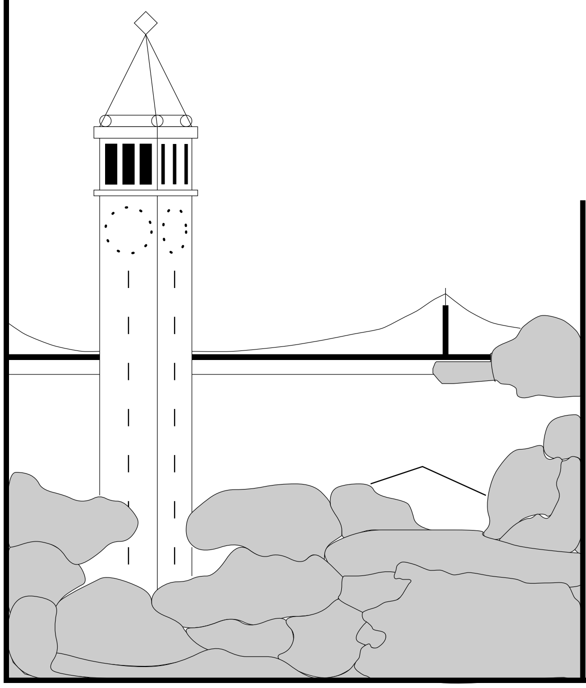
Descriptional Composition of Compiler Components

John Tang Boyland



Report No. UCB//CSD-96-916

September 1996

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Descriptional Composition of Compiler Components

John Tang Boyland*

September, 1996

Abstract

New machine architectures and new programming languages are always appearing, and thus the need for new compilers continues unabated. Even experimental languages and machines need compilers. Compiler writers developing new and/or experimental compilers face competing pressures when designing their large-scale structure. On the one hand, a more modular structure will make it easier to maintain, modify or reuse pieces of the compiler. A more modular compiler is more likely to be correct, and reusable compiler components lead to consistent semantics among the compilers using them. On the other hand, a highly modular structure may lead to inefficiencies in implementation.

Suppose one uses an intermediate representation and divides up the compiler into two parts, one which compiles the source to the intermediate representation and another which translates a program in the intermediate representation to the target machine language. Doing so may make the compiler easier to understand, and furthermore, a well-chosen intermediate representation may prove a suitable target for other source languages, or a suitable source for translating to other machines. On the other hand, the need to create and then traverse this intermediate representation may slow a compiler significantly. If the two parts are described in a high-level declarative formalism, *descriptional composition* can be used to combine the two parts automatically so as to avoid creating and traversing the intermediate structure.

This dissertation presents a declarative compiler description language, APS, and a new method for descriptional composition. The language, based on a variant of attribute grammars, contains a number of features that aid compiler writers in *factoring* descriptions so that each concept can be expressed separately. Both a compiler for Oberon2 and a front-end for APS itself have been written in APS. The back-end of the Oberon2 compiler consists of a translation to a form of the "GCC tree" intermediate representation. Another module gives the translation from this form to source-level C text.

A prototype compiler has been developed for APS that supports descriptional composition. The descriptionally composed version of the Oberon2 back-end with the translation to C text is no larger than the sum of the sizes of the modules from which it is composed, yet it runs almost twice as fast. The Oberon2 compiler is the first successful use of descriptional composition for a realistically complex system, and demonstrates the effectiveness of combining the new APS description language and the new algorithm for descriptional composition presented in this dissertation.

---

*This research was supported in part by the Advanced Research Projects Agency (DoD) under Grant MDA972-92-J-1028, and by the National Science Foundation under Infrastructure Grant CDA-8722788. The content of the information does not necessarily reflect the position or the policy of the U.S. Government.

To my dear grandmother
Virginia Barrett (1917-1996)
who lived to hear that her grandson
had given his dissertation seminar.

And to her as-yet unborn
first great-grandchild.

Contents

Chapter 1

Introduction

Producing compilers for a multitude of different languages and for a multitude of different machines is a task that cries out for software reuse. The number of potential compilers is the product of the number of languages and the number of machines. Already by the late 1950's, a universal intermediate language (UNCOL) was proposed in order to reduce the number of compilers to the sum instead of the product (see Figure 1.1) [91]. Each full compiler would then consist of two stages: the *front end* that translates from a particular programming language to the intermediate language and the *back end* that translates from the intermediate language to a particular machine language.

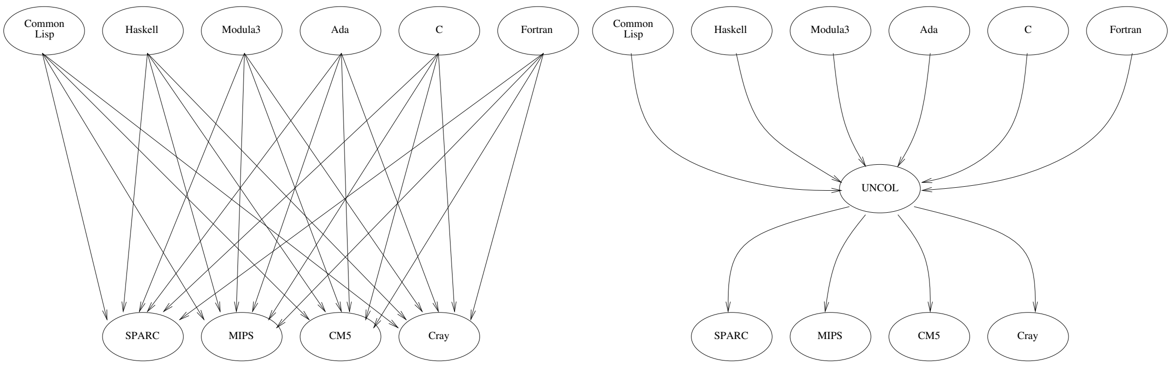


Figure 1.1: The UNCOL problem and solution

The fatal weakness of this scheme was the specification of the intermediate language. If it was too simple, none of the front ends could communicate complicated idioms to the back ends in order to use complex features of particular machines. It would also be difficult to specify important global properties (such as those needed for garbage collection) without over-specifying the implementation in the intermediate language. If, to avoid these problems, the intermediate language contained features from every possible language and machine (an infeasible situation in itself), the intermediate language would be so complex that a back end would be nearly as complicated as the whole set of “direct” compilers it was meant to replace.

One solution is to abandon the goal of having a single intermediate language and to have several intermediate languages, each oriented toward a specific class of languages and specialized for a certain class of machine. Each of these more limited intermediate languages

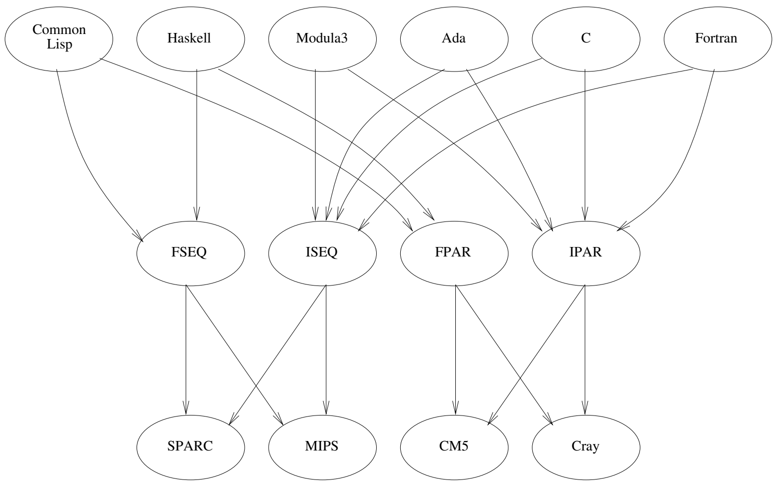


Figure 1.2: Several intermediate languages

would be simpler than an intermediate language which attempts to handle everything. For example, as illustrated in Figure 1.2, one might have specialized intermediate languages for functional as opposed to imperative source languages. Similarly, the intermediate language used for sequential target machines might be different from the one for parallel machines. The register transfer language of *gcc* [89], might be used as the intermediate language (ISEQ) in Figure 1.2 for *I*mperative languages being compiled to *SEQ*uential machines.

If there is no single universal intermediate language, there are still opportunities for sharing between compiler stages. In Figure 1.2 for example, there are two front ends for Ada (one that produces code in an intermediate language destined for sequential machines and one that produces code for parallel machines). These two stages will have to perform many of the same tasks, such as parsing, name resolution and type-checking that have little if any relation to the ultimate machine for which the program is being compiled. This situation is the same for the other languages and to some extent for the machine codes as well.

Thus it is common to break up a compiler into at least *four* stages (see for example *gcc*). Figure 1.3 shows how this situation would work with our example languages and machines. The first stage does tasks that are completely machine independent and produces an intermediate form, that is, a representation of a program produced by one stage and consumed by the next stage. This first intermediate form is often an abstract syntax tree annotated with information gleaned during name resolution and type checking. The next stage produces a language-independent low-level intermediate form taking into account some properties of the target machine but still remaining mostly machine independent. The

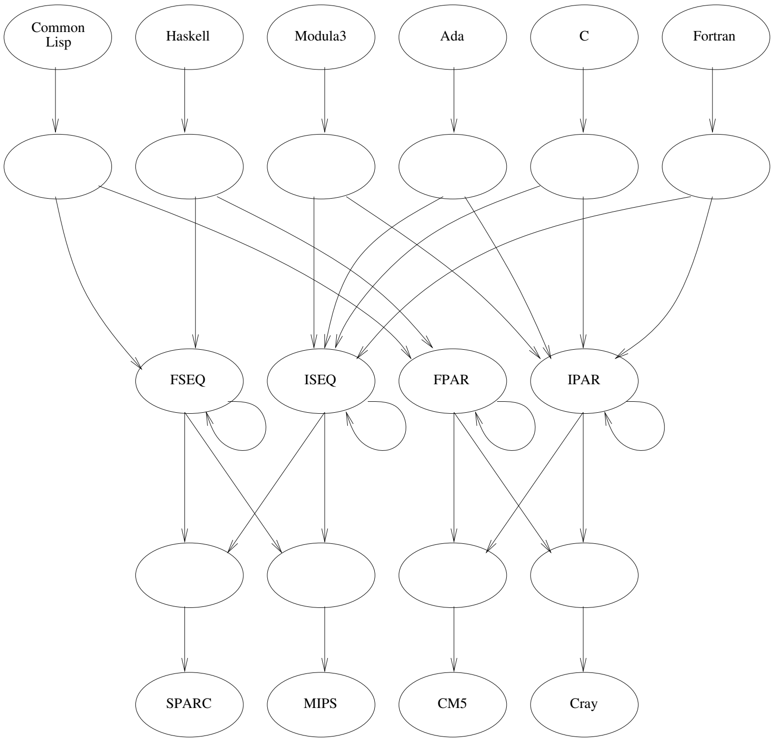


Figure 1.3: Four stage compilers

low-level intermediate form may be optimized independent of both language and machine. Optimizations are represented by cycles in the figure, as these optimizations produce "better" versions of the program in the same intermediate language. The third stage generates a high-level form of the target machine language; assembly language is often used at this point. Using assembly language rather than immediately generating object code removes some operating-system dependencies from this stage. Finally the last stage is completely dependent on machine and operating-system features; it produces an object file and/or an executable.

More sharing is still possible. For example, it might be that a number of languages (say Modula-3, Ada and C) share certain features above that of the language-independent intermediate language, such as the elementary data and control structures common in their Algol heritage. Breaking up the second stage of each compiler for these languages into two stages increases code reuse. The first stage would convert a language specific abstract tree into a basic Algol-family intermediate language. The second stage would convert basic Algol into the appropriate low-level language-independent intermediate language. Similarly, it is likely that the stages that respectively convert the functional and imperative forms of the low-level intermediate language for sequential machines perform many of the same operations. Again breaking these stages into two enables sharing. One stage would be particular to whether a functional language or an imperative language is being compiled and another shared stage specific to the target machine language. It may be profitable to perform optimizations on the intermediate form passed between the two new stages. Farnum [29, 30] proposes this use of multiple intermediate forms in order to increase reuse of optimizations. See Figure 1.4 for how these considerations would affect our picture of the compilers.

One of the stated purposes of UNCOL was to reduce the number of compilers that needed to be written. In Figure 1.1, our example shows a reduction from 24 compilers to 10 compiler stages. The reduction would be greater if more languages or machines were involved. In Figure 1.4, the number of compiler stages, even ignoring the optimizations, has increased beyond that originally required to 29. By the number of stages alone, therefore, the situation is worse. However such counting of compilers and stages makes the implicit assumption that each compiler or stage carries equal weight. This assumption is clearly wrong. A full compiler for Ada that reads in program text and produces SPARC machine code is undeniably more complex than a front end that produces an abstract syntax tree.

Even so, an Ada compiler for the SPARC that consists of six compiler stages (plus optimizations) is likely to be both larger and slower than one monolithic program. In what way, then, can the situation in Figure 1.4 be considered an improvement of that in Figure 1.1? One answer is that the aggregate complexity of the 24 compiler stages of Figure 1.1 is likely to be greater than that of the 29 compiler stages of Figure 1.4. By the criterion of reducing aggregate complexity, it is beneficial to split a stage if one or both of the resulting stages can be reused as shown graphically in Figure 1.5. By breaking up the transformation from X to Z into two (simpler) transformations from X to a new intermediate form XY and from XY to Z, it is possible to reuse the second stage to translate Y to Z, presumably leaving the simpler problem of converting Y to XY.

But aggregate complexity is not the whole of the story. Compilers change. The

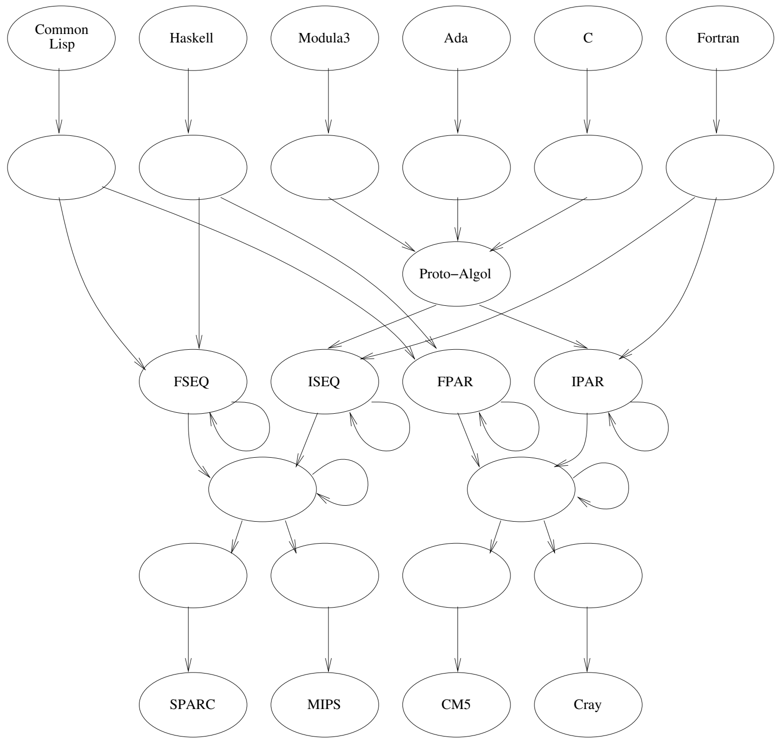


Figure 1.4: Using multiple intermediate forms

language definition may have changed, or there may be a new slightly different version of the target architecture. New optimizations may be added. In any of these cases, all the places in the compiler that deal with the changed concepts or methods must be updated. It eases the revision process if each stage handles as few concepts as possible and if each concept is handled in as few stages as possible. The scope of the change is limited to the affected stages. Thus it may be beneficial to split a stage if the modularity is thereby increased (see Figure 1.6). We say that a description is *factored* when each idea, each conceptual task, is expressed in a single place. Writing a description in factored form properly takes into account the principle that all software should be developed in such a way as to prepare for change.

These benefits of factoring, however, do not take into account the time or space required by the compiler to perform its tasks. Factoring makes a description more modular but increases the number of compiler stages used in a compiler. When a compiler executes as a great number of stages, the compiler's performance is degraded. The compiler may spend most of its time constructing intermediate forms or communicating between stages to the detriment of the actual computation going on. Concern for efficiency can lead a compiler writer to use less factoring than would be desirable for reuse or for anticipating change. For example, Sloane [86] claimed that a significant portion of the memory overhead in an automatically generated Icon compiler could be traced to the use of an intermediate form, and suggests that the decision to use an intermediate form may have been a mistake.

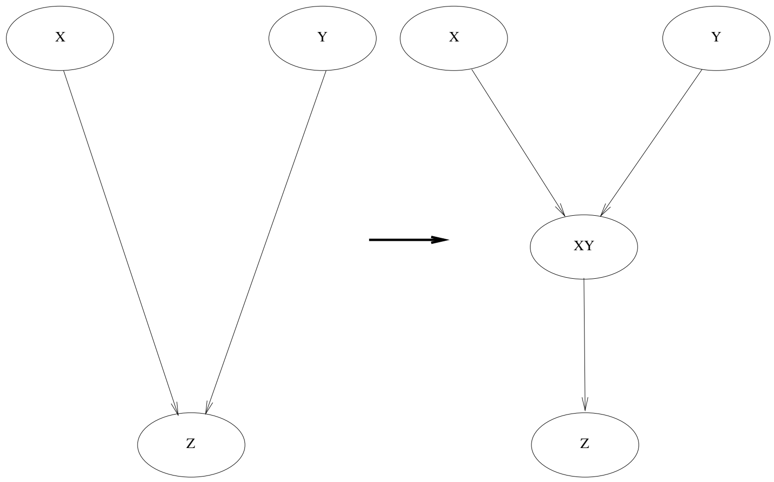


Figure 1.5: Splitting a stage to increase reuse

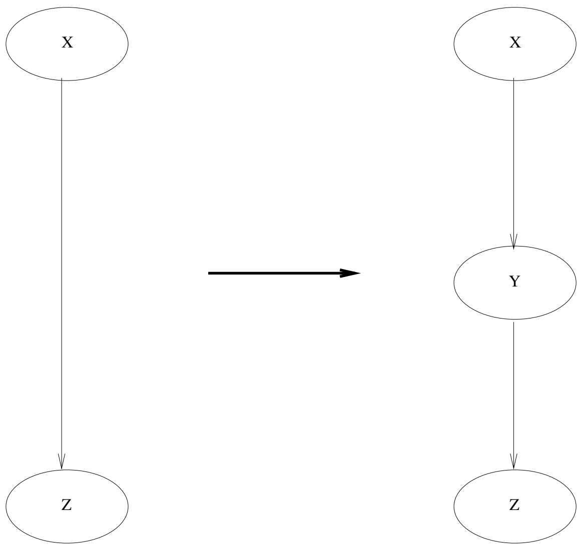


Figure 1.6: Splitting a stage to increase modularity

It is not necessary, however, that a compiler *described* as a series of many stages actually *execute* as a series of serial passes. Compiler stages can be combined in automatic or semi-automatic ways in order to reduce overhead. "Descriptional composition" is one such technique that combines two compiler stages into a single stage. Descriptional composition and other combination techniques allow a compiler writer to write factored descriptions without necessarily incurring overhead during compiler execution time.

Therefore, in order to solve the problems of reuse and change posed in creating compilers for a variety of languages and machines, we need two things. We need a compiler description method that permits factoring at all different levels of the description. We also need tools for implementing these descriptions efficiently, using combination techniques to avoid the cost incurred by naïve implementation of a factored description. In this dissertation, we present a compiler description language that enables factoring at many levels, together with methods for combining factored descriptions to reduce execution overhead.

Section 1.1 discusses factoring and the ways in which factoring can be expressed. Section 1.2 presents combination techniques and how they improve the efficiency of implementations of factored descriptions. Section 1.3 reviews existing compiler description techniques and to what extent they support the factoring methods and the combination techniques of the preceding two sections. In the final section of this chapter we discuss our proposed solution and summarize the rest of the dissertation.

1.1 Factoring In Compiler Descriptions

The process of *factoring* converts a description in which a concept is expressed multiple times into one in which it is expressed just once. The term is taken from mathematics: just as $2x + ax + xz$ can be factored into the multiplication $(2 + a + z)x$, so also can the algorithm described in the left column in Figure 1.7 be factored into the one on the right. In the left column, we have an example of type checking simple expressions of the form $e_1 + e_2$ or $e_1 - e_2$ for two subexpressions e_1 and e_2. If both operands are integers, the resulting expression is also an integer. Otherwise, the result will be a real, with a possible automatic coercion for an integer operand. The program fragment also checks that only integers and reals are being combined. The tests for whether the result is an integer and whether the operation is legal are handled twice, once for $+$ and once for $-$. Also, there are four places where the example determines that an automatic coercion is necessary.

In the factored version in the right column, every one of these tasks is done in only one place. This factoring is made possible by converting the $+$ and $-$ nodes into general *OP* nodes. The factored description also makes use of pattern matching to avoid having to detect coercions redundantly for left and right operands on an expression. Factoring can usually be accomplished in a variety of ways; the example simply gives a flavor of two methods that can be used.

An extreme form of factoring occurs when the same description is used to implement two related but different systems. For example, one description could be implemented both as a batch compiler and as an incremental compiler. Another example is a single description that can be used as a compiler and as a "decompiler" (a program that computes a source-level version of compiled code).

```
for each parse node n of the form x + y
    if type of x is INTEGER and
       type of y is INTEGER then
        type of n is INTEGER
    elsif (type of x is INTEGER or
           type of x is REAL) and
          (type of y is INTEGER or
           type of y is REAL) then
        type of n is REAL
        if type of x is INTEGER then
            coercion is necessary for x
        elsif type of y is INTEGER then
            coercion is necessary for y
        endif
    else
        signal a type error
    endif

for each parse node n of the form x - y
    if type of x is INTEGER and
       type of y is INTEGER then
        type of n is INTEGER
    elsif (type of x is INTEGER or
           type of x is REAL) and
          (type of y is INTEGER or
           type of y is REAL) then
        type of n is REAL
        if type of x is INTEGER then
            coercion is necessary for x
        elsif type of y is INTEGER then
            coercion is necessary for y
        endif
    else
        signal a type error
    endif
```

```
for each parse node n of the form x + y
    convert into x OP y

for each parse node n of the form x - y
    convert into x OP y

for each parse node n of the form x OP y
    if type of x is INTEGER and
       type of y is INTEGER then
        type of n is INTEGER
    elsif (type of x is INTEGER or
           type of x is REAL) and
          (type of y is INTEGER or
           type of y is REAL) then
        type of n is REAL
    else
        signal a type error
    endif

for each parse node n of the form
       x OP y or y OP x
    if type of x is INTEGER and
       type of n is REAL then
        coercion is necessary for x
    endif
```

Figure 1.7: Type-checking simple expressions in redundant and factored form

1.1.1 The Problem of Redundancy

In order to see the benefits of factoring, it is first necessary to look at the problem caused by *redundancy*, the opposite of factoring. Redundancy is present when some concept is expressed more than once. When someone makes a change in a description, they must take into account all the places where a concept is redundantly expressed. This subsection describes the nature of redundancy and then discusses ways in which redundancy arises in descriptions.

If some tool maintains consistency or at least checks the consistency among the redundant expressions of a concept, this redundancy is called *benign*. Otherwise, the redundancy is called *dangerous*, since it could lead to inconsistency. Such (unchecked) consistency can cause a run-time fault. Moreover, dangerous redundancy increases the work needed when maintaining a description over time. A bug fix or improvement in one location must be manually repeated in all other locations redundantly expressing the same concept.

Not all redundancy is dangerous, or even undesirable. Programming languages often require benign redundancy in the form of declarations or interface files, and compilers check that they are consistent. Moreover, not all dangerous redundancy could conceivably be eliminated. For example, comments are desirable because they help explain some segment of code. However, since comments cannot automatically be kept consistent with the code, they are dangerously redundant. In this case, redundancy is necessary.

One instance in which unnecessary dangerous redundancy may arise is in describing parsers. On the one hand, one might choose to use a parser generator such as the Unix utility *yacc*[51]. One the other hand, one might choose to write a parser by hand. The second alternative is much more likely to have instances of dangerous redundancy. Consider the situation when the syntax of a statement in the language being parsed is changed. A parser described using *yacc* typically requires only a single change to the affected rule. On the other hand, in a hand-written parser, the change may be more profound. Not only must the hand-coded parsing of that statement be changed, but if look-ahead sets have changed, changes might be necessary in widely scattered areas in the parser. If those changes are not made, it is likely that the parser will continue to work for most strings in the language even though it is inconsistent. Thus dangerous redundancy can lead to subtle bugs. Thus we see one of the advantages of a formal description; certain properties (such as look-ahead sets) are represented *implicitly* and are computed by a tool and thus consistency is maintained automatically.

The beginning of this chapter discussed a particular form of redundancy: redundancy on the large scale of compiler stages. If we have two compilers for different languages but for the same machine, each compiler will need to perform instruction scheduling (among many other tasks) for that machine. If the compilers do not share an instruction scheduler, then there will be dangerous redundancy. An improvement in the scheduling algorithm used by one compiler does not lead to improvement in the other compiler. A new version of the machine with a different instruction pipeline structure will require both schedulers to be rewritten. Thus it is likely that unless a great effort is made, each compiler will be less effective than it could be.

Dangerous redundancy often arises through *ad hoc reuse* of software, in which code is simply copied from one place to another and then modified as necessary. *Ad hoc* reuse is often the only method of software reuse that is possible; it is often the easiest form of reuse in the short term. Unfortunately, the type of redundancy that arises with *ad hoc* reuse is of the worst possible kind; the redundant expressions of the concept are often in widely separated modules, written by different people and the redundancy is not automatically maintained nor even checked for consistency. *Ad hoc* reuse is thus one of the prime causes of dangerous redundancy.

Ad hoc reuse actually ends up duplicating development and maintenance effort in the future, because inconsistency is more likely to creep in and create subtle bugs in the software. Why then is *ad hoc* reuse ever used? One reason is that *ad hoc* reuse may be the only way to get any reuse at all. For example, the copy may apply to a different type and the programming language may not support polymorphism. Or the source expression may not be modifiable but may need to be generalized in some way. Sadly, a common reason for *ad hoc* reuse is that factoring takes too much time, either for the developer who must rewrite the expression of the concept to generalize it, or in execution, in that a factored description may be too inefficient. For a simple concept expressed as a small fragment of a description, duplicated development and maintenance cost may be smaller than the cost of factoring. However, the more complex a concept and the larger the expression in a description, the more costly *ad hoc* reuse will be in the long run, and the more worthwhile factoring ahead of time will be.

The problem of redundancy therefore is that the existence of multiple expressions of a concept means that multiple changes must be made when the concept is changed or a better method of expressing the concept is to be used. The existence of multiple sites leads to inconsistency when not every site is updated. Redundancy is *benign* if consistency is maintained or at least checked by the tools that implement the description; it is *dangerous* otherwise. It seems unlikely that all dangerous redundancy can be eliminated from a description, but it is desirable that it be reduced as much as practical.

1.1.2 Expressing Factoring

Factoring, the avoidance of redundancy, can be expressed using a number of devices, some of which require specific support from the implementation language. In this subsection we will discuss some of these devices.

It is possible to factor a description with no support whatsoever from the implementation language by using hand-written programs to generate redundant (but consistent) descriptions. One can write a program, possibly with the help of a macro-processor such as M4 [62] or the C preprocessor that can be run with different parameters to generate software fragments that have much in common. The resulting redundancy is benign, as long as it is maintained automatically (perhaps using some form of a "makefile" or script). This method of factoring is often difficult and messy as it involves a lot of small programs that create software fragments. It can be difficult to understand software systems written in this way, since, in effect, each generation tool defines its own language. Moreover, debugging applications built using many generated pieces may be difficult. Therefore, maintenance becomes more expensive. In the case of the programming language C, use of a macro-processor is standardized. Even in C, the macro-processor operates on tokens, not on well-formed subexpressions. Consequently, the pieces into which redundant expressions are factored are not valid language fragments. Errors are reported by the compiler for the expansion, not for the human-written source. That is, since tools outside the language are used to produce consistent versions, the language's own compiler cannot be used to check the fragments by themselves.

Most programming languages provide a way to factor out code that differs only in what particular values are used. The parameterized fragments are variously called "procedures," "functions," or "subroutines." Tools such as compilers can check that a procedure is well-formed without knowing what values will be used for the parameters. In some languages, program units may be parameterized by type as well as by value. Type parameterization (also known as "polymorphism") is an important factoring tool and is better than using textual macro substitution because it works within the language. Errors can be caught by the compiler and expressed in source form. Generics in Ada and templates in C++ provide this functionality.[1] Functional languages such as ML or Haskell provide full checking for functions parameterized by type. Dynamically typed languages such as various versions of Lisp or Smalltalk provide polymorphism, but compilers cannot guarantee type safety.

Even if a programming language provides polymorphism, and especially where polymorphic entities have to be described using special constructs, it may be necessary to rewrite a non-polymorphic fragment into a polymorphic one before it can be reused. This process involves identifying what will be shared and what will not be.

A form of reuse called *inheritance* makes a controlled form of reuse easier, because it is not necessary to decide *a priori* what will be reused and what will not be reused. Inheritance is most common in so-called *object-oriented* programming languages. Inheritance allows one module ("class" in an object-oriented language) to include the features of another module. Some languages permit more than one module to be inherited; this facility is called *multiple inheritance*. Multiple inheritance is desirable for the purposes of factoring because it permits more reuse. In many object-oriented programming languages, however, if one module inherits from another, it is considered a subtype of that module. Unless inheritance can be separated from subtyping, inheritance must either be restricted or it may lead to type insecurities [21, 87]. Several languages, notably Sather [93], completely separate inheritance from subtyping in order to maximize the implementation reuse possible with inheritance.

In the previous subsection, we mentioned that a parser described using *yacc* exhibits factoring; each grammar rule is stated once. Properties such as look-ahead sets can be computed automatically from the description. A parser described by a context-free grammar is more *declarative* than a parser written in a general purpose language since the grammar specifies the task that needs to be done (parsing) but does not specify how that task is to be done. By omitting details of method, a declarative description give more latitude to the implementation. A new implementation may do a better job with the same description. For example, *flex* [77] yields faster scanners than the standard UNIX utility *lex* [66] from the same descriptions.

Another declarative formalism is *pattern matching*. A pattern describes structure,

---

[1] Templates in C++ are weaker that Ada generics in that it is not possible to guarantee that an expansion will not cause an error.

not how to determine a match. A variety of sophisticated implementation techniques are available to implement a description based on pattern matching [47].

A more declarative description may permit widely different execution models. For example, the same description could be implemented both as a batch compiler and as an incremental compiler. Not only is the description writer relieved of the burden of writing two separate descriptions, but it is far more likely that the two compilers will actually implement the same language. Ensuring that separately written batch and incremental compilers use the same language semantics is much harder to verify, especially if the language is undergoing change.

In practice, there is not a hard distinction between declarative and non-declarative description methods. Even a program written in a general purpose language such as C leaves latitude to the implementation; a new compiler can yield a faster implementation of a given program. To the extent that implementation decisions are taken by a shared implementation tool (be it a compiler or a more specialized tool such as *yacc*); they have been factored out of the individual descriptions being implemented.

Thus we see two advantages that more declarative descriptions have over less declarative descriptions as far as factoring is concerned. A more declarative description represents information implicitly (rather than explicitly and redundantly). Moreover, a more declarative description method allows more of the work involved in realizing a working system to be expressed in a shared implementation tool rather than in the descriptions being implemented.

In the start of this chapter, we discussed the sharing of multiple compiler stages as an important example of factoring. Such sharing is possible when a well-defined intermediate structure is used to pass information from one stage to another. The fact that the intermediate "tree" used in *gcc* is documented only with C header files has hindered its use as an intermediate language for other compilers (but recently, more information has become available [61]). Sharing of compiler stages is hindered even further when the connection between two stages is not encapsulated in a structure at all, but in information sent back and forth through many procedural interfaces. This situation exists in the case of *gcc*. Thus it helps factoring if there is a way to declare intermediate structures and to ensure that two stages communicate only through this structure.

In this section, we discussed the dangers of redundancy and *ad hoc* code reuse. We also discussed the following ways in which factoring can be expressed:

- textual macros
- subroutines
- polymorphism
- inheritance
- declarative formalisms (including pattern matching)
- intermediate structures

Textual macros work outside the language and lead to a proliferation of complex meta-tools. The other techniques operate within the description language and thus rely on the implementation for efficiency. Naïve implementation of highly factored descriptions is likely to lead to high overhead at runtime. The following section describes some methods for implementing factored descriptions more efficiently.

1.2 Combination Techniques

We have described a number of factoring techniques, but the relative efficiency of factored descriptions depends on the program that implements the description. The execution cost of an implementation is relevant because the person writing a description may be wary of the cost of factored descriptions. Since factored descriptions require more discipline to write, the envisaged efficiency loss can lead the description writer to create a less factored description. Hence the lack of good implementation techniques not only leads to inefficient implementations, but also hinders factoring. Development and maintenance costs for software production are thus increased.

In this section, we examine some known implementation methods that can be used to reduce or eliminate the overhead in factored descriptions. Better implementations of course yield more efficient programs. More importantly, as program writers become confident that factoring will not necessarily lead to inefficiency, they will write better structured programs.

1.2.1 Inlining and Specialization

Subroutine inlining is a well-known practice for reducing the cost of a factored description. When a subroutine does such a simple task that the overhead of calling the subroutine is a relatively large fraction of the execution time, it can be beneficial at each call site to compile the body of the subroutine rather than compiling a call. If the compiled procedure body is larger than the compiled call, inlining the procedure will increase the size of the compiled program. Thus inlining usually only beneficial for small procedures.

A subroutine or polymorphic entity may be *specialized* by being compiled under some assumptions about its parameters. A specialized entity may often be faster than a general form because known values may be faster to handle than unknown ones, and more significantly because *partial evaluation* (described in the following paragraph) may apply. Specialization can be thought of as inlining of a meta-program taking parameters and producing a specialized entity. Generics in Ada are usually implemented in this way; a compiler will compile a new instance of a generic for each instantiation in the program.

After inlining, it may happen that some of the parameters have known values, or that certain conditions in the inlined subroutine can be determined at compile time. *Constant propagation* and more generally *partial evaluation*, can be used to improve the execution of a program. Constant propagation is most often associated with imperative languages and partial evaluation with functional ones. In constant propagation, the compiler notices which uses of a named entity will always get the same constant value. When the values of entities are known at compile-time, expressions can be evaluated and control-flow simplified at compile-time as well.

1.2.2 Descriptional Composition and Deforestation

When one stage produces a data structure to be traversed and processed by the next stage, combining the two stages is not a simple matter of subroutine inlining. Instead, somehow, the production and traversal actions must be interleaved. Moreover since the structure may not be created in the same way as it is traversed, the control structure of the combined stage must often be more complicated than either of the two original stages. Ganzinger and Giegerich call this task *descriptional composition* [39] (to stress the fact that interleaving is necessary), and Wadler coined the term *deforestation* [97] (because intermediate "trees" are being removed). Other researchers in functional languages use the term *fusion* [65]. The term deforestation is usually used when the production and traversal actions are each expressed in a single function and the term descriptional composition is usually used the tree is produced by one "attribute grammar" and traversed by a second. In principle, however, the process has the same effect in each case.

Descriptional composition allows a programmer to use a method that is clear and clean, but would otherwise seem inefficient in intermediate resources. For example, a factorial function could be defined by creating a list of integers (a library routine) and then reducing the list using multiplication (the reduction function also being a library function). A "deforesting" compiler would compile this function into one that would perform the task without allocating any list elements.

In order that such a combining tool do its task, it must be able to distinguish generation of structure and traversal of structure from other actions done by stages. Moreover, since it should be possible to combine a combined stage with yet another stage, it must also be possible to express the more complicated control of a combined stage in the original formalism. For these reasons, descriptional composition has only been implemented for highly declarative languages and for attribute grammars, the latter being a declarative tree-based formalism that will be discussed later. Moreover, all reported instances of descriptional composition have involved unrealistically simple tasks.

As with inlining, descriptional composition does not always increase efficiency; the resulting stage is often bigger than the two stages combined to form it. However, also as with inlining, the resulting stage often can be improved using partial evaluation.

1.3 Compiler-Description Methods

In this section, we consider methods for describing compilers. For each method we discuss the support for factoring. We also look at specific implementations and to what extent combining methods are available for making factoring efficient. Section 1.3.1 and Section 1.3.2 briefly discuss the issue of writing compilers using general purpose programming languages, first with imperative languages and then with more declarative languages. Section 1.3.3 describes some special purpose compiler description methods. Section 1.3.4 concludes with a discussion of one particular class of these methods, attribute grammars.

1.3.1 Imperative General-Purpose Programming Languages

It would be impossible do justice to all the imperative languages that are useful for writing compilers, but this section attempts to give a brief outline.

One particular language, C, is often used to write compilers because of its flexibility in handling characters and dynamically allocated structures, and because C compilers usually produce efficient code. C, like most imperative languages has subroutines ("functions") and, through its required preprocessor, has textual macros. It lacks any method for polymorphism other than subverting the type system through unsafe binary level type reinterpretations. C has the power to declare complicated pointer-based data structures but the lack of polymorphism or controlled subtyping often reduces their declarative power. C compilers can perform inlining (usually only with compiler directives) and constant propagation, but descriptional composition is problematic, as we shall see.

Other more modern programming languages (for instance Ada, C++, Modula-3 [74], and Oberon2 [73]) provide various forms of polymorphism allowing for more factoring and better descriptions of structure. Specialization of polymorphic entities (*generics* in Ada or *templates* in C++) is usually done as a matter of course.

Common Lisp [90] provides many factoring methods, including fully-general syntactic macros and a powerful object-oriented facility in CLOS. The dynamic semantics of Common Lisp (allowing, for example, a function definition to be replaced at runtime) makes it harder to perform combining techniques such as inlining and partial evaluation without programmer annotations. Various subsets or near subsets of Common Lisp have been proposed to make such optimizations easier such as Goerigk et al's CommonLisp$_0$ [43] or the ISO's draft of ISLISP [49].

One of the features of imperative languages is that structure can be changed destructively. The programmer specifies control explicitly in order to ensure that destructive updates are sequenced correctly. The first feature makes descriptional composition problematic because destructive updates in the stage creating an intermediate structure and those in the stage traversing it may interfere with each other. Moreover, much analysis is required before a compiler can safely modify a programmer-defined single thread of control.

1.3.2 Declarative General-Purpose Programming Languages

In declarative programming languages such as Prolog and Haskell, the programmer expresses *what* values need to be computed rather than *how* they are computed. In such languages, destructive update is restricted or eliminated altogether as a language feature in order to preserve the quality of *referential transparency*, which roughly means that the same expression computes the same value wherever it might occur. Such properties permit more control-flow decisions to be made by the compiler. Consequently, some optimizations are easier to perform.

Haskell is a side-effect-free language; in particular, there are no programmer-visible destructive updates. Moreover it is *lazy*; no actual parameter's value is computed unless it is actually needed. These features provide a challenge to the compiler writer; naïve implementation yields very inefficient programs. Compiler writers have risen to the challenge, not only performing inlining and partial evaluation (see, for example, Peyton Jones and

Launchbury's description of "unboxing" transformations [79]) but also providing Wadler style deforestation [42, 71, 97]. Unfortunately it appears that deforestation is only implemented for simple cases (one recursive function definition using built-in operations) and the technology is not yet sophisticated enough to support the implementation of full compiler stages.

Prolog programs express the logical relations between inputs and outputs and thus the same program may be understood as mapping inputs to outputs or as mapping outputs to inputs. For example, a compiler written in Prolog can theoretically be used as a decompiler (generating source code from object code) [10]. As mentioned earlier, this feature is an extreme form of factoring; the connection between the source and object is described in a single place rather than requiring two specifications. In practice, however, the implementation model used by Prolog often makes "backward" runs prohibitively expensive or even non-terminating.

In languages without assignment, there is no separation between the concept of "value" from "container of values" (that is, an "object"), so there is no concept of *object identity* (as abstraction of *address*). As a result, it is not possible to directly model the attribution of trees in such languages. In fact, it is impossible to distinguish trees from directed acyclic graphs. This feature has its advantages, but for graph algorithms and more specifically for various compilation tasks, this feature can be a hindrance.[2] Chapters 5 and 6 demonstrate the utility of object identity in compiler descriptions.

1.3.3 Special-Purpose Compiler Description Systems

Rather than use a general-purpose programming language to write a compiler, one might want to use a system especially geared toward writing compilers. Waite discusses the conditions under which formal descriptions and the corresponding compiler construction tools are useful [99]. Wilhelm discusses the particular issue of describing compiler transformations [102]. Even without an efficient implementation, a formalism can be useful for prototyping new compilers or for specification (for example, see Kasten's LIDO specification language [56]). Two very important characteristics are that the formal description must be clearer than an operational description in a general-purpose language and should be simpler. To these characteristics, I would add the requirement that a formal specification method encourage factoring, especially factoring methods specific to compiler writing, such as pattern matching. If a formal description requires redundant unchecked specification where an operational solution does not, then the formal description method is flawed, and is less useful even as a non-executable specification.

Formal description methods admit multiple implementation methods, and in the case of very high-level compiler description methods, it is possible to provide implementations of different execution paradigms. For example in FNC2, the same description can be used to describe a batch compiler and an incremental compiler [54].

Special-purpose compiler description methods can be grafted onto an underlying general-purpose programming language, or they can be full languages in their own right. In

[2] However, King and Launchbury's recent work in monads has allowed efficient implementation of standard graph algorithms [63].

the former case, the implementation usually consists of a reduction to the underlying language, leaving less room for special execution paradigms. As pointed out by Jourdan *et al.* the latter case gives the implementor more power for high-level optimizations and multiple execution paradigms [53].

On the other hand, it is difficult to implement a powerful feature efficiently. Often a generally applicable implementation method for a feature will do poorly in cases where little of the power of the feature is exploited. This problem is a common one facing all implementors of higher-level languages. A solution can often be found through more sophisticated analysis that determines when a specialized and efficient mechanism can apply. A designer of a compiler description method is challenged to provide a formalism that is not only easier to use to write compilers, but also (almost) as efficient as hand-coding. The requirement of efficiency becomes stronger if the special purpose formalism does not provide much help to the compiler writer in expressive power.

The formalisms that have received the most attention are those based on attribute grammars [64]. The remainder of this subsection describes some representative systems built on other formalisms. Section 1.3.4 discusses attribute grammars and variations.

OPTRAN

A description in Lipps et al's OPTRAN system [67] consists of two parts, an attribute grammar defined over abstract syntax trees and a set of rewrite rules. Attribute grammars are discussed in Section 1.3.4. An example of an OPTRAN rewrite rule is one to replace the addition of two integer constants by their sum:

```
transform
    <addop,<intconst\1>,<intconst\2>>
into
    <intconst>
apply
    trafoaddop(sscanattr of intconst\1,
               sscanattr of intconst\2,
               sscanattr of intconst);
```

This fragment rewrites any node denoting the addition of two constants by a single constant node. It also computes the `sscanattr` of the new node (that is, the constant's value) by calling a procedure `trafoaddop` that takes two input parameters and one output parameter.

The OPTRAN compiler generates Pascal code, which is linked with hand-written code implementing such procedures as `trafoaddop`. In essence then, OPTRAN serves as a powerful extension to the Pascal language. Unfortunately some of the features of Pascal (such as modifiable global variables) interfered with features in OPTRAN (such as incremental re-evaluation of attributes). Efforts were then directed toward a functional model of transformations as exemplified by Trafola (below) [102].

A later system based on a similar synthesis of attribution and transformation is Farnum's DORA system [30]. DORA uses a more powerful attribution model including pattern-matching [31] and fixpoint computations [29].

Trafola

Heckmann's Trafola [45] is a functional language with particular emphasis on tree pattern matching and transformations. In its foundations, it is related to pure functional languages such as Miranda [94], and thus includes the useful factoring tools of that tradition, in particular lazy polymorphic functions. More importantly, Trafola includes a powerful pattern matching system, including *insert patterns* (notated using ^), in which any descendant of a node can be identified in a single pattern. For example, a simple partial evaluator for arithmetic expressions can be written as follows:

```
Repeat
  {P ^ ('add['integer X,'integer Y]) => P ^ ('integer(X+Y))
  #P ^ ('add[X, integer 0])          => P ^ X
  #P ^ ('add[integer 0, X])          => P ^ X
  #P ^ ('mul['integer X,'integer Y]) => P ^ ('integer(X*Y))
  #P ^ ('mul[X, integer 0])          => P ^ ('integer 0)
  #P ^ ('mul[integer 0, X])          => P ^ ('integer 0)
  # other                            => other }
```

The `Repeat` function takes a tree transformation as its argument and returns a new transformation. This new transformation applies the original transformation repeatedly to a tree until no change is made. The transformation in this case is an anonymous function whose body is inside the braces, `{}`. It takes a tree and returns a possibly new tree. The body consists of a sequence of rewrite rules written *pattern* `=>` *result*; the first matching pattern is chosen. In this example, most of the rules are of the form `P ^` Q `=> P ^` Q'. Such rewrite rules apply the rewrite rule Q `=>` Q' to every subtree in the tree being transformed. The first rewrite rule does the same task as the rule in the OPTRAN example. Chapter 3 discusses the insert operator ^ in greater detail.

Such powerful language features need sophisticated implementations in order to avoid being too costly. Trafola includes an optional type checker (a type inferencer in the style of ML) and as described by Ferdinand [37] uses bottom-up linear time tree parsing in order to make pattern matching more efficient. (See the PROSPECTRA System report for a more complete description of the implementation [3]).

CENTAUR/Typol

The CENTAUR system is a multi-lingual integrated software development environment [19]. As part of that work, researchers have developed a system for type-checking programs that uses the powerful notation of Natural Semantics [55] based on operational semantics. Operational semantics is widely used in describing and proving properties of programming languages; see, for example, the definition of Standard ML [72].

The description language, Typol [24], is related to Prolog in that it is a pure logic language and so does not have the concept of object identity. Typol supports pattern matching (through unification) and conditional rule application. Like Prolog, it has a general form of subroutines and a polymorphic type system, but unlike Prolog, it has a static type system. Typol is a powerful language and thus a general implementation can be inefficient. Various subsets have been defined that can be implemented more efficiently [5, 6].

The following fragment specifies the run-time semantics of "while" statements in a simple imperative language:

$$\frac{s \vdash \text{EXP} : \mathit{true} \quad s \vdash \text{STMTS} : s_1 \quad s_1 \vdash \texttt{while}\ \text{EXP}\ \texttt{do}\ \text{STMTS}\ \texttt{end} : s_2}{s \vdash \texttt{while}\ \text{EXP}\ \texttt{do}\ \text{STMTS}\ \texttt{end} : s_2} \quad (r1)$$

$$\frac{s \vdash \text{EXP} : \mathit{false}}{s \vdash \texttt{while}\ \text{EXP}\ \texttt{do}\ \text{STMTS}\ \texttt{end} : s} \quad (r2)$$

Rules of the form $\frac{\mathit{this} \quad \mathit{that}}{\mathit{something}}$ are read "if *this* and *that* then *something*". The first rule specifies that if in state s, the expression evaluates to *true* and if the new state after executing the body is s_1 and if executing the whole loop again with this new state yields state s_2, then the result of executing the loop starting in state s is s_2. The second rule is simpler; it states that if the expression evaluates to *false*, then the state is not changed after the loop executes.

Back End Generators

A number of generators for the final machine-specific part of a compiler have been developed including *burg* [80], *iburg* [38], Emmelmann et al's BEG [27, 28] and Bradlee et al's Marion [15, 16]. From a description of the machine, these systems generate instruction selection routines that find the cheapest instruction sequence. Some (notably Marion) take into account pipelines and other processor resource issues thus allowing the generator to handle more of the issues that must be addressed by a back end. As lexer and parser generators enable easy specification and modification of lexers and parsers, so these tools greatly simplify the task of creating and maintaining back ends for compilers.

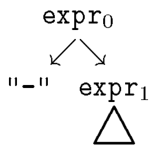
1.3.4 Attribute Grammars

Attribute grammars (first described by Knuth [64]) are a formalism for associating values with instances of productions in a context free grammar. Attribute grammars are useful for specifying front ends for compilers, or indeed entire compilers (see, for example, Farrow's production compiler for Pascal [32]). Attribute grammars have a set of *attribution definitions* associated with each production in a context-free grammar. These definitions show how to compute each attribute instance, that is, values associated with tree nodes in a (possibly abstract) parse tree of a program. For example, the attribute definitions associated with a tree node representing a unary negated expression might be expressed in the following way:

```
expr0 → "-" expr1
        expr0.type :=
                if expr1.type = INT_TYPE then
                        INT_TYPE
                else if expr1.type = REAL_TYPE then
                        REAL_TYPE
                else
                        ERROR_TYPE;
```

```
expr0.code :=
        if expr1.type = INT_TYPE then
                expr1.code || INEG
        else if expr1.type = REAL_TYPE then
                expr1.code || FNEG
        else if expr1.type == ERROR_TYPE then
                expr1.code
        else
                ERROR["illegal operand to negation"];
```

The production in this fragment is $\texttt{expr}_0 \rightarrow$ `"-"` $\texttt{expr}_1$. An instance of this production is a parse tree of the following shape:



In the scope of this production are two attribute definitions.[3] The first defines $\texttt{expr}_0$`.type`, that is, the `type` attribute of the parent node $\texttt{expr}_0$. The definition specifies the type of the parent node as the same as that of child node $\texttt{expr}_1$, as long as the type is `INT_TYPE` or `REAL_TYPE`. Otherwise the type of the parent is computed as `ERROR_TYPE`. The second definition defines $\texttt{expr}_0$`.code`, the code to be generated for the parent. Notice that this definition uses both attributes of the child.

Attribute grammars have a number of strengths. The first strength is that (syntactic) case analysis for attributes of a nonterminal is already exposed. There is a close connection between the productions of the context-free grammar and the attribution definitions. This case analysis provides for a degree of (enforced) factoring.

Another strength is that evaluation order is implicit—the tool that implements an attribute grammar determines an evaluation order so that an attribute is scheduled for evaluation before any attribute that depends on its value. Because evaluation order is implicit, side-effects are not permitted in (formal) attribute grammars. Consequently, a single description can be used to specify both a batch compiler and an incremental compiler.[4]

One of the limitations of attribute grammars is a result of their simplicity—all attributes are defined using local dependencies. In the example, only the nodes $\texttt{expr}_0$ and $\texttt{expr}_1$ were visible to the definitions. It would not be possible to write a definition that uses a child of $\texttt{expr}_1$. In this case, such a rule would be of dubious utility. However, it is natural to use information about a declaration of a variable at some use site of that variable. In a parse tree, however, the declarations and uses rarely have a parent-child relationship. In order to establish a dependency between two nodes widely separated in a tree, it is necessary to transfer needed values through all the intermediate nodes along the path from one to the other in the tree. The need to specify all the intermediate copy rules decreases

[3]This example uses an Algol-like syntax for the attribute definitions.

[4]Getting an efficient incremental language analysis, however, requires more than simply incrementalizing an existing description.

the usefulness of classical attribute grammars because it makes a formal description more complex to read and write. Moreover an attribute grammar may even be more complex than a compiler written in a general purpose programming language.

Another limitation is that all the attribute definitions associated with a given production are specified in one place. In order to understand the contribution a single attribute makes in a compiler, say one used for code generation, it may be necessary to scan the entire description to find all the definitions for that attribute, and then again to see how its values are used.

Classical attribute grammars have no support for pattern-matching, beyond single productions. Pattern-matching must be done "by hand" using attributes. Lack of support for pattern-matching not only makes descriptions more complex, but also leads to dangerous redundancy (as explained in Chapter 3).

Practical tools that implement attribute grammar-based formalisms often tackle these problems by providing extensions to the attribute grammar formalism. As a result, attribute grammars are not standardized or used as an information exchange formalism as context free grammars are used. Classical attribute grammars are too clumsy and extensions are too closely identified with specific tools. There has been some effort to describe extensions independent of any tool and to show a strong theoretical foundation, for example descriptional composition, fixpoint computations, higher order attributes, pattern-based attribution, conditional attribution, and remote attribution. One can hope that these extensions (to be more fully discussed in this dissertation) will be more widely accepted, thus permitting attribute grammars to be more used for explanatory description, in a similar manner to context-free grammars.

One of the earliest attribute grammar systems to be used was Kasten et al's GAG [58], an implementation of classical attribute grammars. The largest system reported built with GAG was an 400 page description for the analysis of Ada [95]. GAG's successor, LIGA [57], supports the use of (externally defined) side-effecting data structures protected by control attributes that order the requests on such data structures. Such descriptions cannot be easily incrementalized. LIGA provides assistance for remote access of nodes. Any definition may refer to an ancestor by type, or to all descendants of a given type. LIGA also has "chain" attributes where each element in a list adds something to an accumulating result. These extensions allow a description to be more concise.

Another early system was Farrow's Linguist [34], which has been used to describe a production Pascal compiler as well as a VHDL compiler [32]. It has been extended with a variant of higher-order attribute grammars to handle the creation of derived structure which can then be traversed and attributed [36]. Farrow has also developed a prototype that supports building of circular structures (such as are used in symbol tables) [35].

Particularly relevant to this dissertation is the MARVIN [40] system. It was developed as an implementation platform for attribute-coupled grammars [41] and their descriptional composition. MARVIN unifies all data structures (the tree itself and auxiliary structures such as environments) under the single concept of sorted algebras. MARVIN is built as an extension to Modula2 [103] and so specifications in MARVIN are a hybrid of declarative and imperative programming; this feature hinders incremental implementation. Moreover, the descriptional composition method used can only handle the very simplest forms of transformations—those that yield a basically isomorphic structure; there is no support for choosing between two output possibilities.

The FNC2 [52, 54] compiler description system can generate three types of compilers from a single description—batch evaluators, parallelized batch evaluators and incremental evaluators. Attribute grammars for FNC2 are written in OLGA [53]. As with LIGA, there are mechanisms for automatically generating copy rules and for referring to attributes of an ancestor (though not of descendants). OLGA has a limited form of pattern matching that can be used to provide default values for attributes. OLGA has a module system that allows tools to be written in several distinct parts, thus encouraging reuse and also aiding readability. Modules can be separately compiled. FNC2 does not currently implement descriptional composition.

Attribute grammars are good formalisms because they use the syntax to determine the dependencies between attributes, and by having an implicit evaluation order, they permit a wide variety of batch and incremental evaluations. Unfortunately, as currently implemented, there is little or no support for pattern matching or descriptional composition. For more information on attribute grammars and implementations, the interested reader should see Deransart and Jourdan's review [23].

1.4 The Thesis

In order to assist the writing of factored reusable compiler descriptions, we want a description method that has support for both large scale and small scale factoring, with implementation methods that use combining techniques for efficient implementation. It also is a great advantage if the same specification can be used for both incremental and batch evaluation.

None of the reviewed systems accomplish all these things. In particular they lack the ability to do large-scale descriptional composition. Moreover, many of the simplest and cleanest systems, even if they had the necessary power, do not provide enough expressiveness, making them fundamentally flawed from a user's point of view. The simpler systems often do not have enough support for factoring methods such as pattern-matching. The result is often dangerous redundancy.

It is our thesis that it is possible to support powerful factoring techniques together with combining methods such as descriptional composition in an expressive compiler description language. This dissertation presents a compiler description language based on attribute grammars with numerous powerful extensions. This language is designed on the principle that factoring should be as unlimited as possible. From a description factored by concept, it is a straightforward task to generate a description factored by attribute or node type; the reverse transformation is completely infeasible. As a result, the proposed description language contains many (useful) features. It is meant to be used; in fact, the compiler for the description language is written in the language (see Appendix C).

1.5 Summary

In order to achieve more reuse both between different compilers and between different versions of the same compiler, compiler descriptions should be written in many atomic reusable stages, each performing a single conceptual task. In other words, compilers, like all software, should be written in a factored style. Considerations of efficiency, however, discourage programmers from using factoring, especially as factoring may require more rigorous design. Current compiler description methods lack the ability to combine factored descriptions to avoid the costs that accompany the use of multiple intermediate forms.

Therefore, we need a compiler description language that not only supports factoring in a variety of ways, but also permits efficient implementation through the use of combining techniques. This dissertation provides such a language (APS) and descriptional composition of compiler components written in APS. Chapter 2 introduces the proposed compiler description language, APS, informally and shows how to express classical attribute grammars. The following chapters describe in turn the major features of the language: pattern matching, logical sequences, remote attribution, circular attribution and modularization. Each chapter shows how these methods increase factoring potential, how they are expressed in APS and what combination techniques are used in implementation. Chapter 9 describes the operation of descriptional composition in the APS compiler. Chapter 10 concludes the dissertation with a report of experiences and some ideas for further work. Appendix A contains a summary of APS. Appendix B contains a compiler for a simple language, Oberon2, and Appendix C provides a front-end for APS described in terms of itself. The Oberon2 compiler is used as a running example for describing the features of APS.

Chapter 2

An Introduction to the APS Compiler Descriptional Language

This chapter introduces APS, a compiler description language. APS extends the attribute grammar formalism with pattern matching, sequences, higher-order features and modules. These extensions enable numerous factoring methods. Section 2.1 introduces the basic concepts of APS, trees and attributes. Section 2.2 explains some of the minor extensions. The chapter closes with a discussion of the basic implementation model used by the current APS compiler. The major extensions follow in their own chapters. A complete summary of APS in given in Appendix A.

The examples used in this chapter come from an Oberon2 [73] compiler presented in Appendix B. Oberon2 is the latest in the line of Niklaus Wirth's languages in the Algol family. Some of the other well-known languages are Pascal, Modula2 and Oberon. Oberon2 is a small language, but not a toy language. It contains modules, record subtyping, overridable methods, open array types, garbage collection, sets and five numeric types. Oberon2 is thus complex enough to be used to evaluate a compiler description language, yet small enough to be easily understood by the reader.

2.1 Basics

The APS description language is a programming language designed for operating on *forests* of *trees*. In a tree, each node other than the *root* has a *parent* node. A root does not have a parent. A *subtree* rooted at a node is the portion of a tree "under" that node. A *forest* consists of an ordered collection of trees and thus has multiple root nodes. Although APS primarily supports trees, one can also define directed-acyclic graphs (DAGs) as well as general graphs. In DAGs, nodes can be shared between parents, but no node can be its own ancestor. There are no such restrictions for general graphs. This chapter, however, only describes the attribution of trees.

APS provides the ability to define *attributes*—typed values that are associated with particular nodes. The programmer specifies the values by expressing an attribute as a function of other values, including other attributes. The APS compiler *schedules* the evaluation of the attributes so that no attribute is used before it is defined. Automatic scheduling is one of the main benefits of attribute-grammar based systems such as APS.

2.1.1 Tree Languages

In APS, trees are *typed*, that is, they obey certain structural rules. These restrictions are given in an APS module that describes a *tree language*. Figure 2.1 gives an example of such a module. A tree language is defined by a set of *phyla* (singular: phylum), and a set of *constructors*. A phylum is a type to which subtrees may belong. The phyla are disjoint; no subtree belongs to more than one phylum. Each constructor is associated with exactly one phylum, although multiple constructors may share the same phylum. A constructor is a function that creates a tree node and labels it with the name of the constructor. By definition, the subtree rooted at this node belongs to the phylum of the constructor. By extension, we say a node belongs to a phylum if the subtree rooted at that node belongs to the phylum.

Some tree languages allow all subtrees in all contexts; in this case, there is one phylum for all constructors. More commonly (as in the example) there is one phylum for each nonterminal in an abstract syntax.

A constructor has zero or more formal parameters. A parameter whose type is a phylum takes a subtree, the root of which becomes a child of the node being created. Other parameters take values to be stored in the node. Following Ganzinger and Giegerich [39], we call these two kinds of parameters *syntactic* parameters and *semantic* parameters respectively. The distinction is important because the structure of the tree is determined by the syntactic parameters. The semantic parameters merely describe what values are stored in the nodes at their creation. Technically, these values are "attributes," but that term is used here only for values added after creation.

The set of trees in a tree language are those that can be built by repeated application of the constructors. There must always be at least one constructor that has no syntactic parameters. Otherwise, tree construction could never start. Constructors that only have semantic parameters are called *leaf* constructors and they create *leaf* nodes.

Phyla play the role of nonterminals in a context-free grammar in the sense that phyla denote sets of subtrees, while nonterminals denote sets of substrings. Constructors play the role of grammar productions.

In Figure 2.1, we have (part of) an APS module that describes an abstract syntax for Oberon2. The full module is given in Appendix B.1. The tree language of this portion has 12 phyla (`Program`, `Block`, `Declaration`, etc) and 19 constructors (`program`, `module_decl`, `block`, etc). The last three phyla (`Modules`, `Declarations` and `Statements`) are *sequence phyla*. A subtree of a sequence phylum is a list of subtrees; each subtree is of the phylum named inside the brackets (`[ ]`). For example, subtrees of phylum `Modules` are lists of subtrees of phylum `Declaration`. Sequence phyla are discussed fully in Chapter 4.

Figures 2.2, 2.3 and 2.4 give an example of how an Oberon2 module is represented in an APS tree. The function `make_symbol` returns a `Symbol` for a string and `integer_constant` returns a `Oberon2Constant` for an integer value. Sequences are constructed with the `{...}` notation.

```
module OBERON2_TREE[] begin
  phylum Program;
  phylum Block;
  phylum Declaration;
  phylum Type;
  phylum Statement;
  phylum Expression;
  phylum Operator;
  phylum IdentDef;
  phylum Use;
  phylum Modules:=SEQUENCE[Declaration];
  phylum Declarations:=SEQUENCE[Declaration];
  phylum Statements:=SEQUENCE[Statement];
  ⋮
  constructor program(modules : Modules) : Program;
  constructor module_decl(name : IdentDef; body : Block) : Declaration;
  constructor block(decls : Declarations; stmts : Statements) : Block;
  constructor var_decl(name : IdentDef; shape : Type) : Declaration;
  ⋮
  constructor fixed_array_type(length : Expression; element_type : Type)
      : Type;
  constructor open_array_type(element_type : Type) : Type;
  ⋮
  constructor boolean_type() : Type;
  constructor integer_type() : Type;
  constructor assign_stmt(lhs : Expression; rhs : Expression) : Statement;
  constructor named_expr(using : Use) : Expression;
  constructor unop(op : Operator; arg : Expression) : Expression;
  constructor binop(op : Operator; arg1, arg2 : Expression) : Expression;
  constructor is_test(value : Expression; test_type : Type) : Expression;
  constructor aref(array : Expression; index : Expression) : Expression;
  constructor constant_expression(value : Constant) : Expression;
  ⋮
  constructor plus() : Operator;
  constructor minus() : Operator;
  ⋮
  constructor identifier(name : Symbol) : IdentDef;    NB: simplified
  constructor use_name(name : Symbol) : Use;
end;
```

Figure 2.1: An Abstract Syntax for Oberon2 (portions elided)

```
MODULE Simple;
  VAR x : Integer;
  VAR y : Integer;
BEGIN
  x := 12;
  y := x + x;
END.
```

Figure 2.2: A simple Oberon2 module

```
program(
  { module_decl(
      identifier(make_symbol("Simple")),
      block(
        { var_decl(
            identifier(make_symbol("x")),
            integer_type()),
          var_decl(
            identifier(make_symbol("y")),
            integer_type()) },
        { assign_stmt(
            named_expr(use_name(make_symbol("x"))),
            constant_expression(integer_constant(12))),
          assign_stmt(
            named_expr(use_name(make_symbol("y"))),
            binop(plus(),
                  named_expr(use_name(make_symbol("x"))),
                  named_expr(use_name(make_symbol("x"))))) })) })
```

Figure 2.3: An APS expression to make a tree for Figure 2.2

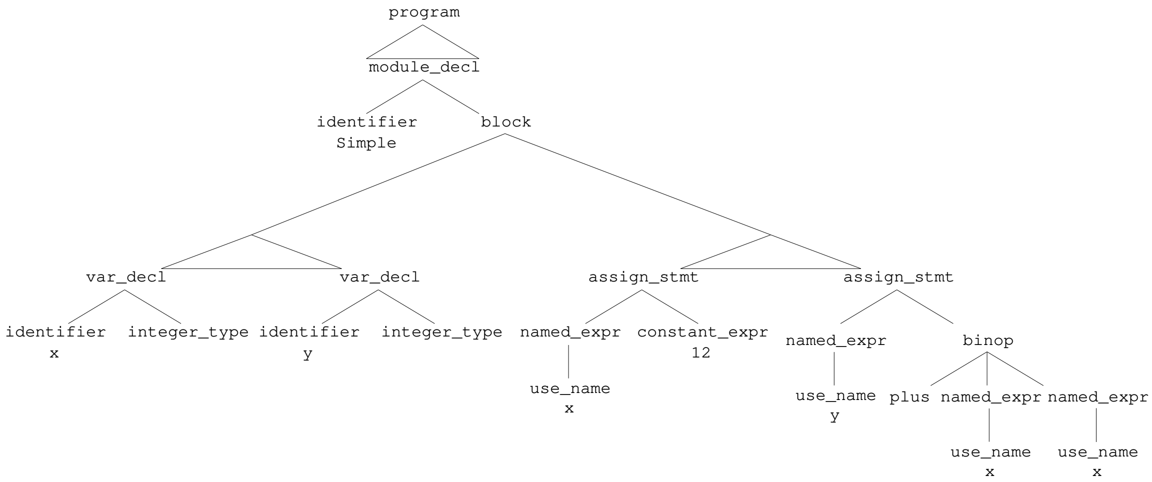


Figure 2.4: The result of evaluating the APS expression in Figure 2.3

2.1.2 Attributes

Attributes must be declared in APS. An attribute declaration names a phylum; each node of this phylum will be associated with an instance of this attribute. Attributes must have a declared type. Additionally a default value may be given. If no other definition of the attribute takes effect, the default value (if any) will be used. If there is no default, the attribute is *undefined*.

One may think of attribute declarations as adding a slot to every node of the phylum. If a names an attribute, and n is some expression that computes a node then n.a refers to the a slot of the node computed from n. Once a value has been determined for one of these slots, it does not change. Moreover the APS compiler ensures that an attribute will be used only after its value is determined.

For example, the following attribute declaration comes from the module that computes the types for expressions (Appendix B.3.3):

```
attribute Expression.expr_type : remote Type := any;
```

This fragment declares an attribute named `expr_type`. An instance of this attribute (having a value that is a remote[1] reference to a `Type` node) will be associated with each node of phylum `Expression`. The default value of each instance is `any`. This value (declared in another module) is used to indicate that the type could not be computed. The error reporting routines treat `any` specially to avoid cascading error messages.

Attribute definitions specify the values of attributes in particular contexts and under certain conditions. The context of an attribute definition is given by a pattern. The syntax used is `match` *pattern* `begin` *definitions* `end`. The following fragment computes the `expr_type` of Oberon2 `IS` expressions (type tests):

[1]See Chapter 5 for an explanation of remote types.

```
match ?e=is_test(...) begin
  e.expr_type := boolean;
end;
```

The pattern matches any node labeled with the `is_test` constructor. The `...` in the pattern means that we don't care what the children (if any) are. Pattern *variables* are defined in a pattern by preceding their names with a question mark (`?`). In this fragment, `e` is the only pattern variable. The equal sign (`=`) signifies that `e` is bound to whatever is matched by `is_test(...)`. There is only one definition in the body of this pattern match. It defines `expr_type` of an `is_test` node to be `boolean` (declared elsewhere).

An attribute may also be defined using the values of other attributes. For example:

```
match ?e=unop(?,?arg) begin
  e.expr_type := arg.expr_type;  -- for minus & not
end;
```

The pattern binds `e` to a node labeled by `unop` and binds `arg` to a child of this node. The subtree rooted at this child was the second argument passed to the constructor when the node was created. The first argument is ignored; the `?` notation is used as a placeholder. For the two unary operators in Oberon2 (`~` and `-`) the result type is the same as the argument type.[2] This example takes advantage of this fact by giving a single definition for both operators. As already mentioned, the APS compiler ensures that attributes are only used after they are defined. There is no danger that the definition of `e.expr_type` could be evaluated before `arg`'s `expr_type` definition. The APS compiler detects and complains about circular dependencies. One can think of the attribute definitions as being a system of constraints that are satisfied simultaneously.

Attribute definitions may also be conditional:

```
match ?e=aref(?array,?) begin
  case array.expr_type begin
    match array_type(?et) begin
      e.expr_type := et.base_type;
    end;
    ⋮
  end;
end;
```

The pattern binds `e` to an `aref` node and `array` to the first subtree child. An `aref` node is used for array references in Oberon2, for example `a[i]`. The body of the pattern match then examines the `expr_type` of the node bound to `array`. APS has a `case` statement for this purpose. The body of the `case` statement includes an example of *nested* pattern matching. The type is only interesting if it is an array type. Otherwise the expression is illegal, and since no other attribution clause applies to array references, the default (`any`) will be used.

[2] Of course, if the argument type were illegal, this fragment would not detect the error. For example, if the expression was `-"hello"`, it would blindly compute the type as `string`. Type checking is done in a different module.

In the case it is indeed an array type, this fragment defines the `expr_type` of the `aref` node. (Ignore the `base_type` attribute for now; it merely canonicalizes a type.) This definition is an example of *conditional attribution*; the attribute definition is only applicable under certain conditions. Otherwise, some other definition must be used. Chapter 3 discusses the interaction between pattern matching and conditional attribution.

2.2 Simple Extensions to Classical Attribute Grammars

In the following chapters, we describe some far-reaching extensions to the basics outlined here, but in this section we describe a few extensions that are both easier to explain and common to many attribute grammar systems.

2.2.1 Conditionals

Attribute grammar systems typically permit conditionals on the right-hand side of attribute equations. However, if a number of attributes depend on the same condition, in rare but significant cases, a static analysis that does not "understand" conditionals will discover a (spurious) attribute circularity. Therefore, APS recognizes conditionals as a fundamental construct. We have shown that standard static analyses of attribute grammars can carry over to conditional attribute grammars [12].

Conditionals in APS are expressed using an `if` 'statement', for example:[3]

```
match ?md=module_decl(?,block({...,import(?,?u=use_name(?name)),...},?))
begin
  mref : remote Declaration :=
      find_local_decl(name,module_scope);
  if mref == nil then
    u.use_decl := nil;
    u.no_decl_reason := "Undeclared module";
  elsif mref == md then
    -- Section 11: "A module must not import itself"
    u.use_decl := nil;
    u.no_decl_reason := "Illegal self-import";
  elsif not check_import(md,{},mref) then
    -- Section 11: "...cyclic import of modules is illegal"
    u.use_decl := nil;
    u.no_decl_reason := "Illegal indirect self-import";
  else
    u.use_decl := mref;
  endif;
end;
```

This fragment handles Oberon2 import clauses. It uses some of the advanced features explained in Chapters 3, 4 and 5. There are four difference cases for the compiler to handle,

[3] The Oberon2 compiler does not require conditional analysis to avoid spurious circularities.

each of which is handled by one of the four conditional clauses in the fragment. Each of the four clauses defines the attribute `using.use_decl` and all but the last define the attribute `using.no_decl_reason`. The second attribute is a string used as an error message when no (legal) module can be found for the import.

It is not necessary that all the definitions in the branch be evaluated at the same time. The only restriction is that none can be evaluated before the value of the condition is known.

2.2.2 Global Variables

A *global variable* is a named value computed using other global variables, functions or constructors. Global variables provide the ability to factor out a common value, or "magic number", and give it a name. From the Oberon2 compiler we have the following fragment:

```
module OBERON2_TYPE[T :: OBERON_TREE[]] extends T
begin
  type Oberon2Type = appropriate type
  -- some preconstructed types
  boolean : Oberon2Type := boolean_type();
  char : Oberon2Type := char_type();
  shortint : Oberon2Type := shortint_type();
  integer : Oberon2Type := integer_type();
  longint : Oberon2Type := longint_type();
  real : Oberon2Type := real_type();
  longreal : Oberon2Type := longreal_type();
  any : Oberon2Type := any_type(); -- used for an erroneous expression
end;
```

This module declares a type (`Oberon2Type`) and several global variables (`boolean`, `char`, etc). Section 2.1.2 gave examples of defining the `expr_type` attribute for expressions that used these declarations. Global variables must be assigned a value at their declaration site. A global variable may be used anywhere in its scope, that is in the module in which it is declared.

2.2.3 Local Variables

Rather than being global to a module, a variable may be declared local to any attribution clause:

```
  match ?b=block(?decls,?stmts) begin
    inner : Scope := nested_contour(b.scope);
    for decl in decls begin
      decl.scope := inner;
    end;
    for stmt in stmts begin
      stmt.scope := inner;
```

```
    end;
    b.saved_scope := inner;
  end;
```

This fragment is an attribution clause from the module that creates a symbol table for an Oberon2 program. The full text can be found in Appendix B.3.2. The attribution clause states that a new nested contour is declared for every block in an Oberon2 program. This new contour is saved in a local variable `inner`. A new contour defines a new scope. Both the declarations and the statements of the block have this scope. Symbol table records have object identity; every time `nested_contour` is called, it returns a new instance. Therefore, this fragment would have a different meaning if the definition of `inner` were substituted for all its uses:

```
  match ?b=block(?decls,?stmts) begin
    decls.scope := ScopeModule$nested_contour(b.scope);
    stmts.scope := ScopeModule$nested_contour(b.scope);   -- Wrong!
  end;
```

This fragment has the declarations and statements in different scopes. Therefore, local variables are important not only for factoring, but also to store shared objects.

In attribute grammar terminology, local variables are known as *local attributes*. We do not use this term, instead reserving the word "attribute" for values associated with nodes. The term "variable" is meant in the mathematical sense of an unchanging possibly unknown value, not in the sense of a store location with a changeable location as in imperative languages. In APS, a variable refers to an entity whose value may be determined by `:=` definitions (also called "assignments"). If a names an attribute, and n is some expression that computes a node then n.a is a variable. In attribute grammar terminology n.a is an *attribute occurrence*. The term "variable" is used in APS in order to separate the concept of a definable value from the concept of associating values with nodes.

2.2.4 Functions

Each attribute definition computes its values as a function of (possibly zero) other values. If the programmer could not define new functions, this paradigm would be very restrictive. Instead, as with any attribute grammar system, APS has function declarations. A simple function may be specified with an expression:

```
  function subset(x,y : Integer) : Boolean
      := logandc2(x,y) = 0;
```

This function (from Appendix B.1.1) determines if the first integer (interpreted as a bitset) is a subset of the second. This function uses the identity $x \subseteq y \iff x \cap \overline{y} = \emptyset$.

A more complicated function may be specified using an attribution clause. In the following example, the result of the function is given as a local variable (named by default `result`):

```
  function make_range(x,y : Constant) : Constants begin
```

```
  case Constants${x,y} begin
    match {some_integer_constant(?v1),
           some_integer_constant(?v2)} begin
      result := {shortint_constant(i) for i : Integer in v1..v2};
    end;
  else
    result := {nil};
  end;
end;
```

This function uses some of the features explained in Chapters 3 and 4. The parameters are placed in a list and then pattern matching is used to determine if each parameter is some form of an integer constant. If both of the parameters are integer constants, it creates the set of integer constants in the range. Otherwise it returns the set with `nil` in it.

2.2.5 Polymorphism

As mentioned in Chapter 1, polymorphism is an important factoring technique, especially in a typed language such as APS. APS uses the technique of *bounded polymorphism*, where any polymorphic entity is defined for types that fit certain *signatures* (sets of features, such as operations, that are available for the type). For example, types with the `NUMERIC[]` signature have a multiplication operation available, and so one could write a polymorphic function `square` that squares its argument:

```
[T :: NUMERIC[]] function square(x : T) : T := x * x;
```

The notation `[T :: NUMERIC[]]` declares a scope (including just the single function definition) in which `T` is a valid type. The function `square` is implicitly exported to the surrounding scope. Then at any point it is used, type inference must be able to determine the value of the implicit type parameter. In fact the infix multiplication operator `*` is defined in a very similar way. At its use in the preceding function definition, the value of its implicit type parameter is determined to be `T`, a valid type of the `NUMERIC[]` signature.

The only way for a type to satisfy a signature is for the module that creates it to assign it that signature, either explicitly or implicitly through the module's "extension" (see Chapter 8). The ability to extend types with new signatures makes this restriction less onerous.

Sometimes what is desired is not fully general polymorphism, but merely overloading: the ability to use the same name for a finite number of different types. APS provides *finite polymorphism* is which a type variable ranges over a given set of types. For example, the following fragment from the Oberon2 compiler symbol table module (Appendix B.3.2) has an example of a finite signature and a finitely polymorphic attribute:

```
signature SCOPABLE := {Declaration,Block,Header,Receiver,
                       Statement,Expression,Use,Type,
                       Case,CaseLabel,Element},
                      var PHYLUM[];
```

```
-- The scope for inserting declarations
[phylum T :: SCOPABLE] attribute T.scope : Scope := root_scope;
```

In this example, the `scope` attribute is declared for each of the phyla named in the `SCOPABLE` signature. This signature ranges over a number of phyla. The additional restriction `var PHYLUM[]` ensures that a type satisfying this signature can be decorated with attributes. Polymorphic attribute declarations must only be finitely polymorphic. Otherwise the system would not know how many different versions of the attribute could exist.

Finitely polymorphic attribution clauses are useful for factoring. For example, the following fragment from later in the same module copies the scope to the children of any node as long as both parent and child types belong to `SCOPABLE`:

```
-- otherwise we just copy scope to the child
[phylum P :: SCOPABLE;
 phylum C :: SCOPABLE] begin
  match ?parent:P=parent(?child:C) begin
    child.scope := parent.scope;
  end;
end;
```

Here `parent` is a special polymorphic constructor that can match any node of the appropriate type and bind one of the children of that node. By limiting the types `P` and `C` to phyla for which the `scope` attribute is defined, we ensure that the attribute definition in the block are properly typed.

2.3 Implementation

Attribute grammars permit sophisticated analysis and corresponding implementation techniques [23], and these techniques are directly transferable to APS. However, efficient static scheduling may require the compiler description writer to introduce contortions in the specification leading to dangerous redundancy. For example, even with a ground-breaking "fiber" analysis, Maddox's Modula2 description needs a separate syntax for constant expressions as well as the normal expression syntax in order to avoid static circularities [70]. Since the main goal of APS is to permit highly factored descriptions, any APS compiler would have to permit some attributes to be dynamically scheduled, despite such scheduling usually being less efficient. For simplicity therefore, the current APS compiler prototype uses dynamic scheduling throughout.

2.3.1 Variable Instances

The instantiation of a module creates instances of its global variables. The creation or prior existence of a tree node implies the existence of the corresponding attribute variable instances. The activation of an attribution clause with a given set of bindings brings into existence any local variable instances. Demand evaluation is used to compute the value for all these *variable instances*. Notice that local variable instances can be created during evaluation of other variable instances.

Global variable instances are created as soon as the module is instantiated. Each instance is associated with a *thunk* for its (required) default value, a parameterless function that when evaluated returns the value. Chapter 6 introduces a kind of global variable that can be assigned in attribution clauses, but for the purposes of this chapter, the value of a global variable instance is the same as its default value.

Once all the nodes of a phylum have been created, the attributes for that phylum are instantiated. The attribute variable instances are not stored in the nodes but stored in a vector indexed by a unique value stored in each node.

Then when the module is *finalized*, all the top-level match clauses in the module are activated for all nodes in the appropriate phyla. The activation of clauses may involve the instantiation of local variables and activation of definitions. When a definition is activated, a thunk to compute the right-hand side is placed on the worklist for that variable instance on the left-hand side. The textual priority of the definition is stored with the thunk. No instance evaluation is performed at this stage, only the activation of clauses and their definitions. A conditional clause that requires attribute evaluation is delayed.

2.3.2 Guards

In the case of a conditional definition, it cannot be determined until after evaluation has begun whether the definition will be activated or not. As a result, a delayed conditional clause is stored as a *guard thunk* on the worklists of variable instances that could be potentially assigned in its scope (other than local variable instances local to the conditional clause). Guard thunks are given higher priority than any definition thunk because their evaluation may add a definition to the variable.

Sometimes, it is not possible to determine the affected variable instance without further attribute evaluation. Since global and local variables are immediately known by their names, this situation can only occur with attribute variables in which the node being attributed is not known without evaluation. In this case, the guard thunk is put on the worklist of the *imprecise guard* variable instance for the attribute. All attribute variable instances are made to depend upon the imprecise guard for the attribute.

2.3.3 Demand Evaluation

All the variable instances created in a module instantiation are stored and then evaluated using demand evaluation. When a variable instance's value is demanded, it is marked as currently undergoing evaluation, and then its worklist is used to compute its value. First, all the guard thunks are evaluated to ensure all its definitions are activated. New guard thunks may be added while evaluating a guard thunk. Evaluation continues until all guards have been evaluated. Then the definitions are sorted and the one with highest priority (earliest textual position) is chosen and evaluated to yield the instance's value. This value is stored in the variable instance, which is then marked as having completed evaluation. Then the value is used in the place it was demanded.

During evaluation of either kind of thunk, further variable instances may themselves be demanded. If a variable instance that is marked as currently undergoing evaluation is demanded, a circular dependency has been exposed and the runtime system terminates evaluation with a diagnostic. The possibility of such a situation shows one major disadvantage of not using static analysis: the lack of circular dependencies cannot be ensured.

2.4 Looking Ahead

The features described in this chapter are sufficient to express classical attribute grammars in APS. Attribute grammars are useful in their own right, but as pointed out in Chapter 1, they have a number of deficiencies when used as compiler descriptions. The following chapters describe the major extensions in APS that address the problems.

Chapter 3

Pattern Matching

Pattern matching is one of the most important tools for factoring a formal compiler description. In classical attribute grammars, all factoring must be done through the case analysis provided by the abstract syntax. Rules are given for each production in the grammar. However, sometimes a concept needs a finer analysis. Some tasks need be done only for some instances of a production, such as when one of the subtrees has a particular shape. In this case, the attribute equations expressing this concept would be guarded by a condition that tests the shape.

Sometimes a concept needs a coarser analysis. Some concepts may apply generally to a whole class of productions. Using classical attribute grammars, such concepts would have to be expressed multiple times. This situation is often an instance of dangerous redundancy.

Often a concept needs both coarser *and* finer analysis than that provided by attribute grammars. That is, the definitions apply to multiple productions, but only for a special case of each. In this case, in a classical attribute grammar, the concept *and its condition* would have to be expressed multiple times, almost certainly being dangerously redundant. Pattern matching, on the other hand, can enable such concepts to be expressed in a single place.

This chapter first describes pattern matching in general. Section 3.2 discusses some of the issues that arise when pattern matching is added to an attribute grammar-like system. Section 3.3 describes how pattern matching is expressed in APS, including how the issues are addressed. Section 3.4 outlines a canonicalization that reduces pattern matching to conditional attribute grammars. This transformation is used in the APS compiler prior to descriptional composition.

3.1 Pattern Matching in Compilers

If a compiler is written in an imperative language with hand-written traversals of the tree, it is straightforward to use the conditionals of the implementation language to express a concept in one place. For example, to detect the incrementing of a variable written as an assignment, `i := i + 1`, one might write code such as:

```
if (stmt->operator == ASGN_STMT &&
    stmt->child[0]->operator == ID &&
    stmt->child[1]->operator == BINOP &&
    stmt->child[1]->child[0] == PLUS &&
    stmt->child[1]->child[1]->operator == ID &&
    stmt->child[1]->child[1]->id = stmts->child[0]->id &&
    stmt->child[1]->child[2]->operator == INTEGER &&
    stmt->child[1]->child[2]->iconst == 1) {
    /* handle an increment of a simple variable */
    ...;
}
```

Writing code like this can be tiresome and the code can also be tiresome to read. *Pattern matching* is a technique that allows such code to be expressed more concisely:

```
if (match(stmt,"ASGN_STMT[ID[?x],BINOP[!PLUS,ID[?x],INTEGER[1]]]")) {
    /* handle an increment of a simple variable */                    (1)
    ...;
}
```

The pattern gives a specification that the shape of the tree must have and the match function returns a boolean saying whether the pattern matches the given (sub)tree. Being in the form of a "picture," a pattern is often clearer than code that does the testing directly. Pattern matching has been implemented in a variety of systems; general purpose languages (for example, Prolog [20], ML [72] and Haskell [48]) and systems specifically tailored for compiler description (Twig [1], Optran [67], Trafola [45], Dora [13, 29], BURS [78] and *burg* [80]). In Prolog, pattern matching is realized in unification.

3.1.1 Pattern Variable Binding

Patterns may specify *variable bindings* and *value patterns*. Variable bindings are wild cards in the pattern. In the example, *name* is a pattern variable and `?`*name* is used to specify a variable binding. If a pattern-matching system is integrated into the language, then the variable bindings would be available as program-variable bindings. In formula (1), integration would mean that `x` would be a valid declared variable in the scope of the `{...}` block. A value pattern contains an expression to be evaluated at run-time. Matching with a value pattern only succeeds if the corresponding part of the tree is the *same*, in some sense, as the value of the value pattern. In formula (1), value patterns are written using `!`*value*. (Queinnec calls value patterns "eval" patterns [81]; Heckmann calls them "importing variable" patterns [45].)

If some pattern variable occurs more than once (such as `?x` in formula (1)) the pattern is *non-linear*. The meaning of such a repetition is that the corresponding values in the tree must be the same. In formula (1), the pattern only matches if the two identifier occurrences in the assignment statement are the same identifier. If two values in the tree are the same for the purposes of a non-linear pattern, but actually are distinguishable in some sense, then it becomes an issue as to *which* of the values is chosen for the binding. For example, we could generalize the previous pattern to handle all cases where an increment is performed, not only for plain variables but also for other assignable locations such as array accesses or record field accesses:

```
if (match(stmt,"ASGN_STMT[?place,BINOP[!PLUS,?place,INTEGER[1]]]")) {
    /* handle increment of "place" */
    ...;
}
```

The pattern matching system would presumably consider two subtrees to be the same if they were structurally identical (for example, if both had the shape `AREF[ID[x],INTEGER[3]]`). But the compiler might store annotations on nodes so it would make a difference whether `place` was bound to the first child of the assignment node or to the second child of the binary operation node. Non-linear patterns can be handled by treating each pattern variable normally the first time it is encountered in the pattern, and all successive times treating it as a value pattern. In effect, the pattern would be used as if it had been written with only one binding, with subsequence occurrences being value patterns:

```
if (match(stmt,"ASGN_STMT[?place,BINOP[!PLUS,!place,INTEGER[1]]]")) {
    /* handle increment of "place" */
    ...;
}
```

The pattern, of course, could have been written this way to begin with. Value patterns therefore subsume non-linear patterns.

3.1.2 Conjunctive and Disjunctive Patterns

A *conjunctive pattern* has two subpatterns, and matches if both subpatterns match the subtree at that position. One use of a conjunctive pattern is to use a pattern-variable binding to give a name to the subtree and also have a subpattern match the subtree:

```
if (match(stmt,"ASGN_STMT[?place,?rhs&BINOP[!PLUS,!place,?expr]]")) {
    /* handle increment of "place" by "expr"*/
    ...;
}
```

The pattern in this example matches an increment of a place expressed in an assignment node. In this example, the two subpatterns of a conjunctive pattern are joined by an ampersand (`&`). The pattern variable, `rhs`, is bound to the whole right-hand side expression tree of the assignment statement (in this case, the binary operation "PLUS" applied to the place and another expression), whereas `expr` is bound to a subtree of the right-hand side.

The logical counterpart to a conjunctive pattern is the *disjunctive pattern*, which matches if either of the subpatterns match. In the following example, a disjunctive pattern is expressed by separating the two subpatterns with `|`. Disjunctive patterns interact with pattern variable binding to cause a number of semantic difficulties. A pattern can cause problems if it attempts to bind variables inside the subpatterns of a disjunctive pattern:

```
if (match(expr,"ID[?x] | AREF[ID[?x],?index]")) {
        x is bound at this point.
        Is index bound at this point?
}
```

This pattern matches either simple identifiers or simple identifiers with an array subscript. The pattern variable `x` is bound in both subpatterns and thus it is reasonable to expect `x` to be bound inside the `{...}` block. On the other hand, `index` is bound only in one choice and so `index` might be bound or it might not. If variable named `index` is declared in an outer scope, then a direct implementation of such patterns leads to the unsettling semantics that this variable declaration sometimes is shadowed and sometimes not.

Another problem with disjunctive patterns is *non-determinism*. Say we have a pattern to detect addition by zero:

```
if match(expr,"BINOP[!PLUS,?x,INTEGER[0]] | BINOP[!PLUS,INTEGER[0],?x]") {
    /* replace "expr" by sub-expression "x" */
}
```

If the tree in question matches `BINOP[!PLUS,INTEGER[0],INTEGER[0]]` then either of the choices in this example could match. In this case, the question is whether `x` is bound to the first `INTEGER[0]` or to the second one.

Because of these difficulties, pattern matching systems with disjunctive patterns, often forbid (or ignore) pattern variable bindings within the subpatterns (for example, DORA) or require all choices to bind the same set of variables (Trafola for example). Moreover, non-determinism is often handled by choosing only the "first" matching choice. Prolog is an exception here as the language directly supports non-determinism. In Prolog, if a pattern matches in multiple ways, each choice of bindings is used. If a choice leads to a failure at some later point, the next choice is tried.

Negated patterns are a further logical extension. Pattern variable binding interacts even worse with negated patterns. In systems with negated patterns, bindings inside a negated pattern are not visible outside the pattern.

3.1.3 Variable-Depth Patterns

Despite having the power to specify conditions for multiple tree nodes, patterns in many formalisms do not normally have the power to match an arbitrary subsection of a tree. For example, in most pattern-matching systems, it is not possible to match a procedure header node and a return statement node that is arbitrarily deep in the procedure body in a single pattern. A few powerful pattern matching systems can handle such situations. Trafola has a non-deterministic construct (the insert pattern) in which an arbitrarily deep subtree may be matched:

```
P ^ Q
```

This pattern will match any tree. The pattern variable `Q` is bound to some subtree anywhere within the tree being matched and `P` is bound to a tree with a "hole" where `Q` was. A tree with a hole can be combined with a subtree to fill the hole. The insert construct is useful for tasks such as our example of matching a procedure header and return statements, but it does not provide a way to specify the shape of the intervening nodes.

Similar constructs are Farnum's "vertical iterators" [29] or Queinnec and Geffroy's "tree patterns" [82]. These constructs have two parts, a *wrapper* with a "hole" and a *termination* pattern. A subtree matches a vertical iterator if it matches the termination or matches the wrapper with the subtree at the "hole" recursively matching the whole iterator. In essence, the wrapper is copied multiple times as needed. By having a wrapper, therefore, a vertical iterator pattern specifies the shape of all the intervening nodes, but having only one wrapper makes it difficult to allow different types of intervening nodes. Queinnec's system includes unrestricted disjunctive patterns and thus does not have this problem.

All three systems allow bindings in the terminator pattern. Farnum's vertical iterators also allow bindings in the wrapper. A variable is bound to the list of all the bindings in all the instances of the wrapper. Queinnec and Geffroy's system handles non-linear pattern matching; multiple instances of the same pattern variable are matched against each other. Farnum's vertical iterators are made deterministic using the "maximal munch" rule; the largest tree to match is chosen. The other systems are non-deterministic.

3.1.4 Semantic Conditions

A pattern cannot always express all the conditions under which some compilation task applies. Often it is necessary to have additional conditions, called *semantic conditions* to contrast with the *syntactic condition* which is the pattern itself. (Heckmann calls semantic conditions "where" patterns, whereas Queinnec calls them "check" patterns).

Value patterns can be thought of as a limited form of semantic condition because they rely on comparing with a run-time value, not with some structure spelled out in the pattern. Pattern matching systems differ as to whether semantic conditions can be integrated into patterns. In a pattern matching system embedded in a general-purpose programming language, it is not usually necessary to have integration, since the programming language usually has constructs (such as "if" statements) to do the testing. Some formal systems have no support for semantic conditions. In *burg* and BEG, it is necessary to translate semantic conditions into syntactic ones or not have them at all. For example, instead of using a semantic condition to specify that a certain tree matches only if some integer constant in it has the value 1, the tree representation must be changed to represent integer constant nodes with value 1 as instances of a special "one" node. This method has limitations and can be clumsy if it actually requires the tree to be transformed to add these special nodes.

3.1.5 Sensitivity to Change

While pattern matching helps factor a description by concept, patterns are very sensitive to changes in the tree structure itself. If a single rule in the abstract grammar describing the structure of the tree changes, it may require widely separated changes in the description. Thus the benefit of pattern matching for factoring must often be balanced against the disadvantage of sensitivity to the tree structure.

Ideally, one would like a factoring method that reduces the sensitivity to changes in the tree structure. Wadler [98] and more recently Palao Gostanza *et al.* [75] have introduced ways to name patterns over trees. Section 3.3.1 introduces a pattern matching construct, the *pattern definition*, that generalizes these approaches.

3.1.6 Theoretical Results and Implementation

Pattern matching on trees has been studied extensively, pattern-variable binding less so. Hoffmann and O'Donnell [47] showed that pattern matching with wild cards (but no pattern variables) including conjunctive, disjunctive and negated patterns can be done in time linear in the size of the tree with a bottom-up tree automaton. More precisely, given a set of patterns that can be preprocessed off line, every match site in the tree of every pattern can be determined in time linear in the size of the tree and in the number of successful matches. Chase [18] gave a faster algorithm for the off-line processing than Hoffmann and O'Donnell. Cai et al [17] have improved the speed ever more. Moreover, Trafola's insert patterns and DORA's vertical iterators can also be matched using bottom-up tree automata.

Farnum has shown that pattern matching can be used in an attribute grammar-like formalism and that linear-time bottom-up tree matching can be used to do the matching and binding. He showed that these *attribute pattern sets* can be converted to classical attribute grammars, although it is not possible to determine statically whether every attribute is defined [31].

3.2 Pattern Matching in Declarative Descriptions

When pattern matching is to be incorporated into a declarative framework, certain issues come up that do not arise with compilers written in a general-purpose programming language. The most basic issue is what happens when multiple patterns match or more importantly what happens when attribute definitions controlled by different patterns conflict. Other issues come up when patterns may match in multiple ways. This section discusses these issues.

3.2.1 Conflicting Attribute Definitions

One way of handling conflicting attribute definitions is to forbid the situation from occurring. It would be an error when two attribute definitions for the same attribute apply to the same node in a tree. If a pattern-matching system uses this rule, negated patterns are essential in order to exclude possibilities. Forbidding conflict, however, violates the principle of factoring because a special case must be expressed in two places: once positively to guard the special attribute definitions and once negatively to guard the normal case.

In an imperative language, one could write something like

```
if match(stmt, "ASGN_STMT[?place,BINOP[!PLUS,?place,INTEGER[1]]]")) {
    /* handle increment of "place" */
    ...;
} else if (match(stmt,"ASGN_STMT[?place,BINOP[!PLUS,!place,?expr]]")) {
    /* handle increment of "place" by "expr"*/
```

```
    ...;
} else if (match(stmt,"ASGN_STMT[?place,?expr]")) {
    /* handle assignment of "place" */
    ...;
} else ...
```

and it is perfectly clear that the later patterns do not apply if the first one matches. One attempt to transfer this method to declarative descriptions would be for each tree node to use the definitions guarded by the first pattern that matches it. This rule is flawed in several ways. First, every set of definitions guarded by a pattern would have to define all the attributes. Thus a single case analysis must be used for all attributes. This situation is possibly better than using an arbitrary case analysis, but it does not not take into account the fact that different kinds of subtasks in a compiler may require different case analyses. One way to ameliorate this situation is to define a different set of guarded definitions for each subtask. This fix, however, unless made considerably more complicated, does not allow attributes with slightly different case analyses to share common cases.

Another serious problem with picking the first matching pattern is that it does *not* avoid conflicting definitions if attributes can be defined for nodes other than the root of the pattern. For example, assume pattern-guarded attribute definitions are used to generate different code for identifier references on the left hand of assignment statements:

```
ASGN_STMT[?expr&ID[?id],?rhs]
        expr.code := "fetch address " ++ id;
?expr&ID[?id]
        expr.code := "fetch " ++ id;
```

These definitions define the code to be generated for a subtree of the form `ID[?id]`. In the context of an assignment, the first definition should be used; otherwise the second is to be used. However, the first definition matches `ASGN_STMT` nodes, not `ID[?id]` nodes. And so when determining the set of definitions for an `ID[?id]` tree, only the second definition matches, even in the context of an assignment. As a result, by the "first matching pattern" rule, the `code` attribute will be defined *twice* for `ID[?id]` subtrees on the left-hand side of assignments. The problem is that even though the patterns match different nodes, the guarded attribute definitions apply to the same node.

If one wishes to add pattern matching to a declarative formalism that, like attribute grammars, permits attribution of nodes other than that matched by the root of the pattern, resolving conflicting attribute definitions must be done in a different way. Rather than prioritizing the patterns, the attribute definitions themselves can be prioritized. The highest priority definition of an attribute is the one used. Farnum's attribute pattern sets [31] and Dueck and Cormack's module attribute grammars [26] use this rule. Pattern-attribution clauses are ordered by their textual appearance and each attribution clause guards a number of attribute definitions. If two attribute definitions apply to the same node in a tree, the textually earlier attribute definition is the one that holds.

One could also use a dynamic definition for "first." For example, "first" could mean the first definition found in a preorder pattern-matching traversal of the tree. The problem with such a definition of "first" is that it can be rather difficult to determine statically when a definition will be overridden by another. Moreover this definition of "first" seems too much tied to a particular implementation method.

In summary, if one wants to add pattern matching to an attribute grammar-like formalism, it seems the only useful policies for resolving attribute definition conflicts are to disallow conflict altogether (with the loss in factoring) or to use some version of Farnum's textual ordering rule.

3.2.2 Multiple Matches

If the pattern that guards some attribute definitions could match in more than one way, ambiguity may arise. For example, what would be the meaning of a pattern-guarded attribution clause such as the following?

```
?expr & (BINOP[!PLUS,?x,INTEGER[0]] | BINOP[!PLUS,INTEGER[0],?x])
        expr.code := x.code;
```

If the tree in question takes the form `BINOP[PLUS,INTEGER[0],INTEGER[0]]`, then either possibility matches. Presumably, one would want some sort of disambiguation that chooses one particular match over the other. Otherwise, if both possibilities were used, `expr.code` would have two conflicting definitions, corresponding to the two possible bindings for `x`. However, sometimes one would like all possible bindings to apply:

```
?expr & (BINOP[?_,?x,?_] | BINOP[?_,?_,?x] | UNOP[?_,?x])
        x.env := expr.env
```

(Here `_` is a "don't care" pattern variable in the manner of Prolog.) This fragment defines the environment attribute of a child of a unary or binary expression node to get the parent's environment attribute. If some disambiguating rule is used to choose between the first two cases (that both match all `BINOP` nodes), then one of the two children of each `BINOP` node will not get a definition of the parent's `env` attribute.

Thus one may wish to use such patterns in two different ways. On the one hand, sometimes a disambiguating rule is desired so that only one binding is used. On the other hand, sometimes it is useful to use all bindings (in the spirit of Prolog). Using all bindings is more elegant that using just one binding, but even in Prolog (in which all bindings are attempted if necessary), there is a method to force a choice when desired.

3.3 Pattern Matching in APS

Our pattern matching system is inspired by Farnum's attribute pattern sets. The pattern matching system in APS includes pattern variables, value patterns and conjunctive patterns, as well as a restricted form of disjunctive patterns subsumed in the novel construct of *pattern definitions*. Semantic conditions are attached to patterns using `if`. Non-linear patterns are not permitted; instead, value patterns must be used. APS does not include negated patterns. It seems cleaner to specify things positively. Moreover Farnum's conflict-resolution rule makes negated patterns less necessary.

```
-- handle binop's in three cases:
-- predicate operators are easy---the result is always boolean;
-- divide (/) is special---it always forces a real result;
-- otherwise we choose the least common type.
match ?e=binop(predicate_operator(),?,?) begin
  e.expr_type := boolean;
end;
match ?e=binop(divide(),?e1,?e2) begin
  case Types${e1.expr_type,e2.expr_type} begin
    match {...,?ty=longreal_type(),...} begin
      e.expr_type := ty;
    end;
    match {...,?ty=set_type(),...} begin
      e.expr_type := ty;
    end;
  else
    e.expr_type := real;
  end;
end;
match ?e=binop(arithmetic_operator(),?e1,?e2) begin
  case Types${e1.expr_type,e2.expr_type} begin
    match {...,?ty=longreal_type(),...} begin
      e.expr_type := ty;
    end;
    match {...,?ty=real_type(),...} begin
      e.expr_type := ty;
    end;
    match {...,?ty=longint_type(),...} begin
      e.expr_type := ty;
    end;
    match {...,?ty=integer_type(),...} begin
      e.expr_type := ty;
    end;
    match {...,?ty=shortint_type(),...} begin
      e.expr_type := ty;
    end;
    match {...,?ty=set_type(),...} begin
      e.expr_type := ty;
    end;
  end;
end;
```

Figure 3.1: An Example of Pattern Matching from Appendix B.3.3

Patterns are used at the top-level to guard attribution clauses, and inside each clause, pattern matching can be used to make further distinctions. These *nested matches* permit factoring of common parts of patterns into a single top-level pattern. All matches (that is, all possibilities of pattern variable bindings) are used in top-level matches. Nested matches may either use all matches (if introduced by the keyword `for`) or use the first match of the first pattern that matches (if introduced by the keyword `case`). The *first* match is found by attempting all choices of a disjunctive pattern in order, top down through the pattern. A `case` clause may also include a default clause (labeled with the `else` keyword) to be used when none of the patterns match. The keyword `case` was chosen for pattern matches that use the first match to emphasize the fact that it performs a case analysis of the tree. The keyword `for` was chosen for pattern matching when all matches are to be used to emphasize the fact that the body of the match may be executed multiple times.

Figure 3.1 provides an example of pattern matching in APS. There are three top-level definitions that compute the type of binary operator expressions. The first attribution clause is the simplest. If the operator is a predicate operator (such as "`<`"), the type of the expression is boolean.

The second clause handles division (`X/Y`). In Oberon2, arithmetic is always done in the maximum precision of the two operands. However, dividing two integers always returns a real. (Integer division is accomplished with the `DIV` operator.) The clause performs a nested match of the list of types of the two operands. Since the `case` keyword is used, the first match is used. The `{...}` notation is explained more fully in Chapter 4. Here, it means that the pattern `{...,?ty=longreal(),...}` matches if either of the operands has `LONGREAL` type. In this case, the result will be a `LONGREAL`. Otherwise, the clause checks if the operation is one on sets (`X/Y` denotes the symmetric set difference of sets `X` and `Y`). If neither of these cases hold, the only (legal) possibility is that the operands are reals or some variety of integer. In this case, the result will be a real and so the `else` clause for the nested match defines the type to be real.

The third clause is similar. As with the clause for `divide`, this clause contains a nested match on the types of the operands. In Oberon2, the numeric types are logically nested:

$$\texttt{LONGREAL} \supset \texttt{REAL} \supset \texttt{LONGINT} \supset \texttt{INTEGER} \supset \texttt{SHORTINT}$$

Arithmetic is done in the smallest numeric type that includes both operands. Thus if either operand is a `LONGREAL`, the result is a `LONGREAL`. Otherwise, the same rule applies to `REAL`, and then down the hierarchy for `LONGINT`, `INTEGER`, and `SHORTINT`. Again, we must handle the case of sets (`A+B`, for example, denoting the union of two sets). The `divide` operator is included in the set of arithmetic operators and thus a division will match both clauses. But the attribute definitions in the second clause will be higher priority than those in the third clause.

The pattern syntax of APS provides a number of minor conveniences. For instance, the children of a constructor can be ignored by using ... to substitute for whichever children there might be:

```
match ?e=is_test(...) begin
  e.expr_type := boolean;
end;
```

If the constructor definition needed to be changed to have more or fewer children, or to have them in a different order, this pattern would still be valid.

Another way in which dependence on the specific form of the tree can be reduced is to use *keyword parameters* to the constructors:

```
-- the block is situated in the scope declaring the receiver and formals
-- Since the body is a "block", local declarations can shadow
-- outer declarations.
match proc_decl(header(receiver:=?rec),?body) begin
  body.scope := rec.scope;
end;
```

This fragment uses the full Oberon2 abstract syntax from Appendix B.1. The `header` constructor takes four parameters, but when writing this fragment, it wasn't necessary to remember the order or what they were, it was only necessary to remember that the one of interest for scoping is the receiver. This parameter is the second one for the constructor, and thus this attribution clause is equivalent to the following:

```
match ?d=proc_decl(header(?,?rec,?,?),?body) begin
  body.scope := rec.scope;
end;
```

Using the name rather than the position not only makes the pattern less dependent on the exact shape of the tree, but is also less prone to error.

3.3.1 Pattern Definitions

Disjunctive patterns and an extension of Farnum's vertical iterators are combined in APS under the novel concept of *pattern definitions*. Pattern definitions are named disjunctive patterns that may take patterns as arguments. They are declared by giving all the alternatives, separated by commas:

```
pattern logical_operator() : Operator := log_or(),log_and(),log_not();
pattern integer_operator() : Operator := mod(),div();
pattern arithmetic_operator() : Operator :=
    plus(),minus(),times(),divide(),integer_operator();
pattern equality_operator() : Operator := equal(),not_equal();
pattern comparison_operator() : Operator :=
    equality_operator(),less(),less_equal(),greater(),greater_equal();
-- predicates are things returning boolean values:
pattern predicate_operator() : Operator :=
    logical_operator(), comparison_operator(), in_set();
-- NB: predicate_operator and arithmetic_operator between them
-- cover the space of operators.
```

None of these examples have parameters. Even without parameters, pattern definitions have two uses; they express disjunctive patterns and they provide factoring. For example, in Figure 3.1, it was not necessary to list all the predicate binary operators. Moreover, patterns such as `arithmetic_operator` are used in more than one location in the compiler description. If APS had only disjunctive patterns without the ability to factor out common patterns, a description could be forced into dangerous redundancy.

Pattern definitions may also take parameters. A pattern definition taking parameters may mimic a constructor:

```
pattern array_type(element_type : Type) : Type
    := open_array_type(?element_type),fixed_array_type(?,?element_type);
```

This pattern matches either of the two array types in Oberon2. Each choice in a pattern definition must bind all the parameters. This pattern definition highlights two aspects of factoring. First, since it can match either of the two array types, only one equation is needed where two might otherwise be needed. The previous chapter had this example:

```
match ?e=aref(?array,?) begin
  case array.expr_type begin
    match array_type(?et) begin
      e.expr_type := et.base_type;
    end;
    ⋮
  end;
end;
```

Without the ability to express both possibilities in a single pattern, it would be necessary to duplicate code:

```
match ?e=aref(?array,?) begin
  case array.expr_type begin
    match open_array_type(?et) begin
      e.expr_type := et.base_type;
    end;
    match fixed_array_type(?,?et) begin
      e.expr_type := et.base_type;
    end;
  end;
end;
```

Thus, the fact that a pattern definition is a *disjunctive pattern* means that only one attribution clause is needed. Secondly, a pattern definition is *named*, and so can be used multiple times. For example, Appendix B.5 includes several other uses of `array_type` including the check that pointer types be declared only for arrays and records:

```
-- "[The pointer base type] must be a record or array type"
match pointer_type(?base) begin
  case base.base_type begin
    match any_type() begin end; -- avoid error cascades
```

```
    match array_type(...) begin end;
    match record_type(...) begin end;
  else
    indicate an error
  end;
end;
```

If for some reason, a new array type were added, a new choice could be added to `array_type`. The uses of `array_type` would then all work for the new array type as well. Pattern definitions thus permit a pattern-based description to be less dependent on the precise structure of the tree.[1]

Pattern definitions may also be recursive. Recursion gives the power to express Farnum's vertical iterators and Queinnec's tree patterns. The Oberon2 compiler does not use any (user-defined) recursive pattern definitions[2]. A simple example of what can be accomplished is a recursive pattern definition to match an arithmetic expression that includes a constant:

```
pattern expression(constant_operand : Expression) : Expression =
  ?constant_operand=constant_expression(...),
  unop(?,expression(?constant_operand)),
  binop(?,expression(?constant_operand),?),
  binop(?,?,expression(?constant_operand));

match assign_stmt(?lhs,?rhs=expression(?const)) begin
  ⋮
end;
```

In the body of the `match`, `rhs` will be bound to an arithmetic expression that includes constants, one of which is bound to `const`. Because a constant may occur as either operand of a binary operator, the pattern has multiple possible matches. Pattern definitions are similar to Palao Gostanza's "active destructors" but the latter may only yield one match (even if the surrounding match then fails).

As already mentioned, in top-level pattern matches in APS, every possible binding is used. Any definitions in the `match` in the preceding example must take this nondeterminism into account. Attribute definitions guarded by multiple match patterns must be written to avoid conflicts; APS provides no method for disambiguating between two instances of the same attribute definition:

```
match assign_stmt(?lhs,?rhs=expression(?const)) begin
  lhs.uses_constant := const.value;    -- error!
  const.used := true;                  -- OK
end;
```

[1] As with constructors, keyword parameters and the ... notation for ignoring parameters may be used.
[2] The APS compiler in Appendix C *does* however have several user-defined recursive pattern definitions.

The first definition is erroneous because it could apply multiple times, leading to multiple (conflicting) definitions of the attribute `lhs.uses_constant`. On the other hand, the second definition is legal because each constant literal in the expression is matched in only one way. Section 3.3.3 describes how the APS compiler determines whether an attribute definition may conflict with itself.

Patterns that may match in multiple ways can also cause problems for semantic conditions (including value patterns) even if the first match is to be used. The problem is that the condition may affect the way in which the resulting match is selected. For example, say we had a nested match that tests if two expressions are being added that share the same constant:

```
match assign_stmt(?lhs,?rhs) begin
  case rhs begin
    match binop(plus(),expression(constant_expression(?value)),
                       expression(constant_expression(!value)))
    begin
      lhs.uses_constant_twice := value;
    end;
  end;
end;
```

Although the first match is used, the pattern match cannot proceed without logical backtracking because the binding of a value in the first term may not have a matching value in the second term. In particular, the preceding fragment is *not* equivalent to the following fragment:

```
match assign_stmt(?lhs,?rhs) begin
  case rhs begin
    match binop(plus(),expression(constant_expression(?value)),
                       expression(constant_expression(?value2)))
    begin
      if value = value2 then
        lhs.uses_constant_twice := value;
      endif;
    end;
  end;
end;
```

The second fragment merely tests if the first constant in each term is the same (and therefore will fail in cases where the first succeeds). If `for` were used in both fragments, they *would* be equivalent.[3] Section 3.3.4 explains the restriction on semantic conditions and value patterns that ensures that `case` matches can always be implemented without backtracking.

[3] Unfortunately, both would also be illegal, because the definition of `lhs.uses_constant_twice` would conflict with itself. Chapter 6 introduces some attribute definitions that would be legal in the context of such `for` statements.

3.3.2 Limits on Pattern Definition Recursion

Recursion in pattern definitions is limited so that the set of the subtrees matched can be determined using a finite-state bottom-up tree automaton. The set of such trees is the set of *regular trees* as defined by Courcelle [22]. Mutual recursion is not permitted, and even direct recursion is limited. Pattern definitions must be *bottom-linear*; that is, all of the parameters must be passed in order unchanged to any recursive call. In other words, if the pattern definition p has arguments $a_1, a_2, \ldots, a_n$, the recursive call must have the form p(?a_1,?a_2,...,?a_n). In type theory, Solomon [88] uses similar restrictions to ensure infinite types are *regular types*. This restriction prevents a pattern definition from requiring long-distance matching. The following pattern definition is illegal:

```
pattern far(e : Expression) : Expression =
  binop(times(),far(binop(plus(),?e,?)),?), -- illegal: not bottom-linear
  ?e;
```

This pattern definition binds its parameter to an expression that is nested by n multiplication and n addition operations:

$$(e+y_1+y_2+\ldots+y_n)*x_1*x_2*\ldots*x_n$$

It is illegal because instead of having a recursive call of the form `far(?e)`, it has an additional matching requirement on the actual parameter.

The term "bottom-linear" is used to evoke the relationship between left-linear (and right-linear) context-free grammars and regular expressions. In a *right-linear* context-free grammar, a non-terminal can only be in the right-most position of a production:

A → a A	right-linear
B → a B b	not right-linear

Left-linear context-free grammars are defined analogously. Context-free grammars describe a strictly larger class of languages that regular expressions, but left-linear and right-linear context-free languages describe exactly the same set of languages as regular expressions.[4]

Although APS requires recursive pattern definitions to be bottom linear, it is interesting to note that *top-linear* pattern definitions behave differently from bottom-linear ones. In a top-linear pattern definition, a recursive call must occur at the top level. For example, the following pattern definition matches any assignment statement and binds its argument to the right-hand side or to any factor of the right-hand side:

```
pattern assigned_factor(x : Expression) =  -- top-linear
    assigned_factor(binop(times(),?x,?)),
    assigned_factor(binop(times(),?,?x)),
    assign_stmt(?,?x);
```

[4] In the usual definition of right-linear context free grammars, mutual "recursion" is permitted but a nonterminal may only occur at the end of a rule. Bottom-linearity could have been similarly defined, but the given rule is more flexible.

This pattern definition is not bottom-linear, but it is top-linear. Recursion in a top-linear pattern definition is useless unless there are parameters. Top-linear pattern definitions can be used to describe all the same sets of trees as bottom-linear pattern definitions. Moreover, they are more powerful, although not as powerful as unrestricted pattern definitions:

```
pattern matched(x,y : Expression) : Expression = -- top-linear
    matched(binop(times(),?x,?),binop(times(),?y,?)),
    binop(plus(),?x,?y);
match ?x=matched(expr_constant(?),expr_constant(?)) begin
  ...
end;
```

The match statement matches the sum of two terms each of which has the *same* number of factors and each of which starts with a constant. Such a pattern cannot be matched using a bottom-up finite state automaton, because it would not be possible to keep track of how many factors each term had. A top-down automaton could perform the matching if it could traverse two sections of the tree in parallel.

3.3.3 Checking Self-Conflict

Lexical ordering is used to resolve conflicts between two definitions that originate from different rules in the specification. However, as stated earlier (on page 49), there is no mechanism in APS for resolving two conflicting instances of the same attribute definition. However, a static check can be performed that ensures that a definition will not conflict with itself. Unfortunately, this check is complex. The complexity is comparable with the definition of LALR(1) necessary for specifying context-free grammar based parsers. Theoretically, anyone specifying a parser using a LALR(1)-based parser generator should understand the technical definition of LALR(1), but usually one can avoid the details. Similarly, most of the time it is not necessary to know the full details of the analysis specified in this section.

A variable binding is said to be *controlling* for an attribute definition if for all match possibilities, every instance of the definition (due to different bindings for pattern variables) has a different value for this variable. In other words, each way that the guarding patterns match in a tree yields a different value for this variable. Attributes may only be defined for variables with controlling bindings. This rule guarantees that no definition can conflict with itself.

It usually works to assume that every binding is controlling everywhere except in the body of a `for` nested inside the scope of the binding. However, this approximation is not safe. The APS compiler uses a static definition of controlling that safely approximates the true definition. This section describes the basics of the static definition. For simplicity, assume that pattern definitions are expanded at all their call sites into the form:

`pattern (`*formals*`)`*body*`(`*actuals*`)`

Choices are separated by `|` and any recursive calls of bottom-linear pattern definitions are replaced with `@`—these features are not legal APS syntax; they merely serve as a notation for an internal form. Moreover, value patterns `!`*expr* can be replaced with a binding and a semantic condition: `?v2734 & if v2734=`*expr*. Using these canonicalizations and ignoring the types of nodes, any pattern can be expressed in one of the following forms:

$$
\begin{array}{ll}
\phi ::= & \texttt{?}\mathit{id} \\
& \texttt{if}\ \mathit{expr} \\
& \phi'\ \texttt{\&}\ \phi'' \\
& \phi'\ \texttt{|}\ \phi'' \\
& \mathit{constructor}\texttt{(}\phi_1\texttt{,}\phi_2\texttt{,}\ldots\texttt{,}\phi_n\texttt{)} \qquad n \geq 0 \\
& \texttt{pattern}\ \texttt{(}\mathit{id}_1\texttt{,}\mathit{id}_2\texttt{,}\ldots\texttt{,}\mathit{id}_n\texttt{)}\phi'\texttt{(}\phi_1\texttt{,}\phi_2\texttt{,}\ldots\texttt{,}\phi_n\texttt{)} \qquad n \geq 0 \\
& \texttt{@}
\end{array}
$$

The structure of a pattern can thus be described as a tree, and one can speak of going "up" (toward the root) or "down" (away from the root) within the pattern.

This section defines sets associated with each subpattern of a pattern that will enable a definition of ambiguity. If the pattern is unambiguous in certain ways, then a pattern variable will be known statically to be controlling.

For each subpattern, ϕ, two sets are defined; they represent the *constructors* and the *positions* of the pattern. The set of constructors for a pattern ($C(\phi)$) is the set of constructors that could label a subtree matched by ϕ. The set of positions of a pattern ($P(\phi)$) is the set of positions that the subtree being matched by ϕ could assume in a matched tree.

$$C(\phi) \subseteq \mathit{Constructors}$$

$$P(\phi) \subset \mathit{Constructors} \times \{1, 2, \ldots\}$$

(where *Constructors* is the set of all constructors, and $\{1, 2, \ldots\}$ is the infinite set of natural numbers). These sets are used to determine whether a pattern is ambiguous if the matcher can only look at one node label in the tree (that is, one constructor) at a time.

The definitions of the sets are given for each possible form of the pattern. The notation here approaches that of an attribute grammar. See Appendix C.5.1 for the realization in APS. The sets of constructors are computed bottom-up and the sets of positions are computed top-down. These equations and set constraints are potentially circular, and so the least fixpoint is used.

If a pattern variable occurs as part of a pattern, then as long as it is not a formal parameter to a pattern definition, nothing is known about the set of constructors it could match:

$\phi ::=$ `?`*id* when *id* is not a formal parameter

$$C(\phi) = \mathit{Constructors}$$

Similarly, semantic conditions can occur anywhere, and so again, nothing is known about the constructors:

$\phi ::=$ `if` *expr*

$$C(\phi) = \mathit{Constructors}$$

Conjunctive patterns take the intersection of constructor sets and propagate the positions to both subpatterns:

$\phi ::= \phi'$ `&` ϕ''

$$C(\phi) = C(\phi') \cap C(\phi'')$$
$$P(\phi') = P(\phi)$$
$$P(\phi'') = P(\phi)$$

A disjunctive pattern is similar, but in this case the constructor sets are unioned.

$\phi ::= \phi'$ `|` ϕ''

$$C(\phi) = C(\phi') \cup C(\phi'')$$
$$P(\phi') = P(\phi)$$
$$P(\phi'') = P(\phi)$$

In a simple constructor call, the set of constructors and the set of positions can be calculated exactly.

$\phi ::= \mathit{constructor}(\phi_1, \phi_2, \ldots, \phi_n)$

$$C(\phi) = \{\mathit{constructor}\}$$
`for` $i \in \{1, \ldots, n\}$
$$P(\phi_i) = \{(\mathit{constructor}, i)\}$$

For an inlined pattern definition, the set of constructors includes all the constructors for the choices in the body. The positions of the choices include the positions for the whole pattern, but also include the positions for all of the recursive calls in the body. The constructors for any recursive call are those for the call as a whole. The constructors for each formal parameter binding are the constructors for the corresponding actual parameter. The positions for each actual parameter includes the positions for each corresponding formal parameter binding:

$\phi ::=$ `pattern` $(id_1, id_2, \ldots, id_n)\phi'(\phi_1, \phi_2, \ldots, \phi_n)$

$$C(\phi) = C(\phi')$$
$$P(\phi') \supseteq P(\phi)$$
`for each` $\phi^{@} = \texttt{@} \in \phi'$
$$C(\phi^{@}) = C(\phi')$$
$$P(\phi') \supseteq P(\phi^{@})$$
`for each` $i \in \{1, 2, \ldots, n\}$
`for each` $\phi'_i = \texttt{?}id_i \in \phi'$
$$C(\phi'_i) = C(\phi_i)$$
$$P(\phi_i) \supseteq P(\phi'_i)$$

Finally, if the match comes from a known place, the positions can be copied from there:

```
match ... ?id ... begin
  ...
    for/case id begin
    ⋮
```

```
        match ϕ begin ... end;
      ⋮
      end;
  . . .
  end;
```

$P(\phi) = P(?id)$

The constructors $C(\phi)$ cannot be similarly "copied back" because pattern matching is not required to be exhaustive.

In all other cases, the positions of a top-level pattern are unknown:

$$P(\phi) = \mathit{Constructors} \times \{1, 2, \ldots\}$$

All these equations are combined and the least fixpoint is taken. After doing this step, it is possible to make a further pass to refine the definitions of $C(\phi)$ to take into account conjunctive patterns. The algorithm given in Appendix C.5.1 takes this approach. As it happens this simplification does not affect the following definitions of ambiguity, but it does make a big difference for the canonicalization in Section 3.4.

The constructor and position sets are used to determine whether a pattern is "one constructor look-ahead" (OCLA) ambiguous for top-down or bottom-up tree parsing. Intuitively a pattern is top-down OCLA-ambiguous if when "traveling" down the pattern, one cannot tell from the constructor on the tree being matched which choice to take at a disjunctive pattern. A pattern is bottom-up OCLA-ambiguous if just knowing the position of a node is not enough to tell which choice of a pattern definition to go into at a pattern definition call, or whether another wrapper should be added when reaching the top of a pattern definition body and if so, which wrapper to choose. The following paragraphs define this intuition more formally.

The only patterns that can be top-down OCLA-ambiguous are choices inside pattern definitions, since disjunctive patterns are only permitted at the user level when packaged in pattern definitions. Because of the nature of bottom-linear patterns, choices and also recursive pattern definition calls can both be bottom-up OCLA-ambiguous. A choice is top-down OCLA-ambiguous if the sets of constructors for the choices are not disjoint. It is bottom-up OCLA-ambiguous for a particular parameter if the sets of positions for the formal parameters are not disjoint. It is bottom-up OCLA-ambiguous for holes if the sets of positions for the holes in each choice are not disjoint.

$\phi ::= \phi' \;\texttt{|}\; \phi''$

$$\mathit{TD\text{-}Ambig}(\phi) \equiv \neg\mathrm{disjoint}(C(\phi'), C(\phi''))$$
$$\mathit{BU\text{-}Ambig}_{id_i}(\phi) \equiv \exists \phi'_i = \texttt{?}id_i \in \phi', \phi''_i = \texttt{?}id_i \in \phi'' : \neg\mathrm{disjoint}(P(\phi'_i), P(\phi''_i))$$
$$\mathit{BU\text{-}Ambig}_{\texttt{@}}(\phi) \equiv \exists \phi_1^{\texttt{@}} = \texttt{@} \in \phi', \phi_2^{\texttt{@}} = \texttt{@} \in \phi'' : \neg\mathrm{disjoint}(P(\phi_1^{\texttt{@}}), P(\phi_2^{\texttt{@}}))$$

A recursive pattern definition is recursively bottom-up OCLA-ambiguous if it is not possible to tell from the current position whether another "wrapper" is to be added or not:

$\phi ::= \texttt{pattern}\ (id_1, id_2, \ldots, id_n)\phi'(\phi_1, \phi_2, \ldots, \phi_n) \qquad n \geq 0$

$$\mathit{BU\text{-}Ambig}(\phi) \equiv \exists \phi^{\texttt{@}} = \texttt{@} \in \phi' : \neg\mathrm{disjoint}(P(\phi^{\texttt{@}}), P(\phi))$$

This information is used to determine whether it is possible to transmit values from a place in a tree matched by one subpattern to another place in the tree matched by a different subpattern. Transmitting here means using classical attribute grammar equations.

Attributes that transmit values between two nodes bound by pattern variables can be generated if the "route" between the two pattern variables has the right properties. Only "routes" between subpatterns in the same choice branch of a pattern definition or between subpatterns outside all choices are meaningful. Scoping ensures that the programmer does not attempt to use values from one branch in a different branch or outside the pattern definition altogether. See Figure 3.2 for a sample route.

The *route* from one subpattern to another consists of a directed graph over the subpatterns in the pattern tree. The route consists of two subgraphs: the *upward part* consisting of arcs up the pattern tree from the source to the *join points*, and the *downward part* consisting of arcs down the pattern tree from join points to the destination. If we reverse the direction of all arcs in the route from A to B, we obtain the route from B to A.

The definition of the route and of join points is given here constructively. If the pattern includes no inlined pattern definition calls, the only join point is the least common ancestor, and the route is the route is simply the directed path up from the source to this join point and then down to the destination. More generally, whenever the least common ancestor is not a pattern definition call, it is the only join point. For example, in the diagram in Figure 3.2, the constructor call `assign_stmt(.,.)` at the top is the only join point on the route from `?lhs` to `?const`.

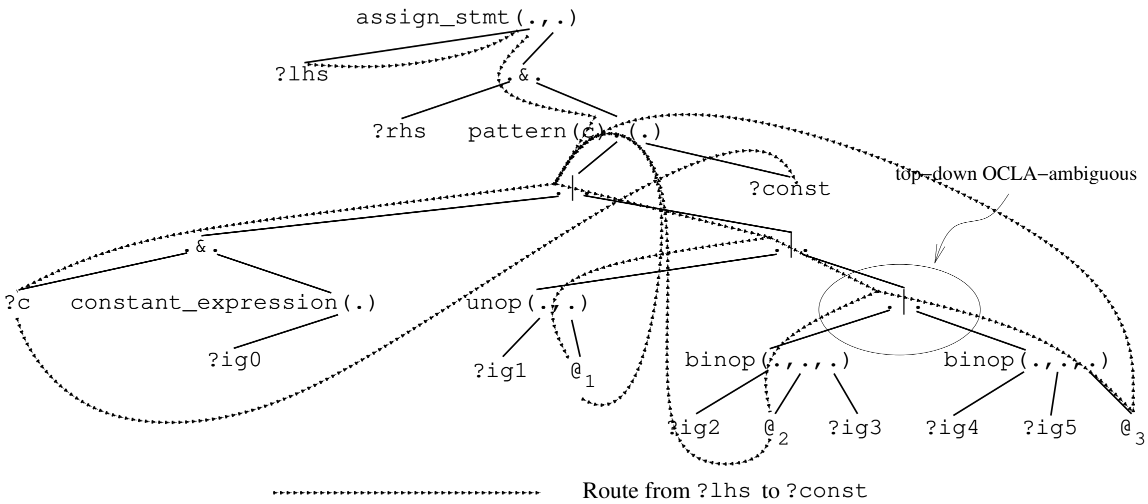
When the least common ancestor is a pattern definition call, then the source must be a descendant of one of the actual parameters to the call, and the destination must be a descendant of a different parameter. In this case, we define the set of join points recursively as the union for each choice in the body of the pattern definition of the join point set for the use of the respective formal parameters. In addition, the route is defined to include the respective routes between the uses of the formal parameters.

The upward route from the source includes the path to the least common ancestor but at every pattern definition call along the path (exclusive of the ends), the route also includes the routes from each use of the corresponding formal parameter and from each hole up to the least common ancestor. In Figure 3.2, the route from `?const` to `?lhs` takes multiple paths through the pattern definition body. The downward part of the route from the least common ancestor to the destination is simply the upward part of the route from the destination to the source.

A route changes direction from going up to going down precisely at a join point. A meaningful route (as defined earlier) will have only two kinds of join points. Either the join point is a constructor call and the route comes up through one parameter position and goes down through a different one, or else it is a conjunctive pattern, in which case the route comes up through one subpattern and down through the other. A route will *not* come up from one choice in a disjunctive pattern and go down the other because routes always follow all choices in parallel.

A route is *fully OCLA-deterministic* if none of the subpatterns on the upward part of the route is bottom-up OCLA-ambiguous and none of the subpatterns on the downward part of the route is top-down OCLA-ambiguous. A route is *partially OCLA-deterministic*

```
assign_stmt(?lhs,?rhs &
                pattern (c)
                    ?c & constant_expression(?ig0) |
                    unop(?ig1,@₁) |
                    binop(?ig2,@₂,?ig3) |
                    binop(?ig4,?ig5,@₃) (?const)
```



ϕ	$C(\phi)$	$P(\phi)$
`pattern(c).(.)`	$\{b,c,u\}$	$\{(a,2)\}$
body	$\{b,c,u\}$	$\{(a,2),(u,2),(b,2),(b,3)\}$
`.&.`	$\{c\}$	$\{(a,2),(u,2),(b,2),(b,3)\}$
$\vdots$	$\vdots$	$\vdots$
$\texttt{@}_1$	$\{b,c,u\}$	$\{(u,2)\}$
`binop | binop`	$\{b\}$	$\{(a,2),(u,2),(b,2),(b,3)\}$
`binop(?ig2,@,?ig3)`	$\{b\}$	$\{(a,2),(u,2),(b,2),(b,3)\}$
$\texttt{@}_2$	$\{b,c,u\}$	$\{(b,2)\}$
`binop(?ig4,?ig5,@)`	$\{b\}$	$\{(a,2),(u,2),(b,2),(b,3)\}$
$\texttt{@}_3$	$\{b,c,u\}$	$\{(b,3)\}$
`?const`	*Constructors*	$\{(a,2),(u,2),(b,2),(b,3)\}$

$a \equiv$ `assign_stmt`, $b \equiv$ `binop`, $c \equiv$ `constant_expression`, $u \equiv$ `unop`

Figure 3.2: A pattern, routes in the pattern, and some of the constructor and position sets

if it would be fully OCLA-deterministic except for some top-down OCLA-ambiguity in pattern matches where only the first match is used. For example, in Figure 3.2, the route from `?const` to `?lhs` is fully OCLA-deterministic. Since this example comes from a top-level match, all matches are used and thus the route from `?lhs` to `?const` is not even partially OCLA-deterministic, because it passes through a top-down OCLA-ambiguous disjunctive pattern. If the pattern match were in a `case` statement then the route from `?lhs` to `?const` would be partially OCLA-deterministic.

A pattern variable binding is *OCLA-controlling* at the point of an attribute definition if every route from it to any subpattern (except those in the bodies of pattern definitions) is at least partially OCLA-deterministic. If a binding is OCLA-controlling, an instance of the binding for a particular tree can be used to determine all the other variable bindings in patterns guarding the attribute definition. First, because none of the patterns along the route from the binding and the root of the pattern is bottom-up ambiguous, a binding can be made for each of these patterns working up from the variable binding. Then other parts of the pattern can have their bindings found deterministically from the spine in the tree corresponding to the route to the root in the pattern. The only problem would be top-down ambiguous subpatterns away from the route to the root. By the definition of partial OCLA-determinicity, such subpatterns can only be in pattern matches, in which the first match is used. But in such matches, the first choice that works is chosen, and thus one can determine the bindings even below the ambiguity. Therefore, if a pattern variable binding is OCLA-controlling for a certain attribute definition, it is necessarily truly controlling for that definition. The converse is not true, because the definition of OCLA-controlling only takes into account single constructor look-ahead.

The APS compiler requires (but currently does not check) that the node named in an attribute definition be OCLA-controlling for that definition. This restriction is not particularly onerous. The differences between a binding being OCLA-controlling and it being truly controlling are that the former does not take into account semantic tests and only uses one node look-ahead for determining ambiguity. No static definition could handle semantic conditions fully because that would require solving the halting problem. Moreover, factoring can often remove the need for multiple node look-ahead.

3.3.4 Limits on Semantic Conditions

A semantic condition needs values. If a semantic condition occurs in a `case` clause, the values that can be used are restricted to be ones uniquely determined at the point where the semantic condition is tested. Each of the values either comes from bindings outside the scope of the pattern (in which case, we assume it comes from the root of the pattern) or from another binding in the same pattern. In either case, the route from the semantic condition to the value must be fully OCLA-deterministic.

3.4 Canonicalization

Descriptional composition can only apply when pattern matching has been expressed using normal attribute definitions. This section describes canonicalizations of pattern matching in the APS compiler as needed for implementation and descriptional composition. The first transformation ("deordering") adds additional boolean attributes so as to express the "first textual definition" rule that prioritizes attribute definitions. This transformation is always used because it greatly simplifies the implementation of attribution equations, especially those including side-effecting "procedure calls" (see Section 6.2.3). The other transformations are only used before descriptional composition to ensure that pattern matching on the intermediate form is expressed in attributes. The result of the transformations is a conditional attribute grammar with a set of unordered attribute definitions for each constructor in the tree language. First, a *direction* is chosen for each attribute, synthesized or inherited. Then local values must be located somewhere within the pattern. The next step generates attributes and definitions to express pattern matching and to carry needed values to the location where the attribute definition needs them. The final step is to collect together all the definitions that apply to a single constructor. At this point the canonicalization is complete, but several simplifications are possible.

3.4.1 Deordering

Attribute definitions are not ordered in attribute grammars, but in APS, ordering is essential for resolving attribute definition conflicts. *Deordering* is the task of removing this dependence on textual order.

Deordering is accomplished by adding new boolean attributes with default value `false`. These boolean attributes are set to true when certain attribute definitions are activated, and tests of the attributes are used to guard later attribute definitions.

One could add a new boolean attribute for each attribute definition. Then each attribute definition is placed in a conditional clause that ensures that no textually earlier definition applies. For example, several definitions of the attribute `a` of the form

x`.a := `e_x`;`
⋮
y`.a := `e_y`;`
⋮
z`.a := `e_z`;`

would yield the following unordered definitions:

x`.a := `e_x`;`
x`.a1 := true;`
⋮
`if not `y`.a1 then`
  y`.a := `e_y`;`
`endif;`
y`.a2 := true;`
⋮
`if not `z`.a1 and not `z`.a2 then`

```
    z.a := e_z;
  endif;
  z.a3 := true;
```

A final attribution clause could be added to handle the default:

```
match ?any:phylum begin
  if not any.a1 and not any.a2 and not any.a3 then
    any.a := default;
  endif;
end;
```

As it happens, it is easier and more efficient to keep defaults implicit.

Deordering as explained in the previous paragraph generates an unordered description whose size in the worst case is on the order of the square of the size of the input. The new attributes themselves contribute only linearly, but the conditions that check the previous definitions can be quadratic in size. Many of the checks may be unnecessary. In particular, if the pattern context ensures that two different definitions could not conflict, the later one need not check the earlier one's boolean attribute. For example, if the first two definitions above are guaranteed not to conflict, there is no need to guard the second definition with a test of `a1`. Moreover, if an attribute definition is guaranteed not to conflict with any later definition, there is no need to generate a boolean attribute for it. The transformed example could thus be simplified:

```
x.a := e_x;
x.a1 := true;
  ⋮
y.a := e_y;
y.a2 := true;
  ⋮
if not z.a1 and not z.a2 then
  z.a := e_z;
endif;
```

In this example, only the last attribute definition conflicts with earlier ones, so instead of having an attribute for each for the first two definitions which is true when the definition is active, one could instead have a single attribute for the last definition that is true when it should *not* be active:

```
x.a := e_x;
x.a3 := true;
  ⋮
y.a := e_y;
y.a3 := true;
  ⋮
```

```
if not z.a3 then
  z.a := e_z;
endif;
```

Only one test is required now. The former kind of generated boolean attributes are called *attractors* to distinguish them from the latter kind, which are called *repeller* attributes.

Further simplifications are possible: sets of attribute definitions in which no definition conflicts with any other can share attractor or repeller attributes. For example, attribute definitions in different branches of the same `if` or `case` statement cannot conflict and can be grouped together.

Such heuristics can greatly reduce the size of the result of the deordering transformation. The following results come from the Oberon2 compiler in Appendix B. For each module, it shows the number of attributes and number of instances (both in assignments and in tests) that are generated according to various heuristics: "naive" deordering which generates one attribute for every definition, except the last; "attractors" and "repellers" only deordering, in which only one or the other kind of boolean attribute is generated; "mixed" deordering where an attempt is made to generate the minimum number of either kind; "folded" deordering, in which attractors and repellers are shared among sets of attributes with no conflicts.

Module	Naive		Attractors		Repellers		Mixed		Folded	
`OBERON2_RESOLVE`	108	1044	46	145	24	123	18	117	14	98
`OBERON2_COMPILE_COMPUTE`	98	1418	1	2	1	2	1	2	1	2
`OBERON2_CHECK`	61	106	7	17	7	17	6	16	6	16
`OBERON2_LAYOUT`	15	15	1	2	1	2	1	2	1	2
`OBERON2_TRANSLATE`	226	2330	69	586	68	585	41	558	29	375
`GCC_TREE`	35	409	22	102	29	81	5	85	5	85
`GCC2C`	170	2314	9	37	20	48	6	34	6	34

More improvement is possible. The current system looks at the $C(.)$ and $P(.)$ sets for pattern variables to see if two attribute definitions might conflict; a more sophisticated system would consider more of the pattern. A later implementation of APS could also do better with polymorphic attributes. Currently, a deordering attribute is declared for all the phyla for which the attribute being deordered is declared, but conflicts may arise for only some of the phyla being attributed.

3.4.2 Choosing a Direction

In an attribute grammar, each attribute is either synthesized or inherited. In the definitions associated with a production, there are attribute definitions for the synthesized attributes of the parent and the inherited attributes of the children. In APS, however, the pattern guarding the attribute definition is not necessarily a single constructor call. Therefore "synthesized" and "inherited" would be meaningless concepts in APS.

Nevertheless, when converting a pattern-matched description to a conditional attribute grammar, it makes a difference which direction is chosen for each attribute. An attribute grammar provides a case analysis on the type of the node for synthesized attributes and on the type of the parent for inherited attributes. Exploiting this limited form of pattern matching well can yield a simpler attribute grammar, than say, having all attributes be synthesized. The methods used to choose a direction for attributes are necessarily heuristics. The description writer may use pragmas to override a poor choice on the part of the heuristic.

A module in APS expressing a simple heuristic is given in Appendix C.5.2. Basically for every assignment, it checks to see how many copies of it would have to be made if it were inherited versus if it were synthesized. It chooses the direction that requires fewer copies.

3.4.3 Locating Local Variables

In an attribute grammar, local attributes are defined locally to a production, but in APS, local variables are declared local to some pattern-matching context. If pattern matching must be converted to classical attribution, such as before descriptional composition, it may be necessary to find a "home" for a local variable. If the local variable is used only in defining attributes of nodes taking part in the same production, it need not be changed. However, if it will be used in defining attributes of nodes farther apart, the value must be transmitted somehow from one node to the other. For instance, in the following example (from Appendix C.2.1), three different nodes are attributed (the braces (`{ }`) are sequence-matching syntax; consider them to be sugar for a constructor with one argument):

```
match ?d=attribute_decl(?,function_type({?formal},{?rd}),default:=?def)
begin
  new_scope : Scope := nested_contour(d.scope);
  formal.scope := new_scope;
  rd.scope := new_scope;
  def.scope := new_scope;
end;
```

In this fragment, the local variable `new_scope` is used in all three definitions and thus before pattern matching can be replaced with attribution, `new_scope` must be canonicalized as an attribute of a node referred to in the pattern.

Any controlling subpattern will do, but it is better if one chooses a place closer to where the local variable is used. It also leads to a more efficient implementation if a constructor call pattern is chosen to minimize the times the value of the local variable must be copied. In the previous example, suppose the `function_type` node is chosen. A new conjunctive pattern is added above the chosen site and a new pattern variable is placed in the other branch. The local value is changed to a synthesized attribute of the node bound by the pattern. For example:

```
attribute Type.LOCATED_new_scope : Scope;
match ?d=attribute_decl(?,?L=function_type({?formal},{?rd}),
                        default:=?def)
begin
  L.LOCATED_new_scope := nested_contour(d.scope);
  formal.scope := L.LOCATED_new_scope;
```

```
    rd.scope := L.LOCATED_new_scope;
    def.scope := L.LOCATED_new_scope;
  end;
```

If no subpattern is controlling, then no attribute can be defined in the scope of the local variable. Thus the local variable's value is not used anywhere and the local variable can be discarded as "dead" code.[5]

3.4.4 Generating Match and Transmission Attributes

This section describes how patterns can be implemented using attributes and attribution clauses. This transformation is necessary before descriptional composition can be used. Since the transformation introduces many attributes, it is not used except as an aid for descriptional composition.

This transformation generates attributes that transmit values from various parts of the tree matched by the pattern to the node matched by the controlling pattern for some attribute definition. The attributes transmit values used in the attribute definition and also transmit pattern matching information, in the form of boolean attributes. Attribute equations are generated along the routes to needed values from the controlling pattern variable for the definition. For the purposes of this discussion, it is assumed that there is only one attribute definition guarded by the patterns being implemented. Multiple definitions can be handled each in turn, with sharing of the generated attributes when possible.

For each subpattern ϕ, and each value $id.a$ used in the definition for which ϕ is on the route between `?id` and the controlling pattern, the transformation will generate a new attribute. This attribute will be synthesized if the ϕ is on the upward part of the route from `?id` to the controlling pattern, and inherited if it is on the downward part of the route. Similarly, for each subpattern ϕ, there will be a boolean pattern-matching attribute; inherited if ϕ is above the controlling binding in the pattern, otherwise synthesized. In order to reduce the number of attributes generated, some value or pattern matching attributes will be merely *renamings*, that is, aliases, of other attributes.

For example, the following clause from Appendix B.3.2 states that the scope of the body of a procedure is the scope that was determined for the receiver in its header:

```
match proc_decl(header(receiver:=?rec),?body) begin
  body.scope := rec.scope;
end;
```

After a simple canonicalization that names every pattern variable and removes keyword parameters for patterns, the clause has the form

```
match proc_decl(header(?g1,?rec,?g2,?g3),?body) begin
  body.scope := rec.scope;
end;
```

The controlling pattern here is `?body`, and so we need to generate attributes to bring the result of the pattern match and the value `rec.scope` to it. After inlining match attributes whose values are always true and substituting renamings, the following code is generated:

[5] Chapter 6 introduces some new kinds of attributes that *can* be assigned in these situations.

```
match ?x=proc_decl(?x1,?x2) begin
  x2.irec_scope?body := x1.srec_scopeheader(?g1,?rec,?g2,?g3);
end;
match ?x=header(?x1,?x2,?x3,?x4) begin
  x.srec_scopeheader(?g1,?rec,?g2,?g3) := x3.srec_scope?rec;
end;
match ?x=proc_decl(?x1,?x2) begin
  x2.scope := x2.irec_scope?body
end;
```

The subscripted names are simply identifiers; their internal structure is not significant.

A more complicated case for pattern matching can be found in Appendix B.5:

```
pattern op(op:Operator; arg:Expression) : Expression :=
    unop(?op,?arg),binop(?op,?arg,?),binop(?op,?,?arg);

  ⋮
match op(logical_operator(),?arg) begin
  Use arg.expr_type to define arg.errors
end;
```

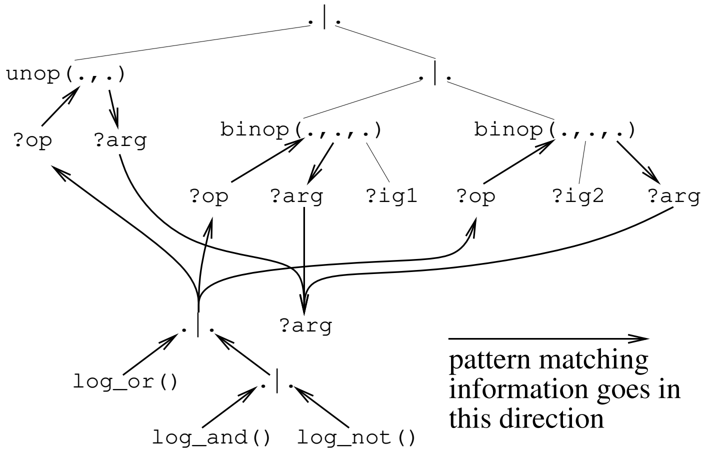
In this example, the only required value is already available at the controlling pattern `?arg`, but implementing pattern matching still requires some boolean attributes to be generated. First pattern definitions are inlined yielding the top fragment in Figure 3.3. Then after performing the transformation, inlining constant true match attributes and using the generated "renamings," the result at the bottom of Figure 3.3 is obtained. The center of the figure shows the flow of information in the pattern to the controlling pattern, `?arg`. This example illustrates once again the notational advantage of using pattern definitions.

The remainder of this section describes the transformation somewhat formally by showing what source-level APS code is generated for each subpattern. As described here, the transformation does not work for constructors with semantic parameters, nor with pattern definitions with choices more complex than pattern definition or constructor calls. It is straightforward to handle these cases, but they needlessly add to the complexity of an already complex discussion.

The transformation assumes attributes are defined for every subpattern in the pattern, but in the presence of pattern functions, this assumption leads to a great proliferation of attributes. Accordingly, the transformation also generates *renaming* declarations that serve as alias declarations with which a single attribute may have multiple names. Renamings are declared with an equal sign (`=`).

In the following discussion, all the routes mentioned are from the controlling binding. First, the discussion centers on patterns on the upward part of the such routes with a case analysis on each kind of pattern. For each kind of pattern, the `generate` clause shows what source-level APS code is generated in this case. For these patterns, inherited attributes are generated. Then the discussion moves to patterns on the downward part of the routes, for which synthesized attributes are generated. Again a case analysis is used.

```
match pattern(op,arg) unop(?op,?arg) |
                     (binop(?op,?arg,?ig1) |
                      binop(?op,?ig2,?arg))
      (pattern() log_or()|log_and()|log_not() ()), ?arg)
begin
  body
end;
```




```
match ?x=log_or() begin
  x.smatch_or|and|not := true;
end;
match ?x=log_and() begin
  x.smatch_or|and|not := true;
end;
match ?x=log_not() begin
  x.smatch_or|and|not := true;
end;
match ?x=unop(?x1,?x2) begin
  if x1.smatch_or|and|not then
    x2.imatch_?arg := true;
  endif;
end;
match ?x=binop(?x1,?x2,?x3) begin
  if x1.smatch_or|and|not then
    x2.imatch_?arg := true;
  endif;
end;
```

```
match ?x=binop(?x1,?x2,?x3) begin
  if x1.smatch_or|and|not then
    x3.imatch_?arg := true;
  endif;
end;
match ?x=unop(?x1,?x2) begin
  if x2.imatch_?arg then
    body
  endif;
end;
match ?x=binop(?x1,?x2,?x3) begin
  if x2.imatch_?arg then
    body
  endif;
end;
match ?x=binop(?x1,?x2,?x3) begin
  if x3.imatch_?arg then
    body
  endif;
end;
```

Figure 3.3: Match attributes introduced to express a complicated pattern.

For disjunctive patterns, the inherited attributes of each of the choices are simply made aliases of the ones for the combination:

$\phi ::= \phi'$ `|` ϕ''
```
        generate
            "imatchφ' = imatchφ;
             imatchφ'' = imatchφ;
             iid_aφ' = iid_aφ;
             iid_aφ'' = iid_aφ;
                for each id.a reached along a route going up through φ"
```

In conjunctive patterns routes coming up from the controlling binding will either continue up through the pattern or go down through the other subpattern. Without loss of generality, assume that they come up through the first subpattern and then continue up through this pattern or down through the second choice. The match attributes of the context and the second subpattern are checked and if both the context and the other subpattern match, attribute equations are activated. Values will come either from the context of the whole pattern or from the other subpattern:

$\phi ::= \phi'$ `&` ϕ''
```
        for each (c, i) ∈ P(φ)
            generate
                "match ?x=c(?x1,...,?xn_c) begin
                   if x.imatchφ and x.smatchφ'' then
                     xi.imatchφ' := true;
                     xi.iid_aφ' := x.iid_aφ
                        for each id.a reached along a route through φ' → φ
                     xi.iid_aφ' := x.sid_aφ''
                        for each id.a reached along a route through φ' → φ''
                   endif;
                 end;"
```

In the case of a constructor call, the pattern match must take into account all the match and value attributes arriving from the arguments of the pattern as well as from the context. Assume that the route from the controlling binding comes up through parameter i:

$\phi ::=$ *constructor*$(\phi_1, \phi_2, \ldots, \phi_n)$
```
        generate
            "match ?x=constructor(?x1,...,?xn) begin
               if x.imatchφ and x1.smatchφ1 and ... and x(i−1).smatchφ(i−1) and
                  x(i+1).smatchφ(i+1) and ... and xn.smatchφn then
                 xi.imatchφi := true;
                 xi.iid_aφi := x.iid_aφ
                      for each id.a reached along a route through φi → φ
                 xi.iid_aφi := xk.sid_aφk
                      for each id.a reached along a route through φi → φk, k ≠ i
```

endif;
end;"

To handle semantic parameters correctly, this rule would have to be changed not to fetch attributes such as $\texttt{smatch}_{\phi_k}$ from non-structural values.

A pattern call is a little more complicated. Assume that the route from the binding comes through the ith parameter. There must not be any bottom-up OCLA-ambiguity with respect to this parameter. In this case, the correct instance of the parameter binding can be determined.

$\phi ::=$ pattern $(id_1, id_2, \ldots, id_n)\phi'(\phi_1, \phi_2, \ldots, \phi_n)$
for each $\phi'_i = ?id_i \in \phi'$
generate
"$\texttt{imatch}_{\phi'_i}$ = $\texttt{imatch}_{\phi_i}$;
iid_$a_{\phi'_i}$ = iid_a_{ϕ_i};
for each $id.a$ reached along a route through $\phi_i \to \phi'_i$"

Recall that recursive calls in pattern definitions are replaced by holes (@) when the pattern is canonicalized (see Section 3.3.3 on page 52). If the pattern is *not* recursive, then the value and match attributes from the context are the ones we need in the body. Otherwise, we must generate new attributes:

$\phi ::=$ pattern $(id_1, id_2, \ldots, id_n)\phi'(\phi_1, \phi_2, \ldots, \phi_n)$
if @ $\notin \phi'$
generate
"$\texttt{imatch}_{\phi'}$ = $\texttt{imatch}_{\phi}$;
iid_$a_{\phi'}$ = iid_a_{ϕ};
for each $id.a$ reached along a route through $\phi' \to \phi$"
else
for each $(c, k) \in P(\phi)$
generate
"match ?x=c(?x1,...?xn_c) begin
if xk.$\texttt{imatch}_{\phi}$ then
xk.$\texttt{imatch}_{\phi'}$:= true;
x.iid_a_{ϕ} := x.iid_$a_{\phi'}$;
for each $id.a$ reached along a route through $\phi' \to \phi$
endif;
end;"
for each $\phi^{@}$ = @ $\in \phi'$
generate
"$\texttt{imatch}_{\phi^{@}}$ = $\texttt{imatch}_{\phi'}$;
iid_$a_{\phi^{@}}$ = iid_$a_{\phi'}$;
for each $id.a$ reached along a route through $\phi' \to \phi^{@}$"

Definitions for synthesized attributes are generated for the downward parts of the routes from the controlling variable binding.

For pattern variable bindings, a synthesized attribute alias is created for each value that needs to be fetched from this location and a (constant true) match attribute is similarly defined.

ϕ ::= ?*id* for non-parameter binding

```
        generate
            "attribute Type.smatch_φ : Boolean := true;
             sid_a_φ = a;
                for each id.a needed somewhere"
```

Since this attribute always gets its default value, any use of `xi.smatch`$_\phi$ can be replaced with `true`.

When all matches are used, semantic conditions are not used to guide the pattern matching. Instead they are turned into normal `if` guards inside the body of the match. Only for `case` clauses are semantic conditions evaluated during the pattern match. In this case, semantic conditions use the values coming in and if the values check, define the match attribute to be true. Semantic conditions require their own match and value attributes generated similarly, except only along routes from the condition to values used in the condition.

ϕ ::= `if` *expr* only if the match is in a `case` clause

```
        for each c ∈ C(φ)
            generate
                "match ?x=c(...) begin
                   if x.imatch_φ then
                     if expr then where each id.a replaced by x.iid_a_φ
                       x.smatch_φ := true;
                     endif;
                   endif;
                 end;"
```

The inherited match attribute is tested to ensure that the needed value attributes are defined. If the match is going to fail anyhow because some other part of the pattern didn't match, the value attributes will not be defined. The value of the match attribute will be ignored in this case, but it is important that evaluating it not cause an error.

In the case of a conjunctive pattern, there may be routes going down into each of the subpatterns to fetch values from there. The conjunctive pattern only matches if both subpatterns match.

ϕ ::= ϕ' `&` ϕ''

```
        for each c ∈ C(φ)
            generate
                "match ?x=c(...) begin
                   if x.smatch_φ' and x.smatch_φ'' then
                     x.smatch_φ := true;
                   endif;
                 end;
```

```
sid_aϕ = sid_aϕ';
    for every id.a reached along a route through ϕ → ϕ'
sid_aϕ = sid_aϕ'';
    for every id.a reached along a route through ϕ → ϕ''"
```

Linearity ensures that the values reached on the two routes are distinct (and thus that the aliases do not conflict). If one of the subpatterns is guaranteed to match (if, for example, it is a pattern variable), then the match attribute can be made an alias of the match attributes of the other subpattern.

For a disjunctive pattern, if the first choice matches, the value attributes are copied from there. Otherwise, if the second choice matches, the value attributes are copied from it:

```
ϕ ::= ϕ' | ϕ''
        for each c ∈ C(ϕ)
            generate
                "match ?x=c(...) begin
                   if x.smatchϕ' then
                     x.smatchϕ := true;
                     x.sid_aϕ := x.sid_aϕ';
                         for every id.a reached along a route coming down through ϕ
                   elsif x.smatchϕ'' then
                     x.smatchϕ := true;
                     x.sid_aϕ := x.sid_aϕ''
                         for every id.a reached along a route coming down through ϕ
                   endif;
                 end;"
```

If the pattern is top-down OCLA-unambiguous (as must be the case if all matches are to be used), the sets of constructors for the two sides are different, and one can generate less code:

```
ϕ ::= ϕ' | ϕ''
        generate
                "x.smatchϕ' = x.smatchϕ;
                 x.smatchϕ'' = x.smatchϕ;
                 sid_aϕ' = sid_aϕ;
                 sid_aϕ'' = sid_aϕ;
                     for every id.a reached along a route coming down through ϕ"
```

For a constructor call, the match succeeds if each of the children match. Each of the values coming up through the node comes through one of the children (and only one because of linearity):

```
ϕ ::= constructor(ϕ1,ϕ2,...,ϕn)
        generate
            "match ?x=c(?x1,...,?xn) begin
```

```
      if x.smatch_φ1 and ... and x.smatch_φn then
        x.smatch_φ := true;
        x.sid_a_φ := x.sid_a_φi;
           for every id.a reached along a route through φ → φi
      endif;
    end;"
```

If a route from the controlling pattern goes down through a pattern call, pattern matching attributes and values for the call come from the body. Each parameter must have the appropriate match and value attributes defined. Similarly for holes in the pattern. The parameter bindings in the body get their attributes from the actual parameters in the pattern call. The holes get their attributes from the body. Again if the pattern is not recursive (there are no holes), more aliases can be used:

```
φ ::= pattern (id1,id2,...,idn)φ'(φ1,φ2,...,φn)
        if @ ∉ φ'
            generate
                "smatch_φ' = smatch_φ;
                 sid_a_φ' = sid_a_φ
                    for every id.a reached along a route through φ → φ'"
        else
          for each c ∈ C(φ)
            generate
                "match ?x=c(...) begin
                    if x.smatch_φ' then
                      x.smatch_φ := true;
                      x.sid_a_φ := x.sid_a_φ'
                        for every id.a reached along a route through φ → φ'
                    endif;
                 end;"
        for each φ'_i = ?id_i ∈ φ'
            generate
                "smatch_φ'_i = smatch_φi;
                 sid_a_φ'_i = sid_a_φi;
                    for every id.a reached along a route through φ'_i → φi"
        for each φ^@ = @ ∈ φ'
            generate
                "smatch_φ@ = smatch_φ';
                 sid_a_φ@ = sid_a_φ';
                    for every id.a reached along a route through φ^@ → φ'"
```

Once all the transmission attributes are generated it is possible to express every original attribute definition without pattern matching. First all the wrappers are handled. Pattern matching is ignored, except for semantic conditions in top-level or `for` patterns; such conditions are turned into normal `if` statements. All `if` statements guarding the attribute definition remain. Then all expressions (those in the wrapping `if`'s and those on the right-hand side of the attribute definition) are converted: *id*`.`*a* is converted into `i`*id*`_`$a_{?v}$ where `?`v is the controlling pattern variable bound to the node being attributed. Finally the attribute definition itself is generated. If the attribute in question is inherited, one copy is generated for each position the variable could take:

```
?v.a := expr;
        for each (c, i) ∈ P(?v)
            generate
                "match ?x=c(?x1,...,?xn_c) begin
                   if condition_1 then
                     ⋱
                       if condition_ω then
                         xi.a := expr;
                       endif;
                     ⋰
                   endif;
                 end;"
```

If the attribute is synthesized, the constructor set for the variable is used:

```
?v.a := expr;
        for each c ∈ C(?v)
            generate
                "match ?x=c(...) begin
                   if condition_1 then
                     ⋱
                       if condition_ω then
                         x.a := expr;
                       endif;
                     ⋰
                   endif;
                 end;"
```

Since most attributes tend to be synthesized, this generating rule shows the importance of using conjunctive patterns to trim down constructor sets.

All these rules make the step of generating transmission attributes rather complex, but the basic issue is simple—the values needed for an attribute definition are copied from node to node until they are in reach of the attribute definition that needs them. Similarly, match attributes ensure that the attribute definition is only activated when the tree matches the guarding patterns.

3.4.5 Collecting

In a traditional attribute grammar, there is only one set of definitions for each production in the grammar. The preceding step has simplified pattern matching so that only one constructor is ever being used, but there are still multiple attribution clauses for each constructor. It is a straightforward task to collect all the `match` clauses together that refer to the same constructor. In fact, collection can be done during generation.

3.4.6 Simplifying

The canonicalization steps mentioned in this section produce a verbose attribute grammar with many artificial attributes expressing ordering or pattern matching. Once the attribute equations are all collected together, a number of simplifications can be attempted.

Many of the match attributes will always be true. For example the inherited attribute to the top of a pattern will always be true. Similarly, synthesized match attributes for pattern variables are always true. In these cases, one can replace uses of the attributes by the value true and often further simplify the expression in which this use occurs.

One can also use a form of "copy propagation" to substitute the values of attributes whose definitions are known and do not involve any computation.

3.5 Summary

Pattern matching is powerful tool, and an important method for factoring a description by concept. APS has a powerful linear pattern matching facility that through named pattern definitions allows patterns themselves to be factored as well. Pattern matching can be added to the attribute grammar formalism assuming a good conflict resolution rule is chosen. Descriptions using pattern matching can be converted to conditional attribute grammar form through a canonicalization described in the last section of this chapter.

Chapter 4

Sequences

This chapter introduces the concept of sequences and describes how support for sequences can ease the task of writing descriptions, reduce dangerous redundancy, and possibly even permit efficient parallel or incremental evaluation.

The chapter discusses issues that arise when including sequences in a formalism with attribution. Then it describes how sequences and their generalization, collections, are supported in APS and how they are integrated with expressions and pattern matching. The chapter concludes with a description of the implementation of sequences, how sequence patterns can be converted into uses of pattern definitions, and how sequence expressions can be converted to simpler expressions.

4.1 Why Sequences?

When describing the structure of a formal entity (such as a string of a programming language), it is often intuitive to speak of arbitrary length sequences (perhaps with separators). For example, one might say that a *block* is the keyword `begin` followed by zero or more *statements* followed by the keyword `end`. This description can be written formally:

block $\rightarrow$ `begin` statement* `end`

However, many formal language tools lack a fundamental concept of a sequence and so the previous rule would be syntactic sugar for the following set of rules:

block $\rightarrow$ `begin` statement-list `end`
statement-list $\rightarrow$
$\rightarrow$ statement statement-list

Sometimes the tool does not even provide syntactic sugar, and the description must be written without using any explicit notion of sequencing.

If the formalism does not support sequences, not even with syntactic sugar, a description writer must generate dreary boilerplate every time a sequence is needed. Moreover, if the formalism does not provide generic operations such as `append`, `nth` or `map`, then again the description writer must rewrite such functionality from scratch. Therefore, the lack of support for sequences may increase the dangerous redundancy of a description.

But syntactic sugar is not sufficient to get all the benefits of sequences. If sequences or, more generally, collections are treated as a fundamental data type by a formalism, more implementation techniques may apply. For example, languages with primitive support for collections may be more easily parallelizable [85].

Another example comes from research in incremental attribute grammar evaluators. If a sequence is treated as mere syntactic sugar for a left-heavy or a right-heavy tree of the elements in the sequence, the height of a tree will be at least as great as the length of the longest sequence in the tree. Since programs rarely are deeply nested but often have long sequences (of declarations at the top-level, or of statements in a procedure), a left-heavy or right-heavy tree for sequences will greatly add to the height of the tree. Incremental re-evaluation algorithms for attribute grammars work best when the height of the tree is limited (see for example, Vogt and Swiestra's work [92]). If some form of balanced tree is used to represent the sequence, the sequence will only contribute logarithmically to the height of the tree. Using balanced sequences can therefore improve incrementality.

Therefore, sequences should be supported by a formalism, not only through syntactic sugar in order to avoid dangerous redundancy, but also as a fundamental data type that admits sophisticated implementation techniques, such as needed for parallelism or incrementality.

4.2 Attribution with Sequences

If sequences of tree nodes are present in an attribute grammar-like formalism, various issues arise. Can sequences or partial sequences be attributed? How can attributes of the children of sequence be used by the parent? How can elements of the sequence use attributes of the parent or of each other?

If sequences are merely syntactic sugar for some left-heavy, right-heavy or even balanced tree representation, any form of attribute flow supported for normal nodes is possible. Typically however, attribute flows in sequences fall in one of the following four patterns:

1. *broadcast* a value to all members of the sequence

2. *collect* a value from each member using some (associative) combining function

3. start with a value and give it to the first member of the sequence. Each member accepts a value and produces a new value. This new value is given to the next member and the last member returns it to the parent (*bucket-brigade* left-to-right)

4. bucket-brigade right-to-left

For example, if the sequence is represented as a balanced tree of elements, each sequence node has one of three forms:

$\texttt{seq}_0 \rightarrow \texttt{seq}_1\ \texttt{seq}_2$
$\texttt{seq} \rightarrow \texttt{element}$
$\texttt{seq} \rightarrow$

```
X → ... seq ...
        seq.value = value to broadcast
seq₀ → seq₁ seq₂
        seq₁.value = seq₀.value
        seq₂.value = seq₀.value
seq → element
        element.value = seq.value
seq →
```

(a) broadcast a value to all children

```
X → ... seq ...
        ... = ... seq.value ...
seq₀ → seq₁ seq₂
        seq₀.value = f(seq₁.value,seq₂.value)
seq → element
        seq.value = element.value
seq →
        seq.value = identity
```

(b) collect using combining function `f`, with identity `identity`

```
X → ... seq ...
        seq.before = starting value
        ... = ... seq.after ...
seq₀ → seq₁ seq₂
        seq₁.before = seq₀.before
        seq₂.before = seq₁.after
        seq₀.after = seq₂.after
seq → element
        element.in = seq.before
        seq.after = element.out
seq →
        seq.after = seq.before
```

(c) bucket-brigade left to right

Figure 4.1: Idioms for expressing 3 attribution flows for a balanced sequence representation. The fourth attribution flow (bucket-brigade right-to-left) is analogous to the third (bucket-brigade left-to-right).

A sequence of the first form is split into nearly equally large subsequences. A sequence of the second form has but one element. The third and last form is the empty sequence. The idioms in Figure 4.1 may be used to execute the four attribute flows. The idioms for left-heavy and right-heavy trees are similar, even simpler, as there are only two forms for each sequence node.

In the idioms given, there is no redundancy in the concept. For instance, in Figure 4.1(b) the combining function must be given only once and the identity value similarly only once. On the other hand, the idioms need a lot of "boilerplate" to get the values to where they are needed. A formalism can help make a description less cluttered and less error-prone if this boilerplate can be generated automatically. LIGA [57] has a `CHAIN` declaration for specifying bucket-brigade attributions precisely for this reason.

If the semantics of a sequence are such that it has a particular structure, then it may be necessary to overspecify computations on the sequence. For example, if a left-heavy representation is used, and a "collect" attribute flow is defined using this decomposition, the combining function will only be used in a left-associative manner. Before an implementation program can apply a transformation to improve incrementality that relies on the function being associative, the program must ensure that indeed it is associative. When on the other hand, combining functions are required to be associative, it is easier to optimize the description.

When the intermediate sequence nodes cannot be attributed—that is, if the sequence is not sugar for a particular representation—the formalism must provide mechanisms to accomplish at least the preceding four attribute flows. It is better if the formalism does not require dangerous redundancy; that is require the combining function of the default value to be specified more than once (without checking). The first two attribute flows (broadcast and collect) are usually the easiest to support. It is more difficult to support bucket-brigades.

The Olga language in the FNC2 system represents sequences implicitly and provides a form of case analysis for the sequence [54]. This case analysis can be used to express a bucket-brigade. If one does not use Olga's equivalent of LIGA's CHAIN declaration, a left-to-right bucket-brigade can be expressed as follows: (This syntax is generic, the fragment is not legal Olga code.)

```
X → ... seq ...
        for each elem ∈ seq
            elem.in = if is_first(elem) then starting value
                                        else previous(elem).out
        ending value = if is_empty(seq) then starting value
                                        else last(seq).out
```

This fragment avoids some of the boilerplate that occurs in a formalism with no primitive support for sequences. On the other hand, it is necessary to state the starting value twice. It is also necessary to specify the `out` attribute for elements more than once. If a change was made in one place but not in the other, the inconsistency may not be found easily. The idioms in Figure 4.1 that operate directly on an exposed balanced tree representation do not have this problem. The `before` and `after` attributes for *sequences* did have to be specified multiple times. This redundancy, however, was in the boilerplate, rather than here where it is more part of the concept.

In summary, if the representation of a sequence is exposed and if attributes may be placed on the intermediate sequence nodes, the description writer has complete freedom to specify an attribution flow. On the other hand, this power may lead to overspecification of a problem. Thus in order to allow optimization, a formalism may hide the representation of sequences, and require the description writer to use primitive sequence operations. However, such an approach has two dangers: the primitives may not be powerful enough to handle what the description writer needs for a concept, and the primitives may lead to dangerous redundancy.

4.3 Sequences and Collections in APS

In APS, sequences are supported primitives but the representation is (mostly) exposed. A sequence is treated as just one kind of collection. Other collection types supported in APS are lists, bags, sets and multi-sets. All the collection types share a "comprehension" syntax similar to that used in SETL [83], Miranda or Haskell and also share a pattern syntax. This section introduces the various collection types and then sequences *per se*. Next this section explains the pattern matching syntax and finally the more complicated comprehension syntax.

4.3.1 Collections

APS provides a *class* (that is, a generic interface) named "`COLLECTION`" and a number of concrete modules that fulfill it: `BAG` represents unordered collections of values where duplicates are permitted; `SET` represents unordered collections without duplicates; `ORDERED_SET` represents collections kept in sorted order without duplicates; `LIST` represents ordered collections and are similar to sequences except without object identity.

```
type BagOfIntegers := BAG[Integer];
type SetOfIntegers := SET[Integer]((=));
type OrderedSetOfIntegers := ORDERED_SET[Integer]((=),(<));
type ListOfIntegers := LIST[Integer];
```

(In APS, type and phylum parameters are enclosed in square brackets `[ ]`.) A bag type need only specify the type of the elements in the bag. A set type must also specify an equality operation to be used to determine whether two elements are the same for the purposes of the set; the function may be user-defined. Ordered sets need a comparison function as well. Operations for ordered sets can be implemented more efficiently than for general sets.

A special module, `SEQUENCE`, also fulfills the `COLLECTION` signature. More precisely, it fulfills the `READ_ONLY_COLLECTION` signature. The only major difference to the description writer is that type inference cannot be used to infer the type of a comprehension (see Section 4.3.3). The `SEQUENCE` module is used to create sequence phyla, such as seen in Chapter 2:

```
phylum Modules=SEQUENCE[Declaration];
```

Here `Modules` is declared as sequence phylum. The representation of sequences is exposed, a sequence is constructed using three constructors: `append`, `single`, and `none`. This representation permits (but does not require) the sequence to be balanced.

4.3.2 Pattern Matching Collections

In APS, sequences (as well as other collection types) may be matched using a special pattern syntax. This syntax allows individual elements to be matched, but does not expose any of the internal nodes used in the representation of the collection type. A collection pattern consists of zero or more separated collection subpatterns separated by commas:

pattern ::= ...| `{` *sub*∗`,` `}`

A subpattern may be a normal pattern, in which case it applies to a single element. Alternatively, it may be three dots (`...`) with an optional element pattern (preceded by the keyword `and`). In this case, the subpattern represents zero or more elements that each must match the element pattern if one is given.

sub ::= *pattern* | `...` | `...` `and` *pattern*

Any variable bindings in the element pattern are ignored elsewhere.

An example of pattern matching sequences comes from the static check module in Appendix B.5:

```
attribute Declaration.proc_is_local : Boolean := false;
match proc_decl(body:=block(decls:={...,?d=proc_decl(...),...})) begin
  d.proc_is_local := true;
end;
```

The `...` performs two purposes in this fragment. The `...` in the pattern `proc_decl(...)` means that the children of the `proc_decl` node are ignored; it is not a sequence pattern. The other instances of `...` concern sequence pattern matching: each of the `...` sections in the pattern `{...,?d,...}`. can stand for any number of declarations. The pattern variable `d` thus may be bound to any of the declarations in the list. The pattern further restricts the binding to procedure declarations. Thus `d` could be bound to any of the local procedure declarations in a procedure block. Since this fragment is a top-level pattern match, all matches are used. Thus any local procedure declaration gets the value `true` for its `proc_is_local` attribute.

Collection patterns may appear wherever patterns are expected. For example, there is no restriction on nested collection patterns:

```
pattern case_label(e : Expression) : CaseLabel
    := single_label(?e),range_label(?e,?),range_label(?,?e);
match case_stmt(?sub,{...,case_clause({...,case_label(?e),...},?),...},?)
begin
  make sure e has the same type as sub
end;
```

For any value in a case label of any clause in a case statement, its type must be the same as the type of the expression being tested. There are three places where there are multiple possibilities in the match. For the case statement, a clause is chosen. Next for that clause, a label is chosen. Finally, if that label is a range, either the start or the end is chosen.

Later in the same module as the previous example, a check is made that a case label does not repeat one that came before. A simple way to do this check is to match any two labels and ensure that they do not have the same value:

```
match {...,single_label(?e1),...,?lab2=single_label(?e2),...}
begin
  if e1.constant_value = e2.constant_value then
    generate an error message
  endif;
end;
```

The definition of OCLA-ambiguity given in Section 3.3.3 is extended to apply to collection patterns: a collection pattern is top-down OCLA-ambiguous if there is more than one list subpattern (`...` or `... and` *pattern*); a subpattern in a collection pattern is bottom-up OCLA-ambiguous if there are more than two list subpatterns or if there are two and the subpattern is not between them. For instance, the first example from this section is top-down OCLA-ambiguous because there are two list subpatterns:

```
{...,?d=proc_decl(...),...}
```

However, the subpattern `?d=proc_decl(...)` is not bottom-up OCLA-ambiguous, because it lies between the two list subpatterns. The second example has two collection patterns. Both are top-down OCLA-ambiguous and in neither case is the sole non-list subpattern bottom-up OCLA-ambiguous:

```
{...,case_clause({...,case_label(?e),...},?),...}
```

The third example has three list subpatterns and so it is top-down OCLA-ambiguous and every subpattern is bottom-up OCLA-ambiguous:

```
{...,single_label(?e1),...,?lab2=single_label(?e2),...}
```

As a result, the binding of `lab2` is nowhere OCLA-controlling.

4.3.3 Collection Comprehensions

Values of a collection type may be created using a polymorphic constructor: `{ }` with a powerful comprehension notation. For example:

```
b : BagOfIntegers := {1,2,3,4,5,4,3,2,1};
s1 : SetOfIntegers := {x for x:Integer in b};
s2 : SetOfIntegers := {0, s1...};
o : OrderedSetOfIntegers := {x+1 if odd(x) for x:Integer in b};
```

The first case simply builds a bag of nine integers. The second case builds a set with all the elements of `b` (from which duplicates are removed). The third case builds a set from another set plus zero. The fourth case has a conditional, `odd(x)`, used to control entry into the set. The resulting set will be `{2,4,6}`.

More formally, a comprehension consists of zero or more comprehension subexpressions separated by commas:

expr ::= ...| `{` *comp*∗`,` `}`

Each comprehension subexpression is either a simple expression or an instance of a *sequence expression*:

comp ::= *expr* | *seq*
seq ::= *expr* ...
 | *comp* `if` *expr*
 | *comp* `for` *id*`:`*type* `in` *expr*
 | *func*`(`seq_1`,`seq_2`,`...`,`seq_n`)` $n > 0$

A simple expression is simply included in the resulting collection. The first form of sequence expression includes all the elements of a collection in the result. The second form either adds or ignores the collection built by the comprehension depending on the boolean value of the expression. The third form adds a collection for each element of another collection. The scope of the identifier is the comprehension subexpression *comp*. Specifying the type for the formal is optional. The last form of a sequence expression is an *implicit map* of a function over arguments of a sequence. If the function has one argument, it is applied to each element of its sequence argument. If there is more than one sequence, there must exist a collection c and each sequence must have one of the forms:

c ...
comp `for` *id*`:`*type* `in` c
func`(`seq'_1`,`seq'_2`,`...`,`seq'_m`)` $n > 0$

where each seq'_j must also have one of the preceding forms. In this case, the implicit map can be trivially rewritten into the form:

func`(`$expr_1$`,`$expr_2$`,`...`,`$expr_n$`)`
 `for tmp in` *expr*

where each *expr* is an ordinary (non-sequence) expression using `tmp`.

The comprehension notation is *polymorphic*; it is valid for any collection type. The APS compiler infers the correct type for the comprehension. If no type can be inferred, an error is reported. The type may be given explicitly by preceding the comprehension with *collection-type*`$`. The basic Algol scope module used in the Oberon2 compiler has a complicated example of a comprehension:

```
private type SortedDecls := ORDERED_SET[remote Decl]((==),(<<));

var function find_local_decl(name : Symbol; scope : remote Contour)
```

```
    : remote Decl
begin
  case SortedDecls${decl if decl_name(decl) = name
                      for decl in scope.local_decls}
  begin
    match {?first,...} begin
      result := first;
    end;
  else
    result := Decl$nil;
  end;
end;
```

The type `SortedDecls` is an ordered set of `Decls` (ignore the equality and ordering relations for now). The case statement forms the set of all `decls` from the contour's local declaration list whose name matches the name being searched for. If this set has a first element (that is, if it has *any* elements), this value is returned, otherwise a nil pointer is returned.

4.3.4 Attribution

All four attribution flows are supported in APS. For example, to broadcast a value to all children of a sequence, one may use a sugared form of `for`:

```
match ?p=funcall(?,?args) begin
  for a : Expression in args begin
    a.scope := p.scope;
  end;
end;
```

This fragment broadcasts a scope attribute to all children of a function call. The `for` "loop" is sugar for the following form that uses the `{ }` notation:

```
match ?p=funcall(?,?args) begin
  for args begin
    match {...,?a,...} begin
      a.scope := p.scope;
    end;
  end;
end;
```

As this transformation demonstrates, there is no notion of a loop with control going around it; the sugared `for` is simply a shorthand for a non-deterministic pattern match.

A collecting attribute flow can be accomplished with an *implicit reduction* over the sequence of children using a special syntax. If one of the arguments to a binary function is a sequence expression, then this argument represents a series of values that the function is to reduce. The base case is the other argument. Reduction starts from the right if the second argument is the sequence expression, otherwise it starts from the left. Since `and` is a binary function written as an infix operator, it can be used to reduce lists of booleans. The following example comes from Appendix B.4:

```
-- a function call is constant if its function is constant
-- (in this case, "constant" means "predefined")
-- and each of its arguments is constant.
match ?e=funcall(?func,?args) begin
  e.expr_constant := func.expr_constant and
      (arg.expr_constant for arg in args);
end;
```

The sequence expression `(arg.expr_constant for arg in args)` is the second operand to `and` and computes a boolean value for each argument to the function call. The preceding fragment is equivalent to

```
match ?e=funcall(?func,?args) begin
  e.expr_constant := func.expr_constant and
                       (arg1.expr_constant and
                         (arg2.expr_constant and
                           ...
                              argn.expr_constant
                           ...
                         )
                       )
```

This reduction notation is powerful and convenient, but it overspecifies reductions done with an associative function, because one must choose either a left associating or a right associating reduction.

Bucket-brigades can be thought as an unbounded number of attribute definitions sharing most of the same structure. For example, a left-to-right bucket-brigade conceptually has the form:

$$
\begin{array}{l}
\mathtt{X} \rightarrow \mathtt{seq} \\
\qquad \mathtt{seq}_1\mathtt{.in = X.start} \\
\qquad \mathtt{seq}_2\mathtt{.in = seq}_1\mathtt{.out} \\
\qquad \vdots \\
\qquad \mathtt{seq}_n\mathtt{.in = seq}_{n-1}\mathtt{.out} \\
\qquad \mathtt{X.result = seq}_n\mathtt{.out}
\end{array}
$$

In APS, comprehension subexpressions may be used in attribute definitions. The preceding example is expressed as follows:

```
match ?x=con(?seq) begin
  elem.in for elem:Element in seq, x.result :=
      x.start, elem.out for elem:Element in seq;
end;
```

(where `con` is the name of the constructor corresponding to the production `X` $\rightarrow$ `seq`). As with the hand-written idiom for defining bucket-brigades, but unlike the Olga example, this fragment does not duplicate any conceptual elements (the starting value, the in and out attributes for each element, and the receiver for the result). The sequence must be duplicated, but the APS compiler checks that the comprehensions are over the same sequence.

As described in this section therefore, APS provides an intuitive notation (using braces `{ }`) to create collections and to express patterns. It is possible to express all four attribution flows without dangerous redundancy. The representation of `SEQUENCE` phyla is exposed to allow attribution, but the representation of the other types is hidden.

4.4 Implementation

Currently the APS compiler implements *all* collections as (potentially balanced) trees with three constructors: `append`, `single` and `none`. The special collection syntax for pattern matching, expressions and attribution is transformed into simpler forms. This section describes the transformations used.

4.4.1 Implementing Collection Patterns

Complicated collection patterns are transformed into pattern definitions. These pattern definitions handle any form the tree may be in, balanced or otherwise. For example, they handle trees in which `none()` nodes are appended to other partial sequences. This property is essential to enable descriptional composition.

Collection subpatterns can have any of the following forms:

`...`
`... and` *pattern*
pattern

The first two are called *sequence subpatterns*. A collection pattern is converted into a pattern definition call with parameters for each of the non-list subpatterns. The pattern definition body is constructed by determining how the sequence could be represented.

First, if `none()` is valid, it is added as a choice in the pattern definition body. This pattern is valid only when there are only sequence subpatterns. Next, if a `single` pattern is valid, it is added. This pattern is valid when there is one non-sequence subpattern, in which case the argument is this subpattern. It is also valid when there are only sequence subpatterns, in which case it has the form `single(?)` if the subpatterns include `...`, otherwise there is one `single` pattern for each subpattern `... and` *pattern*.

Finally all the choices for `append` are added. The sequence pattern may be split anywhere not next to a sequence subpattern, or it may split in the middle of a sequence subpattern. For example, the following sequence pattern has four places where it could be split:

```
{ ?f1,?f2,... and m(),?f3,...}
 |   |    |                 |
```

These places yield the following choices for the pattern definition body:

```
append({},{?f1,?f2,... and m(),?f3,...}),
append({?f1},{?f2,... and m(),?f3,...}),
append({?f1,?f2,... and m()},{... and m(),?f3,...}),
append({?f1,?f2,... and m(),?f3,...},{...})
```

The first and fourth cases will end up including a (bottom-linear) recursive call to the pattern definition.

Now that the pattern definition body is defined, the collection patterns in the body must be replaced. The algorithm remembers previously generated pattern definitions to avoid infinite recursion. See Figure 4.2 for the final set of resulting pattern definitions for the example. If instead of extending the definition of OCLA-ambiguity for sequences, the definition operated on the implementation of sequences, `?f3` would not be OCLA-controlling. A different implementation of sequences would lead to a different definition of OCLA-ambiguity for sequences. It was to avoid this state of affairs that OCLA-ambiguity was extended to operate directly on collection patterns.

4.4.2 Implementing Comprehensions

Collections are created using the `{ }` notation. This comprehension notation can be converted to simpler expressions including uses of generated function definitions. First, the top-level `{ }` notation is converted into nested calls to append the comprehension results.

Assume that type inference has been done for every comprehension yielding a prefix *type*`$` for every comprehension. Let E be the function that implements comprehensions using expressions. It takes two parameters, the sequence expression and a list of implicit mapping functions (given in angle braces $(\langle \ldots \rangle)$; see the end of this section for the purpose of this parameter):

expr ::= *type*`${` *seq*`*, }`
    expr $\Rightarrow$
        `"`*type*`$append(`$E(seq_1)(\langle\rangle)$`,`*type*`$append(`$E(seq_2)(\langle\rangle)$`,...`$E(seq_n)(\langle\rangle)$`...))"`

Each kind of sequence is translated according to its own rule. A single element is placed in a collection (after applying any map functions)

seq ::= *expr*
    $E(seq)(\langle m_1, \ldots, m_n \rangle) =$ `"`*type*`$single(`m_1`(...(`m_n`(`*expr*`))...))"`

Including all the elements of a collection is easy if it has the same type as the collection being created (and it is not involved in any implicit maps). The whole collection can be used directly. Otherwise the postfix ... syntax is treated as sugar for a trivial map over the elements of the collection:

seq ::= *expr* ...
    `if` *expr-type* `is the same as` *type* `and` $m = \langle\rangle$
        $E(seq)(m) = E(expr)(m)$
    `else`
        $E(seq)(m) = E($`tmp for tmp in` *expr*$)(m)$

```
-- {?f1,?f2,... and m(),?f3,...}
pattern pXXAXD(f1,f2,f3 : Element) : Collection =
    append(p(),pXXAXD(?f1,?f2,?f3)),
    append(pX(?f1),pXAXD(?f2,?f3)),
    append(pXXA(?f1,?f2),pAXD(?f3)),
    append(pXXAXD(?f1,?f2,?f3),pD());
pattern p() : Collection = -- {}
    none(),
    append(p(),p());
pattern pX(f1 : Element) : Collection = -- {?f1}
    single(?f1),
    append(p(),pX(?f1)),
    append(pX(?f1),p());
pattern pXAXD(f2,f3 : Element) : Collection = -- {?f2,... and m(),?f3,...}
    append(p(),pXAXD(?f2,?f3)),
    append(pXA(?f2),pAXD(?f3)),
    append(pXAXD(?f2,?f3),pD());
pattern pXA(f2 : Element) : Collection = -- {?f2,... and m()}
    single(?f2),
    append(p(),pXA(?f2)),
    append(pXA(?f2),pA());
pattern pA() : Collection = -- {... and m()}
    none(),
    single(m()),
    append(pA(),pA());
pattern pAXD(f3 : Element) : Collection = -- {... and m(),?f3,...}
    single(?f3),
    append(pA(),pAXD(?f3)),
    append(pAXD(?f3),pD());
pattern pD() : Collection = -- {...}    a simpler definition is possible, of course
    none(),
    single(?),
    append(pD(),pD());
pattern pXXA(f1,f2 : Element) : Collection = -- {?f1,?f2,... and m()}
    append(p(),pXXA(?f1,?f2)),
    append(pX(?f1),pXA(?f2)),
    append(pXXA(?f1,?f2),pA());
```

Figure 4.2: Pattern definitions that implement {`?f1,?f2,... and m(),?f3,...`}

Since APS does not have `if` expressions, an out-of-line `if` statement is generated that computes the result into a newly generated local value `tmp`. Any further out-of-line generations will be placed in the branch with the assignment:

seq_0 ::= seq_1 `if` *expr*

```
        generate
            "tmp : type;
             if expr then
               other out-of-line additions
               tmp := E(seq1)(m);
             else
               tmp := type$none();
             endif;"
        E(seq0)(m) = "tmp"
```

A `for` comprehension is converted into a call of a function that explicitly traverses the structure of a sequence:

seq_0 ::= seq_1 `for` *id* : *element-type* `in` *expr*

```
        generate
            "function map_tmp(s : expr-type) : type begin
               case x begin
                 match expr-type$append(?s1,?s2) begin
                   result := type$append(map_tmp(s1),map_tmp(s2));
                 end;
                 match expr-type$single(?id) begin
                   other out-of-line additions
                   result := E(seq1)(m);
                 end;
                 match expr-type$none() begin
                   result := type$none();
                 end;
               end;
             end;"
        E(seq0)(m) = "map_tmp(expr)"
```

An implicit map is converted by adding the mapping function to the list:

seq ::= *func*(*seq*)

$$E(seq_0)(\langle m_1, \ldots, m_n \rangle) = E(seq_1)(\langle m_1, \ldots, m_n, func \rangle)$$

4.4.3 Reductions

Reductions are similarly implemented, with the additional complexity of a reduction function. Let R be the function that implements (right) reductions. This section shows how R is defined. Left reductions are analogous.

$expr_0$ ::= $func$`(`$expr_1$`,`seq`)`

$$expr_0 \Rightarrow R(func, expr_1, seq)()$$

A simple expression treated as part of a sequence is handled as a base case:

seq ::= $expr$

$$R(f,i,seq)(\langle m_1,\ldots,m_n\rangle) = \texttt{"}f\texttt{(}i\texttt{,}m_1\texttt{(}\ldots\texttt{(}m_n\texttt{(}expr\texttt{))}\ldots\texttt{))"}$$

Reducing over a whole collection is handled as sugar:

seq ::= $expr$ ...

$$R(f,i,seq)(m) = R(f,i,\texttt{tmp for tmp in } expr)(m)$$

As with comprehensions, `if` sequences lead to out-of-line code being generated:

seq_0 ::= seq_1 `if` $expr$

```
generate
    "init : init := i;
     tmp : type;
     if expr then
       other out-of-line additions
       tmp := R(f,init, seq1)(m);
     else
       tmp := init;
     endif;"
R(f, i, seq0)(m) = "tmp"
```

A `for` reduction is converted into a call of a function that explicitly performs the reduction:

seq_0 ::= seq_1 `for` id `:` $element\text{-}type$ `in` $expr$

```
generate
    "function reduce_tmp(init : type; s : expr-type) : type begin
       case x begin
         match expr-type$append(?s1,?s2) begin
           result := reduce_tmp(reduce_tmp(init,s1),s2);
         end;
         match expr-type$single(?id) begin
           other out-of-line additions
           result := R(f,init, seq1)(m);
         end;
         match expr-type$none() begin
           result := init;
         end;
       end;
     end;"
E(seq0)(m) = "reduce_tmp(i,expr)"
```

As with comprehensions, an implicit map is converted by adding the mapping function to the list:

seq ::= $func(seq)$

$$R(f, i, seq_0)(\langle m_1, \ldots, m_n\rangle) = E(f, i, seq_1)(\langle m_1, \ldots, m_n, func\rangle)$$

The overspecification of the form of the reduction (left versus right) means that a special check to test for associative reduction functions is necessary before one can use a better implementation technique for reductions (one that, say, does not use up an amount of stack linear in the length of the collection, as the translation given here does).

4.4.4 Attribution

Broadcast attribution uses pattern matching. After the collection pattern is converted into pattern definitions, broadcast attribution no longer uses anything special. A collecting attribution is expressed as a reduction and is implemented as such. The last two cases, bucket-brigades in either direction, can be translated using the bucket-brigade idiom given in Figure 4.1.

4.5 Summary

The concept of a sequence often crops up when describing programming languages. Support for sequences relieves a description writer from writing sequence boilerplate. It also reduces dangerous redundancy.

In an attribute grammar-like formalism, it is important to support at least the following attribute flows: a broadcast from the parent to all children, a reduction over values for each child and also communication between each child and its nearest neighbors. In APS, sequences and more generally, collections, are provided with powerful pattern matching and comprehension notations. Internally, collections are represented as balanced trees, and all the attribution flows are supported without requiring dangerous redundancy. Collection patterns and collection comprehensions can be translated into simpler patterns and expressions.

Chapter 5

Higher-Order Features

The nodes of a tree being attributed can be viewed as attribute values themselves. This chapter investigates the ramifications of this deceptively simple idea.

For example, when building a symbol table, rather than create a record to store information *about* a declaration, it may be easiest simply to store a reference to the declaration node itself. Then at the point of a use, the symbol table lookup function could return this reference. Depending upon whether the formalism permitted such actions, one could then traverse the subtree rooted at the node referred to, or even query or define its attributes. As long as the tree (or attributes decorating it!) provides information in a useful form, the ability to use node references may simplify a description.

Sometimes, however, the structure of the tree being attributed is more complex than necessary for some task. In this case, it may be useful to create a simplified form of the tree and then carry out the task on this new structure. The results could then be used in other tasks. Compilers often create all sorts of auxiliary data structures: symbol tables, use-def chains, or control flow graphs, just to name a few. Not all of these structures are trees, but still they can be viewed as alternate (possible simplified) representations of the program being compiled. Compilation algorithms are often easier to express on a representation of the program that makes explicit certain relevant properties of the program and omits irrelevant aspects. The ability to define a simplified view of a tree thus aids factoring.

A translation in a compiler can be described using the same facilities. A translation from one intermediate form to another is a computation that reads in a tree and produces a new tree. Chapter 1 argued the benefits of describing compilation as a series of translations, each performing a small step towards the final goal of machine language. By splitting up the task in this way, the various parts of the compiler are more easily reusable and are more resilient to change.

This chapter explores these and other ways in which tree nodes may be treated as values. Because attributes are associated with tree nodes, the ability to store references to nodes in attributes is known as *higher-order attribution*. Section 5.1 describes previous uses of higher-order features in attribution systems. The issues that arise when defining the meaning of higher-order features in the context of attribute grammar-like formalisms are discussed in Section 5.2. In Section 5.3 the support for such features in APS is described and in Section 5.4 the implementation is sketched.

5.1 Higher-Order Features in Attribute Grammars

This section describes higher-order features appearing in the literature and the purpose for which they were used.

Ganzinger and Giegerich's attribute-coupled grammars (ACGs) [39, 41] were the first to use attribute grammars to describe tree translations. An attribute-coupled grammar is simply an attribute grammar that computes a tree using productions in another grammar as node constructors. Ganzinger and Giegerich were also the first to investigate the benefits of descriptional composition. In attribute-coupled grammars, the output tree could only be attributed once attribution was finished for the input tree; there was no possibility of communicating information back from the second attribution to the first. As a result, attribute-coupled grammars were *only* useful for translation, not for producing simplified versions of the tree.

A related formalism is Vogt, Swiestra and Kuiper's higher-order attribute grammars (HOAGs) [92, 96]. Subtrees may be created and then grafted into the tree being attributed. At this point, the newly added subtree may likewise be attributed. One may view HOAGs as a generalization of ACGs in that ACGs describe translations of whole trees at a time whereas in HOAGs a translation can be done for a subtree at a time [92].

Farrow, Marlowe, and Yellin's composable attribute grammars (CAGs) [36] take a different tack. In CAGs, trees may be created that abstract properties in the original tree. In a similar vein, Farnum's optimizer used derived nodes to build a control-flow graph [29]. In either system, the new nodes may be attributed at their creation and attributes may be read from the nodes. These capabilities require node identity and thus it is important that trees are built rather than DAGs. In order to be able to ensure this property statically, a CAG compiler must use a restriction similar to that used to enable descriptional composition in ACGs.

Hedin's Door Attribute Grammars [46] permits nodes to be closely associated with "door objects," whose references may be used as values during attribution. Attributes of the door objects may be read or defined through the references, thus inducing long-range dependencies. Hedin showed that this extension not only simplified the specification but also led to a more efficient incremental implementation.

Interestingly, there has been little if anything published about treating the nodes in an attributed tree as values. The ability to pass around references to nodes and then to fetch attributes from the nodes is convenient for building symbol tables. Section 5.3.6 shows that treating nodes from the attributed tree as values can be reduced to creating and attributing derived nodes as in Farnum's DORA system and in CAGs.

5.2 Issues in Higher-Order Features

When the ability to operate upon tree nodes is added to a formalism, such as an attribute grammar, in which tree nodes induce equations, there are bound to be some complications. This section examines questions, the answers to which profoundly affect the ways in which nodes can be used, and the ways in which the features can be implemented.

5.2.1 Do Nodes Have Identity?

One important issue is whether nodes have identity. The issue arises when one seeks to distinguish a tree from a directed acyclic graph (DAG). In a tree, each node has at most one parent; in a DAG, nodes may have multiple "parents" (also known as "predecessors"). If the nodes returned by a constructor have object identity, then constructor calls are not referentially transparent. That is, if results of dynamic calls to constructors can be distinguished, then it makes a difference whether a constructor is called once and the result used in two places or called twice with the same arguments.

If two nodes that root isomorphic subtrees are indistinguishable, it is not possible in general to fetch attributes from a node passed in an attribute value (see Section 5.2.2), because attributes in general may depend on the context. Thus the presence or absence of node identity makes a great difference in expressiveness. In HOAGs, node identity is absent; subtrees are simply viewed as mappings from inherited attributes to synthesized attributes. Attributes may only be computed for subtrees once they have been rooted in a location and attributes may never be fetched from nodes treated as values. Conversely, in CAGs, attributes may be fetched from nodes treated as values.

On the one hand, referential transparency is a useful property because it makes it easier to transform a program. On the other hand, node identity adds expressiveness to a language. It also permits the useful distinction between trees and DAGs. Node identity can of course be simulated in a referentially transparent language, but at the source-level, constructor calls are either referentially transparent or they are not.

5.2.2 May Attributes Be Read?

As mentioned in the previous section, being able to read attributes of nodes passed as values requires that nodes have identity. The ability to read attributes also complicates the dependency graph. Many efficient evaluation mechanisms have been developed for attribute grammars, exploiting the fact that in classical attribute grammars, all (direct) dependencies are between neighboring nodes in the parse tree. The ability to form direct remote dependencies makes the dependencies much harder to analyze. Johnson and Fischer's non-local attribute grammars require hand-written annotations that inform the attribute scheduler of the kinds of remote dependencies that are possible [50]. More recently, Farrow has proposed an analysis to automate the scheduling [35]. This work has been extended specifically to handle remote attribute access of attributed nodes [11]. Maddox has implemented a corrected version of Farrow's fiber analysis [70].

5.2.3 May Attributes Be Written?

In HOAGs, the inherited attributes for a grafted subtree may be written. In CAGs, the *input* attributes of a node are defined by the attribution rule that creates it. Input attributes are associated with nodes like other attributes, but are defined by the rule that creates the node, not by rules that match its structure. It is not necessary that the creation of the node and the definition of its input attributes occur at the same time during attribution.

In both HOAGs and CAGs, attributes may only be written in these restricted situations. Since HOAGs preclude node identity, there would be no other way to write attributes anyway. But even in CAGs, where node identity is used, attributes may only be written for nodes in the attribution rule where they are created. Allowing writes in any rule where a reference to a node is available runs into problems. It is much more difficult, if not impossible, to guarantee statically that an attribute will receive only a single definition. Even lexical ordering is not sufficient to avoid conflict since the the same attribute definition may be instantiated multiple times in a tree. There is no guarantee that the node used in each definition will be different. The problem is similar to that for *non-controlling* bindings, as described in Chapter 3. Only if there is some way to resolve multiple definitions cleanly, can unrestricted writes be well defined.

5.2.4 Can Cyclic Structures Be Built?

Many useful structures in compilers are cyclic. For example, symbol tables often have cyclic entries for representing self-referential types such as

```
TYPE Node = RECORD
               item:INTEGER;
               next:POINTER TO Node;
            END;
```

Since `Node` is a record type that refers to itself, the symbol table entry for `Node` will be a structure that also refers to itself. The cycle can be cut by representing pointer types specially, but this fix makes things more difficult for the description writer. Such a fix must be carefully described and a new fix is needed for every potentially circular structure. If cycles are forbidden, neither call graphs nor control flow graphs can be represented naturally, to name a few common cyclic structures.

If a constructor is *lazy*, that is, if it can return a valid value before all the arguments are ready for evaluation, then creating cycles is straightforward. Augusteijn has proposed a similar, if less general, solution: one may mark certain actual parameters as lazy [7]. Otherwise, the possibly cyclic edges must be added later. In Olga, a limited form of side-effecting assignment is permitted to fix the parameter to a constructor for the purpose of building cyclic structures [53].

A different way to accomplish the effect of cyclic structures is to use CAG-style input attributes. This method entails assigning attributes to the nodes after they have been created. If a formalism (for example, APS) has both constructor arguments (arguments used to create the node in the first place) and input attributes, and pattern matching can only be used for constructor arguments, then cyclic "edges" defined with input attributes will be harder to traverse.

In any case, cyclic structures require special analysis to achieve correct and efficient evaluation. Computing the dependencies and determining a correct evaluation order can be difficult. Farrow's "fibering" technique was specifically designed in order to provide dependency analysis of cyclic structures [35].

5.2.5 When Can Nodes be Created?

In ACGs and HOAGs, all node creation is done before any evaluation of the attributes of the tree or subtree in which it resides. If it is legal to add nodes to a tree after attribute evaluation has started, or more specifically, if it is possible to create new nodes using values determined during attribution, it can be difficult to determine a safe evaluation order for attributes. A CAG is denoted *non-separable*, if attribute values of the tree being created are used to direction further node creation in the same tree [36]. For example, a node that was once a root may suddenly get a parent. If parent's existence depended indirectly on an attribute that has a different value for a node with a parent versus without a parent, circularity is the result. Munson's Proteus system allows nodes in a tree to be *elaborated*, that is, to be transformed into several new nodes, but this capability is strictly limited so that elaboration can be performed before attribute evaluation [44].

5.3 Higher-Order Features in APS

In APS, nodes may be created in the manner of CAGs, and their input attributes may be defined. APS also includes the novel ability to refer to nodes remotely by reference; tree node references may be passed around in attributes. Higher-order features are used extensively in the Oberon2 compiler and the APS compiler itself.

5.3.1 Node References

In APS, it is legal to refer to the nodes of the tree being attributed. This feature is used heavily in the symbol table module. In order to distinguish node references from nodes as syntactic entities grounded in the tree, APS requires that the type of a node reference to be declared as *remote*. For example, after name resolution in the Oberon2 compiler is done, every use of a named entity has a reference to the declaration to which it is bound:

```
attribute Use.use_decl : remote Declaration := nil;
```

This fragment declares an attribute `use_decl` that for every `Use` refers to a `Declaration`. `Use` and `Declaration` are both phyla in the abstract Oberon2 tree language. The keyword `remote` converts a phylum into a normal type. Any node may be implicitly coerced into a `remote` node reference. The default value for this attribute is `nil`, a polymorphic node reference that is different from any constructed node reference.

Node references may participate in pattern matching. Figure 5.1 contains APS code that checks whether local procedures (procedures declared local to another procedure) are used properly. In Oberon2, references to local procedures cannot be stored in variables or passed to other procedures; local procedures may only be called. The first two sections define boolean attributes: `proc_is_local` is true for procedures declared locally to another procedure; `proc_called` is true for procedure expressions that participate in calls. The last section is the one that uses the `use_decl` node reference.

For any identifier expression, the clause first checks whether the use refers to a procedure declaration. If so, and the procedure is local and the expression is not an immediate call, an error message is generated. The nested pattern match is performed

```
attribute Declaration.proc_is_local : Boolean := false;
match proc_decl(body:=block(decls:={...,?d=proc_decl(...),...})) begin
  d.proc_is_local := true;
end;

attribute Expression.called : Boolean := false;
match funcall(?proc,?) begin
  proc.called := true;
end;
match call_stmt(?call) begin
  call.called := true;
end;

match ?e=named_expr(?using) begin
  case using.use_decl begin
    match ?proc=proc_decl(...) begin
      if proc.proc_is_local and not e.called then
        generate an error message
      endif;
    end;
  end;
end;
```

Figure 5.1: Example of Using Node References from Appendix B.5

using a node reference as a subject and the `proc_is_local` attribute is fetched from a node reference.

Node references (including `nil`) may be compared using `==`, the node identity predicate. Node references (but not `nil`) in a completely built tree may be compared using `<<`, a lexical ordering predicate. Chapter 4 illustrated a fragment of the Oberon2 compiler that used these predicates:

```
private type SortedDecls := ORDERED_SET[remote Decl]((==),(<<));

var function find_local_decl(name : Symbol; scope : remote Contour)
    : remote Decl
begin
  case SortedDecls${decl if decl_name(decl) = name
                        for decl in scope.local_decls}
  begin
    match {?first,...} begin
      result := first;
    end;
  else
    result := Decl$nil;
  end;
end;
```

In the declaration of `SortedDecls`, node identity `(==)` is used to distinguish elements in the set and lexical ordering `(<<)` is used to order the set. In the body of `find_local_decl` therefore, the lexically first declaration that matches the name is the one returned. A function, such as this one, that fetches attributes from references to nodes must be declared `var`. A `var` function can be used by a client of a module only after the tree is built and attributed.

A node reference may refer to any other node in the tree. Therefore node references are never *controlling*. In particular, it is illegal to assign attributes through node references. Similarly no pattern variable bound in a pattern match on a subtree found through a remote reference is controlling. Chapter 6 describes a type of attribute that allows non-controlling assignments.

5.3.2 Creating Nodes

In APS, each call to a constructor creates a new node. In a module performing a translation, nodes are created for an output tree designated by a type. For example, in the Oberon2 compiler, a translation module converts between the Oberon2 abstract syntax and an intermediate form used to communicate with the *gcc* back-end:

```
-- Convert abstract Oberon2 into the GCC tree language defined in
-- gcc-tree.aps in order to interface to rest of GCC compiler.
module OBERON2_TRANSLATE[T :: var OBERON2_TREE[],
                              var OBERON2_RESOLVE[T],
```

```
                                var OBERON2_MACHINE_SIZES[],
                                var OBERON2_COMPILE_COMPUTE[T],
                                var OBERON2_LAYOUT[T]]
    extends T
begin
  type BareGccTree := GCC_TREE[](address_size);
  type GccTree := OBERON2_GENERATE_RUNTIME[BareGccTree];

  attribute Declaration.gcc_decl : GccTree$Declaration;
  ⋮
end;
```

This module uses many of the other modules that attribute abstract Oberon2 trees. It defines a type `GccTree` that is a new instance of the `GCC_TREE` module. Inside this module attributes are declared (for instance `gcc_decl`) that have *syntactic type*, that is, whose type is a phylum of the tree being created. Such attributes are also known as *syntactic attributes*; a value of syntactic type, such as a syntactic attribute of a node, or a local value of syntactic type is known as a *syntactic value*.

The constructors must create tree nodes, not DAGs. This property is enforced through two restrictions. First, syntactic attributes may be fetched only from controlling bindings. More precisely, if a syntactic attribute is fetched from a remote or non-controlling binding, the value is coerced into a node reference. This restriction avoids the problem of using syntactic attributes of non-controlling bindings in more than one place. Without this restriction, it would be possible to build DAGs.

The second restriction is that within any attribution clause, each syntactic value may be used at most once. One use in each branch of a conditional counts as a single use. The check is performed as follows. For every conditional or deterministic pattern match, one branch is chosen. If the resulting clause has more than one lexical use of a syntactic value, the attribution clause is illegal. If no such choice leads to more than one lexical use, it is legal. We have termed this restriction SAMODUR (syntactic at most one dynamic use requirement) [14] and it is a carefully designed weakening of Ganzinger and Giegerich's SSUR (syntactic single use requirement) [39]. Actually SAMODUR applies to the whole set of clauses in a normal attribute grammar and so is stricter that the clause-local restriction spelled out here. Lexical ordering is used again to ensure that two syntactic uses in different clauses do not conflict. The current APS compiler does not check these restrictions, but the run-time system ensures that each node has at most one parent.

All the nodes of a phylum must be created before any attributes of the nodes are needed. Since APS programs do not have an explicit evaluation order, this restriction is expressed by an implicit dependency for every attribute on every node creation. As elaborated further in Chapter 8, this restriction implies that an extending module may either add new nodes to the tree or read attributes from the extended module, but not both.

5.3.3 Abstracting Node Creation

Since node creation is not a referentially transparent process and since all the nodes of a phylum must be created before attribution begins, APS functions are not permitted to create (attributable) nodes. However, APS provides *procedures* for abstracting node creation. They may also be used to abstract the other non-referentially transparent operations introduced in Chapter 6. Procedures may create nodes by calling constructors or other procedures.

For example, in Appendix B.2, a procedure is declared that behaves like a constructor, providing a short-cut for creating Oberon2 type declaration nodes:

```
procedure make_global_type(name : String; base : Type) : Declaration
    := type_decl(identifier(make_symbol(name),exported()),base);
```

Procedures are thus important for factoring purposes.

5.3.4 Built-In Node References

The previous section listed restrictions to ensure that trees, not DAGs, were built. Some data structures used by compilers, such as representations of basic blocks, are in the form of DAGs. For this reason, APS supports DAGs through node reference parameters for constructors. Node reference parameters are declared using a `remote` type. For example, in the scope module that defines the `Contour` phylum, the following two constructors are defined (see Appendix B.3.1):

```
constructor root_contour() : Contour;
constructor nested_contour(parent : remote Contour) : Contour;
```

The first constructor is used to create the root contour, the one that holds declarations of global scope. The second constructor is used to create sub-scopes nested in another scope. Since a scope may have many sub-scopes, the parent scope field of the constructor cannot be a child, it must be a remote reference to the enclosing scope. Since constructors are strict in APS (all the parameter values must be available before a node is returned), node reference parameters can never introduce cycles. For example, it is not possible to create a scope that is nested in itself.

5.3.5 Input Attributes

When cyclic structures are desired, they must be accomplished through *input attributes* as in CAGs. Input attributes must be defined at the point in the rule which creates the node. The interface to the GCC tree has many input attributes; the following fragment from Appendix B.6.1 shows one:

```
-- Uses permit cyclic references within the tree:
constructor a_use() : Use;
input attribute Use.use_decl : remote Declaration := nil;
```

Here every `Use` in the intermediate tree may refer to a `Declaration` somewhere else in the tree, with no requirement of non-circularity.

The module also provides a procedure that does the assignment of the input attribute at the same time:

```
procedure use_remote(d : remote Declaration) : Use begin
  result := a_use();
  result.use_decl := d;
end;
```

This procedure can be used to create a `Use` node that refers to a declaration that has already been created. This example demonstrates that procedures in APS can be thought of as abstractions over attribution clauses, whereas functions in APS are abstractions over expressions.

5.3.6 Auxiliary Trees

Instead of using the nodes of the tree being attributed as values, it is always possible to create an auxiliary node of a local phylum, assign some attributes and then pass a reference to this node instead. This paradigm is used in CAGs. It may be useful as a way to provide simplified views of the nodes. For example, in the Oberon2 compiler, the symbol table module passes around instances of auxiliary nodes representing contours rather than passing around references to the nodes in the abstract parse tree that establish scopes.

The same dependency issues arise, however. It may be necessary to perform some sort of "fibering" analysis to obtain a useful static approximation to the dynamic dependency graph.

5.3.7 Referentially-Transparent Node Creation

Not all values need object identity and so APS provides a way to define structured terms without object identity. Constructors are used to declare the shape of the values, but the result type of a *pure constructor* is a type, not a phylum.

Two isomorphic structured values are indistinguishable. Such objects may not be attributed. The Oberon2 compiler uses such values to represent constant values. The following declarations come from Appendix B.2.

```
constructor shortint_constant(value : Integer) : Constant;
constructor integer_constant(value : Integer) : Constant;
⋮
pattern some_integer_constant(x : Integer) : Constant
    := shortint_constant(?x),integer_constant(?x),longint_constant(?x);

zero : Constant := shortint_constant(0);
```

These constructors are used in the same way as constructors that create nodes with identity. However, pure constructors may also be called by functions, as seen in this example presented in Chapter 2:

```
function make_range(x,y : Constant) : Constants begin
  case Constants${x,y} begin
    match {some_integer_constant(?v1),
           some_integer_constant(?v2)} begin
      result := {shortint_constant(i) for i : Integer in v1..v2};
    end;
  else
    result := {nil};
  end;
end;
```

The pure constructor `shortint_constant` is called for every integer in the range.

5.4 Implementation of Higher-Order Features

Few of the features described here make any real change to the attribute grammar paradigm and thus involve any particularly interesting implementation issues. The distinction between nodes with identity and those without is reflected in the implementation in that it is possible to enumerate the nodes of a phylum. Moreover for each phylum there is a scheduled *closing* time, a point after which no more new nodes may be created for it. At that point, the attributes for the nodes of the phylum are allocated.

Each node of a phylum must have a unique parent. Section 5.3.2 gave static restrictions on the creation of nodes that ensure this property. The current APS compiler does not check creation statically; instead the run-time system signals an error if a node is given a second parent.

Lexical ordering between nodes (that is, the `<<` operator) is handled by examining the tree which contains the nodes. If the nodes are in disjoint trees, an artificial tree is created containing both of the trees. These artificial trees are created only after the phyla at the roots have been closed. Thus lexical ordering is always consistently (if arbitrarily) defined.

As mentioned in Section 5.2.2, direct dependencies between non-neighboring nodes make the dependency graph more complicated. Currently, the APS implementation uses dynamic attribute scheduling which makes remote dependencies trivial to handle. Eventually, the APS compiler may perform static "fibering" analysis.

5.5 Summary

Higher-order features add much expressiveness to a compiler description language as witnessed by the examples from the Oberon2 compiler. The use of higher-order features, however, complicates static dependency analysis. APS provides a full set of higher-order features, and is the first attribute-grammar based system to provide many of them.

Chapter 6

Remote Attribution

Chapter 5 described features that permitted references to nodes in the attributed tree to be transmitted through the attribute system. The attributes of such nodes could be queried there, far from the point where the attributes were defined. The question then arises whether it should be possible also to *define* attributes of such nodes. This chapter discusses this ability, *remote attribution*. The first section answers several questions: why remote attribution is useful; how it helps in factoring a description, and what remote attribution really means. The second section describes how remote attribution is supported in APS through the device of *collection attributes*. The final section discusses implementation.

6.1 Motivation

The higher-order features described in Chapter 5 permit attributes to be queried at remote locations, thus providing information flow from the location of the node in the tree to the place where the reference to the node is used. But these features do not permit attributes to flow in the reverse direction. Why might such reverse flow be useful? It is useful in the case that information must flow not only from a declaration to a use, but also from the use to the declaration. For example, determining whether a declaration has been used at all requires such reverse flow. Similarly, a compiler for an imperative language can perform certain operations if it knows a certain variable is never assigned a value. The number of times a variable binding is used may also be useful; if a variable is used no more than once, it may be possible to replace its sole use with its definition. Thus, the ability to transmit information from where a node reference is used back to its location in the tree is useful.

Of course, determining whether a declaration is used at least once does not require the ability to write attributes remotely. Instead a description writer could write rules for transmitting values back to the declaration node. These rules would parallel the rules that transmitted the node, but information would flow in the reverse direction. If multiple pieces of information were needed, either multiple parallel attribute definitions could be written or the values could be packaged up and sent with one attribute definition. In either case, one ends up with a dangerously redundant specification—the same path is specified more than once and there is no formal mechanism for checking that the specifications are indeed identical. Moreover, choosing whether the values can be packaged together or not depends on whether the two values have compatible dependencies. Determining dependencies and legal packagings should be left to the compiler generator, rather than being hand-specified by the description writer.

There are problems with writing attributes remotely. As described in Section 5.2.3, it can be impossible to determine statically whether an attribute is defined at all, or even whether it is multiply defined. The same issues arise when using normal attribution to avoid the need for remote attribution. For example, if each expression defines an attribute that specifies which declarations are used in the expression, then it would be necessary to combine information for the subexpressions of an expression. If a declaration is used in *any* of the subexpressions, then it is used in the full expression. If one wishes to keep track of the number of uses, it is necessary to combine the counts from each subexpression for each declaration separately.

Thus, in remote attribution there must be some way to specify the ways in which disparate definitions are combined. The following section describes a class of attributes, *collection attributes*, that may participate in remote attribution. Section 6.3 discusses the implementation of these attributes.

6.2 Collection Attributes in APS

This section describes collection attributes in APS, as well as global and local collections. It concludes with more discussion of *procedures*, which were introduced in the previous chapter.

6.2.1 Collection Attributes

Collection attributes in APS are attributes that may be given multiple definitions. These definitions are combined into a single value using a combining operator specified in the attribute declaration. For example, in the Oberon2 compiler (see Appendix B.5), there is a collection attribute for each declaration that records whether that declaration is used anywhere in the program:

```
collection attribute Declaration.decl_used : Boolean :> false, (or);
```

A collection attribute is distinguished by the keyword `collection`. The default value is specified using `:>` in place of `:=` and has two parts, an initial value (used to prime the collection process) and a combining function. If no default is given, the type must satisfy the `COMBINABLE` signature (see Section 8.2.3), and the initial value and combining function are inferred from the type. In this case, the default is `false` and the combining function is `or`. (Since `or` is an infix operator, it must be surrounded with parentheses to make a valid APS expression.) The combining function should be both associative and commutative[1], because the collection attribute definitions are unordered.

All attribute definitions of a collection attribute are used; lexical ordering is irrelevant. In a definition of a collection attribute, any node value of the appropriate phylum

[1] The current APS compiler does not check these properties.

may be used. It may be `remote` or from a non-controlling binding (see Chapter 3). For example, the definitions for `decl_is_used` come from every use in the program:

```
match ?u:Use begin
  if u.use_decl /= nil then
    u.use_decl.decl_used :> true;
  endif;
end;
```

The special definition syntax: *lval* `:>` *rval* is used for defining collection attributes. The `:>` operator evokes the mathematical $>$ operator. Rather than specifying the final value for *lval*, it merely adds a constraint on the final value.

Any use of a collection attribute depends implicitly on every definition, and is guaranteed to retrieve a complete definition. Such guarantees are difficult in an imperative framework, where the collection is updated as a side-effect. Since APS is declarative, each attribute has a single fixed value; automatic dependency analysis ensures this property. For example, the `decl_used` attribute can be used to generate warning messages for unused unexported entities. (Exported entities may also get a use from outside the module.)

```
match ?d=declaration(identifier(?,not_exported())) begin
  if not d.decl_used then
    add_error(d,"Warning: declaration not used");
  endif;
end;
```

Collection attributes are used like any other attribute. This fragment contains an example of a *procedure call* (see Section 6.2.3).

The power of collection attributes (and thus remote attribution in general) is demonstrated in that in a mere ten lines of code, the author was able to add detection of unused variables. The resulting specification is still declarative; execution order is not specified, only the necessary computations. It was unnecessary to define and combine sets to track used identifiers and transport these sets through the whole tree. Collection attributes are an important high-level abstraction.

6.2.2 Global and Local Collection Variables

Not only attributes of tree nodes, but both global and local variables may be collections as well. For example, the clause for checking Oberon2 case statements has a local collection variable:

```
type CaseSet := ORDERED_SET[remote CaseLabel]((==),(<<));
match case_stmt(?,?clauses,?) begin
  collection labels : CaseSet;
  -- first we flatten the case statement
  -- and select only labels without errors
  for clauses begin
    match Cases${...,case_clause({...,?label,...},?),...} begin
```

```
      case label begin
        match single_label(?) begin
          labels :> {label};
        end;
        match range_label(?e1,?e2) begin
          if e1.constant_value > e2.constant_value then
            add_error(label,"Empty range");
          else
            labels :> {label};
          endif;
        end;
      end; -- case label
    end;
  end; -- for clauses
  ⋮
end; -- match case_stmt
```

The local collection variable `labels` is an ordered set of the labels occurring in the case statement. Since no default is given for `labels`, the initial value and combining function are inferred from the type. All `SET` types are `COMBINABLE`; the inferred initial value is the empty set, and the combining function is set union.

Despite the fact that this syntax looks imperative, it is not; the value of a collection variable incorporates all its definitions, not just the ones lexically before the use.

6.2.3 Procedures

As mentioned in Chapter 5, whereas functions abstract expressions, procedures abstract attribution clauses. One example used in the Oberon2 compiler is the `add_error` procedure used in the preceding example. It is defined for a polymorphic type `Node` that ranges over most of the phyla in the Oberon2 abstract tree language:

```
collection attribute Node.errors : Errors;

-- an internal way to report an error
private procedure add_error(node : remote Node; message : String) begin
  node.errors :> {message};
end;
```

The message is put into a set and then added to the `errors` collection attribute for that node. Another example of a procedure from the same file is `ensure_constant`:

```
procedure ensure_constant(x : remote Expression)
begin
  if not x.expr_constant then
    add_error(x,"not constant");
  endif;
end;
```

This procedure includes a check on an attribute and otherwise calls the first procedure. Procedures are an important factoring technique that avoid repetitive definitions. Without procedures, there would be no way to factor out commonalities in attribute definitions.

Since procedures add constraints to an attribution system, they may only be called from other procedures or in attribute clauses. In particular, they may not be called by functions.

The procedures shown here have no return value. As the examples from Chapter 5 show, procedures may also return values.

6.3 Implementation of Collection Attributes

As with the ability to read remote attributes from a node, the most natural implementation of remote attribution is to operate on the node reference. In Chapter 5, the operation was to fetch the attribute from the node. In this case, the operation is to activate a definition of the attribute of the node. The compiler ensures that all definitions of a collection attribute are activated before the value is used. More generally, the compiler ensures that each *collection variable instance* (that is, an instance of a local or global collection variable, or an instance of a collection attributes for a particular node) depends on all its definitions.

As described in Chapter 2, the compiler for APS uses demand evaluation. This section shows how collections are implemented in that context. Alternatively, earlier work has shown that one can use fiber analysis to achieve a static schedule [11].

6.3.1 Locating Definitions

In Chapter 5, remote dependency tracking was straightforward because of the asymmetry: values might be demanded non-locally, but they were defined locally. In that case demand evaluation works easily. Similarly, all the definitions of an instance of a global or local collection variable can be easily determined. Local collection variables can only be defined local to an attribution rule in any case and global collection variable definitions can be found by traversing the tree. However, it is harder to determine all the definitions of the instance of a collection attribute for a particular node.

This difficulty can be solved by noting that although one cannot tell which definitions of a collection attribute are relevant for a particular instance of that attribute, it *is* possible to tell that *some* instance of the collection attribute is being defined. The compiler adds artificial dependencies to ensure that no instance of a collection attribute is evaluated before it is determined which node is being attributed at every definition of that attribute.

As mentioned in Section 2.3.2, an imprecise guard variable is introduced for every attribute (and that includes collection attributes). Every instance of the collection attribute is made to depend on the imprecise guard. In turn, the guard is made to depend on every expression evaluating to a reference to a node for which that collection attribute is defined. Once the imprecise guard has been evaluated, each definition of the collection attribute can be assigned to the particular attribute instance.

It may seem that this solution will induce circularities whenever one collection attribute instance depends on another. It does not because the solution does not lead to dependencies between collection attribute instances but between collection attribute instances and expressions that compute the *references* to the nodes being attributed. Only if one has a situation analogous to

node`.some_collection.some_collection :>` *value*

where the value of a collection attribute instance is a node being used for the definition of another (or possibly the same) attribute instance will this solution detect a circularity.

If the most natural solution to some problem leads to collection attribute definitions of this nature, it may be better to use a circular collection attribute, introduced in Chapter 7.

6.3.2 Combining Values

Once all definitions are allocated to particular collection variable instances, demand evaluation can continue as before. No priority ordering is used for collection variable instances since all definitions contribute to the final value. Then, when the value of the collection variable instance is demanded, all the values for the definitions are demanded. The returning values are combined into a single value using the combining function.

6.4 Summary

This section introduced a novel feature in APS: collection attributes. Collection attributes permit non-local definitions, since there is no need to ensure that a collection attribute for a node has only one definition. Remote attribution is a powerful mechanism for transmitting information in the reverse direction of higher-order attributes. Collection attributes can be added to a demand-driven model by ensuring each instance of a collection attribute can only be evaluated after determining the nodes being attributed at every definition site of the attribute.

Chapter 7

Circular Attribution

Abstract interpretation and more specifically constraint systems over lattices (for example Aiken's set constraints [2]) have been used to specify type-checking or to assist optimization (see, for example Kennedy's methods for specifying data-flow [60]). Often these specifications lead to circular systems of constraints.

Constraint systems where each constraint has the form $v_0 \geq f(v_1, v_2, \ldots, v_n)$ for variables v_i can sometimes be solved. A function f is said to be *monotonic* if substituting successively greater values for its parameters yields a greater value for the result:

$$(v_1' \geq v_1 \wedge v_2' \geq v_2 \wedge \ldots \wedge v_n' \geq v_n) \Longrightarrow f(v_1', v_2', \ldots, v_n') \geq f(v_1, v_2, \ldots, v_n)$$

where $\geq$ is the ordering relation of the lattice. More generally, a function may be monotonic with respect to only some of its parameters. A constraint of the form $v_0 \geq f(v_1, v_2, \ldots, v_n)$ is likewise said to be monotonic (with respect to some subset $V \subseteq \{v_1, v_2, \ldots, v_n\}$) if f is monotonic (with respect to the corresponding parameters). If all the constraints in the system are monotonic and the lattice is finite, a least fixpoint exists and is computable. This chapter shows how such circular systems of constraints can be expressed in a natural way in an attribute grammar formalism, and in particular in APS.

7.1 Circular Attribution in Attribute Grammars

In classical attribute grammar terminology, a *well-formed* attribute grammar is precisely one that does not induce a circular dependency for any tree defined over its context-free grammar. Even the most flexible implementation methods for attribute grammars fail in the presence of circular dependencies, as do any of the more efficient techniques that implement only a subset of the well-formed attribute grammars. Moreover, the kinds of dependencies induced by a (monotonic) set-constraint system are awkward to express without cyclic dependencies. At worst, all one can do is define function-valued attributes that are used by an out-of-line function to compute the fixpoint. Thus any extension of attribute grammars to embrace circular dependencies is likely to require more than a simple transformation to classical attribute grammars; new evaluations techniques must also be defined.

Before discussing the extensions that achieve the semantics needed to support set constraints, two other extensions will be discussed. Augusteijn's Elegant system [7] permits the description writer to mark certain uses of an attribute as lazy and thus explicitly break circularity. This feature is used to build circular structures. However, the semantics of this extension does not admit static dependency analysis if the circular structure is traversed elsewhere in the attribute grammar. This restriction is lifted if one instead uses Farrow's fibering analysis [35]. An alternative approach, one that permits attribute grammars to describe dynamic semantics, is exemplified by Walz and Johnson's "gated attribute grammars," [100] in which attributes may change values, in that special "gated" attributes have two definitions, one of which breaks the cycle. The same effect can be achieved more elegantly using Parigot el al.'s "scheme productions" [76] in which the computation is described over a possible infinite tree, rather than with circular dependencies among attributes.

This chapter, however, primarily concerns the expression of set-constraints and similar recursive definitions in attribute grammars. Farrow has shown that attribute grammars can be extended to handle circular attribution and gives two implementation methods, one using a dynamic dependency graph, and one using static dependency analysis [33]. In either case, the type of the attributes along a cycle must have a complete partial order with an equality test, and the dependencies must be monotonic and satisfy the ascending chain condition. For an attribute dependency expressed as a function f of one other attribute in the cycle, the latter condition can be expressed mathematically:

$$\begin{aligned} \forall_{d_1 < d_2}\, f(d_1) &\leq f(d_2) \text{ (monotonicity)} \\ \forall_{d_1 < d_2 < \ldots}\, \exists_{k>0} f(d_k) &= f(d_{k+1}) \text{ (ascending chain condition)} \end{aligned}$$

If these conditions are met, Farrow calls the attribute grammar a *finitely recursive* attribute grammar. Such an attribute grammar can be evaluated using successive approximations starting from the least element in the complete partial order, and stopping when a fixpoint is reached. The monotonicity condition ensures that a fixpoint exists and the ascending chain condition ensures that successive approximation will reach it.

Farnum's DORA system (discussed in regard to pattern matching in Chapter 3) also provides "recursive attributes" defined over lattices. These attributes can have circular dependencies and they are implemented through the same technique of successive approximation. DORA does not use static analysis, but neither does it build a dynamic dependency graph. Instead all instances of the attribute are evaluated together (possibly with all the instances of other recursive attributes). As a result, it is an error if a non-recursive attribute in the dependency graph lies between two instances of the same recursive attribute, even if it is not involved in a circular dependency itself. However, recursive attributes in DORA can have more complex dependencies than finite recursive attribute grammars. A recursive attribute may be assigned more than once, and an attribute instance being assigned can be computed using other attributes. (In APS terminology, recursive attributes are "collection attributes" as well as permitting circular dependencies.) Thus one can express dependencies such as

$$\forall_{y \in A(x)}\, A(y) \supseteq A(x).$$

This expressive power is useful in forming transitive closures. Farnum's dissertation [29] shows how Shiver's control-flow analysis [84] can be expressed in DORA. Recursive attributes must be defined over a lattice, a (complete) partial order with a least upper bound operator (written $\vee$)[1] that satisfies the following laws:

$$\begin{aligned} x \vee y &\geq x \\ x \vee y &\geq y \\ z \geq x, y &\iff z \geq x \vee y \end{aligned}$$

The dependency functions along a cycle must be monotonic. The DORA system does not check any of these properties.

In related work, Aßmann's OPTIMIX optimizer generator [4] uses *edge-addition rewrite systems* to specify circular program analyses. The restriction to edge addition (as opposed to also permitting edge removal) corresponds to the monotonic dependency condition for DORA.

7.2 Circular Attribution in APS

Circular dependencies are provided in APS through *circular* and *circular collection* values. The first kind corresponds roughly to those supported by Farrow's system and the second to Farnum's recursive attributes. This section describes each kind using examples. Following this introduction is a discussion of various semantic issues.

7.2.1 Circular Attributes

The Oberon2 compiler does not use any circular attributes, but the APS compiler does. Chapter 3 defined whether a pattern variable is "controlling" using set constraints. These set constraints are implemented using circular attributes in Appendix C.5.1. Two circular attributes are declared, one for the $C(\phi)$ (called `first_constructors`[2]) and one for $P(\phi)$ (called `positions`):

```
type Positions := ORDERED_SET[Position]((=),position_less);
type PositionsLattice := UNION_LATTICE[Position,Positions];
⋮
circular attribute Pattern.positions : PositionsLattice;
⋮
circular attribute Pattern.first_constructors : ConstructorsLattice;
```

The type of a circular attribute must be a lattice or at least a complete partial order. Such types have a bottom value and an order predicate. The lattices created by `UNION_LATTICE` have the empty set for bottom, and use subset inclusion for ordering. The corresponding `INTERSECTION_LATTICE` module requires a universal set as well as the element and set types; it creates lattices with the reverse ordering relation, superset inclusion.

Circular attributes are assigned just as normal attributes:

[1] $\sqcup$ is also sometimes used.

[2] This set is later culled down by the pattern context to get the final set `constructors`. This optimization (mentioned in Chapter 3) makes the transformation of pattern matching more efficient.

```
match ?p=choice_pattern(?p1,?p2) begin
  p.first_constructors :=
      p1.first_constructors |\/| p2.first_constructors;
  p1.positions := p.positions;
  p2.positions := p.positions;
end;
```

Here the `|\/|` operator is the *join* operator for the lattice, in this case set union; the join operation is guaranteed monotonic in its arguments. As with normal attributes, circular attributes obey the lexical ordering rule for handling multiple applicable definitions: the first such definition is the one used.

7.2.2 Circular Collection Attributes

Circular collection attributes combine the features of circular attributes (a fixpoint is computed) with collection attributes (remote attribution and multiple definitions are possible). The type of a circular collection attribute must be a `LATTICE` type; a join operation is required as well as the bottom value and ordering relation used by circular attributes. A `LATTICE` type's join operation should satisfy the laws of a least upper bound operator, whereas there is no such need for the combining operation of a `COMBINABLE` type. In particular, `T$join(x,x)` must be the same as `x` for a `LATTICE` type `T`, but `T$combine(x,x)` need not be the same as `x` for a `COMBINABLE` type `T`. For example, addition is an acceptable (and useful) combining operation, but could not serve as a join operation. The APS compiler does not currently check these laws, but all the standard type constructors that create `LATTICE` types observe the laws.

The circular collection value in this example happens to be a local value:

```
-- we collect up all the positions for the body ...
circular collection body_positions : PositionsLattice;
-- ... and then assign them:
body.positions := body_positions;

body_positions :> p.positions;
for body begin
  match pattern_scope(?holep=hole()) begin
    holep.first_constructors := body.first_constructors;
    body_positions :> holep.positions;
  end;
end;
```

The join operation for a `UNION_LATTICE` such as `PositionLattice` is set union. The local value `body_positions` collects together all the positions for recursive calls to the pattern definition (the holes) into a single set. It must be declared circular, because it may depend indirectly on the positions for the body of the definition. It must be declared a collection because it is assigned multiple times; we wish to use the positions of *every* hole in the pattern. As seen here, circular collection attributes are assigned as collection attributes, using the `:>` syntax.

In addition to `LATTICE` types created by `UNION_LATTICE` and `INTERSECTION` lattice, several other lattices or lattice creating modules are standard in APS. There are two predefined boolean lattices, `OrLattice` which orders `false` before `true` and its reverse `AndLattice`. The boolean lattices are treated specially by the APS compiler (see Sections 7.2.4 and 7.2.5). The `MAX_LATTICE` and `MIN_LATTICE` modules create lattices over `COMPARABLE` types, the first uses the comparison relation as the lattice ordering relation, and the second uses its reverse.

All `LATTICE` types are created as extensions (see Chapter 8) over the base type of the lattice. As a result, a value of the `LATTICE` type can be used wherever a value of the base type is expected and vice versa. Were it not for this fact, one would need to convert back and forth, and circular attributes would be much less convenient.

7.2.3 Successive Approximation and Normal Values

The most important aspect of circular attributes is that, despite the fact that they are computed using successive approximation, only the fixpoints are used to compute normal values (those not declared circular). Moreover, none of the intermediate values are *visible*, that is, affect the construction of nodes, the assignment of normal attributes or external procedure calls.

Values not declared circular must not take part in any cycles. Dependencies between value instances one of which is not circular are *simple* dependencies, as are non-monotonic dependencies between circular attributes. Any chain of dependencies including at least one simple dependency link is called a *simple dependency chain.* Simple dependencies may separate strongly-connected subsets of circular instances, but any cycle of dependencies involving a simple dependency is illegal.

7.2.4 Monotonicity

The distinction between simple and monotonic dependencies combined with the ability to describe strongly-connected sets of circular instances separated by simple dependencies gives us more expressiveness than DORA's recursive attributes, but also requires more analysis. In particular, simple (that is non-monotonic) dependencies between two circular instances are permitted. Unlike DORA, the APS compiler checks dependencies for monotonicity.

Even an implicit conversion can be non-monotonic. For example:

```
type MaxIntegerLattice := MAX_LATTICE[Integer](0);
type MinIntegerLattice := MIN_LATTICE[Integer](1000);

circular a : MaxIntegerLattice;
circular b : MinIntegerLattice;

a := b;                 -- non-monotonic!
```

The last assignment is type correct because both variables store integers. However, the use of `b` is this fragment is non-monotonic because it is not of the lattice of the left-hand side of the equation. This situation must be the case since successive approximations of `a` increase from 0 and those of `b` decrease from 1000. If an approximation for `b` were used, rather than its final value, `a` could end up with a value too large.

The use of a circular value in an expression is taken to be monotonic if the type expected by its context is the lattice of the circular value. Thus if we declare a function as taking a value of a lattice and returning one of another, the function is presumed monotonic; it is an (unchecked) error if it is not. For example, if we defined a negation function that took values of `MinIntegerLattice` and returned values of `MaxIntegerLattice`, we could form a monotonic dependency between `a` and `b`:

```
function negate_min(x : MinIntegerLattice) : MaxIntegerLattice := -x;

a := negate_min(b);      -- monotonic
```

Notice that an approximation to `b` is greater than what the final value will be, but an approximation of `-b` is less than the final value of `-b`. That reasoning shows that the declaration of `negate_min` is meaningful. The APS compiler does not attempt to perform such a justification; such declarations are simply trusted.

In addition, certain common situations involving the builtin keywords `and`, `or`, `not` and `in` are known by the APS compiler to be monotonic. Both `and` and `or` can be used monotonically in contexts expecting either an `AndLattice` or an `OrLattice` value. The context is passed on to both parameters. In an `OrLattice` context, an `AndLattice` argument of a `not` expression can be used monotonically, and the same is true with the two lattices switched. In an `OrLattice` context, the set argument in an `in` expression can be used monotonically if it is of some `UNION_LATTICE` type, and similarly for an `AndLattice` context with a `INTERSECTION_LATTICE` set type.

A lattice comparison function is available using the `|<=|` syntax. It is declared as returning merely a `Boolean`, but in an `OrLattice` context, the second argument can be used monotonically, and for an `AndLattice` context, the first argument can be used monotonically. The APS compiler also knows about `|<|`, `|>|` and `|>=|`. It may seem that these rules are complicated, but they simply embody the common sense rule that a value can be used monotonically, that is, an approximate value can be used, only if the approximate value does not cause an effect not caused by the final value.

7.2.5 Wrappers

Dependencies can arise not only though assignments, but also through guards. For example, in the fragment

```
if x then
  y := 0;
endif;
```

the value `y` depends on `x`. When both are circular values, the issue arises as to whether the dependency is monotonic or not. In this case, if `x` is of the `OrLattice` type, then the dependency is monotonic: once the approximation of `x` becomes true, we know its final value must be true as well.

More generally, for the purposes of assignments in the 'then' part of an 'if' clause, the guarding condition is evaluated as an `OrLattice` boolean, but for the purposes of assignments in the the 'else' part, the guarding condition is evaluated as an `AndLattice` boolean. The common sense rule is that the guard is not used until it reaches its final value; the guarded statements are not activated unless the system knows that the guard is guaranteed to stay the same.

For example, suppose we have the following structure:

```
if c_opt and c_pess then
  ...
else
  ...
endif;
```

where `c_opt` (optimistic) is a circular value of type `AndLattice` (whose approximations start true and become false if assigned), and `c_pess` (pessimistic) is a circular value of type `OrLattice` (whose approximations start false and become true if assigned). Both `and` and `or` are monotonic for both boolean lattices. For the purposes of the 'then' part, `c_pess` is used monotonically, but for `c_opt`, we require the final value. Thus any circular value assigned in the 'then' part depends non-monotonically upon `c_opt`. Similarly any circular value assigned in the 'else' part depends non-monotonically on `c_pess`.

In essence, the condition is implemented in two different ways: once for each branch of the 'if.' The APS compiler ensures, however, than any procedure calls in the condition are evaluated only once.

A similar, albeit simpler, situation applies to `for-in` structures:

```
for x in v begin
⋮
end;
```

The expression v for the set can be used monotonically if it is of a `UNION_LATTICE` type. These lattices have the feature that once an approximation includes a certain element, we know for certain that the final value will also include that element, and so it is safe to evaluate the guarded equations with `x` bound to this element.

Figure 7.1 contains several examples of the cases discussed here. The fragment comes from the APS type checker, which uses type variables for type inference. Each type variable gets a set of types it is bound to (`bindings`) and a set of other type variables that have been equated through the type inference process (`chain`). For each variable, we go through the set of equated type variables (ignoring the variable itself) and then force each type variable's `bindings` and `chain` attributes to include the the respective attributes of the other type variable. Here `ty1.chain` depends directly on itself, but the dependency is monotonic, since the use in the `for-in` wrapper is monotonic; the set of equated type variables (`ty1.chain`) can never lose elements.

However, the rest of the fragment involves defining the normal (non-circular) `binding` attribute and the `type_errors` attribute of "`owner`" whose declaration (not shown here) states it to be a normal collection attribute. Since these attributes are not circular,

```
attribute TypeVariable.binding : ContextualType := var_unbound;
var type TypeVariablesLattice :=
    UNION_LATTICE[remote TypeVariable,TypeVariableSet];
type BindingsLattice := UNION_LATTICE[PartialType,BindingSet];
circular collection attribute TypeVariable.chain : TypeVariablesLattice;
circular collection attribute TypeVariable.bindings : BindingsLattice;
circular collection attribute TypeVariable.consistent : AndLattice;
⋮
match ?ty1=type_variable(?decl,?owner) begin  --N.B. simplified
  ty1.chain :> {ty1};
  -- force closure:
  for ty2 in ty1.chain begin
    if ty1 /= ty2 then
      ty2.bindings :> ty1.bindings;
      ty2.chain :> ty1.chain;
      ty1.bindings :> ty2.bindings;
      ty1.chain :> ty2.chain;
    endif;
  end;
  ⋮
  -- set the binding or flag as inconsistent
  if ty1.consistent then
    case ty1.bindings begin
      match {?ty,...} begin
        ty1.binding := fix_partial_type(ty);
      end;
    else
      owner.type_errors :> {"type variable unbound"};
    end;
  else
    owner.type_errors :> {"type variable has inconsistent binding"};
  endif;
```

Figure 7.1: Example of embedded attribute definitions from Appendix C.3

the dependency between them and the circular values `ty1.consistent` and `ty1.bindings` are simple. However, even if they were circular, the dependencies for the ones defined in the 'then' part guarded by the condition `ty1.consistent` would still be simple since the guard is of the wrong lattice to be monotonic.

7.2.6 Procedures

The distinction between functions and procedures becomes more pronounced in the presence of circular attributes. A function represents a pure value abstraction, but an internal procedure, a procedure declared local to the module being implemented, represents an abstraction over the attribution clauses of that module. The formal parameters of the procedure represent by name the actuals and so may represent circular values. Uses of formal parameters may be monotonic. If the result of a procedure may be used in a legal cyclical dependency, it should be declared `circular` and typed with a `LATTICE` type.

External procedures including constructors for a phylum are different since they are assumed to have an unanalyzable effect. Thus an external procedure is only called after its parameters have their final values.

7.3 Implementation

As mentioned earlier, the current APS implementation does not perform static scheduling; instead it uses demand evaluation. But demand evaluation works correctly only for (simple) dependencies between normal instances. When a circular instance is demanded *simply* (that is, in a non-monotonic context), it cannot return an approximate value; it may only return the final (fixpoint) value. For this reason, if a circular instance is not yet fully evaluated, a simple demand dependency is handled as an exception that unrolls all the currently demanded instances. Later, in a separate phase, each circular instance is demanded monotonically. When a circular instance is demanded monotonically, the next approximate value is computed, but each circular instance is recomputed only once per sweep through the instances. There are a number of complications to this basic model; this section describes the implementation method more fully.

7.3.1 Simple versus Monotonic Demand Evaluation

The runtime system goes through two phases when evaluating instances; these phases are repeated as necessary until all instances are evaluated.

The first phase involves monotonically demanding every circular instance until all have reached fixpoints. During this phase, the runtime system repeatedly brings a subset of the still unfinished instances to a fixpoint. Eventually, all circular instances are computed. The set of instances currently being brought to a fixpoint is called the *active set*. Each repetition starts with all remaining unfinished circular instances in the active set. If it turns out that one circular instance has a (possibly indirect) simple dependency on another, the first is removed from the active set.

During the second phase, instances are demanded simply. If any new circular instance (for example, a circular local variable) is created during this phase, it must wait until the next repetition of the first phase before it can return a value. Demanding the value of such a circular instance or any other unfinished circular instance terminates all pending evaluation attempts. These normal instances will not be evaluated until the next repetition of the second phase.

7.3.2 Guards

As mentioned in Section 6.3, a collection attribute instance depends on the imprecise guard for the attribute. The guard ensures that the node for each definition is known. As a result, all the nodes whose collection attribute is being defined must be known before any of the collection attribute's instances may be evaluated. If this feature were carried over to circular collection attributes, then if the node being defined a collection attribute instance depended (indirectly) on an instance of the same collection attribute, then all instances would depend indirectly on each other. As long as the dependencies were monotonic, the instances could still be evaluated, but a single simple dependency between two instances would result is an illegal cycle. In Farnum's DORA system, the equivalent to circular collection attributes has this property; all instances must be evaluated together.

However, simple dependencies are necessary in certain situations, such as in type-inference. For example, one usually wishes to infer the type of a function by itself before using the inferred type for the uses of the function. Such a situation is present in the type checker for APS given in Appendix C.3. Proper implementation of such systems requires some analysis of the possibly affected attribute instances so that the dependencies can be properly limited. Further research is needed on such analysis; in the meantime, the current APS compiler can implement a module under the assumption that all dependency paths through imprecise guards are monotonic. In this case, the guards are always kept on the active set, but are not brought to a fixpoint until every instance they depend on is completely evaluated. The programmer may have to add additional dependencies to ensure a safe evaluation order. The runtime system detects when such an unsafe evaluation order leads to erroneous execution, and terminates the evaluation with a diagnostic.

7.3.3 Simple Dependencies

During monotonic demand evaluation, a monotonic dependency chain of circular instances is maintained (the *pending* set). Then whenever a normal (non-circular) instance is demanded, monotonic evaluation is turned off while it is being computed. If during such an evaluation, an unevaluated circular instance is demanded, all the pending circular instances are marked as having a simple dependency on this latest circular instance and are removed from the active set. Then evaluation of all these instances is terminated and monotonic evaluation continues with the next instance on the active set. Similarly, if a circular instance already marked as having a simple dependency is demanded (either simply or monotonically), all pending instances are marked as before and removed from the active set. In this way the active set is reduced to those circular instances that do not have simple dependencies on unready circular instances.

7.3.4 Finding a Fixpoint

The runtime system repeatedly monotonically demands every instance on the active set. At each iteration, it resets a *changed* flag. Whenever a circular instance gets a new successive approximation, the changed flag is set. If the flag is still clear when the iteration is over, all instances in the active set have reached a fixpoint. At this time, any imprecise guards in the active set are removed, and the rest are marked as finished. Then each instance marked as having a simple dependency is checked to see if the instance on which it depends is finished, and if so the simple dependency marking is removed. Any remaining unfinished circular instances without (known) simple dependencies are placed on the active set and evaluation continues until either all circular instances are finished, or else a cycle is found. In the latter case, an error is reported and evaluation is aborted.

7.4 Summary

Circular attribution is needed in a declarative compiler formalism for expressing many analyses necessary for optimization. Circular collection attributes allow new dependencies to be added monotonically during evaluation. Moreover, they can be used to implement unification without side-effects. APS supports these sophisticated attribution flows.

Chapter 8

Modularity

This chapter describe previous proposals for integrating modules into attribute grammars. Then it describes the module system of APS, and how modules are used to split a language description into smaller coherent parts.

8.1 Other Treatments of Modules in Attribute Grammars

Traditionally attribute grammars are monolithic. One early example is the 24,000 line Ada front end written in GAG/ALADIN [95]. Apart from the factoring provided by the context-free grammar, that description was written as a single entity. At least in this case, the entire attribute grammar was occupied with the closely linked tasks of name resolution and type-checking. However in other cases, such as the complete compiler for Pascal developed by Farrow [32], attribute definitions concerned with code generation are intermingled with ones performing name-resolution.

A number of methods for adding modularity to attribute grammars have been proposed; Watt gives a summary [101]. Some of the proposals for so-called module attribute grammars however, are really more appropriate for smaller levels of granularity than that of program modules. For example Baum's modular attribute grammars [8] allow an attribute grammar to be factored into pieces, each piece of which includes one or more productions from the grammar and all the associated definitions for it. This method provides no support for factoring beyond that in a classical attribute grammar, neither does it provide for separate compilation of modules. Similarly, Dueck and Cormack's modular attribute grammars [26] provide a limited form of untyped pattern matching suitable for specifying attributes over concrete parse grammars, but each module defines a single attribute and cannot be separately implemented. Moreover, their paper suggests that their concept of modularity had been tested only for systems requiring at most a hundred or so attribute definitions.

Larger-scale reuse and conceptual factoring can be accomplished by producing abstractions of the program being analyzed. For example, one can accomplish the task of name resolution over an abstract grammar that consists merely of scope boundaries, variable definitions and uses, and then map the solution to the actual structure used in the program. Kasten's and Waite's modularity mechanism [59] used in the LIGA system [57] supports this idea by allowing a production to inherit clauses from another production. With Farrow, Marlowe and Yellin's composable attribute grammars [36], one creates the abstract structure directly using normal attribute definitions. Composable attribute grammars allow more flexibility in the application abstraction than in LIGA, but the constraint of "separability" (needed for implementation) means that the evaluation of attributes in different modules cannot be intermingled. This restriction limits the kinds of dependencies that can be expressed.

The module system of Olga in Jourdan et al's FNC2 [52] is most like that of a general-purpose programming language's module system (it also resembles that provided in APS). In Olga, attribute grammar modules may import an attributed tree and then either define more attributes for the tree or else produce a new tree to be attributed elsewhere. Olga modules can be separately compiled.

8.2 Modules in APS

Modules provide medium-scale structuring in APS. The appendices give a number of module definitions. Modules export services and may be instantiated, extended or inherited. This section describes modules and how they are used.

8.2.1 Module Declarations

Modules are defined using the keyword `module`. Modules may be parameterized by types or values: type parameters are given in brackets `[...]`; value parameters in parentheses `(...)`. For example the `ALGOL_SCOPE` module in Appendix B.3.1 takes a type parameter named `Decl` and a value parameter named `decl_name`. When a module is instantiated, the result is either a type (by default), or a phylum (indicated, as in this example, by the keyword `phylum`). Section 8.2.2 describes instantiation in greater detail.

Entities in a module may be declared *public* or *private*. By default, an entity is public, but the declaration "`private;`" changes the default to private for the following declarations. For example, in the module for checking Oberon2 programs, only a few entities are public (the full text is given in Appendix B.5):

```
module OBERON2_CHECK[...] ...
begin
  signature ERROR_NODES :=
      {Declaration,Header,Receiver,Type,Statement,Case,
       CaseLabel,Expression,Element,Use}, var PHYLUM[];
  type Errors := BAG[String];
  [phylum Node::ERROR_NODES] begin
    collection attribute Node.errors : Errors;

    -- an internal way to report an error
    private procedure add_error(node : remote Node; message : String) begin
      node.errors :> {message};
    end;
```

```
  end;

  private;
  ⋮
  collection attribute Declaration.decl_used : Boolean :> false, (or);
  ⋮
  attribute Use.use_variable : Boolean := false;
  ⋮
end;
```

The module contains many entities including several attribute declarations (just three are shown here), but only one attribute, `errors`, is public. This attribute is defined over phyla satisfying the signature `ERROR_NODES` and has type `Errors`. All the other named entities are relevant only to the implementation of the module; they neither conflict with other entities of the same name in other modules, nor are directly accessible outside the module.

In order to permit separate compilation, *services*—that is, public entities of a module—are declared *var*, or (by default) *constant*. The services declared as var and all attributes are permitted to depend on attribution clauses in the module. They may also depend upon the results of the uses of *input* services: procedures, constructors for phyla, and attributes or variables declared assignable by the client using the keyword `input`. For example, the `ALGOL_SCOPE` module has an input attribute named `local_decls`, a var function named `find_local_decl`, and a constant type named `Decls`.

Since a client of a module cannot and should not know which var services depend on which input services, any use of an input service must be scheduled prior to any use of a var service. Constant services may be used at any time. At runtime, each instance of a module undergoes a transition called *finalization*. Before this point, input services may be used; after this point, var services may be used. During finalization, the clauses in the module are activated and all variables have their final value computed. The APS compiler must schedule module instance finalizations.

8.2.2 Instantiation

Modules must be instantiated with type and value parameters (if any) to be used. The result of such instantiation is either a type or a phylum. Chapter 4 discussed the `SEQUENCE` module that constructs sequence phyla as well as the modules `BAG`, `LIST`, `SET`, and `ORDERED_SET`. Previous examples have show many instantiations such as

```
  type Decls := BAG[remote Decl];
```

Module instantiations must be named, as seen here.

User-defined modules are instantiated in the same way. For example, the module `ALGOL_SCOPE` is instantiated twice (as it happens) in the Oberon2 symbol table module in Appendix B.3.2. The first instantiation creates the `Contour` phylum:

```
  phylum Contour := ALGOL_SCOPE[Declaration](decl_name);
```

A service provided by the module may be accessed through the type using the notation *type$service*. This notation is borrowed from Russell [9] and CLU [68]. For example, the `root_contour` constructor from `ALGOL_SCOPE` is fetched from `Contour`:

```
root_contour = Contour$root_contour;
pattern root_contour = Contour$root_contour;
⋮
```

Here, `root_contour` is defined as a local alias or *renaming* of the one available from `Contour`. A renaming is distinguished from a declaration by the use of `=`. Since constructors live in both value and pattern name spaces, two renamings are used. A renaming gives the name space; by default the value name space, other name spaces are given by one of the keywords `signature`, `type`, or `pattern`.

Every instantiation of a module yields a different type. This rule known as the "generative type constructor" rule, similar to the better known "name equivalence" rule, means that values of structurally identical types are not type compatible. For example, later in `APS_SYMTAB`, we have another instantiation of the `ALGOL_SCOPE` module:

```
phylum RecordContour := ALGOL_SCOPE[Declaration](decl_name);
```

A value of type `RecordContour` may not be used where a value of type `Contour` is expected and vice versa. Similarly, the two phyla `Formals` and `Fields` are incomparable despite both being declared as `SEQUENCE[Declaration]` (see Appendix B.1). Bounded polymorphism makes the generative type constructor rule less onerous than it would be otherwise.

8.2.3 Extension

One module may *extend* another module's services. More precisely, a module may extend the services provided by an already constructed type (such as that passed as a type parameter). The resulting type is the same as the original type for the purpose of type-equivalence, but has additional services. Inside the scope of the extending module, the original services of the type may be used without `$` qualification. The extension may only use the public entities of the extended type. (This idea of extension was developed from Maddox's work [69].)

Extension is very convenient and is used throughout the Oberon2 compiler. Most modules are parameterized by a type parameter representing the tree and subject to various "signatures". Each then extends the type received in the parameter. For example, the module for checking Oberon2 (from Appendix B.5) has the following header:

```
-- Compile-time checks for Oberon2:
-- we use the information provided by the resolution phase
-- and by compile-time computations and check that there are no errors
module OBERON2_CHECK[T :: var OBERON2_TREE[],
                          var OBERON2_RESOLVE[T],
                          var OBERON2_COMPILE_COMPUTE[T]] extends T
```

The type parameter must have the services provided by OBERON2_TREE, OBERON2_RESOLVE, and OBERON2_COMPILE_COMPUTE. The keyword var here means that services marked var must be available, and in particular attributes have their final values. Each of the restrictions, and the concatenation of the restrictions is a *type signature* (as used by Donahue and Demers for Russell [25]) known simply in APS as a *signature*.

The result of this module invocation is the same as the type parameter, but has the additional services provided by OBERON2_CHECK (basically just the attribute errors).

8.2.4 Inheritance

An extending module cannot modify the implementation of the services already provided by a type. Inheritance, on the other hand, does allow such modifications. It also allows the implementation of systems of *non-separable* (mutually dependent) modules, because the clauses of the inherited module are scheduled with the clauses of the inheriting module.

When a module is inherited, the effect is that its entire body is copied into the body of the inheriting module. This model of inheritance (pure *implementation inheritance*) is derived from that of Sather [93]. All entities, public or private, are available to the renamings and "replacements" (see below) in the body of the inherit clause; any renamed entities are treated as entities of the inheriting module.

In the Oberon2 compiler, the tasks of name resolution and computing the types of expressions must be performed together because in order to resolve a field identifier in a record selection, it is necessary to know the type of the record expression, but in order to know the type of a variable use, it is necessary to first resolve its name. However, for modularity, name resolution and expression type computation are described in separate modules. In order to implement the tasks, they are inherited together into a single module. In particular, this combined module OBERON2_RESOLVE module inherits from the OBERON2_SYMTAB module:

```
inherit OBERON2_EXPR_TYPE[T](use_decl) begin
  var base_type = base_type;
  var expr_type = expr_type;
  var expr_header = expr_header;
  var implicitly_guarded = implicitly_guarded;
end;
```

This example has four value renamings, each marked var because they are public var services of the OBERON2_RESOLVE module.

Outside the body of the inherit clause, the inherited entities are only known if they were renamed in the body of the inherit clause. The renaming requirement is intentional in the design of APS. It ensures that inheriting a module can never result in name clashes. It also allows a person reading the description to know what a name refers to.

The inheritance clause may include *replacements* as well as renamings. Replacements act to consistently modify the contents of the inherited module. Each replacement gives the name of a service used in the inherited module and a name of a service accessible in the inheriting module with which to replace the uses. By simultaneously renaming an inherited service and replacing its uses, one can use the old definition of a function to assist in writing the new definition. The following fragment comes from a module in the APS compiler, that describes a null transformation of an abstract APS tree annotated with binding information. (see Appendix C.4.1). This module uses inheritance and is likewise inherited by other modules performing transformations.

```
inherit COPY_ABSTRACT_APS[Input,CopyRecords] begin
  ⋮
  copy_Use -> copy_Use;
  var inherited_copy_Use = copy_Use;
end;
```

The `COPY_ABSTRACT_APS` module is an automatically generated module which simply copies an APS program in its abstract tree form. The inheriting module does a copy with an additional change: name resolution information is copied to the new tree. In order to avoid duplication of effort, the `COPY_ABSTRACT_APS` module is inherited. However, in order to add the name resolution information to the new tree, this module needs to access `Use` nodes are they are created. Thus it replaces the inherited `copy_Use` procedure with its own version (declared earlier in the file). However, most of the work is identical, and so it also inherits the original version under a different name so it can be used in the new definition of `copy_Use`.

8.3 Implementation

Modules are conceptually functors that accept types and/or values and compute types. In the implementation, they are functions. The services provided by a type are stored in a dictionary, indexed by name space and by name.

When a module is executed, it evaluates constant variables and types. Then it arranges for the constant and input services to be available: constructors, procedures, and input attributes. At this point the resulting type is said to be *initialized*. At some point, the module will be requested to *finalize* its instance. First all attribution clauses are activated and the demand evaluation worklists are initialized. Then demand evaluation is used to bring all variables to their final values. Finally, the input services are removed from the dictionary and the var services are added.

Demand evaluation is used in the calling module to ensure that a module instance is not finalized until after all uses of input services are accomplished, and that no var service is used until after finalization. This situation is accomplished by noting not only which variables are potentially affected by an attribution clause or procedure, but also which *types* are *modified*: that is, for which types input services are used. When it is necessary to postpone the evaluation of an attribution clause or procedure due to demand evaluation, it is placed on the guard thunk list of a guard variable for a type. The type itself is made to depend only on types and values needed to initialize the type, leaving finalization to its guard variable.

8.4 Summary

Modularity is an important property of a compiler description, in keeping with its importance for all other programs. The description language may either encourage or hinder modular design. APS provides modules which have public and private services. These modules may use other modules, extend types or inherit other modules. These methods are used extensively in the Oberon2 and APS compilers described in the appendices to create modular descriptions.

Chapter 9

Descriptional Composition

Descriptional composition combines two stages of a compiler description, the first of which produces a tree attributed by the second, and produces a new stage (called the *composed* stage) that does the work of both. The term *descriptional composition* contrasts with *functional composition*. In functional composition, the two stages are compiled (possibly separately) and then run successively.

Descriptional composition is similar to performing inline substitution of procedures. The difference stems from the fact that the first stage does not directly invoke the the second stage. Instead it merely produces an intermediate structure to be attributed by the second stage. The equivalent advantages and disadvantages of inline substitution apply to descriptional composition. With inline substitution, one avoids the procedure call overhead; with descriptional composition, the intermediate tree need no longer be produced. An inline substitution may allow optimizations such as constant folding; descriptional composition allows similar optimizations to be performed. A disadvantage of inline substitution is that the program will usually grow in size; similarly, a composed stage is likely to be more complicated than either of its parts. As with inline substitution, descriptional composition is most useful when at least one of the stages involved involves work comparable with the work of creating the intermediate form.

Section 9.1 describes previous work in descriptional composition. Then Section 9.2 describes the process as implemented in the current APS compiler. Section 9.3 describes the results of descriptionally composing two stages of the Oberon2 compiler.

9.1 Previous Work

The term descriptional composition was introduced by Ganzinger and Giegerich, who give an algorithm for composing attribute-coupled grammars (ACGs) [39]. Giegerich has described several closure properties of the composition operation [41]. Descriptional composition of two translations is possible when the first translation obeys a certain statically verifiable property. Boyland and Graham show how this restriction can be relaxed in certain cases [14]. In essence, both the original restriction and the later one require that the first translation build a tree; each node must have a unique parent in the tree. Every syntactic attribute must be used exactly once in an attribution rule (Ganzinger and

Giegerich) or at most once for each instance of the rule in which values are chosen for all of its conditionals (Boyland and Graham). Without this restriction, inherited attributes in the descriptionally composed module are multiply defined. Farrow, Marlowe and Yellin [36] remarked that descriptional composition was performed by hand in order to implement some "non-separable" (that is, mutually dependent) modules. This process could have been automated assuming the modules in question satisfied the equivalent restriction for CAGs.

Interestingly, all previous instances of automatic descriptional composition have concerned "toy" examples. In fact, the MARVIN project [40], which set out to implement large-scale attribute-coupled grammars, was apparently abandoned before descriptional composition could be implemented. This lack of realistic examples raises the question of whether the method scales well to larger and more complex cases. This chapter answers this question by showing that descriptional composition can indeed scale well.

9.2 Descriptional Composition in APS

Since the APS compiler cannot predict whether descriptional composition will yield a benefit or not, its use is controlled by the description writer with the pragma `compose` applied to intermediate structure types. Moreover, all the relevant modules must be expanded into a single module that must not export any phyla to be composed, nor any operations on instances of the phyla. In this manner, the compiler has access to the entire body of code that creates and traverses instances of the phyla.

For example, as seen in Appendix B, the Oberon2 compiler has two translations: from Oberon2 abstract syntax to the GCC tree representation and from the tree representation to C (represented as text). The two translations are composed in the module `OBERON2_COMPOSE` given in Appendix B.8. This short module simply hooks together the modules taking part in descriptional composition and provides `pragma`'s for each intermediate phylum to be removed. All the phyla of `GCC_TREE` are marked for removal except `TypePhylum`, as its instances are traversed recursively in a way that inhibits composition. (The reasons are given in Section 9.2.2.)

Descriptional composition removes phyla. For that purpose, it is necessary to regularize and simplify the creating and reading of instances of these phyla. Several canonicalizations are applied to uses of the phyla. Next, analysis of the result is performed to determine what attributes are read or written for which variables carrying instances of the phyla being removed. Finally, descriptional composition per se can be carried out. Since descriptional composition enables a number of simplifications (such as using known conditional values), these simplifications are then performed. The rest of this section describes the process in detail.

9.2.1 Canonicalizations

Constructor Calls The logically first step is to isolate all calls to constructors of the phyla to be removed. Each such constructor call in an expression is converted into the form

$$v_0 \ \texttt{:} \ \mathit{type} \ \texttt{:=} \ \mathit{constructor}\texttt{(}v_1\texttt{,}\ldots\texttt{,}v_n\texttt{);}$$

where each v_i is a (new) local variable. Procedure calls that return instances of the phyla to be removed are treated similarly. For example, the translation for set elements in Oberon2 consists of generating a logical shift:

```
match ?e=single_element(?v) begin
  e.gcc_element :=
      GccTree$lshift(GccTree$make_integer_cst(1,gcc_set_type),
                     v.gcc_expr);
end;
```

This call is rewritten as

```
match ?e=single_element(?v) begin
  arg1 : GccTree$Expression := GccTree$make_integer_cst(1,gcc_set_type);
  arg2 : GccTree$Expression := v.gcc_expr;
  call : GccTree$Expression := GccTree$lshift(arg1,arg2);
  e.gcc_element := call;
end;
```

Pattern Matching Next, pattern matching on instances of the phyla to be removed is converted to attribution. This transformation is described in detail in Section 3.4.4.

Lexical Ordering Expressions of the form e_1 `<<` e_2 are converted into comparisons of a new attribute e_1`.pos < `e_2`.pos` together with equations for the new attribute.[1] These equations are tedious to write, but easy to generate automatically. That is the purpose, indeed, for having lexical position comparison as a built-in operator. This transformation can be avoided altogether if a source-level optimization can remove the relation. The only use of the `<<` relation in the module to convert GCC trees to C text can be removed in this way.

9.2.2 Analysis

Named values (including formal parameters, local and global variables, and attributes of nodes) that may carry references to nodes of the phyla to be removed are marked as *carrying* values. Carrying may be direct (the value is a node reference) or indirect (the value is a collection of node references). For each such value, the analysis stage determines which attributes are read or written for the node references carried. For example, in the following fragment, the `text` attribute is fetched from `e.expr_type`:

```
match ?e=integer_cst(?v) begin
  e.text := "(" || e.expr_type.text || ")" || v;
  e.no_effect := true;
end;
```

[1] The new attribute would not incrementalize well, and so if the composed module is to be implemented as an incremental compiler, it is desirable to convert the comparisons of the attribute into `<<` expressions of the input to the first stage or more locally defined attributes.

Moreover, for each type constructed using the phyla being removed, analysis is used to determine which attributes are fetched for values of these types. For example, a list of dimension expressions is collected for each array expression:

```
-- array expressions carry along their dimensions separately:
type GccRemoteExpressionList := LIST[remote GccTree$Expression];
```

As it happens, the attribute `text` is read from instances passed in such lists.

The analysis currently works using a simple flow-analysis of the carrying values. The attribute reads and writes are propagated back to the place where the node is created. This method can break down in the presence of fetched attributes that themselves carry node references. Descriptional composition gets rid of carrying values by composing them with the attributes fetched from references to nodes carried. If a carrying value is composed with such an attribute, the result is another carrying value that must itself be composed. Thus if an attribute carries a node that indirectly carries another node with the original attribute, the process of descriptional composition may never terminate. In the case of the Oberon2 compiler, `TypePhylum` of the GCC tree module is used in such a way that it possesses many such attributes (if pattern matching were to be removed), because, for example, for every pointer type, there must be an attribute of its base type. As a result, this phylum is excluded from descriptional composition.

9.2.3 Composition

At this point, the process of descriptional composition per se can start. First the top-level match rules for the phyla are substituted where the appropriate nodes are created. Next, new attributes and constructed types are generated. Attribute reads and writes of the node to be removed are converted to reads and writes of the new attributes. At the same time, new copy rules are introduced. Finally, all rules creating and passing node references are removed. Descriptional composition is complete.

Substitution At every point an instance of a phylum to remove is created, all the appropriate top-level clauses that apply to the constructor being called are substituted. For example, an Oberon2 call statement is translated to a GCC `do` statement:

```
match ?s=call_stmt(?call) begin
  s.gcc_stmt := GccTree$do(call.gcc_expr);
end;
```

This fragment is first canonicalized as

```
match ?s=call_stmt(?call) begin
  arg1 : GccTree$Expression := call.gcc_expr;
  Xdo : GccTree$Statement := GccTree$do(arg1);
  s.gcc_stmt := Xdo;
end;
```

The translation to C text contains the following top-level match for `do`:

```
match ?s=do(?expr) begin
  s.text := block_saved(s.indent,expr.saved_decl_texts,
                        expr.text || ";\n");
end;
```

Moreover, the `indent` attribute for statements has a default that uses the node for which it is defined:

```
[T :: INDENTING] attribute (node:T).indent : String :=
    make_indent(node.depth);
```

Thus substitution yields

```
match ?s=call_stmt(?call) begin
  arg1 : GccTree$Expression := call.gcc_expr;
  Xdo : GccTree$Statement := GccTree$do(arg1);
  Xdo.text := block_saved(Xdo.indent,arg1.saved_decl_texts,
                          arg1.text || ";\n");
  Xdo.indent := make_indent(Xdo.depth);
  s.gcc_stmt := Xdo;
end;
```

Generation For every variable carrying instances of the phyla being removed, new variables are generated, one for each attribute read or written as determined by the analysis. The types of these new variables may be newly created. Then, attribute reads and writes are redirected to use these new variables. Copy rules are introduced in parallel with copy rules that carry variables; synthesized attributes in the direction of the copy and inherited attributes in the reverse direction. For example, the preceding expanded example is further elaborated

```
match ?s=call_stmt(?call) begin
  arg1 : GccTree$Expression := call.gcc_expr;
  arg1@saved_decl_texts : SavedDeclTexts :=
      call.gcc_expr@saved_decl_texts;
  arg1@text : String := call.gcc_expr@text;
  Xdo : GccTree$Statement := GccTree$do(arg1);
  Xdo@text : String :=
      block_saved(Xdo@indent,arg1@saved_decl_texts,
                  arg1@text || ";\n");
  Xdo@indent : String := make_indent(Xdo@depth);
  s.gcc_stmt := Xdo;
  Xdo@depth : Integer := s.gcc_stmt@depth;
  s.gcc_stmt@text := Xdo@text;
end;
```

Here `@` is used as an alphabetic character; the variable `arg1@text` replaces the attribute fetch `arg1.text`.

Removal of Phyla Now that instances of the phyla to be removed are used neither in pattern matching nor in attribution, the phyla can finally be removed, together with any variable carrying instances of the phyla and all their attribution rules. In the running example, this transformation produces

```
match ?s=call_stmt(?call) begin
  arg1@saved_decl_texts : SavedDeclTexts :=
      call.gcc_expr@saved_decl_texts;
  arg1@text : String := call.gcc_expr@text;
  Xdo@text : String :=
      block_saved(Xdo@indent,arg1@saved_decl_texts,
                  arg1@text || ";\n");
  Xdo@indent : String := make_indent(Xdo@depth);
  Xdo@depth : Integer := s.gcc_stmt@depth;
  s.gcc_stmt@text := Xdo@text;
end;
```

9.2.4 Simplification

After descriptional composition has been applied, the description has a great many local variables that have their default values. If the variable is used at most once, or if its default value is trivial to recompute, its uses may be replaced by its default value with some gain in performance. For example in the previous fragment, all of the local variables can be substituted and then removed:

```
match ?s=call_stmt(?call) begin
  s.gcc_stmt@text :=
      block_saved(make_indent(s.gcc_stmt@depth),
                  call.gcc_expr@saved_decl_texts,
                  call.gcc_expr@text || ";\n");
end;
```

We now have a component of a compiler that goes directly from an Oberon2 abstract syntax tree to C text without using an intermediate form. This compiler is generated automatically; only the original compiler modules must be maintained by hand.

9.3 Results

Table 9.1 shows the compiled size (in megabytes) and execution time (in seconds) of the Oberon2 compiler before, during and after descriptional composition. The compiled size includes the size of the description itself (which is kept with the compiled code) and does not include the size of the run-time system or basic library. The execution time also gives a sense of the memory used, because the time for garbage collecting (including global garbage collects) is included. All times given are averages over ten successive runs on a lightly loaded HP 715/80 workstation with 64 megabytes of RAM. The current APS compiler used to implement both the composed and non-composed versions is a prototype

Module	Separate		Combined		Composed	
	size	time	size	time	size	time
Front end	1.68	120	1.68	125	1.68	116
Oberon2 to GCC	1.38	58				
GCC Tree	0.44	24	2.29	162	2.12	88
GCC To C	0.63	81				
Total	4.12	283	3.96	287	3.80	204

Table 9.1: Measure of Descriptional Composition
(Compiled program sizes are in megabytes; execution times are in seconds.)

and does not generate optimal code; these performance numbers do not compare favorably with other Oberon2 compilers. The purpose of the performance numbers here is to permit an evaluation of the *relative* benefit of descriptional composition.

The compiler is run on a set of six Oberon2 modules comprising about 200 lines. They consist of the six modules given in Figures 9.1, 9.2, 9.3, 9.4, 9.5, and 9.7. When compiled together, errors messages are only generated for the `System` module (see Figure 9.8). These errors are due to the module not having an implementation in Oberon2, and can be safely ignored.

The time spent reading the input files is charged to the front end and the time spent writing the output files is charged to the GCC to C translation or the composed translation. The first pair of columns shows the sizes and times for the compiler executing modularly. Before descriptional composition can be applied, the modules taking part must be combined into a single module and canonicalized. Some of these transformations may lead to better, some to worse performance. At the same time, combining the modules permits dead code elimination and constant propagation. In order to separate these effects from descriptional composition, per se, the second pair of columns shows how the modular compiler would execute if the combined module was implemented without descriptional composition The final pair of columns shows the additional effect of descriptional composition on the combined module. In all cases, the bottom line combines performance numbers of the front end with the back end. Memory usage in one part of the program may involve more frequent garbage collections in other parts; therefore the execution time for the front end is given for each of the three cases.

It is encouraging that the canonicalizations did not cause much loss of performance in the combined version. In fact, the optimizations possible on the combined version make simple combination attractive. In particular, many of the GCC tree features were not used in the Oberon2 compiler and could be eliminated; these include the “complex” type along with operations for using it, and “union” types, as well as many expression nodes.

The descriptionally composed version runs much faster than either the functionally composed version or the combined version. In fact it runs about as fast as the GCC tree to text transformation alone. This experiment demonstrates the practicality of descriptional composition even in complex situations, when not all intermediate structures can be composed away. Since descriptional composition is a high-level (almost source-level) optimization, these benefits should carry over to a practical APS compiler.

```
MODULE Trees;
  IMPORT Texts, Oberon;

  TYPE
    Tree* = POINTER TO Node;
    Node* = RECORD
      name-: POINTER TO ARRAY OF CHAR;
      left, right : Tree
    END;

  VAR w: Texts.Writer;

  PROCEDURE (t:Tree) Insert* (name : ARRAY OF CHAR);
    VAR p, father : Tree;
  BEGIN p := t;
    REPEAT father := p;
      IF name = p.name^ THEN RETURN END;
      IF name < p.name^ THEN p := p.left ELSE p := p.right END
    UNTIL p = NIL;
    NEW(p); p.left := NIL; p.right := NIL;
    NEW(p.name,LEN(name)+1); COPY(name,p.name^);
    IF name < father.name^ THEN father.left := p ELSE father.right := p END
  END Insert;

  PROCEDURE (t : Tree) Search* (name : ARRAY OF CHAR) : Tree;
    VAR p: Tree;
  BEGIN p := t;
    WHILE (p # NIL) & (name # p.name^) DO
      IF name < p.name^ THEN p := p.left ELSE p := p.right END
    END;
    RETURN p;
  END Search;

  PROCEDURE (t : Tree) Write*;
  BEGIN
    IF t.left # NIL THEN t.left.Write END;
    Texts.WriteString(w,t.name^); Texts.WriteLn(w);
    Texts.Append(Oberon.Log,w);
    IF t.right # NIL THEN t.right.Write END
  END Write;

  PROCEDURE Init* (t : Tree);
  BEGIN NEW(t.name,1); t.name[0] := 0X; t.left := NIL; t.right := NIL
  END Init;

BEGIN Texts.OpenWriter(w)
END Trees.
```

Figure 9.1: `Trees` module from Oberon2 Report.

```
MODULE NewTrees;
  IMPORT Texts, Oberon, Trees;

  TYPE
    NewTree* = POINTER TO NewNode;
    NewNode* = RECORD (Trees.Node)
      message*: ARRAY 10 OF CHAR;
    END;

  VAR wr : Texts.Writer;

  PROCEDURE (t : NewTree) Write*;
  BEGIN
    Texts.WriteString(wr,t.message); Texts.WriteLn(wr);
    Texts.Append(Oberon.Log,wr);
    t.Write^()
  END Write;

  PROCEDURE Init* (t : NewTree; message : ARRAY OF CHAR);
  BEGIN
    Trees.Init(t);
    COPY(message,t.message);
  END Init;

BEGIN Texts.OpenWriter(wr)
END NewTrees.
```

Figure 9.2: `NewTrees` module using `Trees` module.

```
MODULE Test;
  IMPORT Trees;

  PROCEDURE Run*;
  VAR
    t : Trees.Tree;
    t1 : Trees.Tree;
  BEGIN
    NEW(t);
    Trees.Init(t);
    t.Insert("first");
    t.Insert("second");
    t.Insert("third");
    t.Insert("fourth");
    t1 := t.Search("second"); t1.Write
  END Run;

END Test.
```

Figure 9.3: Driver module for module `Trees` .

```
MODULE Test2;
  IMPORT NewTrees;

  PROCEDURE Run*;
  VAR
    t : NewTrees.NewTree;
  BEGIN
    NEW(t);
    NewTrees.Init(t,"Hello");
    t.Write
  END Run;

END Test2.
```

Figure 9.4: Driver module for module `NewTrees`.

```
MODULE Texts;
  IMPORT System;
  TYPE
    Buffer = POINTER TO ARRAY OF CHAR;
    Writer* = RECORD
      buf* : Buffer;
      fill, len : INTEGER;
    END;

  PROCEDURE EnsureLength* (VAR w : Writer; len : INTEGER);
    VAR newbuf : Buffer; newlen : INTEGER;
  BEGIN newlen := len+w.len+10;
    IF len < w.len THEN
      NEW(newbuf,newlen);
      COPY(w.buf^,newbuf^);
      w.buf := newbuf;
      w.len := newlen;
    END
  END EnsureLength;

  PROCEDURE WriteString* (VAR w : Writer; VAR s : ARRAY OF CHAR);
    VAR i : INTEGER; len : INTEGER;
  BEGIN len := System.StringLength(s);
    EnsureLength(w,w.fill+len);
    FOR i := 0 TO len-1 DO
      w.buf[w.fill] := s[i];
      INC(w.fill);
    END;
    w.buf[w.fill] := 0X;
  END WriteString;

  PROCEDURE WriteCharacter* (VAR w : Writer; ch : CHAR);
  BEGIN
    EnsureLength(w,w.fill+1);
    w.buf[w.fill] := ch;
    INC(w.fill);
    w.buf[w.fill] := 0X;
  END WriteCharacter;

  PROCEDURE WriteLn* (VAR w : Writer);
  BEGIN
    WriteCharacter(w,0AX);
  END WriteLn;

  PROCEDURE Append* (VAR f : System.File; VAR w : Writer);
  BEGIN
    System.Write(f,w.buf^);
    w.fill := 0;
    w.buf[w.fill] := 0X;
  END Append;

  PROCEDURE OpenWriter* (VAR w : Writer);
  BEGIN
    NEW(w.buf,10);
    w.fill := 0;
    w.len := 0;
    w.buf[w.fill] := 0X;
  END OpenWriter;
END Texts.
```

Figure 9.5: Simple stubs for standard Oberon2 module `Texts` .

```
MODULE Oberon;
  IMPORT System;
  VAR Log* : System.File;
BEGIN
  Log := System.Stdout;
END Oberon.
```

Figure 9.6: Simple stubs for standard Oberon2 module `Oberon2`.

```
MODULE System;
  TYPE File* = LONGINT;
  VAR Stdout* : File;

  PROCEDURE Write* (f : File; VAR a : ARRAY OF CHAR);
  END Write;

  PROCEDURE StringLength* (VAR a : ARRAY OF CHAR) : INTEGER;
  END StringLength;

END System.
```

Figure 9.7: Headers for some system functions.

```
system:5 [OBERON2_TREE'value_formal
          [OBERON2_TREE'identifier f [OBERON2_TREE'not_exported]] ..]:
  Warning: declaration not used
system:5 [OBERON2_TREE'var_formal
          [OBERON2_TREE'identifier a [OBERON2_TREE'not_exported]] ..]:
  Warning: declaration not used
system:9 [OBERON2_TREE'proc_decl
          [OBERON2_TREE'header
           [OBERON2_TREE'identifier StringLength #] ..]]:
  No RETURN statement for PROCEDURE
system:8 [OBERON2_TREE'var_formal
          [OBERON2_TREE'identifier a [OBERON2_TREE'not_exported]] ..]:
  Warning: declaration not used
```

Figure 9.8: Errors generated when compiling system module.

Chapter 10

Experiences

This chapter describes the status of the APS and Oberon2 compilers. It then details the author's experiences with APS as a compiler descriptional language, and with descriptional composition as an optimization technique. Next it outlines some areas for further work suggested by the results. At the end, this chapter gives the conclusion for the dissertation.

10.1 Status

The current APS compiler and run-time system are written in about 25,000 lines of Common Lisp (including comments and blank lines). The parser (mostly machine generated) is external to Lisp. It reads in APS programs, creates trees in the form given in Appendix C.1 and writes a fully parenthesized version to be read by Lisp. For historical reasons, the compiler is built on top of a rewrite of DORA [29], although it uses almost no facilities of that system except the pattern match compiler. An APS module is translated into Lisp and is compiled by the native Allegro Common Lisp (Franz Inc.) compiler. Sometimes, the sheer size of the generated functions can overwhelm the native compiler, which is optimized for hand-written code. However, this platform has been extremely convenient, especially as Common Lisp provides support for pretty-printing internal forms, and for complicated compile-time macro expansion.

Part of the APS compiler has been rewritten in 10,000 lines of APS itself (mainly the front end). Some of the modules can be seen in Appendix C. This exercise has been valuable for a number of reasons. First, the declarative nature of specifications ensured that the type system could be formally described. Second, since APS is a complex language, the experiment shows that APS is capable of describing the "static semantics" of a language considerably more complex than Oberon2. Third, several extensions were added to APS after their use was shown to improve factoring, readability or performance in the APS compiler.

The Oberon2 compiler consists of a parser (written using the same tools as the APS compiler's parser) and a front end (including translation to the GCC intermediate representation) written in APS. The part written in APS (about 6000 lines) is given in Appendix B. While the compiler has not been fully tested, it compiles a set of six modules using arrays, open arrays, bound procedures, inheritance and overriding. The generated C code is indented for partial readability and it compiles and runs successfully. For example, the following bound procedure comes from the Oberon2 manual [73]:

```
PROCEDURE (t : Tree) Write*;
BEGIN
  IF t.left # NIL THEN t.left.Write END;
  Texts.WriteString(w,t.name^); Texts.WriteLn(w);
  Texts.Append(Oberon.Log,w);
  IF t.right # NIL THEN t.right.Write END
END Write;
```

This procedure compiles into the following C function:

```
void Trees_Node_Write(struct G1992 * T)  {
  if (((*(T)).left)!=((struct G1992 *)((void *)0)))
    {
      struct G1992 * G1913 = (*(T)).left;
      (*((*((*(G1913))._type_spec)).Write_ref))(G1913);
    }
  else
    {}
  {
    G1941 G1878 = (*(T)).name;
    unsigned long G1879 = (G1878).dim1;
    (Texts_WriteString)(&(Trees_w),(G1944){(G1878).dopevector,G1879});
  }
  (Texts_WriteLn)(&(Trees_w));
  (Texts_Append)(&(Oberon_Log),&(Trees_w));
  if (((*(T)).right)!=((struct G1992 *)((void *)0)))
    {
      struct G1992 * G1914 = (*(T)).right;
      (*((*((*(G1914))._type_spec)).Write_ref))(G1914);
    }
  else
    {}
}
```

10.2 Experiences with APS

This section describes experiences in using the APS compiler description language.

10.2.1 Features

At the core of APS is a simple polymorphic functional language. Since experience with such languages is fairly wide-spread, this section concentrates on those features that set APS apart from this framework.

In common with other attribute-grammar systems, APS supports the ability to decorate trees with attributes. Attributes can be thought of as memoed functions taking a single value for which object identity is maintained. The ability to use syntax-directed definitions, especially patterns including nested patterns, makes this paradigm attractive, and even more so helps factor a description by concept.

Collection attributes are extremely useful. For instance, just to give an example that came up in the last stages of creating the Oberon2 compiler, in the process of converting the GCC intermediate form to C text, it is necessary to generate C "typedefs" for intermediate language types. These types are scattered throughout the tree, some attached to expressions, others to explicit type declarations. Since C is a declaration-before-use language, it is necessary to generate the "typedefs" in an order that respects this restriction. In an imperative language, this task would presumably be done by sorting the types into topological order, but it would be difficult to locate all the types. In APS, finding all the instances of a phylum is done using a top-level match. Then, each "typedef" is generated with a priority so that it will follow any other typedef that must precede it. All of these prioritized strings are then placed in a single *sorted* collection attribute. The resulting realization in APS is simple and clear. (See Appendix B.7.)

Pattern definitions (as described in Section 3.3.1), the ability to name views of nodes, including the ability to use non-deterministic "disjunctive" patterns, have proved enormously successful. For example, the simple pattern `array_type` is used extensively in the Oberon2 description to factor attribute rules dealing with arrays. Figure 3.1 in Chapter 3 showed some further ways in which pattern definitions were used to factor the description by concept. The Oberon2 compiler uses over 25 pattern definitions. Further work will be done in describing their contribution outside of the context of APS. Recursive pattern definitions have been less pervasive, although the APS compiler exhibits several useful instances of recursive pattern definitions.

Another innovation in APS of great importance is the *procedure*. Despite its syntax and name, an APS procedure is not an imperative function. No other attribute grammar system to my knowledge has the ability to factor not just expressions but *attribution rules themselves*. Appendix B.5 shows several instances of procedures such as the simple procedure `ensure_variable` or the more complex procedure `ensure_type_guard_applicable` that permit checks (and error messages) to be expressed in a single place. Recursive procedures in APS are essential to the ability to express the standard unification algorithm declaratively. Currently, procedures interact poorly with lexical prioritization of attribute rules. Perhaps this conflict will be resolved in a later version of APS.

Finally, despite the fact that APS has this feature in common with many other languages, it must be noted that polymorphism is essential for factoring. Moreover, the ability, as in APS, of defining entities relative to a limited set of types is particularly useful. They permit polymorphic attributes such as `scope` in Appendix B.3.2, and also allow the rule that scopes are inherited by default from the parent to be described in a simple, general and type-safe manner.

10.2.2 Possible Enhancements

Since the design of APS was frozen at a certain point, several interesting enhancements had to be set aside. In particular, APS lacks the ability to match the parent of a node under consideration; pattern matching only goes "down" the tree, never "up." Several context-dependent concepts therefore, such as checks that a local procedure value is only used as the object of a call, must be expressed using attributes. "Upward" patterns would simplify the expression of such concepts. However, they may also be unintuitive and confusing.

Higher-order functions are useful, and are provided in APS. They are used extensively in the `OBERON2_CONSTANT` module given in Appendix B.1.1. However APS does not permit *patterns* to be passed as parameters to either functions or modules. The lack of "first-class" status for patterns makes factoring harder in some situations where a function applies to several different nodes in an analogous manner. Without "first-class" patterns, dangerous redundancy can creep in.

Currently, patterns are implemented using success continuations, but the power is not used. For instance, pattern matching is often used to take some value out of a structure, and then a function is applied to it and pattern matching used on the result. For example:

```
match ?ty1=named_type(?using) begin
  case using.use_decl begin
    match type_decl(?,?ty2) begin
      ty1.base_type := ty2.base_type;
    end;
  end;
end;
```

If one could define a pattern `use_decl` that could match on the attributes of a node, one could express this rule as follows:

```
match ?ty1=named_type(use_decl(type_decl(?,?ty2))) begin
  ty1.base_type := ty2.base_type;
end;
```

If patterns could be generalized to include the power of CLU-style iterators, such a pattern could be defined as follows:

```
pattern use_decl(d : remote Declaration) : Use begin
  case result.use_decl begin
    match ?d begin
      yield;
    end;
  end;
end;
```

The inside-out nature of patterns—the fact that patterns take a result and produce values for the arguments—may make such an extension opaque.

None of these enhancements change the language dramatically, but neither do they reflect serious expressive failures of the language. It seems then that most of the features one would desire in a declarative compiler description language are present. Considering therefore the execution times of descriptions implemented by the current APS compiler, it would seem that future effort with APS should be expended in more efficient implementation.

The prototype implementation uses closures extensively. Every attribution clause is implemented as a function that executes everything that cannot cause demand evaluation. Any other statement (for example a conditional that tests possibly unevaluated attributes) is converted into a closure. This closure is used as a guard thunk on every attribute instance that could be defined by the statement. The closure, when invoked, evaluates the needed expression and then behaves as an attribution clause, possibly generating further closures. This use of closures requires little analysis, but uses space and time prolifically. Finally attribute definition usually involves storing a closure on the worklist of an attribute instance. Possibilities for better implementation include using more efficient methods for common simple cases, and using state-carrying coroutines rather than closure-creating closures.

10.2.3 Writing and Reading Descriptions

For the most part, APS has been a convenient language for writing descriptions. The Oberon2 compiler was able to be expressed at a similar level to the English report. The type-checking rules, for example, were given in approximately the same order as in the report, and were not much longer. The APS front end in Appendix C was harder to write, but that was because the language was changing while the description was being written.

Since APS does not require declaration before use, and only uses lexical order for definition precedence, the description writer is free to organize the declarations in almost any order. It's not immediately clear what conventions should be followed for best comprehensibility.

The rule that earlier definitions take precedence over later definitions was chosen to approximate the effect of "if" and "case" statements in a variety of languages. This choice has the opposite effect of repeated definition (assignment) in imperative languages. Nevertheless, one reason to believe that the correct choice was made is that exceptions must always be placed before the general rules; if one reads the rules in order, no rule can be contradicted by a later rule.

APS encourages descriptions to be factored by concept rather than by type of node in the tree being attributed. As a result, if one wishes to learn from a description what computations are carried out on a particular node, one must read the entire description, being careful to look for pattern definitions that match the node in question. This fact may seem a deleterious effect of factoring by concept. However, factoring by node can be accomplished by an automatic process, perhaps in a browser, whereas factoring by concept cannot be done automatically. Moreover, in order to understand a description, one should first learn the concepts described rather than the details of what "happens" at each node in the tree.

APS does not require dependencies to be statically analyzable as noncircular. This approach contrasts with Maddox's Colander2. Even though the Colander2 compiler uses sophisticated static analysis, it is still necessary to contort the description to avoid circularity.

Portions of the grammar of Modula2 need to be duplicated, and certain computations must be carried out using tree-walking functions rather than using (simpler) attribute equations. It remains to be seen how a mix of statically analyzable and unanalyzable dependencies can be implemented efficiently. In any case, the choice taken for APS makes descriptions much easier to write, and to understand.

10.3 Discussion of Descriptional Composition

Descriptional composition is a tool of implementation, it does not change the semantics of the modules being composed. It is interesting only when it improves some measure of program performance. In the case of the Oberon2 compiler, there is a sizable improvement in speed.

10.3.1 Analysis

The benefits of descriptional composition are twofold: first, an intermediate structure need not be constructed and then traversed; second, the action of combining two modules permits more computation to be carried out at description compile-time. Logically, however, the descriptionally composed module does all the work of the modules that make it up. In particular all the computation carried out on the intermediate structure that was removed is still carried out, only it has been translated to apply to a different source structure. For example, the GCC tree package includes attributes for computing the types of expression nodes. This computation is still carried out in the descriptionally composed Oberon2 compiler, but with composed attributes operating on the Oberon2 abstract syntax tree.

Since there is often not a one-to-one correspondence between the nodes in the input and output trees in the first stage, the computation that is moved to the first stage may be more complex, that is may require more attributes. For example, even after optimization gets rid of composed attributes that are not needed, the composed version of the Oberon2 back-end still has more attributes (about 100) than all the attributes (about 60) of the modules that were composed to produce it. As a result, implementing the composed version may be more difficult than implementing the modules separately.

Descriptional composition has the most potential for benefit when one or both of the following situations apply:

1. The time spent constructing and traversing the intermediate form is a significant amount of the time used in the modules being composed. This situation was true for all previously reported instances of descriptional composition [13, 14, 39, 41], but is not particularly valid for the Oberon2 compiler described here, since many more attributes (that is, computation) are involved.

2. The composition admits considerable compile-time evaluation and simplification. This situation is manifestly true for the Oberon2 compiler. The size of the composed description was reduced by 75% by compile-time simplifications.

In other situations, where much of the work does *not* involve creating and traversing intermediate structure and where few simplifications can be performed at compile-time, it is unlikely that descriptional composition will show much benefit.

10.3.2 Looking Beyond Descriptional Composition

As described in the introduction, the motivation for investigating the technology of descriptional composition stems from the desire to see efficient implementation of highly factored compiler descriptions. In particular, compilers are easier to modify and reuse if they consist of a large number of modular stages. Descriptional composition, however, only addresses some of the issues that arise.

For example, suppose a compiler stage produces an output tree very similar to its input tree by performing a straightforward canonicalization. If the input tree was highly decorated (with type and binding information, for example), and these decorations are needed for further processing, it will then be necessary to re-decorate the output tree before performing the next stage. While this task can be easily described by reusing a decoration module, it would nevertheless involve computation on this new tree. If the stage were then to be composed with a following stage, descriptional composition only moves the computation back to the input tree, it does not get rid of it.

The problem, however, of redecorating a transformed tree has been extensively treated in the literature on incremental attribution mechanisms. Although incremental systems work dynamically to bring decorations up to date after a transformation, typical implementations use static analysis of possible changes for greater efficiency. One can thus envision that such analysis could be applied to descriptions of the transformation and the decoration. The result would be a specialized version of the decoration module that uses the result of the previous decoration. Such an application of the technology of incremental attribute update in new setting would nicely complement the work performed by descriptional composition.

10.4 Further Work

The APS project is not complete; there are several tasks that I hope to accomplish subsequently. I hope to bring the boot-strapping process to completion. Then, after compiling the compiler with itself (no mean feat given the memory requirements), APS could be weaned from Common Lisp. That would permit APS to be more easily distributed to other researchers.

The APS type checker requires finer dependency analysis than is provided by dynamic dependencies in the current APS compiler and run-time system. Currently, the module is implemented by turning off a safety check and adding extra dependencies (in an isolated section of the module). A technique for determining the necessary dependencies must be developed and implemented.

The current APS compiler performs some global dependency analyses, in particular it places a module's entities in a total order of strongly-connected groups of mutually dependent attributes and functions. If pattern matching is removed as described in Section 3.4.4, finer, production-level analysis in the style of other attribute-grammar based systems would then be possible. However, the two major existing APS programs (the Oberon2 and APS compilers) make heavy use of dynamically non-circular attributes, which will appear circular to any static analysis system. Moreover, remote attribution requires a form of fibering analysis. How to accomplish such analysis is still unclear (and more ground for further work). Earlier work [11] shows that fibering analysis can be adapted to collection attributes, but this technique has not been tested on anything save the smallest examples. Maddox [70] has implemented a method based on isolating circular dependencies in regions. It remains to be seen which method or methods are practical.

On a different note, the current GCC tree intermediate language is implemented through a translation to C source text. One of the first changes needs to be a translation of switch statements; the semantics used by the GCC compiler is unclear, and so they were omitted. Once this translation is complete, the next task is to attach the GCC tree package in APS directly to the GCC compiler back end. This task is made more difficult by the nature of the current interface in the GCC compiler—half structural and half procedural with some highly convoluted memory management thrown in for good measure. However, accomplishing this task would bring rewards because it would enable more efficient compilation and also provide the only clean interface to the GCC back end for front ends. Completing this task would make APS attractive to compiler researchers.

10.5 Conclusions

I embarked upon this research in order to determine whether descriptional composition could be applied usefully in realistic situations. While this goal was accomplished with the conclusion that it was indeed useful, along the way, the importance of a good declarative description language became evident. Attribute grammars are wholly insufficient as a vehicle for expressing compilers and related systems. Thus many of the contributions of the research and of this dissertation are in demonstrating the need for specific extensions and showing how they can be implemented. As a compiler description language, APS seems fairly complete. More pressing, therefore, than further extension is better implementation.

The success of descriptional composition in a realistic setting is encouraging. Moreover, it appears that incremental attribution technology could nicely complement descriptional composition in the practical implementation of modular compiler descriptions.

Bibliography

[1] A. V. Aho, M. Ganapathi, and S. W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Transactions on Programming Languages and Systems*, 11(4):491–516, October 1989.

[2] Alexander Aiken. Set constraints: results, application and future directions. In A. Borning, editor, *Second Workshop on the Principles and Practice of Constraint Programming*, pages 326–335. Springer-Verlag, May 1994.

[3] Martin Alt, Christian Fecht, Christian Ferdinand, and Reinhard Wilhelm. The Trafola-H system. In *The PROSPECTRA system*. 1991.

[4] Uwe Aßmann. How to uniformly specify program analysis and transformation with graph rewrite systems. In Peter A. Fritzson, editor, *Proceedings of Compiler Construction, 6th International Conference, CC'96*, volume 1060 of *Lecture Notes in Computer Science*. Springer-Verlag, April 1996.

[5] Isabelle Attali and Jacques Chazarain. Functional evaluation of strongly non circular Typol specifications. In Pierre Deransart and Martin Jourdan, editors, *Attribute Grammars and their Applications. International Conference WAGA Proceedings*, volume 461 of *Lecture Notes in Computer Science*, pages 157–176, Berlin, Heidelberg, New York, 1990. Springer-Verlag.

[6] Isabelle Attali and P. Franchi-Zannettacci. Unification-free execution of Typol programs by semantic attribute evaluation. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, pages 160–177, Cambridge, Massachusetts and London, England, August 1988. The MIT Press.

[7] Lex Augusteijn. The Elegant compiler generator system. In Pierre Deransart and Martin Jourdan, editors, *Attribute Grammars and their Applications. International Conference WAGA Proceedings*, volume 461 of *Lecture Notes in Computer Science*, pages 238–254, Berlin, Heidelberg, New York, 1990. Springer-Verlag.

[8] B. Baum. Another kind of modular attribute grammar. In Uwe Kastens and P. Pfahler, editors, *Proceedings of Compiler Construction, 4th International Conference, CC'92*, volume 641 of *Lecture Notes in Computer Science*, pages 44–50, Berlin, Heidelberg, New York, October 1992. Springer-Verlag.

[9] Hans Boehm, Alan Demers, and James Donahue. An informal description of Russell. Technical Report TR 80-430, Cornell University, 1980.

[10] Jonathan Bowen. From programs to object code using logic and logic programming. In R. Giegerich and S. L. Graham, editors, *Code Generation - Concepts, Tools, Techniques. Workshops in Computer Science*, pages 173–192. Springer-Verlag, Berlin, Heidelberg, New York, 1992.

[11] John Boyland. Remote attribution. Draft, 1995.

[12] John Boyland. Conditional attribute grammars. *ACM Transactions on Programming Languages and Systems*, 18(1):73–108, January 1996.

[13] John Boyland, Charles Farnum, and Susan L. Graham. Attributed transformational code generation for dynamic compilers. In R. Giegerich and S. L. Graham, editors, *Code Generation - Concepts, Tools, Techniques. Workshops in Computer Science*, pages 227–254. Springer-Verlag, Berlin, Heidelberg, New York, 1992.

[14] John Boyland and Susan L. Graham. Composing tree attributions. In *Conference Record of the Twenty-first Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 375–388, New York, January 1994. ACM Press.

[15] David G. Bradlee, Robert R. Henry, and Susan J. Eggers. The Marion system for retargetable instruction scheduling. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 229–240, New York, June 1991. ACM Press.

[16] David Gordon Bradlee. *Retargetable Instruction Scheduling for Pipeline Processors.* PhD thesis, University of Washington, 1991. Technical report CSE 91-08-07.

[17] J. Cai, R. Paige, and R. Tarjan. More efficient bottom-up multi-pattern matching in trees. *Theoretical Computer Science*, 106(1):21–60, November 1992.

[18] D. R. Chase. An improvement to bottom-up tree pattern matching. In *Conference Record of the Fourteenth ACM Symposium on Principles of Programming Languages*, pages 168–177, New York, January 1987. ACM Press.

[19] D. Clement, J. Incerpi, and G. Kahn. CENTAUR: towards a "software tool box" for programming environments. In F. Long, editor, *Software Engineering Environments. International Workshop on Environments. Proceedings*, pages 287–304, Berlin, Heidelberg, New York, September 1989. Springer-Verlag.

[20] W. F. Clocksin and C. S. Mellish. *Programming in Prolog.* Springer-Verlag, Berlin, Heidelberg, New York, 3rd edition, 1987.

[21] William R. Cook, Walter L. Hill, and Peter S. Canning. Inheritance is not subtyping. In *Conference Record of the Seventeenth ACM Symposium on Principles of Programming Languages*, pages 125–135, New York, January 1990. ACM Press.

[22] Bruno Courcelle. Fundamental properties of infinite trees. *Theoretical Computer Science*, 25(2):95–169, March 1983.

[23] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute Grammars: Definitions, Systems and Bibliography*, volume 323 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 1988.

[24] T. Despeyroux. Typol: a formalism to implement Natural Semantics. Technical Report 94, INRIA, 1988.

[25] James Donahue and Alan Demers. Data types are values. *ACM Transactions on Programming Languages and Systems*, 7(3):426–445, July 1985.

[26] G. D. P. Dueck and G. V. Cormack. Modular attribute grammars. *Computer Journal*, 33(2):164–172, April 1990.

[27] Hemlut Emmelmann. Code selection by regularly controlled term rewriting. In R. Giegerich and S. L. Graham, editors, *Code Generation - Concepts, Tools, Techniques. Workshops in Computer Science*, pages 3–29. Springer-Verlag, Berlin, Heidelberg, New York, 1992.

[28] Hemlut Emmelmann, F.-W. Schroer, and R. Landwehr. BEG—a generator for efficient back ends. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 227–237, New York, July 1989. ACM Press.

[29] Charles Farnum. *Pattern-based languages for prototyping compiler optimizers*. PhD thesis, Computer Science Division—EECS, University of California, Berkeley, December 1990. Technical report UCB/CSD 90/608.

[30] Charles Farnum. DORA — an environment for experimenting with compiler optimizers. In *Proceedings of the IEEE 1992 International Conference on Computer Languages*, April 1992.

[31] Charles Farnum. Pattern-based tree attribution. In *Conference Record of the Nineteenth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 211–222, January 1992.

[32] Rodney Farrow. Generating a production compiler from an attribute grammar. *IEEE Software*, 1(4):77–93, October 1984.

[33] Rodney Farrow. Automatic generation of fixed-point-finding evaluators for circular, but well-defined, attribute grammars. In *Proceedings of the ACM SIGPLAN '86 Symposium on Compiler Construction*, pages 85–98, New York, July 1986. ACM Press.

[34] Rodney Farrow. The Linguist translator-writing system. Technical report, Declarative Systems Inc., June 1989.

[35] Rodney Farrow. Fibered evaluation in Linguist. unpublished, 1990.

[36] Rodney Farrow, T. J. Marlowe, and D. M. Yellin. Composable attribute grammars: Support for modularity in translator design and implementation. In *Conference Record of the Nineteenth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 223–234, January 1992.

[37] Christian Ferdinand. Pattern matching in a functional transformation language using treeparsing. In P. Deransart and J. Maluszyński, editors, *Programming Language Implementation and Logic Programming, Proceedings*, volume 456 of *Lecture Notes in Computer Science*, pages 358–371, Berlin, Heidelberg, New York, 1990. Springer-Verlag.

[38] Christopher W. Fraser, David R. Hanson, and Todd A. Proebsting. Engineering a simple, efficient code-generator generator. *ACM Letters on Programming Languages and Systems*, 1(3):213–226, September 1992.

[39] Harald Ganzinger and Robert Giegerich. Attribute coupled grammars. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 157–170, New York, June 1984. ACM Press.

[40] Harald Ganzinger, Robert Giegerich, and Martin Vach. MARVIN: A tool for applicative and modular compiler specifications. Technical Report 220, University Dortmund, July 1986.

[41] Robert Giegerich. Composition and evaluation of attribute coupled grammars. *Acta Informatica*, 25(4):355–423, 1988.

[42] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Functional Programming Languages and Computer Architecture 6th ACM Conference Proceedings*, pages 223–232, New York, 1993. ACM Press.

[43] Wolfgang Goerigk, Ulrich Hoffmann, and Heinz Knutzen. CommonLisp$_0$ and CLOS$_0$: The language definition. Technical report, Christian-Albrechts-Universität zu Kiel, September 1993.

[44] Susan L. Graham, Michael A. Harrison, and Ethan V. Munson. The Proteus presentation system. In *SIGSOFT '92: Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments*, pages 130–138, New York, December 1992. ACM Press. Appeared as *ACM Software Engineering News 17* (5).

[45] Reinhold Heckmann. A functional language for the specification of complex tree transformations. In Harald Ganzinger, editor, *European Symposium on Programming (ESOP '88)*, volume 300 of *Lecture Notes in Computer Science*, pages 175–190. Springer-Verlag, Berlin, Heidelberg, New York, 1988.

[46] Görel Hedin. An overview of door attribute grammars. In Peter A. Fritzson, editor, *Proceedings of Compiler Construction, 5th International Conference, CC'94*, volume 786 of *Lecture Notes in Computer Science*, pages 31–51. Springer-Verlag, April 1994.

[47] Christoph M. Hoffmann and Michael J. O'Donnell. Pattern matching in trees. *Journal of the ACM*, 29(1):68–95, January 1982.

[48] P. Hudak, Simon Peyton Jones, and P. Wadler (editors). Report on the programming language Haskell, a non-strict purely functional language (version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[49] ISO/IEC JTC 1/SC 22/WG 16. Programming language ISLISP: Working draft 11.4. Technical report, International Standards Organization, July 1994.

[50] Gregory F. Johnson and Charles N. Fischer. A meta-language and system for nonlocal incremental attribute evaluation in language-based editors. In *Conference Record of the Twelfth ACM Symposium on Principles of Programming Languages*, pages 141–151, New York, January 1985. ACM Press.

[51] S. C. Johnson. Yacc: Yet another compiler compiler. In *Unix Programmer's Manual*. Bell Telephone Laboratories, 7th edition, January 1979.

[52] M. Jourdan, D. Parigot, C. Julie, O. Durin, et al. Design, implementation and evaluation of the FNC-2 attribute grammar system. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 209–222, New York, June 1990. ACM Press.

[53] Martin Jourdan, Carole Le Bellec, and Didier Parigot. The OLGA attribute grammar description language: Design, implementation and evaluation. In Pierre Deransart and Martin Jourdan, editors, *Attribute Grammars and their Applications. International Conference WAGA Proceedings*, volume 461 of *Lecture Notes in Computer Science*, pages 222–237, Berlin, Heidelberg, New York, 1990. Springer-Verlag.

[54] Martin Jourdan and Didier Parigot. Internals and externals of the FNC-2 attribute grammar system. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems. International Summer School SAGA Proceedings*, volume 545 of *Lecture Notes in Computer Science*, pages 485–504, Berlin, Heidelberg, New York, 1991. Springer-Verlag.

[55] G. Kahn. Natural semantics. In F. Brandenburg, G. Vidal-Nacquet, and W. Wirsig, editors, *STACS '87: Fourth Annual Symposium on Theoretical Aspects of Computer Sciences*, volume 247 of *Lecture Notes in Computer Science*, pages 22–39, Berlin, Heidelberg, New York, 1987. Springer-Verlag.

[56] Uwe Kastens. Attribute grammars as a specification method. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems. International Summer School SAGA Proceedings*, volume 545 of *Lecture Notes in Computer Science*, pages 15–47, Berlin, Heidelberg, New York, 1991. Springer-Verlag.

[57] Uwe Kastens. Attribute grammars in a compiler construction environment. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems. International Summer School SAGA Proceedings*, volume 545 of *Lecture Notes in Computer Science*, pages 380–400, Berlin, Heidelberg, New York, 1991. Springer-Verlag.

[58] Uwe Kastens, B. Hutt, and E. Zimmermann. *GAG: A Practical Compiler Generator*, volume 141 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 1982.

[59] Uwe Kastens and William M. Waite. Modularity and reusability in attribute grammars. *Acta Informatica*, 31:601–627, 1994.

[60] Ken Kennedy. A survey of data flow analysis techniques. In Stephen S. Muchnick and Neil D. Jones, editors, *Program flow analysis : theory and applications*, pages 5–54. Prentice-Hall, Englewood Cliffs, N.J., 1981.

[61] Richard Kenner. Targetting and retargetting the GNU C compiler. Technical report, New York University Ultracomputer Research Lab, January 1995. POPL '95 Tutorial Notes.

[62] Brian W. Kernighan and Dennis M. Ritchie. The M4 macro processor. In *Unix Programmer's Manual*. Bell Telephone Laboratories, 7th edition, January 1979.

[63] David J. King and John Launchbury. Structuring depth-first search algorithms in haskell. In *Conference Record of the Twenty-second Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 344–354, New York, January 1995. ACM Press.

[64] Donald E. Knuth. Semantics of context free languages. *Mathematical Systems Theory*, 2(2):127–145, June 1968. Corrections appear in *Mathematical Systems Theory 5*, 1 (1971), 95–96.

[65] John Launchbury and Tim Sheard. Warm fusion: Deriving build-cats from recursive definitions. In *Functional Programming Languages and Computer Architecture 7th ACM Conference Proceedings*, pages 314–323, New York, 1995. ACM Press.

[66] M. E. Lesk and E. Schmidt. LEX—a lexical analyzer generator. In *Unix Programmer's Manual*. Bell Telephone Laboratories, 7th edition, January 1979.

[67] Peter Lipps, Ulrich Möncke, and Reinhard Wilhelm. OPTRAN: A language/system for the specification of program transformations—system overview and experiences. In D. Hammer, editor, *Proceedings of the Workshop on Compiler Compilers and High Speed Compilation*, volume 371 of *Lecture Notes in Computer Science*, pages 52–65, Berlin, Heidelberg, New York, October 1988. Springer-Verlag.

[68] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. The MIT Press, Cambridge, Massachusetts and London, England, 1986.

[69] William Maddox. Colander2: A reference manual. Technical Report internal report, Computer Science Division—EECS, University of California, Berkeley, 1993.

[70] William Maddox. *Incremental Static Analysis*. PhD thesis, University of California, Berkeley, 1996.

[71] S. Marlow and P. Wadler. Deforestation for higher-order functions (functional programming). In J. Launchbury and P. Sansom, editors, *Function Programming, Glasgow 1992*, pages 154–165, Berlin, Heidelberg, New York, 1993. Springer-Verlag.

[72] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* The MIT Press, Cambridge, Massachusetts and London, England, 1990.

[73] H. Mössenböck and Niklaus Wirth. The programming language Oberon-2. *Structured Programming*, 12(4):179–195, 1991.

[74] G. Nelson, editor. *Systems Programming in Modula-3.* Prentice-Hall, Englewood Cliffs, N.J., 1991.

[75] Pedro Palao Gostanza, Ricardo Peña, and Manuel Núñez. A new look at pattern matching in abstract data types. In *Proceedings of the ACM SIGPLAN Internation Conference on Functional Programming (ICFP '96)*, pages 110–121. ACM Press, New York, 1996.

[76] Didier Parigot, Gilles Roussel, Etienne Duris, and Martin Jourdan. Attribute grammars: A declarative functional language. Technical Report RR 2662, INRIA, Rocquencourt, France, October 1995.

[77] V. Paxson. Man pages. In flex-2.4.6.tar.z, Free Software Foundation, November 1993.

[78] Eduardo Pelegrí-Llopart and Susan L. Graham. Optimal code generation for expression trees: an application of BURS theory. In *Conference Record of the Fifteenth ACM Symposium on Principles of Programming Languages*, pages 294–308, New York, January 1988. ACM Press.

[79] Simon L. Peyton Jones and John Launchbury. Unboxed values as first class citizens in a non-strict functional language. In J. Hughes, editor, *Functional Programming Languages and Computer Architecture 5th ACM Conference Proceedings*, volume 523 of *Lecture Notes in Computer Science*, pages 636–666. Springer-Verlag, Berlin, Heidelberg, New York, 1991.

[80] Todd A. Proebsting. Simple and efficient BURS table generation. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 322–330, New York, July 1992. ACM Press.

[81] Christian Queinnec. Compilation of non-linear, second order patterns on S-expressions. In P. Deransart and J. Maluszyński, editors, *Programming Language Implementation and Logic Programming, Proceedings*, volume 456 of *Lecture Notes in Computer Science*, pages 340–357, Berlin, Heidelberg, New York, 1990. Springer-Verlag.

[82] Christian Queinnec and Jean-Marie Geffroy. Partial evaluation applied to symbolic pattern matching with intelligent backtrack. In *Workshop for Static Analysis*, October 1992.

[83] J. T. Schwartz, R. B. K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to SETL*. Springer-Verlag, Berlin, Heidelberg, New York, 1986.

[84] Olin Shivers. Control flow analysis in scheme. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174. ACM Press, New York, July 1988.

[85] Jay M. Sipelstein and Guy E. Blelloch. Collection-oriented languages. Technical Report CMU-CS-90-127, Carnegie Mellon University, Pitsburgh, PA, March 1991.

[86] Anthony M. Sloane. An evaluation of an automatically generated compiler. *ACM Transactions on Programming Languages and Systems*, 17(5):691–703, September 1995.

[87] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Conference Proceedings of OOPSLA'86 – Object Oriented Programming Systems, Languages and Applications*, pages 38–45, New York, October 1986. ACM Press. Appeared as *ACM SIGPLAN Notices 21* (11), November 1986.

[88] Marvin Solomon. Type definitions with parameters. In *Conference Record of the Fifth ACM Symposium on Principles of Programming Languages*, pages 31–38, New York, January 1978. ACM Press.

[89] Richard M. Stallman. GCC.

[90] Guy Lewis Steele, Jr. *Common Lisp: The Language*. Digital Press, 2nd edition, 1990.

[91] J. Strong, J. Wegstein, A. Tritter, J. Olsztyn, O. Mock, and T. Steel. The problem of programming communications with changing machines: a proposed solution. *Communications of the ACM*, 1(8):12–18, August 1958. Continued in *Communications of the ACM 1*, 9 (September 1958) 9–15.

[92] Doaitse Swiestra and Harald Vogt. Higher order attribute grammars. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems. International Summer School SAGA Proceedings*, volume 545 of *Lecture Notes in Computer Science*, pages 256–296. Springer-Verlag, Berlin, Heidelberg, New York, 1991.

[93] Clemens Szypersky, Stephen Omohundron, and Stephan Murer. Engineering a programming language: The type and class system of Sather. Technical Report TR-93-064, International Computer Science Institute, November 1993.

[94] D. A. Turner. Miranda: A non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*. Springer-Verlag, New York, 1985.

[95] J. Uhl, S. Drossopoulou, G. Persch, G. Goos, M. Dausmann, G. Winterstein, and W. Kirchgässner. *An Attribute Grammar for the Semantic Analysis of Ada*, volume 139 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, Heidelberg, New York, 1982.

[96] Harald H. Vogt, S. D. Swiestra, and M. F. Kuiper. Higher order attribute grammars. In *Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 131–145, New York, July 1989. ACM Press.

[97] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–48, 1990. Originally published in ESOP 1988, LNCS 300.

[98] Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Conference Record of the Fourteenth ACM Symposium on Principles of Programming Languages*, pages 307–313, New York, January 1987. ACM Press.

[99] W. M. Waite. Use of attribute grammars in compiler construction. In Pierre Deransart and Martin Jourdan, editors, *Attribute Grammars and their Applications. International Conference WAGA Proceedings*, volume 461 of *Lecture Notes in Computer Science*, pages 255–265, Berlin, Heidelberg, New York, 1990. Springer-Verlag.

[100] Janet A. Walz and Gregory F. Johnson. Incremental evaluation for a general class of circular attribute grammars. In *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 209–221, New York, July 1988. ACM Press.

[101] D. A. Watt. Modular description of programming languages. *Computer Journal*, 34(6):A009–28, December 1991.

[102] Reinhard Wilhelm. Tree transformations, functional languages, and attribute grammars. In Pierre Deransart and Martin Jourdan, editors, *Attribute Grammars and their Applications. International Conference WAGA Proceedings*, volume 461 of *Lecture Notes in Computer Science*, pages 116–129, Berlin, Heidelberg, New York, 1990. Springer-Verlag.

[103] Niklaus Wirth. *Programming in Modula-2.* Springer-Verlag, Berlin, Heidelberg, New York, 1982.

Appendix A

Summary of APS

This appendix describes in brief the features of APS. Appendix C gives a definition in APS of the type system. The syntax is given with `typewriter` font templates with *italic* slots. The syntax to be used for a slot may be given in a parenthesized cross-reference. Optional portions are surrounded by italic braces *[ ]*; zero or more repetitions are indicated using an italic star ***; alternatives are separated by a slash /; italic parentheses *( )* are used for grouping. The syntax

nonterminal ::= ... / *template*

is used to give an additional template for a nonterminal.

Section A.11 gives a listing of predefined entities.

A.1 Namespaces

The APS description language has four namespaces: signatures, types, patterns and values. A *signature* is a specification of services that may be accessed through a type that satisfies the signatures. A *type* includes an implementation of services and if it is a phylum, contains all the nodes created in the type. A service (of any namespace) may be fetched from a type using the syntax *type*`$`*service*. A *pattern* is a partial specification of the form of a value. Every *value* has a type.

Section A.2 specifies all the named entities introduced by declarations. The next four sections describe the syntactic units that correspond to the four namespaces: signatures, types, patterns and expressions.

A.2 Named Declarations

There are three classes of declarations; this section describes *named* declarations:

declaration ::= *named'* `|` *unnamed*(A.7) `|` *clause*(A.8)
named' ::= *[*`var` / `private` / `public`*]* *named*

The name of a declaration is a string of alphanumeric characters (including `_`) starting with an alphabetic character. Names introduced in the signature namespace are conventionally all uppercase. Types and phyla are given capitalized names; all other declarations are all lowercase. The following names are reserved words:

> `and, attribute, begin, case, circular, class, collection, constant,`
> `constructor, elscase, else, elsif, end, endif, extends, for,`
> `function, if, in, infix, infixl, infixr, inherit, input, match,`
> `module, not, on, or, pattern, phylum, pragma, private, procedure,`
> `public, remote, signature, then, type, var, with`

The reserved words `constant` and `elscase` are not currently used in APS syntax. The keywords `infix`, `infixl`, `infixr` and `with` are used in situations not described in this appendix. A reserved word may be used as a name if surrounded by parentheses.

A contour (declaration scope) may not have two entities with the same name and namespace unless that name is the special name `_` (declarations given this name cannot be referred to). A declaration in a nested contour *shadows* (makes inaccessible) declarations of the same name and namespace in outer scopes. In other words, APS uses standard Algol-style lexical scoping.

Any declaration declared in a module or class is either public or private. Entities are public unless otherwise indicated by prefixing the declaration with the keyword `private`. This default behavior is switched by the declaration "`private;`" for all following declarations; then a public entity must be declared using the keyword `public`.

A public declaration in a module or class may be declared `var`, or else it is implicitly declared *constant*. Constant variable declarations may not be defined (that is, assigned using `:=` or `:>`). Constant declarations must not depend on var declarations.

Each declaration ends with a semicolon, so lists of declarations need no separation character.

A.2.1 Class Declarations

A class declaration has the form

named ::= ... /
  `class` *name* `[`*type-formals*(A.2.14)`]` *[result-name]* *[*`::` *signatures*(A.3)*]* *[*`begin`
    *top-level**
  `end`*]*`;`
top-level ::= *named* / *unnamed*(A.7)

Class declarations introduce names in the signature namespace. A signature is produced when a class is instantiated (see Section A.3.1). Within the scope of the class, the type satisfying the signature may be referred to as *result-name* (or `Result`, if no name is given). Any type satisfying the signature also satisfies the *parent signatures* (the ones after the double colon "`::`"). If the parent signatures contain var or input capabilities, these capabilities are only present for instantiations of the class with the respective capabilities (see Section A.3.1). Classes are only used to specify interfaces for modules; they are not executed at run-time.

A.2.2 Module Declarations

A module declaration has the form

named ::= ... /
  `module` *name* `[`*type-formals*(A.2.14)`]` [`(`*value-formals*(A.2.13)`)`][`::` *signatures*(A.3)]
    [`phylum`] [*result-name*] [`extends` *type*(A.4)]
  [`begin`
    *top-level**(A.2.1)
  `end`]`;`

A module declaration introduces a name in both the signature and type namespaces.

A module may be instantiated in the same way as a class to form a signature; no value parameters are given in this case. Alternatively, a module may be instantiated as a type (see Section A.4.1), in which case any value parameters must be supplied. The type resulting from instantiating a module *as a type* satisfies by construction the signature resulting from instantiating the module *as a signature*.

Within the scope of the module, the type being created is visible as *result-name* (or `Result`, if no name is given). The type is created as an extension of a base type, or is a new type (see Section A.2.4). An extension of a type is equal to it for the purposes of type-equality, but may provide additional services. Any services provided by the extension (through its inferred signatures) are visible in a virtual contour around the body of the module.

Procedures, types, or variables declared outside the scope of the module cannot be called, modified (see Section A.4), or defined respectively inside the body of the module.

Any parent signature (or parent of a parent signature etc.) specified in the module declaration must be satisfied either by the type being extended, or by matching the entities exported by the module against the entities declared in the signature. See Section A.10 for the operation of matching.

A module cannot be passed as an actual type parameter.

A.2.3 Signature Declarations

A signature declaration introduces a name in the signature namespace:

named ::= ... / `signature` *name* `:=` *signatures*(A.3)`;`

Any type that satisfies *signatures* also satisfies this new signature. In other words, this declaration does not introduce a new signature; it only provides a way to give a name for a list of signatures.

A.2.4 Type Declarations

A type declaration declares either a new type or a computed type. The first kind of type declaration has the form

named ::= ... / `type` *name*`;`

This declaration binds *name* to a new type with signature `var input TYPE[]` (see Section A.3.1). Constructors may be declared for such a type (see Section A.2.10).

The second kind of type declaration (a computed type declaration) has the form

named ::= ... / `type` *name* *[*`::` *signatures*(A.3)*]* `:=` *type*(A.4)`;`

This declaration binds *name* to the type. If no signatures are given, the signatures are inferred from the type. If signatures are given, the type must satisfy them. By giving the signatures therefore, the new type declaration cannot have more signatures than the type from which it is computed.

A.2.5 Phylum Declarations

A phylum is a special kind of type with the the property that all its instances may be matched using a top-level match (see Section A.7.3). There are restrictions on what operations can be performed on a value whose type is a phylum.

As with type declarations, a phylum declaration declares either a new phylum or a computed phylum. The first kind of phylum declaration has the form

named ::= ... / `phylum` *name*`;`

This declaration binds *name* to a new phylum with signature `var input PHYLUM[]`. Constructors may be declared for this phylum (see Section A.2.10).

The second kind of phylum is the same as a computed type declaration:

named ::= ... / `phylum` *name* *[*`::` *signatures*(A.3)*]* `:=` *type*(A.4)`;`

This type must be a phylum.

A.2.6 Variable Declarations

Variable declarations introduce names in the value namespace. A variable declaration has the form:

named ::= ... / *[*`input`*]* *[*`circular`*]* *[*`collection`*]* *name* `:` *type*(A.4) *[predef]*
predef ::= `:=` *default* / `:>` *init*`,`*combine*

Here *default*, *init* or *combine* are all expressions (Section A.6). If the variable declaration is declared as `circular`, the type must satisfy the `LATTICE[]` signature. A composition default (*init* and *combine*) may only be given for a non-circular collection variable. The first part (*init*) gives the initial value for the collection and the second part (*combine*) gives the combining function. If no initial value or no combining function is specified for a non-circular collection variable, the type must satisfy the `COMBINABLE[]` signature, and the missing values are fetched from the type. An *input* variable may be defined external to the module, assuming the type from which the variable is fetched has the appropriate signature with input capability (see Section A.3.1).

A.2.7 Attribute Declarations

Attribute declarations also introduce names in the value namespace. An attribute declaration has the form:

named ::= ... /
  [`input`] [`circular`] [`collection`] `attribute`
    (*phylum*(A.4) / `(`*value-formal*(A.2.13)`)`)`.`*name* `:` *type*(A.4) [*predef*(A.2.6)]

The phylum decorated by this attribute is specified in one of two ways. Either it is specified directly (the first choice) or indirectly, as the type of a value formal (the second choice). The phylum must satisfy the `var PHYLUM[]` signature. The default value or initial value must not include procedure calls. The same restrictions apply to attributes as to variable declarations (Section A.2.6) The value formal may be used in the default or initial value.

An attribute declaration declares a variable (or *slot*) for every node of the phylum being decorated. An attribute declaration introduces a function that may be used to access the variables. (As explained in Section A.6.2, the syntax *node*`.`*attr* may be used as syntactic sugar for *attr*`(`*node*`)`). These variables may be defined inside the module; additionally if the attribute is declared as `input`, they may be defined external to the module as an input service. When fetched from types, uses of the variables are var services. Thus an attribute declaration not declared `input` is indistinguishable outside the module from a var function.

A.2.8 Function Declarations

A function declaration introduces a name in the value namespace and has the form:

named ::= ... /
  `function` *name*`(`*value-formals*(A.2.13)`)`
    [`collection`] [*result-name*] `:` *type*(A.4) [*predef*(A.2.6)]
  [`begin` *declaration** `end`] ;

Within the scope of the function, the result being computed is available as the variable *result-name* (or `result` if no name is given). Functions are *strict*; a function body is not instantiated until the arguments are computed. On the other hand, a function call is presumed monotonic in arguments whose types satisfy the signature `LATTICE[]`, if the result has a lattice type and the call used monotonically.

Procedures, types, or variables declared outside the scope of the function cannot be called, modified, or defined respectively inside the body of the function.

A.2.9 Procedure Declarations

A procedure declaration introduces a name in the value namespace and has the form:

named ::= ... /
  `procedure` *name*`(`*value-formals*(A.2.13)`)`
    [[`circular`] [`collection`] [*result-name*] `:` *type*(A.4) [*predef*(A.2.6)] /

    (*variable-declarations*(A.2.6))*]*
  [`begin` *declaration** `end`*]* ;

A procedure may have no results, one result (whose name by default is `result`) or multiple results. Procedures are *non-strict*; a procedure body may be instantiated before the arguments are computed. A procedure is effectively called exactly once per instance of a call site. That is, a single call can never lead to the effect of two calls, nor can two calls be coalesced into one. A procedure cannot be passed as a parameter nor assigned to a variable.

A.2.10 Constructor Declarations

A constructor declaration introduces its name in both the value and pattern namespaces. It has the form:

named ::= ... / `constructor` *name*`(`*value-formals*(A.2.13)`)` `:` *type*(A.4)`;`

The type must be a new type declared in the same context (function or module) as the constructor. A constructor creates a node when called as a function or a procedure. This node is distinct from any node created by any other constructor. The types of the formal parameters must satisfy the signature `BASIC[]`.

If *type* is not a phylum, then none of the formal parameters types may be phyla (although remote types are permitted (Section A.4.2)). When used as a value, such a constructor is treated as a function. Equality for nodes created by such constructors is structural: two nodes are the same if they were created by the same constructor with the same parameters.

If *type* is a phylum, then every formal parameter with a phylum type is a *child* parameter. The phylum of any child parameter must satisfy the signature `input PHYLUM[]`, because becoming a child has ramifications for a node (for example, it affects its lexical ordering). When used as a value, a constructor is treated as a procedure. Nodes created by such constructors have *identity*; each instance of a constructor call creates a new node. Equality for such nodes uses node identity. The node created by the constructor is the *parent* for each actual parameter corresponding to a child formal parameter. No child may have more than one parent. The predefined polymorphic pattern `parent` matches a node with a child.

When used as a pattern, a constructor call only matches nodes created by the constructor. The pattern arguments are then matched against the actual arguments to the constructor when the node was created.

A.2.11 Pattern Declarations

A pattern declaration introduces a name in the pattern namespace and has the form:

named ::= ... / `pattern` *name*`(`*value-formals*(A.2.13)`)` `:` *type*(A.4) `:=` *choices*(A.5)`;`

Each pattern in *choices* (a comma-separated sequence of patterns) must bind every name that appears as a formal parameter in the pattern declaration.

A pattern call matches any node that matches one of the choices where the actual parameters are matched against the binding produced by the choice. All possibilities are attempted. However, if used in the context of a `case` clause (Section A.8.4), possibilities are attempted only until a complete match is found.

Pattern declarations may not be mutually recursive, and a recursive pattern declaration must be *bottom-linearly* recursive (see Section 3.3.2).

A.2.12 Renamings

A renaming declaration has one of the forms:

named ::= ... /
  `class` *name* `=` *use* `;` /
  `signature` *name* `=` *use* `;` /
  `module` *name* `=` *use* `;` /
  `type` *name* `=` *use* `;` /
  `pattern` *name* `=` *use*′ `;` /
  name `=` *use*′ `;`
use ::= *name* / *type*(A.4)`$`*name*
use′ ::= *use* / `(`*use* `:` *type*(A.4)`)`

The last form of renaming is a *value* renaming.

A renaming introduces a new name in the respective namespace (a module renaming introduces a name in the type namespace). This name refers to the same entity referred to by the use. Value and pattern renamings must use an explicit typing to disambiguate polymorphic uses (see Section A.7.2).

A.2.13 Value Formals

A value formal parameter has the form:

value-formal ::= *(name/*`_`*)* `:` *type*(A.4)

If the name is given as `_`, the formal parameter cannot be used.

Two or more value formal parameters with the same type may be specified with the form:

value-formal ::= ... / *names* `:` *type*(A.4)

The names must be comma-separated. Value formal parameter declarations are separated by semicolons.

A.2.14 Type Formals

A type formal parameter has the form:

type-formal ::= *[*`phylum`*]* *name* *[*`::` *signatures*(A.3)*]*

Two or more type formal parameters with the same signatures may be written with the form:

type-formal ::= ... / [`phylum`] *names* : : *signatures*(A.3)

The names must be comma-separated. Type formal parameter declarations are separated by semicolons.

A name of a type formal may be used in the signatures of any type formal in the same list of type formals.

A.3 Signatures

A signature may be specified by a use of a named signature (Section A.2.12), or by a class instantiation or fixed signature. Sequences of signatures are separated by commas. It is an error if a sequence of signatures contains two or more services in the same namespace with the same name.

A.3.1 Class Instantiation

A class may be instantiated to yield a signature:

signature ::= ... / [`var`] [`input`] *class*`[`*types*(A.4)`]`

The type actuals are *not* checked for conformance to the type formals of the class; conformance is unneeded since classes do not contain implementations.

If the `var` *capability* is present, var services are available. Likewise if the `input` capability is present, input services are available. Type formals may not be declared with both `var` and `input` for the same class.

A.3.2 Fixed Signatures

A fixed signature is described as a set of types:

signature ::= ... / `{`*types*(A.4)`}`

All the types is the set must be distinct. Any type from the set satisfies the signature and no other type does.

A.4 Types

A type may be specified by a use of a named type or by one of the forms given in this section. The signatures of a type fetched from another type cannot be inferred, but can be checked. Thus a fetched type must be assigned as a computed type with explicit signatures before it can be used to fetch other services. For instance, `T$U$V` is never legal because no signatures can be inferred for `T$U`. This rule prevents a circularity in name resolution.

Fetching an input service from a type *modifies* the type. Types may only be modified in certain situations (the same situations in which procedures may be called).

A.4.1 Module Instantiation

A module may be instantiated only directly in a computed type or phylum declaration (see Sections A.2.4 and A.2.5).

type ::= ... / *(*`type`/`phylum`*)* *name* *[*`::` *signatures]* `:=` *module*`[`*types*`]`*[*`(`*exprs*(A.6)`)`*]*

The actual type parameters are checked for conformance with their respective formal type parameters, and similarly for the value parameters. If the type created by the module extends a formal type parameter, the result of the instantiation extends the corresponding actual type parameter.

A.4.2 Node Reference Types

A type for node references has the following form:

type ::= ... / `remote` *phylum*(A.4)

This type is *not* a phylum but it *is* equal to the phylum for the purposes of type checking, and satisfies the same signatures as the phylum.

A.4.3 Private Types

A private type has the form:

type ::= ... / `private` *type*(A.4)

The private type satisfies the same signatures as *type*, but is not considered equal to any other type.

A.4.4 Function Types

A function type has the form:

type ::= ... /
  `function (`*value-formals*(A.2.13)`)` *[*`:` *type* / `(`*variable-declarations*(A.2.6)`)`*]*

Function types do not satisfy any signatures and may not be passed as actual type parameters.

A procedure is considered to have a function type only for the purpose of an explicitly typed polymorphic use (see Section A.2.12). Attributes and functions have function types with one result type.

A.5 Patterns

A pattern must have one of the forms given in this section. No procedures may be called in expressions occurring in patterns.

A.5.1 Pattern Calls

A pattern call has the form:

pattern ::= ... / *use*(A.2.12)`(`*pattern-actuals*`)`
pattern-actual ::= *pattern* / *name* `:=` *pattern* / `...`

The patterns actuals in the pattern call are separated by commas.

If a pattern actual is a keyword parameter (the second possibility), *name* must be the name of a parameter for the pattern or constructor named in *use*. Furthermore all later pattern actuals must be keyword parameters too, but not all formals must be matched. The third possibility `...` may only be used as the last pattern actual and matches any remaining formals.

A.5.2 Pattern Variables

A pattern variable has the form:

pattern ::= ... / `?`*[name] [*`:` *type*(A.4)*]*

A pattern variable matches any node (of the correct type). If no type is given, the type is inferred from the context. The scope of a pattern variable is the entire pattern in which it occurs and any attached block of declarations (if the pattern variable is in a pattern governed by `match`). A name may not be bound more than once in a pattern.

A.5.3 Conjunctive Patterns

A conjunctive pattern has the form:

pattern ::= ... / *pattern* `&` *pattern*

A conjunctive pattern only matches if both of the subpatterns match. The two subpatterns must have the same type.

A.5.4 Conditional Patterns

A conditional pattern has the form:

pattern ::= ... / *pattern* `if` *expression*(A.6)

This pattern matches only if the pattern matches and the (boolean) expression evaluates to "true."

A.5.5 Value Patterns

A value pattern has the form:

pattern ::= ... / `!`*expression*(A.6)

Value patterns only match a value equal to the one in the expression. A value pattern is sugar for the following form:

`?`*name* `if` *name*`=`*expression*

where *name* is a newly created unique name.

The exclamation point may be omitted from a value pattern with a literal value (see Section A.6.1).

A.5.6 Type-Test Patterns

A type-test pattern has the form:

pattern ::= ... / *pattern* `:?` *type*(A.4)

Here *type* must name a type formal with a fixed signature (Section A.3.2). The type of the pattern must be a type in the set specified in the fixed signature. A type-test pattern only matches if the type of the pattern is equal to the actual type parameter given for the type formal.

A.5.7 Sequence Patterns

A sequence pattern has the form:

pattern ::= ... / *[type*(A.4)`$`*]*`{`*elem-patterns*`}`
elem-pattern ::= *pattern* / ... *[*`and` *pattern]*

Here *elem-patterns* is a comma separated sequence of *elem-pattern*. A pattern used after ... `and` is called a *constraint* pattern.

The type of the sequence pattern (*type* if given, otherwise inferred) must satisfy a signature `READ_ONLY_COLLECTION[`*elem-type*`]` for some type *elem-type*. Each element pattern must have type *elem-type*. The pattern ... used in a sequence pattern matches any number of element patterns, each of which must match the constraint pattern, if one is given. The scope of pattern variables in the constraint pattern is limited to that pattern.

A.6 Expressions

An expression may be a use of a name in the value space, or one of the forms mentioned in this section.

A.6.1 Literals

An expression may take the form of a literal string (surrounded by double quotes), character (surrounded by single quotes), real or integer. The types of such literals are respectively `String`, `Character`, `Real` and `Integer` as predefined in Section A.11.

<table>
<tr><th>Precedence</th><th>Operators</th><th>Associativity</th></tr>
<tr><td>-3</td><td><code>and</code></td><td>left</td></tr>
<tr><td>-2</td><td><code>or</code></td><td>left</td></tr>
<tr><td>-1</td><td><code>in not</code></td><td>non associative</td></tr>
<tr><td>0</td><td><code>..</code></td><td>left</td></tr>
<tr><td>3</td><td><code>&& ||</code></td><td>right</td></tr>
<tr><td>4</td><td><code>/= < <= == > >= << >> <<= >>=</code>
<code>|<| |<=| |>| |>=|</code></td><td>non associative</td></tr>
<tr><td>5</td><td><code>\\ ++</code></td><td>right</td></tr>
<tr><td>6</td><td><code>+ -</code> (all unary operators)</td><td>left</td></tr>
<tr><td>7</td><td><code>* % /</code></td><td>left</td></tr>
<tr><td>8</td><td><code>** ^ ^^ ^^=</code></td><td>right</td></tr>
<tr><td>9</td><td>(all other operators)</td><td>left</td></tr>
</table>

Table A.1: Operators in APS

A.6.2 Function Calls

A function call has the form:

expr ::= . . . / *expr*(*exprs*)

Alternatively, if the function has only one parameter, it may be called using the syntax:

expr ::= . . . / *expr* . *use*(A.2.12)

Functions, procedures, attributes and values with function type may be called. Procedure calls are not permitted in certain constructs.

A.6.3 Operations

Certain function calls can be written using prefix or infix notation. Table A.1 lists all infix operators with their precedence (higher numbers have tighter precedence). The name of the function is the same as the binary operator. Unary operators are prefixed with `#` to give the name. To use an operator as a name, it must be enclosed in parentheses.

A.6.4 Implicit Reductions

If exactly one of the two arguments to a function is a sequence expression (see Section A.6.6), the function call is an implicit reduction over the sequence. The other argument is the base case. Reduction starts from the end of the sequence if the sequence is the first argument, otherwise it starts from the start.

A.6.5 Comprehensions

Comprehensions have the form:

expr ::= ... / *[type*(A.4)`$`*]*`{`*comps*`}`
comp ::= *expr*(A.6) / *seq*(A.6.6)

Each element of the comprehension is either a normal expression or a sequence expressions. The elements are separated with commas.

A comprehension expression creates a value of a type that satisfies a signature `COLLECTION[`*elem-type*`]` for some type *elem-type*. If the context of this expression does not determine the type, it must be given in the comprehension expression. Each element must have the type *elem-type*.

A.6.6 Sequence Expressions

Sequence expressions have one of the following forms:

seq ::=
  expr(A.6)`...` /
  seq(A.6.6) `if` *expr* /
  seq `for` *name [*: *type]* `in` *expr* /
  expr`(`*seqs*`)`

Sequences of sequence expressions are separated by commas.

The first form of sequence expression, a *repeat* sequence, uses all the elements of a collection in sequence. This sequence expression has the element type of the collection as its type. The second form conditionally uses a sequence of values, and has the type of the sequence. The third form, an *explicit map*, uses a sequence expression for each element of a collection. Here *name* is bound for each element in the collection given and is visible in the sequence expression. The type of the explicit map is the type of the *body*, that is, the sequence expression, *seq*. The last form is an *implicit map* and applies the function to elements of the respective sequences. There must be at least one argument, and if there is more than one argument the sequences must be formed from the same collection (see Section 4.3.3 for details). The type of an implicit map is the return type of the function.

A.6.7 Pragma Expressions

Pragma expression include several forms not permitted in normal expressions:

pragma-expr ::=
  `class` *use*(A.2.12) /
  `module` *use* /
  `signature` *use* /
  `type` *use* /
  `pattern` *use'*(A.2.12) /
  expr(A.6)

These forms refer to the declaration so named, and are not affected by replacements (see Section A.7.1).

A.7 Other Top-level Declarations

APS includes other unnamed declarations that may be used in the top-level of a module or file.

A.7.1 Inherit Clauses

An inherit clause takes the form:

unnamed ::= ... /
  `inherit` *module*`[`*types*(A.4)`]`*[*`(`*exprs*(A.6)`)`*]* `begin`
    *top-level**(A.2.1)
  `end;`

The module must be a use of a named module. It may not represent a module fetched from a type.

The inherit clause is replaced by a copy of the module, with some changes. In particular, the names declared in the module are only visible to declarations declared in the inherit clause. The names introduced by these declarations are visible in the context surrounding the inherit clause. Renaming are often used to make an inherited entity visible outside the inherit clause.

The sequence of declarations may include *replacements*. A replacement has one of the forms:

replacement ::=
  `type` *name* `->` *use*(A.2.12)`;` /
  `pattern` *name* `->` *use'*(A.2.12)`;` /
  name `->` *use'*(A.2.12)`;`

Replacements do *not* introduce names; they affect the binding of names in the module being inherited. The name on the left of the replacement must be visible in the module being inherited (that is, declared in the module, or external to the module). Any use of the entity thus named in the module is replaced by the use given in the replacement. Definitions and declarations are not replaced. Neither are uses in pragma calls (Section A.7.4). A value or pattern may be replaced only by another with the same type. A type may be replaced only by another type satisfying the same signatures. Additionally, unless the type is declared local to the module being inherited and is a private type (Section A.4.3) or a new type (Section A.2.4), the type being replaced must be equal to the replaced type.

A module declared locally to the module being inherited may be inherited itself.

A.7.2 Polymorphism

Polymorphic declarations are declared in special polymorphic blocks:

unnamed ::= ... /
  `[`*type-formals*(A.2.14)`]` `begin`
    *declaration**
  `end;`

If there is only one declaration, the `begin` and `end` may be omitted.

Any names introduced in *declaration** are introduced in the context surrounding the polymorphic block. Any use of a name somewhere outside the block is a *polymorphic use* and the context of the use must determine actual type parameters for the formal type parameters of the block. The inferred types must conform to their respective type formals. Only patterns and values may be used polymorphically, all other polymorphic uses are illegal.

If all the type formals satisfy fixed signatures (Section A.3.2) then the block is *finitely polymorphic*, otherwise it is *infinitely polymorphic*. The only declarations that may be nested in an infinitely polymorphic block are other polymorphic blocks, function and pattern declarations, pragma calls, renamings and replacements. Polymorphic blocks nested in infinitely polymorphic blocks are restricted as if they were infinitely polymorphic themselves.

A finitely polymorphic block is instantiated for each sequence of types that satisfy the type formal signatures.

A.7.3 Top-Level Match

A top-level match has the form:

unnamed ::= ... /
  `match` *pattern*(A.5) `begin`
    *declaration**
  `end;`

Top-level matches are currently legal only in modules.

The type of the pattern must be a phylum, and must satisfy the signature `var PHYLUM[]`. A top-level match is instantiated for every node of the phylum that matches the pattern and for every set of bindings that match.

A.7.4 Pragma Call

A pragma call has the form:

unnamed ::= `pragma` *name*`(`*pragma-exprs*(A.6.7)`);`

A pragma may direct optimizations or analysis, but does not change the semantics of a description. A use in a pragma is not affected by replacements (Section A.7.1).

A.8 Clause Declarations

The following types of declarations are not currently permitted in the top-level of modules or files, but they are permitted in match clauses, function declarations and prtocedure declarations.

A.8.1 Definitions

A definition takes the form:

clause ::= ... / *expr*(A.6)$_l$ *(*`:=`*/*`:>`*)* *expr*$_r$`;`

The special variable `_` may be used as the target (l-value) of a definition if the value being assigned is to be ignored. This situation is only useful when the right-hand side has a procedure call or the left-hand side is a sequence expression.

The following forms of assignment may use sequence expressions:

clause ::= ... /
  expr$_1$`,` *expr*$_2$ `for` *name* `in` *c* *(*`:=`*/*`:>`*)* *expr*$_3$ `for` *name*$'$ `in` *c*`,` *expr*$_4$`;` /
  expr$_1$ `for` *name* `in` *c*`,` *expr*$_2$ *(*`:=`*/*`:>`*)* *expr*$_3$`,` *expr*$_4$ `for` *name*$'$ `in` *c*`;`

These forms are bucket-brigade assignments (see Section 4.3.4). Each side of a bucket brigade expression refers to $n + 1$ l-values or r-values, where n is the length of the ordered collection c used on both sides of the assignment. The semantics of the assignment is that each l-value is defined with its corresponding r-value. For example, if the collection `l` consists of the two nodes `n1` and `n2`, then

```
n.index for n in l, count := 0, n.index+1 for n in l;
```

has the effect of

```
n1.index := 0;
n2.index := n1.index+1;
count := n2.index+1;
```

Definitions using the symbol `:>` may only be used for variables as collections (see Section A.2.6). The values specified in all the definitions of such variables are combined with the initial value to give the final value of the variable.

Definitions using the symbol `:=` may only be used for variables *not* declared as collections. The lexically first definition of a variable is used for its value. Other definitions are ignored. It is an error if two definitions arise from the same lexical position. Such an error may arise if the definition is instantiated more than once.

A.8.2 Procedure Call

A procedure call takes the form:

clause ::= ... / *use'*(A.2.12)`(`*exprs*`)`*[*`@(`*exprs*`)`*]*`;`

The optional part is to be omitted if the procedure has no result values.[1]

The first sequence of expressions are actual parameters for the procedure call; the second sequence consists of l-values to be assigned the results of the procedure call.

[1]The current APS compiler only accepts calls to such procedures.

A.8.3 Conditional

A conditional declaration has the form

```
clause ::= ... |
  if expr(A.6) then
    declaration*
  (elsif expr then
    declaration*)*
  [else
    declaration*]
  endif;
```

The conditional selects the set of declarations to execute depending upon the value of the conditional expressions. Each declaration depends monotonically on its guarding condition as an `OrLattice`, and monotonically on all preceding conditions as `AndLattice`'s (see Section 7.2.4).

A.8.4 Pattern Matching

Pattern match clauses take the form:

```
clause ::= ... |
  (case|for) expr(A.6) begin
    (match pattern(A.5) begin
      declaration*
    end;)*
  [else
    declaration*]
  end;
```

An `else` clause is legal only for `case` pattern matches.

A `case` clause selects the declarations to execute depending on the value of the *pattern subject* given as *expr*. At most one sequence of declarations will be instantiated for every instantiation of the `case` clause (if an `else` clause is specified, then exactly one sequence).

A `for` clause instantiates every sequence of declarations with every set of bindings that enables a guarding pattern to match the pattern subject.

A.8.5 For-In Clause

A for-in clause has the form

```
clause ::= ... |
  for name in expr(A.6) begin
    declaration*
  end;
```

The declarations in this clause are instantiated for every element in the collection *expr* (whose type must satisfy the signature `READ_ONLY_COLLECTION[`*elem-type*`]`). The instantiation is monotonic in types satisfying the signature `UNION_LATTICE[`*elem-type*`]`.

A.9 Signature Conformance

Signature conformance is defined for a sequence of type actuals and corresponding type formals. Uses of the type formals in the signatures of the type formals are replaced by their corresponding type actuals. Then the computed signatures of the type actuals are compared to the signatures of the corresponding type formals. All parent signatures for classes (and modules) are recursively added to the respective lists. Then for every class instantiation of a type formal, the corresponding type actual must have the same class instantiated in its signatures with at least the capabilities required by the type formal. Furthermore for every fixed signature of the type formal, the type actual must be equal to one of the types specified, or must itself satisfy a fixed signature, all of whose types must be equal to a type in the fixed signature of the type formal.

A.10 Signature Matching

When a module declaration specifies parent signatures and some of these signatures are not satisfied by the extension type, the body of the module must *match* such signatures. This section defines when a module body matches a signature.

A module can *never* match a fixed signature, only class instantiations. Furthermore, for each parent signature of the class that is not satisfied by the type extensions, the module body must match this signature as well.

In order to match a class instantiation, the module body must declare a public entity for each service required by the class that is available for the capabilities given. For instance, if a class contains a public attribute declaration not declared `input`, then a module satisfying an instantiation of the class need only define a corresponding entity if the var capability is present in the instantiation. The corresponding entity must have the same name and namespace.

Next, entities in the value and pattern namespaces must have the same type as the entities in the class they are matching, and type declarations in the module must have at least the signatures of the types in the class. Furthermore, unless a type in the class is a private or new type, the type in the module must be equal to it. Type checking of values and patterns is done assuming the corresponding type declarations are equal to the types they match. A class with entities in the signature namespace cannot be matched by any (other) module.

Constant entities can only be matched by other constant entities. Input attributes or variables in a class can only be matched in a module claiming to satisfy the signature with input capability, if the corresponding declaration (respectively an attribute or variable declaration) in the module is similarly declared `input`.

Finally, the declarations in the class need to be realized in a compatible way. A function declaration can be matched by another function, a constructor for a new type (not a phylum), an attribute or a variable declaration. The same is true for attribute declarations (although an input attribute in a class instantiated with input capability can only be matched by another attribute declaration, as explained in the preceding paragraph). A procedure declaration can be matched by any value declaration (of the correct type, of course). A pattern declaration can be matched by a pattern or constructor declaration. A constructor declaration can only be matched by another constructor declaration. A variable declaration can only be matched by another variable declaration.

A.11 Predefined Entities

The following file of predefined entities is made visible to every APS program

```
--- basic definitions
--- to be included in every APS program


class NULL[];

class BASIC[] :: NULL[] begin
  function equal(_,_ : Result) : Boolean;
end;

-- The function's = and /= are defined polymorphically using equal
[T :: BASIC[]] begin
  (=) = T$equal;
  function (/=)(x,y : T) : Boolean := not T$equal(x,y);
end;

class PRINTABLE[] begin
  function string(_ : Result) : String;
end;

class COMPARABLE[] :: BASIC[] begin
  function less(_,_ : Result) : Boolean;
  function less_equal(_,_ : Result) : Boolean;
end;

[T :: COMPARABLE[]] begin
  (<) = T$less;
  (<=) = T$less_equal;
  function (>)(x,y : T) : Boolean := T$less(y,x);
  function (>=)(x,y : T) : Boolean := T$less_equal(y,x);
end;

-- for two elements of an ordered type,
```

```
-- they are equal or one is less than the other.
-- Integers are ORDERED, sets are not.
class ORDERED[] :: COMPARABLE[];

class NUMERIC[] :: BASIC[] begin
  zero : Result;
  one : Result;
  function plus(_,_ : Result) : Result;
  function minus(_,_ : Result) : Result;
  function times(_,_ : Result) : Result;
  function divide(_,_ : Result) : Result;
  -- unary operators:
  function unary_plus(_ : Result) : Result;
  function unary_minus(_ : Result) : Result;
  function unary_times(_ : Result) : Result;
  function unary_divide(_ : Result) : Result;
end;

-- define infix operations polymorphically:
[T :: NUMERIC[]] begin
  (+) = T$plus;
  (-) = T$minus;
  (*) = T$times;
  (/) = T$divide;
  (#+) = T$unary_plus;
  (#-) = T$unary_minus;
  (#*) = T$unary_times;
  (#/) = T$unary_divide;
end;

[T :: NUMERIC[],ORDERED[]] function abs(x : T) : T begin
  if x < T$zero then
    result := -x;
  else
    result := x;
  endif;
end;


--- Normal objects

private module NULL_TYPE[] :: NULL[];

-- this module must be ready to be instantiated when this file is loaded.
```

```
module TYPE[] :: BASIC[], PRINTABLE[] extends NULL_TYPE[] begin
  function assert(_ : Result);
  function equal(x,y : Result) : Boolean := node_equivalent(x,y);
  function node_equivalent(x,y : Result) : Boolean;
  function string(_ : Result) : String;
end;

private type SampleType := TYPE[];


--- Concrete Types

-- Boolean is a very special predefined type
private module BOOLEAN[] :: BASIC[],PRINTABLE[] begin
  function assert(_:Result);
  function equal(x,y : Result) : Boolean;
  function string(x : Result) : String;
end;

type Boolean :: BASIC[],PRINTABLE[] := BOOLEAN[];

true : Boolean;
false : Boolean;

function (and)(_,_ : Boolean) : Boolean;
function (or)(_,_ : Boolean) : Boolean;
function (not)(_ : Boolean) : Boolean;


private module INTEGER[] :: NUMERIC[],ORDERED[],PRINTABLE[] begin
  function assert(_:Result);
  zero : Result;
  one : Result;
  function equal(x,y : Result) : Boolean;
  function less(x,y : Result) : Boolean;
  function less_equal(x,y : Result) : Boolean;
  function plus(x,y : Result) : Result;
  function minus(x,y : Result) : Result;
  function times(x,y : Result) : Result;
  function divide(x,y : Result) : Result;
  function unary_plus(x : Result) : Result := x;
  function unary_minus(x : Result) : Result;
  function unary_times(x : Result) : Result := x;
  function unary_divide(x : Result) : Result;
```

```
  function string(x : Result) : String;
end;

type Integer :: NUMERIC[],ORDERED[],PRINTABLE[] := INTEGER[];

-- some useful functions:
function lognot(x : Integer) : Integer;
function logior(x,y : Integer) : Integer;
function logand(x,y : Integer) : Integer;
function logandc2(x,y : Integer) : Integer;
function logxor(x,y : Integer) : Integer;
function logbitp(index,set : Integer) : Integer;
function ash(n,count : Integer) : Integer;
function odd(_ : Integer) : Boolean;

[T :: NUMERIC[]] function (^)(x : T; y : Integer) : T begin
  if y = 0 then
    result := T$one;
  elsif y = 1 then
    result := x;
  elsif odd(y) then
    result := x * (x ^ (y - 1));
  else
    result := (x*x)^(y/2);
  endif;
end;


class REAL[] :: NUMERIC[],ORDERED[] begin
  function from_integer(_ : Integer) : Result;
  function to_integer(_ : Result) : Integer;
end;

private module IEEE[] :: REAL[],PRINTABLE[] begin
  function assert(_:Result);
  zero : Result;
  one : Result;
  max : Result; -- maximum positive value
  min : Result; -- minimum positive value
  function equal(x,y : Result) : Boolean;
  function less(x,y : Result) : Boolean;
  function less_equal(x,y : Result) : Boolean;
  function plus(x,y : Result) : Result;
  function minus(x,y : Result) : Result;
```

```
  function times(x,y : Result) : Result;
  function divide(x,y : Result) : Result;
  function unary_plus(x : Result) : Result := x;
  function unary_minus(x : Result) : Result;
  function unary_times(x : Result) : Result := x;
  function unary_divide(x : Result) : Result;
  function from_integer(y : Integer) : Result;
  function to_integer(x : Result) : Integer;
  function string(x : Result) : String;
end;

type IEEEdouble := IEEE[];
type IEEEsingle := IEEE[];
function IEEEwiden(x : IEEEsingle) : IEEEdouble;
function IEEEnarrow(x : IEEEdouble) : IEEEsingle;

type Real = IEEEdouble; -- a pseudonym

private module CHARACTER[] :: ORDERED[],PRINTABLE[] begin
  function assert(_ : Result);
  function equal(x,y : Result) : Boolean;
  function less(x,y : Result) : Boolean;
  function less_equal(x,y : Result) : Boolean;
  function string(x : Result) : String;
end;

type Character :: ORDERED[],PRINTABLE[] := CHARACTER[];

function char_code(x : Character) : Integer;
function int_char(x : Integer) : Character;
tab : Character := int_char(9);
newline : Character := int_char(10);


--- Phylum objects

private module NULL_PHYLUM[] :: NULL[] phylum;

module PHYLUM[] :: BASIC[], PRINTABLE[] phylum extends NULL_PHYLUM[] begin
  -- the version for TYPE is superseded
  function assert(_ : Result);
  --- primitive functions for comparing objects
  ---   (object identity is used.)
  function identical(_,_ : Result) : Boolean;
```

```
  equal = identical;
  function string(_:Result) : String;
  -- every object has an identity as an integer
  function object_id(_:Result) : Integer;
  function object_id_less(_,_:Result) : Boolean;

  -- comparisons are possible between nodes of different phyla:
  [phylum Other] begin
    -- in order to handle <<, we need to implicitly  define an attribute
    -- threaded through everything.  Therefore they are "var function"s
    var function precedes(x : Result; y : Other) : Boolean;
    var function precedes_equal(x : Result; y : Other) : Boolean;
    pragma dynamic(precedes,precedes_equal);

    -- the general parent pattern:
    pattern parent(y : Other) : Result;
    -- matches the result or any descendant
    pattern ancestor(y : Other) : Result;

    -- this function is true if x is an ancestor of y but not equal
    function ancestor(x : Result; y : Other) : Boolean;
    -- this function is true if x is an ancestor of y
    function ancestor_equal(x : Result; y : Other) : Boolean;
  end;

  nil : remote Result;
end;

[T :: PHYLUM[]] begin
  -- the contents of PHYLUM are hand exported:
  (==) = T$identical;
  (##) = T$object_id;
  (<#) = T$object_id_less;
  [phylum Other] begin
    pattern parent = T$parent : function(_:Other) : T;
    pattern ancestor = T$ancestor : function(_:Other) : T;
    (^^) = (T$ancestor : function (_:T;_:Other) : Boolean);
    (^^=) = (T$ancestor_equal : function (_:T;_:Other) : Boolean);
  end;
  nil = T$nil;

  -- some variants are defined:
  function (/==)(x,y : T) : Boolean := not T$identical(x,y);
end;
```

```
[T,U :: var PHYLUM[]] begin
  function (<<)(x : T; y : U) : Boolean := T$precedes(x,y);
  function (<<=)(x : T; y : U) : Boolean := T$precedes_equal(x,y);
  function (>>)(x : T; y : U) : Boolean := U$precedes(y,x);
  function (>>=)(x : T; y : U) : Boolean := U$precedes_equal(y,x);
end;

private phylum SamplePhylum := PHYLUM[];


--- COMBINABLE, COMPLETE_PARTIAL_ORDER, and LATTICE
--- used for collection, circular and
--- circular collection attributes respectively.

class COMBINABLE[] begin
  initial : Result;
  function combine(_,_ : Result) : Result;
end;

class COMPLETE_PARTIAL_ORDER[] :: BASIC[] begin
  bottom : Result;
  function compare(_,_ : Result) : Boolean;
  function compare_equal(_,_ : Result) : Boolean;
end;

-- Sometimes the ordering relations coincides with
-- the natural <, <= relations, but sometimes, it goes
-- in the reverse order, so we use a different name and
-- different symbols.

[T :: COMPLETE_PARTIAL_ORDER[]] begin
  (|<|) = T$compare;
  (|<=|) = T$compare_equal;
  function (|>|)(x,y : T) : Boolean := T$compare(y,x);
  function (|>=|)(x,y : T) : Boolean := T$compare_equal(y,x);
end;

class LATTICE[] :: COMPLETE_PARTIAL_ORDER[] begin
  function join(_,_ : Result) : Result;
  function meet(_,_ : Result) : Result; -- not strictly necessary
end;

[T :: LATTICE[]] begin
```

```
  (|\/|) = T$join;
  (|/\|) = T$meet;
end;

module MAKE_LATTICE[T :: BASIC[]](default : T;
                                  comparef,compare_equalf
                                      : function(_,_:T) : Boolean;
                                  joinf,meetf : function (_,_:T) : T)
    :: COMBINABLE[], LATTICE[] Lattice extends T
begin
  -- Lattices are also convenient as types for collection attributes:
  initial = default;
  combine = joinf;

  bottom = default;
  compare = comparef;
  compare_equal = compare_equalf;
  join = joinf;
  meet = meetf;
end;

private function cand(x,y : Boolean) : Boolean := (not x and y);
private function implies(x,y : Boolean) : Boolean := (not x or y);
private function andc(x,y : Boolean) : Boolean := (x and not y);
private function revimplies(x,y : Boolean) : Boolean := (x or not y);

type OrLattice := MAKE_LATTICE[Boolean](false,cand,implies,(or),(and));
type AndLattice := MAKE_LATTICE[Boolean](true,andc,revimplies,(and),(or));

[T :: ORDERED[]] begin
  function max(x,y : T) : T begin
    if x > y then
      result := x;
    else
      result := y;
    endif;
  end;
  function min(x,y : T) : T begin
    if x < y then
      result := x;
    else
      result := y;
    endif;
  end;
```

```
end;

module MAX_LATTICE[T :: ORDERED[]](min_element : T)
    MaxLattice := MAKE_LATTICE[T](min_element,(<),(<=),max,min);
module MIN_LATTICE[T :: ORDERED[]](max_element : T)
    MinLattice := MAKE_LATTICE[T](max_element,(>),(>=),min,max);

--- various types of collections:

class READ_ONLY_COLLECTION[ElemType] begin
  procedure {.}(_ : ElemType...) : Result;
  pattern {.}(_ : ElemType...) : Result;
  pattern append(_,_ : Result) : Result;
  pattern single(_ : ElemType) : Result;
  pattern none() : Result;
  function member(x : ElemType; l : Result) : Boolean;
end;

class COLLECTION[ElemType] :: READ_ONLY_COLLECTION[ElemType] begin
  function append(l1,l2 : Result) : Result;
  function single(x : ElemType) : Result;
  function none() : Result;
  -- eventually the following declaration will go:
  function {.}(_ : ElemType...) : Result;
end;

[ElemType; T :: READ_ONLY_COLLECTION[ElemType]] begin
  pattern {.} = T${.};
  member = T$member;
  -- alternate formulation as an infix operator:
  (in) = T$member;
end;

[ElemType; T :: COLLECTION[ElemType]] begin
  {.} = T${.};
end;


class READ_ONLY_ORDERED_COLLECTION[ElemType]
    :: READ_ONLY_COLLECTION[ElemType]
begin
  function nth(_ : Integer; _ : Result) : ElemType; -- starting at zero
  function nth_from_end(_ : Integer; _ : Result) : ElemType;
  function position(_ : ElemType; _ : Result) : Integer;
```

```
  function position_from_end(_ : ElemType; _ : Result) : Integer;
end;

class ORDERED_COLLECTION[ElemType]
    :: READ_ONLY_ORDERED_COLLECTION[ElemType], COLLECTION[ElemType]
begin
  function subseq(_ : Result; _,_ : Integer) : Result;
  function subseq_from_end(_ : Result; _,_ : Integer) : Result;
  function butsubseq(_ : Result; _,_ : Integer) : Result;
  function butsubseq_from_end(_ : Result; _,_ : Integer) : Result;
end;

[E;T :: READ_ONLY_ORDERED_COLLECTION[E]] begin
  nth = T$nth;
  nth_from_end = T$nth_from_end;
  position = T$position;
  position_from_end = T$position_from_end;
  function first(x : T) : E := T$nth(0,x);
  function last(x : T) : E := T$nth_from_end(0,x);
end;
[E;T :: ORDERED_COLLECTION[E]] begin
  subseq = T$subseq;
  subseq_from_end = T$subseq_from_end;
  butsubseq = T$butsubseq;
  butsubseq_from_end = T$butsubseq_from_end;
  function firstn(n : Integer; x : T) : T := T$subseq(x,0,n);
  function lastn(n : Integer; x : T) : T := T$subseq_from_end(x,0,n);
  function butfirst(x : T) : T := T$butsubseq(x,0,1);
  function butlast(x : T) : T := T$butsubseq_from_end(x,0,1);
  function butfirstn(n : Integer; x : T) : T := T$butsubseq(x,0,n);
  function butlastn(n : Integer; x : T) : T := T$butsubseq_from_end(x,0,n);
  function butnth(n : Integer; x : T) : T := T$butsubseq(x,n,n+1);
  function butnth_from_end(n : Integer; x : T) : T :=
      T$butsubseq_from_end(x,n,n+1);
end;

-- Sequences may be balanced.
-- They are not COMBINABLE because they are ordered.
module SEQUENCE[phylum ElemType :: input PHYLUM[], BASIC[]]
    :: READ_ONLY_ORDERED_COLLECTION[ElemType], var input PHYLUM[]
    phylum
begin
  -- the version for PHYLUM is superseded
  function assert(_ : Result);
```

```
  procedure {.}(l : ElemType...) : Result;
  pragma modifies({.},type Result);
  pattern {.}(l : ElemType...) : Result;
  function nth(i : Integer; l : Result) : ElemType;
  function nth_from_end(i : Integer; l : Result) : ElemType;
  function position(x : ElemType; l : Result) : Integer;
  function position_from_end(x : ElemType; l : Result) : Integer;
  function member(x : ElemType; l : Result) : Boolean;
  -- NB: the following are used in aps-boot-compiler:
  constructor append(l1,l2 : Result) : Result;
  constructor single(x : ElemType) : Result;
  constructor none() : Result;
end;

-- Bags may be in any order whatsoever
module BAG[ElemType :: BASIC[]] :: COLLECTION[ElemType],COMBINABLE[] begin
  -- the version for TYPE is superseded
  function assert(_ : Result);
  function {.}(l : ElemType...) : Result;
  pattern {.}(l : ElemType...) : Result;
  function member(e : ElemType; l : Result) : Boolean;
  constructor append(l1,l2 : Result) : Result;
  constructor single(x : ElemType) : Result;
  constructor none() : Result;
  initial : Result := none();
  function combine(l1,l2 : Result) : Result; -- can do things.
end;

class CONCATENATING[] begin
  function concatenate(_,_ : Result) : Result;
end;
[T :: CONCATENATING[]] (++) = T$concatenate;

module LIST[ElemType :: BASIC[]] :: BASIC[],CONCATENATING[],
                                    ORDERED_COLLECTION[ElemType]
begin
  -- the version for TYPE is superseded
  function assert(_ : Result);
  function cons(x : ElemType; l : Result) : Result;
  constructor single(x : ElemType) : Result;
  constructor append(l1,l2 : Result) : Result;
  constructor none() : Result;

  function equal(l1,l2 : Result) : Boolean;
```

```
  concatenate = append;
  function member(x : ElemType; l : Result) : Boolean;

  function nth(i : Integer; l : Result) : ElemType;
  function nth_from_end(i : Integer; l : Result) : ElemType;
  function position(x : ElemType; l : Result) : Integer;
  function position_from_end(x : ElemType; l : Result) : Integer;
  function subseq(l : Result; start,finish : Integer) : Result;
  function subseq_from_end(l : Result; start,finish : Integer) : Result;
  function butsubseq(l : Result; start,finish : Integer) : Result;
  function butsubseq_from_end(l : Result; start,finish : Integer) : Result;

  function {.}(l : ElemType...) : Result;
  pattern {.}(l : ElemType...) : Result;
end;

class ABSTRACT_SET[ElemType]
begin
  function member(x : ElemType; l : Result) : Boolean;

  function union(_,_ : Result) : Result;
  function intersect(_,_ : Result) : Result;
  function difference(_,_ : Result) : Result;
end;
[E;T :: ABSTRACT_SET[E]] begin
  (\/) = T$union;
  (/\) = T$intersect;
  (/\~) = T$difference;
end;
[E;T :: ABSTRACT_SET[E],COLLECTION[E]] begin
  function (\)(x : T; elem : E) : T := T$difference(x,{elem});
end;

module SET[ElemType :: BASIC[]](test : function(x,y : ElemType) : Boolean)
    :: BASIC[], COMPARABLE[], COLLECTION[ElemType], ABSTRACT_SET[ElemType],
       COMBINABLE[]
    := private BAG[ElemType]
begin
  function equal(_,_ : Result) : Boolean;
  function less(_,_ : Result) : Boolean;
  function less_equal(_,_ : Result) : Boolean;

  function {.}(_ : ElemType...) : Result;
  pattern {.}(_ : ElemType...) : Result;
```

```
  function member(x : ElemType; l : Result) : Boolean;

  function union(_,_ : Result) : Result;
  function intersect(_,_ : Result) : Result;
  function difference(_,_ : Result) : Result;

  -- for collection attributes (use a different method than BAG)
  function combine(x,y : Result) : Result := union(x,y);
end;

module MULTISET[ElemType :: BASIC[]]
    (test : function(x,y : ElemType) : Boolean)
    :: BASIC[], COMPARABLE[], COLLECTION[ElemType], ABSTRACT_SET[ElemType],
       COMBINABLE[]
    -- NB: MULTISETs are not useful for circular attributes
    := private BAG[ElemType]
begin
  function equal(_,_ : Result) : Boolean;
  function less(_,_ : Result) : Boolean;
  function less_equal(_,_ : Result) : Boolean;

  function {.}(_ : ElemType...) : Result;
  pattern {.}(_ : ElemType...) : Result;
  function member(x : ElemType; l : Result) : Boolean;
  function count(x : ElemType; l : Result) : Integer;

  function union(_,_ : Result) : Result;
  function intersect(_,_ : Result) : Result;
  function difference(_,_ : Result) : Result;

  -- for collection attributes (use a different method than BAG)
  function combine(x,y : Result) : Result := union(x,y);
end;

module ORDERED_SET[ElemType :: BASIC[]]
    (test : function(x,y : ElemType) : Boolean;
     compare : function(x,y : ElemType) : Boolean)
    :: ORDERED_COLLECTION[ElemType] -- and SET[ElemType] too
    := private SET[ElemType](test)
begin
  -- equality can be checked cheaper than for SET
  function equal(_,_ : Result) : Boolean;
  function less(_,_ : Result) : Boolean;
  function less_equal(_,_ : Result) : Boolean;
```

```
  -- combined differently than for unordered SET's
  function {.}(_ : ElemType...) : Result;
  -- member can be inherited

  -- new operation functions:
  function union(_,_ : Result) : Result;
  function intersect(_,_ : Result) : Result;
  function difference(_,_ : Result) : Result;

  -- can be used for collection attributes:
  function combine(x,y : Result) : Result := union(x,y);

  -- ORDERED_SETs have ordering functions:
  function nth(i : Integer; l : Result) : ElemType;
  function nth_from_end(i : Integer; l : Result) : ElemType;
  function position(x : ElemType; l : Result) : Integer;
  function position_from_end(x : ElemType; l : Result) : Integer;
  function subseq(l : Result; start,finish : Integer) : Result;
  function subseq_from_end(l : Result; start,finish : Integer) : Result;
  function butsubseq(l : Result; start,finish : Integer) : Result;
  function butsubseq_from_end(l : Result; start,finish : Integer) : Result;
end;

module ORDERED_MULTISET[ElemType :: BASIC[]]
    (test : function(x,y : ElemType) : Boolean;
     compare : function(x,y : ElemType) : Boolean)
    :: ORDERED_COLLECTION[ElemType] -- and ABSTRACT_SET[ElemType] too
    := private MULTISET[ElemType](test)
begin
  -- equality can be checked cheaper than for MULTISET
  function equal(_,_ : Result) : Boolean;
  function less(_,_ : Result) : Boolean;
  function less_equal(_,_ : Result) : Boolean;

  -- combined differently than for unordered SET's
  function {.}(_ : ElemType...) : Result;
  -- member and count can be inherited

  -- new operation functions:
  function union(_,_ : Result) : Result;
  function intersect(_,_ : Result) : Result;
  function difference(_,_ : Result) : Result;
```

```
  -- can be used for collection attributes:
  function combine(x,y : Result) : Result := union(x,y);

  -- ORDERED_MULTISETs have ordering functions:
  function nth(i : Integer; l : Result) : ElemType;
  function nth_from_end(i : Integer; l : Result) : ElemType;
  function position(x : ElemType; l : Result) : Integer;
  function position_from_end(x : ElemType; l : Result) : Integer;
  function subseq(l : Result; start,finish : Integer) : Result;
  function subseq_from_end(l : Result; start,finish : Integer) : Result;
  function butsubseq(l : Result; start,finish : Integer) : Result;
  function butsubseq_from_end(l : Result; start,finish : Integer) : Result;
end;

module UNION_LATTICE[ElemType;T :: SET[ElemType]]
    UnionLattice := MAKE_LATTICE[T]({},(<),(<=),(\/),(/\));
module INTERSECTION_LATTICE[ElemType;T :: SET[ElemType]](universe : T)
    IntersectionLattice := MAKE_LATTICE[T](universe,(>),(>=),(/\),(\/));

--- I may add a primitive pair constructor that
--- is structurally typed.
module PAIR[T1,T2 :: BASIC[]] begin
  constructor pair(x:T1;y:T2) : Result;
  function fst(p : Result) : T1 begin
    case p begin
      match pair(?x,?) begin
        result := x;
      end;
    end;
  end;
  function snd(p : Result) : T2 begin
    case p begin
      match pair(?,?y) begin
        result := y;
      end;
    end;
  end;
end;

module STRING[] :: ORDERED[], PRINTABLE[] := LIST[Character] begin
  function less(x,y : Result) : Boolean;
  function less_equal(x,y : Result) : Boolean;
  function string(x : Result) : String;
end;
```

```
type String := STRING[];

[T,U :: PRINTABLE[]] function (||)(x : T; y : U) : String
    := T$string(x) ++ U$string(y);

type Range := LIST[Integer];
function (..)(x,y : Integer) : Range begin
  if x = y then
    result := {x};
  elsif x > y then
    result := {};
  else
    mid : Integer := (x+y)/2;
    result := (x..mid) ++ ((mid+1)..y);
  endif;
end;

[ElemType; T :: READ_ONLY_COLLECTION[ElemType]] begin
  function length(l : T) : Integer := 0+(1 for _ in l);
end;
```

Appendix B

A Compiler for Oberon2

This section consists of an Oberon2 compiler written in APS. It translates Oberon2 (in concrete syntax tree form) into a formalized version of the GCC tree intermediate representation.

B.1 Abstract Syntax

```
module OBERON2_TREE[] begin
  phylum Program;
  phylum Block;
  phylum Declaration;
  phylum Header;
  phylum Receiver;
  phylum Type;
  phylum Statement;
  phylum Case;
  phylum CaseLabel;
  phylum Expression;
  phylum Operator;
  phylum Element;

  phylum IdentDef;
  phylum ExportInfo;
  phylum Use;
  --phylum ModuleUse;

  signature PHYLA := {Program,Block,Declaration,Header,Receiver,
                      Type,Statement,Case,CaseLabel,Expression,Operator,
                      Element,IdentDef,ExportInfo,Use,ModuleUse},
                     var PHYLUM[];

  phylum Modules:=SEQUENCE[Declaration];
  phylum Declarations:=SEQUENCE[Declaration];
  phylum Formals:=SEQUENCE[Declaration];
  phylum Fields:=SEQUENCE[Declaration];
```

```
phylum Statements:=SEQUENCE[Statement];
phylum Cases:=SEQUENCE[Case];
phylum CaseLabels:=SEQUENCE[CaseLabel];
phylum Actuals:=SEQUENCE[Expression];
phylum Elements:=SEQUENCE[Element];

signature SEQ_PHYLA := {Modules,Declarations,Formals,Fields,Statements,
                        Cases,CaseLabels,Actuals,Elements}, var PHYLUM[];

type Constant := OBERON2_CONSTANT[];
type Constants = Constant$Constants;

constructor program(modules : Modules) : Program;
constructor module_decl(name : IdentDef; body : Block) : Declaration;
constructor block(decls : Declarations; stmts : Statements) : Block;
constructor no_block() : Block;

constructor import(name : IdentDef; from : Use) : Declaration;
constructor const_decl(name : IdentDef; value : Expression) : Declaration;
constructor type_decl(name : IdentDef; value : Type) : Declaration;
constructor var_decl(name : IdentDef; shape : Type) : Declaration;
constructor forward(header  : Header) : Declaration;
constructor proc_decl(header : Header; body : Block) : Declaration;

constructor header(name : IdentDef;
                   receiver : Receiver;
                   formals : Formals;
                   result : Type) : Header;

constructor receiver(formal : Declaration) : Receiver;
constructor no_receiver() : Receiver;


-- note that formals are separated and new type identifiers
-- introduced if necessary (they aren't open_array types, for example)
constructor value_formal(name : IdentDef; shape : Type) : Declaration;
constructor var_formal(name : IdentDef; shape : Type) : Declaration;
pattern formal(name : IdentDef; shape : Type) : Declaration
    := value_formal(?name,?shape),var_formal(?name,?shape);
-- for the builtin procedures with optional parameters
constructor opt_formal(shape : Type; default : Constant)
    : Declaration;
-- optional parameters of unlimited number and type
-- check in different ways
constructor rest_formal(shape : Type) : Declaration;

constructor field(name : IdentDef; shape : Type) : Declaration;

constructor no_decl() : Declaration;
```

```
pattern declaration(name : IdentDef) : Declaration :=
    module_decl(?name,?),import(?name,?),
    const_decl(?name,?),type_decl(?name,?),var_decl(?name,?),
    forward(header(?name,?,?,?)),proc_decl(header(?name,?,?,?),?),
    formal(?name,?),field(?name,?);

-- a shorthand for getting at the receiver formal:
pattern bound_proc_decl(name : IdentDef; formal : Declaration) : Declaration
    := proc_decl(header:=header(name:=?name,
                                receiver:=receiver(formal:=?formal))),
       forward(header:=header(name:=?name,
                              receiver:=receiver(formal:=?formal)));

constructor no_type() : Type;
constructor named_type(using : Use) : Type;
constructor fixed_array_type(length : Expression; element_type : Type)
    : Type;
constructor open_array_type(element_type : Type) : Type;
constructor record_type(extending : Type; fields : Fields) : Type;
constructor pointer_type(to : Type) : Type;
constructor proc_type(header : Header) : Type;
-- basic types
constructor boolean_type() : Type;
constructor char_type() : Type;
constructor shortint_type() : Type;
constructor integer_type() : Type;
constructor longint_type() : Type;
constructor real_type() : Type;
constructor longreal_type() : Type;

pattern array_type(element_type : Type) : Type
    := open_array_type(?element_type),fixed_array_type(?,?element_type);

pattern basic_type() : Type
    := boolean_type(),char_type(),shortint_type(),integer_type(),
      longint_type(),real_type(),longreal_type();

pattern numeric_type() : Type
    := shortint_type(),integer_type(),longint_type(),
      real_type(),longreal_type();

constructor set_type() : Type;
constructor nil_type() : Type; -- the type of NIL

-- pseduo-types used for builtin procedures and functions:
constructor abs_type() : Type;
constructor long_type() : Type;
constructor max_type() : Type;
constructor short_type() : Type;
constructor type_type() : Type;
```

```
constructor any_type() : Type;

pattern pseudo_type() : Type
    := abs_type(),long_type(),max_type(),short_type(),type_type(),any_type();

-- default type for things computing types.
any : Type := any_type();
boolean : Type := boolean_type();
char : Type := char_type();
shortint : Type := shortint_type();
integer : Type := integer_type();
longint : Type := longint_type();
real : Type := real_type();
longreal : Type := longreal_type();
string : Type := open_array_type(char);
set : Type := set_type();
a_type : Type := type_type();

constructor assign_stmt(lhs : Expression; rhs : Expression) : Statement;
constructor call_stmt(call : Expression) : Statement;
constructor if_stmt(guard : Expression;
                    (then) : Statements;
                    (else) : Statements) : Statement;
constructor case_stmt(expr : Expression;
                      cases : Cases;
                      (else) : Statements) : Statement;
constructor while_stmt(guard : Expression;
                       body : Statements) : Statement;
constructor repeat_stmt(body : Statements; guard : Expression) : Statement;
constructor for_stmt(variable : Expression; -- just an ident
                     start, finish, step : Expression;
                     body : Statements) : Statement;
constructor loop_stmt(body : Statements) : Statement;
constructor with_stmt(variable : Expression; -- just a qualident
                      guard_type : Type;
                      body : Statements;
                      (else) : Statements) : Statement;
constructor exit_stmt() : Statement;
constructor return_stmt(value : Expression) : Statement;

constructor case_clause(labels : CaseLabels; body : Statements) : Case;

constructor single_label(value : Expression) : CaseLabel;
constructor range_label(start,finish : Expression) : CaseLabel;

constructor no_expr() : Expression;
constructor named_expr(using : Use) : Expression;
constructor unop(op : Operator; arg : Expression) : Expression;
constructor binop(op : Operator; arg1, arg2 : Expression) : Expression;
constructor funcall(func : Expression; actuals : Actuals) : Expression;
```

```
constructor is_test(value : Expression; test_type : Type) : Expression;
-- this node doesn't appear until after type checking, because
-- it is syntactically undistinguishable from a procedure call
constructor type_guard(value : Expression; guard_type : Type) : Expression;
-- multiple dimension array sugar removed before this point:
constructor aref(array : Expression; index : Expression) : Expression;
constructor fref(record : Expression; field : Use; super : Boolean)
    : Expression;
constructor fetch(pointer : Expression) : Expression;
constructor set_expr(elements : Elements) : Expression;
constructor constant_expression(value : Constant) : Expression;

constructor log_or() : Operator;
constructor log_and() : Operator;
constructor log_not() : Operator;
constructor plus() : Operator;
constructor minus() : Operator;
constructor times() : Operator;
constructor divide() : Operator;
constructor mod() : Operator;
constructor div() : Operator;
constructor equal() : Operator;
constructor not_equal() : Operator;
constructor less() : Operator;
constructor less_equal() : Operator;
constructor greater() : Operator;
constructor greater_equal() : Operator;
constructor in_set() : Operator;
-- type test parses as its own type of expression.

pattern logical_operator() : Operator := log_or(),log_and(),log_not();
pattern integer_operator() : Operator := mod(),div();
pattern arithmetic_operator() : Operator :=
    plus(),minus(),times(),divide(),integer_operator();
pattern equality_operator() : Operator := equal(),not_equal();
pattern comparison_operator() : Operator :=
    equality_operator(),less(),less_equal(),greater(),greater_equal();
-- predicates are things returning boolean values:
pattern predicate_operator() : Operator :=
    logical_operator(), comparison_operator(), in_set();
-- NB: predicate_operator and arithmetic_operator between them
-- cover the space of operators.

constructor single_element(expr : Expression) : Element;
constructor range_element(start,finish : Expression) : Element;

constructor identifier(name : Symbol; export_info : ExportInfo) : IdentDef;
constructor ignore() : IdentDef;

constructor not_exported() : ExportInfo;
```

```
  constructor readonly_exported() : ExportInfo;
  constructor exported() : ExportInfo;

  constructor use_name(name : Symbol) : Use;
  -- qualified nodes rarely appear in expressions at first because
  --   we can't distinguish a qualified identifier from a field reference
  constructor qualified(from_module : Use; using : Use) : Use;

  [T :: {Expression,Use}] pattern possibly_qualified(mod, u : Use) : T
      := qualified(?mod,?u) :? T, fref(named_expr(?mod),?u,!false) :? T;
end;
```

B.1.1 Constants

```
module OBERON2_CONSTANT[] :: NUMERIC[], COMPARABLE[]
    Constant
begin
  constructor shortint_constant(value : Integer) : Constant;
  constructor integer_constant(value : Integer) : Constant;
  constructor longint_constant(value : Integer) : Constant;
  constructor real_constant(value : IEEEsingle) : Constant;
  constructor longreal_constant(value : IEEEdouble) : Constant;

  constructor set_constant(representation : Integer) : Constant;

  constructor boolean_constant(value : Boolean) : Constant;

  constructor char_constant(value : Character) : Constant;
  constructor string_constant(value : String) : Constant;

  constructor undefined() : Constant;

  nil : Constant := undefined();


  -- since we're using infinite precision integers for all integer
  -- constants, we might as well make it easier to use them factored!
  pattern some_integer_constant(x : Integer) : Constant
      := shortint_constant(?x),integer_constant(?x),longint_constant(?x);

  type Constants := LIST[Constant];

  -- this has to work for sets too, but we can fake it
  -- by having a special coercion function for it.
  zero : Constant := shortint_constant(0);
  -- one is only to satisfy NUMERIC
  one : Constant := shortint_constant(1);


  --- Set creation functions
```

```
-- special functions to make it easier to use sets:

function make_set(l : Constants) : Constant begin
  collection rep : Integer :> 0,logior;
  collection some_nil : Boolean :> false,(or);
  for x : Constant in l begin
    case x begin
      match undefined() begin some_nil :> true; end;
      match some_integer_constant(?n) begin
        -- set the appropriate bit in the result
        -- we are using arbitrary precision integers
        -- so there is no problem with overflow, just massive
        -- space consumption
        rep :> ash(1,n);
      end;
    end;
  end;
  if some_nil then
    result := nil;
  else
    result := set_constant(rep);
  endif;
end;
function make_range(x,y : Constant) : Constants begin
  case Constants${x,y} begin
    match {some_integer_constant(?v1),
           some_integer_constant(?v2)} begin
      result := {shortint_constant(i) for i : Integer in v1..v2};
    end;
  else
    result := {nil}; -- force make_set to return nil
  end;
end;


--- arithmetic functions
-- (We use the builtin names so that we can overload the builtin operators)

function plus(x,y : Constant) : Constant
    := do_op(logior,(or),IEEEdouble$plus,IEEEsingle$plus,Integer$plus,x,y);

function minus(x,y : Constant) : Constant
    := do_op(logandc2,andc2,IEEEdouble$minus,IEEEsingle$minus,Integer$minus,
             x,y);
function andc2(x,y : Boolean) : Boolean := x and not y;

function times(x,y : Constant) : Constant
    := do_op(logand,(and),IEEEdouble$times,IEEEsingle$times,Integer$times,
             x,y);
```

```
function divide(x,y : Constant) : Constant
begin
  case Constants${x,y} begin
    match {some_integer_constant(?),some_integer_constant(?)} begin
      result := divide(to_real(x),to_real(y));
    end;
  else
    result := do_op(logxor,xor,IEEEdouble$divide,IEEEsingle$divide,
                    Integer$divide, -- this parameter is not used
                    x,y);
  end;
end;
function xor(x,y : Boolean) : Boolean := (x and not y) or (not x and y);

function unary_plus(x : Constant) : Constant := x;

-- for unary -, we don't do zero-x because we want it to work for booleans
-- and sets too.  (And even coercing 0 to false will get the wrong value)
function unary_minus(x : Constant) : Constant
    := do_unop(lognot,(not),IEEEdouble$unary_minus,IEEEsingle$unary_minus,
               Integer$unary_minus,x);

function unary_times(x : Constant) : Constant := x;

function unary_divide(x : Constant) : Constant := divide(one,x);

function div(x,y : Constant) : Constant
begin
  case y begin
    match some_integer_constant(!0) begin
      result := nil;
    end;
  else
    result := do_op(logxor,xor, -- unused params
                    IEEEdouble$divide,IEEEsingle$divide, -- unused params
                    Integer$divide, -- only function used
                    x,y);
  end;
end;

function mod(x,y : Constant) : Constant
begin
  case y begin
    match some_integer_constant(!0) begin
      result := nil;
    end;
  else
    result := do_op(logxor,xor,IEEEdouble$divide,IEEEsingle$divide, -- ditto
                    floor, -- only function used
                    x,y);
```

```
  end;
end;
function floor(x,y : Integer) : Integer :=
    x-(x/y)*y;

function abs(v : Constant) : Constant
begin
  -- this is the lazy way:
  -- call the functions defined in this module:
  if less(v,zero) then
    result := unary_minus(v);
  else
    result := v;
  endif;
end;


-- factor out binary operations into a single function
private function do_op(set_op : function(_,_ : Integer) : Integer;
                       boolean_op : function(_,_ : Boolean) : Boolean;
                       longreal_op : function (_,_ : IEEEdouble)
                           : IEEEdouble;
                       real_op : function (_,_ : IEEEsingle) : IEEEsingle;
                       int_op : function (_,_ : Integer) : Integer;
                       x,y : Constant) : Constant
begin
  case do_coerce(x,y) begin
    match {set_constant(?rep1),
           set_constant(?rep2)} begin
      result := set_constant(set_op(rep1,rep2));
    end;
    match {boolean_constant(?b1),
           boolean_constant(?b2)} begin
      result := boolean_constant(boolean_op(b1,b2));
    end;
    match {longreal_constant(?v1),
           longreal_constant(?v2)} begin
      result := longreal_constant(longreal_op(v1,v2));
    end;
    match {real_constant(?v1),
           real_constant(?v2)} begin
      result := real_constant(real_op(v1,v2));
    end;
    match {longint_constant(?v1),
           longint_constant(?v2)} begin
      result := longint_constant(int_op(v1,v2));
    end;
    match {integer_constant(?v1),
           integer_constant(?v2)} begin
      result := integer_constant(int_op(v1,v2));
```

```
    end;
    match {shortint_constant(?v1),
           shortint_constant(?v2)} begin
      result := shortint_constant(int_op(v1,v2));
    end;
  else
    result := nil;
  end;
end;

private function do_unop(set_op : function(_ : Integer) : Integer;
                         boolean_op : function(_ : Boolean) : Boolean;
                         longreal_op : function (_:IEEEdouble) : IEEEdouble;
                         real_op : function (_ : IEEEsingle) : IEEEsingle;
                         int_op : function (_ : Integer) : Integer;
                         x : Constant) : Constant
begin
  case x begin
    match set_constant(?rep) begin
      result := set_constant(set_op(rep));
    end;
    match boolean_constant(?b) begin
      result := boolean_constant(boolean_op(b));
    end;
    match longreal_constant(?v) begin
      result := longreal_constant(longreal_op(v));
    end;
    match real_constant(?v) begin
      result := real_constant(real_op(v));
    end;
    match longint_constant(?v) begin
      result := longint_constant(int_op(v));
    end;
    match integer_constant(?v) begin
      result := integer_constant(int_op(v));
    end;
    match shortint_constant(?v) begin
      result := shortint_constant(int_op(v));
    end;
  else
    result := nil;
  end;
end;


-- comparison functions
function equal(x,y : Constant) : Boolean
    := do_compare(Integer$equal,String$equal,Boolean$equal,
                  IEEEdouble$equal,IEEEsingle$equal,Integer$equal,
                  x,y);
```

```
-- the following operations are not legal on sets in Oberon2
-- but we still define them (for fun).
function less(x,y : Constant) : Boolean
    := do_compare(proper_subset,String$less,bool_less,
                  IEEEdouble$less,IEEEsingle$less,Integer$less,
                  x,y);
function proper_subset(x,y : Integer) : Boolean
    := x/=y and subset(x,y);
function bool_less(x,y : Boolean) : Boolean
    := not x and y;

function less_equal(x,y : Constant) : Boolean
    := do_compare(subset,String$less_equal,bool_less_eq,
                  IEEEdouble$less_equal,IEEEsingle$less_equal,
                  Integer$less_equal,
                  x,y);
function subset(x,y : Integer) : Boolean
    := logandc2(x,y) = 0;
function bool_less_eq(x,y : Boolean) : Boolean
    := not x or y;

private function do_compare(set_op : function(x,y : Integer) : Boolean;
                            string_op : function(x,y : String) : Boolean;
                            bool_op : function(x,y : Boolean) : Boolean;
                            longreal_op : function (x,y : IEEEdouble)
                                : Boolean;
                            real_op : function (x,y : IEEEsingle) : Boolean;
                            int_op : function (x,y : Integer) : Boolean;
                            x,y : Constant) : Boolean
begin
  case do_coerce(x,y) begin
    match {set_constant(?rep1),
           set_constant(?rep2)} begin
      result := set_op(rep1,rep2);
    end;
    match {string_constant(?s1),string_constant(?s2)} begin
      result := string_op(s1,s2);
    end;
    match {char_constant(?c1),char_constant(?c2)} begin
      result := string_op({c1},{c2});
    end;
    match {boolean_constant(?b1),
           boolean_constant(?b2)} begin
      result := bool_op(b1,b2);
    end;
    match {longreal_constant(?v1),
           longreal_constant(?v2)} begin
      --! NaN problems:
      result := longreal_op(v1,v2);
```

```
    end;
    match {real_constant(?v1),
           real_constant(?v2)} begin
      --! NaN problems:
      result := real_op(v1,v2);
    end;
    match {some_integer_constant(?v1),
           some_integer_constant(?v2)} begin
      result := int_op(v1,v2);
    end;
  else
    -- badly typed or not constant
    -- just return false for want of a better value
    result := false;
  end;
end;

-- a shared coercion routine for use by do_compare and do_op:
-- it makes sure that both operands are of the same type by coercing
-- one up to the other.
private function do_coerce(x,y : Constant) : Constants begin
  case Constants${x,y} begin
    match {...,undefined(),...} begin
      result := {};
    end;
    match {...,set_constant(?),...} begin
      result := {to_set(x),to_set(y)};
    end;
    match {...,string_constant(?),...} begin
      result := {to_string(x),to_string(y)};
    end;
    match {...,char_constant(?),...} begin
      result := {to_char(x),to_char(y)};
    end;
    match {...,boolean_constant(?),...} begin
      result := {to_boolean(x),to_boolean(y)};
    end;
    match {...,longreal_constant(?),...} begin
      result := {to_longreal(x),to_longreal(y)};
    end;
    match {...,real_constant(?),...} begin
      result := {to_real(x),to_real(y)};
    end;
    match {...,longint_constant(?),...} begin
      result := {to_longint(x),to_longint(y)};
    end;
    match {...,integer_constant(?),...} begin
      result := {to_integer(x),to_integer(x)};
    end;
    match {...,shortint_constant(?),...} begin
```

```
      result := {to_shortint(x),to_shortint(y)};
    end;
  else
    result := {};
  end;
end;


--- Coercion routines
-- These routines do coercion between the various numeric
-- types that are part of this package.  The operators
-- only request upward coercions but downward coercions are also useful.

function to_set(x : Constant) : Constant begin
  case x begin
    match set_constant(?) begin
      result := x;
    end;
    -- hack: also match zero:
    match !zero begin
      result := set_constant(0);
    end;
  else
    result := nil;
  end;
end;

function to_string(x : Constant) : Constant begin
  case x begin
    match string_constant(?) begin
      result := x;
    end;
    match char_constant(?c) begin
      result := string_constant({c});
    end;
  else
    result := nil;
  end;
end;

function to_char(x : Constant) : Constant begin
  case x begin
    match string_constant({?c}) begin
      result := char_constant(c);
    end;
    match char_constant(?) begin
      result := x;
    end;
  else
    result := nil;
```

```
    end;
  end;

  function to_boolean(x : Constant) : Constant begin
    case x begin
      match boolean_constant(?) begin
        result := x;
      end;
    else
      result := nil;
    end;
  end;

  function to_longreal(x : Constant) : Constant begin
    case x begin
      match longreal_constant(?) begin
        result := x;
      end;
      match real_constant(?v) begin
        result := longreal_constant(IEEEwiden(v));
      end;
      match some_integer_constant(?v) begin
        result :=
            longreal_constant(IEEEdouble$from_integer(v));
      end;
    else
      result := nil;
    end;
  end;

  function to_real(x : Constant) : Constant begin
    case x begin
      match longreal_constant(?v) begin
        result := real_constant(IEEEnarrow(v));
      end;
      match real_constant(?) begin
        result := x;
      end;
      match some_integer_constant(?v) begin
        result :=
            real_constant(IEEEsingle$from_integer(v));
      end;
    else
      result := nil;
    end;
  end;

  function to_longint(x : Constant) : Constant begin
    case x begin
      match longreal_constant(?v) begin
```

```
      result := longint_constant(IEEEdouble$to_integer(v));
    end;
    match real_constant(?v) begin
      result := longint_constant(IEEEsingle$to_integer(v));
    end;
    match longint_constant(?) begin
      result := x;
    end;
    match some_integer_constant(?v) begin
      result := longint_constant(v);
    end;
  else
    result := nil;
  end;
end;

function to_integer(x : Constant) : Constant begin
  case x begin
    match longreal_constant(?v) begin
      result := integer_constant(IEEEdouble$to_integer(v));
    end;
    match real_constant(?v) begin
      result := integer_constant(IEEEsingle$to_integer(v));
    end;
    match integer_constant(?) begin
      result := x;
    end;
    match some_integer_constant(?v) begin
      result := integer_constant(v);
    end;
  else
    result := nil;
  end;
end;

function to_shortint(x : Constant) : Constant begin
  case x begin
    match longreal_constant(?v) begin
      result := shortint_constant(IEEEdouble$to_integer(v));
    end;
    match real_constant(?v) begin
      result := shortint_constant(IEEEsingle$to_integer(v));
    end;
    match shortint_constant(?) begin
      result := x;
    end;
    match some_integer_constant(?v) begin
      result := shortint_constant(v);
    end;
  else
```

```
      result := nil;
    end;
  end;

  -- for turning into an integer for case statements and for ORD
  function ord(x : Constant) : Integer begin
    case x begin
      match some_integer_constant(?v) begin
        result := v;
      end;
      match char_constant(?ch) begin
        result := char_code(ch);
      end;
      match string_constant({?ch}) begin
        result := char_code(ch);
      end;
      match boolean_constant(?b) begin
        if b then
          result := 1;
        else
          result := 0;
        endif;
      end;
    end;
  end;
end;
```

B.2 Predefined Procedures

```
module OBERON2_ADD_PREDEFINED[T :: input OBERON2_TREE[]] extends T begin

  -- we only add nodes to the tree.  We do not export anything.
  private;

  -- we create extra declarations that will be in the root_contour
  -- for that purpose, we have several procedures:
  procedure make_global_const(name : String; value : Expression)
      : Declaration
      := const_decl(identifier(make_symbol(name),exported()),value);
  procedure make_global_type(name : String; base : Type) : Declaration
      := type_decl(identifier(make_symbol(name),exported()),base);
  procedure make_global_proc(name : String; formals : Formals; rt : Type)
      : Declaration
      := forward(header(identifier(make_symbol(name),exported()),
                        no_receiver(),formals,rt));
  procedure make_value_formal(of_type : Type) : Formals
      := Formals${value_formal(ignore(),of_type)};
  procedure make_value_formals(t1,t2 : Type) : Formals
      := Formals${value_formal(ignore(),t1),value_formal(ignore(),t2)};
```

```
procedure make_named_type(name : String) : Type
    := named_type(use_name(make_symbol(name)));

predefined_decls : Declarations :=
    Declarations$
    {-- Section 6.1:
     make_global_type("*ANY*",any),
     make_global_type("BOOLEAN",boolean),
     make_global_type("CHAR",char),
     make_global_type("SHORTINT",shortint),
     make_global_type("INTEGER",integer),
     make_global_type("LONGINT",longint),
     make_global_type("REAL",real),
     make_global_type("LONGREAL",longreal),
     make_global_type("*STRING*",pointer_type(string)),
     make_global_const
         ("TRUE",constant_expression(Constant$boolean_constant(true))),
     make_global_const
         ("FALSE",constant_expression(Constant$boolean_constant(false))),

     -- Section 6.4:
     make_global_const("NIL",constant_expression(Constant$nil)),

     -- Section 10.3:
     -- value returning procedures:
     make_global_proc("ABS",
                      make_value_formal(make_named_type("LONGREAL")),
                      abs_type()),
     make_global_proc("ASH",
                      make_value_formals(make_named_type("LONGINT"),
                                         make_named_type("LONGINT")),
                      make_named_type("LONGINT")),
     make_global_proc("CAP",
                      make_value_formal(make_named_type("CHAR")),
                      make_named_type("CHAR")),
     make_global_proc("CHR",
                      make_value_formal(make_named_type("LONGINT")),
                      make_named_type("CHAR")),
     make_global_proc("ENTIER",
                      make_value_formal(make_named_type("LONGREAL")),
                      make_named_type("LONGINT")),
     make_global_proc("LEN",
                      Formals${value_formal
                                   (ignore(),
                                    open_array_type
                                        (make_named_type("*ANY*"))),
                               opt_formal(make_named_type("LONGINT"),
                                          Constant$integer_constant(1))},
                      make_named_type("LONGINT")),
```

```
make_global_proc("LONG",
                 make_value_formal(long_type()),
                 long_type()),
make_global_proc("MAX",
                 make_value_formal(max_type()),
                 max_type()),
make_global_proc("MIN",
                 make_value_formal(max_type()),
                 max_type()),
make_global_proc("ODD",
                 make_value_formal(make_named_type("LONGINT")),
                 boolean_type()),
make_global_proc("ORD",
                 make_value_formal(make_named_type("CHAR")),
                 make_named_type("INTEGER")),
make_global_proc("SHORT",
                 make_value_formal(short_type()),
                 short_type()),
make_global_proc("SIZE",
                 make_value_formal(type_type()),
                 make_named_type("LONGINT")),
-- proper procedures
make_global_proc("ASSERT",
                 Formals${value_formal(ignore(),boolean_type()),
                          opt_formal(make_named_type("LONGINT"),
                                     Constant$integer_constant(1))},
                 no_type()),
make_global_proc("COPY",
                 Formals$
                     {value_formal
                          (ignore(),
                           open_array_type(make_named_type("CHAR"))),
                      -- not sure if this will work:
                      var_formal
                          (ignore(),
                           open_array_type(make_named_type("CHAR")))},
                 no_type()),
make_global_proc("DEC",
                 -- not really:
                 Formals$
                     {var_formal(ignore(),make_named_type("LONGINT")),
                      opt_formal(make_named_type("LONGINT"),
                                 Constant$integer_constant(1))},
                 no_type()),
make_global_proc("EXCL",
                 Formals$
                     {var_formal(ignore(),set_type()),
                      value_formal(ignore(),
                                   make_named_type("LONGINT"))},
                 no_type()),
```

```
      make_global_proc("HALT",make_value_formal(make_named_type("LONGINT")),
                       no_type()),
      make_global_proc("INC",
                       -- not really:
                       Formals${var_formal(ignore(),
                                           make_named_type("LONGINT")),
                                opt_formal(make_named_type("LONGINT"),
                                           Constant$integer_constant(1))},
                       no_type()),
      make_global_proc("INCL",
                       Formals${var_formal(ignore(),set_type()),
                                value_formal(ignore(),
                                             make_named_type("LONGINT"))},
                       no_type()),
      make_global_proc("NEW",
                       Formals${var_formal
                                    (ignore(),
                                     pointer_type(make_named_type("*ANY*"))),
                                rest_formal(make_named_type("LONGINT"))},
                       no_type())};
end;
```

B.3 Name Resolution

In the Oberon2 compiler, name resolution and computing the types of expressions must be interleaved. The following module brings these parts together:

```
module OBERON2_RESOLVE[T :: var OBERON2_TREE[], var OBERON2_ADD_PREDEFINED[T]]
    :: var OBERON2_SYMTAB[T], var OBERON2_EXPR_TYPE[T]
    extends T
begin
  -- these two modules need to be inherited into one so they
  -- can be implemented in one pass:
  inherit OBERON2_SYMTAB[T](base_type,expr_type) begin
    var use_decl = use_decl;
    var no_decl_reason = no_decl_reason;
    var is_redeclaration = is_redeclaration;
    var overrides = overrides;
    type ForwardSet = ForwardSet;
    var forwarding_decls = forwarding_decls;
    var is_builtin = is_builtin;
  end;
  inherit OBERON2_EXPR_TYPE[T](use_decl) begin
    var base_type = base_type;
    var expr_type = expr_type;
    var expr_header = expr_header;
    var implicitly_guarded = implicitly_guarded;
  end;

  -- "A type guard v(T) ... "
```

```
  -- NB: type guards are read as function calls:
  -- we define an attribute to say when a funcall is a type_guard
  public attribute Expression.is_type_guard : Boolean := false;
  -- We say an expression v(T) is a type guard if T is a type
  -- and v isn't a procedure
  match ?e=funcall(?v,{?T}) begin
    case T.expr_type begin
      match type_type() begin
        if v.expr_header == nil then
          e.is_type_guard := true;
        endif;
      end;
    end;
  end;
  var pattern actual_type_guard(v : Expression; uT : Use) : Expression
      := type_guard(?v,named_type(?uT)),
         ?e=funcall(?v,{named_expr(?uT)}) if e.is_type_guard;
end;
```

B.3.1 Algol Scope

The Oberon2 compiler uses a generic Algol scope module that defines the basic symbol table operations.

```
module ALGOL_SCOPE[phylum Decl :: var PHYLUM[]]
    (decl_name : function(x : remote Decl) : Symbol)
    phylum Contour
begin
  constructor root_contour() : Contour;
  constructor nested_contour(parent : remote Contour) : Contour;

  type Decls := BAG[remote Decl];
  input collection attribute Contour.local_decls : Decls;

  private type SortedDecls := ORDERED_SET[remote Decl]((==),(<<));

  var function find_local_decl(name : Symbol; scope : remote Contour)
      : remote Decl
  begin
    case SortedDecls${decl if decl_name(decl) = name
                          for decl in scope.local_decls}
    begin
      match {?first,...} begin
        result := first;
      end;
    else
      result := Decl$nil;
    end;
  end;
  pragma memo(find_local_decl);
```

```
  var function find_decl(name : Symbol; scope : remote Contour)
      : remote Decl
  begin
    here : remote Decl := find_local_decl(name,scope);
    if here /= Decl$nil then
      result := here;
    else
      case scope begin
        match nested_contour(?parent) begin
          result := find_decl(name,parent);
        end;
      else
        result := Decl$nil;
      end;
    endif;
  end;
end;
```

B.3.2 The Symbol Table

```
-- This module is responsible for creating scopes and inserting declarations
-- into the scopes.  It requires information from name resolution for types
-- so it can locate extended types and receiver types for type-bound types.

module OBERON2_SYMTAB[T :: var OBERON2_TREE[]]
    (base_type : function (_ : remote T$Type) : remote T$Type;
     expr_type : function (_ : remote T$Expression) : remote T$Type) extends T
begin

  -- this attribute expresses the main work of the symbol table: looking
  -- things up.  The next attribute is a string that gives a reason
  -- (error message) for a use not being found (if use_decl is nil)
  attribute Use.use_decl : remote Declaration := nil;
  attribute Use.no_decl_reason : String := "";

  -- extra information that requires the symbol table:
  -- is_redeclaration       true for a decl that incorrectly redeclares another
  -- overrides               in inheritance, the inherited shadowed decl
  -- forwarding_decls       for a forward declaration a list of redeclarations
  attribute Declaration.is_redeclaration : Boolean := false;
  attribute Declaration.overrides : remote Declaration := nil;
  type ForwardSet := ORDERED_SET[remote Declaration]((==),(<<));
  collection attribute Declaration.forwarding_decls : ForwardSet;

  attribute (d:Declaration).is_builtin : Boolean := d.scope==root_scope;

  private;
```

```
--- Scope

phylum Contour := ALGOL_SCOPE[Declaration](decl_name);

root_contour = Contour$root_contour;
pattern root_contour = Contour$root_contour;
nested_contour = Contour$nested_contour;
pattern nested_contour = Contour$nested_contour;
local_decls = Contour$local_decls;
find_local_decl = Contour$find_local_decl;
find_decl = Contour$find_decl;

-- We need a distinct type for record scopes
-- because we need to insert things in record scopes aftering
-- looking things up in regular scopes, and only
-- the type system is powerful enough to prove that this doesn't
-- cause a cycle.

phylum RecordContour := ALGOL_SCOPE[Declaration](decl_name);

pattern root_record_contour = RecordContour$root_contour;
pattern nested_record_contour = RecordContour$nested_contour;
root_record_contour = RecordContour$root_contour;
nested_record_contour = RecordContour$nested_contour;
local_record_decls = RecordContour$local_decls;
find_local_record_decl = RecordContour$find_local_decl;
find_record_decl = RecordContour$find_decl;

private function decl_name(x : remote Declaration) : Symbol begin
  case x begin
    match declaration(identifier(?name,?)) begin
      result := name;
    end;
  end;
end;

type Scope := remote Contour;
type RecordScope := remote RecordContour;

signature SCOPABLE := {Declaration,Block,Header,Receiver,
                       Statement,Expression,Use,Type,
                       Case,CaseLabel,Element},
                      var PHYLUM[];

-- The scope for inserting declarations
-- (The default will only be active for entities not in any module,
-- that is the predefined procedures)
[phylum T :: SCOPABLE] attribute T.scope : Scope := root_scope;
attribute Block.saved_scope : Scope := root_scope;
attribute Declaration.record_scope : RecordScope;
```

```
attribute Type.saved_record_scope : RecordScope;

-- predefined things sit outside the program and can be shadowed by
-- local declarations;
root_scope : Scope := root_contour();

-- this is the scope that modules are declared in:
module_scope : Scope := root_contour();

-- this scope is used to represent a default scope with nothing declared
-- in it:
empty_scope : Scope := root_contour();


--- Establishing scope for entities

match program({...,?mod,...}) begin
  mod.scope := root_scope;
end;

match ?b=block(?decls,?stmts) begin
  inner : Scope := nested_contour(b.scope);
  for decl in decls begin
    decl.scope := inner;
  end;
  for stmt in stmts begin
    stmt.scope := inner;
  end;
  b.saved_scope := inner;
end;

match ?h=header(?,?receiver,?formals,?) begin
  inner : Scope := nested_contour(h.scope);
  receiver.scope := inner;
  for formal in formals begin
    formal.scope := inner;
  end;
end;

-- the block is situated in the scope declaring the receiver and formals
-- Since the body is a "block", local declarations can shadow
-- outer declarations.
match proc_decl(header(receiver:=?rec),?body) begin
  body.scope := rec.scope;
end;

-- each record's fields are scoped inside the record it extends,
match ?rt=record_type(?extending,?fields) begin
  field_scope : RecordScope;
  case extending.base_type begin
```

```
    match ?pt=record_type(...) begin
      field_scope := nested_record_contour(pt.saved_record_scope);
    end;
  else
    field_scope := root_record_contour();
  end;
  rt.saved_record_scope := field_scope;
  for field in fields begin
    field.record_scope := field_scope;
  end;
end;

-- otherwise we just copy scope to the child
[phylum P :: SCOPABLE;
 phylum C :: SCOPABLE] begin
  match ?parent:P=parent(?child:C) begin
    child.scope := parent.scope;
  end;
end;

signature SCOPABLE_SEQ := {Declarations,Formals,Fields,Statements,
                           Cases,CaseLabels,Actuals,Elements}, var PHYLUM[];
[phylum P :: SCOPABLE;
 phylum L :: SEQUENCE[C],SCOPABLE_SEQ;
 phylum C :: SCOPABLE] begin
  match ?parent:P=parent(?:L={...,?child:C,...}) begin
    child.scope := parent.scope;
  end;
end;


--- Inserting elements in scopes and checking for redeclaration
--- and compute forwarding resolutions and find overridden declarations:

-- every declaration inserts itself in the scope
-- except that modules insert themselves into the module scope
-- and procedures with a receiver insert themselves into the record:
match ?d=declaration(identifier(?name,?)) begin
  insert_record_scope : RecordScope := nil;
  insert_scope : Scope := nil;
  is_record_scope : Boolean := false;
  case d begin
    match module_decl(...) begin
      insert_scope := module_scope;
    end;
    match bound_proc_decl(?,?rec_formal) begin
      is_record_scope := true;
      -- section 10.2: "The receiver may either be a variable parameter
      -- of record type T or ..."
      case rec_formal begin
```

```
      match var_formal(?,?rtype) begin
        case rtype.base_type begin
          match ?rt=record_type(...) begin
            insert_record_scope := rt.saved_record_scope;
          end;
        end;
      end;
      -- "... a value parameter of type POINTER TO T
      -- (where T is a record type).  The procedure is bound to the type
      -- T and is considered local to it"
      match value_formal(?,?rtype) begin
        case rtype.base_type begin
          match pointer_type(?ty) begin
            case ty.base_type begin
              match ?rt=record_type(...) begin
                insert_record_scope := rt.saved_record_scope;
              end;
            end;
          end;
        end;
      end;
    end;
  end; -- match bound_proc_decl(...)
  match field(?,?) begin
    insert_record_scope := d.record_scope;
    is_record_scope := true;
  end;
else
  insert_scope := d.scope;
end;

-- insert ourselves in this scope
-- and then check for redeclaration (and forwarding)
-- and overriding of inherited declarations
redeclared : remote Declaration := nil;
if is_record_scope then
  if insert_record_scope /= nil then
    insert_record_scope.local_record_decls :> {d};
    redeclared := find_local_record_decl(name,insert_record_scope);
    case insert_record_scope begin
      match nested_record_contour(?parent_scope) begin
        d.overrides := find_record_decl(name,parent_scope);
      end;
    end;
  endif;
else
  if insert_scope /== nil then
    insert_scope.local_decls :> {d};
    redeclared := find_local_decl(name,insert_scope);
  endif;
```

```
  endif;
  case redeclared begin
    match !nil begin end;
    match !d begin end; -- OK if we find ourselves:
    match forward(...) begin
      redeclared.forwarding_decls :> {d};
    end;
  else
    d.is_redeclaration := true;
  end;
end;


--- Definitions of use_decl and no_decl_reason

-- Field reference (a special case):
-- we have two matching rules for it: one for modules and
-- one for field reference:

-- if the fref is not a module use, this rule won't attribute
-- anything and the normal fref rule will kick in:
[T :: {Expression,Use}, var PHYLUM[]]
    match ?:T=possibly_qualified(?module_use,?using=use_name(?name))
begin
  case module_use.use_decl begin
    match import(?,?module_use) begin
      case module_use.use_decl begin
        match module_decl(?,?body) begin
          case find_local_decl(name,body.saved_scope) begin
            match !nil begin
              using.use_decl := nil;
              using.no_decl_reason := "Identifier not in module";
            end;
            match declaration(identifier(?,not_exported())) begin
              using.use_decl := nil;
              using.no_decl_reason := "Identifier not exported";
            end;
            -- otherwise, just use the decl:
            match ?decl begin
              using.use_decl := decl;
            end;
          end;
        end;
      else
        -- force a default case (avoid cascading error messages)
        using.use_decl := nil;
        using.no_decl_reason := "";
      end;
    end;
  end;
```

```
end;
-- copy information to the qualified use as well.
match ?qu=qualified(?,?eu) begin
  qu.use_decl := eu.use_decl;
end;

-- in Oberon2: x.f can be a shorthand for x^.f
-- and so we have a function that does the optional pointer dereference
-- for us if necessary:
private function optptr_deref(t : remote Type) : remote Type
begin
  case t.base_type begin
    match pointer_type(?t) begin
      result := t;
    end;
  else
    result := t;
  end;
end;

match fref(?expr,?field=use_name(?name),...) begin
  case optptr_deref(expr.expr_type).base_type begin
    match ?rt=record_type(...) begin
      found : remote Declaration :=
          find_record_decl(name,rt.saved_record_scope);
      field.use_decl := found;
      if found == nil then
        field.no_decl_reason := "Undeclared field";
      endif;
    end;
  else
    field.use_decl := nil;
    field.no_decl_reason := ""; -- avoid multiple errors
  end;
end;

-- There are two places where definition is not required before use:
-- when we are dealing with other modules and when we are looking up
-- a local pointer type.

-- for modules, we have to make sure we don't have a circular import,
-- if we do then our "dynamic" attributes could be truly circular.
match ?md=module_decl(?,block({...,import(?,?u=use_name(?name)),...},?))
begin
  mref : remote Declaration :=
      find_local_decl(name,module_scope);
  if mref == nil then
    u.use_decl := nil;
    u.no_decl_reason := "Undeclared module";
  elsif mref == md then
```

```
    -- Section 11: "A module must not import itself"
    u.use_decl := nil;
    u.no_decl_reason := "Illegal self-import";
  elsif not check_import(md,{},mref) then
    -- Section 11: "...cyclic import of modules is illegal"
    u.use_decl := nil;
    u.no_decl_reason := "Illegal indirect self-import";
  else
    u.use_decl := mref;
  endif;
end;

type ModuleSet := ORDERED_SET[remote Declaration]((==),(<<));
-- This function returns false if having the importer import the given
-- module would cause an import of the original module.
-- "true" being returned means it is OK to import the given module.
-- Note that the second argument is used to ensure that a cyclic import
-- elsewhere in the module import structure doesn't cause this function
-- to loop endlessly.  This function could be made into a circular
-- attribute instead.
function check_import(importer : remote Declaration;
                      cycle_check : ModuleSet;
                      a_module : remote Declaration)
    collection result : Boolean :> true, (and)
begin
  if a_module = importer then
    result :> false;
  elsif a_module /= nil and a_module not in cycle_check then
    new_cycle_check : ModuleSet := {cycle_check...,a_module};
    for a_module begin
      match module_decl(?,block(decls:={...,import(?,use_name(?mname)),...}))
      begin
        result :>
            check_import(importer,new_cycle_check,
                         find_local_decl(mname,module_scope));
      end;
    end;
  endif;
end;

-- pointer types can be forward references:
match pointer_type(named_type(?u=use_name(?name))) begin
  u.use_decl := find_decl(name,u.scope);
end;

-- otherwise we check for definition before use:
match ?u=use_name(?name) begin
  found : remote Declaration := find_decl(name,u.scope);
  if found /== nil then
    if found << u or found.is_builtin then
```

```
      u.use_decl := found;
    elsif u.use_decl == nil then
      u.no_decl_reason := "Illegal forward use";
    endif;
  endif;
end;

-- fill in the default error message
match ?u=use_name(?) begin
  if u.use_decl == nil then
    u.no_decl_reason := "Undeclared identifier";
  endif;
end;
end;

-- OBERON2_SYMTAB cannot be compiled separately
-- because it relies on things that rely on
-- this module:
pragma no_separate(module OBERON2_SYMTAB);
```

B.3.3 Expression Types

```
module OBERON2_EXPR_TYPE[T :: var OBERON2_TREE[]]
    (use_decl : function (_ : remote T$Use) : remote T$Declaration)
    extends T
begin
  -- an attribute that handles the type resolution of type names
  -- Note that it must be dynamic because it is copied across
  -- type name uses.
  attribute Type.base_type : remote Type := any;
  pragma dynamic(base_type);

  attribute Expression.expr_type : remote Type := any;
  attribute Expression.implicitly_guarded : Boolean := false;

  -- "expr_type" is not correct for procedure names
  -- used in expressions, so we have another attribute to handle
  -- procedure values ("expr_header")
  attribute Expression.expr_header : remote Header := nil;
  -- if the expr_type is a valid procedure type then the header
  -- is immediately available:
  match ?e:Expression begin
    case e.expr_type begin
      match proc_type(?header) begin
        e.expr_header := header;
      end;
    end;
  end;
```

```
private;


--- BASE TYPES
-- each type has a base type, that is, one that
-- makes sure we get past the name of the type to a real type constructor

match ?ty1=named_type(?using) begin
  case using.use_decl begin
    match type_decl(?,?ty2) begin
      ty1.base_type := ty2.base_type;
    end;
  end;
end;

match ?ty=pseudo_type() begin
  ty.base_type := any;
end;

match ?ty:Type begin
  ty.base_type := ty;
end;


--- EXPRESSION TYPES

-- Use's come in two places:
-- in named_expr's and in fref's (could be imported entity)
-- In either case, we follow similar rules:

attribute Use.use_type : remote Type := any;
attribute Use.use_header : remote Header := nil;

match ?u=use_name(...) begin
  case use_decl(u) begin
    match const_decl(?,?value) begin
      u.use_type := value.expr_type;
    end;
    match type_decl(...) begin
      u.use_type := a_type;
    end;
    match var_decl(?,?shape) begin
      u.use_type := shape.base_type;
    end;
    match proc_decl(?header,?) begin
      -- NB: no type, but instead a header:
      u.use_header := header;
    end;
    match forward(?header) begin
      u.use_header := header;
```

```
    end;
    match formal(?,?shape) begin
      u.use_type := shape.base_type;
    end;
    -- to handle field references:
    match field(?,?shape) begin
      u.use_type := shape.base_type;
    end;
  end;
end;


-- With statements change the type of the variable
-- but we have to watch out for multiple with statements on the same
-- variable.
type WithList := LIST[remote Statement];

[phylum T :: PHYLA] attribute T.active_withs : WithList := {};

match ?w=with_stmt(body:=?body) begin
  for s in body begin
    s.active_withs := {w} ++ w.active_withs;
  end;
end;
-- otherwise we copy down (by default)
[phylum PT,CT :: PHYLA] begin
  match ?p:PT=parent(?c:CT) begin
    c.active_withs := p.active_withs;
  end;
end;
[phylum PT,CT :: PHYLA;
 phylum ST :: SEQ_PHYLA, SEQUENCE[CT]] begin
  match ?p:PT=parent(ST${...,?c:CT,...}) begin
    c.active_withs := p.active_withs;
  end;
end;

match ?e=named_expr(?using) begin
  case e.active_withs begin
    -- case selects the first match!
    match {...,with_stmt(named_expr(?wuse),?guard_type,...),...}
        if wuse.use_decl /== nil
        if wuse.use_decl == using.use_decl
    begin
      e.implicitly_guarded := true;
      e.expr_type := guard_type.base_type;
    end;
  else
    -- otherwise we get the type or header from the use:
    e.expr_type := using.use_type;
```

```
    e.expr_header := using.use_header;
  end;
end;

match ?e=unop(?,?arg) begin
  e.expr_type := arg.expr_type;  -- for minus & not
end;

type Types := LIST[remote Type];

-- handle binop's in three cases:
-- predicate operators are easy---the result is always boolean;
-- divide (/) is special---it always forces a real result;
-- otherwise we choose the least common type.
match ?e=binop(predicate_operator(),?,?) begin
  e.expr_type := boolean;
end;
match ?e=binop(divide(),?e1,?e2) begin
  case Types${e1.expr_type,e2.expr_type} begin
    match {...,?ty=longreal_type(),...} begin
      e.expr_type := ty;
    end;
    match {...,?ty=set_type(),...} begin
      e.expr_type := ty;
    end;
  else
    e.expr_type := real;
  end;
end;
match ?e=binop(arithmetic_operator(),?e1,?e2) begin
  case Types${e1.expr_type,e2.expr_type} begin
    match {...,?ty=longreal_type(),...} begin
      e.expr_type := ty;
    end;
    match {...,?ty=real_type(),...} begin
      e.expr_type := ty;
    end;
    match {...,?ty=longint_type(),...} begin
      e.expr_type := ty;
    end;
    match {...,?ty=integer_type(),...} begin
      e.expr_type := ty;
    end;
    match {...,?ty=shortint_type(),...} begin
      e.expr_type := ty;
    end;
    match {...,?ty=set_type(),...} begin
      e.expr_type := ty;
    end;
  end;
```

```
end;

match ?e=funcall(?func,?actuals) begin
  -- we have to handle a lot of special cases:
  case func.expr_header begin
    match header(result:=abs_type()) begin
      case actuals begin
        match {?first} begin
          e.expr_type := first.expr_type;
        end;
      end;
    end;
    match header(result:=long_type()) begin
      case actuals begin
        match {?op} begin
          case op.expr_type begin
            match shortint_type() begin
              e.expr_type := integer;
            end;
            match integer_type() begin
              e.expr_type := longint;
            end;
            match real_type() begin
              e.expr_type := longreal;
            end;
          end;
        end;
      end;
    end;
    match header(result:=max_type()) begin
      case actuals begin
        match {?op} begin
          case op.expr_type begin
            match set_type() begin
              e.expr_type := integer;
            end;
          else
            e.expr_type := op.expr_type;
          end;
        end;
      end;
    end;
    match header(result:=short_type()) begin
      case actuals begin
        match {?op} begin
          case op.expr_type begin
            match integer_type() begin
              e.expr_type := shortint;
            end;
            match longint_type() begin
```

```
              e.expr_type := integer;
            end;
            match longreal_type() begin
              e.expr_type := real;
            end;
          end;
        end;
      end;
    end;
    -- otherwise a (fairly) normal procedure
    match header(result:=?rt) begin
      e.expr_type := rt.base_type;
    end;
  end;
end;

-- type guards are not syntactically distinguishable from funcalls:
-- we put this after funcall to make sure it doesn't catch things
-- like SIZE(T).  The latter will have a expr_header and will
-- be handled by the earlier patterns

match ?e=funcall(?,{named_expr(?using)}) begin
  case using.use_decl begin
    match type_decl(?,?ty) begin
      e.expr_type := ty.base_type;
    end;
  end;
end;

match ?e=is_test(...) begin
  e.expr_type := boolean;
end;

-- NB: type guards are syntactically indistinguishable from
-- procedure calls so any type guards in the tree must be generated
-- directly.
match ?e=type_guard(?,?guard) begin
  e.expr_type := guard.base_type;
end;

match ?e=aref(?array,?) begin
  case array.expr_type begin
    match array_type(?et) begin
      e.expr_type := et.base_type;
    end;
    match pointer_type(?at) begin
      case at.base_type begin
        match array_type(?et) begin
          e.expr_type := et.base_type;
        end;
```

```
      end;
    end;
  end;
end;

-- field or imported ID
match ?e=fref(?,?field,...) begin
  e.expr_type := field.use_type;
  e.expr_header := field.use_header;
end;

--(We assume this is taken care of in the parser or
--an earlier canonicalization)
--
-- fetch is syntactic sugar for "super":
-- match ?e=fetch(fref(?,?field)) begin
--   case field.use_decl begin
--     match proc_decl(header:=?header) begin
--         e.expr_header := header;
--     end;
--   end;
-- end;

match ?e=fetch(?pe) begin
  case pe.expr_type begin
    match pointer_type(?ty) begin
      e.expr_type := ty.base_type;
    end;
  end;
end;

match ?e=set_expr(...) begin
  e.expr_type := set;
end;

match ?e=constant_expression(?con) begin
  case con begin
    match Constant$shortint_constant(...) begin
      e.expr_type := shortint;
    end;
    match Constant$integer_constant(...) begin
      e.expr_type := integer;
    end;
    match Constant$longint_constant(...) begin
      e.expr_type := longint;
    end;
    match Constant$real_constant(...) begin
      e.expr_type := real;
    end;
    match Constant$longreal_constant(...) begin
```

```
        e.expr_type := longreal;
      end;
      match Constant$string_constant(...) begin
        e.expr_type := string;
      end;
      match Constant$char_constant(...) begin
        e.expr_type := char;
      end;
      match Constant$set_constant(...) begin
        e.expr_type := set;
      end;
      match Constant$boolean_constant(...) begin
        e.expr_type := boolean;
      end;
    end;
  end;
end;

pragma no_separate(module OBERON2_EXPR_TYPE);
```

B.4 Compile-Time Computation

The Oberon2 language implicitly requires certain computations to be carried out at compile-time. In particular, the sizes of records and expressions involving only constant operands must be computed at compile time.

```
type Oberon2Sizes := ORDERED_SET[Integer]((=),(<));

class OBERON2_MACHINE_SIZES[] begin
  byte_bits : Integer;

  char_size : Integer;
  shortint_size : Integer;
  integer_size : Integer;
  longint_size : Integer;
  real_size : Integer;
  longreal_size : Integer;

  sub_alignment_sizes : Oberon2Sizes;
  max_alignment : Integer; -- alignment can be in multiples of this size

  address_size : Integer;
end;

module OBERON2_COMPILE_COMPUTE[T :: var OBERON2_TREE[],
                                    var OBERON2_SYMTAB[T],
                                    var OBERON2_EXPR_TYPE[T],
                                    OBERON2_MACHINE_SIZES[]]
    extends T
```

```
begin

  -- use the machine dependent values to set up the integer types:
  function max_twos_complement(bytes : Integer) : Integer
      :=  (2^(bytes*byte_bits-1))-1;
  function min_twos_complement(bytes : Integer) : Integer
      := -(2^(bytes*byte_bits-1));

  max_shortint : Integer := max_twos_complement(shortint_size);
  min_shortint : Integer := min_twos_complement(shortint_size);
  max_integer : Integer := max_twos_complement(integer_size);
  min_integer : Integer := min_twos_complement(integer_size);
  max_longint : Integer := max_twos_complement(longint_size);
  min_longint : Integer := min_twos_complement(longint_size);

  set_size : Integer := longint_size;

  max_set : Integer := set_size-1;
  min_set : Integer := 0;

  -- return the next legal size
  -- all sizes must be in the set sub_alignment_sizes or a multiple
  -- of max_alignment:
  function round_up_size(size : Integer) : Integer begin
    if size >= max_alignment then
      result := ((size+max_alignment-1)/max_alignment)*max_alignment;
    elsif size in sub_alignment_sizes then
      result := size;
    else
      result := round_up_size(size+1);
    endif;
  end;

  function align_size(size : Integer; alignment : Integer) : Integer begin
    if alignment >= max_alignment then
      result := ((size+max_alignment-1)/max_alignment)*max_alignment;
    elsif alignment = 0 then -- no alignment
      result := size;
    else
      result := ((size+alignment-1)/alignment)*alignment;
    endif;
  end;

  --- TYPES and TYPE SIZES

  -- compute type sizes and alignments
  attribute Type.type_size : Integer := 0;
  attribute Type.fixed_size_type : Boolean := true; -- open arrays aren't

  -- The dependencies are never circular, but static
```

```
-- analysis will fail to find a safe dependency order,
-- and so these attributes are declared to need "dynamic dependency
-- tracking".
pragma dynamic(type_size);
pragma dynamic(fixed_size_type);

-- basic types
match ?ty=boolean_type() begin
  ty.type_size := char_size;
end;
match ?ty=char_type() begin
  ty.type_size := char_size;
end;
match ?ty=set_type() begin
  ty.type_size := set_size;
end;
match ?ty=shortint_type() begin
  ty.type_size := shortint_size;
end;
match ?ty=integer_type() begin
  ty.type_size := integer_size;
end;
match ?ty=longint_type() begin
  ty.type_size := longint_size;
end;
match ?ty=real_type() begin
  ty.type_size := real_size;
end;
match ?ty=longreal_type() begin
  ty.type_size := longreal_size;
end;

match ?ty=named_type(?u) begin
  case u.use_decl begin
    -- a cycle:
    match ?td=type_decl(...) if td ^^ ty begin
      ty.fixed_size_type := true;
      ty.type_size := 0;
    end;
    match type_decl(value:=?subty) begin
      ty.fixed_size_type := subty.fixed_size_type;
      ty.type_size := subty.type_size;
    end;
  end;
end;

-- POINTER TO ARRAY is handled differently:
match ?ty=pointer_type(?at=open_array_type(...)) begin
  ty.type_size := (1 + at.open_ranges)*address_size;
end;
```

```
match ?ty=pointer_type(...) begin
  ty.type_size := address_size;
end;

match ?ty=proc_type(...) begin
  -- The restrictions on local procedures means that there
  -- is no need to keep environments with procedure values
  ty.type_size := address_size;
end;

attribute Type.open_ranges : Integer := 0;
match ?ty=open_array_type(?elem_ty) begin
  ty.fixed_size_type := false;
  ty.open_ranges := 1 + elem_ty.open_ranges;
  -- but it's still useful to have this value around:
  ty.type_size := elem_ty.type_size;
end;
match ?ty=fixed_array_type(?length,?elem_ty) begin
  ty.fixed_size_type := elem_ty.fixed_size_type;
  case length.constant_value begin
    match Constant$some_integer_constant(?v) begin
      ty.type_size := round_up_size(v*elem_ty.type_size);
    end;
  end;
end;

-- the most complicated type:
match ?ty=record_type(?extending,?fields) begin
  prefix_size : Integer;
  case extending begin
    match no_type() begin
      -- need space for a type link
      prefix_size := address_size;
    end;
  else
    prefix_size := extending.type_size;
  end;
  total_size : Integer;
  field.min_offset for field in fields, total_size :=
      prefix_size, field.next_offset for field in fields;
  ty.type_size := round_up_size(total_size);
end;

attribute Declaration.min_offset : Integer := 0;
attribute Declaration.offset : Integer := 0;
attribute Declaration.next_offset : Integer := 0;
match ?d=field(?,?shape) begin
  d.offset := align_size(d.min_offset,shape.type_size);
  d.next_offset := d.offset + shape.type_size;
end;
```

```
--- COMPILE-TIME CONSTANTS

-- "A constant expression in an expression that can be evaluated by
-- a mere textual scan without actually executing the program. ..."

attribute Expression.expr_constant : Boolean := false;
-- sometimes we need the value:
-- (We need a default to avoid getting undefined attribute errors
-- for erroneous programs).  We create new tree nodes in a different
-- forest to represent the values:
attribute Expression.constant_value : Constant := Constant$nil;
-- this information stores information for MIN and MAX:
attribute Expression.constant_type_value : remote Type := nil;

type Errors := BAG[String];
collection attribute Expression.errors : Errors;

procedure add_error(e : remote Expression; message : String) begin
  e.errors :> {message};
end;

-- "It's operands are constants or ..."
match ?e=constant_expression(?v) begin
  e.expr_constant := true;
  e.constant_value := v;
end;

-- NB: an imported constant masquerades as a field reference:
pattern possible_constant_use(u : Use) : Expression
    := named_expr(?u),fref(?,?u,?);

pattern builtin(name : Symbol) : Declaration :=
    ?d=forward(header(identifier(?name,?),...))
    if d.is_builtin;

function builtin_name(x : remote Declaration) : String := ""
begin
  case x begin
    match builtin(?name) begin
      result := symbol_name(name);
    end;
  end;
end;

match ?e=possible_constant_use(?using) begin
  case using.use_decl begin
    match const_decl(?,?value) begin
      e.expr_constant := true;
```

```
      e.constant_value := value.constant_value;
    end;
    -- Presumably: types count as constant too
    -- (for the purpose of MAX and MIN)
    match type_decl(value:=?ty) begin
      e.expr_constant := true;
      e.constant_type_value := ty.base_type;
    end;
    -- "... predeclared functions that can be evaluated
    -- at compile time."
    -- Apparently all the predeclared functions can be evaluated at
    -- compile time.
    match builtin(...) begin
      e.expr_constant := true;
    end;
  end;
end;

-- an arithmetic operation is constant if all operands are constant
match ?e=binop(?op,?e1,?e2) begin
  e.expr_constant := e1.expr_constant and e2.expr_constant;
  if e.expr_constant then
    a1 : Constant := e1.constant_value;
    a2 : Constant := e2.constant_value;
    case op begin
      -- NB: the infix operators are defined for Constant!
      match log_or() begin
        -- NB: type checking is done elsewhere
        e.constant_value := a1 + a2; -- defined to be "or" for booleans;
      end;
      match log_and() begin
        e.constant_value := a1 * a2; -- "and" for booleans
      end;
      match plus() begin
        e.constant_value := a1 + a2;
      end;
      match minus() begin
        e.constant_value := a1 - a2;
      end;
      match times() begin
        e.constant_value := a1 * a2;
      end;
      match divide() begin
        e.constant_value := a1 / a2;
      end;
      match mod() begin
        e.constant_value := Constant$mod(a1,a2);
      end;
      match div() begin
        e.constant_value := Constant$div(a1,a2);
```

```
      end;
      match equal() begin
        e.constant_value := Constant$boolean_constant(a1 = a2);
      end;
      match not_equal() begin
        e.constant_value := Constant$boolean_constant(a1 /= a2);
      end;
      match less() begin
        e.constant_value := Constant$boolean_constant(a1 < a2);
      end;
      match less_equal() begin
        e.constant_value := Constant$boolean_constant(a1 <= a2);
      end;
      match greater() begin
        e.constant_value := Constant$boolean_constant(a1 > a2);
      end;
      match greater_equal() begin
        e.constant_value := Constant$boolean_constant(a1 >= a2);
      end;
      match in_set() begin
        case Constants${a1,a2} begin
          match {Constant$some_integer_constant(?v),
                 Constant$set_constant(?rep)} begin
            e.constant_value := Constant$boolean_constant(logbitp(v,rep)/=0);
          end;
        end;
      end;
    end; -- case op
  endif; -- if constant
end; -- match binop

-- similarly for unop:
match ?e=unop(?op,?e1) begin
  e.expr_constant := e1.expr_constant;
  if e.expr_constant then
    a1 : Constant := e1.constant_value;
    case op begin
      match plus() begin
        e.constant_value := a1;
      end;
      match minus() begin
        e.constant_value := -a1;
      end;
      match log_not() begin
        e.constant_value := -a1; -- "not" for booleans
      end;
    end; -- case op
  endif; -- if constant
end; -- match unop
```

```
-- a function call is constant if its function is constant
-- (in this case, "constant" means "predefined")
-- and each of its arguments is constant.
match ?e=funcall(?func,?args) begin
  e.expr_constant := func.expr_constant and
      (arg.expr_constant for arg in args);
end;

-- Now we also have to figure out the values of these
-- constant expressions:

match ?e=funcall(named_expr(?u),?arg_nodes) begin
  if e.expr_constant then
    args : Constants := {arg.constant_value for arg in arg_nodes};
    case u.use_decl begin
      match builtin(!make_symbol("ABS")) begin
        case args begin
          match {?a} begin
            e.constant_value := Constant$abs(a);
          end;
        end;
      end; -- ABS
      match builtin(!make_symbol("ASH")) begin
        case args begin
          match {Constant$some_integer_constant(?v1),
                 Constant$some_integer_constant(?v2)} begin
            e.constant_value := Constant$longint_constant(ash(v1,v2));
          end;
        end;
      end;
      match builtin(!make_symbol("CAP")) begin
        case args begin
          match {Constant$char_constant(?c)} begin
            e.constant_value := Constant$char_constant(to_upper(c));
          end;
        end;
      end; -- match CAP
      match builtin(!make_symbol("CHR")) begin
        case args begin
          match {Constant$some_integer_constant(?v)} begin
            if v >= 0 and v <= 255 then -- as defined in the manual
              e.constant_value := Constant$char_constant(int_char(v));
            else
              add_error(e,"Character constant out of range: " || v);
            endif;
          end;
        end;
      end; -- match CHR
      match builtin(!make_symbol("ENTIER")) begin
        case args begin
```

```
    match {?a} begin
      e.constant_value := Constant$to_longint(a);
    end;
  end;
end;
match builtin(!make_symbol("LEN")) begin
  -- LEN works on arrays, and thus on strings
  -- and also thus on characters.
  dimension : Integer := 0;
  case args begin
    match {?} begin end;
    match {?,Constant$some_integer_constant(?v)} begin
      if v < 0 then
        add_error(e,"LEN given negative dimension");
      else
        dimension := v;
      endif;
    end;
  else
    add_error(e,"LEN requires a constant dimension");
  end;
  -- now case on the argument value first
  -- to handle characters and strings
  case first(args) begin
    match Constant$char_constant(?) begin
      if dimension > 0 then
        add_error(e,"Strings have only one dimension");
      endif;
      e.constant_value := Constant$longint_constant(1);
    end;
    match Constant$string_constant(?s) begin
      if dimension > 0 then
        add_error(e,"Strings have only one dimension");
      endif;
      e.constant_value := Constant$longint_constant(length(s));
    end;
  else
    -- otherwise look at the type
    e.constant_value :=
        get_dimension(e,first(arg_nodes).expr_type,dimension);
  end;
end; -- match LEN
match builtin(!make_symbol("LONG")) begin
  case args begin
    match {Constant$real_constant(?v)} begin
      e.constant_value := Constant$longreal_constant(IEEEwiden(v));
    end;
    -- the specification is really strange:
    match {Constant$shortint_constant(?v)} begin
      e.constant_value := Constant$integer_constant(v);
```

```
    end;
    match {Constant$integer_constant(?v)} begin
      e.constant_value := Constant$longint_constant(v);
    end;
    -- otherwise undefined (that's what's bad with the spec)
  end;
end;
match builtin(!make_symbol("MAX")) begin
  case arg_nodes begin
    match {?ty} begin
      case ty.constant_type_value begin
        match shortint_type() begin
          e.constant_value := Constant$shortint_constant(max_shortint);
        end;
        match integer_type() begin
          e.constant_value := Constant$integer_constant(max_integer);
        end;
        match longint_type() begin
          e.constant_value := Constant$longint_constant(max_longint);
        end;
        match real_type() begin
          e.constant_value := Constant$real_constant(IEEEsingle$max);
        end;
        match longreal_type() begin
          e.constant_value :=
              Constant$longreal_constant(IEEEdouble$max);
        end;
        match set_type() begin
          e.constant_value := Constant$integer_constant(max_set);
        end;
      end; -- case type
    end;
  end; -- case arg_nodes
end; -- match MAX
match builtin(!make_symbol("MIN")) begin
  case arg_nodes begin
    match {?ty} begin
      case ty.constant_type_value begin
        match shortint_type() begin
          e.constant_value := Constant$shortint_constant(min_shortint);
        end;
        match integer_type() begin
          e.constant_value := Constant$integer_constant(min_integer);
        end;
        match longint_type() begin
          e.constant_value := Constant$longint_constant(min_longint);
        end;
        match real_type() begin
          e.constant_value := Constant$real_constant(-IEEEsingle$max);
        end;
```

```
        match longreal_type() begin
          e.constant_value :=
              Constant$longreal_constant(-IEEEdouble$max);
        end;
        match set_type() begin
          e.constant_value := Constant$integer_constant(0);
        end;
      end; -- case type
    end;
  end; -- case arg_nodes
end; -- match MIN
match builtin(!make_symbol("ODD")) begin
  case args begin
    match {Constant$some_integer_constant(?v)} begin
      e.constant_value := Constant$boolean_constant(odd(v));
    end;
  end;
end; -- ODD
match builtin(!make_symbol("ORD")) begin
  case args begin
    match {?a} begin
      e.constant_value := Constant$integer_constant(Constant$ord(a));
    end;
  end;
end; -- ORD
match builtin(!make_symbol("SHORT")) begin
  case args begin
    match {Constant$longreal_constant(?v)} begin
      e.constant_value := Constant$real_constant(IEEEnarrow(v));
    end;
    match {Constant$longint_constant(?v)} begin
      e.constant_value := Constant$integer_constant(v);
    end;
    match {Constant$integer_constant(?v)} begin
      e.constant_value := Constant$shortint_constant(v);
    end;
  end;
end; -- SHORT
match builtin(!make_symbol("SIZE")) begin
  case arg_nodes begin
    match {named_expr(?u)} begin
      case u.use_decl begin
        match type_decl(value:=?ty) begin
          if ty.fixed_size_type then
            e.constant_value :=
                Constant$longint_constant(ty.type_size);
          else
            add_error(e,"SIZE applied to open array");
          endif;
        end;
```

```
            end;
          end;
        end;
      end; -- SIZE
    end; -- case func
  endif; -- if constant
end; -- match funcall

-- this should be in a standard library:
function to_upper(c : Character) : Character begin
  if 'a' <= c and c <= 'z' then
    result := int_char(char_code(c)+32); -- ASCII specific
  else
    result := c;
  endif;
end;

-- Return the constant value for the given dimension of an array.
--       e is the source expression whose dimension is requested
--       t is the type of expression (or subarray on a recursive call)
--       dimension = 0 means the first dimension.
-- This procedure implements the predefined function LEN at compile-time.
-- It returns "nil" if the dimension could not be computed,
-- (for example if the corresponding dimension is "open").
var procedure get_dimension(e : remote Expression;
                            t : remote Type;
                            dimension : Integer) : Constant := Constant$nil
begin
  case t.base_type begin
    match any_type() begin end; -- don't cascade errors
    match fixed_array_type(?size,?subtype) begin
      if dimension = 0 then
        result := size.constant_value;
      else
        result := get_dimension(e,subtype,dimension-1);
      endif;
    end;
    match open_array_type(?subtype) begin
      if dimension = 0 then
        result := Constant$nil;
      else
        result := get_dimension(e,subtype,dimension-1);
      endif;
    end;
  else
    add_error(e,"LEN given array with not enough dimensions");
  end;
end;

end;
```

B.5 Checking for Correctness

```
-- Compile-time checks for Oberon2:
-- we use the information provided by the resolution phase
-- and by compile-time computations and check that there are no errors
module OBERON2_CHECK[T :: var OBERON2_TREE[],
                          var OBERON2_RESOLVE[T],
                          var OBERON2_COMPILE_COMPUTE[T]] extends T
begin
  signature ERROR_NODES :=
      {Declaration,Header,Receiver,Type,Statement,Case,
       CaseLabel,Expression,Element,Use}, var PHYLUM[];
  type Errors := BAG[String];
  [phylum Node::ERROR_NODES] begin
    collection attribute Node.errors : Errors;

    -- an internal way to report an error
    private procedure add_error(node : remote Node; message : String) begin
      node.errors :> {message};
    end;
  end;

  private;

  -----------------------------------------------------------------
  -- This module follows loosely along the pages of the Oberon-2 --
  -- reference manual [Moessenboeck and Wirth 1993]              --
  -----------------------------------------------------------------


  --- Section 4: Declaration and scope rules

  -- (Much of this has been handled by OBERON2_SYMTAB)

  -- "1. No identifier may denote more than one object within a given scope
  --  (i.e. no identifier may be declared twice in a block)"
  match ?d:Declaration begin
    if d.is_redeclaration then
      add_error(d,"This identifier already declared in this scope");
    endif;
  end;

  -- other cases (where a look fails for some reason)
  match ?using=use_name(?) begin
    if using.use_decl == nil and using.no_decl_reason /= "" then
      add_error(using,using.no_decl_reason);
    endif;
  end;

  -- warn about unused declarations:
```

```
collection attribute Declaration.decl_used : Boolean :> false, (or);
match ?u:Use begin
  if u.use_decl /= nil then
    u.use_decl.decl_used :> true;
  endif;
end;
match ?d=declaration(identifier(?,not_exported())) begin
  if not d.decl_used then
    add_error(d,"Warning: declaration not used");
  endif;
end;

-- "Identifiers marked with "-" in their declarations are read-only
--  in importing modules"
-- Presumably: this mark is only relevant for variables and fields.
pattern normally_writeable(info : ExportInfo) : Declaration
    := var_decl(identifier(?,?info),?),field(identifier(?,?info),?),
       var_formal(identifier(?,?info),?);

attribute Use.use_variable : Boolean := false;
-- NB: using importing variables is read as a field reference:
match ?using:Use begin
  case using.use_decl begin
    match !nil begin
      using.use_variable := true; -- avoid cascading error messages
    end;
    match ?d=normally_writeable(?info) begin
      case info begin
        match readonly_exported() begin
          using.use_variable :=
              using.declared_in_module == d.declared_in_module;
        end;
      else
        using.use_variable := true;
      end;
    end;
  else
    using.use_variable := false;
  end;
end;

--- Section 5: Constant declarations

-- "A constant declaration associates an identifier with a constant value"
match const_decl(?,?expr) begin
  ensure_constant(expr);
end;

procedure ensure_constant(x : remote Expression)
begin
```

```
  if not x.expr_constant then
    add_error(x,"not constant");
  endif;
end;


--- Section 6: Type declarations

-- "A structured type cannot contain itself"
-- This is already illegal because the type name would be a forward
-- reference.

-- Section 6.1: Basic types

-- Section 6.2: Array types

match fixed_array_type(?l,?) begin
  -- from the grammar, array length must be a constant expression
  ensure_constant(l);
  -- presumably: the length must be an integer type:
  ensure_assignment_compatible(l,longint);
end;

-- "[open_array_types] are restricted to pointer base types,
-- element types of open arrays and formal parameter types".
[T :: PHYLA] match ?parent:T=parent(?ty=open_array_type(...)) begin
  case parent begin
    match pointer_type(...) :? T begin end;
    match open_array_type(...) :? T begin end;
    match formal(...) :? T begin end;
  else
    add_error(ty,"Cannot use open array types in this context");
  end;
end;

-- Section 6.3: Record types

-- "a record type can be declared as an extension of another record type"
match record_type(?ty,?) begin
  case ty.base_type begin
    match record_type(...) begin end;
    match any_type() begin end; -- avoid multiple errors
    match no_type() begin end;
  else
    add_error(ty,"Can only extend record types");
  end;
end;

var function record_extends(t1,t2 : remote Type) : Boolean
begin
```

```
  if t1 == t2 then
    result := true;
  else
    case t1 begin
      match record_type(?ext,...) begin
        result := record_extends(ext.base_type,t2);
      end;
    else
      result := false;
    end;
  endif;
end;

-- "All identifiers declared in the extended record must be different
-- from the identifiers declared in its base type record(s)"
-- NB: OBERON2_SYMTAB has already computed an "overrides" attribute
-- that does most of the work for us:
-- Presumably: fields cannot override procedures either
match ?f=field(...) begin
  if f.overrides /== nil then
    add_error(f,"Already declared in a base type record");
  endif;
end;

-- Section 6.4: Pointer types

-- "[The pointer base type] must be a record or array type"
match pointer_type(?base) begin
  case base.base_type begin
    match any_type() begin end; -- avoid error cascades
    match array_type(...) begin end;
    match record_type(...) begin end;
  else
    add_error(base,"Pointer base types must be arrays or records");
  end;
end;

type Types := LIST[remote Type];

-- "If a type T1 is an extension and P1 is of type POINTER TO T1,
-- then P1 is also an extension of P [being POINTER TO T]"
var function pointer_extends(p1,p2 : remote Type) : Boolean
begin
  case Types${p1,p2} begin
    match {pointer_type(?t1),pointer_type(?t2)} begin
      result := record_extends(t1.base_type,t2.base_type);
    end;
  else
    result := false;
  end;
```

```
end;

-- Section 6.5: Procedure types

-- "If a procedure is assigned to a variable of type T, ...
-- "P must not be a predeclared or type-bound procedure nor may it be
-- local to another procedure".
-- Presumably: local procedures may not be passed as procedures either.
-- Apparently this means that local procedures or type-bound
-- procedures may not be used in any way except to be called.

attribute Declaration.proc_is_local : Boolean := false;
match proc_decl(body:=block(decls:={...,?d=proc_decl(...),...})) begin
  d.proc_is_local := true;
end;

attribute Expression.called : Boolean := false;
match funcall(?proc,?) begin
  proc.called := true;
end;
match call_stmt(?call) begin
  call.called := true;
end;

match ?e=named_expr(?using) begin
  case using.use_decl begin
    match ?proc=proc_decl(...) begin
      if proc.proc_is_local and not e.called then
        add_error(e,"Illegal use of local procedure");
      endif;
    end;
  end;
end;

match ?e=fref(?,?using,...) begin
  case using.use_decl begin
    match proc_decl(header:=header(receiver:=receiver(...))) begin
      if not e.called then
        add_error(e,"Illegal use of type-bound procedure");
      endif;
    end;
  end;
end;


-- Section 7: Variable declarations


-- Section 8: Expressions
```

```
-- NB: copy errors from the compile compute module:
match ?e:Expression begin
  e.errors :> {e.T$errors...};
end;

-- 8.1 Operands

-- NB: the manual does not say what a variable is.
-- I inferred the following rules.
attribute Expression.expr_variable : Boolean := false;

procedure ensure_variable(v : remote Expression) begin
  if not v.expr_variable then
    add_error(v,"Not a variable");
  endif;
end;

match ?e=named_expr(?u) begin
  e.expr_variable := u.use_variable;
end;

-- "If a designates an array, then a[e] ... The type of e must be
-- an integer type"
-- Presumably: a must be an array type (or pointer to same)
match aref(?a,?e) begin
  ensure_assignment_compatible(e,longint);
  case a.expr_type begin
    match array_type(...) begin end;
    match any_type(...) begin end;
    match pointer_type(?ty) begin
      case ty.base_type begin
        match array_type(...) begin end;
        match any_type(...) begin end;
      else
        add_error(a,"not a pointer to an array");
      end;
    end;
  else
    add_error(a,"not an array");
  end;
end;

-- "If r designates a record then r.f denotes the field ..."
-- Presumably: r must be a record type (or pointer to the same)
match fref(?r,?,...) begin
  case r.expr_type begin
    match record_type(...) begin end;
    match any_type(...) begin end;
    match pointer_type(?ty) begin
      case ty.base_type begin
```

```
        match record_type(...) begin end;
        match any_type(...) begin end;
      else
        add_error(r,"not a pointer to a record");
      end;
    end;
  else
    add_error(r,"not a record");
  end;
end;

-- "If p designates a pointer, p^ designates the variable which is
-- referenced by p"
-- Presumably: p must be of pointer_type
match ?e=fetch(?p) begin
  e.expr_variable := true;
  case p.expr_type begin
    match pointer_type(...) begin end;
    match any_type(...) begin end;
  else
    add_error(p,"not a pointer");
  end;
end;

-- "If a or r and read-only then a[e] and r.f are read-only"
-- Apparently this ends up meaning:
-- a is variable <=> a[e] is variable
-- r is variable and f is variable  <=> r.f is variable
-- Note that this rule seems to imply that if a^[e] is abbreviated
-- as a[e] then it will have more restricted readonly characteristics,
-- but the example given excludes this interpretation.
match ?result=aref(?a,?) begin
  case a.expr_type begin
    match pointer_type(...) begin
      result.expr_variable := true;
    end;
  else
    result.expr_variable := a.expr_variable;
  end;
end;
match ?result=fref(?r,?f,...) begin
  -- we need to restrict it so it only applies for fields:
  case f.use_decl begin
    match field(...) begin
      case r.expr_type begin
        match pointer_type(...) begin
          result.expr_variable := f.use_variable;
        end;
      else
        result.expr_variable := r.expr_variable and f.use_variable;
```

```
      end;
    end;
  else
    -- an external module reference:
    result.expr_variable := f.use_variable;
  end;
end;


-- "[A type guard] is applicable if:
--  1. v is a variable parameter of record type or v is a pointer and if
--  2. T is an extension of the static type of v"
-- These conditions are identical to those for type tests, so we write
-- a procedure to do the test:
procedure ensure_type_guard_applicable(v : remote Expression;
                                       uT : remote Use)
begin
  case uT.use_decl begin
    match !nil begin end;
    match type_decl(?,?T) begin
      -- case 1b & 2
      if not pointer_extends(v.expr_type,T.base_type) then
        case v.expr_type begin
          match any_type() begin end;
          match pointer_type(...) begin
            add_error(T,"Not a base pointer type");
          end;
          match record_type(...) begin
            case v begin
              match named_expr(?uv) begin
                case uv.use_decl begin
                  match !nil begin end;
                  -- case 1a & 2
                  match var_formal(...) begin
                    if not record_extends(v.expr_type,T.base_type) then
                      add_error(T,"Not a base record type");
                    endif;
                  end;
                else
                  add_error(v,"Not a var formal");
                end;
              end; -- match named_expr
            else
              add_error(v,"Not a pointer");
            end; -- case v
          end; -- match record-type
        else
          add_error(v,"Pointer or record required");
        end; -- case v.expr_type
      endif;
```

```
    end; -- match type_decl(...)
  else
    add_error(uT,"Type identifier required");
  end; -- case uT.use_decl
end;


-- NB: some type guards are read as function calls, but OBERON2_RESOLVE
-- has defined a pattern for us.
match ?e=actual_type_guard(?v,?uT) begin
  ensure_type_guard_applicable(v,uT);
  e.expr_variable := v.expr_variable;
end;

-- "The actual parameters must correspond to the formal parameters
--  as in proper procedure calls (see 10.1)."
-- See section 10.1

-- Section 8.2: Operators

-- "The operands must be expression compatible with respect to the operator
--  (see App. A)."
-- See section A: expression compatible OBERON2_TYPE

pattern op(op:Operator; arg:Expression) : Expression :=
    unop(?op,?arg),binop(?op,?arg,?),binop(?op,?,?arg);

-- Section 8.2.1: Logical operators
-- "These operators apply to BOOLEAN operands ..."
match op(logical_operator(),?arg) begin
  ensure_assignment_compatible(arg,boolean);
end;

-- Section 8.2.2 Arithmetic operators and Section 8.2.3 Set operators
-- "The operators +,-,*, and / apply to operands of numeric type"
-- (but also to sets)
pattern simple_arithmetic_op() : Operator := plus(),minus(),times(),divide();
match op(simple_arithmetic_op(),?arg) begin
  if arg.expr_type /= set then
    ensure_assignment_compatible(arg,longreal);
  endif;
end;
-- but if one is a set, the other must be too:
match binop(simple_arithmetic_op(),?arg1,?arg2) begin
  if arg1.expr_type = set then
    ensure_assignment_compatible(arg2,set);
  elsif arg2.expr_type = set then
    ensure_assignment_compatible(arg1,set);
  endif;
end;
```

```
-- "The operators DIV and MOD apply to integer operands only."
match op(integer_operator(),?arg) begin
  ensure_assignment_compatible(arg,longint);
end;

-- "A set constructor defines the value of a set by listing its elements
--  between curly brackets.  The elements must be integers in the range
--  0..MAX(SET).  A range a..b denotes all integers in the interval
--  [a,b]"
-- The type can be checked at this time but the values at worst
-- can only be checked at runtime:
pattern element(x : Expression) : Element
    := single_element(?x),range_element(?x,?),range_element(?,?x);
match element(?x) begin
  ensure_assignment_compatible(x,longint);
end;

-- Section 8.2.4 Relations
-- "The relations =, #, <, <=, >, and >= apply to the numeric types
--  CHAR, strings, and character arrays ... .  The relations = and #
-- also apply to BOOLEAN and SET, as well as to pointer and procedure
-- types (including the value NIL)."

type TypeList := LIST[remote Type];

-- check < <= > >=
pattern simple_comparison_operator() : Operator
    := less(), less_equal(), greater(), greater_equal();
match ?e=binop(simple_comparison_operator(),?arg1,?arg2) begin
  case TypeList${arg1.expr_type,arg2.expr_type} begin
    match {numeric_type(),numeric_type()} begin end;
    match {string_type(),string_type()} begin end;
    match {...,any_type(),...} begin end;
  else
    add_error(e,"incompatible operands to comparison");
  end;
end;
-- check # and =
match ?e=binop(equality_operator(),?arg1,?arg2) begin
  case TypeList${arg1.expr_type,arg2.expr_type} begin
    match {numeric_type(),numeric_type()} begin end;
    match {string_type(),string_type()} begin end;
    match {boolean_type(),boolean_type()} begin end;
    match {set_type(),set_type()} begin end;
    match {nil_type(),pointer_type(...)} begin end;
    match {pointer_type(...),nil_type()} begin end;
    match {pointer_type(?t0),pointer_type(?t1)}
        if record_extends(t0,t1) or record_extends(t1,t0)
    begin end;
    match {nil_type(),proc_type(...)} begin end;
```

```
    match {proc_type(...),nil_type()} begin end;
    match {proc_type(?h1),proc_type(?h2)}
        if headers_match(h1,h2)
    begin end;
    match {...,any_type(),...} begin end;
  else
    add_error(e,"incompatible operands to equality test");
  end;
end;

-- "x IN s ... x must be of an integer type, and s of type SET."
match ?e=binop(in_set(),?x,?s) begin
  ensure_assignment_compatible(x,longint);
  ensure_assignment_compatible(s,set);
end;

-- "v IS T stands for "the dynamic type of v is T (or an extension of T)"
--  and is called a type test.  It is applicable if
--  1: v is a variable parameter of record type or v is a pointer, and if
--  2: T is an extension of the static type of v"
-- These conditions are the same as for type guards
match is_test(?v,named_type(?uT)) begin
  ensure_type_guard_applicable(v,uT);
end;


--- Section 9: Statements

-- Section 9.1: Assignments

-- "The expression must be assignment compatible with the variable"
match assign_stmt(?lhs,?rhs) begin
  ensure_variable(lhs);
  ensure_assignment_compatible(rhs,lhs.expr_type);
end;

-- Section 9.2: Procedure calls
-- "A procedure call activates a procedure."
-- Presumably: the procedure must be a proper procedure
match call_stmt(?call) begin
  case call.expr_type begin
    match any_type() begin end;
    match no_type() begin end;
  else
    add_error(call,"Not a proper procedure call");
  end;
end;

-- "If a formal parameter is a variable parameter, the corresponding
--  actual parameter must be a designator denoting a variable..."
```

```
-- Presumably: the rules for proper procedure and function procedure
-- parameters is the same.  See Section 10.1

-- Section 9.3: Statement sequences

-- Section 9.4: If statements
-- NB: elsifs are desugared before we see them here.
-- "The Boolean expression preceding a statement sequence is called its
--  guard."
-- Presumably: the expression must be of boolean type
match if_stmt(?guard,...) begin
  ensure_assignment_compatible(guard,boolean);
end;

-- Section 9.5: Case statements
-- "The case expression must either be of an integer type that includes
--  the type of all case labels, or both the case expression and the case
--  labels must be of type CHAR."
-- This can be restated as:
--  the case expression must be of integer type or of type CHAR
--  the case labels must be included in the type of the case expression
match case_stmt(?expr,...) begin
  case expr.expr_type begin
    match any_type() begin end;
    match shortint_type() begin end;
    match integer_type() begin end;
    match longint_type() begin end;
    match char_type() begin end;
  else
    add_error(expr,"Required integer or character");
  end;
end;
pattern case_label(e : Expression) : CaseLabel
    := single_label(?e),range_label(?e,?),range_label(?,?e);
match case_stmt(?sub,{...,case_clause({...,case_label(?e),...},?),...},?)
begin
  ensure_assignment_compatible(e,sub.expr_type);
end;

-- "Case labels are constants and ..."
match case_label(?e) begin
  ensure_constant(e);
end;
-- "... no value may occur more than once."
-- NB: Actually building up the sets is dangerous because
-- the sets could be very large (for example -1000...1000)
-- So instead we gather up all the cases and then do checks on them:
type CaseSet := ORDERED_SET[remote CaseLabel]((==),(<<));
match case_stmt(?,?clauses,?) begin
  collection labels : CaseSet;
```

```
  -- first we flatten the case statement
  -- and select only labels without errors
  for clauses begin
    match Cases${...,case_clause({...,?label,...},?),...} begin
      case label begin
        match single_label(?) begin
          labels :> {label};
        end;
        match range_label(?e1,?e2) begin
          if e1.constant_value > e2.constant_value then
            add_error(label,"Empty range");
          else
            labels :> {label};
          endif;
        end;
      end; -- case label
    end;
  end; -- for clauses
  -- then do a cross check
  for labels begin
    match {...,single_label(?e1),...,?lab2=single_label(?e2),...}
    begin
      if e1.constant_value = e2.constant_value then
        add_error(lab2,"Duplicate case label");
      endif;
    end;
    match {...,single_label(?e1),...,?lab2=range_label(?e2,?e3),...}
    begin
      if e1.constant_value >= e2.constant_value and
          e1.constant_value <= e3.constant_value then
        add_error(lab2,"Includes previous case label");
      endif;
    end;
    match {...,range_label(?e1,?e2),...,?lab2=single_label(?e3),...}
    begin
      if e3.constant_value >= e1.constant_value and
          e3.constant_value <= e2.constant_value then
        add_error(lab2,"Included in previous case label");
      endif;
    end;
    match {...,range_label(?e1,?e2),...,?lab2=range_label(?e3,?e4),...}
    begin
      -- if overlapping, we give an error
      if e2.constant_value >= e3.constant_value and
         e1.constant_value <= e4.constant_value then
        add_error(lab2,"Overlaps previous case label");
      endif;
    end;
  end; -- for
end; -- match case_stmt
```

```
-- Section 9.6: While statements

-- "... the Boolean expression (its guard) ..."
match while_stmt(?guard,...) begin
  ensure_assignment_compatible(guard,boolean);
end;

-- Section 9.7: Repeat statements

-- "... a condition specified by a Boolean expression ..."
match repeat_stmt(?,?guard) begin
  ensure_assignment_compatible(guard,boolean);
end;

-- Section 9.8: For statements

-- "... while a progression of values is assigned to an integer variable ..."
-- "The statement FOR v := beg TO end BY step DO statements END
--  is equivalent to
--  temp := end; v := beg;
--  IF step > 0 THEN
--    WHILE v <= temp DO statements; v := v + step END
--  ELSE
--    WHILE v >= temp DO statements; v := v + step END
--  END
--  temp has the same type as v, step must be a nonzero constant expression."
-- This equivalence adds the following implied condition:
--   beg, end and step are assignment compatible with v
match for_stmt(?v,?start,?finish,?step,?) begin
  ensure_variable(v);
  ensure_assignment_compatible(v,longint);
  ensure_assignment_compatible(start,v.expr_type);
  ensure_assignment_compatible(finish,v.expr_type);
  ensure_assignment_compatible(step,v.expr_type);
  case step begin
    match no_expr() begin end;
  else
    ensure_constant(step);
    case step.constant_value begin
      match Constant$some_integer_constant(0) begin
        add_error(step,"Cannot step by zero");
      end;
    end;
  end;
end;

-- Section 9.9: Loop statements

-- Section 9.10: return and exit statements
```

```
-- "The type of the expression must be assignment compatible (see App. A)
--  with the result type specified in the procedure heading (see Ch. 10)"
-- NB: proper procedures have no_type which is incidentally the type
-- of no_expr for empty RETURN statements.
attribute Statement.return_procedure : remote Declaration := nil;
match ?p=proc_decl(?,block(?,{...,?stmt,...})) begin
  stmt.return_procedure := p;
end;
match ?p:Statement=parent(Statements${...,?c:Statement,...}) begin
  c.return_procedure := p.return_procedure;
end;
match ?s=return_stmt(?value) begin
  case s.return_procedure begin
    match !nil begin
      -- Presumably: RETURN is illegal elsewhere (say in a module body)
      add_error(s,"RETURN statements must be inside PROCEDUREs");
    end;
    match proc_decl(header:=header(result:=?rt)) begin
      ensure_assignment_compatible(value,rt.base_type);
    end;
  end;
end;

-- "An exit statement ... specifies termination of the enclosing loop
--  statement..."
-- Presumably EXIT is illegal elsewhere
attribute Statement.exit_loop : remote Statement := nil;
match ?l=loop_stmt({...,?sub,...}) begin
  sub.exit_loop := l;
end;
match ?p:Statement=parent(Statements${...,?c:Statement,...}) begin
  c.exit_loop := p.exit_loop;
end;
match ?e=exit_stmt(...) begin
  if e.exit_loop == nil then
    add_error(e,"EXIT statements must nest inside LOOPs");
  endif;
end;

-- Section 9.11: With statements
-- "With statements execute a statement sequence depending on the result
-- of a type test and apply a type guard to every occurrence of the tested
-- variable within this statement sequence."
-- NB: WITH has been desugared like IF so we only need to
-- worry about one guard per WITH statement.
-- We require that the type test be applicable:
match with_stmt(?v,named_type(?uT),...) begin
  ensure_type_guard_applicable(v,uT);
end;
```

```
-- The fact that an implicit type guard is added throughout the body
-- is already taken care of in module EXPR_TYPE.
-- (actually in discussion on comp.lang.oberon, it seems more like that
-- the variable gets a new type during the body, thus making assignments
-- to possibly unsafe values illegal unless they have type guards too)
--
-- It seems that in the distributed Oberon2 compiler the static type of
-- the variable is temporarily modified while processing the body;
-- this action seems erroneous if the variable is a formal because it
-- makes certain perfectly safe recursive calls illegal.


-- Section 10: Procedure declarations

-- "The body of a function procedure must contain a return statement
-- that defines its result"
match ?p=proc_decl(header(result:=?rt),block(?,?body)) begin
  case rt begin
    match no_type() begin end;
  else -- yes, a function procedure
    case body begin
      match ancestor(return_stmt(...)) begin end;
    else -- no return statement
      add_error(p,"No RETURN statement for PROCEDURE");
    end;
  end;
end;

-- "The formal parameter lists of the forward declaration and actual
--  declaration must match (see App. A)"
match ?d=forward(?h1) begin
  case d.forwarding_decls begin
    -- NB: case chooses first match, that is the first procedure
    -- in the set.
    match {...,?p=proc_decl(?h2,?),...} begin
      if not headers_match(h1,h2) then
        add_error(p,"Does not match forward declaration");
      endif;
      for x: remote Declaration in d.forwarding_decls begin
        if x /== p then
          case x begin
            match proc_decl(...) begin
              add_error(x,"Forward already satisfied");
            end;
          else
            add_error(x,"Redeclaration of forward with non-procedure");
          end;
        endif;
      end;
    end;
```

```
  else
    if not d.is_builtin then -- predefine procedures aren't satisfied
      add_error(d,"Forward declaration not satisfied");
      for x : remote Declaration in d.forwarding_decls begin
        add_error(x,"Redeclaration of forward with non-procedure");
      end;
    endif;
  end;
end;

-- "The result type of a procedure can neither be a record nor an array"
match header(result:=?rt) begin
  case rt.base_type begin
    match array_type(...) begin
      add_error(rt,"Return type of a procedure may not be an array");
    end;
    match record_type(...) begin
      add_error(rt,"Return type of a procedure many not be a record");
    end;
  end;
end;

-- Section 10.1: Formal parameters

-- different headers for special builtins.
pattern INC_DEC_header(name : IdentDef) : Header
    := header(?name,formals:={var_formal(...),
                              opt_formal(...)});
pattern NEW_header() : Header
    := header(formals:={?,rest_formal(...)});

-- a pattern matching all types that can be considered strings
var pattern string_type() : Type
    := array_type(?c) if is_char_type(c.base_type),
       char_type();
var function is_char_type(c : remote Type) : Boolean begin
  case c.base_type begin
    match char_type() begin
      result := true;
    end;
  else
    result := false;
  end;
end;

type TypeBag := BAG[remote Type];
match ?e=funcall(?proc,?actuals)
    if not e.is_type_guard -- avoid type guards that look like funcalls
begin
  actual_types : TypeBag :=
```

```
    {a.expr_type for a : remote Expression in actuals};
case proc.expr_header begin
  match !nil begin
    add_error(e,"Not a procedure call");
  end;
  -- check for predefined things with optional parameters
  --   INC  DEC
  match INC_DEC_header(...) and
      header(formals:={?fixed,opt_formal(shape:=?opt_type)})
  begin
    for actuals begin
      match {?a,...} begin
        ensure_variable(a);
        ensure_assignment_compatible(a,longint);
      end;
      match {?,?opt,...} begin
        ensure_assignment_compatible(opt,opt_type.base_type);
      end;
      match {?,?,?,...} begin
        add_error(e,"Too many actual parameters to builtin procedure");
      end;
    end;
  end;
  --   LEN  ASSERT
  match header(formals:={?fixed,opt_formal(shape:=?opt_type)}) begin
    for actuals begin
      match {?a,...} begin
        ensure_actual_compatible(a,fixed);
      end;
      match {?,?opt,...} begin
        ensure_assignment_compatible(opt,opt_type.base_type);
      end;
      match {?,?,?,...} begin
        add_error(e,"Too many actual parameters to builtin procedure");
      end;
    end;
  end;
  -- special case for NEW:
  match NEW_header() begin
    case actuals begin
      match {} begin
        add_error(e,"Too few parameters to builtin NEW");
      end;
      -- "NEW(v) [v is] pointer to record or fixed array"
      match {?v} begin
        ensure_variable(v);
        case v.expr_type begin
          match pointer_type(record_type(...)) begin end;
          match pointer_type(fixed_array_type(...)) begin end;
          match pointer_type(open_array_type(...)) begin
```

```
              add_error(e,"Need to give dimensions for NEW of open array");
            end;
            match pointer_type(...) begin end;
            match any_type() begin end;
          else
            add_error(e,"NEW must be given a valid pointer type");
          end;
        end; -- single argument NEW
        -- "NEW(v,x0,...,x1) v: pointer to open array; xi:integer type"
        match {?v,...} begin
          ensure_variable(v);
          case v.expr_type begin
            match pointer_type(?o=open_array_type(...)) begin
              if o.open_dimensions /= length(actuals)-1 then
                add_error(e,"Wrong number of dimensions (expected " ||
                              o.open_dimensions || ")");
              endif;
              for actuals begin
                match {?,...,?range,...} begin
                  ensure_assignment_compatible(range,longint);
                end;
              end;
            end;
            match pointer_type(...) begin
              add_error
                  (e,"Multi-argument NEW may only be used on open arrays");
            end;
            match any_type() begin end;
          else
            add_error(e,"NEW must be given a valid pointer type");
          end;
        end; -- multiple argument NEW
      end; -- case NEW actuals
    end; -- match NEW_header
    -- now the normal case
    match header(formals:=?formals) begin
      m : Integer := min(length(formals),length(actuals));
      if length(formals) > length(actuals) then
        add_error(e,"Too few actual parameters to function (expected " ||
                      length(formals) || ")");
      elsif length(formals) < length(actuals) then
        add_error(e,"Too many actual parameters to function (expected " ||
                      length(formals) || ")");
      endif;
      for i : Integer in 0..m-1 begin
        ensure_actual_compatible(nth(i,actuals),nth(i,formals));
      end;
    end;
  end; -- case header
end; -- match funcall
```

```
-- "Let Tf be the type of a formal parameter f (not an open array) and
--  Ta the type of the corresponding actual parameter a.  For
--  variable parameters, Ta must be the same as Tf, or Tf must be a
--  record type and Ta an extension of Tf.  For value parameters,
--  a must be assignment compatible with f (see App. A)
--     If Tf in an open array, then a must be array compatible with f"
var procedure ensure_actual_compatible(actual : remote Expression;
                                       formal : remote Declaration)
begin
  -- Presumably we check that var actuals are variable
  -- even in the case of an open array type:
  case formal begin
    match var_formal(...) begin
      ensure_variable(actual);
    end;
  end;
  -- now the type check:
  case formal begin
    match formal(?,?shape=open_array_type(...)) begin
      if not array_compatible(actual.expr_type,shape.base_type) then
        add_error(actual,"not array compatible");
      endif;
    end;
    match var_formal(?,?shape) begin
      -- The technical definition of "same" is a little shaky
      -- so I'll use the idea of identical base type:
      if actual.expr_type.base_type /== shape.base_type and
          not record_extends(actual.expr_type.base_type,shape.base_type) then
        add_error(actual,"Not compatible with variable formal");
      endif;
    end;
    match value_formal(?,?shape) begin
      ensure_assignment_compatible(actual,shape.base_type);
    end;
  end;
end;

attribute Type.open_dimensions : Integer := 0;
match ?o=open_array_type(?e) begin
  o.open_dimensions := e.open_dimensions+1;
end;

-- Section 10.2: Type-bound procedures

-- "Globally declared procedures may be associated with a record type
--  declared in the same module"
attribute Declaration.globally_declared : Boolean := false;
[T :: PHYLA] attribute T.declared_in_module : remote Declaration := nil;
match ?m=module_decl(body:=block({...,?decl,...},?)) begin
```

```
  decl.globally_declared := true;
  decl.declared_in_module := m;
end;
[phylum P,C :: PHYLA] begin
  match ?p:P=parent(?c:C) begin
    c.declared_in_module := p.declared_in_module;
  end;
end;
[phylum P,C :: PHYLA; SP :: SEQ_PHYLA, SEQUENCE[C]] begin
  match ?p:P=parent(SP${...,?c:C,...}) begin
    c.declared_in_module := p.declared_in_module;
  end;
end;
match ?p=bound_proc_decl(?,?rec=formal(shape:=?shape)) begin
  if not p.globally_declared then
    add_error(p,"Type-bound procedures must be globally declared");
  elsif shape.base_type.declared_in_module /== p.declared_in_module then
    add_error(p,"Type-bound procedure declared for external record type");
  endif;
  -- "The receiver may either be a variable parameter of record type T or
  --  a value parameter of type POINTER to T (where T is a record type)."
  case rec begin
    match var_formal(...) begin
      case shape.base_type begin
        match record_type(...) begin end;
        match any_type() begin end;
      else
        add_error(p,"Receiver type must be a record type");
      end;
    end;
    match value_formal(...) begin
      case shape.base_type begin
        match pointer_type(?t1) begin
          case t1.base_type begin
            match record_type(...) begin end;
            match any_type() begin end;
          else
            add_error(p,"Receiver type must be a pointer to a record");
          end;
        end;
      else
        add_error(p,"Receiver type must be a pointer to a record");
      end;
    end;
  end;
end;

-- "... However a procedure P' (with the same name as P) may be explicitly
--  bound to T1 in which case it overrides the binding of P. P' is
--  considered a redefinition of P for T1.  The formal parameters of P
```

```
--  P' must match (see App. A).  If P and T1 are exported, P' must be
--  exported."
-- The last condition is a little complicated: what if T' = POINTER TO T is
-- exported but not T itself?  I will take the condition literally and
-- say then the condition does not apply.
match ?pp=proc_decl(header:=?h1) and bound_proc_decl(?,formal(shape:=?shape))
begin
  case pp.overrides begin
    match field(...) begin
      add_error(pp,"Cannot override field");
    end;
    match ?p=proc_decl(header:=?h2) begin
      if not headers_match(h1,h2) then
        add_error(p,"Overriding procedure header does not match");
      endif;
      case p begin
        match bound_proc_decl(identifier(export_info:=exported()),...)
            if shape.base_type.type_exported begin
          case pp begin
            match bound_proc_decl(identifier(export_info:=exported()),...)
            begin end;
          else
            add_error(p,"Overriding procedure must be exported");
          end;
        end;
      end; -- case proc for export info
    end; -- overrides a procedure
  end;
end;

attribute Type.type_exported : Boolean := false;
match type_decl(identifier(export_info:=exported()),?rt=record_type(...))
begin
  rt.type_exported := true;
end;

-- "If r is a receiver parameter declared with type T, r.P^ denotes the
--  (redefined) procedure bound to the base type of T."
-- The parser checks for <expr>.<id>^(...) and generates the "super"
-- fref.
match ?e=fref(?r,?uP,!true) begin
  case r begin
    match named_expr(?u) begin
      case u.use_decl begin
        match !nil begin end;
        match ?d if d.is_receiver_formal begin end;
      else
        add_error(r,"Can only fetch redefined procedures for receivers");
      end;
    end;
```

```
  end;
  case uP.use_decl begin
    match !nil begin end;
    match ?p=proc_decl(...) begin
      if p.overrides == nil then
        add_error(e,"No redefined procedure");
      endif;
    end;
  else
    add_error(uP,"Not a procedure");
  end;
end;

attribute Declaration.is_receiver_formal : Boolean := true;
match receiver(?f) begin
  f.is_receiver_formal := true;
end;

-- "In a forward declaration of type-bound procedure ..."
-- (checked already)


-- Section 11: Modules

-- "A module must not import itself."
-- "... cyclic import of modules is illegal"
-- Both handles already in ob2-symtab (by interfering with the import lookup)


-- Appendix A: Definitions of terms

-- Same Types
-- "Two variables a and b with types Ta and Tb are of the same type if
--  1. Ta and Tb are both denoted by the same type identifier, or
--  2. Ta is declared equal Tb in a type declaration of the form Ta = Tb, or
--  3. a and b appear in the same identifier list in a variable, record
--     field or formal parameter declaration and are not open arrays."
-- The third case is taken care of by the abstract tree generator.
--
-- This definition seems buggy in several ways:
-- 1> It only works for named variables, but the definition requires it
--    to work for generalized variables (for var parameters) or even for
--    expressions (same type is used in several other definitions).
-- For example:
--   type Apples = Integer;
--   var x : Apples;
-- Is x+1 the "same" type as Integer?
--
-- 2> It is not symmetric, rule 2 is one direction only
-- similarly, it is not transitive.  I could understand either lack
```

```
-- (except it makes this rather more complex) if the other were
-- there.
--
-- type Apples = Integer;
--      NewApples = Apples;
-- Apples and Integer are the "same type", NewApples and Apples
-- are the "same" type, but NewApples and Integer are not the same type.
-- Neither are Integer and Apples, or Apples and NewApples.

function same_type(t1,t2 : remote Type) : Boolean begin
  b1 : remote Type := t1.base_type;
  b2 : remote Type := t2.base_type;
  result := b1 == b2 or b1 == any or b2 == any;
end;

-- Equal types
-- "Two types T1 and Tb are equal if
--  1. Ta and Tb are the same type, or
--  2. Ta and Tb are open array types with equal element types, or
--  3. Ta and Tb are procedure types whose formal parameter lists match"
function equal_types(t1,t2 : remote Type) : Boolean begin
  if same_type(t1,t2) then
    result := true;
  endif;
  case t1 begin
    match open_array_type(?et1) begin
      case t2 begin
        match open_array_type(?et2) begin
          result := equal_types(et1,et2);
        end;
      end;
    end;
    match proc_type(?h1) begin
      case t2 begin
        match proc_type(?h2) begin
          result := headers_match(h1,h2);
        end;
      end;
    end;
  end;
  -- otherwise they do not match:
  result := false;
end;

-- Type inclusion
-- "Numeric types include (the values of) smaller numeric types
--  according to the following hierarchy."
function type_includes(t1,t2 : remote Type) : Boolean begin
  t1_index : Integer := numeric_type_index(t1);
  t2_index : Integer := numeric_type_index(t2);
```

```
  result := t1_index /= 0 and t2_index /= 0 and t1_index > t2_index;
end;
private function numeric_type_index(t : remote Type) : Integer begin
  case t.base_type begin
    match !longreal begin result := 5; end;
    match !real begin result := 4; end;
    match !longint begin result := 3; end;
    match !integer begin result := 2; end;
    match !shortint begin result := 1; end;
  else
    result := 0;
  end;
end;

-- Type extension
-- "Given a type declaration Tb = RECORD(Ta) ... END, Tb is a direct
--  extension of Ta, and Ta is a direct base type of Tb.  A type Tb
--  is an extension of a type Ta (Ta is a base type of Tb) if
--  1. Ta and Tb are the same types, or
--  2. Tb is a direct extension of an extension of Ta, [or
--  3.] If Pa = POINTER to Ta and Pb = POINTER to Tb, Pb is an extension
--  of Pa (Pa ius a base type of Pb) if Tb is an extension of Ta."
-- Most of this was handled by the attribute record_extends, and
-- pointer_extends:
var function type_extends(t1,t2 : remote Type) : Boolean :=
    record_extends(t1,t2) or pointer_extends(t1,t2);
-- I don't believe this function is used anywhere.

-- Assignment compatible
-- "An Expression e of type Te is assignment compatible with a variable
--  v of type Tv if one of the following conditions hold:
--  1. Te and Tv are the same type
--  2. Te and Tv are numeric types and Tv includes Te
--  3. Te and Tv are record types and Te is an extension of Tv and the
--     dynamic type of v is Tv.
--  4. Te and Tv are pointer types and Te is an extension of Tv
--  5. Tv is a pointer or a procedure type and e is NIL
--  6. Tv is ARRAY n of CHAR and e is a string constant with m<n chars
--  7. Tv is a procedure type and e is the name of a [global] procedure whose
--     formal parameters match those of Tv"
var procedure ensure_assignment_compatible(e : remote Expression;
                                           Tv : remote Type)
begin
  Te : remote Type := e.expr_type;
  if same_type(Te,Tv) or
      type_includes(Tv,Te) or
      record_extends(Te,Tv) or
      pointer_extends(Te,Tv) or
      is_pointer_or_proc(Tv) and is_nil(Te) or
      string_type_is_big_enough(Tv,e) or
```

```
      proc_type_matches_global(Tv,e) then
  else
    add_error(e,"type mismatch");
  endif;
end;

function is_pointer_or_proc(t : remote Type) : Boolean begin
  case t begin
    match pointer_type(...) begin
      result := true;
    end;
    match proc_type(...) begin
      result := true;
    end;
  else
    result := false;
  end;
end;

function is_nil(t : remote Type) : Boolean begin
  case t begin
    match nil_type(...) begin
      result := true;
    end;
  else
    result := false;
  end;
end;

var function string_type_is_big_enough(Tv : remote Type;
                                       e : remote Expression)
    : Boolean := false
begin
  case Tv begin
    match fixed_array_type(?n,?ct) if is_char_type(ct) begin
      case e.constant_value begin
        match Constant$string_constant(?s) begin
          if Constant$ord(n.constant_value) > length(s) then
            result := true;
          endif;
        end;
      end;
    end;
  end;
end;

var function proc_type_matches_global(Tv : remote Type;
                                      e : remote Expression)
    : Boolean := false
begin
```

```
  case Tv begin
    match proc_type(?h1) begin
      result := headers_match(h1,e.expr_header);
    end;
  end;
end;

-- Appendix A: Array compatible
var function array_compatible(Ta,Tf : remote Type) : Boolean := false
begin
  -- "An actual parameter a of type Ta is array compatible with a formal
  --  parameter f of type Tf if:
  --  1. Tf and Ta are the same type, or
  if same_type(Tf,Ta) then
    result := true;
  else
    case Tf begin
      --  2. Tf is an open array, Ta is any array, and
      --     their element types are array compatible or
      --  3. Tf is ARRAY OF CHAR and a is a string."
      --  (The third is handled as part of # because strings are
      --   considered open arrays of characters in this compiler.)
      match open_array_type(?eTf) begin
        case Ta begin
          match array_type(?eTa) begin
            result := array_compatible(eTa,eTf);
          end;
        end;
      end;
    end;
  endif;
end;

-- Appendix A: "Matching formal parameter lists"
var function headers_match(h1,h2 : remote Header) : Boolean := false
begin
  case h1 begin
    match header(?,?rec1,?f1,?rt1) begin
      case h2 begin
        match header(?,?rec2,?f2,?rt2) begin
          result :=
              -- "Two formal parameter lists match if
              --  1. They have the same number of parameters, and
              length(f1) = length(f2) and
              --  2. They have either the same function result type or none
              (same_type(rt1,rt2) or no_type_p(rt1) and no_type_p(rt2)) and
              --  and
              --  3. parameters at corresponding positions have equal types
              (formal_type_equal(nth(i,f1),nth(i,f2))
                   for i in 0..(length(f1)-1)) and
```

```
              --  4. parameters at corresponding positions are both
              --     either value or variable parameters"
              (formal_match(nth(i,f1),nth(i,f2))
                   for i in 0..(length(f1)-1))
              -- I used to check eceivers too to ensure
              -- forward declarations matched, but this was
              -- 1> unnecessary because forwards only look in the same scope
              -- 2> wrong because then overriding was impossible
              -- REMOVED: and receiver_match(rec1,rec2)
              ;
        end;
      end;
    end;
  end;
end;

function no_type_p(ty : remote Type) : Boolean := false begin
  case ty begin
    match no_type() begin
      result := true;
    end;
  end;
end;

var function formal_type_equal(f1,f2 : remote Declaration) : Boolean := false
begin
  case f1 begin
    match formal(?,?t1) begin
      case f2 begin
        match formal(?,?t2) begin
          result := equal_types(t1,t2);
        end;
      end;
    end;
  end;
end;

function formal_match(f1,f2 : remote Declaration) : Boolean := false
begin
  case f1 begin
    match value_formal(...) begin
      case f2 begin
        match value_formal(...) begin
          result := true;
        end;
      end;
    end;
    match var_formal(...) begin
      case f2 begin
        match var_formal(...) begin
```

```
          result := true;
        end;
      end;
    end;
  end;
end;

var function receiver_match(r1,r2 : remote Receiver) : Boolean := false
begin
  case r1 begin
    match receiver(?f1) begin
      case r2 begin
        match receiver(?f2) begin
          result := formal_type_equal(f1,f2) and formal_match(f1,f2);
        end;
      end;
    end;
    match no_receiver() begin
      case r2 begin
        match no_receiver() begin
          result := true;
        end;
      end;
    end;
  end;
end;
end;
```

B.6 Translation to GCC Trees

The *gcc* compiler can be used as the optimizing back end of a compiler, for example Kenner's Ada compiler [61]. Most of the interface in actually procedural rather than in the form of trees. In order to encapsulate the GCC back end, a full tree language is used. Trees are then taken apart by an interface program that directs the back end.

B.6.1 A Tree Language for the GCC Back End

```
-- Front end structure for GCC described in APS
-- This description follows the gist but not the details of the
-- GCC Tree structure.  In particular, everything is described in structure,
-- whereas in GCC, procedural interfaces are often used.
-- This description was written with the aid of Richard Kenner's
-- 1995 POPL tutorial on the GCC compiler.
module GCC_TREE[](passed_pointer_size : SmallInteger) begin

  pointer_size = passed_pointer_size;

  --- Phyla
```

```
phylum CompilationUnit;

phylum Block;

phylum Declaration;
phylum Statement;
phylum TypePhylum;
type Type := remote TypePhylum; -- tree structure not req'd for types.
phylum Expression;

phylum Use;
type Identifier = Symbol;

phylum Declarations := SEQUENCE[Declaration];
phylum Statements := SEQUENCE[Statement];
type Types := LIST[Type]; -- because not tree structured
phylum Expressions := SEQUENCE[Expression];

phylum Fields := SEQUENCE[Declaration];

constructor compilation_unit(d : Declarations) : CompilationUnit;


--- Declarations

constructor no_decl() : Declaration;

constructor label_decl(name : Identifier) : Declaration;
constructor parm_decl(name : Identifier; ty : Type) : Declaration;
constructor const_decl(name : Identifier; ty : Type;
                       initial : Expression) : Declaration;
constructor var_decl(name : Identifier; ty : Type;
                     initial : Expression) : Declaration;
constructor field_decl(name : Identifier;
                       ty : Type;
                       -- offset is offset in bytes from start of record
                       -- we do not currently handle bit fields
                       offset : Expression) : Declaration;
constructor type_decl(name : Identifier; ty : Type) : Declaration;
-- put a name on a result_decl so it can be matched by declaration(...)
constructor named_result_decl(name : Identifier;
                              ty : Type) : Declaration;
procedure result_decl(ty : Type) : Declaration :=
    named_result_decl(result_name,ty);
pattern result_decl(ty : Type) : Declaration :=
    named_result_decl(?,?ty);
constructor function_decl(name : Identifier;
                          ty : Type;
                          arguments : Declarations;
                          result : Declaration;
```

```
                          body : Block) : Declaration;

pattern declaration(name : Identifier) : Declaration
    := label_decl(?name),parm_decl(name:=?name),const_decl(name:=?name),
       var_decl(name:=?name),field_decl(name:=?name),type_decl(name:=?name),
       named_result_decl(name:=?name),function_decl(name:=?name);

result_name : Identifier := make_symbol("result");

constructor block(decls : Declarations;
                  stmts : Statements) : Block;
constructor no_block() : Block;

-- every declaration gets a mangled name:
input attribute (d:Declaration).assembler_name : Identifier := decl_name(d);
function decl_name(d : remote Declaration) : Identifier begin
  case d begin
    match declaration(?name) begin
      result := name;
    end;
  else
    result := null_symbol;
  end;
end;
-- These should be automatic:
input attribute Declaration.source_file : String := "";
input attribute Declaration.source_line : Integer := 0;

-- flags on declarations:
input attribute Declaration.is_public : Boolean := false;
input attribute Declaration.is_common : Boolean := false;
input attribute Declaration.is_external : Boolean := false;
input attribute Declaration.is_inline : Boolean := false;
-- is_static for var_decls
input attribute Declaration.is_static : Boolean := false;
input attribute Declaration.address_taken : Boolean := false;


--- Use

-- Uses permit cyclic references within the tree:
constructor a_use() : Use;
input attribute Use.use_decl : remote Declaration := nil;
procedure use_remote(d : remote Declaration) : Use begin
  result := a_use();
  result.use_decl := d;
end;


--- Statement
```

```
-- In GCC, these are actually handled through procedural interfaces

constructor no_stmt() : Statement;
constructor seq(stmts : Statements) : Statement;
constructor do(expr : Expression) : Statement;
constructor label(decl : Declaration) : Statement;
constructor goto(label_use : Use) : Statement;
constructor cond(expr : Expression;
                 then_part : Block;
                 else_part : Block) : Statement;
constructor loop(prologue : Block;
                 body : Block) : Statement;
constructor continue() : Statement;
constructor exit() : Statement;
constructor return(value : Expression) : Statement;
-- For C switch and other language case statements.
-- The "exitable" field is true if "exit" can exit from a switch.
-- It is true for C switches because "break" breaks a case not the
-- enclosing loop.
constructor switch(expr : Expression;
                   exitable : Boolean;
                   body : Block) : Statement;
-- in order to handle C switches, case labels are Statements:
constructor single_case(value : Expression) : Statement;
constructor range_case(min,max : Expression) : Statement;
constructor default_case() : Statement;
constructor block_stmt(b : Block) : Statement;


--- Types

constructor type_use(u : Use) : TypePhylum;
constructor integer(unsigned : Boolean;
                    precision : SmallInteger) : TypePhylum;
constructor enumeral(precision : SmallInteger;
                     elements : Declarations) : TypePhylum;
constructor boolean() : TypePhylum;
constructor char() : TypePhylum;
constructor real(precision : SmallInteger) : TypePhylum;
constructor range(min,max : Expression;
                  base : Type) : TypePhylum;

constructor array(domain : Type; range : Type) : TypePhylum;
constructor record(fields : Fields) : TypePhylum;
constructor union(fields : Fields) : TypePhylum;
-- In a qual_union, the fields are marked with an expression
-- that evaluates to true if the
-- field in the union is present.
constructor qual_union(fields : Fields) : TypePhylum;
constructor complex(base : Type) : TypePhylum;
```

```
constructor pointer(base : Type) : TypePhylum;
constructor reference(base : Type) : TypePhylum;
constructor void() : TypePhylum;
constructor function_type(args : Types; return : Type) : TypePhylum;

-- can be changed: default value is a special boolean type:
input boolean_type : Type := boolean();
-- fixed types that may be useful:
char_type : Type := char();
float_type : Type := real(32);
double_type : Type := real(64);
void_type : Type := void();
pointer_type : Type := pointer(void_type);
-- an offset is the same size as a pointer, but an integer type:
offset_type : Type := integer(true,pointer_size);
string_type : Type := pointer(char_type);

-- cyclic types will cause this to crash:
var function base_type(ty : Type) : Type begin
  case ty begin
    match type_use(?u) begin
      case u.use_decl begin
        match type_decl(ty:=?ty2) begin
          result := base_type(ty2);
        end;
      end;
    end;
  end;
  result := ty;
end;


--- Expressions
-- every expression has an expr_type attribute.
-- This type is filled in by default for nodes where the type can be
-- computed easily.  Mandatory attribution is mentioned in comments.

input attribute Expression.expr_type : Type := nil;


constructor no_expr() : Expression;
match ?e=no_expr() begin e.expr_type := void_type; end;


-- constants (expr_type for integer_cst is required)
constructor integer_cst(value : Integer) : Expression;
procedure make_integer_cst(value : Integer; ty : Type) : Expression
begin
  result := integer_cst(value);
  result.expr_type := ty;
```

```
end;
constructor single_real_cst(value : IEEEsingle) : Expression;
constructor double_real_cst(value : IEEEdouble) : Expression;
constructor string_cst(value : String) : Expression;
constructor complex_cst(realpart,imagpart : Expression) : Expression;

match ?e=single_real_cst(?) begin e.expr_type := float_type; end;
match ?e=double_real_cst(?) begin e.expr_type := double_type; end;
match ?e=string_cst(?) begin e.expr_type := string_type; end;
match ?e=complex_cst(?r,?i) begin e.expr_type := complex(r.expr_type); end;


-- uses of declarations
constructor expr_use(u : Use) : Expression;

pattern typed_decl(ty : Type) : Declaration :=
    parm_decl(ty:=?ty), const_decl(ty:=?ty), function_decl(ty:=?ty),
    var_decl(ty:=?ty), field_decl(ty:=?ty);
attribute Declaration.decl_type : Type := void_type;
match ?d=typed_decl(?ty) begin
  d.decl_type := ty;
end;

match ?e=expr_use(?u) begin
  e.expr_type := u.use_decl.decl_type;
end;


-- expression of type void
constructor unit() : Expression;
match ?e=unit() begin e.expr_type := void_type; end;


-- storage references
constructor component_ref(object : Expression; field_use : Use) : Expression;
match ?e=component_ref(?,?fu) begin
  e.expr_type := fu.use_decl.decl_type;
end;

constructor indirect_ref(pointer : Expression) : Expression;
match ?e=indirect_ref(?p) begin
  case base_type(p.expr_type) begin
    match pointer(?ty) begin
      e.expr_type := ty;
    end;
  end;
end;

constructor array_ref(array : Expression;
                      indices : Expressions) : Expression;
```

```
match ?e=array_ref(?a,?) begin
  case base_type(a.expr_type) begin
    match array(?,?ty) begin
      e.expr_type := ty;
    end;
  end;
end;

-- constructing arrays and records:
-- (expr_type required)
constructor (constructor)(objects : Expressions) : Expression;


-- C's comma operator:
constructor compound(e1,e2 : Expression) : Expression;

match ?e=compound(?,?e2) begin
  e.expr_type := e2.expr_type;
end;


-- two types of assignment: regular C assignment
-- and initialization of a variable.  (I'm not sure what the second
-- is used for.)
constructor assign(e1,e2 : Expression) : Expression;
modify = assign; -- original name
constructor initialize(u : Use; e : Expression) : Expression;

match ?e=assign(?e1,?) begin
  e.expr_type := e1.expr_type;
end;

match ?e=initialize(?,?e1) begin
  e.expr_type := e1.expr_type;
end;


-- C's  ... ? ... : ... expression
constructor conditional(c : Expression; e1,e2 : Expression) : Expression;

match ?e=conditional(?,?e1,?) begin
  e.expr_type := e1.expr_type;
end;


-- function call
constructor call(func : Expression; args : Expressions) : Expression;

match ?e=call(?func,...) begin
  case base_type(func.expr_type) begin
```

```
    match function_type(?,?rt) begin
      e.expr_type := rt;
    end;
  end;
end;


-- arithmetic operations (operands and result have same type)
constructor plus(e1,e2 : Expression) : Expression;
constructor minus(e1,e2 : Expression) : Expression;
constructor mult(e1,e2 : Expression) : Expression;

-- 4 times of division for integers:
-- (again operands and result have same type)
-- trunc rounds to zero
-- floor to minus infinity
-- ceil to plus infinity
-- round to nearest integer
constructor trunc_div(e1,e2 : Expression) : Expression;
constructor ceil_div(e1,e2 : Expression) : Expression;
constructor floor_div(e1,e2 : Expression) : Expression;
constructor round_div(e1,e2 : Expression) : Expression;
-- the four corresponding remainder operations
constructor trunc_mod(e1,e2 : Expression) : Expression;
constructor ceil_mod(e1,e2 : Expression) : Expression;
constructor floor_mod(e1,e2 : Expression) : Expression;
constructor round_mod(e1,e2 : Expression) : Expression;
-- division for reals or exact division of integers
constructor div(e1,e2 : Expression) : Expression;
rdiv = div;
div_exact = div;

-- conversions to floating point and back (expr_type must
-- always be specified).
-- truncation of reals to integers: (all four methods as for XXX_div)
constructor fix_trunc(e : Expression) : Expression;
constructor fix_ceil(e : Expression) : Expression; -- unimplemented in GCC!
constructor fix_floor(e : Expression) : Expression; -- unimplemented in GCC!
constructor fix_round(e : Expression) : Expression; -- unimplemented in GCC!
-- conversion of integers to floats:
constructor float(e : Expression) : Expression;

-- unary negation (result has same type as operand)
constructor negate(e : Expression) : Expression;

-- builtins
constructor min(e1,e2 : Expression) : Expression;
constructor max(e1,e2 : Expression) : Expression;
constructor abs(e : Expression) : Expression;
constructor ffs(e : Expression) : Expression; -- "find first set bit"
```

```
-- shifting  (rsult has same type as first operand)
-- on unsigned integers, it is logical, on signed, arithmetic
constructor lshift(e : Expression; amt : Expression) : Expression;
constructor rshift(e : Expression; amt : Expression) : Expression;
constructor lrotate(e : Expression; amt : Expression) : Expression;
constructor rrotate(e : Expression; amt : Expression) : Expression;

-- bitwise operations (operands and result have same type)
constructor bit_or(e1,e2 : Expression) : Expression;
constructor bit_xor(e1,e2 : Expression) : Expression;
constructor bit_and(e1,e2 : Expression) : Expression;
constructor bit_andtc(e1,e2 : Expression) : Expression; -- unimpl'd in GCC!
constructor bit_not(e : Expression) : Expression;
bit_ior = bit_or; -- standard GCC name

-- boolean combinations in which only nonzeroness is relevant.
-- orif and andif are short-circuiting.
constructor truth_andif(e1,e2 : Expression) : Expression;
constructor truth_orif(e1,e2 : Expression) : Expression;
constructor truth_and(e1,e2 : Expression) : Expression;
constructor truth_or(e1,e2 : Expression) : Expression;
constructor truth_xor(e1,e2 : Expression) : Expression;
constructor truth_not(e : Expression) : Expression;

-- comparisons (all but eq and ne legal only for scalars)
-- type is the language boolean type.
constructor lt(e1,e2 : Expression) : Expression;
constructor le(e1,e2 : Expression) : Expression;
constructor gt(e1,e2 : Expression) : Expression;
constructor ge(e1,e2 : Expression) : Expression;
constructor eq(e1,e2 : Expression) : Expression;
constructor ne(e1,e2 : Expression) : Expression;

pattern binop(e1,e2 : Expression) : Expression :=
    plus(?e1,?e2),minus(?e1,?e2),mult(?e1,?e2),div(?e1,?e2),
    trunc_div(?e1,?e2),ceil_div(?e1,?e2),
    floor_div(?e1,?e2),round_div(?e1,?e2),
    trunc_mod(?e1,?e2),ceil_mod(?e1,?e2),
    floor_mod(?e1,?e2),round_mod(?e1,?e2),
    min(?e1,?e2), max(?e1,?e2),
    lshift(?e1,?e2),rshift(?e1,?e2),lrotate(?e1,?e2),rrotate(?e1,?e2),
    bit_or(?e1,?e2),bit_xor(?e1,?e2),bit_and(?e1,?e2),bit_andtc(?e1,?e2),
    truth_andif(?e1,?e2),truth_orif(?e1,?e2),truth_and(?e1,?e2),
    truth_or(?e1,?e2),truth_xor(?e1,?e2);
pattern comparison(e1,e2 : Expression) : Expression :=
    lt(?e1,?e2),le(?e1,?e2),gt(?e1,?e2),ge(?e1,?e2),eq(?e1,?e2),ne(?e1,?e2);
pattern unop(e : Expression) : Expression :=
    negate(?e),abs(?e),ffs(?e),bit_not(?e),truth_not(?e);
```

```
match ?e=binop(?e1,?) begin
  e.expr_type := e1.expr_type;
end;

match ?e=comparison(...) begin
  e.expr_type := boolean_type;
end;

match ?e=unop(?e1) begin
  e.expr_type := e1.expr_type;
end;


-- represents any (explicit or implicit) type conversion.
-- The expr_type must be given.
constructor convert(e : Expression) : Expression;
procedure make_convert(e : Expression; ty : Type) : Expression
begin
  result := convert(e);
  result.expr_type := ty;
end;

-- use when no code should be generated:
constructor nop(e : Expression) : Expression;
-- types same but now cannot be used as an lvalue
constructor non_lvalue(e : Expression) : Expression;
-- non_lvalue for pointer or reference types:
-- (This constructor is redundant)
-- constructor reference(e : Expression) : Expression;


-- C's &  operator:
constructor address(e : Expression) : Expression;

match ?e=address(?e1) begin
  e.expr_type := pointer(e1.expr_type);
end;


-- operators on complex values:
constructor make_complex(real,imag : Expression) : Expression;
constructor conj(e : Expression) : Expression;
constructor realpart(e : Expression) : Expression;
constructor imagpart(e : Expression) : Expression;

match ?e=make_complex(?r,?) begin
  e.expr_type := complex(e.expr_type);
end;
match ?e=conj(?e1) begin
  e.expr_type := e1.expr_type;
```

```
  end;
  pattern real_or_imag_part(e : Expression) : Expression :=
      realpart(?e), imagpart(?e);
  match ?e=real_or_imag_part(?e1) begin
    case base_type(e1.expr_type) begin
      match complex(?ty) begin
        e.expr_type := ty;
      end;
    end;
  end;


  -- C's {post-,pre-}{inc,dec}rement operators: ++ and --
  constructor predecrement(e : Expression; amt : Expression) : Expression;
  constructor preincrement(e : Expression; amt : Expression) : Expression;
  constructor postdecrement(e : Expression; amt : Expression) : Expression;
  constructor postincrement(e : Expression; amt : Expression) : Expression;

  pattern some_crement(e : Expression; amt : Expression) : Expression :=
      predecrement(?e,?amt),preincrement(?e,?amt),
      postdecrement(?e,?amt),postincrement(?e,?amt);

  match ?e=some_crement(?e1,...) begin
    e.expr_type := e1.expr_type;
  end;

  -- remember this expression (in a temporary), it may be get use elsewhere:
  constructor save_expr(e : Expression) : Expression;

  match ?e1=save_expr(?e2) begin
    e1.expr_type := e2.expr_type;
  end;

  -- reuse an expression
  -- (If the object is a save_expr, it will be only evaluated once)
  constructor reuse_expr(e : remote Expression) : Expression;

  match ?e=reuse_expr(?e1) begin
    e.expr_type := e1.expr_type;
  end;
end;
```

B.6.2 Some Runtime-System Additions

```
module OBERON2_GENERATE_RUNTIME[GccTree :: input GCC_TREE[]] extends GccTree
begin
  runtime_compilation_unit : CompilationUnit
      := compilation_unit
      (Declarations${type_guard_failed_decl,
                     array_index_error_decl,
```

```
                   strcmp_decl,
                   allocate_decl, -- for NEW
                   strcpy_decl, -- for COPY
                   assert_decl});

procedure make_external_function(name : String;
                                 arg_types : Types;
                                 return_type : Type) : Declaration
begin
  result := function_decl(make_symbol(name),
                          function_type(arg_types,return_type),
                          Declarations${parm_decl(gensym(),ty)
                                            for ty in arg_types},
                          result_decl(return_type),
                          no_block());
  result.is_public := true;
end;

procedure make_external_var(name : String; ty : Type) : Declaration
begin
  result := var_decl(make_symbol(name),ty,no_expr());
  result.is_public := true;
end;

--- Builtins:

type_guard_failed_decl : Declaration :=
    make_external_function("type_guard_failed",
                           Types${pointer_type,string_type},
                           void_type);

array_index_error_decl : Declaration :=
    make_external_function("array_index_error",
                           Types${offset_type,offset_type},
                           pointer_type);

strcmp_decl : Declaration :=
    make_external_function("System_StringCompare",
                           Types${string_type,string_type},
                           integer(false,pointer_size));

allocate_decl : Declaration := -- used for NEW
    make_external_function("malloc",Types${offset_type},pointer_type);

strcpy_decl : Declaration := -- for COPY
    make_external_function("System_StringCopy",
                           Types${string_type,string_type},
                           string_type);

-- predefined Oberon2 procedures:
```

```
  assert_decl : Declaration :=
      make_external_function("ASSERT",Types${boolean(),offset_type},void_type);
end;
```

B.6.3 Translating Oberon2

```
-- Convert abstract Oberon2 into the GCC tree language defined in
-- gcc-tree.aps in order to interface to rest of GCC compiler.
module OBERON2_TRANSLATE[T :: var OBERON2_TREE[],
                              var OBERON2_RESOLVE[T],
                              var OBERON2_MACHINE_SIZES[],
                              var OBERON2_COMPILE_COMPUTE[T],
                              var OBERON2_LAYOUT[T]]
    extends T
begin
  type BareGccTree := GCC_TREE[](address_size);
  type GccTree := OBERON2_GENERATE_RUNTIME[BareGccTree];

  attribute Declaration.gcc_decl : GccTree$Declaration;
  attribute Declaration.gcc_decls : GccTree$Declarations;
  [T :: {Statement,Case,CaseLabel}, var PHYLUM[]] begin
    attribute T.gcc_stmt : GccTree$Statement;
    pragma source_transfer(gcc_stmt);
  end;
  attribute Type.gcc_type : GccTree$Type;
  -- most types are the same when used in subexpressions,
  -- but the rules on open arrays makes arrays special.
  -- (and anyway we don't want to be copying huge arrays around in
  -- subexpressions, besides, then a[3] := 5 wouldn't work then):
  attribute (t : Type).gcc_expr_type : GccTree$Type := t.gcc_type;
  attribute Expression.gcc_expr : GccTree$Expression;
  attribute Element.gcc_element : GccTree$Expression;

  attribute Program.gcc_program : GccTree$CompilationUnit;

  pragma source_transfer(gcc_decl,gcc_type,gcc_expr_type,gcc_expr,
                         gcc_element,gcc_program);

  private;

  phylum GccDeclarations :: input var SEQUENCE[GccTree$Declaration] :=
      GccTree$Declarations;
  phylum GccStatements :: input var SEQUENCE[GccTree$Statement] :=
      GccTree$Statements;
  type GccTypes :: LIST[GccTree$Type] :=
      GccTree$Types;
  phylum GccExpressions :: input var SEQUENCE[GccTree$Expression] :=
      GccTree$Expressions;
  phylum GccFields :: input var SEQUENCE[GccTree$Declaration] :=
```

```
    GccTree$Fields;

match ?p=program(?modules) begin
  p.gcc_program :=
      GccTree$compilation_unit
      (GccDeclarations${m.gcc_decls...
                           for m in modules});
end;

ignore_symbol : Symbol := make_symbol("ignore");
function gcc_id(ident : remote IdentDef) : Symbol begin
  case ident begin
    match identifier(?name,...) begin
      result := name;
    end;
  else
    result := ignore_symbol;
  end;
end;

-- array expressions carry along their dimensions separately:
type GccRemoteExpressionList := LIST[remote GccTree$Expression];
attribute Expression.gcc_dimensions : GccRemoteExpressionList := {};

-- a paradoxical combination of semantic and syntactic type:
type GccExpressionList := LIST[GccTree$Expression];

-- Similarly: we have to provide access to the object in a method call:
attribute Expression.method_call_object : remote GccTree$Expression;

-- declarations get a mangled name that includes the MODULE.
-- The final assembler name is something like:
--   MODULE_name
attribute Declaration.decl_prefix : String := "";
-- bound procedures get the mangled name: MODULE_type_name
attribute Type.type_prefix : String;

match module_decl(identifier(?sym,...),block(?decls,?)) begin
  prefix : String := symbol_name(sym) ++ "_";
  for decl in decls begin
    decl.decl_prefix := prefix;
  end;
end;

--- Declarations

procedure set_decl_info(decl : GccTree$Declaration;
                        d : remote Declaration) : GccTree$Declaration
begin
  result := decl;
```

```
  -- set the assembler name:
  case d begin
    match bound_proc_decl(identifier(?sym,...),formal(?,?ty)) begin
      method_prefix : String := "";
      case ty.base_type begin
        match pointer_type(?ty) begin
          method_prefix := ty.base_type.type_prefix;
        end;
        match ?ty=record_type(...) begin
          method_prefix := ty.type_prefix;
        end;
      end;
      decl.GccTree$assembler_name :=
          make_symbol(method_prefix ++ symbol_name(sym));
    end;
    match declaration(identifier(?sym,...),...) begin
      if d.decl_prefix = "" then
        decl.GccTree$assembler_name := sym;
      else
        decl.GccTree$assembler_name :=
            make_symbol(d.decl_prefix ++ symbol_name(sym));
      endif;
    end;
  end;
  -- is_public:
  case d begin
    match declaration(identifier(?,not_exported())) begin
      decl.GccTree$is_public := false;
    end;
  else
    decl.GccTree$is_public := true;
  end;
  -- is_external
  case d begin
    match forward(...) begin
      decl.GccTree$is_external := true;
    end;
  end;
end;

match ?d=module_decl(?id,block(?decls,?stmts)) begin
  d.gcc_decl :=
      set_decl_info
      (GccTree$function_decl
           (id.gcc_id,
            GccTree$function_type(GccTypes${},GccTree$void_type),
            GccDeclarations${},
            GccTree$result_decl(GccTree$void_type),
            GccTree$block(GccDeclarations${},
                          GccStatements${stmt.gcc_stmt for stmt in stmts})),
```

```
       d);
  d.gcc_decls := GccDeclarations$
      {decl.gcc_decls... for decl in decls, d.gcc_decl};
end;

match ?d=const_decl(?id,?value) begin
  d.gcc_decl :=
      set_decl_info(GccTree$const_decl(id.gcc_id,
                                       value.expr_type.gcc_expr_type,
                                       value.gcc_expr),
                    d);
end;

match ?d=type_decl(?id,?ty) begin
  d.gcc_decl := set_decl_info(GccTree$type_decl(id.gcc_id,ty.gcc_type),d);
end;

match ?d=var_decl(?id,?ty) begin
  d.gcc_decl := set_decl_info(GccTree$var_decl(id.gcc_id,
                                               ty.gcc_type,
                                               GccTree$no_expr()),
                              d);
end;

match ?d=forward(?h=header(name:=?id)) begin
  d.gcc_decl :=
      set_decl_info(GccTree$function_decl(id.gcc_id,
                                          h.gcc_function_type,
                                          h.gcc_arguments,
                                          h.gcc_result_decl,
                                          GccTree$no_block()),
                    d);
end;

match ?d=proc_decl(?h=header(name:=?id),block(?decls,?stmts))
begin
  d.gcc_decl :=
      set_decl_info(GccTree$function_decl(id.gcc_id,
                                          h.gcc_function_type,
                                          h.gcc_arguments,
                                          h.gcc_result_decl,
                                          gcc_block),
                    d);
  gcc_block : GccTree$Block := ...
      GccTree$block(GccDeclarations${decl.gcc_decls... for decl in decls},
                    GccStatements${stmt.gcc_stmt for stmt in stmts});
end;

match ?d=formal(?id,?) begin
  d.gcc_decl := GccTree$parm_decl(id.gcc_id,d.formal_gcc_type);
```

```
end;
opt_name : GccTree$Identifier := make_symbol("opt");
match ?d=opt_formal(?ty,?) begin
  d.gcc_decl := GccTree$parm_decl(opt_name,ty.gcc_type);
end;


--- Headers

attribute Header.gcc_function_type : GccTree$Type;
attribute Header.gcc_arguments : GccTree$Declarations;
attribute Header.gcc_result_decl : GccTree$Declaration;

match ?h=header(receiver:=?rec,formals:=?formals,result:=?result) begin
  -- NB: function_type "owns" the formal and result types
  -- but this is not a problem because the corresponding parm_decls
  -- and result_decl do not.
  -- NB: receiver becomes first argument
  h.gcc_function_type :=
      GccTree$function_type(GccTypes${rec_gcc_type if bound,
                                      f.formal_gcc_type for f in formals},
                            result.gcc_type);
  h.gcc_arguments := GccDeclarations${rec_gcc_decls... if bound,
                                      f.gcc_decls... for f in formals};
  h.gcc_result_decl := GccTree$result_decl(result.gcc_type);

  bound : Boolean;
  rec_gcc_type : GccTree$Type;
  rec_gcc_decls : GccDeclarations;
  case rec begin
    match receiver(?f) begin
      bound := true;
      rec_gcc_type := f.formal_gcc_type;
      rec_gcc_decls := f.gcc_decls;
    end;
  else
    bound := false;
  end;
end;

-- save type around for the header and for the formal declaration:
attribute Declaration.formal_gcc_type : GccTree$Type;
-- formals of OPEN ARRAY type are passed as a vector
-- the first element is a dope vector (copied for value parameters,
-- but the copy does not show up here).
-- (I assume that GCC that handle arrays passed by value,
--  perhaps this assumption is unwarranted.  Otherwise I have to use
--  records and fiddle around with fields.  A pair type would
--  come in handy).
match ?d=formal(?,?ty=open_array_type(...)) begin
```

```
  d.formal_gcc_type := ty.gcc_type;
end;
-- other array types are passed as a pointer to first element:
-- (again value parameters will be copied first, but not here)
match ?d=formal(?,?ty=fixed_array_type(...)) begin
  d.formal_gcc_type := ty.gcc_type;
end;
-- otherwise var formals are pointer types
match ?d=var_formal(?,?ty) begin
  d.formal_gcc_type := GccTree$pointer(ty.gcc_type);
end;
-- other formals passed by value,
match ?f=value_formal(?,?ty) begin
  f.formal_gcc_type := ty.gcc_type;
end;

-- to keep translation happy:
match ?f=opt_formal(?ty,...) begin
  f.formal_gcc_type := ty.gcc_type;
  f.gcc_decl := GccTree$parm_decl(ignore_symbol,f.formal_gcc_type);
end;
match ?f=rest_formal(?ty) begin
  f.formal_gcc_type := ty.gcc_type;
  f.gcc_decl := GccTree$parm_decl(ignore_symbol,f.formal_gcc_type);
end;

-- a replacement for GccTree$use_remote that doesn't depend
-- on d.gcc_decl.  Note that the second assignment
-- is not required before "result" has a valid value.
var procedure make_use(d : remote Declaration) : GccTree$Use
begin
  result := GccTree$a_use();
  result.GccTree$use_decl := d.gcc_decl;
end;


--- Type Descriptors

attribute Type.decl_for_type : remote Declaration := nil;

attribute Declaration.type_desc_decl : GccTree$Declaration := nil;

-- like make_use, but for the type descriptor:
-- type_decl's used turn into uses of the type_desc_decl
var procedure make_method_ref(d : remote Declaration) : GccTree$Expression
begin
  result := GccTree$address(GccTree$expr_use(u));
  u : GccTree$Use := GccTree$a_use();
  case d begin
    match type_decl(...) begin
```

```
      u.GccTree$use_decl := d.type_desc_decl;
    end;
  else
    u.GccTree$use_decl := d.gcc_decl;
  end;
end;

var procedure make_method_field(d : remote Declaration) : GccTree$Declaration
begin
  index : GccTree$Expression :=
      GccTree$make_integer_cst(d.method_index*address_size,
                               GccTree$offset_type);
  case d begin
    match type_decl(identifier(?name,...),?) begin
      -- break the cycle:
      u : GccTree$Use := GccTree$a_use();
      u.GccTree$use_decl := d.type_spec_type_decl;
      result := GccTree$field_decl(make_symbol(name||"_ref"),
                                   GccTree$pointer(GccTree$type_use(u)),
                                   index);
    end;
    match proc_decl(?h=header(identifier(?name,...),...),...) begin
      result := GccTree$field_decl(make_symbol(name||"_ref"),
                                   GccTree$pointer(h.gcc_function_type),
                                   index);
    end;
  end;
end;

-- record types are handled specially, because the type descriptor
-- must be generated.
attribute Type.spec_type : GccTree$Type := nil;
attribute Declaration.type_spec_type_decl :remote GccTree$Declaration := nil;
match ?d=type_decl(identifier(?name,...),?t=record_type(...)) begin
  t.decl_for_type := d;
  d.type_desc_decl := spec_decl;

  spec_decl : GccTree$Declaration :=
      GccTree$const_decl(gensym(),type_spec_type,init);
  spec_decl.GccTree$assembler_name :=
      make_symbol(d.decl_prefix ++ symbol_name(name) ++ "_typespec");
  spec_decl.GccTree$is_public := true;

  type_spec_type : GccTree$Type := GccTree$record
      (GccFields$
           {GccTree$field_decl
                (make_symbol("typespec_size"),
                 GccTree$offset_type,
                 GccTree$make_integer_cst(0,GccTree$offset_type)),
            make_method_field(m) for m in t.methods});
```

```
    spec_type_decl : GccTree$Declaration :=
        GccTree$type_decl(gensym(),type_spec_type);
    d.type_spec_type_decl := spec_type_decl;
    t.spec_type := type_spec_type;

    init : GccTree$Expression :=
        GccTree$(constructor)
        (GccExpressions${spec_size_expr,
                         make_method_ref(m) for m in t.methods});
    init.GccTree$expr_type := type_spec_type;

    spec_size_expr : GccTree$Expression :=
        GccTree$make_integer_cst(t.desc_size,GccTree$offset_type);

    t.type_prefix := d.decl_prefix ++ symbol_name(name) ++ "_";
    d.gcc_decls := GccDeclarations${d.gcc_decl,spec_type_decl,spec_decl};

end;


 --- Types

 match ?t=no_type() begin
   t.gcc_type := GccTree$void_type; -- this shouldn't be used....
 end;

 match ?t=named_type(?u) begin
   t.gcc_type := GccTree$type_use(make_use(u.use_decl));
 end;

 -- multi-dimensional arrays (even ones with open parts)
 -- are laid out as a single block of memory.
 -- (see note on expressions below...)
 match ?t=fixed_array_type(?length,?elemty) begin
   case length.constant_value begin
     match Constant$some_integer_constant(?v) begin
       t.gcc_type := GccTree$array
           (GccTree$range(GccTree$make_integer_cst(0,GccTree$offset_type),
                          GccTree$make_integer_cst(v-1,GccTree$offset_type),
                          GccTree$offset_type),
            elemty.gcc_type);
     end;
   end;
 end;
 -- open array's are records, the first first the the dope vector,
 -- the remaining fields are the lengths of the dimensions.
 attribute Type.open_ranges : Integer := 0;
 match ?t=open_array_type(?elemty) begin
   -- we don't need to use the base type because
```

```
  -- open array types need not be named:
  t.open_ranges := 1 + elemty.open_ranges;
  t.gcc_type := GccTree$record
      (GccFields$
           {GccTree$field_decl
                (make_symbol("dopevector"),
                 t.gcc_expr_type, -- GccTree$pointer_type,
                 GccTree$make_integer_cst(0,GccTree$offset_type)),
            GccTree$field_decl
                (make_symbol("dim" || i),
                 GccTree$offset_type,
                 GccTree$make_integer_cst(i*address_size,
                                          GccTree$offset_type))
                for i in 1..t.open_ranges});
end;
-- In expressions,
-- however, the type of an array subexpression is a pointer to
-- the contents (one big vector).
-- The ranges come along separately.
match ?t=array_type(?elemty) begin
  case elemty.base_type begin
    match ?et=array_type(...) begin
      t.gcc_expr_type := et.gcc_expr_type;
    end;
    match ?et begin
      t.gcc_expr_type := GccTree$pointer(et.gcc_type);
    end;
  end;
end;

phylum DimensionExpr;
constructor dimension_expr() : DimensionExpr;
attribute DimensionExpr.useable_dimensions : GccTree$Expression;
attribute DimensionExpr.reuseable_dimensions : GccRemoteExpressionList;

-- create a dimension expr with a useable useable_dimensions attribute
procedure make_dimension_expr(array_expr : remote GccTree$Expression;
                              dimension : Integer;
                              topty : remote Type;
                              ty : remote Type)
    : DimensionExpr
begin
  case ty.base_type begin
    match ?bt=array_type(?elemty) begin
      next : DimensionExpr :=
          make_dimension_expr(array_expr,dimension+1,topty,elemty);
      result := dimension_expr();
      new_expr : GccTree$Expression;
      case bt begin
        match open_array_type(...) begin
```

```
          case topty.base_type.gcc_type begin
            match GccTree$record(?fields) begin
              new_expr :=
                  GccTree$save_expr
                  (GccTree$component_ref
                       (GccTree$reuse_expr(array_expr),
                        GccTree$use_remote(nth(dimension+1,fields))));
            end;
          end;
        end;
        match ?t=fixed_array_type(?length,?elemty) begin
          case length.constant_value begin
            match Constant$some_integer_constant(?v) begin
              new_expr := GccTree$make_integer_cst(v,GccTree$offset_type);
            end;
          end;
        end;
      end; -- case bt
      result.useable_dimensions := GccTree$compound(new_expr,
                                                    next.useable_dimensions);
      result.reuseable_dimensions :=
          {new_expr} ++ next.reuseable_dimensions;
    end; -- match array_type
  else
    result := dimension_expr();
    result.useable_dimensions := GccTree$no_expr();
    result.reuseable_dimensions := {};
  end;
end;

-- represent two values:
-- 1> GCC tree of array pointer expression
-- 2> a list of reusable dimensions

phylum ArrayExpr;
constructor array_expr() : ArrayExpr;
attribute ArrayExpr.gcc_array_expr : GccTree$Expression;
attribute ArrayExpr.gcc_array_dimensions : GccRemoteExpressionList;

-- pass in a pointer to the array (fixed) or record (open)
procedure make_array_expr(ty : remote Type;
                          base : GccTree$Expression) ae : ArrayExpr
begin
  ae := array_expr();
  case ty begin
    match array_type(...) begin
      de : DimensionExpr := make_dimension_expr(base,0,ty,ty);
      ptr_expr : GccTree$Expression;
      case ty begin
        match open_array_type(...) begin
```

```
          case ty.gcc_type begin
            match GccTree$record(?fields) begin
              ptr_expr := GccTree$component_ref
                  (GccTree$reuse_expr(base),
                   GccTree$use_remote(first(fields)));
            end;
          end;
        end;
      else
        ptr_expr := GccTree$make_convert(GccTree$reuse_expr(base),
                                         ty.gcc_expr_type);
      end;
      ae.gcc_array_expr := GccTree$compound
          (base,GccTree$compound(de.useable_dimensions,ptr_expr));
      ae.gcc_array_dimensions := de.reuseable_dimensions;
    end;
  else
    ae.gcc_array_expr := base;
    ae.gcc_array_dimensions := {};
  end;
end;


match ?f=field(...) begin
  f.gcc_decl := make_gcc_field_decl(f);
  -- the decl will go into a sequence of a different type (Fields)
  f.gcc_decls := GccDeclarations${};
end;

match ?d:Declaration begin
  -- default
  d.gcc_decl := GccTree$no_decl();
  d.gcc_decls := GccTree$Declarations${d.gcc_decl};
end;

procedure make_gcc_field_decl(f : remote Declaration) : GccTree$Declaration
begin
  case f begin
    match field(?id,?ty) begin
      result := GccTree$field_decl
          (id.gcc_id,
           ty.gcc_type,
           GccTree$make_integer_cst(f.offset,GccTree$offset_type));
    end;
  end;
end;

type_spec_name : GccTree$Identifier := make_symbol("_type_spec");
attribute Type.type_spec_field : remote GccTree$Declaration := nil;
```

```
match ?t=record_type(?parent,?local_fields) begin
  type_spec_field_decl : GccTree$Declaration :=
      GccTree$field_decl(type_spec_name,
                         GccTree$pointer(t.spec_type),
                         GccTree$make_integer_cst(0,GccTree$offset_type));
  t.type_spec_field := type_spec_field_decl;
  t.gcc_type := GccTree$record
      (GccFields${type_spec_field_decl,
                  make_gcc_field_decl(f)
                      for f in parent.base_type.field_list,
                  f.gcc_decl for f in local_fields});
end;

match ?t=pointer_type(?base) begin
  case base begin
    -- POINTER is nop for open array types:
    match open_array_type(...) begin
      t.gcc_type := base.gcc_type;
    end;
  else
    t.gcc_type := GccTree$pointer(base.gcc_type);
  end;
end;

match ?t=proc_type(?header) begin
  t.gcc_type := header.gcc_function_type;
end;

match ?t=boolean_type() begin
  t.gcc_type := GccTree$boolean();
end;

match ?t=char_type() begin
  t.gcc_type := GccTree$char_type;
end;

match ?t=shortint_type() begin
  t.gcc_type := GccTree$integer(false,shortint_size*byte_bits);
end;
match ?t=integer_type() begin
  t.gcc_type := GccTree$integer(false,integer_size*byte_bits);
end;
match ?t=longint_type() begin
  t.gcc_type := GccTree$integer(false,longint_size*byte_bits);
end;
match ?t=real_type() begin
  t.gcc_type := GccTree$real(real_size*byte_bits);
end;
match ?t=longreal_type() begin
  t.gcc_type := GccTree$real(longreal_size*byte_bits);
```

```
end;

match ?t=set_type() begin
  t.gcc_type := GccTree$integer(true,set_size*byte_bits);
end;

match ?t=nil_type() begin
  t.gcc_type := GccTree$pointer_type;
end;

match ?t=pseudo_type() begin
  t.gcc_type := GccTree$pointer_type;
end;


--- Statements

match ?s=assign_stmt(?lhs,?rhs) begin
  s.gcc_stmt := GccTree$do
      (GccTree$assign
           (lhs.gcc_expr,convert_expr(rhs,rhs.gcc_expr,lhs.expr_type)));
end;

match ?s=call_stmt(?call) begin
  s.gcc_stmt := GccTree$do(call.gcc_expr);
end;

match ?s=if_stmt(?cond,?ths,?els) begin
  s.gcc_stmt := GccTree$cond
      (cond.gcc_expr,
       GccTree$block(GccDeclarations${},
                     GccStatements${th.gcc_stmt for th in ths}),
       GccTree$block(GccDeclarations${},
                     GccStatements${el.gcc_stmt for el in els}));
end;

match ?s=case_stmt(?expr,?cases,?default) begin
  s.gcc_stmt := GccTree$switch
      (expr.gcc_expr,
       false, -- EXIT is only for LOOPS
       GccTree$block(GccDeclarations${},
                     GccStatements${c.gcc_stmt for c in cases,
                                    GccTree$default_case(),
                                    (default...).gcc_stmt}));
end;
match ?s=case_clause(?labels,?body) begin
  s.gcc_stmt := GccTree$seq
      (GccStatements${label.gcc_stmt for label in labels,
                      stmt.gcc_stmt for stmt in body});
end;
```

```
--!! change to use constant value...
match ?s=single_label(?value) begin
  s.gcc_stmt := GccTree$single_case(value.gcc_expr);
end;
match ?s=range_label(?start,?stop) begin
  s.gcc_stmt := GccTree$range_case(start.gcc_expr,stop.gcc_expr);
end;


-- WHILE, REPEAT and FOR do not translate in gcc loops
-- so that EXIT can be reserved for LOOP.
match ?s=while_stmt(?expr,?stmts) begin
  -- generate a simple sequence:
  --    goto LCOND
  -- LBODY:
  --    body
  -- LCOND:
  --    if condition goto LBODY
  body_label : GccTree$Declaration := GccTree$label_decl(gensym());
  cond_label : GccTree$Declaration := GccTree$label_decl(gensym());
  s.gcc_stmt := GccTree$seq
      (GccStatements$
           {GccTree$goto(GccTree$use_remote(cond_label)),
            GccTree$label(body_label),
            s.gcc_stmt for s in stmts,
            GccTree$label(cond_label),
            GccTree$cond
                (expr.gcc_expr,
                 GccTree$block(GccDeclarations${},
                               GccStatements$
                                   {GccTree$goto
                                        (GccTree$use_remote(body_label))}),
                 GccTree$no_block())});
end;
match ?s=repeat_stmt(?stmts,?expr) begin
  -- generate a simpler sequence
  -- LBODY:
  --    body
  --    if !condition goto LBODY
  body_label : GccTree$Declaration := GccTree$label_decl(gensym());
  s.gcc_stmt := GccTree$seq
      (GccStatements$
           {GccTree$label(body_label),
            s.gcc_stmt for s in stmts,
            GccTree$cond
                (GccTree$truth_not(expr.gcc_expr),
                 GccTree$block(GccDeclarations${},
                               GccStatements$
                                   {GccTree$goto
                                        (GccTree$use_remote(body_label))}),
```

```
                 GccTree$no_block())});
end;
match ?s=for_stmt(?v=named_expr(?vu=use_name(?)),
                  ?start,?finish,?step,
                  ?stmts) begin
  step_value : Integer := 1;
  case step.constant_value begin
    match Constant$some_integer_constant(?v) begin
      step_value := v;
    end;
  end;
  -- generate code for
  -- temp := finish; v := start;
  -- IF step >0 THEN
  --   WHILE v <= temp DO statements; v := v+step END
  -- ELSE
  --   WHILE v > temp DO statements; v := v+step END
  -- END
  temp_decl : GccTree$Declaration :=
      GccTree$var_decl(make_symbol("temp"),
                       v.expr_type.gcc_expr_type,
                       finish.gcc_expr);
  step_expr : GccTree$Expression :=
      GccTree$make_integer_cst(step_value,v.expr_type.gcc_expr_type);
  init_v : GccTree$Statement :=
      GccTree$do(GccTree$assign
                   (v.gcc_expr,
                    convert_expr(start,start.gcc_expr,v.expr_type)));
  -- we cannot reuse v.gcc_expr, so we could copy it, or
  -- since we know how it was created, can recreate it:
  inc_v : GccTree$Statement :=
      GccTree$do
      (GccTree$assign
           (GccTree$expr_use(GccTree$use_remote(vu.use_decl.gcc_decl)),
            GccTree$plus
                (GccTree$expr_use(GccTree$use_remote(vu.use_decl.gcc_decl)),
                 step_expr)));
  body_label : GccTree$Declaration := GccTree$label_decl(gensym());
  cond_label : GccTree$Declaration := GccTree$label_decl(gensym());
  condition : GccTree$Expression;
  if step_value > 0 then
    condition :=
        GccTree$le
        (GccTree$expr_use(GccTree$use_remote(vu.use_decl.gcc_decl)),
         GccTree$expr_use(GccTree$use_remote(temp_decl)));
  else
    condition :=
        GccTree$le
        (GccTree$expr_use(GccTree$use_remote(vu.use_decl.gcc_decl)),
         GccTree$expr_use(GccTree$use_remote(temp_decl)));
```

```
  endif;
  s.gcc_stmt := GccTree$block_stmt
      (GccTree$block
           (GccDeclarations${temp_decl},
            GccStatements$
                {init_v,
                 GccTree$goto(GccTree$use_remote(cond_label)),
                 GccTree$label(body_label),
                 s.gcc_stmt for s in stmts,
                 inc_v,
                 GccTree$label(cond_label),
                 GccTree$cond
                     (condition,
                      GccTree$block
                          (GccDeclarations${},
                           GccStatements$
                               {GccTree$goto
                                     (GccTree$use_remote(body_label))}),
                      GccTree$no_block())}));
end;

match ?s=loop_stmt(?stmts) begin
  s.gcc_stmt := GccTree$loop
      (GccTree$no_block(),
       GccTree$block(GccDeclarations${},
                     GccStatements${s.gcc_stmt for s in stmts}));
end;
match ?s=exit_stmt() begin
  s.gcc_stmt := GccTree$exit();
end;

match ?s=with_stmt(?expr,?ty,?body,?rest) begin
  s.gcc_stmt := GccTree$cond
      (gen_type_test(expr.gcc_expr,ty),
       GccTree$block(GccDeclarations${},
                     GccStatements${stmt.gcc_stmt for stmt in body}),
       GccTree$block(GccDeclarations${},
                     GccStatements${stmt.gcc_stmt for stmt in rest}));
end;

match ?s=return_stmt(?expr) begin
  s.gcc_stmt := GccTree$return(expr.gcc_expr);
end;


--- Expressions

shortint_gcc_type : GccTree$Type :=
    GccTree$integer(false,shortint_size*byte_bits);
integer_gcc_type : GccTree$Type :=
```

```
    GccTree$integer(false,integer_size*byte_bits);
longint_gcc_type : GccTree$Type :=
    GccTree$integer(false,longint_size*byte_bits);
set_gcc_type : GccTree$Type :=
    GccTree$integer(false,set_size*byte_bits);

procedure gen_constant_value(x : Constant) const : GccTree$Expression begin
  case x begin
    match Constant$shortint_constant(?v) begin
      const := GccTree$integer_cst(v);
      const.GccTree$expr_type := shortint_gcc_type;
    end;
    match Constant$integer_constant(?v) begin
      const := GccTree$integer_cst(v);
      const.GccTree$expr_type := integer_gcc_type;
    end;
    match Constant$longint_constant(?v) begin
      const := GccTree$integer_cst(v);
      const.GccTree$expr_type := longint_gcc_type;
    end;
    match Constant$real_constant(?v) begin
      const := GccTree$single_real_cst(v);
      const.GccTree$expr_type := GccTree$float_type;
    end;
    match Constant$longreal_constant(?v) begin
      const := GccTree$double_real_cst(v);
      const.GccTree$expr_type := GccTree$double_type;
    end;
    match Constant$set_constant(?rep) begin
      const := GccTree$integer_cst(rep);
      const.GccTree$expr_type := set_gcc_type;
    end;
    match Constant$boolean_constant(?b) begin
      v : Integer := 0;
      if b then v := 1; endif;
      const := GccTree$integer_cst(v);
      -- NB: cannot use boolean_type, because it's an input attribute.
      const.GccTree$expr_type := GccTree$boolean();
    end;
    match Constant$char_constant(?v) begin
      const := GccTree$integer_cst(char_code(v));
      const.GccTree$expr_type := GccTree$char_type;
    end;
    match Constant$string_constant(?s) begin
      const := GccTree$string_cst(s);
    end;
    match Constant$undefined() begin -- i.e. nil
      const := GccTree$integer_cst(0);
      const.GccTree$expr_type := GccTree$pointer_type;
    end;
```

```
  else
    -- control should not get here:
    pragma break();
    -- not a good constant value:
    const := GccTree$integer_cst(8888);
    const.GccTree$expr_type := GccTree$pointer_type;
  end;
end;

-- by default every expression uses its constant value:
match ?e:Expression if e.expr_constant and e.expr_header == nil begin
  const : GccTree$Expression := gen_constant_value(e.constant_value);
  case e.constant_value begin
    match Constant$string_constant(?s) begin
      dim : GccTree$Expression :=
          GccTree$make_integer_cst(length(s)+1,GccTree$offset_type);
      e.gcc_dimensions := {dim};
      e.gcc_expr := GccTree$compound(dim,const);
    end;
  else
    e.gcc_expr := const;
  end;
end;

match ?e=no_expr() begin
  e.gcc_expr := GccTree$no_expr();
end;

-- named expressions can be qualified or Module.var references
-- disguised as fref's:
pattern actual_named_expr(u : Use) : Expression :=
    named_expr(?u),
    fref(named_expr(?u),?u,!false) if module_decl_p(u.use_decl);

function module_decl_p(x : remote Declaration) : Boolean begin
  case x begin
    match module_decl(...) begin result := true; end;
  else
    result := false;
  end;
end;

match ?e=actual_named_expr(?u) begin
  base : GccTree$Expression := GccTree$expr_use(make_use(u.use_decl));

  simple : GccTree$Expression;
  case e.expr_type begin
    match ?ty=array_type(...) begin
      ae : ArrayExpr := make_array_expr(ty,base);
      simple := ae.gcc_array_expr;
```

```
      e.gcc_dimensions := ae.gcc_array_dimensions;
    end;
  else
    -- var formals need implicit dereferencing:
    case u.use_decl begin
      match var_formal(...) begin
        simple := GccTree$indirect_ref(base);
      end;
    else
      simple := base;
    end;
  end;

  -- We can't put guards on something which is immediately assigned,
  -- because the generated code is conditional (unsuitable as an lvalue)
  -- and we don't need to anyway since the value is about to be overwritten
  -- anyway (and it must be a legal value because of typechecking).
  if e.implicitly_guarded and not e.immediately_assigned then
    case e.expr_type begin
      match pointer_type(?ty) begin
        e.gcc_expr := gen_type_guard(GccTree$save_expr(simple),ty.base_type);
      end;
    else
      -- gen_type_guard operates on pointer expressions:
      e.gcc_expr := GccTree$indirect_ref
          (gen_type_guard(GccTree$save_expr(GccTree$address(simple)),
                          e.expr_type));
    end;
  else
    e.gcc_expr := simple;
  endif;
end;

-- upward pattern matching could handle this without attributes:
attribute Expression.immediately_assigned : Boolean := false;
match assign_stmt(?expr,?) begin
  expr.immediately_assigned := true;
end;
match ?e1=actual_type_guard(?e2,?) begin
  e2.immediately_assigned := e1.immediately_assigned;
end;


-- arithmetic may involve implicit coercions:
procedure convert_expr(e : remote Expression;
                       tree : GccTree$Expression;
                       arg_ty : remote Type) : GccTree$Expression
begin
  -- convert to base type first:
  ty : remote Type := arg_ty.base_type;
```

```
  -- overly zealous application doesn't hurt:
  if e.expr_type /= ty then
    -- special cases: characters and strings can be coerced into each other
    if e.expr_type = char and is_string_type(ty) then
      -- if it's a constant character, we turn it into a constant
      -- string, otherwise we create an array expression:
      case e.constant_value begin
        match Constant$char_constant(?c) begin
          result := GccTree$string_cst({c});
        end;
      else
        null : GccTree$Expression :=
            GccTree$make_integer_cst(0,GccTree$char_type);
        constructed_array : GccTree$Expression :=
            GccTree$(constructor)(GccExpressions${tree,null});
        constructed_array.GccTree$expr_type :=
            GccTree$array(GccTree$range
                              (GccTree$make_integer_cst(0,integer.gcc_type),
                               GccTree$make_integer_cst(1,integer.gcc_type),
                               integer.gcc_type),
                          GccTree$char_type);
        result := GccTree$address(constructed_array);
        result.GccTree$expr_type := GccTree$string_type;
      end;
    elsif is_string_type(e.expr_type) and ty = char then
      -- again, special case: the string is constant:
      case e.constant_value begin
        -- don't check here for the string being the right length:
        match Constant$string_constant({?c,...}) begin
          result := GccTree$make_integer_cst(char_code(c),GccTree$char_type);
        end;
      else
        result := GccTree$indirect_ref(tree);
        result.GccTree$expr_type := GccTree$char_type;
      end;
    else
      result := GccTree$convert(tree);
      result.GccTree$expr_type := ty.gcc_expr_type;
    endif;
  else
    result := tree;
  endif;
end;

function is_string_type(ty : remote Type) : Boolean begin
  case ty begin
    match !string begin
      result := true;
    end;
    match array_type(?base) begin
```

```
      case base.base_type begin
        match char_type() begin
          result := true;
        end;
      end;
    end;
  end;
  result := false;
end;

procedure make_component_ref(record : GccTree$Expression;
                             field : remote Declaration) : GccTree$Expression
begin
  result := GccTree$component_ref
      (record, -- GccTree$make_convert(record,field.field_record.gcc_type),
       GccTree$use_remote(field.gcc_decl));
end;

-- we add conversions for unary operations to avoid disastrous errors
match ?e=unop(log_not(),?arg) begin
  e.gcc_expr := GccTree$truth_not(convert_expr(arg,arg.gcc_expr,boolean));
end;
match ?e=unop(plus(),?arg) begin
  e.gcc_expr := arg.gcc_expr;
end;
match ?e=unop(minus(),?arg) begin
  -- extra work for sets:
  if e.expr_type = set then
    e.gcc_expr :=
        GccTree$bit_not(convert_expr(arg,arg.gcc_expr,e.expr_type));
  else
    e.gcc_expr :=
        GccTree$negate(convert_expr(arg,arg.gcc_expr,e.expr_type));
  endif;
end;

match ?e=binop(log_or(),?arg1,?arg2) begin
  e.gcc_expr :=
      GccTree$truth_orif(convert_expr(arg1,arg1.gcc_expr,boolean),
                         convert_expr(arg2,arg2.gcc_expr,boolean));
end;
match ?e=binop(log_and(),?arg1,?arg2) begin
  e.gcc_expr :=
      GccTree$truth_andif(convert_expr(arg1,arg1.gcc_expr,boolean),
                          convert_expr(arg2,arg2.gcc_expr,boolean));
end;

match ?e=binop(plus(),?arg1,?arg2) begin
  -- more extra work for sets (also for minus(), times(), divide())
  if e.expr_type = set then
```

```
    e.gcc_expr :=
        GccTree$bit_or(convert_expr(arg1,arg1.gcc_expr,e.expr_type),
                       convert_expr(arg2,arg2.gcc_expr,e.expr_type));
  else
    e.gcc_expr := GccTree$plus(convert_expr(arg1,arg1.gcc_expr,e.expr_type),
                               convert_expr(arg2,arg2.gcc_expr,e.expr_type));
  endif;
end;
match ?e=binop(minus(),?arg1,?arg2) begin
  if e.expr_type = set then
    e.gcc_expr :=
        GccTree$bit_andtc(convert_expr(arg1,arg1.gcc_expr,e.expr_type),
                          convert_expr(arg2,arg2.gcc_expr,e.expr_type));
  else
    e.gcc_expr :=
        GccTree$minus(convert_expr(arg1,arg1.gcc_expr,e.expr_type),
                      convert_expr(arg2,arg2.gcc_expr,e.expr_type));
  endif;
end;
match ?e=binop(times(),?arg1,?arg2) begin
  if e.expr_type = set then
    e.gcc_expr :=
        GccTree$bit_and(convert_expr(arg1,arg1.gcc_expr,e.expr_type),
                        convert_expr(arg2,arg2.gcc_expr,e.expr_type));
  else
    e.gcc_expr := GccTree$mult(convert_expr(arg1,arg1.gcc_expr,e.expr_type),
                               convert_expr(arg2,arg2.gcc_expr,e.expr_type));
  endif;
end;
match ?e=binop(divide(),?arg1,?arg2) begin
  if e.expr_type = set then
    e.gcc_expr :=
        GccTree$bit_xor(convert_expr(arg1,arg1.gcc_expr,e.expr_type),
                        convert_expr(arg2,arg2.gcc_expr,e.expr_type));
  else
    -- In Oberon2 int1/int2 -> real and
    -- so both operands may need to be converted:
    e.gcc_expr := GccTree$div(convert_expr(arg1,arg1.gcc_expr,e.expr_type),
                              convert_expr(arg2,arg2.gcc_expr,e.expr_type));
  endif;
end;
match ?e=binop(mod(),?arg1,?arg2) begin
  e.gcc_expr :=
      GccTree$floor_mod(convert_expr(arg1,arg1.gcc_expr,e.expr_type),
                        convert_expr(arg2,arg2.gcc_expr,e.expr_type));
end;
match ?e=binop(div(),?arg1,?arg2) begin
  e.gcc_expr :=
      GccTree$floor_div(convert_expr(arg1,arg1.gcc_expr,e.expr_type),
                        convert_expr(arg2,arg2.gcc_expr,e.expr_type));
```

```
end;

-- comparing strings uses the builtin strcmp function:
procedure make_strcmp_call(arg1,arg2 : Expression) : GccTree$Expression :=
    GccTree$call(GccTree$expr_use(GccTree$use_remote(GccTree$strcmp_decl)),
                 GccExpressions${arg1.gcc_expr,arg2.gcc_expr});

-- with comparisons, we coerce to first argument's type
match ?e=binop(equal(),?a1,?a2) begin
  if is_string_type(a1.expr_type) then
    e.gcc_expr := GccTree$eq(make_strcmp_call(a1,a2),
                             GccTree$make_integer_cst(0,longint.gcc_type));
  else
    e.gcc_expr := GccTree$eq(a1.gcc_expr,
                             convert_expr(a2,a2.gcc_expr,a1.expr_type));
  endif;
end;
match ?e=binop(not_equal(),?a1,?a2) begin
  if is_string_type(a1.expr_type) then
    e.gcc_expr := GccTree$ne(make_strcmp_call(a1,a2),
                             GccTree$make_integer_cst(0,longint.gcc_type));
  else
    e.gcc_expr := GccTree$ne(a1.gcc_expr,
                             convert_expr(a2,a2.gcc_expr,a1.expr_type));
  endif;
end;
-- we may have to add error-prevention code to prevent this from
-- blowing up if there are errors in the code:
match ?e=binop(less(),?a1,?a2) begin
  if is_string_type(a1.expr_type) then
    e.gcc_expr := GccTree$lt(make_strcmp_call(a1,a2),
                             GccTree$make_integer_cst(0,longint.gcc_type));
  else
    e.gcc_expr := GccTree$lt(a1.gcc_expr,
                             convert_expr(a2,a2.gcc_expr,a1.expr_type));
  endif;
end;
match ?e=binop(less_equal(),?a1,?a2) begin
  if is_string_type(a1.expr_type) then
    e.gcc_expr := GccTree$le(make_strcmp_call(a1,a2),
                             GccTree$make_integer_cst(0,longint.gcc_type));
  else
    e.gcc_expr := GccTree$le(a1.gcc_expr,
                             convert_expr(a2,a2.gcc_expr,a1.expr_type));
  endif;
end;
match ?e=binop(greater(),?a1,?a2) begin
  if is_string_type(a1.expr_type) then
    e.gcc_expr := GccTree$gt(make_strcmp_call(a1,a2),
                             GccTree$make_integer_cst(0,longint.gcc_type));
```

```
  else
    e.gcc_expr := GccTree$gt(a1.gcc_expr,
                             convert_expr(a2,a2.gcc_expr,a1.expr_type));
  endif;
end;
match ?e=binop(greater_equal(),?a1,?a2) begin
  if is_string_type(a1.expr_type) then
    e.gcc_expr := GccTree$gt(make_strcmp_call(a1,a2),
                             GccTree$make_integer_cst(0,longint.gcc_type));
  else
    e.gcc_expr := GccTree$ge(a1.gcc_expr,
                             convert_expr(a2,a2.gcc_expr,a1.expr_type));
  endif;
end;

match ?e=binop(in_set(),?x,?s) begin
  -- x IN S ==> ((1<<X)&S)!=0
  e.gcc_expr :=
      GccTree$ne
      (GccTree$bit_and(GccTree$lshift(one,
                                      convert_expr(x,x.gcc_expr,shortint)),
                       convert_expr(s,s.gcc_expr,set)),
       zero);
  zero : GccTree$Expression := GccTree$make_integer_cst(0,set.gcc_type);
  one : GccTree$Expression := GccTree$make_integer_cst(1,set.gcc_type);
end;

-- predefined function procedures:
-- (As long as a return type or variable formal isn't polymorphic,
--  we can implement as procedures (possibly inlined)).
-- constant: MAX, MIN, SIZE
-- polymorphic: ABS, LEN, LONG, SHORT
-- monomorphic: ASH, CAP, CHR, ENTIER, ODD, ORD
--
-- proper procedures:
-- polymorphic: DEC, INC, NEW
-- monomorphic: ASSERT, COPY, EXCL, HALT, INCL
--
-- we have to generate special code for the polymorphic ones,
-- but we have to make sure each is the builtin procedure, not
-- one of the same name that we are generating code for.

match ?e=funcall(?func,?args) begin
  h: remote Header := func.expr_header;
  case h begin
    -- ABS
    match header(identifier(!make_symbol("ABS"),...),
                 result:=abs_type()) begin
      case args begin
        match {?arg} begin
```

```
      e.gcc_expr := GccTree$abs(arg.gcc_expr);
    end;
  end;
end;
-- LEN
match header(identifier(!make_symbol("LEN"),...),
             formals:={?fixed,opt_formal(shape:=?opt_type)}) begin
  dim : Integer := 0;
  -- only open array expressions are not compile-time computable:
  case args begin
    match {?,?dim_expr} begin
      case dim_expr.constant_value begin
        match Constant$some_integer_constant(?v) begin
          dim := v;
        end;
      end;
    end;
  end;
  case args begin
    match {?arg,...} begin
      e.gcc_expr := GccTree$compound
          (arg.gcc_expr,
           GccTree$reuse_expr(nth(dim,arg.gcc_dimensions)));
    end;
  end;
end;
-- LONG
match header(identifier(!make_symbol("LONG"),...),
             formals:={value_formal(shape:=long_type())}) begin
  case args begin
    match {?arg} begin
      e.gcc_expr :=
          GccTree$make_convert(arg.gcc_expr,e.expr_type.gcc_expr_type);
    end;
  end;
end;
-- SHORT
match header(identifier(!make_symbol("SHORT"),...),
             formals:={value_formal(shape:=short_type())}) begin
  case args begin
    match {?arg} begin
      e.gcc_expr :=
          GccTree$make_convert(arg.gcc_expr,e.expr_type.gcc_expr_type);
    end;
  end;
end;
-- COPY
-- (not polymorphic, but we convert to use strcpy)
-- BUG: this catches all calls to any procedure named COPY.
match header(identifier(!make_symbol("COPY"),...),...) begin
```

```
  case args begin
    match {?x,?v} begin
      e.gcc_expr := GccTree$make_convert
          (GccTree$call
               (GccTree$expr_use
                    (GccTree$use_remote(GccTree$strcpy_decl)),
                GccTree$Expressions${v.gcc_expr,x.gcc_expr}),
           GccTree$void_type);
    end;
  end;
end;
-- DEC
match header(identifier(!make_symbol("DEC"),...),
             formals:={var_formal(...),opt_formal(...)})
begin
  case args begin
    match {?v} begin
      e.gcc_expr := GccTree$predecrement
          (v.gcc_expr,
           GccTree$make_integer_cst(1,v.expr_type.gcc_expr_type));
    end;
    match {?v,?amt} begin
      e.gcc_expr := GccTree$predecrement
          (v.gcc_expr,
           GccTree$make_convert(amt.gcc_expr,v.expr_type.gcc_expr_type));
    end;
  end;
end;
-- INC
match header(identifier(!make_symbol("INC"),...),
             formals:={var_formal(...),opt_formal(...)})
begin
  case args begin
    match {?v} begin
      e.gcc_expr := GccTree$preincrement
          (v.gcc_expr,
           GccTree$make_integer_cst(1,v.expr_type.gcc_expr_type));
    end;
    match {?v,?amt} begin
      e.gcc_expr := GccTree$preincrement
          (v.gcc_expr,
           GccTree$make_convert(amt.gcc_expr,v.expr_type.gcc_expr_type));
    end;
  end;
end;
-- NEW
match header(identifier(!make_symbol("NEW"),...),
             formals:={?,rest_formal(...)}) begin
  case args begin
    match {?ptr,...} begin
```

```
size : Integer := 0;
case ptr.expr_type begin
  match pointer_type(?ty) begin
    size := ty.type_size;
  end;
end;
size_args : GccExpressionList :=
    {GccTree$save_expr(GccTree$make_convert(arg.gcc_expr,
                                            GccTree$offset_type))
         if arg /= ptr
         for arg in args};
call : GccTree$Expression := GccTree$call
    (GccTree$expr_use(GccTree$use_remote(GccTree$allocate_decl)),
     GccExpressions$
         {GccTree$mult
              (size_args...,
               GccTree$make_integer_cst
                   (size,GccTree$offset_type))});
-- Now we need to initialize the structure (if a RECORD)
-- or set the open array dimensions (if an ARRAY)
case ptr.expr_type begin
  match pointer_type(?ty) begin
    case ty.base_type begin
      match ?rt=record_type(...) begin
        assign : GccTree$Expression :=
            GccTree$assign(ptr.gcc_expr,call);
        e.gcc_expr := GccTree$assign
            (GccTree$component_ref
                 (GccTree$indirect_ref(assign),
                  GccTree$use_remote(rt.type_spec_field)),
             GccTree$make_convert
                 (GccTree$address
                      (GccTree$expr_use
                           (GccTree$use_remote
                                (rt.decl_for_type
                                     .type_desc_decl))),
                  GccTree$pointer_type));
      end;
      match ?at=open_array_type(...) begin
        record : GccTree$Expression := GccTree$(constructor)
            (GccTree$Expressions$
                 {call,
                  GccTree$reuse_expr(size)
                      for size in size_args});
        record.GccTree$expr_type := at.gcc_type;
        e.gcc_expr := GccTree$assign
            (ptr.gcc_expr,
             record);
      end;
    end; -- case base type
```

```
          end;
        end; -- case ptr type
      end;
    end; -- case args
  end; -- match NEW
else
  -- a regular call or a fetched call:
  actuals : GccTree$Expressions;
  case h begin
    match header(formals:=?formals) begin
      actuals := GccExpressions$
          {make_actual(arg,arg.gcc_expr,
                       nth(position(arg,args),formals))
               for arg in args,
           make_optional_actual(last(formals))
               if length(formals) > length(args)};
    end;
  end;

  case h begin
    -- an object call:
    match header(receiver:=receiver(?)) begin
      case func begin
        match fref(?obj,?,?) begin
          obj_ptr : GccTree$Expression :=
              GccTree$reuse_expr(func.method_call_object);
          e.gcc_expr :=
              GccTree$call(func.gcc_expr,
                           GccExpressions${obj_ptr,actuals...});
        end;
      end;
    end;
    match header(formals:=?formals) begin
      -- we may need to distinguish between function pointers and functions
      -- but I hope not.
      e.gcc_expr := GccTree$call(func.gcc_expr,actuals);
    end;
  end;
 end;
end;

-- Actual Parameters:
--
procedure make_actual(arg : remote Expression;
                      tree : GccTree$Expression;
                      formal : remote Declaration) : GccTree$Expression
begin
  -- Here's a function to help us detect array types
  function is_array_type(ty : remote Type) : Boolean begin
    case ty.base_type begin
```

```
      match array_type(...) begin result := true; end;
    else
      result := false;
    end;
  end;

  case formal begin
    -- arrays are passed the same whether or not we have a
    -- var or value formal.
    match formal(?,?ty=open_array_type(?)) begin
      result := GccTree$(constructor)
          (GccExpressions$
               {tree,
                GccTree$reuse_expr(nth(i,arg.gcc_dimensions))
                    for i in 0..(ty.open_ranges-1)});
      result.GccTree$expr_type := ty.gcc_type;
    end;
    -- regular arrays are passed just with the pointer
    -- regardless of var/value status:
    match formal(?,?ty) if is_array_type(ty) begin
      result := tree;
    end;
    -- other var formals passed by reference:
    match var_formal(?,?ty) begin
      result := GccTree$address(tree);
    end;
    -- other value formals passed by value
    -- (include large records.), but they must be coerced into
    -- the right type:
    match value_formal(?,?ty) begin
      result := convert_expr(arg,tree,ty);
    end;
  end;
end;

procedure make_optional_actual(f : remote Declaration) : GccTree$Expression
begin
  case f begin
    match opt_formal(?,?default) begin
      result := gen_constant_value(default);
    end;
  end;
end;


-- type tests:
-- the first argument is a pointer and the second a (record) type.
-- we have to check out the type descriptor.
-- From ob2-layout.aps:
-- to tell whether a certain dynamic type "td" is or extends statically
```

```
-- known type T, do the following:
--           if ((int)*td >= T.desc_size &&
--                td[T.desc_index] == T) ...
-- But, now I've typed the whole thing with records, and so we do:
--           if (td.typespec_size >= ... &&
--               td.T_ref = &T) ...
procedure gen_type_test(p : GccTree$Expression;
                        ty : remote Type) : GccTree$Expression begin
  spec_field : remote GccTree$Declaration;
  case ty.gcc_type begin
    match GccTree$record(?fs) begin
      spec_field := first(fs);
    end;
  end;
  fields : remote GccFields;
  case ty.spec_type begin
    match GccTree$record(?fs) begin
      fields := fs;
    end;
  end;
  -- perform the cast and then check it:
  new_p : GccTree$Expression :=
      GccTree$make_convert(p,GccTree$pointer(ty.gcc_type));
  td : GccTree$Expression := GccTree$save_expr
      (GccTree$component_ref(GccTree$indirect_ref(new_p),
                             GccTree$use_remote(spec_field)));
  result :=
      GccTree$truth_andif
      (GccTree$ge(GccTree$component_ref(GccTree$indirect_ref(td),
                                        GccTree$use_remote(first(fields))),
                  GccTree$make_integer_cst(ty.desc_size,
                                           GccTree$offset_type)),
       GccTree$eq
           (GccTree$component_ref
                (GccTree$indirect_ref(GccTree$reuse_expr(td)),
                 GccTree$use_remote(nth(1+ty.desc_index,fields))),
            GccTree$address
                (GccTree$expr_use
                     (GccTree$use_remote(ty.decl_for_type
                                         .type_desc_decl)))));
end;

match ?e=is_test(?e1,named_type(?u)) begin
  e.gcc_expr := gen_type_test(e1.gcc_expr,use_base_record_type(u));
end;

-- a type guard is like a type test, but if the test fails,
-- type_guard_failed is called:
procedure gen_type_guard(p : GccTree$Expression;
                         ty : remote Type) : GccTree$Expression begin
```

```
  case ty.decl_for_type begin
    match type_decl(identifier(?name,...),...) begin
      result :=
          GccTree$conditional
          (gen_type_test(p,ty),
           GccTree$reuse_expr(p),
           -- we have to convert the result back to satisfy the
           -- type checker that doesn't know that the function
           -- never returns:
           GccTree$make_convert
           (GccTree$call
                (GccTree$expr_use
                     (GccTree$use_remote(GccTree$type_guard_failed_decl)),
                 GccExpressions$
                     {GccTree$make_convert(GccTree$reuse_expr(p),
                                           GccTree$pointer_type),
                      GccTree$string_cst(symbol_name(name))}),
            GccTree$pointer_type));
    end;
  end;
end;

match ?e=actual_type_guard(?value,?u) begin
  e.gcc_expr := gen_type_guard(GccTree$save_expr(value.gcc_expr),
                               use_base_record_type(u));
end;

function use_base_record_type(u : remote Use) : remote Type begin
  case u.use_decl begin
    match type_decl(?,?ty) begin
      case ty.base_type begin
        match ?ty=record_type(...) begin result := ty; end;
        match pointer_type(?ty) begin result := ty.base_type; end;
      end;
    end;
  end;
end;

-- arrays:
-- taking a slice of an array is a pointer operation,
-- getting a real element has an indirection in it.
-- This code relies on multiplication of constants being done "smart".
match ?e=aref(?p,?index) begin
  array : GccTree$Expression;
  array_type : remote Type;
  array_dimensions : GccRemoteExpressionList;
  case p.expr_type begin
    match array_type(...) begin
      array := p.gcc_expr;
      array_type := p.expr_type;
```

```
      array_dimensions := p.gcc_dimensions;
    end;
    match pointer_type(?rt) begin
      array_type := rt.base_type;
      ae : ArrayExpr :=
          make_array_expr(array_type,GccTree$save_expr(p.gcc_expr));
      array := ae.gcc_array_expr;
      array_dimensions := ae.gcc_array_dimensions;
    end;
  end;

  max : remote GccTree$Expression := first(array_dimensions);
  index_expr : GccTree$Expression := GccTree$save_expr
      (GccTree$make_convert(index.gcc_expr,GccTree$offset_type));

  -- note array assignment is done by element-to-element copy,
  -- (and is only directly supported for strings)
  -- and so the element pointer needn't be an lvalue:
  element_ptr : GccTree$Expression :=
      GccTree$conditional
      (GccTree$truth_andif(GccTree$ge(index_expr,
                                      GccTree$make_integer_cst
                                          (0,GccTree$offset_type)),
                           GccTree$lt(GccTree$reuse_expr(index_expr),
                                      GccTree$reuse_expr(max))),
       GccTree$plus
           (array,
            GccTree$mult
                (GccTree$reuse_expr(index_expr),
                 GccTree$reuse_expr(d)
                     for d in butfirst(array_dimensions))),
       GccTree$make_convert
           (GccTree$call
                (GccTree$expr_use
                     (GccTree$use_remote(GccTree$array_index_error_decl)),
                 GccExpressions$
                     {GccTree$reuse_expr(index_expr),
                      GccTree$reuse_expr(max)}),
            array_type.gcc_expr_type));
  case e.expr_type begin
    match array_type(...) begin
      e.gcc_expr := element_ptr;
    end;
  else
    e.gcc_expr := GccTree$indirect_ref(element_ptr);
  end;

  e.gcc_dimensions := butfirst(array_dimensions);
end;
```

```
-- field references are either normal field references or they are
-- method references.  In the latter case we have to handle the
-- super flag.  In the former case, we have to special case arrays.
-- We must also dereference pointers as necessary.
-- in any case, we should ignore module.var references.
match ?e=fref(?r,?u,?super) begin
  obj_ptr : GccTree$Expression;
  rec_type : remote Type := nil;
  case r.expr_type begin
    match record_type(...) begin
      obj_ptr := GccTree$address(r.gcc_expr);
      rec_type := r.expr_type;
    end;
    match pointer_type(?rt) begin
      obj_ptr := r.gcc_expr;
      rec_type := rt.base_type;
    end;
  end;

  case u.use_decl begin
    match ?f=field(?,?ty) begin
      -- make_array_expr *does* not work for non-arrays.
      selection : GccTree$Expression :=
          make_component_ref(GccTree$indirect_ref(obj_ptr),f);
      case ty.base_type begin
        match array_type(...) begin
          ae : ArrayExpr := make_array_expr(ty,GccTree$save_expr(selection));
          e.gcc_expr := ae.gcc_array_expr;
          e.gcc_dimensions := ae.gcc_array_dimensions;
        end;
      else
        e.gcc_expr := selection;
      end;
    end;
    match ?m=proc_decl(header:=?h=header(receiver:=receiver(...))) begin
      saved_obj_ptr : GccTree$Expression := GccTree$save_expr(obj_ptr);
      record : GccTree$Expression := GccTree$indirect_ref(saved_obj_ptr);
      e.method_call_object := saved_obj_ptr;
      if super then
        -- simple, no dispatching:
        case rec_type begin
          match record_type(?parent,?) begin
            e.gcc_expr :=
                GccTree$compound
                (record, -- use so that we can reuse later.
                 GccTree$make_convert
                     (GccTree$address
                          (GccTree$expr_use
                               (GccTree$use_remote
                                    (nth(m.method_index,parent.methods)
```

```
                                      .gcc_decl))),
                    GccTree$pointer(h.gcc_function_type)));
          end;
        end;
      else
        -- dispatching:
        td : GccTree$Expression :=
            GccTree$component_ref
            (record,
             GccTree$use_remote(rec_type.type_spec_field));
        fields : remote GccFields;
        case rec_type.spec_type begin
          match GccTree$record(?fs) begin
            fields := fs;
          end;
        end;
        e.gcc_expr :=
            GccTree$indirect_ref
            (GccTree$component_ref
                 (GccTree$indirect_ref(td),
                  GccTree$use_remote(nth(1+m.method_index,fields))));
      endif;
      --e.gcc_expr.GccTree$expr_type := GccTree$pointer(h.gcc_function_type);
    end;
  else
    -- we must have a module reference masquerading as a field reference:
    e.gcc_expr := GccTree$expr_use(GccTree$use_remote(u.use_decl.gcc_decl));
  end;
end;

match ?e=fetch(?p) begin
  case p.expr_type begin
    match pointer_type(open_array_type(...)) begin
      ae : ArrayExpr := make_array_expr
          (e.expr_type,GccTree$save_expr(p.gcc_expr));
      e.gcc_expr := ae.gcc_array_expr;
      e.gcc_dimensions := ae.gcc_array_dimensions;
    end;
  else
    e.gcc_expr := GccTree$indirect_ref(p.gcc_expr);
  end;
end;

gcc_set_type : GccTree$Type := GccTree$integer(true,set_size*byte_bits);

match ?e=set_expr(?elements) begin
  e.gcc_expr :=
      GccTree$bit_or(e.gcc_element for e in elements,
                     GccTree$make_integer_cst(0,gcc_set_type));
end;
```

```
  match ?e=single_element(?v) begin
    e.gcc_element :=
        GccTree$lshift(GccTree$make_integer_cst(1,gcc_set_type),
                       v.gcc_expr);
  end;

  match ?e=range_element(?e1,?e2) begin
    -- {x..y} = (1<<(y+1))-(1<<x)
    e.gcc_element := GccTree$minus
        (GccTree$lshift
             (GccTree$make_integer_cst(1,gcc_set_type),
              GccTree$plus(e2.gcc_expr,
                           GccTree$make_integer_cst(1,gcc_set_type))),
         GccTree$lshift(GccTree$make_integer_cst(1,gcc_set_type),e1.gcc_expr));
  end;

end;
```

B.7 Translating GCC Trees to C text

This module creates string for each compilation unit and a string to be written as the header file. the semantics of the switch statement nodes in GCC tree form is unclear and so switch statements are currently not transkated.

```
module GCC2C[T :: var GCC_TREE[]] extends T
begin

  attribute CompilationUnit.program_text : String;

  attribute Declaration.is_top_level : Boolean := false;

  match ?c=compilation_unit(?decls) begin
    c.program_text :=
        "/* Automatically generated GCC dialect C code */\n" ||
        "/* Generated by the GCC tree to C converted written in APS */\n" ||
        "/* John Boyland, 1996 */\n" ||
        "#include \"header.h\"\n\n" ||
        (decl.text for decl in decls);
    for decl in decls begin
      decl.is_top_level := true;
    end;
  end;

  var header_text : String :=
      "/* Automatically generated GCC dialect C code */\n" ||
      "/* Generated by the GCC tree to C converted written in APS */\n" ||
      "/* John Boyland, 1996 */\n" ||
      type_def_string_part(type_def_string_set...) || "\n" ||
```

```
    (extern_decl_texts...);

-- private;

signature HAS_TEXT := {Block,Declaration,Statement,
                       Type,Expression,Use}, var PHYLUM[];

[T :: HAS_TEXT] attribute T.text : String := "";

attribute Declaration.type_text : String := ""; -- for parm_decl's

function comma(s,base : String) : String
begin
  if base = "" then
    result := s;
  else
    result := s || "," || base;
  endif;
end;


--- Indentation

signature INDENTING := {Statement, Block, Declaration}, var PHYLUM[];
signature SEQ := {Statements,Declarations,Fields}, var PHYLUM[];
[phylum T :: INDENTING] attribute T.depth : Integer := 0;

function make_indent(i : Integer) : String begin
  if i > 0 then
    result := "  " || make_indent(i-1);
  else
    result := "";
  endif;
end;

[T :: INDENTING] attribute (node:T).indent : String :=
    make_indent(node.depth);



---- Saved Expressions

type SavedDeclTexts := LIST[String];
attribute (e:Expression).saved_decl_texts : SavedDeclTexts :=
    {c.saved_decl_texts... for c in e.expression_children};
type ExpressionChildren := ORDERED_SET[remote Expression]((==),(<<));
collection attribute Expression.expression_children : ExpressionChildren;
match ?e1:Expression=parent(?e2:Expression) begin
  e1.expression_children :> {e2};
end;
```

```
match ?e1:Expression=parent(?children:Expressions) begin
  e1.expression_children :> {children...};
end;

match ?e1=save_expr(?e2) begin
  sym : Symbol := gensym();
  e1.text := symbol_name(sym);
  e1.saved_decl_texts := e2.saved_decl_texts ++
      {e2.expr_type.tmp_text || " " || sym || " = " || e2.text || ";\n"};
  pragma print();
end;

var function tmp_text(ty : Type) : String begin
  case ty.base_type begin
    match array(?,?et) begin
      result := et.text ++ " *";
    end;
  else
    result := ty.text;
  end;
end;

function prefix_saved(prefix : String; saved : SavedDeclTexts) : String :=
    ((prefix ++ s) for s in saved) ++ "";

function block_saved(prefix : String; saved : SavedDeclTexts; stmt : String)
    : String
begin
  if saved = {} then
    result := prefix || stmt;
  else
    result := prefix || "{\n" || prefix_saved(prefix++"  ",saved) ||
          "  " || prefix || stmt || prefix || "}\n";
  endif;
end;



---- Type Def String:

type TypeDefString;
constructor type_def_string(priority : Integer; s : String) : TypeDefString;

function type_def_string_part(tds : TypeDefString) : String begin
  case tds begin
    match type_def_string(?,?s) begin
      result := s;
    end;
  end;
end;
```

```
function type_def_string_less(tds1,tds2 : TypeDefString) : Boolean
begin
  case tds1 begin
    match type_def_string(?p1,?s1) begin
      case tds2 begin
        match type_def_string(?p2,?s2) begin
          result := p1 < p2 or p1 = p2 and s1 < s2;
        end;
      end;
    end;
  end;
end;

type TypeDefStringSet :=
    ORDERED_SET[TypeDefString]((=),type_def_string_less);

var collection type_def_string_set : TypeDefStringSet;

attribute Type.priority : Integer := 0;



---- Extern Declarations

type Strings := BAG[String];

var collection extern_decl_texts : Strings;


---- Declarations

attribute Expression.is_initializer : Boolean := false;

match ?d=parm_decl(?,?ty) begin
  d.text := ty.text || " " || d.assembler_name;
  d.type_text := ty.text;
end;

match ?d=const_decl(?,?ty,?init) begin
  d.text :=
      prefix_saved(d.indent,init.saved_decl_texts) ||
      d.indent ||
      "extern " if d.is_external ||
      "static " if (d.is_top_level and not d.is_public) ||
      "const " || ty.text || " " || d.assembler_name ||
      initial_text || ";\n";
  initial_text : String;
  if init.text = "" then
    initial_text:="";
  else
```

```
    initial_text := " = " || init.text;
  endif;
  init.is_initializer := true;
  if d.is_top_level and d.is_public and not d.is_external then
    extern_decl_texts :>
        {"extern const " || ty.text || " " || d.assembler_name || ";\n"};
  endif;
end;

match ?d=var_decl(?,?ty,?init) begin
  d.text :=
      prefix_saved(d.indent,init.saved_decl_texts) ||
      d.indent ||
      "extern " if d.is_external ||
      "static " if (d.is_top_level and not d.is_public) ||
      ty.text || " " || d.assembler_name || initial_text || ";\n";
  initial_text : String;
  if init.text = "" then
    initial_text:="";
  else
    initial_text := " = " || init.text;
  endif;
  init.is_initializer := true;
  if d.is_top_level and d.is_public and not d.is_external then
    extern_decl_texts :>
        {"extern " || ty.text || " " || d.assembler_name || ";\n"};
  endif;
end;

match ?d=field_decl(?,?ty,...) begin
  d.text := d.indent || ty.text || " " || d.assembler_name || ";\n";
end;

match ?d=type_decl(?,?ty) begin
  -- these types are never used:
  d.text := d.indent || "typedef " || ty.text || " " ||
      d.assembler_name || "; /* UNUSED */\n";
end;

match ?d=function_decl(?,?,?args,?rd,?body) begin
  d.text :=
      d.indent ||
      "extern " if d.is_external ||
      "static " if (d.is_top_level and not d.is_public) ||
      rd.text || " " || d.assembler_name || "(" ||
      comma(arg.text for arg in args,"") || ")" ||
      body_text || "\n";
  body_text : String;
  if body.text = "" then
    body_text := ";";
```

```
  else
    body_text := body.text;
  endif;
  if d.is_top_level and d.is_public and not d.is_external then
    extern_decl_texts :>
        {"extern " || rd.text || " " || d.assembler_name || "(" ||
             comma(arg.type_text for arg in args,"") || ");\n"};
  endif;
end;

match ?d=result_decl(ty:=?ty) begin
  d.text := ty.text;
end;



--- Blocks

match ?b=block(?decls,?stmts) begin
  b.text :=
      b.indent || "{\n" ||
      (decl.text for decl in decls) ||
      (stmt.text for stmt in stmts) ||
      b.indent || "}\n";
end;

match ?b=no_block() begin
  b.text := b.indent || "{}\n";
end;



---- Uses

match ?u=a_use() begin
  u.text := symbol_name(u.use_decl.assembler_name);
end;



---- Statements

match ?s=no_stmt() begin
  s.text := "";
end;


match ?s=do(?expr) begin
  s.text := block_saved(s.indent,expr.saved_decl_texts,
                        expr.text || ";\n");
```

```
end;


match ?s=label(?l) begin
  s.text := s.indent || l.assembler_name || ":\n";
end;

match ?s=goto(?u) begin
  s.text := s.indent || "goto " || u.use_decl.assembler_name || ";\n";
end;


match ?s=cond(?expr,?then_part,?else_part) begin
  s.text := block_saved(s.indent,expr.saved_decl_texts,
                        "if (" || expr.text || ")\n" ||
                            then_part.text || else_prefix || "else\n" ||
                            else_part.text);
  else_prefix : String := s.indent;
  if expr.saved_decl_texts /= {} then
    then_part.depth := s.depth+2;
    else_part.depth := s.depth+2;
    else_prefix := s.indent ++ "  ";
  endif;
end;


match ?s=loop(?prologue,?body) begin
  s.text := prologue.text || s.indent || "for (;;)\n" || body.text;
end;

match ?s=continue() begin
  s.text := s.indent || "continue;\n";
end;

match ?s=exit() begin
  s.text := s.indent || "exit;\n";
end;


match ?s=return(?expr) begin
  if expr.text = "" then
    s.text := s.indent ++ "return;\n";
  else
    s.text := block_saved(s.indent,expr.saved_decl_texts,
                          "return " ++ expr.text ++ ";\n");
  endif;
end;

-- switch and cases omitted because semantics of intermediate
-- form is not clear.
```

```
match ?s=block_stmt(?b) begin
  s.text := b.text;
end;

match ?s=seq(?stmts) begin
  for s1 in stmts begin
    s1.depth := s.depth;
  end;
  s.text := (stmts...).text || "";
end;



---- Types


match ?ty=type_use(?u) begin
  case u.use_decl begin
    match type_decl(?,?ty2) begin
      ty.priority := ty2.priority;
      ty.text := ty2.text;
    end;
  end;
end;


--!! many machine dependent assumptions
match ?ty=integer(!true,!pointer_size) begin
  ty.text := "unsigned long";
end;

match ?ty=integer(!true,?) begin
  ty.text := "unsigned";
end;

match ?ty=integer(...) begin
  ty.text := "int";
end;


match ?ty=enumeral(?,?elems) begin
  ty.text := symbol_name(gensym());
  type_def_string_set :>
      {type_def_string(0,"typedef enum " || ty.text ||
                          "{" || comma((elems...).text,"") || "}" ||
                          ty.text || ";\n")};
end;
```

```
match ?ty=boolean() begin
  ty.text := "int";
end;


match ?ty=char() begin
  ty.text := "char";
end;


match ?ty=real(!32) begin
  ty.text := "float";
end;

match ?ty=real(!64) begin
  ty.text := "double";
end;


match ?ty=range(?,?,?base) begin
  ty.text := base.text;
end;


match ?ty=array(range(integer_cst(!0),integer_cst(?max),?),?base) begin
  ty.priority := base.priority+1;
  ty.text := symbol_name(gensym());
  type_def_string_set :>
      {type_def_string(ty.priority,
                       "typedef " || base.text || " " || ty.text ||
                           "[" || (max+1) || "];\n")};
end;


pattern record_or_union(fields : Fields) : TypePhylum :=
    record(?fields), union(?fields);

match ?ty=record_or_union(?fields) begin
  collection max_priority : Integer :> 0, integer_max;
  for field in fields begin
    case field begin
      match field_decl(?,?fty,...) begin
        max_priority :> fty.priority;
      end;
    end;
  end;
  ty.priority := max_priority+1;
  ty.text := symbol_name(gensym());
  for field in fields begin
    field.depth := 1;
```

```
  end;
end;

match ?ty=record(?fields) begin
  type_def_string_set :>
      {type_def_string(ty.priority,
                       "typedef struct " || ty.text || " {\n" ||
                           (fields...).text || "} " ++ ty.text ++ ";\n")};
end;

match ?ty=union(?fields) begin
  type_def_string_set :>
      {type_def_string(ty.priority,
                       "typedef union " || ty.text || " {" ||
                           (fields...).text || "} " ++ ty.text ++ ";\n")};
end;


match ?ty=complex(?base) begin
  ty.text := "__complex__ " || base.text;
  ty.priority := base.priority+1;
end;


match ?ty=pointer(?base) begin
  case base_type(base) begin
    match record(...) begin
      ty.text := "struct " ++ base.text ++ " *";
      ty.priority := 0;
    end;
    match union(...) begin
      ty.text := "union " ++ base.text ++ " *";
      ty.priority := 0;
    end;
  else
    ty.text := base.text || " *";
    ty.priority := base.priority+1;
  end;
end;

match ?ty=reference(?base) begin
  ty.text := base.text || " &";
  ty.priority := base.priority+1;
end;


match ?ty=void() begin
  ty.text := "void";
end;
```

```
match ?ty=function_type(?args,?return) begin
  collection max_priority : Integer :> 0, integer_max;
  max_priority :> return.priority;
  for arg in args begin
    max_priority :> arg.priority;
    case arg.base_type begin
      match pointer(?ty) begin
        -- can't allow "struct T *" before struct T defined.
        max_priority :> ty.priority;
      end;
    end;
  end;
  ty.priority := 1+max_priority;
  ty.text := symbol_name(gensym());
  type_def_string_set :>
      {type_def_string(ty.priority,
                       "typedef " || return.text || " " || ty.text || "(" ||
                           comma((args...).text,"") || ");\n")};
end;



---- Expressions

-- We keep track of expressions that don't do anything
attribute (e:Expression).no_effect : Boolean := false;

-- Saved expressions are always no_effect because
-- the results are computed off line:
match ?e=save_expr(...) begin
  e.no_effect := true;
end;

match ?e=no_expr() begin
  e.text := "";
  e.no_effect := true;
end;

match ?e=integer_cst(?v) begin
  e.text := "(" || e.expr_type.text || ")" || v;
  e.no_effect := true;
end;

match ?e=single_real_cst(?v) begin
  e.text := "(float)" || v;
  e.no_effect := true;
end;

match ?e=double_real_cst(?v) begin
```

```
  e.text := "(double)" || v;
  e.no_effect := true;
end;

function escapify(c : Character) : String begin
  case c begin
    match !'"' begin
      result := "\\\"";
    end;
    match !'\\' begin
      result := "\\\\";
    end;
    match !'\n' begin
      result := "\\n";
    end;
  else
    result := {c};
  end;
end;

match ?e=string_cst(?v) begin
  e.text := {'"',escapify(c)... for c in v,'"'};
  e.no_effect := true;
end;

match ?e=complex_cst(?r,?i) begin
  e.text := r.text || "+" || i.text || "i";
  e.no_effect := true;
end;


match ?e=expr_use(?u) begin
  e.text := u.text;
  e.no_effect := true;
end;

match ?e=unit() begin
  e.text := "(void)0";
  e.no_effect := true;
end;


match ?e=component_ref(?obj,?u) begin
  e.text := "(" || obj.text || ")." || u.text;
  e.no_effect := obj.no_effect;
end;

match ?e=indirect_ref(?p) begin
  e.text := "*(" || p.text || ")";
  e.no_effect := p.no_effect;
```

```
end;

match ?e=array_ref(?a,?indices) begin
  e.text := "(" || a.text || ")" ||
      ("[" || i.text || "]" for i in indices);
  e.no_effect := a.no_effect and (i.no_effect for i in indices);
end;

match ?e=(constructor)(?objs) begin
  ctext : String := "{" || comma(obj.text for obj in objs,"") || "}";
  if e.is_initializer then
    e.text := ctext;
  else
    e.text := "(" || e.expr_type.text || ")" || ctext;
  endif;
  e.no_effect := true and (obj.no_effect for obj in objs);
end;

match ?e=compound(?e1,?e2) begin
  if e1.no_effect then
    e.text := e2.text;
    e.no_effect := e2.no_effect;
  elsif e2.text = "" then
    e.text := "(void)" || e1.text;
  else
    e.text := "(" || e1.text || "," || e2.text || ")";
  endif;
end;


match ?e=assign(?l,?r) begin
  e.text := l.text || "=" || r.text;
end;

match ?e=initialize(?u,?r) begin
  e.text := u.text || "=" || r.text;
end;


match ?e=conditional(?c,?e1,?e2) begin
  e.text := "(" || c.text || ") ? (" ||
      e1.text || ") :( " || e2.text || ")";
end;

match ?e=call(?f,?args) begin
  e.text := "(" || f.text || ")(" ||
      comma(arg.text for arg in args,"") || ")";
end;
```

```
-- nodes which are directly represented by C's binary operations
pattern c_binop_node(e1,e2 : Expression) : Expression :=
    plus(?e1,?e2), minus(?e1,?e2), mult(?e1,?e2), trunc_div(?e1,?e2),
    trunc_mod(?e1,?e2),  div(?e1,?e2), lshift(?e1,?e2), rshift(?e1,?e2),
    bit_or(?e1,?e2), bit_and(?e1,?e2), bit_andtc(?e1,?e2), bit_xor(?e1,?e2),
    truth_andif(?e1,?e2), truth_orif(?e1,?e2), truth_xor(?e1,?e2),
    lt(?e1,?e2),le(?e1,?e2),gt(?e1,?e2),ge(?e1,?e2),eq(?e1,?e2),ne(?e1,?e2),
    make_complex(?e1,?e2),
    --!! The following will not work when applied to pointers:
    predecrement(?e1,?e2),preincrement(?e1,?e2);

pattern c_unop_node(e : Expression) : Expression :=
    address(?e),fix_trunc(?e),negate(?e),bit_not(?e),truth_not(?e),
    conj(?e),realpart(?e),imagpart(?e);

function c_op_name(n : remote Expression) : String begin
  case n begin
    -- binary
    match plus(...) begin result := "+"; end;
    match minus(...) begin result := "-"; end;
    match mult(...)  begin result := "*"; end;
    match trunc_div(...) begin result := "/"; end;
    match trunc_mod(...) begin result := "%"; end;
    match div(...) begin result := "/"; end;
    match lshift(...) begin result := "<<"; end;
    match rshift(...) begin result := ">>"; end;
    match bit_or(...) begin result := "|"; end;
    match bit_and(...) begin result := "&"; end;
    match bit_andtc(...) begin result := "&~"; end;
    match bit_xor(...) begin result := "^"; end;
    match truth_andif(...) begin result := "&&"; end;
    match truth_orif(...) begin result := "||"; end;
    match truth_xor(...) begin result := "^^"; end;
    match lt(...) begin result := "<"; end;
    match le(...) begin result := "<="; end;
    match gt(...) begin result := ">"; end;
    match ge(...) begin result := ">="; end;
    match eq(...) begin result := "=="; end;
    match ne(...) begin result := "!="; end;
    match complex_cst(...) begin result := "+1i*"; end;
    --!! See note above:
    match preincrement(...) begin result := "+="; end;
    match predecrement(...) begin result := "-="; end;
    -- unary
    match address(...) begin result := "&"; end;
    match fix_trunc(...) begin result := "(long)"; end;
    match negate(...) begin result := "-"; end;
    match bit_not(...) begin result := "~"; end;
    match truth_not(...) begin result := "!"; end;
    match conj(...) begin result := "~"; end;
```

```
    match realpart(...) begin result := "__real__"; end;
    match imagpart(...) begin result := "__imag__"; end;
  end;
end;

match ?e=c_binop_node(?e1,?e2) begin
  e.text := "(" || e1.text || ")" || c_op_name(e) || "(" || e2.text || ")";
end;

match ?e=c_unop_node(?arg) begin
  e.text := c_op_name(e) || "(" || arg.text || ")";
end;

-- we need floor_XXX for the Oberon2 compiler
match ?e=floor_div(?e1,?e2) begin
  e.text := "FLOOR_DIV(" || e1.text || "," || e2.text || ")";
end;
match ?e=floor_mod(?e1,?e2) begin
  e.text := "FLOOR_MOD(" || e1.text || "," || e2.text || ")";
end;

match ?e=min(?e1,?e2) begin
  e.text := "MIN(" || e1.text || "," || e2.text || ")";
end;
match ?e=max(?e1,?e2) begin
  e.text := "MAX(" || e1.text || "," || e2.text || ")";
end;
match ?e=abs(?e1) begin
  e.text := "ABS(" || e1.text || ")";
end;

-- the other div and truncs are not implemented yet

match ?e=convert(?e1) begin
  e.text := "(" || e.expr_type.text || ")(" || e1.text || ")";
  e.no_effect := e1.no_effect;
end;


match ?e=reuse_expr(?saved) begin
  e.text := saved.text;
  e.no_effect := saved.no_effect;
end;

-- default indentation rules
[phylum T,U :: INDENTING] begin
  match ?i1:T=parent(?i2:U) begin
    i2.depth := i1.depth+1;
  end;
end;
```

```
  [phylum T,U :: INDENTING; phylum S :: SEQ, SEQUENCE[U]] begin
    match ?i1:T=parent(?s:S) begin
      for i2 in s begin
        i2.depth := i1.depth+1;
      end;
    end;
  end;
end;
```

B.8 Descriptional Composition

```
module OBERON2_COMPOSE[Input :: var OBERON2_TREE[],
                               var OBERON2_RESOLVE[Input],
                               var OBERON2_MACHINE_SIZES[],
                               var OBERON2_COMPILE_COMPUTE[Input],
                               var OBERON2_LAYOUT[Input]]
    extends Input
begin
  var header_text = TextRep$header_text;
  attribute (p:Program).program_text : String :=
      p.gcc_program.TextRep$program_text;

  private;

  inherit OBERON2_TRANSLATE[Input] begin
    type BareGccTree = BareGccTree;
    type GccTree = GccTree;
    gcc_program = gcc_program;
    pragma inline(set_decl_info,make_use,make_component_ref,make_strcmp_call);
  end;

  type TextRep := GCC2C[GccTree];

  pragma expand(type BareGccTree,
                type GccTree,
                type TextRep);

  pragma compose(type GccTree$CompilationUnit,
                 type GccTree$Block,
                 type GccTree$Declaration,
                 type GccTree$Statement,
                 type GccTree$Expression,
                 type GccTree$Use,
                 type GccTree$Declarations,
                 type GccTree$Statements,
                 type GccTree$Expressions,
                 type GccTree$Fields);

  pragma inline(GccTree$make_external_function,
```

```
                GccTree$make_external_var,
                GccTree$result_decl,
                GccTree$use_remote,
                GccTree$make_integer_cst,
                GccTree$make_convert);
end;
```

Appendix C

The APS Compiler: Name Resolution and Type Checking

This section contains portions of the source to the APS compiler written in itself. In particular, it contains a front end that performs name resolution and typechecking. Additionally it contains a transformation for expanding `inherit` declarations, as well as two modules for computing information useful in canonicalizing patterns.

C.1 Abstract Tree

```
module ABSTRACT_APS[] begin

  -- the four basic structures
  phylum Signature;
  phylum Type;
  phylum Expression;
  phylum Pattern;
  -- two related ones;
  phylum Module;
  phylum Class;

  -- Def's and Use's
  phylum Def;
  phylum Use;

  -- structural phyla
  phylum Program;
  phylum Unit;
  phylum Declaration;
  phylum Block;
  phylum Match;
  phylum Direction;
  phylum Default;

  -- sequences
```

```
phylum Units:=SEQUENCE[Unit];
phylum Declarations:=SEQUENCE[Declaration];
phylum Matches:=SEQUENCE[Match];

phylum Types:=SEQUENCE[Type];
phylum Expressions:=SEQUENCE[Expression];
phylum Patterns:=SEQUENCE[Pattern];

phylum Actuals:=SEQUENCE[Expression];
phylum TypeActuals:=SEQUENCE[Type];
phylum PatternActuals:=SEQUENCE[Pattern];

signature PHYLA := {Signature,Type,Expression,Pattern,Module,Class,Def,Use,
                    Program,Unit,Declaration,Block,Match,Direction,Default,
                    Units,Declarations,Matches,Types,Expressions,Patterns,
                    Actuals,TypeActuals,PatternActuals}, var PHYLUM[];

-- The identifier _ is used specially.
underscore_symbol : Symbol := make_symbol("_");

constructor program(name : String; units : Units) : Program;

constructor no_unit() : Unit;
constructor with_unit(name : String) : Unit;
constructor decl_unit(decl : Declaration) : Unit;

constructor no_decl() : Declaration;

-- a "begin" ... "end" block.
constructor block(body : Declarations) : Block;

constructor class_decl(def : Def;
                       type_formals : Declarations;
                       result_type : Declaration;
                       parent : Signature;
                       contents : Block) : Declaration;

constructor module_decl(def : Def;
                        type_formals : Declarations;
                        value_formals : Declarations;
                        result_type : Declaration;
                        parent : Signature;
                          contents : Block) : Declaration;

constructor signature_decl(def : Def;
                           sig : Signature) : Declaration;

constructor phylum_decl(def : Def;
                        sig : Signature;
                        (type) : Type) : Declaration;
```

```
constructor type_decl(def : Def;
                      sig : Signature;
                      (type) : Type) : Declaration;

constructor value_decl(def : Def;
                       (type) : Type;
                       direction : Direction;
                       default : Default) : Declaration;
constructor attribute_decl(def : Def;
                           (type) : Type;
                           direction : Direction;
                           default : Default) : Declaration;

constructor function_decl(def : Def;
                          (type) : Type;
                          body : Block) : Declaration;
constructor procedure_decl(def : Def;
                           (type) : Type;
                           body : Block) : Declaration;
constructor constructor_decl(def : Def;
                             (type) : Type) : Declaration;
constructor pattern_decl(def : Def;
                         (type) : Type;
                         choices : Pattern) : Declaration;

constructor inheritance(def : Def;
                        used : Type;
                        body : Block) : Declaration;

constructor polymorphic(def : Def;
                        type_formals : Declarations;
                        body : Block) : Declaration;

constructor pragma_call(name : Symbol;
                        parameters : Expressions) : Declaration;

constructor top_level_match(m : Match) : Declaration;

-- replacements can only occur in the body of an inheritance
-- and are renamings to take place in the scope of the inherited module:
-- NB: signature, class and module, replacements aren't implemented yet,
--     it can't be type checked
constructor class_replacement((class) : Class; as : Class) : Declaration;
constructor module_replacement((module) : Module; as : Module) : Declaration;
constructor signature_replacement(sig : Signature; as : Signature)
    : Declaration;
constructor type_replacement((type) : Type; as : Type)
    : Declaration;
constructor value_replacement(value : Expression; as : Expression)
    : Declaration;
```

```
constructor pattern_replacement((pattern) : Pattern; as : Pattern)
    : Declaration;

-- renaming can occur anywhere (in inheritance, they particularly useful)
constructor class_renaming(def : Def; old : Class) : Declaration;
constructor module_renaming(def : Def; old : Module) : Declaration;
constructor signature_renaming(def : Def; old : Signature) : Declaration;
constructor type_renaming(def : Def; old : Type) : Declaration;
constructor value_renaming(def : Def; old : Expression) : Declaration;
constructor pattern_renaming(def : Def; old : Pattern) : Declaration;

[phylum T :: {Class,Module,Signature,Type,Expression,Pattern}] begin
  pattern replacement(from,as : T) : Declaration :=
      class_replacement(?from,?as),module_replacement(?from,?as),
      signature_replacement(?from,?as),type_replacement(?from,?as),
      value_replacement(?from,?as), pattern_replacement(?from,?as);
  pattern renaming(def : Def; old : T) : Declaration :=
      class_renaming(?def,?old),module_renaming(?def,?old),
      signature_renaming(?def,?old),type_renaming(?def,?old),
      value_renaming(?def,?old),pattern_renaming(?def,?old);
  pattern some_use(u : Use) : T :=
      class_use(?u) :? T, module_use(?u) :? T, sig_use(?u) :? T,
      type_use(?u) :? T, value_use(?u) :? T, pattern_use(?u) :? T;
end;

pattern some_replacement() : Declaration :=
    class_replacement(?from,?as),module_replacement(?from,?as),
    signature_replacement(?from,?as),type_replacement(?from,?as),
    value_replacement(?from,?as), pattern_replacement(?from,?as);
pattern some_renaming(def : Def) : Declaration :=
    class_renaming(?def,?old),module_renaming(?def,?old),
    signature_renaming(?def,?old),type_renaming(?def,?old),
    value_renaming(?def,?old),pattern_renaming(?def,?old);

-- directions for attributes
constructor direction(is_input : Boolean;
                      is_collection : Boolean;
                      is_circular : Boolean) : Direction;

-- different kinds of Default: a single value or a lattice or nothing
constructor simple(value : Expression) : Default;
constructor composite(initial : Expression;
                      combiner : Expression) : Default;
-- constructor lattice(bottom : Expression;
--                         join : Expression;
--                         equal : Expression) : Default;
constructor no_default() : Default;


-- formals:
```

```
constructor normal_formal(def : Def; (type) : Type) : Declaration;
constructor seq_formal(def : Def; (type) : Type) : Declaration;
pattern formal(def : Def; (type) : Type) : Declaration
    := normal_formal(?def,?(type)), seq_formal(?def,?(type));

constructor type_formal(def : Def; sig : Signature) : Declaration;
constructor phylum_formal(def : Def; sig : Signature) : Declaration;
pattern some_type_formal(def : Def; sig : Signature) : Declaration :=
    type_formal(?def,?sig),phylum_formal(?def,?sig);

-- definition and use:
constructor def(name : Symbol; is_constant, is_public : Boolean) : Def;
constructor use(name : Symbol) : Use;
constructor qual_use(from : Type; name : Symbol) : Use;

-- a useful pattern:
pattern declaration(def : Def) : Declaration
    := class_decl(?def,...), class_renaming(?def,...),
       module_decl(?def,...), module_renaming(?def,...),
       signature_decl(?def,...), signature_renaming(?def,...),
       phylum_decl(?def,...),
       type_decl(?def,...),
       type_renaming(?def,...),
       type_formal(?def,...),
       phylum_formal(?def,...),
       pattern_decl(?def,...),
       constructor_decl(?def,...),
       pattern_renaming(?def,...),
       value_decl(?def,...),
       attribute_decl(?def,...),
       function_decl(?def,...),
       procedure_decl(?def,...),
       value_renaming(?def,...),
       formal(?def,...),
       -- inheritance and polymorphic entities are implicitly named
       inheritance(?def,...),
       polymorphic(?def,...);

pattern some_class_decl(def : Def;
                        type_formals : Declarations;
                        result_type : Declaration;
                        parent : Signature;
                        contents : Block) : Declaration :=
    class_decl(?def,?type_formals,?result_type,?parent,?contents),
    module_decl(?def,?type_formals,?,?result_type,?parent,?contents);

pattern some_type_decl(def : Def; sig : Signature; (type) : Type)
    : Declaration
    := type_decl(?def,?sig,?(type)), phylum_decl(?def,?sig,?(type));
```

```
-- every type declaration or inherit gets a use attached to it:

pattern type_decl_with_predefined_use(def : Def)
    : Declaration
    := type_renaming(?def,...),
       some_type_decl(?def,...),
       some_type_formal(?def,...),
       inheritance(?def,...);

attribute Declaration.predefined_use : Type;
pragma source_transfer(predefined_use);

match ?td=type_decl_with_predefined_use(def(?name,...)) begin
  td.predefined_use := type_use(use(name));
end;


--- now various uses (only the simplest ones for now)

constructor class_use(use : Use) : Class;

constructor module_use(use : Use) : Module;

constructor type_use(use : Use) : Type;
constructor type_inst((module) : Module;
                      type_actuals : TypeActuals;
                      actuals : Actuals) : Type;
constructor no_type() : Type;

constructor value_use(use : Use) : Expression;
no_value = no_expr;
constructor typed_value(expr : Expression; (type) : Type) : Expression;

constructor sig_use(use : Use) : Signature;
constructor sig_inst(is_input,is_var : Boolean;
                     (class) : Class;
                     actuals : TypeActuals) : Signature;
constructor no_sig () : Signature;

constructor pattern_use(use : Use) : Pattern;
constructor no_pattern() : Pattern;
constructor typed_pattern(pat : Pattern; (type) : Type) : Pattern;

-- a list of types: only these type satisfy
constructor fixed_sig(types : Types) : Signature;
-- two signatures:
constructor mult_sig(sig1,sig2 : Signature) : Signature;

-- types have a number of different forms:
constructor remote_type(nodetype : Type) : Type;
```

```
-- constructor void() : Type;

-- constructor internal_list_type(u : Type) : Type;
constructor function_type(formals : Declarations;
                          return_values : Declarations) : Type;
pattern function_typing(formals : Declarations;
                        return_type : Type) : Type
    := function_type(?formals,{value_decl((type):=?return_type)});

-- wrapped around type parameters (for easier checking)
-- constructor type_actual((type) : Type) : Type;
constructor private_type(rep : Type) : Type;

-- patterns
constructor match_pattern(pat : Pattern; (type) : Type) : Pattern;
constructor pattern_call(func : Pattern;
                         actuals : PatternActuals) : Pattern;
-- this is needed for named pattern arguments:
constructor pattern_actual(arg : Pattern;
                             formal : Expression) : Pattern;
--
-- the sugar pattern, e.g.: {?x,...,?y}
-- to remove and replaced with calls to special patterns
-- constructor sequence_pattern(actuals : Patterns; (type) : Type) : Pattern;
constructor rest_pattern(constraint : Pattern) : Pattern;
-- constructor append_pattern(s1,s2 : Pattern) : Pattern;
-- constructor single_pattern(elem : Pattern) : Pattern;
-- constructor nil_pattern() : Pattern;
constructor choice_pattern(choices : Patterns) : Pattern;
constructor and_pattern(p1,p2 : Pattern) : Pattern;
constructor pattern_var(formal : Declaration) : Pattern;
constructor condition(e : Expression) : Pattern;
constructor hole() : Pattern;
constructor pattern_function(formals : Declarations; body : Pattern) : Pattern;

-- statements
-- a begin ... end statement
constructor block_stmt(body : Block) : Declaration;
constructor effect(e : Expression) : Declaration;
-- procall's are added during a canonicalization
constructor multi_call(proc : Expression;
                       actuals : Actuals;
                       results : Actuals) : Declaration;
procall = multi_call; -- old syntax
pattern procall = multi_call; -- old syntax
constructor normal_assign(lhs : Expression; rhs : Expression) : Declaration;
constructor collect_assign(lhs : Expression; rhs : Expression) : Declaration;
pattern assign(lhs : Expression; rhs : Expression) : Declaration :=
    normal_assign(?lhs,?rhs),collect_assign(?lhs,?rhs);
-- replaced with value_decl
```

```
-- constructor local_decl(def : Def;
--                         direction : Direction;
--                         default : Default) : Declaration;
constructor if_stmt(cond : Expression;
                    if_true : Block;
                    if_false : Block) : Declaration;
constructor for_in_stmt(formal : Declaration;
                        seq : Expression;
                        body : Block) : Declaration;
-- constructor for_on_stmt(formal : Def;
--                          seq : Expression;
--                          body : Block) : Declaration;
constructor for_stmt(expr : Expression;
                     matchers : Matches) : Declaration;
constructor case_stmt(expr : Expression;
                      matchers : Matches;
                      default : Block) : Declaration;
constructor matcher(pat : Pattern;
                    body : Block) : Match;

-- expressions
-- (see var_name, too)
constructor integer_const(token : String) : Expression;
constructor real_const(token : String) : Expression;
constructor string_const(token : String) : Expression;
constructor char_const(token : String) : Expression;
constructor undefined() : Expression;
constructor no_expr() : Expression;
constructor funcall(f : Expression; actuals : Actuals) : Expression;
-- these are stuck in in the place of actuals.
-- constructor actual(e : Expression;
--                     formal : Expression) : Expression;
-- We use ... syntax now:
-- constructor reduce(f : Expression;
--                     elems : Expressions) : Expression;
-- constructor sequence(elems : Expressions; (type) : Type) : Expression;
constructor append(s1,s2 : Expression) : Expression;
-- introduced for empty sequences.
constructor empty() : Expression;
-- sugar: the key returned for this constructor
-- constructor constructor_key(name : Expression) : Expression;
-- the following are only legal in pragma's
constructor class_value(c : Class) : Expression;
constructor module_value(m : Module) : Expression;
constructor signature_value(s : Signature) : Expression;
constructor type_value(t : Type) : Expression;
constructor pattern_value(p : Pattern) : Expression;

-- used in set comprehensions
constructor repeat(expr : Expression) : Expression;
```

```
  constructor guarded(expr : Expression; cond : Expression) : Expression;
  constructor controlled(expr : Expression;
                         formal : Declaration;
                         set : Expression) : Expression;
end;
```

C.2 Name Resolution and Environments

Name resolution computes the declaration referred to in each use, and it also computes the binding of type formals in the scope of the declaration. The bindings are made explicit in environments. The `APS_ENVIRON` module defines the `Environment` type.

C.2.1 Symbol Table

```
-- Creates contours for every declaration context and places every declaration
-- in the proper contour.  Lookup isn't handled here.
-- See aps-lookup.
module APS_SYMTAB[Input :: var ABSTRACT_APS[]] extends Input begin

  --- Scope Objects:

  -- each Contour is an object
  phylum Contour;

  constructor root_contour(name : Symbol) : Contour;
  constructor nested_contour(parent : remote Contour) : Contour;

  type Scope := remote Contour;

  type Decls := ORDERED_SET[remote Declaration]((==),(<<));

  -- Basically all declarations here are at the top-level,
  -- but we only find things in files declared with "with" declarations.
  type Scopes := SET[Scope]((==));
  collection attribute Contour.also_search : Scopes;

  collection attribute Contour.value_decls : Decls;
  collection attribute Contour.type_decls : Decls;
  collection attribute Contour.pattern_decls : Decls;
  collection attribute Contour.signature_decls : Decls;

  -- a set of polymorphic(...) things in this scope
  collection attribute Contour.poly_decls : Decls;

  --- Scope Attributes

  -- the scope attribute is the lexical scope of a tree node.
  -- It is not necessarily the scope that is used to find something
  -- (for example in the tree nodes expressing an "inherit" declaration)
```

```
-- These are the nodes where a scope may be placed:
-- (Direction does not appear here)
signature SCOPABLE := {Signature,Type,Expression,Pattern,Class,Module,
                       Unit,Declaration,Block,Use,
                       Match,Default}, var PHYLUM[];

[phylum T :: SCOPABLE] attribute T.scope : Scope;

-- The decl_scope is the scope where a declaration is inserted.
-- It differs from "scope" only for declarations inside an inherit
-- or a polymorphic scope.

attribute (d:Declaration).decl_scope : Scope := d.scope;

pattern some_named_signature_decl(name : Symbol) : Declaration
    := class_decl(def(?name,...),...), class_renaming(def(?name,...),...),
       signature_decl(def(?name,...),...),
       signature_renaming(def(?name,...),...),
       module_decl(def(?name,...),...), module_renaming(def(?name,...),...);

pattern some_named_type_decl(name : Symbol) : Declaration
    := module_decl(def(?name,...),...), module_renaming(def(?name,...),...),
       phylum_decl(def(?name,...),...),
       type_decl(def(?name,...),...),
       type_renaming(def(?name,...),...),
       type_formal(def(?name,...),...),
       phylum_formal(def(?name,...),...);

pattern some_named_pattern_decl(name : Symbol) : Declaration
    := pattern_decl(def(?name,...),...),
       constructor_decl(def(?name,...),...),
       pattern_renaming(def(?name,...),...);

pattern some_named_value_decl(name : Symbol) : Declaration
    := value_decl(def(?name,...),...),
       attribute_decl(def(?name,...),...),
       function_decl(def(?name,...),...),
       procedure_decl(def(?name,...),...),
       constructor_decl(def(?name,...),...),
       value_renaming(def(?name,...),...),
       formal(def(?name,...),...);

-- this pattern is needlessly non-deterministic, but in
-- a "case" statement, this won't cause problems:
pattern some_named_decl(name : Symbol) : Declaration
    := some_named_signature_decl(?name), some_named_type_decl(?name),
       some_named_pattern_decl(?name), some_named_value_decl(?name);

match ?decl=some_named_signature_decl(?name) begin
```

```
  if name /= underscore_symbol then
    decl.decl_scope.signature_decls :> {decl};
  endif;
end;

match ?decl=some_named_type_decl(?name) begin
  if name /= underscore_symbol then
    decl.decl_scope.type_decls :> {decl};
  endif;
end;

match ?decl=some_named_pattern_decl(?name) begin
  if name /= underscore_symbol then
    decl.decl_scope.pattern_decls :> {decl};
  endif;
end;

match ?decl=some_named_value_decl(?name) begin
  if name /= underscore_symbol then
    decl.decl_scope.value_decls :> {decl};
  endif;
end;


-- The root scopes are all the file scopes.
var collection root_scopes : Scopes;

var function find_scope(name : Symbol) : Scope begin
  collection found : Scopes;
  for root_scopes begin
    match {...,?c=root_contour(!name),...} begin
      found :> {c};
    end;
  end;
  case found begin
    match {?one} begin
      result := one;
    end;
  else
    result := nil; -- multiply or not declared
  end;
end;

-- find the basic module and put it on the list to search.
var basic_scope : Scope := find_scope(make_symbol("basic"));

-- signature and module declarations have a scope for looking for things:
-- (using the $ syntax).  Similarly with polymorphic.
-- (Inheriutance is a little different)
attribute Declaration.saved_contour : Contour;
```

```
pragma source_transfer(saved_contour);
pattern with_saved_contour() : Declaration :=
    module_decl(...),signature_decl(...),polymorphic(...),inheritance(...);

-- The following rules all concern propagating the scope attribute
-- through the tree.
match program(?name,?units) begin
  new_scope : Scope := root_contour(make_symbol(name));
  root_scopes :> {new_scope};

  if basic_scope /= nil then
    new_scope.also_search :> {basic_scope};
  endif;

  for u in units begin
    u.scope := new_scope;
  end;
end;

match ?u=with_unit(?name) begin
  wscope : Scope := find_scope(make_symbol(name));
  if wscope /= nil then
    u.scope.also_search :> {wscope};
  endif;
end;

match ?u=decl_unit(?d) begin
  d.scope := u.scope;
end;

match ?s=class_decl(type_formals:=?tfs,result_type:=?result,
                    parent:=?parent,contents:=?contents)
begin
  new_scope : Scope := nested_contour(s.scope);
  for tf in tfs begin
    tf.scope := new_scope;
  end;
  result.scope := new_scope;
  parent.scope := new_scope;
  contents.scope := new_scope;
  -- the body
  body_contour : Contour := nested_contour(new_scope);
  case contents begin
    match block(?decls) begin
      for decl in decls begin
        decl.scope := body_contour;
      end;
    end;
  end;
  -- we save the scope for later searches:
```

```
  s.saved_contour := body_contour;
end;

match ?m=module_decl(type_formals:=?tfs,value_formals:=?vfs,
                     result_type:=?result,
                     parent:=?parent,contents:=?contents)
begin
  new_scope : Scope := nested_contour(m.scope);
  for tf in tfs begin
    tf.scope := new_scope;
  end;
  for vf in vfs begin
    vf.scope := new_scope;
  end;
  parent.scope := new_scope;
  result.scope := new_scope;
  contents.scope := new_scope;
  -- the body
  body_contour : Contour := nested_contour(new_scope);
  case contents begin
    match block(?decls) begin
      for decl in decls begin
        decl.scope := body_contour;
      end;
    end;
  end;
  -- we save the scope for later searches:
  m.saved_contour := body_contour;
end;

pattern some_function_decl(ty : Type; body : Block) : Declaration
    := function_decl(?,?ty,?body),procedure_decl(?,?ty,?body);
match ?d=some_function_decl(function_type(?formals,?results),?body) begin
  new_scope : Scope := nested_contour(d.scope);
  for formal in formals begin
    formal.scope := new_scope;
  end;
  for rd in results begin
    rd.scope := new_scope;
  end;
  body.scope := new_scope;
end;
match ?d=attribute_decl(?,function_type({?formal},{?rd}),default:=?def)
begin
  new_scope : Scope := nested_contour(d.scope);
  formal.scope := new_scope;
  rd.scope := new_scope;
  def.scope := new_scope;
end;
```

```
-- pattern definitions are strange because the parameters are
-- not in the scope of the body:
match ?d=pattern_decl(choices:=choice_pattern(?choices))
begin
  -- each choice has its own scope:
  for choice : Pattern in choices begin
    choice.scope := nested_contour(d.scope);
  end;
end;

match ?p=polymorphic(?,?tfs,?contents) begin
  p.decl_scope.poly_decls :> {p};
  -- otherwise, similar to module_decl
  type_scope : Scope := nested_contour(p.scope);
  for tf in tfs begin
    tf.scope := type_scope;
  end;
  contents.scope := type_scope;
  -- the body
  body_contour : Contour := nested_contour(type_scope);
  case contents begin
    match block(?decls) begin
      for decl in decls begin
        decl.scope := body_contour;
      end;
    end;
  end;
  p.saved_contour := body_contour;
end;

match ?i=inheritance(?,?ty,block(?decls)) begin
  -- The type is resolved in the outer block:
  ty.scope := i.scope;
  -- The declarations are bound in the scope of the module
  -- (it must be a module) being inherited.  The module must
  -- also be lexically in scope (not through an extension).
  -- This property allows us to do the lookup before the
  -- rest of things are looked up:
  inherited_scope : Scope;
  case ty begin
    match type_inst(module_use(use(?name)),...) begin
      case lookup_module(name,i.scope) begin
        match ?m=module_decl(...) begin
          inherited_scope := m.saved_contour;
        end;
      else
        inherited_scope := i.scope;
      end;
    end;
  else
```

```
    inherited_scope := i.scope;
  end;
  i.saved_contour := nested_contour(inherited_scope);
  for decl in decls begin
    [phylum T :: SCOPABLE,{Signature,Type,Expression,Pattern}]
    begin
      case decl begin
        match replacement(?old:T,?new:T) begin
          old.scope := i.saved_contour; -- used to be inherited_scope
          new.scope := i.decl_scope;
        end;
      end;
    end;
    -- defaults:
    decl.decl_scope := i.decl_scope;
    decl.scope := i.saved_contour;
  end;
end;

-- a lookup function that does the lookup for an inherit and
-- avoids looking at extensions:
var function lookup_module(name : Symbol; scope : Scope) : remote Declaration
begin
  case scope.type_decls begin
    match {...,?decl=some_named_decl(!name),...} begin
      case decl begin
        match module_decl(...) begin
          result := decl;
        end;
      else
        result := nil; -- can only inherit from modules
      end;
    end;
  else
    case scope begin
      match nested_contour(?parent_scope) begin
        result := lookup_module(name,parent_scope);
      end;
    else
      result := nil;
    end;
  end;
end;

match ?m=matcher(?pat,?b) begin
  new_scope : Scope := nested_contour(m.scope);
  pat.scope := new_scope;
  b.scope := new_scope;
end;
```

```
-- block gets new scope:
match ?b=block(?decls) begin
  new_scope : Scope := nested_contour(b.scope);
  for decl in decls begin
    decl.scope := new_scope;
  end;
end;

-- for loops get new blocks
match ?f=for_in_stmt(?formal,?seq,?body) begin
  new_scope : Scope := nested_contour(f.scope);
  formal.scope := new_scope;
  body.scope := new_scope;
end;

-- function types get a new scope:
match ?ft=function_type(?formals,?rds) begin
  new_scope : Scope := nested_contour(ft.scope);
  for formal in formals begin
    formal.scope := new_scope;
  end;
  for rd in rds begin
    rd.scope := new_scope;
  end;
end;

-- bodies of repeat patterns get a new scope:
match ?p=rest_pattern(?body) begin
  new_scope : Scope := nested_contour(p.scope);
  body.scope := new_scope;
end;

match ?c=controlled(?expr,?formal,?) begin
  new_scope : Scope := nested_contour(c.scope);
  formal.scope := new_scope;
  expr.scope := new_scope;
end;

-- copy scope to predefined_use
match ?td=type_decl_with_predefined_use(...) begin
  td.predefined_use.scope := td.scope;
end;

-- now a catch-all cases: copy scope to child:
[phylum T,U::SCOPABLE] match ?p:T=parent(?c:U) begin
  c.scope := p.scope;
end;

-- a little sloppy: this should be defined in aps-tree:
signature APS_SEQUENCE :=
```

```
      {Units,Declarations,Matches,Types,Expressions,
       Patterns,Actuals,TypeActuals,PatternActuals}, var PHYLUM[];

  [phylum T,U::SCOPABLE;
   L :: APS_SEQUENCE,SEQUENCE[U]] begin
    match ?p:T=parent(L${...,?c:U,...}) begin
      c.scope := p.scope;
    end;
  end;
end;
```

C.2.2 Environments

```
module APS_ENVIRON[Input :: var ABSTRACT_APS[]]
    extends Input
begin
  --- The (Type) Environment

  type Environment;

  -- the decl for a rib is always a module, signature or polymorphic
  -- declaration:
  pattern rib_decl() : Declaration :=
      module_decl(...), class_decl(...), polymorphic(...), function_decl(...);

  constructor bound_rib(decl : remote Declaration;
                        input_cap, var_cap : Boolean;
                        mapping : EnvironmentMapping;
                        next : Environment) : Environment;
  constructor unbound_rib(decl : remote Declaration;
                          next : Environment) : Environment;
  constructor root_env() : Environment;

  pattern some_rib(decl : remote Declaration;
                   next : Environment) : Environment
      := bound_rib(?decl,next:=?next),unbound_rib(?decl,?next);

  pattern some_unbound() : Environment
      := unbound_rib(...), root_env();

  empty_env : Environment := root_env();

  type ContextualTypes := LIST[ContextualType];

  type EnvironmentMapping :=
      MAP[remote Declaration, ContextualType](return_no_contextual_type);
  function return_no_contextual_type(_ : remote Declaration) : ContextualType
      := no_contextual_type;
  no_contextual_type : ContextualType := no_contextual();
  environment_mapping = EnvironmentMapping$map;
```

```
pattern environment_mapping = EnvironmentMapping$map;
empty_mapping : EnvironmentMapping := environment_mapping({});

-- a useful type:
-- (see aps-signature.aps)
type Environments := BAG[Environment];


--- The environment attribute:

-- the places where an environment is useful
-- (all nodes excluding Program, Unit, Def, Direction)
signature WITH_ENVIRONMENT :=
    {Signature, Type, Expression, Pattern, Use, Block, Declaration,
     Match, Default}, var PHYLUM[];
signature WITH_ENVIRONMENT_SEQ :=
    {Declarations,Matches,Types,Expressions,Patterns,
     TypeActuals,Actuals,PatternActuals}, var PHYLUM[];

[phylum T :: WITH_ENVIRONMENT]
    attribute T.environment : Environment := empty_env;

-- special case: the environment for a function decl's type is the
-- outside environment (the information is available outside,
-- and it confuses later steps if there is a superfluous unbound_rib
-- about the type):
match ?d=function_decl((type):=?ty) begin
  ty.environment := d.environment;
  -- but the return decl is put inside the scope:
  case ty begin
    match function_type(?,?rds) begin
      for rd in rds begin
        rd.environment := unbound_rib(d,d.environment);
      end;
    end;
  end;
end;

-- module, signature and polymorphic decls
-- add ribs to the environment
match ?d=rib_decl() begin
  new_env : Environment := unbound_rib(d,d.environment);
  [phylum CH :: WITH_ENVIRONMENT] begin
    for d begin
      match parent(?ch:CH) begin
        ch.environment := new_env;
      end;
    end;
  end;
  [phylum CH :: WITH_ENVIRONMENT;
```

```
   L :: SEQUENCE[CH],WITH_ENVIRONMENT_SEQ] begin
    for d begin
      match parent({...,?ch:CH,...}:L) begin
        ch.environment := new_env;
      end;
    end;
  end;
end;

-- the environment is spread to the uses hanging on:
match ?td=type_decl_with_predefined_use(...) begin
  td.predefined_use.environment := td.environment;
end;

-- otherwise, just copy from parent to child:
[phylum P,CH :: WITH_ENVIRONMENT] begin
  match ?p:P=parent(?ch:CH) begin
    ch.environment := p.environment;
  end;
end;
[phylum P,CH :: WITH_ENVIRONMENT;
 L :: SEQUENCE[CH],WITH_ENVIRONMENT_SEQ] begin
  match ?p:P=parent({...,?ch:CH,...}:L) begin
    ch.environment := p.environment;
  end;
end;


--- Combining environments

function merge_environ(e1,e2 : Environment) : Environment begin
  case e1 begin
    -- only bound environments change the environment being merged:
    match bound_rib(?decl1,?i1,?v1,?,?next1) begin
      case e2 begin
        match bound_rib(?decl2,?i2,?v2,?mapping2,?next2) begin
          result :=
              bound_rib(decl2, i1 and i2, v1 and v2,
                        merge_environment_mapping(e1,mapping2),
                        merge_environ(e1,next2));
        end;
        match unbound_rib(!decl1,?) begin
          result := e1;
        end;
        match unbound_rib(?,?) begin
          result := merge_environ(next1,e2);
        end;
      else
        -- e2 is root_env(), nothing can happen to it.
        result := e2;
```

```
      end;
    end; -- match
  else
    -- otherwise e1 has no effect:
    result := e2;
  end; -- case e1
end;

function merge_environment_mapping(e : Environment;
                                   m : EnvironmentMapping)
    : EnvironmentMapping
begin
  case m begin
    match environment_mapping(?pairs) begin
      result := environment_mapping({merge_pair(p) for p in pairs});
    end;
  end;
  function merge_pair(p : EnvironmentMapping$PairType)
      : EnvironmentMapping$PairType
  begin
    case p begin
      match (=>)(?from,?to) begin
        result := from=>merge_contextual(e,to);
      end;
    end;
  end;
end;

pragma no_memo(merge_environment_mapping);

function apply_environ(e : Environment;
                       decl : remote Declaration) : ContextualType
begin
  -- trying looking up the decl in the environment
  case e begin
    match bound_rib(mapping:=?mapping) begin
      result := EnvironmentMapping$apply(mapping,decl);
    end;
  else
    result := no_contextual_type;
  end;

end;


--- Contextualized things
-- one for each phylum that can be contextualized (and found!)

module CONTEXTUAL[BaseType :: BASIC[]](default : BaseType) :: BASIC[] begin
  constructor contextual(environ : Environment;
```

```
                        base : BaseType) : Result;
  constructor no_contextual() : Result;
  constructor bad_contextual(message : String) : Result;
  default_base = default;

  -- functions
  function contextual_base(x : Result) : BaseType begin
    case x begin
      match contextual(?,?base) begin
        result := base;
      end;
    else
      result := default_base;
    end;
  end;
  function contextual_environ(x : Result) : Environment begin
    case x begin
      match contextual(?environ,?) begin
        result := environ;
      end;
    else
      result := empty_env;
    end;
  end;
end;
[BaseType; T :: CONTEXTUAL[BaseType]] begin
  contextual = T$contextual;
  no_contextual = T$no_contextual;
  bad_contextual = T$bad_contextual;
  pattern contextual = T$contextual;
  pattern no_contextual = T$no_contextual;
  pattern bad_contextual = T$bad_contextual;
  contextual_base = T$contextual_base;
  contextual_environ = T$contextual_environ;
end;

[BaseType; T :: CONTEXTUAL[BaseType]] begin
  function merge_contextual(env : Environment; x : T) : T begin
    case x begin
      match contextual(?sub,?base) begin
        result := contextual(merge_environ(env,sub),base);
      end;
    else
      result := x;
    end;
  end;
end;

-- contextual declarations must always have as invariant
--   outerrib.decl = decl.parent or
```

```
  --   env = empty_env() and at_top_level(decl)
  type ContextualDeclaration := CONTEXTUAL[remote Declaration](nil);

  type ContextualType := CONTEXTUAL[remote Type](nil);

  type ContextualSignature := CONTEXTUAL[remote Signature](nil);

  [BaseType :: WITH_ENVIRONMENT; T :: CONTEXTUAL[BaseType]] begin
    var function contextualize(base : BaseType) : T begin
      if base == nil then
        result := no_contextual();
      else
        result := contextual(base.environment,base);
      endif;
    end;
  end;

  function add_result_to_environment(ct : ContextualType;
                                     env : Environment) : Environment
  begin
    case env begin
      match bound_rib(?cd=some_class_decl(result_type:=?rd),?i,?v,
                      environment_mapping(?pairs),?next)
      begin
        result :=
            bound_rib(cd,i,v,environment_mapping({rd=>ct,pairs...}),next);
      end;
    end;
  end;
end;

-- used primarily as a signature,
-- but used as a module for modules that do transformations
module APS_BOUND[Input :: var ABSTRACT_APS[], var APS_ENVIRON[Input]]
    extends Input
begin
  input attribute Use.contextual_def : ContextualDeclaration :=
      no_contextual();
  input attribute Declaration.depends_on_self : Boolean := false;
end;
```

C.2.3 Name Resolution

Name resolution needs to use the signatures for each type in the problem used to fetch services (including implicit fetching through the extension of a module). But signature determination requires that name resolution be carried out. Thus signature determination (which requires type determination) is inseparable with name resolution. Furthermore, to prevent name resolution from getting into infinite loops, we need to detect types defined in terms of themselves. For example, name resolution of a service fetched from a type such as

`Bad` should not cause the compiler to go into infinite recursion:

```
type Bad = Bad;
```

Thus name resolution requires that cycles be detected in declarations which likewise name resolution. Again the cycle detection module is inseparable with name resolution.

```
-- This module resolves name references.
-- It also defines an attribute "resolve_error" that
-- contains a string (nil if no error).

-- Note, in order to give a noncircular semantics to APS,
-- it is necessary to perform all regular lookup before any $ qualified
-- lookup and all $ qualified lookup must be doable in parallel.
-- Therefore, scopes (used for qualified lookup) are never inferred
-- for qualified types, modules and signatures.
module APS_RESOLVE[Input :: var ABSTRACT_APS[], var APS_PATTERN[Input],
                            var APS_SYMTAB[Input],
                            var APS_ENVIRON[Input],
                            var APS_PREDEFINED[Input]]
    extends Input
begin
  attribute Use.resolve_error : String;
  -- each use gets a preliminary definition,
  -- to be fixed up by APS_TYPECHECK
  attribute Use.pre_contextual_def : ContextualDeclaration;

  -- possible error messages for module extensions:
  attribute Declaration.extension_error : String := "";

  -- we look up each formal of a pattern_decl in each choice
  -- (But do not generate error messages if not found)
  type DeclList := LIST[remote Declaration];
  attribute Pattern.pattern_formals : DeclList := {};

  private;


  attribute Contour.inherited_environment : Environment := empty_env;

  -- We treat the inheritance declaration as a type declaration to hold the
  -- result of the inheritance.  Since it will be possible to refer to this
  -- declaration through renamings, it must behave like any other type
  -- declaration too.  (See other modules.)
  -- We make a special bound_rib that records type replacements
  -- too.
  match ?i=inheritance(?,?ty=type_inst(...),block(?decls)) begin
    case type_inst_env(ty) begin
      match bound_rib(?md,mapping:=environment_mapping(?previous),next:=?next)
      begin
        collection replacement_pairs : EnvironmentMapping$BaseType;
```

```
      for decl in decls begin
        case decl begin
          match type_replacement(?from=type_use(use(?name)),?as) begin
            case lookup_locally(name,type_namespace(),md.saved_contour,true)
            begin
              match {contextual(some_unbound(...),?decl)} begin
                replacement_pairs :> {decl=>contextualize(as)};
              end;
            end;
          end;
        end;
      end;
      i.saved_contour.inherited_environment :=
          bound_rib(md,true,true,
                    environment_mapping(previous\/replacement_pairs),next);
    end;
  end;
end;


--- Name Resolution:

-- Name resolution happens in two passes, one for unqualified
-- lookup and one for qualified lookup.

attribute Use.unqualified_contextual_def : ContextualDeclaration;

not_found : ContextualDeclaration := no_contextual();
duplicate_find : ContextualDeclaration := bad_contextual("ambiguous");
not_input : ContextualDeclaration := bad_contextual("type fixed");
not_var : ContextualDeclaration := bad_contextual("type unfinished");

attribute Use.namespace : Namespace;
type Namespace;
constructor signature_namespace() : Namespace;
constructor type_namespace() : Namespace;
constructor pattern_namespace() : Namespace;
constructor value_namespace() : Namespace;

match class_use(?use) begin
  use.namespace := signature_namespace();
end;
match sig_use(?use) begin
  use.namespace := signature_namespace();
end;
match module_use(?use) begin
  use.namespace := type_namespace();
end;
match type_use(?use) begin
  use.namespace := type_namespace();
```

```
end;
match value_use(?use) begin
  use.namespace := value_namespace();
end;
match pattern_use(?use) begin
  use.namespace := pattern_namespace();
end;

-- for efficiency, the lookup for the predefined uses are done by hand:
match ?td=type_decl_with_predefined_use(...) begin
  case td.predefined_use begin
    match type_use(?u) begin
      u.unqualified_contextual_def := contextual(td.environment,td);
      u.pre_contextual_def := contextual(td.environment,td);
    end;
  end;
end;

match ?use=qual_use(?inst,?name) begin
  -- force the default here:
  use.unqualified_contextual_def := not_found;
  if inst.scopes = {} then
    use.pre_contextual_def := not_found;
    use.resolve_error := "Type has no signature";
  else
    find : ContextualDeclaration :=
        lookup_in_environments(name,use.namespace,inst.scopes);
    use.pre_contextual_def := find;
  endif;
end;

-- don't bother trying to lookup pattern formal names:
match pattern_actual(?,value_use(?u)) begin
  u.unqualified_contextual_def := not_found;
  u.pre_contextual_def := not_found;
  u.resolve_error := "";
end;

-- don't bother looking up '_'
match ?use=use(!underscore_symbol) begin
  use.pre_contextual_def := not_found;
  use.unqualified_contextual_def := not_found;
  use.resolve_error := "";
end;

-- normal lookup
match ?use=use(?name) begin
  find : ContextualDeclaration := lookup_in_scope(name,use.namespace,
                                                  use.scope,true);
  use.pre_contextual_def := find;
```

```
  use.unqualified_contextual_def := find;
end;

match ?u:Use begin
  case u.pre_contextual_def begin
    match !not_found begin
      u.resolve_error := "not found";
    end;
    match bad_contextual(?reason) begin
      u.resolve_error := reason;
    end;
  else
    u.resolve_error := "";
  end;
end;


--- Lookup

type ContextualDeclarations := BAG[ContextualDeclaration];

-- take a name and try to find it in the scope, return
-- a contextualized entity.
-- (There's an added complication for merging in the
-- inherited environment, if necessary)

var function lookup_in_scope(name : Symbol;
                             namespace : Namespace;
                             s : Scope;
                             internal : Boolean)
    final_result : ContextualDeclaration
begin
  result : ContextualDeclaration; -- as computed in the rest of function
  if s.inherited_environment /= empty_env then
    final_result := merge_contextual(s.inherited_environment,result);
  else
    final_result := result;
  endif;

  collection finds : ContextualDeclarations;

  -- look among all local declarations
  finds :> lookup_locally(name,namespace,s,internal);
  -- look inside local polys
  if namespace = value_namespace() or
      namespace = pattern_namespace() then
    finds :> lookup_in_polys(name,namespace,s,internal);
  endif;
  case finds begin
    match {} begin
```

```
    if internal then
      extended : ContextualDeclaration
          := lookup_in_environments(name,namespace,s.extension);
      if extended = not_found then
        case s begin
          match nested_contour(?parent_scope) begin
            result :=
                lookup_in_scope(name,namespace,parent_scope,internal);
          end;
          match root_contour(...) begin
            -- try other files
            collection finds : ContextualDeclarations;
            for root_scope in s.also_search begin
              if s /= root_scope then
                case lookup_in_scope(name,namespace,root_scope,false)
                begin
                  match !not_found begin end;
                  match ?thing begin
                    finds :> {thing};
                  end;
                end;
              endif;
            end;
            case finds begin
              match {} begin
                result := not_found;
              end;
              match {?find} begin
                result := find;
              end;
            else
              result := duplicate_find;
            end;
          end;
        else -- maybe nil ?
          result := not_found;
        end;
      else
        result := extended;
      endif;
    else
      result := not_found;
    endif;
  end;
  match {?find} begin
    result := find;
  end;
else
  result := duplicate_find;
end;
```

```
end;

function lookup_locally(name : Symbol;
                        namespace : Namespace;
                        s : Scope;
                        internal : Boolean)
    collection : ContextualDeclarations
begin
  -- create the barebones environment for the found things
  for scope_decls(s,namespace) begin
    match {...,?decl=declaration(def(!name,is_public:=?is_public)),...}
        if is_public or internal
    begin
      result :> {contextual(decl.environment,decl)};
    end;
  end;
end;
pragma memo(lookup_locally);

function lookup_in_polys(name : Symbol;
                         namespace : Namespace;
                         s : Scope;
                         internal : Boolean)
    collection : ContextualDeclarations
begin
  -- look among all poly's if a value or type
  -- and create an empty bound_rib environment.
  -- (aps-typecheck will fill in the details).
  for poly in s.poly_decls begin
    env : Environment := bound_rib(poly,true,true,empty_mapping,
                                   poly.environment);
    ps : Scope := poly.saved_contour;
    result :> {merge_contextual(env,cd) for cd in
                   lookup_locally(name,namespace,ps,internal),
               merge_contextual(env,cd) for cd in
                   lookup_in_polys(name,namespace,ps,internal)};
  end;
end;


type ContextualDeclarationList := LIST[ContextualDeclaration];

var function lookup_in_environments(name : Symbol;
                                    namespace : Namespace;
                                    envs : Environments)
    : ContextualDeclaration
begin
  case ContextualDeclarationList$
      {lookup_in_environment(name,namespace,env) for env in envs}
  begin
```

```
    match {... and !not_found,?find if find /= not_found,...} begin
      result := find;
    end;
  else
    result := not_found;
  end;
end;

var function lookup_in_environment(name : Symbol;
                                   namespace : Namespace;
                                   env : Environment)
    : ContextualDeclaration
begin
  case env begin
    match some_rib(?decl,?) begin
      scope : Scope := decl.saved_contour;
      result :=
          merge_contextual(env,lookup_in_scope(name,namespace,scope,false));
    end;
  end;
end;

function scope_decls(s : Scope; namespace : Namespace) : Decls
begin
  case namespace begin
    match signature_namespace() begin
      result := s.signature_decls;
    end;
    match type_namespace() begin
      result := s.type_decls;
    end;
    match value_namespace() begin
      result := s.value_decls;
    end;
    match pattern_namespace() begin
      result := s.pattern_decls;
    end;
  end;
end;


--- Pattern choices

match pattern_decl(?,function_type(?formals,?),choice_pattern(?pats))
begin
  for pat in pats begin
    pat.pattern_formals :=
        {find_pattern_formal(formal,pat.scope.value_decls)
             for formal in formals};
  end;
```

```
end;

function find_pattern_formal(formal : remote Declaration;
                             decls : Decls) : remote Declaration begin
  collection results : Decls;
  case formal begin
    match formal(def(?name,...),...) begin
      for decls begin
        match {...,?f=formal(def(!name,...),...),...} begin
          results :> {f};
        end;
      end;
    end;
  end;
  case results begin
    match {} begin
      result := nil;
    end;
    match {?f} begin
      result := f;
    end;
  else
    result := formal; -- NB: multiply defined
  end;
end;


--- Scopes:

-- See #NOTE 6 in research5.ram for justification:
-- We don't infer signatures for $ things and
-- don't allow $ signatures.  This major change forbids things
-- like T$U$V and makes nested modules almost useless outside
-- of their enclosing module even when they are exported.

-- We have to use inheritance because these modules are non-separable
inherit APS_DETECT_CYCLE[Input](unqualified_contextual_def) begin
  var depends_on_self = depends_on_self;
end;
inherit APS_TYPE_INFO[Input]
    (unqualified_contextual_def,depends_on_self)
begin
  var base_module = base_module;
  var base_class = base_class;
  var type_inst_env = type_inst_env;
  var type_base_known_p = type_base_known_p;
  var contextual_type_equalp = contextual_type_equalp;
  var environment_equalp = environment_equalp;
end;
inherit APS_SIG_INFO[Input]
```

```
      (unqualified_contextual_def,depends_on_self,
       base_class,base_module,
       type_inst_env,type_base_known_p,
       contextual_type_equalp,environment_equalp)
  begin
    signature SCOPED = SIGNATURED;
    [phylum T :: SIGNATURED]
        var scopes = (sig_info : function(_:T):Environments);
  end;


  --- Extension:

  attribute Contour.extension : Environments := {};

  match ?m=module_decl(result_type:=?result_decl) begin
    extension_scopes : Environments := {};
    m.saved_contour.extension := extension_scopes;
    case result_decl.scopes begin
      match {...,bound_rib(!m,...),...}
      begin
        m.extension_error := "Circular extension";
        extension_scopes := {};
      end;
    else
      extension_scopes := result_decl.scopes;
    end;
  end;

  pragma no_memo(lookup_in_environment,lookup_in_polys,lookup_locally);
end;
```

C.2.4 Cycle Detection

```
-- This module determines cycles in contextual_def's
module APS_DETECT_CYCLE[Input :: var ABSTRACT_APS[], APS_PATTERN[Input],
                                 var APS_ENVIRON[Input]]
    (contextual_def :function(_:remote Input$Use):Input$ContextualDeclaration)
    extends Input
begin
  -- whether a type or signature depends on itself:
  attribute Declaration.depends_on_self : Boolean := false;

  private;

  --- Detecting type and signature circularity

  -- We have circular sets that keep track of the transitive closure of
  -- requirements and then see if a declaration precedes itself
  -- in the closure:
```

```
type Decls := ORDERED_SET[remote Declaration]((==),(<<));
type DeclLattice := UNION_LATTICE[remote Declaration,Decls];

signature REQUIRES_SIG :=
    {Declaration,Class,Module,Signature,Type,Use}, var PHYLUM[];

[phylum T :: REQUIRES_SIG]
    circular collection attribute T.requires : DeclLattice;

match ?d:Declaration begin
  d.depends_on_self := d in d.requires;
end;

-- a module is circular if one of its type/module declarations
--    or inheritances requires the module to get a value.
--    or its parent signature requires itself.

match ?d=module_decl(result_type:=?rd,parent:=?parent,contents:=?b)
begin
  d.requires :> parent.requires |\/| rd.requires;
  for b begin
    match block({...,?td=some_type_decl(...),...}) begin
      d.requires :> td.requires;
    end;
    match ancestor(inheritance(?,?used,?)) begin
      d.requires :> used.requires;
    end;
  end;
end;

-- a class is circular if its parent/value requires itself.
match ?d=class_decl(parent:=?parent) begin
  d.requires :> parent.requires;
end;

match ?d=signature_decl(?,?sig) begin
  d.requires :> sig.requires;
end;

-- a type is circular if its value requires itself
match ?d=some_type_decl((type):=?ty) begin
  d.requires :> ty.requires;
end;

-- pattern definitions:
match ?d=pattern_decl(?,?,pattern_scope(pattern_call(some_pattern_use(?u),
                                                     ...)))
begin
  case u.contextual_def begin
```

```
    -- simple recursion permitted
    match contextual(?,!d) begin end;
  else
    -- mutual recursion not permitted
    d.requires :> u.requires;
  end;
end;

-- renamings are similar
match ?d=class_renaming(?,?cl) begin
  d.requires :> cl.requires;
end;
match ?d=module_renaming(?,?mod) begin
  d.requires :> mod.requires;
end;
match ?d=signature_renaming(?,?sig) begin
  d.requires :> sig.requires;
end;
match ?d=type_renaming(?,?ty) begin
  d.requires :> ty.requires;
end;
match ?d=value_renaming(?,some_value_use(?u:Use)) begin
  d.requires :> u.requires;
end;
match ?d=pattern_renaming(?,some_pattern_use(?u:Use)) begin
  d.requires :> u.requires;
end;

match ?u:Use begin
  case contextual_def(u) begin
    match contextual(?,?decl) begin
      u.requires :> decl.requires;
      u.requires :> {decl};
    end;
  end;
end;

match ?cl=class_use(?u) begin
  cl.requires :> u.requires;
end;
match ?s=sig_use(?u) begin
  s.requires :> u.requires;
end;
match ?s=sig_inst(?,?,?cl,?) begin
  s.requires :> cl.requires;
end;
match ?s1=mult_sig(?s2,?s3) begin
  s1.requires :> s2.requires |\/| s3.requires;
end;
```

```
  match ?m=module_use(?u) begin
    m.requires :> u.requires;
  end;
  match ?t=type_use(?u) begin
    t.requires :> u.requires;
  end;
  match ?t=type_inst(?mod,?tactuals,?) begin
    t.requires :> mod.requires;
    t.requires :> {ty.requires... for ty in tactuals};
  end;
  match ?t1=remote_type(?t2) begin
    t1.requires :> t2.requires;
  end;
  match ?t1=private_type(?t2) begin
    t1.requires :> t2.requires;
  end;
end;
```

C.2.5 Type Information

```
module APS_TYPE_INFO[Input :: var ABSTRACT_APS[], APS_PATTERN[Input],
                              var APS_ENVIRON[Input]
                              -- var APS_DETECT_CYCLE[Input]
                     ]
    (contextual_def : function(_:remote Input$Use):Input$ContextualDeclaration;
     depends_on_self :function(_:remote Input$Declaration):Boolean)
    extends Input
begin
  --- This module exports useful functions about types

  no_base : ContextualDeclaration := no_contextual();
  no_base_type : ContextualType := no_contextual();

  -- the base class for a class to be instantiated
  -- and the base module for a type that needs to be instantiated
  attribute Class.base_class : ContextualDeclaration := no_base;
  attribute Module.base_module : ContextualDeclaration := no_base;


  --- base_class

  match ?cl=class_use(?u) begin
    case contextual_def(u) begin
      match contextual(?e,?r=class_renaming(?,?cl2)) begin
        if not r.depends_on_self then
          cl.base_class := cl2.base_class;
        else
          cl.base_class := no_base;
        endif;
      end;
```

```
    match ?cd=contextual(?e,?d=some_class_decl(...)) begin
      cl.base_class := cd;
    end;
  end;
end;

--- base_module

match ?mod=module_use(?u) begin
  case contextual_def(u) begin
    match contextual(?e,?d=module_renaming(?,?mod2)) begin
      if d.depends_on_self then
        mod.base_module := no_base;
      else
        mod.base_module := mod2.base_module;
      endif;
    end;
    match ?cd=contextual(?,module_decl(...)) begin
      mod.base_module := cd;
    end;
  end;
end;


--- Type information

var function type_base_known_p(ty : remote Type) : Boolean begin
  -- that is, we know how the type was created:
  -- by a module invocation without an extension, or as a blank type:
  case make_contextual_base_type(ty.environment,ty) begin
    match contextual(?,type_inst(...)) begin
      result := true;
    end;
    match contextual(?,type_use(?u)) begin
      case u.contextual_def begin
        match contextual(?,some_type_decl(?,?,no_type())) begin
          result := true;
        end;
      end;
    end;
  end;
  result := false;
end;

--- Next compute a useful bit of information for type instances:
-- the bound rib for a module instantiation:

var function type_inst_env(ty : remote Type) : Environment begin
  case ty begin
    match type_inst(?m,?tactuals,?) begin
```

```
      case m.base_module begin
        match contextual(?environ,
                         ?md=module_decl(type_formals:=?tfs,
                                         parent:=?parent,
                                         result_type:=?rd))
        begin
          if length(tfs) = length(tactuals) then
            result :=
                bound_rib
                (md,true,true,
                 environment_mapping
                     ({rd=>contextualize(ty),
                       nth(i,tfs)=>contextualize(nth(i,tactuals))
                           for i in 0..(length(tactuals)-1)}),
                 environ);
          endif;
        end;
      end;
    end;
  end;
  result := empty_env;
end;


--- Functions for comparing environments and types

-- returns true when two environments are the same
-- (ignoring capabilities).
var function environment_equalp(e1,e2 : Environment) : Boolean begin
  case e1 begin
    match root_env() begin
      result := e1 = e2;
    end;
    match unbound_rib(?decl1,?) begin
      case e2 begin
        match unbound_rib(?decl2,?) begin
          result := decl1 == decl2;
        end;
      end;
    end;
    -- capabilities irrelevant for bound ribs:
    -- polymorphic things are structurally typed:
    match bound_rib(?p=polymorphic(type_formals:=?tfs),next:=?n1) begin
      case e2 begin
        match bound_rib(!p,next:=?n2) begin
          result := true and
              (contextual_type_equalp(apply_environ(e1,tf),
                                      apply_environ(e2,tf))
                   for tf in tfs) and
              environment_equalp(n1,n2);
```

```
        end;
      end;
    end;
    -- for other things, we require instance equivalence:
    match bound_rib(?cd=some_class_decl(result_type:=?rd),...) begin
      case e2 begin
        match bound_rib(!cd,...) begin
          result := contextual_eq(apply_environ(e1,rd),apply_environ(e2,rd));
        end;
      end;
    end;
  end;
  result := false; -- default
end;

[T :: BASIC[]; U :: CONTEXTUAL[T]] begin
  var function contextual_eq(c1,c2 : U) : Boolean begin
    case c1 begin
      match contextual(?e1,?t) begin
        case c2 begin
          match contextual(?e2,!t) begin
            result := environment_equalp(e1,e2);
          end;
        end;
      end;
    end;
    result := false;
  end;
end;

-- return true when two types have the same base type:
var function contextual_type_equalp(ct1,ct2 : ContextualType) : Boolean
    := base_type_equalp(contextual_type_base_contextual_type(ct1),
                        contextual_type_base_contextual_type(ct2));

var function base_type_equalp(ct1,ct2 : ContextualType) : Boolean begin
  case ct1 begin
    -- first test the structurally equivalent types:
    -- void type:
    match contextual(?,no_type()) begin
      case ct2 begin
        match contextual(?,no_type()) begin
          result := true;
        end;
      end;
    end;
    -- function types:
    match contextual(?e1,function_type(?fs1,?rds1)) begin
      case ct2 begin
        match contextual(?e2,function_type(?fs2,?rds2)) begin
```

```
          result :=
              length(fs1) = length(fs2) and
              (formal_type_equalp(e1,e2,nth(i,fs1),nth(i,fs2))
                   for i in 0..(length(fs1)-1)) and
              length(rds1) = length(rds2) and
              (formal_type_equalp(e1,e2,nth(i,rds1),nth(i,rds2))
                   for i in 0..(length(rds1)-1));
        end;
      end;
    end;
  else
    -- otherwise by name:
    result := contextual_eq(ct1,ct2);
  end;
  result := false;
end;
--
var function formal_type_equalp(e1,e2 : Environment;
                                f1,f2 : remote Declaration)
    : Boolean
begin
  case f1 begin
    match formal(?,?t1) begin
      case f2 begin
        match formal(?,?t2) begin
          result := formal_same_shape_p(f1,f2) and
              contextual_type_equalp(make_contextual_type(e1,t1),
                                     make_contextual_type(e2,t2));
        end;
      end;
    end;
    match value_decl((type):=?t1) begin
      case f2 begin
        match value_decl((type):=?t2) begin
          result :=
              contextual_type_equalp(make_contextual_type(e1,t1),
                                     make_contextual_type(e2,t2));
        end;
      end;
    end;
  end;
  result := false;
end;
--
function formal_same_shape_p(f1,f2 : remote Declaration) : Boolean begin
  case f1 begin
    match seq_formal(...) begin
      case f2 begin
        match seq_formal(...) begin
          result := true;
```

```
        end;
      end;
    end;
    match normal_formal(...) begin
      case f2 begin
        match normal_formal(...) begin
          result := true;
        end;
      end;
    end;
  end;
  result := false;
end;

pragma no_memo(formal_type_equalp,formal_same_shape_p);

function make_contextual_type(e : Environment; ty : remote Type)
    : ContextualType
begin
  case ty begin
    match type_use(?u) begin
      case u.contextual_def begin
        match contextual(?sub,?decl) begin
          env : Environment := merge_environ(e,sub);
          substituted : ContextualType := apply_environ(env,decl);
          if substituted = no_contextual_type then
            result := contextual(env,decl.predefined_use);
          else
            result := simplify_contextual_type(substituted);
          endif;
        end;
      end;
    end;
  end;
  -- default:
  result := contextual(e,ty);
end;
pragma no_memo(make_contextual_type);

function simplify_contextual_type(ct : ContextualType) : ContextualType
begin
  case ct begin
    match contextual(?env,?ty) begin
      result := make_contextual_type(env,ty);
    end;
  end;
end;
pragma memo(simplify_contextual_type);

var function make_contextual_base_type(e : Environment; ty : remote Type)
```

```
    : ContextualType
begin
  case ty begin
    match type_use(?u) begin
      case u.contextual_def begin
        match contextual(?sub,?decl) begin
          env : Environment := merge_environ(e,sub);
          substituted : ContextualType := apply_environ(env,decl);
          if substituted = no_contextual_type then
            case decl begin
              match type_renaming(?,?ty) begin
                result := make_contextual_base_type(env,ty);
              end;
              -- base types are not expanded:
              match some_type_decl(?,?,no_type()) begin
                result := contextual(env,decl.predefined_use);
              end;
              match some_type_decl(?,?,?ty) begin
                result := make_contextual_base_type(env,ty);
              end;
              match inheritance(?,?ty,?) begin
                result := make_contextual_base_type(env,ty);
              end;
            else
              result := contextual(env,decl.predefined_use);
            end;
          else
            -- try another go around:
            result := contextual_type_base_contextual_type(substituted);
          endif;
        end;
      end;
    end;
    match remote_type(?ty) begin
      result := make_contextual_base_type(e,ty);
    end;
    match type_inst(?mod,?tactuals,?) begin
      case mod.base_module begin
        match contextual(?sub,?md=module_decl(type_formals:=?tfs,
                                              result_type:=?rd))
            if length(tfs) = length(tactuals)
        begin
          env : Environment := merge_environ(e,sub);
          -- if the result is a base type, we do not expand
          case rd begin
            match some_type_decl(?,?,no_type()) begin
              result := contextual(e,ty);
            end;
            match some_type_decl(?,?,?ty1) begin
              env : Environment := merge_environ(e,type_inst_env(ty));
```

```
                result := make_contextual_base_type(env,ty1);
              end;
            end;
          end;
        end;
      end;
    end;
    -- default:
    result := contextual(e,ty);
  end;
  pragma no_memo(make_contextual_base_type);

  var function contextual_type_base_contextual_type(ct : ContextualType)
      : ContextualType
  begin
    case ct begin
      match contextual(?e,?ty) begin
        result := make_contextual_base_type(e,ty);
      end;
    else
      result := no_base_type;
    end;
  end;


end;
```

C.2.6 Signature Information

```
-- This module computes information on signatures for types and signatures.
module APS_SIG_INFO[Input :: var ABSTRACT_APS[],
                             var APS_ENVIRON[Input],
                             -- var APS_DETECT_CYCLE[Input],
                             -- var APS_TYPE_INFO[Input],
                             var APS_PREDEFINED[Input]]
    -- rather than require signatures on the tupes, we
    -- just request the needed attributes (in the form of functions)
    (contextual_def :function(_:remote Input$Use):Input$ContextualDeclaration;
     depends_on_self :function(_:remote Input$Declaration):Boolean;
     base_class :function(_:remote Input$Class):Input$ContextualDeclaration;
     base_module :function(_:remote Input$Module):Input$ContextualDeclaration;
     type_inst_env :function(_:remote Input$Type):Input$Environment;
     type_base_known_p :function(_:remote Input$Type) : Boolean;
     contextual_type_equalp :function(_,_:Input$ContextualType):Boolean;
     environment_equalp :function(_,_:Input$Environment):Boolean)
    extends Input
begin
  -- the base type formal for a possible extension
  attribute Type.base_extension : remote Declaration := nil;
  -- This attribute is useful only for type_inst nodes:
```

```
-- It points to one of the actual parameters.
attribute Type.extension_actual : remote Type := nil;

no_base : ContextualDeclaration := no_contextual();

--- Information about signatures for types and signatures:

-- Historically, SigInfoType was a type formal, but now, it's
-- just a bag of Environments.
type SigInfoType = Environments;

signature SIGNATURED := {Type,Signature,Declaration}, var PHYLUM[];
[phylum T :: SIGNATURED]
    attribute T.sig_info : SigInfoType := empty_sig_info;

--- Useful values/functions for SigInfoType

-- formerly parameters, or overrideable upon inheritance

empty_sig_info : SigInfoType := {};

function make_sig_info(env : Environment; _ : remote Declaration)
    : SigInfoType := {env};

function append_sig_info(envs1,envs2 : SigInfoType)
    : SigInfoType
    := {with_possible_added_caps(env1,envs2) for env1 in envs1,
        env2 if not already_in_sig_info(env2,envs1) for env2 in envs2};
function with_possible_added_caps(env1 : Environment;
                                  envs2 : SigInfoType) : Environment
begin
  case env1 begin
    match bound_rib(?decl,?i1,?v1,?mapping,?next) begin
      case envs2 begin
        match {...,bound_rib(!decl,?i2,?v2,...),...} begin
          result := bound_rib(decl,i1 or i2, v1 or v2, mapping, next);
        end;
      else
        result := env1;
      end;
    end;
  else
    result := env1; -- capabilities irrelevant
  end;
end;
function already_in_sig_info(env1 : Environment;
                             envs2 : SigInfoType) : Boolean
begin
  case env1 begin
    match bound_rib(?decl,...) begin
```

```
      case envs2 begin
        match {...,bound_rib(!decl,...),...} begin
          result := true;
        end;
      else
        result := false;
      end;
    end;
  else
    result := true; -- throw away
  end;
end;

function merge_sig_info(env : Environment; envs : SigInfoType) : SigInfoType
    -- merge_environ will mask capabilities for us
    := {merge_environ(env,env2) for env2 in envs};

function add_result_to_sig_info(td : remote Declaration;
                                envs : Environments) : Environments
    := {add_result_to_environment(contextualize(td.predefined_use),env)
            for env in envs};


-- signature decls are almost the same as signature renamings:
pattern some_signature_decl(def : Def; sig : Signature) : Declaration :=
    signature_decl(?def,?sig), signature_renaming(?def,?sig);

match ?sd=some_signature_decl(?def,?sig) begin
  if not sd.depends_on_self then
    sd.sig_info := sig.sig_info;
  else
    sd.sig_info := empty_sig_info;
  endif;
end;

match ?s=sig_use(?u) begin
  case contextual_def(u) begin
    match contextual(?e,?sd) begin
      --?? This will mask out capabilities if the signature is
      --?? fetched from a type with only limited capabilities.
      s.sig_info := merge_sig_info(e,sd.sig_info);
    end;
  end;
end;

match ?s=sig_inst(?i,?v,?cl,?tactuals) begin
  case cl.base_class begin
    match contextual(?e,?d=some_class_decl(type_formals:=?tfs,
                                           parent:=?parent))
    begin
```

```
      if length(tfs)=length(tactuals) then
        new_env : Environment :=
            bound_rib(d,i,v,
                      environment_mapping
                          ({nth(i,tfs)=>contextualize(nth(i,tactuals))
                                for i in 0..(length(tfs)-1)}),
                      e);
        base_sig_info : SigInfoType := make_sig_info(new_env,d);
        if d.depends_on_self then
          s.sig_info := base_sig_info;
        else
          s.sig_info :=
              append_sig_info(base_sig_info,
                              merge_sig_info(new_env,parent.sig_info));
        endif;
      endif;
    end;
  end;
end;

match ?s=mult_sig(?s1,?s2) begin
  s.sig_info := append_sig_info(s1.sig_info,s2.sig_info);
end;


--- Now signature info for type declarations

-- a type declaration that uses itself gets zippo for sigs:
match ?d:Declaration begin
  if d.depends_on_self then
    d.sig_info := empty_sig_info;
  endif;
end;

-- if no type given, presume that TYPE[] or PHYLUM[] is being called.
match ?td=type_decl(?,no_sig(),no_type()) begin
  td.sig_info := add_result_to_sig_info(td,SampleType_decl.sig_info);
end;
match ?td=phylum_decl(?,no_sig(),no_type()) begin
  td.sig_info := add_result_to_sig_info(td,SamplePhylum_decl.sig_info);
end;

match ?td=some_type_decl(?,?sig,?ty) begin
  case sig begin
    match no_sig() begin
      td.sig_info := add_result_to_sig_info(td,ty.sig_info);
    end;
  else
    td.sig_info := add_result_to_sig_info(td,sig.sig_info);
  end;
```

```
end;

match ?tf=some_type_formal(?,?sig) begin
  tf.sig_info := add_result_to_sig_info(tf,sig.sig_info);
end;

match ?td=type_renaming(?,?ty) begin
  td.sig_info := add_result_to_sig_info(td,ty.sig_info);
end;

-- an inheritance is considered a type declaration:
match ?td=inheritance(?,?ty,?) begin
  td.sig_info := add_result_to_sig_info(td,ty.sig_info);
end;


--- And now sig info for types

match ?ty=type_use(?use) begin
  case contextual_def(use) begin
    match contextual(?e,?td=type_renaming(?,?ty2)) begin
      if not td.depends_on_self then
        -- base extension is pointless unless environment is unbound
        case e begin
          match some_unbound() begin
            ty.base_extension := ty2.base_extension;
          end;
        else
          ty.base_extension := nil;
        end;
        ty.sig_info := merge_sig_info(e,ty2.sig_info);
      else
        ty.base_extension := nil;
        ty.sig_info := empty_sig_info;
      endif;
    end;
    match contextual(?environ,?td) begin
      -- It's not necessary to test depends_on_self
      -- because we already do it for type declarations.
      -- Base extension is pointless unless environment is unbound
      case environ begin
        match some_unbound() begin
          ty.base_extension := td;
        end;
      else
        ty.base_extension := nil;
      end;
      ty.sig_info := merge_sig_info(environ,td.sig_info);
    end;
  end;
```

```
end;

match ?ty=type_inst(?m,?tactuals,?) begin
  case m.base_module begin
    match contextual(?,!NULL_TYPE_decl) begin
      ty.sig_info := {};
    end;
    match contextual(?,!NULL_PHYLUM_decl) begin
      ty.sig_info := {};
    end;
    match contextual
        (?environ,
         ?md=module_decl(type_formals:=?tfs,
                         parent:=?parent,
                         result_type:=?rd=some_type_decl(?,?,?result)))
    begin
      if length(tfs) = length(tactuals) then
        new_env : Environment := type_inst_env(ty);
        parent_sig_info : SigInfoType :=
            merge_sig_info(new_env,parent.sig_info);
        base_sig_info : SigInfoType :=
            append_sig_info(make_sig_info(new_env,md),parent_sig_info);
        new_sig_info : SigInfoType :=
            append_sig_info(base_sig_info,
                            merge_sig_info(new_env,rd.sig_info));

        if md.depends_on_self then
          ty.sig_info := make_sig_info(new_env,md);
        else
          -- NB: we could change the semantics and substitute the tf in
          -- the whole environment, possibly getting other type formals too.
          case result.base_extension begin
            match ?tf=some_type_formal(...) if tf in tfs
            begin
              extending : Type := nth(position(tf,tfs),tactuals);
              ty.extension_actual := extending;
              ty.base_extension := extending.base_extension;
              ty.sig_info := append_sig_info(extending.sig_info,
                                             new_sig_info);
            end;
          else
            ty.sig_info := new_sig_info;
          end;
        endif;
      endif;
    end;
  end;
end;

match ?ty1=remote_type(?ty2) begin
```

```
  ty1.sig_info := ty2.sig_info;
  ty1.base_extension := ty2.base_extension;
end;

match ?ty1=private_type(?ty2) begin
  ty1.sig_info := ty2.sig_info;
  ty1.base_extension := nil; -- hidden by "private"
end;



---- Comparing signatures


function signatures_equalp(envs1,envs2 : Environments) : Boolean :=
    length(envs1)=length(envs2) and
    (((signature_equalp(sig1,sig2) for sig1 in envs1) or false)
         for sig2 in envs2);

function signature_equalp(sig1,sig2 : Environment) : Boolean begin
  case Environments${sig1,sig2} begin
    match {bound_rib(?cd=some_class_decl(type_formals:=?tfs),?i,?v,?m1,?n1),
           bound_rib(!cd,!i,!v,?m2,?n2)} begin
      result :=
          (contextual_type_equalp(apply_environ(sig1,tf),
                                  apply_environ(sig2,tf))
               for tf in tfs) and
          environment_equalp(n1,n2);
    end;
  end;
end;

function signatures_included_in(envs1,envs2 : Environments) : Boolean
begin
  collection each_included : Boolean :> true, (and);
  for env1 in envs1 begin
    case env1 begin
      match bound_rib(?cd=some_class_decl(type_formals:=?tfs),?i1,?v1,?,?n1)
      begin
        case envs2 begin
          match {...,?env2=bound_rib(!cd,?i2,?v2,?,?n2),...} begin
            each_included :>
                (not i1 or i2) and
                (not v1 or v2) and
                (contextual_type_equalp(apply_environ(env1,tf),
                                        apply_environ(env2,tf))
                     for tf in tfs) and
                environment_equalp(n1,n2);
          end;
        else
```

```
          each_included :> false;
        end;
      end;
    end;
  end;
  result := each_included;
end;



---- Limited Signatures:

-- the set of types that a signature is limited to, or {} if
-- no limit (yes this is a dumb idea):
attribute Signature.limited_types : ContextualTypeSet := {};
type ContextualTypeSet := SET[ContextualType](contextual_type_equalp);

function merge_limited(e : Environment; limited : ContextualTypeSet)
    : ContextualTypeSet
    := {merge_contextual(e,ct) for ct in limited};

match ?s=sig_use(?u) begin
  case u.contextual_def begin
    match contextual(?e,?sd=some_signature_decl(?,?s2)) begin
      if not sd.depends_on_self then
        s.limited_types := merge_limited(e,s2.limited_types);
      endif;
    end;
  end;
end;

match ?s=mult_sig(?s1,?s2) begin
  if s1.limited_types = {} then
    s.limited_types :=
        {ty if type_has_signatures(ty,s1.sig_info)
             for ty in s2.limited_types};
  elsif s2.limited_types = {} then
    s.limited_types :=
        {ty if type_has_signatures(ty,s2.sig_info)
             for ty in s1.limited_types};
  else
    -- only legal if neither is empty (that is, unlimited)
    -- NB: if this intersection is empty, the signature
    -- is erroneous (detected in aps-typecheck)
    s.limited_types :=
        {ty if type_has_signatures(ty,s1.sig_info)
             for ty in s2.limited_types} /\
        {ty if type_has_signatures(ty,s2.sig_info)
             for ty in s1.limited_types};
  endif;
```

```
end;

match ?s=fixed_sig(?types) begin
  -- if the set is empty, the signature is erroneous
  s.limited_types := {contextual(t.environment,t)
                          if type_base_known_p(t)
                          for t in types};
end;


-- The following procedure is a weakening of contextual_type_equalp
-- used in certain situations in patterns (in match_patterns :?,
-- and for checking pattern_decl formal bindings).  A pattern
-- can match if the required type is a type formal which is
-- limited to match a set of types and the type in question is
-- in the set.  If the second type is not a type_formal, then
-- normal processing is used.
var function contextual_type_matchp(ct : ContextualType; ty : remote Type)
    : Boolean
begin
  case ty begin
    match type_use(?u) begin
      case u.contextual_def begin
        -- A type formal can only be used in its scope,
        -- so we are guaranteed that the context will be trivial.
        match contextual(unbound_rib(...),some_type_formal(?,?sig)) begin
          limited : ContextualTypeSet := sig.limited_types;
          if limited /= {} then
            result := type_in_limited(ct,limited);
          endif;
        end;
      end;
    end;
  end;
  result := contextual_type_equalp(ct,contextual(ty.environment,ty));
end;
--
var function type_in_limited(ct : ContextualType;
                             limited : ContextualTypeSet)
    : Boolean
begin
  if ct in limited then
    result := true;
  else
    -- ct could be a type formal limited to a subset of the types:
    case ct begin
      match contextual(?e,type_use(?u)) begin
        case u.contextual_def begin
          -- We can ignore the environment because uses of type formals
          -- are always unbound.
```

```
          match contextual(?,?tf=some_type_formal(?,?sig)) begin
            type_limited : ContextualTypeSet
                := merge_limited(e,sig.limited_types);
            if type_limited /= {} and type_limited <= limited then
              result := true;
            else
              result := false;
            endif;
          end;
        end;
      end;
    end;
  endif;
  result := false; -- otherwise
end;

function merge_environments(env : Environment; envs : Environments)
    : Environments
    := {merge_environ(env,e) for e in envs};

var function type_has_signatures(ct : ContextualType;
                                 rsigs : Environments) : Boolean
begin
  case ct begin
    match contextual(?environ,?ty) begin
      tsigs : Environments := merge_environments(environ,ty.sig_info);
      result := signatures_included_in(rsigs,tsigs);
    end;
  end;
end;

end;
```

C.3 Type-Checking

The APS type checker takes a tree for which name resolution has been completed, with the exception of type inference of uses of polymorphic entities. It performs the type inference and as generates error messages related to type-checking. It uses the preliminary name resolution of `APS_RESOLVE` (available in the attribute `pre_contextual_def`) and some type manipulation functions from `APS_TYPE`.

After completion, name binding is complete (`APS_BOUND`, see Appendix C.2.2) and typing information is available for many nodes (`APS_TYPED`):

```
module APS_TYPED[Input :: var ABSTRACT_APS[], var APS_ENVIRON[Input]]
    extends Input
begin
  [phylum T ::{Expression,Pattern,Declaration,Default}, var PHYLUM[]]
      input attribute T.computed_type : ContextualType;
  [phylum T :: {Pattern,Expression,Type}, var PHYLUM[]]
```

```
      input attribute T.actual_formal : remote Declaration := nil;
end;
```

This module is one of the most complicated in the APS compiler, and requires the use of monotonic guards, as described in Section 7.3.2.

```
module APS_TYPECHECK[Input :: var ABSTRACT_APS[],
                              var APS_PREDEFINED[Input],
                              var APS_ENVIRON[Input],
                              var APS_RESOLVE[Input],
                              var APS_DETECT_CYCLE[Input],
                              var APS_TYPE_INFO[Input],
                              var APS_SIG_INFO[Input]]
    :: var APS_BOUND[Input], var APS_TYPED[Input]
    extends Input
    -- The task of this module is to annotate each expression
    -- and pattern with an inferred type.  At the same time it
    -- generates type error messages for mismatched types.
    -- These two tasks are done together because the type inference
    -- algorithm compares types wherever the language requires them
    -- to be the same.
    -- Declarations also get these attributes:
    --   some declarations have missing types (to be inferred)
    --   and thus may also need error messages.
begin
  signature TYPEABLE := {Expression,Pattern,Declaration,Default}, var PHYLUM[];

  [phylum T :: TYPEABLE]
      attribute T.computed_type : ContextualType;

  [phylum T :: {Expression,Pattern,Declaration,Default,
                Type,Signature,Class,Module},
               var PHYLUM[]]
      collection attribute T.type_errors : TypeErrors;

  [phylum T :: {Pattern,Expression,Type}, var PHYLUM[]]
      attribute T.actual_formal : remote Declaration := nil;

  type TypeErrors := BAG[String];

  -- this module also completes the name lookup by filling in the
  -- type variables:

  attribute (u:Use).contextual_def : ContextualDeclaration :=
      u.pre_contextual_def;
  -- re-export from APS_TYPE
  depends_on_self = Input$depends_on_self;


  -- private;
```

```
[T :: SIGNATURED]
    sigs = (sig_info : function(_:T):Environments);

-- the rest of this module is highly convoluted:



---- VAR ENVIRONMENTS, CONTEXTS and TYPE VARIABLES

--- Var Environments
-- Var environments are an addition to regular Environments
-- (some ribs, those for poly, use type variables)

type VarEnvironment;
type TypeVariables := LIST[TypeVariable];
type VarEnvironmentMapping :=
    MAP[remote Declaration,PartialType](return_error_type);
function return_error_type(_:remote Declaration) : PartialType := error_type;
var_mapping = VarEnvironmentMapping$map;
pattern var_mapping = VarEnvironmentMapping$map;

constructor var_rib(poly : remote Declaration;
                    input_cap, var_cap : Boolean;
                    variables : TypeVariables;
                    next : VarEnvironment) : VarEnvironment;
constructor var_bound_rib(decl : remote Declaration;
                          input_cap, var_cap : Boolean;
                          mapping : VarEnvironmentMapping;
                          next : VarEnvironment) : VarEnvironment;
constructor no_var(e : Environment) : VarEnvironment;

pattern some_var_rib(decl : remote Declaration;
                     input_cap, var_cap : Boolean;
                     next : VarEnvironment) : VarEnvironment :=
    var_rib(?decl,input_cap:=?input_cap,var_cap:=?var_cap,next:=?next),
    var_bound_rib(?decl,input_cap:=?input_cap,var_cap:=?var_cap,next:=?next);

pattern some_bound_rib(decl : remote Declaration;
                       input_cap, var_cap : Boolean) : VarEnvironment :=
    var_bound_rib(?decl,?input_cap,?var_cap,...),
    no_var(bound_rib(?decl,?input_cap,?var_cap,...));

empty_var_env : VarEnvironment := no_var(empty_env);

-- the merge functions are a bit complicated but leave most of the
-- hard work to merge_environ:
function merge_var_environ(ve1,ve2 : VarEnvironment) : VarEnvironment
begin
  case ve1 begin
    match no_var(?e1) begin
```

```
      result := merge_environ_var_environ(e1,ve2);
    end;
    match some_var_rib(?,?i1,?v1,?) begin
      case ve2 begin
        match no_var(?e2) begin
          result := merge_var_environ_environ(ve1,e2);
        end;
        match var_rib(?poly,?i2,?v2,?tvs,?next) begin
          result := var_rib(poly, i1 and i2, v1 and v2, tvs,
                            merge_var_environ(ve1,ve2));
        end;
        match var_bound_rib(?decl,?i2,?v2,?mapping,?next) begin
          result := var_bound_rib(decl, i1 and i2, v1 and v2,
                                  merge_var_mapping(ve1,mapping),
                                  merge_var_environ(ve1,next));
        end;
      end;
    end;
  end;
end;
function merge_environ_var_environ(e1 : Environment; ve2 : VarEnvironment)
    : VarEnvironment
begin
  case e1 begin
    match root_env() begin
      result := ve2;
    end;
    match unbound_rib(...) begin
      result := ve2;
    end;
    match bound_rib(?,?i1,?v1,...) begin
      case ve2 begin
        match no_var(?e2) begin
          result := no_var(merge_environ(e1,e2));
        end;
        match var_rib(?poly,?i2,?v2,?tvs,?next) begin
          result := var_rib(poly,i1 and i2,v1 and v2,tvs,
                            merge_environ_var_environ(e1,ve2));
        end;
        match var_bound_rib(?decl,?i2,?v2,?mapping,?next) begin
          ve1 : VarEnvironment := no_var(e1);
          result := var_bound_rib(decl, i1 and i2, v1 and v2,
                                  merge_var_mapping(ve1,mapping),
                                  merge_environ_var_environ(e1,next));
        end;
      end;
    end;
  end;
end;
function merge_var_environ_environ(ve1 : VarEnvironment; e2 : Environment)
```

```
    : VarEnvironment
begin
  case ve1 begin
    match no_var(?e1) begin
      result := no_var(merge_environ(e1,e2));
    end;
    match some_var_rib(?decl1,?i1,?v1,?next1) begin
      case e2 begin
        match bound_rib(?decl2,?i2,?v2,?mapping2,?next2) begin
          result :=
              var_bound_rib(decl2, i1 and i2, v1 and v2,
                            merge_var_environ_mapping(ve1,mapping2),
                            merge_var_environ_environ(ve1,next2));
        end;
        match unbound_rib(!decl1,...) begin
          result := ve1; -- replace unbound section with var environment
        end;
        match root_env() begin
          result := no_var(e2);
        end;
      else
        -- e2 is an unbound rib, we look inside
        -- ve1 for the ending to it.  We may never find one
        -- and end up using the first rule of this function
        result := merge_var_environ_environ(next1,e2);
      end;
    end;
  end;
end;
pragma no_memo(merge_environ_var_environ);

function merge_var_mapping(ve : VarEnvironment;
                           vm : VarEnvironmentMapping)
    : VarEnvironmentMapping
begin
  case vm begin
    match var_mapping(?pairs) begin
      result := var_mapping({merge_pair(p) for p in pairs});
    end;
  end;
  function merge_pair(p : VarEnvironmentMapping$PairType)
      : VarEnvironmentMapping$PairType
  begin
    case p begin
      match (=>)(?from,?to) begin
        result := from=>merge_partial_type(ve,to);
      end;
    end;
  end;
end;
```

```
function merge_var_environ_mapping(ve : VarEnvironment;
                                   m : EnvironmentMapping)
    : VarEnvironmentMapping
begin
  case m begin
    match environment_mapping(?pairs) begin
      result := var_mapping({merge_pair(p) for p in pairs});
    end;
  end;
  function merge_pair(p : EnvironmentMapping$PairType)
      : VarEnvironmentMapping$PairType
  begin
    case p begin
      match (=>)(?from,?to) begin
        result := from=>make_partial_contextual_type(ve,to);
      end;
    end;
  end;
end;
pragma no_memo(merge_var_mapping,merge_var_environ_mapping);

function apply_var_environ(ve : VarEnvironment;
                           d : remote Declaration) : PartialType
begin
  case ve begin
    match var_bound_rib(mapping:=?mapping) begin
      result := VarEnvironmentMapping$apply(mapping,d);
    end;
    match no_var(?e) begin
      result := make_var(apply_environ(e,d));
    end;
  else
    result := error_type;
  end;
end;


--- Conversion
-- in aps-resolve, bogus bound_ribs are created for polymorphically
-- used entities.  here we first convert them to VarEnvironments
-- (in a procedure since we need to create type variables)
-- and then convert them back using the final bindings for type variables.

[phylum T :: TYPEABLE] begin
  var procedure environ2var_environ(owner : remote T;
                                    e : Environment) : VarEnvironment begin
    case e begin
      match bound_rib(?poly=polymorphic(?,?tfs,?),?i,?v,?,?next) begin
        result := var_rib(poly,i,v,
                          {type_variable(tf,owner) for tf in tfs},
```

```
                          environ2var_environ(owner,next));
      end;
    else
      result := no_var(e);
    end;
  end;
end;

var function fix_var_environ(ve : VarEnvironment) : Environment begin
  case ve begin
    match var_rib(?poly=polymorphic(?,?tfs,?),?i,?v,?tvs,?next) begin
      result :=
          bound_rib(poly,i,v,
                    environment_mapping({nth(i,tfs)=>nth(i,tvs).binding
                                         for i in 0..(length(tfs)-1)}),
                    fix_var_environ(next));
    end;
    match var_bound_rib(?decl,?i,?v,?mapping,?next) begin
      result := bound_rib(decl,i,v,
                          fix_var_mapping(mapping),
                          fix_var_environ(next));
    end;
    match no_var(?e) begin
      result := e;
    end;
  end;
end;

var function fix_var_mapping(vm : VarEnvironmentMapping)
    : EnvironmentMapping
begin
  case vm begin
    match var_mapping(?pairs) begin
      result := environment_mapping({fix_pair(pair) for pair in pairs});
    end;
  end;
  function fix_pair(pair : VarEnvironmentMapping$PairType)
      : EnvironmentMapping$PairType
  begin
    case pair begin
      match (=>)(?tf,?pt) begin
        result := tf=>fix_partial_type(pt);
      end;
    end;
  end;
end;

pragma no_memo(fix_var_mapping);
```

```
--- VAR_CONTEXTUAL
-- The var contextual module is a variation on the contextual module.
-- During the type inference process, we create type variables
-- for unknown types and these type variables are used in
-- VarEnvironments as the bindings for poly type formals.

module VAR_CONTEXTUAL[BaseType :: BASIC[]](default : BaseType) :: BASIC[]
begin
  constructor var_contextual(var_environ : VarEnvironment;
                             base : BaseType) : Result;
  constructor no_var_contextual() : Result;
  default_base = default;
end;
[BaseType; VT :: VAR_CONTEXTUAL[BaseType]] begin
  var_contextual = VT$var_contextual;
  no_var_contextual = VT$no_var_contextual;
  pattern var_contextual = VT$var_contextual;
  pattern no_var_contextual = VT$no_var_contextual;
end;
[BaseType; VT :: VAR_CONTEXTUAL[BaseType]] begin
  function merge_var_contextual(env : VarEnvironment; x : VT) : VT begin
    case x begin
      match VT$var_contextual(?sub,?base) begin
        result := VT$var_contextual(merge_var_environ(env,sub),base);
      end;
    else
      result := x;
    end;
  end;
end;
[BaseType; VT :: VAR_CONTEXTUAL[BaseType]; T :: CONTEXTUAL[BaseType]] begin
  function make_var(c : T) : VT begin
    case c begin
      match T$contextual(?e,?base) begin
        result := VT$var_contextual(no_var(e),base);
      end;
    else
      result := VT$no_var_contextual();
    end;
  end;
  function var_merge_contextual(ve : VarEnvironment; c : T) : VT begin
    case c begin
      match T$contextual(?e,?base) begin
        result := VT$var_contextual(merge_var_environ_environ(ve,e),base);
      end;
    else
      result := VT$no_var_contextual();
    end;
  end;
```

```
  var function fix_var_contextual(v : VT) : T begin
    case v begin
      match VT$var_contextual(?ve,?base) begin
        result := T$contextual(fix_var_environ(ve),base);
      end;
    else
      result := T$no_contextual();
    end;
  end;
end;

var function fix_partial_type(pt : PartialType) : ContextualType begin
  -- we have to detect the use of a type variable and
  -- substitute it out.
  --?? Is this the only case we need to substitute?
  tv : remote TypeVariable := detect_type_variable(pt);
  if tv /= nil then
    result := tv.binding;
  else
    result := fix_var_contextual(pt);
  endif;
end;

pragma no_memo(fix_partial_type);



-- Partial types and partial signatures are computed during
-- the type inference process. They have to be fixed before being exported.

type PartialType := VAR_CONTEXTUAL[remote Type](nil);
type PartialTypes := LIST[PartialType];

type VarContextualDeclaration := VAR_CONTEXTUAL[remote Declaration](nil);

type PartialSig := VarEnvironment;
type PartialSigs := LIST[PartialSig];

error_type : PartialType := no_var_contextual();
error_sigs : PartialSigs := {};

[phylum T :: TYPEABLE]
    attribute T.partial_type : PartialType := error_type;

[phylum T :: TYPEABLE] match ?x:T begin
  x.computed_type := fix_partial_type(x.partial_type);
end;

pattern partial_sig(decl : remote Declaration;
                    input_cap, var_cap : Boolean) : PartialSig :=
```

```
    no_var(bound_rib(?decl,?input_cap,?var_cap,...)),
    var_bound_rib(?decl,?input_cap,?var_cap,...);

function make_partial_sigs(envs : Environments) : PartialSigs
    := {no_var(env) for env in envs};

function partial_type_sigs(pt : PartialType) : PartialSigs begin
  case pt begin
    match var_contextual(?ve,?ty) begin
      result := merge_partial_sigs(ve,make_partial_sigs(ty.sigs));
    end;
  else
    result := error_sigs;
  end;
end;

function merge_partial_sigs(ve : VarEnvironment;
                            partial_sigs : PartialSigs)
    : PartialSigs
    := {merge_var_environ(ve,venv) for venv in partial_sigs};


--- Type Variables
-- Type variables arise in two different ways:
-- 1> directly through untyped formals (in patterns or implied patterns)
--    (or through polymorphic primitives such as the empty sequence
--     (), undefined_expr, or ...)
-- 2> through polymorphic entities (for which we need an environment)
-- NB: the type formal is nil for <1>

phylum TypeVariable;
[phylum T :: TYPEABLE]
    constructor type_variable(tformal : remote Declaration;
                              owner : remote T) : TypeVariable;

pattern some_type_variable(tformal : remote Declaration) : TypeVariable
    := type_variable(?tformal,?:Expression),
       type_variable(?tformal,?:Pattern),
       type_variable(?tformal,?:Declaration);

-- sometimes ane entity's type must be inferred
-- (for example, every ?x in a pattern) and for that
-- we need to generate a PartialType from a TypeVariable
-- we do this by using a predefined empty polymorphic entity lying around:

blank_type_formal_use : remote Type
    := get_blank_type_formal_use(blank_poly_decl);
function get_blank_type_formal_use(d : remote Declaration)
    : remote Type
begin
```

```
  case d begin
    match polymorphic(?,{?},block({type_renaming(?,?ty)})) begin
      result := ty;
    end;
  end;
end;
pragma no_memo(get_blank_type_formal_use);

function var_partial_type(tv : remote TypeVariable) : PartialType
    := var_contextual(var_rib(blank_poly_decl,true,true,{tv},
                              no_var(blank_poly_decl.environment)),
                      blank_type_formal_use);

--- Type Variable attributes:

-- these attributes are the result of the work of this module:
-- each type variable gets a final binding to a type:
attribute TypeVariable.binding : ContextualType := var_unbound;
var_unbound : ContextualType := no_contextual();

-- Every type variable gets
-- a set of other type variables it is equal to and
-- a set of bindings to partial types

var type TypeVariableSet := ORDERED_SET[remote TypeVariable]((==),(<<));
type BindingSet := SET[PartialType]((=));

var type TypeVariablesLattice :=
    UNION_LATTICE[remote TypeVariable,TypeVariableSet];
type BindingsLattice := UNION_LATTICE[PartialType,BindingSet];
circular collection attribute TypeVariable.chain : TypeVariablesLattice;
circular collection attribute TypeVariable.bindings : BindingsLattice;
circular collection attribute TypeVariable.consistent : AndLattice;
circular collection attribute TypeVariable.fits_signature : AndLattice;

[T :: TYPEABLE] match ?ty1=type_variable(?decl,?owner:T) begin
  ty1.chain :> {ty1};
  -- force closure:
  -- (this is probably not too efficient)
  for ty2 in ty1.chain begin
    if ty1 /= ty2 then
      ty2.bindings :> ty1.bindings;
      ty2.chain :> ty1.chain;
      ty1.bindings :> ty2.bindings;
      ty1.chain :> ty2.chain;
    endif;
  end;
  -- for each two bindings, we call a procedure that does
  -- type inference for two types that must be identical:
  for b1 in ty1.bindings begin
```

```
  for b2 in ty1.bindings begin
    ty1.consistent :> require_type_equal(b1,b2);
  end;
end;
-- each binding must satisfy the signature
-- (necessary to get other type variables bound)
case decl begin
  match !nil begin end;
  match ?td=some_type_formal(...) begin
    formal_sigs : PartialSigs :=
        merge_partial_sigs(owner.use_var_environ,
                           make_partial_sigs(td.sigs));
    for b in ty1.bindings begin
      if not require_sig(b,formal_sigs) then
        ty1.fits_signature :> false;
      endif;
    end;
  end;
end;
-- add message if signature doesn't fit
if not ty1.fits_signature then
  owner.type_errors :> {"type variable binding does not fit signatures"};
endif;
-- set the binding or flag as inconsistent
if ty1.consistent then
  case ty1.bindings begin
    match {?ty,...} begin
      ty1.binding := fix_partial_type(ty);
    end;
  else
    owner.type_errors :> {"type variable unbound"};
  end;
else
  owner.type_errors :> {"type variable has inconsistent binding"};
endif;
-- check limited type formals:
case decl begin
  match !nil begin end;
  match some_type_formal(?,?sig) begin
    limited : ContextualTypeSet :=
        {merge_contextual(owner.use_fixed_environ,ct)
             for ct in sig.limited_types};
    if limited /= {} and
        not type_in_limited(ty1.binding,limited) then
      owner.type_errors :>
          {"type variable binding does not fit limitation"};
    endif;
  end;
end;
-- check binding is not private:
```

```
  case ty1.binding begin
    match contextual(?env,?ty) begin
      if type_is_private(env,ty) then
        owner.type_errors :> {"type variable binding is private"};
      endif;
    end;
  end;
end;

-- return whether the type is implementable without looking at private
-- parts of other modules:
function type_is_private(env : Environment; ty : remote Type) : Boolean
begin
  case ty begin
    match type_use(?u) begin
      case u.pre_contextual_def begin
        match contextual(?sub,?decl) begin
          new_env : Environment := merge_environ(env,sub);
          result := decl_is_private(new_env,decl);
        end;
      end;
    end;
    match remote_type(?ty1) begin
      result := type_is_private(env,ty1);
    end;
    match private_type(?ty1) begin
      -- this is a different sense of private than that being tested here
      result := type_is_private(env,ty1);
    end;
  else
    -- some sort of error
    result := true;
  end;
  -- default
  result := false;
end;

function decl_is_private(env : Environment;
                         decl : remote Declaration) : Boolean
begin
  case env begin
    match root_env() begin
      result := false;
    end;
    match unbound_rib(...) begin
      result := false;
    end;
    match bound_rib(...) begin
      case decl begin
        match declaration(def(is_public:=?is_public)) begin
```

```
          result := not is_public;
        end;
      else
        result := true;
      end;
    end;
  end;
end;

pragma no_memo(decl_is_private);



---- CIRCULARITY CHECKING

match ?d:Declaration begin
  if d.depends_on_self then
    d.type_errors :> {"Circular renaming or dependency"};
  endif;
end;



---- SIGNATURE CHECKING

--- Checking uses of fixed_sig: {...} signatures
-- These signatures are very special and can only be used
-- to constrain formal parameters. not as a parent signature
-- for modules or signatures.  Furthermore, it is not legal
-- to have an empty set of possible types (this rule is not
-- logically necessary, but makes the implementation convenient;
-- it almost certainly indicates an error).  And all the types
-- in a fixed_sig must be known to be different (all must
-- have at their base a known call to TYPE)

match ?s=fixed_sig({}) begin
  s.type_errors :> {"illegal empty finite signature"};
end;

match fixed_sig({...,?ty,...}) begin
  if not type_base_known_p(ty) then
    ty.type_errors :> {"does not have a known base type"};
  endif;
end;

match ?s=mult_sig(?s1,?s2) begin
  if s.limited_types = {} and
      (s1.limited_types /= {} or s2.limited_types /= {}) then
    s.type_errors :> {"illegal empty finite signature"};
  endif;
```

```
end;

match some_class_decl(parent:=?parent) begin
  if parent.limited_types /= {} then
    parent.type_errors :> {"illegal use of finite signature"};
  endif;
end;


--- Checking signatures for duplicates
-- signatures may not include duplicate copies of a signature
-- unless they're bindings of the formals are equal

match ?s=mult_sig(?s1,?s2) begin
  for s1.sigs begin
    match {...,?e1=bound_rib(?cd=some_class_decl(def:=def(?name,...),
                                                 type_formals:=?tfs),...),
           ...}
    begin
      for s2.sigs begin
        match {...,?e2=bound_rib(!cd,...),...} begin
          if false or (not contextual_type_equalp(apply_environ(e1,tf),
                                                  apply_environ(e2,tf))
                          for tf in tfs)
          then
            s.type_errors :> {"contradictory " || name || " signatures"};
          endif;
        end;
      end;
    end;
  end;
end;


-- Each type declaration must satisfy its declared signatures:
match ?td=some_type_decl(?,?sig,?ty) begin
  -- we could compute the sigs and then add the result, but the
  -- work is already done for teh declaration itself.  In that case,
  -- we want to avoid testing the inferred signatures against themselves.
  -- So we only do the test when signatures are given:
  case sig begin
    match no_sig() begin end;
  else
    if not require_sig(make_simple_partial_type(ty),
                       make_partial_sigs(td.sigs)) then
      td.type_errors :> {"Assignment does not satisfy signatures"};
    endif;
  end;
end;
```

```
pattern some_type_equality(def : Def; (type) : Type) : Declaration
    := type_renaming(?def,?(type)), some_type_decl(?def,?,?(type));

match ?d=type_replacement(?old,?new) begin
  if not require_sig(make_simple_partial_type(new),
                     make_partial_sigs(old.sigs)) then
    d.type_errors :> {"Replacement does not satisfy signatures"};
  endif;
  case old begin
    match type_use(?u) begin
      case u.pre_contextual_def begin
        -- don't check base type if private:
        match contextual(?,some_type_decl(?,?,private_type(...))) begin end;
        -- Now we cannot just compare the two types becasue by the
        -- substitution in the inheritance they will be identical.
        -- We have to delve a little deeper.
        -- We do that by forming the base type *before* mergeing with
        -- the environment:
        match contextual(?e,?decl) begin
          pt : PartialType :=
              make_partial_contextual_type
              (no_var(e),make_contextual_base_type(decl.environment,
                                                   decl.predefined_use));
          if not require_type_equal(pt,make_simple_partial_type(new)) then
            d.type_errors :> {"Replacement not the same type"};
          endif;
        end;
      end;
    end;
  end;
end;

match ?d=signature_replacement(...) begin
  d.type_errors :> {"Signature replacement is not permitted"};
end;


--- Signature instantiation

-- each signature instance must be given enough parameters
match ?si=sig_inst(?,?,?cl,?tactuals) begin
  case cl.base_class begin
    match contextual(?,some_class_decl(type_formals:=?tfs)) begin
      if length(tactuals) /= length(tfs) then
        if length(tactuals) < length(tfs) then
          si.type_errors :> {"not enough type parameters to class"};
        else
          si.type_errors :> {"too many type parameters to class"};
        endif;
      endif;
```

```
    end;
  end;
end;

pattern some_class_decl_or_renaming() : Declaration :=
    class_renaming(...), class_decl(...), module_decl(...);

match ?cl=class_use(?u) begin
  case u.pre_contextual_def begin
    match contextual(?,some_class_decl_or_renaming(...)) begin end;
    match contextual(...) begin
      cl.type_errors :> {"not a class"};
    end;
  end;
end;

match ?s=sig_use(?u) begin
  case u.pre_contextual_def begin
    match contextual(?,some_class_decl_or_renaming(...)) begin
      s.type_errors :> {"must be instantiated"};
    end;
  end;
end;


---- TYPE CHECKING

--- Type Requirements
-- Type variables get their bindings by being

-- for every kind of expression, pattern or declaration,
-- we do some checking:

function make_simple_partial_type(raw : remote Type) : PartialType begin
  if raw == nil then
    result := error_type;
  else
    result := var_contextual(no_var(raw.environment),raw);
  endif;
end;

pattern environment_rib(e : Environment) : Environment :=
    ?e, some_rib(next:=environment_rib(?e));
pattern var_environment_rib(e : Environment) : VarEnvironment :=
    no_var(environment_rib(?e)), some_var_rib(next:=var_environment_rib(?e));
pattern var_environment_var_rib(ve : VarEnvironment) : VarEnvironment :=
    ?ve, some_var_rib(next:=var_environment_var_rib(?ve));

-- NB: this complicated function is called to get the argument or
-- result types of a function_type that may have been contextual.
```

```
-- It is necessary to look up what the types mean in the context of
-- the call in order to get the right types to use for type inference.
function make_partial_type(ve : VarEnvironment;
                           ty : remote Type) : PartialType
begin
  -- compare with make_contextual_type in aps-type.aps
  case ty begin
    match !nil begin
      result := error_type;
    end;
    match type_use(?u) begin
      case u.pre_contextual_def begin
        match contextual(?sub,?decl) begin
          venv : VarEnvironment := merge_var_environ_environ(ve,sub);
          substituted : PartialType := apply_var_environ(venv,decl);
          if substituted = error_type then
            result := var_contextual(venv,decl.predefined_use);
          else
            result := simplify_partial_type(substituted);
          endif;
        end;
      end;
    end;
  end;
  -- default:
  result := var_contextual(ve,ty);
end;

function simplify_partial_type(pt : PartialType) : PartialType begin
  case pt begin
    match var_contextual(?venv,?ty) begin
      result := make_partial_type(venv,ty);
    end;
  end;
end;

function apply_and_simplify(ve : VarEnvironment; td : remote Declaration)
    : PartialType
    := simplify_partial_type(apply_var_environ(ve,td));

function make_partial_contextual_type(ve : VarEnvironment;
                                      ct : ContextualType) : PartialType
begin
  case ct begin
    match contextual(?e,?ty) begin
      result := make_partial_type(merge_var_environ_environ(ve,e),ty);
    end;
  else
    result := error_type;
  end;
```

```
end;

function make_contextual_type_partial(ct : ContextualType) : PartialType
begin
  case ct begin
    match contextual(?e,?ty) begin
      result := make_partial_type(no_var(e),ty);
    end;
  else
    result := error_type;
  end;
end;

function merge_partial_type(ve : VarEnvironment;
                            pt : PartialType) : PartialType
begin
  case pt begin
    match var_contextual(?sub,?ty) begin
      result := make_partial_type(merge_var_environ(ve,sub),ty);
    end;
  else
    result := error_type;
  end;
end;

pragma no_memo(make_partial_type,make_partial_contextual_type,
               make_contextual_type_partial,merge_partial_type);

function make_partial_base_type(ve : VarEnvironment; ty : remote Type)
    : PartialType
begin
  case ty begin
    match type_use(?u) begin
      case u.pre_contextual_def begin
        match contextual(?sub,?decl) begin
          venv : VarEnvironment := merge_var_environ_environ(ve,sub);
          substituted : PartialType := apply_var_environ(venv,decl);
          if substituted = error_type then
            case decl begin
              match type_renaming(?,?ty) begin
                result := make_partial_base_type(venv,ty);
              end;
              -- base types are not expanded:
              match some_type_decl(?,?,no_type()) begin
                result := var_contextual(venv,decl.predefined_use);
              end;
              match some_type_decl(?,?,?ty) begin
                result := make_partial_base_type(venv,ty);
              end;
              match inheritance(?,?ty,?) begin
```

```
                result := make_partial_base_type(venv,ty);
              end;
            else
              result := var_contextual(venv,decl.predefined_use);
            end;
          else
            -- try another go around:
            result := partial_type_base_type(substituted);
          endif;
        end;
      end;
    end;
    match remote_type(?ty) begin
      result := make_partial_base_type(ve,ty);
    end;
    match type_inst(?mod,?tactuals,?) begin
      case mod.base_module begin
        match contextual(?sub,?md=module_decl(type_formals:=?tfs,
                                              result_type:=?rd))
            if length(tfs) = length(tactuals)
        begin
          -- if the result is a base type, we do not expand
          case rd begin
            match some_type_decl(?,?,no_type()) begin
              result := var_contextual(ve,ty);
            end;
            match some_type_decl(?,?,?ty1) begin
              venv : VarEnvironment :=
                  merge_var_environ_environ(ve,type_inst_env(ty));
              result := make_partial_base_type(venv,ty1);
            end;
          end;
        end;
      end;
    end;
  end;
  -- default:
  result := var_contextual(ve,ty);
end;

pragma no_memo(make_partial_base_type);

function partial_type_base_type(pt : PartialType)
    : PartialType
begin
  case pt begin
    match var_contextual(?ve,?ty) begin
      result := make_partial_base_type(ve,ty);
    end;
  else
```

```
    result := error_type;
  end;
end;


--- The engine
-- the following proedures do all the basic work of type checking

pattern collection_class(elem_formal : Declaration) : Declaration :=
    !READ_ONLY_COLLECTION_decl and
    class_decl(type_formals:={?elem_formal});

pattern collection_partial_sig(elem_formal : remote Declaration)
    : VarEnvironment
    := var_bound_rib(collection_class(?elem_formal),...),
       no_var(bound_rib(collection_class(?elem_formal),...));

[T :: TYPEABLE] begin
  procedure require_type(x : T; pt : PartialType) begin
    if not require_type_equal(x.partial_type,pt) then
      x.type_errors :> {"Type mismatch"};
    endif;
  end;
  var procedure require_sequence_type(x : T; pt : PartialType) : PartialType
  begin
    case pt.partial_type_sigs begin
      match {...,?ve=collection_partial_sig(?elem_formal),...} begin
        result := apply_and_simplify(ve,elem_formal);
      end;
    else
      result := error_type;
      x.type_errors :> {"not a sequence type"};
    end;
  end;
end;


-- require_type_equal:
-- If one of the two types is a type variable, bind it.
-- Otherwise, if if both are builtin types (functions or
-- internal list types) ensure that they are compatible structurally.
-- Otherwise, compare the base type declarations.
-- Note we can tell whether something is a type variable by checking
-- it's base declaration and seeing if it is a type formal bound to
-- a type in the var environment.
var procedure require_type_equal(t1,t2 : PartialType) : Boolean begin
  if t1 /= error_type and t2 /= error_type then
    pbt1 : PartialType := partial_type_base_type(t1);
    pbt2 : PartialType := partial_type_base_type(t2);
    tv1 : remote TypeVariable := detect_type_variable(pbt1);
```

```
  tv2 : remote TypeVariable := detect_type_variable(pbt2);
  if tv1 /= nil then
    if tv2 /= nil then
      tv1.chain :> {tv2};
      tv2.chain :> {tv1};
    else
      tv1.bindings :> {t2};
    endif;
    result := true;
  elsif tv2 /= nil then
    tv2.bindings :> {t1};
    result := true;
  else
    case pbt1 begin
      match var_contextual(?,no_type()) begin
        case pbt2 begin
          match var_contextual(?,no_type()) begin
            result := true;
          end;
        end;
      end;
      match var_contextual(?ve1,function_type(?fs1,?rds1))
      begin
        case pbt2 begin
          match var_contextual(?ve2,function_type(?fs2,?rds2))
          begin
            result :=
                length(fs1) = length(fs2) and
                (require_formal_type_equal(ve1,ve2,nth(i,fs1),nth(i,fs2))
                     for i in 0..(length(fs1)-1)) and
                length(rds1) = length(rds2) and
                (require_formal_type_equal(ve1,ve2,nth(i,rds1),nth(i,rds2))
                     for i in 0..(length(rds1)-1));
          end;
        end;
      end;
    else -- not a builtin type, require type eq'ness
      case pbt1 begin
        match var_contextual(?ve1,?ty) begin
          case pbt2 begin
            match var_contextual(?ve2,!ty) begin
              result := require_var_environ_equal(ve1,ve2);
            end;
          end;
        end;
      end;
    end;
    result := false; -- default
  endif;
else
```

```
    result := true;
  endif;
end;

var procedure require_formal_type_equal(ve1,ve2 : VarEnvironment;
                                        f1,f2 : remote Declaration) : Boolean
begin
  case f1 begin
    match formal(?,?t1) begin
      case f2 begin
        match formal(?,?t2) begin
          result := formal_same_shape_p(f1,f2) and
              require_type_equal(make_partial_type(ve1,t1),
                                 make_partial_type(ve2,t2));
        end;
      end;
    end;
    match value_decl((type):=?t1) begin
      case f2 begin
        match value_decl((type):=?t2) begin
           result :=
              require_type_equal(make_partial_type(ve1,t1),
                                 make_partial_type(ve2,t2));
        end;
      end;
    end;
  end;
  result := false;
end;

function detect_type_variable(pt : PartialType)
    : remote TypeVariable
begin
  case pt begin
    match var_contextual(var_rib(polymorphic(type_formals:=?tfs),
                                 variables:=?tvs),
                         type_use(?u)) begin
      case u.pre_contextual_def begin
        match contextual(?,?tf) if tf in tfs begin
          result := nth(position(tf,tfs),tvs);
        end;
      end;
    end;
  end;
  -- default
  result := nil;
end;

var procedure require_var_environ_equal(ve1,ve2 : VarEnvironment) : Boolean
    -- compare with environment_equalp in aps-type.aps
```

```
begin
  case ve1 begin
    match no_var(?e1) begin
      case ve2 begin
        match no_var(?e2) begin
          result := environment_equalp(e1,e2);
        end;
      else
        -- swap so as to have the no_var always in second place
        result := require_var_environ_equal(ve2,ve1);
      end;
    end;
    -- for classes and modules, name equivalence is used
    match var_bound_rib(?d1=some_class_decl(result_type:=?rd),...)
    begin
      case ve2 begin
        match some_bound_rib(?d2,...) begin
          result :=
              d1==d2 and
              require_type_equal(apply_and_simplify(ve1,rd),
                                 apply_and_simplify(ve2,rd));
        end;
      end;
    end;
    -- for poly's structural equivalence is used
    -- (this case should not appear unless some polymorphic uses
    --  were inferred in pre_contextual_def (currently not))
    match var_bound_rib(?d1=polymorphic(type_formals:=?tfs),next:=?n1) begin
      case ve2 begin
        match no_var(bound_rib(?d2,next:=?n2)) begin
          result := d1==d2 and
              (require_type_equal(apply_and_simplify(ve1,tf),
                                  apply_and_simplify(ve2,tf))
                   for tf in tfs) and
              require_var_environ_equal(n1,no_var(n2));
        end;
        match var_rib(...) begin
          -- swap to handle var rib as first always
          result := require_var_environ_equal(ve2,ve1);
        end;
        match var_bound_rib(?d2,next:=?n2) begin
          result :=
              d1==d2 and
              (require_type_equal(apply_and_simplify(ve1,tf),
                                  apply_and_simplify(ve2,tf))
                   for tf in tfs) and
              require_var_environ_equal(n1,n2);
        end;
      end;
    end;
```

```
    -- var rib's are always polys and thus we use structural equivalence
    match var_rib(?p1=polymorphic(type_formals:=?tfs),?,?,?tvs1,?n1) begin
      case ve2 begin
        match no_var(bound_rib(?p2,next:=?n2)) begin
          result := p1 == p2 and
              (assign_type_variable(nth(i,tvs1),
                                    apply_and_simplify(ve2,nth(i,tfs)))
                   for i in 0..(length(tvs1)-1)) and
              require_var_environ_equal(n1,no_var(n2));
        end;
         match var_bound_rib(?p2,next:=?n2) begin
          result := p1 == p2 and
              (assign_type_variable(nth(i,tvs1),
                                    apply_and_simplify(ve2,nth(i,tfs)))
                   for i in 0..(length(tvs1)-1)) and
              require_var_environ_equal(n1,n2);
        end;
        match var_rib(?p2,?,?,?tvs2,?n2) begin
          result := p1 == p2 and
              (chain_type_variables(nth(i,tvs1),nth(i,tvs2))
                   for i in 0..(length(tvs1)-1)) and
              require_var_environ_equal(n1,n2);
        end;
      end;
    end;
  end;
  result := false; -- default
end;
--
-- a convenient way to force type variables together.
var procedure chain_type_variables(tv1,tv2 : remote TypeVariable) : Boolean
begin
  tv1.chain :> {tv2};
  tv2.chain :> {tv1};
  result := true;
end;
--
-- likewise for making assignments
var procedure assign_type_variable(tv : remote TypeVariable;
                                   pt : PartialType) : Boolean
begin
  tv2 : remote TypeVariable := detect_type_variable(pt);
  if tv2 /= nil then
    result := chain_type_variables(tv,tv2);
  else
    tv.bindings :> {pt};
    result := true;
  endif;
end;
```

```
-- This procedure doesn't need to worry about fixed_sig's:
-- These are handled elsewhere.  We go through each required
-- signature and find it (or not) and then require the
-- appropriate var environments to be equal.  This specification
-- is correct because a type is allowed only one incarnation of
-- a named signature.
var procedure require_sig(pt : PartialType; req_sigs : PartialSigs)
    collection all_found : Boolean :> true, (and)
begin
  sigs : PartialSigs := partial_type_sigs(pt);
  for req in req_sigs begin
    case req begin
      match ?req_ve=partial_sig(?sd,?req_input,?req_var) begin
        case sigs begin
          match {...,?ve=partial_sig(!sd,?i,?v),...}
          begin
            all_found :>
                (i or not req_input) and
                (v or not req_var) and
                require_sig_types_equal(ve,req_ve);
          end;
        else
          all_found :> false;
        end;
      end;
    end;
  end;
end;

-- a useful function:
function var_environ_next(ve : VarEnvironment) : VarEnvironment begin
  case ve begin
    match no_var(some_rib(next:=?next)) begin
      result := no_var(next);
    end;
    match some_var_rib(next:=?next) begin
      result := next;
    end;
  end;
end;

-- require that the arguments are the same.
var procedure require_sig_types_equal(ve1,ve2 : VarEnvironment) : Boolean
begin
  case ve1 begin
    match some_bound_rib(?cd=some_class_decl(type_formals:=?tfs),...) begin
      result :=
          require_var_environ_equal(var_environ_next(ve1),
                                    var_environ_next(ve2)) and
```

```
          (require_type_equal(apply_and_simplify(ve1,tf),
                              apply_and_simplify(ve2,tf))
               for tf in tfs);
    end;
  end;
  -- otherwise undefined (partial_sigs must be some kind of bound_rib).
end;


--- Built In Types:

function make_partial_type_for_decl(decl : remote Declaration) : PartialType
begin
  if decl == nil then
    result := error_type;
  else
    result := make_simple_partial_type(decl.predefined_use);
  endif;
end;

boolean_partial_type : PartialType :=
    make_partial_type_for_decl(Boolean_decl);
integer_partial_type : PartialType :=
    make_partial_type_for_decl(Integer_decl);
real_partial_type : PartialType :=
    make_partial_type_for_decl(IEEEdouble_decl);
string_partial_type : PartialType :=
    make_partial_type_for_decl(String_decl);
char_partial_type : PartialType :=
    make_partial_type_for_decl(Character_decl);

void_partial_type : PartialType :=
    make_simple_partial_type(Void_type);


--- Declarations

pattern some_typed_decl((type) : Type) : Declaration
    := value_decl((type):=?(type)),
       attribute_decl((type):=?(type)),
       function_decl((type):=?(type)),
       procedure_decl((type):=?(type)),
       constructor_decl((type):=?(type)),
       pattern_decl((type):=?(type)),
       formal((type):=?(type));

-- handle a special case:
-- formals don't need types in some cases:
match ?d=formal((type):=no_type()) begin
  tv : TypeVariable := type_variable(nil,d);
```

```
  d.partial_type := var_partial_type(tv);
end;

match ?d=some_typed_decl((type):=?ty) begin
  d.partial_type := make_simple_partial_type(ty);
end;

[T :: {Expression,Pattern}, var PHYLUM[]] begin
  match ?d=renaming(?,?old:T) begin
    if d.depends_on_self then
      d.partial_type := error_type;
    else
      d.partial_type := old.partial_type;
    endif;
  end;
end;

-- declaration type checks:

match value_decl((type):=?ty,default:=?def) begin
  require_type(def,make_simple_partial_type(ty));
end;
match attribute_decl((type):=function_typing(?,?ty),
                     default:=?def)
begin
  require_type(def,make_simple_partial_type(ty));
end;

match pattern_decl((type):=function_typing(?formals,?ty),
                   choices:=choice_pattern({...,?pat,...}))
begin
  require_type(pat,make_simple_partial_type(ty));
  for i in 0..(length(formals)-1) begin
    formal : remote Declaration := nth(i,formals);
    pformal : remote Declaration := nth(i,pat.pattern_formals);
    case formal begin
      match formal(def(?name,...),?ty) begin
        if pformal == nil then
          pat.type_errors :> {"Formal " || name || " not bound"};
        elsif pformal == formal then
          pat.type_errors :> {"Formal " || name || " multiply bound"};
        elsif not contextual_type_matchp(pformal.computed_type,ty) then
          pformal.type_errors :> {"Does not match formal"};
        endif;
      end;
    end;
  end;
end;

[T :: {Expression, Pattern}, var PHYLUM[]] begin
```

```
  match replacement(?old:T,?new:T) begin
     require_type(new,old.partial_type);
  end;
end;


--- Defaults

match ?d=simple(?e) begin
  d.partial_type := e.partial_type;
end;
match ?d=composite(?e,?) begin
  d.partial_type := e.partial_type;
end;

-- this pattern used to cover more cases:
pattern combination_default(initial,func : Expression) : Default
    := composite(?initial,?func);

match combination_default(?e,?func) begin
  case partial_type_base_type(func.partial_type) begin
    match var_contextual(?ve,function_typing(?formals,?rt)) begin
      if length(formals) /= 2 then
          func.type_errors :> {"not a binary function"};
      endif;
      case e.partial_type begin
        match !error_type begin end;
        match ?pt begin
          for formal in formals begin
            if not require_type_equal(make_partial_contextual_type
                                          (ve,formal.computed_type),pt) then
              func.type_errors :>
                  {"function formal does not have required type"};
            endif;
          end;
          if not require_type_equal(make_partial_type(ve,rt),pt) then
            func.type_errors :>
                {"function result does not have required type"};
          endif;
        end;
      end;
    end;
  else
    func.type_errors :> {"not a function"};
  end;
end;



--- Uses
```

```
[phylum T :: TYPEABLE] begin
  attribute T.use_var_environ : VarEnvironment := empty_var_env;
  attribute T.use_fixed_environ : Environment := empty_env;
end;

[phylum T :: {Expression,Pattern}, var PHYLUM[]] begin
  pattern typed_use(u : Use) : T :=
      value_use(?u) :? T, pattern_use(?u) :? T;

  match ?x=typed_use(use(!underscore_symbol)) begin
    tv : TypeVariable := type_variable(nil,x);
    x.partial_type := var_partial_type(tv);
  end;
  match ?x=typed_use(?u:Use) begin
    case u.pre_contextual_def begin
      match contextual(?env,?decl) begin
        ve : VarEnvironment := environ2var_environ(x,env);
        x.use_var_environ := ve;
        -- pragma break();
        x.partial_type := make_partial_contextual_type
            (ve,decl.computed_type); -- NB: need fixed type here.
        x.use_fixed_environ := fix_var_environ(ve);
        u.contextual_def := contextual(x.use_fixed_environ,decl);
      end;
    end;
  end;
end;


--- Calls
-- we combine pattern calls and function calls into
-- a single analysis.

-- This analysis needs to handle both
-- variable arity functions (built in) and also
-- named or elided parameters (for patterns)

type DeclList := LIST[remote Declaration];

[phylum T :: {Expression,Pattern}, var PHYLUM[]] begin
  -- list of formals currently unsatisfied
  attribute T.formals_before : DeclList;
  attribute T.formals_after : DeclList;

  -- whether or not the arguments are matched by position
  -- (true until a named actual)
  attribute T.positional_before : Boolean;
  attribute T.positional_after : Boolean;
end;
```

```
[phylum T :: {Expression,Pattern}, var PHYLUM[];
 phylum L :: {Actuals,PatternActuals},SEQUENCE[T], var PHYLUM[]] begin

  pattern some_actual(actual : T; formal : Expression) : T
      := pattern_actual(?actual,?formal) :? T;

  -- return whether too few arguments:
  -- (so error message can be placed).
  var procedure match_formals_actuals(ve : VarEnvironment;
                                      formals : remote Declarations;
                                      actuals : L) : Boolean
  begin
    -- the formals remaining after processing the arguments
    remaining_formals : DeclList;

    (actuals...).formals_before, remaining_formals :=
        {formals...}, (actuals...).formals_after;

    -- whether no named parameters exist:
    all_positional : Boolean;

    (actuals...).positional_before, all_positional :=
        true, (actuals...).positional_after;

    for x in actuals begin
      -- first handle the case of a named formal
      case x begin
        match some_actual(?actual,?v=value_use(?u=use(?formal_name))) begin
          case x.formals_before begin
            match {...,?f=formal(def(!formal_name,...),?ty),...} begin
              u.contextual_def := contextual(fix_var_environ(ve),f);
              require_type(actual,make_partial_type(ve,ty));
              x.actual_formal := f;
              x.formals_after := {f1 if f1/=f for f1 in x.formals_before};
            end;
          else
            v.type_errors :> {"unknown formal"};
            x.formals_after := x.formals_before;
          end;
          x.positional_after := false;
        end;
      else

        -- Give error message if not positional
        if not x.positional_before then
          x.type_errors :> {"positional argument after named argument"};
        endif;
        x.positional_after := x.positional_before;
```

```
      -- handle case of a sequence formal:
      case x.formals_before begin
        match {?f=seq_formal(?,?ty)} begin
          require_type(x,make_partial_type(ve,ty));
          x.actual_formal := f;
          x.formals_after := x.formals_before;
        end;
      else

        -- handle case of rest things
        case x begin
          match rest_pattern(no_pattern()) :? T begin
            x.actual_formal := nil;
            x.formals_after := {};
          end;
          match rest_pattern(?) :? T begin
            x.type_errors :> {"rest patterns cannot be used here"};
            x.formals_after := {};
          end;
        else

          -- normal case
          case x.formals_before begin
            match {?f=formal(?,?ty),...} begin
              require_type(x,make_partial_type(ve,ty));
              x.actual_formal := f;
              x.formals_after := butfirst(x.formals_before);
            end;
          else
            x.type_errors :> {"too many arguments"};
            x.formals_after := {};
          end;

        end; -- case for rest
      end; -- case for sequence formals
    end; -- case for named formal
  end; -- for actuals

  case remaining_formals begin
    match {} begin end;
    match {seq_formal(...)} begin end;
  else
    result := not all_positional;
  end;
  result := true;
end; -- procedure match_formals_actuals
-- pragma inline(match_formals_actuals);

-- not useful outside this block:
pattern call(func : T; actuals : L) : T
```

```
      := pattern_call(?func,?actuals) :? T, funcall(?func,?actuals) :? T;
  match ?res=call(?func,?actuals) begin
    case partial_type_base_type(func.partial_type) begin
      match var_contextual(?ve,function_typing(?formals,?rtype)) begin
        res.partial_type := make_partial_type(ve,rtype);
        if not match_formals_actuals(ve,formals,actuals) then
          res.type_errors :> {"too few arguments"};
        endif;
      end;
    else
      res.partial_type := error_type;
      func.type_errors :> {"not of function type"};
    end;
  end; -- match call
end; -- end poly

match ?d=multi_call(?proc,?actuals,?results) begin
  case partial_type_base_type(proc.partial_type) begin
    match var_contextual(?ve,function_type(?formals,?result_decls)) begin
      if not match_formals_actuals(ve,formals,actuals) then
        d.type_errors :> {"too few arguments"};
      endif;
      if length(results) = length(result_decls) then
        for i in 0..(length(results)-1) begin
          case nth(i,result_decls) begin
            match value_decl((type):=?ty) begin
              require_type(nth(i,results),make_partial_type(ve,ty));
            end;
          end;
        end;
      else
        d.type_errors :> {"wrong number of results (expected " ||
                              length(result_decls) || ")"};
      endif;
    end;
  else
    proc.type_errors :> {"not of some functional type"};
  end;
end;

attribute Type.type_formals_before : DeclList;
attribute Type.type_formals_after : DeclList;

-- return whether too few arguments:
-- (so error message can be placed).
var procedure match_type_formals_actuals(env : Environment;
                                          formals : remote Declarations;
                                          actuals : TypeActuals)
    : Boolean
begin
```

```
  -- the formals remaining after processing the arguments
  remaining_type_formals : DeclList;

  (actuals...).type_formals_before, remaining_type_formals :=
      {formals...}, (actuals...).type_formals_after;

  result := length(remaining_type_formals) = 0;

  venv : VarEnvironment := no_var(env);

  for x in actuals begin
    -- get the formal:
    case x.type_formals_before begin
      match {} begin
        x.type_errors :> {"too many arguments"};
      end;
      match {?f=some_type_formal(?,?sig),...} begin
        x.actual_formal := f;
        x.type_formals_after := butfirst(x.type_formals_before);
        if not require_sig(var_contextual(no_var(x.environment),x),
                           merge_partial_sigs(venv,
                                              make_partial_sigs(f.sigs)))
        then
          x.type_errors :> {"type does not conform to signature"};
        endif;
        -- check limitation
        ct : ContextualType := contextual(x.environment,x);
        limited : ContextualTypeSet
            := merge_limited(env,sig.limited_types);
        if limited /= {} and not type_in_limited(ct,limited) then
          x.type_errors :>
              {"type actual does not fit limitation"};
        endif;
      end;
    end; -- case type_formals_before
  end; -- for actuals
end; -- procedure match_type_formals_actuals
-- pragma inline(match_type_formals_actuals);


--- Patterns

-- things other than calls:

match ?p=and_pattern(?p1,?p2) begin
  p.partial_type := p1.partial_type;
  require_type(p2,p1.partial_type);
end;

match ?p=condition(?cond) begin
```

```
  require_type(cond,boolean_partial_type);
  p.partial_type := error_type; -- OK since always second argument of an AND
end;

match ?p=pattern_var(?d) begin
  p.partial_type := d.partial_type;
end;

match ?p=rest_pattern(?r) begin
  p.partial_type := r.partial_type;
end;

match ?p=no_pattern() begin
  p.partial_type := error_type; -- i.e. anything
end;

-- occur only in later transformations:
match ?p=hole() begin
  p.partial_type := error_type;
end;
match ?p=pattern_function(...) begin
  p.partial_type := error_type;
end;


--- Statements

match effect(?e) begin
  require_type(e,void_partial_type);
end;

match assign(?lhs,?rhs) begin
  require_type(rhs,lhs.partial_type);
end;

match if_stmt(?cond,...) begin
  require_type(cond,boolean_partial_type);
end;

match for_in_stmt(?formal,?seq,...) begin
  require_type(formal,require_sequence_type(seq,seq.partial_type));
end;

pattern for_or_case_stmt(expr : Expression;
                         matchers : Matches) : Declaration
    := for_stmt(?expr,?matchers), case_stmt(?expr,?matchers,?);

match for_or_case_stmt(?expr,{...,matcher(?pat,?),...}) begin
  require_type(pat,make_var(expr.computed_type));
end;
```

```
--- Expressions

-- simple expressions:
match ?e=integer_const(...) begin
  e.partial_type := integer_partial_type;
end;
match ?e=real_const(...) begin
  e.partial_type := real_partial_type;
end;
match ?e=string_const(...) begin
  e.partial_type := string_partial_type;
end;
match ?e=char_const(...) begin
  e.partial_type := char_partial_type;
end;

match ?e=undefined() begin
  e.partial_type := error_type;
end;
match ?e=no_expr() begin
  e.partial_type := void_partial_type;
end;

match ?e1=typed_value(?e2,?ty) begin
  pt : PartialType := make_simple_partial_type(ty);
  require_type(e2,pt);
  e1.partial_type := pt;
end;
match ?e1=typed_pattern(?e2,?ty) begin
  pt : PartialType := make_simple_partial_type(ty);
  require_type(e2,pt);
  e1.partial_type := pt;
end;

match ?p1=match_pattern(?p2,?ty) begin
  if not contextual_type_matchp(p2.computed_type,ty) then
    p2.type_errors :> {"Pattern does not match type"};
  endif;
  p1.partial_type := make_simple_partial_type(ty);
end;

match ?e=append(?e1,?e2) begin
  e.partial_type := e1.partial_type;
  require_type(e2,e1.partial_type);
end;

match ?e=repeat(?seq) begin
  e.partial_type := require_sequence_type(seq,seq.partial_type);
```

```
end;

match ?e1=guarded(?e2,?cond) begin
  require_type(cond,boolean_partial_type);
  e1.partial_type := e2.partial_type;
end;

match ?e1=controlled(?e2,?formal,?seq) begin
  require_type(formal,require_sequence_type(seq,seq.partial_type));
  e1.partial_type := e2.partial_type;
end;

-- bogus values (for use only in pragmas)
pattern pragma_value() : Expression :=
    signature_value(...),type_value(...),pattern_value(...);

attribute Expression.used_in_pragma : Boolean := false;

match pragma_call(?,{...,?e,...}) begin
  e.used_in_pragma := true;
end;

match ?e=pragma_value(...) begin
  e.partial_type := error_type;
  if not e.used_in_pragma then
    e.type_errors :> {"legal only in a pragma"};
  endif;
end;


--- Type instantiation

-- We need to check signatures of type actuals and
-- the types of value actuals.  We also must check
-- uses of modules to make sure they are always instantiated.
-- We must also check that type instantiation only happens in
-- type declarations.

match ?ty=type_inst(?m,?tactuals,?actuals) begin
  case m.base_module begin
    match contextual(?env,?md=module_decl(type_formals:=?tfs,
                                          value_formals:=?vfs))
    begin
      -- if the types don't match up, there isn't much ability to continue
      if length(tfs) /= length(tactuals) then
        ty.type_errors :> {"wrong number of type parameters"};
      else
        built_env : Environment := type_inst_env(ty);
        _ := match_type_formals_actuals(built_env,tfs,tactuals);
        if not match_formals_actuals(no_var(built_env),vfs,actuals) then
```

```
        ty.type_errors :> {"too few value arguments"};
      endif;
    endif;
  end; -- match contextual base module
 end;
end;

pattern some_module_decl_or_renaming() : Declaration :=
    module_decl(...), module_renaming(...);

match ?mod=module_use(?u) begin
  case u.pre_contextual_def begin
    match contextual(?,some_module_decl_or_renaming(...)) begin end;
    match contextual(...) begin
      mod.type_errors :> {"not a module"};
    end;
  end;
end;

match ?ty=type_use(?u) begin
  case u.pre_contextual_def begin
    match contextual(?,some_module_decl_or_renaming(...)) begin
      ty.type_errors :> {"must be instantiated"};
    end;
  end;
end;

-- check where instantiation occurs
-- (legal only in type decls)

attribute Type.ok_to_instantiate_here : Boolean := false;

match some_type_decl((type):=?ty) begin
  ty.ok_to_instantiate_here := true;
end;

match inheritance(?,?ty,?) begin
  ty.ok_to_instantiate_here := true;
end;

match ?ty1=remote_type(?ty2) begin
  ty2.ok_to_instantiate_here := ty1.ok_to_instantiate_here;
end;

match ?ty1=type_inst(?,{...,?ty2,...},?) begin
  ty2.ok_to_instantiate_here := ty1.ok_to_instantiate_here;
end;

match ?ty1=private_type(?ty2) begin
  ty2.ok_to_instantiate_here := ty1.ok_to_instantiate_here;
```

```
end;

match ?ty=type_inst(...) begin
  if not ty.ok_to_instantiate_here then
    ty.type_errors :> {"cannot instantiate a module here"};
  endif;
end;


--- EXCESS

-- to enable this module to compile, we have to make the unspecific
-- guards nonstrict and then add new dependencies to ensure that
-- dependencies are correct.  Eventually these dependencies should be
-- added automatically.

pragma nonstrict_guards(module APS_TYPECHECK);

module CONTROL[] :: COMBINABLE[] := Boolean
begin
  initial : Result := false;
  combine : function (_,_:Result):Result := (or);
  [T] function depends_on(_:T) : Result := false;
  function use_control(_ : Result) : Boolean := false;
end;
type Control := CONTROL[];

[T] depends_on = (Control$depends_on : function(_:T):Control);
use_control = Control$use_control;


-- certain sets of type variables must be evaluated together

[phylum T :: TYPEABLE] begin
  collection attribute T.type_variables : TypeVariableSet;
  collection attribute T.partial_type_uses : Control;
  match ?tv=type_variable(?,?owner:T) begin
    owner.type_variables :> {tv};
  end;
  match ?x:T begin
    x.partial_type_uses :> depends_on(x.partial_type);
  end;
end;

signature EP_SEQUENCE :=
    {Expressions,Patterns,Actuals,PatternActuals}, var PHYLUM[];
[phylum T :: TYPEABLE,{Expression,Pattern,Default}] begin
  match ?p:T begin
    case p begin
      -- conditions may use pattern variables and so
```

```
      -- we have them link separately and don't pull in the
      -- uses from the condition into the pattern:
      match condition(?c) :? T begin
        link_type_variables(c.type_variables,c.partial_type_uses);
      end;
      -- controlled things infer a type for the sequence and formal
      -- together and once done, compute the type of the body.
      match controlled(?expr,?f,?seq) :? T begin
        f.type_variables :> seq.type_variables;
        link_type_variables(f.type_variables,seq.partial_type_uses);
        p.type_variables :> expr.type_variables;
        p.partial_type_uses :> expr.partial_type_uses;
      end;
    else
      [phylum U :: TYPEABLE,{Expression,Pattern,Declaration}]
      begin
        for p begin
          match parent(?c:U) begin
            p.type_variables :> c.type_variables;
            p.partial_type_uses :> c.partial_type_uses;
          end;
        end;
      end;
    end;
  end;
end;
[phylum T,U::TYPEABLE,{Expression,Pattern};
 L :: EP_SEQUENCE,SEQUENCE[U]] begin
  match ?p:T=parent(L${...,?c:U,...}) begin
    p.type_variables :> c.type_variables;
    p.partial_type_uses :> c.partial_type_uses;
  end;
end;

procedure link_type_variables(type_variables : TypeVariableSet;
                              control : Control)
begin
  pragma dynamic(control);
  for tv in type_variables begin
    if use_control(control) then
      -- NB always false
      -- but I'm playing it safe anyway:
      tv.bindings :> {};
      tv.chain :> {tv};
      tv.consistent :> true;
      tv.fits_signature :> true;
    endif;
  end;
end;
```

```
  -- everything but Pattern, Expression and Default
  signature EBPED := {Signature,Type,Program,Unit,Declaration,
                      Block,Match,Direction}, var PHYLUM[];

  [phylum T :: EBPED;
   phylum U :: TYPEABLE,{Expression,Pattern,Default}] begin
    match ?:T=parent(?c:U) begin
      link_type_variables(c.type_variables,c.partial_type_uses);
    end;
  end;
  [phylum T :: EBPED;
   phylum U :: TYPEABLE,{Expression,Pattern,Default};
   L :: EP_SEQUENCE,SEQUENCE[U]] begin
    match ?:T=parent(L${...,?c:U,...}) begin
      link_type_variables(c.type_variables,c.partial_type_uses);
    end;
  end;
end;
```

C.4 Expanding Inheritance

Inheritance inherits from the "no-op" copy transformation.

C.4.1 The "No-Op" Copy Transformation

This transformation creates a new tree with exactly the same structure as the one being attributed. It also decorates the new tree with name binding information.

```
module APS_NOP_COPY[Input :: var ABSTRACT_APS[],
                             var APS_ENVIRON[Input],
                             var APS_BOUND[Input]]
                             --var APS_TYPE[Input]]
    extends Input
begin
  -- This module is useless unless inherited:
  private;

  ---- COPY RECORD

  -- The copy records record what things we need to fix up remote references
  -- for.  In this file, we only need to copy contextual declarations and
  -- types, the environment change part is not currently used.
  -- If one needs more things done, they must be added in an inheriting module.

  type CopyRecords;

  constructor copy_records(environ : Environment;
                           remote_records : RemoteRecords) : CopyRecords;
```

```
type RemoteRecords := BAG[RemoteRecord];

type RemoteRecord;
constructor decl_remote_record(before : remote Declaration;
                               after : remote NewTree$Declaration)
    : RemoteRecord;
constructor type_remote_record(before : remote Type;
                               after : remote NewTree$Type)
    : RemoteRecord;

no_copy_records : CopyRecords := copy_records(empty_env,{});

function merge_copy_records(cr1,cr2 : CopyRecords) : CopyRecords begin
  case cr1 begin
    match copy_records(?e1,?) begin
      case cr2 begin
        match copy_records(?e2,?rr2) begin
          -- We ignore rr1 because we will only need to look at
          -- local copy records, and also sometimes rr1 = {}
          result := copy_records(merge_environ(e1,e2),rr2);
        end;
      end;
    end;
  end;
end;

function extend_env(env : Environment; cr : CopyRecords) : CopyRecords
begin
  case cr begin
    match copy_records(?sub,?rrs) begin
      result := copy_records(merge_environ(env,sub),rrs);
    end;
  end;
end;

function extend_remote_records(cr : CopyRecords; rrs : RemoteRecords)
    : CopyRecords
begin
  case cr begin
    match copy_records(?env,?rrs2) begin
      result := copy_records(env,{rrs...,rrs2...});
    end;
  end;
end;


--- Functions that use copy records.

-- first the environment is used in copy_Environment
```

```
function copy_Environment(env : Environment;
                          copy_records : CopyRecords)
    : NewTreeWithEnviron$Environment
begin
  case copy_records begin
    match copy_records(?outer,?) begin
      result := inherited_copy_Environment(merge_environ(outer,env),
                                           copy_records);
    end;
  end;
end;

-- (One function for each kind of thing recorded in a remote record.)

function copy_remote_Declaration(decl : remote Declaration;
                                 copy_records : CopyRecords)
    : remote NewTree$Declaration
begin
  case copy_records begin
    match copy_records(?,{...,decl_remote_record(!decl,?new),...}) begin
      result := new;
    end;
  else
    result := decl.copied_Declaration;
  end;
end;

function copy_remote_Type(ty : remote Type; copy_records : CopyRecords)
    : remote NewTree$Type
begin
  case copy_records begin
    match copy_records(?,{...,type_remote_record(!ty,?new),...}) begin
      result := new;
    end;
  else
    result := ty.copied_Type;
  end;
end;


--- Recording copies:

-- after a copy, we need to record the original->copy binding.

[phylum T :: PHYLUM[]] begin
  function new_copy_records(new_thing : remote T) : CopyRecords
  begin
    collection remote_records : RemoteRecords;
    for new_thing begin
```

```
      match ancestor(?d:NewTree$Declaration) begin
        remote_records :> {decl_remote_record(d.original_Declaration,d)};
      end;
      match ancestor(?ty:NewTree$Type) begin
        remote_records :> {type_remote_record(ty.original_Type,ty)};
      end;
    end;
    result := copy_records(empty_env,remote_records);
  end;
end;


--- Copy Attributes

-- depends on self can be copied directly to the new tree:
match ?new:NewTree$Declaration begin
  new.FinishedTree$depends_on_self :=
      new.original_Declaration.depends_on_self;
end;

-- copy the contextual def
match ?use:Use begin
  use.copied_Use.FinishedTree$contextual_def :=
      copy_ContextualDeclaration(use.contextual_def,
                                 local_use_copy_records(use));
end;

-- overridden when the use is being moved.
function local_use_copy_records(u : remote Use) : CopyRecords
    := no_copy_records;

procedure copy_Use(u : remote Use; copy_records : CopyRecords)
    : NewTree$Use
begin
  new_use : NewTree$Use := inherited_copy_Use(u, copy_records);
  new_use.FinishedTree$contextual_def :=
      copy_ContextualDeclaration(u.contextual_def,copy_records);
end;

inherit COPY_ABSTRACT_APS[Input,CopyRecords] begin
  type NewTree = Copy;
  type NewUnits = Copy$Units;
  type NewDeclarations = Copy$Declarations;
  type NewActuals = Copy$Actuals;
  type NewPatternActuals = Copy$PatternActuals;

  -- copied attributes:
  var copied_Program = copied_Program;
  var copied_Unit = copied_Unit;
  var copied_Declaration = copied_Declaration;
```

```
    var copied_Expression = copied_Expression;
    var copied_Pattern = copied_Pattern;
    var copied_Type = copied_Type;
    var copied_Use = copied_Use;
    var copied_Units = copied_Units;
    var copied_Declarations = copied_Declarations;

    -- original attributes:
    var original_Declaration = original_Declaration;
    var original_Type = original_Type;

    -- copy procedures
    copy_Unit = copy_Unit;
    copy_Declaration = copy_Declaration;
    copy_Type = copy_Type;
    copy_Expression = copy_Expression;
    copy_Pattern = copy_Pattern;
    copy_Units = copy_Units;
    copy_Declarations = copy_Declarations;
    copy_Use -> copy_Use;
    var inherited_copy_Use = copy_Use;
  end;

  inherit APS_COPY_ENVIRON[Input,NewTree,CopyRecords] begin
    var type NewTreeWithEnviron = CopyWithEnviron;

    var inherited_copy_Environment = copy_Environment;
    copy_ContextualDeclaration = copy_ContextualDeclaration;
    copy_Environment -> copy_Environment;
    copy_remote_Declaration -> copy_remote_Declaration;
    copy_remote_Type -> copy_remote_Type;
  end;

  public var type FinishedTree := APS_BOUND[NewTreeWithEnviron];
end;
```

Inheritance is implemented using a procedure because an inherited module may recursively inherit another module.

```
module APS_EXPAND_INHERIT[Input :: var ABSTRACT_APS[],
                                   var APS_ENVIRON[Input],
                                   var APS_TYPE[Input],
                                   var APS_BOUND[Input],
                                   var APS_RENAME[Input]]
    extends Input
begin
  var type Expanded :: var ABSTRACT_APS[], var APS_ENVIRON[Expanded],
                       var input APS_BOUND[Expanded]
      := private FinishedTree;

  type Errors := BAG[String];
```

```
collection attribute Declaration.inherit_errors : Errors;

private ;


--- Mappings

constructor inheritance_copy_records
    (env : Environment;
     remote_records : RemoteRecords;
     replacement_records : ReplacementRecords) : CopyRecords;

type ReplacementRecords := BAG[ReplacementRecord];

type ReplacementRecord;

-- to perform the inheritance, we need to keep track of
-- replacements (both explicit and those implied by the
-- type and value actuals to the type instance)
-- as well as the copy records.  We have to worry about four
-- kinds of replacements: one for each namespace
-- (and note that the value replacement should not affect lvalues).

constructor value_replacement_record(before,after : ContextualDeclaration)
    : ReplacementRecord;
constructor pattern_replacement_record(before,after : ContextualDeclaration)
    : ReplacementRecord;
constructor type_replacement_record(before,after : ContextualDeclaration)
    : ReplacementRecord;
constructor signature_replacement_record(before,after:ContextualDeclaration)
    : ReplacementRecord;

pattern some_replacement_record(before,after : ContextualDeclaration)
    : ReplacementRecord :=
    value_replacement_record(?before,?after),
    pattern_replacement_record(?before,?after),
    type_replacement_record(?before,?after),
    signature_replacement_record(?before,?after);

-- factoring device:
function replacement_record_matchp(rr1,rr2 : Replacement) : Boolean begin
  case ReplacementRecords${rr1,rr2} begin
    match {... and value_replacement_record(...)} begin
      result := true;
    end;
    match {... and pattern_replacement_record(...)} begin
      result := true;
    end;
    match {... and type_replacement_record(...)} begin
      result := true;
```

```
    end;
    match {... and signature_replacement_record(...)} begin
      result := true;
    end;
  end;
end;
-- yet another factoring device:
function make_replacement_record(before,after : ContextualDeclaration;
                                 model : ReplacementRecord)
    : ReplacementRecord
begin
  case model begin
    match value_replacement_record(...) begin
      result := value_replacement_record(before,after);
    end;
    match pattern_replacement_record(...) begin
      result := pattern_replacement_record(before,after);
    end;
    match type_replacement_record(...) begin
      result := type_replacement_record(before,after);
    end;
    match signature_replacement_record(...) begin
      result := signature_replacement_record(before,after);
    end;
  end;
end;

pattern some_copy_records(environ : Environment;
                          remote_records : RemoteRecords) : CopyRecords
    := copy_records(?environ,?remote_records),
       inheritance_copy_records(?environ,?remote_records,...);

function merge_copy_records(cr1,cr2 : CopyRecords) : CopyRecords begin
  case cr1 begin
    match inheritance_copy_records(?e1,?,?rps1) begin
      case cr2 begin
        match inheritance_copy_records(?,?rms2,?rps2) begin
          --environments never used for inheritance_copy_records
          result := inheritance_copy_records
              (e1,rms2,
               {rps1...,merge_replacement_record(rps1,rp2) for rp2 in rps2});
        end;
        match copy_records(?,?rms2) begin
          result := inheritance_copy_records(e1,rms2,rps1);
        end;
      end;
    end;
    match copy_records(?e1,?) begin
      case cr2 begin
        match inheritance_copy_records(?,?rms2,?rps2) begin
```

```
          result := inheritance_copy_records(e1,rms2,rps2);
        end;
        match copy_records(?,?rms2) begin
          result := copy_records(e1,rms2);
        end;
      end;
    end;
  end;
end;
function merge_replacement_record(rps : ReplacementRecords;
                                  rr : ReplacementRecord)
    : ReplacementRecord
begin
  case rr begin
    -- needs factoring:
    match some_replacement_record(?mid,?after) begin
      case rps begin
        match {...,?rr2=some_replacement_record
                   (?before,?new if contextual_decl_equalp(mid,new))
                   if replacement_record_matchp(rr,rr2),...}
        begin
          result := make_replacement_record(before,new,rr);
        end;
      end;
    end;
  else
    result := rr;
  end;
end;

function contextual_decl_equalp(cd1,cd2 : ContextualDeclaration) : Boolean
begin
  case cd1 begin
    match contextual(?e1,?decl) begin
      case cd2 begin
        match contextual(?e2,!decl) begin
          result := environment_equalp(e1,e2);
        end;
      end;
    end;
  end;
  result := false;
end;


--(NB: Would be shorter if we had an expression 'match' predicate)
function value_replacement_record_p(cr : ReplacementRecord) : Boolean := false begin
  case cr begin
    match value_replacement_record(...) begin result := true; end;
  end;
```

```
end;
function pattern_replacement_record_p(cr : ReplacementRecord) : Boolean := false
begin
  case cr begin
    match pattern_replacement_record(...) begin result := true; end;
  end;
end;
function type_replacement_record_p(cr : ReplacementRecord) : Boolean := false begin
  case cr begin
    match type_replacement_record(...) begin result := true; end;
  end;
end;
function signature_replacement_record_p(cr : ReplacementRecord) : Boolean := false
begin
  case cr begin
    match signature_replacement_record(...) begin result := true; end;
  end;
end;
function no_replacement_record_p(_ : ReplacementRecord) : Boolean := false;


-- The semantics of replacement is that Contextual Declarations are
-- looked up once to see if there is a replacement and once again
--(NB: This would be easier if we could pass patterns as parameters)

function convert_ContextualDeclaration
    (cd : ContextualDeclaration;
     copy_records : CopyRecords;
     replacement_record_match : function(_:CopyRecord) : Boolean)
    : NewTreeWithEnviron$ContextualDeclaration
begin
  replaced : ContextualDeclaration := cd;
  case copy_records begin
    match {...,?rr=some_replacement_record(?before,?after)
               if contextual_declaration_equalp(before,cd)
               and replacement_record_match(rr),...} begin
      replaced := after;
    end;
  end;
  --?? Note on old version of copy_ContextualDeclaration
  --?? I don't understand fully.  It may be relevant here:
  --:: we have to clean up the environment because it
  --:: may include inherited bound environments from *other*
  --:: inheritances!
  result := copy_ContextualDeclaration(replaced,copy_records);
end;

[Base; T :: CONTEXTUAL[Base]] begin
  function remove_inheritance(x : T) : T begin
    case x begin
```

```
      match contextual(?environ,?thing) begin
        result := contextual(remove_inheritance_from_environ(environ),
                             thing);
      end;
    else
      result := x;
    end;
  end;
end;

function remove_inheritance_from_environ(e : Environment) : Environment
begin
  case e begin
    match bound_rib(?decl,contextual(?,inheritance(...)),?,?,?,?next) begin
      result := unbound_rib(decl,decl.environment);
    end;
    match bound_rib(?decl,?ra,?i,?v,?tas,?next) begin
      result := bound_rib(decl,remove_inheritance(ra),i,v,
                          {remove_inheritanceta(ta) for ta in tas},
                          remove_inheritance_from_environ(next));
    end;
  else
    result := e;
  end;
end;


--- Performing Inheritace

-- Most nodes need a inherited_copy_records
-- attribute to keep track of surrounding inherit blocks
-- For most nodes in the tree (those not inside an inherit block),
-- this attribute will be empty:

[phylum T :: PHYLA]
    attribute T.inherited_copy_records : CopyRecords := no_copy_records;


-- empty for most declarations,
-- one element for replacements and
-- large for inheritance "declarations"
attribute Declaration.inheritance_replacements : ReplacementRecords := {};

-- caching useful information
attribute Declaration.inheritance_module : remote Declaration := nil;

-- we have to assign newTree$Declarations to a type so we can use it
type NewDeclarations :: SEQUENCE[NewTree$Declaration] :=
    NewTree$Declarations;
```

```
-- This attribution clause overrides one inherited from
-- APS_TREE_COPY via APS_NOP_COPY:

match ?i=inheritance(?ty=type_inst(type_use(?u),?tactuals,?vactuals),
                     ?b=block(?decls))
begin
  -- override default copy to avoid using trees twice
  b.copied_Block := NewTree$block({});        -- ignored
  ty.copied_Type := NewTree$no_type();        -- ignored

  case u.expanded_def begin
    match contextual(?env,?md=module_decl(type_formals:=?tfs,
                                          value_formals:=?vfs,
                                          result_decl:=?rtd,
                                          precontents:=block(?pdecls),
                                          contents:=?mb=block(?idecls)))
        if not md.depends_on_self  -- avoid inheriting circular modules
        if length(tfs) = length(tactuals) -- don't inherit if the ...
        if length(vfs) = length(vactuals) -- ... instantiation is bad
    begin
      unbound_env : Environment := mb.environment;

      -- in order to make inheritance of modules with inheritance
      -- easier, we cache two useful attributes:

      replacements : ReplacementRecords :=
          {decl.inheritance_replacements... for decl in decls,
           type_replacement_record(contextual(unbound_env,nth(i,tfs)),
                                   as_contextual_decl(nth(i,tactuals)))
               for i in 0..(length(tfs)-1),
           value_replacement_record(contextual(unbound_env,nth(i,vfs)),
                                    as_contextual_decl(nth(i,vactuals)))
                for i in 0..(length(vfs)-1)};

      copy_records : CopyRecords :=
          merge_copy_records
          (i.inherited_copy_records,
           inheritance_copy_records
               (i.environment,new_copy_records
                    -- NB: note circular dependence broken
                    -- by procedure parameter
                    (i.new_Declarations,replacements)));

      i.inheritance_replacements := replacements;
      b.inherited_copy_records := copy_records;
      i.inherited_module := md;

      new_env : Environment :=
          bound_rib(md,contextual(i.environment,i.predefined_use),true,true,
```

```
                    {contextual(t.environment,t) for t in tactuals},
                    env);

      -- the inheritance decl is replaced
      i.copied_Declaration :=
          copy_Declaration(rtd,copy_records);
      i.new_Declarations :=
          {i.copied_Declaration,
           copy_Declarations(pdecls,copy_records)...,
           decls.copied_Declarations...,
           copy_Declarations(idecls,copy_records)...};
    end;
  else -- bad inheritance, just replace with the body:
    i.copied_Declaration := NewTree$no_decl();
    i.new_Declarations := decls.copied_Declarations;
    i.inherit_errors :> {"bad inheritance"};
  end;
end;

[phylum T :: {Signature,Type,Expression,Pattern}] begin
  pattern with_use(u : Use) : T :=
      sig_use(?u), qual_sig(sig_use(?u)),
      type_use(?u), qual_type(?,type_use(?u)),
      pattern_use(?u), qual_pattern(?,pattern_use(?u)),
      value_use(?u), qual_value(?,value_use(?u)),
      typed_pattern(with_use(?u),?), typed_value(with_use(?u),?);

  function as_contextual_decl(x : T) : ContextualDeclaration begin
    case x begin
      match with_use(?u) begin
        result := u.contextual_def;
      end;
    else
      result := not_found;
    end;
  end;
end;

match ?d=value_replacement(with_use(?ufrom),with_use(?uto)) begin
  d.replacements :=
      {value_replacement_record(remove_inheritance(ufrom.contextual_def),
                                uto.contextual_def)};
end;
match ?d=pattern_replacement(with_use(?ufrom),with_use(?uto)) begin
  d.replacements :=
      {pattern_replacement_record(remove_inheritance(ufrom.contextual_def),
                                  uto.contextual_def)};
end;
match ?d=type_replacement(with_use(?ufrom),with_use(?uto)) begin
  d.replacements :=
```

```
      {type_replacement_record(remove_inheritance(ufrom.contextual_def),
                               uto.contextual_def)};
end;
match ?d=signature_replacement(with_use(?ufrom),with_use(?uto)) begin
  d.replacements :=
      {signature_replacement_record(remove_inheritance(ufrom.contextual_def),
                                    uto.contextual_def)};
end;


-- transfer inherited_copy_records down:
[phylum P,C :: PHYLA] begin
  match ?p:P=parent(?c:C) begin
    c.inheritance_copy_records := p.inheritance_copy_records;
  end;
end;


--- Uses

-- we need to convert uses according to the proper namespace

attribute Use.replacement_record_predicate
    : function(_:CopyRecord) : Boolean
    := no_replacement_record_p;

match ?e=value_use(?u) begin
  if e.lvaluep then
    u.replacement_record_predicate := no_replacement_record_p;
  else
    u.replacement_record_predicate := value_replacement_record_p;
  endif;
end;
match pattern_use(?u) begin
  u.replacement_record_predicate := pattern_replacement_record_p;
end;
match type_use(?u) begin
  u.replacement_record_predicate := type_replacement_record_p;
end;
match signature_use(?u) begin
  u.replacement_record_predicate := signature_replacement_record_p;
end;
match module_use(?u) begin
  u.replacement_record_predicate := type_replacement_record_p;
end;
match class_use(?u) begin
  u.replacement_record_predicate := signature_replacement_record_p;
end;

-- copy the contextual def
```

```
match ?use:Use begin
  use.copied_Use.FinishedTree$contextual_def :=
      copy_ContextualDeclaration(use.contextual_def,
                                 use.inheritance_copy_records,
                                 u.replacement_record_predicate);
end;

procedure copy_Use(u : remote Use; copy_records : CopyRecords)
    : NewTree$Use
begin
  new_use : NewTree$Use := inherited_copy_Use(u, copy_records);
  new_use.FinishedTree$contextual_def :=
      convent_ContextualDeclaration(u.contextual_def,copy_records,
                                   u.replacement_record_predicate);
end;


--- Sequence flattening

-- when we copy Declarations (or Units), we flatten the result:

match ?decls=Declarations$single(?decl) begin
  case decl begin
    match inheritance(...) begin
      decls.copied_Declarations := copy_Declarations(decls);
    end;
  else
    decls.copied_Declarations := decl.new_Declarations;
  end;
end;

procedure copy_Declarations(decls : remote Declarations;
                            copy_records : CopyRecords)
    : NewDeclarations
begin
  case decls begin
    match Declarations$single(?decl) begin
      case decl begin
        match inheritance(?,block(?decls)) begin
          case decl.inheritance_module begin
            match module_decl(result_decl:=?rdecl,
                              contents:=block(?idecls))
            begin
              --Q: What about decl.inheritance_copy_records ?
              --A: We recreate them as we traverse the structure.
              new_copy_records :=
                  merge_replacement_records(copy_records,
                                            decl.inheritance_replacements);
              result := NewDeclarations$
                  {copy_Declarations(rdecls,new_copy_records)...,
```

```
                    copy_Declarations(decls,copy_records)...,
                    copy_Declarations(idecls,new_copy_records)...};
            end;
          else
            result := NewDeclarations${};
          end;
        end;
        match no_decl() begin
          result := NewDeclarations${};
        end;
        match some_replacement() begin
          result := NewDeclarations${};
        end;
      else
        result := NewDeclarations${copy_Declaration(decl,copy_records)};
      end;
    end;
  else
    result := inherited_copy_Declarations(decls,copy_records);
  end;
end;


-- we have to ensure that in places where copied_Declaration is used
-- that we turn off the new_Declarations attribute:

match ?md=module_decl(result_type:=?result_decl) begin
  result_decl.new_Declarations := {};
end;

match ?sd=signature_decl(result_type:=?result_decl) begin
  result_decl.new_Declarations := {};
end;

match ?ft=function_type(?formals,?rd=value_decl(...)) begin
  rd.new_Declarations := NewDeclarations${};
end;

match ?e=controlled(formal:=?formal) begin
  formal.new_Declarations := NewDeclarations${};
end;


-- To allow a Declaration to generate multiple decls in its copy,
-- we define new attributes coped_Declarations and copied_Units
-- Parts of these declarations must override the definitions inherited
-- below and so must come before the inherit.

attribute Declaration.new_Declarations : NewDeclarations;
```

```
match ?d=no_decl() begin
  d.new_Declarations := NewDeclarations${};
end;

match ?d=some_replacement() begin
  d.new_Declarations := NewDeclarations${};
end;

-- otherwise, just grab the (local) copy
match ?d:Declaration begin
  d.new_Declarations := NewDeclarations${d.copied_Declaration};
end;


-- and we have to do something similar with units too:

type NewUnits :: SEQUENCE[NewTree$Unit] := NewTree$Units;

attribute Unit.New_Units : NewUnits;

match ?u=no_unit() begin
  u.new_Units := NewUnits${};
end;

match ?u=decl_unit(?d) begin
  u.new_Units := NewUnits${NewTree$decl_unit(d)
                               for d in d.new_Declarations};
end;

match ?u:Unit begin
  u.new_Units := NewUnits${u.copied_Unit};
end;

match ?s=Units$single(?u) begin
  s.copied_Units := u.new_Units;
end;


--- Inheritance using inheritance

-- these are right at the end so all matches in this module override
-- the ones here:
inherit APS_NOP_COPY[Input] begin
  type NewTree = NewTree;
  type NewTreeWithEnviron = NewTreeWithEnviron;
  type FinishedTree = FinishedTree;

  -- sequence types
  type NewDeclarations = NewDeclarations;
  type NewUnits = NewUnits;
```

```
    -- copied attributes:
    copied_Unit = copied_Unit;
    copied_Declaration = copied_Declaration;
    copied_Block = copied_Block;
    copied_Signature = copied_Signature;
    copied_Type = copied_Type;
    copied_Use = copied_Use;
    copied_Units = copied_Units;
    copied_Declarations = copied_Declarations;

    -- copy functions
    copy_Declaration = copy_Declaration;
    copy_Declarations -> copy_Declarations;
    inherited_copy_declarations = copy_Declarations;

    -- original attributes:
    original_Declaration = original_Declaration;
    original_Type = original_Type;

    -- CopyRecord declarations:
    type CopyRecords = CopyRecords;
    type RemoteRecords = RemoteRecords;
    copy_records = copy_records;
    pattern copy_records = copy_records;
    pattern copy_records -> some_copy_records;
    merge_copy_records -> merge_copy_records;
    [phylum T :: PHYLUM[]]
        new_copy_records =
        (new_copy_records : function(_:remote T):CopyRecords);

    -- miscellaneous:
    copy_ContextualDeclaration = copy_ContextualDeclaration;
    copy_remote_Declaration = copy_remote_Declaration;
    copy_remote_Type = copy_remote_Type;
    local_use_copy_records -> inheritance_copy_records;
    copy_Use -> copy_Use;
  end;
end;
```

C.5 Pattern Matching

C.5.1 Computing Pattern Lookaheads

```
module APS_LOOK_AHEAD[Input :: var APS_TREE[],
                               var APS_PATTERN[Input],
                               var APS_ENVIRON[Input],
                               var APS_BOUND[Input]]
    extends Input
begin
```

```
type Constructors := ORDERED_SET[remote Declaration]((==), (<<));
type ConstructorsLattice := UNION_LATTICE[remote Declaration,Constructors];

type Position = PAIR[remote Declaration,Integer];
function position_equal(p1,p2 : Position) : Boolean begin
  case {p1,p2} begin
    match {pair(?c,?i),pair(!c,!i)} begin
      result := true;
    end;
  else
    result := false;
  end;
end;
function position_less(p1,p2 : Position) : Boolean begin
  case {p1,p2} begin
    match {pair(?c,?i1),pair(!c,?i2)} begin
      result := i1 < i2;
    end;
    match {pair(?c1,?),pair(?c2,?)} begin
      result := c1 << c2;
    end;
  end;
end;
type Positions := ORDERED_SET[Position]((=),position_less);
type PositionsLattice := UNION_LATTICE[Position,Positions];

attribute (p : Pattern).constructors : ConstructorsLattice
    := p.first_constructors;
circular attribute Pattern.positions : PositionsLattice;

collection attribute Declaration.phylum_constructors : Constructors := {};
collection attribute Declaration.phylum_positions : Positions := {};

private;

-- collect all the constructors for a phylum:
match ?cd=constructor_decl(...) begin
  cd.phylum_for_constructor :> {cd};
end;

-- collect all the positions a phylum may appear in:
attribute Formal.formal_index : Integer := 0;
match ?cd=constructor_decl(as_value:=a_value((type):=
                                                function_type(?formals,?)))
begin
  formal.formal_index for formal:Formal in formals,_ :=
      1,formals.formal_index+1 for formal:Formal in formals;
  for formal:Formal in formals begin
    case formal begin
      match a_value((type):=?ty) begin
```

```
        case ty.type_as_phylum begin
          match ?pd=phylum_decl(...) begin
            pd.phylum_positions :> {pair(cd,formal.formal_index)};
          end;
        end;
      end;
    end;
  end;
end;

-- First we compute bottom-up the constructors for each pattern
-- (The attribute must be circular because of pattern definition recursion).
-- Later and_patterns are used to cut down on the constructors for
-- each pattern particpating in an and_pattern.
circular attribute Pattern.first_constructors : ConstructorsLattice;

-- the rules for and_patterns and choice_patterns are analagous,
-- buit and_pattern has a rule for the final constructors set:

match ?p=and_pattern(?p1,?p2) begin
  p.first_constructors :=  -- |/\| here means set intersection
      p1.first_constructors |/\| p2.first_constructors;
  p1.positions :> p.positions;
  p2.positions :> p.positions;
  -- cut down final constructors result:
  p1.constructors := p.constructors;
  p2.constructors := p.constructors;
end;

match ?p=choice_pattern(?p1,?p2) begin
  p.first_constructors :=
      p1.first_constructors |\/| p2.first_constructors;
  p1.positions := p.positions;
  p2.positions := p.positions;
end;

match ?p=simple_pattern(?pf,?actuals) begin
  case pf begin
    -- for pattern calls:
    match pattern_function(?formals,?body) begin
      p.first_constructors := body.first_constructors;
      -- we collect up all the positions for the body ...
      circular collection body_positions : PositionsLattice;
      -- ... and then assign them:
      body.positions := body_positions;

      body_positions :> p.positions;
      for body begin
        match pattern_scope(?holep=hole()) begin
          holep.first_constructors := body.first_constructors;
```

```
          body_positions :> holep.positions;
        end;
      end;
      for arg:Pattern in actuals begin
        -- similarly, for the arguments, we get a collection:
        circular collection arg_positions : Positions;
        circular collection arg_constructors : Constructors;
        arg.positions := arg_positions;
        arg.constructors := arg_constructors;
        for body begin
          match pattern_scope(?pv=pattern_variable(...)
                                if pv.binding_formal == arg.pattern_formal)
          begin
            pv.first_constructors := arg.first_constructors;
            arg_positions :> pv.positions;
            arg_constructors :> pv.constructors;
          end;
        end;
      end;
    end;
    -- for constructor calls:
  else
    case pf.pattern_declaration.pattern_as_constructor begin
      match ?cd=constructor_decl(...) begin
        p.first_constructors :> {cd};
        for arg:Pattern in actuals begin
          case arg.pattern_formal begin
            -- make sure not an error (nil) and ...
            match ?formal=a_value((type):=?ty) begin
              case ty.type_as_phylum begin
                -- ... make sure a child field:
                match ?pd=phylum_decl(...) begin
                  arg.positions :>
                      {pair(cd,formal.formal_index)};
                end;
              end;
            end;
          end; -- case arg.pattern_formal
        end; -- for arg in args
      end; -- match constructor_decl
    end; -- case constructor for pf
  end; -- case pf
end;

-- We need to have a special case for when further pattern matching is done
-- on a pattern variable for which we already have positions:
pattern pattern_match_stmt(expr : Expression; matchers : Matches) : Statement
    = for_stmt(?expr,?matchers),case_stmt(?expr,?matchers,?);
match pattern_match_stmt(?e=value_name(?),{...,matcher(pat:=?pat),...})
begin
```

```
    case e.value_binding.as_pattern_var begin
      match ?p=pattern_var(?) begin
        pat.positions := p.positions;
      end;
    end;
  end;

  -- handle pattern vars, semantic conditions for constructors
  -- handle top-level matches for positions:
  -- NB: lexical ordering makes these definitions defaults:
  match ?p:Pattern begin
    case p.pattern_type.type_as_phylum begin
      match ?pd=phylum_decl(...) begin
        p.first_constructors := pd.phylum_constructors;
        p.positions := pd.phylum_positions;
      end;
    end;
  end;

end;
```

C.5.2 Choosing a Direction

```
-- Specify a direction for each attribute, synthesized (or inherited)

module APS_DIRECTION[Input :: var APS_TREE[]] extends Input
begin
  input attribute Declaration.synthesized : Boolean := true;
end;
```

```
-- Choose a direction for each attribute, synthesized or inherited

module APS_DIRECTION[Input :: var APS_TREE[],
                              var APS_ENIVRON[Input],
                              var APS_BOUND[Input],
                              var APS_LOOK_AHEAD[Input]]
    :: var APS_DIRECTED[Input]
    extends Input
begin
  attribute Declaration.synthesized : Boolean;

  private;

  constant inherited_symbol : Symbol := make_symbol("inherited");
  constant synthesized_symbol : Symbol := make_symbol("synthesized");

  collection attribute pragma_specified_inherited : Boolean := false, (or);
  collection attribute pragma_specified_synthesized : Boolean := false, (or);

  -- The pragma takes precedence over any calculations:
```

```
match ad=attribute_decl(...) begin
  if ad.pragma_specified_synthesized then
    -- (Really, we should check that the attribute isn't specified
    -- as both inherited and synthesized)
    ad.synthesized := true;
  elsif ad.pragma_specified_inherited then
    ad.synthesized := false;
  endif;
end;

match pragma_decl(?sym,{?v=value_name(...)}) begin
  case v.value_as_attribute begin
    match ?ad=attribute_decl(...) begin
      case sym begin
        match !inherited_symbol begin
          ad.pragma_specified_inherited :> true;
        end;
        match !synthesized_symbol begin
          ad.pragma_specified_synthesized :> true;
        end;
      end;
    end;
  end;
end;

-- we make a count of how many equations there will be if the
-- attribute is synthesized versus inherited.  The smaller set wins:
collection attribute Declaration.num_if_synthesized : Integer := 0, (+);
collection attribute Declaration.num_if_inherited : Integer := 0, (+);

match ad=attribute_decl(...) begin
  ad.synthesized := ad.num_if_synthesized <= ad.num_if_inherited;
end;

attribute Declaration.unknown_synthesized : Integer;
attribute Declaration.unknown_inherited : Integer;
match ?ad=attribute_decl(...) begin
  case ad.phylum_for_attribute begin
    match ?pd=phylum_decl(...) begin
      ad.unknown_synthesized := length(pd.phylum_constructors);
      ad.unknown_inherited := length(pd.phylum_positions);
    end;
  end;
end;

pattern attr_ref(node : Expression; a : Expression) : Expression
    = funcall(?a,{?node}) if a.value_binding.as_attribute /= nil;
match assign_stmt(attr_ref(?a,?node),?) begin
  case a.value_binding.as_attribute begin
    match ?ad=attribute_decl(...) begin
```

```
        local num_syn : Integer := ad.unknown_synthesized;
        local num_inh : Integer := ad.unknown_inherited;
        case node.value_binding begin
          match ?vd=value_decl(...) begin
            case vd.value_decl_as_pattern_var begin
              match ?pv=pattern_var(...) begin
                num_syn := length(pv.constructors);
                num_inh := length(pv.positions);
              end;
            end;
          end;
        end;
        ad.num_if_synthesized :> num_syn;
        ad.num_if_inherited :> num_inh;
      end;
    end;
  end;
end;
```