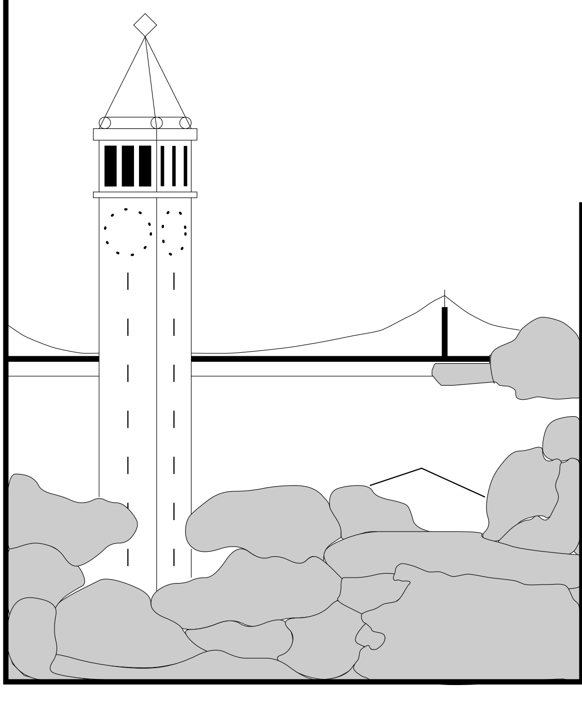
Handling the Complexities of a Real-World Language: A Harmonia Language Module for C

Stephen McCamant



Report No. UCB/CSD-02-1192

August 22, 2002

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

Handling the Complexities of a Real-World Language: A Harmonia Language Module for C

Stephen McCamant*
University of California, Berkeley
smcc@cs.berkeley.edu

August 22, 2002

Abstract

The syntax of popular programming languages often includes features that don't conform to the simplest models of program translation. Though designed to be easy for conventional compilers to handle, these features can cause trouble for language analysis in other environments. In implementing support for the C language within the Harmonia incremental framework, we've needed unconventional approaches to deal with the language's quirks. Closely following the language specification, we describe how a flex-based lexer, an ambiguous context-free grammar, and an object-oriented syntax-tree based analysis can be built to function well in a text editor or other interactive environment. By resolving syntactic ambiguities during name resolution, we collect accurate semantic information about identifiers and types, providing the basis for enhanced services in our augmented version of XEmacs.

1 Introduction

Harmonia is a framework for building interactive, language based tools for viewing, editing and transforming program source code [Har]. Harmonia is language-independent, providing incremental lexer and parser drivers and a framework for semantic analysis in which language-specific tables and code are contained in dynamically loadable libraries, called language modules. About two dozen of these modules have been written so far, providing lexical and syntactic specifications for a variety of languages, including many general-purpose programming languages as well as little languages specific to the Harmonia project.

We describe the construction of a language module to support the C programming language, specifically the recent ISO standard 9899:1999, 'C99' for short. (References of the form (§ 1.2.3.4) refer to rules in that standard, [JTC99a], which served as the main specification for what to implement, such as rule 4 in section 1.2.3). The C language module is the most complex language specification built for the project to date, mainly because of the extensive semantic analysis performed. While our system can perform semantic analysis for several other languages, including a small object-oriented language used in compiler implementation classes and our syntax specification language, C is the first 'real-world' language module to include the most significant phases of semantic analysis required to determine whether a program is legal.

The analysis of C presents several challenges, since the design of the language did not contemplate the sort of interactive environment Harmonia presents. Several features of C, including its separate compilation, use of the preprocessor, and insistence on declaration before use, show its origin in batch compilation systems with limited resources. While these features are at worst minor annoyances for modern C compiler writers, they present much more fundamental problems for the design of a Harmonia language module, which must work in a more restricted environment. In particular, the framework provides two major constraints: first, the boundaries between the usual lexing, parsing, and semantic analysis phases must be strict. Because our lexer and parser operate incrementally on changed regions of a syntax tree, phases can't make assumptions about what parts of other analysis phases have completed, or indeed even about a left-to-right ordering of analysis. Second, because our syntax tree is a unified representation of all the persistent information about

*This work was supported in part by NSF grants CCR-9988531 and CCR-0098314

a program, the analyses must all work on a common data structure. The structure of our single tree is thus a trade-off between the needs of the parser and of later analyses, and the results of semantic analysis are not new tables of information but annotations on the same tree.

Compensating for the rigidness of the framework, however, we also have access to some very powerful analysis tools. The most important of these is our generalized LR (GLR) parser, which is what allows us to deal with a troublesome part of C's grammar relating to type names. The C grammar gives different treatment to names declared with `typedef` to represent types, but there is nothing in the grammar to distinguish such names from ordinary identifiers. In a batch processing compiler, the conventional way to handle this ambiguity is to feed symbol table information back from the semantic analysis phase to the lexer, so it can give typedef names a different lexeme from normal identifiers. This approach would not work with Harmonia's pass separation: the semantic information simply is not available at the right time. Instead, we write an ambiguous grammar for C, and then use Harmonia's GLR parser, which can efficiently find all the alternative representations of the token stream and represent them in a packed 'parse DAG' structure. (Roughly speaking, a GLR parser 'forks' into separate streams of control whenever there is a conflict in the LR table; the streams rejoin when their states match). It is then left to a later semantic analysis to choose between the alternatives using information on the meaning of names. This general approach had been conceived as part of the rationale for the incremental GLR parsing algorithm ([WG97]), but not previously implemented in a full-scale system.

2 Lexing and Parsing

2.1 Lexing and the Phases of Analysis

With the exception of the translation steps commonly done by a preprocessor, the lexical structure of C is straightforward. Our lexical description is written using the standard notation of flex [Pax95], making liberal use of defined sub-patterns to encapsulate complicated regular expressions. A start condition is used to simplify the description of matching `/* */` comments, but none of flex's other more advanced features, such as trailing contexts or creating multiple tokens in an action, are needed. In particular, flex's longest-match heuristic conveniently coincides with the rules of C. The complete flex specification can be found in Appendix A.

The lexer tries to determine as much semantically useful information about a token as possible, though since the lexer interface is limited to giving one of a fixed set of token names to a string of text, this is limited to simple tasks like including the type modifiers on integer constants (see Figure 1). In contrast, transformations that involve changing the text of the program, such as the concatenation of adjacent string constants and the replacement of escape characters within them, are deferred to semantic analysis. The standard description of C [JTC99a] has these transformations occurring before parsing, but because our mechanism is incremental, we must keep a representation that matches the source code directly.

Conceptually, the operations of a C compiler on a source file are described as a sequence of many phases (§ 5.1.1.2), though in practice implementations compile either in a single pass, or with a preprocessor pass combining phases 1-4 (Figure 2). Since the scope of this project does not cover correct support of constructs enabled by the preprocessor, our system only fully supports the processing phases traditionally carried out by a compiler proper, and not the work of a preprocessor. As an exception, a few commonly used preprocessor features are recognized so that they can be passed over in source files where they appear. The most obvious example of a construct treated this way are comments (both the matching `/* */` and C++-like `//` types), but our lexer also recognizes backslash-newline line continuations when they appear in whitespace, and ignores preprocessor directive lines starting with `#`. The notorious ANSI 'trigraph' feature, in which for instance the sequence `??<` can be used anywhere in place of the character { that doesn't appear in some non-ASCII character sets, is not supported. On the other hand the feature that replaced them, 'digraphs'

```
{INTEGER}(ll|LL)                      { RETURN_TOKEN(INT_CONST_LL); }
{INTEGER}([Uu](ll|LL)|(ll|LL)[Uu]) { RETURN_TOKEN(INT_CONST_ULL); }
```

Figure 1: The lexer recognizes the various possibilities for type modifiers on integer literals, which along with the size of the integer (determined in semantics) decide the type of the expression. For instance, an integer literal ending in `ll` will have either signed or unsigned `long long` type (§ 6.4.4.1.5). (An excerpt from the full flex rules found in Appendix A).

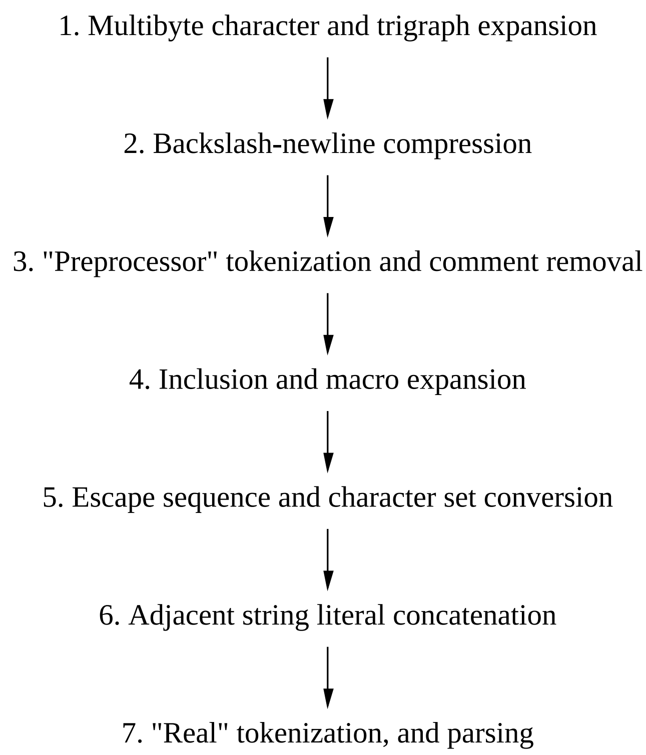


Figure 2: Phases of frontend processing specified in the C standard (conceptual). For an explanation of details such as the difference between preprocessor tokenization and "real" tokenization, see (§ 5.1.1.2).

like <% that can only appear as complete tokens, can conveniently be supported because their occurrence is isolated in single lexer rules. Also, the Java-style universal character names (`\uXXXX` and `\UXXXXXXXX`) are recognized, though not translated when they appear in identifiers, since the Harmonia framework doesn't support Unicode.

2.2 Parsing and Grammar

Harmonia's grammar specification language supports the standard EBNF ?, +, and * operators for specifying optional elements and sequences. A suffix of ? denotes that an element can occur once or not at all, a suffix of * that it can occur zero or more times, and a suffix of + that it can occur one or more times. In addition, Harmonia sequences can have separators, like the commas that occur between the elements of a list. These features allow us to treat sequences abstractly; rather than being left-recursive or right-recursive, they are balanced into trees by the parser and traversed by iterator objects.

Our context-free grammar for C is based on the standard, first from the second edition of K&R [KR88], then updated with the changes from the C99 standard [JTC99a]. Besides some abbreviation of the standards' verbose production names, the main differences can be thought of as reversing the changes made in translating the language's abstract grammar into the unambiguous context-free grammar presented in the standards. The standards' use of an *opt* subscript on nonterminals is a special case of Harmonia's ? operator, and most of its nonterminals with names ending in *-list* correspond to sequences that Harmonia can express with its + and * operators, depending on whether the nonterminal is optional when it occurs (Figure 3).

Whenever possible, the grammar has been arranged so as to minimize the size and complexity of the resulting parse trees; this reduces the memory usage of the representation, but more importantly simplifies the specification of semantic analyses. This is achieved by making as much use as possible of the EBNF operators, and then substituting the nonterminals with a single alternative (often incorporating EBNF) into the right-hand sides where they appear, whenever doing so is sensible. It is not possible to eliminate every single-production rule in this way, because the EBNF operators

Standard C Grammar	Harmonia Grammar
jump-statement: `return` *expression*$_{opt}$	`jump_stmt: RETURN exprs?`

compound-statement:
  `{` *block-item-list*$_{opt}$ `}`
block-item-list:
  block-item
  block-item-list block-item
	`comp_stmt: '{' block_item* '}'`

enum-specifier:
  `enum` *identifier* `{` *enumerator-list* `}`
enumerator-list:
  enumerator
  enumerator-list `,` *enumerator*
	`enum_spec: ENUM IDSYM '{' enum+[','] '}'`

Figure 3: Harmonia's EBNF operators allow us to abstract the patterns of optional elements and sequences in the C grammar.

multiplicative-expression:
  cast-expression
  multiplicative-expression `*` *cast-expression*
  multiplicative-expression `/` *cast-expression*
  multiplicative-expression `%` *cast-expression*
additive-expression:
  multiplicative-expression
  additive-expression `+` *multiplicative-expression*
  additive-expression `-` *multiplicative-expression*

Figure 4: The syntax of multiplicative and additive expressions, as specified in the C standard (§ A.2.1).

are not allowed to nest in the current implementation. However, this nesting might not be desirable anyway, since there are semantic behaviors that can usefully be factored out in the many places they appear.

The most important example of this transformation is the grammar of expressions. The C standard grammar is written to unambiguously specify the precedence of operators by using a different nonterminal for the expressions that occur at every level of precedence: *additive-expression*, *shift-expression*, *relational-expression*, and so on (Figure 4). While suitable for their purposes, this choice would be undesirable for Harmonia, as it would introduce many additional levels of one-child ('chain') nonterminals in the resulting parse trees, and require additional levels of indirection in the specification of semantic analysis. Unfortunately it is not possible to merge all of the expression nonterminals, since several of them occur in other productions of the grammar. We have been able to combine these rules into productions for three nonterminals, corresponding to the C standard's *expression*, *assignment-expression*, and *conditional-expression*, which for grammar brevity we abbreviate as `exprs`, `assign_expr`, and `expr` (see Figure 5 for an example). The complete EBNF grammar can be found in Appendix B.

2.3 Static Ambiguity Resolution

As in any LR-style tool, this merging of productions introduces spurious ambiguity in the grammar; besides the most numerous cases of precedence and associativity in expressions, ambiguity also arises from the most natural way of describing C's `if` statements with optional `else` clauses, the well-known 'dangling-else' difficulty. Since all of these ambiguities can be statically resolved based of the properties of a grammar, we would like to be able to resolve them at the time of constructing parse tables, so the parser need not worry about them at analysis-time. Harmonia's current facility for resolving these ambiguities, however, is limited to the capabilities provided by bison [CP99], which work intuitively only for resolving conflicts between binary operators: operators can be ordered according to precedence and declared to have certain associativities with `%left`, `%right` and `%nonassoc`, and productions can also be given precedence matching that of an operator with a `%prec` declaration.

As is the standard practice with bison and similar tools, the conflicts between most operators were resolved simply by declaring the operators to be either left- or right-associative in the correct precedence order. The other expression

```
%left<operator>     MINUS { alias '-' } PLUS { alias '+' }
%left<operator>     TIMES { alias '*' } DIV { alias '/' } \
                    MOD { alias '%' }
%%
expr: /* ... */
    | expr '+' expr
    | expr '-' expr
    | expr '*' expr
    | expr '/' expr
    | expr '%' expr
      /* ... */
```

Figure 5: The same part of the expression grammar shown in Figure 4, as a Harmonia grammar specification. By declaring the associativity and precedence of operators as properties of the tokens, we can specify the syntax of expressions in a way that matches the abstract view used by the semantics, without the need for the many chain productions found in the standard's unambiguous grammar.

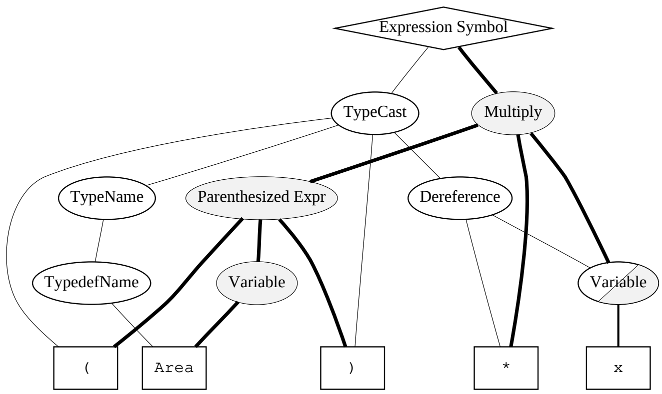


Figure 6: The parse DAG resulting from the expression `(Area)*x`. If `Area` is the name of a user-defined type, then the expression is a type-cast of a dereference (the lighter subtree containing the TypeCast and Dereference nodes), while if `Area` is a variable, the expression is a multiplication (the darker subtree with Multiply and Dereference). Harmonia's GLR parser combines these two interpretations (conceptually parse trees) into a single parse graph in which common children are shared. (For clarity, some chain nonterminals have been omitted).

productions like the conditional (`? :`) operator and array indexing can also be parsed correctly with appropriate precedence declarations, though it is not easy to see by inspection that the `?` and `:` tokens should, for instance, be marked as being right-associative to achieve the desired grouping. (The best way to make sense of the rule is that the sequence `?`-*expression*-`:` is to act like a right-associative binary operator).

The dangling-else ambiguity can also be resolved using precedence declarations, though not without extra complications to the abstract syntax tree. Ideally, we would like to specify that the sequence of an `else` token and its corresponding statement are optional in the production for an `if` statement, but in this case the table conflict would be between shifting the `else` token and reducing an empty optional `else`-clause. The nonterminal representing an optional `else`-clause is automatically generated in Harmonia's translation from EBNF, however, so it is not possible to give it a precedence declaration. Instead, we are forced to represent the choice by a pair of productions in the grammar. For C, we have chosen to write two productions, one each for `if` statements with and without `else` clauses.

2.4 Semantic Ambiguity Resolution

With all of these static ambiguities resolved, however, we are still left with the most important and troublesome ambiguity in the C grammar, which arises from the use of `typedef`. In several places in the grammar, most commonly in declarations

$$\mathit{Declaration} ::= \mathit{DeclarationSpecifier}^{+}\ \mathit{Declarator}^{*}_{\text{','}}\ \text{';'}$$

```
unsigned const int   x, *p;
```

Figure 7: The syntax of C declarations (simplified from Appendix B), and an example

a. `int (*x)[5];`
b. `struct foo { int x; };`

Figure 8: Other features of declaration syntax

a. `int (x);`
b. `int;`

Figure 9: Strange looking declarations that are still legal according to our grammar (b is semantically illegal under (§ 6.7.2))

but also when a parenthesized expression might be a cast operator, C needs to distinguish between the name of a type and other kinds of identifiers. This is easiest to understand in the case of a parenthesized expression: a type name in parentheses can only be a cast, while other things appearing inside parentheses are simply to be grouped for parsing. For instance, an expression like `(Area)*x` has two possible parses (shown in Figure 6) depending on whether `Area` is the name of a variable or a type.

The ambiguity is less intuitively obvious, though more common, in declarations. The basic structure of a C declaration is one or more 'declaration specifiers' followed by zero or more 'declarators'; for instance, in the declaration in Figure 7, `unsigned`, `const` and `int` are declaration specifiers, while `x` and `*p` are declarators. There are several possibilities in C's declaration syntax that are rarely exploited in practice: for instance, an extra set of parentheses are allowed around a declarator for grouping, as in Figure 8 line a, and declarations without declarators are possible, which is the reason for the oft-forgotten semicolon after a structure definition like Figure 8 line b, which is in fact a declaration of no instances of the structure.

For more standard kinds of declarations, examples that have the same structure, like those in Figure 9, look unusual, but our grammar allows them for the sake of uniformity in the rules. There are also some rules of the language that cannot be practically expressed in the grammar: for instance, a declaration is limited to certain combinations of specifiers (you cannot have a `short long int`), but the number of possible combinations is large: for instance, the keywords in the type most commonly referred to as `unsigned long int` can in fact appear in any of the 6 possible permutations, along with optional `const` and `volatile` qualifiers, and the `int` can also be omitted (Figure 10). While the set of possible specifications for this type is a context-free (in fact regular) language, there are so many possibilities that it would not be practical to encode them in our grammar — the grammar for type specifiers would be larger than the entire rest of the grammar. Instead, the grammar allows any combination of declaration specifiers, and the additional restrictions are enforced during semantic analysis.

This flexibility of syntax has a regrettable interaction with the `typedef` facility. Using `typedef`, any identifier can be made to act just as a built-in type name like `int` would, and typedef names are not marked in the way say structure tags are (by a preceding `struct` keyword). Since our lexer and parser, acting as truly prior passes, do not have access to the name information which will later be collected in semantic analysis as to which identifiers are typedef names and which are not, the only thing to do is to treat every identifier as the same kind of token. The grammar contains ambiguous productions, by which in some circumstances an identifier can be reduced either to a nonterminal representing a user defined type or to one for a regular object identifier. This lexical-level ambiguity turns into a number of unresolved conflicts in the construction of our LALR(1) parse table, and eventually through the action of the GLR parser ([WG97]) into multiple alternatives compactly represented in a parse DAG as in Figure 6. The method by which we choose between these alternatives is described in section 3.2. Even innocent-looking declarations like `int x;` are ambiguous, since the parser cannot tell that `x` isn't a type name. (Even though it is not legal to have both an `int` and typedef name in the same declaration, this constraint cannot cleanly be expressed at the level of the grammar, as described above).

```
const unsigned long int   unsigned long const int
const unsigned int long   unsigned int const long
const long unsigned int   long unsigned const int
const long int unsigned   long int const unsigned
const int unsigned long   int unsigned const long
const int long unsigned   int long const unsigned

unsigned const long int   unsigned long int const
unsigned const int long   unsigned int long const
long const unsigned int   long unsigned int const
long const int unsigned   long int unsigned const
int const unsigned long   int unsigned long const
int const long unsigned   int long unsigned const
```

```
const unsigned long
const long unsigned
unsigned const long
unsigned long const
long const unsigned
long unsigned const
```

Figure 10: The same type can a have quite a few different names. While the set of legal names forms a regular language, it would be impractical to represent the possibilities directly in our grammar.

2.5 Preprocessor Constructs in the Grammar

In order to make the language module more usable on non-preprocessed source files, which is a practical necessity for an editor, we have modified our C grammar to pass the most common preprocessor constructs through without complaint. For instance, when preprocessor directives appear at the top level or in between statements, they are ignored by the parser; this handles most uses of them, though the results of semantic analysis on such a parse tree are unreliable, and constructs like the conditional inclusion of parts of a function definition are unlikely to be recognized correctly. Also, because of their similar-looking syntax, most uses of macros can be parsed as if they were constant identifiers or function calls, though this too obviously will not work well for semantic analysis. As an additional use of GLR-based ambiguity, we allow 'function calls' to take arguments that are really the names of types, as a sop to 'function calls' that are really unrecognized macros, though of course such interpretations are determined not to be functions in semantic analysis.

3 Semantic Analysis

The general framework for semantic analysis in Harmonia is an object-oriented language for operations of syntax trees, ASTDef [Bos01], which is translated into C++ methods for syntax tree node objects. A semantic analysis is a collection of methods defined on the classes for different nodes of a language's syntax trees, usually a recursive traversal of the tree of one sort or another. Our semantic analysis for C consists of two passes, one that collects names and resolves ambiguities, and another that collects and checks type information. These will be described in the following subsections.

3.1 Name Resolution

One of the most basic kinds of semantic information, and the one that is probably most immediately useful in an editing context, is the linkage between entities referred to by the same name: the naming of variables, functions, and other language features is the most basic non-textual level of structure in source code. For C, name resolution is additionally important because the information derived by name resolution is what is needed to resolve the grammatical ambiguities caused by typedef names. In fact the interplay between name resolution and disambiguation is two-sided, since a resolution of the typedef ambiguities is also needed to correctly recognize what identifiers are being declared. Therefore, the two activities of name resolution and disambiguation need to occur in a single left-to-right pass over the syntax tree.

If it were not for the interaction with disambiguation, name resolution for C would be fairly straightforward, though the language does include a large number of details to attend to. There are several different kinds of namespaces in a general C program: aside from macros (handled by the preprocessor) and keywords (recognized by the lexer and parser), a given identifier could name:

- a data object, a function, an enumeration constant, or a typedef name (all of which share the same namespace, and can be shadowed in inner blocks),
- an entire structure, union, or enumeration (which share a separate namespace, but also respect block structure),
- a statement label for `goto` (for which there is one flat namespace per function), or
- a member of a structure or union (for which there is one namespace per structure or union).

Each of these namespaces is represented by a different hashtable during name resolution (using an existing Harmonia template class for possibly-nested namespace tables), though the tables for identifiers and struct/union/enum-tags are linked so that they can easily be scoped together. Collecting name information is then a relatively straightforward process of walking the syntax tree from left to right, adding and removing hashtables from a (conceptual) stack when scopes are entered and exited, adding entries when declarations are processed and looking entries up when name uses are encountered. For later reference, we also retain information about the context in which declarations are encountered, and the linkage of identifiers (that is, whether they are visible from other compilation units; though we do not yet handle multiple compilation units, linkage has implications for some other language rules).

3.2 Disambiguation

Conceptually separate from name resolution, though overlapping in time, is the process of disambiguation. Operating on the ambiguous DAG produced by the parser, we choose for each ambiguity which alternative we prefer.

3.2.1 Ambiguities in the Harmonia Tree

When Harmonia's GLR parser constructs multiple sub-trees representing one section of the source code, an extra level of indirection is added to the syntax tree at the top of the region with multiple interpretations. Rather than the parent of the region having one or the other of the interpretations as a child, its child is an abstract node representing the common left-hand-side of the possible productions, and having the alternative subtrees as children. (These nodes are referred to as 'symbol nodes', after nodes in a previous version of the GLR algorithm that intervened in every parent-child relationship [Rek92], but a term like 'choice node' would probably be more descriptive). Associated with each symbol node is the notion of a primary alternative: the parser chooses this alternative arbitrarily, but it is key to the way clients of the analysis view the document. The alternatives of symbol nodes can share children, so in general the parse 'tree' is really a directed acyclic graph, but the interface exported to clients is just a tree, specifically the tree that results from choosing only the primary alternative at each symbol node. (In fact, many clients use a filtered view of the tree that omits the symbol node levels completely).

3.2.2 Disambiguation by Elimination

There are several ways in which structural ambiguities in a language might be resolved. For instance, some ambiguities might be resolvable based on information available in the tree at the point of ambiguity (the lowest node dominating both alternatives). In other cases, one might want to heuristically evaluate the choices and give each a score, picking the alternative with the best score. In the case of C, the most direct approach follows from the form of the language specification: besides the grammar, additional (in some sense semantic) rules are provided which further restrict the set of legal programs, to a subset of what is allowed by the context-free grammar. To resolve an ambiguity, we carry out a semantic analysis on each of the alternatives at an ambiguity, being sure to look out for violations of the additional semantic rules. Whenever we encounter one of these errors, we reject the alternative in which it appears; if all the rules are implemented correctly, then at most one alternative will be left for each ambiguity, which if it exists will be the correct one. (If the program has a genuine error, it is possible that none of the errors will be correct; in this case we concatenate the error messages for each alternative to present a composite message). For this procedure to be efficient, the disambiguating errors need to occur early in the alternative subtree, to minimize the analysis effort duplicated between the alternatives (in the case of nested ambiguities, there is a danger of exponential running time), but the ambiguities in C seem to satisfy this criterion.

The C rules we enforce to resolve ambiguities are:

- The declarator in a function definition must declare an object of function type. (§ 6.9.1.2)
- A typedef name in parentheses as a parameter declaration is an abstract declarator for a function with a single parameter, not a redundantly parenthesized parameter declarator (for a parameter whose name would shadow the typedef). (§ 6.7.5.3.11)
- An old-style function declaration may not declare a parameter with the same name as an existing typedef name. (§ 6.9.1.6)
- A declaration may include at most one typedef name. (§ 6.7.2.2)
- A declaration that includes a typedef name may not include any other type specifier. (§ 6.7.2.2)
- Every declaration must include at least one type specifier. (Note that this rule is not part of the C89 standard, in which `int` can be implicit; C89 has a weaker rule that requires every declaration that redefines a typedef name to include at least one other type specifier, which would also serve to resolve this ambiguity). (§ 6.7.2.2).
- An identifier can be used as a typedef name only if it has been declared to be one.
- An identifier cannot be used as a primary expression if it has been defined as a typedef name. (§ 6.5.1.2)

3.2.3 Data Structures for Disambiguation

Implicit in this plan of tentatively analyzing alternatives is the need, after we have finished analyzing a subtree, to either discard the work we had done in analysis as incorrect, or incorporate it into our representation for analyzing the rest of the program. Since our analysis collects information by updating a number of data structures, we need to be able to either avoid making those changes until we have decided on the right alternative, or roll them back when we discard an alternative. For the latter approach, the facility we need is similar to the dynamic scoping operators of languages like LISP (`fluid-let`) and Perl (`local()`), but no such facility is available in ASTDef or the underlying C++. Instead, we must implement it by hand for each relevant data structure. For simple types that can be assigned in C++, we can create an `auto`-allocated object whose destructor restores a saved value at block exit with a simple template. For name tables, we extend Harmonia's nesting `Scope` templates with a new kind of scope (called an `OverlayScope`) that for read accesses appears transparent, as if it were part of its parent scope, but stores new entries separately. Once we have decided whether an alternative is correct, the overlay can either be discarded, or its entries can be copied into the outer scope (which might itself be an overlay in the case of nested ambiguities). Finally, for changes we make to Harmonia's syntax tree for token sub-category classification, (described below), we keep a list of potential changes in a separate structure, and only apply them once we have left an outermost ambiguity.

3.2.4 Disambiguation and Parent Pointers

For efficient traversals, Harmonia's nodes also keep pointers to their parents, but this notion becomes more complicated in the presence of multiple alternatives. If a node is shared between alternatives (as at least tokens always are in the current implementation), it can have more than one parent, and in order to present a consistent interface to the tree-style clients, there needs to be a distinguished primary parent, corresponding to the choices of primary alternatives elsewhere in the tree, so that a node's primary parent is the one by which it is reached in the tree of primary alternatives (see Figure 11). Our disambiguation consists, in the end, of changing the primary alternative of symbol nodes, so when doing so we must also reset the parent pointers in the region of the tree below the symbol node being changed but above any nested symbol nodes, which we do with a simple traversal (see Figure 12). An additional constraint is that we must carry out this alternative and parent-pointer setting for every symbol node that dominates a symbol we wish to

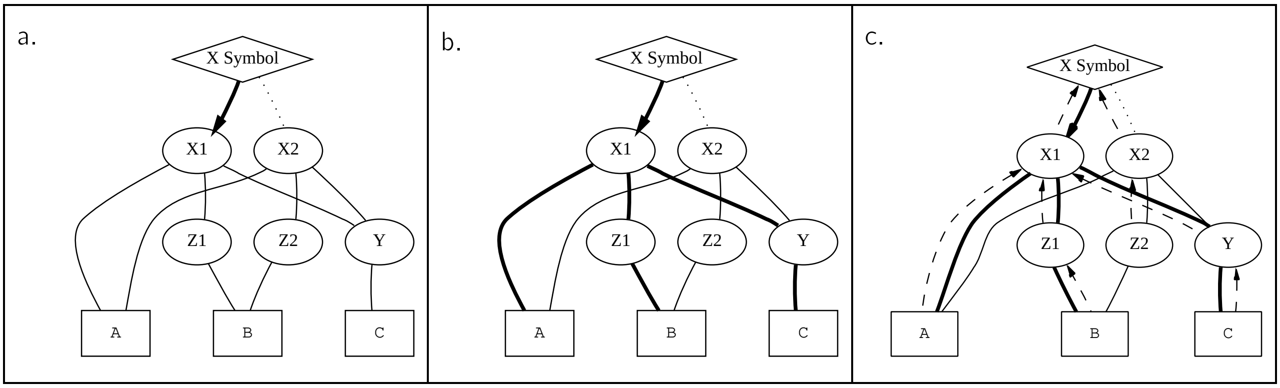


Figure 11: Symbol nodes and primary parents. Every symbol node has one of its alternatives designated as 'primary' (here X1, shown by the bold arrow in a). The nodes reachable by paths from the root which only use primary alternatives form the primary subtree (shown with bold edges in b). Every node also has a single primary parent link (shown with dashed lines in c), which points to the parent by which it can be reached along a path in the primary subtree.

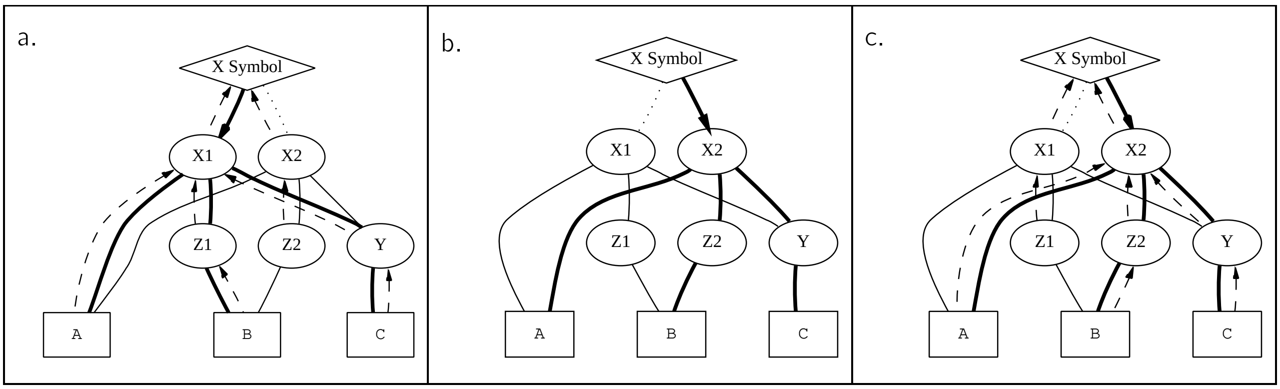


Figure 12: Changes in alternatives require changes in parent links. In (a), X1 is the primary alternative. If X2 rather than X1 is chosen as the primary alternative for the symbol node (bold arrow in b), the membership of the primary subtree changes (bold edges in b), and corresponding changes are needed in the primary parent pointers (dashed arrows in c).

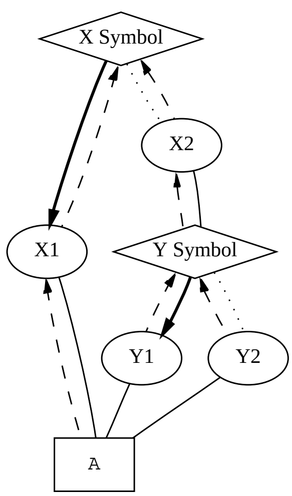


Figure 13: In the presence of nested symbol nodes, we cannot set the primary alternative for symbol node Y, without later choosing an alternative at symbol node X, lest the primary parent of a shared child (A) be set incorrectly.

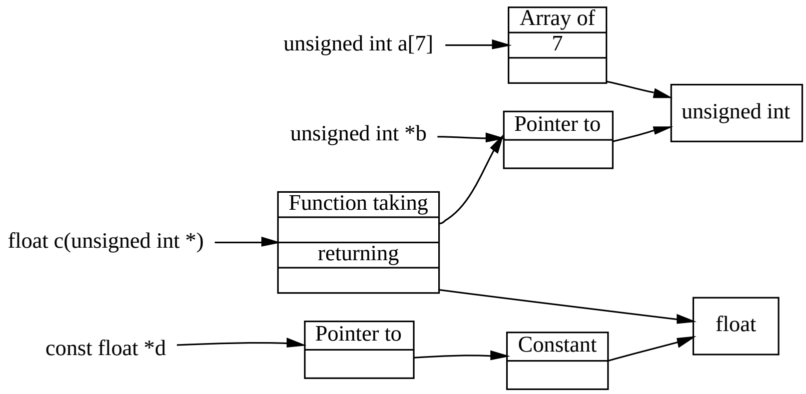


Figure 14: C's types are represented by linked data structures eventually pointing back to a set of basic types.

disambiguate (even if we have no basis for choosing between the alternatives at a higher symbol node), to ensure that every node whose parent might have been reset during our analysis has in fact had its parent set to the correct parent (Figure 13). (In a sense, this situation is like the ones above in calling for a mechanism to undo the parent pointer changes we made inside an eventually-discarded alternative analysis, but there is not a clean way to encapsulate the needed information-saving in a way that could be used in other languages' analyses. Harmonia's tree versioning supports rolling back an incomplete transaction's worth of changes, but not nesting such transactions).

3.3 Exporting Analysis Results

The collection of hash tables and other structures we use during the analysis to collect name resolution information are efficient and powerful for the needs of the analysis, but they are rather complicated. For Harmonia, we'd like to make that information available to client programs using a standard interface for version-sensitive syntax tree attributes, so that they can be accessed by applications written in any supported language. To do this we represent the information as tree annotations of a restricted form (strings, integers, and pointers to other nodes can be attached to nodes of a particular type), and take advantage of Harmonia's facilities for efficiently recording the history of values that change over time. The semantic information is then available to the Harmonia XEmacs mode as well as other Harmonia applications. Though an infrastructure for multiple semantic analysis passes does not really exist, later passes also limit themselves to this form of information, rather than the transient structures like hash tables, so that analysis in the future can be provided more flexibly.

The most important form of annotation produced by name resolution is links (pointers) from each use of an object identifier to its declarator; these are supplemented by links between the declarator and the identifier being declared, from `goto` statements to their targets, from identifiers with linkage to other linked identifiers, and from incomplete structure and union types to their complete definitions. Another useful kind of information produced during name resolution is a further classification of identifiers (which cannot be distinguished from one another by the lexer) into sub-categories such enumeration constants, structure tags, statement labels, variable declarations, typedef names, and so on. (Many of these categories correspond to a token's syntactic context, but until disambiguation has completed a token might equally well be thought of as being in two different syntactic contexts).

3.4 Type Checking

With disambiguation out of the way, the type checking phase of semantic analysis can proceed in much the same way as it would in a conventional compiler. The differences in doing the analysis for Harmonia stem from not being able to transform our program representation (since we must retain the tree in the same format the parser found it), and performing the analysis to the specification in the abstract, rather than for the benefit of any later compilation phases.

For a language that is sometimes criticized for a lack of type enforcement, C has a quite complex type system, with a large number of built-in types and many ways of constructing complex types out of simpler ones. To handle these uniformly, our representation is a linked, object-oriented one: the basic types like `unsigned int` have distinguished

a. `int ***x;` `int x[1][1][1]` $\lfloor \lfloor \lfloor \mathtt{x}_{\text{int}***}\,\mathtt{[1]} \rfloor_{\text{int}**}\,\mathtt{[1]} \rfloor_{\text{int}*}\,\mathtt{[1]} \rfloor_{\text{int}}$

b. `int x[5][5][5];` `int x[1][1][1]` $\lfloor \lfloor \lfloor \mathtt{x}_{\text{int[5][5][5]}\mapsto\text{int}(*)\text{[5][5]}}\,\mathtt{[1]} \rfloor_{\text{int[5][5]}\mapsto\text{int}(*)\text{[5]}}\,\mathtt{[1]} \rfloor_{\text{int[5]}\mapsto\text{int}*}\,\mathtt{[1]} \rfloor_{\text{int}}$

Figure 15: The decay of array types (in an array lvalue) into pointer types allows a true multi-dimensional array (b, where `x` is declared as `int x[5][5][5]`) to use the same dereference syntax as a nested pointer (a, where it is declared as `int ***x`). In this figure and the next, the types of sub-expressions are represented by subscripts in the traditional C type notation, and type conversions are represented by a $\mapsto$ arrow.

a. `int f(void);` `(&f)()` $\lfloor \lfloor \mathtt{(\&f}_{\text{int(void)}}\mathtt{)} \rfloor_{\text{int}(*)\text{(void)}}\,\mathtt{()} \rfloor_{\text{int}}$

b. `int f(void);` `f()` $\lfloor \mathtt{f}_{\text{int(void)}\mapsto\text{int}(*)\text{(void)}}\,\mathtt{()} \rfloor_{\text{int}}$

c. `int f(void);` `(*f)()` $\lfloor \lfloor \mathtt{(*f}_{\text{int(void)}\mapsto\text{int}(*)\text{(void)}}\mathtt{)} \rfloor_{\text{int(void)}\mapsto\text{int}(*)\text{(void)}}\,\mathtt{()} \rfloor_{\text{int}}$

Figure 16: Whenever a function type appears in an expression (except as an argument to `&` or `sizeof`), it is converted to a pointer-to-function type (§ 6.3.2.1.4). Thus, if `f` is a function (declared for instance as `int f(void)`), any of the syntaxes `(&f)()`, `f()`, or `(*f)()`, as in a, b, or c, may be used to invoke it [JTC99b].

objects, and then derived types like `unsigned int[7]` ('array of 7 unsigned ints') are other objects that point to their constituent types (see Figure 14 for examples). For efficiency, the results of this type construction are memoized: the first time we make a seven-element array of unsigned integers, we record that type in a hash table that is part of the unsigned integer type object, so that when that derived type is needed again we can just retrieve the same type object. Corresponding to the different ways in which a derived type can be constructed in C, every type object also keeps track of:

- the type of pointers to that type,
- a hash table holding the types of arrays of that type of various lengths and other qualifiers,
- a hash table holding the types of functions returning that type with various argument lists, and
- the seven possibilities of versions of the type qualified by combinations of `const`, `volatile`, and `restrict` (`restrict` is a C99 extension).

In addition to this inclusion relationship between type objects, there is also an inheritance hierarchy among the classes of type objects, reflecting the classification of types in the standard: for instance, all the integer types have a common `IntegerType` parent class. The information that the rest of the analysis needs about types is provided by queries that are virtual functions on the type objects, taking advantage of commonalities of related types to simplify the specification.

Besides the type information proper, the description of expressions in the C standard classifies them in some additional ways, mainly having to do with whether they can change value or be modified. Following the taxonomy in [PB89], our language module captures this with a notion of expression class like 'address constant' or 'array lvalue', including various conversions between classes. This notion of class is overloaded to include several relatively unrelated constraints that apply to expressions in context. Identifiers that represent arrays and pointers turn into element pointers and function pointers respectively when used in expressions; these conversions are part of the way the standard formally specifies the meaning of array subscripting and indirect function calls. (The need for these conversion rules can be seen in the somewhat unintuitive examples in Figures 15 and 16).

Expressions that designate objects in memory are classified as lvalues, though not all lvalues can actually be modified; there is a separate category of 'modifiable lvalues' that actually coincides with the expressions that can be assigned to. Finally, different parts of the language require different kinds of constant expressions: integer constants (such as in case statements), arithmetic constants, and address constants (as in static initializers). The type checking pass computes the type and expression class of each expression in the tree in one pass: both are essentially computed bottom up, though in some cases a mismatch between the class of expression required by an operator and the bottom-up computed class requires a conversion, which also changes the type (for instance, when an expression that could be an lvalue is used in an rvalue context, qualifiers like `const` are dropped from the type). To simplify passing this information around, the return value of the type checking routines is conceptually an ⟨expression-class, type⟩ pair, encapsulated in a single object.

3.5 Obstacles to Pass Separation

An additional complication of C, relative to the structure that is been described so far, is that name resolution and type checking cannot actually be treated as completely separate passes. In modern versions of C, each structure or union has a separate name space for its members, so that for instance two different structures can have fields with the same name. As a consequence, it is not possible to tell which field with a given name a structure access operator (`.` or `->`) is accessing without knowing the type of the object being accessed. Because of this ordering constraint, the name resolution of structure members is postponed from the main name resolution pass, and done instead when the expression is type checked. Because a table listing the members of each structure or union type was prepared during name resolution, though, this operation is just a simple lookup; in particular, no additional disambiguation is required.

Our division of the analysis into passes also complicates some aspects of C's semantics that are structured around a left-to-right ordering of the code in a compilation unit. C has the notion of an 'incomplete type'; for instance, the type of a structure is incomplete if only a forward declaration like `struct s;` has been seen, and not a definition that gives the members of the structure. In order to ensure that a one-pass compiler can lay out data structures, C requires that a type be complete when it is used in certain ways, such as the element type of an array. Straightforwardly following our two phase approach, the incomplete and completing definitions for a type would be found and linked together during the name resolution pass, while the completeness of object definition types would be verified during type checking. By the time we reach the type checking pass, however, this approach has lost any information about where in the tree an incomplete type was visible (assuming that we eventually saw a complete definition), since the data structures only reflect the most up-to-date information about the type. To correctly enforce this requirement, we need to either separate the representations of complete from incomplete versions of a type or check the relative position of the declaration and its use in the tree: we have for the moment chosen the first approach as simpler.

3.6 Practical Aspects of Semantic Analysis

ASTDef's object-oriented framework has some advantages for separating the concerns of different analysis and different kinds of nodes, but it has the effect of de-localizing the control flow in an analysis — the code rarely continues on for more than a few lines in a single method before returning to code for some other kind of node. Thus the control flow is best understood in terms of the inclusion relationship between rules in the grammar, shown in Figure 17. The core of each analysis is a traversal that walks through the entire tree, but it is necessary to define smaller sub-traversals to access information from nested tree structures (for instance, to find the name of the variable being declared in a declarator).

Because an analysis is not a first class entity in ASTDef, we haven't generally designated data types and variables as belonging to a particular analysis. It is natural to associate some declarations with particular kinds of node, but in many other cases data structures are needed to pass information between the code analyzing different nodes. For data structures, the best approach appears to be to put them in a header file shared by the entire module; they are then easy to use, though information hiding between analyses is not enforced. For variables that conceptually would be global to an analysis (like a name table in name resolution or the set of basic types in type checking), what we have done is create a 'context' structure that is passed by reference among all of the analysis methods, and filled with whatever information is necessary. (Adding new parameters to methods one by one quickly becomes cumbersome: our name resolution and type checking passes have about 250 methods each).

Another task that is part of implementing semantic analyses in Harmonia is giving labels to the elements on the right-hand side of each production so that they can be referred to from code. These names in the syntax specification are used as the basis for the names of accessor methods on the node objects, which are the way the semantic analysis passes traverse the tree. For elements that are modified by EBNF operators, at least two names are required: one that names the element 'outside' the modifier (for a sequence specification, this gives the iterator), and one for each element within the modified sub-production (there can be more than one if the operator applies to a parenthesized sequence of elements). Accessors are needed for each item that is used by a semantic analysis, so for most languages they are not needed on elements of the concrete syntax like parentheses and semicolons that play no further semantic role after

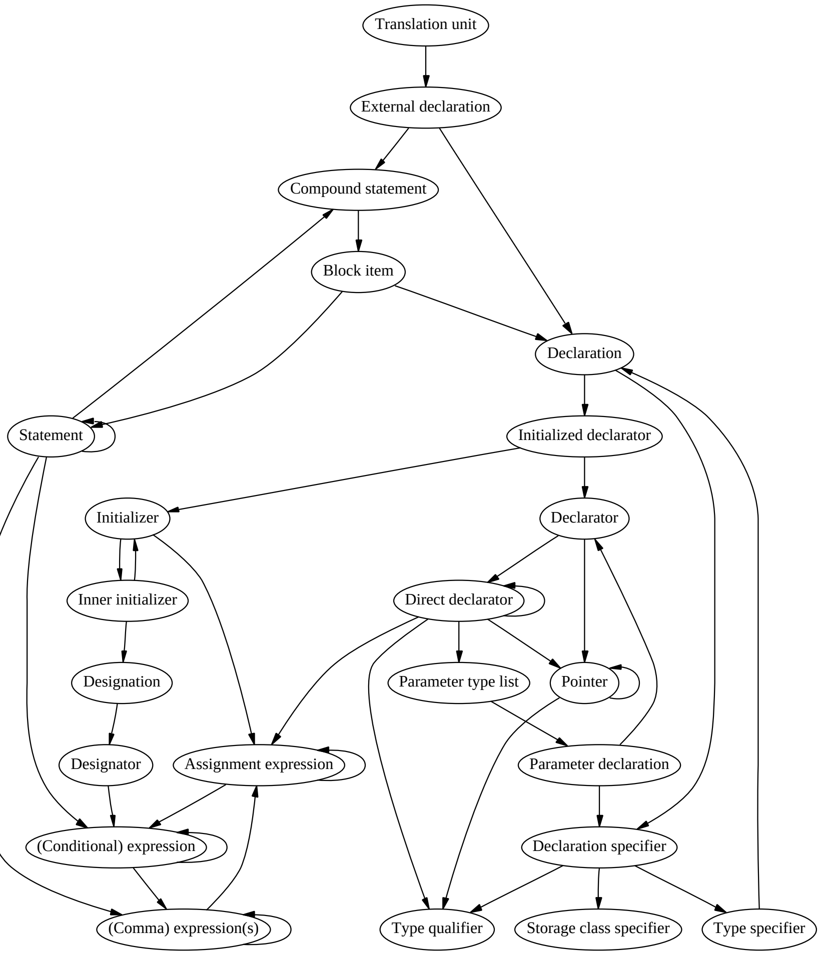


Figure 17: Rule inclusion relationships in C (simplified)

```
prog: BEGIN (stmt ';')+ END?
prog: begin:BEGIN stmts:(stmt:stmt semi:';')+ end:(end:END)?
```

Figure 18: A production, before and after accessor names have been added.

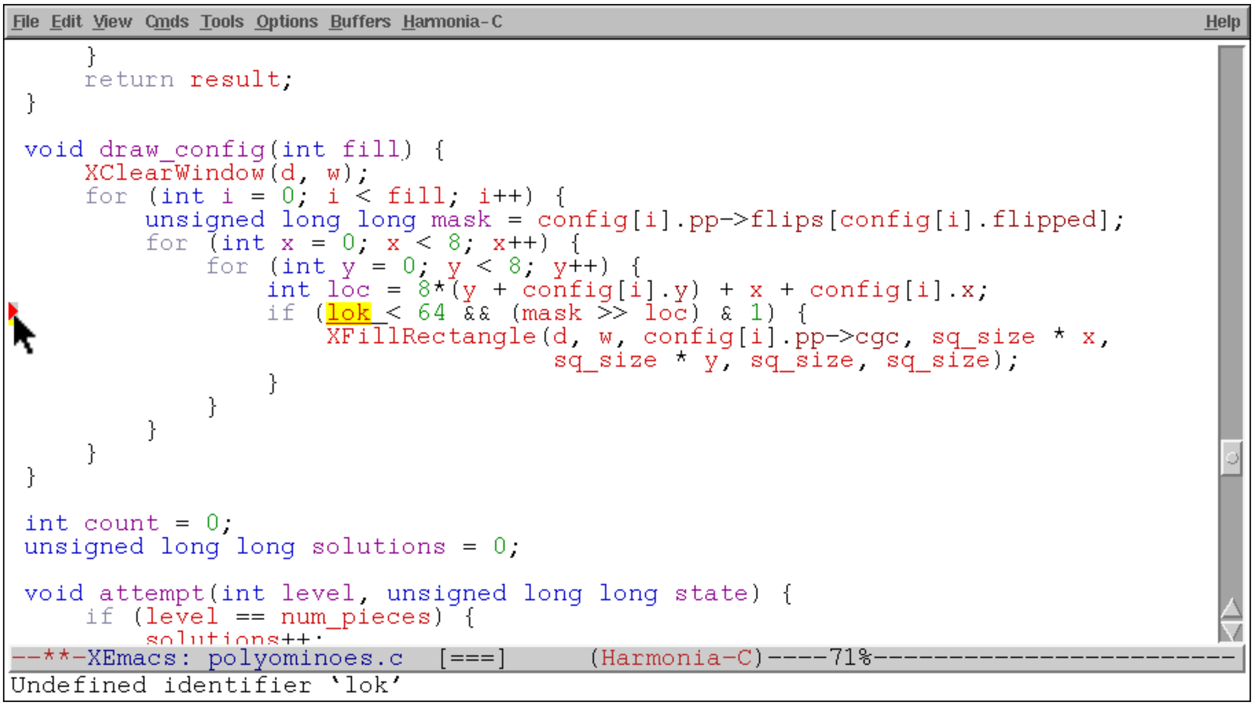


Figure 19: Our language module in use in a Harmonia-enabled version of XEmacs. A marker in the margin points out a semantic error (a misspelled identifier), which is displayed in the modeline when the marker is under the mouse cursor.

parsing. For C, however, we have added accessors for every item in the grammar as part of a project that used a tree traversal for output purposes. Figure 18 shows an example of this transformation.

4 Exploiting Semantic Information

The additional semantic information that the C language module computes in its analysis is used by Harmonia's XEmacs editing mode to provide more powerful and more correct services to users (Figure 19). One example of this power is the improvement in syntax highlighting that can be achieved by using more complete information about the role tokens play in a program. For instance, it is easy enough for any editor to highlight the keywords like `float` and `short` that specify types, since the set of such keywords is fixed. If the editor is to provide a consistent view of C's syntax, however, other type specifiers, both standard ones like `size_t` and types defined in the user's program, should be highlighted in the same way. It is not possible to correctly distinguish such type names with a simple text match, since they consist of the same kind of sequence of letters as regular identifiers, but when information from name resolution is available, it is trivial to recognize those tokens whose sub-categorization (based on an earlier declaration) marks them as typedef names.

With the correct syntactic structure determined after disambiguation, we can also take advantage of Harmonia-Mode's structural editing features [Too02]. For instance, we can use structural navigation to traverse a file one function at a time, or drill down to structure of statements and expressions. Structurally-filtered views let us temporarily hide the comments in a file, the bodies of functions, or other syntactically defined subsets of the code to concentrate on whatever else is important for the current task. Last but not least, we can put syntactic modifiers on what would otherwise be a textual I-search, for instance to find the word 'token' only when it's the name of a variable, not when it occurs in strings or comments. A related feature, which would be easy to implement but has not yet been, would be to make the links between uses of identifiers to their definitions available as hyperlinks, allowing users to navigate 'semantically' through the naming structure of a program.

Another convenient use of interactive analysis is to display semantic errors directly in the text editor as soon as the user has made them. Since our analysis does most of the checks that a C compiler does, it can also catch most of the same errors that a compiler does (it cannot yet find problems in data flow, though, like the use of an uninitialized variable). There is also a certain advantage to handling the ambiguities of the C language explicitly, rather than deciding in the lexer what kind of identifier a lexeme is, as most batch C compilers do. Our module can display the errors that disqualified all of the ambiguous alternatives, while many compilers can do little more than signal a generic 'syntax error' when the structure of a declaration did not match the category it guessed for an identifier.

5 Areas for Future Work

5.1 Analysis of C

Though the module described above is a close-to-complete implementation of the rules of standard C, there is still a significant amount of work needed to produce the best tool for day-to-day editing. First, there are some small pieces missing from the type checking analysis as it stands: compound assignment operators and complex initializers are not type checked, enumeration constants are not matched with their definitions, and constant expressions are not folded to their correct values (so, for instance, we cannot check whether the case labels in a switch statement are really distinct). There are also some other analyses traditionally performed by compilers that would be useful, such as control and data flow to find unreachable code and uninitialized variables.

There are two other missing complexities that are the main practical limits on the utility of the current module: the lack of support for either preprocessing or multiple compilation units. The preprocessor unfortunately plays a large part in the meaning of many C programs, through macro definitions, conditional compilation, and header inclusion. In most cases, these facilities are used in fairly well structured ways: macros serve either as constant expressions or like inlined functions, conditionals bracket syntactically complete sequences of statements, and header files are just used to specify interfaces [EBN97]. One approach then is to handle only 'well-behaved' preprocessor uses, and ignore or rewrite the rest. A more ambitious approach would be to exactly replicate the behavior of the preprocessor, performing the same text transformations a preprocessor does as a pre-lexing stage of analysis, and matching the results of the rest of the analysis back onto the non-preprocessed source. While this is the only strategy that is sure to work with any existing C program, there are significant challenges involved in bridging the gap between the two representations of a program. The division of C programs into multiple files can be considered part of the challenge of working with the preprocessor, but it is perhaps the most important special case, since even C programs that consist of only one file generally require functions from the standard library. We have some plans for doing analysis incrementally at the granularity of program units (discussed in [Bos01]), but they are better explored first in a language with simpler inter-unit semantics than C.

5.2 Harmonia's Analysis Infrastructure

Harmonia's ASTDef framework is flexible enough to support many analysis strategies with efficient (though batch-structured) code, but the process of writing the code for an analysis is more tedious than it needs to be. One area that could be streamlined is the selection of accessor names for all the components of each production. For C, it was quite uncommon for the name of an accessor to convey any extra information about the role that the corresponding item played semantically: the names usually duplicated or were abbreviations of the item itself, or plurals in the case of sequences. (Recall the `stmts:(stmt:stmt` example from Figure 18.) It would be easier on implementers, more predictable, and make the syntax specification easier to read if such conventional accessor names were generated automatically, and only the exceptions were explicitly specified.

Another helpful change would be to make the concept of a traversal of the tree a first-class notion in the ASTDef language. Roughly, a traversal would be a depth-first left-to-right walk over some subset of nonterminals in a grammar, implemented with a recursive method on each nonterminal node object with an identical name and argument list. The name resolution and type checking passes described above would correspond to two main traversals, with a few sub-traversals extracting extra information in some contexts. From the point of view of structuring code it would be convenient for a traversal to declare its own types and global (to the traversal) variables; the latter could be implemented by the same sort of context variable we used for C, but without the need to explicitly declare it. Within this structure, the more predictable parts of a traversal could be specified declaratively and implemented automatically: for instance, if on a particular nonterminal a traversal simply recurses on all the nonterminal's children (say as type checking does on all the declarations in a file, all the statements inside a block, etc.), this could be specified compactly. Some other common patterns are a traversal that recurses on a nonterminal's children one by one until a particular return value is returned (as our name resolution does when it finds an alternative-disqualifying error), or recurses on a particular child (as the accessor traversal to find the name of the variable being declared in a declarator does).

As alluded to above, a feature that would have been quite helpful for this project, though perhaps only because of our particular disambiguation strategy, would be a uniform way to make 'undo-able' data structures in which an abortive foray can be efficiently retracted. It likely isn't feasible to solve this problem for general C++ data structures, but the infrastructure Harmonia provides could be improved to make it easier to undo changes to the central syntax tree structure. The tree already has mechanisms for versioning that include much of the change tracking that would be needed, but some of the current assumptions about how they are used would have to be modified (either allowing the unfinished analysis transactions to nest, or having an analysis use a whole subtree of versions).

A final possible direction for evolution would be to re-introduce the distinction between concrete and abstract syntax, to allow some of a language's complexity to show up declaratively in the syntax but not in the tree structure used by analysis. It's generally easier for an implementer to express a language constraint in the syntax description, when possible, than as a rule in semantic analysis. On the other hand, there are countervailing pressures to simplify the description of syntax, particularly as it affects the tree structure seen by later analyses and clients. This pressure is quite strong in Harmonia, because the lack of an abstract syntax means that every additional complexity added in the syntax requires extra method calls and levels of indirection in each semantic pass that operates on that tree structure. In C, this tradeoff appeared most visibly in specifying the possible combination of declaration specifiers (the complexity in Figure 10). Faced with the choice of expressing the constraints with perhaps 100 lines of grammar or what turned out to be 500 lines of mainly error-checking code, we chose the latter, since each additional line of grammar complexity would cause a factor of 5 to 10 times as many lines of semantic analysis overhead. There are of course other trade-offs involved in the choice of where to enforce language constraints: while a parser can't automatically generate very informative error messages, an incremental parser like Harmonia's can trace an error back to a previous change, which is intuitive in the context of an editor. Incrementalizing semantic analyses is more difficult, and instead to achieve informative error messages we had to invert the description of the standard, which lists what is allowed, to efficiently recognize instead what combinations are not allowed. While this ultimately gives very informative messages, the inversion itself is a complex transformation, and the resulting code would be difficult to modify if a new type specifier were introduced (as several were in C99). Harmonia's structure has been simplified relative to earlier projects by unifying the notions of concrete and abstract syntax, but this simplification has come at the price of requiring compromises to the design of both. (Another challenge that should be mentioned to the prospect of pushing more complexity into the grammar are the scalability limitations we have encountered with our current systems for parser tables and syntax tree code generation).

6 Conclusion

We've shown that with the right tools, we can develop analyses for a real-world programming language, C, almost directly from its specification. Using the Harmonia framework, we've constructed an incremental lexer directly from a standard flex specification, and an incremental GLR parser from an EBNF grammar. Using a specialized language that translates into C++ code, we've implemented batch semantic analyses that collect and verify name and type information, and by combining the GLR parser's support for ambiguous grammars with semantic disambiguation, we can cleanly support ambiguities, such as those associated with `typedef` names, that are usually handled with inter-pass feedback. Grouped into a Harmonia language module, these analyses allow language-aware tools, such as an enhanced version of XEmacs, to provide enhanced, correct user services.

A Flex lexical specification

```
%x INCOMMENT

HEXDIGIT      [0-9a-fA-F]
OCTESCAPE     \\[0-7]{1,3}
HEXESCAPE     \\x{HEXDIGIT}+
```

```
UNIV          \\u{HEXDIGIT}{4}|\\U{HEXDIGIT}{8}
ESCAPE        (\\[ntvbrfa\n\\?'"])|{OCTESCAPE}|{HEXESCAPE}|{UNIV}
STRING        \"([^\\\n"]|{ESCAPE})*\"
CHARLIT       \'([^\\\n']|{ESCAPE})*\'
WSCHAR        [ \n\t\f\v\r]
WHITESPACE    {WSCHAR}+|({WSCHAR}*\\\n)+{WSCHAR}*
IDENT         [_a-zA-Z]([_a-zA-Z0-9]|{UNIV})*
DIGIT         [0-9]
NUMBER        {DIGIT}+
ZNUMBER       ([1-9]{DIGIT}*)|0
INTEGER       {ZNUMBER}|(0[0-7]+)|(0[xX][0-9a-fA-F]+)
EXPONENT      [Ee][+-]?[0-9]+
FRACTIONAL    ([0-9]+\.)|([0-9]*\.[0-9]+)
DECFLOAT      {FRACTIONAL}{EXPONENT}?|[0-9]+{EXPONENT}
HEXFRACT      {HEXDIGIT}*\.{HEXDIGIT}+|{HEXDIGIT}+\.
BINEXP        [pP][+-]?[0-9]+
HEXFLOAT      0[xX]({HEXFRACT}|{HEXDIGIT}+){BINEXP}
FLOAT         {DECFLOAT}|{HEXFLOAT}
DIRECTIVE     {WSCHAR}*#(.*\\\n)*.*
%%
{WHITESPACE}                      { RETURN_TOKEN(WSPC); }
<INCOMMENT>"*/"                   { BEGIN(INITIAL); RETURN_TOKEN(COMMENT); }
<INCOMMENT>.|\n                   { yymore(); break; }
"/*"                              { BEGIN(INCOMMENT); yymore(); break; }
"//".*                            { RETURN_TOKEN(COMMENT); }
^{DIRECTIVE}\n                    { RETURN_TOKEN(PREPROC); }
"["                               { RETURN_TOKEN(LBRACK); }
"]"                               { RETURN_TOKEN(RBRACK); }
"<:"                              { RETURN_TOKEN(LBRACK); }
":>"                              { RETURN_TOKEN(RBRACK); }
"("                               { RETURN_TOKEN(LPAREN); }
")"                               { RETURN_TOKEN(RPAREN); }
"{"                               { RETURN_TOKEN(LBRACE); }
"}"                               { RETURN_TOKEN(RBRACE); }
"<%"                              { RETURN_TOKEN(LBRACE); }
"%>"                              { RETURN_TOKEN(RBRACE); }
"."                               { RETURN_TOKEN(DOT); }
"->"                              { RETURN_TOKEN(PTR); }
"++"                              { RETURN_TOKEN(INC); }
"--"                              { RETURN_TOKEN(DEC); }
"&"                               { RETURN_TOKEN(BAND); }
"*"                               { RETURN_TOKEN(TIMES); }
"+"                               { RETURN_TOKEN(PLUS); }
"-"                               { RETURN_TOKEN(MINUS); }
"~"                               { RETURN_TOKEN(BNOT); }
"!"                               { RETURN_TOKEN(NOT); }
"/"                               { RETURN_TOKEN(DIV); }
"%"                               { RETURN_TOKEN(MOD); }
"<<"                              { RETURN_TOKEN(LSHIFT); }
">>"                              { RETURN_TOKEN(RSHIFT); }
"<"                               { RETURN_TOKEN(LT); }
">"                               { RETURN_TOKEN(GT); }
```

```
"<="                                    { RETURN_TOKEN(LE); }
">="                                    { RETURN_TOKEN(GE); }
"=="                                    { RETURN_TOKEN(EQ); }
"!="                                    { RETURN_TOKEN(NE); }
"^"                                     { RETURN_TOKEN(BXOR); }
"|"                                     { RETURN_TOKEN(BOR); }
"&&"                                    { RETURN_TOKEN(LAND); }
"||"                                    { RETURN_TOKEN(LOR); }
"?"                                     { RETURN_TOKEN(QUESTION); }
":"                                     { RETURN_TOKEN(COLON); }
";"                                     { RETURN_TOKEN(SEMI); }
"..."                                   { RETURN_TOKEN(ELLIPSIS); }
"="                                     { RETURN_TOKEN(ASSIGN); }
"*="                                    { RETURN_TOKEN(MULASSIGN); }
"/="                                    { RETURN_TOKEN(DIVASSIGN); }
"%="                                    { RETURN_TOKEN(MODASSIGN); }
"+="                                    { RETURN_TOKEN(ADDASSIGN); }
"-="                                    { RETURN_TOKEN(SUBASSIGN); }
"<<="                                   { RETURN_TOKEN(SHLASSIGN); }
">>="                                   { RETURN_TOKEN(SHRASSIGN); }
"&="                                    { RETURN_TOKEN(ANDASSIGN); }
"^="                                    { RETURN_TOKEN(XORASSIGN); }
"|="                                    { RETURN_TOKEN(ORASSIGN); }
","                                     { RETURN_TOKEN(COMMA); }
{INTEGER}                               { RETURN_TOKEN(INT_CONST); }
{INTEGER}[Uu]                           { RETURN_TOKEN(INT_CONST_U); }
{INTEGER}[Ll]                           { RETURN_TOKEN(INT_CONST_L); }
{INTEGER}([Uu][Ll]|[Ll][Uu])            { RETURN_TOKEN(INT_CONST_UL); }
{INTEGER}(ll|LL)                        { RETURN_TOKEN(INT_CONST_LL); }
{INTEGER}([Uu](ll|LL)|(ll|LL)[Uu])      { RETURN_TOKEN(INT_CONST_ULL); }
{FLOAT}[fF]                             { RETURN_TOKEN(FLOAT_CONST); }
{FLOAT}                                 { RETURN_TOKEN(DOUBLE_CONST); }
{FLOAT}[lL]                             { RETURN_TOKEN(LONGDOUBLE_CONST); }
{STRING}                                { RETURN_TOKEN(STR_CONST); }
L{STRING}                               { RETURN_TOKEN(WIDE_STR_CONST); }
{CHARLIT}                               { RETURN_TOKEN(CHAR_CONST); }
L{CHARLIT}                              { RETURN_TOKEN(WIDE_CHAR_CONST); }
auto                                    { RETURN_TOKEN(AUTO);}
break                                   { RETURN_TOKEN(BREAK);}
case                                    { RETURN_TOKEN(CASE);}
char                                    { RETURN_TOKEN(CLANGCHAR);}
const                                   { RETURN_TOKEN(CONST);}
continue                                { RETURN_TOKEN(CONTINUE);}
default                                 { RETURN_TOKEN(DEFAULT);}
do                                      { RETURN_TOKEN(DO);}
double                                  { RETURN_TOKEN(DOUBLE);}
else                                    { RETURN_TOKEN(ELSE);}
enum                                    { RETURN_TOKEN(ENUM);}
extern                                  { RETURN_TOKEN(EXTERN);}
float                                   { RETURN_TOKEN(CLANGFLOAT);}
for                                     { RETURN_TOKEN(FOR);}
goto                                    { RETURN_TOKEN(GOTO);}
```

```
if                                  { RETURN_TOKEN(IF);}
inline                              { RETURN_TOKEN(INLINE);}
int                                 { RETURN_TOKEN(CLANGINT);}
long                                { RETURN_TOKEN(CLANGLONG);}
register                            { RETURN_TOKEN(REGISTER);}
restrict                            { RETURN_TOKEN(RESTRICT);}
return                              { RETURN_TOKEN(RETURN);}
short                               { RETURN_TOKEN(CLANGSHORT);}
signed                              { RETURN_TOKEN(SIGNED);}
sizeof                              { RETURN_TOKEN(SIZEOF);
static                              { RETURN_TOKEN(STATIC);}
struct                              { RETURN_TOKEN(STRUCT);}
switch                              { RETURN_TOKEN(SWITCH);}
typedef                             { RETURN_TOKEN(TYPEDEF);}
union                               { RETURN_TOKEN(UNION);}
unsigned                            { RETURN_TOKEN(UNSIGNED);}
void                                { RETURN_TOKEN(VOID);}
volatile                            { RETURN_TOKEN(VOLATILE);}
while                               { RETURN_TOKEN(WHILE);}
_Bool                               { RETURN_TOKEN(_BOOL);}
_Complex                            { RETURN_TOKEN(_COMPLEX);}
_Imaginary                          { RETURN_TOKEN(_IMAGINARY);}
{IDENT}                             { RETURN_TOKEN(IDSYM); }
<*>.|\n                              ERROR_ACTION;
%%
```

B EBNF Grammar

This appendix summarizes the grammar used by our language module, based on the C99 standard grammar (§ A.2) with additions for non-preprocessed code. In the grammar below, the notations $x?$, x^*, and x^+ mean zero or one, zero or more, or one or more $x$es, respectively, as in standard EBNF. The notation x_a^+ means one or more $x$es, separated by as if there is more than one, and similarly x_a^*.

trans_unit ::= *ext_decl*$^+$

ext_decl ::= *decl_spec*$^+$ *declr* *decl** *comp_stmt* | *decl* | PREPROC

decl_spec ::= *stor_class_spec* | *type_spec* | *type_qual* | `inline`

decl ::= *decl_spec*$^+$ *init_declr*$^*_{','}$ ';'

stor_class_spec ::= `auto` | `register` | `static` | `extern` | `typedef`

type_name ::= *type_spec_qual*$^+$ *abst_declr*?

type_spec_qual ::= *type_spec* | *type_qual*

type_spec ::= `void` | `char` | `short` | `int` | `long` | `float`
| `double` | `signed` | `unsigned` | `_Bool` | `_Complex`
| `_Imaginary` | `struct` IDSYM? '{' *struct_decl*$^+$ '}'
| `union` IDSYM? '{' *struct_decl*$^+$ '}' | `struct` IDSYM
| `union` IDSYM | `enum` IDSYM? '{' *enum*$^+_{','}$ ','? '}'
| `enum` IDSYM | IDSYM

type_qual ::= `const` | `volatile` | `restrict` *init_declr* ::= *declr* ('=' *init*)?

designator ::= '[' *expr* ']' | '.' IDSYM

designation ::= *designator*$^+$ '='

inner_init ::= *designation*? *init*

init ::= *assign_expr* | '{' *inner_init*$^{+}_{','}$ ','? '}'

enum ::= IDSYM ('=' *expr*)?

declr ::= *pointer*? *direct_declr*

abst_declr ::= *pointer* | *pointer*? *direct_abst_declr*

direct_declr ::= IDSYM | '(' *pointer*? *direct_declr* ')'
  | *direct_declr* '[' *type_qual** *assign_expr*? ']'
  | *direct_declr* '[' *type_qual** `static` *type_qual** *assign_expr* ']'
  | *direct_declr* '[' *type_qual** '*' ']'
  | *direct_declr* '(' *param_type_list* ')'
  | *direct_declr* '(' IDSYM$^{*}_{','}$ ')'

direct_abst_declr ::= '(' *abst_declr* ')' | *direct_abst_declr*? '[' *assign_expr*? ']'
  | *direct_abst_declr*? '[' '*' ']'
  | *direct_abst_declr* '(' *param_type_list*? ')'
  | '(' *param_type_list*? ')'

struct_decl ::= *type_spec_qual*$^+$ *struct_declr*$^{+}_{','}$ ';' | PREPROC

struct_declr ::= *declr* | *declr*? ':' *expr*

pointer ::= '*' *type_qual** *pointer*?

param_decl ::= *decl_spec*$^+$ *declr* | *decl_spec*$^+$ *abst_declr*?

param_type_list ::= *param_decl*$^{+}_{','}$ (',' '...')?

for_init ::= *exprs*? ';' | *decl*

stmt ::= IDSYM ':' *stmt* | `case` *expr* ':' *stmt* | `default` ':' *stmt*
  | *exprs*? ';' | *comp_stmt* | `if` '(' *exprs* ')' *stmt*
  | `if` '(' *exprs* ')' *stmt* `else` *stmt* | `switch` '(' *exprs* ')' *stmt*
  | `while` '(' *exprs* ')' *stmt* | `do` *stmt* `while` '(' *exprs* ')' ';'
  | `for` '(' *for_init* *exprs*? ';' *exprs*? ')' *stmt*
  | `goto` IDSYM ';' | `continue` ';' | `break` ';'
  | `return` *exprs*? ';' | PREPROC

block_item ::= *decl* | *stmt*

comp_stmt ::= '{' *block_item** '}'

exprs ::= *exprs* ',' *assign_expr* | *assign_expr*

arg_expr ::= *assign_expr* | *type_name*

assign_expr ::= *expr* '=' *assign_expr* | *expr* '*=' *assign_expr*
  | *expr* '/=' *assign_expr* | *expr* '%=' *assign_expr*
  | *expr* '+=' *assign_expr* | *expr* '-=' *assign_expr*
  | *expr* '<<=' *assign_expr* | *expr* '>>=' *assign_expr*
  | *expr* '&=' *assign_expr* | *expr* '^=' *assign_expr*
  | *expr* '|=' *assign_expr* | *expr*

expr ::= *expr* '?' *exprs* ':' *expr* | *expr* '||' *expr* | *expr* '&&' *expr*
| *expr* '|' *expr* | *expr* '^' *expr* | *expr* '&' *expr* | *expr* '==' *expr*
| *expr* '!=' *expr* | *expr* '<' *expr* | *expr* '>' *expr*
| *expr* '<=' *expr* | *expr* '>=' *expr* | *expr* '<<' *expr*
| *expr* '>>' *expr* | *expr* '+' *expr* | *expr* '-' *expr* | *expr* '*' *expr*
| *expr* '/' *expr* | *expr* '%' *expr* | '(' *type_name* ')' *expr*
| '&' *expr* | '+' *expr* | '-' *expr* | '~' *expr* | '!' *expr* | '*' *expr*
| *expr* '(' $arg\_expr^{*}_{,}$ ')' | '++' *expr* | '--' *expr* | *expr* '++'
| *expr* '--' | `sizeof` *expr* | `sizeof` '(' *type_name* ')'
| '(' *exprs* ')' | *int_const* | *char_const* | *float_const*
| *str_const* | '(' *type_name* ')' '{' $inner\_init^{+}_{,}$ ','? '}'
| IDSYM | *expr* '.' IDSYM | *expr* '->' IDSYM
| *expr* '[' *exprs* ']'

int_const ::= INT_CONST | INT_CONST_U | INT_CONST_L
| INT_CONST_UL | INT_CONST_LL | INT_CONST_ULL

str_const ::= $\mathrm{STR\_CONST}^{+}$ | $\mathrm{WIDE\_STR\_CONST}^{+}$

char_const ::= CHAR_CONST | WIDE_CHAR_CONST

float_const ::= FLOAT_CONST | DOUBLE_CONST | LONGDOUBLE_CONST

C Code Examples

C.1 Syntax Specification

The syntax specification, processed by Harmonia's `ladle2` program, uses an extension of Bison's syntax [CP99], extended with EBNF operators, names for productions (after =>), which turn into the names of node classes, and accessor names for items (before :), which become the names of accessor methods on those objects. The following example shows the grammar of statements (compare to the EBNF above):

```
stmt: label:IDSYM c:':' stmt:stmt                             => "LabeledStatement"
    | kw:CASE expr:expr c:':' stmt:stmt                       => "CaseLabelStatement"
    | kw:DEFAULT c:':' stmt:stmt                              => "DefaultLabelStatement"
    | expr:(expr:exprs)? semi:';'                             => "ExprStatement"
    | stmt:comp_stmt                                          => "CompoundStatement"
    | kw:IF l:'(' pred:exprs r:')' t_stmt:stmt %prec BARE_IF_PREC
                                                              => "IfNoElseStatement"
    | k1:IF l:'(' pred:exprs r:')' t_stmt:stmt
                         k2:ELSE f_stmt:stmt                  => "IfElseStatement"
    | kw:SWITCH l:'(' val:exprs r:')' stmt:stmt               => "SwitchStatement"
    | kw:WHILE l:'(' val:exprs r:')' stmt:stmt                => "WhileStatement"
    | k1:DO stmt:stmt k2:WHILE l:'(' val:exprs r:')' semi:';' => "DoWhileStatement"
    | kw:FOR l:'(' init:for_init pred:(expr:exprs)? semi:';'
                   inc:(expr:exprs)? r:')' stmt:stmt          => "ForStatement"
    | kw:GOTO label:IDSYM semi:';'                            => "GotoStatement"
    | kw:CONTINUE semi:';'                                    => "ContinueStatement"
    | kw:BREAK semi:';'                                       => "BreakStatement"
    | kw:RETURN ret:(expr:exprs)? semi:';'                    => "ReturnStatement"
    | p:PREPROC                                               => "PreprocStatement"
    ;
```

C.2 Name Resolution

The `resolve_indents()` function, which is defined for each kind of node, fills out the `idents` table it takes as an argument and watches for errors that would invalidate an alternative. If it finds such an error, it returns `false` up the call tree to the closest symbol node. Here is the implementation for uses of typedef names:

```
// type_spec -> IDSYM:type_name
operator TypedefTypeSpec extends ParentNode, type_spec, GLRStateMixin {
    // Pointer to the Declarator that defined us
    public versioned slot Vptr definition = NULL;
    versioned Semantics attribute definition
        = TypedVal((Node*)definition.value(vg(), gvid));

    public virtual method bool resolve_idents(ResolveContext *ctx,
                                              IdentTable *idents) {
        // Get our string
        PooledString id = get_type_name()->pooled_string();

        // Our kid is a typedef name, at least from this parent
        get_type_name()->set_subspecies(ctx, IDSYM::TYPEDEF_NAME);
        if (!idents->resolve_ident(id)) {
            // No definition is scope
            assert(!idents->has_ident_binding(id));
            ambig_semant_error(ctx, this, "undefined typedef name"
                               " %s", id.chars());
            return !ctx->inside_ambig;
        }
        IdentInfo *info = idents->resolve_ident(id);
        assert(info);
        if (info->type != IdentInfo::TYPEDEF_ID) {
            // Wrong kind of previous definition
            ambig_semant_error(ctx, this, "identifier is not "
                               "a typedef name: %s", id.chars());
            return !ctx->inside_ambig;
        } else {
            // Looks good
            definition.set_value(vg(), info->definition);
            compute_synth_attrs(true); // Set change bits
            return true;
        }
    }
}
```

C.3 Type Checking

The type checking of subtraction is typical among binary operators. First we recursively check the types of the two arguments, then check that they have the right expression classes, and then check for one of three allowed type patterns: the difference of two arithmetic types, of a pointer minus an integer, or of two compatible pointer types.

```
operator Subtract extends expr {
    public virtual method ExprType determine_type(TypecheckContext *ctx) {
        ExprType left, right, tmp;
        if (!(left = get_left()->determine_type(ctx)))
```

```
    return ExprType(ERROR_CLASS, 0);
if (!(right = get_right()->determine_type(ctx)))
    return ExprType(ERROR_CLASS, 0);
if ((tmp = left.to_rvalue())) {
    left = tmp;
} else {
    set_semant_error(this, "Can't use %s as left arg to `-'",
                     left.name().c_str());
}
if ((tmp = right.to_rvalue())) {
    right = tmp;
} else {
    set_semant_error(this, "Can't use %s as right arg to `-'",
                     right.name().c_str());
}
ExprType result;
if (left.ty->is_arithmetic() && right.ty->is_arithmetic()) {
    // Difference of two arithmetic types is the usual converted
    // result type
    ArithmeticType *left_converted = (ArithmeticType*)left.ty;
    ArithmeticType *right_converted = (ArithmeticType*)right.ty;
    result.ty =
        ArithmeticType::usual_conv(&left_converted, &right_converted);
    if (left.is_int_const() && right.is_int_const())
        result.cl = INTEGER_CONST;
    else if (left.is_arith_const() && right.is_arith_const())
        result.cl = ARITH_CONST;
    else
        result.cl = RVALUE;
} else if (right.ty->is_integer() && left.ty->is_ptr_to_object()) {
    // Pointer minus an integer is a pointer of the same type
    result.ty = left.ty;
    if (right.is_int_const() && left.is_addr_const())
        result.cl = CONSTANT;
    else
        result.cl = RVALUE;
} else if (left.ty->is_pointer() && right.ty->is_pointer()) {
    // Pointer minus a pointer is a (particular) integer type
    CType *left_ref_uq = ((PointerType *)left.ty)->referent_type
        ->unqualified();
    CType *right_ref_uq = ((PointerType *)right.ty)->referent_type
        ->unqualified();
    if (!CType::composite_type(left_ref_uq, right_ref_uq))
        // Can't do 'int *' - 'char *'
        set_semant_error(this, "Incompatible pointer types in "
                         "difference: %s - %s",
                         left.ty->name().c_str(),
                         right.ty->name().c_str());
    result.ty = ctx->the_PtrdiffType;
    result.cl = RVALUE;
} else {
    // Any other type combination is illegal
```

```
            set_semant_error(this, "Wrong types to ‘-’: %s - %s",
                             left.ty->name().c_str(),
                             right.ty->name().c_str());
            result.ty = 0;
            result.cl = ERROR_CLASS;
            return result;
        }
        string msg = "Difference has type " + result.ty->name();
        set_string_prop(MOUSE_OVER_MSG, msg.c_str());
        compute_synth_attrs(true);
        return result;
    }
}
```

D Source Code Availability

The complete source code of the flex, grammar and semantic specifications is also available online [McC02].

References

[Bos01] Marat Boshernitsan. Harmonia: A flexible framework for constructing interactive language-based programming tools. Technical Report CSD-01-1149, Computer Science Division, EECS Department, University of California, Berkeley, June 2001.

[CP99] Robert Corbett and Project GNU. `bison` release 1.28, 1999.

[EBN97] Michael Ernst, Greg J. Badros, and David Notkin. An empirical analysis of C preprocessor use. Technical Report TR-97-04-06, University of Washington, 1997.

[Har] Harmonia Research Group. Harmonia project home page. `http://www.cs.berkeley.edu/~harmonia`.

[JTC99a] JTC1/SC22/WG14. *International Standard 9899: Programming languages — C*. ISO/IEC, 1999.

[JTC99b] JTC1/SC22/WG14. *Rationale for International Standard — Programming languages — C*. Number N897 in working group documents. ISO/IEC, draft revision 2 edition, October 1999.

[KR88] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall, second edition, 1988.

[McC02] Stephen McCamant. C language module source files, 2002. `http://www.cs.berkeley.edu/~harmonia/projects/c-language/index.html`.

[Pax95] Vern Paxson. `flex 2.5.4` man pages, November 1995. Free Software Foundation.

[PB89] P.J. Plauger and Jim Brodie. *Programmer's Quick Reference Series: Standard C*. Microsoft Press, 1989.

[Rek92] Jan Rekers. *Parser Generation for Interactive Environments*. Ph.D. dissertation, University of Amsterdam, 1992.

[Too02] Michael Toomim. *Harmonia-Mode User's Guide*, 2002. `http://www.cs.berkeley.edu/~harmonia/projects/harmonia-mode/introduction.html`.

[WG97] Tim A. Wagner and Susan L. Graham. Incremental analysis of real programming languages. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, pages 31–43, Las Vegas, NV, June 1997.