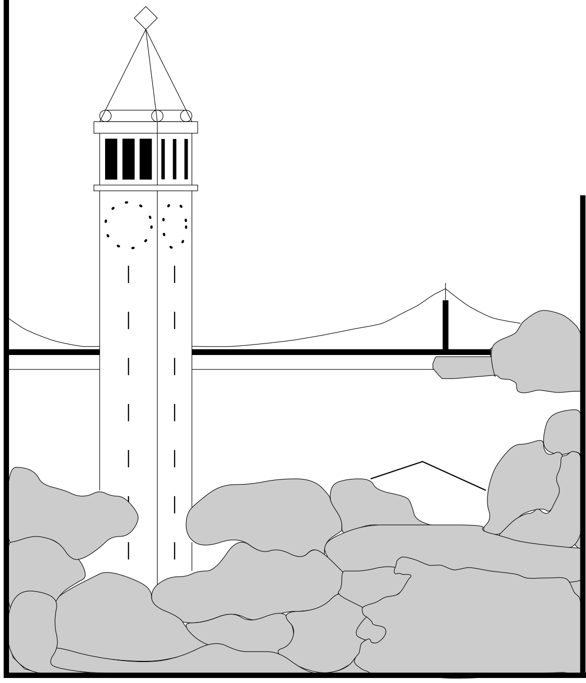
The High-Level Intermediate Language L

Luigi Semenzato



Report No. UCB/CSD 93-760

July 25, 1993

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

Abstract

L is an extensible high-level intermediate language. Its intended application is the construction of compiler back-ends and run-time libraries for high-level programming languages with complex built-in data structures, such as Hilfinger and Colella's FIDIL language. L provides a general-purpose abstract machine, Basil, and machinery for extending it. L has been used to define the abstract machine INFIDEL, described in a separate report. We call L a "high-level" intermediate language for several reasons. First, the target machine of L is a generic processor, or multiprocessor, programmable in C. The "assembly language" used by L is C. Second, L strives to be usable both as an intermediate language and a programming language. The typical extension of L is expected to have a large run-time library, also written in L. Many operators in the extended abstract machine are implemented by function calls to this library.

The High-Level Intermediate Language L

Luigi Semenzato

July 25, 1993

Research supported at UC Berkeley by DARPA and the National Science Foundation under grant DMS-8919074.

Contents

Chapter 1

Overview

L is an extensible high-level intermediate language. Its intended application is the construction of compiler back-ends and run-time libraries for very-high-level programming languages, such as FIDIL [HC89]. L provides a general-purpose abstract machine, Basil, and machinery for extending it. L has been used to define the abstract machine INFIDEL, described separately [Sem93].

We define L a *high-level* intermediate language for several reasons. First, the target machine of L is a generic processor, or multiprocessor, programmable in C. The "assembly language" used by L is C. Second, L strives to be usable both as an intermediate language and a programming language. The typical extension of L is expected to have a large run-time library, also written in L. Many operators in the extended abstract machine are implemented by function calls to this library.

The basic L abstract machine, Basil, is essentially C with a few additions, most notably automatic storage management. Basil can be extended as needed by adding new operators and types.

New operators in the extended machine define functionality not present in Basil. This functionality can be achieved in two ways. Expressions involving extended machine operators can be rewritten using the L transformation system; or they can be translated to calls to a run-time library.

L is useful when the ideal target machine has a large number of primitive operators and types. L operators are lightweight: they are easy to define and transform, or implement. The rewriting system is similar to Lisp macro expansion; it is powerful but at the same time reasonably simple to understand and use. Many type-specific optimizations can be performed during the transformation. A subset of Basil (also extensible), called *pure Basil*, is executable during compilation, thus providing a mechanism for compile-time constant folding.

The overhead of learning to write L code or transformations is small for those who know C and Common Lisp. The semantics of the primitive abstract machine Basil are those of C. The syntax is that of Common Lisp. The transformation system resembles Lisp macros. The programming language used to specify transformations is Common Lisp.

We introduce L by first describing the Basil virtual machine, and then showing how it is extended. At the end we include a reference manual for L/Basil. We assume the reader to be familiar with C and Common Lisp.

Chapter 2

Basil

Basil is a high-level universal assembly language. Roughly speaking, it has the semantics of C and the syntax of Lisp.

The resemblance to C is dictated by engineering reasons. Basil is translated into C in a straightforward fashion. There are high-quality C compilers available for every processor on the market today. This is likely to remain true for the foreseeable future. These C compilers are finely tuned to provide processor-specific optimizations (such as register allocation) that would be expensive to reproduce. Many architectural features that do not fit well within the C model (for example, vector or parallel processing) are easily accessible from C, typically through architecture-specific language extensions. There is, of course, a partial or total loss of portability when using such features. Basil makes no attempt to hide architectural differences beyond what C does. This is the task of the extended virtual machine. The transformation from the extended machine to Basil is then machine dependent; but there are so many machine-independent parts that any other approach is wasteful.

The use of Lisp syntax is also dictated by engineering reasons. The extensions mechanisms of L involve code analysis and rewriting. It is important that the visible representation of code closely match the internal one. It would be difficult to conceive a more concise syntax than that of Lisp for this purpose. And although this syntax looks undeniably hostile to the uninitiated, it is simple and consistent, and it takes surprisingly little training to acquire familiarity with it.

Basil serves not only as a target language, but also as a programming language on its own, for writing run-time libraries. From this point of view, Basil has a code transformation mechanism similar to that of Lisp-like languages. In most languages an abstract data type is defined by a set of types and a set of procedures operating on objects of those types. To this, Basil adds type-specific optimizing transformations. Thus transformation in Basil have a dual purpose: they define the mapping of a high-level virtual machine into a low-level one (vertical transformations), and can optimize library code within the same machine (horizontal transformations). The transformation system is presented in chapter 3.

Another strong point of Basil is garbage collection. Its availability encourages a Lisp-like programming style, in which it is convenient to define and manipulate complex descriptors. Automatic garbage collection is not always the best solution for large objects, such as those often found in scientific programs; for this reason the `malloc` and `free` primitives of C are also available. The garbage collector is meant to be used for small objects, typically used in descriptors. The two mechanisms coexist gracefully and with a small distributed overhead.

This chapter attempts to give a concise description of Basil. A complete description of all available Basil operators is given in Appendix B.

2.1 A Tiny Example

A Basil program to compute a factorial looks like this:

```
(deflfun (int fact) ((int x))
  (return
   (if (= x 0)
     1
     (* x (fact (- x 1))))))
```

and is compiled into this C code (or something very similar):

```
int fact(int x)
{
    return x == 0 ? 1 : x * fact(x - 1);
}
```

The Basil translator attempts to produce readable C code that has a clear correspondence to its Basil source: for instance, the name of a Basil variable is preserved, if possible, in the translation. When a conflict may arise, the name is modified as little as possible, typically by adding a numeric suffix. Debugging is done directly on the C output.[1] Unlike C, expressions and statements may be freely intermixed in Basil, and blocks may return values.

2.2 Syntax

A Basil expression is either a list, a symbol, or another Lisp object representing a constant (integer, float, structure, etc.). When it is a list, its first element must be either a symbol, called the *operator* of the expression, or a form whose first element is `slambda`, which stands for *simple-lambda*. In the latter case, the form is equivalent to the result of applying the simple-lambda to the arguments (the rest of the form), using simple-lambda substitution rules. By these rules, *every* instance of a formal in the `slambda` body is replaced by the corresponding unevaluated actual. This is done with complete disregard of position. For example, in Common Lisp we have:

`((lambda (f x y) (f x y)) 4 5 6)` → `(f 5 6)`

but in Basil we get:

`((slambda (f x y) (f x y)) 4 5 6)` → `(4 5 6)`

2.3 Values and Locations

As in C, certain Basil expressions have a meaning that is context-dependent, and evaluate either to a *location* (a container where a value can be stored) or a *value* (a data object of some type). Variables are interpretable both ways; so are the array reference `aref` and record reference operators. The positions where such expressions represent locations are the left-hand side of an assignment, a pass-by-reference argument, and the initializing expression in a `reflet`.

Internally, such expressions are disambiguated during expansion. There are different array reference and record reference operators for the value and the location case; and variable references are either of the form `(ref` *x*`)`, meaning the location named by *x*, or `(val` *x*`)`, meaning the value contained in that location.

[1] It is possible to insert debugging information into Basil and have it passed to the C compiler, but not convenient to do so manually. The debugging facilities are intended for Basil code generated by a compiler front-end (see section 2.11).

2.4 Local Declarations

The `let` form is the main local variable-binding construct. It has a syntax similar to Lisp's `let`:

(let ({*clause*}*) {*expression*}$^+$)

A *clause* has the form ((*type name*) [*initial-value*]). The general rule is that the syntax of a Basil name-binding form can be derived from the equivalent Lisp form by replacing *name*, where *name* is the entity being defined, with (*type name*). In most cases a shorthand is available: using *name* where (*type name*) is expected stands for (t *name*), where `t` is the *universal union pointer type* (see section 2.7). The precise shorthand rules for `let` are:

$$\begin{array}{rcl} x & \longrightarrow & ((\texttt{t}\ x)) \\ (x\ e) & \longrightarrow & ((\texttt{t}\ x)\ e) \end{array}$$

The value returned by a `let` form is the value of the last expression in the body. If the last expression does not produce a value (has `void` type), the `let` form does not have a value either.

There are three variations of `let`: `reflet`, `clet` and `slet`.

A `reflet` form declares *reference constants*. They are names for assignable locations, like in `C++`. An initializing expression must be provided for each constant, and it must be interpretable as a location.

The `clet` operator declares constants. The initializing expression in each clause must be present. If the expression has a known value at compile time, then the constant is replaced by that value in the body, but only where it is useful to do so (see section 3.3.4). Operations that change the value of the constant in the body have unpredictable effects.

The `slet` operator defines symbol macros. A clause has the structure (*name form*). Occurrences of *name* in the body are replaced by *form*.

Local operators are defined with `loplet` and `soplet`. They are described in section 3.2, together with their global equivalents `deflop` and `defsop`. Operators should be defined freely, not just for extending the virtual machine interface, but for expressing internal abstractions whenever convenient.

2.5 Global Declarations

The operators `deflfun` and `deflfun-pure` define L functions. `deflvar` defines global variables, and `defsconstant` global symbol macros. `deflstruct` defines structure types (see section 2.7.2). There are also operator-defining operators, described in section 3.2.

A `deflfun` form has the following syntax:

`(deflfun (`*type name*`) (`*arg*$*$`)` *expression*$+$`)`

Each *arg* has the syntax `(`$\{$`val | ref`$\}$ *type name*`)`. These shorthands are available:

x	$\longrightarrow$	`(val t` *x*`)`
`(`*type name*`)`	$\longrightarrow$	`(val` *type name*`)`

Unlike Lisp, and rather more like C, a function must use an explicit `return` statement to return a value.[2]

When the operator `deflfun-pure` is used instead of `deflfun`, it means two things. It tells the L compiler that the function is side-effect free, or close enough for our purposes; and it instructs it to compile a version of the function that can be executed during compilation. So the function is translated into Lisp as well as C. The Lisp version will execute successfully only if the function has been written in pure Basil (see section 2.10). Any call to a pure function whose arguments have a compile-time constant value results in the function being evaluated at compile time, and the function-call form being replaced by the result.

2.6 Control Statements

Basil has a few control statements that derive from C, and a few from Lisp. Here we give a brief overview of them. The reference manual has all the details.

2.6.1 Loops

The `for` statement is the one from C. The control variable in a `for` needs to be declared separately. The `dotimes` statement instead is from Lisp, and creates a

[2]This choice is not strictly due to laziness: it helps in preserving a likeness between L source and C output, therefore making debugging easier.

new binding for the control variable. The control clause of `dotimes` may also specify under which circumstances the body should be expanded inline. The `loop` statement is from Lisp, while the `while` statement is from C. None of these statements returns a value.

2.6.2 Conditionals

The `if` statement is the one of Lisp, and may return a value. When translating into C, the L compiler decides whether to use the C `if` or the C conditional expression (question mark-colon). The `case` statement is also from Lisp, and may return a value.

2.7 Types

The types of Basil are those of C, with the addition of the type `t`, the *universal union pointer*. The L programmer can also define new type expressions (see section 3.3.2).

A caveat: the L compiler currently does not do any type checking—not counting, of course, that done by the C compiler on the output files.

2.7.1 Universal Union Pointers

This type is used to emulate the behavior of dynamically-typed Lisp variables. It is called `t`, which in Lisp denotes the union of all types. In a correct Basil program, a variable with type `t` contains a pointer to a heap-allocated instance of a structure defined by a `deflstruct` form (see section 2.7.2). Such structure instances are garbage-collected, just as in Lisp.

2.7.2 Structures

The L form `deflstruct` defines new structure types, similarly to the `defstruct` form of Lisp, and the structure declaration of C. Processing this form produces several new operators related to the structure. As an example, if the structure has name `s` and fields `x` and `y`, the following operators are defined:

`make-s`. The constructor function. This function takes two arguments, one for each slot in the structure, and returns a boxed instance of that structure (that is, a pointer to a heap-allocated instance) with its slots initialized from the argument values. This instance is garbage-collected. The return type of `make-s` is `t`, and the returned value should be stored in a variable, or other container, of type `t`, or other type that is eventually transformed into `t`.

`s-p`. The type predicate. It takes a universal pointer as its argument, an returns true if the pointer points to an object of type `s`.

`s-x`, `s-y`. The boxed field accessors. They take a universal union pointer and return the location of each slot in the boxed structure instance. This location is interpreted as a value in the proper contexts.

`s-x*`, `s-y*`. The unboxed field accessors. These are the equivalent of the C structure field selectors `.x` and `.y`.

A few more operators and Lisp functions are defined for each structure. They are used to purify Basil code that, for efficiency considerations, is impure, but must execute at compile time anyhow. They are described in section 2.10.

2.8 Lists

Basil has a few operators to manipulate Lisp-style lists. They have rather predictable names (`cons`, `list`, `nconc`, `append`, `mapcar`, etc.). The main difference is that operators like `mapcar` and `mapcan` take an operator, instead of a function, as their second argument. By necessity then all list-mapping operators generate inline code. The operator may either be a symbol or an `slambda` form.

2.9 Separate Compilation

A Basil program consists of definitions contained in one or more files, also called `modules`. Some modules may use definitions from other modules: for instance, structure or operator definitions. If in order to process module x the compiler needs information contained in module y, then module y should contain the directive `(need` x`)`.

All global entities (global variables and functions) are visible across all modules, without the need for C-style header files (they are generated automatically). Section 3.1 describes the compilation process in some detail.

2.10 Pure Basil

Pure Basil is a subset of Basil that can be translated and executed in Lisp (it is *that* subset of Basil that can be *easily* translated into Lisp—again, an engineering choice). Its purpose is to avoid having to maintain two sets of source code for certain routines: one for the run-time library, the other for the compiler.

Of all Basil types, only pointer types are not available in pure Basil (except, of course, `t`). Structure types are fully supported, both boxed and unboxed (unboxed structures are emulated by boxed ones that aren't allowed to be shared through multiple pointers). Array types are also supported.

In some cases of practical significance, one would like to take advantage of C to implement certain low-level abstractions efficiently using pointer arithmetic. It is possible to maintain two separate implementations of these low-level abstractions, one for C, the other for Lisp, and use them in pure Basil code, as long as they have the same interface (see section 3.3.1). Target-specific transformations (`target-case`) are useful in this situation. The details of these are given in the reference manual.

2.11 Debugging

Because of the way the Lisp reader works,[3] it is not possible to debug Basil source code directly. One must use the C debugger on the C output, much in the same way one would use the assembly-language debugger on C code compiled without debugging information. My experience has shown that this is a fairly easy task, to the point that I often prefer to debug the C version of pure Basil functions, rather than the Lisp version (it's nice to have a choice—Lisp has better printing, but C has better single-stepping).

When Basil is used as a target language, it is possible to pass debugging information along. There are two types of information: *source-position* and *name* information.

2.11.1 Source Position

Source-position information associates a file name and a range of positions to Basil forms. It can be provided in two ways. If the compiler front end is written in Lisp, the Basil code produced by it is Lisp objects (lists, mostly). Each code object may

[3] And because I did not have a compelling desire to rewrite it.

be associated to a `source-position` structure. This contains a file name, and a starting and ending line and character positions. Example:

```
#s(source-position :file-name "source.file"
                   :from-line 10
                   :from-char 0
                   :to-line 12
                   :to-char 4)
```

The association is done by the function `(setf source-position)`.

If the front end is not written in Lisp, it can communicate with the L system via ASCII text.[4] The L reader assigns a special meaning to the dispatching character macro `#!`. The construct

`#!` *form fl fc tl tc*

reads as *form*, and associates to it a `source-position` structure. The line and character positions are given by to *fl*, *fc*, *tl*, and *tc*. The file name is taken from the value of the Lisp variable `*source-name*`, which can be set wherever necessary with an escape-into-Lisp form (`eval`).

If the front end does not provide character positions, the syntax

`#2!` *form fl tl*

saves space and time.

As the compiler transforms Basil code into C, it attempts to maintain source-position information as much as possible.

2.11.2 Name

The `name-information` structure associates information to a source name. It has the following fields:

`scope` A `source-position` structure indicating the range of positions in the source file for which the association of the name with this structure is valid. It should be `nil` if the scope is global.

[4] A binary format could be used for higher efficiency. A format that could make sense is that of a Lisp fast-load file, even though it is system-dependent, and in most cases not documented.

`definition` A `source-position` structure pointing to the definition of this object.

`target-name` The name used by the front-end to refer to the entity named by the source name.

`other` An association list containing other information to be passed to the debugger (for instance, the source type).

Name information is passed to the debugger, possibly after changing the target name. The function `(setf name-information)` is used to associate a name-information structure to a name.

2.12 Caveats and Limitations

2.12.1 Debugging

None of the facilities described in section 2.11 on debugging is currently implemented.

2.12.2 Pointer Discipline

The implementation of the conservative C garbage collector requires some discipline on the programmer's side.

The garbage collector considers an object to be in use if it can find a pointer to the beginning of that object from some memory location that is potentially in use. Such objects are left alone; the others may be reclaimed.

There are two situations in which the collector may recycle objects that the programmer considers alive. We present them, together with the precautions that must be taken.

Pointers to the middle of objects

The first problem occurs when the program allocates an object and subsequently maintains a pointer to the middle of the object as the only reference to it. The system assumes that such object is unreachable, and may incorrectly free it. The program must maintain a pointer to the beginning of the object to insure its survival.

In some rare cases, a program may be optimized so that even when it appears to maintain a variable pointing to the beginning of an object, the variable is dead and its value discarded. This may occur when the only use of the object consists in writing and reading values in some location at a non-null offset inside it.

It is possible to modify the garbage collector to fix this, but we haven't done it, because this situation is easy to avoid, and the modification would both slow down the collector and make it more conservative (and therefore less efficient).

Manually vs. automatically collected objects

Pointers to a garbage-collected object must always be stored in memory locations (or processor registers) that the collector will scan during its search for potentially in-use objects (the mark phase). The following areas of memory are *not* scanned: blocks allocated with `malloc` (as opposed to `malloc-gc`), and static data areas other than those listed in a data structure called the *static root table*.

Global variables of type `t`, or other types that may contain `t` pointers, are automatically placed in the static root table, so the programmer must be careful only when using casts. But if for instance the programmer needs temporary space for a variable-length array of `t`, it may be unsafe to use `malloc` for this purpose, and `malloc-gc` should be used instead. In this case it is a good practice to use `free-gc` to free the temporary array, instead of letting the collector reclaim it. This insures that a pointer to the beginning of the object does not become dead before the object itself.

2.12.3 Static Analysis

The L compiler performs very little static analysis. L programs are (mostly) statically typed, but the compiler does not check that declarations and uses are consistent. In most cases type mismatches result in errors during the C compilation; in some cases they cause run-time errors and the L compiler aborts. To debug in these situations, the compiler should be run within an interactive Lisp session.

Chapter 3

L Extension Mechanisms

L offers a framework for constructing high-level abstract machines. Such machines can be used as a target for a compiler, or programmed directly. They are built mainly by adding new operators to the basic abstract machine Basil, described in chapter 2. To explain how to define new operators, we need first to describe how the L compiler works.

3.1 The L Compiler

3.1.1 Phases

The L compiler is organized in *phases*. Every phase is composed of one or more passes. The first pass is called the *expansion pass*; it is followed by zero or more *ad-hoc passes*. Each pass takes a piece of L code, and returns it, possibly after transforming it and producing side effects.

Each phase is named by a Lisp keyword.[1] The compilation of Basil into C uses two phases: the `:basil` phase and the `:c` phase. Most of the compiling transformations occur in the expansion pass of each phase. The ad-hoc passes are reserved for transformations that are impossible or awkward to perform during expansion.

Basil can be extended by adding operators to the `:basil` phase, or by adding a new phase before that. The INFIDEL virtual machine is an extension of the `:basil` phase. At this moment we have no experience in adding new phases.

[1] We use keywords, instead of simple symbols, because the former are package-independent.

We suggest that part of the FIDIL compiler (the infamous "middle end") could be implemented as such.

3.1.2 The Expansion Pass

The expansion pass is where the L compiler does the bulk of its work. Operators may have an associated phase-specific expansion function. This function is defined in a manner similar to a Common Lisp macro, and it has similar semantics. During the expansion pass, each L form is visited, and if its operator has an associated expansion function, that function is called, passing the form to it. The form is replaced with the object returned by the expansion function. If this object is a form with a different operator, it is expanded again, if applicable. Then subexpressions in that form are recursively expanded. This top-down order can be altered by an expansion function that calls `expand` directly on subexpressions of the current form.

As the expansion rules imply, an expansion function may return a form with the same operator. However, expansion functions must be idempotent. The result of multiple applications to a given expression must be the same as for a single application: that is, a legal expansion function f satisfies $f(f(x)) \equiv f(x)$. For instance, the following definition for the hypothetical operator `op` is erroneous:

```
(deflop op (x)
  `(op (g ,x)))
```

When the expansion function returns `t` as its second value, the replacement and its subexpressions are assumed to be completely expanded. This is an important optimization, but it does not affect the semantics of expansion in a correct rewriting system.

This organization has two useful properties. One is that all or most of the information relative to an operator is contained in a single place, instead of being spread across the description of several transformations involving that operator. The other is that all the transformations are done in a single pass, instead of having several transformation passes that can interfere with each other, and that are difficult to order properly.

3.1.3 Code and Extensions

The unit of compilation in L is called the *module* and corresponds to a file, call it `mod.l`. A module contains one or more L forms. The compilation of each form produces *target code*, or *compiler extensions*, or both. We define compiler extension any information used by the compiler to process other forms. For instance, an operator definition is purely an extension. The definition of an L function is mostly code, but it also produces a small extension: the binding of the return type to the function name. This binding is needed to determine the type of a call to that function.

The compilation of a module produces a few files containing C code (`_mod.c`, `_mod.h`, and `_mod.d`), a file with debugging information (`_mod.db`), and a file containing Lisp code (`_mod.cl`) that, when loaded, recreates the compiler extensions of `mod.l`. For efficiency, the extensions file `_mod.cl` can be compiled. For more efficiency, it can also be preloaded into a Lisp image.

Expansion functions whose execution has side effects on the compiler state must also record it in the compiler-extension file, by placing in it a form that produces the same side effect. To help organizing this, the `extension` Lisp macro evaluates its argument and outputs it to the compiler extension file. The form is then replaced by its evaluated argument.

3.2 Operator Definition

The semantics of a new operator are specified in one of three ways: a rewriting rule, a library function, or an output function.

Operator rewriting is similar to Lisp macro expansion. The rewriting rule of an operator is a function that takes a form with that operator, and returns a replacement form. This function is called the *expansion function*. Unlike Lisp, an L expansion function may return a form with the same operator without causing an infinite loop.

Operators can also be implemented by library routines. Such operators may still have an associated rewriting rule, which are typically used to recognize special uses of that operator, whereas the library function implements the general case.

Operators that are neither rewritten nor implemented as library functions must have an output function. As an extension mechanism, the output function is meant to be used when the target architecture has features that are accessible only through syntactic extensions to C. The argument to an operator's output function is a form with that operator. The function outputs C code, and returns no values.

Two more attributes characterize an operator, the *type descriptor* and the *syntax template*.

The type descriptor specifies the type of an expression with that operator. It can be an L type expression, when the operator has a fixed type; a type-computing function, when the operator is generic or overloaded; or the `:expand` keyword, when the type should be computed by first expanding the expression. The function `l-type` returns the type of an expression according to these specifications.

The syntax template must be specified for operators that are not transformed during the expansion phase. It specifies the structure of an expression with that operator, and enables the L programmer to define the equivalent of Lisp special forms.

New operators are defined by the `deflop` macro. Because of its relevance, we include here the definition of `deflop` from the reference manual:

`deflop` *op args* `&key :type :template :output &rest` *body* [*Lisp macro*]

`deflop` defines a new operator with name `op`. *args* specifies the arguments of an expression with such operator, using the syntax of the argument list in a Common Lisp macro definition. *body* is the body of the expansion function of the operator. If there are no statements, the operator does not have an expansion function.

The `:type` keyword argument[2] can be:

- an L type expression. Any expression with this operator has the given L type.
- a Lisp function. This function takes a form with the given operator as argument, and it returns an L type expression;
- the `:expand` keyword. This indicates that to compute its type, the expression must be expanded first.
- the `:first`, `:second`, etc. keywords. This is a shorthand for a type function that returns the type of the first, second, etc. argument.

The default value for the `:type` argument is `:expand`.

The `:template` keyword argument is a form that specifies the structure and some semantics of an expression with that operator. Templates are used for various kinds of code traversal during and after the expansion phase. If an operator has an expansion function, and expands into a form with a different operator, then its template is useless.

A template is a list of template symbols and other templates. The template symbols have semantic significance as explained:

[2]These are not keyword arguments in the strict Common Lisp sense, because positionally the precede the body of the expansion function. However, they are parsed in the obvious way.

`expr`. An L expression that returns a value.

`location`. An L expression that returns a location: for instance, the left hand side of an assignment operator, or a pass-by-reference parameter position.

`name`. A symbol that is not evaluated: for instance, a variable in the declaration position of a `let` form, or other variable-binding constructs.

`*`. Some other kind of information associated with the form.

In addition, the following form has a special meaning:

`(list` *template*`)`. Zero or more occurrences of *template*. This may only appear as the last subform of a form.

The default value for the operator template is `(list expr)`.

The two variations of `deflop`, `deflop-c` and `deflop-lisp`, define an expansion function for the `:c` or `:lisp` phase. Type and template are not phase-specific.

3.3 Uses of the Expansion Mechanism

3.3.1 Conditional Expansion

The Lisp variable `*target*` is bound, during the `:basil` phase, to either `:c` or `:lisp`, depending whether the current expansion has C or Lisp as its target (pure Basil is compiled into both). A macro expansion function can check the value of `*target*` and return different code for each case.

The Basil operator `target-case` packages this functionality concisely. It takes two arguments. The first is code to use when compiling into C, the second when compiling into Lisp.

3.3.2 Types as Expressions

Because types are treated as expressions in L, the operator definition and rewriting mechanism can be used for types as well. For instance, the n-dimensional domain type in INFIDEL is defined by this simple code:

```
(deflop-c domain (n)
  (declare (ignore n))
  t)
```

This definition means that the type expression `(domain` n`)` is converted to `t` in the `:c` phase. Its effect is that domain values are represented by universal union pointers.

3.3.3 Compile-time Evaluation

Some information about the values of the operands of an expression may be available at compilation time. When *complete* information is available—that is, the values themselves are known—it may be convenient to evaluate the expression and replace it with its value.

This situation occurs frequently in INFIDEL, for several classes of *descriptor* objects. These descriptors are *data* objects describing *control*: on what data to operate, how to partition large objects, etc. A simple example of what could be considered a descriptor in FORTRAN is the upper bound of a `DO` loop with a constant lower bound. INFIDEL descriptors are more complex.

Expansion functions are responsible for folding expressions. Two Lisp functions are used as an aid to this purpose. The function `lconstantp` returns true if its argument is L code with a known constant value. The function `value-of` returns that value.

3.3.4 Demand-Driven Constant Propagation

Constant propagation is useful if the cost of reproducing a constant is lower than the cost of maintaining it (storing and reloading, or keeping it in a register). When this is not the case, constant propagation may still be useful when the knowledge of a constant value provides opportunities for further optimization. The L operator `clet` propagates *expensive* constants: those that are expensive to reproduce, but may be useful to know. If the initializing expression for a constant c in a `clet` clause expands into the constant value V, then `(lconstantp` c`)` $\rightarrow$ `t` during expansion of the `clet` body, and `(value-of` c`)` $\rightarrow V$. However, occurrences of c in the body are not converted into V. Only if all occurrences of c disappear during expansion, then the initializing clause for c is also removed.

This behavior is obtained by the use of the `code-value` operator, which takes three arguments: code that generates that value (must always be present); the value itself, or the keyword `:unknown` if it is not known; and an association list of keyword-value pairs, available to the programmer to propagate type-specific information. This operator has no expansion function in the `:basil` phase, and it expands into its first argument (the code) during the later phase (`:c` or `:lisp`).

Chapter 4

Reference

This chapter describes the Basil operators and the transformation mechanism of L. The style and syntax follow the conventions of the Common Lisp reference manual.

4.1 Program Structure and Transformation

A Basil/L program is composed of modules. Each module contains one or more top-level forms. Each form may be a fragment of Basil code, or an L compilation directive. Basil code is transformed into C or Lisp code, which is then compiled and executed. The transformation is divided in *phases*, summarized in figure 4.1.

A phase is composed of passes. The first pass is the *expansion pass*, described in 4.2. This is followed by zero or more *ad-hoc* passes. A programmer can extend Basil by adding operators to the `:basil` phase, using `deflop`, or by adding a new phase before it.

The `:basil` phase transforms Basil code into a nameless low-level form, which can be fed to the `:c` or `:lisp` phases. The operators `deflop-c` and `deflop-lisp` add operators to these phases: they are documented for completeness, but it is not expected that they will be used in normal L programming.

4.2 Operator Rewriting

Forms are expanded top-down. A top-level form in a program is first read as a Lisp object (a tree of lists and other Lisp objects). Then the expander (the function

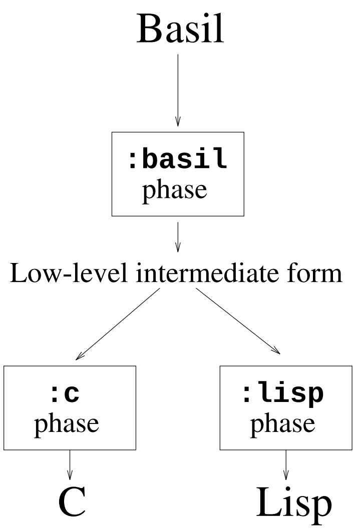


Figure 4.1: The Basil phases

`expand`) traverses it destructively and returns an expanded tree. A form whose operator has an associated expansion function, as defined by `deflop` or one of its variants, is replaced with the object returned by the expansion function. The form returned is itself expanded unless one or more of the following situations occur:

1. the operator of the form does not have an expansion function;
2. the operator of the form is equal to the operator of the form before expansion;
3. the expansion function returns `t` as its second value, indicating that the form and its subforms have been completely expanded.

After a form has been expanded, its arguments in expression positions are expanded as needed, unless the expansion function has returned `t` as its second value.

Operator expansion functions are phase-specific. The `deflop` operator defines the expansion function for the `:basil` phase. `deflop-c` and `deflop-lisp` define the expansion function for the `:c` and `:lisp` phases.

`deflop` *op args* `&key :type :template :output &rest` *body* [*Lisp macro*]

`deflop` defines a new operator with name `op`. *args* specifies the arguments of an expression with such operator, using the syntax of the argument list in a Common

Lisp macro definition. *body* is the body of the expansion function of the operator. If there are no statements, the operator does not have an expansion function.

The `:type` keyword argument[1] can be:

- an L type expression. Any expression with this operator has the given L type.
- a Lisp function. This function takes a form with the given operator as argument, and it returns an L type expression;
- the `:expand` keyword. This indicates that to compute its type, the expression must be expanded first.
- the `:first`, `:second`, etc. keywords. This is a shorthand for a type function that returns the type of the first, second, etc. argument.

The default value for the `:type` argument is `:expand`.

The `:template` keyword argument is a form that specifies the structure and some semantics of an expression with that operator. Templates are used for various kinds of code traversal during and after the expansion phase. If an operator has an expansion function, and expands into a form with a different operator, then its template is useless.

A template is a list of template symbols and other templates. The template symbols have semantic significance as explained:

`expr`. An L expression that returns a value.

`location`. An L expression that returns a location: for instance, the left hand side of an assignment operator, or a pass-by-reference parameter position.

`name`. A symbol that is not evaluated: for instance, a variable in the declaration position of a `let` form, or other variable-binding constructs.

`*`. Some other kind of information associated with the form.

In addition, the following form has a special meaning:

`(list` *template*`)`. Zero or more occurrences of *template*. This may only appear as the last subform of a form.

[1]These are not keyword arguments in the strict Common Lisp sense, because positionally the precede the body of the expansion function. However, they are parsed in the obvious way.

The default value for the operator template is `(list expr)`.

The two variations of `deflop`, `deflop-c` and `deflop-lisp`, define an expansion function for the `:c` or `:lisp` phase. Type and template are not phase-specific.

`defsop` *name args form* [*Lisp macro*]
`defsop-c` *name args form* [*Lisp macro*]
`defsop-lisp` *name args form* [*Lisp macro*]

Define a "simple" operator. The lambda-list *args* must be a simple list of symbols. The replacement form for *name* is obtained by this process: occurrences in *form* of symbols from *lambda-list* are substituted (via the Lisp function `subst`) with the corresponding actual parameters, and the resulting form is returned. The type and template for *name* are the default ones from `deflop`. Example:

```
(defsop 1+ (x)
  (+ x 1))
```

`loplet` $(\{($ *name args* $\{$*expr*$\}^+)\}^+)$ *body* [*L operator*]
`soplet` $(\{($ *name args* $\{$*expr*$\}^+)\}^+)$ *body* [*L operator*]

These forms are used to define operators locally to a group of statements, just like `macrolet` in Lisp. `loplet` resembles `deflop`, `soplet` resembles `defsop`. Two examples follow:

```
(soplet ((int-square (x)
           (let (((int tmp) x))
             (* tmp tmp))))
  (int-square 42))

(loplet ((square (x)
           `(let (((,(l-type x) tmp) ,x))
             (* tmp tmp))))
  (square 42.0))
```

`lconstantp` *object* [*Lisp function*]

`lconstantp` is true if and only if its argument evaluates to a known value. `lconstantp` is false when its argument is a symbol or an expression that would expand to a known value; to check if such objects have a known value, they must be expanded first with the `expand` function. See also the tt value-of function for examples.

`value-of` *object* [*Lisp function*]

`value-of` returns the value, as a Lisp object, of an L expression that evaluates to a constant. Examples:

```
(lconstantp 42) --> t
(value-of 42) --> 42
(lconstantp (quote 42)) --> t
(value-of (quote 42)) --> 42
(lconstantp (code-value '(val x) 42 nil)) --> t
(value-of (code-value '(val x) 42 nil)) --> 42
(lconstantp 'x) --> nil
```

`l-type` *l-expr* [*Lisp function*]

`l-type` returns the type of *l-expr* if it can be computed in the current environment; otherwise it causes an error.

`expand` *code* [*Lisp function*]

`expand` returns the expansion of *code* using the current operator definitions. This function is useful in writing expansion functions, particularly those that perform constant folding. For instance, a constant folding `+` can be defined as follows:

```
(deflmacro + (x y)
  (values
   (let ((x (expand x))
         (y (expand y)))
      (if (and (lconstantp x) (lconstantp y))
          `(quote ,(+ (value-of x) (value-of y)))
          `(+ ,x ,y)))
   t))
```

The macro returns the second value `t` to indicate that no further expansion is necessary.

`code-value` *code value other* [*L operator*]

A code-value form is used to propagate values and other attributes of values in a on-demand fashion. *code* contains L code. In the target-specific phase, a code-value expression is replaced by *code*. In the `:basil` phase, the constantness of a code-value form is determined by *value*. When the value of *value* is `:unknown`, the value of the form is unknown; in all other cases, the *value* field represents the compile-time value of the object.

Code-value forms, together with the functions `lconstantp` and `value-of`, allow a form of constant propagation in which constant values are propagated only to the points in which they are effectively used. This is useful when reconstructing the constant value at run time is expensive.

other is an association list of keyword-value pairs, and it is available to the programmer for type-specific information. See also `deflstruct` for an example of its use.

4.3 Variables, Constants, and Blocks

Symbols in an expression position are variables or symbol macros. Their meaning is either global or local, and it can be redefined locally.

`defsconstant` *name form* [*Lisp macro*]
`defsconstant-c` *name form* [*Lisp macro*]
`defsconstant-lisp` *name form* [*Lisp macro*]

After one of these definitions, occurrences of *name* in an expression position are replaced with *form* during the corresponding expansion pass.

`slet` `(`$\{$`(`*var form*`)`$\}^+$`)` $\{$*expression*$\}^+$ [*L macro*]

Occurrences of each *var* are expanded with the corresponding *form* within the body, except where *var* is rebound by another variable-binding form. The expanded code is a `let` form with an empty argument list.

`clet` `(`$\{$`((`*type var*`)` *form*`)`$\}^+$`)` $\{$*expression*$\}^+$ [*L operator*]

Each *form* is expanded, and if it expands into a constant, the corresponding *var* is bound to a code-value form with *value* set to that constant, and *code* set to *var*.

Occurrences of *var* in the body expand into that code-value form.

If after expansions of the body all occurrences of *var* disappear, and `form` expanded into a constant, then the initialization clause is removed.

`let` ({`((`*type var*`)` [*form*]`)`}*) {*expression*}+ [*L operator*]

`let` is the main local-variable binding construct. The structure of the variable declarations is similar to that of a Lisp `let`, except that the variable is replaced by the pair `(`*type var*`)`, where *type* is an L type expression.

When *type* is `t`, the pair `(t` *var*`)` may be abbreviated to *var*. Also, if *var* has type `t` and is not initialized, the parentheses enclosing *var* may be dropped. Examples:

```
;;; x has type int and is initialized to 42.
(let (((int x) 42)) ...)

;;; x has type int and is not initialized.
(let (((int x))) ...)

;;; x has type t and is not initialized.
(let (x) ...)

;;; int has type t and is initialized to x.  Eh?  What?
(let ((int x)) ...)
```

The last example is not recommended.

`reflet` ({`((`*type var*`)` *form*`)`}+) {*expression*}+ [*L operator*]

Each *form* must be interpretable as a location. The compiler makes *var* an alias for the location denoted by *form* in the body. *var* can be used both to store and retrieve the value in that location.

`genvar` `&optional` *string* [*Lisp function*]
`genvar-global` `&optional` *string* [*Lisp function*]

`genvar` returns a symbol with a name that is unique within the current function. The symbol can be used to represent local variables. `genvar-global` returns a symbol with a globally unique name. The optional *string* argument is used in forming the name.

The naming convention in L is that names beginning with an underscore are reserved for variables generated during macro expansion. Of them, the ones ending with an underscore are reserved for names that need to be globally unique.

4.4 Types and Structures

The primitive types of L are those of C (char, int, float, double, void), with the addition of type `t`, the universal union pointer. Derived types can be obtained with the `pointer`, `array`, and `fun` constructors, and the `defltype` and `deflstruct` forms.

Types are treated just like expressions during macro expansion.

`t` [*L type*]

Variables of this type may point to any structure defined by `deflstruct`, and allocated by its structure constructor. This is the L equivalent of Lisp's dynamic typing. The type `t` makes it easy to define "generic" functions and data structures, such as a *list* type. Objects of type `t` are garbage-collected in the C execution environment.

`pointer` *type* [*L type operator*]
`array` *length type* [*L type operator*]
`fun (`{*argtype*}*`)` *return-type* [*L type operator*]

These constructors produce C's derived types.

`defltype` *name type* [*L operator*]

This construct makes *name* equivalent to *type*, similarly to C's *typedef*.

`deflstruct` *name* {*field*}$^+$ [*L operator*]

`deflstruct` defines the L structure *name* and creates several functions and operators. Each *field* is a `(`*type field-name*`)` pair, or a single *field-name*, which stands for `(t` *name*`)`. Example:

```
(deflstruct point
  (int x)
  (int y))
```

The next few entries describe the operators defined by `deflstruct`.

`alloc-`*name* [*L operator*]

This function allocates an uninitialized instance of structure *name*, and returns a pointer to that structure. The return type of this function is `t`. Calling this function is the standard way of allocating instances of *name* on the heap.

name`-p` *pointer* [*L operator*]

name`-p` returns true if *pointer* points to a structure of type *name*, otherwise it returns nil.

name`-`*field* *pointer* [*L operator*]

name`-`*field* is the field selector operator for the field *field* of *name* when accessing the structure through a pointer. *pointer* should be of type `t` and should point to an instance of *name*.

If *pointer* expands to a code-value form, the field selector checks if the structure represents a partial value. This is true if the *other* field of the form contains a `(:partial-value .` *instance*`)` pair. *instance* must be an instance of structure *name* with each field initialized either to `:unknown` or its compile-time value. If the field *field* of *instance* is not `:unknown`, the field selector form expands to that field, otherwise it remains unchanged.

name`-`*field*`*` *expr* [*L operator*]

name`-`*field*`*` is the field selector for an instance of *name*. *expr* should be of type *name*.

`make-`*name* $\{field\}^+$ [*L operator*]

This operator is a By Order of Arguments (BOA) constructor for structure *name*. When any of its arguments expands to a constant, `make-`*name* expands to a code-value form representing a partial value (see *name-field* above). (This functionality is currently unimplemented).

`make-`*name* {*field*}$^+$ [*L function for C*]

This function is a By Order of Arguments (BOA) constructor for structure *name*, and is available only to the C output.

`expand-`*name struct* [*Lisp function*]

A Lisp function that converts a Lisp structure instance into C code. *struct* should be an instance of structure *name*. This function translates structure instances into C code during the `:c` expansion pass.

This concludes the descriptions of L operators created by `deflstruct`.

`type-p` *object type* [*L operator*]

`type-p` is true if *object* is a pointer to a structure of type *type*, otherwise is false.

`new` *type* [*L function for C*]

`new` allocates space for a structure of type *type*, just like `alloc-`*type*, but is only defined in C.

`new*` *type size* [*L function for C*]

`new` allocates space for a variable-sized structure of type *type*. It is only defined in C.

4.5 Functions and Statements

L functions are semantically similar to Lisp functions, except the value must be returned by a `return` statement.

`deflfun` (*type name*) *arglist* {*statement*}$^+$ [*L operator*]
`deflfun-pure` (*type name*) *arglist* {*statement*}$^+$ [*L operator*]

`deflfun` defines an L function with name *name* and return type *type*. *arglist* is a list triplets `(`*mode type name*`)`, where *mode* is either `val` or `ref`, denoting argument passing by value or by reference, *type* is the argument type, and *name* the argument name.

`deflfun-pure` is like `deflfun`, but makes a Lisp version of the function available for execution during compilation. Furthermore, when all arguments to a call to a pure functions are compile-time constant, the call is evaluated during the expansion pass, and it is replaced by the result.

Both `deflfun` and `deflfun-pure` define an operator with the same name of the function. This definition can be overridden by a `deflop` after the function definition. Should it occur before, its effect is obliterated by the `deflfun`.

4.5.1 Control Statements

`if` *clause then else* [*L operator*]
`when` *clause* $\{$*statement*$\}^+$ [*L operator*]
`unless` *clause* $\{$*statement*$\}^+$ [*L operator*]
`for (`*init test repeat*`)` $\{$*statement*$\}^+$ [*L operator*]
`while` *clause* $\{$*statement*$\}^+$ [*L operator*]
`loop` $\{$*statement*$\}^+$ [*L operator*]
`break-loop` [*L operator*]
`return` [*expression*] [*L operator*]
`case` *keyform* $\{$`(`$\{$`(`$\{$*key*$\}^*$`)` | *key*$\}$ $\{$*form*$\}^*$`)`$\}^*$ [*L operator*]

Most of these statements have obvious meaning in C or Lisp. They all have `void` type, except `if` and `case`, which may return a value. `return` is the C return, not the Lisp return. `loop` is an infinite loop. `break-loop` transfers control to the first statement that follows the enclosing `loop`.

`dotimes (`*var count* `&key` *inline inline-limit*`)` $\{$*statement*$\}^+$ [*L operator*]

`dotimes` works like in Lisp, but it also allows the two keyword parameters *inline* and *inline-limit*. If *count* expands into a constant, and *inline* is `t`, or *inline-limit* is specified and *count* is less or equal to *inline-limit*, then the loop is expanded in line. The default value for *inline* is `nil`, and for *inline-limit* is 1. tt dotimes has `void` type.

`dotimes*` (*var count*) $\{$*statement*$\}^+$ [*L operator*]

`dotimes*` works similarly to `dotimes`, except it has no keyword parameters, and *var* must already be declared.

4.5.2 Assignments

`setq` *place value* [*L operator*]
`setf` *place value* [*L operator*]

`setq` and `setf` are equivalent. They store a copy of *value* in *place*.

`incf` *place* `&optional` *delta* [*L operator*]
`decf` *place* `&optional` *delta* [*L operator*]

These are the same as in Lisp.

4.5.3 Arithmetic operators

The standard arithmetic operators are available.

4.6 Lists

The following list-related operators are available:

`cons` *car cdr* [*L operator*]
`car` *list* [*L operator*]
`first` *list* [*L operator*]
`cdr` *list* [*L operator*]
`rest` *list* [*L operator*]
`last` *list* [*L operator*]
`push` *element list* [*L operator*]
`pop` *list* [*L operator*]
`nconc` $\{$*list*$\}^*$ [*L operator*]
`append` $\{$*list*$\}^*$ [*L operator*]
`mapcar` *operator* $\{$*list*$\}^+$ [*L operator*]
`mapcan` *operator* $\{$*list*$\}^+$ [*L operator*]

`mapc` *operator* $\{list\}^+$ [*L operator*]
`dolist` (*var list* `&key` *inline-limit*) *L operator* $\{expr\}^+$ [*L operator*]
`length` *list* [*L operator*]
`nreverse` *list* [*L operator*]
`sort-list` *function list* [*L operator*]

The syntax and semantics of these operators is identical to their synonymous ones in Common Lisp; with the exception of the mapping operators (`mapcar`, `mapcan`, `mapc`) which take an operator (or *slambda* form) as their second argument; and `dolist`, which does not return a value, and takes a keyword argument specifying a maximum list length for inlining the loop. The *function* argument of `sort-list` must be an operator that is implemented as a function.

4.7 Compilation

An L program is typically developed in separate files, also called modules. L has no mechanism for hiding information across modules: a function declared in a module is visible from all other modules.

Files containing L sources have the suffix `.l`. Compiling *module*`.l` produces several files: `_`*module*`.c`, containing C function and variable definitions; `_`*module*`.h`, with C declarations to interface between modules; `_`*module*`.d`, with auxiliary definitions for C compilation; and `_`*module*`.cl`, which contains compiler state information necessary to compile other modules.

module`.c` includes at its beginning the file `all.h`. For a system composed of the n modules m_1`.l`, ..., m_n`.l`, the programmer must make the file `all.h` include m_1`.h`, ..., m_n`.h`, and the file `all.d` must include m_1`.d`, ..., m_n`.d`.

The forms in each L source file are read and processed one by one by the function `l-compile-form`.

The special variable `*auxiliary-c-forms*` is set to the empty list before processing the first form in a file. Macro expansion functions can add definitions to this list. After processing the last form in the file, the forms contained in it are output as C code. To insure that these forms are expanded in the correct environment, the programmer should manually process them by applying `expand` and `translate-into-c` to each.

`l-compile` $\{module\}^+$ [*Lisp macro*]

`l-compile` compiles the code contained in the file(s) *module*`.l`. Its arguments are not evaluated.

`l-compile-module` $\{$*module*$\}^+$ [*Lisp function*]

This function compiles one or more modules. Its arguments must be symbols. The file names of the modules are obtained by appending `".l"` to the lower-case version of the name of each symbol argument.

`l-compile-form` *form* [*Lisp function*]

Compile the L form *form*.

`need` $\{$*module*$\}^+$ [*L operator*]

This form speficies that to compile correctly the current module the compiler needs information contained in *module*.

`eval` *form* [*L operator*]

This is an escape-into-Lisp mechanism. The expansion of this form is obtained by evaluating *form* as a Lisp form.

`literal` *string* [*L operator*]

This is an escape-into-C mechanism. *string* is output literally into the C output file.

`target-select` *c-code lisp-code* [*L operator*]

The expansion function of this operator returns *c-code* when the final target of the compilation is C, *lisp-code* when it is Lisp.

`extension` *form* [*Lisp macro*]

The argument of an `extension` form must be a function call form. A form that evaluates to the same value as the function call form is output to current compiler extension file. The replacement of the `extension` form is its argument. Example:

```
(let ((x 'x-value)
```

```
        (y 'y-value))
    (extension (f x y)))
```

is replaced by `(f x y)`, and the form `(f 'x-value 'y-value)` is output to the extension file. The arguments of `f` must have a printable representation that can be read back in.

`*target*` [*Lisp variable*]

This variable is bound to either `:c` or `:lisp`, depending on the current compilation target.

`*auxiliary-c-forms*` [*Lisp variable*]

`*auxiliary-c-forms*` is set to `nil` at the beginning of the compilation of a module. After all the forms in the module have been processed, the contents of this variable undergo phase `:c` transformations and is output. An L form should be placed in this list only after application of `translate-into-c`. Example:

```
(push (translate-into-c form) *auxiliary-c-forms*)
```

`translate-into-c` *form* [*Lisp function*]

`translate-into-c` returns an L form that can be directly passed to the `:c` phase for further processing. It should be used with `*auxiliary-c-forms*`.

Bibliography

[HC89] Paul N. Hilfinger and Phillip Colella. *FIDIL: A Language for Scientific Programming*, chapter 5, pages 97–138. Frontiers in Applied Mathematics. SIAM, 1989.

[Sem93] Luigi Semenzato. The INFIDEL Virtual Machine. Technical Report UCB/CSD 93-761, Computer Science Division (EECS), Univ. California, Berkeley, July 1993.

Index