The C-- Language Specification
Version 2.0
(CVS Revision 1.128)

Norman Ramsey Simon Peyton Jones Christian Lindig

June 27, 2009

Contents

List of Figures

List of Tables

1 Introduction

C-- is a compiler-target language. The idea is that a compiler for a high-level language translates programs into into C--, leaving the C-- compiler to generate native code. C--'s major goals are these:

- C-- encapsulates compilation techniques that are well understood, but difficult to implement. Such techniques include instruction selection, register allocation, instruction scheduling, and optimization of imperative code with loops.
- C-- is a *language*, rather than a library (such as `gcc`'s RTL back end). As such, it has a concrete syntax that can be read by people, and a semantics that is independent of any particular implementation.
- C-- is a portable *assembly* language. It is, by design, as low-level as possible while still concealing details of the particular machine architecture.
- C-- is independent of both source programming language and target architecture. Its design accommodates a variety of source languages and leaves room for back-end optimization, all without upcalls from the back end to the front end.
- C-- is efficient—or at least admits efficient implementation. So C-- provides almost as much flexibility and performance (assuming a good C-- compiler) as a custom code generator.

A *client* of C-- is divided into two parts. The *front-end compiler* generates programs written in the *C-- language*, which the C-- compiler translates into efficient machine code. This code interoperates with the *front-end run-time system*, which implements such services as garbage collection, exception dispatch, thread scheduling, and so on. The front-end run-time system relies on mechanisms provided by the C-- run-time system; they interact through the *C-- run-time interface*. This document describes the C-- language and run-time interface. It is intended for authors who wish to write clients of the C-- system.

The C-- team has prepared some demonstration front ends that may help you learn to use C--.

- Paul Govereau's `tigerc` compiler includes a compiler for Tiger (the language of Andrew Appel's compiler text) as well as a small run-time system with garbage collector. The `tigerc` compiler is written in Objective Caml.
- Norman Ramsey's `lcc` back end works with `lcc` to translate C into C--. It is written in C.
- Reuben Olinsky's interpreter, which is packaged with the Quick C-- sources, includes sample run-time clients for garbage collection and exception dispatch.

All this code is available through the C-- web site at `http://www.cminusminus.org`.

1.1 What C-- is not

C-- has some important non-goals:

- C-- is *not* an execution platform, such as the Java Virtual Machine or the .NET Common Language Runtime. These virtual machines are extremely high-level compared to C--. Each provides a rich type system, garbage collector, class loader, just-in-time compiler, byte-code verifier, exception-dispatch mechanism, debugger, massive libraries, and much more besides. These may be good things, but they clearly involve much more than encapsulating code-generation technology, and depending on your language model, they may impose a significant penalty in both space and time.

- C-- is not "write-once, run-anywhere". It conceals most architecture-specific details, such as the number of registers, but it exposes some. In particular, C-- exposes the word size, byte order, and alignment properties of the target architecture, for two reasons. First, to hide these details would require introducing a great deal of complexity, inefficiency, or both—especially when the front-end compiler needs to control the representation of its high-level data types. Second, these details are easy to handle in a front-end compiler. Indeed, a compiler may benefit, because it can do address arithmetic using integers instead of symbolic constants such as `FloatSize` and `IntSize`.

1.2 Run-time services

C--'s main goal is to encapsulate code generation. One reason that this encapsulation has proved troublesome in the past is that code generation interacts with the provision of *run-time services*, such as:

- Garbage collection
- Exception handling
- Concurrency
- Debugging

These services are typically implemented through intimate collaboration between the code generator and a *run-time system*, but such close collaboration is impeded by the abstractions needed to make a code generator reusable. Nevertheless, it would be wrong for C-- to offer such services; no one semantics, object layout, and cost model could possibly satisfy all clients.

Instead, C-- also comes with a small run-time system, which provides some low-level, primitive mechanisms, on top of which a C-- client can implement high-level services. The C-- *run-time interface* offers abstractions that, for example, enable a garbage collector to walk the stack and an exception-dispatch mechanism to unwind or cut the stack. The run-time interface is described in Section 8.

1.3 A glimpse of C--

Much of C-- is unremarkable. C-- has parameterized procedures with declared local variables. A procedure body consists of a sequence of statements, which include (multiple) assignments, conditionals, `goto`s, calls, and `jump`s (tail calls).

To give a feel for C--, Figure 1 presents three C-- procedures, each of which computes the sum and product of the integers $1..n$. Figure 1 illustrates two features that are common in assemblers, but less common in programming languages. First, a procedure may return multiple results. For example, each procedure in Figure 1 returns two results, and `sp1` contains a call to a multi-result procedure (namely `sp1` itself). Second, a C-- procedure may explicitly tail-call another procedure. For example, `sp2` tail-calls `sp2_help` (using "`jump`"), and the latter tail-calls itself. A tail call has the same semantics as a regular procedure call followed by a return, but it is guaranteed to deallocate the caller's resources (notably its activation record) before the call.

All memory access is explicit. For example, the statement

```
bits32[x] = bits32[y] + 1;
```

loads a 32-bit word from the memory location whose address is in the variable `y`, increments it, and stores it in the memory location whose address is in the variable `x`.

```
/* Ordinary recursion */
export sp1;
sp1( bits32 n ) {
  bits32 s, p;
  if n == 1 {
      return( 1, 1 );
  } else {
      s, p = sp1( n-1 );
      return( s+n, p*n );
  }
}


/* Tail recursion */
export sp2;
sp2( bits32 n ) {
  jump sp2_help( n, 1, 1 );
}

sp2_help( bits32 n, bits32 s, bits32 p ) {
  if n==1 {
      return( s, p );
  } else {
      jump sp2_help( n-1, s+n, p*n );
  }
}
```

```
/* Loops */
export sp3;
sp3( bits32 n ) {
    bits32 s, p;
    s = 1; p = 1;

  loop:
    if n==1 {
      return( s, p );
    } else {
      s = s+n;
      p = p*n;
      n = n-1;
      goto loop;
}   }
```

Figure 1: Three procedures that compute the sum $\sum_{i=1}^{n} i$ and product $\prod_{i=1}^{n} i$, written in C--.

2 Fundamentals of C--

C-- combines features of assembly languages and compiler intermediate codes. To provide context for the detailed descriptions in the rest of the manual, this section introduces the major concepts of C--.

2.1 Classification of errors

Although C-- compilers detect some *compile-time errors*, and the C-- run-time system detects some *checked run-time errors*, C-- is by design an unsafe language with many *unchecked run-time errors*—it's an assembler. A compile-time error or checked run-time error is one the the C-- implementation guarantees to detect. An unchecked run-time error is one that it is up to the front-end compiler and run-time system to avoid; after an unchecked run-time error, the behavior of a C-- program is arbitrary. Examples of unchecked run-time errors include unaligned memory accesses that are not so annotated in the source, passing the wrong number or types of arguments, and returning the wrong number or types of results.

2.2 C-- procedures

All code is part of some C-- *procedure*, which is defined at the top level of a compilation unit. Procedures are described in detail in Section 5. Their two unusual features are:

- They support fully general, optimized tail calls (§6.8).
- A procedure may return more than one result (§6.8).

A procedure is a first-class value—for example it may be passed as a parameter, or stored in a data structure—and a call may be indirect.

There are no limits on the number of parameters a C-- procedure expects or the number of results it can return, but the number and types of parameters and results for any particular procedure are fixed at compile time. C-- has no support for "varargs".[1] Calls and returns are *unchecked* and *unsafe*. C-- provides multiple calling conventions; the default calling convention is chosen by the back end. The default calling convention supports proper tail calls, but they must be written explicitly by the front end, using the keyword `jump`. C-- also provides the C calling convention, which does not support tail calls.

A C-- procedure may include labels and computed gotos. It is an *unchecked* run-time error for a goto to cross a procedure boundary. For control flow across procedure boundaries, the analogs of goto and label are `cut to` and `continuation`. As in assembly languages, a label is visible throughout the compilation unit in which it appears, so for example, a front end can build a jump table. A continuation is visible only in the procedure in which it appears.

The primary purpose of a procedure is to be executed, but a C-- procedure is also a form of initialized data. This fact is important if, for example, a front end wants to define initialized data that mimics the layout of a heap object containing executable code.

2.3 Registers and memory

Like machine instructions, C-- programs manipulate *registers* and *memory*. In C--, variables are like registers; they have no addresses, and assigning to one cannot change the value of another. Variables may be *local* (private) to a procedure, but C-- also supports *global register variables* that are shared by all procedures. An implementation of C-- does its best to map variables onto actual hardware registers.

[1]There is a proposal hanging fire that would extend C-- with support for a variable number of arguments, but this proposal would *not* be compatible with C varargs or with C's `stdarg.h`. Various research projects notwithstanding, varargs in the C style are incompatible with optimized tail calls.

Labels, procedure names, imported names, and other names that refer to memory locations are *immutable*. Just as in an assembler, one does not assign to these names; instead, one uses explicit fetch and store operations that specify both the memory address and the size of the value to be transferred. Addresses and address arithmetic are in units of the target "`memsize`," which should be specified to be the number of bits in the normal addressing unit of the machine. If not specified explicitly, a program's target `memsize` defaults to 8 bits.

Memory access uses the byte order of the target machine, which must be specified explicitly using a target `byteorder` directive.[2] Fetches and stores include (implicit) assertions about alignment of addresses; it is an *unchecked* run-time error to violate these assertions.

2.3.1 Rights to memory

Memory is a resource that must be shared by C-- and its client. The "rules of the road" are as follows:

- The C-- run-time system includes initialized data that belongs to C--. The client may not read or write this data.
- When the C-- compiler translates a compilation unit, it generates initialized data that is used either to implement the dynamic semantics of procedures (e.g., generated machine instructions, jump tables) or to support the C-- run-time system (e.g., stack maps). The client must not write this data, and it is not guaranteed to be granted read access to the data.
- The heap belongs to the client. The front-end run-time system manages all allocation; the C-- run-time system never allocates.
- Stacks are shared between the client and C--. Initially, the client owns the system stack, any OS-thread stacks, and any stacks allocated on the heap. But when the client calls into a C-- procedure, C-- takes over the rights to the stack. The C-- stack is a private data structure, and the front end may not read or write locations on the stack, *except* that the front end may read or write any locations declared within `stackdata`.

2.3.2 Aliasing

A C-- compiler is intended to perform many scalar and loop optimizations, including instruction scheduling, redundancy elimination, peephole optimization, and loop-invariant code motion. The C-- compiler can do a better job if it can tell when two memory references may *interfere* or *alias*. Because may-alias information can be very difficult to recover from low-level codes, C-- provides annotation mechanisms by which the front end can communicate may-alias information to the optimizer:

- Each memory reference (load or store) may be annotated with a list of *alias names* (§7.3.2).
- Each procedure call may be annotated with a list of alias names read and a list of alias names written (§6.8.1).
- The front end provides an interpretation of the alias names that enables the back end to determine whether two references may alias.

The annotations on memory references and on calls are specified in this manual. But the mechanism by which the front end provides an interpretation is not specified; instead it is left up to each individual implementation of C--. We expect that once we have sufficient experience with C--, a future version of this specification will describe a mechanism that every implementation of C-- will be required to provide.

[2]This requirement ensures that every well-typed C-- program has a semantics that is independent of target machine.

If C-- variables are like machine registers, why aren't there different types of variables? For example, why doesn't C-- distinguish integer variables from floating-point variables, since most machines distinguish integer registers from floating-point registers? Earlier versions of C-- did draw such distinctions, but when we thought about how this information was *used*, we gradually moved to other mechanisms.

A C-- compiler needs to know *which registers to use* to hold the values of variables and to pass parameters and results. We can't predict what kinds of registers a machine will have. Many machines have just integer and floating-point registers, but not all. The 68k Dragonball, which is used in older Palm Pilots, has separate registers for integers and pointers. Some DSP chips have two sets of general-purpose registers, sometimes called the X and Y registers. The StrongARM has predicate registers. Using a different C-- type for each kind of machine register would create more work for front ends, and it would make it difficult to generate code for new kinds of architectures without having to change C--. We therefore decided that part of the role of C-- should be to hide not only the number of registers on the machine, but also what kinds of registers exist.

A C-- back end has all the information it needs to place local variables in machine registers, because it can see all the locations where such variables are defined and used. But a back end does not always have enough information to know where to put formal and actual parameters, because those locations are determined by the calling convention, and many calling conventions use different locations for values depending on the high-level types of those values. For example, most calling conventions pass integer values in integer registers and floating-point values in floating-point registers. To help determine where parameters should be passed, C-- uses "kinds" on formal parameters, actual parameters, and results. The kinds are string literals, so we expect they will accommodate new source-level types and calling conventions far into the future.

Figure 2: Sidebar: The C-- type system

2.4 The type system

The C-- type system serves only two purposes. It helps the back end choose the proper machine instruction for each operation, depending on the sizes of the operands, and it helps the back end make effective use of condition codes. *The type system does not protect the programmer.* As in a real assembler, distinctions between signed, unsigned, and floating-point values are in the operators, not in the types of the operands.

There are just two types in C--:

- A k-bit *value* has type `bits`k; for example, a four-byte word is specified as `bits32`. Values may be stored in variables, passed as parameters, returned as results, and fetched from and stored in memory.
- A Boolean value has type `bool`. Booleans govern control flow in conditionals. Boolean values are not first class: they cannot be stored in variables or passed to procedures.

A rationale appears in Figure 2.

For each target architecture, each implementation of C-- designates two special `bits`k types:

The native pointer type is a type `bits`p of machine addresses. For example, the name of a procedure, like `sp1` in Figure 1, denotes an immutable value of the native pointer type. The size p of the native pointer type is called the *native pointer size.* There is no separate pointer type.

The native word type is a type `bits`w that is the type of a literal that is not explicitly typed (§3.3.3). The size w of the native word type is called the *native word size.*

2.5 Kinds

The C-- type system deliberately does not distinguish floating-point numbers from integers or signed integers from unsigned ones. Instead, a C-- implementation is expected to decide (say) what sort of register to use for a C-- variable depending on what operations are performed on that variable. In this way, C-- is not vulnerable to such architecture-specific variations as (say) whether pointers and data values must be held in different register banks.

There are two places, however, where a C-- implementation does not have enough information to make an effective decision:

- The arguments and results of a procedure. For example, in a procedure *definition* it may be clear that the first parameter is treated as a floating-point number, but the procedure *calls* may not be able to "see" the definition, or even know statically what procedure is being called.
- To establish a suitable storage location for each global variable (§4.4.1) would require a whole-program analysis.

C-- therefore allows the programmer to annotate these constructs with optional *kinds.* A kind is simply a literal string; a C-- implementation must advertise what kinds it understands. Kinds supply additional information to the implementation, to enable it to make better calling sequences and allocation decisions.

Kinds for procedure definitions and calls are described in Sections 5.1 and 6.8 respectively. *It is an unchecked error for a C-- program to supply different kinds at the definition and call site of a procedure.* It follows that a C-- implementation may safely use kinds to drive the calling convention.

Kinds for global variables are described in Section 4.4.1.

2.6 Compilation units

Like an assembly-language file, a C-- compilation unit specifies the creation of a sequence of named *sections.* Each section may contain a mix of code, initialized data, and uninitialized data. Labels point to locations within sections and are visible throughout a compilation unit. A C-- compiler may produce output in the form of assembly language, object code, or perhaps binary code and data directly in memory.

2.7 Naming and visibility

C-- uses a single name space for values. A name may denote a register variable, which is mutable, or one of several kinds of immutable value. Not all immutable values are link-time or compile-time constants; for example, a continuation value is immutable but is not available until run time.

C-- has a separate name space for types and another separate name space for the names of primitive operators.

C-- uses three nested scoping levels: the program, the compilation unit, and the procedure. C-- does *not* have "definition before use;" a name is visible in the *entire* compilation unit or procedure in which it appears. The "normal" scope is the compilation unit, but local register variables, continuations, and stack labels are private to procedures, and their names may hide names with larger scope. Scoping does not always coincide with the syntactical structure of a program: names declared in sections are part of the unit scope and labels (for `goto` or memory access) *always* have unit scope.

As in Haskell and Modula-3, a C-- name is visible throughout the scope in which it is declared. The order of declarations is irrelevant, and it is a checked compile-time error to declare the same name twice in the *same* scoping level. A name in an inner scope hides the same name in an outer scope.

A name has program scope if and only if it is explicitly exported by the compilation unit that defines it. If the name is used in another unit, the other unit must import it explicitly.

2.8 The run-time model

To the front end, C-- procedures appear to run on a stack. The front-end run-time system can use the run-time interface to inspect and modify one stack and one activation record at a time. The front-end runtime can cause control to resume at a different activation or different continuation within an activation. It can also inspect and modify the values of local and global register variables, as well as memory.

To enable a C-- compiler to optimize a program without introducing errors, the front end must annotate the source code with information about how the run-time system may change control flow. The front end may also annotate the source code with information about how the run-time system may change the values of variables. For example, if a variable is not changed by the run-time system, it may be marked `invariant`. It is an *unchecked* run-time error for the front-end run-time system to disregard an annotation.

To record front-end information, like which C-- variables point to heap objects, the front end can put *spans* around statements and procedures. These spans can be used at run time to map the program counter to arbitrary values deposited by the front end.

We anticipate that users of C-- will wish to create multiple threads of computation that run on multiple stacks. But the design of suitable abstractions for managing stack overflow and stack underflow remains a research problem. For this reason Version 2.0 of this specification is regretfully single-stacked. Support for multiple threads will appear in a future version.

The run-time interface is described in Section 8.

2.9 Portability

C-- is not a write-once-run-anywhere language. As a matter of design, it exposes some aspects of the machine architecture. These are aspects that are relatively easy for a front-end compiler to take account of and relatively hard for a C-- compiler to abstract.

A C-- compiler should advertise, for each particular target architecture, the following facts:

Supported data types: which `bits`k types are supported (§2.4).

Native types: the `bits`k types that implement the native pointer type and native word type (§2.4, §4.7).

Addressing unit: the number of bits m in the smallest addressable unit of memory (§4.7).

Byte order: in a load or store that is wider than the addressing unit, does the lower-addressed location contain the most-significant or least-significant bits of the data (§4.7).

Primitive operators: which primitive operators are supported, and at what sizes (§7.4). Also what primitive operators are available in forms that use alternate-return continuations and what those continuations mean.

Back-end capabilities: which expressions the back end can generate code for (§7).

Foreign interface: What `foreign` calling conventions are supported (§5.4).

Kinds in calling conventions: What kinds may be used in calls and returns, and what the kinds mean. In particular, it must be clear what kind and C-- type corresponds to each C type.

Kinds in register declarations: What kinds may be used in global register declarations, and how those kinds are used to request different kinds of registers.

Names of hardware registers: What hardware registers are available to be used as global register variables, and by what names those hardware registers are known.

3 Syntax

As befits an assembler, the syntax of C-- is designed for flexibility, not beauty. For example, most "top-level" declarations may actually appear anywhere in a program, even in the middle of a procedure. Commas may be used as separators or as terminators, at the discretion of the front end. Some primitives are expressible both in the standard prefix form and in a C-like infix form.

The syntax of C-- is given in Figures 3 and 4 on pages 14 and 15.

This rest of this section describes the lexical aspects of C--, while subsequent sections deal with the syntactic structure.

3.1 Character set

The source code of a C-- program uses the ASCII character set only. (A future extension may support Unicode.)

3.2 Line renumbering

When reporting errors and warnings, a C-- compiler uses a source-code location that includes a file name and line number. C-- numbers the lines in a compilation unit starting from one. The end of a line is marked by a newline character. To facilitate interoperability with other tools, a line directive can be used to associate source code lines to a location in a different file. A line directive has the following form:

`#` *number* `"`*file*`"`

A line directive must be on a line of its own and must start with the `#` character; *number* is a decimal number in the file name *file*. One or more space or tab characters must be used to delimit the three tokens of a line directive. The syntax of the line directive is the one used by the C preprocessor `cpp`.

The line directive associates the *following* line in the source code with line *line* in *file*. Line numbering continues from there on linearly and thus all subsequent lines are considered to be from file *file*.

3.3 Lexemes

All lexemes are formed from printable, 8-bit, ASCII characters. They include *names*, *integer literals*, *floating-point literals*, *character literals*, and *string literals*, as well as symbols and reserved words.

3.3.1 White space

Whitespace may appear between lexemes. Whitespace is a nonempty sequence of characters consisting only of space-like characters: space, tab, newline, return, and form feed.

3.3.2 Lexical names

A maximal sequence of letters, digits, underscores, dots, dollar signs, and `@` signs is a name, unless the sequence is a reserved word or the sequence begins with a digit. Uppercase and lowercase letters are distinct.

As examples, the following are C-- names:

```
x                   _foo.name_abit_12.long
foo                 Sys.Indicators
_912                .9Aname
aname12             $1
@name
```

unit ⇒ {*toplevel*}

toplevel ⇒ `section` *string* `{` {*section*} `}`
| *decl*
| *procedure*

section ⇒ *decl*
| *procedure*
| *datum*
| `span` *expr* *expr* `{` {*section*} `}`

decl ⇒ `import` *import* {`,` *import*} [`,`] `;`
| `export` *export* {`,` *export*} [`,`] `;`
| `const` [*type*] *name* `=` *expr* `;`
| `typedef` *type* *name* {`,` *name*} [`,`] `;`
| [`invariant`] *registers* `;`
| `pragma` *name* `{` *pragma* `}`
| `target` {`memsize` *int* | `byteorder` (`little` | `big`) | `pointersize` *int* | `wordsize` *int*} `;`

import ⇒ [*string* `as`] *name*

export ⇒ *name* [`as` *string*]

datum ⇒ *name* `:`
| `align` *int* `;`
| *type* [*size*] [*init*] `;`

init ⇒ `{` *expr* {`,` *expr*} [`,`] `}`
| *string*
| *string16*

registers ⇒ [*kind*] *type* *name* [`=` *string*] {`,` *name* [`=` *string*]} [`,`]

size ⇒ `[` [*expr*] `]`

body ⇒ {*decl* | *stackdecl* | *stmt*}

procedure ⇒ [*conv*] *name* `(` [*formals*] `)` `{` *body* `}`

formal ⇒ [*kind*] [`invariant`] *type* *name*

actual ⇒ [*kind*] *expr*

kind ⇒ *string*

formals ⇒ *formal* {`,` *formal*} [`,`]

actuals ⇒ *actual* {`,` *actual*} [`,`]

stackdecl ⇒ `stackdata` `{` {*datum*} `}`

Figure 3: Syntax of C--, part I

```
stmt         ⇒ ;
             | if expr { body } [else { body }]
             | switch expr { {arm} }
             | span expr expr { body }
             | lvalue {, lvalue} [,] = expr {, expr} [,] ;
             | name = %% name ( [actuals] ) {flow} ;
             | [kindednames =] [conv] expr ( [actuals] ) [targets] {flow | alias} ;
             | [conv] jump expr [( [actuals] )] [targets] ;
             | [conv] return [< expr / expr >] [( [actuals] )] ;
             | name :
             | continuation name ( [kindednames] ) :
             | goto expr [targets] ;
             | cut to expr ( [actuals] ) {flow} ;

kindednames  ⇒ [kind] name {, [kind] name} [,]

arm          ⇒ case range {, range} [,] : { body }

range        ⇒ expr [.. expr]

lvalue       ⇒ name
             | type [ expr [assertions] ]

flow         ⇒ also (cuts | unwinds | returns) to name {, name} [,]
             | also aborts [,]
             | never returns [,]

alias        ⇒ (reads | writes) name {, name} [,]

targets      ⇒ targets name {, name} [,]

expr         ⇒ int [:: type]
             | float [:: type]
             | ' char ' [:: type]
             | name
             | type [ expr [assertions] ]
             | ( expr )
             | expr op expr
             | ~ expr
             | % name [( [actuals] )]

type         ⇒ bitsn
             | name

string16     ⇒ unicode ( string )

conv         ⇒ foreign string

assertions   ⇒ aligned int [in name {, name}]
             | in name {, name} [aligned int]

op           ⇒ ` name ` | + | - | * | / | % | & | | | ^ | @<< | >> | == | != | > | < | >= | <=
```

Figure 4: Syntax of C--, part II

These are not C-- names:

```
3illegal
import
section
```

3.3.3 Integer literals

An integer literal is a sequence of digits and denotes a number. An integer literal must fit into a stated number of bits, which is determined by the type of the literal. By default, this type is `bits`k, where k is the native word size (§2.4), but the type may be made explicit by providing a `::` *type* suffix. The *type* must denote a `bits`k type.

Unlike a C-- value, an integer literal is either signed or unsigned. The sign of a literal determines how C-- decides whether it fits into its type:

- An unsigned literal fits into k bits if it evaluates to a two's-complement representation in which all but the least significant k bits are zero.
- A signed literal fits into k bits if it begins with a minus sign and it evaluates to a two's-complement representation in which all but the least significant $k-1$ bits are one. That is, the most significant bit of a negative literal must be one.
- A signed literal fits into k bits if it begins with a digit and it evaluates to a two's-complement representation in which all but the least significant $k-1$ bits are zero. That is, the most significant bit of a positive literal must be zero.

It is a checked compile-time error when an integer literal does not fit into its type.

An integer literal may be notated in several ways; the notation determines the base of the literal and whether it is signed or unsigned.

1. A literal starting with `0x` or `0X` is in hexadecimal notation (base 16) and is *unsigned.*
2. A literal starting with `0` is in octal notation (base 8) and is *unsigned.*
3. A literal starting any of the digits `1` through `9` and ending in the letter `u` or `U` is in decimal notation (base 10) and is *unsigned.* Also, a literal `0u` or `0U` is in decimal notation and is *unsigned.*
4. A literal starting with any of the digits `1` through `9` and ending in a digit is in decimal notation (base 10) and is *signed.*
5. A literal starting with a minus sign, followed by any of the digits `1` through `9`, and ending in a digit, is in decimal notation (base 10) and is *signed.*

The following EBNF grammar defines the syntax *int* for integer literals:

$$\begin{array}{ll}
\textit{int} & \Rightarrow (\textit{signed-int} \mid \textit{unsigned-int})\ \big[\texttt{::}\ \textit{type}\big] \\
\textit{signed-int} & \Rightarrow \big[\texttt{-}\big]\,\textit{dec} \\
\textit{unsigned-int} & \Rightarrow \textit{hex} \mid \textit{oct} \mid \textit{dec}\ (\texttt{u} \mid \texttt{U}) \mid \texttt{0}\ (\texttt{u} \mid \texttt{U}) \\
\textit{decdigit} & \Rightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
\textit{hexdigit} & \Rightarrow \textit{decdigit} \mid \texttt{a} \mid \texttt{b} \mid \texttt{c} \mid \texttt{d} \mid \texttt{e} \mid \texttt{f} \mid \texttt{A} \mid \texttt{B} \mid \texttt{C} \mid \texttt{D} \mid \texttt{E} \mid \texttt{F} \\
\textit{octdigit} & \Rightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7}
\end{array}$$

$$\begin{array}{ll} \textit{dec} & \Rightarrow (\texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9})\ \{\textit{decdigit}\} \\ \textit{hex} & \Rightarrow \texttt{0}\ (\ \texttt{x} \mid \texttt{X})\ \textit{hexdigit}\ \{\textit{hexdigit}\} \\ \textit{oct} & \Rightarrow \texttt{0}\ \{\textit{octdigit}\} \end{array}$$

The following examples are integer literals:

```
5  01234  23::bits8  077::bits16  0x00 255U::bits8 -128::bits8
```

For each bit vectory, there are four ways to write that vector as an integer literal: hex, octal, signed decimal, and unsigned decimal. Here is an example:

```
0x81::bits8   0201::bits8   129U::bits8   -127::bits8
```

Each of the following integer literals will be rejected because it does not fit into its type:

```
255::bits8  -129::bits8
```

3.3.4 Floating-point literals

A floating-point literal denotes a floating-point number. Its type is `bits`k where k is the native word size. Like an integer literal, a floating-point literal can be followed by a *type* to give it an explicit type `bits`k. It is a checked compile-time error when the floating-point literal cannot be represented by k bits. The syntax of a floating-point literal is described by the following EBNF grammar for *float*:

$$\begin{array}{lll} \textit{float} & \Rightarrow & \{\textit{decdigit}\}\ \texttt{.}\ \{\textit{decdigit}\}\ [\textit{exp}]\ [\texttt{::}\ \textit{type}] \\ & \mid & \{\textit{decdigit}\}\ \textit{exp}\ [\texttt{::}\ \textit{type}] \\ \textit{exp} & \Rightarrow & (\texttt{e} \mid \texttt{E})\ [\texttt{+} \mid \texttt{-}]\ \{\textit{decdigit}\} \end{array}$$

A floating-point literal may not begin with a dot. The following are examples of floating-point literals:

```
3.1415  3e-5 1e+2  23.3e-4  2.71828e0::bits64
```

The mapping of floating-point literal to bit vector is determined by the C-- program's floating-point semantics. At present, the only semantics supported is the IEEE 754 semantics, but C-- provides for future extension by means of a (currently undocumented) `target float` directive.

3.3.5 Character literals

A character literal is a value of type `bits8` by default. It is specified as an ASCII character enclosed in single quotes: `'a'` denotes the decimal value 97 (ASCII code for "a"). The characters `'` and `\` must be escaped with `\` if used in a character literal, as must any non-printing characters. Like an integer or floating-point literal, a character literal can be followed by an optional type denoting the number of bits k to hold the value: `'a'::bits16` is a `bits16` value.

$$\begin{array}{lll} \textit{character} & \Rightarrow & \texttt{'}\ \textit{char}\ \texttt{'}\ [\texttt{::}\ \textit{type}] \\ \textit{char} & \Rightarrow & \textit{printable character} \\ & \mid & \backslash\ \textit{escape} \end{array}$$

In order to use the ASCII codes of non-printing characters and the quote character these symbols can be denoted with an escape mechanism. A backslash followed by one or more digits or letters is interpreted according to the following table:

Escape Sequence	Interpretation
`\a`	Alert (Bell)
`\b`	Backspace
`\f`	Form feed
`\n`	Newline
`\r`	Carriage return
`\t`	Horizontal tab
`\\`	Backslash
`\'`	Single quote
`\"`	Double quote
`\?`	Question mark
`\x`{*hexdigit*}	The value of the *hexdigit* sequence, which must contain at least one and at most two *hexdigit*s
`\`{*octdigit*}	The value of the *octdigit* sequence, which must contain at least one and at most three *octdigit*s

The hexadecimal and octal notation enable specification of a character by its numeric code. It is a checked compile-time error to specify a value too large to be represented in the number of bits available for the literal.

Examples of legal character literals:

```
'a'  'a'::bits16  '\0' '\0x0' '\010'  '\r'
```

3.3.6 String literals

A string literal is an abbreviation for a sequence of character literals and can be used only to define the initial value of data. A string literal is terminated by double quotes and must contain only printable characters. Each character is value of type `bits8`. To include non-printable characters the backslash-escape mechanism defined for character literals is used: characters following a backslash are interpreted according to the table in Section 3.3.5. It is a checked compile-time error when a numerical escape code specifies a value larger than 8 bits. Examples:

```
"hello"  "world\0"  "(%d) (%s) \n"
```

string ⇒ `"` {*printable character*} `"`

3.3.7 Reserved words

The following identifiers are reserved words and may not be used as C-- names.

```
aborts align aligned also as big bits byteorder case const continuation cut
cuts else equal export foreign goto if import in invariant invisible jump
little memsize pragma reads register return returns section semi span stackdata
switch target targets to typedef unicode unwinds writes
```

Future revisions of the C-- may require additional reserved words, but no C-- reserved word will ever contain an `@`, `$`, or `.` character, so a front end using these characters is guaranteed never to collide with a reserved word.

3.4 Comments

C-- supports the standard C comment conventions. A one-line comment starts with `//`; the comment includes all characters up to, but not including, the next newline character. A multi-line comment starts with `/*` and ends with `*/`. Multi-line comments do not nest; a multi-line comment ends at the first `*/` after the opening `/*`. It is a checked compile-time error when an opened comment is not closed in a compilation unit. Comments are not recognized inside string and character literals.

```
/* This is a multi-line
   comment */

// */ this is a one-line comment
/*// this comment
   ends at the end of this line, not at the end of the line above */
```

4 Top-level structure of a compilation unit

The top level of a C-- compilation unit consists of sections that hold procedures, data, and declarations. Top-level declarations and *sections* define names that are visible in the entire compilation unit, even before their definition. Procedures can contain local declarations, which shadow global declarations except when noted. A local declaration is visible in the entire body of a procedure. Declarations control:

- The import and export of names into and out of a compilation unit (§4.1).
- Names for constants (§4.2).
- Names for types (§4.3).
- Names for registers (§4.4).
- The layout of initialized and uninitialized data sections (§4.5).
- The characterization of the target architecture (§4.7).

Many of these declarations may also appear inside a procedure body, as discussed in §5.2.

4.1 Import and export

decl ⇒ `import` *import* $\{$ `,` *import*$\}$ $[$`,`$]$ `;`
   | `export` *export* $\{$ `,` *export*$\}$ $[$`,`$]$ `;`

Names that are to be used outside of the C-- compilation unit must be exported with the `export` declaration. Likewise, names that the compilation unit uses and does not declare must be imported with the `import` declaration. It is a checked compile-time error to import a name that is also declared locally. Only labels or the names of procedures can be imported and exported; to share a C-- register variable across compilation units, declare it as a global variable in each unit (§4.4.1). An imported and exported name always denotes a link-time constant of the native pointer type (§2.4). *test this*

An example:

⟨*toplevel example*⟩≡
```
import printf, sqrt; /* C procedures used in this C-- program */
export f3;           /* To be used outside   this C-- program */
```

Like any other C-- name, an imported name is visible throughout the scope in which the `import` declaration appears; it is not necessary to import a name before using it.

Each name that is explicitly exported or imported is guaranteed to appear in the symbol table of the compiled object code with precisely the name given in the source C-- program. A name that is used only internally within a compilation unit might be mangled before appearing in the symbol table, or it might not appear in the symbol table at all.

To avoid conflicts with reserved words, an external name can be renamed during import or export. For example:

⟨*toplevel example*⟩+≡
```
import "jump" as ext_jump;
```

imports the external symbol `jump` and makes it accessible as `ext_jump` inside the compilation unit. In the same way, `export ext_jump as "jump"` exports the name `ext_jump` under the external name `jump`. The string notation makes it possible to import and export names that are not legal C-- identifiers.

It is legitimate to `export` or `import` the same name more than once.

`add mul divu modu and or xor shl shrl com eq ne gtu ltu geu leu`

Table 1: Primitive Operations available for constant declaration.

4.2 Constants

decl ⇒ `const` [*type*] *name* `=` *expr* `;`

The `const` declaration gives a name to a compile-time constant *value*. The defining expression of a constant can refer only to numerical and character literals, other constants, and some primitive operations. The type of the expression is the type of the constant; since expression types are inferred they cannot be declared. The scope of a constant defined in a procedure is the body of the procedure. Constants are values and share the same name space with other values like procedures and labels. It is a checked compile-time error when a constant definition refers directly or indirectly to itself.

⟨*toplevel example*⟩+≡

```
const pi        = 3.1415;       /* type bits k - native word size */
const mega      = kilo * kilo;  /* type bits k - native word size */
const kilo      = 1024;         /* type bits k - native word size */
const nl        = '\n';         /* type bits8  */
```

A constant is a value and thus has type `bits`*k*. It is a checked compile-time error to declare a constant of any other type, i.e. `bool`. Thus, `const true = 1 == 1` is illegal.

Table 1 lists the primitive operations available during constant evaluation. The corresponding infix abbreviations can be used to access them.

C-- does not distinguish floating-point constants from integer constants. An expression such as `kilo + pi` is well typed but may not have the value it would have in C.

4.3 Type definitions

decl ⇒ `typedef` *type name* {`,` *name*} [`,`] `;`

The `typedef` declaration introduces a name for a value type. The name and the defining type are totally interchangeable in a program. For example, after the declarations below, the new type `word` is indistinguishable from `bits32`. Type names live in the type name space, which is different from the value name space.

⟨*toplevel example*⟩+≡

```
typedef bits32 code;
typedef bits32 word;
typedef bits32 data;
```

When a named type is used, it is resolved at compile time. It is a checked compile-time error when a type declaration refers directly or indirectly to itself or when another declaration for that name exists in the same scope. Local type declarations have local scope and thus shadow global type declarations.

4.4 Register variables

decl ⇒ [`invariant`] *registers* `;`

registers ⇒ [*kind*] *type* *name* [`=` *hardware-string*] {`,` *name* [`=` *hardware-string*]} [`,`]

A `C--` *variable* should be thought of as a machine register. In particular, one cannot take the address of a `C--` variable, and distinct `C--` variables therefore cannot possibly alias with one another or with any memory location. Unlike real registers, `C--` variables come in unlimited numbers. A `C--` program may let the compiler simulate unlimited variables using a limited number of hardware registers, or it may reserve particular hardware registers to hold the values of particular `C--` variables.

Any register variable may be declared `invariant`. Such a declaration amounts to a guarantee, provided by the front end, that the front-end run-time system will not change the value of the variable while a `C--` computation is suspended.

N.B. propos
extension:
`invisible`

There are two distinct sorts of `C--` variables: global (§4.4.1) and local (§4.4.2).

4.4.1 Global variables

A variable declared at the top level of a `C--` program is called a *global variable* or *global register.* It is visible throughout the entire compilation unit in which it appears, and its value is preserved by interprocedural control flow (call, return, `cut to`) that uses the native `C--` calling convention. (For managing global registers when calling foreign functions, see Section 8.5 on page 56).

For example, given the declaration

⟨*toplevel example*⟩+≡
```
bits64 hp;
```

the implementation maps variable `hp` to a machine register if there is one available, otherwise the implementation maps `hp` to a memory location. The `C--` program cannot tell the difference; it should view all variables as registers.

A declaration of one or more global-register variables may be tagged with a *kind* that requests hardware registers drawn from a particular class.

⟨*toplevel example*⟩+≡
```
"address" bits64 hptr, hplim;
"float"   bits80 epsilon;
```

The set of kinds that may be used with each type must be specified by the `C--` implementation.

A declaration of a global-register variable may also request a particular hardware register by following the variable's name with `=` *hardware-string*, where *hardware-string* is a string literal that names the register requested. For example, the declaration

⟨*toplevel example*⟩+≡
```
bits32 rm = "IEEE 754 rounding mode";
```

makes `rm` a synonym for the hardware register that holds the rounding mode used for floating-point computation.

As with kinds, an implementation must specify which hardware registers may be requested by name.

So that separately-compiled `C--` modules can agree about the way machine registers are allocated, `C--` requires that *all separately compiled modules have* identical *top-level variable declarations*, including kinds and named hardware registers. An implementation of `C--` must detect violations no later than link time. The means of enforcement may vary among implementations, but it is likely that a front end will have to identify one compilation unit as the "master," and that `C--` will ensure that all other compilation units match the master. It is also likely that error messages will be mysterious.

4.4.2 Local variables

A *local variable*, declared in the body of a particular C-- procedure, is private to that procedure, and it dies when the procedure returns.

A declaration of a local variable may not have a kind and may not request a particular hardware register.

4.5 Data sections

A *section* defines the memory layout of initialized and uninitialized data. Both initialized and uninitialized data reserve memory to be used at run time; only initialized data specifies the contents.

A *datum* in a section may define a label, set alignment, or reserve space for an array of cells. If the values of those cells are specified, the reserved space is *initialized data*; otherwise it is *uninitialized data.* A value is specified by a link-time constant expression.

Here is an example that creates initialized data. The example is explained in detail below.

⟨*toplevel example*⟩+≡
```
section "data" {
    foo:  bits32[]  {1::bits32,2::bits32,3::bits32,ff};  // ff is a forward reference
          bits32[]  {1::bits32,2::bits32};
    ff:   bits32[]  {2,3}; bits32[]{ff,foo};
    str:  bits8[]   "Hello world\0";
          bits16[10];
}
```

A compilation unit can define any number of named sections. Two sections with the same name are treated as one section with their contents concatenated. Each section maintains a *location counter.* The location counter tracks the address at which the next datum is stored. Every initialized or uninitialized datum (such as `bits8[10]`) increases the location counter by its size (§4.6). A *label* (such as `foo` or `ff`) captures the value of a location counter and makes it accessible as a link-time constant (§4.5.1).

No padding is added in a section, so it is possible to find any desired datum by starting from a label and adding the correct offset, as in, for example, the expression `bits16[foo+4]`. The address `foo+4` need not point to the beginning of a data element; it can point into the middle.

By default, each location counter is unaligned; even if the target architecture has alignment constaints, C-- never aligns the location counter automatically. A location counter's alignment may be increased using an `align` directive. To align a label (and hence the datum it points to) to a specific boundary, an alignment directive (§4.6.1) has to be placed before the label. In the following example, `baz` and `quux` might or might not be the same address, but `quux` is guaranteed to be aligned on an 8 `memsize` boundary.

⟨*toplevel example*⟩+≡
```
section "data" {
    baz:  align 8;
    quux: bits32 {0};
}
```

It is possible to define a section with no labels; there is no way to access the resulting data.

A C-- procedure is a special form of initialized data (§4.6.2).

A sequence of directives may be wrapped in a `span` (§4.6.3). *[Simon says: Say more here][Norman says: What more needs to be said?]*

4.5.1 Labels

A *label declaration* consists of a name followed by a colon. It associates the name with the current location counter in the section in which it appears. The name therefore denotes a link-time constant which points to a fixed memory location; its type is the native pointer type. The scope of a label is the entire compilation unit in which it appears; thus, a label can be used in its scope before it is declared.

A label provides no information about the type of data it points to.

⟨*toplevel example*⟩+≡
```
section "text" {
    hello:  bits8[] "hello world\n\0";
    /* hello is of the native pointer type */
}
```

4.6 The data directive

A *datum* reserves memory and may also define its initial contents. More specifically, a *datum* is defined by a type, the number of elements of that type to reserve memory for, and optionally the initial values of the elements:

$$\begin{array}{lcl} \textit{datum} & \Rightarrow & \textit{type}\ \big[\textit{size}\big]\ \big[\textit{init}\big]\ \texttt{;} \\ & | & \ldots \end{array}$$

$$\textit{size} \Rightarrow \texttt{[}\ \big[\textit{expr}\big]\ \texttt{]}$$

The amount of memory reserved is determined by the type and the number of elements. Since both the size and the initial values are syntactically optional, a number of variants exist. The general rule is that a program can specify the size explicitly or let the number of initial values determine the size implicitly.

1. *type* `;` Memory is reserved for one element of type *type* with unspecified initial value.
2. *type* `[` *expr* `]` `;` The compile-time constant expression *expr* defines a nonnegative integer n. Memory is reserved for n elements of type *type* with unspecified initial values.
3. *type* `[ ] {` $\big[$*expr* $\big\{$`,` *expr*$\big\}$ $\big[$`,`$\big]\big]$ `} ;` The syntax allows initialization to be specified by zero or more link-time constant expressions, with the comma used either as a separator or a terminator. The number of link-time constant expressions defining the initial values determine a number $n \geq 0$ of elements of type *type*. Memory is reserved for n elements of type *type* and the values from the list of expressions are used to initialize it. It is a checked compile-time error when any expression's type is different from *type*.
4. *type* `[` *expr* `] {` $\big[$*expr* $\big\{$`,` *expr*$\big\}$ $\big[$`,`$\big]\big]$ `} ;` This is the same case as before except that the size given by the compile-time constant must exactly match the number of elements used for the initial values.
5. `bits8 [ ]` *string* `;` The bytes defined by the non-empty string *string* define the amount of reserved memory and its initial value. Unlike a C string, a C`--` string is not implicitly terminated by a null character. If null-termination is desired, the C`--` literal must end in `\0`.
6. `bits8 [` *expr* `]` *string* `;` This is the same case as before except that the number of bytes defined by the string must match the compile-time constant for the size.
7. C`--` is eventually intended to support Unicode in both programs and string literals, but this part of the design is not yet developed. We welcome suggestions.
8. To use any other syntactically possible variant is a checked compile-time error.

A link-time constant as used to define an initial value belongs to one of the following cases:

1. The link-time constant is a compile-time constant.
2. The link-time constant is a label or imported name.
3. The link-time constant is a sum or difference of other link-time constants.

Since the representation of an initialized value might depend on the byte order, the only way to ensure that a reference to the initialized value sees the proper value is to ensure that the reference uses the same type that was used to initialize the value (§7.3). For example, if a datum is initialized with

```
section "data" { foo: bits16{17}; },
```

when that datum is read back with `bits8[foo]`, the value will be 0 or 17 depending on whether the C`--` program specifies `target byteorder big` or `target byteorder little`, but if the same datum is read with `bits16[foo]`, it is guaranteed to be 17 regardless of the byte order that is specified.

4.6.1 Alignment

The `align` *n* directive ensures the location counter is a multiple of *n*, by increasing it if necessary. If the location counter is increased, the contents of any "padding" memory are not specified. The parameter *n* of an alignment directive must be an integer power of two, otherwise this is a checked compile-time error.

In the following example, `one_` is aligned to a 4-byte boundary and `pi_` to an 8-byte boundary. In both cases padding may be inserted: for example, between the last byte of the `bits32` and the first byte of the `bits64` (the datum pointed to by `one_`) there may be padding in order to place the `bits64` on an 8-byte boundary.

⟨*toplevel example*⟩+≡
```
section "data" {
                align 4;
    one_:       bits32 {1};
                align 8;
    pi_:        bits32 {3.1415};
}
```

In a sequence of alignment directives, the directive with the largest parameter defines the alignment:

⟨*toplevel example*⟩+≡
```
section "data" {
             align 1;
             align 16;
             align 8;
    label:  bits32[] {1,2,3};
}
```

In this example, the first `bits32` value `1` is placed on a 16-cell boundary. (The cell is the `target memsize`, typically 8 bits, so on a byte-addressed machine, a 16-cell boundary is a 16-byte boundary.)

4.6.2 Procedures as section contents

Declarations of procedures are initialized data. They can appear in a *section*, possibly interleaved with declarations of *datum*:

⟨*toplevel example*⟩+≡
```
section "data" {
    const PROC = 3;

    bits32[] {p_end, PROC};
    p (bits32 i) {
        loop:
                i = i-1;
                if (i >= 0) { goto loop ; }
                return;
    }
    p_end:
}
```

The example shows how to create a data structure that includes a procedure and a pointer to the end of the procedure. The expression `p_end` is a link-time constant expression and thus can be used to initialize data.

A procedure that appears at top level outside of any `section` declaration is deemed to appear in the `"text"` section.

4.6.3 Spans

In a section, a `span` directive may wrap declarations, procedures, data, and other spans. The C-- run-time system function `Cmm_GetDescriptor` looks up information using the spans that enclose a program point at which computation is suspended. Spans may be used to mark source-code regions of interest.

Needs examp

Syntactically, a `span` directive has two expressions and a body. The first expression, called the *key*, is a compile-time constant expression of the native word size; it identifies all spans sharing the same value for the key. The second expression, called the *value*, is a link-time constant expression of the native pointer type. It contains user-supplied data that is returned by the runtime system when the corresponding span is found to be the smallest span that contains the requested key and encloses the current program counter.

Each *expr* in a span is syntactically restricted: if the expression is not a single lexical token, such as a name or an integer literal, it must be enclosed in parentheses.

4.7 Target Directive

So that a C-- compilation unit may have a well-defined semantics independent of the target machine, we require that the front end specify fundamental properties of the target architecture it expects.

- *Addressing unit*: The `memsize` directive specifies the number of bits in the smallest addressable unit of memory, which is called a *cell*. The default is `memsize 8`.
- *Byte order*: The `byteorder` directive determines the semantics of references to values in memory that are larger than one addressable unit. C-- recognizes only two byte orders: `big` and `little`. For example, the Intel Pentium is a `byteorder little` architecture, and the Sun SPARC is a `byteorder big` architecture. Byte order cannot be defaulted; it must be given explicitly.
- *Pointer size*: The `pointersize` directive specifies the number of bits in a value of the native pointer type. The default is `pointersize 32`.
- *Word size*: the `wordsize` directive specifies the number of bits in a literal constant whose type is not given explicitly. The default is `wordsize 32`.

The following example describes the Intel Pentium:

⟨*toplevel example*⟩+≡
```
target
    memsize 8
    byteorder little
    pointersize 32
    wordsize 32 ;
```

A compilation unit may contain many `target` declarations, but they must be consistent; it is a checked compile-time error if a single compilation unit contains inconsistent target directives. Every compilation unit must specify byte order; it a checked compile-time error if a compilation unit specifies no byte order.

The `target` directives in all compilation units must describe the same architecture. It is an *unchecked* link-time error when target descriptions across compilation units mismatch.

This should checked. Wh the target is encoded into MD5 encode linker symbo should be detectable at link-time.

The `target` directive provides a sanity check and helps ensure that the *meaning* of a C-- program is independent of the target machine, but no C-- compiler is obligate to *translate* an arbitrary C-- program for an arbitrary target machine. For example, a 32-bit C-- compiler need not be able to translate a C-- program containing 64-bit code. Every C-- compiler must advertise what `target` directives it can accomodate for targets it supports.

5 Procedures

A C-- procedure definition plays a dual role: it is both data and code. Like initialized data, it reserves space with specific contents, but the contents are not values; they are target-machine instructions. A procedure also has a dynamic semantics. The front-end run-time system can call a C-- procedure, which can then call another, and so on—which is the whole point, after all.

Overall, C-- procedures are quite a bit like C procedures, but there are many differences:

- A C-- procedure may not only accept an arbitrary number of parameters, but may return an arbitrary number of results.
- The number of arguments passed to a C-- procedure is fixed; there is no support for "varargs."
- Calls to procedures are not typechecked. The number of actual parameters and types and kinds of those parameters at a call site must be identical to the number, types, and kinds of the formal parameters in the definition of the procedure being called. The calling convention at the call site must also be identical to the calling convention in the procedure definition. Any mismatch is an *unchecked* run-time error.
- A procedure cannot be called from an expression.

5.1 Procedure definition

A procedure definition has the following syntax:

procedure ⇒ [*conv*] *name* `(` [*formals*] `)` `{` *body* `}`

formals ⇒ *formal* {`,` *formal*} [`,`]

formal ⇒ [*kind*] [`invariant`] *type name*

- *conv* is the calling convention used to call the procedure (§5.4).
- *name* is name of the procedure; it denotes an immutable (link-time constant) value of the native pointer type. This value can be stored in data structures, etc., and then later retrieved and used in call and `jump` statements (§6.8 and §6.8). Like other code labels, the name of a procedure is visible throughout a compilation unit, or if named in an `export` declaration, throughout the program.
- *formals* are the formal parameters, if any. The formal parameters are in the scope of the procedure body and denote run-time values of the declared types. A call to a procedure passes arguments *by value.* This means that changes to a parameter from within the procedure are not visible at the call site.

  A parameter declared as `invariant` tells C-- that the front-end run-time system promises not to change the value of a parameter while the computation is suspended.

  Unlike C, C-- does not permit a procedure to accept a variable number of arguments.
- A *kind*, which is a string literal, is an implementation-dependent way to guide parameter passing. It does not affect the semantics of a C-- program except to make the program undefined if kinds do not match. A missing kind is equivalent to a kind of `""`. An implementation of C-- should advertise what kinds it understands; most implementations should understand the kinds `"float"` for floating-point numbers, `"address"` for pointers, and `""` for integers and other data.

  (The results returned by a C-- procedure also require kinds, but these kinds appear on `return` statements in the *body*, not as part of the procedure's header.)

- The body of a procedure is a sequence of *decl*s (declarations of local register variables or of stack-allocated data), interleaved with a sequence of *stmt*s (statements). The sequence of statements specifies a control-flow graph. Every path in such a graph must end in `jump`, `return`, or `cut to`; a procedure in which control "falls off the end" is rejected by the C-- compiler with a checked compile-time error.

Because C-- is an unsafe language, a C-- procedure has no return types; types of results are known at call sites and `return`s.

For example, procedure `goo` expects one 32-bit argument. Inside `goo`'s body, the local register variable `x` is declared. `goo` assigns to `x`, then makes a tail call to procedure `bar`.

⟨*toplevel example*⟩+≡
```
goo(bits32 y) {
  bits32 x;

  x = y + 1;
  jump bar(x);
}
```

When a procedure is called, a new *activation* for that procedure is created. The activation holds the values of the procedure's parameters and local register variables. An activation dies when its procedure returns or `jumps` to another procedure, when a younger activation cuts past it to an older activation on the same stack, or when the front-end run-time system cuts or unwinds the stack past that activation.

5.2 Procedure body and nested declarations

$$\textit{body} \Rightarrow \{\textit{decl} \mid \textit{stackdecl} \mid \textit{stmt}\}$$

A procedure body consists of a mixture of declarations, `stackdata` declarations (§5.3), and statements (§6), in any order.

A procedure may contain nested declarations (*decl* in the syntax of Figure 3) of any kind, including `const`, `typedef`, `import` and so on. Each such declaration is visible throughout the *entire* body of the procedure; as long as each declaration occurs somewhere within the procedure's body, the order and placement of the declarations are irrelevant.

Declarations appearing in a procedure body have the same syntax and semantics as if they occurred at top level, with the following exceptions:

Scope. A local declaration within a procedure obeys the following scope rules:

- The name declared is visible throughout the entire procedure body and only within that body.
- If the same name is declared at top level, the local declaration hides the top-level declaration throughout the procedure.

It is a checked compile-time error to declare a single name more than once within a single procedure.

Lifetime. A register variable declared within a procedure refers to a distinct value for each activation of the procedure in which it appears. The value lives only as long as the activation. A local variable cannot be shared between different activations of the same procedure.

Local registers. A local register cannot have a kind, and it cannot be mapped to a hardware register using `=` *string* following the type in its declaration.

5.3 Allocating space on the stack

To handle a high-level value that can't conveniently fit in a register, like a record or an array, C-- can allocate an area in the procedure's activation record. The syntax is the same as for uninitialized data, except the labels and declarations appear in `stackdata { ... }`. No initial value can be provided.

⟨*toplevel example*⟩+≡
```
f2 (bits32 x) {
    bits32 y;

    stackdata { p : bits32;
                q : bits32[40];
    }

    /* Here, p and q are the addresses of the relevant chunks of data.
     * Their type is the native pointer type of the machine.
     */
   return (q - p);
}
```

As with `data`, the names `p` and `q` are immutable values of the native pointer type, but they are not link-time constants; they may be different in every activation of `f`. A C-- program may use address arithmetic on `p` and `q` to refer to any memory location within the `stackdata` definition, but to use address arithmetic on `p` and `q` to refer to a memory location *outside* the `stackdata` definition is an *unchecked* run-time error. Because `p` and `q` don't outlive `f`, it is also an *unchecked* run-time error to use `p` or `q` to refer to memory after the activation containing them has died.

The scope of the names declared within `stackdata` is the entire procedure body (as for nested declarations), and the lifetime of the allocated memory is the same as that of the activation of the enclosing procedure.

C-- does not provide dynamically sized stack allocation.

5.4 Foreign calling conventions

To use a foreign-language calling convention for a procedure, the name of the calling convention should be declared before the procedure name with the `foreign` keyword. Here, `foo` is called using the standard `C` calling convention. It then makes a call to `bar`, using the native C-- calling convention. Then `bar` returns, and finally `ceefun` returns, again using the standard `C` calling convention.

⟨*toplevel example*⟩+≡
```
export ceefun;
foreign "C" ceefun() {
  bits32 x;
  x = bar(x);
  foreign "C" return (x);
}

bar(bits32 a) {
  return (a + 1);
}
```

The calling conventions that every implementation of C-- is required to support are:

1. "C--" is the native convention and is guaranteed to support tail calls.
2. "C" is the standard C calling convention. Every implementation of C-- must say what C-- size and kind go with each C type.

   Depending on the platform, it may or may not be possible to use the run-time interface with a foreign "C" activation on the stack (§8.1.1).
3. "paranoid C" is like the standard C calling convention, except that a "paranoid" C function does not restore callee-saves registers. It would be dangerous to define a C-- function with this convention, because such a function would not be safely callable from C. But it *is* useful to use this "paranoid" convention to call a foreign C function. When calling a foreign function using a paranoid convention, C-- does not rely on the C compiler to save registers properly. This means it is always possible to use the run-time interface with a foreign "paranoid C" activation on the stack (§8.1.1).

Every implementation of C-- must advertise what calling conventions and kinds it recognizes. Implementations are encouraged to support the following kinds:

"float"	A floating-point number
"address"	A memory address or pointer
""	Any non-float, non-pointer data, including signed and unsigned integers, bit vectors, records passed by value, and so on

When calling a foreign function, it is up to the front end to handle parameters that are not passed by value. For example, if a C convention requires that a struct argument be passed by reference, or if it requires that a struct result be replaced by a "hidden" argument that points to a stack slot in the caller's frame, it is up to the front end to generate C-- code that passes a suitable reference, not the struct itself.

6 Statements

A statement can read and write memory and registers, and a statement can change the flow of control. Statements appears only within procedures. Their syntax is as follows:

```
stmt ⇒ span expr expr { body }
     | ;
     | lvalue {, lvalue} [,] = expr {, expr} [,] ;
     | name = %% name ( [actuals] ) {flow} ;
     | if expr { body } [else { body }]
     | switch [[ range ]] expr { {arm} }
     | name :
     | goto expr [targets] ;
     | continuation name ( [kinded-names] ) :
     | cut to expr ( [actuals] ) {flow} ;
     | [kinded-names =] [conv] expr ( [actuals] ) [targets] {flow} ;
     | [conv] jump expr [( [actuals] )] [targets] ;
     | [conv] return [< expr / expr >] [( [actuals] )] ;

body ⇒ {decl | stackdecl | stmt}
```

6.1 Span

The `span` statement provides a key-value pair that can be looked up by the C-- run-time system. A span encloses a sequence of statements; whenever control is suspended within the span (e.g., at a call site), the C-- run-time system function `Cmm_GetDescriptor` can "see" the key-value pair. More precisely, because spans can be nested, when `Cmm_GetDescriptor` is called on an activation suspended at a particular point, it is given a key, and it returns the value of the smallest enclosing span with that key (§8.4). The dynamic semantics of the `span` is the semantics of the statements it encloses. A `span` can be also part of a section, where it encloses procedures (§4.6.3).

Syntactically, a `span` directive has two expressions and a body. The first expression, called the *key*, is a compile-time constant expression of the native word size; it identifies all spans sharing the same value for the key. The second expression, called the *value*, is a link-time constant expression of the native pointer type. It contains user-supplied data that is returned by the runtime system when the corresponding span is found to be the smallest span that contains the requested key and encloses the current program counter.

Each *expr* in a span is syntactically restricted: if the expression is not a single lexical token, such as a name or an integer literal, it must be enclosed in parentheses.

6.2 Empty statement

The empty statement, which is written as a bare semicolon (`;`), can appear anywhere a statement can. It has no effects.

6.3 Assignment

C-- assignments are *multiple assignments*; C-- computes the values of all right-hand sides and addresses before performing any assignments. An assignment statement assigns values from a list of expressions or a value returned by a primitive operator.

$$\begin{array}{lll} \mathit{stmt} & \Rightarrow & \mathit{lvalue}\ \big\{\texttt{,}\ \mathit{lvalue}\big\}\ \big[\texttt{,}\big]\ \texttt{=}\ \mathit{expr}\ \big\{\texttt{,}\ \mathit{expr}\big\}\ \big[\texttt{,}\big]\ \texttt{;} \\ & | & \mathit{name}\ \texttt{=}\ \texttt{\%\%}\ \mathit{name}\ \texttt{(}\ \big[\mathit{actuals}\big]\ \texttt{)}\ \big\{\mathit{flow}\big\}\ \texttt{;} \end{array}$$

The type of each *lvalue* in an assignment must match the type of the corresponding expression on the right-hand side (§7.5). To violate this rule is a checked compile-time error.

$$\begin{array}{lll} \mathit{lvalue} & \Rightarrow & \mathit{name} \\ & | & \mathit{type}\ \texttt{[}\ \mathit{expr}\ \big[\mathit{assertions}\big]\ \texttt{]} \end{array}$$

Either an *lvalue* is the name of a register variable or it is a reference to memory. The rules governing references to memory are explained in Section 7.3, but in brief, *type* gives the size of the location to which the right-hand value is assigned, *expr* gives the address of that location, and any *assertions* make claims about aliasing and alignment.

Here is an example of a multiple assignment, including a swap instruction.

⟨*toplevel example*⟩+≡
```
swap (bits32 x) {
    bits32 p,q;

    p,q = x, x+1::bits32;
    bits32[p], bits32[q] = bits32[q], bits32[p];
    return;
}
```

The `%%` form for assigning the results of a primitive operator is intended to provide clients a way to recover from a primitive operation that may fail. For example, it may be possible to provide an alternate-return continuation (§6.8.1) to detect division by zero. For this version of the C-- specification, however, the details have not yet been worked out.

6.4 Conditional statement

Conditional execution of code is accomplished with the `if` statement. It has the following syntax:

$$\mathit{statement} \Rightarrow \texttt{if}\ \mathit{expr}\ \texttt{\{}\ \big\{\mathit{body}\big\}\ \texttt{\}}\ \big[\texttt{else \{}\ \big\{\mathit{body}\big\}\ \texttt{\}}\big]$$

The expression *expr* is evaluated at run time and must have type `bool`; that is, it must denote a condition, not a value. Most commonly, such an expression will be the result of applying a relational operator, but this is not a requirement.

As in C, the `else` branch is optional, and both statement blocks may be empty, but unlike C, C-- requires the curly braces even for single statements, as here:

```
if x == 0 { x = x + 1;}
```

Also unlike C, C-- does not require the condition to be parenthesized.

When the condition is true, the block of statements immediately following the condition is executed. Otherwise, if an optional `else` branch has been specified, its block of statements is executed. Execution of either block resumes at the first statement after the `if` or `if/else`, unless of course the code in the block changes flow of control with a `goto`, `return`, `cut to`, or `jump`.

Here's an example:

⟨*toplevel example*⟩+≡

```
f0(bits32 x) {
    bits32 y;

    y = 0;
    if (y >= bits32[foo+8]) {
      y = y + 1;
      return (y);
    } else {
      x = x - 1;
      if x != 0 {
         y = y + 2;
      }
      return (y);
    }
  }
```

6.5 The `switch` statement

A `switch` statement evaluates an expression *expr* and based on the value, executes an *arm*. Each arm is guarded by *range*s of (compile-time constant) expressions. The `switch` statement executes the first arm (in the order of appearence) that is guarded by a value that is equal to the value of the evaluated expression. Here's the syntax:

$$\begin{array}{ll} \textit{stmt} & \Rightarrow \texttt{switch}\ \textit{expr}\ \texttt{\{}\ \big\{\textit{arm}\big\}\ \texttt{\}} \\ \textit{arm} & \Rightarrow \texttt{case}\ \textit{range}\ \big\{\texttt{,}\ \textit{range}\big\}\ \big[\texttt{,}\big]\ \texttt{:}\ \texttt{\{}\ \textit{body}\ \texttt{\}} \\ \textit{range} & \Rightarrow \textit{expr}\ \big[\texttt{..}\ \textit{expr}\big]^{3} \end{array}$$

The meaning of a range $e_1..e_2$ is all the bit patterns n such that e_1`` `leu` ``n and n`` `leu` ``e_2; the meaning of a list of ranges is their union. This use of unsigned comparison may seem a bit strange, but it gives the expected results for both positive and negative two's-complement ranges. (For ranges that cross zero, however, the results may be most *unexpected*. Beware!) A range with e_1 `` `gtu` `` $e2$ is empty. Ranges are evaluated at compile-time and thus are compile-time constant expressions.

- The expression *expr*, called the *scrutinee*, is evaluated at run time and yields a value.
- Every *arm* is guarded by a non-empty list of non-empty ranges. The types of all values of the range must be equal *and* must match the type of the expression. The switch executes the first arm (in order of appearance) that is guarded by a range into which falls the value of the scrutinee. Unless the arm changes the flow of control, control then flows from the arm to the statement after the `switch`. Unlike in C, "fall through" between arms is impossible.

  An implementation of C-- may assume that earlier arms are more likely to be executed.

- It is an *unchecked* run-time error to execute a switch statement in which the value of the scrutinee does not appear in the ranges of any of the arms. The arms must cover all possible values of *expr*. Unlike C, C-- does not include a `default` arm in a `switch` statement. A default arm can be simulated by guarding the last arm with a range such as `0 .. 0xffffffff`.

In the following example, expression `x+23` is assumed to yield a value in between 0 and 7. If the value is 1, 2, or 3, then the first branch is taken. If the value is 5, then the second branch is taken. If the value is 0, 4, 6, or 7, then the third branch is taken.

⟨*toplevel example*⟩+≡
```
f6 (bits32 x, bits32 y) {
    import bits32 f;

    switch (x + 23) {
      case 1,2,3   : { y = y + 1;}
      case 5       : { y = x + 1; x = y;}
      case 0,4,6,7 : { y = f();
                       if y == 0 { x = 1;}
                     }
    }
    return (x, y);
}
```

6.6 Control labels and `goto`

A control label is used as the target of a `goto` to change control flow *within* a procedure. Like a data label, a control label is declared by giving its name, followed by a colon. The name denotes a link-time constant value of the native pointer type. Unlike C, C-- provides first-class labels and computed `goto`s. As in an assembler, the scope of a control label is the entire compilation unit in which the label appears. In particular, a label can appear in initialized data in another section.

The syntax is

stmt ⇒ *name* :
  | `goto` *expr* [`targets` *name* {, *name*} [,]] ;

In general, the `goto` is a "computed" or "indirect" goto. The expression *expr* must evaluate to the value of one of the labels in the `targets` list; otherwise, executing the `goto` is an *unchecked* run-time error. Each label in the targets list must be defined in the local procedure, otherwise this is a checked compile-time error. In the special case where the *expr* consists of the name of a (local) label, the `targets` list may be omitted, and it defaults to that label. Otherwise, it is a checked compile-time error to omit the `targets` list.

⟨*toplevel example*⟩+≡
```
f3 () {
  stackdata {
    bar: bits8[64];
  }

  f3_label:
    bits64[bar]     = 18::bits64;
    bits64[bar+4*8] = bits64[bar];
    if (%zx32(bits8[bar+4*8]) == 18) { return; }
    goto f3_label;
}
```

6.7 Continuations and `cut to`

A continuation is a bit like a control label, except that it enables control flow *between* procedures, not just within a single procedure. Like a label definition, a continuation statement marks a point of control and introduces a name that has a C-- value. But unlike a control label, a continuation encapsulates a particular *activation* of a procedure as well as a point in the source code. Also, a continuation can take parameters.

The name of the continuation denotes a value of the native pointer type. The name is visible only inside the procedure in which it appears. This value, which encapsulates a stack pointer and program counter, is different for every activation of the procedure in which the `continuation` appears. The value is first-class—that is, it can be passed around and stored in data structures—but it has a limited lifetime: it dies when its procedure activation dies.

```
stmt    ⇒ continuation name ( [[kind] name {, [kind] name} [,]] ) :
        | cut to expr ( [actuals] ) {flow} ;

flow    ⇒ also (cuts | unwinds | returns) to name {, name} [,]
        | also aborts
        | never returns
```

The `cut to` statement changes the flow of control to a previously captured continuation value *expr*: control returns to the activation of the continuation and resumes at the continuation statement. The continuation being `cut to` is typically fetched from a data structure or register.

A `cut to` statement is the *only* way to pass control to a continuation. It is a checked compile-time error for control to fall through to a `continuation` statement or for a continuation to be on the targets list of a `goto`.

A continuation can receive values that are passed from `cut to`. To receive values, the continuation statement lists local variables into which the values should be placed. The values passed from `cut to` are stored in these variables. In is an *unchecked* run-time error when the number, types, and kinds of values passed by `cut to` do not match the number, types, and kinds of the formal parameters of the receiving continuation.

A continuation's value is of the native pointer type. It is a checked compile-time error to `cut to` a value not of the native pointer type. It is an *unchecked* run-time error to `cut to` a continuation inconsistent with the *flow* annotation on the `cut to`:

- The `also aborts` annotation indicates that the `cut to` may cut to a continuation in an older activation in the same stack, destroying the current activation. This is the default if {*flow*} is empty.
- The `also cuts to` annotation names all continuations in the current activation that may be cut to.
- The `also returns to` and `also unwinds to` annotations are meaningless for a `cut to`. To include one of these annotations on a `cut to` is a checked compile-time error.
- The `never returns` annotation is superfluous for a `cut to`, but it is permissible.

A continuation is live as long as its procedure activation is live. It is an *unchecked* run-time error to `cut to` a dead contінutation.

6.8 Procedure calls, tail calls, and returns

A call statement, which is also called a *call site*, *suspends* a procedure activation and starts executing another procedure. This procedure may call other procedures, and so on, but eventually it resumes the suspended activation by executing a *matching return*. Alternatively, another procedure may execute the matching return after a sequence of *tail calls*. A tail call does not suspend an activation; instead, the activation making the tail call dies and is replaced by an activation of the called procedure. For this reason, a tail call does not have a matching return. Finally, an activation suspended at a call site may be resumed at a continuation, which is reached by a `cut to`, a `return`, or through the run-time system (§8.4). All methods of transferring control between procedures—including call, tail call, return, and `cut to`—may pass values.

6.8.1 Calls and tail calls

Because a procedure call is a statement, not an expression, it cannot be used inside an expression. For example, a phrase such as `y = f(g(x)) + 1;` is not legal C--.

In general, a C-- call or tail call is indirect: the procedure to be called is computed by an expression whose value need not be known until run time. The syntax of call and tail call is as follows:

```
stmt         ⇒ [kinded-names =] [conv] expr ( [actuals] ) [targets] {flow | alias} ;
             | [conv] jump expr [( [actuals] )] [targets] ;

kinded-names ⇒ [kind] name {, [kind] name} [,]

actuals      ⇒ [kind] expr {, [kind] expr} [,]

targets      ⇒ targets name {, name} [,]

flow         ⇒ also (cuts | unwinds | returns) to name {, name} [,]
             | also aborts
             | never returns

alias        ⇒ (reads | writes) [name {, name} [,]]
```

- The results of a call are assigned to variables. It is an *unchecked* run-time error when the number, types, or kinds of the variables at a call site differs from the number, types, or kinds of the values at the matching return.

  A tail call has no results; the calling activation dies immediately.

- For both a call and a tail call, an optional *conv* identifies the calling-convention to use when calling the procedure. A missing calling convention is equivalent to `foreign "C--"`. Other calling conventions are needed for interoperability with foreign code (§5.4). Every convention supports calls, but a foreign convention is not guaranteed to support tail calls. In particular, the foreign `"C"` convention does *not* support tail calls.

  It is an *unchecked* run-time error if the calling convention at a call site differs from the calling convention at the definition of the procedure called. It is also an *unchecked* run-time error if the calling convention at the call site differs from the calling convention at the matching return.

- For both a call and a tail call, the expression *expr* must evaluate to the address of a procedure. It is a checked compile-time error if *expr*'s type is not the native pointer type; it is an *unchecked* run-time error if *expr* evaluates to something that is not a procedure.

- For both a call and a tail call, the list of expressions *actuals* is evaluated at run time and passed by value to the called procedure. Because in a correct C-- program, evaluating an expression has no side effect, order of evaluation is not specified.

  A *kind* attached to an actual parameter has an implementation-dependent meaning—it guides the C-- compiler in putting the actual parameter in an appropriate location. A missing kind is equivalent to a kind of `""`. It is an *unchecked* run-time error if the number, types, or kinds of actual parameters differs from the number, types, or kinds of formal parameters at the definition of the procedure called.

- For both a call and a tail call, the optional list "`targets`" enumerates the names of the procedures that *expr* can evaluate to. If a list is present, the C-- compiler may use it to optimize (for example, by using a nonstandard calling convention). If the list is supplied, it is an *unchecked* run-time error if *expr* evaluates to the address of a procedure not on the list.

- When a called procedure returns normally, the flow of control continues at the statement following the call. The called procedure, however, can use `cut to`, `return`, or the C-- run-time system to pass control to a continuation and effectively perform a non-local return. The call must be annotated accordingly using *flow* annotations. It is an *unchecked* run-time error when the called procedure transfers control to a continuation not listed in the *flow* annotations, or when it transfers control by a means not consistent with the *flow* annotations. The annotations' requirements are as follows:
  - An `also aborts` indicates that the called procedure may cut to a continuation in an older activation in the same stack, destroying the current activation. This is the default when no annotation is given.
  - An `also cuts to` lists continuations to which the called procedure can transfer control using `cut to`.
  - An `also unwinds to` lists continuations to which the called procedure can transfer control with the help of the C-- run-time system (§8.4).
  - An `also returns to` lists continuations to which the called procedure can transfer control using `return`, as discussed below.
  - A `never returns` annotation indicates that control never returns using the normal return continuation, but only by one of the paths indicated by the other annotations above.

  A continuation listed in `also cuts to`, `also unwinds to`, or `also returns to` must be defined in the same procedure as the call site, or else it is a checked compile-time error.
- The optional *alias* annotation may identify *alias names* (§2.3.2) that are read or written by the procedure call. These alias names are used to determine when the call may interfere with a load or a store operation. If not annotated, the call is assumed to read or write any legal memory location, which means that it may interfere with any load or store. If annotated with an empty list of names, the call is assumed not to read or write any location visible to the calling context, which means it interferes with no loads or stores. Otherwise, the meaning of the list of names is determined by the front end (§7.3.2).
- A future version of this specification may describe additional annotations by means of which a front end will be able to tell C-- how a call depends on or affects the values of global register variables.

A continuation reached only by `also cuts to` or `also returns to` may have any arguments at all. But a continuation that is reached by `also unwinds to` must have arguments whose sizes and kinds correspond to C types. This restriction is necessary because such arguments are passed directly from C using the `Cmm_MakeUnwindCont` function (§8.4).

Here is an example of using a tail call to write an infinite loop with no stack growth:

⟨*toplevel example*⟩+≡
```
f7(bits32 x, bits32 y) {
    jump f7(y, x);   /* Loop forever */
}
```

6.8.2 Returns

A `return` statement transfers control (and results) back to a matching call site. A return kills the activation of the procedure containing the return, so the once the return is executed, the local variables and continuations of the procedure die. The syntax is:

stmt	⇒ [*conv*] `return` [`<` *expr* `/` *expr* `>`] [`(` [*actuals*] `)`] `;`
actuals	⇒ [*kind*] *expr* {`,` [*kind*] *expr*} [`,`]

The return statement may be qualified with the calling convention to be used (§5.4); a missing qualification is equivalent to `foreign "C--"`. It is an *unchecked* run-time error for the convention at a return to differ from the convention at the matching call site.

The *actuals* are the expressions whose values are returned. A procedure may return any number of values. Because in a correct `C--` program, evaluating an expression has no side effect, order of evaluation is not specified. It is an unchecked run-time error for the number, kinds, or types of the actuals to differ from the number, kinds, or types of the variables on the left-hand side of the matching call site. As in a call, a missing kind is equivalent to a kind of `""`.

A `return` statement normally passes values to the left-hand side of its call site, and execution of the suspended activation resumes with the statement following the call site. But a `return` may also return to a continution that is listed in the call site's `also returns to` annotation. In this case, the *actuals* of the return are assigned to the formal parameters of the continuation, and execution of the suspended activation resumes with the continuation. The choice of continuation is made by using the form `return <` i `/` n `>` [`(` [*actuals*] `)`]. The matching call site must be annotated with exactly n `also returns to` continuations, and execution resumes with the continuation numbered i, where continuations are numbered from 0. If $i = n$, execution resumes after the matching call site, just like a normal return. If the `<` i `/` n `>` notation is missing, it is equivalent to `< 0 / 0 >`. The expressions i and n are evaluated at compile time. It is a checked compile-time error if i or n is not a compile-time constant expression (§4.2), if i or n has a type that is not the native word type, or if i falls outside the range $0..n$. Also, it is an *unchecked* run-time error if n differs from the number of `also returns to` continuations at the matching call site. Finally, each of the expressions i and n is syntactically restricted: either the expression is a single name or literal, or else it is enclosed in parentheses.

Here is an example showing `return` with multiple values.

⟨*toplevel example*⟩+≡
```
f4() {
    bits32 x, y;
    x, y = f5(5);
    return (x,y);
}
f5(bits32 x) {
   return (x, x+1);
}
```

7 Expressions

A C-- expression can be a literal, a name, a reference to a value in memory, or a primitive operator applied to other expressions (see Figure 4 on page 15). Evaluating a C-- expression produces either a k-bit *value* or a Boolean *condition*. A value may be stored in a variable, passed as a parameter, returned as a result, or fetched from and stored in memory. A condition governs control flow in `if`. An expression that produces a k-bit value has type `bits`*k*; an expression that produces a condition has type `bool`. As in an assembler, distinctions between signed integers, unsigned integers, pointers, and floating-point values are in the operators, not in the types of operands.

The type of an expression is determined by the types of the names and operators that appear in the expression. C-- has no overloading, no implicit conversions, and no typecasts.

An implementation of C-- need not be able to generate code for a C-- program containing expressions of type `bits`*k* for arbitrary k. Instead, each implementation is required to advertise what expressions it can generate code for. It is reasonable to expect that a C-- compiler generating code for a 32-bit platform will be able to translate expressions of type `bits32`, and so on.

7.1 Literals

$$
\begin{aligned}
\textit{expr} \Rightarrow\ & \textit{int}\ \big[\texttt{::}\ \textit{type}\big] \\
\mid\ & \textit{float}\ \big[\texttt{::}\ \textit{type}\big] \\
\mid\ & \texttt{'}\ \textit{char}\ \texttt{'}\ \big[\texttt{::}\ \textit{type}\big]
\end{aligned}
$$

The simplest building blocks of expressions are literals. Any literal may be given an explicit size by using the notation `::` *type*. There are literal expressions for integers, floating-point numbers, and characters, but not strings; string literals can be used only to initialize data (§4.6).

Integer literals default to the native word size. An integer literal produces the bit vector that is the two's-complement representation of the integer (§3.3.3).

Floating-point literals default to the native word size. The meaning of a floating-point literal (down to the bit level) will be covered only under a future version of this specification. The intent is that the a decimal floating-point literal will produce the "most appropriate" IEEE 754 bit vector, whatever that means. A client that wants a definite IEEE floating-point value is advised to emit an integer literal (perhaps using hexadecimal notation) that denotes the appropriate bit vector.

Character literals have type `bits8` by default (§3.3.5).

Both floating-point and integer literals have type `bits`*k* and thus are not distinguished in the type system (§7.5). So for example, the expression `3.1415 + 1` is legal in C--, but since `+` denotes two's-complement integer addition, the result might be unexpected: the bit pattern of `3.1415` is considered as an integer and added to `1`, so essentially what you get is `3.1415` plus one unit in the last place. In particular, `3.1415` is *not* rounded to `3` and thus the result is not `4`. More useful expressions include `%f2i32(3.1415, rm) + 1` and ``3.1415 `fadd` %i2f32(1, rm)``, where `rm` refers to rounding modes.

7.2 Names

> *expr* ⇒ *name*

A *name* in an expression denotes one of the following values:

1. A C-- register register variable; its type is as declared.
2. A constant; its type is inferred from its defining expression.
3. A code label (possibly a procedure); its type is the native pointer type.
4. A data label, either in a section or on the stack; its type is the native pointer type.
5. A continuation; its type is the native pointer type.

Every has a type of the form `bits`k. Since a condition for an `if` statement has type `bool`, a name cannot denote a condition. A value of type `bits`k can be converted to a condition of type `bool` by comparing it with zero.

7.3 References to memory

> *expr* ⇒ *type* `[` *expr* $\left[\textit{assertions}\right]$ `]`

A memory reference *type* `[`*expr*`]` consists of an address *expr* and a *type*, possibly with *assertions*. The type *type* is of the form `bits`k and describes the size of the memory object being read (or written); k must be a multiple of an addressable memory unit as declared by `target memsize` (§4.7). The typical `memsize` is 8, so typically `bits8` and `bits16` are legal but `bits17` is not. The address *expr* determines the location(s) in memory to which the reference refers. The type of the address must be the native pointer type, which is checked at compile time. As described below, the reference may include assertions that describe the address's alignment or the (abstract) set of values from which it may be drawn.

A memory reference refers to a location, or possibly an aggregate of locations. For example, a reference of type `bits32` might refer to an aggregate of four 8-bit bytes. When a reference appears as an *expr*, it denotes the contents of its location(s). When the reference appears as an *lvalue*, on the left-hand side of an assignment, it denotes the location(s) into which the corresponding value on the right-hand side should be stored (§6.3).

When a memory reference refers to an aggregate of locations—that is, when *type* is larger than will fit in a single addressable memory unit—addressable memory units are aggregated according to the `target byteorder` (§4.7). For example, if a reference of type `bits32` refers to an aggregate of four 8-bit bytes, and if the `target byteorder` is `big`, then the first byte (the one at *expr*) forms the most significant byte of the reference, and so on.

It is an *unchecked* run-time error for a C-- program to refer to a location in memory to which it does not have rights (§2.3.1).

Here are two examples, both of which assume that `target memsize` is 8.

- The assignment "`bits8[label+4] = 'A';`" stores the byte `0x41` (value of charcter literal `'A'`) into the location `label+4`; `label` must be a value of the native pointer type.
- The expression "`bits32[label + i*4]`" denotes the contents of the specified memory location; the expression `label+i*4` must evaluate to a native pointer that is aligned on a 4-byte boundary.

7.3.1 Alignment and memory access

A memory reference of the form `bits`*k* `[`*expr*`]` implicitly asserts that *expr* evaluates to an address that is aligned on an *n*-byte boundary, where *k* is *n* times the `target memsize`. If the address is not so aligned, it is an *unchecked* run-time error. For example, in the typical case of `target memsize 8`, the address in a `bits16` reference must be aligned to 2 bytes. If the front end cannot guarantee the expected alignment, it must annotated the reference with an alignment assertion, which gives the maximum alignment that can be expected:

assertions ⇒ `aligned` *int*

Such an assertion guarantees that the address is aligned on an *int*-unit boundary; to violate the guarantee is an *unchecked* run-time error. For example, a completely unaligned access should be annotated `aligned 1`. Examples:

- The assignment "`bits64[label aligned 4] =` *expr* `;`" is a 8-byte store to a 4-byte aligned address.
- The expression "`bits32[label aligned 1]`' is a 4 byte-reference at an unaligned address.

7.3.2 Aliasing assertions

A typical front end usually has some information about how memory references must differ (may not alias). For example, a front end may be able to identify stores to a newly allocated, uninitialized object, which cannot possibly interfere with loads from an old, initialized object. By communicating this information to the C-- back end, the front end enables to the back end to do a better job scheduling loads, for example.

The front end communicates this information in two parts.

- Each memory reference may be annotated with a list of names. The syntax is

  assertions ⇒ `in` *name* $\{$ `,` *name*$\}$

- The front end provides a procedure that, when presented with two lists of names, tells whether the corresponding references may alias. The mechanism by which the front end provides this procedure is not covered under this specification.

It is anticipated that there will be a simple default mechanism that will suffice for many clients, but no such mechanism is covered by this specification.

7.4 Applications of primitive operators

C-- has over 75 primitive operators. An implementation of C-- should advertise what subset it supports.

Each primitive operator produces one result, the type and size of which is determined by the name of the operator and the types and sizes of its operands. Primitive operators are free of side effects: they do not change memory, registers, or the flow of control. If the application of a primitive operator causes a system exception, such as division by zero, this is an *unchecked* run-time error. (A future version of this specification may provide a way for a program to recover from such an exception.)

Table 2 lists the primitive operators grouped by function. Table 4 lists and explains the operators, dividing them into just two groups—floating-point operators and other operators—and in alphabetical order within each group. Many primitive operators have polymorphic types, which are explained in §7.4.2.

Floating-point operations

Comparisons

`feq`	Equal
`fne`	Unequal
`fge`	Greater than or equal to
`fgt`	Greater than
`fle`	Less than or equal to
`flt`	Less than
`fordered`	Ordered (comparable)
`funordered`	Unordered (incomparable)

Arithmetic

`fabs`	Absolute value
`fadd`	Add
`fdiv`	Divide
`fmul`	Multiply
`fmulx`	Multiply, extended
`fneg`	Negate
`fsqrt`	Square root
`fsub`	Subtract

Conversions

`f2f`*k*	Convert float to float (change size)
`i2f`*k*	Convert integer to float
`f2i`*k*	Convert float to integer

Particular values

`NaN`*k*	Not a number
`minf`*k*	Minus infinity
`pinf`*k*	Plus infinity
`mzero`*k*	Minus zero
`pzero`*k*	Plus zero

Rounding modes

`round_down`	Toward $-\infty$
`round_nearest`	Toward nearest
`round_up`	Toward $+\infty$
`round_zero`	Toward 0

Non-floating-point operations

Arithmetic

`add`	Add
`addc`	Add with carry in
`carry`	Carry out (from `addc`)
`sub`	Subtract
`subb`	Subtract with borrow in
`borrow`	Borrow out (from `subb`)
`neg`	Negate
`mul`	Multiply (signed or unsigned)
`mulux`	Multiply unsigned, extended
`mulx`	Multiply signed, extended
`div`	Signed divide (round to $-\infty$)
`quot`	Signed quotient (round to 0)
`divu`	Unsigned divide (round to 0)
`mod`	Signed modulus (with `div`)
`rem`	Signed remainder (with `quot`)
`modu`	Unsigned modulus (with `divu`)

Overflow checking

`add_overflows`	`mulu_overflows`
`div_overflows`	`quot_overflows`
`mul_overflows`	`sub_overflows`

Boolean operations

`bit`	Convert Boolean to bit
`bool`	Convert bit to Boolean
`conjoin`	Boolean and
`disjoin`	Boolean or
`false`	Falsehood
`not`	Boolean complement
`true`	Truth

Bit operations

`and`	Bitwise and
`com`	Bitwise complement
`or`	Bitwise or
`xor`	Bitwise exclusive or
`rotl`	Bit rotate left
`rotr`	Bit rotate right
`shl`*	Shift left
`shra`*	Shift right, arithmetic
`shrl`*	Shift right, logical
`popcnt`	Population count

*Table 4 describes restrictions on operands

Comparisons

`eq`	Equal
`ne`	Unequal
`ge`	Greater than or equal to (signed)
`gt`	Greater than (signed)
`le`	Less than or equal to (signed)
`lt`	Less than (signed)
`geu`	Greater than or equal to, unsigned
`gtu`	Greater than, unsigned
`leu`	Less than or equal to, unsigned
`ltu`	Less than, unsigned

Width changing

`lobits`*k*	Extract low bits
`sx`*k*	Sign extend
`zx`*k*	Zero extend

Table 2: Primitive operators grouped by function

Operator	**Associativity**	**Meaning**
`~` `-` (unary)	right	`com neg`
`/ * %`	left	`divu mul modu`
`- +`	left	`sub add`
`>> <<`	left	`shrl shl`
`&`	left	`and`
`^`	left	`xor`
`|`	left	`or`
`>= > <= < != ==`	none	`geu gtu leu ltu ne eq`
`` ` ``*name*`` ` ``	none	*name*

Operators at the top of the table have highest precedence; operators on the same line have equal precedence.

Table 3: Infix operators with precedence and associativity

7.4.1 The syntax of primitive operators

$$
\begin{array}{lll}
\textit{expr} & \Rightarrow & \texttt{\%}\ \textit{name}\ \left[\texttt{(}\ \left[\textit{actuals}\right]\ \texttt{)}\right] \\
& | & \textit{expr}\ \textit{op}\ \textit{expr} \\
& | & \texttt{\~{}}\ \textit{expr} \\
& | & \texttt{-}\ \textit{expr} \\
\\
\textit{op} & \Rightarrow & \texttt{`}\ \textit{name}\ \texttt{`}\ \Big|\ \texttt{+}\ \Big|\ \texttt{-}\ \Big|\ \texttt{*}\ \Big|\ \texttt{/}\ \Big|\ \texttt{\%}\ \Big|\ \texttt{\&}\ \Big|\ \texttt{|}\ \Big|\ \texttt{\^{}}\ \Big|\ \texttt{<<}\ \Big|\ \texttt{>>}\ \Big|\ \texttt{==}\ \Big|\ \texttt{!=}\ \Big|\ \texttt{>}\ \Big|\ \texttt{<}\ \Big|\ \texttt{>=}\ \Big|\ \texttt{<=}
\end{array}
$$

Each application of a primitive operator takes one of four syntactic forms:

- In standard prefix form, the operator's name is preceded by a "`%`" symbol, and the arguments appear in parentheses, as in `%mul(n, m)`. Every operator can be written in this form.
- In symbolic infix form, the operator's symbol is written between the two arguments, as in `n * m`. Only operators listed Table 3 have symbols and can be written in this form; the table shows the associativity, precedence, and meaning of each operator symbol. (The associativity and precedence of infix operators are as in C, but there is a big difference: in C`--`, an infix operator is an abbreviation for *exactly one* primitive operator. For example, `*` is always integer multiplication.)
- In backquoted infix form, the operator's name is written in backquotes between its two arguments, as in ``n `mul` m``. Only binary operators may be written in this form. A backquoted operator has least precedence and no associativity; when multiple backquoted operators are used in the same expression, they must be disambiguated with parentheses.
- In symbolic prefix form, the operator's symbol appears before its argument. There are only two prefix operators: `~` *expr* means `%com(`*expr*`)` (bitwise complement), and `-` *expr* means `%neg(`*expr*`)` (integer negation).

The names of primitive operators do not occupy the same name space as other C`--` names, so for example a procedure named `add` can coexist with the primitive `%add`. The set of primitive operators is fixed at compile time; an expression cannot call a user-defined procedure.

The lexical treatment of the percent sign depends on whitespace: If a percent sign is followed by whitespace, it is an infix operator. If a percent sign is followed by a letter, it is part of the standard prefix form of an operator. For example, `10 % b` denotes the same expression as `%modu(10, b)`, but `10%b` is parsed as `10` followed by the name of a primitive `b`.

7.4.2 Primitive operators and types

Most primitive operators are *polymorphic*—that is, they accept arguments of more than one type. For example, `add` can be use to add values of any width. We write the type of a polymorphic operator using $\forall$; for example,

$$\begin{array}{ll} \texttt{add} & \forall\alpha.\texttt{bits}\alpha \times \texttt{bits}\alpha \rightarrow \texttt{bits}\alpha \\ \texttt{eq} & \forall\alpha.\texttt{bits}\alpha \times \texttt{bits}\alpha \rightarrow \texttt{bool} \end{array}$$

The type of `add` can be read "for any width α, `add` takes two values of type `bits`α and delivers a result of the same type". In general, we form the type of an operator, *optype*, as follows:

$$\begin{array}{lcll} \textit{optype} & ::= & t_1 \times \ldots \times t_n \rightarrow t & \text{monomorphic operator, } n \geq 0 \\ & | & \forall\alpha.t_1 \times \ldots \times t_n \rightarrow t & \text{polymorphic operator, } n \geq 0 \\ t & ::= & \texttt{bool} & \\ & | & \texttt{bits}k & \text{some particular } k > 0 \\ & | & \texttt{bits}\alpha & \alpha \text{ bound by } \forall\alpha \text{ in a polymorphic type} \end{array}$$

In C--, it is always possible to deduce the type of an operator application from the types of its arguments. To enable this deduction, C-- requires that in any application of the primitive operators `f2f`k, `i2f`k, `f2i`k, `NaN`k, `lobits`k, `minf`k, `mzero`k, `pinf`k, `pzero`k, `sx`k, or `zx`k, the k in the operator's name must be replaced by the width of the desired result. For example, to zero-extend a byte in memory to a 32-bit word, one might write `%zx32(bits8[a])`. To zero-extend the same byte to 64 bits, one would write `%zx64(bits8[a])`.

7.5 Typing rules for expressions

An expression in C-- has a single type, which is either `bits`k or `bool`. The type (and k) are identified at compile time, according to the following rules.

- The type of a literal annotated with `::` *type* is *type*.
- The type of an unannotated integer or floating-point literal is the native word type, i.e., `bits`k, where k is the `target wordsize`.
- The type of an unannotated character literal is `bits8`.
- The type of a name declared as a register variable is its declared type.
- The type of a name defined as a constant expression is the type of the constant expression.
- The type of a name that is imported or is the name of a continuation or a label (in initialized data; in `stackdata`; or in a procedure, including the name of the procedure) is the native pointer type, i.e., `bits`k, where k is the `target pointersize`.
- The type of a reference to memory memory *type*`[`*expr* $\big[$*assertions*$\big]$`]` is the declared *type*.
- The type of an application of a primitive operator `%`*name*`(`$\big[$*actuals*$\big]$`)` is the type returned returned by the primitive operator, provided the application is well typed. (The type of an application notated in infix or prefix-symbolic form is the same as the type of the same application notated in standard prefix form.)

  An application is well typed if (a) it is possible to instantiate the operator's type (by substituting for the $\forall$-bound variable, if any) to $t_1 \times t_2 \times \ldots \times t_n \rightarrow t$; (b) if the operator is `f2f`k, `i2f`k, `f2i`k, `NaN`k, `lobits`k, `minf`k, `mzero`k, `pinf`k, `pzero`k, `sx`k, or `zx`k, then t is type `bits`k; and (c) the operator is applied to n arguments, where argument i has type t_i. In this case, the type of the application is t.

  An application that is not well typed results in a checked compile-time error.

C-- primitive operators	
IEEE 754 Floating-point operations	
`i2f`k	$\forall\alpha.$`bits`$\alpha \times$ `bits2` $\rightarrow$ `bits`k Converts a two's-complement integer of α bits into an IEEE 754 floating-point value of k bits, using the rounding mode given as the second argument.
`f2f`k	$\forall\alpha.$`bits`$\alpha \times$ `bits2` $\rightarrow$ `bits`k Converts an IEEE 754 floating-point value of α bits to a floating-point value of k bits, using the rounding mode given as the second argument.
`f2i`k	$\forall\alpha.$`bits`$\alpha \times$ `bits2` $\rightarrow$ `bits`k Converts an IEEE 754 floating-point value of α bits to a two's-complement integer value of k bits, using the rounding mode given as the second argument.
`fabs`	$\forall\alpha.$`bits`$\alpha \rightarrow$ `bits`α Absolute value of a floating-point value.
`fadd`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \times$ `bits2` $\rightarrow$ `bits`α Floating-point add with explicit rounding mode.
`fdiv`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \times$ `bits2` $\rightarrow$ `bits`α Floating-point divide with explicit rounding mode.
`feq`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bool` Floating equality. Compares floating-point numbers and produces a Boolean directly.
`fgt`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bool` Floating greater than. Compares floating-point numbers and produces a Boolean directly.
`fge`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bool` Floating greater than or equal. Compares floating-point numbers and produces a Boolean directly.
`flt`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bool` Floating less than. Compares floating-point numbers and produces a Boolean directly.
`fle`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bool` Floating less than or equal. Compares floating-point numbers and produces a Boolean directly.
`fmul`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \times$ `bits2` $\rightarrow$ `bits`α Floating-point multiply with explicit rounding mode.
`fmulx`	$\forall\alpha,\beta.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bits2`α Floating-point multiply, extended: Multiply two α-bit, floating-point numbers and return the exact 2α-bit product, with no rounding.
`fne`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bool` Floating inequality. Compares floating-point numbers and produces a Boolean directly.
`fneg`	$\forall\alpha.$`bits`$\alpha \rightarrow$ `bits`α Negation of a floating-point value.
`fordered`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bool` Floating ordered predicate. Returns true if and only if two floating-point numbers are comparable.
`fsqrt`	$\forall\alpha.$`bits`$\alpha \times$ `bits2` $\rightarrow$ `bits`α Square root of a floating-point value, with explicit rounding mode.
`fsub`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \times$ `bits2` $\rightarrow$ `bits`α Floating-point subtract with explicit rounding mode.

Table 4: C-- primitive operators

C-- primitive operators, continued	
`funordered`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bool` Floating unordered predicate. Returns true if and only if two floating-point numbers are not comparable.
`minf`k	$\forall\alpha.$`bits`k IEEE 754 representation of minus infinity.
`mzero`k	$\forall\alpha.$`bits`k IEEE 754 representation of minus zero.
`NaN`k	$\forall\alpha.$`bits`$\alpha \rightarrow$ `bits`k `NaN n` is an IEEE 754 NaN with significand `n`. For a 32-bit NaN, the significand has $\alpha = 23$ bits. For a 64-bit NaN, the significand has $\alpha = 52$ bits. It is an *unchecked* run-time error to pass a zero significand to `%NaN`.
`pinf`k	$\forall\alpha.$`bits`k IEEE 754 representation of plus infinity.
`pzero`k	$\forall\alpha.$`bits`k IEEE 754 representation of plus zero.
`round_down`	`bits2` IEEE 754 rounding mode for rounding down.
`round_up`	`bits2` IEEE 754 rounding mode for rounding up.
`round_nearest`	`bits2` IEEE 754 rounding mode for rounding to nearest.
`round_zero`	`bits2` IEEE 754 rounding mode for rounding toward zero.

Other operations, alphabetically by name	
`add`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bits`α Two's-complement integer addition.
`addc`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \times$ `bits1` $\rightarrow$ `bits`α Two's-complement integer addition with carry in.
`add_overflows`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bool` Tells whether a signed integer addition would overflow, i.e., if an addition would produce a result with a sign different from the sign of both arguments.
`and`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bits`α Bitwise and.
`bit`	`bool` $\rightarrow$ `bits1` Convert boolean value to a 1-bit value.
`bool`	`bits1` $\rightarrow$ `bool` Convert a 1-bit value to a Boolean.
`borrow`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \times$ `bits1` $\rightarrow$ `bits1` The bit `borrow`(x, y, b_i) is the "borrow bit" needed when computing the subtraction $x - y - b_i$, where x and y are two's-complement α-bit integers, and b_i is the "borrow bit".
`carry`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \times$ `bits1` $\rightarrow$ `bits1` The bit `carry`(x, y, c_i) is the "carry out" bit produced by adding $x + y + c_i$ where x and y are two's-complement integers, and c_i is the "carry in" bit.
`com`	$\forall\alpha.$`bits`$\alpha \rightarrow$ `bits`α Bitwise complement of value.
`conjoin`	`bool` $\times$ `bool` $\rightarrow$ `bool` Boolean conjunction. x `` `conjoin` `` $y \equiv$ `if` x `then` y `else false`.

Table 4: C-- primitive operators

C-- primitive operators, continued	
`disjoin`	`bool` $\times$ `bool` $\rightarrow$ `bool` Boolean disjunction. x ‘`disjoin`‘ $y \equiv$ `if` x `then true else` y.
`div`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bits`α Two's-complement signed integer division, rounding towards minus infinity. It is an *unchecked* run-time error to divide by zero.
`div_overflows`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bool` Tells whether signed integer division would overflow, that is, if it would divide the most negative integer by -1 to produce a positive result too large to be represented in two's-complement notation. It is an *unchecked* run-time error to ask whether division by zero would overflow.
`divu`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bits`α Unsigned integer division, rounding down (towards both zero and minus infinity). It is an *unchecked* run-time error to divide by zero.
`eq`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bool` Equality. True, if values are bitwise equal.
`false`	`bool` Boolean falsehood.
`ge`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bool` Greater than or equal. Compares two's-complement signed integers.
`geu`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bool` Greater than or equal, unsigned. Compares two unsigned integers.
`gt`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bool` Greater than. Compares two's-complement signed integers.
`gtu`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bool` Greater than, unsigned. Compares two unsigned integers.
`le`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bool` Less or equal. Compares two's-complement signed integers.
`leu`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bool` Less or equal, unsigned. Compares two unsigned integers.
`lobits`k	$\forall\alpha.$`bits`$\alpha \rightarrow$ `bits`k The k least significant bits of the argument.
`lt`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bool` Less than. Compares two's-complement signed integers.
`ltu`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bool` Less than, unsigned. Compares two unsigned integers.
`mod`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bits`α Signed modulus, satisfying x ‘`mod`‘ $y = x - y$ ‘`mul`‘ (x ‘`div`‘ y) for all $y \neq 0$. It is an *unchecked* run-time error to take the modulus when dividing by zero.
`modu`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bits`α Unsigned modulus, satisfying x ‘`modu`‘ $y = x - y$ ‘`mul`‘ (x ‘`divu`‘ y) for all $y \neq 0$. It is an *unchecked* run-time error to take the modulus when dividing by zero.
`mul`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bits`α Multiply two α-bit integers and return the least significant α bits of the (signed) product. The most significant α bits of the product are silently discarded. Notice that the answer is the same whether a signed or an unsigned multiply is used, so there is only one variant. (N.B. It may still be worth providing two variants in order to take better advantage of instructions that detect overflow.)

Table 4: C-- primitive operators

C-- primitive operators, continued	
`mul_overflows`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bool` Tells whether an extended, signed multiply would require the high word, i.e., whether the most significant α bits of the product would differ from the sign bit of the least significant α bits.
`mulu_overflows`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bool` Tells whether an extended, unsigned multiply would require the high word, i.e., whether the most significant α bits of the product would differ from zero.
`mulux`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bits2`α Multiply, unsigned, extended: Multiply two α-bit, unsigned integers and return the exact 2α-bit (unsigned) product.
`mulx`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bits2`α Multiply, extended: Multiply two α-bit, two's-complement, signed integers and return the exact 2α-bit (signed) product.
`ne`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bool` Inequality. Compares two's-complement signed integers.
`neg`	$\forall\alpha.$`bits`$\alpha \rightarrow$ `bits`α Two's-complement negation.
`not`	`bool` $\rightarrow$ `bool` Logical complement
`or`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bits`α Bitwise or.
`parity`	$\forall\alpha.$`bits`$\alpha \rightarrow$ `bits1` Computes the number of 1 bits in its argument, modulo 2.
`popcnt`	$\forall\alpha.$`bits`$\alpha \rightarrow$ `bits`α Returns the number of one bits in its argument.
`quot`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bits`α Two's-complement, signed integer division, rounding towards zero. It is an *unchecked* run-time error to divide by zero.
`quot_overflows`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bool` Tells whether signed integer quotient would overflow, that is, if it would divide the most negative integer by -1 to produce a positive result too large to be represented in two's-complement notation. It is an *unchecked* run-time error to ask whether quotient by zero would overflow.
`rem`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bits`α Remainder, satisfying x ‘`rem`‘ $y = x - y$ ‘`mul`‘ (x ‘`quot`‘ y) for all $y \neq 0$. It is an *unchecked* run-time error to take the remainder when dividing by zero.
`rotl`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bits`α Rotate left.
`rotr`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bits`α Rotate right.
`shl`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bits`α Shift left. It is an *unchecked* run-time error for the shift amount to be as large as α. If in doubt, shift by $n \bmod \alpha$.
`shra`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bits`α Shift right, arithmetic. It is an *unchecked* run-time error for the shift amount to be as large as α. If in doubt, shift by $n \bmod \alpha$.
`shrl`	$\forall\alpha.$`bits`$\alpha \times$ `bits`$\alpha \rightarrow$ `bits`α Shift right, logical. It is an *unchecked* run-time error for the shift amount to be as large as α. If in doubt, shift by $n \bmod \alpha$.

Table 4: C-- primitive operators

C-- primitive operators, continued	
`sub`	$\forall\alpha.\texttt{bits}\alpha \times \texttt{bits}\alpha \rightarrow \texttt{bits}\alpha$ Two's-complement integer subtraction.
`subb`	$\forall\alpha.\texttt{bits}\alpha \times \texttt{bits}\alpha \times \texttt{bits1} \rightarrow \texttt{bits}\alpha$ Two's-complement integer subtraction with borrow in.
`sub_overflows`	$\forall\alpha.\texttt{bits}\alpha \times \texttt{bits}\alpha \rightarrow \texttt{bool}$ Tells whether a signed integer subtraction would overflow, i.e., if the two arguments differ in sign and the result would have the sign of the right-hand argument.
`sx`k	$\forall\alpha.\texttt{bits}\alpha \rightarrow \texttt{bits}k$ Sign-extend an α-bit, two's-complement, signed integer to produce a k-bit, two's-complement, signed integer, where $k > \alpha$.
`true`	`bool` Boolean truth.
`xor`	$\forall\alpha.\texttt{bits}\alpha \times \texttt{bits}\alpha \rightarrow \texttt{bits}\alpha$ Bitwise exclusive or.
`zx`k	$\forall\alpha.\texttt{bits}\alpha \rightarrow \texttt{bits}k$ Zero-extend an α-bit value to produce a k-bit value, where $k > \alpha$. Argument and result represent the same value when interpreted as unsigned integers.

Table 4: C-- primitive operators

8 The C-- run-time interface

A typical program created using C-- includes not only compiler-generated C-- code but also a hand-written run-time system (the *front-end runtime*). Although it may be helpful to write small parts of the front-end run-time in C--, the bulk of the front-end runtime is normally written in a high-level language, such as C.

As a program executes, control may be transferred back and forth between C-- code and the front-end runtime. The *C-- run-time interface* gives the front-end runtime the ability to inspect and modify the state of a suspended C-- computation. The interface can also be used to create a new C-- computation and to transfer control to it.

8.1 The C-- run-time model

This section presents the model of computation that is supported by the C-- run-time system.

8.1.1 Activations, stacks, and continuations

A C-- computation uses a stack of *activations* to hold its state. An *activation*, which is created when a procedure is called, holds the values of that procedure's local variables (including parameters) and stack data, as well as any temporary values that are used during the evaluation of expressions. When the procedure returns or makes a tail call, the activation is destroyed. Activations may also be destroyed by a `cut to`, as discussed below.

Activations are stored on a *stack*, which grows from *older* to *younger* activations. (In order to avoid confusion about the direction of stack growth, we use the term "youngest end" rather than "top" of the stack.) A C-- stack may reside on the system stack or in memory allocated by the front-end run-time system.

A stack can include activations of foreign procedures as well as of procedures compiled with C--. When an activation of a foreign procedure is visited during a stack walk, it looks like an activation of a C-- procedure with no spans, no formal parameters, no local variables, and no `stackdata` labels.

Depending on the specification of the foreign calling convention, it may or may not be possible to use the run-time interface to recover values of callers' variables. The problem is that although all foreign conventions specify *what* registers are saved, not all conventions specify *where* they are saved. Each implementation of C-- must advertise what capabilities it supports for each foreign convention. Implementations of C-- are encouraged to provide a "paranoid" version of each foreign calling convention, to be used when it is necessary to guarantee access to local variables through the run-time interface.

The state of a C-- stack can be encapsulated as a *continuation* (§6.7). Such a continuation, when presented to the run-time interface, enables a client to inspect and modify the stack.

A C-- continuation is not a thread, and C-- does not implement a thread scheduler. Instead it offers hooks that enable the front-end runtime to implement a scheduler.

8.1.2 Transferring control between the front-end run-time system and C--

Control is transferred between the front-end run-time system and C-- code using ordinary calls and returns.

- The front-end run-time system may call a C-- procedure directly, provided the procedure is defined to use the `foreign "C"` calling convention (§5.4).
- The front-end run-time system may return to a C-- procedure, which must have called it with with a `foreign "C"` call.
- A C-- procedure may call the front-end run-time system with a `foreign "C"` call.

- A C-- procedure may return with a `foreign "C" return`, provided it was originally called by a front-end C procedure. (Any number of tail calls may intervene between the C call and the C return. For example, it is possible for the front-end run-time system to make a `foreign` call to C-- procedure `f`, which makes a tail call to `g`, which returns with a `foreign "C"` return.)

8.1.3 Transferring control between C-- stacks

A C-- procedure running on one stack may transfer control to a C-- procedure running on another stack by executing `cut to k`, where `k` is a continuation. This mechanism can be used to coroutine between C-- stacks.

When a C-- procedure executes `cut to k`, the continuation `k` refers to a particular activiation on a particular stack. The `cut to k` destroys all activations younger than `k` on the destination stack. Therefore, if the continuation `k` points into the currently active stack, the activations from the current activation to (but not including) `k` are destroyed. This usage is more like raising an exception than like a coroutine.

If a C-- procedure wants to transfer control to the front-end run-time system on a different stack, it must do so in two steps: first change stacks using `cut to`, then get to the front-end runtime using a `foreign` call or return.

8.1.4 Walking a stack

A C-- continuation provides not only a target for `cut to` but also a representation of its stack. Such a continuation can be presented to the C-- run-time system and used to walk the stack. A front-end runtime can visit each activation on a stack, find and modify local variables of each activation, and even create a new continuation that (when `cut to`) will resume execution at a given activation.

8.2 Overview and numbering conventions

Table 5 presents an overview of the C-- run-time interface. There are a couple of surprises in the exported types: Although C-- does not distinguish a code pointer from a data pointer, the C interface must do so lest the C compiler complain. And although the C programming language forces the interface to expose the representation of `struct cmm_activation`, *it is an unchecked error for a front-end run-time system to refer to any field of this structure.*

8.2.1 Numbering

The C-- runtime interface uses ordinal numbers to refer to parameters, variables, `stackdata` labels, and continuations. The following rules apply for numbering:

- Global variables are numbered from 0 in the order in which they appear in the source code.
- Formal parameters and local variables of a single procedure are numbered together, from 0, in the order in which they appear in the source code. The numbering sequence is separate for each C-- procedure.

  ⟨*toplevel example*⟩+≡
  ```
  fn (bits32 x /* 0 */, bits32 y /* 1 */) {
      bits32 m /* 2 */, n /* 3 */;
      return (y,x);
  }
  ```

Types:

`Cmm_CodePtr`	Native pointer (to code)
`Cmm_DataPtr`	Native pointer (to data)
`Cmm_Activation` ≡ `struct cmm_activation`	An opaque structure representing an activation of a C-- procedure.
`Cmm_Cont` ≡ `struct cmm_cont`	An opaque structure representing a C-- continuation.

Stack-walking and -unwinding functions:

```
Cmm_Activation Cmm_YoungestActivation (const Cmm_Cont *k);
int            Cmm_isOldestActivation (const Cmm_Activation *a);
Cmm_Activation Cmm_NextActivation     (const Cmm_Activation *a);
int            Cmm_ChangeActivation   (Cmm_Activation *a);
Cmm_Cont*      Cmm_MakeUnwindCont     (const Cmm_Activation *a, unsigned n, ...);
void           Cmm_CutTo              (const Cmm_Cont *k);
```

Span-lookup function:

```
Cmm_Dataptr Cmm_GetDescriptor(const Cmm_Activation *a, Cmm_Word token);
```

Access to global variables, local variables, and stack labels:

```
unsigned Cmm_LocalVarCount    (const Cmm_Activation *a);
void*    Cmm_FindLocalVar     (const Cmm_Activation *a, unsigned n);
void     Cmm_LocalVarWritten  (const Cmm_Activation *a, unsigned n);
void*    Cmm_FindDeadLocalVar (const Cmm_Activation *a, unsigned n);
void*    Cmm_FindStackLabel   (const Cmm_Activation *a, unsigned n);
```

Table 5: Overview of the run-time interface

- Labels appearing in `stackdata` in a single procedure are numbered from 0 in the order in which they appear in the source code. The numbering sequence is separate for each C-- procedure.
- At a call site, continuations in `also unwinds to` are numbered from 0 in the order in which they appear in the source code. If a continuation appears more than once in `also unwinds to` lists at a call site, that continuation has more than one number. The numbering for each call site may be different.[4]

  ⟨*unimplemented toplevel example*⟩≡
  ```
  fc () {
      f() also unwinds to c1, c0, c2;
      return (99);

      continuation c0 (): return (11);

      continuation c1 (): return (22);

      continuation c2 (): return (33);
  }
  ```

  When control is suspended at the call to `f`, continuation `c1` has number 0 for the `MakeUnwindCont` runtime function.

[4]This numbering enables the C-- run-time system to check the index's correctness using a single compare instruction. The alternative, to use one numbering for the procedure, precludes some front-end tricks for sharing continuation tables across call sites.

8.3 Creating a new stack

A future version of this specification will show how to run a C-- computation on a stack allocated by the front-end run-time system. Until the future arrives, implementors are encouraged to experiment with ways of supporting this feature.

8.4 Walking a stack and inspecting its contents

These functions allow the front-end runtime to walk a stack. The starting point is always a C-- continuation, which encapsulates the state of a stack.

An *activation handle*, a C value of type `*Cmm_Activation`, encapsulates a single activation. The information so encapsulated includes a pointer into a C-- stack, the program counter to which control will return, and the values of the callee-saves registers. (The availability of callee-saves registers is what distinguishes an activation handle from a continuation.)

To get a handle for the youngest activation on a stack, the front end calls `Cmm_YoungestActivation`, passing a continuation. To get older activations, the front end calls `Cmm_NextActivation` or `Cmm_ChangeActivation`. Given an activation, the front end can get to its local variables using `CmmFindLocalVar`.

`Cmm_Activation Cmm_YoungestActivation(const Cmm_Cont *k)` Returns an activation handle that encapsulates the activation from which continuation `k` came. `Cmm_YoungestActivation` returns a structure, not a pointer, so that the C compiler can arrange to allocate memory for the structure in the caller's frame. As noted above, it is an *unchecked* run-time error for the front-end runtime to mess with the internals of this structure.

`int Cmm_IsOldestActivation(const Cmm_Activation *a)` Returns nonzero if and only if `*a` is the oldest activation on its stack; that is, iff `*a` has no caller.

`Cmm_Activation Cmm_NextActivation(const Cmm_Activation *a)` Returns the activation to which `a` will return. A checked run-time error occurs unless `!Cmm_IsOldestActivation(a)`.

`int Cmm_ChangeActivation(Cmm_Activation *a)` Combines testing and walking in one call:

```
rc = Cmm_ChangeActivation(&a);
```

is equivalent to

```
if (Cmm_IsOldestActivation(&a))
  rc = 0;
else {
  a = Cmm_NextActivation(&a);
  rc = 1;
}
```

but it may be faster.

`Cmm_Cont *Cmm_MakeUnwindCont(Cmm_Activation *a, unsigned n, ...)` returns a *parameterless* C-- continuation that, when `cut to`, passes `Cmm_MakeUnwindCont`'s parameters to an `also unwinds to` continuation of the call site at which the activation `a` is suspended. The parameter `n` identifies the call-site continuation by indexing the `also unwinds to` lists. The remaining parameters are the values to be passed to that continuation. It is a checked run-time error for `n` to be out of range. It is an *unchecked* run-time error for the number or the C types of the remaining parameters to be inconsistent with the number, C-- types, and kinds of the formal parameters of the continuation. (As usual, it is up to each implementation of C-- to advertise what size and kind correspond to each C type.)

Calling `Cmm_MakeUnwindCont(a, n, arg1, arg2, ...)` may write into `*a`, and it kills every activation that is younger than `a` and shares a stack with `a`. After this call, therefore, it is an *unchecked* run-time error to do any of the following:

- To `cut to` the continuation returned by `Cmm_MakeUnwindCont` after the memory associated with `*a` has been reused
- To pass `a` to any procedure in this interface
- To read or write any local variable of activation `a` or of any activation that is younger than `a` and shares a stack with `a`
- To `cut to` any other continuation in activation `a` or in any activation that is younger than `a` and shares a stack with `a`
- To `return` to activation `a` or to any activation that is younger than `a` and shares a stack with `a`

N.B. It is an *unchecked* run-time error to kill an activation that is suspended at a call site *unless* that call site is annotated with `also aborts`.

The intent of the restrictions above is that once you have called `Cmm_MakeUnwindCont`, you can either `cut to` the continuation right away, or provided you preserve `*a`, you can save the continuation in a data structure and `cut to` it later. A common application is to use `Cmm_MakeUnwindCont` to trim the stack during exception handling.

It is not possible to use `Cmm_MakeUnwindCont` to unwind a stack to the *normal* return continuation. If this outcome is desired, the front end must instead explicitly create an `also unwinds to` continuation that goes to the same point as the normal return continuation.

`void Cmm_CutTo(Cmm_Cont *k)` Cuts to a parameterless continuation. Useful for applying to a continuation returned by `Cmm_MakeUnwindCont`.

`void *Cmm_FindLocalVar(const Cmm_Activation *a, unsigned n)` where `n` is an index into the formal parameters and local variables of activation `a`, using the numbering conventions of §8.2.1.

- If `n` refers to a live variable, `FindLocalVar` returns a pointer to a location containing the value of that variable. The pointer can be used to read and change the value of the variable. It is an *unchecked* run-time error to change the value of a variable that has been declared `invariant`.
- If `n` refers to a dead variable, `FindLocalVar` returns `NULL`.
- It is a checked run-time error for `n` to be out of range.

Because `Cmm_FindLocalVar` returns a pointer to the location containing the variable, the front-end runtime can use this pointer to modify the variable (provided it is not annotated `invariant`). But if it does so, it must afterward announce the modification to C-- by calling `Cmm_LocalVarWritten`. To modify a local variable without making this call is a checked run-time error.

`void Cmm_LocalVarWritten (const Cmm_Activation *a, unsigned n)` announces to C-- that the front-end run-time system has changed the value of the local variable numbered `n`. For the front-end run-time system to change the value of a local variable *without* announcing the change is an *unchecked* run-time error. The announcement must take place before the next `Cmm_NextActivation`, `Cmm_ChangeActivation`, `Cmm_MakeUnwindCont`, or `cut to`.

`unsigned Cmm_LocalVarCount(const Cmm_Activation *a)` returns the number of formal parameters and local variables of activation `a`.

`void *Cmm_FindDeadLocalVar(const Cmm_Activation *a, unsigned n)` `n` is an index into the formal parameters and local variables of activation `a`. If `n` refers to a dead variable, `FindLocalVar` returns a pointer to a location containing the last known value of that variable, or `NULL` if no such location exists. Otherwise, `FindDeadLocalVar` returns `Cmm_FindLocalVar(a, n)`. It is an *unchecked* run-time error to write through a pointer returned by `FindDeadLocalVar`. This function might be used to support a debugger.

`void *Cmm_FindStackLabel(const Cmm_Activation *a, unsigned n)` where `n` is the index of one of the stack labels of the activation. `FindStackLabel` returns the (immutable) value of that label, namely a pointer into the stack-allocated data area. It is a checked run-time error for `n` to be out of range.

`Cmm_Dataptr Cmm_GetDescriptor(const Cmm_Activation *a, Cmm_Word key)` Returns the value associated with the smallest span tagged with `key` and containing the program point where the activation `a` is suspended. Returns `(Cmm_Dataptr) 0` if there is no such span.

8.5 The global registers

A running C-- computation may use *global registers* (§4.4.1). Conceptually, these global variables live in fixed machine registers. They are left undisturbed by all control transfers to C-- code (call, return, `cut to`, etc).

If, however, a C-- program makes a `foreign` call to non-C-- code, such as the front-end run-time system, the global-register variables are saved in a place from which they are no longer accessible by either C-- code or the run-time system. If a C-- computation wishes the the global registers to be made accessible, it must explicitly save them before a foreign call and restore them afterward.

9 Frequently asked questions

1. *Can the* `import` *import directive be used to import a variable declared in another compilation unit?*

   No. Only labels and procedures can be imported. If you want to share a C-- global register variable across compilation units, that variable needs to be declared in every compilation unit. Once declared, C-- global register variables are automatically shared across compilation units. N.B. Every compilation unit must contain *identical* global-variable declarations.

2. *I want something like a C global variable, but I don't want to burn a register. What do I do?*

   Here's an example that shows how one might translate a C global variable (of `struct` type) into C--.

   The C source code containing the definition of the global variable `pt`

   ```
    struct point { int x; int y; } pt;
   ```

   becomes the C-- source code

   ```
   section "data" {
     pt: align 4; bits32[2];
     export pt;
   }
   ```

   In another compilation unit, the C source code containing the external reference

   ```
   extern struct point { int x; int y; } pt;
   ```

   becomes

   ```
   import pt;
   ```

   And in either compilation unit, the C source code

   ```
   pt.x = pt.y + 1;
   ```

   becomes the C– source code

   ```
   bits32[pt] = bits32[pt+4] + 1;
   ```

3. *How do I program floating-point computations with the hardware rounding modes?*

   Here is one example:

   ```
   target byteorder little;

   bits2 rm = "IEEE 754 rounding mode";

   p() { return ("float" %i2f32(3, rm)); }

   foreign "C" main(bits32 argc, "address" bits32 argv) {
     bits32 x;
     "float" x = p();

     foreign "C" printf("address" answer, "float" %f2f64(x, rm));
     foreign "C" return(0);
   }
   ```

```
section "data" {
  answer: bits8[] "Integer 3 converts to floating-point %4.2lf\n\0";
}

import printf; export main;
```

4. *How do I build my own jump tables?*

   Ordinary code labels are visible throughout the compilation unit in which they appear. So label each jump target, put a table in initialized data (in a data section *outside* your procedure), and put in a computed `goto` labeled with a `targets` directive, e.g.,

```
section "data" { align 4; jump_tab: bits32[] { L1, L2, L3 }; }
f (bits32 x) {
  goto bits32[jump_tab + 4*x] targets L1, L2, L3;
  L1: return (1);
  L2: return (2);
  L3: return (3);
}
```

10 Common mistakes

- Getting a floating-point result from C without a `"float"` kind is a mistake.
- Passing a 32-bit floating-point value to C's `printf` function is a mistake. The rules of C require that when there is no prototype, a `float` be promoted to `double`, so for example the correct way to print a 32-bit floating-point value `x` is

  ```
  bits32 x;
  foreign "C" printf("address" str, "float" %f2f64(x, System.rounding_mode));
  ```

11 Potential extensions

This section describes potential extensions, which may become part of C-- version 2.1.

11.1 Run-time information about C-- variables

The current version of C-- provides a front end all the capabilities it needs to manipulate variables. Arbitrary information about a variable can be coded as initialized data, and this information can be associated with the variable using a span which (again by coded initialized data) relates the information to the variable's number. But although C-- is in this sense complete, experience is showing us that it is not always convenient. In particular, we have observed two problems:

- To use C-- today, a front end must contain a fair amount of mechanism, e.g., to hold initialized data for emission at the end of a procedure, to keep track of the numbers of interesting variables, and so on. This sort of mechanism is no problem for a large, existing compiler, which probably already contains something suitable. But the mechanism is a nuisance for a small, simple compiler, such as a student's compiler.
- The interface is not tuned to make any particular common case easy. For example, one common case is to identify some set of variables as pointers and to iterate over these variables, e.g., for garbage collection. For this common case, there are two obvious things a front end can do:
  - Accumulate a list of the numbers of interesting variables, code that list as initialized data, point to the list with a span, and iterate over the list at run time.
  - Associate a bit with each variable, make the bit zero for uninteresting variables and one for interesting ones, code the bit vector as initialized data, point to the bit vector with a span, iterate over all variable numbers at run time, and test the bits.

We are thinking of extending C-- with several mechanisms that will help ease the burden of targeting C--, especially for simple front ends.

Indirect initialized data We propose that a C-- program be able to emit initialized data from anywhere in the program that an expression is expected. The proposed syntax would be to add the following production to the grammar:

expr ⇒ `indirect` *string* `{` $\{$*datum*$\}$ `}`

This is syntactic sugar for allocating $\{$*datum*$\}$ as initialized data in the section named by the *string*. The value of the expression is a pointer to the initialized data, after initial alignment (if any).

For example, the phrase

```
foreign "C" printf(indirect "rodata" { bits8[] "hello, world\n\0"; });
```

would be internally rewritten to

```
foreign "C" printf(L72);
...

section "rodata" { L72: bits8[] "hello, world\n\0"; }
```

where `L72` is an internally generated name.

Similarly, the phrase

```
span 1 (indirect "rodata" {
  align 4;
  bits32[] { 3, 10, 30, 99 };
}) { ... }
```

would be rewritten to

```
span 1 L73 { ... }
  ...
section "rodata" {
  align 4;
  L73:
  bits32[] { 3, 10, 30, 99 };
}
```

where `L73` is an internally generated name.

If two `indirect` expressions have the same context and are allocated in a read-only section, perhaps they could share space.

Variable metadata When a front end wants to associate metadata with a variable, it must put the metadata in an array indexed by variable number, then point to the array with a span. For a simple front end, it would be easier simply to include metadata when a variable is declared. Our proposal is

- Every local C-- variable is associated with one word of metadata, specified when the variable is declared (and defaulting to 0). Like a span value, a metadata value must be a link-time constant expression.
- In the run-time interface, a variable's metadata is just like its location: if the variable is live, its metadata can be found with `Cmm_FindMetadata`.

The proposed syntax would be to replace the existing grammar's definition of *registers* with the following definition:

registers ⇒ [*kind*] *type name register-modifier* {, *name register-modifier*} [,]

register-modifier ⇒ [= *string* | `metadata` *expr*]

The proposed run-time interface would be

```
void* Cmm_FindMetadata(const Cmm_Activation *a, unsigned n);
```

If a variable number `n` is live in activation `a`, `Cmm_FindMetadata(a, n)` returns that variable's metadata; otherwise, it returns 0.

For example, suppose one wishes a variable's metadata to be a pointer to a two-word block of memory. The first word tells if the metadata is a pointer; the second word gives the source-language name of the variable. One could write

```
bits32 i@437
  metadata indirect "rodata" {
    align 4;
    bits32[] { 1, indirect "rodata" { bits8[] "deads_list\0"; } };
  };
```

To print the name of a variable at run time, one could write

```
struct metadata {
  int is_pointer;
  const char *source_name;
}
...
struct metadata *m;
m = Cmm_FindMetadata(a, n);
if (m) printf("var name is %s\n", m->source_name);
...
```

Iteration groups A common pattern in run-time systems is to iterate over a group of variables that are (within the group) considered indistinguishable. For example,

- A garbage collector may wish to iterate over all pointer variables while ignoring non-pointer variables.
- A diagnostic stack walker may with to iterate over all C-- variables that represent source-language variables (e.g., to print their names and values) while ignoring C-- variables that represent temporaries introduced by the front-end compiler.

As noted above, the present version of C-- requires the front end either to code a list of the relevant variable numbers or somehow to associate group membership with each variable.

A possible alternative would be to tag each variable with a list of iteration groups to which it could belong. A possible syntax would be to add

register-modifier ⇒ `iteration` *expr*

as a new *register-modifier*, with the understanding that *expr* must be a compile-time constant expression that evaluates to a small, nonnegative integer.

Iteration groups would be supported by the following additions to the run-time interface:

```
typedef void (*Cmm_iterator)(void *location, void *metadata, void *closure);
void Cmm_Iterate(const Cmm_Activation *a, unsigned iteration_number, Cmm_iterator iterate, void *closure);
```

A call to `Cmm_Iterate(a, n, iter, cl)` has the effect of the following loop:

```
for each live variable v in iteration group n do
    let i be v's number
    iter(Cmm_FindLocalVar(i), Cmm_FindMetadata(i), cl);
```

An iterator visits only *live* variables.

Variations on the numbering of variables At present, every local variable in a C-- procedure is numbered by the C-- compiler. At run time, the number can be passed to `Cmm_FindLocalVar` (and under the new proposal, `Cmm_FindMetadata`). Several users have objected to this interface on the grounds “why should I spend space on stack maps for variables that I know I am never going to care about at run time?” An implementation of C-- seems to have two alternatives:

- Use space proportional to the total number of variables. `Cmm_FindLocalVar` and `Cmm_FindMetadata` require a range check and an array lookup, taking constant time.
- Use space proportional to the number of *live* variables. `Cmm_FindLocalVar` and `Cmm_FindMetadata` require a range check and a binary search, taking time logarithmic in the number of live variables.

We are considering several alternatives:

- The front end could label some variables as “unnumbered.” A variable that is unnumbered and is not part of any iteration group would in effect be completely invisible to the run-time interface, so no resources would be consumed storing information about that variable.
- Instead of letting C-- number the variables, the front end could number them explicitly. Variables not explicitly numbered would be unnumbered. (Because this convention would not be backwards compatible, we are a bit leery of this idea.)
- Some combination of the above. For example, the front end could explicitly number some variables and label others as unnumbered. Variables *not* annotated by the front end would continue to be numbered implicitly.

Global variables Knotty questions remain about how best to save and restore global variables, especially when multiple C-- stacks are active. A simple solution is to have front ends generate code to save and restore globals explicitly, but if C-- has to allocate some globals into memory, there may be a more efficient strategy.

11.2 Running C-- threads

In a single-threaded environment, C-- code runs on the system stack, just like C code. In a multi-threaded environment, however, it may be necessary to multiplex multiple user-level threads (possibly using OS threads into the bargain). To multiplex multiple threads requires run-time support to create a new stack and compile-time support to check for stack overflow.

11.2.1 Creating a C-- stack and running a thread on it

The front-end run-time system can create a C-- stack using `Cmm_CreateStack`. To transfer control to the new stack, the front end uses `cut to`. The interface includes a *limit cookie*, which is to be used to implement a stack-overflow check.

```
typedef struct cmm_stack_limit Cmm_Limit;
Cmm_Cont *Cmm_CreateStack(Cmm_CodePtr f, Cmm_DataPtr x, void *stack, unsigned n, Cmm_Limit **limitp);
```

Function `Cmm_CreateStack` returns a C-- continuation that, when `cut to`, executes the C-- call `f(x)` on the stack `stack`.

- The parameter `f` must be the address of a C-- procedure that takes exactly one argument, which is a value of the native pointer type. To pass any other `f` to `Cmm_CreateStack` is an *unchecked* run-time error.

  It is a checked *run-time error for the procedure addressed by* `f` *to return*—this procedure should instead finish execution with a `cut to`.
- When queried using the C-- run-time interface, a continuation returned by `Cmm_CreateStack` looks like a stack with one activation. That activation makes the two parameters `f` and `x` visible through `Cmm_FindLocalVar`; these parameters can be changed using the run-time interface (for example, if a garbage collection intervenes between the time the continuation is created and it is used).
- When a continuation returned by `Cmm_CreateStack` is `cut to`, it is as if the stack makes a tail call `jump f(x)`. In particular, the activation of `f` now appears as the oldest activation on the stack. As noted, it is a checked run-time error for this activation to return.
- The parameter `stack` is the address of the stack, which is `n` bytes in size. After calling `Cmm_CreateStack`, the stack belongs to C--, so it is an *unchecked* run-time error for the front end to read or write any part of this stack except through `stackdata` areas in active procedures (or through pointers provided by the C-- run-time interface).
- The parameter `limitp` is a pointer through which the C-- run-time system can write a *stack-limit cookie*. This limit cookie is used in overflow checking as described below.

To implement threads, a front end will typically allocate a large thread-control block to represent a thread, and the C-- stack will be part of this block. The rest of the block may contain a C-- continuation for the thread, the stack-limit cookie, thread-local data, the priority of the thread, links to other threads, and so on. All of this information is outside the purview of C--, however.

11.2.2 Stack overflow checking and handling

C-- code running on a finite stack is vulnerable to stack overflow. To protect C-- code from stack overflow, the C-- programmer must insert an explicit stack-limit check in every C-- procedure.[5] The details of the

[5]We would like to find a way to enable a front end to amortize a single stack-limit check over multiple procedures.

stack-limit check are not yet specified, but it might take the following form:

`limitcheck` *limit* `fails to` *k*

where `limitcheck` is a keyword, and *limit* is a C-- expression that evaluates to the stack-limit cookie for the stack on which the procedure is executing. If there is not enough room on the stack, C-- cuts to continuation *k*, passing the limit cookie and a continuation that can be used to move the computation to a larger stack, as described below. Typically *k* will be a stack-overflow handler set up by the front-end run-time system and running on a stack of its own.

(The C-- client gets to decide whether to hold the limit cookie in a global register or somewhere else. It is the client's responsibility to make sure that the `limit` expression is valid. For example, if the limit cookie is held in a global register, and `cut to` transfers control to a different C-- thread, the client must arrange to change the value of the register.)

To move a C-- thread to a larger stack,[6] we imagine additions to the run-time interface something like the following:

```
typedef struct cmm_reloc Cmm_Reloc;
Cmm_Cont *Cmm_MoveStack(Cmm_Cont *k, Cmm_Limit *limit, void *newstack, unsigned n,
                        Cmm_Limit **limitp, Cmm_Reloc **relocp);
```

The `MoveStack` function takes a continuation `k` and limit cookie `limit`. It moves the computation to a new stack. It returns the analogous continuation on the new stack, and it also writes a new stack-limit cookie through been pointer `limitp`. Finally, it writes *relocation information* through pointer `relocp`. Relocation information is used to move two kinds of items:

- Pointers to user-allocated `stackdata` on the old stack
- C-- continuations that refer to activations on the old stack[7]

`MoveStack` is not limited to cases of stack overflow; any stack can be moved whenever no computation is running on that stack.

Because a run-time system might conceivably move multiple stacks at once (e.g., at a garbage collection), relocation information is used in arrays. An array is represented by a pointer to `Cmm_Reloc*`, together with a count.

```
void      Cmm_SortRelocation   (Cmm_Reloc *relocs[], unsigned reloc_count);

Cmm_Cont *Cmm_RelocateCont     (Cmm_Cont *k, Cmm_Reloc *relocs[], unsigned reloc_count);
void     *Cmm_RelocateStackdata(void     *p, Cmm_Reloc *relocs[], unsigned reloc_count);
```

An array must be sorted before being passed to `RelocateCont` or `RelocateStackdata`.

It is up to the front-end run-time system to call `RelocateCont` or `RelocateStackdata` on any continuation or pointer that might refer to a stack that has moved. Using such a continuation or pointer without relocating it is an *unchecked* run-time error. N.B. It is safe simply to pass *every* continuation and stack pointer to these relocation functions; a continuation or pointer that does not refer to a moved stack will not be affected.

Example

```
typedef enum { NEW, RUNNING, SLEEPING, DEAD } state;

struct tcb {
  Cmm_Cont *k;
```

[6]We would like at some point to support segmented stacks with underflow detection, but for the next revision, that isn't in the cards.

[7]A potential alternative is to require that `MoveStack` do a stack walk to fix up every continuation stored in the stack, but that doesn't help with pointers to continuations on the old stack, and it might do unnecessary work (both in walking and in fixing up continuations that will not be used).

```
  state state;
  Cmm_Limit *limit;
  void *stack;
  unsigned size;
};

struct tcb *new_thread(unsigned n) {
  struct tcb *tcb = malloc(n + sizeof(*tcb));
  assert(tcb);
  tcb->state = new;
  tcb->stack = tcb+1;
  tcb->size  = n;
  tcb->k = Cmm_CreateStack(run_thread, tcb, tcb->stack, n, &tcb->limit);
  return tcb;
}
```