The C-- Language Reference Manual

Simon Peyton Jones   Thomas Nordin   Dino Oliva   Pablo Nogueira Iglesias

April 23, 1998

Contents

1 Introduction

`C--` is a portable assembly language designed to be a good backend for high level languages (particularly for those that make use of garbage-collection) and to run fast on a number of todays major computer architectures. It is also designed to have as few dependencies as possible on the underlying hardware, but speed and ease of use has sometimes taken precedence over orthogonality and minimality. `C--` should be rich enough to be a viable backend for most mainstream and research compilers.

This paper should be sufficiently self-supporting so that anyone who knows an imperative language and is acquainted with computers should be able to write her/his own `C--` programs after reading this document.

2 Syntax definition

The syntax of `C--` is given in Figures 1 and 2.

2.1 General

A `C--` program file is written in eight bit ASCII characters. It consists in a sequence of `data` layout directives (Section 4), and/or procedure definitions (Section 5), and/or `import` declarations, and/or `export` declarations (Section 2.5), interleaved in any order.

A `C--` *compilation unit* is a `C--` program file that can be successfully compiled and that is suitable for linking.

`C--` does not support input/output. Nevertheless, it can be accomplished using a foreign language call (Section 3.8).

Program	*program*	$\rightarrow$	*pal* [*program*]	
Pal	*pal*	$\rightarrow$	*data*	
		$\mid$	[*conv*] *Name* `(`arg_1`,` $\ldots arg_n$`)` [*data*] *block*	$n \geq 0$
		$\mid$	`import` $Name_1$`,` $\ldots Name_n$ `;`	$n \geq 1$
		$\mid$	`export` $Name_1$`,` $\ldots Name_n$ `;`	$n \geq 1$
Data	*data*	$\rightarrow$	`data {`$datum_1 \ldots datum_n$`}`	$n \geq 1$
Datum	*datum*	$\rightarrow$	*Name* `:`	
		$\mid$	*type*[`[`*sconst*`]`][`{`$expr_1$`,` $\ldots expr_n$`}`] `;`	$n \geq 1$
		$\mid$	*type* `[]{`$expr_1$`,` $\ldots expr_n$`}` `;`	$n \geq 1$
		$\mid$	`word1[]`*AsciiString* `;`	*Abbreviation*
		$\mid$	`word2[]`*UnicodeString* `;`	*Abbreviation*
		$\mid$	`align`*n* `;`	*Alignment directive*
Simple Constants	*sconst*	$\rightarrow$	*Num*	*Integer constant*
		$\mid$	`'`*char*`'`	*Ascii char. constant*
		$\mid$	`unicode('`*char*`')`	*Unicode char. constant*
Constants	*const*	$\rightarrow$	*sconst*	*Simple constants*
		$\mid$	*FNum*	*Float number constant*
		$\mid$	*Name*	*Symbolic constant*
		$\mid$	*AsciiString*	*String constant*
		$\mid$	*UnicodeString*	*Unicode string constant*
Strings	*AsciiString*	$\rightarrow$	`"`$char_1 \ldots char_n$`"`	$n \geq 0$
	UnicodeString	$\rightarrow$	`unicode("`$char_1 \ldots char_n$`")`	$n \geq 0$
Convention	*conv*	$\rightarrow$	`foreign` *convkind*	*Convention declaration*
Conventions	*convkind*	$\rightarrow$	`C`	*Calling Conventions*
		$\mid$	`Pascal`	
		$\mid$	`...`	
Formal Arguments	*arg*	$\rightarrow$	*type Name*	
Type	*type*	$\rightarrow$	`word`*n* $\mid$ `float`*m*	$n \in \{1,2,4,8\}$, $m \in \{4,8\}$
Block	*block*	$\rightarrow$	`{`$stm_1 \ldots\ stm_n$`}`	$n \geq 0$

Figure 1: `C--` syntax

$$
\begin{array}{llcll}
\textit{Statements} & \textit{stm} & \rightarrow & \texttt{skip;} & \textit{Null statement} \\
 & & \mid & \textit{type}\ \textit{Name}_1\texttt{,}\ \ldots \textit{Name}_n\texttt{;} & \textit{Var. decl.},\ n \geq 1 \\
 & & \mid & \textit{Name}\ \texttt{=}\ \textit{expr}\ \texttt{;} & \textit{Assignment} \\
 & & \mid & \textit{type}[\texttt{\{align}n\texttt{\}}]\,\texttt{[}\textit{expr}\texttt{]}\ \texttt{=}\ \textit{expr}\ \texttt{;} & \textit{Memory write, align. } n \\
 & & \mid & \texttt{if}\ \textit{expr}\ \textit{rel}\ \textit{expr}\ \textit{block}\ [\texttt{else}\ \textit{block}] & \\
 & & \mid & \texttt{switch}[\texttt{[}\textit{sconst}_1\texttt{..}\textit{sconst}_n\texttt{]}]\ \textit{expr}\ \texttt{\{}\textit{swt}_1 \ldots \textit{swt}_n\texttt{\}} & n \geq 1 \\
 & & \mid & \textit{block} & \textit{Scoping} \\
 & & \mid & \textit{Name}\texttt{:} & \textit{Local control label} \\
 & & \mid & \texttt{goto}\ \textit{Name}\ \texttt{;} & \textit{Goto local label} \\
 & & \mid & \texttt{jump}\ \textit{expr}\ \texttt{(}\textit{expr}_1\texttt{,}\ \ldots \textit{expr}_n\texttt{)}\ \texttt{;} & n \geq 0,\ \textit{Jump to expr} \\
 & & \mid & [\textit{conv}]\ [\textit{Name}_1\texttt{,}\ \ldots \textit{Name}_m\texttt{=}]\ \textit{expr}\ \texttt{(}\textit{expr}_1\texttt{,}\ \ldots \textit{expr}_n\texttt{)}\ \texttt{;} & n,\ m \geq 0 \\
 & & \mid & [\textit{conv}]\ \texttt{return}\ \texttt{(}\textit{expr}_1\texttt{,}\ \ldots \textit{expr}_n\texttt{);} & n \geq 0 \\
 & & & & \\
\textit{Expressions} & \textit{expr} & \rightarrow & \textit{const} & \\
 & & \mid & \textit{Name} & \textit{Variable or label} \\
 & & \mid & \textit{type}\ [\texttt{\{align}n\texttt{\}}]\,\texttt{[}\textit{expr}\texttt{]} & \textit{Memory read, align. } n \\
 & & \mid & \texttt{(}\textit{expr}\texttt{)} & \\
 & & \mid & \textit{expr}\ \textit{op}\ \textit{expr} & \\
 & & \mid & \textit{prim}\ \texttt{(}\textit{expr}_1\texttt{,}\ \ldots\texttt{,}\ \textit{expr}_n\texttt{)} & n \geq 1 \\
 & & & & \\
\textit{Operators} & \textit{op} & \rightarrow & \texttt{+}\textit{flag} \mid \texttt{-}\textit{flag} \mid \texttt{*}\textit{flag} \mid \texttt{/}\textit{flag} \mid \texttt{\%}\textit{flag} & \textit{Arithmetic} \\
 & & \mid & \texttt{\&} \mid \texttt{|} \mid \texttt{\^{}} \mid \texttt{<<} \mid \texttt{>>}\textit{flag} \mid \texttt{\~{}} & \textit{Bitwise} \\
 & & & & \\
\textit{Primitives} & \textit{prim} & \rightarrow & \texttt{neg}\textit{flag} \mid \texttt{abs}\textit{flag} \mid \texttt{sign}\textit{flag} & \\
 & & \mid & \texttt{exponent}\textit{flag} \mid \texttt{fraction}\textit{flag} \mid \texttt{scale}\textit{flag} & \\
 & & \mid & \texttt{succ}\textit{flag} \mid \texttt{pred}\textit{flag} \mid \texttt{ulp}\textit{flag} & \\
 & & \mid & \texttt{trunc}\textit{flag} \mid \texttt{round}\textit{flag} & \\
 & & \mid & \texttt{intpart}\textit{flag} \mid \texttt{fractpart}\textit{flag} & \\
 & & \mid & \textit{type}\ \textit{flag} & \textit{Type Casts} \\
 & & & & \\
\textit{Flags} & \textit{flag} & \rightarrow & & \textit{No Flag} \\
 & & \mid & \texttt{o} & \textit{UnOrdered} \\
 & & \mid & \texttt{u} \mid \texttt{t} \mid \texttt{ut} & \textit{Unsigned and Trapping} \\
 & & \mid & \texttt{f} \mid \texttt{fz} \mid \texttt{fn} \mid \texttt{fp} & \textit{Floating and Rounding} \\
 & & \mid & \texttt{ft} \mid \texttt{ftz} \mid \texttt{ftn} \mid \texttt{ftp} & \textit{Floating and Trapping} \\
 & & & & \\
\textit{Relations} & \textit{rel} & \rightarrow & \texttt{==}\textit{flag} \mid \texttt{!=}\textit{flag} \mid \texttt{>}\textit{flag} \mid \texttt{<}\textit{flag} \mid \texttt{>=}\textit{flag} \mid \texttt{<=}\textit{flag} & \\
 & & & & \\
\textit{Switch branch} & \textit{swt} & \rightarrow & \textit{sconst}_1\texttt{,}\ \ldots \textit{sconst}_n\ \texttt{:}\ \textit{block} & n \geq 1 \\
 & & \mid & \texttt{default}\ \texttt{:}\ \textit{block} &
\end{array}
$$

Figure 2: Statements in C--

2.2 Comments

Comments start with `/*` and end with `*/`. They cannot be nested.

2.3 Names

Names are made up of letters, digits, underscore and dots. A name cannot begin with a number character or with a dot followed by a number character. Upper and lower case are distinct. Imported names should also follow these restrictions.

Names are identifiers for registers or memory addresses (Section 3.7).

The following are examples of legal `C--` names:

```
x
foo
_912
aname12
_foo.name_abit_12.long
Sys.Indicators
```

These are two illegal `C--` names:

```
.9Aname
3illegal
```

2.4 Name scope

Procedure and label names are always global inside a `C--` compilation unit (or program). Local variable names and local control labels are only in scope of the procedure body where they are declared. There is no nested scoping of names inside a procedure. Procedure and label names may be used before they are declared.

2.5 The `import` and `export` declarations

Names that are to be used outside of the `C--` program must be exported with the `export` declaration. Likewise, names that the `C--` program uses and does not declare must be imported with the `import` declaration. Only procedure and (pointer) label names may be exported.

Imported names should follow the syntatic restriction mentioned in Section 2.3.

An example where a few C external names are imported and a few `C--` names are exported:

```
import printf, sqrt; /* C procedures used in this C-- program */
export foo, bar;     /* To be used outside   this C-- program */
```

Names that are explicitly exported and imported are guaranteed to be unchanged by the compiler. All other names might be renamed.

An `import` or an `export` declaration may appear anywhere in the program where a `data` layout directive or a procedure definition does.

2.6 Constants

Constants can be (signed) integers, (signed) floating point numbers, characters, strings and names. `C--` follows C's syntax for denoting integer, floating point, character, and string constants.

2.6.1 Integer and floating point numbers

Integer constants have of type `word`. Floating point constants have type `float`. Their size is architecture-dependent.

2.6.2 Characters and strings

Character and string constants are treated as integers and as pointer labels respectively. Character constants are ASCII characters surrounded by single quotes. String constants are a sequence of ASCII characters surrounded by double quotes.

A character constant is treated as an integer whose value is the character's 8-bit ASCII code. Therefore, character constants have type `word1`. `C--` uses C's escape sequences to denote special characters, such as `\n` for the new line and `\t` for the tabulator.

For example, character constant `'H'` is a `word1` with value 72.

String constants are like labels that point to the first `word1` of an array of `word1`s stored in static memory. Therefore, they have type `word`n where n is the particular architecture's natural pointer size. *String constants are not automatically null-terminated.*

For example, the string `"Hello World"` is viewed as a label that points to the first byte of the array of bytes with values 72, 101, 108, 108, 111, 32, 87, 111, 114, 108, 100, stored in static memory.

It is possible to have UTF-8 integers for single characters and for string characters.[1]

The syntax to specify an UTF-8 constant is:

`unicode(`$constant$`)`

where $constant$ is a character constant or a string constant. The type of UTF-8 characters is `word2`, for it requires two bytes—two ASCII characters—to code a Unicode character. UTF-8 strings are pointers to the first `word2` of an array of `word2`s stored in static memory, and therefore, they have type `word`n, where n is the architecture's natural pointer size.

3 Fundamental concepts in `C--`

3.1 Memory

Memory is an array of bytes from which different sized *types* (Section 3.4) can be read and written. The size of the addressable memory is implementation dependent (Section 3.6). All addresses and

[1]UTF-8 is an encoding of Unicode characters into 8-bit ASCII characters that does not use any of the ASCII control characters to perform the coding. Unicode is an abbreviation for Universal Multiple-Octet Coded Character Set (UCS), and it is defined in ISO/IEC 10646. It is an international standard for encoding computer character sets that differs from historical ASCII. UTF-8 stands for Universal Transformation Format, 8-Bit form. See `http://www.unicode.org/unicode/standard/utf8.html` for more information.

offsets are specified in bytes. No guarantee about endianess is given, i.e. a portable program should either not depend on a specific endianess or find it out.

3.2 Data segment

The data segment is the part of memory where the static, initialised or uninitialised, data is allocated. The data segment is read/write, so the values stored can be changed at runtime. The size and initial content of the data segment is determined at compile time (Section 4). `C--` does not provide dynamic memory allocation natively, nonetheless, it can be accomplished with foreign language calls (Section 3.8 and Section 5.3).

3.3 Code segment

The code segment is the part of memory where the executable program code is stored (Section 5). The code segment consists of a series of procedure definitions.

`C--` does not currently provide a mechanism for creating code at runtime.

3.4 Types

There are only two kinds of types provided by `C--`, namely `word` and `float`. These types can have different sizes.

- The size of a `word` can be 1, 2, 4 or 8 bytes.
- The size of a `float` can be 4 or 8 bytes.

A type must be qualified with a size. Thus `word1` and `word2` are different types.

There is no pointer type. The `word`n type can be used for pointers (addresses), where n is the particular architecture's *natural pointer size*, i.e.: n is the number of bytes needed to hold a memory address in the particular architecture.

For example, a four byte word is specified as `word4`, an eight byte float is specified as `float8` and so on.

Types are used in

1. **Declaration statements** (Section 5.2.2), to declare the type of local variables.
2. **Memory write statements and memory read expressions** (Section 5.2.4 and Section 6.2, respectively) to indicate the type of the value written/read.
3. **Data layout directives** (Section 4) to indicate the type of the allocated datum.

3.5 Local variables (or registers)

Any number of local variable names may be declared inside procedure bodies. They are typed storage locations that don't have an address. The term "local variable name" is interchangeable with the term "register", since there is an unlimited supply of (virtual) registers: i.e. a local variable name will be mapped to a machine register if there is one available, otherwise it is mapped to a memory location (e.g. the stack), but the mapping is transparent and local variables should be viewed as registers.

3.6 Addresses

To specify where in memory to read or write we need an address. Any expression that evaluates to a `word`n can be used as an address, where n is the architecture's *natural pointer size*, i.e.: n is the number of bytes needed to hold a memory address in the particular architecture.

Absolute addresses can be used but what they refer to is implementation dependent. Their type is also `word`n.

3.7 Names

A name declares either a register or a memory address. Register names are procedure's local variables. Memory address names are either labels (Section 4.1) or procedure names (Section 5).

3.8 Foreign language interface

The foreign language interface is a way for `C--` programs to use other calling conventions for procedure inter-operation with foreign code. This interface is nearly 100% portable across architectures (Section 5.3).

4 Data layout directives

Memory in the data segment is allocated using the `data` directive. A memory block is organised as a sequence of typed data. Each *datum* is thought of as an array of bytes that may be initialised.

Here is an example that allocates and initializes some memory. In particular, it allocates 8 datums of types `word4`, `word2`, `float8`, `word4`, `word1`, `word2`, `word1` and `word8` respectively. The example is explained in more detail in the remainder of this section:

```
data {
  foo:  word4[4]{1,2,3,ff};  /* ff is a forward reference */
        word2[4]{1,2};
  ff:   float8[2]{2.8,3.1}; word4[2]{ff,foo};
  str:  word1[]"Hello world\0";
  ustr: word2[]unicode("Hello world\0");
        word1;
  xs:   word8[]{"This is an", "array of", "word8's"};
}
```

There may be any number of data layout directives in a `C--` program.

4.1 Labels

Labels are the means to refer to the allocated memory. They should be viewed as pointers and not as memory locations. A label declaration consists of a name followed by a colon. Once declared, a label is a name (and so an expression) that refers (points) to a memory address. Therefore it has `word`n type, where n is the particular architecture's natural pointer size. Labels may be used before their are declared, e.g. label `ff` is used in the initialisation of the data directive's first datum before it is declared pointing to the third.

Note that labels do not provide any information about the type of the data pointed to by them.

A label points to the first byte after its declaration. Here is an example in which four labels point to the same datum:

```
data { foo: label1: label2: bar: word4} /* just allocates */
```

Memory is always allocated without padding inside a single data layout directive, so it is possible to find any given data in the data segment by starting from a label and adding the right offset, as in, for example, the read expression `word2[foo+4]`. Indeed, `foo+4` does not have to point to the beginning of a data element. It may point to any other data byte, but it is assumed by `C--` that it is 2-byte aligned.

To align a label (and hence the datum it points to) to a specific boundary, an alignment directive (Section 4.3) has to be placed before the label. In the following example, `foo` and `bar` might or might not be the same address, but `bar` is guaranteed to be aligned on an eight byte boundary.

```
data { foo: align8;
       bar: word4{0};
     }
```

It is be possible to have a stupid data layout directive with no labels that is inaccessible.

4.2 Initialisation

Memory is allocated by specifying the type of the datum, the number of datum's elements to allocate, and the initial value for each element. The particular syntax is:

$type$`[`n`]{`$constant\text{-}list$`};`

where n specifies how many elements of the type $type$ have to be allocated, and $constant\text{-}list$ provides the initial value (of type $type$) for each allocated element, in the form of a comma-separated list of constants or constant expressions (i.e. expressions whose value is known at compile time).

There are a number of possible variants:

1. If `[`n`]` is not provided, only one element is allocated. The `{`$constant\text{-}list$`}` may or may not be provided. If provided, it should contain only one constant or constant expression to which the element is initialised. If not provided, no initial value is given. For example:

   ```
   data { lb1: word1; }
    /* Allocates one byte (contains garbage) */
   data { lb2: word1{17}; }
    /* Allocates one byte and initialises it to (ASCII code) 17 */
   ```

```
data { lb3: word4{17}; }
 /* Allocates one 4-byte word and initialises it to integer 17   */
```

2. If `[`n`]` is provided, then n elements are allocated. The {*constant-list*} may or may not be provided. If not provided, no initial value is given. If provided, it should contain c constants or constant expressions, such that $c \leq n$. Element i $(i : 0 \ldots (n-1))$ is initialised to the value of the constant or constant expression j $(j : 0 \ldots (c-1))$ in {*constant-list*}, such that $j = i \; mod \; c$. For example:

```
data { lb1: word1[17]; }
 /* Allocates 17 bytes (that contain garbage) */
data { lb2: word1[17]{0}; }
 /* Allocates 17 bytes and initialises all of them to 0 */
data { lb3: word4[6]{1,2,3}; }
 /* Allocates six 4-byte words and initialises them
  * to 1,2,3,1,2,3 respectively.
  */
data { lb4: word4[6]{1,2,3,1,2,3}; }
 /* Allocates six 4-byte words and initialises them
  * to 1,2,3,1,2,3 respectively.
  */
data { lb5: word4[4]{1,2,3}; }
 /* Allocates four 4-byte words and initialises them
  * to 1,2,3,1 respectively.
  */
```

3. There is also the possibility to have abbreviations when $n = c$.

type `[]{`*constant-list*`};`	*is an abbreviation for*	*type* `[c]{`*constant-list*`};`
`word1[]"`$char_1 \ldots char_n$`";`	*is an abbreviation for*	`word1[n]{'`$char_1$`',...,'`$char_n$`'};`
`word2[]unicode("`$char_1 \ldots char_n$`");`	*is an abbreviation for*	
	`word2[n]{unicode('`$char_1$`'),...,unicode('`$char_n$`')};`	

For example:

```
data { s1: word1[6]{'h','e','l','l','o','\0'}; }
data { s2: word1[]"hello\0"; }
 /* Both directives  allocate 7 bytes and initialise
  * them to the same ASCII code integers.
  */
data { f1: float8[3]{3.5, 4.4, 6.98}; }
data { f1: float8[] {3.5, 4.4, 6.98}; }
 /* Both directives allocate three 8-byte floats and initialise
  * them to the same floating point numbers.
  */
```

Since the initialised value might have dependencies on the endianess, the only way to guarantee that a memory read (Section 6.2) gets the same initialised (or written) value, is to read the datum or the element with the same type as it was initialised (or written). For example, if a datum was initialised with `data {foo: word2{17};}`, if read back with `word1[foo]` the value might be 0 or 17 depending on the architechture, but if read with `word2[foo]` it is guaranteed to be 17.

4.3 Alignment

For performance reasons, and also to comply with some architecture requirements, it is sometimes necessary to specify the alignment of data. The `align`n directive inserts padding as needed, ensuring that the next datum is placed on an n byte boundary. The value of the padding is unspecified.

In the following example, `foo` is aligned to a 4-byte boundary and `bar` to an 8-byte boundary. In both cases padding may be inserted: for example, between the last byte of the `word4` and the first byte of the `float8` (the datum pointed to by `bar`) there may be padding in order to place the `float8` on an 8-byte boundary.

```
data { align4;
       foo: word4{1};
       align8;
       bar: float8{1.7};
     }
```

5 Procedures

Procedures are the means to place information in the code segment. They are very similar to high-level language procedures. Procedures can optionally take arguments, contain static data declarations and return values.

5.1 Procedure definition

A procedure definition has the following syntax:

$conv\ proc\_name$ `(`$type\ arg_1$`,` $\ldots$ `,`$type\ arg_n$`)`
`{` $body$ `}`

where:

- $conv$ is the (optional) calling convention declaration (Section 5.3) needed for inter-operation.
- $proc\_name$ is the procedure *name*, which stands for the procedure entry point address and, as a name, will be an expression (Section 3.7). It may be also used in call (Section 5.2.8) and `jump` (Section 5.2.9) statements.
- $type\ arg_1$`,` $\ldots$`,` $type\ arg_n$ is the formal argument list. Each formal argument consists of a *name* preceded by its type. If the procedure takes no arguments, the list should be empty. Unlike C, procedures with a variable number of arguments are not supported. The formal arguments are only in scope of the procedure body.
- $body$ is the procedure body enclosed in curly braces. It may consists of a sequence of statements, such as local variable declarations, assignments, memory reads, local control label declarations, `goto`s, etc. It is unspecified what happens if control flows off the edge of a procedure body, that is, every control path in a procedure body should finish in a `jump` (Section 5.2.9) or in a `return` (Section 5.2.10) statement. Therefore, a procedure body should have at least one statement.

The return type needs not be specified in the definition.

For example, procedure `foo` is defined as a procedure that expects one `word4` argument. Inside the procedure body, the local variable (or register) `x` is declared, followed by an assignment statement and a `jump` statement to procedure `bar`.

```
foo(word4 y) {
  word4 x;

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

5.2 Statements

5.2.1 `skip;`

This is just the null statement and can be inserted anywhere an ordinary statement can. It does not have any effects. It is used for clarity instead of the error-prone stand-alone semicolon.

5.2.2 Declaration

A declaration statement has the following syntax:

$$type\ name_1 \texttt{,} \ldots \texttt{,} name_n \ \texttt{;}$$

It declares the local variable names $name_1 \ldots name_n$ of type $type$. These names will be mapped to (virtual) machine registers. As names, they are also expressions of type $type$.

Local variables have to be declared before they are used.

A declaration statement may appear anywhere inside the procedure body. All declarations are treated as if they were declared at the beginning of the procedure body. All the local variable names must be unique. It is not possible to redeclare a name.

5.2.3 Assignment

An assignment statement has the following syntax:

$$name = expr \ \texttt{;}$$

It stores the value of $expr$ in the local variable (or register) $name$, where $expr$ has the same type as $name$.

5.2.4 Memory write

A memory write statement has the following syntax:

$$\texttt{word}n\texttt{[}expr_1\texttt{]} \ \texttt{=} \ expr_2 \ \texttt{;}$$

to write `word`n values, or

`float`n`[`$expr_1$`] = `$expr_2$`;`

to write `float`n values.

Expression $expr_1$ has type `word`n, where n is the particular architecture's natural pointer size, and its value is the memory address in which the value of $expr_2$ is written. Expression $expr_1$ will typically contain one or more labels. Expression $expr_2$ should be of type `word`n or `float`n respectively, otherwise the value written in memory is unspecified.

The following example stores the ASCII integer code of 'A' in the 4th byte of the datum pointed to by `label`

```
word1[label+4] = 'A';
```

The address yielded by $expr_1$ is assumed aligned to the size of the type, namely, n. A memory write can optionally be qualified with an alignment flag `{align`a`}`, so the syntax is now:

`word`n`{align`a`}[`$expr_1$`] = `$expr_2$`;`
`float`n`{align`a`}[`$expr_1$`] = `$expr_2$`;`

A few examples of memory writes with flagged alignment:

- `float8{align4}[label]` = $expr$ does a 8-byte write but assumes that `label` is aligned to a 4 byte boundary.
- `word4{align1}[label]` = $expr$ does a 4-byte write but assumes that `label` is aligned to a byte boundary (pointer to a byte).
- `word1{align4}[label]` = $expr$ does a 1-byte write but assumes that `label` is aligned to a 4 byte boundary.

5.2.5 `if` and relational operations

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

`if` $expr_1$ rel $expr_2$ `{ ... } else { ... }`

The `else` branch is optional and the statement blocks may be empty, as in `if x == 0 {}`, but the curly braces are mandatory even for single statements, as in

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

The condition test is very simple: it consists of a relational operation, rel, that takes two expressions as arguments. The term "operation" is used instead of "operator", therefore avoiding confusion with `C--` operators that are used in expressions (Section 6). Relational operations are only used in `if` condition tests; they cannot be used anywhere else.

This is the set of relational operations:

Name	*Relation*
`==`	*Equality*
`!=`	*NonEquality*
`>`	*Greater Than*
`>=`	*Greater Than or Equal*
`<`	*Less Than*
`<=`	*Less Than or Equal*

They can all be combined with these flags:

Flag	*Meaning*
	Signed comparison (default)
`u`	*Unsigned comparison*
`f`	*Floating point comparison*
`fo`	*Floating point unordered comparison, if supported*

When the condition test holds, the block of statements immediately following the condition test is executed. Otherwise, if an optional `else` branch has been specified, its block of statements is executed. After execution of the any of these blocks, control resumes at the first statement after the `if` or `if/else`.

In the following example, `>=` is used in the `if` test condition without a flag (default signed comparison), and `!=` is used combined with flag `u` to test whether the unsigned integer held in `x` is zero.

```
f(word4 x)
{
  word4 y;

  y = 0;
  if y >= word4[foo+8] {
    y = y + 1;
    return (y);
  } else {
    x = x -u 1;
    if x !=u 0 {
       y = y + 2;
    }
    return (y);
  }
}
```

5.2.6 `switch`

The `switch` statement performs multiway branching depending on the value of a `word` expression. The particular syntax is:

`switch [`$sconst_1$`..`$sconst_n$`]` *expr* `{`

$$
\begin{array}{rl}
sconst_{11}\texttt{,}\ \ldots\texttt{,}\,sconst_{1i} & \texttt{: }\{\ldots\} \\
\vdots & \\
sconst_{m1}\texttt{,}\ \ldots\texttt{,}\,sconst_{mj} & \texttt{: }\{\ldots\} \\
\texttt{default} & \texttt{: }\{\ldots\}
\end{array}
$$
}

where:

- $expr$ is an expression that yields a `word` value.
- $sconst_{k1}$`,` ... `,` $sconst_{kl}$`:` { ... } is branch k $(k : 1 \ldots m)$, in which multiple simple-constant[2] alternatives $(l : 1 \ldots i, j \ldots)$ may be specified.

  When the value of $expr$ is any of $sconst_{k1} \ldots sconst_{kl}$, branch k is taken, executing its block of statements and resuming control at the first statement after the `switch`. There is no fall through between different branches: `C--` assumes that earlier branches are more likely to be taken.
- `default` is the (optional) default branch that is taken when none of the others are taken. The effect is unspecified if none of the branches are taken—none of the $sconst_{ij}$ match $expr$'s value—and no default branch is provided.
- `[`$sconst_1$`..`$sconst_n$`]` is an (optional) range of simple constants in which the value of $expr$ is guaranteed to be. This range is a hint to the compiler. No bounds checking is performed at run-time to see whether $expr$'s value is in the range.

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 `default` branch is taken.

```
switch [0..7] x + 23 {
  1,2,3   : { y = y + 1;}
  5       : { y = x + 1; x = y;}
  default : { y = f();
              if y == 0 { x = 1;}
            }
}
```

5.2.7 Local control labels and `goto`

Local control labels are used in conjuction with the `goto` statement to alter the control flow within a procedure body. A local control label declaration consists of a label name followed by a colon. This kind of control label is not a *name* in the sense of Section 3.7, and so, it should not be confused with the pointer labels mentioned so far. The *only thing* that can be done with a local control label is to provide it as argument to `goto` statements.

In turn, a `goto` statement transfers control to the label it takes as argument. Only a local control label can be the argument of a `goto`.

[2] That is, `word` integers or characters. See Figure 1.

In the following example, the `goto` statement forces the control flow to resume to the very first statement after the label declaration.

```
bar()
{
label:
  word8[foo] = 18;
  word8[foo+4*8] = word8[bar];
  goto label;
  return();
}
```

5.2.8 Procedure call

A call statement invokes a procedure in the conventional way of function invocation, so all the invoking procedure's local variables are saved across the call. The particular syntax is:

$$name_1, \ldots, name_n = conv\ expr\ (expr_1, \ldots, expr_m);$$

where:

- $name_1, \ldots, name_n =$ is the local variable name list. The results returned back by the procedure are stored in each variable in the order in which they are returned, from left to right, by the invoked procedure's `return` statement (Section 5.2.10). If the invoked procedure returns no values, the name list should be omitted, otherwise the values of the names are unspecified after the call.
- $conv$ is the (optional) calling convention declaration needed for inter-operating with foreign code. (Section 5.3)
- $expr$ is any expression that evaluates to a procedure address. It will typically be a (procedure) name.
- $expr_1, \ldots, expr_m$ is the (optional) actual argument list, where each actual argument $expr_i$ is an expression. All the expressions are passed by value to the called procedure. If no arguments are passed, the list should be empty, as in, for example, `x = f();`.

It is unspecified what the effects are if the number and the types of the actual arguments in a call statement do not match the number and the types of the formal arguments of the invoked procedure. It is also unspecified what the effects are if the number and the types of the names in the name list do not match the number and the types of the results returned by the invoked procedure.

Call statements are not expressions and so cannot be used inside expressions. Procedure calls are complete statements. Things such as `y = f(g(x)) + 1;` are not allowed. Recall, however, that procedure *names*, as such, are expressions with the procedure address as value.

The following example is self-explanatory:

```
foo()
{
  word4 x, y;
  x, y = bar(5);
```

```
  return (x,y);
}
bar(word4 x)
{
 return (x, x+1);
}
```

5.2.9 `jump`

The `jump` statement performs a control jump but carrying parameters. It has as target any expression that evaluates to a procedure address and can optionally transfer arguments to that procedure. The syntax is:

`jump` $expr$ `(`$expr_1$`,...,`$expr_n$`);`

where:

- $expr$ is any expression that evaluates to a procedure address. It will typically be a (procedure) name.
- $expr_1$`,...,`$expr_n$ is the (optional) actual argument list, where each actual argument $expr_i$ is an expression. All the expressions are passed by value to the target procedure. If no arguments are passed, the list should be empty, as in, for example, `jump bar ();`.

All local variables die when jumping. It is unspecified what the effects are if the number and the types of the actual arguments in a `jump` statement do not match the number and the types of the formal arguments of the invoked procedure. An example of an infinite loop with no stack growth:

```
bar(word4 x, word4 y)
{
  jump bar(y, x);    /* Loop forever */
}
```

5.2.10 `return`

The `return` statement transfers control back to the call statement issued by an invoking procedure. Optionally, it can also transfer values back. All the local variables of the procedure issuing the `return` die when returning. The syntax is:

`return (`$expr_1$`,...,`$expr_n$`);`

where $expr_i$ are the expressions whose values will be returned. If no values are returned, the expression list should be empty, as in `return ();`. Note that in `C--`, a procedure may return multiple values.

The return statement may be qualified with the calling convention to be used (Section 5.3)

It is unspecified what the effects are if the number and the types of the values returned do not match between a `return` and the call statement.

```
bar(word4 z)
{
  return (1+z, z/3);
{
foo(word4 z)
{
  return ();
{
```

5.3 Foreign language interface

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` uses the standard `C` calling convention.

```
export foo;
foreign C foo()
{
  word4 x;
  jump bar(x);
}
```

The calling convention should be also specified in the same way in call statements and in `return` statements, if it is not `C--`'s calling convention.

```
import printf, fun;
goo()
{
  word4 i;
  foreign C fun(5);
      /* fun has type int -> void  */
  foreign C i = printf(str, arg);
      /* printf() returns an int   */
  return ();
}
bar(word4 a)
{
  a = a + 1;
  foreign C return (a); /* uses C's convention to return 'a'   */
}
```

There supported calling conventions are:

1. C
2. Pascal

All foreign language functions/procedures must have been imported with import declarations. All `C--` procedures directly invoked from a foreign language must have been exported with export declarations.

When calling a `C--` procedure from a foreign program, the types and sizes of the actual arguments should match the types and sizes of the formal arguments in the particular platform, otherwise the effects are unspecified. The same applies for the types and sizes of returned values.

When inter-operating with foreign languages, since the size of a particular foreign language type may differ between platforms, and since `C--` types always has fixed-size types, it is impossible for `C--` to be completely platform independent when inter-operating with foreign languages.

6 Expressions

6.1 Introduction

An `C--` expression can be a constant, a name, a memory read, a primitive, or an operator applied to other expressions. `C--` makes a distinction between integer and floating point expressions, i.e., expressions that yield `words` or `floats` as result.

The integer and floating point model is based on the LIA-1 standard (ISO/IEC 10967-1:1994(E)) and if there are any inconsistencies between this manual and LIA-1, the LIA-1 standard is correct, unless otherwise noted.

Signed and unsigned numbers are not distinguished. Instead, like any other assembler, it is the operations that are typed.

The type of any subexpression is always known and there are no automatic type casts or type conversions.

The following sections cover all the `C--` operators, all the `C--` primitives, and the memory read expression.

6.2 Memory read

Memory read expressions have the following syntax:

> `word`m`[`$expr$`]`
> Type: `word`n $\rightarrow$ `word`m

to read a `word`m value, and

> `float`m`[`$expr$`]`
> Type: `word`n $\rightarrow$ `float`m

to read a `float`m value.

Expression $expr$ has type `word`n, where n is the particular architecture's natural pointer size. Its value is the address of the memory location to read from. It will typically contain one or more labels. The size m indicates how many bytes to read from that location.

The following example expression reads a 4-byte `word` from the second byte pointed to by label `p`:

```
word4[p+1]
```

The address yielded by $expr$ is assumed aligned to an m-byte boundary. A memory read can optionally be qualified with an alignment flag `{align`a`}`. The syntax is:

```
wordm{aligna}[expr]
floatm{aligna}[expr]
```

A few examples of memory reads with flagged alignment:

- `float8{align4}[label]` reads a 8-byte `float` but assumes that `label` is aligned to a 4 byte boundary.
- `word4{align1}[label]` reads a 4-byte `word` but assumes that `label` is aligned to a byte boundary (pointer to a byte).
- `word1{align4}[label]` reads 1-byte but assumes that `label` is aligned to a 4 byte boundary.

6.3 Operators, precedence, and evaluation order

`C--` operators are typed, i.e. there is a different set of operators for the two types provided by `C--`. The following table lists the available operators for signed `word`s and `float`s. Operators for *unsigned* `word`s can be obtained appending the u flag to the signed `word` operators.

Each operator in the table is described in more detail in Section 6.7 and Section 6.8.

Operator	*type*	*flags it can take*	*class*	*Description*
`*`	word$n$ × word$n$ → word$n$	`t, u, h`	Arithmetic	Integer multiplication
`*f`	float$n$ × float$n$ → float$n$	`t, z, n, p`	Arithmetic	Floating point multiplication
`/`	word$n$ × word$n$ → word$n$	`t, u`	Arithmetic	Integer division
`/f`	float$n$ × float$n$ → float$n$	`t, z, n, p`	Arithmetic	Floating point division
`+`	word$n$ × word$n$ → word$n$	`t, u`	Arithmetic	Integer addition
`+f`	float$n$ × float$n$ → float$n$	`t, z, n, p`	Arithmetic	Floating point addition
`-`	word$n$ × word$n$ → word$n$	`t, u`	Arithmetic	Integer substraction
`-f`	float$n$ × float$n$ → float$n$	`t, z, n, p`	Arithmetic	Floating point substraction
`%`	word$n$ × word$n$ → word$n$	`t`	Arithmetic	Integer modulo
`~`	word$n$ → word$n$		Bitwise	Complement
`&`	word$n$ × word$n$ → word$n$		Bitwise	AND
`|`	word$n$ × word$n$ → word$n$		Bitwise	OR
`^`	word$n$ × word$n$ → word$n$		Bitwise	XOR
`<<`	word$n$ × word$n$ → word$n$		Bitwise	Left shift
`>>`	word$n$ × word$n$ → word$n$	`u`	Bitwise	Right shift

Operators should have as arguments expressions of the appropriate type, otherwise the result may be unspecified.

The next table lists the C-- operators in decreasing order of precedence. Operators in the same row have the same precedence. The reader can see that C-- operators follow the precedence order and the associativity of their C counterparts.

Operators			*Associates*
~			right
* *f	/ /f	%	left
+ +f	- -f		left
<<	>>		left
&			left
^			left
|			left

6.4 Primitives

C-- provides a set of primitive operators besides those described above. The general syntax of a primitive is:

prim_name ($expr_1$, ... $expr_n$)

where $expr_i$ are expressions and *prim_name* is the primitive's name. A primitive name is not a *name* in the sense of Section 3.7. Primitive names alone are not expressions that stand for the primitive's entry point address, since primitives are not procedures but built-in operators. Primitives can only be used inside expressions using the syntax given above.

There are word primitives and float primitives. The following table lists all the C-- primitives. See Sections 6.7 and 6.8 for detailed information on each particular primitive.

	Primitive	*Type*
word *primitives*	abs	word$n$ → word$n$
	neg	word$n$ → word$n$
	sign	word$n$ → word$n$
float *primitives*	absf	float$n$ → float$n$
	exponentf	float$n$ → word$n$
	fractionf	float$n$ → float$n$
	fractpartf	float$n$ → float$n$
	intpartf	float$n$ → float$n$
	negf	float$n$ → float$n$
	predf	float$n$ → float$n$
	roundf	float$n$ × word$n$ → float$n$
	scalef	float$n$ × word$n$ → float$n$
	signf	float$n$ → word$m$
	succf	float$n$ → float$n$
	truncf	float$n$ × word$n$ → float$n$
	ulpf	float$n$ → float$n$

6.5 Exception handling

Operators may cause system exceptions such as, for example, overflow or divide-by-zero. Operators can keep record of the exception resulted from their application if they are appended with the `t` (trap) flag. The exception kind is recorded in the global *register*[3] `Sys.Indicators`, which is a bit vector with a bit for every kind of exception. To find out which exception has ocurred, `C--` provides some predefined global constants. They are also bit vectors, with the particular bit that encodes the exception set to 1 and all the others set to 0.

System exceptions and constants are listed in the following table:

$Exceptions$	$lia1except$	$\rightarrow$	`Sys.IntegerOverflow`	
		$\mid$	`Sys.FloatingOverflow`	
		$\mid$	`Sys.Underflow`	
		$\mid$	`Sys.Undefined`	
		$\mid$	`Sys.Inexact`	IEC 559
		$\mid$	`Sys.DivideByZero`	IEC 559
		$\mid$	`Sys.Invalid`	IEC 559
$Constants$	$lia1info$	$\rightarrow$	`Sys.wordn.MaxSigned`	
		$\mid$	`Sys.wordn.MinSigned`	
		$\mid$	`Sys.wordn.MaxUnSigned`	
		$\mid$	`Sys.wordn.MinUnSigned`	
		$\mid$	`Sys.floatn.Radix`	
		$\mid$	`Sys.floatn.Precision`	
		$\mid$	`Sys.floatn.ExpMin`	
		$\mid$	`Sys.floatn.ExpMax`	
		$\mid$	`Sys.floatn.Denorm`	
		$\mid$	`Sys.floatn.IEC559`	
		$\mid$	`Sys.floatn.Max`	
		$\mid$	`Sys.floatn.Min`	
		$\mid$	`Sys.floatn.MinN`	
		$\mid$	`Sys.floatn.Epsilon`	

It is easy to find out the kind of exception that has resulted from an operator application using the bitwise operators on `Sys.Indicators` and the appropriate global constants. For example, to capture, handle, and clean up an overflow exception that resulted from the application of a `word` addition operator, one could write:

```
foo(word4 y) {
  word4 x;

  x = y +t y
  if Sys.Indicators & Sys.IntegerOverflow {
      /* Write here code to handle exception */

      /* Clear handled exception */
      Sys.Indicators = Sys.Indicators & ~Sys.IntegerOverflow;
  }
```

[3] `Sys.Indicators` may be viewed as a global *variable*, but indeed, it is the only possible global variable in a `C--` program.

```
}
```

The `Sys.Indicators` register can be treated as any other register, i.e. it can be cleared, bits can be flipped and so on.

6.6 Casting

6.6.1 word$n$

Type: word$m$ $\rightarrow$ word$n$
Accepts flags: `u`
With `t` flag it sets: N/A
Description:

> If $n > m$, it typecasts returning the higher order bits.
> If $n < m$, it typecasts returning a value in which the new higher order bits are either filled with the highest order bits of its argumet value (sign extension), or filled with zeroes used with the `u` flag.

Example expression: `word4('A')` —sign extension. `word2u(abyte)` —zero fill.

6.6.2 float$n$

Type: word$m$ $\rightarrow$ float$n$
Accepts flags: `t`
With `t` flag it sets: `Sys.FloatingOverflow`
Description:

> Typecasts an integer into a floating point number, if `t` is used it raises `Sys.FloatingOverflow` if the argument is to big to fit.

Example expression: `float8(foo)`

6.7 `word` operators and primitives

All `wordn` types are bounded and the min and max values are provided as constants : `Sys.word`n`.MaxSigned`, `Sys.word`n`.MinSigned`, `Sys.word`n`.MaxUnSigned` and `Sys.word`n`.MinUnSigned`.

The following definitions will be used to explain the individual operators. I denotes the set of possible integer values. N denotes the subset of I of positive integer values.

$$\begin{array}{lcl} I & = & \{x \in Z \mid \texttt{Sys.word}n\texttt{.MinSigned} \leq x \leq \texttt{Sys.word}n\texttt{.MaxSigned}\} \\ N & = & \{x \in Z \mid \texttt{Sys.word}n\texttt{.MinUnSigned} \leq x \leq \texttt{Sys.word}n\texttt{.MaxUnSigned}\} \\ & & \\ wrap_I(x) & = & x \,\texttt{modulo}\, (\texttt{Sys.word}n\texttt{.MaxSigned} - \texttt{Sys.word}n\texttt{.MinSigned} + 1) \\ wrap_I(x) & \in & I \\ & & \\ wrap_N(x) & = & x \,\texttt{modulo}\, (\texttt{Sys.word}n\texttt{.MaxUnSigned} - \texttt{Sys.word}n\texttt{.MinUnSigned} + 1) \\ wrap_N(x) & \in & N \end{array}$$

6.7.1 + and −

Type: $\texttt{word}n \times \texttt{word}n \to \texttt{word}n$
Accepts flags: `t` and `u`
With `t` flag it sets: `Sys.IntegerOverflow`
Description:

$$\begin{array}{lll} x \pm y & = x \pm y & if\ x \pm y \in I \\ & = wrap_I(x \pm y) & if\ x \pm y \notin I \\ \\ x \pm_t y & = x \pm y & if\ x \pm y \in I \\ & = \texttt{Sys.IntegerOverflow} & if\ x \pm y \notin I \\ \\ x \pm_u y & = x \pm y & if\ x \pm y \in N \\ & = wrap_N(x \pm y) & if\ x \pm y \notin N \\ \\ x \pm_{ut} y & = x \pm y & if\ x \pm y \in N \\ & = \texttt{Sys.IntegerOverflow} & if\ x \pm y \notin N \end{array}$$

Example expression: `foo + 17`

6.7.2 *

Type: $\texttt{word}n \times \texttt{word}n \to \texttt{word}n$
Accepts flags: `t`, `u`, and `h`
With `t` flag it sets: `Sys.IntegerOverflow`
Description:

$$\begin{array}{lll} x * y & = x * y & if\ x * y \in I \\ & = wrap_I(x * y) & if\ x * y \notin I \\ \\ x *_t y & = x * y & if\ x * y \in I \\ & = \texttt{Sys.IntegerOverflow} & if\ x * y \notin I \\ \\ x *_u y & = x * y & if\ x * y \in N \\ & = wrap_N(x * y) & if\ x * y \notin N \\ \\ x *_{ut} y & = x * y & if\ x * y \in N \\ & = \texttt{Sys.IntegerOverflow} & if\ x * y \notin N \\ \\ x *_h y & = high(x * y) & the\ higher\ order\ bits \\ x *_{uh} y & = high(x * y) & the\ higher\ order\ bits \end{array}$$

Example expression: `foo * 17`

6.7.3 /

Type: $\texttt{word}n \times \texttt{word}n \to \texttt{word}n$
Accepts flags: `t` and `u`

With `t` flag it sets: `Sys.IntegerOverflow` or `Sys.Undefined`
Description:

$$\begin{aligned} x/y &= \lfloor x/y \rfloor & if\ \lfloor x/y \rfloor \in I \\ &= wrap_I(\lfloor x/y \rfloor) & if\ \lfloor x/y \rfloor \notin I \\ &= undefined\, value & if\ y = 0 \end{aligned}$$

$$\begin{aligned} x/_t y &= \lfloor x/y \rfloor & if\ y \neq 0\ and\ \lfloor x/y \rfloor \in I \\ &= \texttt{Sys.IntegerOverflow} & if\ y \neq 0\ and\ \lfloor x/y \rfloor \notin I \\ &= \texttt{Sys.Undefined} & if\ y = 0 \end{aligned}$$

$$\begin{aligned} x/_u y &= \lfloor x/y \rfloor & if\ \lfloor x/y \rfloor \in N \\ &= wrap_N(\lfloor x/y \rfloor) & if\ \lfloor x/y \rfloor \notin N \\ &= undefined\, value & if\ y = 0 \end{aligned}$$

$$\begin{aligned} x/_{ut} y &= \lfloor x/y \rfloor & if\ y \neq 0\ and\ \lfloor x/y \rfloor \in N \\ &= \texttt{Sys.IntegerOverflow} & if\ y \neq 0\ and\ \lfloor x/y \rfloor \notin N \\ &= \texttt{Sys.Undefined} & if\ y = 0 \end{aligned}$$

Example expression: `foo / 17`

6.7.4 `%`

Type: `word`n $\times$ `word`n $\rightarrow$ `word`n
Accepts flags: `t`
With `t` flag it sets: `Sys.Undefined`
Description:

$$\begin{aligned} x\%y &= x - (\lfloor x/y \rfloor * y) & if\ y \neq 0 \\ &= undefined\, value & if\ y = 0 \end{aligned}$$

$$\begin{aligned} x\%_t y &= x - (\lfloor x/y \rfloor * y) & if\ y \neq 0 \\ &= \texttt{Sys.Undefined} & if\ y = 0 \end{aligned}$$

Example expression: `foo % 17`

6.7.5 `neg` and `abs`

Type: `word`n $\rightarrow$ `word`n
Accepts flags: `t`
With `t` flag it sets: `Sys.IntegerOverflow`
Description:

$$
\begin{aligned}
neg(x) &= -x & if\ \ -x \in I\\
&= wrap_I(-x) & if\ \ -x \notin I
\end{aligned}
$$

$$
\begin{aligned}
negt(x) &= -x & if\ \ -x \in I\\
&= \texttt{Sys.IntegerOverflow} & if\ \ -x \notin I
\end{aligned}
$$

$$
\begin{aligned}
abs(x) &= |x| & if\ |x| \in I\\
&= wrap_I(|x|) & if\ |x| \notin I
\end{aligned}
$$

$$
\begin{aligned}
abst(x) &= |x| & if\ |x| \in I\\
&= \texttt{Sys.IntegerOverflow} & if\ |x| \notin I
\end{aligned}
$$

Example expression: `neg(foo)`

6.7.6 `sign`

Type: `word`$n \rightarrow$ `word`n
Accepts flags: N/A
With `t` flag it sets: N/A
Description:

$$
\begin{aligned}
sign(x) &= 1 & if\ x \geq 0\\
&= 0 & if\ x = 0\\
&= -1 & if\ x \leq 0
\end{aligned}
$$

Example expression: `sign(foo)`

6.7.7 `&`, $|$, and `^`

Type: `word`$n \times$ `word`$n \rightarrow$ `word`n
Accepts flags: N/A
With `t` flag it sets: N/A
Description:

$$
\begin{aligned}
x\,\&\,y &= x\ AND\ y & Bitwise\ AND
\end{aligned}
$$

$$
\begin{aligned}
x\,|\,y &= x\ OR\ y & Bitwise\ OR
\end{aligned}
$$

$$
\begin{aligned}
x\,\hat{}\,y &= x\ XOR\ y & Bitwise\ XOR
\end{aligned}
$$

Example expression: `foo & 17`

6.7.8 `~`

Type: `word`$n \rightarrow$ `word`n
Accepts flags: N/A
With `t` flag it sets: N/A
Description:

$$
\tilde{}x = NOT\ x \quad Bitwise\ complement
$$

Example expression: `~ 17`

6.7.9 << and >>

Type: word$n$ $\times$ word$n$ $\rightarrow$ word$n$
Accepts flags: `u`
With `t` flag it sets: N/A
Description:

x<<n $\quad Left\ shift\ n\ bits\ logically$

x>>n $\quad Right\ shift\ n\ bits\ logically$
x>>$_u n$ $\quad Right\ shift\ n\ bits\ arithmeticaly$

Example expression: `foo << 17`

6.8 `float` operators and primitives

The individual operators will just have a short description, for a more through discussion on the different operators consult, LIA-1 or IEC559 as appropriate.

The representation used for floating point numbers is: it is either zero or

$X = \pm g * r^e = \pm 0.f_1 f_2 \ldots f_p * r^e$

where $0.f_1 f_2 \ldots f_p$ is the p-digit $fraction$ g (represented in base, or radix, r) and e is the exponent.

The exponent e is an integer in [$emin$, $emax$]. The fraction digits are integers in [0, $r - 1$]. If the floating point number is $normalized$, f_1 is not zero, and hence the minimum value of the fraction g is $1/r$ and the maximum value is $1 - r^{-p}$.

This description gives rise to five parameters that completely characterize the values of a floating point type and they are available as `word` valued constants in `C--` :

$ParameterName$	$Specifies$
`Sys.float`n`.Radix`	$base\ (r)$
`Sys.float`n`.Precision`	$number\ of\ radix\ digits\ provided\ (p)$
`Sys.float`n`.ExpMin`	$smallest\ exponent\ value\ (emin)$
`Sys.float`n`.ExpMax`	$largest\ exponent\ value\ (emax)$
`Sys.float`n`.Denorm`	$1\ if\ type\ has\ denormalized\ values,\ 0\ if\ not$
`Sys.float`n`.IEC559`	$1\ if\ type\ conforms\ to\ IEC559,\ 0\ if\ not$

If `Sys.float`n`.IEC559` is equal to 1, most floating point operators support the four different rounding modes defined by IEC559. This is accomplished by adding a flag signifying which rounding mode is desired. The different flags and their rounding modes are:

$Flag$	$Round\ to$
	$Nearest$
`z`	$Zero$
`p`	$Positive\ Infinity$
`n`	$Negative\ Infinity$

A few definitions that we will use for explaining the individual operators:

$$
\begin{array}{llll}
r & \in & Z & The\ radix\ of\ F \\
p & \in & Z & The\ precision\ of\ F \\
emin & \in & Z & The\ smallest\ exponent\ of\ F \\
emax & \in & Z & The\ largest\ exponent\ of\ F \\
denorm & \in & Boolean & Whether\ F\ contains\ denormalized\ values
\end{array}
$$

$$
\begin{array}{lll}
F_N & = & \{0, \pm i * r^{e-p} \mid i, e \in Z, r^{p-1} \leq i \leq r^p - 1, emin \leq e \leq emax\} \\
F_D & = & \{\pm i * r^{e-p} \mid i, e \in Z, 1 \leq i \leq r^{p-1} - 1, e = emin\}
\end{array}
$$

$$
\begin{array}{llll}
F & = & F_N \cup F_D & if\ denorm = True \\
F & = & F_N & if\ denorm = False
\end{array}
$$

$$
\begin{array}{lllll}
fmax & = & max\{z \in F \mid z > 0\} & = (1 - r^{-p}) * r^{emax} & \\
fmin_N & \in & min\{z \in F_N \mid z > 0\} & = r^{emin-1} & \\
fmin_D & \in & min\{z \in F_D \mid z > 0\} & = r^{emin-p} & \\
fmin & \in & min\{z \in F \mid z > 0\} & = fmin_D & if\ denorm = True \\
 & & & = fmin_N & if\ denorm = False \\
epsilon & = & r^{1-p} & &
\end{array}
$$

6.8.1 `+f`, `-f`, and `*f`

Type: `float`n $\times$ `float`n $\rightarrow$ `float`n
Accepts flags: `t`, `z`, `n`, and `p`
With `t` flag it sets: `Sys.FloatingOverflow` or `Sys.Underflow`
Supported rounding modes (if IEC559): all
Description:

The usual basic arithmetic operators with the proper rounding and trapping notification.

Example expression: `foo +f 17.0`

6.8.2 `/f`

Type: `float`n $\times$ `float`n $\rightarrow$ `float`n
Accepts flags: `t`, `z`, `n`, and `p`
With `t` flag it sets: `Sys.FloatingOverflow` or `Sys.Underflow` or `Sys.Undefined`
Supported rounding modes (if IEC559): all
Description:

The usual basic arithmetic division with the proper rounding and trapping notification.

Example expression: `foo /f 17.0`

6.8.3 `signf`

Type: `float`n $\rightarrow$ `word`m
Accepts flags: N/A
With `t` flag it sets: N/A
Supported rounding modes (if IEC559): N/A

Description:

$$\begin{array}{rcll} signf(x) & = & 1 & if\ x \geq 0.0 \\ & = & 0 & if\ x = 0.0 \\ & = & -1 & if\ x \leq 0.0 \end{array}$$

Example expression: `signf(-12.450)`

6.8.4 `negf` and `absf`

Type: `float`n $\rightarrow$ `float`n
Accepts flags: N/A
With `t` flag it sets: N/A
Supported rounding modes (if IEC559): N/A
Description:

$$\begin{array}{rcll} negf(x) & = & -x & \\ \\ absf(x) & = & |x| & \\ \\ signf(x) & = & 1.0 & if\ x \geq 0.0 \\ & = & 0.0 & if\ x = 0.0 \\ & = & -1.0 & if\ x \leq 0.0 \end{array}$$

Example expression: `negf(foo)`

6.8.5 `exponentf`

Type: `float`n $\rightarrow$ `word`n
Accepts flags: `t`
With `t` flag it sets: `Sys.Undefined`
Supported rounding modes (if IEC559): N/A
Description:

$$\begin{array}{rll} exponentf(x) & = \lfloor log_r|x| \rfloor + 1 & if\ x \neq 0.0 \\ & = \texttt{Sys.Undefined} & if\ x = 0.0 \end{array}$$

Example expression: `exponentf(foo)`

6.8.6 `fractionf`

Type: `float`n $\rightarrow$ `float`n
Accepts flags: N/A
With `t` flag it sets: N/A
Supported rounding modes (if IEC559): N/A
Description:

$$\begin{array}{rll} fractionf(x) & = x/r^{\texttt{exponentf}(x)} & if\ x \neq 0.0 \\ & = 0 & if\ x = 0.0 \end{array}$$

Example expression: `fractionf(foo)`

6.8.7 `scalef`

Type: `float`n $\times$ `word`n $\rightarrow$ `float`n
Accepts flags: `t`, `z`, `n`, and `p`
With `t` flag it sets: `Sys.FloatingOverflow` or `Sys.Underflow`
Supported rounding modes (if IEC559): all
Description:

Scales its argument by an integer power of the radix.

Example expression: `scalef(17.0, 3)`

6.8.8 `succf` and `predf`

Type: `float`n $\rightarrow$ `float`n
Accepts flags: `t`
With `t` flag it sets: `Sys.FloatingOverflow`
Supported rounding modes (if IEC559): N/A
Description:

$$\begin{aligned} succf(x) &= min\{z \in F \mid z > x\} && if\ x \neq fmax \\ &= \texttt{Sys.FloatingOverflow} && if\ x = fmax \end{aligned}$$

$$\begin{aligned} predf(x) &= max\{z \in F \mid z < x\} && if\ x \neq -fmax \\ &= \texttt{Sys.FloatingOverflow} && if\ x = -fmax \end{aligned}$$

Example expression: `succf(17.0)`

6.8.9 `ulpf`

Type: `float`n $\rightarrow$ `float`n
Accepts flags: `t`
With `t` flag it sets: `Sys.Underflow` or `Sys.Undefined`
Supported rounding modes (if IEC559): N/A
Description:

$$\begin{aligned} ulpf(x) &= r^{e_F(x)-p} && if\ x \neq 0\ and\ r^{e_F(x)-p} \in F \\ &= \texttt{Sys.Underflow} && if\ x \neq 0\ and\ r^{e_F(x)-p} \notin F \\ &= \texttt{Sys.Undefined} && if\ x = 0 \end{aligned}$$

Example expression: `ulpf(17.0)`

6.8.10 `truncf`

Type: `float`n $\times$ `word`n $\rightarrow$ `float`n
Accepts flags: N/A
With `t` flag it sets: N/A
Supported rounding modes (if IEC559): N/A
Description:

$$
\begin{aligned}
truncf(x,n) &= \lfloor x/r^{e_F(x)-n} \rfloor * r^{e_F(x)-n} && if\ x \geq 0 \\
&= -\texttt{truncf}(-x,n) && if\ x < 0
\end{aligned}
$$

Example expression: `truncf(17.0, 3)`

6.8.11 `roundf`

Type: $\texttt{float}n \times \texttt{word}n \rightarrow \texttt{float}n$
Accepts flags: `t`, `z`, `n`, and `p`
With `t` flag it sets: `Sys.FloatingOverflow`
Supported rounding modes (if IEC559): all
Description:

$$
\begin{aligned}
roundf(x,n) &= rn_F(x,n) && if\ |rn_F(x,n)| \leq fmax \\
&= \texttt{Sys.FloatingOverflow} && if\ |rn_F(x,n)| > fmax
\end{aligned}
$$

Example expression: `roundf(17.0, 3)`

6.8.12 `intpartf` and `fractpartf`

Type: $\texttt{float}n \rightarrow \texttt{float}n$
Accepts flags: N/A
With `t` flag it sets: N/A
Supported rounding modes (if IEC559): N/A
Description:

$$
intpartf(x) = \texttt{signf}(x) * \lfloor |x| \rfloor
$$

$$
fractpartf(x) = x - \texttt{intpartf}(x)
$$

Example expression: `intpartf(17.0, 3)`

7 Further Work

This is the TO DO list. It lists all the open issues and the stuff that remains to be added to this manual.

DATE: Mon Apr 20 11:58:56 BST 1998

Expressions and constants

- Type conversion (not casting, but conversion):
  float → word
  word → float
- Casting float → word.
- Add w/ carry operator missing.
- How to define symbolic name constants.
- Be more specific about number constants and their syntax. Saying that they're like C's is not enough.
- Regarding pointers.

  Pointer arithmetic, e.g.: `float4[ptr + i] = float4[ptr];`

  If natural pointer size is 8 then ptr has type word8; adding integer offset register i (say, of type word4), if numbers have word4 type then expression `ptr+i` needs a explicit casting: `ptr + word8(i)`. More architecture-dependancy in the code. (See "Preprocessing")

  Also, `C--` has only one pointer type `word`n, where n is the architecture's natural pointer size. Could that cause troubles for architectures that offer two or more pointer sizes? Maybe not because we could just get the bigger (less restrictive) size, but I am not sure if that makes sense.
- Easy to tell expression type.

Preprocessing

C's preprocessing directives can be used in `C--` (it would be easy to do, the `C--` compiler just passes the C preprocessor to the `C--` program before compiling it). It would help for offset-calculation expressions in architectures where the type of an integer is say word4 and the type of a label is word8, and lots of typecast have to be done (since casting is not automatically done), for offsets, in expressions like:

```
        word4[label+ word8(3)] = 43;
```

where the integer is typecast, we could have:

```
#ifdef ... /* pointer size == integer size == 8*/
#then
  #define CAST(x) (x)
#else      /* pointer size != integer size == 4 */
  #define CAST(x) word8((x))

        word4[label+CAST(3)] = 43;
```

to avoid to modify the `C--` code anytime we want to port the code. For more complicated expressions rewriting will be tedious.

Talk about preprocessing in the manual.

Data layout directives

Allow expressions in data declarations to specify the number of elements. These expressions should have values known at compile-time. That is, instead of `type[sconst]...`, it should be: `type[expr]...` in Figure 1.

Procedures and statements

- Local declarations. Why the block as an statement?
  The manual says that no nested scoping is possible in NESTED SCOPING section.
- Stack directive.
- Which calling conventions are there? Now: C, Pascal.
- Have optional list in return, e.g., instead of `return ();`, just `return;`?
- `switch`: could it take also expressions as alternatives? Right now each alternative must be a list of constants, but the former is not difficult to implement, although it is inefficient.

  Explain better how the optional range improves compiler performance in the switch statement.
- Static data in procedures: global or local scope? Indicate in manual.

  Static data declarations in procedures. The following has been removed from Section 5:

  When declaring static data in a procedure, the data has to be declared between the name plus formal argument list, and the procedure body. In memory, the data will be placed immediately before the procedure's entry point, so that the procedure name points to the memory after the data. In the following example, `word4[foo - 4]` has the value `4711`.

  ```
  foo()
  data { word4{31415, 4711}; }
  {
    word4 x;
    x = x + 1;
    return(x);
  }
  ```