Low Level Layered Language

L-LANGUAGE

(Draft 1a)

Robert L. Walton

April 26, 2023

Table of Contents

1 Introduction

This document describes **L-Language**, the Layered Language System Low Level Language.

The L-Language is a system programming language built on the following two main ideas:

Type Checking Segregation Hypothesis A strongly typed-checked general-purpose computer-efficient language is impossible. What is possible is to segregate non-type-checkable code into small inline library functions and into macro functions, with code that uses these functions being strongly type-checked.

Fully Capable Macro Sublanguage Hypothesis It is better for a programming language to have a builtin macro language that is a general purpose interpreted language than it is for the programming language to build into itself many more limited and specialized type declaration and flow control features.

The author of this document does not plan to implement the L-Language. However, a parser for the L-Language is being built on top of the 'layered' system for lexical analysis and parsing, and is being used to debug the design of the 'layered' system.

2 Overview

A typical L-Language statement is:

```
int X = Y - C#"0"
```

This allocates a new variable `X` of type `int` and sets its value to the value of the variable `Y` minus the constant `C#"0"` (which is the character code of the character `0`). The 'variable' `X` is readable, but after it is initialized it is not writable.

The following is another example:

```
av *READ-WRITE* uns8 @bp @= local[81]
av uns8 @cp = "Hello!"
int i = 0
while i < @cp.upper:
    bp[i] = cp[i]
    next i = i + 1
bp[cp.upper] = 0
```

Here '`local[81]`' creates an aligned vector of 81 `uns8` (8-bit unsigned) numbers in the current function frame and returns an aligned vector pointer, or `av`, to the vector, marking the vector elements as `*READ-WRITE*`. `"Hello!"` is a constant vector of `uns8` numbers and is similar except that it marks the vector elements `co`, for 'constant', which is the implied default qualifier for `@cp`, and therefore is not explicitly given. Vector pointers can be used with indices to reference elements of their vectors, and have `upper` and `lower` bounds on

these indices. Here the `lower` bounds are their defaults, which are `0`.

Here `@bp` is a variable whose name begins with '`@`' and whose value is therefore a pointer. Such a variable has an associated indirect variable `bp` whose name is missing the initial '`@`'. The expression `@bp[i]` designates a pointer to the `i`+1'st element of the vector pointed at by `@bp`, but the expression `bp[i]` designates the value of the element. Similarly for `@cp[i]` and `cp[i]`.[1]

The qualifier `*READ-WRITE*` says that a value can be read or written, the default qualifier `co`, or 'constant', says a value can be read but will never be written no matter what, the qualifier `ro`, or read-only, says that the value can be read but cannot be written using the variable name given, though it might be written by some other piece of code that accesses the value under another name. The qualifier `*WRITE-ONLY*` says that the value cannot be read but can be written using the variable name given, but might be read by some other piece of code or other device.

There are also three qualifiers that specify the lifetime of the target of a pointer: `*LOCAL*`, `*GLOBAL*`, and `*HEAP*`. The default is `*LOCAL*` which means, roughly, that the pointer is valid until the code block in which the pointer was first calculated is no longer executing. Typically pointer variables are default-`*LOCAL*`, as in the example. The `"Hello!"` constant in the example is a `*GLOBAL*` pointer, meaning that the pointer is valid during the entire program execution, but such pointers are implicitly convertable to `*LOCAL*` pointers, as in the example. `*HEAP*` pointers point at garbage collectable values and obey special rules that we shall not discuss here.

Variables in function frames and module memory have names, like `X`, `Y`, and `Z`, and values that are constants. These values most frequently have a size equal to the natural word size of the computer (typically 32 or 64 bits), or several times that size: `intd` is a two word (double) integer and `intq` is a four word (quad) integer. Although the value of a variable is constant, the value may point at a memory location that is read-write.

An aligned vector pointer `av` is a quad integer (`intq`) containing:

- A 'base pointer' `int` holding the byte address of an `int` in memory that contains the 'base (byte) address' of the vector. Note that the `av` value does not contain the base address, but contains instead this pointer to where the base address is stored in memory. This scheme allows the base address to be change without changing the `av` value.
- An 'offset' `int` that is added to the base address to form the byte address of the vector element that has index `0` in the vector (this element does not exist if `0` is not an allowed index).
- A 'lower bound' `int` which is the minimum allowed value of the index `int`.

[1] '`@`' is analogous to C++ '`&`' used in a variable declaration, but here '`@`' can be used with different types of pointers, can be used without restrictions for structure members, and can be used with mutable pointers.

- An 'upper bound' `int` which is the maximum allowed value of the index `int` plus 1.

There are other types of pointer. An `fv`, or 'field vector', is like an `av` aligned vector except that the offset `int` has a bit address in its high order part and a field size in bits in its low order part. The `ap` and `fp` types are similar but do not have the bounds and cannot be indexed. Lastly there is the direct pointer, `dp`, that is just a single `int` containing a byte address; this is most useful for calling C language functions. New pointer types may be defined by the user.

Variables whose names begin with '`@`' take pointer values, and the variable's name with the initial '`@`' removed is called the associated target variable and names the value pointed at. Thus `@V` is a pointer valued variable and `V` is the value `@V` points at. For example:

```
int X = 5
ap *READ-WRITE* int @Y   // `@= local' is implied
Y = X + 2                // Now Y == 7
Y = Y - 4                // Now Y == 3
X = X + 1                // Illegal!  X is co
ap ro int @Z = @Y        // Copies pointer value @Y.
                         // Pointer conversion from *READ-WRITE*
                         // to ro is legal.
```

Here the implied '`@= local`' allocates an `int` to the current function frame, zeros it, and returns an '`ap *READ-WRITE* int`' pointer to its location.

Instead of making a variable point at a `*READ-WRITE*` location you can update the constant variable using the `next` construct:

```
int X = 5
int Y = X + 2           // Now Y == 7; Y is co
next Y = Y - 4          // Now Y == 3; Y is co
Y = Y + 1               // Illegal!  Y is co
```

Here '`next Y`' is a new variable, distinct from `Y`, but with the same type, pointer type, qualifiers, and name '`Y`', which hides the previous variable of the same name. The advantage of doing this is that it makes compilation more efficient by keeping variables constant (i.e., `co`), and it improves debuggability by retaining the different values of the variable for inspection by a debugger.

Loops use the '`next ...`' construct. For example:

```
// Compute sum of 4, 5, and 6.
//
int sum = 0
int i = 4
next sum, next i = while i <= 6:
    next sum = sum + i
```

```
        next i = i + 1
```

which is semantically equal to:

```
int sum = 0
int i = 4
next sum, next i:
    next sum = sum + i
    next i = i + 1
next sum, next i:
    next sum = sum + i
    next i = i + 1
next sum, next i:
    next sum = sum + i
    next i = i + 1
```

The '`next sum`' and '`next i`' before the '`:`', which are the output variables for the block of code containing the two '`+`' statements, can also be implied as they appear as output variables of the '`+`' statements, so the above loop can be written as:

```
int sum = 0
int i = 4
while i <= 6:
    next sum = sum + i
    next i = i + 1
```

L-Language has a full set of number types: `int8`, `uns8`, `int16`, `uns16`, `flt16`, ..., `int128`, `uns128`, `flt128`; for signed integer, unsigned integer, and floating point respectively. The types `int`, `uns`, `flt` are just these types for the target machine word size. The types `intd`, `intq`, `unsd`, `unsq` are just integer types for twice (double) or four times (quad) the target machine word size. The `bool` type is a single bit interpreted as true if 1 and false if 0: it is in essence a 1-bit unsigned integer, but it is not considered to be a number type.

User defined types have values that consist of a sequence of bytes containing fields. Fields in turn can contain subfields. An example is:

```
type my type:
              uns32                    // Container for:
    [31-24]   uns8 op code             //   Operation
    [31]      bool has constant        //   Format indicator
    [23-0]    int constant             //   Constant
    [23-16]   uns8 src1                //   Source Register
    [15-8]    uns8 src2                //   Source Register
    [7-0]     uns8 des                 //   Destination Register

. . . . . .
```

```
my type X:
    X.op code = 5       // This is an initialization block
    X.src1 = 2          // for X in which X is write-only.
    X.src2 = 3
    X.des = 3
uns op = X.op code      // Now op == 5
int d = X.des           // Now d == 3
ap *READ-WRITE* my type @Y    // `@= local' is implied
Y.op code = 129
fp *READ-WRITE* int @C = @Y.constant
ap *READ-WRITE* uns8 @OP = @Y.op code
next op = OP            // Now op == 129
bool B = Y.has constant // Now B == 1
C = -1234               // Now Y.constant = -1234
```

In this example there is one field in a `my type` value, an unlabeled `uns32` integer. Inside this unlabeled field there are 6 subfields, the first of which is an `uns8` integer occupying the highest order 8 bits of the unlabeled field, bits 31-24, where bits are numbered 0, 1, 2, ... from low to high order. The second subfield is a 1-bit `bool` value that occupies the high order bit, bit 31, of the unlabeled field. Note that subfields can overlap.

Defined type values are aligned on byte boundaries when they are stored in memory. Therefore the '`op code`' subfield is on a byte boundary, and the location of `OP` is an `ap` aligned pointer. Although the `constant` subfield is on a byte boundary, it is shorter than an `int`, and therefore the location of `C` must be an `fp` field pointer. If '`op code`' were in bits 30-23 instead of 31-24, it would not be on a byte boundary and the location of `OP` would also have to be an `fp` field pointer.

Note that '`Y.op code`' is a `*READ-WRITE* uns8` while '`@Y.op code`' is a `co` pointer to a `*READ-WRITE* uns8`.

Names in L-Language can have multiple lexemes, as in the type name '`my type`', the subfield name '`op code`', and what L-Language calls the associated member name '`.op code`' which can be used to access the field.

Another example is:

```
type my type:
    pack
    uns8    kind            // Object Kind
    [7] bool animal         // True if Animal
    [6] bool vegetable      // True if Vegetable
    flt64   weight          // Object Weight
    align   64
    *LABEL*   extension
```

```
    ***                         // Enables type extension.

type my type:
    *ORIGIN*  extension
    flt64   height              // Object Height
    flt64   width               // Object Width
    ***                         // Enables type extension.

type my type:
    *ORIGIN*  extension
    flt64   volume              // Object Volume; overlays height.
                                // No further type extension allowed.

type your type:
    *INCLUDE* my type // Copy sub-declarations of my type
    *ORIGIN* *SIZE*   // *SIZE* is max origin seen so far.
    av uns8 @name     // Aligned vector pointer to name
                      // character string

. . . . . .

my type X:
    X.kind = BOX
    X.weight = 55
    X.height = 1023
    X.width = 572

your type Y:
    Y.kind = BEER
    Y.weight = 0.45
    Y.volume = 48
    Y.@name = "John Doe's Lager"
```

Here `my type` and `your type` are defined by statements called *type-declarations*. Each of these *type-declarations* contains a sequence of sub-declarations, e.g., for `my type` the first two sub-declarations are '`pack`' and '`uns8 kind`'. There is a current offset in bits that starts at `0` and is updated by each sub-declaration. A sub-declaration such as '`uns8 kind`' allocates a field (i.e., `kind`) at the current offset and adds the size of the field to the current offset.

In the example the fields are `kind`, `weight`, `height`, etc. Fields can be packed or aligned; aligned is the default. An aligned number has an offset that is a multiple of the length of the number. Here fields are initially packed so that since `kind` has offset 0 bytes and size 1 byte, `weight` has offset 1 byte. Subfields `animal` and `vegetable` are 1-bit values inside `kind`.

The `align 64` sub-declaration moves the current offset forward to a 64-bit boundary and causes fields beyond it to be aligned and not packed. A number is aligned if its offset is a multiple of its length. Alignments must be powers of two. A defined type has an alignment equal to the least common multiple (in this case just the largest) of the alignments of its aligned fields.

A `*LABEL*` is like a zero length field that has no value and is used to associate an origin-label with the current offset. Here `extension` has the offset value of 128 bits (16 bytes). The `*ORIGIN*` sub-declaration resets the current offset to the offset of a given origin-label, or to `*SIZE*`, which denotes the current size of the type in bits (which may increase with later sub-declarations).

The '`***`' sub-declaration at the end of a *type-declaration* defining a user defined type indicates that the definition may be continued by a later *type-declaration*, as is done for `my type` above. The sub-declarations of the later *type-declaration* are simply appended to those of previous *type-declarations*.

The `*INCLUDE*` sub-declaration copies all the sub-declarations from another user defined type. If the user defined type is defined by multiple *type-declarations*, only sub-declarations from the *type-declarations* in the current scope (see 12 [p91]) are copied.

Defined types can be extended (as per the example), and fields can overlay each other. A defined type value has a size in bytes just large enough to accommodate all its fields. If a defined type has multiple *type-declarations*, this size may not be known until load time.

Values of `const` type are compile-time values, and are not available at run-time. Number constants consisting of digits and optional signs, decimal points, and exponents, are converted to IEEE 64-bit floating point values, as are special lexemes such as `inf`, `+inf`, `-inf`, and `nan`. Other number constants represent rationals with unbounded integral numerators and denominators; for example, `D#"1/3"` represents the precise rational one-third. Number constants can be converted to run-time numbers during compilation. However it is a compile error if the result will not fit into the runtime number. This happens, for example, if either `1.1` or `1e20` is converted to an `int32`.

Quoted strings denote string `const` values that can be converted during compilation to run-time vectors with `co` unsigned integer elements that encode the string in UTF-8, UTF-16, or UTF-32.

Lastly there are map `const` values that can hold lists and dictionaries. Map values can be mutable at compile-time, but cannot be converted to run-time values.

Expressions, statements, and functions that use only `const` values execute at compile-time and can be used to compute compile-time `const` values including maps that represent code.

By default, functions in L-Language are inline. For example,

```
function int r = max ( int x, int y ):
    if x < y:
```

```
        r = y
    else:
        r = x

int x = ...
int y = ...
int z = max ( x, y )
```

L-Language does not support implicit conversions of run-time function results[2], but does support implicit conversion of variables[3] and constants. Any number constant or rational constant may be converted implicitly to any run-time number type as long as the constant value can be stored exactly in a variable of the run-time type or the run-time type is floating point (in which case there may be loss of precision or conversion to an infinity). Any number type variable may be implicitly converted at run-time to a type that will hold all the possible values of the variable, or to any floating point type (in which case the run-time conversion result may be less precise or an infinity).

Language expressions have **target types**. For a function call, the function result cannot be implicitly converted to the target type. However a function call that returns a `const` result is replaced by its value at compile time, and this value can be implicitly converted to the target type. Also, variables may be implicitly converted to their target type.

Builtin operators, such as '+', have operands of the same type as their result.

An example of all this is:

```
flt w = 1.1                 // flt is target type of 1.1
int x = 123                 // int is target type of 123
int y = 2e5                 // int is target type of 2e5
int z = 1e100               // illegal; int is target type of 1e100
                            // which is too large to fit
flt r1 = x                  // implicit conversion is legal as x is a
                            // variable name and not a function call
int r2 = 5 + max ( x, y )   // int is target type of +, 5, max, x, y
int r3 = max ( y, w )       // illegal; int is target type of w and
                            // flt cannot implicitly convert to int
int r4 = max ( x, 123 )     // int is target type of max, x, 123
int r5 = max ( x, 123.4 )   // illegal; int is target type of 123.4
                            // which cannot be stored in a int
const c1 = 100
const c2 = 1000
int r6 = max ( x, c1 + c2 ) // legal; c1 + c2 is replaced by 1100
                            // which has int target type
```

[2]In this matter L-Language follows ADA.
[3]Unlike ADA.

For each type `T`, a function:

```
function T r = T ( T v );
    r = v
```

is provided. Such an 'identity' function might seem useless, but in fact it can be used in an expression '`T(`*e*`)`' to force the target type of *e* to be `T`. As an example consider the following, where there are builtin functions:

```
function N r = ( N v1 ) "+" ( N v2 )
function bool r = ( N v1 ) "<" ( N v2 )
```

for every number type `N`:

```
int x = ...
bool b1 = ( x + 3 < 5 )
    // Illegal: any target type N to which int is implicitly
    // convertible can be used as the target type of "+" and "<" so
    // this is ambiguous.
bool b2 = ( flt ( x ) + 3 < 5 )
    // Legal: type of flt ( x ) can only be flt, so that is the type
    // of "+" and therefore "<".  Note that flt forces implicit
    // conversion of x from int to flt.
bool b3 = ( x + 3 < int ( 5 ) )
    // Legal: type of int ( 5 ) can only be int, so that is the type
    // of "<" which becomes the target type of "+".  Note that int
    // forces implicit conversion of const to int.
```

Integer arithmetic ignores overflows (as in C and C++); for example, if integer + produces a value too large for its target, the result is undefined and may or may not cause program termination. There are similar explicit conversion functions from a floating point type `F` to any integer type `I` with prototypes:

```
function I r = floor ( F v )
function I r = ceiling ( F v )
function I r = truncate ( F v )
function I r = round ( F v )
```

that take a floating point value and round it toward negative infinity (`floor`), positive infinity (`ceiling`), zero (`truncate`), or nearest (`round`). If the value is too large to be stored in `I`, the result is undefined and may or may not cause program termination.

The `ro` qualifier name can be used by itself as an explicit conversion function name to convert `co` or `*READ-WRITE*` qualifiers to `ro` qualifiers. This can handle cases where a function returns a `*READ-WRITE*` pointer to set an `ro` pointer.

```
function ap *READ-WRITE* int r = foo ( ... )
ap ro int @p = ro ( foo ( ... ) )
```

Although function results cannot be implicitly converted, variables can be, and implicit conversion of `co` or `*READ-WRITE*` to `ro` is defined for pointer-valued variables.

In addition to using target types to select which overloaded function is being called, the types of implicitly convertible arguments, that is, variables and constants, can be used. Essentually function definitions that require fewer implicit conversions are preferred. For example,

```
int x = ...
bool b = ( x < 6 )
    // There is a separate "<" operator for every number type,
    // so x and 6 could both be converted to, say, flt.  But this
    // is two implicit conversions, whereas using the "<" operator
    // for int requires only one conversion, so this is used.
bool b = ( x < 6.5 )
    // Again int is chosen for "<", but this time 6.5 produces a
    // compiler error when converted to an int.  Inconveribility
    // of particular constant values is NOT considered in selecting
    // among overloaded function definitions.
```

It is possible to define compile-time functions:

```
function const r = max ( const x, const y ):
    if x < y: r = y
    else:     r = x

const x = 2e5
const y = 3e6
const z = 9e4
const w = max ( x, max( y, z ) )
```

Such functions are not available at run-time, and are not really inline, as there is no distinction between inline and out-of-line for compile-time functions.

Inline function definitions may make use of type wildcards. A name that is a single word beginning with `T$` is a type wildcard that denotes an arbitrary type. Thus the example:

```
function T$r r = max ( T$r x, T$r y ):
    if x < y: r = y
    else:     r = x

const x = 2e5
int y = 27e4
int z = max ( x, y )      // T$r is int, x converts to int.
```

```
const w = 34e4
const v = max ( x, w )    // T$r is const, all values are const.
```

A wildcard type of a result variable gets its value from the target type of a function call. A wildcard type of an argument can get its value from the argument type, but only if the later is a variable, or more generally, a reference expression (e.g., `x[i]`). Typing is mostly done top-down using target types, but reference expressions get types bottom up from the variable explicitly named in the reference expression.

Pointer types can be wildcards which must have names that are single words beginning with `P$`. A list of qualifiers can also be a wild card named by a single word beginning with `Q$`. An example is:

```
function uns r = strlen ( P$s Q$s uns8 @s ):
    dp ro uns8 @sdp = *UNCHECKED* ( @s )
    r = call "strlen" ( @sdp )
```

which converts the pointer of type `P$s` to a pointer of type `dp` (direct pointer) and calls the 'foreign' C programming language subroutine `strlen` with the direct pointer. The `*UNCHECKED*` function is needed to produce a direct pointer from other pointer types, though this cannot be done for some pointer types (e.g., field pointer types).

A pointer type has two places where a qualifier may appear, as in

```
type my control block:
    co ap *GLOBAL* uns32 @cr
    ....
```

in which `@cr` is a constant pointing at a global `uns32` location `cr`.

Pointer types cannot be cascaded, but there is a work-around using defined types:

```
ap av flt64 @p = .....        // Illegal!

struct my pointer
    av flt64 @q

. . . . . .

ap my pointer @p = ...        // OK
flt64 v = p.q[0]              // OK
```

Any inline function can create new code that is inserted after the statement containing a call to the function. The code is expressed as a `const` map value in the format output by the code parser. As a simple example, if the inline function contains:

```
const T = `my number'     // == {"my", "number"}
const V = `my variable'   // == {"my", "variable"}
*INCLUDE* (T, V):
```

```
T V = x + y
T z = max ( V, 1000 )
```

then a statement calling the inline function will be followed by the code:

```
my number my variable = x + y
my number z = max ( my variable, 1000 )
```

which the parser renders as the `const` map value:

```
{ { { "my", "number", "my", "variable" }, "=",
    { { "x" }, "+", { "y" } } },
  { { "my", "number", "z" }, "=",
    { "max", { { "my", "variable" }, { "1000" },
               ".initiator" => "(",
               ".separator" => ",",
               ".terminator" => ")" } } } }
```

Code to be inserted can also be computed directly as a `const` map value without using '`*INCLUDE*`' statements.

3 Lexemes

An L-Language source file is a sequence of bytes that is a UTF-8 encoding of a sequence of UNICODE characters. This is scanned into a sequence of **lexemes**.

Unless otherwise specified, the term '**character**' in this document means a 32-bit UNICODE character.

Lexemes are defined in terms of the following character classes:

horizontal-space-character ::= characters in UNICODE category **`Zs`**
(includes *ASCII-single-space*)
| *horizontal-tab-character*

vertical-space-character ::= *line-feed* | *carriage-return*
| *form-feed* | *vertical-tab*

space-character ::= *horizontal-space-character* | *vertical-space-character*

graphic-character ::= characters in UNICODE categories **`L`**, **`M`**, **`N`**, **`P`**, and **`S`**

control-character ::= characters in UNICODE categories **`C`** and **`Z`**

isolated-separating-character ::=
characters in UNICODE categories **`Ps`**, **`Pi`**, **`Pe`**, and **`Pf`**;
includes **`{ ( [ « » ] ) }`**

separating-character ::= **`|`** | *isolated-separating-character*

leading-separator-character ::= **`‘`** | **`¡`** | **`¿`**

trailing-separator-character ::= **`’`** | **`!`** | **`?`** | **`.`** | **`:`** | **`,`** | **`;`**

quoting-character ::= `"`

letter ::= characters in UNICODE category `L`
ASCII-digit ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
digit ::= characters in UNICODE category `Nd` (includes *ASCII-digits*)
lexical-item-character ::= *graphic-character* other than *separating-character* or `"`

Comments may be placed at the ends of lines:

comment ::= `//` *comment-character**

comment-character ::= *graphic-character* | *horizontal-space-character*

Lexemes may be separated by *white-space*, which is a sequence of *space-characters*, but, with some exceptions mentioned just below, is not itself a lexeme:

white-space ::= *space-character*$^+$
horizontal-space ::= *horizontal-space-character*$^+$
vertical-space ::= *vertical-space-character*$^+$

The following is a special virtual lexeme:

indent ::= virtual lexeme inserted just before the first *graphic-character* on a line

Indent lexemes have no characters, but do have an **indent**, which is the indent of the graphic character after the indent lexeme. The **indent** of a character is the number of columns that precede the character in the character's physical line. *Control-characters* other than *horizontal-space-characters* take zero columns, as do characters of classes `Mn` (combining-marks) and `Me` (ending marks). All other characters take one column, except for tabs, that are set every 8 columns. Indent lexemes are used to form logical lines and blocks ($4\,^{p20}$).

One kind of *vertical-space* is given special distinction:

line-break ::= *vertical-space* containing exactly one *line-feed*

This is the *line-break* lexeme.

Non-*indent*, non-*line-break white-space*, such as occurs in the middle of text or code outside comments, is discarded and not treated as a lexeme. Such *white-space* may be used to separate lexemes.

Horizontal-space-characters other than single space are illegal inside *quoted-string* lexemes (defined below). *Vertical-space* that has no *line-feeds* is illegal (see below). *Control-characters* not in *white-space* are illegal. Characters that have no UNICODE category are *unrecognized-characters* and are illegal:

misplaced-horizontal-space-character ::=
horizontal-space-character, other than ASCII-single-space
misplaced-vertical-space-character ::=
vertical-space-character other than *line-feed*

illegal-control-character :::=
control-character, but not a *horizontal-space-character* or
vertical-space-character

unrecognized-character :::=
character with no UNICODE category or
with a category other than `L`, `M`, `N`, `P`, `S`, `C`, or `Z`

Sequences of these characters generate warning messages, but are otherwise like *horizontal-space*:

misplaced-horizontal :::= *misplaced-horizontal-space-character*$^+$
misplaced-vertical :::= *misplaced-vertical-space-character*$^+$
illegal-control :::= *illegal-control-character*$^+$
unrecognized :::= *unrecognized-character*$^+$

Misplaced-horizontal only exists inside a *quoted-string*, but the other three sequences can appear anywhere. When they occur, these sequences generate warning messages, but otherwise they behave like *horizontal-space*. Specifically, outside *quoted-strings* and *comments* these sequences can be used to separate other lexemes, just as *horizontal-space* can be used, whereas inside *quoted-strings* and *comments* these sequences do nothing aside from generating warning messages.

The lexemes in a L-Language program are specified in Figure 1 [p18]. This specification assumes there are no illegal characters in the input; see text above to account for such characters.

The symbol '`:::=`' is used in syntax equations that define lexemes or parts of lexemes whose syntactic elements are character sequences that must not be separated by *white-space*. The symbol '`::=`' is used in syntax equations that define sequences of lexemes that may and sometimes must be separated by *white-space*.

There is a special ***end-of-file*** lexeme that occurs only at the end of a file.

Files are scanned into sequences of lexemes which are then divided into logical lines as per 4 [p20]. After each logical line is formed, *indent*, *comment*, *line-break*, and *end-of-file* lexemes are deleted from the logical line.

A ***special-character-representative*** can consist of a UNICODE character name surrounded by angle brackets. Examples are `<NUL>`, `<LF>`, `<SP>`, `<NBSP>`. There are three other cases: `<Q>` represents the doublequote `"`, `<NL>` (new line) represents a line feed (same as `<LF>`), and `<UUC>` represents the '**unknown UNICODE character**' which in turn is used to represent illegal UTF-8 character encodings.

A *special-character-representative* can also consist of a hexadecimal UNICODE character code, which must begin with a digit. Thus `<0FF>` represents ÿ whereas `<FF>` represents a form feed.

Quoted string lexemes separated by the '`#`' mark are glued together if they are in the same logical line. Thus

lexeme ::= *numeric-word* | *word* | *natural* | *number* | *numeric*
| *mark* | *separator* | *quoted-string*
| *indent* |*line-break* | *comment* | *end-of-file*

strict-separator ::= *isolated-separating-character* | `|`$^+$

leading-separator ::= `‘`$^+$ | `¡`$^+$ | `¿`$^+$

trailing-separator ::= `’`$^+$ | `!`$^+$ | `?`$^+$ | `.`$^+$ | `:`$^+$ | `;` | `,`

separator ::= *strict-separator* | *leading-separator* | *trailing-separator*

quoted-string ::= `"` *character-representative** `"`

character-representative ::= *graphic-character* other than `"`
| *ASCII-single-space-character*
| *special-character-representative*

special-character-representative ::= `<` { *upper-case-letter* | *digit* }$^+$ `>`

lexical-item ::= *lexical-item-character*$^+$ not beginning with `//`

lexical-item ::= *leading-separator** *middle-lexeme*$^?$ *trailing-separator**

middle-lexeme ::= *lexical-item* not beginning with a *leading-separator-character*
or ending with a *trailing-separator-character*

numeric-word ::= *sign*$^?$ `nan` | *sign*$^?$ `inf` [where *letters* are case insensitive]

word ::= *middle-lexeme* that contains a *letter* before any *digit*
and is not a *numeric-word*

natural ::= *decimal-digit*$^+$ not beginning with `0` | `0`
[but lexical type may be changed; see p19]

number ::= *sign*$^?$ *integer-part exponent-part*$^?$ that is not a *natural*
| *sign*$^?$ *integer-part*$^?$ *fraction-part exponent-part*$^?$
[but lexical type may be changed; see p19]

integer-part ::= *decimal-digit*$^+$ ***fraction-part*** ::= `.` *decimal-digit*$^+$

exponent-part ::= *exponent-indicator sign*$^?$ *decimal-digit*$^+$

sign ::= `+` | `-` ***exponent-indicator*** ::= `e` | `E`

numeric ::= *middle-lexeme* that contains a *digit* before any *letter*
and is not a *natural* or *number*

mark ::= *middle-lexeme* not containing a *letter* or a *digit*

indent ::= see p16 *line-break* ::= see p16

comment ::= see p16 *end-of-file* ::= see p17

Figure 1: L Language Program Lexemes

```
"This is a longer sentence" #
    " than we would like."
"And this is a second sentence."
```

is equivalent to

```
"This is a longer sentence than we would like."
"And this is a second sentence."
```

This is useful for breaking long quoted string lexemes across line continuations. But there is an important case where there is not an exact equivalence between the glued and unglued versions. `"<" # "LF" # ">"` is not equivalent to `"<LF>"`. The former is a 4-character quoted string, the characters being `<`, `L`, `F`, and `>`. The latter is a 1-character quoted string, the character being a line feed.

The definition of a **middle-lexeme** is unusual: it is what is left over after removing *leading-separators* and *trailing-separators* from a *lexical-item*. The lexical scan first scans a *lexical-item*, and then removes *leading-separators* and *trailing-separators* from it. Also *trailing-separators* are removed from the end of a *lexical-item* by a right-to-left scan, and not the usual left-to-right scan which is used for everything else. Thus the *lexical-item* '`¿4,987?,,::`' yields the *leading-separator* '`¿`', the *middle-lexeme* '`4,987`', and the four *trailing-separators* '`?`', '`,`' '`,`' and '`::`'.

Words, *numerics*, and *marks* in the same logical line are glued together if the first ends with '`#`' and the second begins with '`#`'. Thus

```
This is a continued-#
    #middle# #-lexeme.
```

is equivalent to

```
This is a continued-middle-lexeme.
```

For compatibility, two consecutive '`#`' marks may be used to glue together two quoted strings, as in

```
"This is a continued-"#
    #"quoted"# #"-string".
```

which is equivalent to

```
"This is a continued-quoted-string".
```

A *numeric-word*, *natural*, or *number* lexeme is a C/C++ constant, and conversely a C/C++ decimal constant not ending in a *decimal-point* or a C/C++ special floating point value (e.g., `+Inf`) is a *numeric-word*, *natural*, or *number* lexeme. All these lexemes are given an IEEE double precision number value after the manner of C/C++, and then their lexical type is changed as follows:

- If the value is not a finite number, the new type is *numeric-word*. For example, this applies to `1e500` which converts to the same value as `+inf`.

- If the value is an integer in the range $[0, 10^{15})$ the new type is *natural*. For example, this applies to `1e3` which converts to the same value as `1000`.
- Otherwise the new type is *number*. For example, this applies to `1e20` or `1.1`.

In contrast, a *numeric*, like `02/28/2022`, represents a character string and in this is like a *word*. The lexeme `2/3` is also a *numeric* and is not used to represent a number; instead the lexeme pair `D# "2/3"` is used to represent a rational number constant (value of `const` type).

4 Logical Lines, Blocks, and Statements

Each non-blank physical line begins with an *indent* lexeme that is followed by a lexeme that cannot be an *indent*, *line-break*, or *end-of-file*.

Lexemes are organized into **logical lines**. A logical line begins immediately after an *indent* lexeme, and the **indent** of the logical line is the indent of this *indent* lexeme (i.e., the indent of the first graphic character of the logical line).

A logical line ends with the next *indent* lexeme whose indent is not greater than the indent of the logical line, or with an *end-of-file*. Thus physical lines with indent greater than that of the current logical line are **continuation lines** for that logical line.

A code file is a sequence of '**top level**' logical lines that are required to have indent `0`.

A logical line may end with a **block** that is itself a sequence of logical lines that have indents greater than the indent of the logical line containing the block. The block is introduced by a '`:`' at the end of a physical line, provided the '`:`' is not inside brackets or quotes (e.g., not inside `( )` or `‘ ’`). If the first *indent* lexeme after the '`:`' has an indent that is not greater than the indent of the logical line containing the '`:`', the block is empty. Otherwise the indent of this *indent* lexeme becomes the **indent** of the block and the indent of all the logical lines in the block. The first logical line of the block starts immediately after this *indent* lexeme. The block ends just before the first logical line with lesser indent than the block indent, or the end of file. More specifically, the last logical line of the block ends with an *indent* whose indent is less than the block indent, or with an *end-of-file*.

Examples are:

```
this is a top level logical line ending with a block:
    this is the first line of the block
    this is the
         second line of the block
    this is the third line of the block:
        this is the first line of a subblock
        this is the second line
                of the subblock:
```

```
            this is the only line of a sub-subblock
        this is the third line of the subblock
    this is the fourth line
            of the block:
        this is the only line of the second subblock
    this is the fifth line of the block
         and it ends with an empty subblock:
this is the second top level
     logical line
```

A warning message is output if two indents that are being compared differ by more than `0` and less than `2` columns, in order to better detect indentation mistakes.

Line-break lexemes are effectively ignored. A sequence of *line-break* lexemes is followed by an *indent* or *end-of-file* which is not ignored. Blank physical lines are represented by sequences of more than one *line-break* lexeme, and are effectively ignored.

A logical line that contains *comments*, but no lexemes other than *comments*, *line-breaks*, *indents* and a possible *end-of-file*, is a '**comment line**'.

It is an error to begin non-comment logical lines with a *comment*. *Comments* can be used freely in the middle of or at the end of any logical line, or at the beginning of a comment line.

It is an error for the first logical line of a file to have an indent that is greater than `0`, the top level indent.

It is an error for a block to be in the middle of a logical line. This means that the first *indent* following the block must have an indent no greater than that of the logical line containing the block.

Examples are:

```
// this is a logical line that is a single comment

// this is a logical line that has two
    // comments

this is a logical line // with a comment
     // and another comment
     with three comments // and a last comment

this is a logical line ending with a block:
     First line of the block
     Second line of the block
// Comment that ends block
// Comment that is in error because
```

```
    it begins a logical line that this continues

this is a logical line with a block:
     First line of the block
     Second line of the block
  but the block is in error because it is before
  this continuation of the logical line that contains
  the block

this is a logical line ending with a block:
        First line of the block
        Second line of the block
  // comments that end the block, but are in error,
  // because they continue the logical line
  // containing the block
```

After a logical line has been formed, any *indent*, *comment*, *line-break*, and *end-of-file* lexemes in the logical line are removed from the logical line. If the result is empty, e.g., the logical line is a comment line, it is discarded. Otherwise the modified logical line becomes a L-Language '***statement***'.

Therefore a file is a sequence of top-level statements.

Since a logical line can end with a block that itself consists of a sequence of logical lines, a statement can end with a block that itself consists of a sequence of statements.

5 Expressions

Expressions are built from operators, such as `+` and `*`, and primaries, such as variable names and function calls.

Operators are characterized by fixity, precedence, and format. The L-Language operators are listed in Figures 2 [p24], 3 [p25], and 4 [p26].

Given this, expressions have the following syntax, where an *P-expression* is an expression all of whose operators that are outside brackets have precedence equal to or greater than P:

expression ::= *L-expression*
P-expression ::= { *(P+1)-expression* | *P-operator)* }$^+$
where no two *(P+1)-expressions* may be adjacent
and the *P-operators* must obey the fixity rules below
P-operator ::= operator of precedence P
(H+1)-expression ::= *primary*
primary ::= *primary-element*$^+$ [see p28]
primary-element ::= *non-operator-lexeme* | *bracketed-subexpression*

where P is any precedence in the range [L-1,H]

Generally a *P-expression* consists of a sequence of *(P+1)-expressions* separated by operators of precedence *P*. Precedence L-1 is reserved for an error operator used to fix up parsing errors, precedence H-1 is reserved for highest precedence prefix operators, and precedence H is reserved for highest precedence postfix operators.

The operators can have any combination of the following **base fixities**:

`initial`	*P-operator* must be the first thing in its *P-expression*.
`final`	*P-operator* must be the last thing in its *P-expression*.
`left`	*P-operator* must be immediately preceded by a *(P+1)-expression* in its *P-expression*.
`right`	*P-operator* must be immediately followed by a *(P+1)-expression* in its *P-expression*.
`afix`	*P-operator* must be after a (not necessarily immediately) preceding *P-operator* in its *P-expression*.

The following **combination fixities** are defined:

`prefix`	`initial` + `right`
`infix`	`left` + `right`
`postfix`	`left` + `final`
`nofix`	none of `initial`, `final`, `left`, or `right`

All of these but `initial` and `prefix` can be combined with `afix`.

Line Level Operators
Must Occur Outside Parentheses and Brackets
At Top Level or Inside {* ... *}

Operator	Meaning	Fixity	Format	Precedence
if	conditional	prefix	conditional	0000
else if				
else	terminating conditional	initial	terminating conditional	
:	conditional completion	afix right	(none)	
subblock	conditional or declaration completion	afix		
	assignment or loop	postfix	postfix	
type	declaration	prefix	declaration	
pointer type				
function				
reference function				
out-of-line function				
is type		afix infix	(none)	
is function				
--->	abbreviate	infix	binary	
=	assignment	left	assignment	1000
+=	increment	infix	binary	
-=	decrement			
***=**	multiply by			
/=	divide by			
|=	include			
&=	mask			
^=	flip			
<<=	shift left			
>>=	shift right			
@=	pointer assignment	infix	binary	1100

Figure 2: L-Language Line Operators

Non-Line Level Operators: Part I
May Occur Inside or Outside Parentheses and Brackets

Operator	Meaning	Fixity	Format	Precedence
`,`	separator	nofix	separator	2000
`loop`	iterator	prefix	unary	3000
`while`				
`until`				
`exactly`			iteration	
`at most`				
`times`	iteration modifier	afix	(none)	
`if`	selector	infix	selector	10000
`else`		infix afix	(none)	
`BUT NOT`	logical and not	infix	binary	11000
`AND`	logical and	infix	n-ary	11100
`OR`	logical or			
`NOT`	logical not	prefix	unary	11200
`==`	is equal	infix	(none)	12000
`!=`	is not equal			
`<`	is less than			
`<=`	is less than or equal			
`>`	is greater than			
`>=`	is greater than or equal			
`+`	addition	infix	sum	13000
`-`	subtraction			
`|`	bitwise or		n-ary	
`&`	bitwise and			
`^`	bitwise xor			
`/`	division	infix	binary	13100
`*`	multiplication		n-ary	13200
`**`	exponentiation		binary	13300
`<<`	left shift			
`>>`	right shift			

Figure 3: L-Language Non-Line Operators

Non-Line Level Operators: Part II
May Occur Inside or Outside Parentheses and Brackets

Operator	Meaning	Fixity	Format	Precedence
`+`	no-op	prefix	unary	H-1
`-`	negation			
`~`	bitwise complement			
`#`	length			
`D#`	decimal rational			
`B#`	binary rational			
`X#`	hexadecimal rational			
`C#`	character rational			

Figure 4: L-Language Non-Line Operators

The operators in Figures 2^{p24}, 3^{p25}, and 4^{p26} have precedences in the range *[L,H]*. Precedence *(L-1)* is reserved for the 'error operator' which is a nofix operator inserted by the parser to 'fix up' parsing errors so parsing can continue.

The first *P-operator* in a *P-expression* determines the *P-expression*'s **format**, which is one of the following, where in describing expression formats we use:

'expression' to mean *P-expression*,
'operator' to mean *P-operator*,
and 'operand' to mean *(P+1)-expression*:

`conditional` The expression must consist of the operator followed by an operand followed by either a `:` operator and an operand or by just a subblock operator (`:` indented paragraph, which can be an operator).

`terminating conditional` The expression must consist of the operator followed by either a `:` operator and an operand or by just a subblock operator.

`postfix` The expression must consist of an operand followed by the operator.

`declaration` The expression must consist of an operator followed by an operand (that may contain `=` and `,`) followed sometimes by a subblock operator.

`binary` The expression must consist of an operand followed by the operator followed by an operand. There must be only one operator in the expression.

`assignment` The expression must consist of an operand followed by the operator followed by an optional operand.

selector	The expression operators must all be either **if** or **else**. The expression must consist of alternating operands and operators and begin and end with an operand. The two possible operators alternate, with **if** first.
separator	All operators in the expression must be identical. There are no other constraints on the expression. An implied empty operand is inserted between two consecutive operators, at the beginning if the expression begins with an operator, and at the end if the expression ends with an operator. Then the operators are deleted from the expression and the expression operator is attached to the expression as its **.separator** attribute.
n-ary	All operators in the expression must be identical. The expression must consist of alternating operands and operators and begin and end with an operand.
unary	The expression must consist of the operator followed by an operand.
sum	The expression operators must all be either **+** or **-**. The expression must consist of alternating operands and operators and begin and end with an operand.

There are a few additional special syntactic rules:

1. Non-line bitwise operators (**|**, **&**, **^**, **<<**, **>>**, and **~**) cannot be mixed with non-line arithmetic operators (**+**, **-**, **/**, *****, and ******) outside parentheses in a subexpression. E.g., '`x + (y * ~ z)`' is illegal but '`x + (y * (~ z))`' is legal.

Full semantics of operators and expressions is described later, but the following examples give an idea of some of this semantics:

`T v = x + y * z`
: Here `T` is the **target type** of the expression '`x + y * z`' and thus must be the result type of the prototype of the '`+`' function, since function results cannot be implicitly converted. Because it is the result type of `+` and arithmetic operators (with a few exceptions) have operands that are of the same type as their result, `T` is also the target type of `x` and `*`, and since it is the target type of `*` it will be the target type of `y` and `z`. Implicit conversions of variables are allowed, so `x`, `y`, and `z` will all be converted to type `T` before any computation is done.

`T v = x if y else z`
: If `y` is not a `const`, it is evaluated with target type `bool`. If that value is `true`, `x` is evaluated and returned; otherwise `z` is evaluated and returned. Both `x` and `z`, have target type `T`.

  However if `y` is a `const` value, the right-side of the statement is replaced by `x` or `z`, whichever is discarded is also not compiled, and if it would be in error were it compiled, the error is not detected (unless it is a parsing error).

`bool v = x AND y`
If either operand evaluates to `FALSE`, compile-time evaluation stops and the statement is replaced by '`bool v = FALSE`'.

Otherwise same as '`bool v = y if x else FALSE`'.

The `const` values `TRUE` and `FALSE` are implicitly convertible to run-time `bool true` (`1`) and `false` (`0`), respectively.

`x < y < z`
This is logically equivalent to '`x < y` AND `y < z`', except that `y` is evaluated at most once.

If any comparison evaluates to `FALSE` at compile-time, compile-time evaluation stops and the entire expression is replaced by '`FALSE`'.

If a single comparison evaluates to `TRUE` at compile-time, that comparison is removed from the `AND`-containing version of the expression.

If run-time evaluation is necessary, some operands need to be evaluated at run-time, and a target type `T` needs to be found for these operands. A single target type `T` must work as the target type of all the run-time operands. If several target types `T` work and one is the type of a reference expression operand, that is used. Otherwise if several work, it is a compile error.

Changing the expression to '`T(x) < y < z`' will force the target type to be `T` if that works, or a compile error otherwise.

`v[x+5] = y`
The target type of subscript expressions such as '`x + 5`' is `int`.

`~ x`
The '`~`' operator evaluates on signed integers as if they were represented in two's complement by binary values of unbounded size, and similarly for other bitwise operators.

`x ** y`
Requires that `y` be a `const` non-negative integer; `x ** 0 == 1` and `x ** 1 == x` for all `x`.

`x += y`
Means '`x = x + y`', where `x` must be `*READ-WRITE*`.

`next x += y`
Means '`next x = x + y`'. '`next x`' must be defined.

6 Primaries

A **primary** is an *expression* that has no operators outside parentheses or brackets:

primary ::= *constant-primary*
| *reference-expression* [p37]
| *function-call* [p39]
| *bracketed-expression* [p40]

constant-primary ::= *constant* other than *rational-constant*

constant ::= see p32

rational-constant ::= see p33

Note that a *rational-constant* is an operator (e.g. `D#`) followed by a *string-constant*, and therefore contains an operator and is not a *primary*.

6.1 Names

A **name** is a sequence of lexemes used to name things like variables and functions. Names are building blocks of primaries.

name ::= *initial-name-item continuing-name-item**
initial-name-item ::= *name-item* other than *natural*
continuing-name-item ::= *name-item* not containing '`.`'
name-item ::= *word* containing no '`.`' that follows a character that is not a '`.`'
[i.e., '`.`'s can only be at the beginning of the *word*]
[see text about splitting words with embedded '`.`'s]
| *natural* [see p19]
| *quoted-mark* containing no '`.`' that follows a character
that is not the beginning `"` or another '`.`'
[i.e., '`.`'s can only be at the beginning of the *mark*]
[see text about splitting marks with embedded '`.`'s]
| *quoted-separator* not containing '`.`'s
quoted-mark :::= `"` *mark* `"`
quoted-separator :::= `"` *separator* `"`

Words and *marks* containing embedded '`.`'s are split into parts which contain '`.`'s only at their beginning. Thus:

`bill.1.weight..size`	splits into	`bill` `.1` `.weight` `..size`
`p.@.*`	splits into	`p` `.@` `.*`
`.@.*`	splits into	`.@` `.*`
`.,.*`	splits into	`.,` `.*`
`.*.p`	splits into	`.*` `.p`

However it is a compile error if one of the parts is not a *word* or *mark*, as in the examples where `.1` is a *number* and `.,` is not a legal lexeme.

Quoted-marks are not split.

A function may be defined with a *name* that is a *quoted-mark*, such as `"+"`. It may then be called by an expression in which the quotes are omitted, such as `x + y`. Similarly a function may be defined with a function-term `".*"` and called by the expression `p.*` which is split into `p .*`.

Name items beginning with more than one '`.`' are reserved for use by systems and compilers (e.g., `..size` in the example). Name items that are words containing '`$`' or that both begin and end with '`*`' are similarly reserved. For example, words of the form '`T$`...' are reserved for use as type wildcards.

A name may begin with a *word* that is a *module-abbreviation* that designates a code module: see 9.7 [p87]. For example `std` abbreviates the builtin standard module.

L-Language uses several kinds of names:

simple-name ::= *word* not containing any '`.`'s or '`@`'s
module-abbreviation ::= *simple-name*
ma ::= *module-abbreviation*
pointer-type-name ::= *ma*? *simple-name*

basic-name ::= *name* not containing a '`.`', *quoted-mark*, or *quoted-separator*
type-name ::= *ma*? *basic-name* not containing '`@`'s
variable-name ::= *ma*? *basic-name*
pointer-variable ::= *variable-name* whose *basic-name* begins with an `@`
target-variable ::= *variable-name* whose *basic-name* does not begin with an `@`
statement-label ::= *basic-name* not containing '`@`'s

member-name ::= *name* beginning with a *word* or *quoted-mark* containing a '`.`',
but not containing a *quoted-separator*
(note: all '`.`'s in a *name* must be at the beginning of the *name*)
pointer-member-name ::= *member-name* with '`@`' following the initial '`.`'s
target-member-name ::= *member-name* that is not a *pointer-member-name*

data-label ::= *basic-name* | *member-name*

function-term-name ::= *name* not containing a '`.`'

qualifier-name ::= `co` | `ro` | `*READ-WRITE*` | `*WRITE-ONLY*`
| `*GLOBAL*` | `*LOCAL*` | `*HEAP*`

`co` abbreviates 'constant' meaning 'never changes'
`ro` abbreviates 'read-only' meaning 'other code may change'
`*GLOBAL*` has global (forever) lifetime
`*LOCAL*` has lifetime that ends when current stack frame is destroyed
`*HEAP*` has lifetime managed by a garbage collector

operator-word ::= `if` | `else` | `while` | `until` | `AND` | `OR` | `NOT` | `BUT`

function-keyword ::= `no` | `not` | `function`
| `"="` | `","` | `"("` | `")"` | `"["` | `"]"`

wild-card ::= *simple-name* beginning with *wild-card-prefix*

wild-card-prefix :::= one of:

`T$`	name is assigned a *type-name*
`P$`	name is assigned a *pointer-type-name*
`Q...$`	*qualifier-wild-card*; name is assigned a list of *qualifier-names* subject to *qualifier-wild-card-flags* '...'

qualifier-wild-card ::= `Q` *qualifier-wild-card-flag** `$`

qualifier-wild-card-flag :::= one of:

`R`	readable, excludes `*WRITE-ONLY*`	
`W`	writable, excludes `ro` and `co`	
`L`	allows `*LOCAL*`	(see rule 6 below)
`G`	allows `*GLOBAL*`	(see rule 6 below)
`H`	allows `*HEAP*`	(see rule 6 below)

where the following rules should be followed, least there be various confusing syntax or semantic errors (some, but not all, violations of these rules will be detected as compilation errors):

1. A *type-name* should not begin with a *pointer-type-name*.
2. A *pointer-type-name* should not begin with a *type-name*.
3. *Function-term-names* and *basic-names* should not begin with a *module-abbreviation* or contain *function-keywords*.
4. *Names* not used as operators should not begin with initial *operators*, should not end with final *operators*, and should not contain *operators* that are neither initial nor final.
5. *Names* that are not *qualifier-names* should not contain *qualifier-names*, with the exception that a *qualifier-name* by itself can be a *function-term*.
6. If any of `L`, `G`, or `H` are used as a *qualifier-wild-card-flags*, those not used indicate that their corresponding lifetime types are not allowed. If none of `L`, `G`, or `H` are used, all lifetime types are allowed (as if `LGH` has been used).

For example, the name resolver treats a sequence of names in certain contexts as having the form:

$$\{\ \textit{qualifier-name}^\star\ \ \textit{ma}^?\textit{pointer-type-name}\}^?$$
$$\textit{qualifier-name}^\star\ \ \textit{ma}^?\textit{type-name}\ \ \textit{ma}^?\textit{variable-name}$$

where $\textit{ma}^?$ denotes an optional *module-abbreviation*, and while scanning this sequence from left to right, the name resolver does not back up after identifying a *qualifier-name*, *module-name*, *pointer-type-name* or *type-name* in the sequence.

Variable-names, *type-names*, and *pointer-type-names* that begin with a *module-abbreviation* are called **external**. Non-external names are called **internal**.

A name can abbreviate another name, using the statement:

abbreviation-statement ::= *abbreviating-name* ---> *abbreviated-name*

For example:

```
bool ---> std bool
```

Note that it is whole names that are abbreviated, and not parts of names.

The ---> operator executes at compile time. The *abbreviation-statement* must be within the scope[p91] of a definition of the *abbreviated-name*, which must be one of the following kinds:

pointer-type-name
type-name
qualifier-name
pointer-variable
target-variable
statement-label
pointer-member-name
target-member-name

The *abbreviating-name* will be of the same kind as the *abbreviated-name*, and must follow the syntax rules of that kind. For example, if the *abbreviated-name* is a *target-name*, the *abbreviating-name* cannot begin with '`@`'.

Note that *function-term-names* used in *function-calls* cannot be abbreviated.

6.2 Constants

A **constant** is a value of type `const` computed at compile-time. One type of constant, the map constant, is not actually constant and can be changed.

There are five of types of constants:

constant ::= *special-constant*
| *string-constant*
| *number-constant*
| *rational-constant*
| *map-constant*

special-constant ::= `TRUE` | `FALSE` | `UNDEF` | `NONE`
| `*LOGICAL-LINE*` | `*INDENTED-PARAGRAPH*`

string-constant ::= *quoted-string*

The meanings of the *special-constants* are:

`TRUE`	The boolean value true. Convertible to `bool ( 1 )`.
`FALSE`	The boolean value false. Convertible to `bool ( 0 )`.
`UNDEF`	The value exists but is undefined (unknown).
`NONE`	The value does not exist.
`*LOGICAL-LINE*`	see 11 [p90]
`*INDENTED-PARAGRAPH*`	see 11 [p90]

A special constant is not equal to any other constant. The constant `TRUE` can be implicitly converted to the run-time `bool` value `1`. The constant `FALSE` can be implicitly converted to the run-time `bool` value `0`.

A ***string-constant*** is just a *quoted-string* lexeme that denotes a character string: see p18 and p17.

String constants can be used to load run-time vectors with `uns8`, `uns16`, or `uns32` type elements. UTF-8, UTF-16, or UTF-32 encodings are used according to element size.

A ***number-constant*** is an *natural*, *number*, or *numeric-word* lexeme converted to an IEEE 64-bit floating point number.

A number constant may be converted to a run-time number type such as `int32` or `flt64`. It is a compile error to convert to an integer type that cannot hold the exact value of the number. Conversion to a run-time floating type is however never a compile error. If necessary the converted value is `+Inf` or `-Inf` or loses precision.

Rational-constants and *map-constants* are described in the following sections.

6.2.1 Rational Constants

A **rational constant** is a rational number with unbounded numerators and denominators, where the denominator is at least 1 and the numerator and denominator have no common factors (other than 1). If the denominator is 1, the rational is called a **rational integer**.

Rational constants are computed at compile-time by the operators:

Operator	Argument String
`D#`	*decimal-constant-string*
`B#`	*binary-constant-string*
`X#`	*hexadecimal-constant-string*
`C#`	*character-constant-string*

Each of these operators takes a constant string as its sole argument. The syntax of the argument strings is:

sign ::= `+` | `-`
exponent ::= { `e` | `E` } *sign*$^{?}$ *dit*$^{+}$
decimal-constant-string
::= `"` *decimal-natural* *decimal-fraction*$^{?}$ *exponent*$^{?}$ `"`
| `"` *decimal-natural* `/` *decimal-natural* `"`
decimal-natural ::= *dit*$^{+}$ { `,` *dit dit dit* }*
decimal-fraction ::= `.` { *dit dit dit* `,` }* *dit*$^{+}$
dit ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
binary-constant-string
::= `"` *binary-natural* *binary-fraction*$^{?}$ *exponent*$^{?}$ `"`
| `"` *binary-natural* `/` *binary-natural* `"`
binary-natural ::= *bit*$^{+}$ { `,` *bit bit bit bit* }*
binary-fraction ::= `.` { *bit bit bit bit* `,` }* *bit*$^{+}$
bit ::= `0` | `1`
hexadecimal-constant-string
::= `"` *hexadecimal-natural* *hexadecimal-fraction*$^{?}$ *exponent*$^{?}$ `"`
| `"` *hexadecimal-natural* `/` *hexadecimal-natural* `"`
hexadecimal-natural ::= *hit*$^{+}$ { `,` *hit hit* }*
hexadecimal-fraction ::= `.` { *hit hit* `,` }* *hit*$^{+}$
hit ::= `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` | `a` | `b` | `c` | `d` | `e` | `f` | `A` | `B` | `C` | `D` | `E` | `F`
character-constant-string ::= `"` *character-representative* `"`
character-representative ::= see p18

where

- Denominators in fractions must not be zero.

Decimal naturals may have commas every 3 digits from the end and decimal fractions may have commas every 3 digits from the decimal point. Similarly for binary naturals and fractions with commas every 4 binary digits, and with hexa-decimal naturals and fractions with commas every 2 hexa-decimal digits. If there is a decimal point, there must be at least one integer digit and one fraction digit.

For decimal constants without `/` the denominator is a power of `10`; for binary constants without `/` the denominator is a power of `2`; and for hexadecimal constants without `/` the denominator is a power of `16`.

The value of a character constant is the integral UNICODE code point of the *character-representative*.

A rational constant may be converted to a run-time number type such as `int32` or `flt64`. It is a compile error to convert to an integer type that cannot hold the exact value of the rational constant. Conversion of a rational constant to a run-time floating type is however

never a compile error. If necessary the converted value is `+Inf` or `-Inf` or precision is lost.

6.2.2 Map Constants

A **map constant** has two parts, a list (a.k.a, a vector) and a dictionary. Either or both can be empty.

A map constant is computed by a *map-expression* whose syntax is:

map-expression ::= `{ }`
| `{` *map-list* `}`
| `{` *map-dictionary* `}`
| `{` *map-list*`,` *map-dictionary* `}`
| *phrase-constant*
| *expression-constant*
| *type-constant*
| *pointer-type-constant*

map-list ::= *list-element* { `,` *list-element* }*

map-dictionary ::= *dictionary-entry* { `,` *dictionary-entry* }*

dictionary-entry ::= *dictionary-label* `=>` *dictionary-value*

list-element ::= *const-expression*

dictionary-label ::= *const-expression* evaluating to a string

dictionary-value ::= *const-expression*

const-expression ::=
`const` valued *expression* evaluatable at compile-time, see 9.4 ^{p79}

expression ::= see p23

phrase-constant ::= `‘` *expression* `’`

expression-constant ::= `{*` *expression* `*}`

type-constant ::= *type-name*

pointer-type-constant ::= *pointer-type-name*

type-name ::= see p30

pointer-type-name ::= see p30

Maps cannot be represented at run-time.

By abuse of language, **list** is used to refer to a map whose dominant mode of access is to go through the map list elements sequentially. Similarly **vector** is used to refer to a map whose dominant mode of access is to access the map list elements randomly using subscripts. And **dictionary** is used to refer to a map whose dominant mode of usage is to access the map's dictionary elements.

Dictionary entries are also called **attributes**. For lists and vectors, they are also called

annotations.

Each *map-expression* creates a distinct map: no two such maps are `==`. A map created by a *map-expression* is initially set so that it and all its dictionary entries are read-only. This can be changed: see p113.

An ***expression-constant*** is shorthand for the *map-constant* produced when the *expression* is parsed: see p90. Generally, parsing an expression groups expression elements into sublists and moves bracket and separator punctuation to annotations (dictionary entries). Some examples are:

The expression:	Is equivalent to:

`{* X = ( Y + 1 ) *}`	`{ { "X" }, "=",`
	`  { { "Y" }, "+", { 1 }, ".initiator" =>"(",`
	`  ".terminator" =>")" } }`

`{* X, Y = Y, X *}`	`{ { "X", "Y", ".separator" => "," },`
	`  "=",`
	`  { "Y", "X", ".separator" => "," } }`

`{* X + Y * Z *}`	`{ { "X" }, "+", { { "Y" }, "*", { "Z" } } }`
`{* X 3 = Y Z + 1 *}`	`{ { "X", 3 } "=", { { "Y", "Z" } "+", { 1 } } }`

In an `{*` *expression* `*}` constant, line level operators (those in Figure 2 p24) are recognized if and only if they are outside parentheses in the *expression*.

Phrase-constants are like *expression-constants* except that operators (including separators, e.g. '`,`') are not recognized. Brackets are recognized and create sublists. Some examples contrasting with *expression-constants* are:

The Expression:	Is Equivalent To:
`'X = Y + 1'`	`{ "X", "=", "Y", "+", 1 }`
`{* X = Y + 1 *}`	`{ { "X" }, "=", { { "Y" } "+" { 1 } } }`

`'X = ( Y + 1 )'`	`{ "X", "=", { "Y", "+", 1,`
	`  ".initiator" => "(",`
	`  ".terminator" => ")" } }`

`{* X = ( Y + 1 ) *}`	`{ { "X" }, "=", { { "Y" }, "+", { 1 },`
	`  ".initiator" => "(",`
	`  ".terminator" => ")" } }`

Map constants containing parsed code can be computed by *include-statements*: see 10 p89.

Type-names and *pointer-type-names* can be used at compile-time as if they were variables of type `const` with map values. These map values are partly read-only, with the read-only part including elements with labels like `.size` for the size in bits of run-time values of the type. Users can add their own elements if these do not conflict with the names of the read-only elements. See p117.

6.3 Reference Expressions

A ***reference-expression*** computes either a pointer to a location in memory, or the value of such a location, or stores a value in a location, or appears to do these things via calls to reference functions.

Syntactically a typical *reference-expression* begins with a *variable-name* or '**next** *variable-name*' and then adds suffixes that begin with either a **[** bracket or a *member-name* (which begins with '**.**': see p30). A *reference-expression* that is a *variable-name* or '**next** *variable-name*' by itself or inside parentheses is called **simple**. More complex *reference-expressions* are called **compound** and consists of two parts: a **base** that is a smaller *reference-expression*, and an **offset** that follows the base with a **[** or *member-name*. The base may be parenthesized, and if it is, the offset may have a *module-abbreviation* preceeding the base, but the rest of the offset follows the base.

So a *reference-expression* begins with a *variable-name* or '**next** *variable-name*' that may be preceded only by left parentheses and *module-abbreviations*. The *variable-name* or '**next** *variable-name*' is called the **root** of the *reference-expression*.

The type of a *reference-expression* is computed bottom up starting with the type of the base which has been previously computed bottom up. The type of the *variable-name* in the root is specified by context. The type of the base of a compound *reference-expression* is known by recursion. This type, the syntax of the offset, and the types of the any subexpressions that are arguments in the offset determine the type of a compound *reference-expression*.

This is different from the computation of the type of a general sub-*expression*, which proceeds top-down using a target type provided by the *statement* containing the sub-*expression*. Since the type of a *reference-expression* does not depend on its containing *statement*, implicit conversion of a *reference-expression* value to a *statement* provided target type is allowed, whereas implicit conversion of a sub-*expression* that is a *function-call* is not allowed.

The syntax of *reference-expressions* is:

reference-expression ::= *variable-name*
| **next** *variable-name*
| *constant*
| *reference-expression* **[** *index-list* **]**
| *reference-expression member-name*
| *reference-call*
| **(** *reference-expression* **)**

index-list ::= *index* { **,** *index* }*

index ::= *expression*

variable-name ::= see p30

constant ::= see p32

member-name ::= see p30

reference-call ::= see p81

A *reference-call* is a call to a reference function. For details see Reference Functions [p80]. The rest of this section applies to builtin *reference-expressions* that are not *reference-calls*.

A *member-name* of the form '`.`*data-label*' may be used to select a field or subfield of a user defined type [p53] value or a dictionary entry of a *map-dictionary*.

An *index* may be used to select an element of a vector or array in a user defined type value, or the element of a vector pointed at by a pointer, or an element of a *map*. An *index* can be a *const* string. For a run-time *reference-expression* a string *index* is a *data-label*: '`["M"]`' is equivalent to '`.M`'. For a compile-time *reference-expression* the string is used to select a map *dictionary-entry*. Otherwise the *index* must be a positive or negative integer. Bounds imposed by user defined types or stored in a pointer are used to check that the *index* is within range. The target type of a run-time *index* is **`int`**.

Within an *index-list* the comma (`,`) is treated as equivalent to `][`, so, for example, `[x,y]` is equivalent to `[x][y]`.

If the *variable-name* in the root of a *reference-expression* is a *pointer-variable*, the reference expression computes a pointer. E.g., '`@V[5]`' computes a pointer to the 5+1'st element of the vector pointed at by `@V`.

If the *variable-name* in the root of a *reference-expression* is a *target-variable* with an associated *pointer-variable*, the reference expression refers to the value pointed by the pointer that would be computed if the *target-variable* was replaced by its associated *pointer-variable*.

If the *variable-name* in the root of a *reference-expression* is a *target-variable* (i.e., does not begin with '`@`') without an associated *pointer-variable*, the reference expression refers to the variable itself, to a field or dictionary entry of the variable's value, or to a vector or list element of the variable's value.

When a *reference-expression* consists of a base that designates a **container** followed by a *member-name* that designates a field or subfield of this container, and the container is not a `const` value, both the container and the field or subfield have qualifiers. If the container is in a function frame its sole qualifier is `co` or `*`INIT*, as determined by context. Otherwise the container is the target of a pointer, and its qualifiers are the target qualifiers of the pointer type. A field's qualifiers are those its *field-declaration* [p53]. A subfield's qualifiers are those of its containing field.

The qualifiers of a pointer computed by the reference expression, or of the target value referenced by the reference expression, are computed from the container and field qualifiers as per the following:

Reference Expression Result
Pointer Target or Value Qualifiers
Given Container and Field Qualifiers

Container Access Qualifier	Field Access Qualifier				
	(none)	co	ro	*READ-WRITE*	*WRITE-ONLY*
co	co	co	co	co	(illegal)
ro	ro	co	ro	ro	(illegal)
READ-WRITE	*READ-WRITE*	co	ro	*READ-WRITE*	*WRITE-ONLY*
WRITE-ONLY	*WRITE-ONLY*	(illegal)	(illegal)	*WRITE-ONLY*	*WRITE-ONLY*
INIT	*INIT*	*INIT*	*INIT*	*INIT*	*INIT*

- Here *INIT* is a pseudo-qualifier used for containiners that are being initialized. It has the same effect as *WRITE-ONLY* except that it forces fields/subfields of the container to be *INIT* even if they are co or ro.
- The *reference-expression* has the same lifetime qualifier (*GLOBAL*, *HEAP*, or *LOCAL*) and lifetime depth as the container (see 13 [p96]).

Map constants are represented internally by pointers to where the map is stored, so that if X is a variable equal to, i.e., pointing at, a map, then Y = X copies the pointer to the map to the variable Y. By default map constants are read-only and cannot be changed, but it is possible to mark a whole map as read-write, and to independently mark dictionary members as either read-only or read-write. Dictionary members are marked read-only when they are initially created.

The following example illustrates computation with map constants:

```
const X = {"A", "B"}
X[0] = "C"                // Illegal, X is read-only.
read-write ( X )
X[0] = "C"                // Now X is {"C", "B"}.
const Y = X               // Now X and Y are both {"C", "B"}.
Y[1] = "D"                // Now X and Y are both {"C", "D"}.
const Z = { Y, "M" }      // Now Z is {{"C", "D"}, "M"}.
Z[0].W = "N"              // Now Z is {{"C", "D", "W" => "N"}, "M"},
                          // X and Y are both {"C", "D", "W" => "N"}.
X.W = "P"                 // Illegal, X.W is read-only.
read-write ( X, "W" )
X.W = "P"                 // X and Y are both {"C", "D", "W" => "P"}.
                          // Now Z is {{"C", "D", "W" => "P"}, "M"},
```

6.4 Function Calls

The syntax of function calls is:

function-call ::=
module-abbreviation$^{?}$ *parenthesized-call-argument-list** *call-term*$^{+}$

call-term ::= *call-term-name call-argument-list**
| **no** *call-term-name*
| **not** *call-term-name*

call-term-name ::= *function-term-name* with quotes optionally removed from *quoted-marks* and *quoted-separators*

function-term-name ::= see p30

call-argument-list ::= **(** *actual-argument* { **,** *actual-argument* }* **)**
| **[** *actual-argument* { **,** *actual-argument* }* **]**
| **( )** | **[ ]**

parenthesized-call-argument-list ::=
call-argument-list with parentheses **( )** (and not square brackets **[ ]**)

actual-argument ::= *expression*

expression ::= see p23

- *Call-term-names* cannot be abbreviated.
- *Call-terms* of the form '**no x**' and '**not x**' are equivalent to '**x(FALSE)**'.

Thus a *function-call* is a sequence of *function-term-names* and *call-argument-lists*. Note that *Function-term-names* cannot contain '**.**'s and therefore cannot be *member-names*, which are reserved for *reference-calls*^{p81}. Also **[ ]** bracketed *call-argument-lists* cannot be placed before a *call-term-name*, a syntactic distinction to their use in *reference-calls*.

Function-calls are matched to function prototypes. The *call-term-names* in a match are identical to the *function-term-names* taken from the prototype being matched, except that quotes (**"**) in a prototype *quoted-mark* or *quoted-separator* may (or may not) be omitted in the *function-call*. The first step in matching is to scan the *function-call* to identify the *call-term-names*. There is no parser backing up after this is done: if the results of this initial scan do not lead to a satisfying match, the entire call-prototype match fails.

6.5 Bracketed Expressions

The syntax of a *bracketed-expression* is:

bracketed-expression ::= *ma*$^{?}$ **(** *expression* **)**
| **[** *expression* **]**
| **{** *expression* **}**
| **‘** *expression* **’**
| **{*** *expression* ***}**

ma ::= *module-abbreviation* [see p30]

Arithmetic subexpressions and some function argument lists are bracketed with **()** brackets. Reference expression index lists and some function argument lists are bracketed with **[]**

brackets. Expressions that compute map constants are bracketed with `{ }`, `‘ ’`, or `{* *}`, brackets (see 6.2.2 p35).

An expression of the form ‘*ma* `(` *expression* `)`’ is just syntactic sugar for ‘`(` *ma expression* `)`’, except that the *expression* is parsed before the *ma* is moved inside the `()`’s. Thus if `mom` is a *module-abbreviation*, ‘`mom ( x + y * z )`’ is syntactic sugar for

```
( mom x "+" ( y "*" z ) )
```

in which the parenthesis pair surrounding ‘`y "*" z`’ is implied. This allows the *module-abbreviation* to be applied to the outermost operator in the *expression*.

7 Assignment Statements

Assignment-statements have a list of variables on the left side of an `"="` operator which receive values from a list of expressions or a block of code on the right side of the operator. The left-side variables and the `"="` may be omitted if the right side produces no values, or if all left-side variables have the form of ‘`next` *variable-name*’ and are implied by the right side.

The forms of an *assignment-statement* are:

assignment-statement ::= *expression-assignment-statement*
| *call-assignment-statement*
| *block-assignment-statement*
| *deferred-assignment-statement*
| *loop-assignment-statement*

expression-assignment-statement ::=
assignment-result { `,` *assignment-result* }* `=` *expression-list*

expression-list ::= see p45

call-assignment-statement
::= *assignment-result* { `,` *assignment-result* }* `=` *assignment-call*
| *assignment-call*

assignment-call ::= *function-call* | *call-expression*

function-call ::= see p39

call-expression ::= see p85

block-assignment-statement ::=

block-variable-declaration { `,` *block-variable-declaration* }*
{ `=` { `do` *block-label* }$^?$ }$^?$ `:`
statement*
exit-subblock*
| `do` *block-label*$^?$ `:`
statement*
exit-subblock*

deferred-assignment-statement ::=
deferred-variable-declaration { `,` *deferred-variable-declaration* }* `=*DEFERRED*`

loop-assignment-statement ::=
next-variable-declaration { `,` *next-variable-declaration* }* `=`
iteration-control `:`
statement*
exit-subblock*
| *iteration-control* `:`
statement*
exit-subblock*

exit-subblock ::= *exit-label* `exit:`
statement*

iteration-control ::= see p50

block-label ::= *statement-label*

exit-label ::= *statement-label*

statement-label ::= see p30

assignment-result ::= *result-variable-declaration*
| *next-variable-declaration*
| *reference-expression*

reference-expression ::= see p37

result-variable-declaration ::=
type-name target-variable
| *pointer-type-name qualifier-name** *type-name pointer-variable*

next-variable-declaration ::= `next` *variable-name*

block-variable-declaration ::=
type-name target-variable
| `next` *target-variable*
| *pointer-type-name qualifier-name** *type-name pointer-variable*
{ `@=` *allocation-call* }$^?$
| `next` *pointer-variable* { `@=` *allocation-call* }$^?$

deferred-variable-declaration ::=

type-name target-variable
| *pointer-type-name qualifier-name** *type-name pointer-variable*
{ `@=` *allocation-call* }$^{?}$

qualifier-name ::= see p30
pointer-type-name ::= see p30
type-name ::= see p30
variable-name ::= see p30
target-variable ::= see p30
pointer-variable ::= see p30
allocation-call ::= *function-call*

where

- A line-ending `:` may be omitted if the *statement* subblock following is empty.
- The allocation subexpression '`@= local`' may be omitted for *pointer-variable* declarations whose targets have the `*LOCAL*` qualifier provided the *allocator-call* has no arguments (e.g., '`@= local[...]`' cannot be omitted).

Associated with *block-assignment-statements* and *loop-assignment-statements* there are *control-statements* to control the flow of execution within the more complex *assignment-statement*:

control-statement ::= *block-control-statement* [p47]
| *loop-control-statement* [p50]

A *...-variable-declaration* allocates memory for its variables in the frame of the currently executing out-of-line function. The sizes of these variables must be known at compile time. For sizes not known at compile time, a pointer to the variable can be allocated and the '`local`' function called to set the pointer. The '`local`' function allocates memory to the stack after the currently executing out-of-line function's frame.

Expression-assignment-statements set the values of their variables to the values of the *expressions* in the *expression-list*. *Call-assignment-statements* set the values of their variables to the values returned by the *function-call*.

A *block-variable-declaration* initializes its variables according to the declaration syntax as follows:

Non-`next` variable declaration with no *allocation-call*: The memory is zeroed.
`Next` variable declaration with no *allocation-call*: Previous value of the named variable.
Variable declaration with *allocation-call*: Value returned by the *allocation-call*.

Zeroed numbers are zero, while zeroed pointers typically cause segmentation faults when de-referenced.

The variables declared by *block-variable-declarations* without *allocation-calls* are given the

qualifier `*INIT*` in *statements* of the *block-assignment-statement* and `co` after the *block-assignment-statement*. The `*INIT*` qualifer is equivalent to `*WRITE-ONLY*` except that it forces fields and elements of the value to also be `*INIT*` [p39].

The *pointer-variables* declared by *block-variable-declarations* with *allocation-calls* are initialized by the *allocation-call* (see p47) and are thereafter `co`, but the memory pointed at is zeroed initially, `*INIT*` in *statements* of the *block-assignment-statement*, and subject to the *pointer-variable qualifiers* after the *block-assignment-statement*.

If a *declaration* declares a *pointer-variable*, an associated *target-variable* is implicitly declared at the same time whose name is the *pointer-name* with the initial '`@`' removed. The *target-variable* is not itself allocated to memory, but instead references the value the *pointer-variable* points at.

Note that a variable declaration does not allow qualifiers on anything but the target of a pointer. The implicit qualifier of a declared variable is `co` after the *assignment-statements* meaning that the value of the variable once initially set is never changed. The qualifiers of a *target-variable* associated with a *pointer-variable* are those of the target of its associated pointer.

A *next-variable-declaration* for a *variable* `v` must occur in the scope of either a non-*next-variable-declaration* for `v` or another *next-variable-declaration* for `v`. Furthermore, `v` cannot be a *target-variable* associated with a *pointer-variable*. The *next-variable-declaration* redeclares `v` making a new variable that hides the previously declared `v`. The new variable has the same types and qualifiers as the previous variable named `v`.

A *next-variable-declaration* for variable `v` enables '`next v`' to be used like a *variable-name* in *reference-expressions* within the *statements* of the *assignment-statement*. Use of `v` within these statements outside of '`next v`' refers to the value of `v` just before the *assignment-statement* was executed.

Under some circumstances '`next v`' will be implicitly added to the *assignment-result* list of a *call-statement* (see p46), the *block-variable-declaration* list of a *block-assignment-statement* (see p48), or the *next-variable-declaration* list of a *loop-assignment-statement* (see p51).

A *deferred-variable-declaration* behaves like a *block-variable-declaration* with an empty block, but in addition allows the variable to be redeclared in a subsequent 'companion' *block-variable-delcaration* which does not reallocate the variable.

For loops, 4 copies of the variables allocated by an iteration are allocated to the out-of-line function frame when the function is called, and the loop cycles among these copies. A call to '`local`' inside a loop sets a pointer allocated to one of these 4 copies. If such a pointer is already allocated by a previous loop iteration, its memory is reused, or doubled in size and reallocated. This means that the total memory allocated to a pointer inside a loop copy by '`local`' will never be more than 4 times the size of the maximum memory needed for any single loop iteration. As there can be 4 copies of the iteration variables, a call to '`local`' inside a loop may allocate to the stack at most 4x4 = 16 times as much memory as any

single iteration call to '`local`'.

7.0.1 Expression Assignment Statements

The syntax of an *expression-assignment-statement* is:

expression-assignment-statement ::=
assignment-result { **,** *assignment-result* }* **=** *expression-list*

expression-list ::= *expression* { **,** *expression* }*

assignment-result ::= see p42

expression ::= see p23

where

- The number of *expressions* must equal the number of *assignment-results*.
- The *expression-list* must not consist of a single *expression* which is a *function-call* (else the *statement* is a *call-assignment-statement* as described in 7.0.2 [p46]). However the *expression-list* may consist of a single *reference-call*.

Sub-*expressions* computable at compile-time are evaluated in left to right order and replaced by their `const` values before the *statement* is compiled into run-time code. At run-time *expressions* are evaluated in left-to-right order and then the *expression* values are stored in the *assignment-results*.

Variable names declared by *result-variable-declarations* that are *assignment-results* are not visible to the *expressions*. In particular, if '`next V`' is an *assignment-result*, the name '`V`' in an *expression* will refer to the variable that exists before the *expression-assignment-statement*.

The type of an *assignment-result* becomes the **target type** of its corresponding *expression*. If the *expression* is a *function-call*, the type of the first result of the *function-prototype* must match the target type of the *expression*, and the types of the prototype arguments become the target types of the *actual-argument* sub-expressions.

If an *expression* is a *reference-expression* or a *const-expression* (which is replaced by a constant), it will be implicitly converted to its target type if possible.

An example is:

```
int x = 5              // target type of 5 is int
flt y = 6              // target type of 6 is flt
flt z = 7              // target type of 7 is flt
flt r1 = x + y         // target type of +, x, y is flt
flt r2 = x + y * z     // target type of +, x, *, y, z is flt
next x = z             // illegal; target type of z is int
```

Here the `+` and `*` operator functions are only defined for cases where their operand types are

the same as their result type, and `int` variables may be implicitly converted to `flt` but not vice-versa.

7.0.2 Call Assignment Statements

The syntax of a *call-assignment-statement* is:

call-assignment-statement
::= *assignment-result* { **,** *assignment-result* }* **=** *assignment-call*
| *assignment-call*

assignment-call ::= *function-call* | *call-expression*

assignment-result ::= see p42

function-call ::= see p39

call-expression ::= see p85

A *call-assignment-statement* with *assignment-results* follows the same general rules as *expression-assignment-statements* except that its right side is a single *function-call* or the functionally similar *call-expression*.

The types of the function prototype results cannot be implicitly converted to the types of the *assignment-results*.

The right side of a *call-assignment-statement* may have an implied '**next** *v*' if the *function-prototype* has a '**next** *w*' *prototype-result-declaration* and a *prototype-argument-declaration* of the form '... *type-name w*', and *v* is a *variable-name* that by itself is the actual argument matched to *w*. See p68.

7.0.3 Block Assignment Statements

The syntax of *block-assignment-statements* is:

block-assignment-statement ::=
block-variable-declaration { **,** *block-variable-declaration* }*
{ **=** { **do** *block-label* }$^?$ }$^?$ **:**
statement*
exit-subblock*
| **do** *block-label*$^?$ **:**
statement*
exit-subblock*

block-variable-declaration ::= see p42

exit-subblock ::= *exit-label* **exit:**
statement*

exit-label ::= *statement-label*

statement-label ::= see p30
statement ::= see p22
exception-subblock ::= *exit-subblock* whose *exit-label* is `*EXCEPTION*`
block-control-statement ::= *goto-exit-statement* | *throw-exception-statement*
go-to-exit-statement ::= see p47
throw-exception-statement ::= see p48

where

- The *exit-label* `*EXCEPTION*` is special and any *exception-subblock* must be the last *exit-subblock* in the *block-assignment-statement*.

The *block-assignment-statement* first allocates and initializes memory in the current function frame for the variables declared by the *block-variable-declarations*.

Then any *statements* and *exit-subblocks* are executed. During this execution block variables not set by an *allocation-call* are `*INIT*` [p39], and after this execution, these variables become `co`. During this execution pointer variables set by an *allocation-call* are `co`, but their target type is changed to `*INIT*`, and after execution the target type qualifiers become whatever the pointer variable declarations specify.

When a declaration has an *allocation-call*, its variable must have pointer type, and the *allocation-call* is executed to set the pointer before any *statements* in the *block-assignment-statement* are executed. The *allocation-call* is executed with a pre-pended argument list consisting of two `uns` values in `()` parentheses. The first value is the number of bytes to be allocated, and the second value is the byte alignment of the memory to be allocated. The *prototype-pattern* of the called function's prototype [p67] must begin with '`( uns length, uns alignment )`', although the argument names may be different. The called function must allocate a block of memory with the required number of bytes and alignment and zero that block. The prototype must have exactly one result variable whose type is identical to the the pointer type of the *pointer-variable* being set (but the prototype result type may contain wildcards).

As a general rule, allocator functions that return a value of type `av` or `fv` or a user defined vector pointer have a `[]` argument list with a single argument giving a vector size `N`. The allocator allocates not a single block of the given length and alignment, but instead a vector of `N` such blocks, with zero padding between the blocks if necessary to obtain proper alignment for each block. However, this is by convention and is not a builtin requirement of the L-Language. The convention is followed by the builtin allocators (e.g., `local`).

A *go-to-exit-statement* within a block may exit the block or enter an *exit-subblock* of the block:

go-to-exit-statement ::= `go to` *go-to-label* `exit`
go-to-label ::= *block-label* | *exit-label* other than `*EXCEPTION*`

Unless a *go-to-exit-statement* is executed, a block exits after the last *statement* in the block, and an *exit-subblock* exits its containing block after the last *statement* in the *exit-subblock*.

A *go-to-exit-statement* in an *exit-subblock* may only enter a subsequent *exit-subblock* or exit any of its containing *block-assignment-statements* by using that block's *block-label*.

An *exception-subblock* is entered if any preceeding *statement* in the *block-assignment-statement* executes a:

throw-exception-statement ::= `throw *EXCEPTION*`

It is also entered if any preceeding statement execution has a fault, such as a memory fault or an integer divide-by-zero fault. If a `throw *EXCEPTION*` statement is executed in an *exception-subblock*, it behaves as if the containing *block-assignment-statement* were replaced by a `throw *EXCEPTION*` statement. If a *block-assignment-statement* has no *exception-subblock*, it behaves as if it did have an *exception-subblock* whose only statement was a `throw *EXCEPTION*` statement.

Go-to-exit-statements define various possible execution paths through a *block-assignment-statement* (these are paths in an acyclic graph). It is a compile error if a *statement* within the *block-assignment-statement* uses a declaration and the statement can be reached by a path that does not contain the declaration. Note that declarations not in *exit-subblocks* have scope that includes the *exit-subblocks*, but declarations within an *exit-subblock* have scope that ends with the end of the *exit-subblock*. A function prototype is 'used' if and only if it matches a *function-call*.[4]

There is an exception to the last paragraph for *exception-subblocks*. In these any `co` variable allocated in the *block-assignment-statement* may be read, but the value will be zero or `null` if the variable's declaration has not been executed. The '`is set`' function p127 should be used to test a variable in this case.

If '`next` *variable-name*' is used as an *assignment-result* of some *statement* within a *block-assignment-statement* that is not within the scope of a *result-variable-declaration* for the *variable-name* that is also within the *block-assignment-statement*, then '`next` *variable-name*' will be automatically added to the *variable-declarations* of the *block-assignment-statement*, if it is not already there. For example:

```
int x = 5
do:
    next x = x + 1
```

is equivalent to:

```
int x = 5
next x =:
    next x = x + 1
```

[4]Stack space for every stack variable that might be used by an out-of-line function is allocated when the out-of-line function is called. Space allocated by the `local` function is treated differently: see p105.

7.0.4 Deferred Assignment Statements

The syntax of *deferred-assignment-statements* is:

deferred-assignment-statement ::=
deferred-variable-declaration { `,` *deferred-variable-declaration* }* `= *DEFERRED*`
deferred-variable-declaration ::= see p42

Each variable declared by a *deferred-variable-declaration* of a *deferred-assignment-statement* must be declared identically, except for addition or subtraction of an *allocation-call*, as a *block-variable-declaration* of a *block-assignment-statement* that is within the scope [p91] of the *deferred-assignment-statement*. The *block-assignment-statement*, known as the **companion** of the *deferred-variable-declaration*, computes the value of the declared variable, except in the case of a pointer variable whose value is set by an *allocation-call* in the *deferred-assignment-statement*. If a *deferred-assignment-statement* is in a module (9.7 [p87]), companions of its *deferred-variable-declarations* must be in that module or its bodies.

A pointer variable may have an *allocation-call* in either its *deferred-assignment-statement* or in its companion, but not both.

Deferred-assignment-statements and their companions must be top-level.

Deferred-assignment-statement variable initialization is the same as *block-assignment-statement* variable initialization, except that after the statement the variables are made *ro* and not *co* if they are not set by an *allocation-call* in the *deferred-assignment-statement*. Code that reads such `ro` variables before companions compute their values will read zero. For pointers this will typically reference undefined memory which will cause a memory fault if accessed. For code after a companion in the same file as the companion the variable qualifier will be changed to `co`.

For pointers allocated by a *deferred-assignment-statment*, the target value will be zeroed and the the pointer target will be given the `ro` qualifier, except for code after the pointer's companion in the same file as the companion, for which the pointer target will be given its declared qualifiers. In particular, function bodies after the companion in the companion's file will see these declared qualifiers.[5]

The variables declared in a *deferred-assignment-statement* are treated as normal *block-variable-declaration* variables inside their companions. In particular, inside their companions the `ro` variables are `*INIT*` [p39], and the `co` pointer variables set by an *allocation-call* in their *deferred-assignment-statement* have their target type changed to `*INIT*` inside their companions.

[5] If the companion is in a body, that body may not be initialized (thereby executing the companion) until after code that imports the body's module is initialized: see 9.7.1 [p88].

7.0.5 Loop Assignment Statements

A *loop-assignment-statement* has the syntax:

loop-assignment-statement ::=
next-variable-declaration { `,` *next-variable-declaration* }* `=`
iteration-control-list `:`
statement*
exit-subblock*
| *iteration-control-list* `:`
statement*
exit-subblock*

iteration-control-list ::= *iteration-control* { `,` *iteration-control* }*

iteration-control ::= `loop` *loop-label*$^?$
| `exactly` *int-expression* `times`
| `at most` *int-expression* `times`
| `while` *bool-expression*
| `until` *bool-expression*

loop-label ::= *statement-label*

statement-label ::= see p30

int-expression ::= *expression* evaluating to an `int`

bool-expression ::= *expression* evaluating to a `bool`
or to a *const* value that is either `"TRUE"` or `"FALSE"`

loop-control-statement ::= *break-statement* | *continue-statement*

break-statement ::= `break` *loop-label*$^?$

continue-statement ::= `continue` *loop-label*$^?$

A *loop-assignment-statement* is the semantic equivalent of a sequence of zero or more copies of the statement with its *iteration-controls* deleted, making these copies into *block-assignment-statements*. Each copy is called an **iteration** of the *loop-assignment-statement*. The number of iterations is determined at run-time by the *iteration-controls* and *loop-control-statements*.

A simple example is:

```
int sum = 0
int i = 1
next sum, next i = while i < 4:
    next sum = sum + i
    next i = i + 1
```

which is semantically equivalent to:

```
int sum = 0
```

```
int i = 1
next sum, next i =:
    next sum = sum + i
    next i = i + 1
next sum, next i =:
    next sum = sum + i
    next i = i + 1
next sum, next i =:
    next sum = sum + i
    next i = i + 1
// Now sum == 6 and i == 4
```

However at run-time the variable values of all but the last 4 iterations of the *loop-assignment-statement* are discarded, which would not be the case if the compiler actually inserted iterations in the source code. This only affects debugging.

The *iteration-controls* are independent of each other: `loop x` just provides a *loop-label*, `exactly x times` and `at most x times` both terminate the loop after `x` iterations (both do the same thing), `while x` terminates the loop if `x` is `false` at the start of an iteration, and `until x` terminates the loop if `x` is `true` at the start of an iteration. There can be at most one `loop x`, but there can be multiple variants of the other *iteration-controls*.

The *break-statement* exits the current iteration of the *loop-assignment-statement* and prevents further iterations. A *continue-statement* exits the current iteration of the *loop-assignment-statement* but lets the *iteration-control* determine whether there will be any more iterations. If there are nested loops, a *loop-label* may be used with these statements to designate which nested iteration is being exited.

As in *block-assignment-statements*, if '`next V`' occurs as an *assignment-result* within the loop *statements* but is not within the scope of a *result-variable-declaration* for `V` that is also within the *loop-assignment-statement*, '`next V`' will be added to the *next-variable-declaration* list of the *loop-assignment-statement*. Therefore the above example could be written as:

```
int sum = 0
int i = 1
next sum = while i < 4:
    next sum = sum + i
    next i = i + 1
```

or

```
int sum = 0
int i = 1
while i < 4:
    next sum = sum + i
    next i = i + 1
```

8 Conditional Statements

A *conditional-statement* executes another *statement* or block of *statements* according to what a *bool-expression* evaluates to. *Conditional-statements* have the syntax:

conditional-statement ::=
`if` *bool-expression* **`:`**
statement*
| **`else if`** *bool-expression* **`:`**
statement*
| **`else :`**
statement*
| **`if`** *bool-expression* **`:`** *statement*
| **`else if`** *bool-expression* **`:`** *statement*
| **`else :`** *statement*

bool-expression ::= see p50

where

- An '`else if`' or '`else`' *statement* must be immediately preceded by an '`if`' or '`else if`' *statement*.

An example is:

```
int x = 5
int y = 6
int z:
    int sum = x + y          // Sets sum = 11
    int product = x * y      // Sets product = 30
    if sum < product:
        z = sum              // Sets z = 11
    else: z = product        // Is NOT executed
// Now z = 11
```

9 Declarations

The following is a complete list of declarations:

declaration ::=	*result-variable-declaration*	[p42]
	next-variable-declaration	[p42]
	block-variable-declaration	[p42]
	prototype-result-declaration	[p67]
	prototype-argument-declaration	[p67]
	declaration-statement	

declaration-statement ::= *type-declaration* [p53]
| *pointer-type-declaration* [p62]
| *inline-function-declaration* [p66]
| *reference-function-declaration* [p80]
| *out-of-line-function-declaration* [p84]

9.1 Type Declarations

The syntax of a type declaration is:

type-declaration ::= `type` *defined-type-name* `:`
type-subdeclaration*
| `type` *defined-type-name* `: *DEFERRED*`
defined-type-name ::= *type-name*
type-name ::= see p30

type-subdeclaration ::= *field-declaration*
| `align` *alignment*$^?$
| `pack`
| `*INCLUDE*` *defined-type-name*
| `*LABEL*` *origin-label*
| `*ORIGIN*` *origin-label*
| `*OFFSET*` *integer-const-expression*
| `***`
| `*EXTERNAL*`
| `*OFFSET* *SIZE*`

field-declaration ::= *field-without-subfields-declaration*
| *field-with-subfields-declaration*

field-without-subfields-declaration ::=
qualifier-name* *field-type-name* *target-label* *field-dimension**
| *qualifier-name** *pointer-type-name*
qualifier-name* *field-type-name* *pointer-label* *field-dimension**

field-with-subfields-declaration ::=
qualifier-name* `std`$^?$ *number-type-name* *target-label*$^?$ *field-dimension**
subfield-declaration$^+$

field-type-name ::= *type-name* that is not `const`
type-name ::= see p30

number-type-name ::= `int` | `int8` | `int16` | `int32` | `int64` | `int128`
| `uns` | `uns8` | `uns16` | `uns32` | `uns64` | `uns128`
| `flt` | `flt16` | `flt32` | `flt64` | `flt128`
pointer-type-name ::= see p30

field-label ::= *data-label*
pointer-label ::= *field-label* beginning with zero or more '`.`'s followed by an '`@`'
target-label ::= *field-label* that is not a *pointer-label*
origin-label ::= *basic-name* not beginning with `@`
data-label ::= see p30
basic-name ::= see p30

field-dimension ::= `[` *dimension-size* `]`
dimension-size ::= *const-expression* with non-negative integer value
const-expression ::=
`const` valued *expression* evaluatable at compile-time, see 9.4 ^{p79}

subfield-declaration ::=
bit-range subfield-type-name subfield-label subfield-dimension*
subfield-type-name ::= *number-type-name* | `bool`
subfield-label ::= *target-label*
subfield-dimension ::= `[` *dimension-size* `]`

bit-range ::= `[` *onlybit* `]`
| `[` *highlowbits* `]`
| `[` *highbit* `-` *lowbit* `]`
onlybit ::= `const-expression` with non-negative integer value
highlowbits ::= *dit*+ `-` *dit*+ [this is a single lexeme]
highbit ::= `const-expression` with non-negative integer value
lowbit ::= `const-expression` with non-negative integer value
dit ::= see p34

alignment ::= `const-expression` with power of 2 integer value

The *type-subdeclarations* are processed in order. At the beginning of each, there is a align/pack switch value and an offset-in-bits integer. These determine the offset in bits of the next *field-declaration* field encountered relative to the beginning of each datum of the defined type being declared. There may be several *type-declarations* for the same defined type (see 9.1.1 ^{p55}), and at the beginning of the first the align/pack switch is set to align and the offset is set to zero.

If the align/pack switch is in the **align** position and the next *type-subdeclaration* is a *field-declaration*, the current offset will be incremented before becoming the offset of the field being declared. The increment will be just enough to make the offset an exact multiple of

the field's type's alignment. The alignment of a number type is its size in bits. The alignment of a defined type is the least common multiple of the alignment of any of its fields, which, since all alignments are powers of two, is the same as the largest alignment of any of the fields.

An '**align** *N*' sub-declaration behaves like an unnamed field of alignment *N* bits and zero length, and in addition sets the align/pack switch to 'align'. An '**align**' sub-declaration just sets the align/pack switch to 'align'.

A **pack** sub-declaration sets the align/pack switch to 'pack'.

An ***INCLUDE*** sub-declaration copies all the *type-subdeclarations* of the given defined type into the current sequence of *type-subdeclarations*.

A ***DEFERRED*** *type-declaration* declares a *defined-type-name* without declaring the definition of the named type. Such a *type-declaration* is typically used allow the *defined-type-name* to be used as the target type of a pointer.

The *defined-type-name* in a *type-declaration* is declared before the *type-declaration*'s sub-declarations are processed, so these sub-declarations may use the *defined-type-name* as a pointer target type.

A ***LABEL*** sub-declaration assigns the current offset to the given *origin-label* and provides the *origin-label* for use by subsequent `*ORIGIN*` sub-declarations. Each *origin-label* may be defined only once.

An '***ORIGIN*** *origin-label*' sub-declaration changes the current offset to that of the given *origin-label*. The *origin-label* must be defined by a preceding `*LABEL*` sub-declaration.

An '***OFFSET*** *integer-const-expression*' sub-declaration changes the current offset to an integer. Offsets set this way may be negative. Offsets are in bits.

9.1.1 Defined Type Expansions

A defined type can have just one or several *type-declarations*. The set of all its *type-declarations* is called the **expansion tree** of the defined type.

The set of fields and subfields of the defined type is the union of those defined by any *type-declaration* in the defined type's expansion tree. However these may not have unique names, as long as each name is unique in the scope of the *type-declaration* that defines the name (because two *type-declarations* in the expansion tree can have disjoint scopes).

All but one of the *type-declarations* in an expansion tree must be within the scope[p91] of another *type-declaration* in the expansion tree, and the expansion tree is a tree-graph in which each *type-declaration* X is the root of a subtree containing all *type-declarations* Y in the tree that are within the scope of X (or equivalently, that have a scope that is a subset of the scope of X). Thus the **parent** of a non-root Y is the *type-declaration* X in the expansion tree with the smallest scope that contains Y.

Non-root *type-declarations* in the expansion tree are called *expansions*, and the root *type-declaration* is called just the **root**.

A *defined-type-name* must be inside the scope of the root of its expansion tree to be visible, and hence usable.

A **leaf** in an expansion tree is a *type-declaration* that is not the parent of any other *type-declaration* in the expansion tree (equivalently, a leaf is a tree node with no children). An expansion tree is **linear** if it has only one leaf (equivalently, no tree node has more than one child). The sole leaf of a linear expansion tree is called a **linear leaf**.

Expansion trees are either internal or external.

Each *type-declaration* in an **internal expansion tree** that is not a linear leaf must end with the `***` sub-declaration, and all *type-declarations* in the tree must be in the same module or its bodies. However the *defined-type-name* may be external. If the tree has a linear leaf that does not have a `***` sub-declaration, the *defined-type-name* is **non-expandable** within the scope of the linear leaf *type-declaration*. Outside that scope, but within the scope of the tree root, the *defined-type-name* is said to be expandable.

Each *type-declaration* in an **external expansion tree** must end with the `*EXTERNAL*` sub-declaration. The *defined-type-name* is expandable whereever it is defined (i.e., within the scope of the tree root).

The final size and alignment of an expandable *defined-type-name* are not known until load time. Allocators use the load time size and alignment to allocate memory for a value of the type and then zero that memory. Thus the '`local`' allocator can allocate a datum of an expandable *defined-type-name* and return a pointer to the datum.

A *defined-type-name* must be non-expandable if it is used as a *type-name* in a variable declaration, or as a *field-type-name* in a *type-subdeclaration*, or as a *defined-type-name* in an `*INCLUDE*` *type-subdeclaration*. However an expandable *defined-type-name* can be used as the target of a pointer type used as a variable type or field type.

A *field-label* or *subfield-label* declared within a *type-declaration* may only be used within the scope of the *type-declaration* in which the *field-label* or *subfield-label* is declared, and may not be re-declared within this scope.

Similarly an *origin-label* may only be used only within its defining *type-declaration* X and within other *type-declarations* in the same expansion tree that are within the scope of X, and may not be re-declarated in the places it can be used.

Because it is possible for different *type-declarations* in an expansion tree to have disjoint scopes, a given label may refer to different things in these disjoint scopes: e.g., it might be a *field-label* in one scope and an *origin-label* in the other, or it might name completely different fields in the two scopes.

At the beginning of each *type-declaration* the align/pack switch is set to align, the offset is set to zero if the *type-declaration* is the root of its expansion tree, and the offset is undefined and

must be set by an *ORIGIN* or *OFFSET* statement before it is used if the *type-declaration* is a non-root.

LABEL and *ORIGIN* sub-declarations are typically used to overlay sections of a defined type's value, and create what in other languages are union types. Care must be taken in using union values as both type-violations and unexpected field allocations can result.

An '***OFFSET* *SIZE***' sub-declaration changes the current offset to the size of values of the type being defined, as it is computed at the point where the *OFFSET* *SIZE* sub-declaration is encountered. This is just the maximum of all offset values previously computed in the current *type-declaration* and its ancestors in the expansion tree. The '***OFFSET* *SIZE***' sub-declaration is not allowed in a *type-declaration* if any of the ancestors of the *type-declaration* in its expansion tree has more than one child.

9.1.2 Type Fields

A field of a value of a user defined type is accessed by prepending '**.**' to the *field-label* to form a *member-name* in a *reference-expression* (see p38). An example is:

```
type my type:
    uns8    kind         // Object Kind
    flt     weight

my type X:
    // Within this block X is write-only.
    X.kind = HIPPOPOTAMUS
    X.weight = 152.34
uns8 kind = X.kind
flt weight = X.weight
```

If a *field-label* is a *pointer-label*, an associated *target-label* is declared consisting of the *pointer-label* minus its first '**@**'. The *target-label* references a virtual field consisting of the target value stored at the location pointed at by the *pointer-label* field's pointer. An example[6] is:

```
type list element:
    int value
    ap list element @after

// Make circular list.
//
ap list element @Y      // `@= local = *DEFERRED*' is implied
ap list element @X:     // `@= local' is implied
    X.value = 1
```

[6] As '**next**' is a keyword, we use '**after**' here.

```
    X.@after = @Y
ap list element @Y:
    Y.value = 2
    Y.@after = @X
//
// Now X.value == 1 and X.after.value == 2 and
// similarly Y.value == 2 and Y.after.value == 1,
// while X.@after == @Y and Y.@after == @X are
// pointers to local memory.
```

A field of the defined type can only be accessed by code in the scope of a *type-declaration* declaring the field.

An example is:

```
type my type : *DEFERRED*
ap *READ-WRITE* my type @X    // `@= local = *DEFERRED*' is implied
                    // Legal, my type members need not be declared.
                    // Size and alignment of my type values is
                    // computed at load time.  The allocated value
                    // will be zeroed.
ap ro my type @Y = @X  // Legal, only ap copied.
ap my type @Z:      // Legal, `@= local' is implied
    Z = X           // Legal, the value at @X is copied to
                    // the value at @Z.  However in this
                    // case the value is completely zero.

type my type:       // Definition of my type that was *DEFERRED*.
    *LABEL* origin  // `origin' is set to offset 0
    int I           // Offset of I is 0.
    ***
X.I = 55            // Legal, .I has been declared.  X is
                    // *READ-WRITE*.

type my type:       // Expansion of my type
    *ORIGIN* *SIZE*
    int J           // Now X.J == 0
    ***
X.J = 66            // Legal, .J has been declared.

type my type:       // Expansion of my type
    *ORIGIN* origin
    int K1
    int K2
```

```
// Now X.K1 == X.I == 55; X.K2 == X.J == 66; but you must know
// how offsets are assigned to believe this.
```

9.1.3 Type Subfields

Subfields are parts of the previously declared number type field. The bits occupied by a subfield are given by its *bit-range*, where bits are numbered 0, 1, ... from the low order end of numbers.

A subfield value may have fewer bits than the number-type of the subfield. For integer types, the value is the low order bits of the integer, with the high order bits added when the value is read by with adding 0 bits for unsigned integers or copies of the highest order bit for signed integers. For floating types, the value is missing low order mantissa bits, which are added as zeros. If a value outside the representable range is stored, it is not an error. Integer values are truncated, and floating values have low order mantissa bits dropped (there is no rounding). However, it is a compile error to have a floating type whose exponent part plus 1 mantissa bit cannot be stored in the subfield value.

Subfield-labels and *field-labels* have the same standing within *reference-expressions*. Both have associated *member-names* made by adding a single '`.`' to the beginning of the *field-label* or *subfield-label*. For example:

```
type my type:
    uns8     kind        // Object Kind
    [0] bool animal      // True if Animal
    [1] bool vegetable   // True if Vegetable
    flt      weight

my type X:
    // Within this block X is write-only.
    X.kind = HIPPOPOTAMUS
        // Also sets animal bit and clears vegetable bit.
    X.weight = 152.34
uns8 kind = X.kind
bool animal = X.animal
bool vegetable = X.vegetable
flt weight = X.weight
```

9.1.4 Type Dimensions

If a *field-declaration* with *field-label* `F` contains a single *field-dimension* `[`n`]` then n fields are allocated to ascending offsets, using zero padding if necessary to align all n fields. The labels

of these fields are `F[`i`]` for $0 \leq i < n$. If there is a subfield labeled `S` of the field, `S[`i`]` refers to the subfield in `F[`i`]`. For example:

```
type character attributes:
    uns8 [128]
    [0] bool is graphic

character attributes X:
    int i = 0
    while i < 128:
        X.is graphic[i] = 32 < i && i < 127
        next i = i + 1
bool line feed is graphic = X.is graphic [C#"<LF>"]
bool A is graphic = X.is graphic [C#"A"]
```

In a *field-declaration* two *field-dimensions* `[`$n1$`][`$n2$`]` is treated as syntactic sugar for `[`$n1$`*`$n2$`]` with `F[`$i1$`][`$i2$`]` being syntactic sugar for `F[`$i1$`*`$n2$`+`$i2$`]`. Similarly `[`$n1$`][`$n2$`][`$n3$`]` is syntactic sugar for `[`$n1$`*`$n2$`*`$n3$`]` with `F[`$i1$`][`$i2$`][`$i3$`]` being syntactic sugar for `F[`$i1$`*`$n2$`*`$n3$`*+`$i2$`*`$n2$`+`$i3$`]`. And so forth for any number of *field-dimensions*.

If a *subfield-declaration* with *subfield-label* `S` contains *subfield-dimension* `[`n`]` then n subfields are allocated to the containing field, starting with the bits designated by the *subfield-declaration*'s *bit-range* and adding the number of bits in the *bit-range* to each integer in the *bit-range* for each successive subfield. For example:

```
type hex digits:
    uns32
    [3-0] uns hex digit [8]

hex digits X:
    int i = 0
    while i < 8:
        X.hex digit[i] = i
        next i = i + 1
// Now X == X#"76543210"
```

A *subfield-declaration* with *subfield-label* S and more than one *subfield-dimension* is treated in the same manner as a *field-declaration* with more than one *field-dimension*. For example, `[`$n1$`][`$n2$`][`$n3$`]` is syntactic sugar for `[`$n1$`*`$n2$`*`$n3$`]` with `S[`$i1$`][`$i2$`][`$i3$`]` being syntactic sugar for `S[`$i1$`*`$n2$`*`$n3$`*+`$i2$`*`$n2$`+`$i3$`]`.

If a *field-declaration* with *field-label* `F` has a *field-dimension* and also a subfield with *subfield-label* `S`, then `S[`i`]` references the subfield in the field value `F[`i`]`. If in addition the subfield has a *subfield-dimension*, `S[`i`][`j`]` references the subfield selected by `[`j`]` in the field value `F[`i`]`.

In all cases '`][`' may be replaced by '`, `' (the space after the comma is required), so that, for example, '`[`*i*`][`*j*`]`' is equivalent to '`[`*i*`, `*j*`]`'.

9.1.5 Type Conversions

When the compiler is confronted with code such as:

```
T1 v1 = ...
T2 v2 = v1
```

where `T1` and `T2` are different types, the compiler just rewrites the code to:

```
T1 v1 = ...
T2 v2 = *IMPLICIT* *CONVERSION* ( v1 )
```

and compiles the rewritten code. If you define a function with the prototype

```
function T2 r = ma? *IMPLICIT* *CONVERSION* ( T1 v )
```

the compiler will used this function. Otherwise the compiler will try to chain implicit conversions together to get to a successful compile. Of course such chaining will only work if there is at least one `*IMPLICIT* *CONVERSION*` function with target type `T2`. If there is more than one such function, and none have argument type `T1`, ambiguity may lead to a compile error.

You can define `*IMPLICIT* *CONVERSION*` functions provided at least one of the two types `T1` and `T2` is user defined, and not builtin, and `T2` is not `const`. There are builtin `*IMPLICIT* *CONVERSION*` functions in which both types are builtin: see 18.2 p120.

The set of `*IMPLICIT* *CONVERSION*` functions defines a graph in which types are nodes and `*IMPLICIT* *CONVERSION*` functions are directed edges. This graph must be acyclic.

When defining an implicit conversion from type `T1` to type `T2`, each value of type `T1` should be exactly representable by a value of type `T2`. This rule should be followed, but is not checked by the compiler.

For types `T1` and `T2` you can also define an explicit conversion:

```
function T2 r = ma? T2 ( T1 v )
```

If the result may not properly represent the value `v`, you may wish to define instead an unchecked conversion:

```
function T2 r = ma? *UNCHECKED* ( T1 v )
```

The compiler will not allow you to define such functions if both `T1` and `T2` are builtin or if `T2` is `const`, but some such functions are builtin: see 18.3 p121.

The types `T1` and `T2` may also be pointer types: see p65. Or one may be a pointer-type and one a non-pointer type. However, `T2` cannot be `const`, and you cannot define your own `*IMPLICIT* *CONVERSION*` function is both types are builtin.

9.2 Pointer Type Declarations

A pointer type has an **associated data type** specified by a pointer type declaration. The syntax is:

pointer-type-declaration ::=
    `pointer type` *defined-pointer-type-name* **`is type`** *type-name*
pointer-type-name ::= see p30
type-name ::= see p30

An `*UNCHECKED*` conversion is implicitly defined from the associated data type the given pointer type, and an explicit conversion is implicitly defined in the other direction.

It is important that there be a 1-1 correspondence between pointer types and their associated data types. In particular, `int` must not be used as the associated data type of more than one pointer type. This is why associated data types are generally user defined types.

For example, the following are builtin:

```
pointer type dp is type std data for dp
pointer type ap is type std data for ap
type std data for dp:
    int address
type std data for ap:
    dp ro int @base
    int offset

function dp Q$1 T$1 @r = std *UNCHECKED* ( std data for dp ddp )
function ap Q$1 T$1 @r = std *UNCHECKED* ( std data for ap dap )
function std data for dp r = std data for dp ( dp Q$1 T$1 @ptr )
function std data for ap r = std data for ap ( ap Q$1 T$1 @ptr )
    // These functions just copy the argument value to the
    // result value changing the type of the value.  Here
    // Q$1 is a wild-card that matches any list of qualifier-names,
    // and T$1 is a wild-card that matches any type-name.

// This function enables implicit conversion of `dp ...' to
// `ap ...', where the latter has the constant 0 for a base
// and the dp value for its offset, provided the target is
// *GLOBAL*.
//
dp *GLOBAL* int std @zero    // value zero is initialized to 0
function std ap QG$1 T$1 @r = std *IMPLICIT* *CONVERSION*
        ( std dp QG$1 T$1 @p ):
    std data for dp ddp = std data for dp ( dp QG$1 T$1 @p )
```

```
        std data for ap dap:
            dap.@base  = std @zero
            dap.offset = ddp.address
        @r = *UNCHECKED* ( dap )

    // This function enables *UNCHECKED* conversion of `ap ...' to
    // `dp ...' where the latter is the sum of the base and offset
    // of the ap.
    //
    function std data for dp r = std *POINTER* *UNCHECKED* *CONVERSION*
            ( std data for ap dap ):
        std data for dp ddp:
            ddp.address = dap.base + dap.offset
        r = ddp

    // This function enables conversion of `ap ...' to `dp ...'
    // when an ap pointer is being used to access a value or
    // member or element of a value.  The dp is the sum of the
    // base and offset of the ap.
    //
    function std data for dp r = std *POINTER* *ACCESS* *CONVERSION*
            ( std data for ap dap ):
        std data for dp ddp:
            ddp.address = dap.base + dap.offset
        r = ddp
```

A *pointer-type-declaration* '`pointer type` P `is type` D' implicitly declares the functions:

```
function P Q$1 T$1 @r = ma? *UNCHECKED* ( D data )
function D r = D ( P Q$1 T$1 @ptr )
```

where $ma^?$ denotes the *module-abbreviation* of P, if any. These just copy values changing type.

Reading and writing values using a pointer of type P can be accomplished by the functions:

```
reference function T$1 r = ma? ( P QR$1 T$1 @p ) ".*"
reference function ma? ( P QW$1 T$1 @p ) ".*" = T$1 r
```

where *ma*, if present here, refers to the module in which P is defined.

Note that these are reference functions [p80]. A `.@` function, if it exists, overrides these functions: see below.

These functions can be defined after P is defined. They allow the pointer to be used to read a copy of the value pointed at, or write the value, but do not allow members or elements of the value to be accessed (members and elements of the copy may be accessed).

These functions are implicitly called when the target of a pointer variable is read or written. For example:

```
ap int @p:
    p = 5
        // This translates to:
        //   @p.* = 5
int x = p
  // This translates to:
  //   int x = @p.*
```

When `.*` is inserted into code in this way, it is inserted without any *module-abbreviation*, so if there are conflicting definitions in different imported modules there will be a compile error.

An alternative strategy is to convert a pointer of type $P1$ to a pointer of type $P2$ that allows members and elements to be accessed. Suppose we are given:

```
pointer type P1 is type D1
pointer type P2 is type D2
```

Then we can define a reference function with the prototype:

```
reference function P2 Q$1 T$1 @r = ma? ( P1 Q$1 T$1 @p ) ".@"
```

which converts a pointer of type $P1$ to a pointer of type $P2$, where $ma^?$, if present, refers to a module in which both $P1$ and $P2$ are defined. This conversion function is automatically called without any *module-abbreviation* if a value pointed at by a pointer of type $P1$ is to be accessed, in preference to calling the `.*` functions above. Then if $P2$ is `dp`, `ap`, `fp`, `av`, or `fv`, the $P2$ pointer can be used to not only read or write the value, but to also read or write members or elements of the value.

Since `@p.*` and `@p.@` are syntactically *reference-calls*[p81], these expressions can only match reference function prototypes, and such matches ignore the prototype result types, and use only the prototype argument types. Thus the inserted calls, which have no module abbreviation, can be matched only if there is at most one `".*"` function in the current context with wildcards `Q$1` and `T$1` for a given $P1$, and similarly for `".@"`.

Alternatively you can define a function with prototype

```
function D2 r = ma? *POINTER* *ACCESS* *CONVERSION* ( D1 data )
```

where *ma* refers to any module in which both $D1$ and $D2$ are defined. This implicitly defines the reference function:

```
reference function P2 Q$1 T$1 @r =
            ma? ( P1 Q$1 T$1 @p1 ) ".@":
    D1 d1 = D1 ( @p1 )
    D2 d2 = *POINTER* *ACCESS* *CONVERSION* ( d1 )
    @r = *UNCHECKED* ( d2 )
```

where *ma* is the same module as that of the *POINTER* *ACCESS* *CONVERSION* function, if any.

When the compiler is confronted with code such as:

```
P1 ... @p1 = ...
P2 ... @p2 = @p1
```

where P1 and P2 are different pointer types, the compiler just rewrites the code to:

```
P1 ... @p1 = ...
P2 ... @p2 = *IMPLICIT* *CONVERSION* ( @p1 )
```

and compiles the rewritten code. If you define a function with the prototype

```
function P2 Q$1 T$1 @r =
            ma? *IMPLICIT* *CONVERSION* ( P1 Q$1 T$1 @p )
```

the compiler will used this function. Otherwise the compiler will try to chain pointer implicit conversions together to get to a successful compile. Of course such chaining will only work if there is at least one *IMPLICIT* *CONVERSION* function with target type P2 .... If there is more than one such function, and none have argument type P1 ..., ambiguity may lead to a compile error.

You can define a function with the above prototype, or you can define

```
function D2 r = ma? *POINTER* *IMPLICIT* *CONVERSION* ( D1 data )
```

which implicitly defines the function:

```
function P2 Q$1 T$1 @r =
            ma? *IMPLICIT* *CONVERSION* ( P1 Q$1 T$1 @p1 ):
    D1 d1 = D1 ( @p1 )
    D2 dd = *POINTER* *IMPLICIT* *CONVERSION* ( d1 )
    @r = *UNCHECKED* ( d2 )
```

You may want the conversion from $P1$ to $P2$ to be explicit and unchecked instead of implied. This can be achieved by defining:

```
function P2 Q$1 T$1 @r = ma? *UNCHECKED* ( P1 Q$1 T$1 @p )
```

Alternatively you can define a function with prototype

```
function D2 r =ma? *POINTER* *UNCHECKED* *CONVERSION* ( D1 data )
```

which implicitly defines the function:

```
function P2 Q$1 T$1 @r = ma? *UNCHECKED* ( P1 Q$1 T$1 @p1 ):
    D1 d1 = D1 ( @p1 )
    D2 dd = *POINTER* *UNCHECKED* *CONVERSION* ( d1 )
    @r = *UNCHECKED* ( d2 )
```

The example at the beginning of this section contains examples of *POINTER* ... *CONVERSION* functions.

In the above function prototypes you can use different wildcard names, e.g, `T$XXX` instead of `T$1`. You can also use non-wildcards, e.g., `int` instead of `T$1`.

An example implementing a new pointer type is:

```
type file:
    *READ-WRITE* av uns8 @name
    . . . . .
av *READ-WRITE* file @files @= global [1000]
ap *READ-WRITE* int @number of files @= global

// Implement a file descriptor (fd) that addresses a file
// in files.  The fd contains an index and addresses
// files[index].

type data for fd:
    int index

pointer type fd is type data for fd

reference function ap Q$1 file @r = ( fd Q$1 file @p ) .@:
    data for fd d = data for fd ( @p )
    ap file *READ_WRITE* @f = @files[d.index]
    @r = @f  // Implicitly converts *READ-WRITE* to Q$1

function fd *READ-WRITE* file @r = allocate fd:
    data for fd d:
        d.index = number of files
    @r = *UNCHECKED* ( d )
    number of files = number of files + 1

fd *READ-WRITE* file @f @= allocate fd
f.@name = ...
. . . . .
av uns8 @n = f.@name
. . . . .
```

9.3 Inline Function Declarations

The syntax of a function declaration is:

function-declaration ::= *function-prototype* `:` *statement*$^{+}$
| *function-prototype* `: *DEFERRED*`

function-prototype ::=
`function` *prototype-result* `=` *module-abbreviation*$^{?}$ *prototype-pattern*
| `function` *module-abbreviation*$^{?}$ *prototype-pattern*
| `function` *module-abbreviation*$^{?}$ *prototype-pattern* `=` *input-variable-list*

prototype-result ::=
prototype-result-list
| `macro` *variable-name*

prototype-result-list ::=
prototype-result-declaration { `,` *prototype-result-declaration* }*

prototype-result-declaration
::= *result-variable-declaration*
| *next-variable-declaration*

module-abbreviation ::= see p30

result-variable-declaration ::= see p42

next-variable-declaration ::= see p42

input-variable-list ::=
prototype-argument-declaration { `,` *prototype-argument-declaration* }*

prototype-argument-declaration
::= *result-variable-declaration* { `?=` *default-value* }$^{?}$
| `bool` *variable-name* { `?=` *default-value* }$^{?}$
| `macro` *variable-name* { `?=` *default-value* }$^{?}$
| *result-variable-declaration* `!=` *required-value*

default-value ::= *expression*

required-value ::= *const-expression*

expression ::= see p23

const-expression ::=
`const` valued *expression* evaluatable at compile-time, see 9.4 ^{p79}

prototype-pattern ::= *parenthesized-pattern-argument-list** *pattern-term*$^{+}$

pattern-term ::= *function-term-name* *pattern-argument-list**

function-term-name ::= see p30

function-variable-name ::=
variable-name N which also has the form:
module-abbreviation$^{?}$ *function-term-name*
and that appears in a *function-prototype* of the form:
`function` N `= ...`

pattern-argument-list

::= `(` *prototype-argument-declaration* { `,` *prototype-argument-declaration* }* `)`
| `[` *prototype-argument-declaration* { `,` *prototype-argument-declaration* }* `]`

parenthesized-pattern-argument-list ::=

pattern-argument-list with parentheses `( )` (and not square brackets `[ ]`)

- A *prototype-pattern function-term-name* must not be an initial segment of any other *function-term-name* in the same *prototype-pattern*.
- *Function-term-names* (in a *prototype-pattern*) may not be *member-names*[p30] (compare with *reference-function-declaration*[p81]).
- *Pattern-argument-lists* appearing before the first *function-term-name* may not use square `[ ]` brackets (compare with *reference-function-declaration*[p81]).
- Result and argument *variable-names* in a *function-prototype* must not begin with a *module-abbreviation*.
- For a *prototype-result-declaration* of the form '**next** *v*', *v* must be the *variable-name* in a *prototype-argument-declaration* of the form '... *type-name v*', and any actual argument associated to the *prototype-argument-declaration* by some *function-call* must be a *reference-expression*.
- Result and argument *variable-names* in a *function-prototype* must be distinct, with an exception for the previous note.
- The first *prototype-argument-declaration* in an *input-variable-list* must not have a *default-value*.
- In a *pattern-argument-list* or *input-variable-list* a *prototype-argument-declaration* with no *default-value* cannot follow a *prototype-argument-declaration* with a *default-value*.
- A wild-card[p31] name of the form `T$...` is treated in a *function-prototype* as a *type-name*. A wild-card name of the form `P$...` is treated as a *pointer-type-name*. A wild-card name of the form `Q...$...` is treated as a *qualifier-name* and must not be combined with other *qualifier-names* in the same *result-variable-declaration* or *prototype-argument-declaration*.
- A `*DEFERRED*` *function-declaration* may not have a '`macro` *variable-name*' *prototype-result*.

An example of an inline function declaration and an inline function call is:

```
function F ( int x ?= 5 ) G ( int y ) H ( int z ?= 7 ) I ( int w ):
    . . . . . .
F I ( 8 ) G ( 6 )    // Equivalent to F ( 5 ) G ( 6 ) H ( 7 ) I ( 8 )
```

The *function-term-names* in the declaration are matched to those in the call, but need not have the same order in the call, except for the first *function-term-name* which must be the same in the declaration and the call. Thus the *call-terms* of the call are re-ordered to match

the order of the *pattern-terms* of the declaration. If one of the *pattern-terms* is omitted in the call, but its arguments have *default-values* the *pattern-term* with its *default-values* will be inserted into the call (here `H ( 7 )` is inserted). Similarly with an *argument-list* that is omitted (here `( 5 )` is inserted).

An example containing an *input-variable-list* is:

```
function F [ int x ] = int y,  int z ?= 5:
    . . . . . .
F[10] = 6
```

which is treated as if `=` were a *function-term-name* that must be the last such in the call, and the comma separated values after `=` in the call and *prototype-argument-declarations* after `=` in the prototype were surrounded by parentheses `( )`. Note that for an argument list in the prototype to match an argument list in the call, both must be surrounded by the same brackets; either both have `( )` or both have `[ ]`, except that implied parentheses in the call are treated as `( )` during matching.

As an example of this last,

```
r = x + y
```

is treated for matching purposes is as

```
( r ) "=" ( ( x ) "+" ( y ) )
```

Note that *quoted-marks* and *quoted-separators* in *function-term-names* may appear with or without quotes in *call-term-names.*[p40] Thus we have the example:

```
function int z = ( int x ) "@@" ( int y ):
    . . . . . .
int v = 5 @@ 6       // Legal
int w = 5 "@@" 6     // Legal
```

However, quoting an operator will cause it to be not recognized as an operator. For example:

`x + y * z` parses as `{ { "x" }, "+", { { "y" }, "*", { "z" } } }`

whereas

`x "+" y * z` parses as `{ { "x", "+", "y" }, "*", { "z" } }`

In the latter, `{ "x", "+", "y" }` cannot be recognized as a *function-call* because the arguments `"x"` and `"y"` are not bracketed (i.e., are not `{ "x" }` and `{ "y" }`), and will also not be recognized as a *reference-expression* (because `"+"` cannot be in a *variable-name* or begin a *member-name*) or *constant*.

A *pattern-term* with the syntax:

boolean-pattern-term ::=
function-term-name **`( bool`** *variable-name* **`?=`** *default-value* **`)`**

triggers special syntax in a call that matches the prototype. In the call:

function-term-name	is equivalent to	*function-term-name* `( TRUE )`
no *function-term-name*	is equivalent to	*function-term-name* `( FALSE )`
not *function-term-name*	is equivalent to	*function-term-name* `( FALSE )`
omitted *function-term-name*	is equivalent to	*function-term-name* `(` *default-value* `)`

Thus the example:

```
function F ( int x ) OPTION ( bool y ?= TRUE )
    . . . . . .
F ( 5 )           // Equivalent to F ( 5 ) OPTION ( TRUE )
F ( 5 ) OPTION    // Equivalent to F ( 5 ) OPTION ( TRUE )
F ( 5 ) no OPTION // Equivalent to F ( 5 ) OPTION ( FALSE )
```

A function prototype result may be named '**next** *v*' if the prototype has an argument named *v*, in which case a *function-call* matching the prototype must match the prototype argument *v* to a *reference-expression e* that is not a constant and does not have the form '**next** *variable-name*'. Then the *assignment-result* corresponding to the prototype '**next** *v*' may be omitted, and will be taken to be '**next** *e*' if *e* is a *variable-name* naming a stack **co** variable for which '**next** *e*' is legal, and will otherwise be taken to be simply *e*.

For example:

```
function next x = inc ( int x ):
    next x = x + 1
. . . . . .
int y = ...
inc ( y )  // Equivalent to `next y = y + 1'.
ap *READ-WRITE* int @z:
    z = ...
inc ( z )  // Equivalent to `z = z + 1'.
```

A *prototype-argument-declaration* with the syntax:

prototype-argument-declaration ::=
macro *variable-name* { **?=** *default-value* }$^{?}$

causes the argument value in a call to be the parse of the actual argument, which is a `const` value. If the *default-value* is used, it is evaluated as a *const-expression*.

If a *required-value* is given in a prototype, the call must have an equal `const` valued actual argument value in order for the call to match the prototype. Note that the argument variable type need not be `const`, as `const` values can be converted to run-time values: equality will be checked after conversion to the argument type. Matches to prototypes with more *required-values* are preferred over matches to prototypes with less *required-values*. Thus the example:

```
function F ( int x ) G ( int y != 5 ):  // First F declaration
    . . . . . .
function F ( int x ) G ( int y ?= 5 ):  // Second F declaration
```

```
    . . . . . .
int z = 5
F ( 8 )            // Matches only second F declaration
F ( 8 ) G ( 5 )    // Matches preferred first F declaration
F ( 8 ) G ( 6 )    // Matches only second F declaration
F ( 8 ) G ( z )    // Matches only second F declaration
                   // (z is not const valued).
```

A *prototype-result* of the form '`macro` *variable-name*' makes the function a **macro function**. When called, *variable-name* must be set to a `const` value that is a parsed expression which replaces the *function-call*.

A '`*DEFERRED*`' *function-declaration* permits inline functions defined between it and a later non-`*DEFERRED*` companion *function-declaration* to call the function. An example is:

```
function F2 ( const i): *DEFERRED*
function F1 ( const i):
    if i != 0:
        <do F1 thing>
        F2 ( i - 1 )

F1 ( 1 )   // Compile Error: Call F2(0) cannot be expanded.

function F2 ( const i )
    if i != 0:
        <do F2 thing>
        F1 ( i - 1 )

F1 ( 5 )   // Legal, expands to:
           //    <do F1 thing>
           //    <do F2 thing>
           //    <do F1 thing>
           //    <do F2 thing>
           //    <do F1 thing>
           // Would not be legal if the deferred
           // function declaration were omitted,
           // as then no F2 declaration would be
           // visible to the statements of F1.
```

Here the statements of `F1` compile in the context of the declaration of `F1` and need the `*DEFERRED*` declaration of `F2` in that context to enable these statements to call `F2`. Given that a call is enabled, the situation where the statements of `F2` are provided later is permitted.

A `*DEFERRED*` declaration and its companion non-`*DEFERRED*` declaration must have identical prototypes, except:

- Default values must appear only in the `*DEFERRED*` declaration and are omitted in the companion.
- Required values need not have the same computing expressions in the two declarations, but these expressions must evaluate to the same values. Note that the two expressions are each evaluated where their prototype is declared, and therefore are evaluated in two different contexts.

A `*DEFERRED*` inline *function-declaration* may have at most one companion.

The prototype of an inline *function-declaration* is visible to the *statements* of that same declaration, and therefore an inline function can call itself without having any `*DEFERRED*` companion.

Recursion in inline function calls must be limited by `const` variables such as the counter `i` in the above example, for if it is not, there will be a compile error when the compiler decides the inline nesting is too deep or the code generated by one statement is too much.

A macro function cannot be `*DEFERRED*`.

9.3.1 Inline Call-Prototype Matching

Each *function-call* in a statement must be matched to a single *function-prototype*, else compilation of the statement fails with a compile error.

This section applies to matching non-*const-expressions* and non-*reference-calls*. The rules of this section are modified when matching *const-expressions* assording to section 9.4 [p79] and *reference-calls* according to section 9.5.1 [p83]. Both *const-expression* and *reference-call* analysis is a bottom-up process that must be done before applying the rules of this section. In particular, this section assumes that all *const-expressions* have been replaced by `const` value, and that the types of all *reference-expressions* are know.

Conceptually, a **matching map** is built for each subexpression of the statement that is not in a subblock within the statement (i.e., not within a substatement of the statement). This maps target-types to one of:

a function prototype	the subexpression is a function call that matches the function prototype and only that prototype
`*FAIL*`	the subexpression is a function call that matches zero or more than one function prototypes
`*EXACT*`	the subexpression is a reference expression or *const-expression* whose type is exactly the target-type

If the subexpression is a *reference-expression* value e whose type is not exactly the target-type, the compiler replaces e by

IMPLICIT *CONVERSION* (*e*)

and attempts to find a function prototype for this revised expression.

The implications of this last are brought out by the following example:

```
int x = 5.5 - 1.5     // Sets x = 4
int y = 5.5 - 1.0     // Fails trying to set y = 4.5
```

The right sides of these statements are *const-expressions* that evaluate to `4.0` and `4.5` respectively before the rules of this section are applied. These `const` values are reference expressions and therefore the statements become:

```
int x = *IMPLICIT* *CONVERSION* ( 4.0 )  // No error.
int y = *IMPLICIT* *CONVERSION* ( 4.5 )  // Compile error.
```

In both cases the builtin function:

```
int x = std *IMPLICIT* *CONVERSION* ( const y )
```

is matched to its *function-call*, but in the first case this function compiles to run-time code that returns the `int` value `4`, while in the second case this function announces a compile error because it cannot convert `const` value `4.5` to an `int` value.

Note that matching maps are not used in the modified versions of this section's algorithm that are used for *const-expressions* and *reference-expressions*.

The matching map for a particular subexpression in a statement does not depend on the parts of the statement that contain the subexpression, but only depends on the subexpression. So matching maps could be built working bottom up in the statement, i.e., starting with innermost subexpressions. But as most entries in a subexpression matching map will be unused, its entries are instead computed as needed, and memoized in the subexpression matching map to avoid recomputation. Thus matching is a top down process, starting with the outermost function call and working downward through argument subexpressions, computing matching map entries as needed.

The *function-call* to *function-prototype* matching algorithm therefore inputs just the *function-call* and its target type, without any argument subexpression details other than the matching maps of each of its arguments. For a function call that is the right side of the `=` in a *call-assignment-statement* there may be zero, or more than one target type, but no matching map is required, as the target types are dictated by the *call-assignment-statement* left side.

In this context, **call-prototype matching** is done as follows:

1. If the call begins with a *module-abbreviation* and the *prototype-pattern* either does not begin with a *module-abbreviation*, or begins with one that names a different module from that of the call, the call-prototype match fails.

   If the *function-call* and *prototype-pattern* both begin with a *module-abbreviation* (identifying the same module), or if neither begins with a *module-abbreviation* (both are non-external), the match is marked as **module proficient**.

2. The *function-term-names* in the prototype are matched to *call-term-names* in the call. To match, the names must be identical, except that quotes in prototype *quoted-marks* and *quoted-separators* may be (but need not be) removed in the call (thus prototype `"+"` matches call `+` and also call `"+"`).

   The match is made by scanning the call from left-to-right while identifying sequences of lexemes that match *function-term-names* in the prototype. After identifying a name, the scan skips to just after the name. If several names match at the same position, the longest is chosen. There is no backup; once a name match is made, it is never unmade. The scan may match a single prototype name to several points in the call, but if this happens, the call-prototype match fails. If the first prototype name fails to match the first call name, the call-prototype match fails, but otherwise names may be matched in any order.

3. The *function-term-names* found in the call are used to determine the extent of *call-terms* in the call. For starters, each *call-term* consists of its *call-term-name* and everything following up to the next *call-term-name* or end of call. If the prototype begins with *pattern-argument-lists*, the situation is treated as if both prototype and call began with identical virtual *term-names*.

   Next if a *call-term-name* is matched to a *boolean-pattern-term function-term-name* and if its *call-term* has no *call-argument-lists*, then if the preceding *call-term* ends in '**`no`**' or '**`not`**', this last is removed from the preceding *call-term* and '`(FALSE)`' is appended to the current *call-term*, while otherwise '`(TRUE)`' is appended to the current *call-term*.

4. A *call-term* must match its corresponding prototype *pattern-term* according the rules that follow. Failure of any call-prototype term match causes the prototype-call match to fail.

5. For a *call-term* to match its corresponding *pattern-term*, both must have the same number of *argument-lists*, the same brackets (either `( )` or `[ ]`) for corresponding *argument-lists*, and the same number of arguments in corresponding *argument-lists*, after the *call-term* has been **adjusted**. The following are permitted adjustments.

   For every *pattern-term* that has no corresponding *call-term* (because its *function-term-name* was not found in the call), a *call-term* consisting of just the *pattern-term*'s *function-term-name* is appended to the *function-call*. After this the *call-terms* are re-ordered so their order matches that of their associated *pattern-terms*.

   A *call-term argument-list* with implied parentheses is treated as if it had `( )` parentheses.

   If in a left-to-right scan of a *call-term*, a *call-argument-list* with `( )` is expected but no `(` is found, and instead a *call-argument-list* with `[ ]` or the end of the *call-term* is found, the empty list `()` is inserted.

   Note that *argument-lists* with `[ ]` brackets cannot be omitted or have their `[ ]` brackets omitted.

If a *call-argument-list* is shorter than the corresponding *pattern-argument-list*, and all omitted arguments at the end of the *call-argument-list* have *default-values* in the *pattern-argument-list*, the *default-values* corresponding to the omitted arguments are added to the end of the *call-argument-list*. The *default-values* are compiled in the context of the prototype and not the context of the call: see p94.

At this point the *pattern-argument-lists* in the prototype *pattern-term* must match in order all the *call-argument-lists* in the *call-term*, both in type of brackets (either '`( )`' or '`[ ]`') and in number of arguments, else the call-prototype match fails.

6. If all the above is successful, then *actual-arguments* in the call are matched to corresponding *prototype-argument-declarations* in the prototype according to the rules that follow. Failure of any of these matches causes the call-prototype match to fail.

7. This section is skipped for macro functions (functions whose *prototype-result* has the form '`macro` *variable-name*').

   If the *function-call* is the right side (part after `=`) in a *call-assignment-statement*, the number of *assignment-results* in the *call-assignment-statement* must not be greater than the number of *prototype-result-variable-declarations*, else the call-prototype match fails.

   If the *function-call* is not the right side in a *call-assignment-statement*, the situation is treated as if it were the right side of a *call-assignment-statement* whose left side consists of the call's target type followed by a virtual *variable-name*.

   The *assignment-results* are matched to the *prototype-result-declarations* going from left to right. The type of each *prototype-result-declaration* is then matched to the type of its matching *assignment-result*.

   Any wildcards[p31] in a prototype result type are filled in from the information in the corresponding assignment target type. If a wildcard gets more than one value from this process, the call-prototype match fails.

   Then if any prototype result type is not identical to its corresponding assignment target type, the call-prototype match fails (i.e., there is no implicit conversion of function result types).

8. *Prototype-argument-declarations* are matched to an *actual-arguments* and processed left to right.

   If a *prototype-argument-declaration* PAD is matched to an *actual-argument* AA and PAD has a wildcard, AA must be *reference-expression*, else the call-prototype match fails. The wildcard is assigned from information in the *reference-expression*'s type (*reference-expression* types are computed bottom-up, ignoring the part of a *statement* containing the *reference-expression*: see 9.5.1[p83]). Note that a constant that replaces a *const-expression* is itself a *reference-expression* of type `const`.

   If different values are assigned to the same wildcard by this process (by different prototype-actual argument matches), the call-prototype match fails.

9. If a *prototype-argument-declaration* has a *required-value*, its matching *actual-argument* must be a *const-expression* with a value equal to the *required-value*, after both are converted to the argument type specified by the *prototype-argument-declaration*, else the call-prototype match fails. Note that the argument type need not itself be `const`.

   If the argument type is `const` and the values being compared are maps, the maps must be identical (they are compared as pointers and not as lists of elements).

10. Matches between *prototype-argument-declaration* types and acceptable *actual-arguments* subexpression target types are then checked, except for `macro` arguments (that require no run-time typing).

    If any *prototype-argument-declaration* type is mapped to `*FAIL*` by the corresponding *actual-argument* matching map, the call-prototype match fails.

    If any is mapped to `*EXACT*`, the call-prototype map is marked as **conversion proficient**. Note that only a single argument needs to be `*EXACT*` in order for the entire call-prototype match to be marked conversion proficient.

If after applying these rules each match is assigned a rank equal to the sum of:

- the number of required arguments the prototype has
- the negative of the number of `T$...` wildcards the prototype has
- a very large number if the match is conversion proficient
- an even larger number if the match is module proficient

Then if there is a single match with maximum rank, that match is accepted, and otherwise all matches fail and there is a compile error. Note that the rank can be negative.

The following are examples using the builtin prototypes

```
function N r = std (N v1) "+" ( N v2 )
function bool r = std (N v1) "==" ( N v2 )
function flt64 r = std flt64 (N v1)
```

which exist for every builtin number type `N`.

```
int32 x = ...
flt64 r1 = x + 5
    // Target type flt64 selects N = flt64 for "+".
bool b1 = ( x == 5 )
    // Target type bool and *EXACT* argument x select
    // N = int32 for "==".
flt64 y = 5.5
bool b2 = ( x == y )
```

```
        // Target type bool and *EXACT* argument y select
        // N = flt64 for "==";  N = int32 version of "=="
        // fails because int32 r = *IMPLICIT* *CONVERSION* (y)
        // *FAIL*s.
    bool b2 = ( flt64 ( x ) == 5 )
        // All flt64 functions have flt64 result type,
        // and as there is no implicit conversion of flt64
        // function results, "==" must have N = flt64.
        // The only flt64 function with *EXACT* argument
        // is the one with N = int32.
```

9.3.2 Macro Functions and Statements

A **macro function** is an inline function whose *prototype-result* has the form '**macro** *variable-name*'.

Call-prototype matching for macro functions ignores any target type, and ignores actual argument types that match **macro** prototype arguments, so *function-term-names* of macro function often need to be more unique than they need to be for non-macro inline functions.

If call-prototype matching for a *function-call* matches the prototype of a macro function declaration, the macro function is immediately called, and its result, which must be a `const` parsed expression, replaces the *function-call*. This new *function-call* then undergoes call-prototype matching as if it had been written into the original code.

If a macro function has a '**macro** *V*' argument, then during execution of the function *V* is assigned the parsed actual argument as a **const** value. For example:

```
function macro r = inc ( macro v ):
    return { v, "+=", { 1 } }
ap *READ-WRITE* int @x:
    x = 5
inc ( x ) // Same as {{"x"}, "+=", {1}}.
    // Now x == 6.
```

During execution, a macro function cannot access non-**macro** prototype arguments. However, a macro function can use special builtin function:

`function std const r = name of ( const variable name )`

> Must be called inside an inline function where '`variable name`' is a *variable name* declared in a non-macro *prototype-argument-declaration*.
>
> Returns a *variable-name* of the form `V$...` that can be used inside run-time code produced by the inline function to reference the actual argument associated with the *prototype-argument-declaration* of the '`variable name`'.

If the *prototype-argument-declaration* has the form '*TTT V*' where *TTT* consists of runtime types and qualifiers and *V* is a *variable-name*, and if

W `= name of ( "`*V*`" )`,

is executed in the inline function, and if `"V$..."` is returned as the value of *W*, then code of the form:

TTT `V$...=` ***actual-argument***

is executed at runtime just before any runtime code generated by the *function-call* to the inline function. `V$...` will be a unique variable name not duplicated in any other code within the same compilation.

For example:

```
function macro r = inc ( macro v, int w ):
    const W = name of ( "w" )
    return { v, "+=", { W } }
ap *READ-WRITE* int @x:
    x = 5
inc ( x, 10 ) // Same as:
              //     int V$1 = 10
              //     {{"x"}, "+=", {V$1}}.
    // Now x == 15.
```

Macro functions can also be used to create new statements that combine a macro function with an indented paragraph:

macro-statement

::= *assignment-result* { `,` *assignment-result* }* `=` *function-call* `:`
statement*
| *function-call* `:`
statement*

- The *function-call* must call a macro function.

Macro-statements may be used to translate user defined declarations into builtin declarations, or to translate user defined loop or other flow control constructs into builtin executable statements.

When the macro function is executing in a *macro-statement* the entire parsed *macro-statement* is the value of the **`*STATEMENT*`** const variable. The macro function can output a edited version of this parsed *macro-statement* in the `*INCLUSION*` variable, as described in the section on inclusions: see p88. The macro function returns the `const` value `NONE` which signals that the compilation of the *macro-statement* is complete. Therefore the *macro-statement* does nothing but set the `*INCLUSION*` variable.

Alternatively, the macro function may return an *expression* which must be a *function-call* to another macro function which will be executed in turn. If multiple macro functions are

thusly called by a *macro-statement*, each macro function execution adds to or modifies the `*INCLUSION*` variable value computed by the previous macro function execution.

An example is:

```
// Iterate for v = 0, 1, 2, ..., limit-1.
//
function macro r = for ( macro V, macro LIMIT ):
    const P = *STATEMENT*[1] // Indented paragraph
    *INCLUDE* (V, LIMIT, P):
        int V = 0
        while V < ( LIMIT ):
            P
            next V += 1
    r = NONE

// Compute sum = 1 + 2 + ... + 15
//
int sum = 0
for ( x, 15 ):
    next sum += x + 1

// The above for ... statement is expanded to:
//
//     int x = 0
//     while x < ( 15 ):
//         next sum += x + 1
//         next x += 1
```

9.4 `Const`-Expressions

A ***const-expression*** is an *expression* that evaluates at compile time to a `const` value.

More specifically, *const-expressions* are discovered by a bottom up transversal of a statement, and when discovered are replaced by their `const` values.

During this bottom up transversal, an *expression* is deemed to be a *const-expression* if and only if it is a `const` constant, a *reference-expression* evaluating to a *const* value, or a *function-call* such that:

1. All call arguments are `const` constant values (after replacement of *const-subexpressions* by their values).

2. An unambiguous function prototype is found matching the *function-call*, using the

algorithm of 9.3.1 [p72], where the only prototypes considered are those for which:

(a) all prototype arguments and results have the `const` type

(b) if the *function-call* to be matched is not the right side of a *call-assignment-statement* [p46], there is at least one result, and the first result will be the value of the *function-call*

(c) if the *function-call* to be matched is the right side of a *call-assignment-statement*, the prototype has at least as many prototype results as the *call-assignment-statement* has *assignment-results*

When a *const-expression* is found by the bottom-up transversal, it is immediately evaluated at compile time. If the *const-expression* is not the right side of a *call-assigment-statement*, its value replaces the *const-expression*. If the *const-expression* is the right side of a *call-assigment-statement*, the *call-assignment-statement* is converted to an *expression-assignment-statement* with a list of the proper number of `const` results of the *const-expression* becoming the *expression-list*.

It is a compile error if a function called during this evaluation attempts to execute run-time code: e.g., if it attempts to allocate a variable to the run-time stack. The function may, however, use the `*INCLUSION*` variable (see 10 [p88]) to generate run-time code that will be compiled outside the context of the function execution - this code will be compiled immediately after code generated by compiling the statement containing the *const-expression*.

9.5 Reference Function Declarations

A **reference function** is a function that can be called as part of evaluating a *reference-expression*. The part is a *reference-call* that is either the entire *reference-expression* or is a base for the rest of the *reference-expression*.

The prototype of a *reference-function* begins with an optional *module-abbreviation* followed by a parenthesized single argument declaration that is matched to the base of the *reference-call*. The type of the *reference-call* is computed from the type of this base - there is no target type.

A *reference-call* is built by extending its base with either a `[ ]` bracketed *index* or a *member-name* optionally followed by other arguments and call terms. The base may be parenthesized, and if it is, may be preceded by a *module-abbreviation*.

Reference-calls are similar to (non-reference) *function-calls*, *reference-function-declarations* are similar to inline *function-declarations*, and reference call-prototype matching is similar to non-reference call-prototype matching, but there are significant differences.

The syntax of a *reference-call* is:

reference-call ::=
ma$^{?}$ *reference-expression member-name*
parenthesized-call-argument-list*
parenthesized-call-term*
ma$^{?}$ *reference-expression square-bracketed-call-argument-list*$^{+}$
parenthesized-call-term*

reerence-expression ::= see p37

call-term ::= see p40

parenthesized-call-term ::= *call-term* without `[ ]` brackets

call-argument-list ::= see p40

parenthesized-call-argument-list ::= *call-argument-list* with `( )` parentheses

square-bracketed-call-argument-list ::= *call-argument-list* with `[ ]` brackets

ma ::= *module-abbreviation* [see p30]

- If the *reference-call* begins with a *module-abbreviation* (*ma*), the *reference-expression* must be parenthesized. Otherwise the *module-abbreviation* will be parsed as part of the *reference-expression*.
- *Call-argument-lists* with square brackets are restricted to appearing just after the *reference-expression*.
- The *member-name* in a *reference-call* may not be abbreviated.

The syntax of a *reference-function-declaration* is:

reference-function-declaration ::=
reference-function-prototype `:`
statement^{+}
| *reference-function-prototype* `: *DEFERRED*`

reference-function-prototype
::= `reference function`
result-variable-declaration `=`
module-abbreviation$^{?}$ *reference-prototype-pattern*
| `reference function`
module-abbreviation$^{?}$ *reference-prototype-pattern* `=`
input-variable-list

module-abbreviation ::= see p30

result-variable-declaration ::= see p42

reference-prototype-pattern

::= *reference-pattern-argument* *member-name* `()`$^{?}$
parenthesized-pattern-argument-list*
parenthesized-pattern-term*
| *reference-pattern-argument* *square-bracketed-pattern-argument-list*$^{+}$
parenthesized-pattern-term*

reference-pattern-argument ::= `(` *result-variable-declaration* `)`

pattern-term ::= see p67

parenthesized-pattern-term ::= *pattern-term* without `[ ]` brackets

function-term-name ::= see p30

member-name ::= see p30

pattern-argument-list ::= see p68

parenthesized-pattern-argument-list ::=
pattern-argument-list with `( )` parentheses

square-bracketed-pattern-argument-list ::=
pattern-argument-list with square brackets `[ ]` (and not parentheses `( )`)

- The rules for a *function-declaration* (p68) apply where applicable to a *reference-function-declaration*, with exceptions as indicated here.
- The *reference-pattern-argument* must be followed by a *member-name*^{p30} or square-bracketed *pattern-argument-lists*.
- The first actual argument in a *reference-call* is a smaller *reference-expression* that has already been deliminated and which is surrounded in the *reference-call* by parentheses `( )` that may be implied and not explicit, unless the *reference-call* starts with a *module-abbreviation*.
- The *reference-pattern-argument* in the prototype cannot contain default or required values or be a `macro` argument.
- A *member-name* cannot be followed immediately by a *pattern-term*; a `()` must be placed immediately after the *member-name* to indicate the boundary between it and the following *pattern-term*. This is the only situation in which `()` should be used.
- *Pattern-argument-lists* with square brackets are restricted to appearing just after the *reference-pattern-argument*.

When a *reference-call* computes a pointer, the pointer target qualifiers are computed in the same manner as for builtin *reference-expressions* (see p38) using the base *reference-expression* qualifiers as the container qualifiers and the protype *result-variable-declaration* qualifiers as field qualifiers.

9.5.1 Reference Call-Prototype Matching

Two major differences between *reference-call*-prototype matching and inline *function-call*-prototype matching are:

- The end of the *reference-call* may be before the end of its containing *reference-expression* and must be determined by special rules, while the end of a *function-call* is the end of its containing *expression*.
- There is no target type for a *reference-call*, and implicit conversion cannot be applied to its first argument *reference-expression*.

With this in mind, the steps of the inline call-prototype matching algorithm of section 9.3.1 [p72] are modified to make a reference call-prototype matchine algorithm as follows:

Step 1 [p73] If a *reference-expression* begins with a *module-abbreviation*, the *module-abbreviation* must immediately precede a parenthesized *reference-expession*.

Step 2 [p74] The end of the *reference-call* is determined before proceeding with this step.

If the base *reference-expression* in the call is followed by a *member-name*, then the *reference-call* is terminated just before the next *member-name* or next '`[`' bracket or at the end of the containing *expression*.

If the base *reference-expression* in the call is followed by *square-bracketed-call-argument-lists*, the *reference-call* is extended to include exactly as many such lists as there are *square-bracketed-pattern-argument-lists* in the *reference-prototype-pattern*. After this the *reference-call* is terminated just before the next *member-name* or next '`[`' bracket or at the end of the containing *expression*. If there are fewer *square-bracketed-call-argument-lists* than *square-bracketed-pattern-argument-lists*, the entire call-prototype match fails.

Step 7 [p75] This step is skipped completely. The result of a *reference-call* is the prototype result after wildcards have been assigned by arguments.

Step 10 [p76] Implicit conversion of the *reference-expression* that is the first argument is not permitted. I.e., the *reference-pattern-argument* type must map to `*EXACT*`.

Example:

```
// The *UNCHECKED* function used below is a builtin function
// that performs a variety of conversions which violate type
// checking.  Here it has the prototype:
//
//    av Q$ T$ @r = std *UNCHECKED*
//        ( ap Q$ T$ @p, int offset, int lower, int upper )
//
// that takes the ap @p, adds the offset argument to the
```

```
// pointer offset, and returns this as an av with bounds
// lower and upper.

// In my vector X, a vector with flt64 elements is located
// at X.offset from the address of X and allows index range
// from 0 through X.length-1.
//
type my vector:
    int offset        // Offset of first element in bytes.
    int length        // Number of elements.
    align 64
    *LABEL* first     // Offset of first element in bits.

reference function ap Q$1 flt64 @x =
        ( ap Q$1 my vector @v ) [ int index ]:
    av Q$1 flt64 @p = *UNCHECKED*
            ( @v, v.offset, 0, v.length )
        // See *UNCHECKED* above.
    @x = @p[index]

type my data:
    *INCLUDE* my vector
    flt64[2]

ap *READ-WRITE* my data @D:    // `@= local' is implied
    D.offset = D.first / 8     // 8 converts bits to bytes.
    D.length = 2

D[0] = 5.5      // Computes and uses pointer @D[0]
D[1] = -7.33    // Computes and uses pointer @D[1]
flt64 x = D[0]  // Now x == 5.5
flt64 y = D[1]  // Now y == -7.33
flt64 z = D[2]  // Run time error: 2 >= upper bound of av
                // that equals D.length.

D[2] = 1.0      // Run-time error: bounds limit exceeded.
```

9.6 Out-of-Line Function Declarations

An out-of-line function prototype is a limited subset of an inline function prototype which ensures that there is a single ordered list of arguments. To obtain a more flexible interface,

an out-of-line function call should be embedded in an inline function that pre-processes the arguments.

The syntax of an out-of-line function declaration is:

out-of-line-function-declaration ::=
out-of-line-function-prototype `:`
statement^{+}
| *out-of-line-function-prototype* `: *DEFERRED*`

out-of-line-function-prototype ::=
`out-of-line function` { *prototype-result-list* `=` }$^{?}$
out-of-line-function-name *pattern-argument-list*$^{?}$

out-of-line-function-name ::=
module-abbreviation$^{?}$ *basic-name* | *foreign-function-name*

foreign-function-name ::= *quoted-string*

prototype-result-list ::= see p67

module-abbreviation ::= see p30

basic-name ::= see p30

pattern-argument-list ::= see p68

- The rules for inline *function-declarations* on p67 must be followed where applicable.
- '`?=`' `bool` defaults are not allowed.
- `macro` arguments are not allowed.
- Wild-cards are not allowed.
- Functions with *foreign-function-names* are called **foreign**. These must all be declared as `*DEFERRED*`.
- The rules for inline '`*DEFERRED*`' non-foreign *function-declarations* and their companions on p71 must be followed for `*DEFERRED* out-of-line-function-declarations` and their companions.

Out-of-line function calls must be the right side (after the `=`) of *call-assignment-statements*,^{p46} they cannot be subexpressions. A non-foreign out-of-line function can be called with a normal *function-call*^{p39}. A foreign out-of-line function must be called with a:

call-expression ::=
`call` *function-expression call-argument-list*$^{?}$

function-expression ::= *reference-expression*

reference-expression ::= see p37

call-argument-list ::= see p40

where the *function-expression* must evaluate to either:

- a `const` *foreign-function-name*
- a function-type value: see 9.6.1 [p86]

Unlike inline functions, an out-of-line function can be called from a statement for which only a `*DEFERRED*` declaration of the out-of-line function is visible. A missing companion declaration is not a compile-time error, but will be a run-time error if the function is actually called at run-time.

Like inline functions, a `*DEFERRED*` *out-of-line-function-declaration* can have only one companion. If a `*DEFERRED*` *out-of-line-function-declaration* is external, [p93] its companion may be anywhere in the scope of the declaration, including in another module or another module's body that imports the declaration's module. This allows a module to call out-of-line functions defined by a companion in another module that imports the first module.

9.6.1 Function Type Declarations

A function type whose values are pointers to out-of-line functions may be declared by:

function-type-declaration ::=
    `type` `function-type-name` `is` *function-type-prototype*
function-type-name ::= *type-name*
type-name ::= see p30
function-type-prototype ::=
    `function` { *prototype-result-list* `=` }? `()` *pattern-argument-list*?

Here the *function-type-prototype* is just like an *out-of-line-function-prototype* except that the *out-of-line-function-name* is replaced by `()` and the word '`out-of-line`' is omitted as being superfluous.

The only operations defined on function type values are copying them, comparing them with `==` and `!=`, and calling them. A call to such a value must be a *call-expression* [p85] and must be the right side (after the `=`) of a *call-assignment-statement.* [p46].

A function constant can be declared by:

function-constant-declaration ::=
        function-type-name function-constant-name `:`
            statement+
function-constant-name ::= *target-variable*
target-variable ::= see p30

Here the first line behaves like an *out-of-line-function-prototype* made by taking the *function-type-prototype* specified by the *function-type-name* and replacing the `()` *out-of-line-function-name* by the *function-constant-name* while adding '`out-of-line`' to its beginning. In addition the *function-constant-name* is declared as a run-time `co` variable whose value has the

type named by the *function-type-name*. Internally, this value is a run-time pointer to the out-of-line function.

9.7 Module and Body Declarations

A **module** is a file whose first statement is a *module-declaration*:

module-declaration ::= *simple-module-declaration*
| *simple-module-declaration*`:`
import-clause*
simple-module-declaration ::= `module` *module-name* `as` *module-abbreviation*
module-name ::= *quoted-string*
module-abbreviation ::= see p30
import-clause ::= `import` *module-name* `as` *module-abbreviation*

- A *module-declaration* may only appear as the first statement of a module file.
- In a *module-declaration* all *module-abbreviations* must be distinct, and all *module-names* must be distinct.

The compiler maps *module-names* to POSIX file names in an implementation dependent manner. The file that contains the module cannot contain anything else.

The *module-abbreviation* associated with a *module-name* may differ in different files. Specifically, the *module-abbreviation* for a module used in the module's own module file need not be the same as the *module-abbreviations* used for the module in files that import the module.

The module `"standard"` with module abbreviation `std` is builtin and contains the builtin types and functions. The *import-clause*

```
import "standard" as std
```

is implied in every *module-declaration* and *body-declaration*.

A **body** is a file whose first statement is a *body-declaration*:

body-declaration ::= `body` *body-name* `of` *module-name*`:`
body-clause*
body-name ::= *quoted-string*
module-name ::= see p87
body-clause ::= *import-clause* | *after-clause*
import-clause ::= see p87
after-clause ::= `initialize after` *body-name*

- A *body-declaration* may only appear as the first statement of a body file.

- In a *body-declaration* the *module-abbreviations* of imported modules must be distinct and must be different from the *module-abbreviation* used by the body's module, and all *module-names* and *body-names* must be distinct.

The compiler maps *body-names* to POSIX file names in an implementation dependent manner. The file that contains the body cannot contain anything else.

A **body** is an extension of the module named in the first line of the *body-declaration*.

A body implicitly imports the module it extends. Within the body that module has the same *module-abbreviation* that it had in the module's own file. The other modules imported in the module's own file are not implicitly imported to the body. The body must import whatever other modules it uses explicitly.

The *after-clauses* name other bodies, not necessarily in the same module, and determine the order in which bodies are initialized: see 9.7.1 p88.

9.7.1 Program Initialization

A module is initialized by executing its top level *statements* in the order in which they appear in the module. Similarly a body is initialized by executing its top level *statements* in the order in which they appear in the body.

The order in which modules and bodies are initialized is determined by the following rules.

1. If a module or body imports another module, the imported module is initialized before the importing module or body.
2. A module is initialized before any of its bodies.
3. If a body contains an *after-clause*, the body is initialized after the body named in the *after-clause*.

The conceptual directed graph whose nodes are modules and bodies and whose arrows connect each module or body to the modules and bodies it must be initialized after is called the '**initialization graph**' and must be acyclic.

10 Inclusions

Each *statement* S has its own `*INCLUSION* const` variable that contains a list of *statements* that is prepended immediately after the compilation of S is finished to the current list of *statements* to be compiled. Before S is compiled its `*INCLUSION*` variable is initialized to the empty list. Inline functions compiled during the compilation of S can read and write the `*INCLUSION*` variable of S.

Since *statements* can be nested, compilation of *statements* can be nested, and at any given time the innermost `*INCLUSION*` variable hides outer `*INCLUSION*` variables. Thus at any time during compilation there is a current `*INCLUSION*` variable.

If one statement is contained in another, the statements in the inner `*INCLUSION*` are appended to the outer `*INCLUSION*` when the inner statement finishes compiling.

It is not necessary to include certain annotations in statements or expressions added to the `*INCLUSION*` variable value.

Specifically, it is not necessary to include the `.position` or the following `.initiator`s or `.terminator`s:

Unnecessary Annotations

`.initiator`	`.terminator`
`*LOGICAL-LINE*`	`"<LF>"`
`"("`	`")"`

because the `.position` of the function call will be added if no `.position` is given, logical lines can be identified from context, and `()` bracketed subexpressions are equivalent to subexpressions with implied brackets (i.e., with no `.initiator` or `.terminator`).

However, any `.separator` and the following must be included:

Necessary Annotations

`.initiator`	`.terminator`
`":"`	`*INDENTED-PARAGRAPH*`
`"["`	`"]"`

The *include-statement* can be used to parse statements and append them to the end of the list which is the value of a `const` variable:

include-statement ::=
    `*INCLUDE*` *include-variable*$^{?}$ *include-argument-list*$^{?}$ **`:`**
        statement*
    `*INCLUDE*` *include-variable*$^{?}$ *include-argument-list*$^{?}$ **`:`** *statement*

include-variable ::= *target-variable* [see p30]

include-argument-list ::= **`(  )`** | **`(`** *include-argument* { **`,`** *include-argument* }* **`)`**

include-argument ::= *word* beginning with a capital *letter*

An *include-statement* is executed at compile time; its *statements* are parsed and the parser output is appended to the list designated by the *include-variable*, which must be a `const` variable. The *include-variable* defaults to `*INCLUSION*`.

Each *include-argument* must be a `const` variable assigned a value before the *include-statement* compiles. Everywhere this variable appears in the parsed *statements* of the *include-statement*, the value of the variable is substituted for the variable name.

Substitution for *include-arguments* obeys the following rules:

- If the *include-argument* value is a list, and if the instance being substituted is an element of a list, the *include-argument* list is spliced into the instance containing list.

  Thus if `X = {"A", "B"}` then

  `"Y", "=", { "X" }` becomes `"Y", "=", { "A", "B" }`

  The *include-argument* value list may be empty. If `X = {}` then

  `"Y", "=", { "foo", "X" }` becomes `"Y", "=", { "foo" }`

  If you do not want a list valued *include-argument* to be spliced in, use `( )` parentheses around the instance being substituted. Thus if `X = {"A", "*", "B"}` then

  ```
  "Y", "=", { ("X"), "+", 1 }
  ```

  becomes

  ```
  "Y", "=", { ("A", "*", "B"), "+", 1 }
  ```

- Otherwise the non-list value of the *include-argument* replaces the instance in the parsed *statement*. Thus if `X = "A"` then

  `"Y", "=", { "X", "*", 2 }` becomes `"Y", "=", { "A", "*", 2 }`

11 Parser Output

The output produced by the parser when it parses code is as follows. In the following ***LOGICAL-LINE*** and ***INDENTED-PARAGRAPH*** name special constants.

Recall that the input is a sequence of logical lines. Also, *rational-constants*[p33] are not produced by the parser: they are parsed as an operator name *word* followed by a *quoted-string*.

For a logical line, the parser produces a list with the annotations:

```
".initiator" => *LOGICAL-LINE*, ".terminator" => "<LF>"
```

The list elements are strings and numbers representing lexemes, and lists representing subexpressions.

Recall that an indented paragraph may appear at the end of a logical line.

For an indented paragraph the parser produces a list which has the annotations:

```
".initiator" => ":", ".terminator" => *INDENTED-PARAGRAPH*
```

The list elements are logical lines.

For an explicitly bracketed subexpression the parser produces a list which has the annotations:

```
".initiator" => "(", ".terminator" => ")"
```

or

```
".initiator" => "[", ".terminator" => "]"
```

The list elements are strings and numbers representing lexemes, and lists representing subexpressions.

For an implicitly bracketed subexpression the parser produces a list which has no `.initiator` and `.terminator` annotations. The list elements are strings and numbers representing lexemes, and lists representing subexpressions.

The parser does not introduce implied brackets where there are explict or implied brackets, but will introduce implied brackets surrouding quoted strings. For example,

`x + y` parses as `{ { "x" }, "+", { "y" } }`

`( x ) + y` parses as

```
{ { "x", ".initiator" => "(", ".terminator" => ")" },
  "+", { "y" } }
```

`( x, "y" )` parses as

```
{ { "x" }, { { "y", ".type" => "<Q>" } },
  ".initiator" => "(", ".terminator" => ")",
  ".separator" => "," }
```

Operators that are separators, such as '`,`', are not included as elements of a list, but become a **`.separator`** annotation of the list. Thus,

`( x, y )` parses as

```
{ "x", "y",
  ".separator" => ",",
  ".initiator" => "(",
  ".terminator" => ")" }
```

Quoted strings become a list with the string as a single element and a `.type` annotation equal to `"<Q>"` (recall that `<Q>` in a quoted string represents the double quote `"`). Thus the lexeme `"Hello"` becomes:

```
{ "Hello", ".type" => "<Q>" }
```

Operator operands become lists in parser output; for example, the statement '`X = Y`' outputs '`{ { "X" }, "=", { "Y" } }`'.

An example is given in Figure 5 [p92].

12 Scope

A *declaration* has a **scope**, that is the set of statements in which any names or prototypes defined by the *declaration* are recognized.

Generally the scope of a *declaration* includes the *statements* in any *block* at the end of the *statement* containing the *declaration* (recall that a *statement* is a logical line that can end in a *block*), and all *statements* following the *statement* containing the *declaration* up to the end

Parser Input:

```
if X < Y:
    X = Y
    Y = Y + 5 * Z
    A 1, B = B, A 1
    const P = "HOHO"
    const Q = 5 + P
    const R = ( 5 + Q )
```

Parser Output:

```
{ "if",
  { { "X" }, "<", { "Y" } },
  { { { "X" }, "=", { "Y" },
      ".initiator" => *LOGICAL-LINE*, ".terminator" => "<LF>" },
    { { "Y" }, "=",
      { { "Y" }, "+", { { 5 }, "*", { "Z" } } },
      ".initiator" => *LOGICAL-LINE*, ".terminator" => "<LF>" },
    { { { "A", 1 }, { "B" }, ".separator" => "," },
      "=",
      { { "B" }, { "A", 1 }, ".separator" => "," },
      ".initiator" => *LOGICAL-LINE*, ".terminator" => "<LF>" },
    { { "const", "P" }, "=", { { "HOHO", ".type" => "<Q>" } },
      ".initiator" => *LOGICAL-LINE*, ".terminator" => "<LF>" },
    { { "const", "Q" }, "=",
      { { 5 }, "+", { "P" } },
      ".initiator" => *LOGICAL-LINE*, ".terminator" => "<LF>" },
    { { "const", "R" }, "="
      { { 5 }, "+", { "Q" },
        ".initiator" => "(", ".terminator" => ")" },
      ".initiator" => *LOGICAL-LINE*, ".terminator" => "<LF>" },
    ".initiator" => ":", ".terminator" => *INDENTED-PARAGRAPH*
  },
  ".initiator" => "*LOGICAL-LINE*", ".terminator" => "<LF>"
}
```

Figure 5: Parser Output Example

of the smallest *block* containing this *statement*.

The **context** of a statement is the set of declarations whose scope the statement is in.

A **top-level** *declaration* is a *declaration* that is not in a *statement* inside any *block*. The scope of a top-level *declaration* lasts to the end of the file containing the *declaration*. The scope of a top-level *declaration* in a module file is extended to include each body file of the module.

Some *declarations* are **external**. These must be top-level declarations in a module (and not in a body). The scope of an external *declaration* is extended to include all modules and bodies that import the module containing the *declaration*.

A *result-variable-declaration*, *block-variable-declaration*, or *deferred-variable-declaration*, is external if the variable-name declared begins with a *module-abbreviation*.

A *next-variable-declaration* is external if the variable-name declared begins with a *module-abbreviation*. However, the *next-variable-declaration* must be in the same module file as the *result-variable-declaration* whose variable name it shares.

A *type-declaration*, *pointer-type-declaration*, or *function-type-declaration* is external if the type-name, *pointer-type-name*, or *function-type-name* declared begins with a *module-abbreviation*.

A *inline-function-declaration*, *reference-function-declaration*, or *out-of-line-function-declaration* is external if the *prototype-pattern* in the *declaration* is immediately preceded by a *module-abbreviation*.

Prototype-result-declaration and *prototype-argument-declaration variable-names* cannot begin with a *module-abbreviation*, and therefore these *declarations* can never be external.

In general, a deferred declaration in a module may have a companion in that module or in a body of that module, but not in modules or bodies that import the declaration's module. As one exception, a deferred external *out-of-line-function-declaration* may have its companion anywhere within the scope of the original declaration. As a second exception, a deferred *type-declaration* must have its companion in the same file as the deferred *type-declaration*. Note that a deferred *type-declaration* may have only one companion, but the companion may have many expansions in the scope of the companion (the companion is the root of its expansion tree: see 9.1.1 p55).

Type-declarations that end with the `*EXTERNAL*` sub-declaration must have external *type-names* (beginning with a *module-abbreviation*). Expansions of such a *type-declaration* may appear anywhere within scope of the initial root *type-declaration*p56, but must have a *type-name* that references the same module as the *type-declaration* they are expanding (they need not use the same *module-abbreviation* to do so).

If two *type-declarations* declaring the same *defined-type-name* have overlapping scope, they must follow the expansion tree rules of 9.1.1 p55. *Type-subdeclarations* of any *type-declaration* have the scope of the *type-declaration* in which they appear, and if they appear in an expan-

sion, their scope is that of the expansion and not the scope of any ancestor *type-declaration* in their expansion tree. *Origin-labels* declared by a *type-subdeclaration* are only visible inside *type-declarations* in the same expansion tree.

A *module-abbreviation* that makes a *declaration* external must abbreviate the module in which the *declaration* occurs, with the exception of expansions of *type-declarations* that end with the `*EXTERNAL*` sub-declaration [p56] and companions of deferred *out-of-line-function-declarations.* [p86]

If two different *result-variable-declarations*, *block-variable-declarations*, *deferred-variable-declarations*, *next-variable-declarations*, *function-constant-declarations*, *pointer-type-declarations*, or *function-type-declarations* of the same *name* have overlapping scope, one of these scopes must include the other, and the declaration with the smaller scope is said to '**hide**' the other declaration. Hiding of this kind is a compile error (note this 'hiding error' does not apply to *type-declarations*).

A *next-variable-declaration* is allowed within the scope of a previous *result-variable-declaration*, *block-variable-declaration*, or *next-variable-declaration* of the same *variable-name* if it is not within a smaller block than the previous declaration. Note that a *next-variable-declaration* has the same syntax as a *reference-expression*, and its use as an implicitly `*INIT*` [p39] *reference-expression* is allowed within a subblock of the *block-assignment-statement* or *loop-assignment-statement* that assigns a value to the declared `next` variable. Also note that *next-variable-declarations* can be implied in *block-assignment-statements* [p48] and *loop-assignment-statements* [p51].

Function prototypes cannot hide each other. If the current context contains two function declarations whose prototypes both match a call, the call is ambiguous and in error, even if the scope of one declaration is within the scope of the other.

Statement-labels (*block-labels*, *exit-labels*, and *loop-labels*), have as their scope the block in which they are defined. It is a compile error if *statement-labels* hide each other.

When a *function-call* to an inline function is expanded, the context of the expansion is not the current context but rather the context of the inline *function-declaration* that provided the *statements* executed by the call. Also the context in which any *default-value* expression provided by a *function-declaration* is compiled is the not the current context but rather the context of that *function-declaration.*

Similarly when a *reference-call* to a reference function is expanded, the context of the expansion is not the current context but rather the context of the inline *reference-function-declaration* that provided the *statements* executed by the call. Also the context in which any *default-value* expression provided by a *reference-function-declaration* is compiled is the not the current context but rather the context of that *reference-function-declaration.*

Code produced by inclusions during *statement* compilation is compiled in the context immediately following the *statement.* See Inclusions (p88).

An example is:

```
module "my_own_module" as mom:
    // `import "standard" as std' is implied
    import "George's_own_module" as gom
    // gom contains:
    //    function int32 z = gom ( int32 x ) "+" ( int32 y )

int32 mom my external constant = ...
int32 my internal constant = ...

function int32 y = mom my external function ( int32 x ):
    ... function body omitted ...
function int32 y = my internal function ( int32 x ):
    ... function body omitted ...

function int32 z = my inline function ( int32 x, int32 y ):
    int32 z1 = gom ( x + y )
        // Uses gom's + operator.
        // Compiles as as `gom (x) "+" (y)'.
    int32 z2 = std ( x + y )
        // Uses builtin std's + operator.
        // Compiles as as `std (x) "+" (y)'.
    z  = z1 + z2
        // Compiles as `z = ( (x) "+" (y) )'.
        // Compile error, ambiguous: both std + operator
        // and gom's + operator match the call to "+".
```

More specifically, when a function declaration is used, the *module-abbreviation* beginning the function call may be omitted if the function declaration is the only function declaration within scope that matches the usage, according to the module proficiency rules of section 9.3.1 [p72]. Thus in the context of the above example the lines:

```
int32 y = mom my external function ( x )
int32 y = my external function ( x )
```

are equivalent if no *function-prototype*

```
function int32 r = ma my external function ( int32 v )
```

is in the current context, where *ma* is a module abbreviation for a module other than ‘`my_own_module`’.

13 Lifetimes

The **lifetime** of a variable, i.e., a piece of memory, is the time interval from the time that the variable is allocated to the time that the variable is deallocated. The compiler tracks lifetimes by assigning each variable a lifetime type and in some cases a separate lifetime depth.

The possible **variables** are:

assignment variable The value stored in a `co` variable created by a *result-variable-declaration*, *block-variable-declaration*, *next-variable-declaration*, or *deferred-variable-declaration*, of an *assignment-statement*.

argument variable The value stored in a `co` variable created by an *prototype-argument-declaration* in a *function-prototype* when the function is called.

result variable The value stored in variable created by an *prototype-result-declaration* in a *function-prototype* when the function is called.

target variable A value that is part of the target of a pointer (e.g., a field, element of a field, etc.).

There are three possible variable **lifetime types**:

`*GLOBAL*` The lifetime starts when the variable is created and stops when the program terminates.

`*LOCAL*` The lifetime starts when the statement declaring and creating the variable executes and stops when scope of the variable's declaration ends.

`*HEAP*` The lifetime starts when the variable is allocated to the heap (i.e., garbage collectible memory), and stops when the variable can no longer be referenced by following a chain of pointer values the root of which is a variable that is either `*GLOBAL*` or is `*LOCAL*` with a lifetime that has not yet terminated.

All external variables are `*GLOBAL*` These are all top-level assignment variables in module files. Other assignment variables, all argument variables, and and all result variables are `*LOCAL*`. A target variable has its lifetime type specified by the target qualifier of the pointer that points to the target containing the variable. For example, the target of a pointer of type `ap *GLOBAL* int` has `*GLOBAL*` lifetime. This pointer can be stored in a `*LOCAL*` assignment variable.

A pointer whose target has a particular lifetime is said to have that lifetime: thus we have `*GLOBAL*` pointers, `*LOCAL*` pointers, etc. In general a pointer cannot be stored in a variable unless one of the **lifetime rules** of this section permits it.

The goal of these rules is to keep a pointer with a given lifetime from being stored in a variable with a longer lifetime.

The first rules are:

(L1) A `*GLOBAL*` pointer can be stored in any variable.

(L2) A `*HEAP*` pointer can be stored in any variable.

(L3) A `*GLOBAL*` lifetime type qualifier can be implicitly converted to a `*HEAP*` lifetime type qualifier.

Note: This requires the garbage collector to be able to identify `*GLOBAL*` pointers and treat them differently.

(L4) A `*GLOBAL*` lifetime type qualifier can be implicitly converted to a `*LOCAL*` lifetime type qualifier of depth 0 (see discussion of depth below).

(L5) A `*HEAP*` pointer can be implicitly converted to a `*LOCAL*` pointer of depth 0 provided the `*HEAP*` pointer is stored in an assignment variable or argument variable.

Note 1: the variable in which the `*HEAP*` pointer is stored is `co` and will exist as long as the `*LOCAL*` pointer exists.

Note 2: because of this conversion, both `*HEAP*` and `*LOCAL*` pointers in the stack will have to be updated by by some kinds of garbage collection: see 14.1 p102, Copying Garbage Collection.

(L6) A `*LOCAL*` pointer can only be stored in a `*LOCAL*` variable. It cannot be stored in a `*GLOBAL*` or `*HEAP*` variable.

Note: The variable in which a pointer is stored must have by default a pointer type that is the same as the type of the pointer value being stored: e.g., when a `*HEAP*` pointer is stored in a `*LOCAL*` variable, the `*LOCAL*` variable's type must be a pointer type with the `*HEAP*` qualifier. In particular, (L1) and (L2) do not permit implied lifetime qualifier conversions.

However, (L3) and (L4) do permit `*GLOBAL*` pointers to be stored in variables which have a pointer type with the `*HEAP*` or `*LOCAL*` qualifier, and (L5) permits `*HEAP*` pointers to be stored in variables which have a pointer type with the `*LOCAL*` qualifier.

Also see 18.2.1 p120.

An example is:

```
type list element:
    ap *GLOBAL* list element @after

// In the following, `@= global' is implied because the pointers
// are *GLOBAL*.

// Circular list:
```

```
//
ap *GLOBAL* *READ-WRITE* list element @last:  // `@= global' is implied
    last.@after = null
    // Last element is pointed at by last.@after
    // First element is pointed at by last.after.@after
    // Empty list has last.@after == null

// Put two elements in list.
//
ap *GLOBAL* list element @X  // `@= global = *DEFERRED*' is implied
ap *GLOBAL* list element @Y  // `@= global = *DEFERRED*' is implied

ap *GLOBAL* list element @X:
    // Put X on empty list
    X.@after = @X
    last.@after = @X
    // List now consists of (X)
ap *GLOBAL* list element @Y:
    // Add Y to end of list
    Y.@after = last.after.@after
    last.after.@after = @Y
    last.@after = @Y
    // List now consists of (X, Y)
```

LOCAL pointers can only be stored in assignment, argument, or result variables, or in targets of *LOCAL* pointers. *LOCAL* pointers have lifetimes that depend upon the extent of the scope of the declaration that created them. Scopes are nested, and declarations have depths in this nesting. The depth of a pointer is the depth of its declaration statement, and pointers with greater depth have smaller scopes and shorter lifetimes. *LOCAL* pointers with greater depth must not be stored in targets of *LOCAL* pointers of less depth.

So we introduce the notion of **lifetime depth** of *LOCAL* pointers. Our rule on when a *LOCAL* pointer can be stored a *LOCAL* pointer's target is:

(L7) A *LOCAL* pointer of lifetime depth $d1$ can be stored in the target of a *LOCAL* pointer of lifetime depth $d2$ if $d2 = d1 \neq 0$.

Not that this seems more restrictive than necessary. The less restrictive rule is simply $d2 \leq d1$. However the compiler cannot enforce this last rule. To use the rule it would be necessary to record the lifetime depth of a *LOCAL* pointer in its run-time value and check the rule at runtime, and we choose not to do this.

The lifetime depth of a pointer created by a '`@= local ...`' *allocation-call*[p47] is the depth of the statement that assigns the variable. The lifetime depth of the value of an assignment variable, an inline prototype argument variable, or an inline prototype result variable is the

lifetime depth of the value assigned to the variable. Note that in the case that the pointer assigned is `*GLOBAL*`, this depth is 0 by (L4).

The rules for assigning depth to *statements* are as follows:

(S1) A *statement* that is not inside any other statement (i.e., is not in a sub-block; i.e., is top-level) is assigned depth 1.

(S2) A *statement* that is inside a sub-block S at the end of an *assignment-statement* of depth D, and is not also inside a sub-block of S, is assigned depth $D + 1$.

(S3) An inline *function-call* is treated as creating a sub-block immediately inside the *statement* containing the *function-call*.

Thus if an inline function is called from a *statement* of depth D, the *statements* in the sub-block S at the end of the inline function declaration that are not inside a sub-block of S will have depth $D + 1$.

Note: If an inline function is called from a *statement* of depth D, the depth D' of the function's declaration must such that $D' \leq D$, since the calling *statement* must be in the scope of the declaration. So when the *statements* of sub-block S access `*LOCAL*` pointer values defined by the context of the declaration, rule (L7) will give the same answers as it would if the *statements* of S had depth $D' + 1$ instead of $D + 1$.

(S4) Let S be the sub-block at the end of an out-of-line function declaration that is of depth D. Then any statement in S that is not also in a sub-block of S is assigned depth $D + 1$.

Because of (L7), a `*LOCAL*` pointer read from the target of a `*LOCAL*` pointer must have the same depth as that pointer. So the compiler always knows the depth of a `*LOCAL*` pointer and can enforce (L7).

Rule (L7) permits a set of `*LOCAL*` objects created at the same depth to point at each other. An example is:

```
type list element:
    ap list element @after

// In the following, `@= local' is implied because the pointers
// default to *LOCAL*.  Also, non-external assignment-variables
// of top level statements have depth 1.

// Circular list:
//
ap *READ-WRITE* list element @last:  // `@= local' is implied
    last.@after = null
```

```
        // Last element is pointed at by last.@after
        // First element is pointed at by last.after.@after
        // Empty list has last.@after == null

    // Put two elements in list.
    //
    ap list element @X  // `@= local = *DEFERRED*' is implied
    ap list element @Y  // `@= local = *DEFERRED*' is implied

    // Now @last, @X, @Y, last.@after, X.@after, etc. all have depth 1

    ap list element @X:
        // Put X on empty list
        X.@after = @X
        last.@after = @X
        // List now consists of (X)
    ap list element @Y:
        // Add Y to end of list
        Y.@after = last.after.@after
        last.after.@after = @Y
        last.@after = @Y
        // List now consists of (X, Y)
```

The same example using inline functions is:

```
    type list element:
        ap list element @after   // target is *LOCAL* by default

    // Circular list:
    //
    ap *READ-WRITE* list element @last:      // `@= local' is implied
        last.@after = null
        // Last element is pointed at by last.@after
        // First element is pointed at by last.after.@after
        // Empty list has last.@after == null
        // The variable `last' has depth 1

    // Add element to end of circular list:
    //
    function add ( ap *READ-WRITE* @element ):
        // element depth is taken from actual argument
        if last.@after == null:
            element.@after = @element
```

```
        else:
            element.@after = last.after.@after
            last.after.@after = @element
        last.@after = @element

    // Put two elements in list.
    //
    ap *READ-WRITE* list element @X          // `@= local' is implied
                                             // @X and X are depth 1
    ap *READ-WRITE* list element @Y          // `@= local' is implied
                                             // @Y and Y are depth 1
    add ( @X )           // X is first element
    add ( @Y )           // Y is second element
```

For out-of-line functions, an argument value that is a `*LOCAL*` pointer is given depth 0, preventing `*LOCAL*` pointers taken from other arguments from being stored in its target.

14 Memory Management

Space for variables allocated by *variable-declarations* is allocated to the currently executing out-of-line function frame or to the stack after this frame: see p43 and p44 for details.

Pointers to the heap (**heap pointers**) normally use the stub-body concept: the heap pointer points not at the body of a heap datum, but instead at a stub which begins with a body pointer at the body. This can be used in a variety of garbage collection schemes. All these schemes require that some special action be taken when a heap pointer is read into a stack variable, or when a heap pointer is written into a heap datum, or when any location in the stack or heap that stores a heap pointer has its value changed.

However instead of using stub-body, heap pointers can optionally be interpreted as pointing directly at the heap datum.

Specifically, the compiler recognizes the following options:

`stub` or **`no-stub`**

> With the `stub` option, heap pointers point at a stub and the first word of the stub points at the heap datum body. With the `no-stub` option, heap pointers point directly at the body.

`copying-gc`, **`marking-gc`**, **`counting-gc`**, or **`no-gc`**

> These options specify garbage collection (**GC**) algorithm being used. These algorithms are described below.

`read-gc` or **`write-gc`**

These are sub-options of GC, indicating whether the GC is read-oriented or write-oriented. See the descriptions of GC below.

All kinds of GC run interleaved with non-GC execution. These compiler options control default inline functions that do the following during non-GC execution:

- Read a non-pointer from a heap datum.
- Read a pointer from a heap datum.
- Write a pointer to a heap datum.
- Write a non-pointer to a heap datum.
- Write a pointer to a non-loop iteration stack variable.
- Write a pointer to a loop iteration stack variable.
- Deallocate a location that contains a pointer.

The three kinds of GC are discussed in detail in the following sub-sections.

14.1 Copying Garbage Collection

In copying garbage collection the stub of a datum is the datum's first word. When the datum is first allocated, this stub points at itself. Then during GC the body will be copied and for a time have two stubs, the stub in the old body that was the source of the copy, and the stub of the new body that was the destination of the copy. Both will point at the new body, and the new body will hold the datum itself, while the old body will no longer be accessed except for its stub.

GC works in cycles. At the start of each cycle, all heap data are in a contiguous virtual memory space called the old space. A new large contiguous virtual memory space is allocated called the new space. The boundary address between these can be use to tell if a body is in old space or new space: just compare the body pointer with the boundary address. The object of the GC cycle is to copy all active data from old space to new space, update all active heap pointers to point at new space stubs, and then discard old space completely.

GC performs a basic operation on heap pointers which we will call **pointer-update**. In this a heap pointer is checked to see if it points at new space, and if not, is replaced by a pointer that points at new space. If the heap pointer points at a body pointer in old space that itself points at old space, the body pointed at is moved to new space, and the heap pointer is replaced by a pointer to the body in new space. If the heap pointer points at a body pointer in old space that points at new space, the heap pointer is simply replaced by that body pointer.

When a new body is created, it is allocated to new space. Places for new or copied bodies in new space are allocated at the 'end' of new space.

GC goes through new space from beginning to end updating all the pointers in bodies it encounters. This is called 'scavenging'. At any time there is an address that is the boundary between the scavenged bodies at the beginning of new space and the non-scavenged bodies at the end of new space. This address can be use to tell if a new space body has been scavenged.

The root pointers of GC are in the global memory (memory allocated at load time) and the stack.

The GC can be either read-oriented or write-oriented.

A read-oriented GC updates heap pointers when they are read from bodies by non-GC execution. The GC begins by updating all heap pointers in global memory and the stack, and from this point on, all pointers in global memory and the stack point to new space and no special action is required when a pointer is written to a body (which itself will be in new space) by non-GC execution.

A write-oriented GC updates heap pointers when they are written into scavenged bodies by non-GC execution. After all objects in new space have been scavenged, the GC updates all pointers in global memory and the stack, and if this does not move any bodies to new space, GC is done; otherwise GC resumes scavenging. Heap pointers in global memory and the stack need not be updated during non-GC execution, and no special action is taken when reading a heap pointer from a body.

The advantage of write-oriented over read-oriented is that write operations are less frequent than read operations and therefore write-oriented may be more efficient. The disadvantage is that in order to finish, the GC must update all pointers in global memory and the stack without non-GC execution changing the stack.

It is possible to implement a deallocate operation which deallocates a body, except for its body pointer. A deallocated body has a body pointer pointing at a large area of inaccessible virtual memory, so a memory fault will occur if the body is accessed. To copy a deallocated body one just makes a copy of just the body pointer without changing this body pointer, which is left pointing at inaccessible memory. The update operation must do extra work to detect deallocated bodies if they are permitted.

The `no-stub` option may not be used with copying GC.

14.2 Marking Garbage Collection

In marking GC, stubs are allocated to a separate space from bodies, and bodies are not copied during GC. Bodies are copied by a separate activity called compaction that is independent of GC. The advantages are that there is less body copying and also that less memory space is required. Also deallocated bodies do not require the update operation to do extra work.

The simplest marking GC uses stubs that begin with a body pointer followed by two list pointers (for doubly linked lists), a marked flag, and a scavenged flag (the flags can generally be put in the same words as the list pointers). There are two lists of stubs: an old space list and a new space list. To move a stub from old space to new space, it is unlinked from the old space list and linked onto the end of the new space list, and its marked flag is set. When a datum is scavenged, its scavenged flag is set.

Otherwise marking GC is just like copying GC.

A variant has only one list pointer associated with the stub and there is just one stub list. The marked and scavenged flags are used as before. At its end, GC goes through the list of all stubs and frees unmarked stubs along with their bodies. However, the list of stubs to be scavenged must be maintained separately, typically as a list of vectors whose elements point at stubs to be scavenged. When a stub is first marked it is put on the list of stubs to be scavenged.

14.3 Counting Garbage Collection

In counting GC each stub has a reference count. The fundamental non-GC operation is storing a pointer P in a location. The required steps are:

(1) save the location's previous value S
(2) add one to the reference count of the stub pointed at by P
(3) subtract one from the reference count of the stub pointed at by S; if that reference count is now zero, collect the stub and its body
(4) store P in the location

This operation must be used when a pointer location is updated in the stack or in a body. There is also an operation for deallocating a location containing a pointer, which omits steps (2) and (4).

Bodies may or may not be allocated separately from stubs. If separate (the `stub` option), deallocated bodies may be implemented and bodies may be compacted separately from GC. Or the `no-stub` option may be used with counting GC.

Of course reference counting GC cannot collect data containing pointer loops, such as circular lists.

14.4 Locating Global and Local Heap Pointers

A list of global and local variables that contain `*HEAP*` pointers needs to be made available to the garbage collector.

From the point of view of the garbage collector, variables are held in two stacks: the **global stack** and the **local stack**. These stacks are each a sequence of blocks. Each block has

at its beginning a **block type** ID which references load time data that tells the length of the block and contains a map specifying which words of the block contain `*HEAP*` pointers. Some types of blocks contain a vector of similar elements, and for these there is a separate count of the number of vector elements at the beginning of the block and the block type data only specifies that length of an element and which words of the element contain `*HEAP*` pointers.

An out-of-line function call begins by allocating a block in the local stack to hold its `*LOCAL* co` variables. All the variables are allocated at the beginning of the call execution, so this block does not vary in size or layout during the function call execution. Executions of '`@= local`' during the function call will add a block to the local stack if the allocated memory size cannot be determined at compile time. At the end of the out-of-line function call, all the blocks it allocated to the local stack are deallocated.

In the case of a statement containing '@= local' in a loop iteration, execution of the statement will re-use the block allocated by the last execution of that statement, unless that block is too small, in which case a new block of at least twice the size of the last block will be allocated. This scheme allocates at most 4 times as much memory as is actually used by the statement during any of its executions.

When a block is allocated it is zeroed. This allows the garbage collector to avoid variables containing `*HEAP*` pointers that have not yet been set.

The execution of each module and body file initialization is treated the same as an execution of an out-of-line function call, except that at load time, before initialization, a block holding the external variables of each module is allocated to the global stack. This allows addresses of external variables to be computed at load time.

Statements containing `@= global` can be executed anytime and allocate blocks to the global stack.

Blocks in the global stack are never deallocated.

15 Non-Function Operators

Some operators map onto functions. For example,

`x + y` maps onto `x "+" y`

However the following operators do not map onto functions:

prefix **`type`**
prefix **`pointer type`**
prefix **`function`**
prefix **`reference function`**
prefix **`out-of-line function`**
afix infix **`is type`**

afix infix **is function**
See Declarations [p52].

prefix **if**
prefix **else if**
initial **else**
afix right **:**
See Conditional Statements [p52].

afix subblock
postfix subblock
See Assignment Statements [p41] and Conditional Statements [p52].

prefix **loop**
prefix **while**
prefix **until**
prefix **exactly** ... **times**
prefix **at most** ... **times**
See *iteration-control* [p50].

infix **=**
See Assignment Statements [p41].

infix **+=**
infix **-=**
infix ***=**
infix **/=**
infix **|=**
infix **&=**
infix **^=**
infix **<<=**
infix **>>=**
The statement '`x += y`' is syntactic sugar for '`next x = x + y`' if `x` is a local `co` variable, and for '`x = x + y`' otherwise.

Similarly for the other operators of the form '*B*`=`' where *B* is a binary operator, the statement '`x` *B*`= y`' is syntactic sugar for '`next x = x` *B* `y`' or '`x = x` *B* `y`'.

infix **@=**
See Allocation Call [p43].

infix **--->**
See Abbreviation Statements [p32].

nofix **,**
Becomes a `.separator` annotation on a list. See the operator separator format [p27].

infix **if**
infix afix **else**
Must be used in an *else-expression* with target type T which has the syntax:

else-expression ::= *if-expression* { **else** *if-expression* }* **else** *T-expression*
if-expression ::= *T-expression* **if** *bool-expression*
bool-expression ::= *expression* with target type `bool` if T is not `const`
and otherwise target type `const`
T-expression ::= *expression* with target type T

The *bool-expressions* are evaluated left to right until one evaluates to `true` or `TRUE`. Then the corresponding *T-expression* (the one in the same *if-expression* as the *bool-expression*) is evaluated and returned. If all *bool-expressions* evaluate to `false` or `FALSE`, the *T-expression* after the last `else` is evaluated and returned.

infix **BUT NOT**
'`x BUT NOT y`' is syntactic sugar for '`x AND ( NOT ( y ) )`'.

infix **AND**
'`x AND y`' is syntactic sugar for '`y if x else FALSE`'.

infix **OR**
'`x OR y`' is syntactic sugar for '`TRUE if x else y`'.

infix **NOT**
'`NOT x`' is syntactic sugar for '`FALSE if x else TRUE`'.

16 Builtin Abbreviations

Module-abbreviations are automatically deduced for function calls, but not for data types, pointer types, or variables used as global constants, such as **true** or **false**. In order to avoid having to input a *module-abbreviation* with every type name, *abbreviation-statements* are used (p32). The following are the builtin abbreviations:

```
                  int  --->  std int
                 intd  --->  std intd
                 intq  --->  std intq
                 int8  --->  std int8
                int16  --->  std int16
                int32  --->  std int32
                int64  --->  std int64
               int128  --->  std int128

                  uns  --->  std uns
                 unsd  --->  std unsd
                 unsq  --->  std unsq
                 uns8  --->  std uns8
                uns16  --->  std uns16
                uns32  --->  std uns32
                uns64  --->  std uns64
               uns128  --->  std uns128

                 bool  --->  std bool
                 true  --->  std true
                false  --->  std false

                 TRUE  --->  std TRUE
                FALSE  --->  std FALSE
                 NONE  --->  std NONE
                UNDEF  --->  std UNDEF
      LOGICAL-LINE*  --->  std *LOGICAL-LINE*
INDENTED-PARAGRAPH*  --->  std *INDENTED-PARAGRAPH*


                  flt  --->  std flt
                 fltd  --->  std fltd
                 fltq  --->  std fltq
                 flt8  --->  std flt8
                flt16  --->  std flt16
                flt32  --->  std flt32
                flt64  --->  std flt64


                   dp  --->  std dp
                   ap  --->  std ap
                   fp  --->  std fp
                   av  --->  std av
                   fv  --->  std fv
```

17 Compile Time Functions and Compiler Constants

Inline functions that have `const` results and arguments and do not produce run-time code are called **compile-time** functions. The functions described in the following sections are builtin compile-time functions.

Some compilation related functions and constants are in the '`compiler`' module, which is abbreviated here as '`com`'.

Unless stated otherwise, builtin compile-time functions obey the following rules:

1. Boolean values are represented by the special constants `TRUE` and `FALSE`.
2. Errors in arguments, such as passing a `string` when a `number` or `rational` is required, result in the function doing nothing but returning the `UNDEF` special constant and producing a compiler error message.
3. If a result or argument is said to be an integer, it may be either a `number` with an integral value, or a `rational` with denominator `1`.
4. A function (e.g., `"+"` or `"=="`) with at least one `number` argument will convert all `rational` arguments to `numbers` before using them, will do all internal calculations with `numbers` and not `rationals`, and will return any numeric results as `numbers`.
5. `Number` values too positive or negative to store are converted to `+Inf` or `-Inf`. `Number` values too small to store are converted to `+0` or `-0`, preserving the sign of the value.

17.1 Compile Time General Functions

```
function const r = std (const v1) "==" ( const v2 )
function const r = std (const v1) "!=" ( const v2 )
```

If any argument is a number and the other is rational, the rational is converted to a number before the comparison. Otherwise comparisons of `const` values of different types treat the values as unequal.

```
function const r = std type ( const v1 )
```

Returns the type of `v1` as one of the strings:

```
"special"   "number"   "rational"   "string"   "map"
```

17.2 Compile Time Numeric Functions

Unless specified otherwise, if one argument is a number and the others are rational, the rational arguments are converted to numbers before the function executes.

```
function const r = std number ( const v1 )
function const r = std rational ( const v1 )
```

These convert their argument to a number or rational. If a single argument is a string, it must have the format of a number or rational constant (rational operator followed by quoted string). If the argument is a map, it may have the format of a number or rational constant in ‘ ’ quotes, or it may be a list of two strings, the first element being a rational operator and the second the string it operates on. E.g.,

```
`5.5'          { "5.5" }
`B# "1.1"'     { "B#", "1.1" }
```

A conversion error produces an `UNDEF` result and an error message. Note that finite numbers can always be converted to rationals, and rationals can always be converted to numbers, though these may be `+Inf` or `-Inf`.

```
function const r = std "+" ( const n1 )
function const r = std "-" ( const n1 )
function const r = std (const n1) "+" ( const n2 )
function const r = std (const n1) "-" ( const n2 )
function const r = std (const n1) "*" ( const n2 )
function const r = std (const n1) "/" ( const n2 )
```

Standard arithmetic operators on numbers or rationals `n1` and `n2`, done using IEEE number or rational arithmetic. For numbers, dividing by `0`, adding `+Inf` to `-Inf`, a `NaN` argument, etc. return `NaN` and no compiler error message. For rationals, dividing by `0` returns `UNDEF` and outputs a compiler error message.

```
function const r = std (const i1) "&" ( const i2 )
function const r = std (const i1) "|" ( const i2 )
function const r = std (const i1) "^" ( const i2 )
function const r = std (const i1) "<<" ( const i2 )
function const r = std (const i1) ">>" ( const i2 )
```

Standard bitwise operators on integers `i1` and `i2` that are treated as two's complement. For the shift operators `"<<"` and `">>"`, `i2`, the amount of the shift, must not be negative. Overflows for number `<<` shift produce an `UNDEF` result and a compiler error message.

```
function const r = std (const n1) "==" ( const n2 )
function const r = std (const n1) "!=" ( const n2 )
function const r = std (const n1) "<" ( const n2 )
function const r = std (const n1) "<=" ( const n2 )
function const r = std (const n1) ">" ( const n2 )
function const r = std (const n1) ">=" ( const n2 )
```

Standard comparison operators on numbers or rationals `n1` and `n2`. Infinities are treated as actual numbers with absolute value larger than any real number: e.g., if `x` is not a `NaN`, ‘`x <= +Inf`’ is always `TRUE` and ‘`x == +Inf`’ is `TRUE` iff `x` is `+Inf`. If an argument

is a NaN, all comparisons return FALSE except != which returns TRUE.

```
function const r1, const r2 = std floor (const n1, const n2 ?= 1 )
function const r1, const r2 = std ceiling (const n1, const n2 ?= 1 )
function const r1, const r2 = std truncate (const n1, const n2 ?= 1 )
function const r1, const r2 = std round (const n1, const n2 ?= 1 )
```

These divide n1 by n2 and return r1 as the result rounded to an integer and r2 as the remainder. Here floor rounds toward negative infinity, ceiling rounds towards positive infinity, truncate rounds toward zero, and round rounds to the nearest integer, or to the even integer if there are two nearest integers.

If an argument is a number, return NaNs if the divisor is zero, an argument is a NaN, or both arguments are infinities, but do not output a compiler error message. If both arguments are rationals and the divisor is zero, return UNDEF and output a compiler error message.

```
function const r = std numerator ( const r1 )
function const r = std denominator ( const r1 )
```

These functions return the numerator and denominator of a rational. Both numerator and denominator are integer rationals.

```
function const r = std is nan ( const v1 )
function const r = std is infinite ( const v1 )
function const r = std is finite ( const v1 )
```

Return TRUE if v1 is a NaN number (is nan), is +Inf or -Inf (is infinite), or is a number that is neither of these (is finite), and FALSE otherwise.

17.3 Compile Time String Functions

```
function const r = std "#" ( const s )
```

Returns the length of string s as a non-negative integer number.[7] Note that # is a prefix operator.

```
function const r = std (const s1) "+" ( const s2 )
```

Returns the concatenation of string s1 and string s2.

```
function const r = std sprintf ( const format, const a1 = "",
                                 const a2 = "", const a3 = "",
                                 const a4 = "", const a5 = "" )
```

Returns the string made by calling the UNIX sprintf function as per:

```
sprintf ( format, a1, a2, a3, a4, a5 )
```

where format is a string. Not all of the data arguments a1, a2, a3, a4, and a5 need

[7] The length of a string cannot be above 2^{48} while numbers can precisely store integers up to 2^{53}.

be used by the `format`. Data arguments may be numbers or strings. Rational data arguments are converted to numbers first. Map data arguments are not allowed.

```
function const r = std (const s1) "==" ( const s2 )
function const r = std (const s1) "!=" ( const s2 )
function const r = std (const s1) "<" ( const s2 )
function const r = std (const s1) "<=" ( const s2 )
function const r = std (const s1) ">" ( const s2 )
function const r = std (const s1) ">=" ( const s2 )
```

Standard lexigraphic comparison operators on strings `s1` and `s2`. Characters are compared by comparing their `UTF-8` representations as strings of unsigned 8-bit bytes. With exceptions for unnormalized `UTF-8` encodings,[8] this is equivalent to comparing the characters by comparing their UNICODE codes as unsigned 32-bit integers.

`function const r = std` **`explode`** `( const s )`

Returns a map that is a vector whose elements are unsigned integer `numbers` equal to the UNICODE codes of the characters of string `s`.

`function const r = std` **`implode`** `( const m )`

Given a map `m` that is a vector whose elements are unsigned integer `numbers` that are UNICODE codes of characters, return the string whose characters are those specified by the map elements in the order specified by the map.

`function const r = std` **`compile re`** (const s)

Compile the regular expression represented by the string `s` and return an integer that references the compiled expression.

Regular expressions are those recognized by the `pcre32` subroutine library for linux: see `pcrepattern[3]` in the linux documentation. The only line ends recognized by `\R` and `$` are LF, CR, and CRLF (no other `pcre32` options are used). By default `^` matches the beginning of the string being matched and `$` matches the end. This can be changed by the `(?i)` option setter in the regular expression.

`function` **`free re`** (const i)

Free the memory used by the compiled regular expression referenced by the integer `i`. Does nothing if `i` does not reference a compiled regular expression.

`function const r = std` **`match re`** (const i, const s)

Matches the string `s` to the compiled regular expression referenced by the integer `i`. Returns a map `r` that is a vector of substrings matched. If there is no match this is an empty list. If there is a match, `r[0]` is the string matched. If there are subpattern matches, `r[i]` is the string matched by the `i`'th subpattern.

[8] An unnormalized `UTF-8` encoding for a character is one taking more bytes than necessary. For example, `NUL` with UNICODE code `0`, can be encoded in 1-byte if normalized, or in 2-, 3-, or 4- bytes unnormalized.

During matching `s` is stored as an exploded vector of unsigned 32-bit unicode values. Substrings matched are subvectors which are imploded to make `const` string values.

`function const r = std` **`scan`** (const s)

Scan the string `s` and return a map that is a vector containing the list of lexemes in `s`. Brackets and operators are not specially recognized and are returned as strings. Quoted strings inside `s` are returned as vector elements that are maps of the form:
`{` *represented-string*`, ".type" => "<Q>" }`

Syntax errors produce compiler error messages and return `UNDEF`.

`function const r = std` **`parse brackets`** (const s)

Equivalent to '*value of* `s`'. See *phrase-constants*: p36.

Specifically, parse the string `s` recognizing brackets but not recognizing operators and return a map. Syntax errors produce compiler error messages and return `UNDEF`.

`function const r = std` **`unparse brackets`** (const v, const f ?= "%0.16g")

Inverse of `parse brackets`. Return a string S such that ' "S" ' equals `v` with numeric rounding differences. Numbers are printed in S using `f` as a printf format.

`function const r = std` **`parse`** (const s)

Equivalent to `{*` *value of* `s *}`. See *expression-constants*: p36.

Specifically, parse the string `s` and return a map. Syntax errors produce compiler error messages and return `UNDEF`.

`function const r = std` **`unparse`** (const v, const f ?= "%0.16g")

Inverse of `parse`. Return a string S such that `{* "`S`" *}` equals `v` with numeric rounding differences. Numbers are printed in S using `f` as a printf format.

17.4 Compile Time Map Functions

A `const` map value is actually a pointer to the map, and not the whole map itself. A *map-constant* creates a new map, distinct from every other map (so you can have multiple different empty maps).

A map may be read-write or read-only. Read-only maps cannot be modified. Each *map-constant* makes a separate read-only map that can be made read-write permanently or temporarily by the following:

`function const r = std` **`read-write`** `( const m )`
`function const r = std` **`read-only`** `( const m )`

Makes the map `m` read-write or read-only and returns `m`.

When the map is created by a *map-constant*, it is made read-only.

Each map *dictionary-entry* can be separately made read-write or read-only. When created, the entry is read-write. This can be changed by the following:

`function const r = std` **`read-write`** `( const m, const s )`
`function const r = std` **`read-only`** `( const m, const s )`

Makes the map label `s` (a string) of the map `m` read-write or read-only, and returns `m`. The map itself may be read-only or read-write.

When a value is first written at a label, the label is created for the map and set to read-write.

In addition, map labels may be **protected**. Such entries are read-only to the code being compiled, and may either be permanently read-only or may be written only by the compiler during compilation.

`function const r = std "#" ( const m )`

Returns the length of the vector part of the map `m` as a non-negative integer `number`.[9] Note that `#` is a prefix operator.

`function const r = std` **`labels`** `( const m )`

Returns a list of the *dictionary-labels* (see p35) of map `m`. The *dictionary-labels* are strings. The list may be empty.

`reference function const r = std ( const m ) [ const s ?= NONE ]`
`reference function std ( const m ) [ const s ?= NONE ] = const v`

Here `m` is map and `m[s]` is used to reference a vector element or dictionary entry of `m` as follows:

1. If the value of `s` is a non-negative integer number (and not a rational), the `s`+1'st element of the vector of `m` is referenced. If this is being read and it does not exist, `NONE` is returned and there is no compile error. If the element is being written but does not exist, or the map is read-only, a compile error results; otherwise the element value is changed.
2. If the value of `s` is a negative integer, `s` is replaced by `# m + s`, and things are as in the last paragraph (note `# m + s < # m`). The element does not exist if `# m + s < 0`.
3. If the value of `s` is a string, the dictionary entry of `m` with label `s` is referenced. If this is being read and it does not exist, `NONE` is returned. If it is being written and it does not exist, and if the map is read-write, the entry is created and made read-write. Else if the entry exists and is read-write, and the map is read-write, the entry value is changed. Else if the entry exists and is read-only, or the map is read-only, a compile error results.

[9] The length of a `map` cannot be above 2^{48} while `numbers` can precisely store integers up to 2^{53}.

4. If the value of `s` is `NONE`, reading pops a value from the end of the vector of `m` and returns the value popped (as in '`v = m[]`'), or just returns `NONE` if the vector is empty, and writing pushes the value written to the end of the vector of `m` (as in '`m[] = v`').
5. Otherwise if `s` is neither an integer or a string, a compile-error results.

`function const r = std copy ( const m )`
`function const r = std copy top ( const m )`

Returns a new map whose contents is a copy of the contents of map `m`. The new map is read-only if and only if `m` is, and the labels in the new map are read-only if and only if the corresponding labels in `m` are.

If any vector or dictionary element values are maps, they are copied recursively by `copy`, but not by `copy top`, which only copies the element values of `m` and does not copy recursively.

Changing the values of elements of the new map will not change the contents of `m`. For `copy` modifying the element values that are maps will not change `m`, but for `copy top`, modifying these element values will change the elements of `m`.

`function const r = std duplicate ( const m )`
`function const r = std duplicate top ( const m )`

Ditto, but if any read-only map is to be copied, the pointer to the map is copied and no new map is made.

`function const r = std slice ( const m, const i, const n )`

If `i >= 0`, returns a new map consisting of just a vector of the elements `m[i]`, `m[i+1]`, ..., `m[i+n-1]`. If any of these elements do not exist, they are omitted (e.g., if `i >= # m` or `i + n <= 0` or `n <= 0` the empty map is returned).

If `i` < 0 it is replaced by `# m + i`. If `# m + i < 0`, `i` is incremented by 1 and `n` decremented by 1 until `# m + i == 0`.

`function const r = std splice ( const m, const i, const n, const v )`

If `i >= 0` and `n > 0` (these must be integers), edits `m` by replacing the vector element sequence `m[i]`, `m[i+1]`, ..., `m[i+n-1]` by the vector elements of `v`. Dictionary elements of `v` are ignored; dictionary elements of `m` are unchanged.

If `i >= # m` the elements of `v` are pushed to the end of `m`.

Else if `n <= 0`, the elements of `v` are inserted before `m[i]`.

Else if `i + n > # m`, `n` is decreased until `i + n == # m`.

If `i` < 0 it is replaced by `# m + i`. If `# m + i < 0`, `i` is incremented by 1 and `n` decremented by 1 until `# m + i == 0`, and then if `n <= 0` the vector elements are inserted at the beginning of the vector.

The edited map `m` is returned.

`function const r = std truncate ( const m, const i )`

An optimized version of `splice` that removes the elements `m[i]`, `m[i+1]`, ..., from the end of the vector of `v`. The edited map `m` is returned. It is not an error if no elements are removed.

If `i` < 0 it is replaced by `# m + i`. If `# m + i < 0`, all elements of the vector are removed.

`function const r = std push ( const m, const v )`

Appends `v` to the vector of `m` and returns `m`.

`function const r = std push ( const m, const v, const i )`

Executes `push(m,v) i` times. It is an error if i is a negative integer.

`function const r = std append ( const m1, const m2 )`

Appends the vector elements of the map `m2` to the vector of `m1` and returns `m1`.

`function const r = std pop ( const m )`

Deletes the last vector element of `m` and returns it. Returns `NONE` if `m` is empty.

`function const r = std pop ( const m, const i )`

Deletes the last `i` vector elements of `m` and returns a map containing them in the same order. If there are fewer than `i` vector elements in `m`, the returned vector will have only `# m` elements.

17.5 Type, Field, Subfield, and Pointer Type Maps

Types, fields, subfields, and pointer types are described at compile-time by `const` map values which user code can access. These are read-write as a whole, but some of their labels are protected. The following sections describe protected labels provided by the compiler. Unless otherwise specified, these have values that do not change during compilation.

Compiled code may add its own labels to these maps. To prevent conflict, the labels provided by the compiler begin with '`.`', so that to use them to access a map dictionary entry you must use double dots: '`..`'. E.g., `int..size`.

In the following a **name string** is a string consisting of a sequence of one or more *words* and *natural-numbers*, separated by single spaces, and beginning with a *word*. *Natural-numbers* are represented by strings of 1 to 9 decimal digits with no high-order zeros (zero is represented by '`0`'). Name strings are used to represent type, field, and subfield names.

Module abbreviations in a name string are replaced by **compiler module abbreviations** which are words of the form `M$`n, where n is a natural number. `M$0` is always the abbreviation for the `std` module. These compiler module abbreviations are specific to the entire

compilation and are not dependent on which module or body a definition appears in.

`com module dictionary`

A dictionary mapping compiler module abbreviations to strings that are *module-names*. For example,

```
com module dictionary["M$0"] == "standard"
```

17.5.1 Type Maps

At compile-time a *type-name* can be used as a `const` type *variable-name* that names a read-only variable with a map value called a **type map**.

The compiler defined attributes of a type map are:

`.type => "type"`

`.name`

The name of the type as a name string.

`.size`
`.alignment`

The `.size` is the number of bits taken by a value of the given type at run-time. The `.alignment` is the alignment in bits of an aligned value of the given type at run-time. E.g., `int64..size == 64, int64..alignment == 64`.

These may increase during compilation of type expansions, and will be `UNDEF` for `*DEFERRED*` types not yet defined by the compilation.

`.expandable`
`.external`

The `.expandable` attribute is `TRUE` if the current list of type subdeclarations ends with `***` or `*EXTERNAL*`, and `FALSE` otherwise. The `.external` attribute is `TRUE` if the current list of type subdeclarations ends with `*EXTERNAL*`, and `FALSE` otherwise.

These may change during compilation of type expansions, and are `UNDEF` for `*DEFERRED*` types not yet defined by the compilation.

`.fields`

A map listing field maps p118 for the fields of the type. The dictionary entries list named fields by name. The vector entries list unnamed fields (which have subfields).

Fields may be added during compilation of type expansions. Will be empty if a type has no fields, or if the type is currently `*DEFERRED*` and not yet defined by the compilation.

17.5.2 Field Maps

Each field of a type has a `const` map value called a **field map** which is in the `.fields` dictionary of a type map. The compiler defined attributes of a field map are:

`.type => "field"`

`.name`

The name of the field (*target-label* or *pointer-label*) as a name string. May be `NONE` for a field with subfields.

`.parent`

Type map of the type of containing this field.

`.offset`

Offset in bits of the field within a value of its parent type.

`.pointer-type`

Pointer type map for the pointer type of the field, or `NONE`.

`.pointer-qualifiers`

List of strings, each a *word* naming a qualifier of the field pointer type, or `NONE` if there is no pointer type. May be empty list.

`.field type`

Type map for the type of the field if the `.pointer-type` is `NONE`, or the `.pointer-type` target if the `.pointer-type` is not `NONE`, or `NONE` for a `*LABEL*`.

`.qualifiers`

List of strings, each a *word* naming a qualifier of the field. May be empty list. If `.pointer-type` is not `NONE`, these qualifiers apply to the pointer value of the field and not to its target.

`.dimensions`

List of strictly positive integers, the dimensions of the field, or `NONE` if no dimensions.

`.subfields`

Dictionary of subfield maps for the subfields of the type. The labels of the dictionary entries are the names of the subfields. Empty if there are no subfields.

17.5.3 Subfield Maps

Each subfield of a field has a `const` map value called a **subfield map** which is in the `.subfields` dictionary of a field map. The compiler defined attributes of a subfield map are:

`.type => "subfield"`

`.name`

The name of the subfield (*target-label*) as a name string.

`.parent`

Field map of the field of containing this subfield.

`.bits`

A list of two integers: {*highbit*, *lowbit*}.

`.subfield type`

Type map for the type of the subfield. This is always a `std` number type or `std bool`.

`.dimensions`

List of strictly positive integers, the dimensions of the subfield, or `NONE` if no dimensions.

17.5.4 Pointer Type Maps

At compile-time a *pointer-type-name* can be used as a `const` type *variable-name* that names a read-only variable with a map value called a **pointer type map**.

The compiler defined attributes of a pointer type map are:

`.type => "pointer type"`

`.name`

The name of the pointer type as a name string.

`.data type`

Type map for the data type of the pointer type.

17.6 Compile-Time Machine Parameters

`const com atomc types = { "int", "uns", ... }`

These are the types for which atomic operations p127 are defined.

`const com hardware overflow = ...` [`TRUE` or `FALSE`]

`TRUE` iff hardware computes the overflow `bool` for integer addition and subtraction. p125 `FALSE` if this is computed when needed by software (much more slowly).

18 Builtin Run-Time Functions and Constants

Run-time functions execute at run-time, and but may have parts that execute at compile-time, and may even return `const` results.

The L-Language built-in run-time functions are very basic and provide only functionality that cannot be efficiently provided by library functions.

18.1 Builtin Run-Time Constants

```
bool std true = 1
bool std false = 0
```

18.2 Builtin Implicit Conversions

See 9.1.5 [p61] for non-builtin conversions.

18.2.1 Qualifier Implicit Conversions

Implicit conversions of qualifiers may occur whenever a pointer value is copied, unlike other implicit conversions. The following are qualifier conversions:

1. `co` may be replaced by `ro`
2. `*READ-WRITE*` may be replaced by `ro`
3. `*READ-WRITE*` may be replaced by `*WRITE-ONLY*`
4. `*GLOBAL*` may be replaced by `*HEAP*`
5. `*GLOBAL*` may be replaced by `*LOCAL*` with depth `0`
6. `*HEAP*` may be replaced by `*LOCAL*` of depth 0 when copying from a *reference-expression* that is nothing but a *pointer-variable* (i.e., is a stack variable which is `co`).

18.2.2 Numeric Implicit Conversions

Any value of number or `bool` type `N1` can be implicitly converted to a value of number type `N2` if every value of type `N1` can be precisely represented by a value of type `N2`. More specifically, the implicit conversions are defined by:

```
function N2 r = std *IMPLICIT* *CONVERSION* ( N1 v )
```

in the following cases:

	N1		
N2	flt64	flt32	flt16
flt64	no	yes	yes
flt32	no	no	yes
flt16	no	no	no
int...	no	no	no
uns...	no	no	no

	N1			
N2	int64	int32	int16	int8
flt64	no	yes	yes	yes
flt32	no	no	yes	yes
flt16	no	no	no	yes
int64	no	yes	yes	yes
int32	no	no	yes	yes
int16	no	no	no	yes
int8	no	no	no	no
uns...	no	no	no	no

	N1				
N2	uns64	uns32	uns16	uns8	bool
flt64	no	yes	yes	yes	yes
flt32	no	no	yes	yes	yes
flt16	no	no	no	yes	yes
int64	no	yes	yes	yes	yes
int32	no	no	yes	yes	yes
int16	no	no	no	yes	yes
int8	no	no	no	no	yes
uns64	no	yes	yes	yes	yes
uns32	no	no	yes	yes	yes
uns16	no	no	no	yes	yes
uns8	no	no	no	no	yes

18.3 Builtin Explicit Conversions

See 9.1.5 [p61] for non-builtin conversions.

18.3.1 Numeric Explicit Conversions

```
function F r = std F (N v)
```

Where `F` is any builtin floating point type and `N` is any builtin number or `bool` type.

Converts `v` to the type `F`. The result may be `+Inf` or `-Inf`, or precision may be lost.

```
function I r = std floor (F v1, F v2 ?= 1.0 )
function I r = std ceiling (F v1, F v2 ?= 1.0 )
function I r = std truncate (F v1, F v2 ?= 1.0 )
function I r = std round (F v1, F v2 ?= 1.0 )
```

Where `I` is any builtin signed integer type and `F` is one of `flt`, `flt64`, or `flt32`.

These divide `v1` by `v2` and return `r` as the result rounded to an integer. Here `floor` rounds toward negative infinity, `ceiling` rounds towards positive infinity, `truncate` rounds toward zero, and `round` rounds to the nearest integer, or to the even integer if there are two nearest integers.

The floating point flags set are those set by division (see below), plus the inexact flag may be set if the division quotient is not a integer, plus the invalid flag is set if the result is outside the range storable in `I`. In the last case, and the result is an indefinite integer. [p126]

```
function I1 r = std *UNCHECKED* ( I2 v )
```

Where `I1` and `I2` are any builtin integer types such that at least one of the following is true:

1. `I1` is shorter than `I2`
2. `I1` and `I2` are of equal length and one is `int...` while the other is `uns...`
3. `I2` is `int...` while `I1` is `uns...`.

These in effect convert `v` to a bit-string of unbounded length (e.g., by two's complement sign extension) and then truncate it to the length of `I1`.

18.4 Builtin Floating Point Operations

A floating point `NaN` is a quiet NaN with zero significand bits, except for the highest order bit which is one (to indicate that the NaN is quiet).

```
function F r = std "+" ( F v1 )
function F r = std "-" ( F v1 )
function F r = std (F v1) "+" ( F v2 )
function F r = std (F v1) "-" ( F v2 )
function F r = std (F v1) "*" ( F v2 )
function F r = std (F v1) "/" ( F v2 )
```

Where `F` is one of `flt`, `flt64`, or `flt32`.

Standard arithmetic operators on numbers `v1` and `v2`, done using IEEE floating point arithmetic.

Floating point operations may set the following floating point flags:

Invalid — Set in the following cases. Returns `NaN`.

```
+Inf + -Inf          -Inf + +Inf
+Inf - +Inf          -Inf - -Inf
+Inf * 0             0 * +Inf
-Inf * 0             0 * -Inf
+Inf / +Inf          +Inf / -Inf
-Inf / +Inf          -Inf / -Inf
+0 / +0              +0 / -0
-0 / +0              -0 / -0
```

Divide by Zero — Set when a non-zero value is divided by a zero value. Returns `+Inf` or `-Inf` with sign determined by the signs of the zero and non-zero values in the usual way.

Overflow — Set when the computed value is a number outside the range that can be stored because its absolute value is too large. Returns `+Inf` or `-Inf`.

Underflow — Set when the computed value is a number outside the range that can be stored because its absolute value is too small. Returns `+0` or `-0`.

Inexact — Set when the computed value cannot be precisely stored but is inside the range of absolute values that can be stored. Returns the nearest value that can be stored, with ties going to the value whose least significant bit is zero.

```
function F r = std floor (F v1, F v2 ?= 1.0 )
function F r = std ceiling (F v1, F v2 ?= 1.0 )
function F r = std truncate (F v1, F v2 ?= 1.0 )
function F r = std round (F v1, F v2 ?= 1.0 )
```

Where `F` is one of `flt`, `flt64`, or `flt32`.

These divide `v1` by `v2` and return `r` as the result rounded to an integer. Here `floor` rounds toward negative infinity, `ceiling` rounds towards positive infinity, `truncate` rounds toward zero, and `round` rounds to the nearest integer, or to the even integer if there are two nearest integers.

The floating point flags set are those set by division (see above), plus the inexact flag may be set if the division quotient is not a integer. If the division quotient is an infinity, no flags are set and `r` is set to the quotient.

```
function bool r = std (F v1) "==" ( F v2 )
function bool r = std (F v1) "!=" ( F v2 )
function bool r = std (F v1) "<" ( F v2 )
```

```
function bool r = std (F v1) "<=" ( F v2 )
function bool r = std (F v1) ">" ( F v2 )
function bool r = std (F v1) ">=" ( F v2 )
```

Where `F` is one of `flt`, `flt64`, or `flt32`.

Standard comparison operators on floating point numbers `v1` and `v2`. Infinities are treated as actual numbers with absolute value larger than any real number: e.g., if `x` is not a `NaN`, '`x <= +Inf`' is always true and '`x == +Inf`' is true iff `x` is `+Inf`. If an argument is a `NaN`, the result is undefined and an invalid flag is set.

```
function bool r = std is nan ( F v1 )
function bool r = std is infinity ( F v1 )
function bool r = std is finite ( F v1 )
```

Where `F` is one of `flt`, `flt64`, `flt32`, or `flt16`.

`is nan`	Returns `true` if `v1` is any NaN number (not just `NaN`), and `false` otherwise.
`is infinity`	Returns `true` if `v1` is `+Inf` or `-Inf`, and `false` otherwise.
`is finite`	Returns `true` if `is nan` and `is infinity` both return `false`, and returns `false` otherwise.

```
type std FP flags:
     uns flags
     [...]  bool invalid (operand for a given operation)
     [...]  bool divide by zero
     [...]  bool overflow
     [...]  bool underflow
     [...]  bool inexact
function std FP flags r = std FP flags register
function std FP flags register = std FP flags v
```

The hardware floating point flags can be read into a datum of type `FP flags` and an `FP flags` datum can be written to the hardware floating point flags by using the `FP flags register` functions. The exact bits in the data that contain the flags are implementation dependent, and are here represented by '`...`'.

18.5 Builtin Integer Operations

```
function I r, bool cout, bool ovfl = std "+" ( I v1, bool cin = 0 )
function I r, bool cout, bool ovfl = std "-" ( I v1, bool cin = 1 )
function I r, bool cout, bool ovfl = std (I v1) "+" ( I v2, bool cin = 0 )
function I r, bool cout, bool ovfl = std (I v1) "-" ( I v2, bool cin = 1 )
```

Where I is one of: `int int128 int64 int32 int16 int8`
`uns int128 uns64 uns32 uns16 uns8`

Standard arithmetic operators on `v1` and `v2` treated as binary unsigned integers. When values are interpreted as two's complement signed integers, these operations also give valid results.

`cin` is added to the result; `cout` is the carry from the result. Operands are made negative by bitwise complementing them and adding 1 by setting `cin = 1`.

`ovfl` is set to `1` if and only if the operation overflows when values are interpreted as signed two's complement integers. If S0, S1, and Sr are the signs of `v1`, `v2`, and `r`, for two operand "+" this would equal `S0 = S1 ≠ Sr`. `ovfl` is computed by some hardware, but is expensive to compute if not supported by hardware. See `com hardware overflow`. p119

```
function I r = std (I v1) "*" ( I v2 )
function U r, U cout = std (U v1) "*" ( U v2, U cin1 = 0, U cin2 = 0 )
```

Where I is one of: `int int128 int64 int32 int16`
`uns uns128 uns64 uns32 uns16`

and U is one of: `uns uns64 uns32`

The version without carries is the standard arithmetic multiply which truncates results that are outside the range of `I`.

The version with carries is integer multiply of N-bit unsigned integers to produce a 2N-bit product to which both `cin1` and `cin2` are added. Of the result `r` is the low order N bits and `cout` is the high order N bits.

```
function I r = std (I v1) "/" ( I v2 )
function U r, U cout = std (U v1, U cin = 0) "/" ( U v2 )
```

Where I is one of: `int int128 int64 int32 int16`
`uns uns128 uns64 uns32 uns16`

and U is one of: `uns uns64 uns32`

The version without carries is the standard arithmetic divide. An exception trap occurs if `v2 = 0`.

The version with carries is integer divide of a 2N-bit unsigned dividend made by concatenating `cin` (high order) and `v1` (low order) by an N-bit unsigned divisor `v2`. The result is an N-bit quotient `r` and an N-bit remainder `cout`. An exception trap occurs if `v2 ≤ cin` (this includes that case where `v2 = 0`).

```
function I r = std "~" ( I v1 )
function I r = std (I v1) "&" ( I v2 )
function I r = std (I v1) "|" ( I v2 )
function I r = std (I v1) "^" ( I v2 )
```

Where `I` is one of: `int int128 int64 int32 int16 int8`
`uns uns128 uns64 uns32 uns16 uns8 bool`

Standard bitwise operators, complement (`~`), and (`&`), or (`|`), and exclusive or (`^`), on integers `v1` and `v2` that are treated as two's complement if signed.

```
function I r = std (I v1) "<<" ( int v2 )
function I r = std (I v1) ">>" ( int v2 )
```

Where `I` is one of: `int int128 int64 int32 int16 int8`
`uns uns128 uns64 uns32 uns16 uns8`

Standard bitwise shifts of integer `v1` by the amount `v2`. If signed, `v1` is treated as two's complement. The amount of shift, `v2`, must be in the range [0,`I..size`); values out of range produce undefined results. Bits shifted out at the left or right side are discarded.

```
function bool r = std (I v1) "==" ( I v2 )
function bool r = std (I v1) "!=" ( I v2 )
function bool r = std (I v1) "<" ( I v2 )
function bool r = std (I v1) "<=" ( I v2 )
function bool r = std (I v1) ">" ( I v2 )
function bool r = std (I v1) ">=" ( I v2 )
```

Where `I` is one of: `int int128 int64 int32 int16 int8`
`uns uns128 uns64 uns32 uns16 uns8 bool`

Standard comparison operators on integers `v1` and `v2`.

```
function I r = std indefinite integer
```

Where `I` is any signed integer type.

This function returns the value that is returned when some operations, e.g. `round`, cannot return a correct integer value.[10]

The value is typically -2^{S-1} where S is the size of `I` in bits.

18.6 Builtin Pointer Operations

```
function P$1 Q$1 T$1 r = std null
```

This function returns zero converted to the pointer type.

Accessing the target of a null pointer usually produces a segmentation fault, as the virtual page with address zero is usually undefined.

This function cannot be called by an `allocation-call` p43.

```
function P$1 QL$1 *LOCAL* T$1 r = std local ( uns length, uns alignment )
```

[10] See documentation of the intel IA-64 FIST instruction.

This function allocates a value of type `T$1` to local memory (the current out-of-line function execution's frame in the local stack: see p105) and returns a pointer to the value.

This function can only be called by an `allocation-call` [p43]. See p47.

```
function P$1 QG$1 *GLOBAL* T$1 r = std global ( uns length, uns alignment )
```

This function allocates a value of type `T$1` to global memory (the global stack: see p105) and returns a pointer to the value.

This function can only be called by an `allocation-call` [p43]. See p47.

```
function bool r = std is set ( N value )
function bool r = std is set ( P$1 Q$1 T$1 pointer )
```

where `N` is any number type.

Returns `true` if the `value` is not zero or the `pointer` is not `null`, and `false` otherwise. This function should only be used inside `*EXCEPTION*` *exit-subblocks* [p47].

19 Atomic Operations

Atomic operations read and write memory locations that are shared between multiple CPUs or processes using the same RAM memory.

When a process executes a program, the process may move memory reads and writes around so they are no longer done when they would be done were the program statements executed in strict sequential order. The execution does this if the effect of the program is not changed, under the assumption that the program is the only user of the RAM memory. But if there are multiple CPUs, or a multi-tasking system asynchronously switching one CPU between processes, this assumption is not correct.

There is also the possibility that if a memory location is being read by CPU 1 and at the same time written by CPU 2, what CPU 1 reads will consist partly of the location value before CPU 2's write and partly of the location value after CPU 2's write.

An operation is **atomic** if:

1. All read and or write operations and all atomic operations for the current process that would be before (or after) this atomic operation in strict sequential program execution are before (or after) this atomic operation in actual optimized program execution.
2. This atomic operation cannot be interrupted by a read or write executed by a different CPU or process of part of the memory being read or written by this atomic operation.

Atomic operations can only be performed on locations of **atomic type**. Atomic types are

listed in the `com atomic types`[p119] compile-time variable. The word-length integer types, `int` and `uns`, are always atomic types.

`function A r = std atomic read ( ap QR$1 A p )`

For `A` an atomic type, read and return the location pointed at by `p`. In addition:

1. All read or atomic operations for the current process that would be before (or after) this operation in strict sequential program execution are before (or after) this operation in actual optimized program execution.
2. The read cannot be interrupted by a write executed by a different CPU or process of part of the memory being read.

`function std atomic write ( ap QW$1 A p, A v )`

For `A` an atomic type, write the value `v` to the location pointed at by `p`. In addition:

1. All write or atomic operations for the current process that would be before (or after) this operation in strict sequential program execution are before (or after) this operation in actual optimized program execution.
2. The write cannot be interrupted by a read executed by a different CPU or process of part of the memory being read.

`function std bool r = atomic compare and set ( ap QRW$1 A p, A vr, A vw )`

For `A` an atomic type, read the location pointed at by `p` and compare it to `vr`. If equal, write `vw` to the location, and return `true`. If not equal, just return `false`. In addition:

1. All read and write and atomic operations for the current process that would be before (or after) this operation in strict sequential program execution are before (or after) this operation in actual optimized program execution.
2. The operation cannot be interrupted by a read or write executed by a different CPU or process of any of the memory this operation reads or writes.

Example 1:

A device has registers in global memory which are shared between the device an a process. The registers are either owned by the process and the device is inactive, or the registers are owned by the device which is executing an operation. There is a register with a GO bit which is turned on by the process to activate the devices, and another register with a READY bit which is set by the devince when its current operation is done.

The GO bit is set by an atomic write, which guarentees that all writes to registers in the code before the GO bit is set are actually done before the GO bit is set.

The READY bit is read by an atomic read, which guarentees that all reads from registers in the code after the READY bit has been read as being on are actually done

after the READY bit has been read as being on.

Example 2:

A device has registers as in Example 1. The device also has a large memory accessed via two registers: A and D. A holds the address of a location in device memory, and D holds the contents of that location: reading D reads the location contents, and writing D writes the location contents.

Register A is set by an atomic write and then register D is read by an atomic read. Because atomic operations execute in the same order as they appear in the code, the read will be done after the write.

Example 3:

A data structure which is shared among processes perhaps executing on different CPUs is guarded by a lock consisting of two integer memory locations: B (before) and A (after). When the structure is not being written, B == A. A writer first increments B, then updates the structure, and then increments A.

To get a write lock, the writer reads A atomically and then does a compare and set on B that checks that B == A and if so writes B+1 to B and acquires the lock.

To read, the reader reads A atomically and saves the value V, then reads data from the structure, then reads B atomically and checks that the value read equals V. If the check passes, the data is uncorrupted by writing that is simultaneous with the reading. If the check fails, the data may be corrupted. The reader must be sure corrupted data does not destroy the integrity of the reader's execution, but can read B atomically and check the value against V at any time to see if the data read so far is uncorrupted.