# Gamma Programming Language Specification

**GitHub Page**

# Contents

# Introduction

The Gamma programming language is a *source language* in which to write *source code*. A *translator translates* source code into *object code* written in an *object language*.

This document describes the process of translating Gamma source code and the *behavior* of object code, to facilitate comparison of translations thereof.

## Interpretation

This document has a prescriptive intent: a translator behaves so that all statements in this document hold.

## Conventions

The following symbols and patterns shall be interpreted as follows:

- `a` represents an occurrence in Gamma source code of the string "a", or the regular expression matching the string "a";
- $a$ represents the regular expression explicitly named "a";
- $R|S$ represents the union of the regular expressions named "R" and "S";
- $R - S$ represents the difference between the regular expressions named "R" and "S";
- $R^? = |R$ represents the optional application of the regular expression named "R";
- $R^+$ represents the repetition of the regular expression named "R" one or more times;
- $R^* = |R^+$ represents the repetition of the regular expression named "R" zero or more times;
- $R^-$ represents the complement of the regular expression named "R";
- $\mathtt{a}$ represents an occurrence in Gamma source code of a token in the class named "a";
- `{a}` represents one or more occurrences of the string "a";
- `[a]` represents the optional occurrence of the string "a";
- `a|b` represents the occurrence of either one of the strings "a" or "b".

# 1 Syntax

The Gamma programming language is a LR(1) language of *translation units*, which are sequences of *tokens*. Tokens, in turn, are strings in a regular language over an *alphabet*.

Translators accept only *well-formed* source code. Source code is well-formed only if the statements not qualifying it as otherwise in this document hold.

# 1.1 Language

## 1.1.1 Alphabet

The alphabet of the Gamma programming language is in the Unicode 10 character set. Several subsets are named here for future reference in this document:

- Letters, consisting of the Unicode General Categories *Li*, *Lu*, *Lt*, *Lm* and *Lo*, named $letter$;
- Digits, consisting of the following subsets:
  - the decimal digits `0123456789`, named $digit$,
    and the nonzero digits $ndigit = digit - \texttt{0}$;
  - the binary digits `01`, named $bdigit$;
  - the octal digits `01234567`, named $odigit$;
  - the hexadecimal digits `0123456789ABCDEF`, named $hdigit$;
- Alphanumerics, named $alpha = digit|letter$;
- Punctuation and delimiters, consisting of `()[]{}'":;,`;
- Symbols, consisting of `@.\/%^*+-#&|~=<>!?:`;
- Whitespace, consisting of the Unicode General Categories *Zs*, *Zl*, *Zp* and *Cc*; and
- The underscore `_`.

## 1.1.2 Tokens

The tokens of the Gamma programming language consist of the union of the following sets of strings:

- The keywords `space type sym with data code if elif case is else do while for jump exit return end`;
- The operators `#:: %:: . @ / % ^^ ** * + - # & | ~ == =< >= < > >< ## && || ! ?: ::`;
- The delimiters `() [] {}`;
- Names, defined by the regular expression $name$
  $(\texttt{\_}|letter)(\texttt{\_}|alpha)^*$
- Identifiers, defined by the regular expression $identifier$
  $name(\backslash name)^*$
- Numbers, defined by the regular expressions $number$
  $ndigit(digit^+(.digit^+)^?|(.digit^+)^?(\texttt{e-}^? ndigit\, digit^*)^?)$
  $\texttt{0}(\texttt{b}\, bdigit^+|\texttt{o}\, odigit^+|\texttt{x}\, hdigit^+)$
- Strings, defined by the regular expression $string$
  $\texttt{"}(\texttt{"}^-|\backslash(\texttt{"}|\backslash))^*\texttt{"}$
- Labels, defined by the regular expression $label$
  $name\texttt{:}$
  where $name$ is the label's *value*
- Comments, defined by the regular expressions
  $\backslash\texttt{*}(\texttt{*}\backslash)^-\texttt{*}\backslash$
  $\backslash\backslash(\backslash\backslash|newline)^- newline$

# 1.2 Translation Units

Translation units are either *header units* or *source units*.

For symbols not defined in this section, see **§ A**.

## 1.2.1 Header Units

A *header unit* is a translation unit which contains type definitions or *declarations*.

### 1.2.1.1 Namespaces

A namespace introduces a name used to create identifiers.

A namespace is defined with the following syntax:
```
space name {declaration} end
```

### 1.2.1.2 Types

A type is the association of a set of *values* with *operations* over these values and other types.

Types are defined with the following syntax:
```
type symbol[{, symbol}]
```

### 1.2.1.3 Declarations

A declaration describes a *symbol*. A symbol is the association of an *name* with a type.

Symbols are declared with the following syntax:
```
sym symbol[{, symbol}]
```

## 1.2.2 Source Units

A *source unit* is a translation unit which contains *type names* or *definitions*. Definitions describe either *datums* or *subprograms* at the beginning of execution (see **§ 3**).

### 1.2.2.1 Data

A datum is the association of a symbol with an *initial value* of the symbol's type.

Datums are defined with the following syntax:
```
data datum[{, datum}]
```

### 1.2.2.2 Subprograms

A subprogram is the association of a symbol with a sequence of *instructions*.

A subprogram is defined with the following syntax:

```
code symbol
       instructions
end
```

A subprogram has a *body* containing one or more *instructions*. An instruction is either a *statement* or a *definition*.

#### 1.2.2.2.1 Statements

A statement is either a *simple statement*, a *primitive statement*, or a *composite statement*. Any statement may be *labeled*.

### Simple Statements

A simple statement changes the values of the module's data. A simple statement has the following form:
`expression_term [(assignment|= [{expression_term =}])expression]`
where `assignment` may be any one of the following *compound assignments*:

- `/=`, `%=`;
- `^^=`, `**=`;
- `*=`;
- `+=`, `-=`;
- `#=`;
- `&=`;
- `|=`, `~=`;
- `##=`;
- `&&=`; and
- `||=`, `!=`.

### Primitive Statements

A primitive statement is either a *JUMP statement*, an *EXIT statement*, or a *RETURN statement*.

A **JUMP statement** passes control to the containing iterative statement or the statement labeled by the argument. A JUMP statement has the following form:
`jump ([name])`
where `name` is its argument.

An **EXIT statement** passes control to the statement following the containing iterative statement or a containing iterative statement labeled by the argument. An EXIT statement has the following form:
`exit ([name])`
where `name` is its argument.

A **RETURN statement** passes control to the subprogram that called the subprogram containing it, at the point at which the containing subprogram was called. A RETURN statement has the following form:
`return ([expression])`
where `expression` is its argument.

### Composite Statements

Composite statements contain other instructions in one or more *blocks*. A composite statement is either a *conditional statement* or an *iterative statement*.

A **conditional statement** evaluates *guards* and uses the values to execute one block or *branch* out of several. A conditional statement is either an *IF statement* or a *CASE statement*.

**IF statements** have the following form:

```
if expression do
       instructions
[{elif expression do
       instructions}]
[else
       instructions]
end
```

where *expression* is a guard and *instructions* is a branch.

**CASE statements** have the following form:

```
case expression
{{is expression} do
       instructions}
[else
       instructions]
end
```

where *expression* is a guard and *instructions* is a branch.

An **iterative statement** executes a block as long as its *condition* is true. An iterative statement is either a *WHILE statement*, a *DO statement*, or a *FOR statement*.

**WHILE statements** have the following form:

```
while expression do instructions end
```

where *expression* is the condition and *instructions* is the block.

**DO statements** have the following form:

```
do instructions until expression end
```

where *expression* is the condition and *instructions* is the block.

**FOR statements** have the following form:

```
for [datum[{,datum}]];expression;simple_statement do instructions end
```

where:

- [*datum*[{, *datum*}]] are the initializations;
- *expression* is the condition;
- *simple_statement* is the update; and
- *instructions* is the block.

#### 1.2.2.2.2 Dynamic Definitions

A dynamic definition inside a subprogram body defines data or names or declares either data or subprograms with scope (see § 2.2.3) and lifetime (see § 3.1.1) limited to that of the containing block.

Symbols are declared with the following syntax:

```
sym symbol[{, symbol}]
```

Datums are defined with the following syntax:

```
data datum[{, datum}]
```

Subprograms are defined with the following syntax:

```
code symbol
        instructions
end
```

Names are defined with the following syntax:

```
with rename[{, rename}]
```

# 2 Semantics

A finite, non-empty group of translation units describes a *module*. Translators accept only *well-defined* modules. A module is well-defined only if the statements not qualifying it as otherwise in this document hold.

A module is an independent association of *data* and *code* stored in *memory*. A module's memory is modeled as a succession of *cells*, each of which is a *string* of *bits*. All cells have the same finite, fixed, translator-defined size. Each cell is assigned a unique natural integer called an *address*. If two addresses *succeed* each other, the cells they are assigned to are consecutive.

# 2.1 Types

A type, for the purposes of this specification, is an association of a set of *values* and *operations* over them. Memory stores *representations* of the values as bit strings.

## 2.1.1 Properties

All types may be *named* or *qualified*. If *T* is a name for a type *S* and *S* is a name for a type *U*, then *T* is a name for *U*. If *T* is a qualified type and *S* is a type derived from *T*, then *S* is a qualified type.

A type is *complete* if all of its values can be assigned a *size*. All complete types have an *alignment*.

### 2.1.1.1 Qualification

A type may be qualified as either *constant* or *result*.

### 2.1.1.2 Size

The size of a type is the number of consecutive cells in the representations of its values. The size of a type *type* is written `#::`*type*.

### 2.1.1.3 Alignment

The alignment of a type is the difference between the addresses of the two closest cells storing values thereof. The alignment of a type *type* is written `%::`*type*.

## 2.1.2 Varieties

Types are either *basic* or *derived*.

### 2.1.2.1 Basic

A basic type is either an *enumerated type* or an *aggregate type*.

#### 2.1.2.1.1 Enumerated Type

An enumerated type is a type whose values are enumerated, i.e. listed explicitly. An enumerated type is described with the following syntax:

`{` *name*`{, `*name*`}` `}`

Every enumerated type is complete; except where stated elsewhere, the size and alignment are translator-defined.

#### 2.1.2.1.2 Aggregate Type

An aggregate type is a type whose values are associations of *members*. An aggregate type is either a *record type* or a *union type*.

A **record type** is a type whose values are in the *product* of the types of its members. A record type is described with the following syntax:
`{ symbol{, symbol} }`
Every record type is complete: the size is the sum of the sizes of its members' types, and the alignment is the least common multiple of the alignments of its members' types. The members of a record type value have separate allocations, and allocations are in the order of appearance in the source code.

A **union type** is a type whose values are in the *union* of the types of its members. A union type is described with the following syntax:
`{ symbol{; symbol} }`
Every union type is complete: the size is the maximum of the sizes of its members' types, and the alignment is the least common multiple of the alignments of its members' types. The members of a union type value share a single allocation.

Operations accepting aggregate type values are:

- `.: type(aggregate_type, member_type)`, *member access*;
- `@: type(@aggregate_type, member_type)`, *member of pointed-to value access*.

## 2.1.2.2 Derived

A derived type is either a *pointer type*, an *array type* or a *signature type*.

### 2.1.2.2.1 Pointer

A pointer type's values store the addresses of values of that type. A pointer type is described with the following syntaxes:
For a pointer to a basic or pointer type `type`,
`@type`

For a pointer to an array type `type[]`,
`(@type)[]`

For a pointer to a signature type `[return_type]([symbol[{,symbol}]])`,
`(@[return_type])([symbol[{, symbol}]])`

Every pointer type is complete: the size and alignment are translator-defined. Nevertheless, pointer types have the special property that their alignment is the largest of all enumerated types.

The only operation accepting pointer type values, for every complete type `type`, is `@: type(@type)`, the *pointer dereference*.

The only operation yielding pointer type values, for every type `type`, is `.: @type(type)`, the *localization*.

### 2.1.2.2.2 Array

An array type's values associate finite numbers of values of a single type with an index. Array types may be either *static* or *dynamic*.

A **static array type** is described with the following syntax:
`type[expression]`

A **dynamic array type** is described with the following syntax:
`type[]`

Static array types are complete; dynamic array types are not.

Operations accepting and/or yielding array type values are:

- `#: nsize(`*type*`[])`, *size*;
- `+: @`*type*`(`*type*`[], nsize)`, *offset*;
- `[]:` *type*`(`*type*`[], nsize)`, *element access*.

(see § 2.1.3.1)

#### 2.1.2.2.3 Signature

A signature type's values describe a subprogram. A signature type is described with the following syntax: `[`*type*`]([`*symbol*`[{, `*symbol*`}]])`

Every signature type is complete: the size and alignment are translator-defined.

The only operation accepting signature type values is `()`, the *subprogram call*.

## 2.1.3 Provided

Translators name *fixed-point* and *floating-point number types*.

### 2.1.3.1 Fixed-Point Types

The following fixed-point types are defined by this specification in every module:

- the *natural integer type* `n1`, containing $[0; 2^8 - 1]$;
- the *relative integer type* `z1`, containing $[1 - 2^7; 2^7 - 1]$; and
- `byte`, a translator-defined name for either `n1` or `z1`.

In particular, `n1` and `z1` are assigned a size and alignment of $1$, so their values are exactly the values that each cell of memory can hold.

Translators may define the following additional fixed-point types:

- the *natural integer types*:
  - `n2`, containing $[0; 2^{16} - 1]$;
  - `n4`, containing $[0; 2^{32} - 1]$;
  - `n8`, containing $[0; 2^{64} - 1]$;
  - `n16`, containing $[0; 2^{128} - 1]$;
  - `n32`, containing $[0; 2^{256} - 1]$;
  - `nmax`, a name for the largest natural integer type defined by the translator; and
  - `nsize`, a name for the natural integer type able to represent the largest address available.
- the *relative integer types*:
  - `z2`, containing $[1 - 2^{15}; 2^{15} - 1]$;
  - `z4`, containing $[1 - 2^{31}; 2^{31} - 1]$;
  - `z8`, containing $[1 - 2^{63}; 2^{63} - 1]$;
  - `z16`, containing $[1 - 2^{127}; 2^{127} - 1]$;
  - `z32`, containing $[1 - 2^{255}; 2^{255} - 1]$;
  - `zmax`, a name for the largest relative integer type defined by the translator; and
  - `zsize`, a name for the relative integer type able to represent half the largest address available.

Their sizes are in a geometric progression of common ratio 2; their other properties are translator-defined.

The following operations accept and yield fixed-point type values, ordered by decreasing precedence:

1. - `+`: *type*(*type*), *unary plus*;
   - `-`: *type*(*type*), *unary minus*;
   - `~`: *type*(*type*), *unary bitwise complement*;
2. - `/`: *type*(*type*, *type*), *integer division*;
   - `%`: *type*(*type*, *type*), *remainder*;
3. - `**`: *type*(*type*, *type*), *bit shift*;
   - `^^`: *type*(*type*, *type*), *bit rotate*;
4. `*`: *type*(*type*, *type*), *multiplication*;
5. - `+`: *type*(*type*, *type*), *addition*;
   - `-`: *type*(*type*, *type*), *subtraction*;
6. `#`: *type*(*type*, *type*), *bitwise exclusive-or*;
7. `&`: *type*(*type*, *type*), *bitwise and*;
8. - `|`: *type*(*type*, *type*), *bitwise or*; and
   - `~`: *type*(*type*, *type*), *bitwise nor*;

where *type* is any of the above fixed-point types.

#### 2.1.3.2 Floating-Point Types

The floating-point types `d2` and `d4` are named in every module. Translators may name the additional floating-point types `d1`, `d8`, `d16`, `d32` and `dmax`. In particular, `d1`, if defined, is assigned a size and alignment of `1`.

The floating-point types' sizes are in a geometric progression of common ratio 2; their values and other properties are translator-defined.

The following operations accept and yield floating-point type values, ordered by decreasing precedence:

1. - `+`: *type*(*type*), *unary plus*;
   - `-`: *type*(*type*), *unary minus*;
2. `/`: *type*(*type*, *type*), *division*;
3. `*`: *type*(*type*, *type*), *multiplication*;
4. - `+`: *type*(*type*, *type*), *addition*; and
   - `-`: *type*(*type*, *type*), *subtraction*;

where *type* is any of the above floating-point types.

#### 2.1.3.3 Boolean Type

The boolean type `bool: {true, false}` is named in every Gamma module.

The following operations accept and/or yield boolean type values, ordered by decreasing precedence:

1. - `==`: `bool`(*typede*, *typede*), *equality*;
   - `><`: `bool`(*typede*, *typede*), *inequality*;
   - `<`: `bool`(*typepe*, *typepe*), *less than*;
   - `>`: `bool`(*typepe*, *typepe*), *greater than*;
   - `=<`: `bool`(*typepe*, *typepe*), *less than or equal*;
   - `>=`: `bool`(*typepe*, *typepe*), *greater than or equal*;
2. `!`: `bool(bool)`, *logical not*;

3. `##: bool(bool,bool)`, *logical exclusive-or*;
4. `&&: bool(bool,bool)`, *logical and*;
5. – `||: bool(bool,bool)`, *logical or*; and
   – `!: bool(bool,bool)`, *logical nor*;

where:

- $typede$ is any enumerated or derived type; and
- $typepe$ is any enumerated or pointer type.

An operation accepting boolean type values is `?: :` $type$(`bool`, $type$, $type$), the *ternary operator*.

# 2.2 Data

A module's data is accessed via *identifiers*, *symbols*, *datums* and *references*.

## 2.2.1 Identifier

An identifier is a sequence of names.

## 2.2.2 Symbols

A symbol is the association of an identifier and a type. Symbols which have the same identifier are called *candidates*, and are said to be *overloaded*.

If a symbol is declared twice in the same scope, the module in question is ill-defined.

## 2.2.3 Datums

A *datum* is an association of a symbol with a value. A datum has *global*, *unit*, *subprogram*, or *block* scope.

If a datum is defined twice in the same scope, the module in question is ill-defined.

### 2.2.3.1 Global

A datum has global scope if it is declared in a header unit.

### 2.2.3.2 Unit

A datum has unit scope if it has global scope or is defined outside of a subprogram.

### 2.2.3.3 Subprogram

A datum has subprogram scope if it has unit scope or is a subprogram parameter.

### 2.2.3.4 Block

A datum has block scope if it has subprogram scope or is defined in a block.

## 2.2.4 References

A *reference* is the association of an occurrence of a name with a datum. The datum's type and the type inferred at the occurrence match.

The process by which the reference is created is called *linkage*, and proceeds as follows:

1. If a WITH statement in the identifier's scope defines its first name, then said first name is replaced by its definition.
2. The identifier and required type are associated to create a *dummy symbol*.

3. If no candidates in the name’s scope match the dummy symbol, the module is ill-defined.
4. If a datum in the candidate’s scope is found with a matching name and type, the reference is created. Otherwise, the reference is recorded.

# 2.3 Code

A module's code describes changes in the module's data in a well-defined order.

All code in a module is contained in subprograms.

A subprogram is either a procedure or a function; procedures do not have a return type, whereas functions do.

## 2.3.1 Labels

Labels, like datums, have scope. In particular, labels always have block scope.

## 2.3.2 Statements

### 2.3.2.1 Simple Statements

#### 2.3.2.1.1 Assignment

In an assignment, the targets are writable, the source is readable, and the source's type matches the target's type.

### 2.3.2.2 Primitive Statements

#### 2.3.2.2.1 JUMP Statement

If a JUMP statement has no argument, it is in the block of an iterative statement, otherwise its argument is the name of a label in its scope.

#### 2.3.2.2.2 EXIT Statement

If an EXIT statement has no argument, it is in the block of an iterative statement, otherwise its argument is the name of the label of an iterative statement containing it.

#### 2.3.2.2.3 RETURN Statement

If a RETURN statement is in a procedure, then it has no argument, otherwise its argument is an expression whose type matches to the return type of the function containing it.

### 2.3.2.3 Composite Statements

Composite statements introduce a scope.

#### 2.3.2.3.1 Conditional Statements

A conditional statement evaluates expressions and uses the values to select bodies or *branches* to which to pass control.

**IF statements** execute branches *guarded* by expressions which equal true when the conditional executes. Each branch's guard is the expression which immediately precedes it syntactically.

**CASE statements** execute a branch guarded by a constant value which equals the expression passed as the statement's argument when it executes. Each branch's guard expression is the union of all of the constant values preceding it without an intervening instruction. No two guard expressions evaluate to the same value.

#### 2.3.2.3.2 FOR Statements

In a FOR statement, the datums defined in the initializations are scoped to, and the expression and simple statement are performed in, the scope of its block.

#### 2.3.2.3.3 DO Statements

In a DO statement, the control expression is evaluated in the scope of its block.

## 2.3.3 Definitions

A definition inside a subprogram body defines either data or a subprogram with a scope limited to that of the containing block and allocated with automatic lifetime. The first reference to any symbol in its own scope is a write.

# 3 Execution

Execution is the span of time during which the actions described by the module's code are performed on its data.

The actions actually performed during execution are called its *behavior*. The modifications of datums of global scope are called its *observable behavior*.

Behavior is either:

- *defined*, if it is described in this document;
- *translator-defined*, if it is described in a document associated with the translator;
- *unspecified*, if it is explicitly described as such or not predicted in this document; or
- *undefined*, if it is explicitly described as such or not described in this document.

# 3.1 States

During execution, a module's state consists of:

- the entities which are *allocated*, with their values; and
- the instruction being executed in that state.

## 3.1.1 Allocation

An allocation is an assignment of consecutive addresses in memory to a value. An allocation is designated by the lowest address reserved.

An allocation's *lifetime* is the portion of execution in which it may be referred to. An allocation has either *static*, *automatic*, *dynamic* or *temporary* lifetime.

### 3.1.1.1 Static

An allocation's lifetime is static if it is equivalent to the execution time of the module in which it is initialized.

### 3.1.1.2 Automatic

An allocation's lifetime is automatic if it is equivalent to the execution time of the block in which it is initialized.

### 3.1.1.3 Dynamic

An allocation's lifetime is dynamic if it does not exceed the lifetime of a pointer pointing to it.

### 3.1.1.4 Temporary

An allocation's lifetime is temporary if it is created to hold the result of an expression (see **§ 3.3**).

## 3.1.2 Initial

The *initial state* of a module occurs before all of its other states. Before the initial state:

1. all recorded references are created by creating datums for them (see **§ 2.2.4**);
2. all datums with unit scope are allocated statically.

The first state of the module determines whether it will behave as a *program* or as a *library*.

### 3.1.2.1 Programs

A module behaves as a program when its first state is specified. The first state is specified by:

- the initial values of datums with unit scope; and
- the subprogram executed first, called the *main subprogram*.

The means of specifying the main subprogram is translator-defined.

#### 3.1.2.2 Libraries

A module behaves as a library when the first state is not specified.

# 3.2 Instructions

When an instruction is *executed*, its behavior is performed.

## 3.2.1 Dynamic Definitions

### 3.2.1.1 Declarations

When a declaration is executed, the datums are created and allocated with automatic lifetime in order of occurrence. For each symbol, the first reference following the declaration is a write which gives the datum's initial value.

If a conditional contains the first writes to a symbol, it does so in all of its branches; otherwise, if the symbol is referred to after the conditional in question, the behavior is undefined.

### 3.2.1.2 Definitions

When a definition is executed, the datums are created and allocated with automatic lifetime in order of occurrence.

## 3.2.2 Statements

### 3.2.2.1 Simple

#### 3.2.2.1.1 Assignments

When an assignment is executed:

1. The right-hand and left-hand sides are evaluated.
2. If the assignment is compound, then the value of the datum referred to by the left-hand side is combined with the value of the right-hand side via the compound assignment's operation into a temporary that becomes the right-hand side.
3. The value of the allocation referred to by the left-hand side is set to the value of the allocation referred to by the right-hand side.
4. The instruction occurring after the assignment in question is executed.

#### 3.2.2.1.2 Procedure calls

Procedures are called as follows:

1. The arguments are evaluated.
2. The procedure referred to is determined.
3. The procedure's parameters are bound to the values of the arguments.

4. The first instruction in the procedure's body is executed.
5. The instruction occurring after the procedure call in question is executed.

## 3.2.2.2 Primitive

### 3.2.2.2.1 JUMP Statement

When a JUMP statement is executed, the behavior depends on the label specified.

- If the label is the empty string, then the containing iterative statement is re-executed.
- Otherwise, the statement with the label inside the containing subprogram is executed.

### 3.2.2.2.2 EXIT Statement

When an EXIT statement is executed, the behavior depends on the label specified.

- If the label is the empty string, then the statement following the containing iterative statement is executed.
- Otherwise, the statement following the containing iterative statement with the label is executed.

### 3.2.2.2.3 RETURN Statement

When a RETURN statement is executed, the body of the containing subprogram ends execution.

If the RETURN statement's argument evaluates to the address of a datum or subprogram of subprogram scope, the behavior is undefined.

## 3.2.2.3 Composite

When a composite statement is executed, it introduces a new scope.

### 3.2.2.3.1 Conditional Statements

When an IF statement is executed:

1. If the first guard is `true`, the branch associated with it is executed.
2. Step 1 is re-performed, with the next guard substituting the first guard.

When a CASE statement is executed:

1. The argument is evaluated.
2. The branch associated with the value to which the argument is equal is executed.

### 3.2.2.3.2 Iterative Statements

When a WHILE statement is executed:

1. If the guard is `false`, the statement following the WHILE statement in question is executed.
2. The block is executed.
3. Step 1 is re-performed.

When a DO statement is executed:

1. The block is executed.
2. If the guard is `true`, the statement following the DO statement in question is executed.
3. Step 1 is re-performed.

When a FOR statement is executed:

1. The initializations are executed.
2. If the guard is `false`, the statement following the FOR statement in question is executed.
3. The block is executed.
4. The update is executed.
5. Step 2 is re-performed.

## 3.3 Expressions

When an expression is evaluated, the expression's result is stored in a temporary allocation (see **§ 3.1.1**), and the allocations of its arguments are deallocated.

### 3.3.1 Constant

A constant is bound to a temporary allocation.

#### 3.3.1.1 Integer

The type of an integer constant is a translator-defined unsigned type.

### 3.3.2 References

When a reference is evaluated, the expression's allocation is the allocation of the datum that the reference refers to. The usage of the allocation referred to determines whether the reference is a *read*, *write*, or a *call*.

#### 3.3.2.1 Read

A reference is a read when the value of the expression is set to the value at the allocation referred to.

#### 3.3.2.2 Write

A reference is a write when the value at the allocation referred to is set to the value of the right-hand-side of an assignment.

#### 3.3.2.3 Call

A reference is a call when the address of the allocation referred to determines the subprogram to call.

### 3.3.3 Function calls

When a function call is evaluated:

1. The arguments are evaluated.
2. The function's parameters are bound to the values of the arguments.
3. Control is passed to the first instruction in the function's body.
4. Control returns at the first RETURN statement reached, with the statement's argument being the function's return value.
5. The result is the function's return value.

# 3.3.4 Operations

## 3.3.4.1 Derived Types

### 3.3.4.1.1 Pointer Types

Let $p$ be an expression of pointer type, and $e$ an expression whose allocation is not of temporary lifetime.

When `@`$p$ is evaluated, the result is the value in the allocation whose address is $p$.

When `.`$e$ is evaluated, the result is the address of the allocation of $e$.

### 3.3.4.1.2 Array Types

Let $a$ be an expression of type $type$`[]`, and $e$ an expression of type `nmax`.

When `#`$a$ is evaluated, the result is the number of allocations assigned to $a$ at the moment of evaluation.

When $a$ `+` $e$ is evaluated:

- if $e$ is less than `#`$a$:
  1. a temporary $a1$ of type `{ p: @`$type$` ; n: nmax }` has `p` initialized to the address of the first allocation of $a$;
  2. a temporary $a2$ of $a1$'s type has `n` initialized to $a1$`.`$n$ `+` $e$`*#::`$type$;
  3. the result is $a2$`.`$p$;
- otherwise, the behavior is undefined.

When $a$`[`$e$`]` is evaluated:

- if $e$ is less than `#`$a$, the result is `@(`$a$ `+` $e$`)`;
- otherwise, the behavior is undefined.

### 3.3.4.1.3 Signature Types

Let $f$ be an expression of type $return\_type$`(`$argument\_type$`)`.

When $f$`(`$args$`)` is evaluated, the result of the expression is the return value of the function call.

## 3.3.4.2 Basic Types

### 3.3.4.2.1 Aggregate Types

Let $e$ be an expression of aggregate type, $f$ an expression of pointer to aggregate type, and $i$ a name.

When $e$`.`$i$ is evaluated, the result is the value of the member of $e$ named $i$.

When $f$`@`$i$ is evaluated, the result is the value of the member of `@`$f$ named $i$.

## 3.3.4.3 Fixed-Point Types

Let `NMAX` be the largest value in `nmax`, `ZMAX` the largest value in `zmax`, and `ZMIN` the smallest value in `zmax`.

Let $e$ and $f$ be expressions evaluating to values of fixed-point type.

#### 3.3.4.3.1 Unary Plus

When $+e$ is evaluated, the result is $e$.

#### 3.3.4.3.2 Unary Minus

When $-e$ is evaluated:

- if $e$ is negative, or is positive and is less than the absolute value of `ZMIN`, then the result is the negative of $e$;
- otherwise, the behavior is undefined.

#### 3.3.4.3.3 Unary Bitwise Complement

When $\sim e$ is evaluated, each bit in the binary representation of the result is the complement of the corresponding bit in the binary representation of $e$.

#### 3.3.4.3.4 Integer Division

When $e/f$ is evaluated:

- if $f$ is nonzero, then the result is the truncated quotient of $e$ and $f$;
- otherwise, the behavior is undefined.

#### 3.3.4.3.5 Remainder

When $e\%f$ is evaluated:

- if $f$ is nonzero, then the result is the remainder of $e$ and $f$;
- otherwise, the behavior is undefined.

#### 3.3.4.3.6 Multiplication

When $e*f$ is evaluated:

- if the product of $e$ and $f$ is either less than `NMAX` or greater than `ZMIN`, then it is the result of the expression;
- otherwise, the behavior is undefined.

#### 3.3.4.3.7 Bit Shift

When $e**f$ is evaluated:

- if the absolute value of $f$ is smaller than the number of bits in the binary representation of the values of the type of $e$, then:
  - if $e$ is positive, then the result is the product of $e$ and $2^f$;
  - otherwise, the result is translator-defined;
- otherwise, the behavior is undefined.

#### 3.3.4.3.8 Bit Rotate

When $e\hat{}\hat{}f$ is evaluated:

- if $f$ is positive, the result corresponds to the cyclic permutation of the bits in the binary representation of $e$ by $f$ positions to the left;
- otherwise, the result corresponds to the cyclic permutation of the bits in the binary representation of $e$ by $f$ positions to the right.

#### 3.3.4.3.9 Addition

When `e + f` is evaluated:

- if the sum of $e$ and $f$ is less than `NMAX` or greater than `ZMIN`, then it is the result;
- otherwise, the behavior is undefined.

#### 3.3.4.3.10 Subtraction

When `e - f` is evaluated:

- if the difference between $e$ and $f$ is less than `NMAX` or greater than `ZMIN`, then it is the result;
- otherwise, the behavior is undefined.

#### 3.3.4.3.11 Bitwise Exclusive-Or

When `e # f` is evaluated, each bit in the binary representation of the result is the binary 'exclusive-or' of the corresponding bits of the binary representations of $e$ and $f$.

#### 3.3.4.3.12 Bitwise And

When `e & f` is evaluated, each bit in the binary representation of the result is the binary 'and' of the corresponding bits of the binary representations of $e$ and $f$.

#### 3.3.4.3.13 Bitwise Or

When `e | f` is evaluated, each bit in the binary representation of the result is the binary 'or' of the corresponding bits of the binary representations of $e$ and $f$.

#### 3.3.4.3.14 Bitwise Nor

When `e ~ f` is evaluated, each bit in the binary representation of the result is the binary 'nor' of the corresponding bits of the binary representations of $e$ and $f$.

### 3.3.4.4 Floating-Point Types

Let `DMAX` be the largest value in `dmax`, `DMIN` the smallest value in `dmax`, and `DMIN_U` the smallest unnormalized value in `dmax`.

Let $e$ and $f$ be expressions evaluating to values of floating-point type.

#### 3.3.4.4.1 Division

When `e/f` is evaluated:

- if $f$ is nonzero, the result is the truncated quotient by long division of $e$ and $f$;
- otherwise, the behavior is undefined.

#### 3.3.4.4.2 Multiplication

When $e$`*`$f$ is evaluated:

- if the product of $e$ and $f$ is less than `DMAX` or is greater than `DMIN_U`, it is the result;
- otherwise, the behavior is undefined.

#### 3.3.4.4.3 Addition

When $e$ `+` $f$ is evaluated:

- if the sum of $e$ and $f$ is less than `DMAX` or is greater than `DMIN_U`, it is the result;
- otherwise, the behavior is undefined.

#### 3.3.4.4.4 Subtraction

When $e$ `-` $f$ is evaluated:

- if the difference between $e$ and $f$ is less than `DMAX` or is greater than `DMIN_U`, it is the result;
- otherwise, the behavior is undefined.

### 3.3.4.5 Boolean Type

Let:

- $x$ and $y$ be expressions evaluating to boolean type,
- $e$ and $f$ be expressions evaluating to any enumerated type,
- $u$ and $v$ be expressions evaluating to any array type,
- $p$ and $q$ be expressions evaluating to any pointer type.

#### 3.3.4.5.1 Equality

When $e$ `==` $f$ is evaluated, the result is `true` if and only if $e$ and $f$ have the same value.

When $u$ `==` $v$ is evaluated, the result is `true` if and only if:

- $u$ and $v$ have the same address,
- $u$ and $v$ have the same length, and
- $u$`[0]` and $v$`[0]` are of the same type.

When $p$ `==` $q$ is evaluated, the result is `true` if and only if $p$ and $q$ have the same value.

#### 3.3.4.5.2 Inequality

When $e$ `><` $f$ is evaluated, the result is `false` if and only if $e$ and $f$ have the same value.

When $u$ `><` $v$ is evaluated, the result is `false` if and only if:

- $u$ and $v$ have the same address,
- $u$ and $v$ have the same length, and
- $u$`[0]` and $v$`[0]` are of the same type.

When $p$ `><` $q$ is evaluated, the result is `false` if and only if $p$ and $q$ have the same value.

#### 3.3.4.5.3 Less Than

When $e$ `<` $f$ is evaluated, the result is `true` if and only if $e$ precedes $f$.

When $p$ `<` $q$ is evaluated:

- if $p$ and $q$ are addresses in the allocation of the same aggregate or array:
  - the result is `true` if and only if $p$ precedes $q$;
- otherwise, the behavior is undefined.

#### 3.3.4.5.4 Greater Than

When $e$ `>` $f$ is evaluated, the result is `true` if and only if $e$ succeeds $f$.

When $p$ `>` $q$ is evaluated:

- if $p$ and $q$ are addresses in the allocation of the same aggregate or array:
  - the result is `true` if and only if $p$ succeeds $q$;
- otherwise, the behavior is undefined.

#### 3.3.4.5.5 Less Than or Equal

When $e$ `=<` $f$ is evaluated, the result is `true` if and only if $e$ precedes or is equal to $f$.

When $p$ `=<` $q$ is evaluated:

- if $p$ and $q$ are addresses in the allocation of the same aggregate or array:
  - the result is `true` if and only if $p$ precedes or is equal to $q$;
- otherwise, the behavior is undefined.

#### 3.3.4.5.6 Greater Than or Equal

When $e$ `>=` $f$ is evaluated, the result is `true` if and only if $e$ succeeds or is equal to $f$.

When $p$ `>=` $q$ is evaluated:

- if $p$ and $q$ are addresses in the allocation of the same aggregate or array:
  - the result is `true` if and only if $p$ succeeds or is equal to $q$;
- otherwise, the behavior is undefined.

#### 3.3.4.5.7 Logical Not

When `!`$x$ is evaluated, the result is `false` if and only if $x$ is `true`.

#### 3.3.4.5.8 Logical Exclusive-Or

When $x$ `##` $y$ is evaluated, the result is the logical exclusive-or of $x$ and $y$.

#### 3.3.4.5.9 Logical And

When $x$ `&&` $y$ is evaluated:

1. $x$ is evaluated.
   If $x$ is `false`, the result is `false`;
2. otherwise $y$ is evaluated and is the result.

#### 3.3.4.5.10 Logical Or

When $x$ `||` $y$ is evaluated:

1. $x$ is evaluated.
   If $x$ is `true`, the result is `true`;
2. otherwise $y$ is evaluated and is the result.

#### 3.3.4.5.11 Logical Nor

When $x$ `!` $y$ is evaluated:

1. $x$ is evaluated.
   If $x$ is `true`, the result is `false`;
2. otherwise `!`$y$ is evaluated and is the result.

### 3.3.4.6 Ternary Operator

Let $p$ be an expression of boolean type, $e$ and $f$ expressions of compatible types.

When $p$`?`$e$`:`$f$ is evaluated:

1. $p$ is evaluated.
2. – If $p$ is `true`, then $e$ is evaluated and is the result;
   – otherwise $f$ is evaluated and is the result.

# A Grammar

```
module               = {declaration}
                     | {definition}
declaration          = space name {declaration} end
                     | sym symbol[{, symbol}]
                     | type symbol[{, symbol}]

symbol               = label type
type                 = [{prefix_type}]term_type
term_type            = (name|user_type)qualifier[{postfix_type}]
                     | (prefix_type type)[{postfix_type}]
user_type            = {(name{, name}|symbol({, symbol}|{; symbol}))}
prefix_type          = @qualifier
postfix_type         = (([symbol[{, symbol}]])|[[expression]])qualifier
qualifier            = [!|?]

definition           = type symbol[{, symbol}]
                     | data datum[{, datum}]
                     | code symbol
                                instructions
                       end
datum                = name = expression
rename               = name = identifier
instructions         = {dynamic_definition|[label] statement}
dynamic_definition   = sym symbol[{, symbol}]
                     | with rename[{, rename}]
                     | data datum[{, datum}]
                     | code symbol
                                instructions
                       end
statement            = simple_statement | primitive_statement | composite_statement
simple_statement     = expression_unary [(assignment|= [{expression_unary =}]) expression]
assignment           = (/|%|^^|**|+|-|#|&|~||)=
primitive_statement  = jump ([name])
                     | exit ([name])
                     | return ([expression])
composite_statement  = if expression do
                                instructions
                       [{elif expression do
                                instructions}]
                       [else
                                instructions]
                       end
                     | case expression
                       {{is expression} do
                                instructions}
                       [else
```

```
                                    instructions]
                              end
                            | while expression do instructions end
                            | do instructions until expression end
                            | for [datum[{, datum}]];expression;simple_statement do instructions end
expression                  = expression_ternary [:: type]
expression_ternary          = expression_lnor [{? expression_lnor : expression_lnor}]
expression_lnor             = expression_land [{(|||!) expression_land}]
expression_land             = expression_lxor [{&& expression_lxor}]
expression_lxor             = expression_comp [{## expression_comp}]
expression_comp             = expression_bnor [(==|><|<|>|=<|>=) expression_bnor]
expression_bnor             = expression_band [{(||~) expression_band}]
expression_band             = expression_bxor [{& expression_bxor}]
expression_bxor             = expression_add [{# expression_add}]
expression_add              = expression_mul [{(+|-) expression_mul}]
expression_mul              = expression_shi [{* expression_shi}]
expression_shi              = expression_div [{(**|^^) expression_div}]
expression_div              = expression_term [{(/|%) expression_term}]
expression_term             = [{prefix_operator}] term
prefix_operator             = @|.|#|+|-|~|!
term                        = (identifier|(expression))[{term_operator}]
                            | literal
term_operator               = [expression]
                            | ([expression[{,expression}]])
                            | (.|@)name
literal                     = number|string|array|aggregate
array                       = [expression[{, expression}]]
aggregate                   = {expression[{, expression}]}
```