Gamma Programming Language Specification

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";
- $\mathfrak{a}$ represents an occurrence in Gamma source code of a token in the class named "a";
- `{a}` represents zero 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".

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 Unicode :
  - the decimal digits `0123456789`, named $digit$,
    and the nonzero digits $ndigit = digit - \mathtt{0}$;
  - 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 `type data code if elif case is else do while for jump exit return end`;
- The operators `#:: %:: . @ / % ^^ ** * + - # & | ~ == =< >= < > >< ## && || ! ?: ::`;
- The delimiters `() [] {}`;
- Identifiers, defined by the regular expression
  $(\_|letter)(\_|alpha)^*$
- Numbers, defined by the regular expressions
  $ndigit(digit^+(.digit^*ndigit)^?|(.digit^*ndigit)^?(\mathtt{e-}^?ndigit\,digit^*)^?)$
  $\mathtt{0}(\mathtt{b}\,bdigit^+|\mathtt{o}\,odigit^+|\mathtt{x}\,hdigit^+)$
- Strings, defined by the regular expression
  $\mathtt{"}(\mathtt{"}^-|\backslash(\mathtt{"}|\backslash))^*\mathtt{"}$
- Labels, defined by the regular expression
  $identifier\mathtt{:}$

1.2 Translation Units

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

1.2.1 Header Units

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

1.2.1.1 Types

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

A type is defined with the following syntax:

```
type identifier :: type{, identifier :: type}
```

1.2.1.2 Declarations

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

Symbols are declared with the following syntaxes:

```
data identifier :: type{, identifier :: type}
code identifier :: signature_type
```

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 (se § ??).

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 identifier = expression{, identifier = expression}
```

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 identifier([identifier :: type{, identifier :: type}])[ :: type]
      instructions
end
```

A subprogram has a *body* composed of *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 is either an *assignment* or a *procedure call*.

An assignment changes the values of the module's data. An assignment has the following form:
`expression_unary (assignment|= {expression_unary =}) expression`
where `assignment` may be any one of the following *compound assignments*:

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

A procedure call executes a procedure's body, binding the values of the arguments to the parameters.

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 ([identifier])`

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 ([identifier])`

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])`

Composite Statements

Composite statements contain other instructions in a *block*, introducing a scope. A composite statement is either a *conditional statement* or an *iterative statement*.

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

IF statements execute branches *guarded* by expressions which equal true when the conditional executes. IF statements have the following form:

```
if (expression)
       instructions
{elif (expression)
       instructions}
[else (expression)
       instructions]
```

```
end
```

CASE statements execute a branch guarded by an expression which equals the expression passed as the statement's argument when it executes. CASE statements have the following form:

```
case (expression)
is (expression) {is (expression)}
        instructions
{is (expression) {is (expression)}
        instructions}
[else (expression)
        instructions]
end
```

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 statments have the following form:

```
while (expression) instructions end
```

DO statments have the following form:

```
do instructions while (expression) end
```

FOR statements have the following form:

```
for (initializations;expression;simple_statement) instructions end
```

1.2.2.2.2 Definitions

A definition inside a subprogram body defines either data or a subprogram with a scope and extent limited to that of the containing subprogram. A definition is either *scoped* or *unscoped*.

Scoped definitions restrict the scope of the datums or subprograms that they define to a block of a containing composite statement.

For data,

```
data ([identifier]) identifier = expression{, identifier = expression}
```

For subprograms,

```
code ([identifier]) identifier ([identifier :: type{, identifier :: type}])[ :: type]
        instructions
end
```

Unscoped definitions restrict the scope of the datums or subprograms that they define to their containing subprogram.

For data,

```
data identifier = expression{, identifier = expression}
```

For subprograms,

```
code identifier ([identifier :: type{, identifier :: type}])[ :: type]
        instructions
end
```

A finite, non-empty group of translation units describes a *module*. A module is an independent association of *data* and *code*.

Translators accept only *well-defined* modules. A module is well-defined only if the statements not qualifying it as otherwise in this document hold.

2.1 Data

A module's data and code are stored in its *memory*.

2.1.1 Data Model

A module's memory is modeled as a succession of *cells*, each of which is a *string* of *bits*. All cells are of a finite, fixed, translator-defined size. Each cell is assigned a unique natural integer called an *address*.

2.1.2 Allocation

An allocation is the association of a range of addresses in memory with a value of a type. An allocation is designated by the lowest address in the range.

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

2.1.2.1 Static

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

2.1.2.2 Automatic

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

2.1.2.3 Dynamic

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

2.1.2.4 Temporary

An allocation's extent is temporary if it is created to hold the result of an operation.

2.1.3 Symbols

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

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

2.1.4 Datums

A *datum* is an association of a symbol with a initial 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.

2.1.4.1 Global

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

2.1.4.2 Unit

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

2.1.4.3 Subprogram

A datum has subprogram scope if it has unit scope, is a subprogram parameter or is defined in an unscoped definition (see **§ 2.3.2.2**).

2.1.4.4 Block

A datum has block scope if it has subprogram scope or is defined in a scoped definition (see **§ 2.3.2.1**).

2.1.5 References

A *reference* is the association of an occurrence of an identifier with a datum having a value whose type is compatible with the type required by the location where the identifier occurs.

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

1. The identifier and required type are associated to create a *dummy symbol*.
2. The declared candidates at the identifier's occurrence are searched.
   If no candidates match the dummy symbol, the module is ill-defined.
3. The datums in the scope of the identifier's occurrence are searched.
   If a datum with a symbol matching the dummy symbol's type is found, the next step is skipped.
4. If the current scope is the global scope, the module is ill-defined.
   Otherwise, Step 3 is re-performed with the immediately enclosing scope.
5. The reference is associated with the datum.

2.2 Types

A type, for the purposes of this specification, is an association of a set of *values* and *operations* over them.

2.2.1 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*.

2.2.1.1 Qualification

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

2.2.1.2 Size

The size of a type is the number of consecutive addresses assigned to a value thereof. The size of a type `type` is written `#::type`.

2.2.1.3 Alignment

The alignment of a type is the smallest difference between two different addresses assignable to a value thereof. The alignment of a type `type` is written `%::type`.

2.2.2 Operations

An operation is an association of values of one or more argument types with values of a result type.

2.2.3 Varieties

Types are either *basic* or *derived*.

2.2.3.1 Basic

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

2.2.3.1.1 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:

```
{ identifier{, identifier} }
```

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

2.2.3.1.2 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:
`{ identifier :: type{, identifier :: type} }`
Each member of a record type has its own allocation, and allocations are in the order of appearance in the source code. Every record type is complete: for any record type *type*, the alignment of *type* is the maximum of the alignments of its members' types.

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:
`{ identifier :: type{; identifier :: type} }`
Each member of a union type shares a single allocation. Every union type is complete: for any union type *type*, the alignment of *type* is the maximum of the alignments of its members' types.

Operations on aggregate types are:

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

2.2.3.2 Derived

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

2.2.3.2.1 Array

An array type's values associate finite numbers of allocations of values 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 on values of array type are:

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

2.2.3.2.2 Signature

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

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

The only operation on values of signature type is `()`, the *function call*.

2.2.3.2.3 Pointer

A pointer type's values store the address of a datum. 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*]`(`[*type*{`,` *type*}]`)`,
`(@`[*return_type*]`)(`[*type*{`,` *type*}]`)`

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 on pointer types, for every complete type *type*, is `@ ::` *type*`(@`*type*`)`, the *pointer dereference*.

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

2.2.4 Provided

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

2.2.4.1 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 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*), *bitwise linear shift*;
   - `^^ ::` *type*(*type*, *type*), *bitwise circular shift*;
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.

2.2.4.2 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` is defined to have 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 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.

2.2.4.3 Boolean

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

The following operations 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 over boolean type values, but not associated with them, is `?:` :: $type$`(bool,`$type$`,`$type$`)`, the *ternary operator*.

2.3 Code

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 Statements

2.3.1.1 Labels

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

2.3.1.1.1 Assignment

An assignment changes the values of the module's data.

If:

- either side of the assignment is not of a type compatible with the other,
- the left-hand side is not writable, or
- the right-hand side is not readable

the module is ill-defined.

2.3.1.2 Primitive Statements

2.3.1.2.1 JUMP Statement

A JUMP statement passes control to the containing iterative statement or the statement labeled by the argument. If a JUMP statement not contained within an iterative statement has an empty argument, the module is ill-defined.

2.3.1.2.2 EXIT Statement

An EXIT statement passes control to the statement following the containing iterative statement or a containing iterative statement labeled by the argument. If an EXIT statement is not contained within an iterative statement or has an argument which refers to a composite statement not containing it, the module is ill-defined.

2.3.1.2.3 RETURN Statement

A RETURN statement passes control to the subprogram that called the subprogram containing it, at the point at which the containing subprogram was called. If a function contains a RETURN statement with no argument or a procedure contains a RETURN statement with an argument, the module is ill-defined, no diagnostic required.

2.3.1.3 Composite Statements

Composite statements introduce a scope.

2.3.1.3.1 Conditional Statements

A conditional statement evaluates expressions and uses the values to select a block or *branch* out of several 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. If a CASE statement has two guard expressions which evaluate to the same constant value, the module is ill-defined.

2.3.1.3.2 FOR Statements

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

2.3.2 Definitions

A definition inside a subprogram body defines either data or a subprogram with a scope limited to that of the containing subprogram and allocated with automatic extent.

2.3.2.1 Scoped

Scoped definitions (see **§ 1.2.2.2.2**) restrict the scope of the datums or subprograms that they define to a block of a containing composite statement.

If a scoped definition occurs outside of a composite statement or refers to a label that does not label a containing composite statement, the module is ill-defined.

2.3.2.2 Unscoped

Unscoped definitions (see **§ 1.2.2.2.2**) restrict the scope of the datums or subprograms that they define to their containing subprogram.

The successive states of a module as expressed by the values of its data at particular moment during execution are called its *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 not predicted in this document; or
- *undefined*, if it is not described in this document.

3.1 States

During execution, a module's state or context consists of:

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

3.1.1 Initial

The first state of the module is the *initial state*. Before the initial state, datums are *allocated*.

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

3.1.1.1 Programs

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

- the initial values of datums of unit and global scope; and
- the subprogram containing the first statement to be executed, called the *main subprogram*.

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

3.1.1.2 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 Definitions

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

If a conditional does not contain definitions of a particular datum in all of its branches, behavior is undefined if the datums are referred to in statements occurring after the conditional in question in the same scope.

If two branches of a conditional containing definitions of a particular datum are executed, behavior is undefined if the datums are referred to in statements occurring after the conditional in question in the same scope.

3.2.2 Statements

3.2.2.1 Simple

3.2.2.1.1 Assignments

When an assignment is executed:

1. The right-hand side is evaluated.
2. The left-hand side is evaluated.
3. 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.
4. 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.
5. 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 of pointer or enumerated type or to a datum 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. The first guard is evaluated.
   If it evaluates to 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. The guard is evaluated.
   If it evaluates to false, the statement following the WHILE statement in question is executed.
2. The body is executed.
3. Step 1 is re-performed.

When a DO statement is executed:

1. The body is executed.
2. The guard is evaluated.
   If it evaluates to false, 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. The guard is evaluated.
   If it evaluates to false, the statement following the FOR statement in question is executed.
3. The body is executed.
4. The update is executed.
5. Step 2 is re-performed.

3.3 Expressions

When an expression is evaluated, the expression's value is stored in a temporary allocation (see **§ 2.1.2**), and the allocations of its arguments are deallocated.

3.3.1 Literals

A literal value is bound to a temporary allocation.

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 extent.

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 value 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` is initialized to the address of the first allocation of a converted to `nmax`.
  2. The value is $a1 + e*\#::type ::@type$.
- otherwise, the behavior is undefined.

When $a[e]$ is evaluated:

- If e is less than $\#a$:
  1. A temporary $a1$ is initialized to $a + e$.
  2. The value is $@(a1::@type)$.
- 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 value of the expression is the return value of the function call;

3.3.4.1.4 Pointer Types

Let p be an expression of type $@type$.

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

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 an identifier.

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

When $f@i$ is evaluated, the value 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 value 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 value 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 value 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 value 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 value 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 value of the expression;
- otherwise, the behavior is undefined.

3.3.4.3.7 Bitwise Linear 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 value is the product of e and $2^{\mathtt{f}}$;
  - otherwise, the value is translator-defined;
- otherwise, the behavior is undefined.

3.3.4.3.8 Bitwise Circular Shift

When e^^f is evaluated, the value corresponds to the cyclic permutation of the bits in the binary representation of e by f positions.

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 value;
- 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 value;
- 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 value 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 value 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 value 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 value 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 value 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 value;
- 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 value;
- 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 value;
- 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 e and f have the same value, and `false` otherwise.

When u == v is evaluated, the result is `true` 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,

and `false` otherwise.

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

3.3.4.5.2 Inequality

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

When u >< v is evaluated, the result is `false` 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,

and `true` otherwise.

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

3.3.4.5.3 Less Than

When e < f is evaluated, the result is `true` if e precedes f, and `false` otherwise.

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 p precedes q, and `false` if not;
- otherwise, the behavior is undefined.

3.3.4.5.4 Greater Than

When e > f is evaluated, the result is `true` if e succeeds f, and `false` otherwise.

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 p succeeds q, and *false* if not;
- otherwise, the behavior is undefined.

3.3.4.5.5 Less Than or Equal

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

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 p precedes or is equal to q, and `false` if not;
- otherwise, the behavior is undefined.

3.3.4.5.6 Greater Than or Equal

When e =< f is evaluated, the result is `true` if e succeeds or is equal to f, and `false` otherwise.

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 p succeeds or is equal to q, and `false` if not;
- otherwise, the behavior is undefined.

3.3.4.5.7 Logical Not

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

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 {declaration}
                     | definition {definition}
declaration          = data identifier :: type{, identifier :: type}
                     | code identifier :: type
                     | type identifier :: type{, identifier :: type}
type                 = identifier qualifier
                     | {identifier :: type(, identifier :: type{, identifier :: type}|;)}
                     | {prefix_type}(prefix_type{prefix_type}type)postfix_type
prefix_type          = @qualifier
postfix_type         = (signature_type|array_type)qualifier
qualifier            = [!|?]

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

```
                    | while (expression) instructions end
                    | do instructions while (expression) end
                    | for (initializations;expression;simple_statement) instructions end
expression          = 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_unary {(/|%) expression_unary}
expression_unary    = {prefix_operator}term
prefix_operator     = @|.|#|+|-|~|!
term                = (identifier|(expression)){term_operator}
                    | literal
term_operator       = [expression]
                    | ([expression{,expression}])
                    | (.|@)term
literal             = number|string|array|aggregate
array               = [expression{, expression}]
aggregate           = {expression{, expression}}
```