S I M U L E T T A

Language Definition

by

Øystein Myhre Andersen

This document is reconstructed from various sources, mainly from the archive after Simula as.
It is manually checked against a printed version and updated regarding discrepancies found in the source code
.

Oslo 15. may 2022
Øystein Myhre Andersen

Table of contents

1. Introduction

The Simuletta language is a special language developed for the production of a portable Simula system. It is used to compile the Simula runtime system to S-Code.

The S-Code definition has had great influence on the design of the Simuletta language. In addition to ordinary value and reference variables, so-called name and field variables are defined in Simuletta. These correspond directly to the concept of general and attribute addresses in S-Code.

Reference variables as well as field and name variables may be fully or partially qualified by specification. As in Simula a qualification may be extended either implicitly by an assignment or explicitly. But unlike Simula there is no runtime checking on the legality of such extension.

Unlike most assembly languages, Simuletta contains a number of structured statements such as if-statement and case-statement.

Data to be manipulated may be of one of the usual types such as integer, or the information may be structured as a record with named components, or even into higher order structures such as linked lists or networks. Operations (such as +) are included for the massage of primitive data, while pointer and structure manipulating instructions (such as the dot notation) are included to support general graph traversion. Data structures and new types are defined as in S-Code using the record concept. Types which are defined this way are specified by infix(record'identifier).

The main control structuring tool is the routine concept. A routine is inherently non-recursive, all parameter transmissions are by value (but it is possible to transfer pointer values). Unlike other languages, Simuletta permits explicit naming of the location, in which the return address is saved, so that a routine not necessarily returns to its point-of-call. Routines also establish name hiding: all named (identifiers) defined within the routine are invisible from the outside, they lose their meaning when the routine is left.

A module defines a closed name scope (just like routines). It is, however, possible to selectively open the scope, making certains aspects of a module accessible outside the module, while other aspects remain hidden. Type definitions, routine identifications, labels, and named constants can be made visible in this manner. The inteface module serves as a global area for variables and constants.

1.1 References.

1. "Portable Simula Revisited"
2. "Simula Standard"
3. "S-PORT, Definition of S-Code"
4. "S-PORT, The Environment Interface"

1.2. **Terminology.**

Atomic unit

A data storage unit. The size is the highest common factor of the sizes of all the data quantities which will be manipulated during the execution of a program. The size is implementation dependent. Atomic units may impose a finer resolution on the storage than the machine allows.

Area

A vector of one or more consecutive atomic units.

Object unit

An area of implementation-defined fixed size; the size will always be an integral number of machine addressable storage cells. This is the allocation unit (storage cells may not be directly usable because of alignment problems).

Quantity

Used with the meaning: something that (at run time) may be manipulated by the executing program.

Record

An area with a structure imposed by a structured type defined by a record descriptor.

Object

A record which is not a component of any record. An object will always comprise an integral number of object units.

Static quantity

The quantity exists throughout the program execution.

Dynamic quantity

The quantity is created during program execution.

Segment

A contiguous storage area containing machine instructions.

Current program point

The place which will contain the next target machine instructions

generated is called the current program point.

Constant area

A storage area used for the allocation of constants. Dependent

upon the architecture of the target machine constants may be

allocated interspersed with instructions (i.e. in program

segments) , in a separate storage or elswhere.

1.3. **Language definition.**

The Simuletta language is defined by a two-level syntax. This is done to reflect the common implementation technique – using a separate scanning preprocessor cooperating with the parsing of program text. The first level of syntax defines a grammer for the scanner, while the second level is the real language syntax.

The syntax will be described with the aid of metalinguistic formulae. The notation used is BNF (as in Simula Common Base Language) with the following extensions:

- Meta symbols are written in lowe case without brackets,
  terminal symbols are underlined lower case.

- Alternative right hand sides for a production may be separated by ::= as well as |.

- Productions may be annotated with comments. Whenever the symbol string:
  -- < any printable character >*
  occures it should be ignored until the end of the line, i.e. It is equivalent to the emty string.

- Part of a right hand side may be enclosed in angular brackets followed by one of the characters ?, *, or + with the following meanings:

  < symbol string >? -- "symbol string" is optional, it may occure zero times or once.

  < symbol string >* -- "symbol string" may occure zero or more times at this point.

  < symbol string >+ -- "symbol string" must occure one or more times at this point.

  Alternatives for the symbol within angular brackets may be separated
  by | , e.g. < letter | digit >+

- Spaces and line breaks are used simply to separate the symbols of a
  production, they haveno other significance.

- Particular instances of a meta-symbol may be given a prefix separated
  from the symbol by an apostrophy, e.g. Record'identifier.

  The prefix (record) has the sole purpose of identifying the meta-symbol (identifier)
  in the accompanying description, it has no syntactic significance whatsoever.

2. **The Basic Syntax**

A Simuletta program text must be represented according to ISO standard, ISO 646-1973, and consists of a sequence of line records. If a line record consists of more than 72 positions, an implementation may treat as significant only the first 72 positions. There are two kinds of line records; Compiler directives and Program lines.

2.1. **Compiler directive lines.**

Compiler directives are identified by having a % character (ISO code 37) in the first position of the line record. Such lines are always taken as compiler control lines and the interpretation of their content is implementation dependent.

2.2. **Program lines.**

Line records which do not have a % character in the first position are taken as Program lines.

The program text consists of a sequence of symbols, comments and spaces. For the purpose of including explanatory texts among the symbols, the following comment convention holds:

> Whenever the symbol string:
>
> -- <any printable character>*
>
> occurs in a program source text it should be ignored until the end of line,
> i.e. It is equivalent to the empty string of symbols.
>
> Whenever the symbol ; occures in a program source text it is ignored.
> It is only used in the source text for documentation purposes.

The basic symbols which constitutes the program text are:

- Identifiers
- Keywords
- Mnemonics
- Simple Values
- Special symbols

A basic symbol must be contained in a single line record, i.e. It cannot be continued from one line to the next. The extent of a symbol is decided by a left-to-right scan of a Program line beginning at position 1 or at the first position containing a non-space character following an already recognized basic symbol, trying to recognize the largest possible string of characters which fits the syntax of a basic symbol.

In the remaining part of this document, the words "program text" mean the sequence of basic symbols obtained by scanning the Program lines excluding the Compiler directives and comments.

2.3. **Basic Symbols.**

```
basic_symbol
      ::= identifier  |  keyword  |  mnemonic  |  simple_value  | special_symbol

identifier
      ::=  letter  <  letter  |   digit  |  _  >*

mnemonic
      ::=  letter  <  letter  |   digit  |  _  >*

keyword
      ::= AND  |  BEGIN  |  BODY  |  BOOLEAN  |  CALL  |  CHARACTER  |  CONST
        |  DEFINE  |  DO  |  ELSE  |  ELSIF  |  END  |  ENDCASE  |  ENDIF
        |  ENDMACRO  |  ENDREPEAT  |  ENTRY  |  EXIT  |  EXPORT  |  EXTERNAL
        |  FALSE  |  FIELD  |  GLOBAL  |  GOTO  |  IF  |  IMPORT  |  INFIX
        |  INFO  |  INSERT  |  INTEGER  |  KNOWN  |  LABEL  |  LONG  |  REM
        |  MACRO  |  MODULE  |  NAME  |  NOBODY  |  NOFIELD  |  NONAME  |  NONE
        |  NOSIZE  |  NOT  |  NOWHERE  |  OR  |  OTHERWISE  |  PROFILE  |  QUA
        |  RANGE  |  REAL  |  RECORD  |  REPEAT  |  REF  |  ROUTINE
        |  SYSINSERT  |  SHORT  |  SIZE  |  SYSROUTINE  |  SYSTEM  |  THEN
        |  TRUE  |  VAR  |  VARIANT  |  VISIBLE  |  WHEN  |  WHILE  |  XOR

simple_value
      ::= integer_number
      ::= real_number
      ::= long_rel_number
      ::= character_value
      ::= string_value

integer_number
      ::= < digit >+
      ::= mnemonic

real_number
      ::= < digit >*  .  < digit >+
      ::= < digit >*  <   .  < digit >+  >?  &  < + | - >?   < digit >+

long_real_number
      ::= < digit >*  .  < digit >+  &&  < + | - >?   < digit >+
      ::= < digit >*  <   .  < digit >+  >?  &&  < + | - >?   < digit >+

character_value
      := <  '  < any ISO_character except  '  >+  '  >+

string_value
      ::= <  "  < any ISO_character except  "  >+  "  >+

ISO_character
      ::= ISO_code
      ::= any printable character from the ISO alphabet.

ISO_code
      ::=  !  < digit >+  !
```

```
special_symbol
      ::=  +  |  -  |  *  |  /   |  %
      ::=  <  | <=  |  =  |  >=  |  >  |  <>
      ::=  ,  |  .  |  :  |  :=  |  (  |  )

letter
      ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z|
          a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z

digit
      ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Letters do not have individual meanings, they are used for forming other items such as identifiers, string values etc. Digits are used for forming numbers, identifiers and string values. A special symbol has a fixed meaning which for the most part is obvious or else will be given at the appropriate place in the sequel. Note that spaces are not permitted inside identifiers, keywords or mnmonics.

For keywords, only the upper case versions are given – all combinations of upper and lower case letters that spell a word given above are also considered reserved words.

2.3.1 **Identifiers.**

A symbol is recognized as an identifier if it belongs to the syntax class identifier and it is neither a keyword nor defined as a mnemonic.

The length of an identifier is restricted to at most 72 characters. The case (upper or lower) of any character is not significant.

The identifiers may be chosen freely, they have no inherent meaning, but serve for the identification of language quantities e.g. Simple variables, labels, records, etc.

The same identifier cannot be used to denote two different quantities except when these quantities have disjointed scopes (the scope of a quantity is the set of statements and expressions in which the declaration of the identifier associated with that quantity is valid).

2.3.2 **Keywords.**

A symbol is recognized as a keyword if it belongs to the syntax class keyword defined above. These are reserved and cannot be used as identifiers or mnemonics. In the remaining part of this document keywords are written in underlined lower case.

2.3.3 **Mnemonics.**

A symbol is recognized as a mnemonic if it belongs to the syntax class mnemonic, and a mnemonic definition containing this symbol has already been encountered, and it is not a keyword. These mnemonics become reserved and cannot be used as identifiers. After a mnemonic definition is encountered during the scan of the basic program text, every occurrence of a particular mnemonic signal a substitution in the input stream. Literal'mnemonics are substituted while Macros are expanded.

Note that mnemonics are not scope restricted within the current unit of compilation, and automatically made visible through module inclusion.

```
mnmonic_definition
      ::= macro_definition
      ::= literal_definition

literal_definition
      ::= define  literal'mnemonic  =  < literal_value >+
            <  ,  literal'mnemonic  =  < literal_value >+  >*

literal_value  ::=  basic_symbol
```

```
Example: (Literal Mnemonics)

      define  A=1,B=2,C=3,D=4,E=5,F=6

Thus:                  range(A:F) X=E;
is changed to:         range(1:6) X=5;
during the scan.
```

Macro Mnemonics.

```
macro_definition
      ::=  macro  macro'mnemonic  (  parcount'integer_number  )
           begin  < macro_element >*  endmacro

macro_element
      ::=  any basic_symbol except  endmacro  and  %
```

The macro definition determines the substitution schema for the macro expansion processing (see below). The macro body will not de analysed when the definition is processed, and nested definitions are illegal. The parcount is the number of actual parameters given when the macro is called, i.e. zero means no parameters, one means exactly one parameter in the call etc.

Macro expansion.

After a macro mnemonic definition is encountered during the scan of the basic program text, every occurrence of that particular mnemonic signals a macro expansion.

```
macro_expansion
      ::=  macro'mnemonic  (  < macro_parameter_list >?  )

macro_parameter_list
      ::=  macro_parameter  <  ,  macro_parameter >*

macro_parameter
      ::=  basic_symbol
      ::=  %  < basic_symbol >*  %
```

The macro expansion should take place by textual replacement in the input stream. When a macro expansion is recognized the complete state of the Scanner is saved and the Scanner enters macro expansion mode. The corresponding macro definition is identified, and the macro call is replaced "textually" after the following rules:

- All macro body elements are processed in the sequence they occur in the definition.
- If the element is a sequence of basic symbols, these symbols are inserted.
- If the element is a parameter number, the content of the corresponding actual parameter is inserted in the same manner.

When the replacement defined above has taken place, the Scanner leaves expansion mode, restores the state saved and continues processing with the first inserted symbol.

Example: (Macro Mnemonics)

```
macro ALLOC(2);
begin %1 := nxt; nxt := nxt + %2;
      if nxt > lim
      then GARB(%2);
           %1 := nxt; nxt := nxt + %2;
      endif;
endmacro
```

Thus:

```
ALLOC(%x.y%,%size(array)+pp.lng%)
```

is replaced by:

```
x.y := nxt; nxt := nxt + size(array)+pp.lng;
if nxt > lim
then GARB(size(array)+pp.lng);
     x.y := nxt; nxt := nxt + size(array)+pp.lng;
endif;
```

during the scan.

2.3.4. **Numbers.**

Numbers are represented in decimal notation. Integers and decimal numbers has their conventional meaning.

Real numbers may have, and long real numbers always have, scale factors announced by single or duble ampersand (&). Scale factors are interpreted as integral power of 10.

E.g. 3&6 stands for 3 times 10 raised to the 6^{th} power, i.e. 3 millions
while &2 stands for 10 raised to the 2^{th} power, i.e. 100

Examples:

Integers:

```
0               200             083
177             743             7
```

Reals

```
.0              200.84          .083&-02
177.            07.43&8         &7
.5384           9.34&+10        3&-4
0.7300          2&-4            &+5
```

Long reals:

```
.5384&&1        9.34&&+10       &&-4
0.730&&6        2&&-4           16&&+5
```

2.3.5 **Character and String Values.**

256 different characters are defined (corresponding to an 8-bit representation). The lower half of this ordered set is the ISO 646 character set (encoded accordingly), the interpretation of the upper half is implementation-dependent.

Within a character (or string) value, any printing character except the character (or string) quote represents itself. In order to include the complete ISO alphabet any character may be represented inside a string value (or a character value) by its ISO code (decimal notation) and surrounded by code quotes (! - ISO code 33). The ISO code cannot consist of more than three digits and must have a legal value, i.e. <= 255. If these conditions are not satisfied the construction is interpreted as a character sequence. The string quote may be represented as ISO code 34, and the character quote by ISO code 39.

Examples:

Character value:	represents
`'A' '!3458!'`	`A!3458!`
`'!' '2!'`	`!2!`
`'!2!'`	`The STX character`

String value:	represents
`"AB"  "CDE"`	`ABCDE`
`"!2!ABCDE!2!"`	`ABCDE  enclosed in STX and ETX characters`
`"AB"` `"CDE"`	`"ABCDE"`
`"!" "33" "!"`	`!33!`

3. General Structure of Simuletta-Programs

```
simuletta_program
      ::= interface_module
      ::= sub_module

interface_module
      ::= global module'identifier
          begin < < visible >?  decl_in_interface >*  end

decl_in_interface
      ::= mnmonic_definition
      ::= global_declaration
      ::= constant_declaration
      ::= record_declaration
      ::= routine_declaration

sub_module
      ::= module module'identifier
          begin < < visible >?  decl_in_module >*
                < statement >*  end

decl_in_module
      ::= module_inclusion
      ::= mnmonic_definition
      ::= local_declaration
      ::= constant_declaration
      ::= record_declaration
      ::= routine_declaration

main_program
      ::= begin  < decl_in_module >*  < statement >* end
```

A system programmed in Simuletta will in the general case consist of a main program, which receive control when the translated system is to begin execution, and several modules which may provide type definitions, routine support (in the form of a runtime system, as is the case in portable Simula) etc. One particular module, the interface module, is used to define the global data structure, basic type definitions and parts of the environment interface routines. The module'identifier is a system-unique identification for the module. It is used in module inclusions.

Each program or module will contain three main classes of program elements:

- type definitions (**record** descriptors) govern the structure and manipulation of data quantities.

- data quantity declarations (**const**, **local**) control the actual (static) allocation and identification of data.

- statements and instructions, possibly grouped in routines, specify (together with the type information) the target machine instructions to be generated.

Rather than compiling a complete program, the Simuletta Compiler can be instructed to compile a part of a progran (module) which subsequently will be referenced by another module or a main program.

The set of identifiers specified in the **visible** defines the externally visible attributes of the module. Candidates are all global variables, constants, records, profiles, bodies and routines, but not variables local to a module. Only program elements specifically mentioned to be **visible** are transferred via the module inclusion given in another compilation unit; the program elements defined through an module inclusion cannot themselves be made visible outside the module being compiled.

The interface module serves two main puposes:

- it specifies a set of statically allocated variables (the globals)
- it specifies parts of the interface to the system environment.

An interface module is included as any other module with the restriction, that only one particular interface module can ever occur in the executing program, i.e. It must be checked that different modules do not include different versions of the interface module.

A complete listing of the Simula interface module may be found in (4).

3.1 Global variable declarations.

```
global_declaration
      ::= type  globalid  <  ,  globalid  >*

globalid
      ::= identifier  < system identifier >?  < = value >?
      ::= identifier  < system identifier >? ( repeat'integer_number )
            < = ( value < , value >* ) >?
```

Global variable declarations may only occur in the interface module. The global declaration specifies static allocation of a variable of the given type. The type will define the internal structure of the quantity as well as the set of operations permitted. Each such variable is an object, and may thus be addressed either by an object reference or a general reference.

Repetitions

A global variable may be defined as a repetition. If the repeat'number is greater than zero a vector of identical elements is defined, containing that number of elements and accessed through indexing with indices starting at zero. The allocation of the elements is done in such a way that the size of each element provides enough information to permit access to one relative to another. A repeat number less than one is illegal.

It is important to realise that the repetition concept does not impose any structure upon the quantity (as e.g. array declaration does in Simula). Whenever a repeated quantity is selected (by an idetifier) the first element is selected directly.

Initialisation.

A global variable may be initialised to a given value. This initialisation will take place before execution of the program proper take place. If a global variable is not initialisated, the initial value is undefined. Globaly declared variables are not initialized unless initial values are given along with declarations.

Examples:

```
integer a,b(4),c(10);   range(0:1000) k,l=34; -- k is undefined
                                              -- l is initialized to 34
real q(2)=(3.14,2.83);  ref() x =system CURDRV;

ref(object) d(8);       infix(complex) z;

field(integer) i,j;     name(ref(object)) dp;
```

3.2 **Local variable declarations.**

```
local_declaration
      ::= type localid  <  ,  localid  >*

localid
      ::= identifier  < ( repeat'integer_number ) >?
```

Local variable declarations may occur in all modules except the interface module. They may also occur in the main program. The local declaration specifies static allocation of a variable of the given type. The type will define the internal structure of the quantity as well as the set of operations permitted. Each such variable is an object, and may thus be addressed either by an object reference or a general reference.

Local variables have a restricted scope, they are only visible from the inside of the module where they are declared, and they cannot br specified visible in the visible list. Local variable may not be initialized. However, since they are statically allocated, they do not lose their values when the module (or main program) is left. Local variables are similar to **own** variables i Algol.

A local variable may be defined as a repetition.

Examples:

```
integer a,b(4),c(10);           range(0;1200) k,l;

real q(2)                       ref() x;

ref(object) d(8);               infix(complex) z;

field(integer) j;               name(ref(object)) dp;
```

3.3 **Constant declarations.**

```
constant_declaration
      ::= const  type  constantid  <  ,  constantid  >*

constantid
      ::= identifier = value
      ::= identifier ( repeat'integer_number ) = ( value < , value >* )
```

Constants may be declared in all modules and in the main program. A constant area is created to hold the specified value. The type will define the internal structure of the quantity as well as the set of operations permitted. Each constant is an object, and may thus be addressed either by an object reference or a general reference.

It must be possible to evaluate all values occurring within a named constant at compile-time.

Examples:

```
const integer a=13,b(4)=(1,2,3,4)

const range(0:255) k=4;

const real q(2)=(3.14,2.83)

const ref(object) d(3)=(ref(a),ref(b),none)

const infix(complex) z=record:complex(re=0.0,im=1.0)

const field(integer) i=field(R.k)

const name(ref(object)) dp=name(x.a(3))
```

3.4 **Inclusion of a module.**

```
module_inclusion
      ::= insert     module_ident  <  ,  module_ident  >*
      ::= sysinsert  module_ident  <  ,  module_ident  >*

module_ident
      ::= module'identifier  <  =  external_id'string_value  >?
```

This instruction causes the Simuletta Compiler to include a module. The external_id is used to identify the module with respect to an operating system. If no external_id is given, the Simuletta Compiler will search the module definition library for a module identified by the module'identifier. If it cannot be found, it is an error.

sysinsert is used for system modules (Simula rts, Simob, etc.), while **insert** is used for user defined modules.

The **visible** elements of the module (as specified in the visible list) are now brought into the current compilation unit.

3.5 **Linkage of modules.**

The linkage of the executable code will generally be carried out in a manner standard to the target system. Some knowledge about the modules must, however, be commucated to the Simuletta Compiler: identification and type binding etc. of names external to the program being compiled. Such information is procured from a data base (the module definition library) maintained by the Simuletta Compiler itself (see appendix D.3). The naming conventions used, the structure of and access methods to this data base are highly target system dependent.

4. **Record Definition.**

```
record_declaration
      ::= record  record'identifier  <  :  prefix'identifier  >?
            <  record_info  >?
          begin  common_part  < variant_part >*  end

record_info
      ::= info "DYNAMIC"
      ::= info "TYPE"

common_part
      ::=  <  type  attrid  <  ,  attrid  >*  >*

variant_part
      ::= variant  <  type  attrid  <  ,  attrid  >*  >*

attrid
      ::= attribute'identifier  <  (  repeat'integer_number  )  >?
```

The record'identifier become associated with the record and serves as identification of the record withih its scope. Records are used to define a structure upon a class of objects or as generators for structured type. Records are also used as building boxes for other record definitions through prefixing. A record consists of attributes, grouped together into a commen part possibly followed by variants.

Examples:

```
record complex; begin real re,im end

record R;
begin integer a,b(4); character c(80);
      variant infix(complex) z(3);
      variant real x,y;
end

record S : R;
begin variant label x;
      variant boolean b; ref() y(0);
end
```

Note that objects of record S may be interpreted in 4 different ways depending upon which pair of variants is selected.

4.1 **Prefixing**

If specified, the prefix must refer a record without an indefinite repetition. The prefix record may, however, contain variants. A record without prefix is said to be defined on prefix level zero, while a record prefixed by R is said to be defined on prefix level one higher than the prefix level of R.

4.2 **Attributes.**

Each attribute definition defines the type of an accessible atribute of the record. There is no inherent correspondence between the order of the attributes in the record descriptor and the allocation order inside a record. However, when the record descriptor is processed by the Simuletta Compiler, an attribute reference value will be associated with each attribute identifier defined. These associations cannot later be changed, which implies that the order prefix ... common part must be preserved.

The attribute'identifier become associated with the attribute. The scope of these attrubute identifiers is the record definition itself. They are only accessible through the so-called 'dot-access' (see 6.1).

An attribute may contain several components which are all of the same specified type. A nonzero repeat'number defines the number of components. If no repeat'number is given, one is assumed. A repeat'number of zero indicastes that the number of components is indefinite. An indefinite repetition in the record descripto must be the (lexically) very last attribute defined, i.e. it occurs immediately before **end**.

4.3 **Record information.**

The record_info string is used to give information to the Simuletta Compiler on the use of certain records. Three classes of records are distinguished, any record descriptor will one of these classes:

- All records used to form structured types which are used in expressions shall contain the record **info** "TYPE".

- A small set of records are used as prefixes to every dynamic quantity created by the run time system during execution; such types shall contain **info** "DYNAMIC". This information may be necessary in order to determine the dynamic size, since the target system may prohibit general use of the address space for such objects, e.g. dynamic reference should be an even byte address.

- If a record descriptor does not contain (either directly or through prefix) any such specification, the record wil not be used for any of the two mentioned purposes.

It may not be necessary for the Simuletta Compiler to utilise the record information fiels; in that case the specifications may be ignored and no distinction should be made by the compiler.

4.4 **Records with variants.**

If several variant parts are given, it specifies alternative interpretations of the record. One particular interpretation is obtained by taking all the common parts together with one selected variant part on each prefix level. Thus the number of alternative interpretations of a record is the product of the number of varianta parts on each prefix level (counting one if there is no variant part on a particular prefix level). The correspondence between pairs of attributes from different variant parts on the same prefix level is not defined. Records are always allocated with a size corresponding to the largest alternative (at allocation an indefinite repetition cannot occur).

4.5 **Allocation order.**

The Simuletta Compiler is free to reorder or pack the attributes in any convenient way, as long as the above mentioned restrictions are observed.

In summary:

- the order prefix...common part...alternatives must be preserved,
- once ordered and packed the order is invariant, which means that a prefix cannot be re-packed,
- an indefinite repetition must be allocated at the end of the structure,
- records are always allocated with a size corresponding to the largest alternative (at allocation an indefinite repetition cannot occur).

5. **Types and Values.**

```
type
   ::= integer
   ::= short integer
   ::= range  (  lower'integer_number  :  upper'integer_number  )
   ::= real
   ::= long real
   ::= size
   ::= Boolean
   ::= character
   ::= label
   ::= entry  (  < profile'identifier >?  )
   ::= structured_type
   ::= object_reference
   ::= attribute_reference
   ::= general_reference

structured_type
   ::= infix  (  record'identifier  <  :  fixrep'integer_number  >?  )

object_reference
   ::= ref  (  < record'identifier >?  )

attribute_reference
   ::= field  (  < qualifying'type >?  )

general_reference
   ::= name  (  < qualifying'type >?  )

value
   ::= integer_number
   ::= real_number
   ::= long_real_number
   ::= Boolean_value
   ::= character_value
   ::= string_value
   ::= label_value
   ::= routine_value
   ::= structured_value
   ::= object_reference_value
   ::= attribute_reference_value
   ::= general_reference_value
```

The various types besically denote properties of values. Any data quantity must belong to some type. The type will define the internal structure of the quantity as well as operations that may be performed upon it. Types are used as generators in global, constant, local or parameter definitions and as specifiers in declarations.

The distinction between resolved and non-resolved type is made because of the indefinite repetition, which may occur in structured types. Such a type cannot be used as a generator, or in further type definition, without determining the actual number of elements in the repetition.

Whenever the Simuletta Compiler should perform type checking, neither the actual number of elements in such an indefinite repetition, nor the actual range specified for an **integer** quantity is of any significance, unless it is explicitly indicated in the text. A structured type can be extended by using prefixing. Such types are not type compatible.

The reference concept corresponds to the intuitive notion of a "name" or a "pointer". It also reflects the addressing capability of computers: in certain simple cases a reference could be implemented as the memory address of a stored value.

The type of the value of a particular variable is defined by the declaration of the variable.

Examples:

```
integer                     range(0:10000)
real                        ref()
ref(object)                 infix(complex)
entry()                     infix(item:14)
field(integer)              name(ref(object))
```

5.1 **Integer types.**

The type declarations **integer**, **short integer** and **range** serves to declare identifiers representing variables capable of containing a subset of the integer values with a machine dependent range and representation. The full integer range must correspond at least to a 16-bit representation, but if possible it should be at least 32 bits.

Quantities of type **integer** may be restricted in range. A **range** has a contiguous value domain which is part of the domain for **integer**. The motivation for **range** is storage economy for variables with a resticted value domain.

The Simuletta Compiler allocates a **range** to a storage unit which at least comprises the closed interval [lower...upper] specified for the **range**. The value domain of a **range** is system dependent and defined to be the domain of the storage unit allocated to the **range**.

The short integer correspond to a range specified with a suitable interval, at least including [-32000,32000], and (almost) symetrical around zero.

Although ranges occur in arithmetic expressions, there is no arithmetic operations (or relations) defined on ranges. It is the responsibility of the Simuletta Compiler to convert to **integer** before the operation is performed, and to convert to **range** before assignment to a range. One consequence of this is that intermediate results never can be range restricted.

It is required that the Simuletta Compiler checks domain overflow for assignments to ranges originating from **short integer**s, while such a check is not required for other ranges. The Simuletta Compiler may choose to always generate domain checks on assignment, hence treating any range as it must treat **short integer**.

Integer numbers are of type **integer**.

5.2 **Reals.**

The type declaration real serves to declare identifiers representing variables capable of containing a subset of the real values, which is representable on the target machine. All real numbers are of type **real**.

5.3 **Long reals.**

The type declaration **long real** serves to declare identifiers representing variables capable of containing a subset of the **real** values with greater precision than the real values. The range of **long real** may differ from that of **real**. Restricted implementations may choose to ignore the distinction between real and long real, treating both as real. All long real numbers are of type **long real**.

5.4 **Booleans.**

The type declaration **Boolean** serves to declare identifiers representing variables capable of containing the values **true** or **false**.

```
Boolean_value
      ::= true
      ::= false
```

5.5 **Object size.**

The type declaration **size** serves to declare identifiers representing variables capable of containing object size values. Values of this type describe object sizes and distances between objects, measured as the distance between two machine addresses. They may be represented as integers (with sign), but they are not compatible with **integer**. The empty size (correspomding to zero) is **nosize**.

The size of a record wil be the size of the prefix plus the size of the common part plus the size of the "largest" variant part, that is the alternative occupying the largest number of object units. If the record contains an indefinite repetition the size is measured as if this attribute is absent.

```
size_value
      ::= nosize
      ::= size  (  record'identifier  )
      ::= size  (  record'identifier  :  integer'expression  )
```

The last form is used for records containing an indefinite repetition. The result is the size of the record obtained by using the value of the integer'expression insted of the indefinite repetition. The record referenced must contain an **info** "DYNAMIC", and it must contain an indefinite repetition, otherwise: error.

5.6 **Characters**.

The type declaration **character** serves to declare identifiers representing variables capable of containing the character values (see sect. 2.3.5).

5.7 **Structure types.**

The type declaration **infix**(R) serves to declare identifiers representing variables capable of containing structured values.

Any defined record may be used as a pattern of a structured type. If the record contains an idefinite repetition, this repetition must be resolved through the use of the fixrep construction, thereby determining the actual number of elements in the repetition.

For every record R without an indefinite repetition there is associated a structured type **infix**(R).

For every record I which contain an indefinite repetition there is associated a set of structured types **infix**(I:n) n=1,2, ... The value 'n' resolves the indefinite repetition in the record I.

Subordinate types.

A structured type S2 is said to be "subordinate" to a second structured type S1 if and only if one of the following situations applies:

- S1 is generated by the record R1 without **fixrep**,
  S2 is generated by the record R2 (with or without **fixrep**)
  and R2 have R1 in its prefix chain.

- S1 is generated by the record R with **fixrep** n2,
  S2 is generated by the same record with **fixrep** n1
  and n1 >= n2

Informally speaking, S2 is subordinate to S1 if S2 is an extension of S1.

Strctured Values.

```
Structured_value
   ::= record  :  record'identifier  (  attrvalue < ,  attrvalue >*  )

attrvalue
   ::= attribute'identifier  =  value
   ::= attribute'identifier  =  (  value  <  ,  value  >*  )
```

The attribute'identifier defines which attribute is to be of the associated value, i.e. The sequence need not be the same in the structured type and in the record value. Strict type correspondence is required between an attribute and the associated value.

All attribute identifiers of the record must occur at most once, but it is legal not to associate to some attributes (they are associated default values according to the table in sect. 9.1).

An indefinite repetition is resolved by the number of values in the corresponding attribute value list. The order of the values correspond to the indexing order, i.e. Attribute(0) takes the first value in the list, etc.

If the record contains alternative parts, a specific alternativ is selected by naming one of its attributes. Once an alternativ has been selected, it is an error if an attribute from any other alternative occur. If no alternative is selected then no alternative part is produced.

Examples:

```
record:complex ( im = 0.137 , re = -2&-2 )

record:template
        ( x = ( record:complex ( im = 1.0 , re = 0.0 ) ,
                record:complex ( im = 0.0 , re = 1.0 ) ) ,
          y = 1327 , a = ( name(b), name(b) , noname ) )
```

5.8 **Strings**

It is adopted as a convention that the interface module should contain the record definition:

```
record string; info "TYPE";
begin name(character) chradr; integer nchr end;
```

The type infix(string) then becomes available in every Simuletta programs.

String values.

String values are introduced by means of a syntactical transformation of the source text:

> Let S be an occurrence of a string value in the source text, and let S consists of the characters C1, C2, ... ,Cn. The occurence of the string S is replaced by:
>
> ```
> record:string(chradr=name(<id>),nchr=<n>)
> ```
>
> and a new constant is added among the declarations in the program:
>
> ```
> const character <id>=(C1,C2, ... ,Cn)
> ```
>
> where <id> is an identifier not occuring elsewere in the text and <n> is the number of characters in S.

5.9 **Data reference types.**

Addresses in Simuletta are designed to permit the description of objects which are arranged in implementation and machine-dependent ways. In particular it is envisaged that some Simuletta Compilers may pack information into available storage in ways which require to be descibed using complex addresses. These considerations have led to the following types of address:

5.9.1 **Object references.**

Associated with an object there is a unique "object reference" which identifies the object. And for any record R there is an associated reference type **ref**(R). A quantity of that type is said to be qualified by the racord R. Its value is either an object address, or the special value **none** which represents "no object address". The qualification restricts the range of values to objects records included in the qualifying record. The range of values includes the value **none** regardless of the qualification.

An object reference is said to be "subordinate" to a second object reference if the qualification of the former is a subrecord of the record which qualifies the later.

The type **ref**() is an unqualified object reference. All object references are subordinate to the unqualified one.

Object Reference Values

```
Object_reference_value
      ::= none
      ::= ref  (  object'identifier  )
```

The value is the object reference of the global or constant quantity identified. **none** refers to no object unit. The type of the value is **ref**() i.e. an unqualified object reference.

5.9.2 **Attribute references.**

Associated with an attribute of an object there is a unique "attribute reference" which identifies that attribute relatively to the object. An attribute reference correspond to the intuitive notion of a reletive pointer (offset) to an attribute within an object.

For any type T there is associated an "attribute reference" type **field**(T). A **field**(T) quantity is said to be qualified by the type T. Its value is either an attribute reference, or the special value **nofield** which represents "no attribute address". The qualification restricts the range of values to attribute of that specific type or a subordinate type in the case of object references. The range of values includes the value **nofield** regardless of the qualification. The type **field**() is unqualified, thus the range of values is not restricted in this case.

An attribute reference is said to be "subordinate" to a second attribute reference if the qualification of the former is subordinate to the qualification of the later. All attribute references are subordinate to the unqualified one.

Attribute Reference Values

```
attribute_reference_value
      ::= nofield
      ::= field  (  record'identifier  <  .  identifier  >+  )
```

The value is the attribute reference.
The type of the value is **field**(type of attribute).

5.9.3 **General references.**

Associated with a variable there is a unique "general reference" which identifies that variable. A general reference correspond to the intuitive notion of a direct pointer to a variable. A general reference carry necessary information to make it it possible to convert it into a pair:

general reference --> [object reference , attribute reference]

For any type T there is associated a "general reference" type **name**(T).

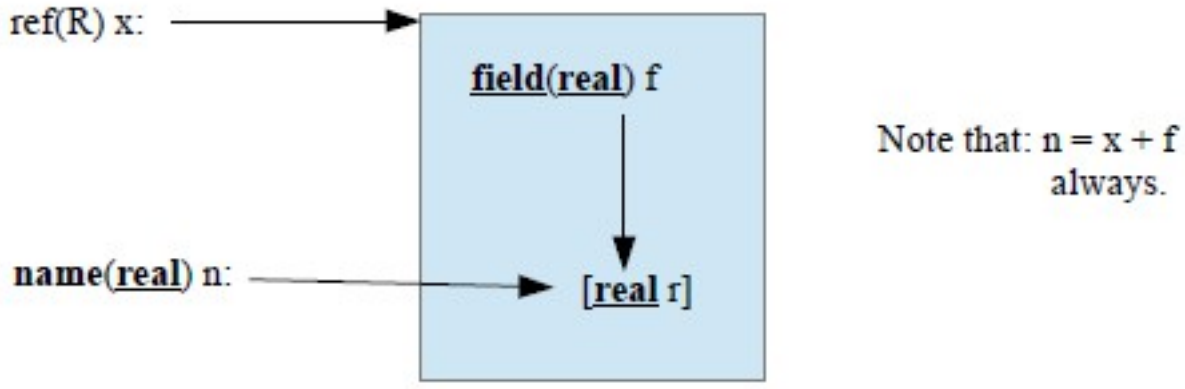
A **name**(T) quantity is said to be qualified by the type T. Its value is either a general address, or the special value **noname** which represent "no general address". The qualification restricts the range of values to variables of that specific type or a subordinate type in the case of object references. The range of values includes the value **noname** regardless of the qualification. The type **name**() is unqualified, thus the range of values is not restricted in this case.

An general reference is said to be "subordinate" to a second general reference if the qualification of the former is subordinate to the qualification of the later. All general references are subordinate to the unqualified one.

General Reference Values

general_reference_value
::= **noname**
::= **name** (variable)

The value is the general address of the variable.
The type of the value is **name**(type of the variable).



Correspondence between the different data references.

5.10 **Instruction address types.**

The type declaration **label** serves to declare identifiers representing variables capable of containing label values. Labels carry the address of a program point. They are independent of the other types of address.

nowhere designates no program point.

The type declaration **entry** serves to declare identifiers representing variables capable of containing entry point values. An entry point value carry the address of a routine body, a peculiar routine cannot occur. They are independent of (and inconvertible to) the other address types.

nobody designates no entry point.

```
label_value
        ::= nowhere
        ::= label'identifier

entry_value
        ::= nobody
        ::= entry  (  routine'identifier  )
```

Labels defined withih a routine body cannot be used to form label values.

6. **Expression.**

```
expression
      ::= factor
      ::= unary_operator  factor
      ::= expression  binary_operator  factor
      ::= if Boolean'expression then expression else expression
      ::= type_conversion

factor
      ::= value
      ::= variable
      ::= routine_activation
      ::= (  expression  )

unary_operator
      ::=  + | - | not

binary_operator
      ::=  + | - | * | / | rem
      ::=  and | or | xor
      ::=  <> | < | <= | = | >= | >
```

Every expression is of a certain type. The type of a factor is the type of the value, variable, etc. In binary operations the type of the result is derived from the actual operation and the type of the operands. Both brances of an conditional expression must be of the same type, which become the type of the expression.

NOTE: The normal operator precedence rules do not hold.
Any expression is evaluated stricly from left to right, i.e.:

```
a*b+c   =>   (a*b)+c          i.e.  get a, get b, do *, get c, do +
a+b*c   =>   (a+b)*c          i.e.  get a, get b, do +, get c, do *
```

Note: Evaluation of an expression can result in an interrupt situation at run time. Handling of such situations are treated in more detail in the document "The Environment Interface" (4).

Examples:

```
x                                 x*y
34                                a/(a+1)
sqrt(name(x)+field(f)))           (x>=4) and (y+z=0)
not ((a*b+c) = d)                 x = 1.32&-2
-x                                (a+(b+c)) > (d+e)
a*b + c                           if a>0 then b*(c+d) else e
```

6.1 **Variable.**

```
variable
      ::= simple_variable  <  (  index'expression  )  >?

simple_variable
      ::= identifier
      ::= var  (  general_reference'expression  )
      ::= object_reference'expression  .  identifier
      ::= structured_type'expression  .  identifier
```

Examples:

```
a                 a(i+4) qua R.x(P(i+j)*2)
f(a,b,c)          var(n+o)
```

Any variable may be supplied with an index expression of type **integer**. In this case the designated quantity is treated as a repetition. Let V be a simple_variable, the the first element of a repetition may be accessed by V or V(0), the second by V(1) and so on. There is no index checking so the legality decision is left to the user.

Dot access.

The 'dot access' is used in two different ways, to remotely access object attributes, or to select attributes within strucktured types.

Let X be an expression of type ref(R), then the remote identifier X.A is valid if A is an attribute of R. If the value of X is **none**, then the remote access is undefined, i.e. It is illegal but there is no runtime checking on it. The use of **qua** enables qualification extension or restriction for remote access. The phrase 'X **qua** Q' means 'change the qualification of X to Q in this context. The remote identifier X **qua** Q.A is valid if A is an attribute of Q and either R is subordinate to Q or Q is subordinate to R.

Let Y be an expresion of type **infix**(S), then the selected identifier Y.B is valid if B is an attribute of S.

The type of a remote or selected identifier is the type specified for the attribute.

Dynamic Access.

Let V be an expression of type **name**(T), then the construction **var**(V) is used to access the content of the variable referenced by V. The type of **var**(V) is T. To cover the case of unqualified references, the optional qualification setting is introduced, thus the type of **var**(V **qua** U) is U.

Exstreme care should be executed when using this access method because no validity checking is performed. For example, the assignment:

```
var( name(i) qua real ) := 13.4;
```

where 'i' is declared **integer**, makes it possible to assign a **real** value into an **integer** variable without conversion. The meaning of this is highly implementation dependent.

Dynamic Quantities.

The addressing of dynamic quantities poses a problem as their descriptors are incomplete. Dynamic quantities are continually being created and destroyed during program execution, and the Simuletta Compiler is not in control of their creation and allocation in storage; this task is the responsibility of the user.

In order to complete the definition of descriptors of dynamic objects the Simuletta Compiler must provide a mechanism for associating object references (to be generated at run time) with natural numbers or object indices known at compile time. The scheme adopted should give complete freedom til the Simuletta Compiler in choosing an appropriate implementation strategy; this can have a considerable effect on run-time performance.

The following convention is adopted:

> Every Simuletta program acts as if the following global variable was defined:
>
> ```
> ref() display(M)
> ```
>
> where M is an implementation defined integer value greater than 7.

The well-known implementation technique using a "display vector" being continually updated during program execution is, for instance, catered for in this proposal.

Accessing attributes of dynamic objects is carried out by using remote identifiers, e.g.

```
display(4) qua entity.pp
```

6.2 **Arithmetic Operators** (+ - * / **rem**).

The binary operators +, -, *, and / are defined for **integer**, **real** and **long real** operands. Both operands must be of the same type, which becomes type of the result. The binary operator **rem** is only defined for **integer** operands. Mixed type arithmetic expressions i.e. Expressions with a mixture of **integer**, **real** and **long real** operands, are illegal. All arithmetic on subranges of **integer** should be performed in full integer arithmetic.

Reminder is defined as: `x rem y == x - (x/y)*y`

6.3 **Boolean Operators** (**and**, **or**, **xor**, **not**)

The logical operators **and**, **or**, **xor** and **not** are only defined for **Boolean** operands, always giving a result of type **Boolean**. There is no guarantee that all expressions involved are evaluated.

6.4 **Relational Operators** (< , <= , = , >= , > , <>)

The operators <= , >= and <> stands for less than or equal, greater than or equal, and unequal respectively.

Both operands must be of the same type. The result obtained by applying one of the relational operators is always of tye **Boolean**. The operators = and <> are defined for all types, while the operators < , <= , >= and > are defined for all arithmetic types as well as characters and object references.

Comparison between character values is done according to the ISO 646 code (i.e. The corresponding integer values are compared). Assuming an integer representation of **size** values, comparison is performed by comparint the numerical values of this representation. Reference values are compared by comparing the corresponding machine addresses (regarded as ordinal numbers). Comparison between quantities of structured types is performed component by component.

	<	<=	=	>=	>	<>
Boolean			+			+
character	+	+	+	+	+	+
integer	+	+	+	+	+	+
real	+	+	+	+	+	+
long real	+	+	+	+	+	+
size	+	+	+	+	+	+
ref	+	+	+	+	+	+
field			+			+
name			+			+
label			+			+
entry			+			+
infix			+			+

Table of legal relational operations

(+ marks valid relation for the designated type)

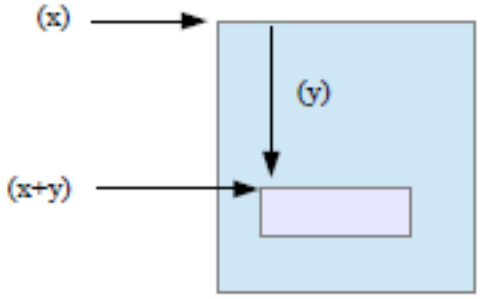
6.5 **General Reference Expression.**

The addition operation is also defined for certain reference types:

```
ref(..)   +  field(T)    producing a result of type name(T)
name(..)  +  field(T)    producing a result of type name(T)
```

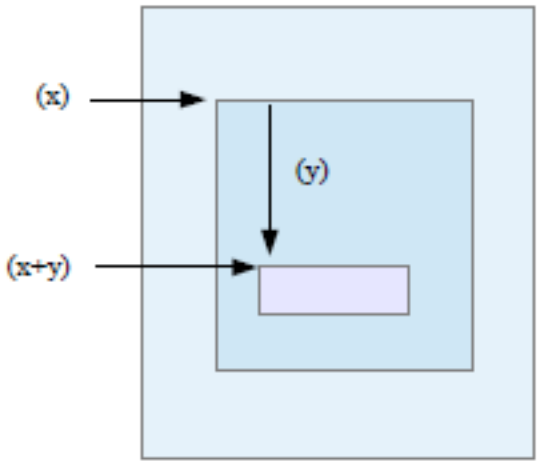
Case 1. `ref(..) + field(T) => name(T)`

Let x be an expression of type **ref**(R), and let y be an expression of type **field**(T), then the expression 'x+y' is defined to be the general reference of type **name**(T) which identifies the attribute referenced by y in the object referenced by x.



Case 2. `name(infix(..)) + field(T) => name(T)`

Let x be an expression of type **name**(**infix**(S)), and let y be an expression of type **field**(T), then the expression 'x+y' is defined to be the general reference of type **name**(T) which identifies the attribute referenced by y in the attribute referenced by x.



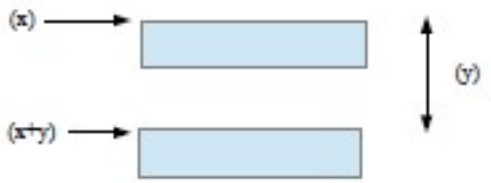
6.6 **Object Reference and Size Expression.**

The addition and subtraction operations are also defined for:

```
ref(..)  +  size     producing a result of type  ref()
ref(..)  -  size     producing a result of type  ref()
ref(..)  -  ref(..)  producing a result of type  size
```

Case 1. **ref**(..) + **size** => **ref**()

Let x be an expression of type ref(R), and let y be an expression of type size, then the expression 'x+y' is defined to be the unqualified object reference which is allocated y object units after the start of x.

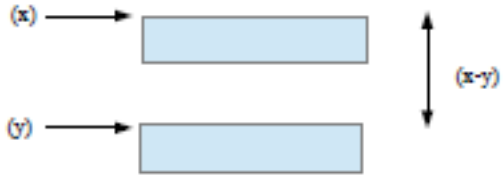


Case 2. **ref**(..) - **size** => **ref**()

Let x be an expression of type ref(R), and let y be an expression of type size, then the expression 'x-y' is defined to be the unqualified object reference which is allocated y object units before the start of x.

Case 3. **ref**(..) - **ref**(..) => **size**

Let x and y be object reference expressions of type ref(R), then the expression 'x-y' is defined to be the distance from x to y.
The type of the result is **size**.



6.7 **Type Conversion.**

```
type_conversion
      ::=  expression  qua  type
```

The value of the expression is converted to the specified type. Not all conversions are valid, see the table below. An attempt to perform an invalid conversion is an error

The conversion performed will in some cases be illegal because of the actual value; one example would be to try to convert a **real** to an **integer**, if the actual value of the **real** is outside the range of **integer**. Such errors should be checked for at run time, in cases where they can occur. These conversions are marked ? in the table below.

Conversion from **name** to **ref** means; take the object reference part of the general reference and return as result. The type of the result is **ref**() .

Conversion from **name** to **field** means; take the attribute reference part of the general reference and return as result. The type of the result is **field**().

An object reference may be converted to a general reference. In that case the object reference is extended with an enpty attribute reference and the pair comprises the result. The type of the result is **name**(), i.e. an unqualified general reference.

real (**long real**) to **integer** conversion is performed after the rule: I = entier (R + 0.5)
(Entier: the gratest integer not grater than the argument)

from: \ to:	**character**	**integer**	**real**	**long real**	**field**	**ref**	**name**
character		+					
integer	?		?	?			
real		?		?			
long real		?	?				
ref							+
name					+	+	

Table of legal conversions.

\+ Always legal and exact

? The lagality depends on the actual value being converted. Loss of accuracy is not considered an error when converting from integer values to real values. In other cases execution-time checks may have to be inserted in order to avoid loss of information due to truncation.

7. **Statements.**

```
Statement
      ::= if_statement
      ::= assignment_statement
      ::= goto_statement
      ::= routine_activation
      ::= built_in_routine
      ::= repeat_statement
      ::= case_statement
      ::= label'identifier :
```

The units of operation within the language are called statements. They will normally be executed consecutive as written. However, this sequence of operations may be broken by **goto** statements, shortened by **if** and **case** statements, and legthened by **repeat** statements.

In order to make it possible to define dynamic succession, labels may be specified among the statements.

7.1 **Goto Statement.**

```
goto_statement
      ::= goto  label'expression
```

A goto statement interrupts the normal sequence of operations, by defining its sucessor explicitly by the value of a label expression. The expression following **goto** must be of type **label**.

No goto statement can lead from the outside and into a routine body (but the opposite is legel).

A **goto** statement is undefined if the label expression eveluates to **nowhere**.

NOTE: Every label defined within a routine body must have one and only one corresponding **goto** statement in the same routine.

Labels defined within a routine body cannot be used to form label values.

Examples:

```
goto L

goto if b then x.L(3) else w(4).S
```

7.2 **Assignment Statement.**

```
assignment_statement
      ::= < variable  := >+  expression
```

Assignment statements serve for assigning the value of an expression to one or several destinations. Assignment to a routine import parameter is not alloed. If assignment is made to an indexed variable, the value of the index expression must lie within the appropriate repetition bounds. Otherwise the action of the program becomes undefined. However, there is no runtime checking on the legality of a particular index value.

The assignment process is understood to take place as follows:

- The expression is eveluated
- Then for each destination, starting with the rightmost:
  - The destinations which are of a certain degree of complexity are evaluated, e.g. indexing, dot access, dynamic access.
  - The value of the expression is assigned to the destinations.

Example: The assignment statement: `x.y := x.y.z := w`
is evaluated as: `temp := w;  x.y.z := temp;  x.y := temp;`

The type of all destinations must be same as the type of the expression, except for reference types which are treated in detail below.

Reference Assignment.

Let T be the type of a particular destination, and let T' be the type of the expression, then the assignment is valid if

- T is equal to T'
- or T is subordinate to T'
- or T' is subordinate to T

The last case involves qualification extension. There is no runtime check on the validity of an qualification extension implied by an assignment.

Examples:

```
x := y + z

x.b := var(n+o qua real) := f(a,b,c)

x(2).b(4).c.d := q
```

7.3 **If Statement.**

```
if_statement
      ::= if  Boolean'expression  then  < statement >*
             <  elsif  Boolean'expression  then  < statement >*  >*
               <  else  < statement >*  >?   endif
```

If statements cause certain statements to be executed or skipped depending on the value of certain Boolean expressions.

The operation of the if statement starts by evaluating the Boolean expression following **if**. If the value is **true**, then the statements following **then** are executed while all other statements are skipped.

If the value of the Boolean expression is **false**, then control is passed to the first **elsif**, if any. All **elsif** branches are then treated in sequence until an Boolean expression evaluates **true**, in which case the statements following the corresponding **then** is executed while all other statements are skipped.

If no other statements are executed and an **else** branch is present, the statements following **else** are executed.

Boolean expressions are evaluated as they are necessary in order to determine the further flow of the program control.

Examples:

```
if b then a:=a+1 endif


L: if if x = none then false else x.a > 0
   then T: repeat x:=x.suc while x <> none do p(x) endrepeat
   else x:=z.suc; goto T endif


   if k=1 then ds:='a'
elsif k=7 then ds:='f'
elsif k=3 then ds:='r'
elsif k=9 then ds:='q' else ds:='X' endif
```

7.4 **Repeat Statement.**

```
repeat_statement
      ::= repeat  < statement >*
            while  Boolean'expression  do  < statements >*  endrepeat
```

Repeat statements cause certain statements to be executed repeatedly.
The expression controlling repetition must be of type Boolean.

The repeat statements

```
repeat  S1  while  B  do  S2  endrepeat
```

is equivalent to

```
L:  S1  if  B
        then  S2; goto L
        endif
```

where L is an identifier not occuring elswhere in the program text.

Examples:

```
Loop: i:=0; j:=101;
      repeat i:=i+1; j:=j-2 while i < j
      do  A(j):=A(i); A(i):=A(i)+A(j+1) endrepeat

      repeat while x <> none do x.a:=P(x) endrepeat

      repeat Q(t) while t > eps do endrepeat
```

7.5 **Case Statement.**

```
case_statement
      ::= case  lower'integer_number  :  upper'integer_number
                  (  integer'expression  )
                  <  when_list  :  < statements >*  >+
                  < otherwise  < statement >*  >?
            endcase

when_list
      ::= when  which'integer_number  <  ,  which'integer_number  >*
```

The case statement causes certain statements to be executed or skipped dependig on the value of an integer expression (the selector). This value is range bounded to the specified case interval (lower,upper), but there is no runtime check on this.

To each group of statements in the case statement one or more 'which numbers' is attached. All 'which numbers' must be different and taken from the case interval.

The operation of the case statement starts by evaluating the selector. If this value is attached to some statement group, that group of statements is executed while all other are skipped.

If the selector value is not attached to any group, the statements following **otherwise** are executed. If there is no **otherwise** group, then no action is performed.

Examples:

```
case 7:13 (i)
when 9,11:   case 1:3 (j)
             when 2:   A(j):=A(j+)
             otherwise A(j):=1;
             endcase
when 7,8,10: x.a:=P(i)
when 13:     x.a:=x.b:=0
endcase


define north=1,west=2,south=3,east=4;

. . . .

case north:east (direction)
      when north: k:=draw(U); progress(k)
      when east,west: progress(k+1);
      when south: k:=draw(-U); progress(k-1);
endcase
```

8. **Routines.**

```
routine_declaration
      ::= profile_declaration
      ::= body_declaration
      ::= singular_routine
      ::= known_routine
      ::= system_routine
      ::= external_routine
```

Routines in Simuletta corresponds to subroutines in other languages but with certain restrictions:

- All parameters are passed to and from the routines by value.
  Import parameters are restricted to be 'read only'.

- Routines are not recusive.

- The return address may be made available to the routine (by means of an exit definition), thus allowing the return address to be changed by the routine itself.

The profile defines the parameters and exit descriptors for the routine, while the body defines the local variables as well as the statement sequence to be executed when the routine is activated. Each routine must have exactly one routine profile associated with is, whereas a profile may be associated with several routine bodies. The same profile shall not be associated with more than one body in any dynamic sequence of routines calling routines, since this would imply re-use of the (possibly static) allocation record defined by the profile.

8.1 **Routine profiles.**

```
profile_declaration
      ::= < global  (  identifying'string_value  )  >?
            profile profile'identifier
            parameter_specification  end

parameter_specification
      ::= < import  <  type  localid  <  ,  localid  >*  >+  >?
          < export type identifier  >?
      ::= < import  <  type  localid  <  ,  localid  >*  >+  >?
          < exit label identifier  >?

localid
      ::= identifier  <  (  repeat'integer_number  )  >?
```

The import (input) parameters and the export (return) parameter are transmittet 'by value'. The import parameters are restricted to be 'read only'. Each **import** (**export**) definition will declare a quantity local to the routine body (bodies) later associated with the profile. The order in which the parameters are given in the profile will define the order of the actual parameters in a routine activation.

An import parameter defined as a repetition must correspond to a list of expressions in the call, the count specifies the maximum permissible number of values to be transferred. Note that the actual count is not transferred, it may be defined as a separate import parameter.

An **exit** definition identifies the area containing the return address of the routine. The exit is of type **label**. It becomes accessible to the routine exactly as any other local quantity, and makes it possible for the routine to change its return address. Such a routine cannot be called from other routines.

Interface profiles (**global profile**) may occur in the head of the interface module only. The profile becomes visible from the runtime environment through the identifying string. Every profile associated to a routine address evaluation, which is an actual parameter to a system routine, should be specified as an interface profile. In fact, the interface specification is redundant, but can be used to simplify code production for routine address values.

8.2 Routine bodies.

```
body_declaration
      ::= body  (  profile'identifier  )
              routine'identifier  routine_body

routine_body
      ::= begin  <  local_variable  >*  <  statement  >*  end

local_variable
      ::= type  localid  <  ,  localid  >*
```

A routine body consists of local variables and a sequence of statements to be executed when activated. No quantities defined in the body are visible outside that body. The initial values of the local variables when entering a routine body are undefined.

The routine'identifier identifies the routine body and is used in routine activation, while profile'identifier connects the body to the relevant routine profile.

Examples:

```
profile P;
   import integer a,b(20); label l;
   exit label x;
end


body(P) R;
begin integer i; i:=1;
      repeat i:=i+1 while i<a do b(i):=b(i-1)+b(i) endrepeat
      x:=l;
end;
```

8.3 **Singular routines.**

```
singular_routine
      ::= routine  routine'identifier
              parameter_specification  routine_body
```

Singular routines are introduced to make it possible to define a routine in one construction instead of being forced to define both a profile and a routine body.

Example:

```
routine R;
   import integer a,b(20); label l;
   exit label x;
begin integer i; i:=1;
      repeat i:=i+1 while i<a do b(i):=b(i-1)+b(i) endrepeat
      x:=l;
end;
```

8.4 **Peculiar routines.**

Peculiar routines (known, system and external) are introduced to permit the Simuletta Compiler to handle each one in the most convenient system- and routine- dependent manner. The routine identifier specified will be used to refer to the routine in routine activation.

All peculiar routines are identified by an identifying string. The string contain at most 6 characters with the case of any character being insignificant (e.g. 'a' is equivalent to 'A'). All identifying strings for peculiar routines contain only alphanumeric characters, the first of which is a letter.

8.4.1 **Known routines.**

```
known_routine
      ::=  known  (  identifying'string_value  )  routine'identifier
              parameter_specification  routine_body
```

A known routine has a body defined in Simuletta. The Simuletta Compiler may know the working of the routine and may replace the body with an optimised code sequence. It is intended to be used in cases where a standard Simuletta routine will be in dager of being inefficient in some implementations, or when the routine call can be replaced by an in-line code sequence at each call.

Example:

```
known("RUT1")  Rk;
import ref(prototype) pp; exit label x;
begin range(1:10) bl;
      ...  statements  ...
      x:= ...
end
```

8.4.2 **System Routines,**

```
system_routine
      ::=  sysroutine  (  identifying'string_value  )  routine'identifier
              parameter_specification  end
```

System routines provide the interface to the runtime environment of the program, or they represent routines which are impossible to program in Simuletta (or potentially prohibitively inefficient in all implementations). Thus no body will be given. Such routines (e.g. date_and_time) are typically provided by the operating system on the target machine, and may require special intervention from the Simuletta Compiler, since the calling conventions and parameter passing mechanisms will be system-dependent.

Example:

```
sysroutine("DATTIM") date_and_time;
import infix(string) item; export integer filled end
```

8.4.3 **External Routines.**

```
external_routine
      ::=  external  (  nature'string_value  , identifying'string_value  )
             routine'identifier  parameter_specification  end
```

External routines are routines written in other languages. The exact nature of the routine is specified by the nature string. External routines are implementation dependent.

Example:

```
external("FORTRAN","SQRT") sqrt
import real x; export real result end;
```

8.5 **Routine Activation.**

```
routine_activation
      ::= routine'identifier  <  argument_list  >?
      ::= call  profile'identifier ( entry'expression ) < argument_list >?

argument_list
      ::= ( argument  < , argument >* )

argument
      ::= expression
      ::= ( expression < , expression >* )
```

A routine activation serves to call for the execution of a routine body. First the profile is connected; this will provide information about the number and types of the parameters. Then the parameter values are evaluated and transferred, and finally the actual routine to activated is connected, either explicitly by giving the routine'identifier or implicitly by evaluating the entry'expression.

Each argument correspond to an import parameter. If several values are to be transmitted to a parameter defined as a repetition, then a list of expressions must be used for that argument. Type checking during parameter transmission follows similar rules as given for the assignment statement (see sect. 7.1).

Examples:

```
R(10,if b then chr(a+3) else c,P)

call P(if b then E else x.r(2))) (x,y)

Q(a,b,(1,4,-3))
```

9. Standard Routines and Functions.

```
built_in_routine
      ::= object_initialisation
      ::= temp_routine

built_in_function
      ::= temp_function
```

9.1 Initialisation of Allocated Areas.

```
object_initialisation
      ::= zeroarea (  object_reference'expression  ,
                      object_reference'expression )

      ::= initarea (  record'identifier  ,  object_reference'expression )

      ::= dinitarea (  record'identifier , fixrep'integer_number   ,
                       object_reference'expression )
```

For the purpose of giving dynamically allocated areas initial values, three standard routines are defined. These routines will allways be used in the following manner:

- when an area has been allocated by the system environment, or when a possible garbage collection has returned free storage, the area(s) will be zero-filled by 'zeroarea',

- when a particular area has been acquired (somehow) to be structured by some type, one of the routines 'initarea' or 'diniarea' is issued.

This usage pattern must be enforced by the user, i.e. when 'initarea' is to be evaluated the Simuletta Compiler may assume that the area to be initialised has been zero-filled. Thus an implementation may choose to realise either 'zeroarea' or the iniarea-pair, or it may choose a mixed strategy, zero filling the area ('zeroarea' implmented) and partly implementing '(d)initarea' for those components which do not have a zero representation. It should be obvious that the complete implementation of all will be redundant and will probably lead to considerable runtime overhead.

zeroarea(x,y):	The area between x and y (x included, y not) wil be zero-filled.
initarea(R,x):	The structure of R is imposed upon the area, and the area is initialised according to the table below. Only the common part of an instance of a structure will be initialised, ignoring both the prefix and any variant part(s). The structure is initialised component by component according to the table below.
diniarea(R,fix,x):	The value of 'fix' is used to resolve the type, i.e. fixing the number of elements in the indefinite repetition, following that the evaluation proceeds exactly as for 'initarea'.

Important Note: A possible prefix part or any variant parts will never be altered by '(d)initarea'.

Area initialisation values

type:	initialised to:
Boolean	**false**
Character	NUL (ISO repr 0)
integer	0
real	0.0
long real	0.0&&1
size	**nosize**
field	**nofield**
ref	**none**
name	**noname**
label	**nowhere**
entry	**nobody**
infix(R)	- each attribute init. as above.

Note: If these values are represented as zero and if 'zeroarea' is implemented, the routines 'initarea' and 'dinitarea' may safely be ignored !

9.2 **Intermediate Results.**

```
temp_routine
      ::= init_pointer( object_reference'expression )
      ::= set_pointer( object_reference'expression )

temp_function
      ::= max_temps           -- resulting type: size
      ::= get_pointer         -- resulting type: ref()
```

During the S-Compilation the compile stack will regularly contain items, which describe partially evaluated expressons such as e.g. incomplete address calculations. The execution of the corresponding machine instructions will, at runtime, give rise to intermediate results; of necessity these must be held in some form of anonymous storage, '**the temporary area**'. The actual implementation of this area should be highly target machine dependent, thus the machine registers may be used if a sufficienly large number of registers is available.

In full S-Code the intermediate results are copied between the temporary area and some object at certain places in the code. Such code are e.g. generated by the Simula Front-end Compiler.

To be able to manipulate such save-objects from Simuletta, the above routines and functions are defined. The temp control instructions are used as follows:

- 'init_pointer' is called in preparation of a complete scan through the pointers of a save-object.

- During the scan 'get_pointer' will be called repeatedly, yielding the pointers successivly.

- If the pointer is to be updated, one 'set_pointer' will follow the corresponding 'get_pointer', so that the pointer just inspected will be updated.

Apart from the actual temporaries saved, some additional information must be present in the save-objects. For the purpose of explanation we will call this additional attribute '**SaveMarks**'.
The **SaveMarks** is some representation of the structure of the save-object, which allows for sequential access to all pointer values saved. (E.g. A bit map indicating the positions of the pointers in the save-object).

The parameters to the instructions 'get_pointer' and 'set_pointer' are implicit, i.e. they refer to the save-object referenced by the most recent call (at runtime) on 'init_pointer', successive calls on 'get_pointer' scans through the pointers in the save-object, and a call on 'set_pointer' refers to the pointer accessed by the most resent call on 'get_pointer'.

For the purpose of explanation we introduce two anonymous variables '**SaveObject**' and '**SaveIndex**'. **SaveObject** is set by 'init_pointer' and referenced by 'get_pointer' and 'set_pointer'. **SaveIndex** is initialized by 'init_pointer', updated by 'get_pointer' and referenced by 'set_pointer'. In an implementation some representation of **SaveObject** and **SaveIndex** could be kept in dedicated registers or in main storage. The use of the variables is explained in detail below.

max_temps:	The value of this function is the size of the biggest save-object which will ever occur.
init_pointer(x):	A scan of the save-object 'x' is initialized, i.e. **SaveObject** is set to refer to the object 'x', and **SaveIndex** is initialized.
get_pointer:	If SaveIndex refers to the 'last' pointer of the save-object referred by **SaveObject** or no pointer exists in the object, the value **none** is returned to signal that the scan of the object should be terminated. Otherwise **SaveIndex** is updated to describe the 'next' pointer in the save-object. In case the value of the 'next' pointer is **none**, the pointer is skipped, i.e. iterate this description, otherwise the value of the referred pointer is returned.
set_pointer(x):	The value 'x' is inserted into the pointer variable referred by **SaveObject** and **SaveIndex**. Note that 'set_pointer' does not update **SaveIndex**. **Important note**: Only the object reference part of a general reference is updated. This instruction is issued by the Garbage Collector during the storage compaction, and objects are always moved as a whole.

Appendix – The Complete Syntax of the Simuletta Language

```
simuletta_program
      ::= interface_module
      ::= sub_module

interface_module
      ::= global module'identifier
          begin < < visible >?  decl_in_interface >*  end

decl_in_interface
      ::= mnmonic_definition
      ::= global_declaration
      ::= constant_declaration
      ::= record_declaration
      ::= routine_declaration

sub_module
      ::= module module'identifier
          begin < < visible >?  decl_in_module >*
                < statement >*  end

decl_in_module
      ::= module_inclusion
      ::= mnmonic_definition
      ::= local_declaration
      ::= constant_declaration
      ::= record_declaration
      ::= routine_declaration

main_program
      ::= begin  < decl_in_module >*  < statement >* end

global_declaration
      ::= type  globalid  <  ,  gloalid  >*

globalid
      ::= identifier  < system identifier >?  < = value >?
      ::= identifier  < system identifier >? ( repeat'integer_number )
            < = ( value < , value >* ) >?

local_declaration
      ::= type localid  <  ,  localid  >*

localid
      ::= identifier  < ( repeat'integer_number ) >?

constant_declaration
      ::= const  type  constantid  <  ,  constantid  >*

constantid
      ::= identifier = value
      ::= identifier ( repeat'integer_number ) = ( value < , value >* )
```

```
module_inclusion
      ::= insert     module_ident  <  ,  module_ident  >*
      ::= sysinsert  module_ident  <  ,  module_ident  >*

module_ident
      ::= module'identifier  <  =  external_id'string_value  >?

record_declaration
      ::= record  record'identifier  <  :  prefix'identifier  >?
            <  record_info  >?
          begin  common_part  < variant_part >*  end

record_info
      ::= info "DYNAMIC"
      ::= info "TYPE"

common_part
      ::=  <  type  attrid  <  ,  attrid  >*  >*

variant_part
      ::= variant  <  type  attrid  <  ,  attrid  >*  >*

attrid
      ::= attribute'identifier  <  (  repeat'integer_number  )  >?


type
   ::= integer
   ::= short integer
   ::= range  (  lower'integer_number  :  upper'integer_number  )
   ::= real
   ::= long real
   ::= size
   ::= Boolean
   ::= character
   ::= label
   ::= entry  (  < profile'identifier >?  )
   ::= structured_type
   ::= object_reference
   ::= attribute_reference
   ::= general_reference

structured_type
   ::= infix  (  record'identifier  <  :  fixrep'integer_number  >?  )

object_reference
   ::= ref  (  < record'identifier >?  )

attribute_reference
   ::= field  (  < qualifying'type >?  )

general_reference
   ::= name  (  < qualifying'type >?  )
```

```
value
   ::= integer_number
   ::= real_number
   ::= long_real_number
   ::= Boolean_value
   ::= character_value
   ::= string_value
   ::= label_value
   ::= routine_value
   ::= structured_value
   ::= object_reference_value
   ::= attribute_reference_value
   ::= general_reference_value

Boolean_value
      ::= true
      ::= false


size_value
      ::= nosize
      ::= size  (  record'identifier  )
      ::= size  (  record'identifier  :  integer'expression  )

Structured_value
   ::= record  :  record'identifier  (  attrvalue < ,  attrvalue >*  )

attrvalue
   ::= attribute'identifier  =  value
   ::= attribute'identifier  =  (  value  <  ,  value  >*  )

Object_reference_value
      ::= none
      ::= ref  (  object'identifier  )

attribute_reference_value
      ::= nofield
      ::= field  (  record'identifier  <  .  identifier  >+  )

label_value
      ::= nowhere
      ::= label'identifier

entry_value
      ::= nobody
      ::= entry  (  routine'identifier  )
```

```
expression
      ::= factor
      ::= unary_operator  factor
      ::= expression  binary_operator  factor
      ::= if Boolean'expression then expression else expression
      ::= type_conversion

factor
      ::= value
      ::= variable
      ::= routine_activation
      ::= (  expression  )

unary_operator
      ::=  + | - | not

binary_operator
      ::=  + | - | * | / | rem
      ::=  and | or | xor
      ::=  <> | < | <= | = | >= | >

variable
      ::= simple_variable  <  (  index'expression  )  >?

simple_variable
      ::= identifier
      ::= var  (  general_reference'expression  )
      ::= object_reference'expression  .  identifier
      ::= structured_type'expression  .  identifier

type_conversion
      ::=  expression  qua  type


Statement
      ::= if_statement
      ::= assignment_statement
      ::= goto_statement
      ::= routine_activation
      ::= built_in_routine
      ::= repeat_statement
      ::= case_statement
      ::= label'identifier :

goto_statement
      ::= goto  label'expression

assignment_statement
      ::= < variable  := >+  expression

if_statement
      ::= if  Boolean'expression  then  < statement >*
            <  elsif  Boolean'expression  then  < statement >*  >*
              <  else  < statement >*  >?  endif

repeat_statement
      ::= repeat  < statement >*
            while  Boolean'expression  do  < statements >*  endrepeat
```

```
case_statement
      ::= case  lower'integer_number  :  upper'integer_number
                  (  integer'expression  )
                  <  when_list  :  < statements >*  >+
                  < otherwise  < statement >*  >?
          endcase

when_list
      ::= when  which'integer_number  <  ,  which'integer_number  >*


routine_declaration
      ::= profile_declaration
      ::= body_declaration
      ::= singular_routine
      ::= known_routine
      ::= system_routine
      ::= external_routine

profile_declaration
      ::= < global  (  identifying'string_value  )  >?
            profile profile'identifier
            parameter_specification  end

parameter_specification
      ::= < import  <  type  localid  <  ,  localid  >*  >+  >?
          < export type identifier  >?
      ::= < import  <  type  localid  <  ,  localid  >*  >+  >?
          < exit label identifier  >?

localid
      ::= identifier  <  (  repeat'integer_number  )  >?

body_declaration
      ::= body  (  profile'identifier  )
               routine'identifier  routine_body

routine_body
      ::= begin  <  local_variable  >*  <  statement  >*  end

local_variable
      ::= type  localid  <  ,  localid  >*

singular_routine
      ::= routine  routine'identifier
              parameter_specification  routine_body

system_routine
      ::=  sysroutine  (  identifying'string_value  )  routine'identifier
              parameter_specification  end

external_routine
      ::=  external  (  nature'string_value  , identifying'string_value  )
             routine'identifier  parameter_specification  end
```

```
routine_activation
      ::= routine'identifier  <  argument_list  >?
      ::= call  profile'identifier ( entry'expression ) < argument_list >?

argument_list
      ::= ( argument  < , argument >* )

argument
      ::= expression
      ::= ( expression < , expression >* )

built_in_routine
      ::= object_initialisation
      ::= temp_routine

built_in_function
      ::= temp_function

object_initialisation
      ::= zeroarea (  object_reference'expression  ,
                      object_reference'expression )

      ::= initarea (  record'identifier  ,  object_reference'expression )

      ::= dinitarea (  record'identifier , fixrep'integer_number   ,
                       object_reference'expression )

temp_routine
      ::= init_pointer( object_reference'expression )
      ::= set_pointer( object_reference'expression )

temp_function
      ::= max_temps           -- resulting type: size
      ::= get_pointer         -- resulting type: ref()
```