
LORIA - UMR 7503 - (INRIA - CNRS - University Henri Poincaré)
Campus Scientifique, BP 239,
54506 Vandœuvre-lès-Nancy Cedex
FRANCE

Programmer's Reference Manual



Lisaac V.0.2

The power of simplicity at work for Operating Systems

by

SONNTAG Benoît
BOUTET Jérôme

{bsonntag, boutet}@loria.fr

Mis en page avec la classe thloria.

Contents

1	Introduction	5
1.1	Motivation	5
1.2	The LISAAC compiler	6
1.3	Why Using LISAAC	7
1.4	Notations	8
2	Quickstart - for beginners	9
2.1	LISAAC : a prototype based language	9
2.2	Notations	10
2.3	Objects	10
2.4	Slots	11
2.4.1	Methods and functions	11
2.4.2	Local variables	16
2.5	Compilation and running	16
2.6	How to write	17
2.6.1	Types	17
2.6.2	My first LISAAC program	18
2.6.3	How to print	18
2.6.4	How to read	18
2.6.5	Conditionals: <i>if else</i>	19
2.6.6	A loop: <i>do_while</i>	20
2.7	LISAAC : an object oriented language	20
2.7.1	Clone	21
2.7.2	Inheritance	23
3	Language Reference	27
3.1	Lexical and syntax overview	27
3.1.1	Lexical overview	27
3.1.2	Syntax overview	29

Contents

3.2	Sections identifiers	30
3.2.1	The HEADER section	31
3.2.2	The INHERIT section	32
3.2.3	The MAPPING section	47
3.2.4	The INTERRUPT section	48
3.2.5	The EXTERNAL section	49
3.2.6	Other sections	50
3.3	Type names	55
3.3.1	Genericity	55
3.3.2	Invariant's type control	55
3.3.3	Particular type: SELF type	56
3.3.4	Default value of a slot according to its type.	58
3.4	Slots	58
3.4.1	Shared slots	58
3.4.2	Non shared slots	62
3.4.3	Expanded slots	64
3.5	Slot descriptors	70
3.5.1	Keyword slots	70
3.5.2	Binary messages	71
3.5.3	Unary messages	72
3.6	Message send, late binding	74
3.7	Assignment	74
3.7.1	Typing rules	75
3.7.2	Implicit-receiver messages	75
3.7.3	A particular assignment: ?=	76
3.7.4	Binary message send	77
3.7.5	Unary message send	77
3.8	Statement lists	77
3.8.1	Return values of lists	78
3.8.2	Use of lists	79
3.8.3	Local variables in statement lists	82
3.9	Statement blocks	83
3.9.1	Return values of blocks	84
3.9.2	Use of blocks	84
3.9.3	Argument and local variables in statement blocks	86
3.10	Auto-cast	87
3.11	Tools for programming by contract	87

3.11.1	Requires and Ensures	89
3.11.2	Invariant	90
3.11.3	Result and Old	91
3.11.4	Inheritance	92
3.12	Externals	93
3.12.1	Slot external	93
3.12.2	C code in Lisaac	93
3.12.3	Lisaac code in C	95
3.12.4	Lisaac external	96
4	The LISAAC Library	97
4.1	OBJECT	97
4.2	NUMERIC	98
4.3	CHARACTER	100
4.4	BOOLEAN	100
4.5	BLOCK	102
4.6	NATIVE_ARRAY	102
4.7	STRING	103
4.8	FIXED_ARRAY	105
4.9	STD_INPUT	106
4.10	STD_OUTPUT	107
4.11	COMMAND_LINE	107
4.12	Default values	107
5	The LISAAC World	109
5.1	Glossary of useful selectors	109
5.1.1	Assignment	109
5.1.2	Cloning	109
5.1.3	Comparisons	109
5.1.4	Numeric operations	110
5.1.5	Logical operations (BOOLEAN) (<i>see 5.2.1</i>)	110
5.1.6	Bitwise operations (INTEGER)	110
5.1.7	Control	110
5.1.8	Debugging	111
5.2	Control Structures: Booleans and Conditionals	111
5.2.1	Booleans expression	111
5.2.2	Conditionals	112

Contents

5.3	Loops	112
5.3.1	Pre-tested looping	112
5.3.2	Post-tested looping	113
5.3.3	Iterators looping	113
5.4	Collections	113
5.4.1	List of collections	113
5.4.2	Example	113
	Bibliography	115
	Index	117

Chapter 0001b

Introduction

LISAAC is the first object-oriented language based on prototype concepts really compiled, with system programming facilities. Actually, two languages are at the origin of LISAAC : the Self language [US87] for its flexibility, the concept of dynamic inheritance and the Eiffel language [Mey94] for its static typing and security (programming by contract). The LISAAC compiler produce optimized C Ansi Code, which can then be compiled on every architecture with an appropriate C Compiler (GCC or others), which made LISAAC a real multi-platform language. Moreover, results of compiled objects show that it is possible to obtain executable from a high-level prototype-based language that are as fast as C programs.

1.1 Motivation

The design as well as the implementation of the ISAAC ¹ operating system [Son00] led us to design a new programming language named LISAAC .

LISAAC integrates communications protection mechanisms, system interruptions support as well as drivers memory mapping. The use of prototypes and especially dynamic inheritance fit a flexible operating system in the making.

The purpose of our project is to break with the internal rigidity of current operating systems architecture that mainly depends, in our opinion, on the low-level languages that have been used to write them.

Thus, our Isaac operating system has been fully written with a high-level prototype-based language.

The evolution of programming languages currently fulfills nowadays data-processing needs and constraints in terms of software conception and production.

Nevertheless, modern languages such as object-oriented languages have not brought a real alternative to procedural programming languages like C in the development of modern operating systems.

Historically, during the creation of an operating system, constraints related to the hardware programming have been systematically fulfilled with a low-level language, such as the C language.

This choice leads in general to a lack of flexibility that can be felt at the applicative layer.

¹Isaac:Object-oriented Operating System.

Chapter 1. Introduction

Our thoughts led us to design and implement a new object-oriented language with extra facilities useful for the implementation of an operating system. In order to achieve that goal, we started to look for an existing object-oriented language with powerful characteristics in terms of flexibility and expressiveness.

Our language also comes from an experiment in the creation of an operating system based on dynamic objects, whose possibilities are a subtle mix of Self with Eiffel, with the addition of some low-level capabilities of the C language. From the Eiffel language, we borrowed a kind of static typing form and programming by contract, using assertions to securise programming.

Our language LISAAC is the first compiled prototype-based language really usable. Compiled objects remain objects with all their capacities, which preserved all the expressivity. Hardware facilities are included in the native language, such as mapping capability or interrupt management.

1.2 The LISAAC compiler

The LISAAC compiler produce optimized C Ansi Code, which can then be compiled on every architecture with an appropriate C Compiler (GCC or others).

The compiler is fully written in LISAAC , the bootstrap was done since 2004, january.

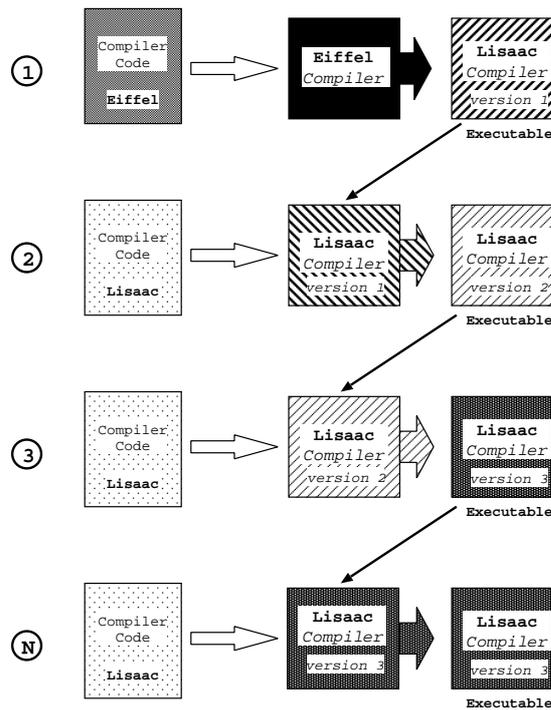
The mechanism of the bootstrap is explained in the following figure.

State 1: the first version of the LISAAC compiler is written in another language (here Eiffel), and compiled as every other Eiffel code with the Eiffel Compiler. It produces an executable, the first version of LISAAC compiler.

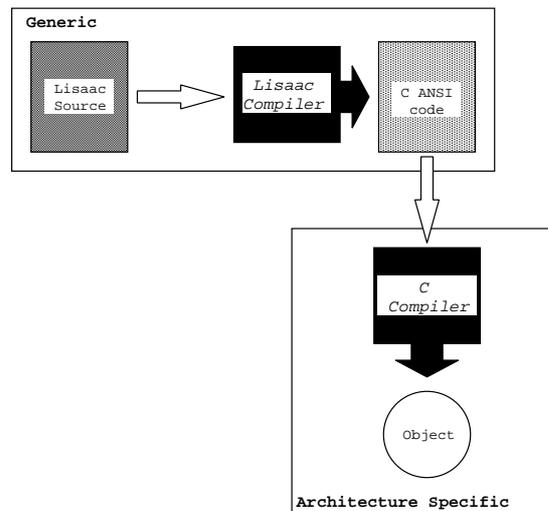
State 2: the code of the compiler is fully translated in LISAAC . We use our LISAAC compiler (version 1) to compile the code (as we can done for every program written in LISAAC). It produces an executable, the second version of LISAAC compiler. In fact, if there is no error, version 1 and version 2 operates equally. This compiler produce executable code from LISAAC , but it was built using Eiffel technology, so it is depend of Eiffel.

State 3: the code of the compiler, written in LISAAC and used at the state 2, is compiled again, using the version 2 of our LISAAC compiler. It produces the version 3 of the LISAAC compiler.

Every iteration of the state 3 don't change the produced executable, we are in a stable state. Obviously, the code of the compiler must be sure with no error before starting the bootstrap operation. The advantage of this bootstrap is that we are now totally independent of another language: the compiler is now written in LISAAC , with a usable version compiled with itself: the compiler is built using only LISAAC technology.



The compiler can be run on every architecture who have a C compiler.



1.3 Why Using LISAAC

LISAAC was first developed to implement ISAAC Operating System but became an independent object oriented language, who can be used to write all kind of programs.

It deals with numerous advantages: it's a powerful high level language, based on prototype concepts. Programming security was from the beginning a real aspect of LISAAC , with the static typing and assertion management (programming by contract) such as *Requires*, *Ensures* and *Invariant*, and lot of verification during compilation.

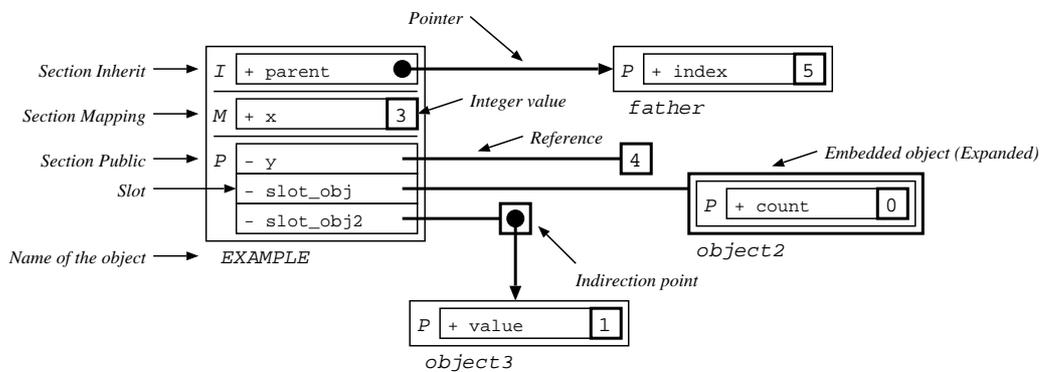
Numerous high level optimizations give efficiency and speed to compiled code.

A large library, fully written in LISAAC , supply the programmer with a large scale of built-in prototypes and functions, such as:

- Number (signed / unsigned 8, 16, 32, 64 bits integer; real; infinite accuracy integer)
- Collections: variable arrays, linked-lists, dictionary (associativity key-value), set
- Hash coding
- Memory management
- Input / Output
- File System (Unix / Linux ; Windows / Dos)
- Image format (bitmap; vectorial)
- Graphic (8, 15, 16, 24, 32 bits)
- Time and Date

1.4 Notations

In this document, you'll find memory representation of objects. Here is the caption of the figures.



Chapter 0010b

Quickstart - for beginners

After the read of this chapter, you would be able to write simples programs in LISAAC .
The next chapter will give you more of LISAAC , allowing you to use all the capabilities and the power of LISAAC .

2.1 LISAAC : a prototype based language

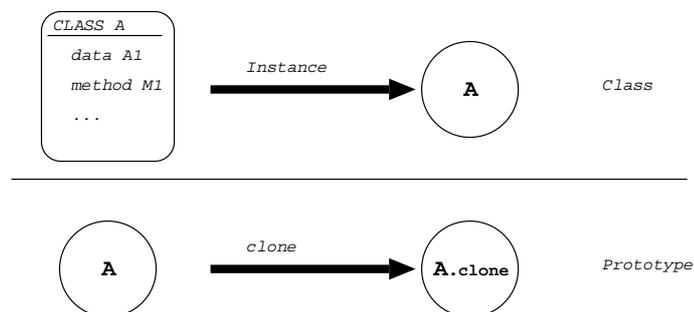
LISAAC is an object oriented language based on prototype concepts.

Class and prototype languages differs on few but important points. In a class language, you have to instance an object from its description in order to make it alive.

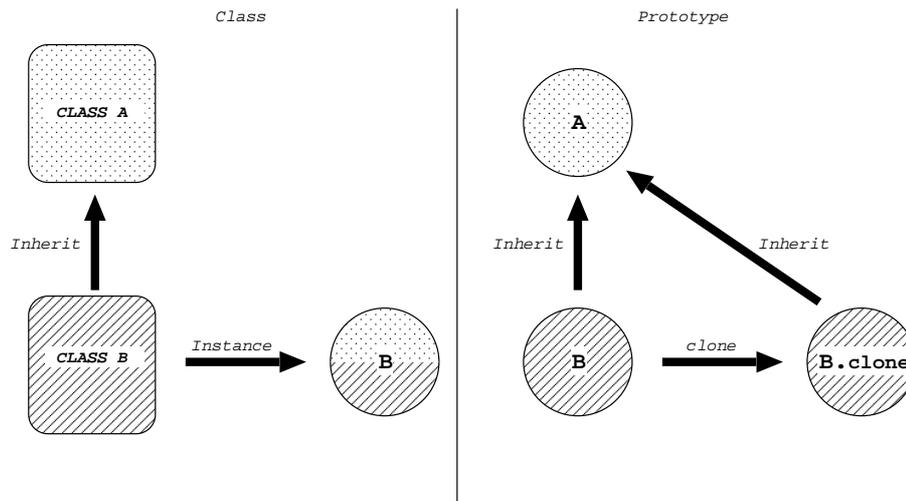
In a prototype language, a description of object is already alive. In LISAAC you can use directly the “master” object without instanciate it.

This particular object is written in capitals and can be use as any object.

Other objects are obtained by cloning the “master” object. The **clone** routine is not a hard-coded function but a function from library.



With this property, we can see that inheritance is particular. Objects inherits from alive objects, with their own live. It permits numerous variations from inheritance, depending of its type (+ or -), as sharing parents between 2 cloned object, or dynamic inheritance (in changing the reference on the parent). We will see this later.



Objects are the fundamental entities in LISAAC ; every entity in a LISAAC program is represented by one or several objects. Even control is handled by objects: blocks (3.9 page 83) are LISAAC closures used to implement user-defined control structures. An object is composed of a set of slots. A slot is a name-value pair. Slots may contain references to other objects. When a slot is found during a message lookup (see section 3.2.2.0 page 44), the object in the slot is evaluated.

2.2 Notations

LISAAC is case sensitive, and respects the following constraints:

Variables and slots are written with small letters (x , *counter*, ...).

Type of objects (or "master" name object / prototype) are in capital letters (INTEGER, BOOLEAN, ...).

Keywords are written with small letters but starts with a capital letter (**Section**, **HEADER**, ...).

Symbol `:=` is an affectation. Be careful not to use symbol `=` which compares 2 objects and returns a boolean.

You will see symbol `+` or `-` before the slots. It defines the type of the slot and is mandatory ! Its role will be explained in the following pages.

A sequence ends with `;`. If not, the compiler continues to the following line.

You can define a list of sequence between `(` and `)`. See section 3.8 on page 77 for more information on instruction lists.

Comments begin with `//` and stops at the end of the line.

For comments on more than one line, start with `/*` and end at `*/`.

2.3 Objects

In LISAAC , objects are the fundamental entities. Everything is represented by one or more objects, from a simple Integer or Boolean to more complex entities like arrays or window.

One object is written in one and only one file, named with the name of the object and with the extension `.li`.

For example, `INTEGER.LI`, `BOOLEAN.LI`, `WINDOW.LI`, ...

A source code of an object is divided in sections.

Section Header is needed. In this section, you define the name of the object and the category. Here we use the APPLICATION category. For the use of other categories see later. Then you have the **Section Public**, in which you will write the slot who will be executed (see next paragraph).

Section, **Header** and **Public** are keywords.

Example: file HELLO_WORLD.LI

Section Header

```
+ name      := HELLO_WORLD;    // Name is in capital letters
- category  := APPLICATION;    // Category is in capital letters
```

Section Public

```
/* ...*/
```



Note that after **Section xxxx** there is no `;`.

2.4 Slots

An object is composed by slots, which is a service given by the object.

A slot can be a data as well as code (function or method).

A slot is defined with a name. It can also add a static type for datas and functions.

A slot is prefixed by `+` or `-` sign, which give the type of slot (to simplify, with `-` value is shared between objects, with `+` value is local to the object, we describe this later).

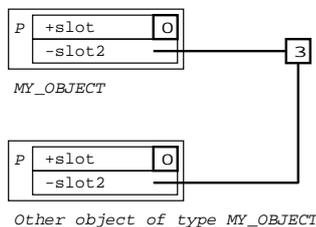
The type is defined with the sign `:` as prefix of the type.

Section Header

```
+ name      := MY_OBJECT;    // Name is in capital letters
- category  := APPLICATION;  // Category is in capital letters
```

Section Public

```
+ slot:INTEGER;                // Value local to the object, init with INTEGER default value
- slot2:INTEGER := 3;          // Value shared between objects, init with value '3'
```



All of this will be explain later.

2.4.1 Methods and functions

Simple slots

As said before, a slot can also be function or method.

Methods(or routines) are a fundamental notion in object-oriented languages, together with their

Chapter 2. Quickstart - for beginners

companion concept late binding (or message send, routine call, method call, dynamic dispatch, ...).

There is 2 types of code slot: the first one is executed within the load of the object. It's defined as a default value of a variable.

```
+ slot :INTEGER := 3 + 4;      // Operation '3 + 4' is evaluated at init
+ slot2:INTEGER := slot * 2;
```

The second type is executed only on the call of the slot. It's defined with the <- symbol.

```
+ slot :INTEGER := 3;
+ slot2 :INTEGER <- ( 5 + slot; ) ; // Operation '5 + slot' evaluated when calling slot2
```

More complex code can be defined within parenthesis.

```
+ slot :INTEGER := 3;
+ slot2 :INTEGER := 4;
+ slot3 <-      // Slot without return value
(
  slot := slot + 3;
  slot2 := slot + 5;
  slot2 := slot2 * 3;
);
```

Note that you can write all your text on the same line, but it's more easy to read to align the code like that. The return value of a code is the value of the last code (without ;) before)

```
+ slot :INTEGER := 3;
+ slot2 :INTEGER <-
(
  slot           // Return Value, evaluated with call of code5
);
+ slot3:INTEGER :=
(
  slot := slot + 6;
  slot := slot2 * 5;
  slot           // Return Value, evaluated during load of the object
);
```

 Be carefull, the type of return of the slot must be the same as the value of the type of returned value.

```
+ slot :INTEGER := 3;
+ slot2 :INTEGER := 4;
+ slot3 :BOOLEAN <-
(
  slot = slot2           // Returns TRUE if equal, FALSE if not
```

```

);
+ slot4:INTEGER <-
(
  slot3                                // Error: type are different
);

```

Call of a slot

The call of a slot depends on when it happens.

File object1.li

Section Header

```

+ name := OBJECT1;
- category := APPLICATION;

```

Section Public

```

+ slot :INTEGER := 3;
+ slot2 :INTEGER <-
(
  slot * 2 + 4                        // Call of 'slot' from the same object
);

```

An object is initialized with NULL value, and a call on it will fall in error. In a prototype language (see 2.1 page 9), the 'master' object (written in capital) is alive without need to instantiate it. Other objects are created by cloning the 'master' object using the **clone** slot. For more information see 2.7 page 20.

File main_object.li

Section Header

```

+ name := MAIN_OBJECT;
- category := APPLICATION;

```

Section Public

```

+ slot_object:OBJECT1;
+ slot_object2 :INTEGER <-
(
  slot_object := OBJECT1.clone;      // clone of the OBJECT1 object.
  slot_object.slot2 + 5              // Call of 'slot2' on 'slot_object' of OBJECT1 type
);

```

The symbol `.` defines a call of a slot on another object.

Slots with arguments

You can also call a slot with parameters.

```

+ slot a:INTEGER :INTEGER <-        // 1 parameter. You don't need parenthesis
(
  a * 2

```

```
);  
  
+ slot2 (a,b:INTEGER) :INTEGER <- // 2 parameters of the same type  
(  
  a + b  
)  
);  
  
+ slot3 (a:INTEGER,b:CHARACTER) :INTEGER <- // 2 parameters, type different  
(  
  b.print;  
  a * 3  
)  
);  
  
+ slot4 :INTEGER <-  
(  
  slot 3 + slot2 (2,3) + slot3 (4,'y')           // call of slots  
)  
);
```

You can define your own keywords to separate parameters.

```
+ slot a:INTEGER value b:INTEGER :INTEGER <- // value is my defined-keyword  
(  
  a + b * 2  
)  
);  
  
+ slot2 (a,b:INTEGER) write c:CHARACTER :INTEGER <-  
(  
  c.print;  
  a * b  
)  
);  
  
+ slot3 a:INTEGER multiply b:INTEGER add c:INTEGER:INTEGER <-  
(  
  a * (b + c)  
)  
);  
  
+ slot4 :INTEGER <-  
(  
  slot 3 + (slot2 (2,3) write 'c') + (slot3 4 multiply 5 add 6) // call of slots  
)  
);
```

Assignment of slots



Be carefull, you can't assign a value to a slot outside the object, as defined in the following example.

Section Header

```
+ name := OBJECT1;
```

```
- category := APPLICATION;
```

Section Public

```
+ value:INTEGER := 3;
```

Section Header

```
+ name := MAIN_OBJECT;
```

```
- category := APPLICATION;
```

Section Public

```
+ slot_object:OBJECT1;
```

```
- method <-
```

```
(
```

```
  slot_object := OBJECT1.clone;
```

```
  slot_object.value := 4;           // The compiler will stop
```

```
);
```

This is done to protect slots of objects. When you define an object, you must specify the slots who can change by creating methods dedicated for that. You can find this tedious, but it will insure that you have the total control of what is done with your object. Just imagine a slot counter who can be modified by anybody working with your object, ... You can then define conditions inside the method to protect more your object.

Example: use of a 'setter'

Section Header

```
+ name := OBJECT1;
```

```
- category := APPLICATION;
```

Section Public

```
+ value:INTEGER := 3;
```

```
- set_value v:INTEGER <- // Define your own setter
```

```
(
```

```
  (v > 0).if {
```

```
    value := v;
```

```
  } else {
```

```
    value := 0;
```

```
  };
```

```
);
```

Section Header

```
+ name := MAIN_OBJECT;
```

```
- category := APPLICATION;
```

Section Public

```
+ slot_object:OBJECT1;
```

```
- method <-
```

```
(
```

```
  slot_object := OBJECT1.clone;
```

```
  slot_object.set_value 4;
```

```
);
```

2.4.2 Local variables

You can define local variable inside your slot. The syntax is the same as for a slot. A local variable will be often not shared (+). The local variable is initialized with the default value of its type.

```
+ slot a:INTEGER :INTEGER <-
( + var1:INTEGER;
  + var2,var3:INTEGER;
  + result:INTEGER;
  var1 := a * 2;
  var2 := a + 4;
  var3 := a - 5;
  result := var1 + var2 - var3;
  result
);
```



Note that you must define all the variable in the first lines of your slot, without code inside this definition list.

```
+ slot a:INTEGER :INTEGER <-
( + var1:INTEGER;
  + var2:INTEGER;
  + result:INTEGER;
  var1 := a * 2;
  var2 := a + 4;
  + var3:INTEGER;                               // The compiler will stop with error
  var3 := a - 5;
  result := var1 + var2 - var3;
  result
);
```

2.5 Compilation and running

To compile your LISAAC programs, you'll have simply to type:

```
lisaac my_object.li
```

It produces 2 files: `my_object.c` and `my_object`, who is an executable. By default LISAAC uses GCC to compile the C code produced.

Running an object

In your main object, you must have only one slot in the **Section Public**. It will be executed at the run of your compiled file.

Section Header

```
+ name      := OBJECT_TO_RUN;
```

```

- category := APPLICATION;
Section Public
+ value:INTEGER := 3;
- go <-
(
  value.print;
);

```

When compiling this program there will be an error: 2 entry points. To correct this error, the **value** slot must be written in a **Section Private**, which is a particular Section, visible only in the current object.

```

Section Header
+ name      := OBJECT_TO_RUN;
- category := APPLICATION;
Section Private
+ value:INTEGER := 3;
Section Public
- go <-
(
  value.print;
);

```

For more information on slots, methods and method calls, see section 3.5 page 70 and section 3.6 page 74.

2.6 How to write

2.6.1 Types

There is no built-in types in LISAAC . Every type is from the library (you can check the source code to see how it is implemented).

The base types you can use are :

- INTEGER with the arithmetic operations and lot of other (implemented in NUMERIC object, parent of all number types)

Notations: *12*, *12d*: decimal value
1BAh, *0FFh*: hexadecimal value
01010b, *10b*: binary value
14o, *6o*: octal value
10KB, *10MB*, *10GB*: system facility

- BOOLEAN: you have 2 'values' for BOOLEAN: TRUE or FALSE. Each of this values are also objects.
- CHARACTER: a simple character

Notations: *'a'*, *'Z'*, *'4'*: simple character
'\n', *'\t'*, *'\r'*: escape character
'\10|', *'\0Ah|'*: code character

- `STRING_CONSTANT`: composed by multiple characters, cannot be modified, defined between " "

Notations: `"Hello World\n"`: simple string

- `STRING`: string to built with functions of the library
- `FIXED_ARRAY`: an array with fixed lower bound, with lot of operations
- `BLOCK`: a block of code, defined between { and }

See chapter on the library for more details.

2.6.2 My first LISAAC program

Here is the classical “Hello World” program, that writes to the standard output:

```
Edit File HELLO_WORLD.LI
```

Section Header

```
+ name := HELLO_WORLD;  
- category := APPLICATION;
```

Section Public

```
- main := "Hello world !".print;    // the slot executed
```

Compile with: `lisaac hello_world` or `lisaac hello_world.li` It produces an executable file called `hello_world`.

In this first LISAAC program, **main** is the root of the system, or beginning of execution (main program). The name doesn’t matter, it would be “start”, “begin” or whatever you want.

The only instruction in the main program is evaluated (i.e. executed) immediately at program startup.

Everything is object in LISAAC , as you can see in this example: the slot **print** is called on the String object “Hello world !”.

See chapter 3 for more explanation.

2.6.3 How to print

As we see before, the method **print** is a library method in the `STRING` prototype. But there is also the same method for `NUMERIC` types.

```
"Hello World !".print;  
3.print;  
my_string.print;    // object of STRING type (created before, of course)
```

2.6.4 How to read

Now, let’s also read from the standard input:

Section Header

```
- name := HOW_TO_READ;
- category := APPLICATION;
```

Section Public

```
- main :=          // a multi-line main
(
  "Enter your name : ".print;
  IO.read_string;
  ("Welcome, " + IO.last_string).print;
);
```

`last_string` returns a reference to the last string that was entered from the standard output. Note the use of the `IO` predefined, initial prototype, for input-output.

2.6.5 Conditionals: *if else*

A basic control structure in many languages is the `if - then - else` construct. In LISAAC, the `then` is omitted; the appropriate construction². As we see before, everything is object, this conditional method deals with the same pattern: `condition.if block_true else block_false`

`condition` is a `BOOLEAN` object (true or false) on which you call the method `if` with 2 parameters: `block_true` and `block_false` (objects of type `BLOCK`), separated by the keyword `else`

Section Header

```
- name := IF_ELSE;
- category := APPLICATION;
```

Section Public

```
- main :=
( + gender:CHARACTER;    // a local variable

  IO.put_string "Enter your gender (M/F) : ";
  IO.read_character;
  gender := IO.last_character;

  (gender == 'M').if {    // conditional
    IO.put_string "Hello Mister !"; // then part
  } else {
    IO.put_string "Hello Miss !";  // else part
  };
);
```

Note that you can use `"my_string".print` or else `IO.put_string "my_string"`. It has the same effect.

Note the use of a local variable `gender` to hold the user's answer. See section 3.8.3 page 82 for local variable declaration in lists of instructions.

²`if else` in LISAAC is not a language construct *per se*, but a simple method call.

The conditional is made of a boolean expression (`gender == 'M'`) to which the message **if else** is sent. See section 3.6 about message send, and section 3.9.2.0 page 85 about booleans and conditionals.

Note that a list of instructions and an expression are the same syntactical construct, between parentheses. See section 3.8.1 page 78 about return values in lists of instructions.

The `{ /* ... /* }` define a list of instruction like a classic list, but its type is BLOCK and its evaluation is delayed (see section 3.9 page 83).

2.6.6 A loop: *do_while*

Here is a conditional loop in LISAAC :

Section Header

```
- name := DO_WHILE;
- category := APPLICATION;
```

Section Public

```
- main :=
  ( + gender:CHARACTER;

    IO.put_string "Enter your gender (M/F) : ";

    {
      IO.read_character;
      gender := IO.last_character;
    }.do_while {(gender != 'M') && {gender != 'F'}}; // conditional loop

    (gender == 'M').if {
      IO.put_string "Hello Mister !";
    } else {
      IO.put_string "Hello Miss !";
    };
  );
```

The input block is executed at least once, and continues as long as the loop condition remains true. This kind of loops, as well as others, is explained in section 3.9.2.0, page 85.

2.7 LISAAC : an object oriented language

LISAAC is an object oriented language. You can build an application using more than one object, it's what is done when you call methods on library objects. The compiler automatically link all of the needed objects to your main object (see compiler chapter for more informations).

When you run a program, only the 'master' objects (written in capital) are alive. Others are initialized with NULL and you can't use them (there will be a compiler stop).

Section Header

```
+ name := OBJECT1;
- category := APPLICATION;
```

Section Public

```
+ slot <- /* ... */
```

Section Header

```
+ name      := MAIN_OBJECT;
- category := APPLICATION;
```

Section Public

```
- main <-
( + my_object:OBJECT1;
  OBJECT1.slot;      // No problem, you use the 'master' object
  my_object.slot;    // Compiler will stop in error 'CALL ON NULL'
);
```

If you want to use an object you have to use the 'clone' operation from the 'master' object (see 2.7.1).

The 'Self' object

We call *self* the current living object. When you call a slot inside an object, it implicitly call the slot of the *self* object. The keyword **Self** can be used to explicitly call the *self* object (like "this" in Java and C++ or "Current" in Eiffel).

Section Header

```
+ name      := EXAMPLE;
- category := APPLICATION;
```

Section Public

```
+ slot_data:INTEGER := 3;

- main <-
(
  Self.slot_data.print;    // produce exactly the same code as slot_data.print;
);
```

 Note that the *self* is different between all the objects, even if they have the same type, because **Self** is an object.

2.7.1 Clone

You can clone an object to create a new object of the same type. The method **clone** is defined in the OBJECT type in the library.

The slot name must be defined with '+' if you want to clone it. As we see before, you have to use clone in order to work with an object.

Section Header

```
+ name      := OBJECT1;
- category := APPLICATION;
```

Section Public

```
+ slot <- /* ... */
```

Section Header

```
+ name      := MAIN_OBJECT;
- category := APPLICATION;
```

Section Public

```
- main <-
( + my_object:OBJECT1;
  my_object := OBJECT1.clone;
  my_object.slot; // No problem there, my_object is not Null
);
```

Example: memory representation (we don't represent slots 'set_x' and 'set_count' to simplify the example, see later the real representation)

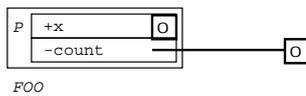
Section Header

```
+ name      := FOO;
- category := APPLICATION;
```

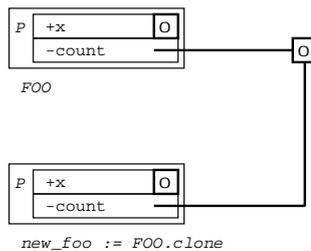
Section Public

```
+ x      :INTEGER;
- set_x v:INTEGER <- ( x := v; );

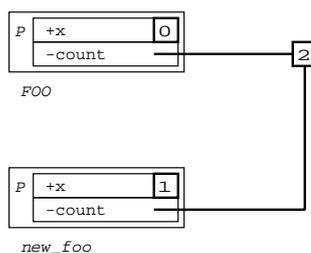
- count:INTEGER;
- set_count v:INTEGER <- ( count := v; );
```



```
new_foo := FOO.clone;
```



```
new_foo.set_x 1;
new_foo.set_count 2;
```



2.7.2 Inheritance

You can define inheritance for objects. You can define as many parents as you want. A parent is defined in a **Section Inherit** with slots following the same rules as other slots. A parent is also an object on which you can send messages. If a slot called on an object is not found in this object, the *lookup* algorithm search in the parents to find the correct slot. This algorithm do an ordered search from the first declared slot in the inheritance section.

Example: Let us see an inheritance with the parent defined with 'P'

Object FATHER

Section Header

```
+ name      := FATHER;
- category := APPLICATION;
```

Section Public

```
+ x      :INTEGER;
- inc_x <- ( x := x + 1; );
- count:INTEGER;
- inc_count <- ( count := count + 1; );
```

Object SON

Section Header

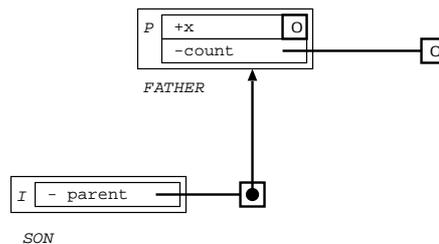
```
+ name      := SON;
- category := APPLICATION;
```

Section Inherit

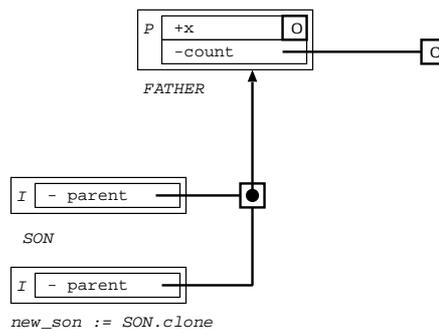
```
- parent:FATHER := FATHER;      // name of the slot doesn't matter
```

Section Public

```
- change_parent p:FATHER <- ( parent := p; );
```

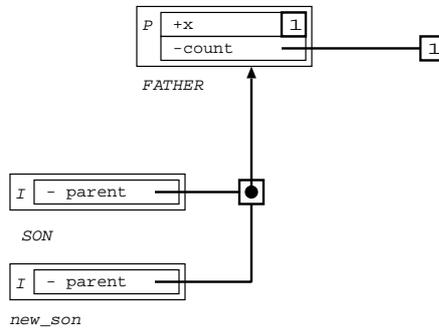


```
new_son := SON.clone;
```

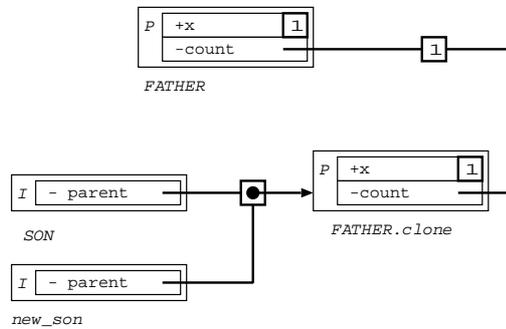


Chapter 2. Quickstart - for beginners

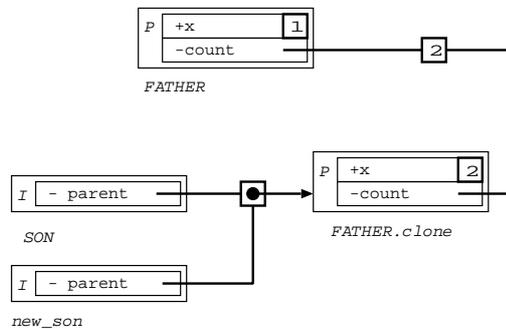
```
new_son.inc_x;
new_son.inc_count;
```



```
new_son.change_parent (FATHER.clone);
```



```
new_son.inc_x;
new_son.inc_count;
```



Example 2: Let us see the same example with the parent defined with '+'

Object FATHER

Section Header

```
+ name      := FATHER;
- category := APPLICATION;
```

Section Public

```
+ x      :INTEGER;
- inc_x  <- ( x := x + 1; );
- count :INTEGER;
- inc_count <- ( count := count + 1; );
```

Object SON

Section Header

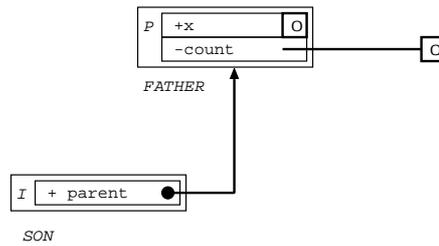
```
+ name      := SON;
- category := APPLICATION;
```

Section Inherit

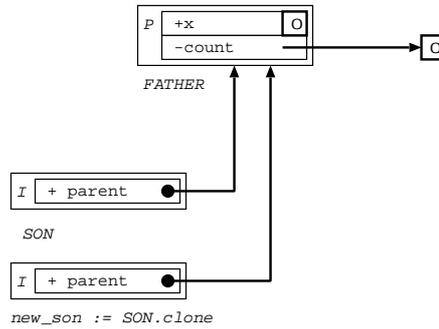
```
+ parent:FATHER := FATHER;
```

Section Public

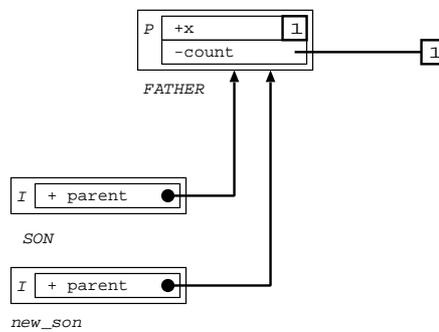
```
- change_parent p:FATHER <- ( parent := p; );
```



```
new_son := SON.clone;
```

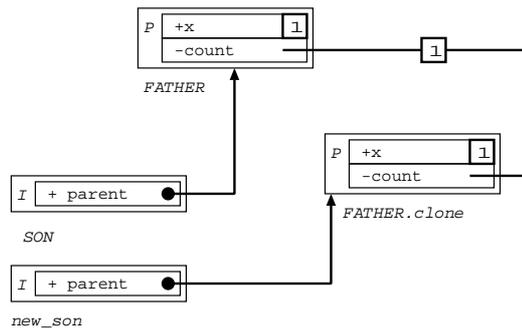


```
new_son.inc_x;
new_son.inc_count;
```

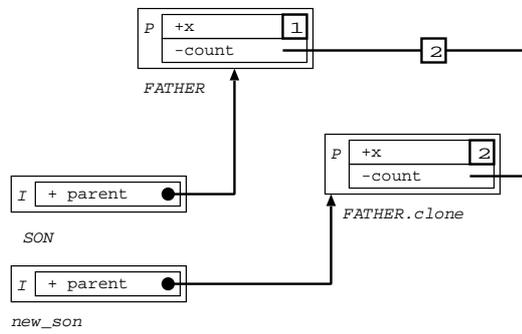


```
new_son.change_parent (FATHER.clone);
```

Chapter 2. Quickstart - for beginners



```
new_son.inc_x;  
new_son.inc_count;
```



Chapter 0011b

Language Reference

3.1 Lexical and syntax overview

Most features of the LISAAC language come from Self. Like Self, LISAAC does not have hard-coded instructions for loops or hard-coded instructions for test statements.

The following syntax of LISAAC is described using “Extended Backus-Naur Form” (EBNF). Terminal symbols are enclosed in single quotes or are written using lowercase letters. Non-terminal are written using **uppercase** letters. The following table describes the semantic of meta-symbols used:

<i>Symbol</i>	<i>Function</i>	<i>Description</i>
<code>(/* ... */)</code>	grouping	a group of syntactic constructions
<code>[/* ... */]</code>	option	an optional construction
<code>{ /* ... */ }</code>	repetition	a repetition (zero or more times)
<code> </code>	alternative	separates alternative constructions
<code>→</code>	production	separates the left and right hand sides of a production

3.1.1 Lexical overview

The following rules draws up the list of the final syntactic elements of the grammar:

<i>Symbol</i>	<i>Description</i>
<i>section</i>	section identifier
<i>identifier</i>	slot name, ...
<i>operator</i>	unary or binary operator symbol
<i>integer</i>	constant of type INTEGER
<i>cap_identifier</i>	type name: name of object or prototype
<i>characters</i>	constant of type CHARACTER
<i>string</i>	constant of type STRING
<i>external</i>	external C code
<i>affect</i>	symbol assignment slots
<i>style</i>	clone comportement
<i>type</i>	unary typing operation
<i>result</i>	result identifier value

Chapter 3. Language Reference

<i>section</i>	→ "HEADER" "INHERIT" "PRIVATE" "PUBLIC" "MAPPING" "INTERRUPT"
<i>identifier</i>	→ LOWER_CASE { (LOWER_CASE DECIMAL_DIGIT ' _ ') }
<i>operator</i>	→ OP_CHAR { OP_CHAR } <i>except affect symbol '='</i>
<i>integer</i>	→ NUMBER { "KB" "MB" "GB" }
<i>cap_identifier</i>	→ UPPER_CASE { (UPPER_CASE DECIMAL_DIGIT) ' _ ' }
<i>characters</i>	→ ''' (NORMAL_CHAR ESCAPE_CHAR) '''
<i>string</i>	→ ' " ' { NORMAL_CHAR ESCAPE_CHAR } ' " ' }
<i>external</i>	→ ' ` ' { NORMAL_CHAR ESCAPE_CHAR } ' ` ' }
<i>affect</i>	→ " : = " " < - " " ? = "
<i>style</i>	→ ' + ' ' - ' }
<i>type</i>	→ "Expanded" "Separate"
<i>result</i>	→ "Result" { ' _ ' DECIMAL_DIGIT { DECIMAL_DIGIT } }
OP_CHAR	→ ' ! ' ' @ ' ' # ' ' \$ ' ' % ' ' ^ ' ' & ' ' < ' ' ' ' * ' ' + ' ' - ' ' = ' ' ~ ' ' / ' ' ? ' ' > ' ' \ ' }
LOWER_CASE	→ ' a ' ' b ' ... ' z ' }
UPPER_CASE	→ ' A ' ' B ' ... ' Z ' }
OCTAL_DIGIT	→ ' 0 ' ' 1 ' ... ' 7 ' }
DECIMAL_DIGIT	→ ' 0 ' ' 1 ' ... ' 9 ' }
HEXA_DIGIT	→ ' 0 ' ' 1 ' ... ' 9 ' ' a ' ... ' f ' ' A ' ... ' F ' }
NORMAL_CHAR	→ <i>any character except ' \ ' and ' ' ' }</i>
ESCAPE_CHAR	→ ' \ t ' ' \ b ' ' \ n ' ' \ f ' ' \ r ' ' \ v ' ' \ a ' ' \ 0 ' ' \ \ ' ' \ ' ' ' \ " ' ' \ ? ' NUMERIC_ESCAPE
NUMERIC_ESCAPE	→ ' \ ' <i>integer</i> ' \ ' }
NUMBER	→ OCTAL_DIGIT { OCTAL_DIGIT } ' o ' DECIMAL_DIGIT { DECIMAL_DIGIT } [' d '] HEXA_DIGIT { HEXA_DIGIT } ' h ' }

Numbers

Notation of numbers: *12*, *12d*: decimal value
1BAh, *0FFh*: hexadecimal value
01010b, *10b*: binary value
14o, *6o*: octal value
10KB, *10MB*, *10GB*, *10hKB*: system facility

Characters

Notations: *'a'*, *'Z'*, *'4'*: simple character
'\n', *'\t'*, *'\r'*: escape character
'\10', *'\0Ah'*: code character

The complete list of escape sequence is:

\a : bell
\b : backspace
\f : formfeed

`\n` : newline
`\r` : carriage return
`\t` : horizontal tab
`\v` : vertical tab
`\\` : backslash

You can define a number as a string between backslashes. You can specify the type of the number (d or nothing for decimal, h for hexadecimal, o for byte, b for binary)
For example: `'\123'`, `'\123d'`, `'\4A3h'`, `'\101o'`, `'\10010110b'`.

String

A `STRING_CONSTANT` is composed by multiple characters, and can't be modified. It is defined between " ".

Notations: `"Hello World\n"`: simple string

For a better view of the source code, you can "cut" a string with the backslash character following by the character 'space', a tabulation or a Carry Return. The string will re-start on the following backslash character.

For example: `"This is \
\ an example for the \
\ string."` will be transformed by the compiler in: `"This is an example for the string"`

3.1.2 Syntax overview

In order to clarify the presentation for human reading, the following grammar of LISAAC is ambiguous. (Actually, the LISAAC parser use precedence and associativity rules to resolve ambiguities.)

PROGRAM	→	{ "Section" (<i>section</i> TYPE_LIST) { SLOT } } [CONTRACT ';']
SLOT	→	<i>style</i> ['(LOCAL)'] TYPE_SLOT [':' (TYPE '(TYPE_LIST)')] [<i>affect</i> DEF_SLOT]
TYPE_SLOT	→	<i>identifier</i> [LOC_ARG { <i>identifier</i> LOC_ARG }] '\'' <i>operator</i> '\'' [("Left" "Right") <i>integer</i>] LOC_ARG]
DEF_SLOT	→	[CONTRACT] EXPR [CONTRACT]
LOC_ARG	→	<i>identifier</i> ':' TYPE '(LOCAL)'
LOCAL	→	{ <i>identifier</i> [':' TYPE] ', ' } <i>identifier</i> ':' TYPE
TYPE_LIST	→	TYPE { ', ' TYPE }
TYPE	→	[<i>type</i>] PROTOTYPE
PROTOTYPE	→	<i>cap_identifier</i> ['[' TYPE_LIST { <i>identifier</i> TYPE_LIST }]]
EXPR	→	EXPR_PREFIX ([<i>affect</i> EXPR] { <i>operator</i> EXPR_PREFIX })
EXPR_PREFIX	→	{ <i>operator</i> } EXPR_MESSAGE
EXPR_MESSAGE	→	EXPR_BASE { '.' SEND_MSG }
EXPR_BASE	→	EXPR_PRIMARY SEND_MSG "Old" <i>identifier</i>
EXPR_PRIMARY	→	"Self" PROTOTYPE <i>result</i> <i>integer</i> <i>characters</i> <i>string</i> '(GROUP)' '{' [LOC_ARG ';'] GROUP }' <i>external</i> [':' ['(] TYPE ['(TYPE_LIST)'] [')']]
GROUP	→	DEF_LOCAL { EXPR ';' } [EXPR { ', ' { EXPR ';' } EXPR }]
CONTRACT	→	' [DEF_LOCAL { (EXPR ';' "...") }]'
DEF_LOCAL	→	{ <i>style</i> LOCAL ';' }
SEND_MSG	→	<i>identifier</i> [ARGUMENT { <i>identifier</i> ARGUMENT }]
ARGUMENT	→	EXPR_PRIMARY <i>identifier</i>

3.2 Sections identifiers

The identifier of a section makes it possible to choose the interpretation of the slots which are in this section. The interpretation of the slots relates to various aspects:

- heading and versioning information (cf. 3.2.1)
- the mode of application of the *lookup* mechanism: inheritance slot (see 3.2.2) or normal message slot
- the exception mode (see 3.2.4)
- the data structure mapping mode (see 3.2.3)
- the link with C code mode (see 3.2.5)
- the classical code section (see 3.2.6)

3.2.1 The HEADER section

The HEADER section is mandatory. It is used to enumerate the general parameters of the prototype. In this section, only the slots containing constants (character string, or numerical constants) are authorized. This section must include the **name** slot which indicates the name of the prototype itself.

Other optional slots can be added to comment on the prototype. The **category** slot indicates the category of the prototype with respect to its level of protection related to the other prototypes. There are 3 level of protection and a special level: KERNEL, DRIVER, APPLICATION and DOCILE.

- A KERNEL object can use only objects of KERNEL level.
- A DRIVER object can use objects of KERNEL or KERNEL level.
- An APPLICATION object can use objects of all levels.

A DOCILE object can be used by any other object and take the category of this object. Objects of the library are DOCILE.

In addition, some conventions on the names of the slots have been fixed for the purpose of maintenance and to ensure consistency of the information of the HEADER section.

You can't modify any slot during the execution: imagine for example the consequences of modifying the CATEGORY slot !

<i>Slot name</i>	<i>Type</i>	<i>Description</i>
'name'	PROTOTYPE	prototype's name (<i>mandatory</i>)
'category'	KERNEL, DRIVER APPLICATION,DOCILE	protection level default is APPLICATION
'version'	REAL	version number
'date'	STRING__CONSTANT	release date
'comment'	STRING__CONSTANT	Comment
'author'	STRING__CONSTANT	author's name
'bibliography'	STRING__CONSTANT	programmer's reference
'language'	STRING__CONSTANT	encoding country language
'bug_report'	STRING__CONSTANT	bugs report list
'type'	<i>external</i>	C equivalent type (if any)
'default'	EXPRESSION	Default value of the prototype (see 3.3.4)
'external'	<i>external</i>	C code which will be included in the C compiled file

Section Header

```
- name := MY_PROTOTYPE;
- category := APPLICATION;
- version := 1;
- date := "2004/06/05";
- comment := "An example";
- author := "Jerome Boutet";
- bibliography := "http://www.isaacos.com";
- language := "English";
- bug_report := "None :-)";
- type := 'unsigned long';
- default := 100;
- external := '#INCLUDE <STDIO.H>';
```

Objects and clone

There are 3 kinds of objects, defined with the slot **name**.

- Slots defined with the the `-` symbol, like

```
- name := MY_NAME;
```

are non clonable. You can only use the `MY_NAME "master"` object.

- Slots defined with the the `+` symbol, like

```
+ name := MY_NAME;
```

are clonable. You can use the `MY_NAME "master"` object and every clone of it. Be carefull, in this case the object must inherit an object containing the **clone** method (in most of the cases object OBJECT).

- Slots defined with the the `+` symbol and the **Expanded** keyword, like

```
+ name := Expanded MY_NAME;
```

are expanded ones. You don't have to clone to use the object: every object of this type is alive. In most of the cases expanded objects are simple objects, such as `INTEGER`, `CHARACTER`, `BOOLEAN`,... Usually you have to define the slot **default** and a **type** associated.

3.2.2 The INHERIT section

This section describes the inheritance slots of the object. Like in Self, a prototype can have several parent slots (multiple inheritance is allowed). The only limitation is that parents and sons must have the same `CATEGORY`. The slots of this section being mostly used by the *lookup* mechanism, only slots without arguments are authorized.

Most of the time, a slot of the `INHERIT` section refers to another prototype, by simply indicating its name. It is also possible to define a parent slot using an instruction list.

 It is not possible to define a parent slot using an instruction block, because that does not have significance.

The assignment of a parent slot may occur at any time during execution to dynamically change the ancestors of the prototype. A parent slot with no value at a given time (`NULL`) is prohibited by the *lookup* algorithm (see section 3.2.2.0 page 44).

The number of inheritance slots is fixed in the source code. Adding a new inheritance slot during the execution is not allowed in `LISAAC`.

Slots in the `INHERIT` section are not visible from outside of the object itself. Accessing a parent slot simply returns the corresponding parent object (if any).

The order in which the slots are declared is very important for the *lookup* algorithm while seeking a message. The inheritance slots are examined with respect to the order in which the source text is written, in a depth-first way, without taking into account possible conflicts (see lookup algorithm 3.2.2.0). If a slot called on an object is not found in this object, the *lookup* algorithm search in the parents to find the correct slot and return the first found.

Section Header

```
+ name      := FATHER1;
- category := APPLICATION;
```

Section Public

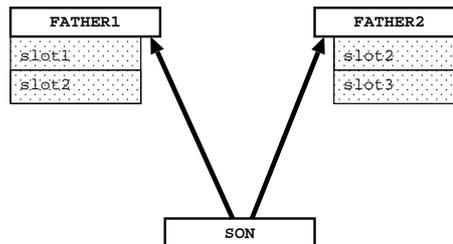
```
+ slot1 <- /* ... */
+ slot2 <- /* ... */
```

Section Header

```
+ name      := FATHER2;
- category := APPLICATION;
```

Section Public

```
+ slot2 <- /* ... */
+ slot3 <- /* ... */
```

**Section Header**

```
+ name      := SON;
- category := APPLICATION;
```

Section Inherit

```
- parent1:FATHER1 := FATHER1;
- parent2:FATHER2 := FATHER2;
```

Section Header

```
+ name      := TEST;
- category := APPLICATION;
```

Section Public

```
- main :=
( + object_son:SON;
  object_son := SON.clone;
  object_son.slot1; // From FATHER1
  object_son.slot2; // From FATHER1
  object_son.slot3; // From FATHER2
);
```

You can also redefine slots in the sons. Slot must follow the same typing profile as its parent (for parameters and result, see also 3.3.2 page 55) but you can change the kind of slot (+, - and Expanded).

Section Header

```
+ name      := FATHER1;
- category := APPLICATION;
```

Section Public

```
+ slot1 v:INTEGER :INTEGER <- /* ... */
+ slot2 <- /* ... */
```

Section Header

```
+ name      := FATHER2;
- category := APPLICATION;
```

Section Public

```
+ slot2 <- /* ... */
+ slot3 t:INTEGER <- /* ... */
```

Section Header

```
+ name      := SON;
- category := APPLICATION;
```

Section Inherit

```
- parent1:FATHER1 := FATHER1;
- parent2:FATHER2 := FATHER2;
```

Section Public

```
- slot1 v:INTEGER :INTEGER <- /* ... */ // slot1 is now shared

+ slot3 t:INTEGER <- /* ... */
```

Section Header

```
+ name      := TEST;
- category := APPLICATION;
```

Section Public

```
- main :=
( + object_son:SON;
  object_son := SON.clone;
  object_son.slot1 4.print; // From SON (redefinition)
  object_son.slot2;       // From FATHER1
  object_son.slot3 5;     // From SON (redefinition)
);
```

The name of the inheritance slots doesn't matter. We often named it "parent" but it's for more visibility, it's not a reserved keyword. But it's mandatory to precise the type of the parent, as for any data slot.

As every slot in LISAAC , inheritance slots have 3 different behaviours.

Shared inheritance

A parent can be defined with the the - symbol:

Section Inherit

```
- parent:FATHER := FATHER;
```

In this case every clone of the object share the same parent object. If a son object change its parent, every clone of this son have their parent changed.

Object FATHER

Section Header

```
+ name      := FATHER;
- category := APPLICATION;
```

Section Public

```
+ x      :INTEGER;
- inc_x  <- ( x := x + 1; );
- count :INTEGER;
- inc_count <- ( count := count + 1; );
```

Object SON

Section Header

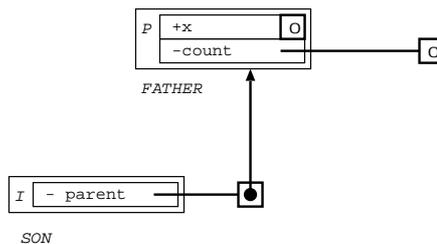
```
+ name      := SON;
- category := APPLICATION;
```

Section Inherit

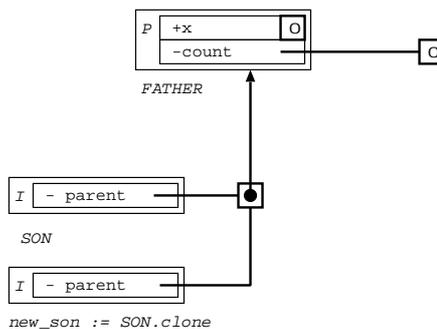
```
- parent:FATHER := FATHER;
```

Section Public

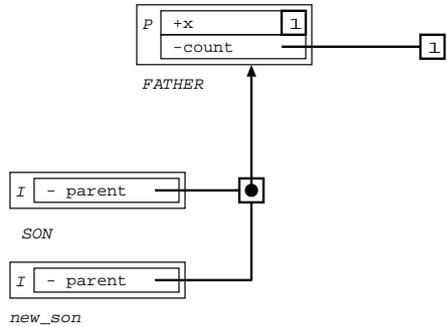
```
- change_parent p:FATHER <- ( parent := p; );
```



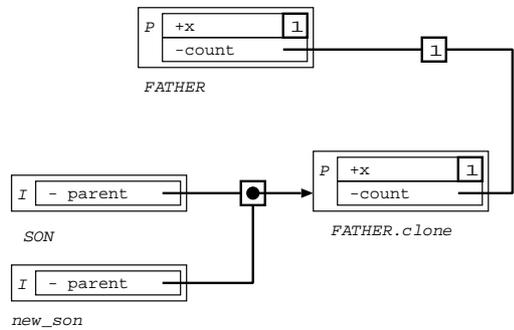
```
new_son := SON.clone;
```



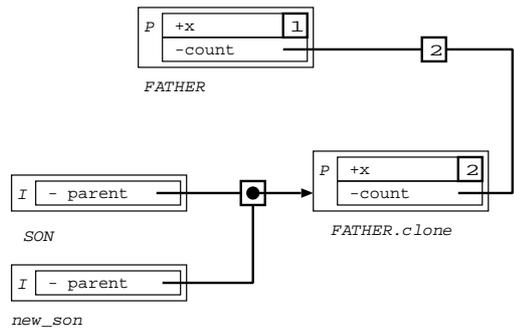
```
new_son.inc_x;
new_son.inc_count;
```



```
new_son.change_parent (FATHER.clone);
```



```
new_son.inc_x;
new_son.inc_count;
```



Non shared inheritance

A parent can be defined with the the + symbol:

Section Inherit

```
+ parent:FATHER := FATHER;
```

In this case every clone of the object share the same parent object at its creation. If a son object change its parent, other clones of this son haven't their parent changed.

Object FATHER

Section Header

```
+ name      := FATHER;
- category := APPLICATION;
```

Section Public

```
+ x      :INTEGER;
- inc_x <- ( x := x + 1; );
- count:INTEGER;
- inc_count <- ( count := count + 1; );
```

Object SON

Section Header

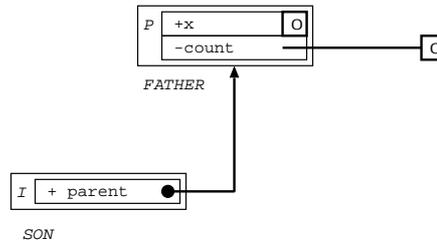
```
+ name      := SON;
- category := APPLICATION;
```

Section Inherit

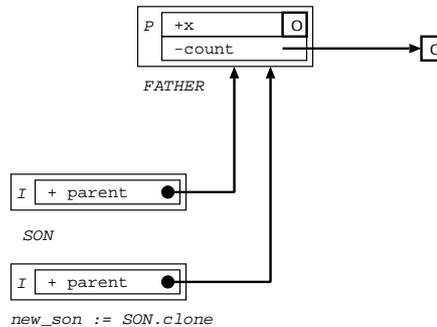
```
+ parent:FATHER := FATHER;
```

Section Public

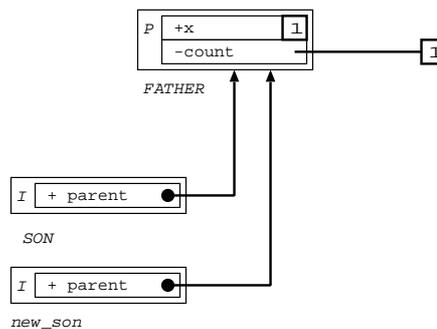
```
- change_parent p:FATHER <- ( parent := p; );
```



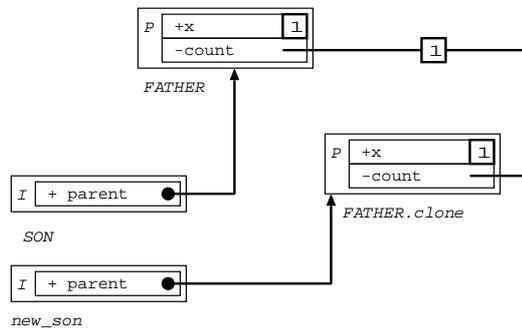
```
new_son := SON.clone;
```



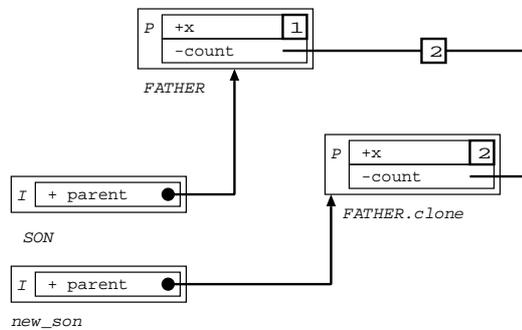
```
new_son.inc_x;
new_son.inc_count;
```



```
new_son.change_parent (FATHER.clone);
```



```
new_son.inc_x;
new_son.inc_count;
```



Expanded inheritance

A parent can be defined with the the `+` symbol and the **Expanded** keyword:

Section Inherit

```
+ parent:Expanded FATHER;
```



In this case, you don't have to affect the value of the parent.

You have an "auto-clone" for parents: each time you clone a son, you have the clone of its parents.

Object FATHER

Section Header

```
+ name := FATHER;
- category := APPLICATION;
```

Section Public

```
+ x :INTEGER;
- inc_x <- ( x := x + 1; );
- count:INTEGER;
- inc_count <- ( count := count + 1; );
```

Object SON

Section Header

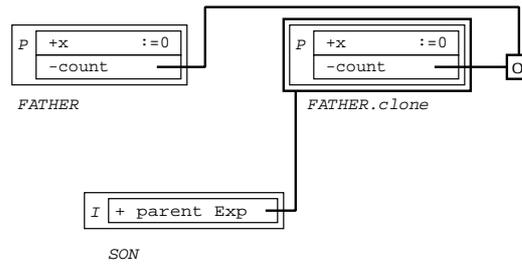
```
+ name      := SON;
- category := APPLICATION;
```

Section Inherit

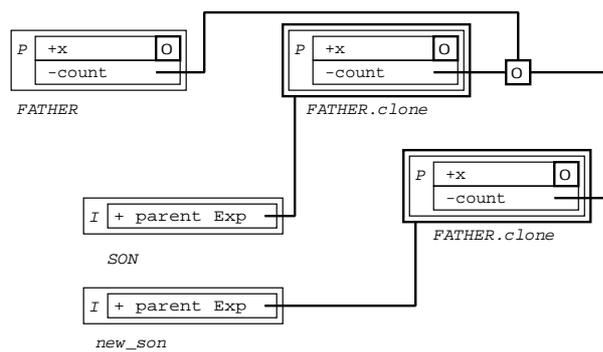
```
+ parent:Expanded FATHER;
```

Section Public

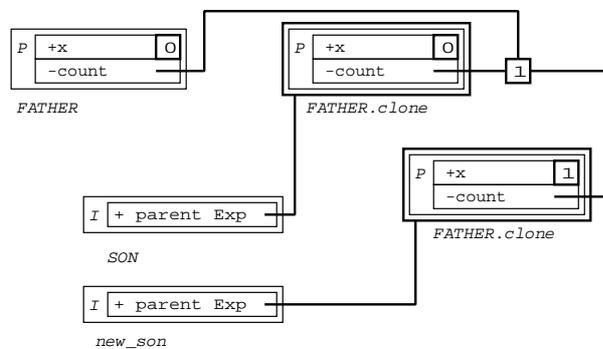
```
- change_parent p:FATHER <- ( parent := p; );
```



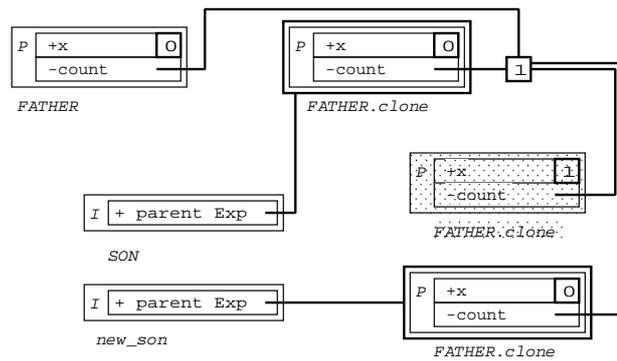
```
new_son := SON.clone;
```



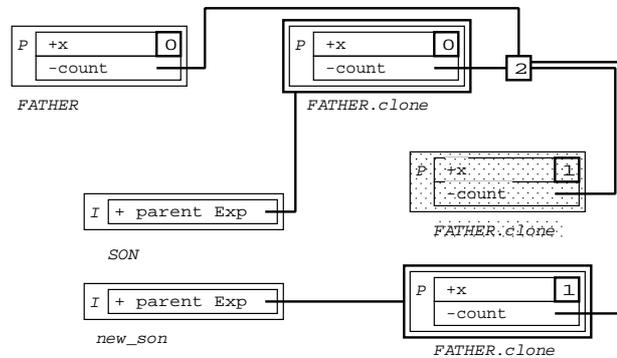
```
new_son.inc_x;
new_son.inc_count;
```



```
new_son.change_parent (FATHER.clone);
```



```
new_son.inc_x;
new_son.inc_count;
```



Note: This kind of inheritance is similar to inheritance in object oriented languages based on class.

A parent can also be defined with the the - symbol and the **Expanded** keyword. The parents are now shared.

Section Inherit

```
- parent:Expanded FATHER;
```

The value of the parent is already initialised.

Object FATHER

Section Header

```
+ name := FATHER;
- category := APPLICATION;
```

Section Public

```
+ x :INTEGER;
- inc_x <- ( x := x + 1; );
- count:INTEGER;
- inc_count <- ( count := count + 1; );
```

Object SON

Section Header

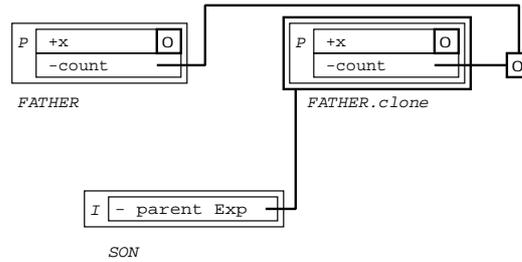
```
+ name      := SON;
- category := APPLICATION;
```

Section Inherit

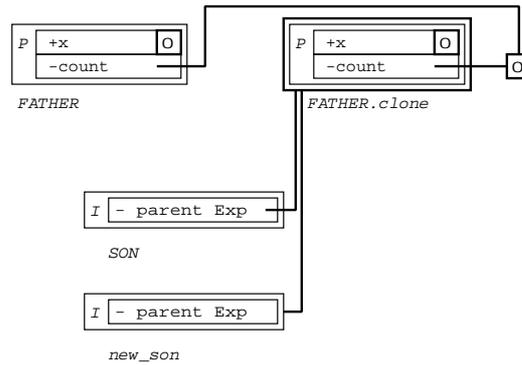
```
- parent:Expanded FATHER;
```

Section Public

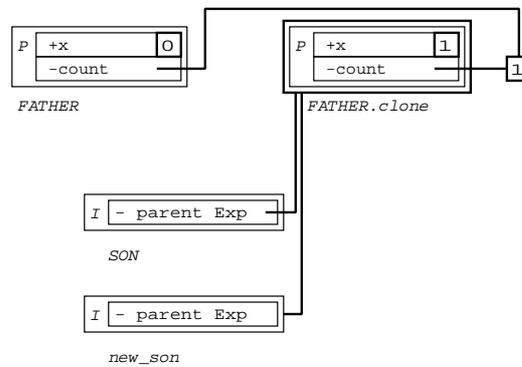
```
- change_parent p:FATHER <- ( parent := p; );
```



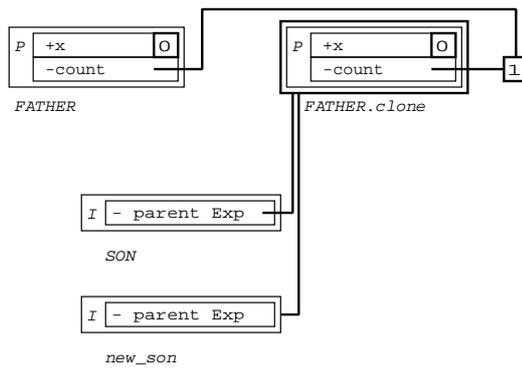
```
new_son := SON.clone;
```



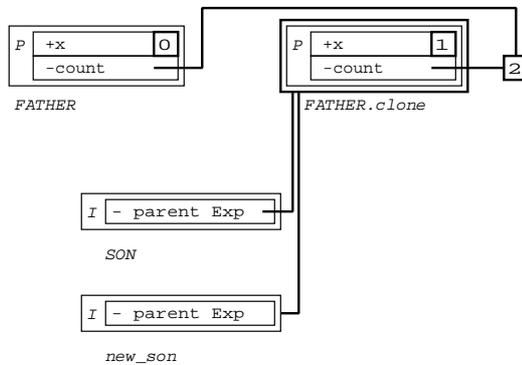
```
new_son.inc_x;
new_son.inc_count;
```



```
new_son.change_parent (FATHER.clone);
```



```
new_son.inc_x;
new_son.inc_count;
```



Immediate/delayed evaluation



The evaluation of the heritage slots depends on their order of declaration

- Evaluation after the loading of the prototype:

Section Inherit

```
+ parent:EXPR := EXPR;
```

- Evaluation every time the lookup algorithm reaches this slot (this should be avoided, because it is obviously very expensive):

Section Inherit

```
- parent:OBJECT <- search_parent;
```

Here, *search_parent* is a method to evaluate the parent.

Other example:

Section Inherit

```
- parent:OBJECT <-
( + result:OBJECT;
```

```

    (flag_depend).if {
      result := VALUE;
    } else {
      result := AFFECT;
    };
    result
  );

```

 the `flag_depend` slot must be present in the lower of the inheritance tree.

Static type and visibility of the slots

The static type of a slot parent must correspond to the first common ancestor of the parents possible dynamics.

About the visibility of the slots, the static tree of heritage shows the slots accessible.

Section Header

```

+ name := A;
- category := APPLICATION;

```

Section Public

```

+ bar <- /* ... */
+ foo <- /* ... */

```

Section Header

```

+ name := B;
- category := APPLICATION;

```

Section Inherit

```

+ parent:A := A;

```

Section Public

```

+ bar <- /* ... */
+ toto <- /* ... */

```

Section Header

```

+ name := C;
- category := APPLICATION;

```

Section Inherit

```

+ parent:A := A;

```

Section Public

```

+ titi <- /* ... */
+ foo <- /* ... */

```

Section Header

```

+ name := D;
- category := APPLICATION;

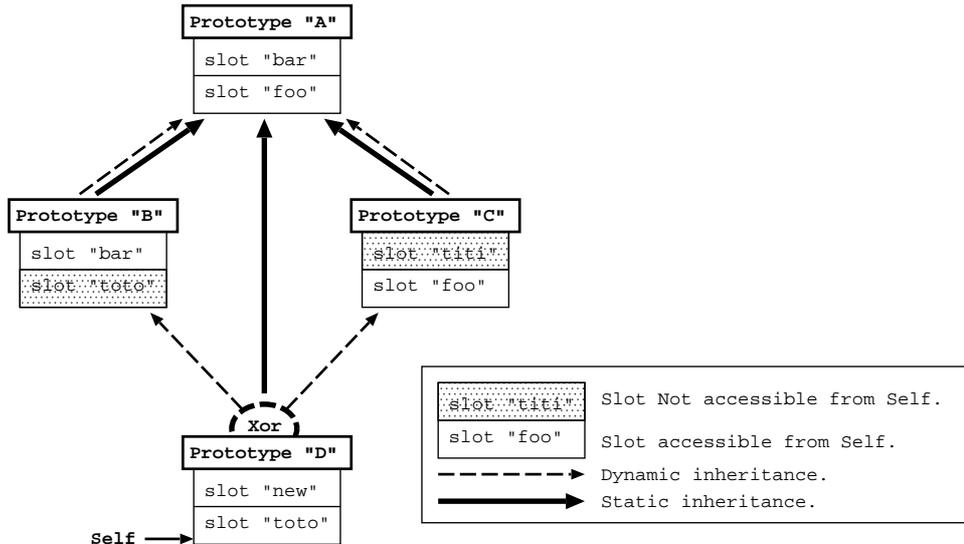
```

Section Inherit

```
+ parent:A <-
(
...          // code that can be dynamically B or C
);
```

Section Public

```
+ new <- /* ... */
+ toto <- /* ... */
```



Section Header

```
+ name := TEST;
- category := APPLICATION;
```

Section Public

```
- main :=
( + object_d:D;
  object_d := D.clone;
  object_d.bar; // Ok, from A or from B (dynamic inheritance)
  object_d.foo; // Ok, from A or from C (dynamic inheritance)
  object_d.new; // Ok, from D
  object_d.toto; // Ok, from D (redefinition in D)
  object_d.titi; // Error: slot not accessible
);
```

The lookup algorithm

The lookup algorithm is the name of the algorithm used to resolve message send (or dynamic dispatch). It determines which precise method is called.

Let *M* be the complete name of the called method, with commas or keywords, if any (see slot names in section 3.5 page 70). Let *R* be the receiver of the message send; in case the receiver is implicit, *R* is **self**. Let *T* be the dynamic type of *R*.

The lookup algorithm works as follows:

1. Look for method M in the current prototype T , searching code slots.
 Since there is no overloading in LISAAC, there should be at most one slot matching M .
 If one was found, the lookup algorithm stops, the target method has been found and the message send can proceed.
 If none was found, continue with step 2.
2. Recursively look for method M in all the parents of the current prototype T , until one is found or all parents have been examined.
 If the matching method has been found, the lookup algorithm stops, the target method has been found and the message send can proceed.
 If none was found, which indicates an error from the developer, an error message is emitted.

Note that at step 1, since there is no overloading in LISAAC, there should be at most one slot matching M . The order of declaration of code slots in T is thus irrelevant.

Conversely, the order of declaration of parent slots is highly relevant. Indeed, during step 2, parent slots are searched recursively, that is in depth-first manner. They are also examined in the order of declaration in the source code (top to bottom). As a consequence, in case of multiple inheritance, if n parent slots ($2 \leq n$) refer to prototypes that contain the searched method M , it is the M contained in the first of those n parent slots that shall be called. Thus multiple inheritance conflicts in LISAAC are solved in a (depth-first) “first come, first served” manner.

```
- lookup msg:STRING set_visited v:SET[OBJECT] :BLOCK <-
  ( + result:BLOCK;
    + i:INTEGER;

    (! v.has self).if {
      // cycle detection.
      v.add self;

      // Search in current object.
      i := list.lower;
      {(i <= list.upper) && {result = NULL}}.while_do {
        (list.item i.name == msg).if {
          // message found.
          Result := list.item i.value;
        };
        i := i + 1;
      };

      (result = NULL).if {
        // Search in parent object.
        i := parent_list.lower;
        {(i <= parent_list.upper) && {result = NULL}}.while_do {
          result := parent_list.item i.lookup msg set_visited v;
          i := i + 1;
        };
      };
    };
```

```

    };
    result
);

```

Resending messages: The equivalent of *super* in Smalltalk or *resend* in Self.

A message call applied to some parent slot is the natural mechanism to achieve the equivalent of *super* in Smalltalk or *resend* in Self. This means that the message is sent to the parent with the current object context. You can bypass the lookup algorithm by precisising the parent on which you call the slot.

Section Header

```

- name := FATHER1;
- category := APPLICATION;

```

Section Public

```

- method <- /* ... */

```

Section Header

```

- name := FATHER2;
- category := APPLICATION;

```

Section Public

```

- method <- /* ... */

```

Section Header

```

- name := SON

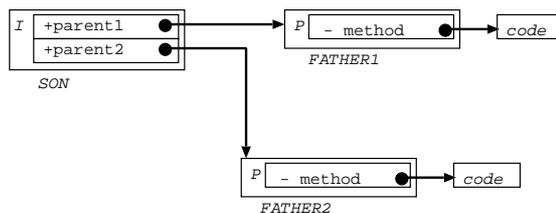
```

Section Inherit

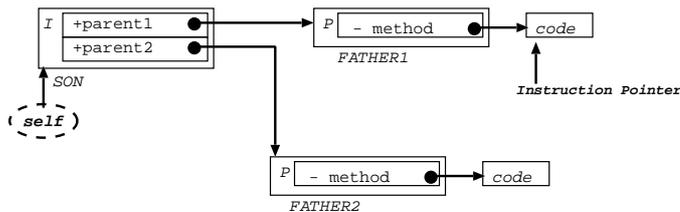
```

+ parent1 := FATHER1;
+ parent2 := FATHER2;

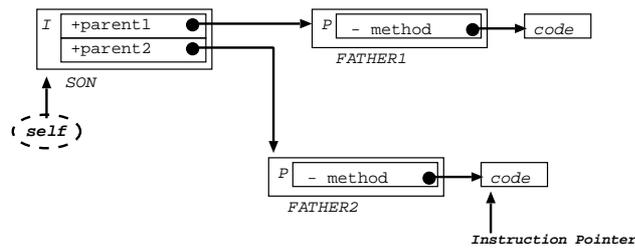
```



method;



parent2.method;



3.2.3 The MAPPING section

The MAPPING section purpose is to format data slots description according to some fixed hardware data structure.

In such a section, the compiler follows exactly the order and the description of slots as they are written to map exactly the corresponding hardware data structure.

Thus, one is able to write data slots description according to the hardware to handle.

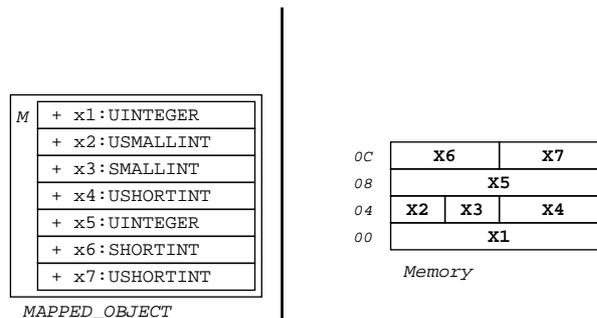
You can only define slots with the + symbol, and only datas (not code).

Otherwise, these attributes are used exactly as the others not in the mapping section (reading or writing).

Section Mapping

```
+ x1:UNTEGER;      // 4 bytes, unsigned
+ x2:USMALLINT;   // 1 byte,  unsigned
+ x3:SMALLINT;    // 1 byte,  signed
+ x4:USHORTINT;   // 2 bytes, unsigned
+ x5:UNTEGER;     // 4 bytes, unsigned
+ x6:SHORTINT;    // 2 bytes, signed
+ x7:USHORTINT;   // 2 bytes, unsigned
```

These prototype match exactly a 16 byte physical structure.



Slots inside some MAPPING section are considered private for any other objects. Slots can only be defined with the + property. No slot outside this section can be defined with the + property.

Section Mapping

```
+ x1:USHORTINT;
- x2:UNTEGER;      // Compiler will stop in error
```

Section Public

```
+ count:INTEGER := 3; // Compiler will stop in error
- slot:INTEGER <- /* ... */ // 0k
```

The mapping can also be used to represent files.

3.2.4 The INTERRUPT section

The goal of the INTERRUPT section is to handle hardware interruptions. In this section you can define methods (code slots) that will be executed only while there is an interrupt associated.

Each slot is associated with one of the processor’s interruptions [Hum90].

These slots differ from others in their generated code. For example, their entry and exit codes are related to the interrupt processing.

Their invocations are asynchronous and borrow the quantum of the current process.

Generally, these slots are little time consumers and they don’t require specific process’ context for their executions.

It is thus necessary to be careful while programming such slots to ensure the consistency of the interrupted process.

Define your method (without return value, because you don’t explicitly call it) as any other classical method.

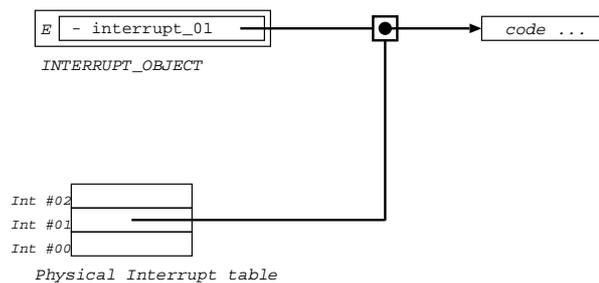
Then associate the adress of your method with the effective interrupt jump adress (it depends of your architecture). This can be done using a system mapped object.

When your interrupt physically happens, there is the call of your associated method, which returns a pointer on the code.

The compiler will not optimize local variables of your interrupt method because of its particularity: the call depends of the context and cannot be anticipated during compilation.

Section Interrupt

```
- interrupt_01 <- /* ... */;
```



You must define as C external (3.12 page 93) the following macros: `__BEGIN_INTERRUPT__` and `__END_INTERRUPT__`. These macros will be executed every time an interrupt function is activated. The code of these macros depends on the architecture. Example for X86 follows.

Section Header

```
- name~~~~~:= INTERRUPT_MANAGER;
- category~~~:= KERNEL;
- external := ‘
```

```

#define __BEGIN_INTERRUPT__ volatile unsigned long eax;
    volatile unsigned long ebx;
    volatile unsigned long ecx;
    volatile unsigned long edx;
    volatile unsigned long esi;
    volatile unsigned long edi;

    asm volatile (
        /* BEGIN INTERRUPT */
        movl %%eax,%0
        movl %%ebx,%1
        movl %%ecx,%2
        movl %%edx,%3
        movl %%esi,%4
        movl %%edi,%5
        /* BEGIN CODE */
        : "=m"(eax), "=m"(ebx), "=m"(ecx), "=m"(edx), "=m"(esi), "=m"(edi)
        : /* no input */
        : "eax", "edx", "ecx", "ebx", "ebp", "esi", "edi", "memory");

#define __END_INTERRUPT__ asm volatile (
    /* END CODE */
    movl %0,%%eax
    movl %1,%%ebx
    movl %2,%%ecx
    movl %3,%%edx
    movl %4,%%esi
    movl %5,%%edi
    movl %%ebp,%%esp
    popl %%ebp
    iret
    /* END INTERRUPT */
    : /* no output */
    : "m"(eax), "m"(ebx), "m"(ecx), "m"(edx), "m"(esi), "m"(edi)
    : "eax", "edx", "ecx", "ebx", "ebp", "esi", "edi", "memory");

';

```

3.2.5 The EXTERNAL section

When a slot is define in LISAAC , its real name (the name of the slot after compilation) is different in the produced C code because of the compiler (optimization, specialization, ...).

You can define a special section, **Section External**, which specified that the function here define must keep their name after compilation. This capability is very useful when you want to link the produced C code with existing code.

This section is more detailed in section 3.12 page 93.

3.2.6 Other sections

Other sections shared the same objective: they all are section of code and datas. The difference between these sections are only the visibility of their slot (method and datas). There is 3 kind of sections of this type: the **Private** section, the **Public** section and the *prototype list* section.

Section Private

It's the most restrictive section. The slots defined in it are only accessibles inside the current object (the *self* object) but not for its descendants.

Section Interrupt, **Section Mapping** and **Section Inherit** are considered **Private**.

Section SELF

The slots defined in it are only accessible inside the current object but also for its descendants. Note that its the keyword SELF is written in capital, which is a different as other keywords.

Section *prototype list*

This section is defined with the keyword **Section** followed by a list of prototypes (in capital, separated by ,) which are allowed to call the slots (example: **Section** INTEGER,BOOLEAN,STRING). The *self* object have also the right to call it.

Section Public

It's the most permissive section. The slots defined in it are accessibles from all the objects.

You can define as many sections as you want.

Section Header

```
+ name := FIRST;  
- category := APPLICATION;
```

Section Private

```
+ slot_private <- /* ... */
```

Section SELF

```
+ slot_self <- /* ... */
```

Section FIRST

```
+ slot_list1 <- /* ... */
```

Section FIRST,SECOND

```
+ slot_list2 <- /* ... */
```

Section Public

```
+ slot_public <- /* ... */
```

<i>Object</i>	private	self	list1	list2	public
<i>self</i> object only Not its descendants	OK	OK	OK	OK	OK
<i>self</i> object and its descendants	X	OK	OK	OK	OK
Type FIRST ('master' object FIRST all its clones and descendants)	X	X	OK	OK	OK
Type SECOND ('master' object SECOND all its clones and descendants)	X	X	OK	OK	OK
Any type except FIRST and SECOND	X	X	X	X	OK

Examine this example in details:

Section Header

```
+ name := FIRST;
- category := APPLICATION;
```

Section Private

```
+ slot_private <- /* ... */
```

Section SELF

```
+ slot_self <- /* ... */
```

Section FIRST

```
+ slot_list1 <- /* ... */
```

Section FIRST,SECOND

```
+ slot_list2 <- /* ... */
```

Section Public

```
+ slot_public <- /* ... */
```

```
+ slot_test <-
(
  slot_private;      // Allowed
  slot_self;        // Allowed
  slot_list1;       // Allowed
  slot_list2;       // Allowed
  slot_public;      // Allowed
);
```

```
+ slot_test2 <-
(+ object_first:FIRST;
  object_first := FIRST.clone;
  object_first.slot_private; // Forbidden
  object_first.slot_self;   // Forbidden
  object_first.slot_list1;  // Allowed
  object_first.slot_list2;  // Allowed
  object_first.public;     // Allowed
);
```

Section Header

```
+ name := SECOND;
- category := APPLICATION;
```

Section Public

```
+ slot_test <-
( + object_first:FIRST;
  object_first := FIRST.clone;
  object_first.slot_private; // Forbidden
  object_first.slot_self; // Forbidden
  object_first.slot_list1; // Forbidden
  object_first.slot_list2; // Allowed
  object_first.public; // Allowed
);
```

Section Header

```
+ name := OTHER;
- category := APPLICATION;
```

Section Public

```
+ slot_test <-
( + object_first:FIRST;
  object_first := FIRST.clone;
  object_first.slot_private; // Forbidden
  object_first.slot_self; // Forbidden
  object_first.slot_list1; // Forbidden
  object_first.slot_list2; // Forbidden
  object_first.public; // Allowed
);
```



The call of a slot in **Section Private** or **Section SELF** is restricted to the implicit call. You can't use the **Self** object.

Section Header

```
+ name := FIRST;
- category := APPLICATION;
```

Section Private

```
+ slot_private <- /* ... */
```

Section SELF

```
+ slot_self <- /* ... */
```

Section Public

```
+ slot_test <-
(
  slot_private; // Allowed
  slot_self; // Allowed
  self.slot_private; // Forbidden
  self.slot_self; // Forbidden
);
```

Accessibility and inheritance

Inheritance share the same accessibility between parents and sons. For example, if a slot is defined in a **Public** section in a parent, it is also **Public** for its descendants. Note that a **Private** slot is not visible from the descendants. If you define a visibility for a prototype, it is also available for its descendants. Look at the accessibility as if the considered slot was effectively in the current object and not in its parents.

Section Header

```
+ name := FATHER;
- category := APPLICATION;
```

Section Private

```
+ slot_private <- /* ... */
```

Section SELF

```
+ slot_self <- /* ... */
```

Section FATHER

```
+ slot_list1 <- /* ... */
```

Section FIRST

```
+ slot_list2 <- /* ... */
```

Section Header

```
+ name := SON;
- category := APPLICATION;
```

Section Inherit

```
+ parent:FATHER := FATHER;
```

Section Public

```
+ slot_test <-
(
  slot_private;           // Forbidden
  slot_self;             // Allowed
  slot_list1;            // Allowed (FATHER and all its descendants)
  slot_list2;            // Forbidden
);
```

```
+ slot_test2 <-
( + object_son:SON;
  object_son := SON.clone;
  object_son.slot_private; // Forbidden
  object_son.slot_self;    // Forbidden
  object_son.slot_list1;   // Allowed
  object_son.slot_list2;   // Forbidden
);
```

Section Header

```
+ name := FIRST;
- category := APPLICATION;
```

Section Public

```

+ slot_test <-
( + object_son:SON;
  object_son := SON.clone;
  object_son.slot_private;      // Forbidden
  object_son.slot_self;        // Forbidden
  object_son.slot_list1;       // Forbidden
  object_son.slot_list2;       // Allowed
);

```

Accessibility restricted to a prototype is also valid for its descendants. In the previous example, call on slot_list2 is allowed in all the objects of FIRST type and for all its descendants.

Section Header

```

+ name := SON_FIRST;
- category := APPLICATION;

```

Section Inherit

```

+ parent:FIRST := FIRST;

```

Section Public

```

+ slot_test <-
( + object_son:SON;
  object_son := SON.clone;
  object_son.slot_private;      // Forbidden
  object_son.slot_self;        // Forbidden
  object_son.slot_list1;       // Forbidden
  object_son.slot_list2;       // Allowed
);

```

You must also keep the same accesibility type when you redefine a slot in a son.

Section Header

```

+ name := FATHER;
- category := APPLICATION;

```

Section Private

```

+ slot_private <- /* ... */

```

Section FIRST

```

+ slot_list1 <- /* ... */

```

Section Header

```

+ name := SON;
- category := APPLICATION;

```

Section Inherit

```

+ parent:FATHER := FATHER;

```

Section Private

```

+ slot_private <- /* ... */ // Ok, it respects the same accessibility

```

Section PUBLIC

```

+ slot_list1 <- /* ... */ // Error: accessibility is different
//                          between FATHER and SON

```

3.3 Type names

Type names are noted with prototype names. A keyword in uppercase (capital letter) identify them.

```
+ color:INTEGER;
```

3.3.1 Genericity

To ease the implementation of containers like arrays, linked lists and dictionaries for example, we also added a form of genericity (parametric types) such as the one defined in Eiffel [Mey94].

```
+ array:ARRAY[CHARACTER];
```

To define such a prototype using genericity, you'll define between '[' and ']' the abstract types used, separated by commas ',' or by keywords. In the definitions of slots, you can use your abstract type

Section Header

```
+ name := GENERICITY_EXAMPLE[E,F];
- category := APPLICATION;
```

Section Public

```
- slot:F <-
( + elt:E;
  /* ... */
);
```



The name of the prototype is the entire name, with '[' and ']'.

Section Header

```
+ name := TEST;
- category := APPLICATION;
```

Section Public

```
- slot <-
( + gen:GENERICITY_EXAMPLE;      // Error: the type does not exist
  + gen2:GENERICITY_EXAMPLE[STRING,INTEGER]; // OK
  /* ... */
);
```

Note that when you use the genericity-prototype, you have to precise the real types you want.

3.3.2 Invariant's type control

The redefinition of a slot must have the same profile as her parent (standard type and name for the arguments and the return value).

Section Header

```
- name := FATHER;
- category := APPLICATION;
```

Section Public

```
+ to_string arg:INTEGER :STRING <- /* ... */
```

Section Header

```
- name := SON;  
- category := APPLICATION;
```

Section Inherit

```
- parent:FATHER:= FATHER;
```

Section Public

```
+ to_string arg:INTEGER :STRING <- /* ... */ // Ok, follow the same profile
```

Section Header

```
- name := SON;  
- category := APPLICATION;
```

Section Inherit

```
- parent:FATHER := FATHER;
```

Section Public

```
+ to_string arg:REAL :ARRAY[CHARACTER] <- /* ... */ // Error: not the same profile
```

3.3.3 Particular type: SELF type

The type SELF represents a prototype which is exactly the same type as the current prototype.

Section Header

```
- name := EXAMPLE;  
- category := APPLICATION;
```

Section Public

```
+ slot:SELF <- ( /* ... */ );
```

Here the SELF type is exactly EXAMPLE.

Another example using inheritance:

Section Header

```
+ name := FATHER;  
- category := APPLICATION;
```

Section Public

```
- create:SELF <-  
( + result:SELF;  
  result := SELF.clone;  
  result  
);
```

Section Header

```
+ name := SON;  
- category := APPLICATION;
```

Section Inherit

```
- parent:FATHER := FATHER;
```

Section Header

```
- name := TEST;
- category := APPLICATION;
```

Section Public

```
- main:=
( + object_father:FATHER;
  + object_son:SON;
  object_father := FATHER.create; // Type FATHER
  object_son := SON.create;      // Type SON
);
```

We can see with this last example that even if the slot who returns SELF type is defined in a parent, it's the current object who define the real type of SELF.



SELF type is available only if the result is calculated. You can't write

```
- slot:SELF;
```

Because if you have inheritance and the slot SELF in the parent, in the children the type is different.

Section Header

```
+ name := FATHER;
- category := APPLICATION;
```

Section Public

```
- a:SELF;
```

Section Header

```
+ name := SON1;
- category := APPLICATION;
```

Section Inherit

```
- parent:FATHER := FATHER;
```

Section Public

```
- affect_a <- ( a := SELF; ); // Here SELF is SON1
```

Section Header

```
+ name := SON2;
- category := APPLICATION;
```

Section Inherit

```
- parent:FATHER := FATHER;
```

Section Public

```
- affect_a <- ( a := SELF; ); // Here SELF is SON1
```

Section Header

```
- name := TEST;
```

```
- category := APPLICATION;
Section Public
- main :=
( + object_son1:SON1;
  + object_son2:SON2;
  object_son1 := SON1.clone;
  object_son2 := SON2.clone;
  object_son1.affect_a; // Ok
  object_son2.affect_a; // Error of typing, a is type SON1 and can't be then SON2
);
```

3.3.4 Default value of a slot according to its type.

A default value can also be defined in the slot **default** in the **Section Header**. It can be a value or an expression evaluated at initialisation of the slot or the local slot (at start of execution of the method).

Section Header

```
- name := EXAMPLE;
- category := APPLICATION;
- default := NULL;
```

If you use the prototype without initializing it, its value will be NULL.

3.4 Slots

3.4.1 Shared slots

If the slot is preceded by the - character, its value is shared between all the clones of the prototype (global slot).

Overview

Section Header

```
+ name := FOO;
- category := APPLICATION;
```

Section Public

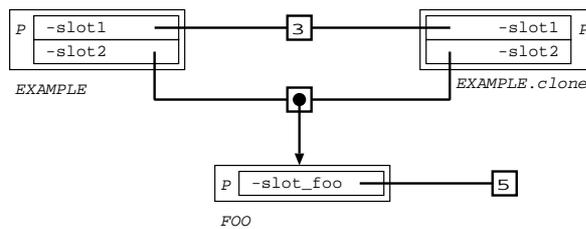
```
- slot_foo:INTEGER := 5;
```

Section Header

```
+ name := EXAMPLE;
- category := APPLICATION;
```

Section Public

```
- slot1:INTEGER := 3;
- slot2:FOO := FOO;
```



The difference between **slot1** and **slot2** is that **INTEGER** is **Expanded**. We will see this in section 3.4.3 page 64.

Note that the 2 objects shared the same pointer on the FOO object. So if you change the pointer, it changes for all the clones.

Non expanded objects have their default value set to NULL.

Section Header

```
+ name := FOO;
- category := APPLICATION;
```

Section Public

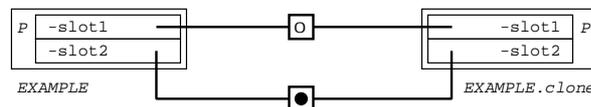
```
- slot_foo:INTEGER := 5;
```

Section Header

```
+ name := EXAMPLE;
- category := APPLICATION;
```

Section Public

```
- slot1:INTEGER;
- slot2:FOO;
```



Assignment

Section Header

```
+ name := FOO;
- category := APPLICATION;
```

Section Public

```
- slot_foo:INTEGER := 5;
```

Section Header

```
+ name := EXAMPLE;
- category := APPLICATION;
```

Section Public

```
- slot1:INTEGER;
- slot2:FOO;
- inc_slot1 <- ( slot1 := slot1 + 1; );
```

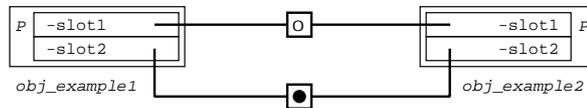
```
- set_slot2 f:FOO <- ( slot2 := f; );
```

Section Header

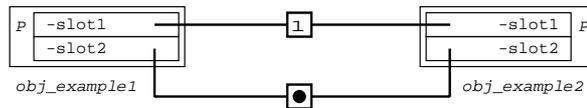
```
+ name := TEST;
- category := APPLICATION;
```

Section Public

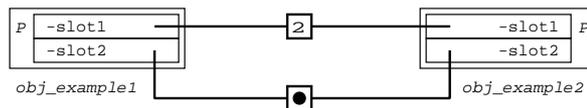
```
- main :=
( + obj_example1,obj_example2:EXAMPLE;
  + obj_foo1,obj_foo2:FOO;
  obj_example1 := EXAMPLE.clone;
  obj_example2 := EXAMPLE.clone;
```



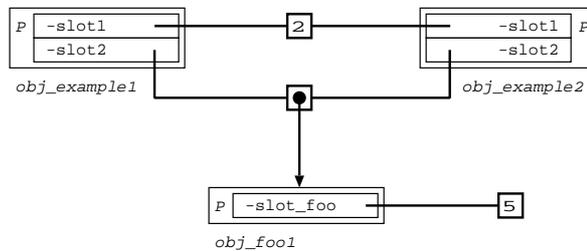
```
obj_example1.inc_slot1;
```



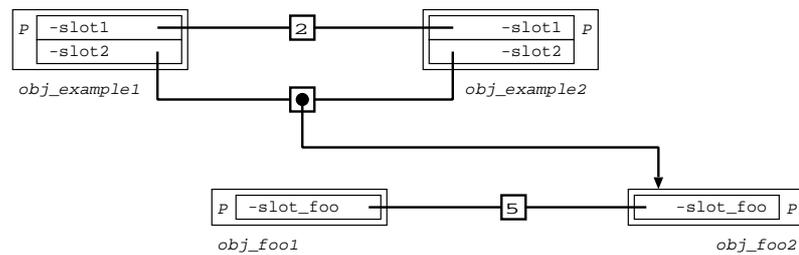
```
obj_example2.inc_slot1;
```



```
obj_foo1 := FOO.clone;
obj_example1.set_slot2 obj_foo1;
```



```
obj_foo2 := FOO.clone;
obj_example2.set_slot2 obj_foo2;
```



You can assign an object only with an object of the same type of its descendants.

Section Header

```
+ name := FATHER;
- category := APPLICATION;
```

Section Header

```
+ name := SON;
- category := APPLICATION;
```

Section Inherit

```
- parent:FATHER := FATHER;
```

Section Header

```
+ name := OBJECT_OTHER;
- category := APPLICATION;
```

Section Header

```
+ name := TEST;
- category := APPLICATION;
```

Section Private

```
- slot:FATHER;
```

Section Public

```
- main :=
( + s:SON;
  + o:OBJECT_OTHER;
  o := OBJECT_OTHER.clone;
  slot := o; // Error: not the same type
  s := SON.clone;
  slot := s; // Ok: descendant of FATHER
);
```



For Expanded types, you must match exactly the same type (see 3.4.3 page 64).

3.4.2 Non shared slots

If the slot is preceded by the + character, its value is not shared between all the clones of the prototype.

Overview

Section Header

```
+ name := FOO;
- category := APPLICATION;
```

Section Public

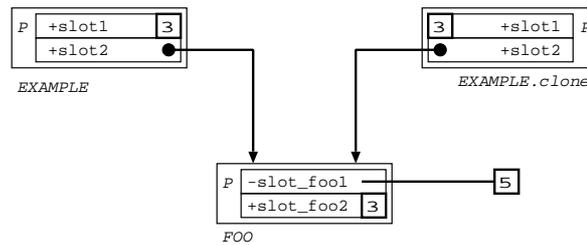
```
- slot_foo:INTEGER := 5;
```

Section Header

```
+ name := EXAMPLE;
- category := APPLICATION;
```

Section Public

```
+ slot1:INTEGER := 3;
+ slot2:FOO := FOO;
```



The difference between **slot1** and **slot2** is that INTEGER is **Expanded**. We will see this in section 3.4.3 page 64.



You can think that the 2 objects shared the same object FOO. It's false. They have each other their own pointer on the same object, which is very different. The pointers refers to the same object FOO because of its initialization. Examples come to illustrate this.

Non expanded objects have their default value set to NULL.

Section Header

```
+ name := FOO;
- category := APPLICATION;
```

Section Public

```
- slot_foo1:INTEGER := 5;
+ slot_foo2:INTEGER := 3;
```

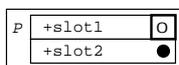
Section Header

```
+ name := EXAMPLE;
```

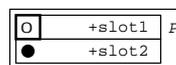
```

- category := APPLICATION;
Section Public
+ slot1:INTEGER;
+ slot2:FOO;

```



EXAMPLE



EXAMPLE.clone

Assignment

Section Header

```

+ name := FOO;
- category := APPLICATION;

```

Section Public

```

- slot_foo1:INTEGER := 5;
+ slot_foo2:INTEGER := 3;

```

Section Header

```

+ name := EXAMPLE;
- category := APPLICATION;

```

Section Public

```

+ slot1:INTEGER;
+ slot2:FOO;
- inc_slot1 <- ( slot1 := slot1 + 1; );
- set_slot2 f:FOO <- (slot2 := f; );

```

Section Header

```

+ name := TEST;
- category := APPLICATION;

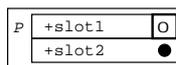
```

Section Public

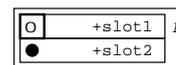
```

- main :=
( + obj_example1,obj_example2:EXAMPLE;
+ obj_foo1,obj_foo2:FOO;
obj_example1 := EXAMPLE.clone;
obj_example2 := EXAMPLE.clone;

```



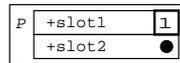
obj_example1



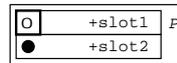
obj_example2

```
obj_example1.inc_slot1;
```

Chapter 3. Language Reference

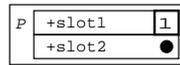


obj_example1

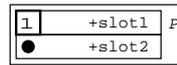


obj_example2

```
obj_example2.inc_slot1;
```

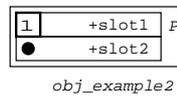
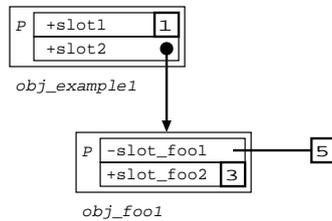


obj_example1



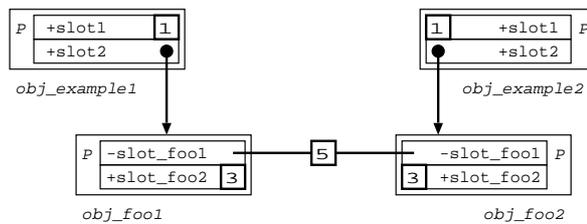
obj_example2

```
obj_foo1 := FOO.clone;
obj_example1.set_slot2 obj_foo1;
```



obj_example2

```
obj_foo2 := FOO.clone;
obj_example2.set_slot2 obj_foo2;
```



3.4.3 Expanded slots

If the slots use the keyword **Expanded**, its value is cloned and embedded (in memory) in the prototype. The keyword can be used either with **+** or **-**.

Overview

Let's first see with Sharable slots.

Section Header

```
+ name := FOO;
```

```
- category := APPLICATION;
```

Section Public

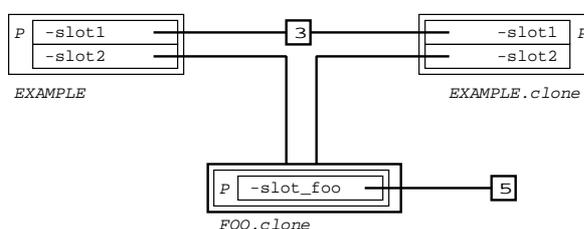
```
- slot_foo1:INTEGER := 5;
+ slot_foo2:INTEGER := 4;
```

Section Header

```
+ name := EXAMPLE;
- category := APPLICATION;
```

Section Public

```
- slot1:Expanded INTEGER := 3;
- slot2:Expanded FOO;
```



If the object is already EXPANDED, the use of the keyword **Expanded** for the slot don't change anything. It's why for **slot1** there is no difference with the non Expanded and Sharable slot (3.4.1) (INTEGER is already EXPANDED). Note that you don't have to initialise a slot with an **Expanded** object, it is already cloned and have their default value. This is a major difference with non Expanded slots.

It's the same thing to define an EXPANDED object and assign it with a slot as defining a non EXPANDED object and assign it with an EXPANDED slot.

Section Header

```
+ name := Expanded FOO;
- category := APPLICATION;
```

Section Public

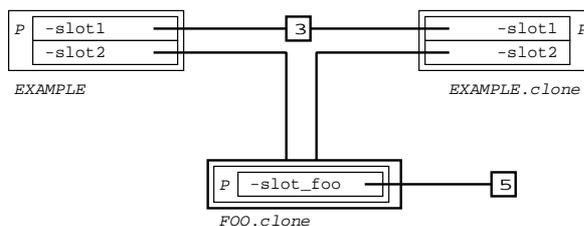
```
- slot_foo1:INTEGER := 5;
+ slot_foo2:INTEGER := 4;
```

Section Header

```
+ name := EXAMPLE;
- category := APPLICATION;
```

Section Public

```
- slot1:INTEGER := 3;
- slot2:FOO;
```



Let's now see with Non Sharable slots.

Section Header

```
+ name := FOO;
- category := APPLICATION;
```

Section Public

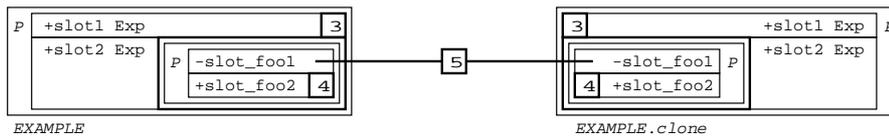
```
- slot_foo1:INTEGER := 5;
+ slot_foo2:INTEGER := 4;
```

Section Header

```
+ name := EXAMPLE;
- category := APPLICATION;
```

Section Public

```
+ slot1:INTEGER := 3;
+ slot2:Expanded FOO;
```



The object `FOO` is directly embedded in the `EXAMPLE` object.

Assignment



You can assign Expanded objects only with objects of exactly the same type (not descendants, which defers from non Expanded objects). See also 3.7.1 page 75.

Section Header

```
+ name := FATHER;
- category := APPLICATION;
```

Section Header

```
+ name := SON;
- category := APPLICATION;
```

Section Inherit

```
- parent:FATHER := FATHER;
```

Section Header

```
+ name := TEST;
- category := APPLICATION;
```

Section Private

```
+ slot:Expanded FATHER; // slot value is not Null by default (clone of FATHER)
```

Section Public

```

- main :=
( + f:FATHER;
  + f_exp:Expanded FATHER;
  + s:SON;
  + s_exp:Expanded SON;
  slot := f_exp;           // Ok, the 2 types are exactly the same
  f := FATHER.clone; // f value is NULL by default
  slot := f;              // f is copied into slot
  slot := s_exp;         // Error: not of the same type
                          // (even it inherits from FATHER)

  s := SON.clone;
  slot := s;             // Error: s is not of the same type
  slot := FATHER.clone; // FATHER.clone is copied into slot
);

```

To explain all this restrictions, remember that an Expanded object is embedded in another. So you can replace it only by an object of the same size (in terms of memory).

Let see an example of assignment. It's very important to notice that if you have a slot with an Expanded parameter, this parameter is passed by copy.

Section Header

```

+ name := FOO;
- category := APPLICATION;

```

Section Public

```

- slot_foo1:INTEGER := 5;
- inc_foo1 <- ( slot_foo1 := slot_foo1 + 1; );
+ slot_foo2:INTEGER := 4;

- inc_foo2 <- ( slot_foo2 := slot_foo2 + 1; );

```

Section Header

```

+ name := EXAMPLE;
- category := APPLICATION;

```

Section Public

```

+ slot1:Expanded FOO;
- slot2:Expanded FOO;
- set_slot1 f:Expanded FOO <- ( slot1 := f; ); // argument must be Expanded
- set_slot2 f:Expanded FOO <- ( slot2 := f; );

```

Section Header

```

+ name := TEST;
- category := APPLICATION;

```

Section Public

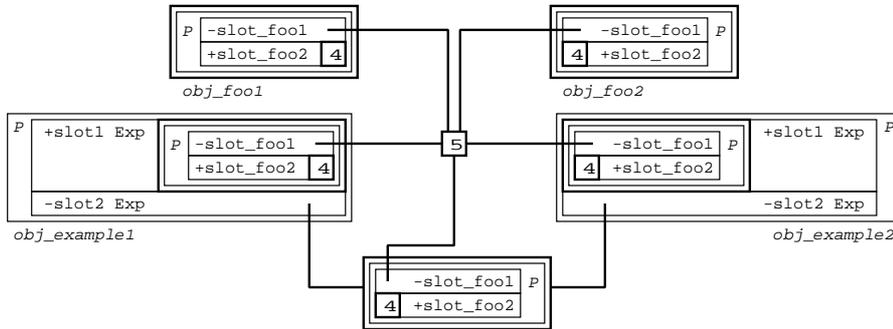
```

- main :=
( + obj_example1,obj_example2:EXAMPLE;
  + obj_foo1,obj_foo2:Expanded FOO;
  obj_example1 := EXAMPLE.clone;
);

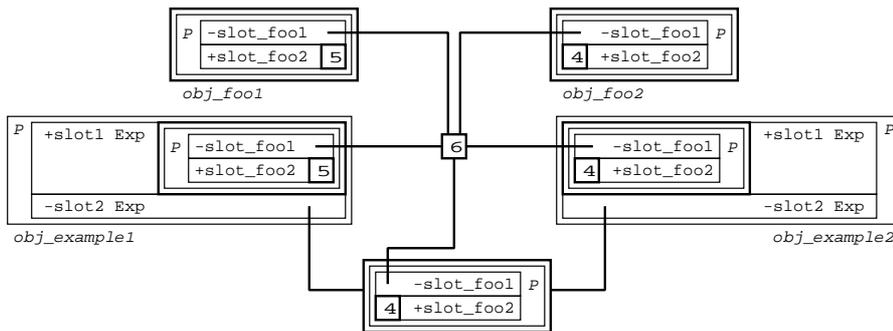
```

Chapter 3. Language Reference

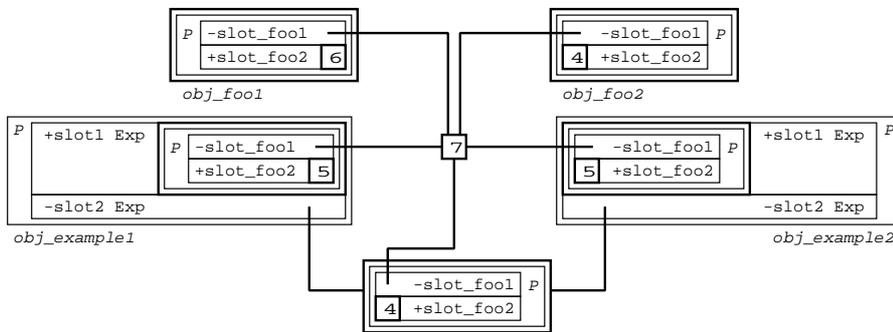
```
obj_example2 := EXAMPLE.clone;
```



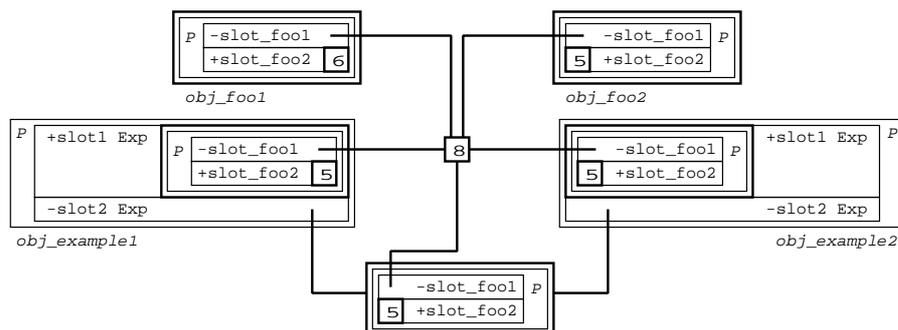
```
obj_foo1.inc_foo1;
obj_foo1.inc_foo2;
obj_example1.set_slot1 obj_foo1; // obj_foo1 cloned when passed through SET_SLOT1
```



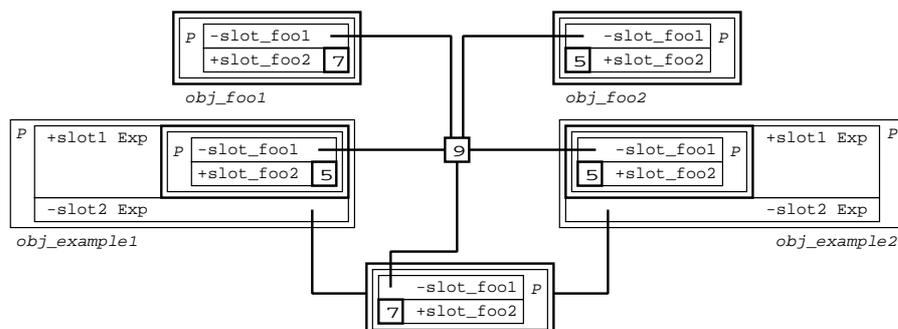
```
obj_foo1.inc_foo1;
obj_foo1.inc_foo2;
obj_example2.set_slot1 obj_foo1;
```



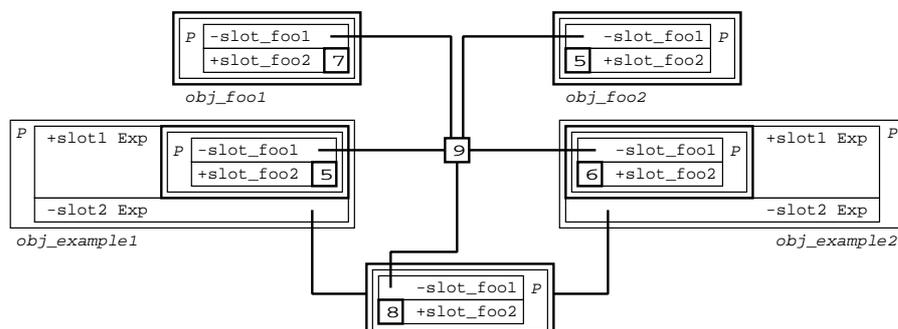
```
obj_foo2.inc_foo1;
obj_foo2.inc_foo2;
obj_example2.set_slot2 obj_foo2;
```



```
obj_foo1.inc_foo1;
obj_foo1.inc_foo2;
obj_example1.set_slot2 obj_foo1;
```



```
obj_example1.slot2.inc_foo2;
obj_example2.slot1.inc_foo2;
```



⚠ The object who will be use with the **Expanded** keyword must be defined clonable (use of + in the name of the prototype).

3.5 Slot descriptors

An object may hold any number of slots which must be in the section codes (see 3.2.6 page 50). Slots can contain data (values and references) or methods (code).

3.5.1 Keyword slots

Code slot may have arguments, which are separated by lowercase keywords. Numbers and the underscore are authorized for name of the slot and keywords (but the sequence ' __ ' is prohibited).

Here are the various way to identify a slot in LISAAC :

1. Argumentless slot definition without return value:

```
- init <- /* ... */
```

2. Argumentless slot definition with return value:

```
- get_color:INTEGER <- /* ... */
```

This slot returns an integer value.

3. Definition of a slot with argument and no return value:

```
- make count:INTEGER <- /* ... */
```

This slot takes an integer argument.

4. Definition of slot with argument and return value :

```
- qsort tab:ARRAY[CHARACTER] :BOOLEAN <- /* ... */
```

5. Definition of slot with argument list and keywords without return value:

```
- qsort tab:ARRAY[CHARACTER] from low:INTEGER to high:INTEGER <-  
/* ... */
```

This slot has three arguments and no return value. Note how the keywords help understand what the slot does.

6. Definition of slot with argument, keywords and return value :

```
- sort t:ARRAY[CHARACTER] from low:INTEGER to high:INTEGER :BOOLEAN <-  
/* ... */
```

 It's important to notice that after keywords you have only one argument. But an argument can be a vector argument, or LIST, as defined in 3.8 page 3.8.

```
- put_pixel (x,y:INTEGER) <- /* ... */
- draw_line (x,y:INTEGER) to (x1,y1:INTEGER) color (r,g,b:INTEGER) <- /* ... */
```

 Arguments are read only ! you can't modify an argument in a method, even if it is a list.

A message (or method) is identified by taking into account the message name as well as its keywords (if any). The names and positions of the keywords thus are very important.

```
- slot arg1:INTEGER from arg2:INTEGER <- /* ... */
- slot arg1:INTEGER from arg2:INTEGER to arg3:INTEGER <- /* ... */
```

The 2 slots defined in the previous example are considered different.

 Overloading is not allowed. Therefore, two messages can't differ only by the type of their arguments or the type of return. You can't also have 2 slots who differs only with the existence of a return value.

```
- slot arg1:INTEGER from arg2:INTEGER <- /* ... */
- slot arg1:BOOLEAN from arg2:INTEGER <- /* ... */ // Forbidden !
- slot arg1:INTEGER from arg2:INTEGER :INTEGER <- /* ... */ // Forbidden !
```

3.5.2 Binary messages

In LISAAC , everything is object and even the simplest operation is done using messages. For example, the binary operation ' 2 + 3 ' is a call of the message ' + ' on the object ' 2 ' using ' 3 ' as argument.

You can define binary operators in LISAAC as defined in the following. It is also possible to chose the associativity and the priority of operators, like for example in the ELAN language [PBy00].

To declare the associativity of an operator, the keywords *left* or *right* may be used.

Priorities are specified by a positive integer value. Priorities start at 1 (lowest priority) and have no upper limit³

The default associativity is *left*, and the default priority is 1.

Here is for example the code for the ****** (power) binary operator, that is left-associative and has priority 150.

```
- ** right 150 exp:INTEGER :INTEGER <-
  ( + result:INTEGER;

  (exp == 0).if {
    result := 1;
  } else {
    ((exp & 1) == 0).if {
      result := ((Self * Self) ** (exp / 2));
```

³Except the maximum allowed for 32 bits integers, of course.

```

    } else {
      result := (Self * (Self ** (exp-1)));
    };
  };
  result
);

```

You can use it with:

```
a := 2 ** 3;
```

Here is the possible code for an | binary operator that would be left-associative and have priority 40 in INTEGER:

```
- | left 40 other:INTEGER :INTEGER <- (!((Self) & (!other)));
```

You can use it with:

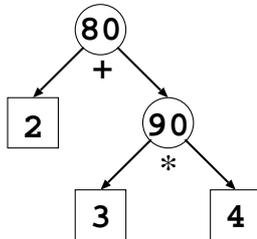
```
a := b | c;
```

```
- '+' left 80 other:SELF :SELF <- (Self - -other);
- '**' left 90 other:SELF :SELF <- /* ... */
```

In the expression

```
a := 2 + 3 * 4;
```

the first operation done is **3 * 4** then the addition.



Note that you will find a list of the binary operator more used in the glossary (see 5.1 page 109).

! Operators '=' and '!=' are reserved for reference comparisons. They have a right associativity and a priority of 50.

3.5.3 Unary messages

The only unary operators allowed are prefixed ones (put at the left of the the receiver).

A canonical example is the unary minus, whose code in INTEGER is:

```
- '-':INTEGER <- zero - Self;
```

You can use it with:

```
a := - 3;
```

Another common unary prefix operator in LISAAC is the question-mark '?'. It is used to allow a rudimentary contract-programming mechanism, very much like the `assert` mechanism of C or Java, but in a much less powerful way than the `require/ensure` Eiffel mechanism. See also 3.11.1 page 89. Here is the code for the ? unary prefix operator define in BLOCK object:

```
- '?' <-
(
  ((_debug_level > 0) && {! value }).if {
    check_crash;
  };
);
```

Note that `_debug_level` is a predefined flag set by the compiler according to the parameters chosen by the developer at compile time. You can use it with:

```
? {a = 3}; // You will see later that between {} you define a BLOCK object.
```

Here is an illustration of the use of ? to implement a kind of routine pre- and post-conditions:

```
- gcd other:INTEGER :INTEGER <-
  // Great Common Divisor of 'self' and 'other'
  ( + result:INTEGER;
    ? {Self >=0}; // a precondition
    ? {other>=0}; // a second one
    (other == 0).if {
      result := Self;
    } else {
      result := other.gcd (Self % other);
    };
    ? {result == other.gcd Self}; // a postcondition
    result
  );

- factorial:INTEGER <-
  ( + result:INTEGER;
    ? {Self>=0};

    // factorial
    (self == 0).if {
      result := 1;
    } else {
      result := (Self * (self - 1).factorial);
    };
    result
  );
```

Note that once the object code has been tested and debugged, the developer can switch off these assertions in the final delivery version by using a simple compile-time option.

3.6 Message send, late binding

The syntax of message calls in LISAAC strongly looks like message calls in Self.

<i>Message kind</i>	<i># of Arguments</i>	<i>Precedence</i>	<i>Associativity</i>
keyword	≥ 0	highest	left-to-right
unary	0	medium	right
binary	1	lowest	left or right*

* Default associativity for binary messages is *left*, but it can be changed, because associativity is defined at the time of the slot declaration (see section 3.5.2 page 71).

 The priority defined by an integer for the binary expressions apply only between the binary operators.

Arguments may be separated by commas or may use keywords as well (the method name is splitted into words to separate arguments), as seen in section 3.5.1 page 70.

3.7 Assignment

The declaration of a slot defines its evaluation mode: immediate or delayed. The slots with immediate evaluation will be evaluated in the order of their declarations (order of the lookup, see 3.2.2.0). They are evaluated at the load time of the object in memory. The starting point of a program will thus naturally be defined by a slot of this type.

- **Definition with ':=': immediate evaluation** The slot is evaluated immediately, that is automatically, when the prototype is loaded :

```
- max_character:INTEGER := (2 ** 8) - 1;
```

The main slot containing the program is declared this way and is thus evaluated as the initial root prototype is loaded :

```
- main :=
  ( IO.put_string "Hello world !"; );
```

- **Definition with '?=': immediate evaluation** The slot is evaluated immediately, that is automatically, when the prototype is loaded :

```
- to_value_if_possible:VALUE ?= Self;
```

If the result is bad type then the result is NULL. See more in the section 3.7.3 page 76.

- **Definition with '<-' : delayed evaluation** Slots declared this way are evaluated only when explicitly requested by a message send:

```
- display <-
  ( IO.put_string "Hello world !"; );
```

A normal slot method is declared this way. In order to trigger the evaluation of **display**, it has to be called, like in the following program:

```
- main :=
  ( display; ); // explicit call
```

3.7.1 Typing rules

Assignment follows strict rules in order to respect typing. Examine all the possible cases of the assignment $A := B$.

B A	TYPE	SELF	Expanded TYPE	Separate TYPE	Expanded SELF	Separate SELF
TYPE	B.sub A	B.sub A	FALSE	FALSE	FALSE	FALSE
SELF	A == B	A = B	FALSE	FALSE	FALSE	FALSE
Expanded TYPE	A == B (Copy B)	A == B (Copy B)	A = B (Copy B)	FALSE	A == B (Copy B)	FALSE
Separate TYPE	FALSE	FALSE	FALSE	B.sub A	FALSE	B.sub A
Expanded SELF	A == B (Copy B)	A == B (Copy B)	A == B (Copy B)	FALSE	A = B (Copy B)	FALSE
Separate SELF	FALSE	FALSE	FALSE	A == B	FALSE	A = B

Notes: 'B.sub A' means that B has the same type as A or of its descendants (B is sub type of A). 'A = B' means that A and B have the same reference (pointer). 'A == B' means that A and B have the same structure (deep equal).

3.7.2 Implicit-receiver messages

Keyword messages are frequently written without an explicit receiver. Such messages use the current living object (named **Self**) as the implied receiver. When you call a slot inside an object, it implicitly call the slot of the *self* object. The keyword **Self** can be used to explicitly call the *self* object.

Section Header

```
+ name := EXAMPLE;
- category := APPLICATION;
```

Section Public

```
+ slot_data:INTEGER := 3;
```

```
- main <-
(
  Self.slot_data.print;    // produce exactly the same code as slot_data.print;
);
```



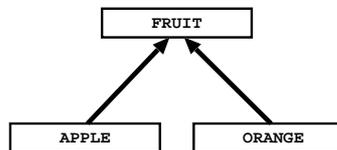
Note that the *self* is different between all the objects, even if they have the same type.



Unary and binary messages do not accept the implicit receiver, they require an explicit one.

3.7.3 A particular assignment: ?=

We see that we can assign an object slot with an object of the same type or of its descendants.



```
+ f:FRUIT;
+ a:APPLE;
a := APPLE;
f := a;
```

But you can't assign an object with one of its parents.

```
+ f:FRUIT;
+ a:APPLE;
f := FRUIT;
a := f;    // Error: static type FRUIT is invalid with static type APPLE
```

You can use the assignment defined with `?=` to assign an object with an other object of the same dynamic type. During compilation, the static type can be different but it don't stop with error.

```
+ f:FRUIT;
+ a:APPLE;
(test).if { f := APPLE; }
else      { f := ORANGE; };
a ?= f;
```

During the compilation, the dynamic type of `f` is not known at the time of assignment but there is no typing error because static type of `f` is `FRUIT`, of which inherits `APPLE`, static type of `a`.

The results depends of the dynamic type of `f`. If the dynamic type of `f` is exactly the same as `a`, the assignment is done as a standard assignment. If the dynamic types are different, the receiver `a` is assigned with `NULL`.

3.7.4 Binary message send

Here is a series of examples to illustrate the above precedence rules:

<i>Source code</i>	<i>is interpreted as</i>
<code>2 + "25".to_integer + 5</code>	<code>((2 + ("25".to_integer)) + 5)</code>
<code>object.set_value 2+2</code>	<code>((object.set_value 2) + 2)</code>
<code>2+2 .to_string</code>	<code>(2 + (2.to_string))</code>

3.7.5 Unary message send

<i>Source code</i>	<i>is interpreted as</i>
<code>object.set_value -2</code>	<code>((object.set_value) - (2))</code>
<code>-2.to_string</code>	<code>(- (2.to_string))</code>
<code>- + - 2</code>	<code>(- (+ (- 2)))</code>

Other example:

```
? {array != NULL};
```

3.8 Statement lists

A statement list, or simply “list”, is a sequence of one or several statements, contained between parentheses ‘(’ ... ‘)’’. Statements are both considered as instructions (doing something) and expressions (having a value), at the same time. Consecutive statements are separated by a semicolon ‘;’. If you want a return value, the result must be the last expression, without ending by ‘;’. You can return multiple values, as a vector of values (values separated with a comma ‘,’’, respecting the order).

Without return value	With one return value	With N return value
<code>(local;</code>	<code>(local;</code>	<code>(local;</code>
<code> expr1;</code>	<code> expr1;</code>	<code> expr1;</code>
<code> expr2;</code>	<code> expr2;</code>	<code> expr2;</code>
<code> expr3;</code>	<code> expr3;</code>	<code> result,</code>
<code> expr4;</code>	<code> result</code>	<code> result2</code>
<code>)</code>	<code>)</code>	<code>)</code>

A list is immediately evaluated when reached by the execution flow. Thus, a routine whose argument is the (single-statement) list ‘(3 + 2)’ receives as argument the result of the evaluation, 5, not the list itself⁴.

```
- make count:INTEGER <- /* ... */
/* ... */
make (3 + 2);
make 5;
/* ... */
```

You can also have code and return value for arguments:

⁴This is the contrary for statement blocks, see section 3.9 page 83.

```
make ("Here is the call with a list !".print; 3 + 2);  
/* ... */
```

Consequently, there is absolutely no difference between a one-statement list in LISAAC and an expression as classically defined in most programming languages.

3.8.1 Return values of lists

The type and return value of a list are determined by the last expression (statement) of the list, after the last semicolon ';' and right before the closing parenthesis ')'.
'

For example, the following list returns an INTEGER value:

```
(  
  a := foo;  
  5.factorial    // INTEGER value returned  
)
```

Note that there is no semicolon after the call to **factorial**.

This list also quite intuitively returns an INTEGER:

```
(2 * (5 + 3))    // two nested lists, both returning INTEGER
```

This list can returns more complex objects, such as BOOLEAN:

```
( a | ( b & c ) ) // two nested lists, both returning BOOLEAN
```

or whichever object:

```
(  
  "Here we create a clone of EXAMPLE object".print;  
  EXAMPLE.clone  
)
```

As said before, you can return multiple values by separating results with commas ', '.

```
( 3, 5 )        // two INTEGER value returns
```

Return values don't need to have the same type.

```
(  
  ( a | ( b & c ) ),  
  8  
)              // two return values, a BOOLEAN and an INTEGER
```

Lists could have code and multiple return values:

```
(  
  "Multiple return values".print;  
  EXAMPLE.clone,  
  ( a | ( b & c ) ),  
  6  
)
```

 You can put code between results, but you can't mix result and not results as explained in the following example:

```
(
  "Hello".print;
  3,
  "Ok".print; // Error: there is a result before, you must end with a result
);

(
  "Hello".print;
  3,
  "Ok".print;
  0 // Ok
);
```

A list may also have no return value at all:

```
(
  a := foo;
  5.factorial; // void return
)
```

In this example, there was a semicolon after the call to **factorial**. Intuitively, since there is nothing between the last semicolon and the closing parenthesis (or an “empty statement” only), nothing is returned from the list after it has been evaluated.

3.8.2 Use of lists

Expressions

It's the classical use of a LIST which one can find in other languages.

```
( 2 + 4 ) * 7 // list with a single return value
```

Methods

From the beginning of this manual, we define methods using lists.

```
- slot <-
(
  "Hello !".print; // List with no return value
);
```

Functions with one result

We see that the result must be the last expression before the end of the list, without using the semicolon. The definition of the return type is done after `:`.

```
- zero:INTEGER <-  
(  
  "Call zero function !".print;  
  0  
);
```

Functions with multiple results

The results are separated by a comma, at the end of the list. The definition of the return types is done after :, separated by commas.

```
- coord:(INTEGER,INTEGER) <-  
(  
  "Call coord function !".print;  
  x,  
  y  
);
```

You can also return different types.

```
- slot:INTEGER,BOOLEAN <-  
(  
  "Call slot function !".print;  
  count,  
  (count > 0)  
);
```

Arguments

Slots accept only one argument as defined in 3.5. But an argument can be a vector.

```
- put_pixel (coord_x,coord_y:INTEGER) <-  
(  
  x := coord_x;  
  y := coord_y;  
);  
  
- put_pixel (coord_x,coord_y:INTEGER) color (r,g,b:INTEGER) <-  
(  
  x := coord_x;  
  y := coord_y;  
  red := r;  
  green := g;  
  blue := b;  
);
```

Arguments in a list don't have to be of the same type, as we can imagine reading the previous examples. It's simply more readable to put keywords to separate arguments of different types.

```
- slot (value:INTEGER,condition:BOOLEAN,text:STRING) <- /* ... */
```

Call of slots

If a slot is defined with a list-argument, you must use a list to call this slot.

```
- put_pixel (coord_x,coord_y:INTEGER) <-
(
  x := coord_x;
  y := coord_y;
);
```

You call this slot with:

```
put_pixel (x,y);
```

You can also call it using a function returning a list:

```
- coord:INTEGER,INTEGER <-
(
  x,
  y
);
```

The call of the slot can be:

```
put_pixel coord;
```



You can't transform a call with keywords in a call with list.

```
- slot value:INTEGER from low:INTEGER to high <- /* ... */
```

Call `slot (3 , 4 , 5)`; is forbidden, it represents a slot defined with

```
- slot (value,low,high:INTEGER) <- /* ... */
```

Assignment

You can assign a list only with a list.

```
( a , b ) := ( 3 , 7 );
( x , y ) := coord;
```

You can also redefine a function (a liste assigned with delayed evaluation '<-'):

```
- msg_error msg:STRING <-
(
  "Error : ".print;
  msg.print;
);
```

```
- debug_mode <-
(
  msg_error msg <- // you don't have to precise the type of the argument
```

```
(
  "Error : ".print;
  msg.print;
  display_stack;
);
);
```



The redefinition of a function must respect the same profile for arguments.

Special case: receiver of message is a list

You can define a list as a receiver for a message.

```
- (b:BOOLEAN) slot a:INTEGER to c:INTEGER <- /* ... */
```

The call is done on the double result list:

```
(self,TRUE).slot 1 to 2;
```

The receiver of the message is the first element of the vector.

3.8.3 Local variables in statement lists

A list has its own environment and scoping. It is possible to declare variables that are local to the list and thus accessible from any statement inside the list but not from outside.

```
( + j,k:INTEGER;
  + array:ARRAY[STRING];
  /* ... */
)
```

Locals in lists have to be declared at the beginning of the list, before the first statement. Therefore, the following declaration is incorrect in LISAAC :

```
( + j,k:INTEGER;          // declaration, OK
  some_method_call;      // statement, OK
  + array:ARRAY[STRING]; // INVALID declaration !!
  /* ... */
)
```

Local variables declared with 'l' preserves their values with each call (variable persistent), as for the keyword 'static' for locals in C.

```
( + j,k:INTEGER;          // declaration, OK
  - counter_call:INTEGER;
  /* ... */
  counter_call := counter_call + 1;
)
```

3.9 Statement blocks

Statement blocks, or simply “blocks”, have a number of similarities with lists (see section 3.8 page 77).

A block is a sequence of one or several semicolon-separated statements (instructions), contained between braces ‘{’ ... ’}’. A block is an instance of prototype `BLOCK`.

Blocks are `LISAAC` closures like a list. Their evaluation is carried out in their definition environment. Contrary to lists, blocks are evaluated only when explicitly sent a **value** message. When a block receives an acceptable **value** message, its statements are executed in the context of the current activation of the method in which the block is declared. This allows the statements in the block to access variables that are local to the block’s enclosing method and any enclosing blocks in that method. This set of variables comprises the lexical scope of the block. It also means that within the block, **Self** refers to the receiver of the message that activated the method, not to the block object itself.

A block can take an arbitrary number of arguments and can have its own local variables, as well as having access to the local variables of its enclosing method.

One of the common uses of blocks in `LISAAC` is to implement library-defined control structures (see section 3.9.2.0 page 85).

Here, an example of a current use of a block.

```
(list = NULL).if {
  "List is empty!".print;
};
```

The block (`if` first’s argument) is evaluated only if conditional is true.

As for lists, you can have no return value or one or multiple return.

Without return value	With one return value	With N return value
{ <i>local</i> ;	{ <i>local</i> ;	{ <i>local</i> ;
<i>expr1</i> ;	<i>expr1</i> ;	<i>expr1</i> ;
<i>expr2</i> ;	<i>expr2</i> ;	<i>expr2</i> ;
<i>expr3</i> ;	<i>expr3</i> ;	<i>result</i> ,
<i>expr4</i> ;	<i>result</i>	<i>result2</i>
}	}	}

A block is equivalent with a list when you call the **value** message on it.

```
( local;
  expr1;
  expr2;
  expr3;
  expr4;
)
```

is equivalent with

```
{ local;
  expr1;
  expr2;
```

```
    expr3;  
    expr4;  
}.value
```

3.9.1 Return values of blocks

The value returned by a block is determined exactly like that of a list (see section 3.8.1 page 78).

The following examples thus are quite straight forward.

The following block returns an INTEGER value:

```
{  
  a := foo;  
  5.factorial    // integer value returned  
}
```

There is no semicolon after the call to **factorial**.

The right-hand-side of the || operator is a single-statement block that returns a boolean:

```
test := (j < upper) || {result != NULL};
```

A block may also have no return value at all:

```
{  
  a := foo;  
  5.factorial; // void return  
}
```

There was a semicolon after the call to **factorial**. Since there is nothing between the last semicolon et the closing curly braket (or an “empty statement” only), nothing is returned from the block after it has been evaluated.

3.9.2 Use of blocks

When using a block as an argument, it’s not the result of the block that is passed (as for lists) but the block itself. This property has an incidence on the way you declare the slots.

```
- slot b:BLOCK <- /* ... */
```

The call must be with a slot object:

```
slot {/* ... */};
```



You must ensure that what is defined in the block is independent from the context.

Let’s see an example.

```
- my_block:BLOCK;  
  
- slot <-  
( + a:INTEGER;  
  my_block := { a }; // Forbidden !  
);
```

When the evaluation of the return block (with the message **value**), the local variable 'a' don't exist ! the result can't be evaluated.

An example of correct use:

```
- slot <-
( + a:BLOCK;
  a := { "World!".print; };
  "Hello ".print;
  a.value;
);
```

Blocks are used in library to define conditionnals, loops and iterations. You will find more in the section Library (see 4).

Expressions

```
(a != NULL) && { b = 3}
```

In the definition of the binary slot **&&** you find the evaluation of the block. In the FALSE prototype:

```
- '&&' left 20 other:BLOCK :BOOLEAN <- FALSE;
```

In the TRUE prototype:

```
- '&&' left 20 other:BLOCK :BOOLEAN <- other.value;
```

Conditionals

```
(a>b).if {"Yes!".print;} else {"No!".print;};
```

In the definition of the slot **if ... else** you find the evaluation of the block. In the FALSE prototype:

```
- if true_block:BLOCK else false_block:BLOCK <-
(
  false_block.value;
);
```

In the TRUE prototype:

```
- if true_block:BLOCK else false_block:BLOCK <-
(
  true_block.value;
);
```

Loops

Do While

```
{ j := j + 1; j.print; }.do_while {j<10};
```

The slot `do_while` is defined directly in the BLOCK object:

```
- do_while test:BLOCK <-  
(  
  value;           // call of value on BLOCK self  
  test.value.if {  
    do_while test; // Defined recursively  
  };  
);
```

Iterator

```
1.to 10 do {"Hello!".print};
```

The slot `to ... do` is defined in the NUMERIC object:

```
- to limit_up:SELF do blc:BLOCK <-  
(  
  (self<=limit_up).if {  
    blc.value self;  
    (self + 1).to limit_up do blc;  
  };  
);
```

3.9.3 Argument and local variables in statement blocks

Locals in blocks are declared like locals in lists (see section 3.8.3 page 82):

```
my_block := { + j,k:INTEGER; // Locals list.  
              + array:ARRAY[STRING];  
              /* ... */  
            };  
/* ... */  
my_block.value;
```

You can also call a slot with an argument. It's defined as local variables, but without sign before.

```
my_block := { arg:INTEGER; // Argument  
              + i,j:INTEGER; // Locals list.  
              /* ... */  
            };  
/* ... */  
my_block.value 3;
```

An argument can also be a vector of arguments.

```

my_block := { (arg1:INTEGER, arg2:STRING); // Argument list.
              + j,k:INTEGER;
              /* ... */
            };
/* ... */
my_block.value (3, "Ok !");

```

The same restrictions as for locals in lists also apply: local have to be declared before any statement and after possible the arguments.

3.10 Auto-cast

Sometimes you want to transform an object in another object, especially for the numbers. This can be done with the "Auto-cast" facility. In the HEADER section, in the slot **name**, you can define the prototypes in which the object can be "auto-casted" with the '->' symbol.

Section Header

```

+ name := PROTO1 -> PROTO2,PROTO3;
- category := APPLICATION;

```

Section Public

```

- to_proto2:proto2 <- ( /*...*/ )
- to_proto3:proto3 <- ( /*...*/ )

```

 In the public section, you must define functions called **to_name_of_type** (here **to_proto2** and **to_proto3**) which are automatically called when there is an autocast. These function must return the corresponding type.

Section Header

```

+ name := TEST;
- category := APPLICATION;

```

Section Public

```

- main :=
( + a:PROTO1;
  + b:PROTO2;
  /* ... */
  b := a; // similar to: b := a.to_proto2;
  /* ... */
);

```

 Auto-cast is not transitive: if A can be auto-casted into B, and B into C, A can't be auto-casted in C. You must explicitly precise the auto-cast of A into C if you need this.

 Auto-cast is not inherited.

3.11 Tools for programming by contract

Compiler furnishes 2 native functions :

```
- debug_level:NUMERIC      // Flag indicating the level of debug mode
- print_runtime_stack      // Print the stack (only if debug mode is on)
```

Example: output with **print_runtime_stack**

```
===== BOTTOM =====
Line #00015 Column #00011 in ESSAI.
Line #00154 Column #00010 in STRING.
Line #00233 Column #00008 in NUMERIC.
Line #00089 Column #00017 in BLOCK.
Line #00056 Column #00023 in OBJECT.
===== TOP =====
```

One example of use of this 2 functions is assertions. Assertions are code conditions who are verified during execution when the object is compiled with the debug option. There is 2 types of assertions: first is a unary message of BLOCK.

```
- '?' <-
(
  ((debug_level > 0) && {! value }).if {
    check_crash;
  };
);
```

The block condition must return one BOOLEAN value. In OBJECT prototype the **check_crash** code is defined:

```
- check_crash <-
(
  print_runtime_stack;
  die_with_code_exit_failure_code;
);

- exit_failure_code:INTEGER := 1;

- die_with_code code:INTEGER <- SYSTEM.exit code;
```

The **exit** slot of **SYSTEM** depends of the architecture on which you run the program.

For example, for UNIX, the slot **exit** is defined as following:

```
- exit code:INTEGER <- 'exit(@code)'; // External, see 3.12.2
```

The assertions can be put wherever you want in the code.

```
/* ... code ... */
? {
  "We verified your code".print;
  j > 0
};
/* ... code ... */
```

The second type of assertion is a binary message of NUMERIC: This type of assertion depends of the level of debug set while compiling.

```

- '?' b:BLOCK <-
(
  ((debug_level > Self) && {! b.value }).if {
    check_crash;
  };
);

```

This kind of assertion can be put anywhere in the code:

```

/* ... code ... */
5 ? {
  "Debug test level 5".print;
  j > 0
};
3 ? {
  "Debug test level 3".print;
  j < 10
};
1 ? {
  "Debug test level 1".print;
  i > 0
};
/* ... code ... */

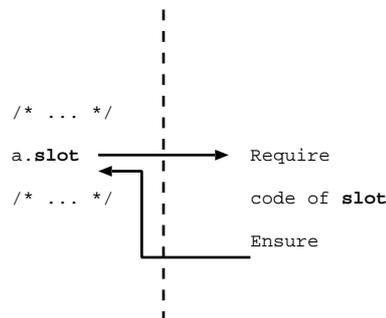
```

If you compile using the level 4 for debug, only the 2 last assertions will be verified. Using this kind of assertions let you assign priorities into verifications.

The stack is written from bottom to top, it indicates the way the program follow during execution. You can then easily find where the condition is false.

3.11.1 Requires and Ensures

To securise programming, you can put code requires, ensure and invariant in code: conditions who have to be verified each time you call a message. Before the call, the conditions are called **Require**, and after the call **Ensure**.



Require and Ensure are defined between [and]. The Require is written between the slot header and the code.

```

- slot <-

```

```
[
  // Require
]

(
  // Code
)

[
  // Ensure
];
```

In Require or Ensure section you can then write your code as for any other method. You can define local variables, but their visibility is limited in the Require **or** in the Ensure (a local variable defined in the Require is not visible in the code or in the Ensure). Local variables defined in the code are also only limited to the code section.

```
- slot <-
[
  + a:INTEGER;
  a := 3;
  ? {count > a};
]
( + b:INTEGER;
  b := a; // Error: 'a' is not defined in the Code section
  b := b * 3;
)
[
  ? {count > a}; // Error: 'a' is not defined in the Ensure section
  ? {count < b}; // Error: 'b' is not defined in the Ensure section
];
```

In the Require and Ensure section, you can define as many assertions as you want.

3.11.2 Invariant

You can define at the end of code invariant conditions, who must be verified each time you call a message on an object. The invariant is defined between [and].

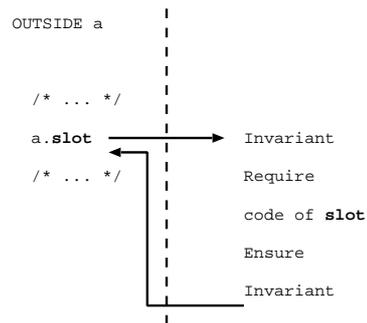
Section Header

```
- name := /* ... */
/* ... */
```

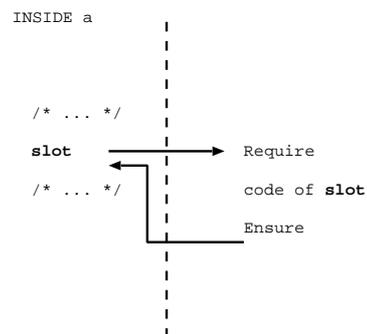
Section Public

```
/* ... */
[
  ? {lower <= upper + 1};
];
```

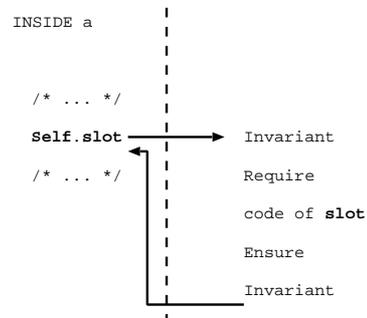
The invariant is verified each time you call a message with the explicit receiver, before and after the call.



If the call is done inside the living object, the invariant is not verified.



If inside the living object, you explicitly call the **Self** object, the invariant will be verified.



3.11.3 Result and Old

You can use the keywords **Old** and **Result** $_x$ to add more verifications. The keyword **Old** can be used in Ensures and Invariant, and is written before a function to indicate the value of this slot *before* the call of the current slot.

```
- slot <-
(
  count := count + 1;
)
[
  ? {count = Old count + 1};
];
```



You can use **Old** only with slots with no arguments.

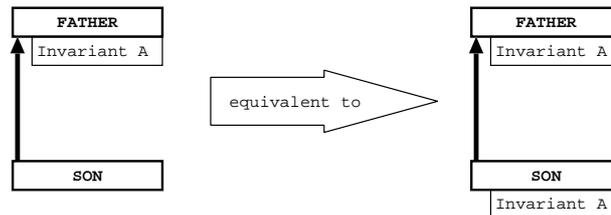
The keyword **Result**_{*x*} represents the result of the function and can be used only in Ensures. If there is only one result, use simply the keyword **Result**.

```
- slot:INTEGER <-
( + a:INTEGER;
  a := count;
  (a > 0).if {
    a := a + 1;
  } else {
    a := 1;
  };
  a
)
[
  ? { Result >= 1 };
];

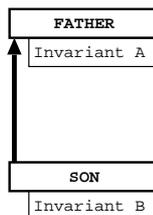
- slot2:INTEGER,INTEGER <-
(
  count + 1,
  count - 1
)
[
  ? { Result1 >= 1 };
  ? { Result2 > 0 };
];
```

3.11.4 Inheritance

Objects inherit invariants from their parents, following the lookup algorithm.

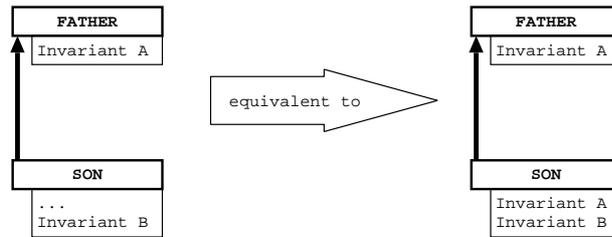


If an invariant is defined in an object, it replace those of its static parent.



An object can also inherit invariant from its parent and add its own invariant. This is done by using dots (...). The invariant of the parent is inserted where the dots are written.

```
[
  ...
  ? {count > 0};
];
```



It's the same pattern for Require and Ensures. If an object has a slot with no Require or/and Ensure, this slot inherits of the Require / Ensure of the slot of the parent of the object (if any). The Require / Ensure defined in a slot replace those of the parent of the object, and the dots are the way to insert in an existing Require / Ensure the conditions of the parent.

3.12 Externals

To include C code in LISAAC language, there is two way, the slot **external** in the **Section Header** or directly in the LISAAC code as following. The C code is defined between ‘.

3.12.1 Slot external

All the C code defined in the slot **external** is directly included in the code. You can define includes, functions, macros, ...

Section Header

```
- name := EXAMPLE;
- category := APPLICATION;
- external := ‘#include <stdio.h>
               // Hardware 'print_char'
               int print_char(char car)
               {
                 fputc(car,stdout);
               }‘;
```



This C code is NOT verified by the LISAAC compiler.

3.12.2 C code in Lisaac

Anywhere in the code (in the definition of a slot, even in Require / Ensure or Invariant) you can insert C code.

Chapter 3. Language Reference

```
- slot <-  
( + a:INTEGER;  
  a := count;  
  'fputc('Y',stdout)';  
);
```

You can use LISAAC local variable or argument with '@' before the variable.

```
- slot <-  
( + a:CHARACTER;  
  'fputc(@a,stdout)';  
);
```

 Global variable (slots) are not permitted in the external, if you have to work with it, use a local variable.

```
- data:CHARACTER := 'Y';  
- slot <-  
(  
  'fputc(@data,stdout)';           // Forbidden !  
);
```

 Variable used in external are read-only.

```
- slot <-  
( + a:INTEGER;  
  '@a ++';                         // Forbidden !  
);
```

You can assign a variable with the result of an external, but you have to precise the return type after `:`.

```
- slot <-  
( + a:INTEGER;  
  a := '@a + 1':INTEGER;  
);
```

You can also precise the dynamic type of the return, if any, as a list of types between parenthesis.

```
- slot <-  
( + a,b:INTEGER;  
  + c:BOOLEAN;  
  a := count;  
  b := size;  
  c := '@a == @b':BOOLEAN(TRUE,FALSE);  
);
```

 The compiler optimize the code by deleting variables that are not used and the code of the external if the result is not used (dead code). It can be hazardous if you don't use the return value of a C function but if you really need the function to be executed.

```
- slot <-
( + a:CHARACTER;
  a := 'getchar()':CHARACTER;
);
```

If you don't use 'a', the variable and the assignment will be simply deleted ! You can force an external to be persistent by using parenthesis around the result type.

```
- slot <-
( + a:CHARACTER;
  a := 'getchar()':(CHARACTER);      // persistent external
);
```

In this case, the result is precised as optionally used and the compiler will not optimize the code and don't delete the external, even if the result is not used.

3.12.3 Lisaac code in C

As explained in 3.2.5, the **Section External** is reserved to define slots which keep their LISAAC name in the generated C code. You can then link the produced C code with other programs keeping the name of the functions.

For example, a slot defined as:

Section Public

```
- slot v:INTEGER :INTEGER <- /* ... */
```

could be compiled and produce a C function

```
static int slot__H8(unsigned long v__GGC) // code is an internal coding of the compiler
```

If you define the slot in a **Section External** you keep the name:

Section External

```
- slot v:INTEGER :INTEGER <- /* ... */
```

This code will be compiled in:

```
int slot (unsigned long v_UCC)          // It keeps the name of the function
```

You can't define function with keywords. If an external function must have multiple arguments use a list:

Section External

```
- slot (a,b:INTEGER,c:CHARACTER) :INTEGER <- /* ... */
```

Which will be compiled in:

```
int slot (a__EDC,b__UFC:integer,c__CCD:character)
```

Note that in this case, the function is not static, and can be accessed by other programs (not inlined).

3.12.4 Lisaac external

Externals composed with a simple integer are LISAAC externals (compiler native functions).
Example:

```
'2';
```

These externals are used for example to define basis operations.

```
- '>' right 60 other:SELF :BOOLEAN <- '1';  
- '-' left 80 other:SELF :SELF <- '2';  
/* ... */
```

Chapter 0100b

The LISAAC Library

In this chapter you will find the description of some prototypes and functions, some of them are at the core of the library, other are the most commonly used.

4.1 OBJECT

OBJECT is the base prototype who contains all the core functions needed to program efficiently. All the prototypes of the library inherits, directly or not, of this prototype. When defining your own prototypes, don't forget to inherit of OBJECT if you want to use its functions.

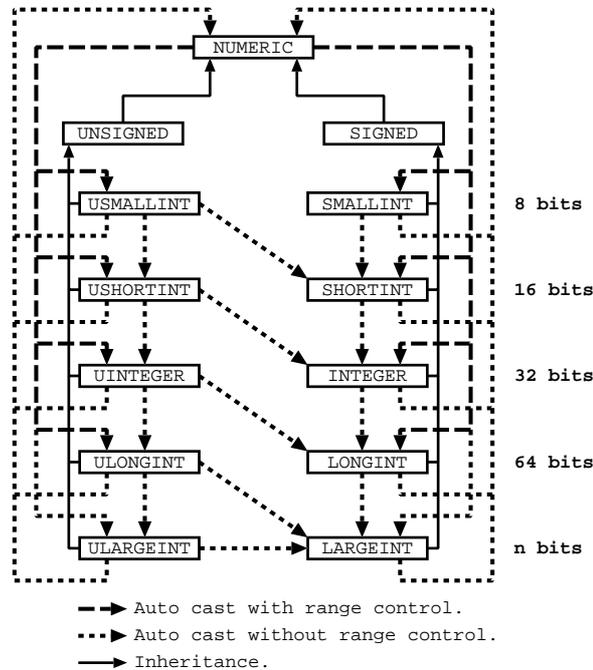
The most commonly slots are:

```
//
// Compiler consideration.
//
- object_size:INTEGER // size of the current prototype (in bytes)
- is_debug_mode:BOOLEAN // indicates if the object was compiled using
// the debug option

//
// Control Error.
//
- is_clonable:BOOLEAN // indicates if object is clonable (name defined with '+')
- print_runtime_stack // print stack as defined in 3.11
- die_with_code code:INTEGER // Terminate execution with exit status code
- crash_with_message msg:ABSTRACT_STRING // Terminate execution writing msg
//
// Common Function.
//
- '==' right 60 other:SELF :BOOLEAN // TRUE if objects are equal
// (to redefine in each object)
- '!==' right 60 other:SELF :BOOLEAN
- clone:SELF // clone of the object
- to_pointer:POINTER // return a pointer on this object
```

4.2 NUMERIC

All the numbers inherits of the NUMERIC prototype. There are facilities of conversion between the types, as you can see on the following figure.



Not all the types and conversions are drawn here for simplification needs.

For example, a USMALLINT can be converted to a SHORTINT without range control, using auto-cast.

```
+ a:SHORTINT;
+ b:USMALLINT;
/* ... */
a := b;
```

When you explicit a value, this value is NUMERIC. The conversion is done by the assignment.

```
+ a:INTEGER;
/* ... */
a := 3; // 3 is NUMERIC, auto-casted to INTEGER, with range control.
```

NUMERIC and all its sons are expanded types, already living.

The slots most commonly used are:

```
//
// Arithmetic operations
//
- '+' left 80 other:SELF :SELF // add
- '-' left 80 other:SELF :SELF // subtract
- '*' left 100 other:SELF :SELF // multiply
```

```

- '/' left 100 other:SELF :SELF // divide
- '%' left 100 other:SELF :SELF // modulo
- '**' right 120 exp:SELF :SELF // power
- '+' :SELF // positive (unary message)
- '-' :SELF // negative (unary message)

//
// Bitwise operations
//
- '&' left 100 other:SELF :SELF // bitwise and
- '|' left 80 other:SELF :SELF // bitwise or
- '^' left 80 other:SELF :SELF // bitwise xor
- '~' :SELF // bitwise complement (unary message)
- '>>' left 100 other:NUMERIC :SELF // logical shift right
- '<<' left 100 other:NUMERIC :SELF // logical shift left

//
// Comparisons :
//
- '=' right 60 other:SELF :BOOLEAN
- '!=' right 60 other:SELF :BOOLEAN
- '>' right 60 other:SELF :BOOLEAN
- '<' right 60 other:SELF :BOOLEAN
- '<=' right 60 other:SELF :BOOLEAN
- '>=' right 60 other:SELF :BOOLEAN
//
// Loops
//
- to limit_up:SELF do blc:BLOCK // iterate forward from Self to limit_up
- downto limit_down:SELF do blc:BLOCK // iterate backward from Self to limit_down

- to limit_up:SELF by step:SELF do blc:BLOCK // iterate forward with step
- downto limit_down:SELF by step:SELF do blc:BLOCK // iterate backward with step

//
// Print
//
- to_hexadecimal:STRING // return a string with the hexadecimal value
- print // print the value to the standard output
//
// Debug
//
- '?' b:BLOCK // assertion, see 3.11

```

Example: use of loops

```

1.to 10 do { i:INTEGER;          // i is the argument of the block
  i.print;
};

16.downto 0 by 2 do { i:INTEGER;
  i.print;
};

```

4.3 CHARACTER

CHARACTER is an expanded prototype, represented by one byte, with a character value. It can be autocasted in smallint without range control. Here is a few commonly used slot:

```

//
// Conversions
//

- code:SMALLINT          // ASCII Code
- to_upper:CHARACTER    // return the equivalent character in upper case
- to_lower:CHARACTER    // return the equivalent character in lower case

//
// Tests
//

- is_letter:BOOLEAN     // Is it a letter ('a' .. 'z' or 'A' .. 'Z') ?
- is_digit:BOOLEAN      // Belongs to '0'..'9'.

```

A character is defined between '.

```

+ c:CHARACTER;
c := 'z';
'z'.is_letter.if /* ... */

```

4.4 BOOLEAN

BOOLEAN is an expanded type from which inherits TRUE and FALSE. By default a BOOLEAN is dynamically FALSE. Conditionnals methods are declared in BOOLEAN but their real definition stand in TRUE or FALSE, the dynamic types of BOOLEAN.

For example, the `if ... else` method is declared deferred (to be redefine) in BOOLEAN.

```

- if b_true:BLOCK else b_false:BLOCK <- deferred;

```

In TRUE the slot is redefine:

```
- if b_true:BLOCK else b_false:BLOCK <- b_true.value;
```

In FALSE the slot is redefine:

```
- if b_true:BLOCK else b_false:BLOCK <- b_false.value;
```

Just examine the following call:

```
(a > b).if { "Yes!".print; } else { "No!".print; };
```

The list (a > b) returns a boolean, which will dynamically be TRUE or FALSE. Then the evaluation of the slot `if ... else` will be done in the corresponding prototype and returns the real result by late binding.

- (a > b) returns TRUE : the code evaluated is `b_true.value` so { "Yes!".print; }
- (a > b) returns FALSE: the code evaluated is `b_false.value` so { "No!".print; }

All the functions of BOOLEAN follow the same pattern using late binding.

```
//
// Logical operations :
//
- '!' :BOOLEAN           // not (unary slot)
- '&' left 20 other:BOOLEAN :BOOLEAN // and (strict, total evaluation)
- '|' left 10 other:BOOLEAN :BOOLEAN // or (strict, total evaluation)
- '&&' left 20 other:BLOCK :BOOLEAN // and then (semi strict)
- '||' left 10 other:BLOCK :BOOLEAN // or else (semi strict)
- '^' left 10 other:BOOLEAN :BOOLEAN // xor
- '->'right 25 other:BOOLEAN :BOOLEAN // imply

//
// Conditionals
//
- if true_block:BLOCK :BOOLEAN
- if true_block:BLOCK else false_block:BLOCK
- elseif cond:BLOCK then block:BLOCK :BOOLEAN
- elseif cond:BLOCK then block:BLOCK else block_else:BLOCK
```

Here is a few examples of using BOOLEAN.

```
+ a,b,c,d,e:BOOLEAN;
/* ... */
e := (a | b) && { c -> d};
/* ... */
(a ^ c).if { "Ok!".print; } // return a BOOLEAN
      .elseif { d } then { "Ko!".print; } else { "Maybe!".print; };
/* ... */
```

4.5 BLOCK

BLOCK is a particular prototype because it is implicitly constructed using braces { and }. For more informations about blocks see 3.9. Here is some functions defined in the BLOCK prototype.

```
//
// Conditional :
//
- '||' left 10 other:BLOCK :BOOLEAN // or else
- '&&' left 20 other:BLOCK :BOOLEAN // and then
//
// Loop :
//
- while_do body:BLOCK // while self is TRUE, evaluate body
- do_while test:BLOCK // evaluate self while test is TRUE
- until_do body:BLOCK // until self is TRUE, evaluate body
- do_until test:BLOCK // evaluate self until test is TRUE
//
// Debug
//
- '?' // assertion, see 3.11
```

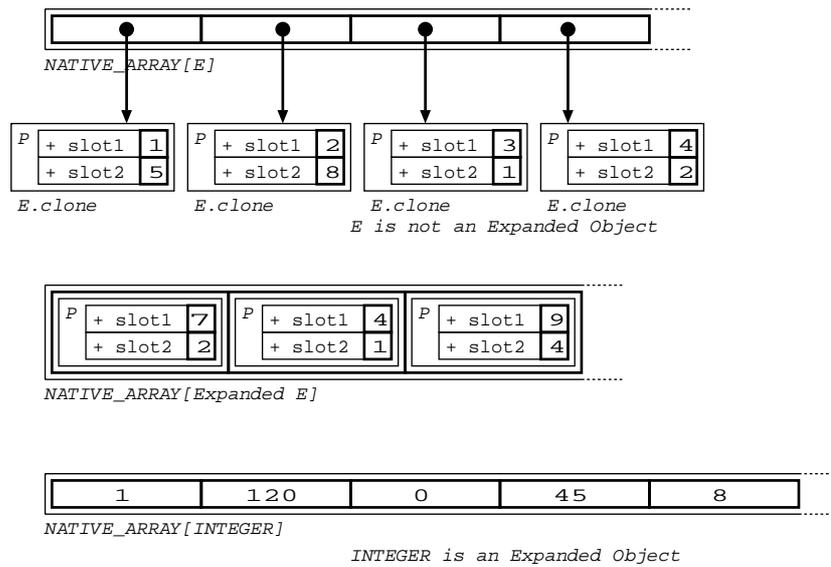
Example: using loops

```
+ j:INTEGER;
{ j < 10 }.while_do { j.print; j := j + 1; };
j := 0;
{ j.print; j := j + 1; }.do_until { j >= 10 };
```

4.6 NATIVE_ARRAY

NATIVE_ARRAY is a particular collection prototype using genericity. It's an expanded type who have the particularity to be directly matched on memory datas. This prototype is at the core of all the collections (arrays).

 Be careful when using a NATIVE_ARRAY, there is no bound control, it's equivalent to a variable defined with (void *) in C. The use of NATIVE_ARRAY is reserved to experts because of its low level. If you want more informations about the use, watch the code of this prototype and how it is used in collections.



4.7 STRING

There 3 type of string in the library: `ABSTRACT_STRING`, `STRING_CONSTANT` and `STRING`.

`ABSTRACT_STRING` is an abstract prototype, who define the standard operations on a string.

`STRING_CONSTANT` inherits of `ABSTRACT_STRING`. A `STRING_CONSTANT` can't be modified after being created. You can create it as following:

```
+ a:STRING_CONSTANT := "Hello world !";
```

`STRING` also inherits of `ABSTRACT_STRING`. This object can be modified in many ways.

This 3 prototypes are similar in their internal representation.

Section Header

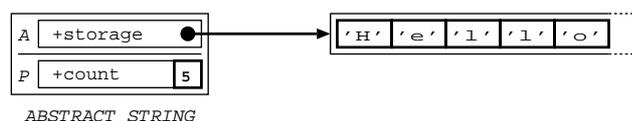
```
- name := ABSTRACT_STRING -> STRING; // can be autocast in STRING
- category := DOCILE;
/* ... */
```

```
Section ABSTRACT_STRING // ABSTRACT_STRING and its descendants
```

```
+ storage:NATIVE_ARRAY[CHARACTER];
```

Section Public

```
/* ... */
```



In `ABSTRACT_STRING` you can find the following slots (visible from `STRING` and `STRING_CONSTANT`, because of the inheritance.

```
//
// Features
//
```

Chapter 4. The LISAAC Library

```
+ count:INTEGER           // Number of elements of storage
- lower:INTEGER := 1;     // The elements are numbered from 1 to count
- upper:INTEGER           // Number of the last element
- capacity:INTEGER       // Number of reserved elements for storage

//
// Access
//

- item index:INTEGER :CHARACTER           // Element number index
- '==' left 40 other:ABSTRACT_STRING :BOOLEAN // True if strings have the same text
- same_as other:ABSTRACT_STRING :BOOLEAN  // Case insensitive '=='

//
// Testing
//

- has ch:CHARACTER :BOOLEAN           // True if 'ch' is present
- has_substring other:ABSTRACT_STRING :BOOLEAN // True if 'other' is present

//
// Operations
//
- '+' other:ABSTRACT_STRING :STRING // New STRING, concatenation of Self and other.
- substring start_index:INTEGER to end_index:INTEGER :STRING // Create a substring

A STRING_CONSTANT is particular because it can't be modified.

- to_string:STRING // create a STRING object from a STRING_CONSTANT

A STRING object is not an expanded prototype so it must be cloned from the 'master' object.

//
// Creation
//

- create needed_capacity:NUMERIC :SELF // Create with needed_capacity but empty
- create_from_string str:ABSTRACT_STRING :SELF // Create with a copy of str

//
// Modifications
//

- clear // Count is reseted, but capacity remain identical
- append other:ABSTRACT_STRING // Append other to Self
- prepend other:ABSTRACT_STRING // Prepend other to Self
```

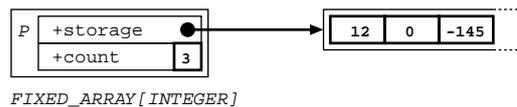
- `put ch:CHARACTER to index:INTEGER` // Put *ch* at position *index*
- `add_last ch:CHARACTER` // Append *ch* to Self
- `to_lower` // Convert all the characters to lower case
- `to_upper` // Convert all the characters to upper case

4.8 FIXED_ARRAY

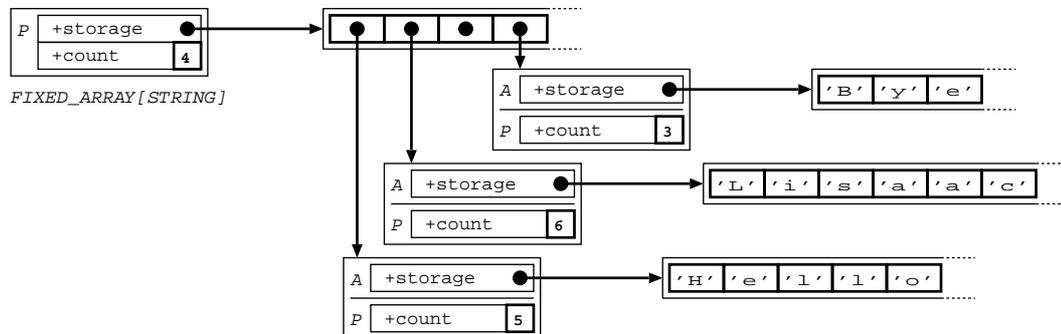
`FIXED_ARRAY` is an array with a fixed lower bound using genericity. You can define a `FIXED_ARRAY` of any object. As for `STRING`, `FIXED_ARRAY` has a storage:

```
+ storage:NATIVE_ARRAY[E]; // Internal access to storage location
+ count:INTEGER; // Number of elements of the array
```

Example: `FIXED_ARRAY` of an expanded object



Example: `FIXED_ARRAY` of an non expanded object



```
//
```

```
// Features
```

```
//
```

```
- lower:INTEGER := 0; // The elements are numbered from 0 to count - 1
- upper:INTEGER // Number of the last element
- capacity:INTEGER // Number of reserved elements for storage
```

```
//
```

```
// Creation
```

```
//
```

```

- create new_count:numeric :SELF // Create an array of new_count elements
                                // initialized to the default of 'E'

- create_with_capacity new_count:numeric :SELF // Create an empty array
                                                // of new_count elements reserved

//
// Access
//
- item index:INTEGER :E // Element number index
//
// Testing
//
- '==' right 60 other:SELF :BOOLEAN // TRUE if objects have the same elements
//
// Modifications
//

- subarray min:NUMERIC to max:NUMERIC :SELF // Create a subarray
- append other:SELF // Append other array
- add_last element:E // Append element
- put element:E to i:NUMERIC // Put element at position i
- clear // Count is reseted, but not capacity

```

4.9 STD_INPUT

STD_INPUT is used to modelize the standard input for the program. You can use directly the master object STD_INPUT when calling slots. Clone of this prototype is useful only if you have multi inputs.

```

- read_character:CHARACTER // return the character read
- read_line_in str:STRING // put the line read in str (must be not NULL)
- last_integer // last integer read
- read_integer // read integer and put result to last_integer

```

Examples: use of functions

```

+ c:CHARACTER;
c := STD_INPUT.read_character;

```

4.10 STD_OUTPUT

STD_OUTPUT is used to modelize the standard output for the program. You can use directly the master object STD_OUTPUT when calling slots. Clone of this prototype is useful only if you have multi outputs.

```
- put_character c:CHARACTER          // write a single character on the output
- put_string s:ABSTRACT_STRING      // write a string
- put_new_line                       // write a new line
```

Examples: use of output

```
STD_OUTPUT.put_character 'Y';
STD_OUTPUT.put_string "Hello world !";
```

4.11 COMMAND_LINE

COMMAND_LINE represents the command line of the call of the executable. If you have to get arguments of the command, use this prototype.

```
- count:INTEGER                      // number of arguments
- item idx:INTEGER :STRING_CONSTANT // argument number idx
                                     // name of the executable is 0,
                                     // first argument is 1
```

Example: use of functions

```
COMMAND_LINE.item 1.print;
```

4.12 Default values

<i>Type</i>	<i>Value</i>
NUMERIC	0
CHARACTER	'\0'
BOOLEAN	FALSE
FALSE	FALSE
<i>nothing</i>	<i>() or VOID</i>
<i>other object</i>	NULL

Chapter 4. The LISAAC Library

Chapter 0101b

The LISAAC World

5.1 Glossary of useful selectors

This glossary lists some useful selectors. It is by no means exhaustive.

Name: *Arity:* *Associativity:* *Semantics:*

5.1.1 Assignment

<code>:=</code>	binary	right	Assignment with value
<code>?=</code>	binary	right	Assignment with value or NULL if bad type
<code><-</code>	binary	right	Assignment with code

5.1.2 Cloning

<code>clone</code>			create a clone
--------------------	--	--	----------------

5.1.3 Comparisons

<code>=</code>	binary	left	reference identity
<code>!=</code>	binary	left	not equal (reference)
<code>==</code>	binary	left	structural equality (first level)
<code>!==</code>	binary	left	not equal (structural)
<code><</code>	binary	left	less than
<code>></code>	binary	left	greater than
<code><=</code>	binary	left	less than or equal
<code>>=</code>	binary	left	greater than or equal
<code>hash_code</code>			hash value

5.1.4 Numeric operations

+	binary	left	add
-	binary	left	subtract
*	binary	left	multiply
/	binary	left	divide
%	binary	left	modulus
**	binary	left	exponential
+	unary	right	positive
-	unary	right	negative

5.1.5 Logical operations (BOOLEAN) (see 5.2.1)

&	binary	left	and (strict, total evaluation)
&&	binary	left	and then (semi-strict)
	binary	left	or (strict, total evaluation)
	binary	left	or else (semi-strict)
^ or ^^	binary	left	xor
->	binary	left	imply
=>	binary	left	imply a block
!	unary	right	not (negation)

5.1.6 Bitwise operations (INTEGER)

&	binary	left	bitwise and
	binary	left	bitwise or
^	binary	left	bitwise xor
~	unary	right	bitwise complement
<<	binary	left	logical left shift (filled low bits by zero)
>>	binary	left	logical right shift (filled high bits by zero)

5.1.7 Control

Conditional (see 5.2.2)

A.if_true B	evaluate B if A is True, no return value
A.if_false B	evaluate B if A is False, no return value
A.if B	evaluate B if A is True, result is receiver A
A.if B else C	evaluate B if A is True, C if A is False
A.if B.elseif C then D	evaluate first arg if False, if arg is True then second arg is evaluate, result is the first arg evaluation
A.if B.elseif C then D else E	evaluate first arg if False, if arg is True then second arg is evaluate, else the third arg is evaluate

A.when V then B	once the receiver is equal to first argument, the second one is evaluated
A.when V1 to V2 then B	if the receiver is in the interval V1-V2, the last argument is evaluated
A.when V1 or V2 then B	if the receiver is V1 or V2, the last argument is evaluated

Basic looping (BLOCK) (*see 5.3*)

loop repeat the block forever

pre-tested looping (BLOCK) (*see 5.3.1*)

A.while_do B	while receiver A evaluates to True, repeat the block B argument
A.until_do B	while receiver A evaluates to False, repeat the block B argument

post-tested looping (BLOCK) (*see 5.3.2*)

B.do_while A	repeat the receiver block B while the argument A evaluates to True
B.do_until A	repeat the receiver block B until the argument A evaluates to True

Iterators (INTEGER) (*see 5.3.3*)

V1.to V2 do B	iterate forward
V1.to V2 by S do B	iterate forward, with stride
V1.downto V2 do B	iterate backward
V1.downto V2 by S do B	iterate backward, with stride

5.1.8 Debugging

?	unary	right	crash if argument expression is False (BLOCK)
? B	binary		crash if block is False and level of debug higher (NUMERIC)

5.2 Control Structures: Booleans and Conditionals

5.2.1 Booleans expression

The boolean expression occurs by sending of message to TRUE or FALSE object.

```
test := ((a | b) & c) -> d;  
test2 := ((i>3) | (j<=20));
```

In this example, all the expressions are evaluated.

Typically, there is a “or” and “and” operators which evaluates that by need the right part of the expression.

```
test := (a || {! b}) && {c -> test};  
// If a is False then '! b' is evaluate.  
// If (a || ! b) is True then 'c -> test' is evaluate.  
  
test2 := ((i>3) || {j<=20});  
// If (i>3) is False 'j<=20' is evaluate.
```

5.2.2 Conditionals

A fundamental control structure in LISAAC , like in many languages, is the conditional. In LISAAC , the behavior of conditionals is defined by two unique boolean objects, TRUE and FALSE. Boolean objects respond to the **if else** message by evaluating the appropriate BLOCK argument.

For example, TRUE implements **if else** this way:

```
- if true_block:BLOCK else false_block:BLOCK <- true_block.value;
```

That is, when TRUE is sent the **if else** message, it evaluates the first block and ignores the second. Conversely, the **if else** implementation in FALSE is:

```
- if true_block:BLOCK else false_block:BLOCK <- false_block.value;
```

5.3 Loops

The numerous ways to do loops in LISAAC , enumerated in section 5.1 above, are best illustrated by examples.

5.3.1 Pre-tested looping

Here are two loops that test for their termination condition at the beginning of the loop:

```
{ conditional expression }.while__do { /* ... */ };  
  
{ conditional expression }.until__do { /* ... */ };
```

In each case, the block that receives the message repeatedly evaluates itself and, if the termination condition is not met yet, evaluates the argument block. The value returned by both loop expressions is **void**. **while__do** tests the condition and loops while it is true, whereas **until__do** tests the condition and loops until it is true. In both case, since the test is done before any looping, the loop block may not be executed at all.

For illustration purposes, here is the implementation of the **while__do** message in BLOCK:

```
- while__do loop_body:BLOCK <-  
  ( ? {loop_body != NULL};
```

```

    Self.value.if {
      loop_body.value;
      Self.while_do loop_body;
    };
  );

```

Of course, `self` is optional.

5.3.2 Post-tested looping

It is also possible to put the termination test at the end of the loop, ensuring that the loop body is executed at least once:

```

{ /* ... */ }.do_while { conditional expression };

{ /* ... */ }.do_until { conditional expression };

```

5.3.3 Iterators looping

```

1.to 10 do { i:INTEGER;
  /* ... */
};

10.downto 1 do { i:INTEGER;
  /* ... */
};

```

The 'i' argument of the block of execution contains the current value of the iteration.

5.4 Collections

5.4.1 List of collections

ARRAY : 1-dimension resizable array
 ARRAY2: 2-dimension resizable array
 ARRAY3: 3-dimension resizable array
 FIXED_ARRAY : 1-dimension fixed array
 FIXED_ARRAY2: 2-dimension fixed array
 FIXED_ARRAY3: 3-dimension fixed array
 LINKED_LIST : 1 way linked list
 LINKED2_LIST: 2 ways linked list
 SET: mathematical set of hashable objects
 DICTIONNAY: associative memory

5.4.2 Example

```

+ a:FIXED_ARRAY[INTEGER];
+ b:INTEGER;

```

Chapter 5. The LISAAC World

```
a := FIXED_ARRAY[INTEGER].create 10;  
a.put 5 to 0;  
a.put 2 to 1;  
b := a.item 0;
```

Bibliography

- [Hum90] R. Hummel. Interruption and exception. In *Intel486 Microprocessor Family Programmer's Reference Manual*, pages 83–104, 1990.
- [Mey94] Bertrand Meyer. *Eiffel, The Language*. Prentice Hall, 1994.
- [PBy00] H. Dubois . . . P. Borovansk y, H. Cirstea. Library reference manual. In *ELAN*, pages 20–24, 2000.
- [Son00] B. Sonntag. <http://www.isaacOS.com>. Site web: Isaac (Object Operating System)., 2000.
- [US87] D. Ungar and R. Smith. Self: The Power of Simplicity. In *2nd Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*, pages 227–241. ACM Press, 1987.

Bibliography

Index

- :=, 74
- ?, 88, 111
- ?=, 74, 76
- <-, 74

- Argument, 13
- Argument, list, 80
- Assertion, 88
- Assignment, 14, 74, 109
- Assignment, list, 81
- Assignment, slot, 59, 63, 66
- Auto cast, 87

- Binary message, 71
- Bitwise operation, 110
- Block, 83
- BLOCK, prototype, 102
- Block, return value, 84
- Boolean, expression, 111
- BOOLEAN, prototype, 100

- C Code, 93, 95
- Call of a slot, 13
- Call of slot, 81
- Character, 28
- CHARACTER, prototype, 100
- Class, 9
- Clone, 21, 32, 109
- Collection, 113
- COMMAND_LINE, prototype, 107
- Comparison, 109
- Compilation, 16
- Conditional, 19, 85, 110, 112
- Contract, programming by, 87

- Default value, 58, 107
- do until, 113
- do while, 20, 85, 113

- Eiffel, 5
- Ensure, 89

- Escape sequence, 28
- Evaluation, delayed, 74
- Evaluation, immediate, 74
- Expression, 79, 85
- External, 49, 93

- Final elements, grammar, 27
- FIXED_ARRAY, prototype, 105
- Function, 79, 80

- Genericity, 55
- Grammar, 30

- if else, 19, 112
- Inheritance, 23, 32
- Inheritance, accessibility, 52
- Inheritance, evaluation of parents, 42
- Inheritance, expanded, 38
- Inheritance, non shared, 24, 36
- Inheritance, programming by contract, 92
- Inheritance, shared, 23, 35
- Interruption, 48
- Invariant, condition, 90
- Invariant, type, 55
- Iterator, 111, 113

- Keyword, 70

- Late binding, 74
- Library, 97
- Lisaac external, 96
- List, 77
- List, argument, 80
- List, assignment, 81
- List, local variable, 82
- List, return value, 78
- Local variable, 16, 82, 86
- Logical operation, 110
- Lookup algorithm, 44
- Loop, 20, 85, 111, 112

- Mapping, 47

Index

- Message, binary, 71, 77
- Message, implicit-receiver, 75
- Message, resend, 46
- Message, send, 74
- Message, unary, 72, 77
- Method, 79

- NATIVE_ARRAY, prototype, 102
- Number, 28
- Numeric operation, 110
- NUMERIC, prototype, 98

- Object, 9, 10
- Object Oriented Language, 20
- OBJECT, prototype, 97
- Old, keyword, 91
- Operation, bitwise, 110
- Operation, logical, 110
- Operation, numeric, 110

- Print, function, 18
- Private, section, 50
- Prototype, 9
- Public, 50

- Read, function, 18
- Require, 89
- Result, keyword, 91
- Run, 16

- Section External, 49
- Section Header, 31
- Section identifier, 30
- Section Inherit, 32
- Section Interrupt, 48
- Section Mapping, 47
- Section Private, 50
- Section Public, 50
- Section SELF, 50
- Section *prototype list*, 50
- Section, other, 50
- self, object, 21
- SELF, prototype, 56
- SELF, section, 50
- Semantic, 27
- Slot, 11, 58, 70
- Slot, argument, 13
- Slot, assignment, 14, 59, 63, 66
- Slot, call, 13, 81
- Slot, expanded, 64
- Slot, external, 93
- Slot, keyword, 70
- Slot, non shared, 62
- Slot, redefinition, 34
- Slot, shared, 58
- Slot, simple, 11
- Slot, visibility, 43
- STD_INPUT, prototype, 106
- STD_OUTPUT, prototype, 107
- String, 29
- STRING, prototype, 103
- Syntax, 29

- Type, 17, 55
- Type, invariant, 55
- Typing rules, 75

- until do, 112

- Variable, local, 16, 82, 86

- while do, 112