# Lisaac 0.1
# Programmer's Reference Manual

The power of simplicity at work for Operating Systems

Benoît Sonntag[1],
Dominique Colnet, Olivier Zendra, Jerome Boutet

12th September 2003

[1]This work is at the core of Benoît Sonntag's ongoing PhD.

# Contents

# Chapter 1

# Introduction

The design as well as the implementation of the Isaac[1] operating system [Son00] led us to design a new programming language named LISAAC [2]. Many features of the LISAAC language come from the Self programming language [US87]. Aside Self's capacities, LISAAC integrates communications protection mechanisms as well as other tools related to the design of operating systems. System interruptions support as well as drivers memory mapping have been considered in the design of LISAAC . The use of prototypes and especially dynamic inheritance fit a flexible operating system in the making. The first benchmarks of our 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 very nature of current operating systems comes from studies, languages, hardware and needs going back to a number of years. 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.

Operating systems (OSes) require high performance in terms of execution speed and memory usage, but also simple internal low-level operations. We also believe that object-oriented operating systems must not be on top of a virtual machine (VM) but directly installed on hardware components, because it is essential to be able to reach the very best performances. It is desirable and possible to fully use the hardware in order to provide, at the operating system level, services that are currently supplied by software layers (garbage collector, inter process communication, common memory buffers, ...).

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.

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. Actually, two languages are at the origin of LISAAC : the Self language [US87] for its flexibility and the Eiffel language [Mey94] for its static typing and security. 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. Compared to Self, LISAAC is a bit limited, especially in the way that source files are compiled.

[1]Isaac: Object-oriented Operating System.
[2]LISAAC : Object-oriented, prototype-based, language, used to develop the Isaac OS.

## 1.2 The LISAAC compiler

## 1.3 The Roots of Behavior

The entrance point of a Lisaac program is not fixed as in much language (*main* in C or Java language).

Under UNIX, in the main program, there must be only one slot in the PUBLIC section, which is the entrance point.

Under ISAAC OS, every slot declared in the PUBLIC section, in the MACRO Object are the entrance points.

# Chapter 2

# Quickstart - for beginners

## 2.1 Use of the compiler: cookbook

## 2.2 How to write

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

```
section HEADER
  - name := HELLO_WORLD;      // it's the program name...

section PUBLIC

  - main := "Hello world !".print;
```

In this first LISAAC program, **main** is the name of a method (or slot that contains executable code) and is the root of the system, or beginning of execution (main program).

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

Comments beginning with '//' and end at line's end.

See chapter 4 for more explanation.

## 2.3 How to read

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

```
section HEADER
  - name := HOW_TO_READ;

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.

Note that the list of instructions that **main** comprises is between parentheses — '(' and ')'. See section 3.2.5 on page 26 for more information on instruction lists.

## 2.4 Conditionals: *if else*

A basic control structure in many languages is the `if - then - else` construct. In LISAAC , the `then` is omitted; the appropriate construction[1] is thus:

```
section HEADER
  - name := IF_ELSE;

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 "You are a male !";     // then part...
      } else {
        IO.put_string "You are a female !";   // 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 3.2.5.2 page 27 for local variable declaration in lists of instructions.

The conditional is made of a boolean expression `(gender == 'M')` to which the message **if else** is sent. See section 3.2.4 about message send, and section 4.3 page 45 about booleans and conditionals.

Note that a list of instructions and an expression are the same syntactical construct, between parentheses. See section 3.2.5.1 page 26 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 3.2.6).

## 2.5 A loop: *do_while*

Here is a conditional loop in LISAAC :

```
section HEADER
  - name := DO_WHILE;

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 "You are a male !";
```

[1]**if else** in LISAAC is not a language construct *per se*, but a simple method call.

```
    } else {
      IO.put_string "You are a female !";
    };
  );
```

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 4.4, page 46.

## 2.6 Methods and late binding

Methods( or routines) are a fundamental notion in object-oriented languages, together with their companion concept late binding (or message send, routine call, method call, dynamic dispatch...).

Here is the definition and use of a **get_age** method in LISAAC :

```
section HEADER
  - name := FIRST_METHOD;

section PRIVATE

  - get_age birth_year:INTEGER :INTEGER <-    // method definition...
    //          ^ argument         ^ return type
    (
      2002 - birth_year    // assuming 2002 is the current year...
    );

section PUBLIC

  - main :=
    ( + year, age:INTEGER;

      IO.put_string "Enter your birth year (1976 for example) : ";

      {
        IO.read_integer;
        year := IO.last_integer;
      }.do_while {year < 1900};

      age := get_age year;   // method call...
      IO.put_string "You are ";
      IO.put_integer age;
      IO.put_string " years old.";
    );
```

Remember that a method is a slot containing code. The **get_age** method has one integer argument, `birth_year`, and returns an integer value. For more information on slots, methods and method calls, see section 3.2.3 page 21 and section 3.2.4 page 25.

# Chapter 3

# 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:

| *Symbol* | *Function* | *Description* |
|---|---|---|
| ( ... ) | grouping | a group of syntactic constructions |
| [ ... ] | option | an optional construction |
| { ... } | repetition | a repetition (zero or more times) |
| \| | alternative | separates alternative constructions |
| $\rightarrow$ | 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:

| *Symbol* | *Description* |
|---|---|
| *section* | section identifier |
| *identifier* | slot name, ... |
| *operator* | unary or binary operator symbol |
| *integer* | constant of type INTEGER |
| *cap_identifier* | type name: name of object or prototype |
| *characters* | constant of type CHARACTER |
| *string* | constant of type STRING |
| *external* | external C code |
| *affect* | symbol assignment slots |
| *priority* | associativity priority (binary operator) |
| *style* | Clone comportement |

Note the comments beginning with '//' and end at line's end. To put in comment part of a program, one uses the sequence '/*' for beginning of the comment and '*/' for the end.

```
section          →  'PRIVATE' | 'PUBLIC' | 'KERNEL' | 'HEADER'
                 |  'DRIVER' | 'NETWORK' | 'APPLICATION'
                 |  'MAPPING' | 'INTERRUPT' | 'INHERIT'
identifier       →  LOWER_CASE { ( LOWER_CASE | DECIMAL_DIGIT ) }
                 |  'self'
operator         →  OP_CHAR { OP_CHAR }  except affect symbol
integer          →  OCTAL_DIGIT { OCTAL_DIGIT } 'o'
                 |  DECIMAL_DIGIT { DECIMAL_DIGIT } [ 'd' ]
                 |  HEXA_DIGIT { HEXA_DIGIT } 'h'
cap_identifier   →  UPPER_CASE { ( UPPER_CASE | DECIMAL_DIGIT ) }
characters       →  ''' { NORMAL_CHAR | ESCAPE_CHAR } '''
string           →  '"' { NORMAL_CHAR | ESCAPE_CHAR } '"'
external         →  '`' { NORMAL_CHAR | ESCAPE_CHAR } '`'
affect           →  ':=' | '<-' | '?='
priority         →  'left' | 'right'
style            →  '*' | '+' | '-'
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      →  any character except '\' and '''
ESCAPE_CHAR      →  '\t' | '\b' | '\n' | '\f' | '\r' | '\v' | '\a'
                 |  '\0' | '\\' | '\'' | '\"' | '\?'
                 |  NUMERIC_ESCAPE
NUMERIC_ESCAPE   →  '\'integer'\'
```

### 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.) The LISAAC grammar is immediately followed by an example to illustrate the syntax (3.1.3).

```
PROGRAM        →  { 'section' ( section | { TYPE ',' } TYPE ) { SLOT } }
SLOT           →  style TYPE_SLOT [ ':' TYPE ] [ affect EXPR ] ';'
TYPE_SLOT      →  identifier [ LOCAL { identifier LOCAL } ]
               |  ''' operator ''' [priority [integer]] [identifier ':' TYPE]
LOCAL          →  { identifier [ ':' TYPE ] ',' } identifier ':' TYPE
TYPE           →  cap_identifier [ '[' { TYPE ',' } TYPE ']' ]
EXPR           →  { identifier { identifier } affect } EXPR_INFIX
EXPR_INFIX     →  { EXPR_PREFIX operator } EXPR_PREFIX
EXPR_PREFIX    →  { operator } EXPR_MESSAGE
EXPR_MESSAGE   →  ( EXPR_PRIMARY | SEND_MSG ) { '.' SEND_MSG }
EXPR_PRIMARY   →  integer | characters | string | TYPE
               |  '(' GROUP ')' | '{' GROUP '}'
               |  external [ ':' ['('] TYPE [ '(' { TYPE ',' } TYPE ')' ] [')']]
GROUP          →  { ' LOCAL ';' } { style LOCAL ';' } { EXPR ';' } [ EXPR ]
SEND_MSG       →  identifier [ L_ARGUMENT { identifier L_ARGUMENT } ]
L_ARGUMENT     →  { ARGUMENT ',' } ARGUMENT
ARGUMENT       →  EXPR_PRIMARY | identifier
```

### 3.1.3 Sources examples

This section presents some examples to illustrate syntax. These examples will be used to describe semantics.

#### 3.1.3.1 Integer number notation

| *Style* | *Example* |
|---|---|
| bits number | 10110010b |
| octal number | 1743o |
| hexadecimal number | 17FCh , 0BEBEh , 0baffeh |
| decimal number | 255 , 128d |

#### 3.1.3.2 Hello world

```
section HEADER
  - name := HELLO_WORLD;  // name is mandatory

section PUBLIC

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

#### 3.1.3.3 Quicksort

```
section HEADER
  - name := QUICKSORT;          // name is mandatory...
      // but other slots in HEADER section are optional
  - category := MICRO;
  - date := "Oct 28 2001";
  - version := 1;
  - comment := "The quick sort example in Lisaac.";
  - author := "Benoit Sonntag.";

  section PRIVATE

  + tableau:ARRAY[CHARACTER];

  - qsort tab:ARRAY[CHARACTER] from low:INTEGER to high:INTEGER <-
    ( + i, j:INTEGER;
      + x, y:CHARACTER;
      ? {tab!=NULL};
      ? {low>=tab.lower};    // assertions...
      ? {high<=tab.upper};

      i := low;
      j := high;
      x := tab.item ((i + j) >> 1);
      {
        {tab.item i < x}.while_do {
          i := i + 1;
        };
        {x < tab.item j}.while_do {
          j := j - 1;
        };
        (i <= j).if {
          y := tab.item i;
          tab.put (tab.item j) to i;
          tab.put y, j;
          i := i + 1;
          j := j - 1;
        };
      }.do_while {i <= j};

      (low < j).if {
        qsort tab from low to j;
      };
      (i < high).if {
        qsort tab from i to high;
      };
    );

section PUBLIC

  - main :=
    ( + j, size:INTEGER;

      size := 200000;

      tableau := ARRAY[CHARACTER].clone;
      tableau.make_capacity size;

      j := tableau.lower;
      {j < size}.while_do {
        tableau.put(CHARACTER.random),j;
        j := j + 1;
      };

      qsort tableau from 1 to size;
    );
```

---

## 3.2 Semantics of Lisaac

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 (3.2.6) 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 3.2.12 page 39), the object in the slot is evaluated.

### 3.2.1 Section 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. 3.2.1.1)
- the mode of application of the *lookup* mechanism: inheritance slot (cf. 3.2.1.2) or normal message slot
- the protection and accessibility level of the slots (cf. 6.1.1)
- the compilation mode of the code: remote code mode, close code mode or hardware interruption/exception mode 6.1.2, data structure mapping mode 6.1.3, . . .

#### 3.2.1.1 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; see the Quicksort HEADER section, page 15 for examples. The `category` slot indicates the category of the prototype with respect to its level of protection related to the other prototypes (cf. 6.1.1 page 52).

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 (see fig. 3.1 below).

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

| *Slot name* | *Type* | *Description* |
|---|---|---|
| 'name' | PROTOTYPE | prototype's name (*mandatory*) |
| 'category' | MACRO, MICRO | category for compiler |
| 'privilege' | KERNEL, DRIVER<br>APPLICATION, NETWORK | protection level<br>default is APPLICATION |
| 'version' | INTEGER | version number |
| 'date' | STRING | release date |
| 'comment' | STRING | Comment |
| 'author' | STRING | author's name |
| 'bibliography' | STRING | programmer's reference |
| 'language' | STRING | encoding country language |
| 'bug_report' | STRING | bugs report list |
| 'type' | *external* | C equivalent type (if any) |
| 'default' | EXPRESSION | Default value of the prototype |
| 'external' | *external* | C code which will be included in the C compiled file |

Figure 3.1: Conventions on slot names

```
section HEADER
  - name := MY_PROTOTYPE;
  - category := MACRO;
  - privilege := APPLICATION;
  - version := 1;
  - date := "2002/12/19";
  - 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>`;
```

#### 3.2.1.2 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 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.

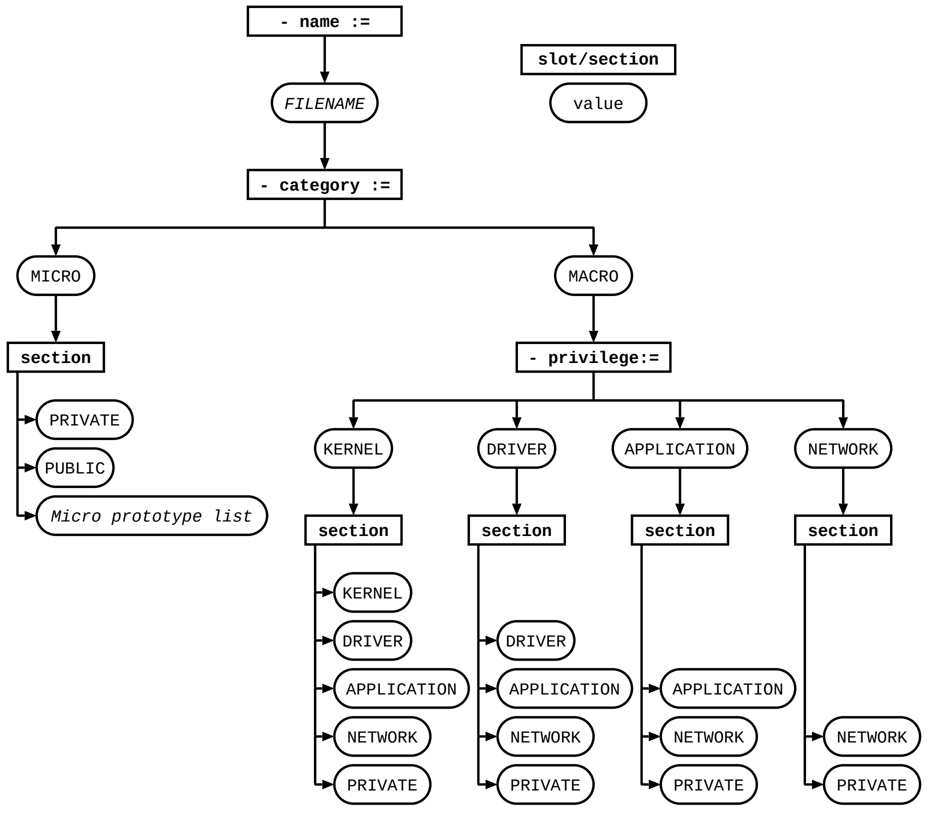


Figure 3.2: Conventions on slot names

⚠ : 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 3.2.12 page 39).

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 3.2.12).

## Share/clonable or immediate/delayed evaluation

⚠ : The evaluation of the heritage slots depends on their order of declaration (see 3.2.7):

- Evaluation with each cloning or after the loading of the prototype:
  - Each created object inherits a different EXPR parent object:

```
section INHERIT
  + parent:EXPR := EXPR; // new one each time...
```

  - All created objects inherit the same BOOLEAN parent object:

```
section INHERIT
```

```
            - main_parent:BOOLEAN := BOOLEAN;  // a single shared parent
```

- 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
    );
  ```

- If the parent is defined with '*' symbol, the parent is cloned. For example:

  ```
  section INHERIT
    * parent:FOO := FOO;
  ```

  If in another prototype we have:

  ```
  FOO := BAR.clone;
  ```

  In our first prototype, the parent is not modified.

**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 (for example fig. 3.3).
About the visibility of the slots, the static tree of heritage shows the slots accessible.

#### 3.2.1.3 The MAPPING section

In this section, you can define (only with the '+' property) attributes which map exactly (size and order) the physical type. You can only define attributes of simple types (numbers or character). Otherwise, these attributes are used exactly as the others not in the mapping section.

```
section MAPPING
  + field1:USMALLINT;
  + field2:UINTEGER;
  + field3:USMALLINT;
```

These prototype match exactly a 6 byte physical structure.

#### 3.2.1.4 The INTERRUPT section

*************** JBJB: je sais pas l'expliquer !!! *************************

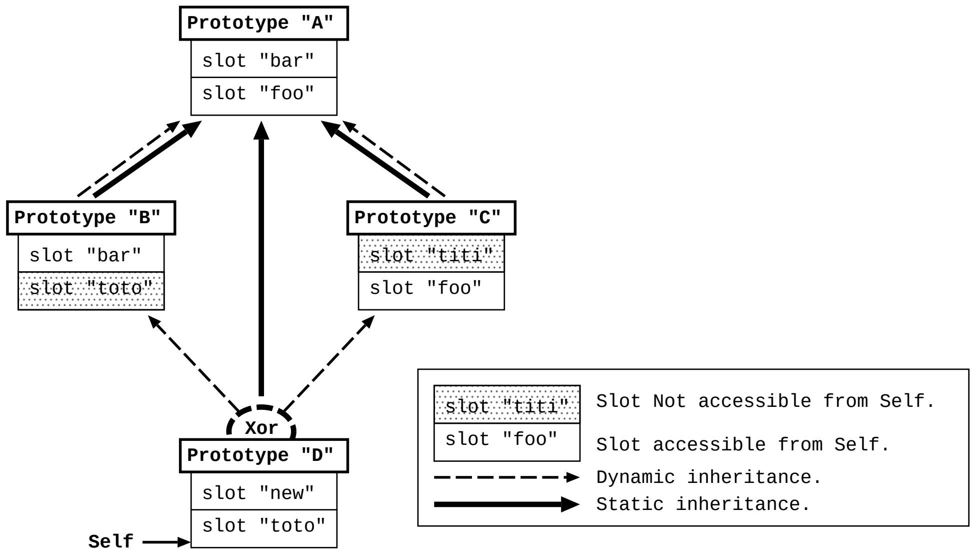


Figure 3.3: Static type and visibility

### 3.2.2 Type names

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

Examples:

+ **color**:INTEGER;

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

#### 3.2.2.1 Invariant's type control

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

---

**Legal example:**

**section** HEADER
  - **name** := FOO;

**section** PUBLIC
  + **to_string** arg:INTEGER :STRING <- ( ... );

**section** HEADER
  - **name** := BAR;

**section** INHERIT
  - **parent** := FOO;

**section** PUBLIC
  + **to_string** arg:INTEGER :STRING <- ( ... );

---

**Illegal example:**

```
section HEADER
  - name := FOO;

section PUBLIC
  + to_string arg:INTEGER :STRING <- ( ... );


section HEADER
  - name := BAR;

section INHERIT
  - parent := FOO;

section PUBLIC
  + to_string arg:REAL :ARRAY[CHARACTER] <- ( ... );
```

---

#### 3.2.2.2 Particular type: SELF type

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

```
section HEADER
  - name := OBJECT;

section PUBLIC
  + clone:SELF <- ( ... );
```

#### 3.2.2.3 Default value of a slot according to its type.

| *Type* | *Value* |
|---|---|
| INTEGER | 0 |
| CHARACTER | '\0' |
| BOOLEAN | FALSE |
| FALSE | FALSE |
| *nothing* | ( ) *or* VOID |
| *other object* | NULL |

A default value can also be defined in the slot DEFAULT in the HEADER SECTION. It can be a value or an expression evaluated at the first call of the prototype.

For example:

```
  - default:= 0;
```

If you use the prototype without initializing it, its value will be '0'.

### 3.2.3 Slot descriptors

An object may hold any number of slots. Slots can contain data (values and references) or methods (code).

Code slot may have arguments, which are separated by commas or by lowercase keywords. Numbers and the underscore are authorized (but the sequence '__' is prohibited). 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 (see the various possibles cases in the examples section 3.2.3.1).

Note that there is overloading is not allowed. Hence, two messages can't differ only by the type of their arguments.

The identifier slots of the objects and the local identifier slots in a block or list instruction are in the same place (single space).

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

The slots preceded by the '+' character obliges the cloning of its value to each cloning of the prototype.

If the slots preceded by the '*' character, its value is cloned and embedded (in memory) in the prototype.

### 3.2.3.1 Keyword slots

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

1. Argumentless slot definition:

   ```
   - get_color:INTEGER <- color;
   ```

   This slot returns an integer value.

2. Definition of slot with argument list and keywords:

   ```
   - 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.

3. Definition of slot with argument list but no keyword:

   ```
   - qsort tab:ARRAY[CHARACTER], low, high:INTEGER <-
     ( ... );
   ```

   This is a kind of variant of 2, where the keyword are omitted and replaced by commas. Although 3 is shorter than 2, it is a bit less clear and thus should be used with some care.

4. Definition of slot with or without keyword in argument list:

   ```
   - qsort tab:ARRAY[CHARACTER] between low,high:INTEGER <-
     ( ... );
   ```

5. Definition of slot with argument and return value :

   ```
   - qsort tab:ARRAY[CHARACTER] :BOOLEAN <-
     ( ... );
   ```

⚠ : Slot '**qsort,,**' and slot '**qsort from to**' are two different, unrelated slots. The "shortcut" version 3 can't be used in place of version 2.

### 3.2.3.2 Binary messages

In LISAAC , it is 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 limit[1]

[1]Except the maximum allowed for 32 bits integers, of course.

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 prioriy 150.

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

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

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));
```

Note that you will find a list of the binary operator more used in the glossary (see 4.1).

#### 3.2.3.3 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;
```

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. Here is the code for the ? unary prefix operator define in BLOCK object:

```
- '?' <-
  (
    (_debug_flag_compiler && {! self.value}).if {
      crash;
    };
  );
```

Note that `_debug_flag_compiler` is a predefined flag set by the compiler according to the parameters chosen by the developper at compile time.

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 developper can switch off these assertions in the final delivery version by using a simple compile-time option.

#### 3.2.3.4 Implicit-receiver messages

Keyword messages are frequently written without an explicit receiver. Such messages use the current object (named `self`) as the implied receiver.

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

#### 3.2.3.5 Slot evaluation

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 3.2.12). They are evalueted 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 or cloned:

  ```
  - 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 or cloned:

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

  If the result is bad type then the result is NULL.

- **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
```

### 3.2.4 Message send, late binding

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

| *Message kind* | *# of Arguments* | *Precedence* | *Associativity* |
|---|---|---|---|
| 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 3.2.3.2 page 22).

⚠ : The priority defined by a 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 3.2.3.1 page 22.

#### 3.2.4.1 Keyword message send

A *keyword message* has a receiver and may have one or several arguments, separated by commas or keywords. Keyword messages have been explained under the *Slot identifier* section 3.2.3 page 21.
Using the slots defined in that section, calls examples are:

1. Argumentless slot call:

   ```
   x := get_color;
   ```

2. Call to a slot with argument list and keywords:

   ```
   qsort my_array from my_array_low to my_array_high;
   ```

3. Call to a slot with argument list but no keyword:

   ```
   qsort my_array, my_array_low, my_array_high;
   ```

4. Call to a slot with and without keyword in the list of argument:

   ```
   qsort my_array between my_array_low, my_array_high;
   ```

⚠ : Bracket should be put the arguments when there is more one item (except for the operators unary).

#### 3.2.4.2 binary message send

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

| *Source code* | *is interpreted as* |
|---|---|
| `2 + "25".`**to_integer** `+ 5` | `( ( 2 + ( "25".`**to_integer** `) ) + 5 )` |
| **object.set_value** `2+2` | `( (` **object.set_value** `2 ) + 2 )` |
| `2+2 .`**to_string** | `( 2 + ( 2.`**to_string** `) )` |

#### 3.2.4.3 Unary message send

| *Source code* | *is interpreted as* |
|---|---|
| **object.set_value** `-2` | `( (` **object.set_value** `) - ( 2 ) )` |
| `-2.`**to_string** | `( - ( 2.`**to_string** `) )` |
| `- + - 2` | `( - ( + ( - 2 ) ) )` |

Other example:

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

### 3.2.5 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 ';'.

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 itself[2]. Consequently, there is absolutely no difference between a one-statement list in LISAAC and an expression as classically defined in most programming languages.

#### 3.2.5.1 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 integers
```

Finally, the left-hand-side of the || operator is a single-statement list that returns a boolean:

```
{(j < upper) || {result != NULL}}.while_do {
  ...
};
```

A list may also have no return value at all:

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

[2]This is the contrary for statement blocks, see section 3.2.6 page 28.

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

#### 3.2.5.2 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.

A dot is use to indicate the declaration of one or several local variables at the beginning of a list:

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

Locals in lists have to be declared at the beginning or 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 '-' preserves their values with each call (variable persistent).

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

#### 3.2.5.3 Arguments in statement lists

The arguments are useful has redefine the code of a method.

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

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

⚠ : Another use is illegal.

### 3.2.6 Statement blocks

Statement blocks, or simply "blocks", have a number of similarities with lists (see section 3.2.5 page 26).

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.

On of the common uses of blocks in LISAAC is to implement library-defined control structures (see section 4.4 page 46).

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..

#### 3.2.6.1 Return values of blocks

The value returned by a block is determined exactly like that of a list (see section 3.2.5.1 page 26).

The following examples thus are quite straightforward.

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.

#### 3.2.6.2 Argument and local variables in statement blocks

Locals in blocks are declared like locals in lists (see section 3.2.5.2 page 27):

```
my_block := {  arg1:INTEGER;   // Arguments list.
               arg2:STRING;
               + j,k:INTEGER;    // Locals list.
               + array:ARRAY[STRING];
              ...
             };
...
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.

### 3.2.7 Objects in memory: clone and assignment

As seen in section 3.2.3 page 21 you have to choose between 3 descriptors for slots, '+', '-' and '*'.

```
section HEADER
  - name := POINT;
section INHERIT
  + parent := GRAPHICS;
section PUBLIC
  + x:INTEGER;
  + y:INTEGER;
  + color:INTEGER;
  + move new_x,new_y:INTEGER <-
    (
      x := new_x;
      y := new_y;
    );
```

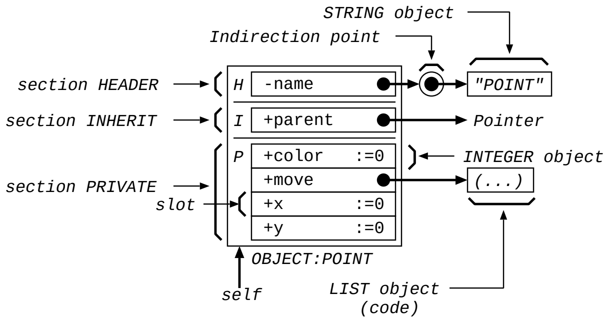


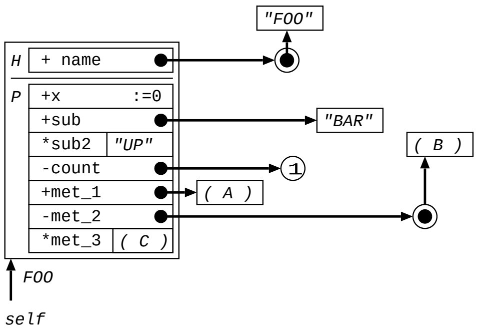
Figure 3.4: Notation for the figures memories

### 3.2.8 Data and method slot

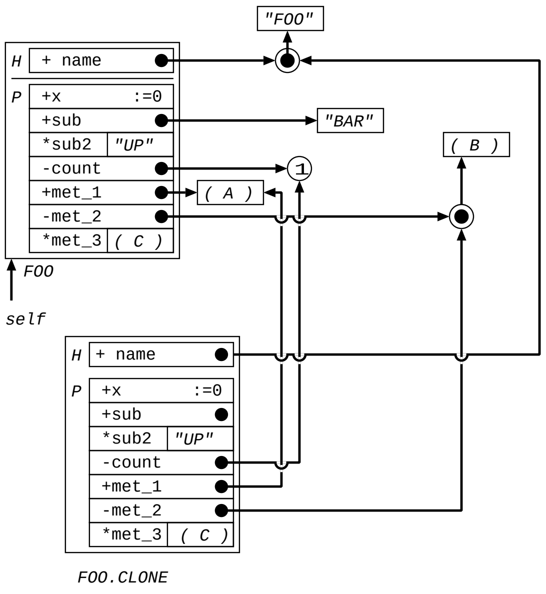
Note that you can't clone a prototype if its name is defined with '-'.

```
section HEADER
  + name := FOO;
section PUBLIC
  + x:INTEGER := 0;
  + sub:STRING := "BAR";
  * sub2:STRING := "HELLO";
  - count:INTEGER := 1;
  + met_1 <- ( ... );
  - met_2 <- ( ... );
  * met_3 <- ( ... );
```

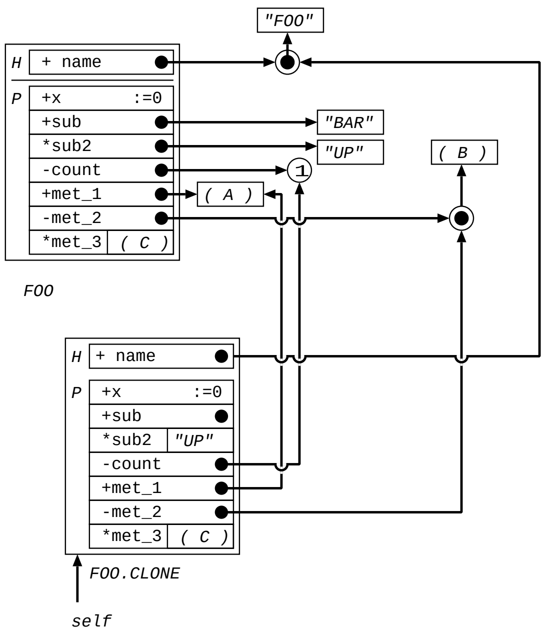
#### 3.2.8.1 clone



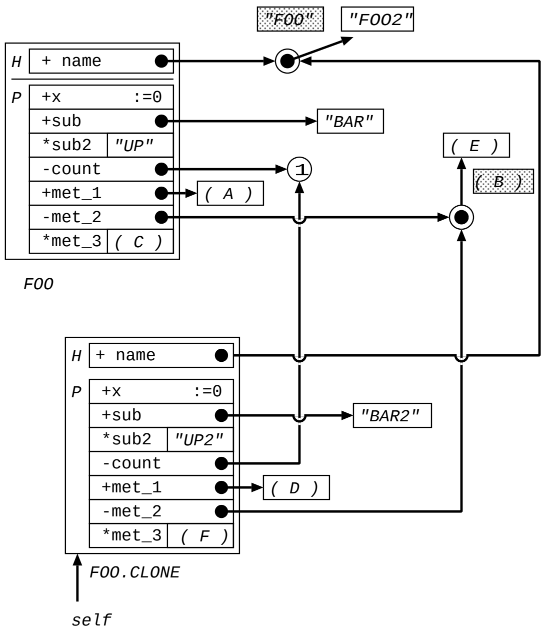
```
new_foo := FOO.clone;
```



### 3.2.8.2 assignment



**name** := *"FOO2"*;
**x** := 3;
**sub** := *"BAR2"*;
**sub2** := *"UP2"*;
**count** := 2;
**met_1** <- ( ... );
**met_2** <- ( ... );
**met_3** <- ( ... );



### 3.2.9 Parent slot: with share

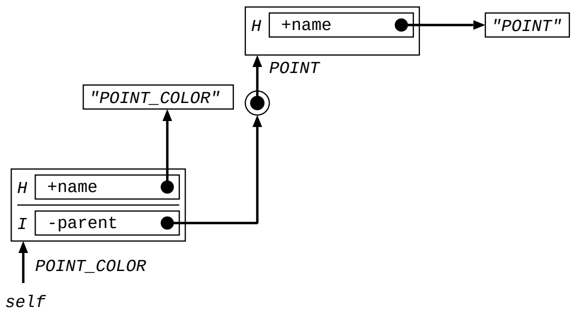
Prototype POINT

```
section HEADER
  + name := POINT;
```

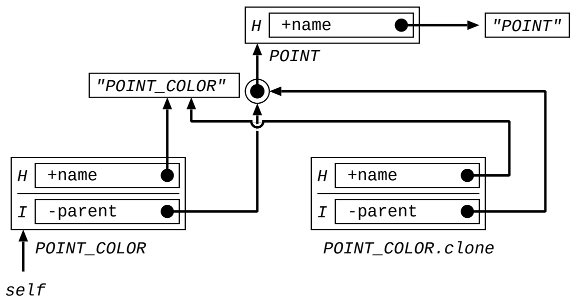
Prototype POINT_COLOR

```
section HEADER
  + name := POINT_COLOR;
section INHERIT
  - parent:POINT := POINT;
```

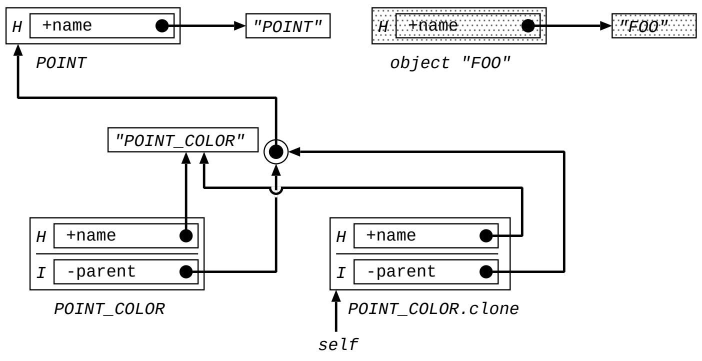
#### 3.2.9.1 clone



```
new_point := POINT_COLOR.clone;
```

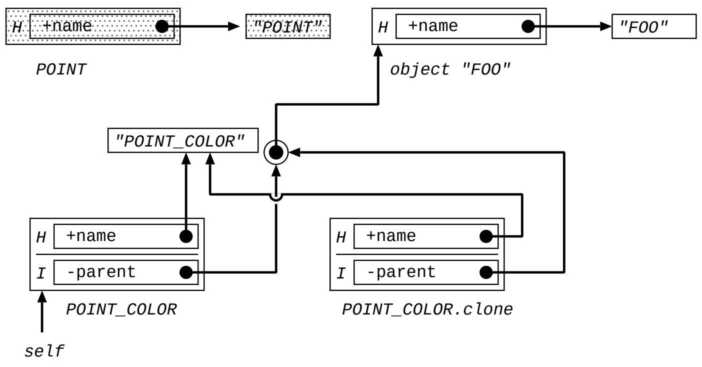


#### 3.2.9.2 assignment



Self is new_point (POINT_COLOR.clone)

**parent** := FOO;



### 3.2.10 Parent slot: with 'clone'

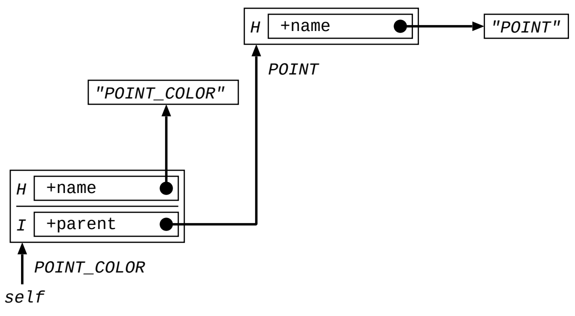
Prototype POINT

**section** HEADER
  + **name** := POINT;

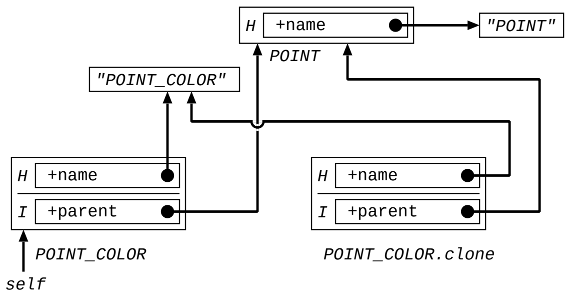
Prototype POINT_COLOR

**section** HEADER
  + **name** := POINT_COLOR;
**section** INHERIT
  + **parent**:POINT := POINT;

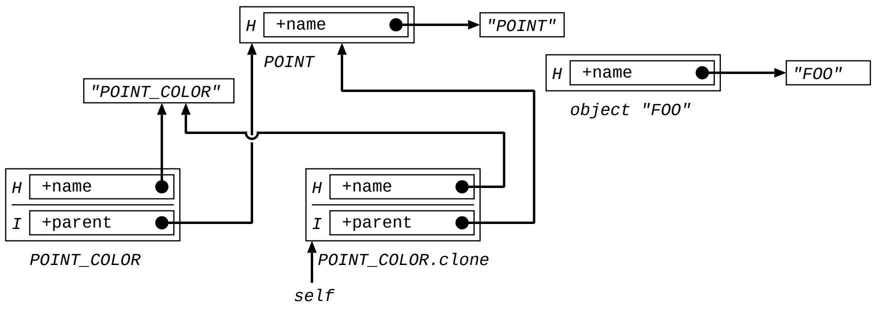
#### 3.2.10.1 clone



new_point := POINT_COLOR.**clone**;

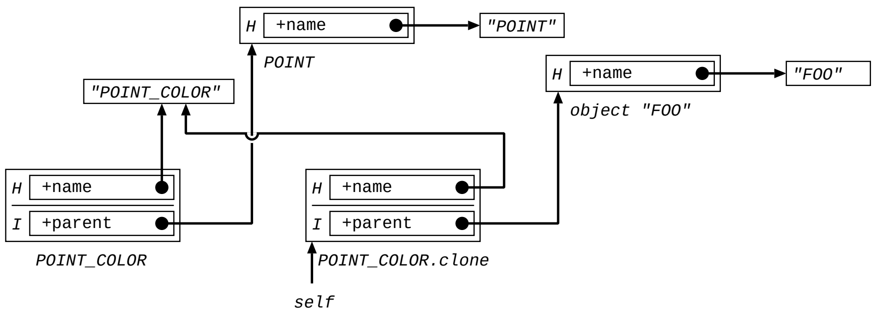


#### 3.2.10.2 assignment



Self is new_point (POINT_COLOR.clone)

**parent** := FOO;



### 3.2.11 Parent slot: with 'auto-clone'

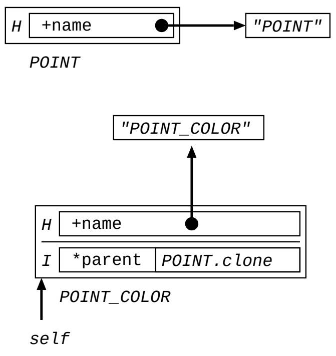
Prototype POINT
The slot NAME of the parent can't be defined with '-' because in this case you can't have PARENT.CLONE

**section** HEADER
  + **name** := POINT;

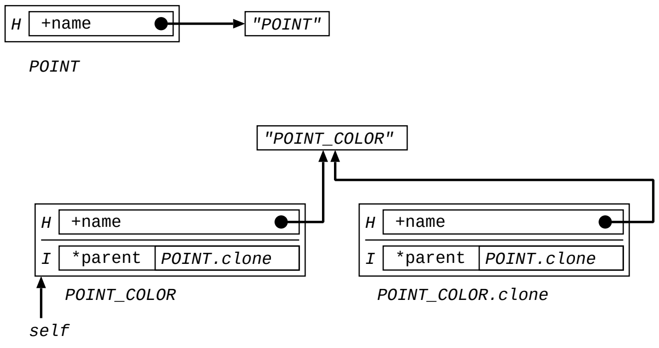
Prototype POINT_COLOR

**section** HEADER
  + **name** := POINT_COLOR;
**section** INHERIT
  * **parent**:POINT := POINT;

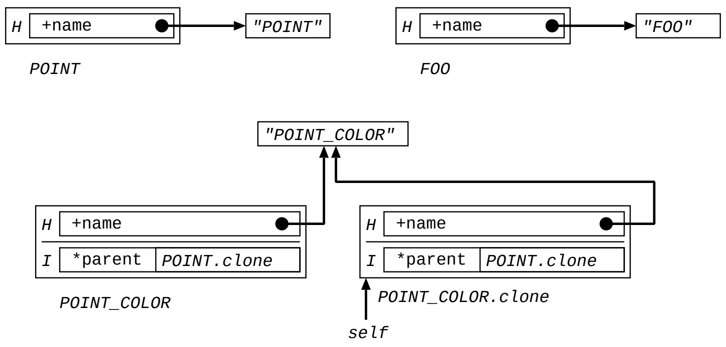
#### 3.2.11.1 clone



new_point := POINT_COLOR.**clone**;

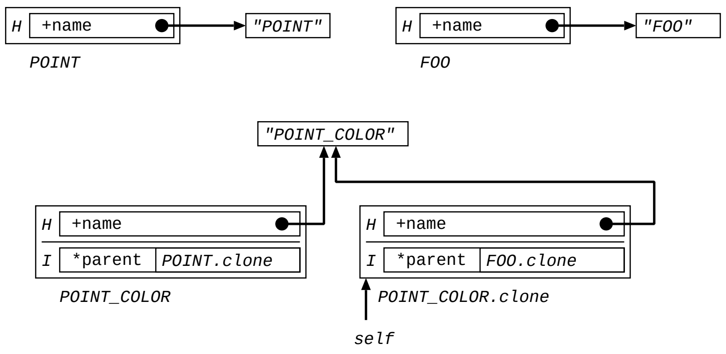


#### 3.2.11.2 assignment



Self is new_point (POINT_COLOR.clone)

**parent** := FOO;



### 3.2.12 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 3.2.3 page 21). 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 developper, 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
  );
```

### 3.2.13 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 he equivalent of *super* in Smalltalk or *resend* in Self. This means that the message is sent to the parent with the current object context (*self* on the figure **??**).

Example of redefinition of a method:

```
section INHERIT

  - main_parent := FOO;

  - second_parent := BAR;

section PUBLIC

  - print :=
    (
      second_parent.print; // I selected method defined in the second parent.
    );

  - old_print :=
    (
      main_parent.print; // Call 'print' of first parent.
    );
```

### 3.2.14 The resend lookup algorithm

Resending message: ...

## 3.3 Auto-cast

In the HEADER section, in the slot name, you can define the prototype which can be "auto-casted" with the '->' symbol. For example:

```
section HEADER
   + name := PROTO1 -> PROTO2,PROTO3;
section PUBLIC
   - to_proto2 <- ( ... )
   - to_proto3 <- ( ... )
```

In the public section, you must have methods called to_proto2, ... (for our example) which are automatically called when there is an autocast.

```
section HEADER
   + name := TEST;
section PUBLIC
   - main :=
(
        + a:PROTO1;
        + b:PROTO2;
        ...
        b := a;   // similar to: b := a.to_proto2;
        ...
);
```

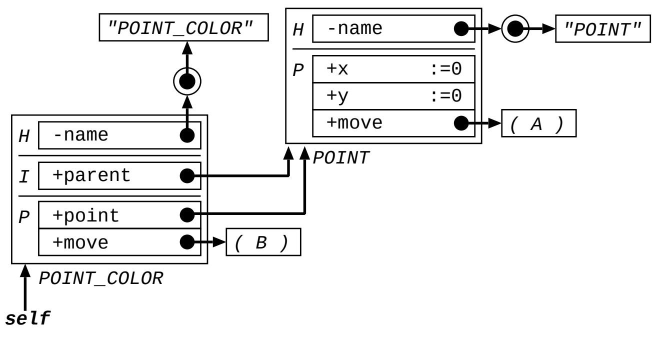


Figure 3.5: Start condition

**point.move** `2,3;`

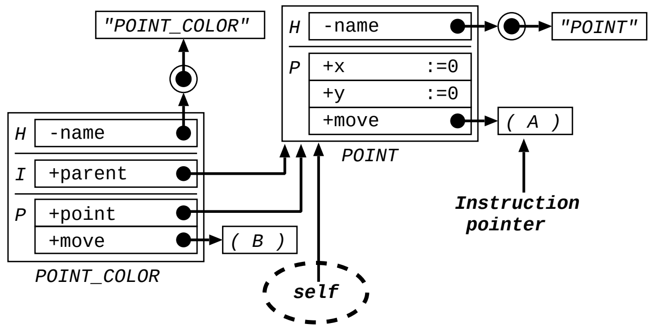


Figure 3.6: During execution of traditional call

**parent.move** `2,3;`

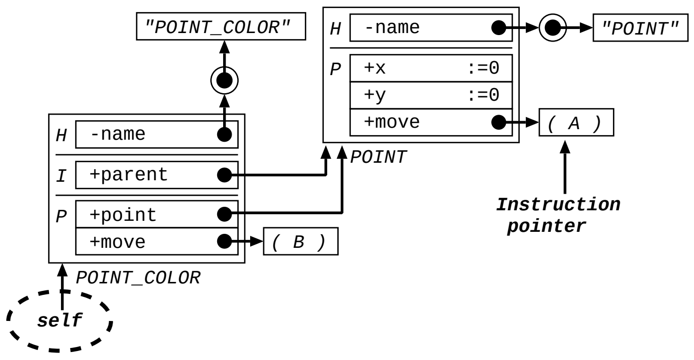


Figure 3.7: During execution of 'super' call

## 3.4 String

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\.

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 transformated by the compiler in: "This is an example for the string"

Chapter 4

# The LISAAC World

## 4.1 Glossary of useful selectors

This glossary lists some useful selectors. It is by no means exhaustive.

| *Name:* | *Arity:* | *Associativity:* | *Semantics:* |
|---|---|---|---|

### 4.1.1 Assignment

| | | | |
|---|---|---|---|
| := | binary | right | Assignment with value |
| ?= | binary | right | Assignment with value or NULL if bad type |
| <- | binary | right | Assignment with code |

### 4.1.2 Cloning

| | | | |
|---|---|---|---|
| clone | | | create a clone |

### 4.1.3 Comparisons

| | | | |
|---|---|---|---|
| = | binary | left | reference identity |
| != | binary | left | not equal (reference) |
| == | binary | left | structural equality (first level) |
| !== | binary | left | not equal (structural) |
| < | binary | left | less than |
| > | binary | left | greater than |
| <= | binary | left | less than or equal |
| >= | binary | left | greater than or equal |
| hash_code | | | hash value |

### 4.1.4 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 |

### 4.1.5 Logical operations (BOOLEAN) *(see 4.3.1)*

| | | | |
|---|---|---|---|
| & | 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 |
| ! | unary | right | not (negation) |

### 4.1.6 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) |

### 4.1.7 Control

#### 4.1.7.1 Selection *(see 4.3.2)*

| | |
|---|---|
| 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 |

#### 4.1.7.2 basic looping (BLOCK) *(see 4.4)*

| | |
|---|---|
| loop | repeat the block forever |

#### 4.1.7.3 pre-tested looping (BLOCK) *(see 4.4.1)*

| | |
|---|---|
| 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 |

#### 4.1.7.4 post-tested looping (BLOCK) *(see 4.4.2)*

| | |
|---|---|
| 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 |

#### 4.1.7.5 Iterators (INTEGER) *(see 4.4.3)*

| | |
|---|---|
| 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 |

### 4.1.8 Debugging (BLOCK)

| | | | |
|---|---|---|---|
| ? | unary | right | crash if argument expression is False |

## 4.2 World Organization

## 4.3 Control Structures: Booleans and Conditionals

### 4.3.1 Booleans expression

The boolean expression occurs by sending of message to TRUE or FALSE object. For example:

```
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.

For example:

```
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.
```

### 4.3.2 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;
```

## 4.4 Loops

The numerous ways to do loops in LISAAC , enumerated in section 4.1 above, are best illustrated by examples.

### 4.4.1 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.

### 4.4.2 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 };
```

### 4.4.3 Iterators looping

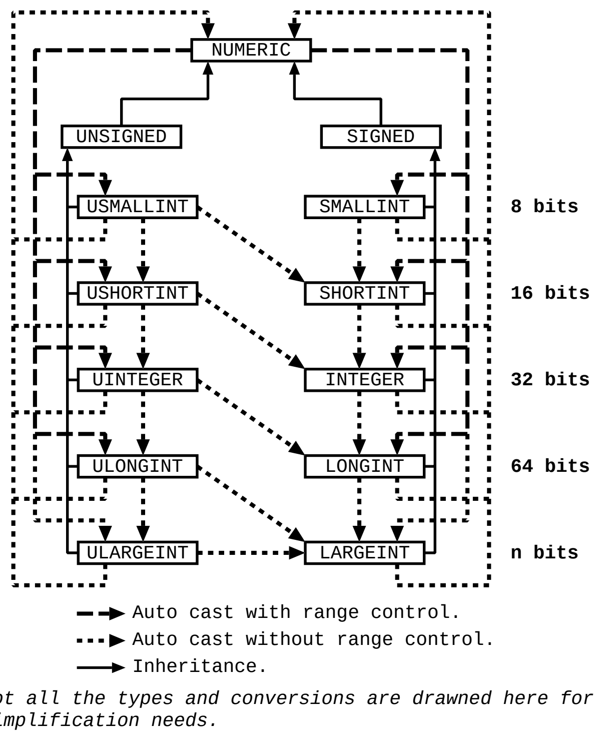
```
1.to 10 do { i:INTEGER;
  ...
};

10.downto 1 do { i:INTEGER;
  ...
};
```

The 'i' argument of the block of execution contains the current value of the iteration.

## 4.5 Collections

### 4.5.1 List of collections

ARRAY : 1-dimension resizable array
ARRAY2: 2-dimension resizable array
ARRAY3: 3-dimension resizable array



*Not all the types and conversions are drawned here for simplification needs.*

Figure 4.1: Number model

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

### 4.5.2 Example

```
+ a:FIXED_ARRAY[INTEGER];
+ b:INTEGER;
a := FIXED_ARRAY[INTEGER].create 10;
a.put 5 to 0;
a.put 2 to 1;
b := a.item 0;
```

### 4.5.3 Numbers

With numbers, the autocast has been implemented only for the case where there is never a problem (unsigned 8 bits in unsigned 16 bits, signed 16 bits in signed 32 bits, ...). For other cases, you must do an explicit cast (see Figure 4.5.3

## 4.6 I/O with Operating System

# Chapter 5

# Compiler Consideration

## 5.1 Usage

## 5.2 Primitive List

| *identifier* | *prototype* | *description* |
|---|---|---|
| 0 | INTEGER x INTEGER → INTEGER | subtract *(mandatory)* |
| 1 | INTEGER x INTEGER → INTEGER | multiply |
| 2 | INTEGER x INTEGER → INTEGER | divide *(mandatory)* |
| 3 | INTEGER x INTEGER → INTEGER | and bitwise *(mandatory)* |
| 4 | INTEGER x INTEGER → INTEGER | or bitwise |
| 5 | INTEGER x INTEGER → BOOLEAN | equal |
| 6 | INTEGER x INTEGER → BOOLEAN | greater than *(mandatory)* |

## 5.3 C glue

To include C code in Lisaac language, there is two way:
The slot EXTERNAL in the HEADER SECTION.
Directly in the Lisaac code as following:
‘C code‘
You can work with Lisaac variable (but only local variable) using ’@’: ‘@my_local‘.
You can also get a result of a function by specifying the Lisaac static type of the result and in brackets the dynamic type if exists.
‘C code‘:STATIC_TYPE(DYNAMIC_TYPE1,DYNAMIC_TYPE2);

Example:

```
‘@a == @b‘:BOOLEAN(TRUE,FALSE);
```

To optimize the generate code, we separate the external C code in 2 types:
• What we call persistent code. It’s a code without return value. The compilation can’t delete it because the code is used by the application.
• The non-persistent code: it’s the code which returns a value. If the result is not used somewhere else in the application, the compilator delete it. But the code inside the external (which is not analysed by the compilator) may be useful. So in that case, you have to specify that you want this external code to be persistent in specifying the type of the return between parenthesis.
‘C code‘:(STATIC_TYPE(DYNAMIC_TYPE1,DYNAMIC_TYPE2));

Example:

```
c := ‘write(@f,@buf,@size)‘:(INTEGER);
```

If ’c’ is not used, the compiler would delete this affectation to optimize the code. Or if this is done, it is an error ! So the result is between parenthesis to ensure the compiler would’nt delete this code.

```
c := ‘write(@f,@buf,@size)‘:(INTEGER);
```

## 5.4 C type

```
section HEADER
  ...
  - type := ‘char‘;
```

The compiler will directly translate the prototype in the type defined (here 'char').

## 5.5 C external

```
section HEADER
  ...
  - external := ‘#include <...>‘;
```

The compiler will directly include the external in the code.

Chapter 6

# LISAAC in Isaac Object Operating System

## 6.1 The LISAAC language and the Isaac system: an overview

It is clear for us that an operating system must use all the power of hardware components. For this reason, the Isaac operating system does not run on top of a virtual machine but directly on hardware components (i.e. directly on the micro-processor), hence the choice of a compiler approach to avoid – at run time – the interpretor overhead.

As in the case of the Eiffel language, an Isaac object is defined by its source code stored in a file of the same name as the object it defines. As in the traditional approach, this file is processed by our LISAAC compiler in order to produce the corresponding executable binary file.

To facilitate the portability of the system, our compiler operates closely with GCC [sc00] and the ELF [elf95] binary linker. A compatibility at the source level for applications written in C has been studied, but the mechanisms involved are not in the scope of this article, nevertheless, we want to stress out that the interfacing used remains coherent with the Isaac object model.

As in Self, a prototype may have a name or can be an anonymous one. All prototypes are clonable and may be modified at runtime. Even the inheritance relationship between prototypes may change at runtime.

Mostly for security reasons as well as for organization of our model, there are two kinds of objects: PRIVATE objects (also called micro-objects) and other objects (called macro-objects). A macro-object is usually used to represent a complex object like some hardware component (e.g. mouse, keyboard, ...) or some complex software component (file, bitmap, ...). Micro-objects are PRIVATEs and are composed in order to constitute some macro-object. Micro-objects are usually simple (e.g. integer, string, ...) and are used for macro-objects implementation. Nevertheless, micro-objects may represent more complex data like linked lists or dictionaries for example.

Actually, as we will see later, the syntax to describe macro-objects and micro-objects is the same. The major difference comes from the compilation strategy used (fig. 6.1). Each macro object is compiled separately to produce the corresponding executable object. All interactions between macro-objects are dynamically typed (i.e. no verification at compile time). Inside some macro-object, all PRIVATE objects are checked statically using an Eiffel-like approach [ZCC97] [CZ99].

Thus, interactions between macro-objects are checked only at runtime. For example, on the Intel architecture [Pro], this is achieved using hardware mechanism. Interactions between micro-objects are checked only at compile time. Those objects are compiled in the context of some macro-object which use them and do not require the creation of a genuine Isaac prototype as such. Finally, to ease the implementation of containers like arrays, linked lists and dictionaries for example, we also added a form of genericity such as the one defined in Eiffel [Mey94].

Dynamic typing allows flexible interaction or communication between operating system components. In LISAAC source code, all macro-objects are all declared with the type mark OBJECT, the most general prototype.

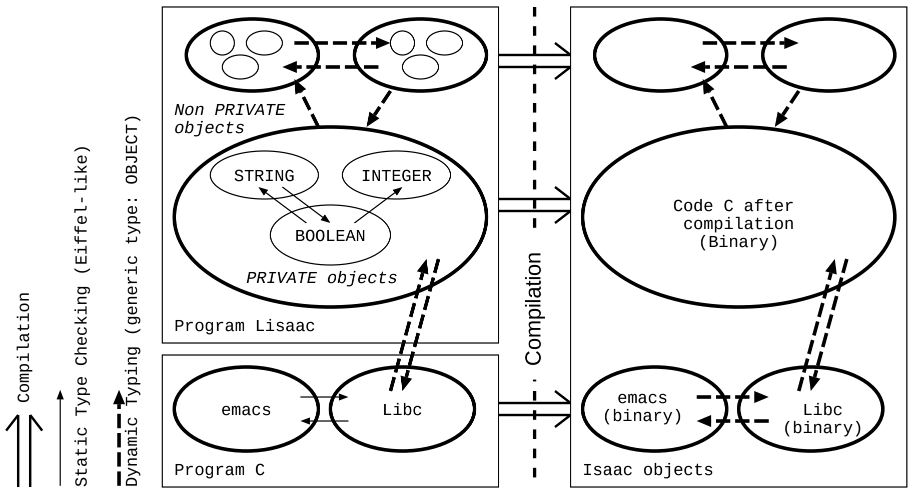


Figure 6.1: Only PRIVATE objects are checked statically when other objects are typed dynamically.

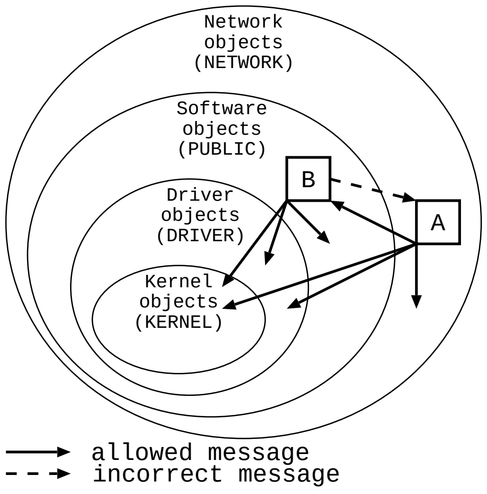


Figure 6.2: The four levels to protect objects communication.

### 6.1.1 Sections PRIVATE, KERNEL, DRIVER, APPLICATION and NETWORK

These names of section define the level of accessibility and protection of the corresponding slots. For example, the slots defined in a section PRIVATE are visible only inside this prototype. They are not accessible from another place, even by descendants (one does not inherit the section PRIVATE).

The other sections (KERNEL, DRIVER, NETWORK and APPLICATION) make it possible to fix the policy of interaction between prototypes knowing that: the PUBLIC category is reserved for the unreliable objects, the KERNEL category is dedicated to the critical objects of the system, the DRIVER category is reserved for *hardware drivers* objects, and finally, the NETWORK category is reserved for the objects coming from a network or being carried out on a distant machine.

Without presenting here in details all the rules of communications (fig. 6.2), let us quote for example that an unreliable object (category APPLICATION) cannot be used by a critical object (category KERNEL). The goal of this rule is to avoid putting in danger the integrity of the system at the time of the call of a functionality of the core.

Thus, there are four levels of protection checked at runtime: KERNEL, DRIVER, APPLICATION and NETWORK. Checking can be achieved by using built-in hardware protection mechanism mixed with software protection when the processor has only two hardware protection levels (intel processor have four level of protection while motorola has only two levels). An assumption of responsibility of protections by the processor and a single space of addressable memory allow a protected and powerful communication between the objects [Son01].

### 6.1.2 Section INTERRUPT

The goal of the INTERRUPT section is to handle hardware interruptions. In such a section code slots are allowed. 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.

### 6.1.3 Section MAPPING

The MAPPING section purpose is to format data slots description according to some fixed hardware data structure. The main goal of MAPPING section is to describe in LISAAC device drivers. 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. Slots inside some MAPPING section are considered private for any other objects.

# Appendix A

# Appendix.

## A.1 The design of Isaac in LISAAC

The intent of this section is to show that a high-level prototype-based language fits very well with an operating system design and implementation. Here are some selected examples from the Isaac operating system as it is implemented in LISAAC .

### A.1.1 Hardware components *versus* software components

In the Isaac system objects hierarchy, true physical hardware objects (e.g. keyboard, mouse, memory, ...) are distinguished from system-software objects (e.g. file, vector, bitmap, ...). In a very natural way, inheritance is used to separate hardware objects from software objects. The reason of this segregation is that hardware objects are not clonable by another PUBLIC object. Using other words: one cannot clone a screen if the physical new screen does not exists. Hardware components are obviously natural critical resources. Conversely, software components, SOFT_OBJECT (fig. A.1), inherit the traditional `Clone` method with PUBLIC accessibility. Also note that the common set of named object is available to all thanks to the general OBJECT prototype.

### A.1.2 Dynamic inheritance at work for video drivers

Figure A.2, represents the Isaac's video architecture. The VIDEO object can change dynamically its parent slot to inherit BITMAP_15 or BITMAP_16 or BITMAP_24 or BITMAP_32 as well. Actually, dynamic inheritance is obviously used to change dynamically the bitmap resolution (one or more times). The reference to the VIDEO object remains unchanged for clients allowing the resolution mode to be changed transparently (i.e. only the parent slot of the VIDEO object is modified).

Moreover, the VIDEO object can redefine any BITMAP's functionalities in order to take as many advantages as possible from the hardware graphic device (graphics accelerator embedded on the board, bit depth, ...).

### A.1.3 The Isaac file system

The implementation of the Isaac file system is another example of dynamic inheritance usage (fig. A.3). The abstract INODE object is inherited by FILE and DIRECTORY as well. The actual INODE object parent slot indicate the appropriate file representation: FAT_16BITS or EXT_2FS for example. Then, the file representation itself may inherit FLOPPY when this inode is on a floppy disk or IDE in the case of some hard disk. Once again, dynamic inheritance is extremely useful and flexible in this case.

### A.1.4 The quick-sort benchmark

Our compiler, while complete, is still under development, and it is not possible to carry out many reliable tests of performances. Nevertheless, a simple test of the quick-sort algorithm is very promising since the performances are equivalent to those of a similar C program. In spite of the absence of hard-coded test/loop statements in LISAAC , the generated C code is very similar to the hand-written C

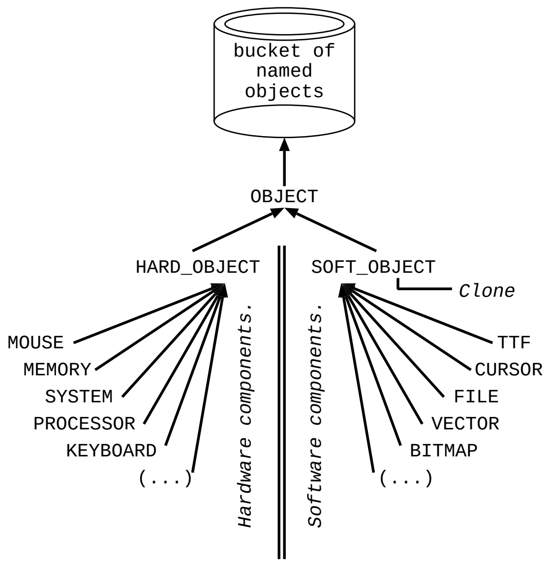


Figure A.1: Hardware components and software components segregation.

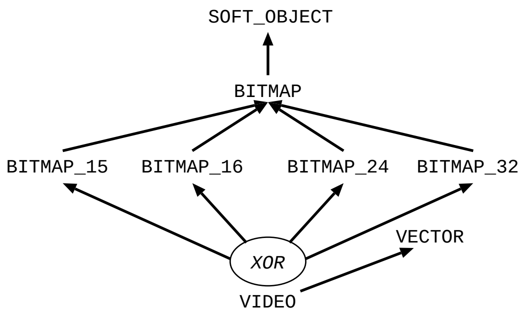


Figure A.2: Dynamic inheritance to select the appropriate VIDEO DRIVER.

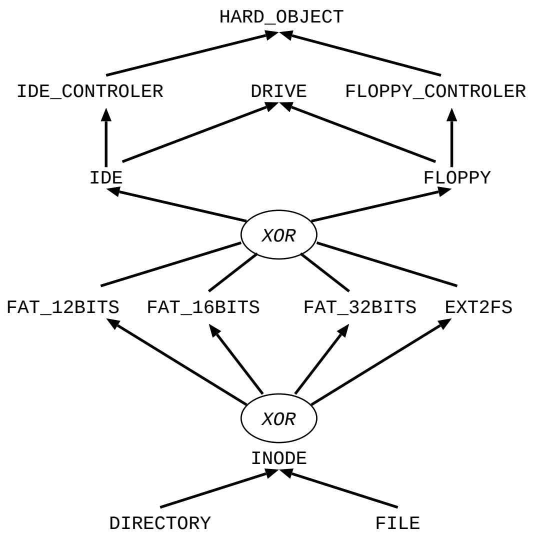


Figure A.3: Another example of dynamic inheritance (file system selection).

| *Compiler* | *user time -O0* | *user time -O3* |
|---|---|---|
| gcc 2.95.2 | 84.030 s | 33.840 s |
| SmallEiffel -.75 | 87.920 s | 36.850 s |
| Lisaac | 82.980 s | 33.620 s |

Figure A.4: Execution time of the quick-sort benchmark.

code, hence the similar performances. To achieve the translation from LISAAC to C, our compiler use traditional removal of recursivity as well as inlining, code specialization and data flow analysis. Most of our compilation strategy come from our experiment with the SmallEiffel compiler [ZCC97] and from the *Cartesian Product Algorithm* of Ole Agesen for the Self language [Age95].

Figure A.4 shows user times which we obtained on the same algorithm of quick-sort out of C, SmallEiffel and LISAAC . Those tests were carried out on INTEL Pentium II to 333 MHz with 128Mo of RAM memory under Linux 2.2.17 and GCC 2.95.2..

# Bibliography

[Age95] Ole Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In *9th European Conference on Object-Oriented Programming (ECOOP'95)*, volume 952 of *Lecture Notes in Computer Sciences*, pages 2–26. Springer-Verlag, 1995.

[CZ99] D. Colnet and O. Zendra. Optimizations of Eiffel programs: SmallEiffel, The GNU Eiffel Compiler. In *29th conference on Technology of Object-Oriented Languages and Systems (TOOLS Europe'99), Nancy, France*, pages 341–350. IEEE Computer Society PR00275, June 1999. LORIA 99-R-061.

[elf95] Tool Interface Standard (TIS) Committee. *Executable and Linking Format (ELF) Specification*, 1995. v1.2.

[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.

[Pro] Intel Processor. http://www.sandpile.org/docs/intel/80386.htm.

[sc00] EGCS steering committee. `http://gcc.gnu.org`. Site web : GNU Compiler Collection., 2000.

[Son00] B. Sonntag. `http://www.isaacOS.com`. Site web: Isaac (Object Operating System)., 2000.

[Son01] B. Sonntag. Article in French about: Usage of the processor memory segmentation with a high-level language. In *2ime Confrence Franaise sur les systmes d'Exploitation, (CFSE'2)*, pages 107–116. ACM Press, 2001.

[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.

[ZCC97] O. Zendra, D. Colnet, and S. Collin. Efficient Dynamic Dispatch without Virtual Function Tables. The SmallEiffel Compiler. In *12th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*, volume 32, number 10 of *SIGPLAN Notices*, pages 125–141. ACM Press, October 1997. LORIA 97-R-140.