UNIVERSITÉ DE STRASBOURG

Programmer's Reference Manual

Object Prototype

Lisaac V.0.4

The power of simplicity at work for You

Sonntag Benoît (`benoit.sonntag@lisaac.org`)

Document history

Version 0.1   September 12, 2003
Benoit Sonntag, Dominique Colnet,
Olivier Zendra, Jerome Boutet

Version 0.2   October 20, 2004
Benoit Sonntag, Jerome Boutet

Version 0.3   September 24, 2007
Benoit Sonntag, Alexandre Chabert

Version 0.31 September 22, 2008
Benoit Sonntag, Pierre-Alexandre Voye

Version 0.4   October 4, 2017
Benoit Sonntag

Contents

Chapter 0001b

Introduction / *Introduction*

Lisaac is the first object-oriented language based on prototype concepts really compiled, with system programming facilities. Two languages are at its origin: the Self language [US87] for its flexibility and the concept of dynamic inheritance as well as the Eiffel language [Mey94] for its static typing and security (programming by contract). The Lisaac compiler produces optimized Ansi C code later compilable on any architecture equipped with an appropriate C Compiler (GCC or others), thus making Lisaac a truly multi-platform language. Moreover, performance results of compiled objects show that it is possible to obtain executables from a high-level prototype-based language that are as fast as C programs.

> *Lisaac est le premier langage objet basé sur le concept des prototype à être réellement compilé, tout en étant muni de facilités pour la programmation système. Celui-ci trouve son origine dans deux langages : Le langage Self [US87] pour sa flexibilité et le concept d'héritage dynamique, ainsi que le langage Eiffel pour le typage statique et la programmation par contrat. Le compilateur Lisaac produit un code Ainsi C optimisé compilable sur toute architecture suportant un compilateur C (GCC ou autre), faisant de Lisaac un langage réellement multiplateforme. Les performances obtenu avec la compilation d'objet démontre qu'il est possible d'obtenir un binaire aussi rapide que du C, même avec un langage objet à prototype.*

Future work

> *Voici le futur !!!*
> *Welcome to the future*
> *in the Matrix !*

1.1 Motivation / *Avant propos*

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 fits perfectly the construction of a flexible operating system.

The purpose of our project is to break from 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, Isaac has been fully written in a high-level prototype-based language.

---

[1] Isaac:Object-oriented Operating System.

The evolution of programming currently fulfills nowadays data-processing needs and constraints in terms of software conception and production.
Nevertheless, modern languages such as object-oriented ones have never brought a real alternative to their procedural counterparts like C in the development of modern operating systems.

Historically, during the creation of an OS, programming constraints related to the hardware have been systematically fulfilled with a low-level language, such as C.
This choice leads generally 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 equipped 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.

Lisaac also comes from an experiment in the creation of an operating system based on dynamic objects, which possibilities are a subtle mix of Self and Eiffel, with the addition of some low-level capabilities of the C language.
Our language is the first compiled prototype-based language really usable. Compiled objects remain objects with all their capabilities and expressivity preserved. Hardware facilities are included natively, such as mapping or interrupt management.

> *La conception du système d'exploitation* Isaac [a] *nous a mené à la conception d'un nouveau langage nommé Lisaac. Lisaac intègre des mécanismes de protections de communications, de support des interruptions, de même que le mappage en mémoire de données pour les pilots de périphériques. L'objectif de notre projet est de se départir de la rigidité interne des systèmes d'exploitations dépendant principalement, selon nous, des langages bas niveau utilisés pour les écrire. Ainsi, Isaac a été intégralement écrit avec un langage haut niveau;*
>
> *L'évolution du marché du logiciel et de la technologie implique de nouvelles contraintes en terme de conception et de développement qui implique d'utiliser des langages de haut niveau. Malgré cet état de fait, les systèmes d'exploitations n'ont pour le moment d'autres alternative que d'être écris dans des langages procéduraux comme C.*
>
> *Ces choix et contraintes impliquent un manque patent de flexibilité qui peut seulement être atteint au niveau applicatif. Notre conception de la problématique nous a conduit à implémenter un nouveau langage objet flexible, expressif et puissant, muni de facilités permettant la programmation système. Lisaac est ainsi le fruit d'un marriage subtil entre Self et Eiffel, adjoint de facilités systèmes. Les objets compilés restent ainsi des objets conservant toutes leurs capacités et expressivité.*
>
> ---
> [a]Isaac:Object-oriented Operating System.

1.2 The Lisaac compiler / *Le compilateur Lisaac*

The Lisaac compiler produces 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 boostrap having been done in 2004, january.
The bootstrap mechanism 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 source code of the first compiler is fully translated in Lisaac. We then use our compiled Lisaac compiler version 1 to compile the new code (as we can do 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 operate equally, the only difference being the Eiffel dependance for the first one.

State 3: the source code of the compiler version 2, written entirely in Lisaac, is compiled again, this time using the version 2 of our Lisaac compiler. It produces the version 3 of the Lisaac compiler.

Every iteration of the state 3 doesn't change the produced executable, we are in a stable state. Of course, the code of the compiler has to be error free before starting the bootstraping operation. The advantage of this boostrap 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.

> *Le compilateur Lisaac produit du code C optimisé compilable sur toutes les architectures proposant un compilateur C (GCC ou autres). Le compilateur est totalement écrit en Lisaac, le boostrap ayant eu lieu en janvier 2004.*
>
> *Le bootstrap se déroule de la mnière suivante :*
>
> *Etape 1 : La première version du compilateur Lisaac est écrit dans un autre langage (ici Eiffel, langage objet classe choisi pour sa puissance, sa généricité, l'implémentation native des contrats) et compilé comme tout programme Eiffel par le compilateur SmartEiffel. Il produit un exécutable, devenant le premier compilateur Lisaac.*
>
> *Etape 2 : Le code source du premier compilateur est totalement traduit en Lisaac. Nous utilisons ensuite notre compilateur (issu de l'étape 1) lisaac pour compiler le nouveau code (comme nous pouvons le faire avec tout programme écrit en Lisaac). Cela produit un exécutable, la seconde version du compilateur Lisaac. S'il n'y a pas d'erreur, la version 1 et 2 fonctionnent de la même manière, la seul différence étant que le premier est écrit en Eiffel.*
>
> *Etape 3 : Le code source du compilateur de l'étape 2, totalement écrit en Lisaac est compilé une nouvelle fois, cette fois en utilisant la 2ème version du compilateur. Cela produit la 3ème version du compilateur.*
>
> *Chaque itération de l'étape 3 ne change pas l'exécutable produit : nous sommes dans un état stable. Bien évidemment, le code du compilateur doit être exempt d'erreur avant de d'emarrer l'opération de bootstraping. L'avantage de cette opération de bootstrap est de nous permettre d'être totalement indépendant d'un autre langage : le compilateur est maintenant écrit en Lisaac, avec une version compilé par lui même. Le compilateur est ainsi produit en utilisant la technologie Lisaac.*
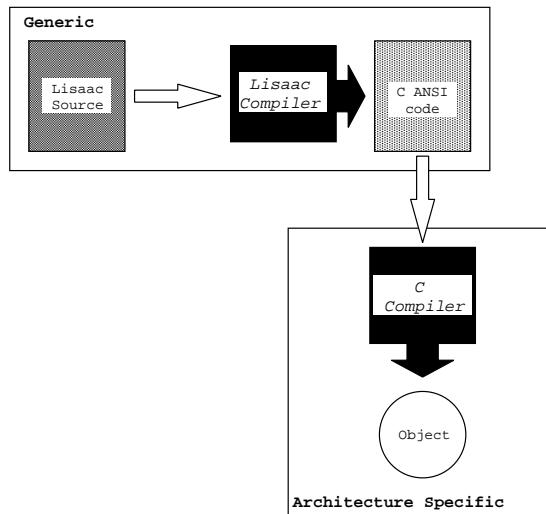
① Compiler Code **Eiffel** → **Eiffel** *Compiler* → Lisaac *Compiler* *version 1* Executable

② Compiler Code **Lisaac** → **Lisaac** *Compiler* *version 1* → **Lisaac** *Compiler* *version 2* Executable

③ Compiler Code **Lisaac** → Lisaac *Compiler* *version 2* → **Lisaac** *Compiler* *version 3* Executable

Ⓝ Compiler Code **Lisaac** → **Lisaac** *Compiler* *version 3* → **Lisaac** *Compiler* *version 3* Executable

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

Le compilateur peut être utilisé sur toute architecture possédant un compilateur C

Generic

Lisaac Source → *Lisaac Compiler* → C ANSI code

C Compiler → Object

Architecture Specific

1.3   Why Using Lisaac

Lisaac was first developped to implement the ISAAC Operating System but became an independent object oriented language, perfectly usable to write all kind of programs.

It has numerous advantages: it's a powerful high level language, based on prototype concepts. Security has been a real aspect from the start, with the static typing and assertion management (programming by contract) such as *Requires*, *Ensures* and *Invariant*, and lots of verifications during compilation.

Many high-level optimizations also provide efficiency and speed to the 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 (fixed or float); 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

Lisaac a été avant tout implémenté pour développer le système d'exploitation Isaac, mais devenant par la suite un langage généraliste.
Il a de nombreux avantages : C'est un langage haut niveau très puissant, basé sur les concepts prototypes. La sécurité logiciel fut un aspect pris en compte dès le d'epart, avec le typage static et la gestion des contrats comme les Requires, Ensures and Invariant, ainsi que les nombreuses vérifications durant la compilation.
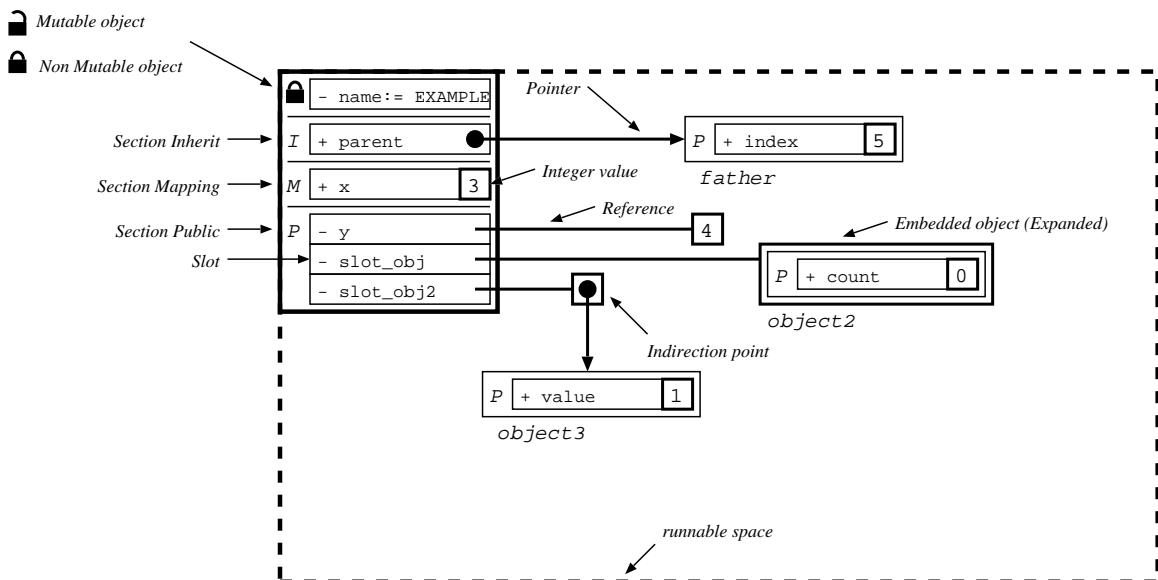
1.4 Notations

In this document, you'll find memory representation of objects. Here is the caption of the figures.
Dans ce document, vous trouverez des figures représentant la présence en mémoire des objets.
La liste des figures sont les suivantes :

Chapter 0010b

Quickstart - for beginners

After reading this chapter, you will be able to write simple programs.
The next chapter will teach you more about Lisaac, allowing you to use all of its capabilities and power.

> *Dans ce manuel, vous trouverez de nombreuses représentations d'objets en mémoire. Ici la charte graphique utilisée pour les représenter.*

2.1   Lisaac: a prototype based language
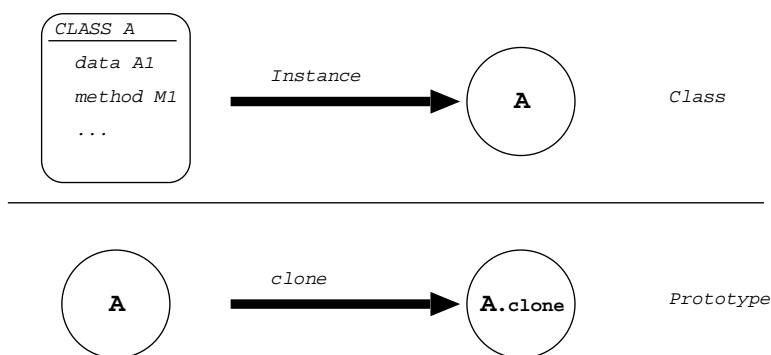
Lisaac is an object oriented language based on prototype concepts.
Class and prototype languages differ on few but important points.   In a class language, you have to instanciate an object from its description in order to make it alive.
In a prototype language, a description of an object is already alive. In Lisaac you can directly use the "master" object without having instanciated it.
This particular object name is written in capitals and can be used as any other object.
Other objects are obtained by cloning the "master" one. The **clone** routine is actually not a hard-coded function but one from the library.
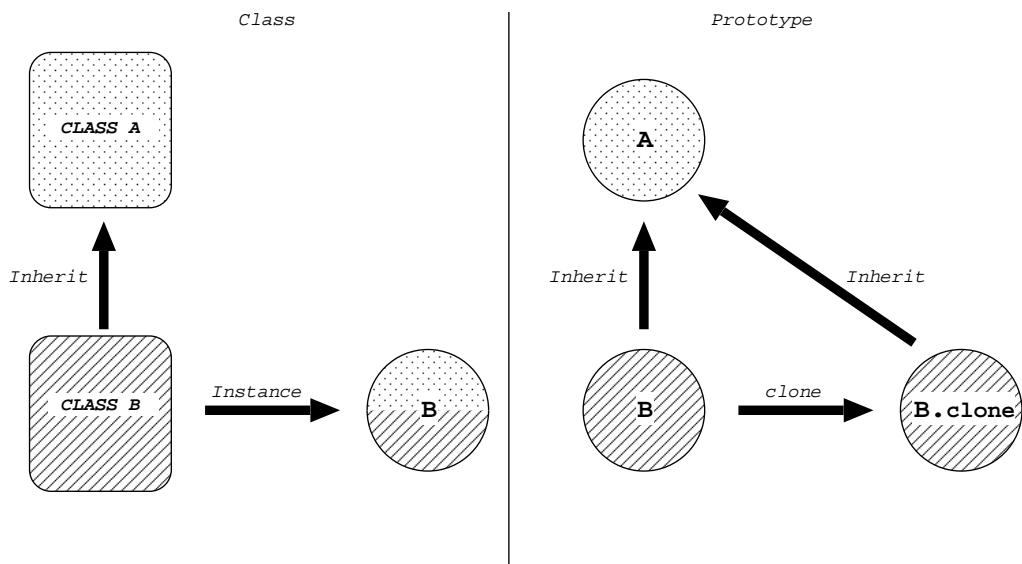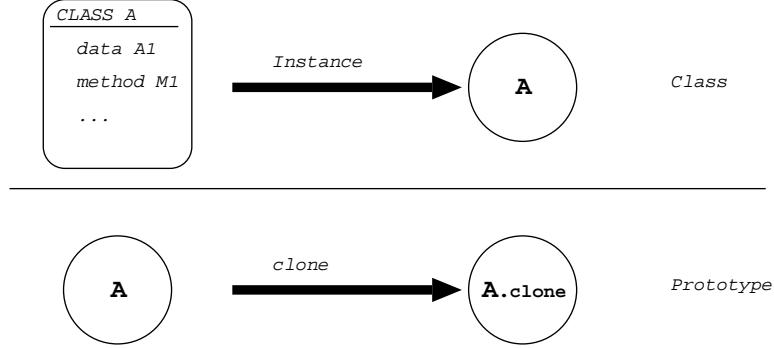
We can see from this property that inheritance is particular. Objects inherit from alive objects, with their own live. It permits numerous variations from inheritance, depending of its type (**+** or **-**), such as sharing parents between 2 cloned object, or dynamic inheritance (by changing the reference of the parent). We will see this later.
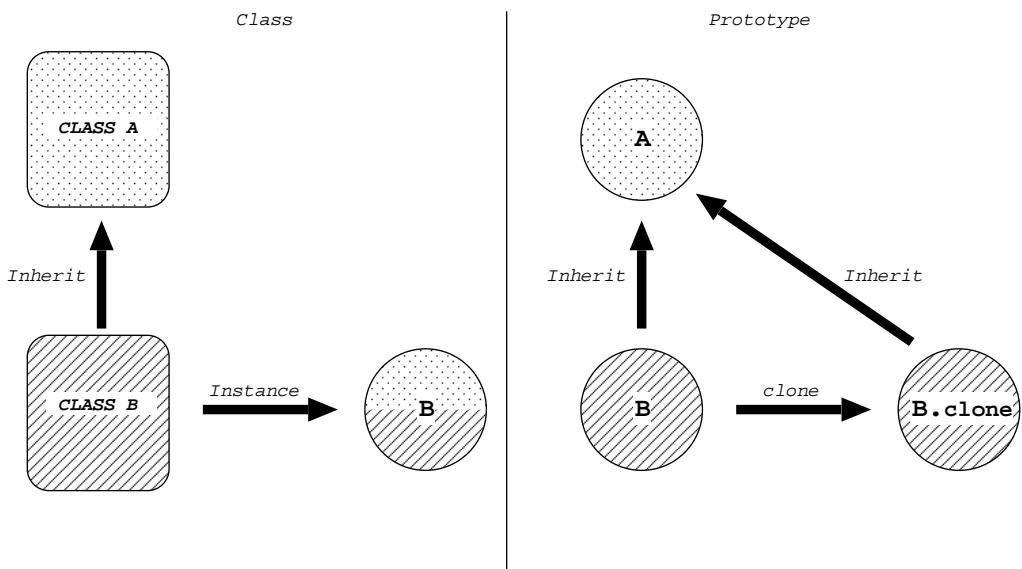
Objects are the fundamental entities in Lisaac; every entity in a program is represented by one or more objects. Even controls are handled by objects: blocks (3.11 page 93) are Lisaac closures used to implement user-defined control structures. An object is composed of a set of slots. A slot is a name-value pair. Slots may contain references to other objects. When a slot is found during a message lookup (see section 3.3.2 page 53), the object in the slot is evaluated.

Lisaac est un langage objet orienté prototype. Les classes et les prototypes diffèrent de façon importante sur quelques points. Dans un langage à classe, vous devez instancier un objet à partir de sa description pour le rendre vivant. Dans un langage objet à prototype, la description de votre objet est dors et déjà vivante. Dans le langage Lisaac, vous pouvez déjà utiliser l'objet "Maître" sans avoir à l'instancier.

Cet objet particulier est écrit en lettre capitale et peut être utiliser comme d'autres objets. Les autres objets peuvent être obtenus en clonant l'objet "Maître". La méthode **clone** n'est pas une primitive du compilateur : elle tout simplement défini dans le prototype OBJECT.

Nous pouvons voir par ces propriétés que dans ce modèle, l'héritage est particulier : Les objets héritent d'autres objets vivants, qui vivent leur propre vie. Il est ainsi permis de nombreuses variations, en fonction de son type (+ ou -), comme partager un parent entre 2 objets clonés, ou l'héritage dynamique (en changeant la référence au parent). Nous verrons tout cela plus tard.

L'objet est l'entité fondamentale en Lisaac : toute entité est un programme représenté par un ou plusieurs objets. Toutes les structures de contrôle sont prise en main par des objets : les blocks (3.11 page 93) sont en lisaac les fermetures utilisées pour implémenter les structures de contrôles. Un objet est constitué d'un ensemble de slots Un slot est une paire nom-valeur. Les slots peuvent contenir des références à d'autres objets Quand un slot est trouvé durant un lookup de message (voir la section 3.3.2 page 53), l'objet du slot est évalué.

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 in small letters but start 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 3.10 on page 87 for more
information on instruction lists.
Comments begin with `//` and stop at the end of the line.
Multi-lines comments start with `/*` and end at `*/`.

> *Lisaac est sensible à la casse et respecte les contraintes suivantes : Variables et slots sont*
> *écrit en minuscule (x,counter,. . .).*
> *Type et objets (ou le nom de l'objet/prototype "Maître") sont en lettres majuscules (*INTE-
> GER,BOOLEAN,. . . *).*
> *Mot-clés sont écrit en lettre minuscule, la première lettre étant en majuscule (***Sec-**
> **tion***,Header,. . .).*
>
> *Le symbole* `:=` *est une affectation. Faites bien attention à ne pas utiliser le symbole*
> `=` *qui compare deux objets et retourne un booléen.*
> *Vous verrez le symbole* `+` *or* `-` *avant chaque slot. Il définie le type, la portée du slot et est*
> *obligatoire. Son rôle sera expliqué dans les prochains pages.*

2.3 Objects

In Lisaac, objects are the fundamental entities. Everything is represented by one or more of
them, from a simple Integer or Boolean to more complex entities like arrays or windows.
An object is written in one and only one file, named as the name of the object and followed by
the extension **.li** .
For example, `integer.li`, `boolean.li`,`window.li`, . . .

The source code of an object is divided in sections.
Section Header is needed. In this section, you define the name of the object. Then you have
the **Section Public**, in which you define the slot which will be executed at initialization (more
in later sections).

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

Example: file `hello_world.li`

Section Header
```
  + name      := HELLO_WORLD;   // Name is in capital letters
```
Section Public

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

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

En lisaac, l'entité fondamentale est l'objet. Toutes les entités du language sont constitués de l'un d'entre deux, que ce soit un simple entier ou booléen, ou un objet plus complexe comme une collection ou une fenêtre.

Un objet est décrit dans un et un seul objet, nommé comme le nom de l'objet - suivi de l'extention **.li** .

Par exemple, `integer.li`, `boolean.li`,`window.li`, ...

Le code source de l'objet est divisé en sections. La **Section Header** *est obligatoire. C'est dans cette section que vous pouvez définir le nom de l'objet, entre autres choses. Vous trouverez ensuite la* **Section Public**, *dans laquelle vous pouvez définir le slot qui sera exécuté à l'initialisation.*

Section, **Header** *et* **Public** *sont des mots clés.*

Exemple: le fichier `hello_world.li`

Section Header
```
  + name      := HELLO_WORLD;   // Le nom est toujours en lettres majuscules
```
Section Public
```
  /* ...*/
```

⚠ *Notez qu'il n'y a pas de* **;** *après* **Section xxxx**.

2.4   Slots

An object is composed of slots, which are services given by the object.
A slot can be data as well as code (function or method).
A slot is defined by a name. It can also add a static type for data and functions.
A slot is prefixed by the **+** or **-** sign, which gives its type (to simplify, with **-** values are shared between objects while values are local to the object with **+**).
The type is defined after the sign **:**.

Section Header
```
  + name      := MY_OBJECT; // Name is in capital letters
```
Section Public
```
  + slot:INTEGER;             // Value local to the object,
                              // and init with INTEGER default value
  - slot2:INTEGER := 3;       // Value shared between objects, init with value 3
```
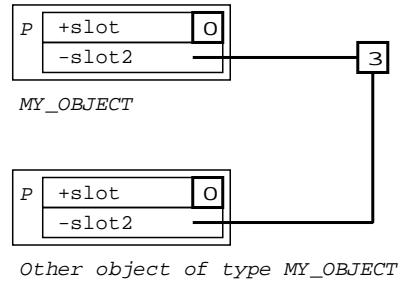
Un objet est composé de slots, qui sont autant de serices proposés par cet objet.
Un slot peut être des données autant que du code (fonction ou méthode).
*Un slot est toujours préfixé par le signe **+** or **-**, qui donne sa portée. Pour simplifier, avec* **-**, *les objets sont partagés entre objets, tandis qu'avec **+**, les données sont locales à l'objet.*
*Le type est dfini aprs le signe **:**.*

```
┌──┬────────────┬──┐
│P │ +slot      │O │
│  ├────────────┤──┤        ┌───┐
│  │ -slot2    ─┼──┼────────┤ 3 │
└──┴────────────┴──┘        └─┬─┘
                             │
MY_OBJECT                    │
                             │
┌──┬────────────┬──┐         │
│P │ +slot      │O │         │
│  ├────────────┤──┤         │
│  │ -slot2    ─┼──┼─────────┘
└──┴────────────┴──┘

Other object of type MY_OBJECT
```

2.4.1 Methods and functions

Simple slots

Comme précédemment défini, un slot peut représenter une fonction ou une méthode. Les méthodes (ou routines) sont une notion fondamentales des langages orienté objets, avec le concept connexe qu'est la liaison dynamique (ou envoi de message, appel de routine, appel de méthode, résolution dynamique . . .).
Il y a deux types de slots. Le premier est exécuté à l'initialisation de l'objet et défini en tant que valeur par défaut d'une variable.

```
// L'operation '3 + 4' est evaluee a l'initialisation.
+ slot:INTEGER := 3 + 4;
+ slot2:INTEGER := slot * 2;
```

Le second type est exécuté uniquement lors de l''appel du slot, et est défini par le symbole <-symbol.

```
+ slot :INTEGER := 3;
// L'operation '5 + slot' est evaluee lors de l'appel de slot2
+ slot2 :INTEGER <-
( 5 + slot );
```

Un code plus complexe peut bien evidemment tre défini entre parenthses

```
+ slot :INTEGER := 3;
+ slot2 :INTEGER := 4;
+ slot3 <- // Slot sans valeur de retour
(
  slot := slot + 3;
  slot2 := slot + 5;
  slot2 := slot2 * 3;
);
```

Notez que vous pouvez écrire votre code sur une seule ligne, bien que ce soit moins lisible. La valeur de retour est la dernire valeur, sans ;.

```
+ slot :INTEGER := 3;
+ slot2 :INTEGER <-
(
  slot // Valeur de retour, evaluee a l'appelle de slot3
);
+ slot3:INTEGER :=
(
  slot := slot + 6;
  slot := slot2 * 5;
  slot // Valeur de retour evaluee lors du chargement de l'objet.
);
```

⚠️*Faites bien attention à la cohérence du type de l'objet de retour défini en entte de fonction et le type de l'objet renvoyé.*

As said before, a slot can also represent a function or method.
Methods(or routines) are a fundamental notion in object-oriented languages, together with their companion concept late binding (or message sending, routine call, method call, dynamic dispatch, . . .).

There are two types of code slots. The first one is executed at the load of the object and is defined as the 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 and is 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 program 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 slot3
);
+ slot3:INTEGER :=
(
  slot := slot + 6;
  slot := slot2 * 5;
  slot // Return Value, evaluated during load of the object
);
```

⚠ Be carefull, the return type of the slot must be the same as 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.
L'appel d'un slot dépend du moment où il arrive.

Fichier `object1.li`

Section Header
  + name := OBJECT1;
Section Public
  + slot :INTEGER := 3;
  + slot2 :INTEGER <-
  (
     slot * 2 + 4 // Appel de 'slot' a partir du meme objet
  );

File `object1.li`

Section Header
  + name := OBJECT1;
Section Public
  + slot :INTEGER := 3;
  + slot2 :INTEGER <-
  (
     slot * 2 + 4 // Call of 'slot' from the same object
  );

An object is initialized with the NULL value, and a call on it will create an error. In a prototype language (see 2.1 page 11), the "master" object (written in capital letters) is alive without needing to instanciate it. Other objects are created by cloning it using the **clone** slot. For more information see 2.7 page 26.

File main_object.li

Section Header
  + name := MAIN_OBJECT;

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 slot call 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 of different types
(
  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 <-
(
  // call of slots
  slot 3 + (slot2 (2,3) write 'c') + (slot3 4 multiply 5 add 6)
);
```

Assignment of slots

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

Section Header
```
  + name      := OBJECT1;
```

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

Section Header
```
  + name      := MAIN_OBJECT;
```

Section Public
```
  + slot_object:OBJECT1;
  - method <-
  (
    slot_object := OBJECT1.clone;
    slot_object.value := 4;              // The compiler will stop
  );
```

This is done to protect slots from objects. When you define an object, you must specify the slots which 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 which can be modified by anybody working with your object, ...You can also define conditions inside the method to further protect your object.

Example: use of a 'setter'

Section Header
```
  + name      := OBJECT1;
```

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;
```

Section Public
```
  + slot_object:OBJECT1;
  - method <-
  (
```

```
    slot_object := OBJECT1.clone;
    slot_object.set_value 4;
);
```

2.4.2   Local variables

You can define local variables inside your slot. The syntax is the same as a slot. A local variable often won't be 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 variables 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 an error
  var3 := a - 5;
  result := var1 + var2 - var3;
  result
);
```

2.5   Compilation and running

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

```
lisaac my_object.li
```

It produces two files: `my_object.c` and `my_object`, which is an executable. By default Lisaac uses GCC to compile the produced C code.

Running an object

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

Section Header
```
 + name      := OBJECT_TO_RUN;
```

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;
```

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

For more information on slots, methods and method calls, see section 3.7 page 80 and section 3.8 page 84.

2.6 How to write

2.6.1 Types

There are 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 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
  10_000, 0FC4_0ABCh: reading facility

- BOOLEAN: you have 2 'values' for BOOLEAN: TRUE or FALSE. Each of those 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 built with functions of the library

- FIXED_ARRAY: an array with fixed lower bound and lots of possible 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;

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 single 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've seen 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;

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 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.[1] As we've seen 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;
```

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 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 3.10.3 page 92 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 3.8 about message sending, and section 3.11.3 page 96 about booleans and conditionals.

Note that a list of instructions and an expression are the same syntactical construct, between parentheses. See section 3.10.1 page 88 about return values in lists of instructions.

The { /* ... */ } defines a list of instruction like a classic list, but its type is BLOCK and its evaluation is delayed (see section 3.11 page 93).

---

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

2.6.6   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 "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 loop, as well as others, are explained in section 3.11.3, page 97.

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 links 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;
```

Section Public
```
   + slot <- /* ... */
```
Section Header
```
  + name      := MAIN_OBJECT;
```

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

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;

Section Public
  + slot_data:INTEGER := 3;

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

⚠ Note that the *self* is different between each 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 seen before, you have to use clone in order to work with an object.

Section Header
  + name   := OBJECT1;

Section Public
   + slot <- /* ... */

Section Header
  + name   := MAIN_OBJECT;

Section Public
  - main <-
(**+ my_object**:OBJECT1;
  my_object := OBJECT1.**clone**;
  my_object.slot;   // No problem there, my_object is not Null
);
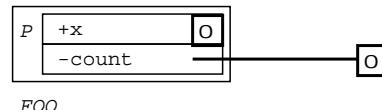
Example: memory representation (we don't represent slots 'set_x' and 'set_count' to simplify the example, see later the real representation)

Section Header
  + name   := FOO;
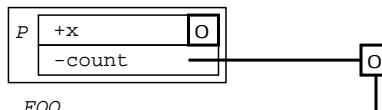
Section Public
```
  + x       :INTEGER;
  - set_x v:INTEGER <- ( x := v; );

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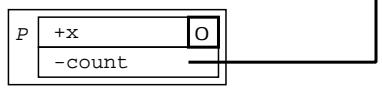

FOO

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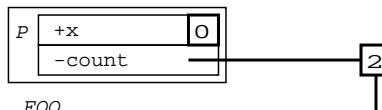

FOO
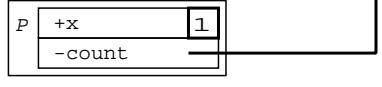
new_foo := FOO.clone

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


FOO

new_foo

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 align="center">Object FATHER</p>

Section Header
```
  + name      := FATHER;
```

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

<p align="center">Object SON</p>

Section Header
```
 + name      := SON;
```
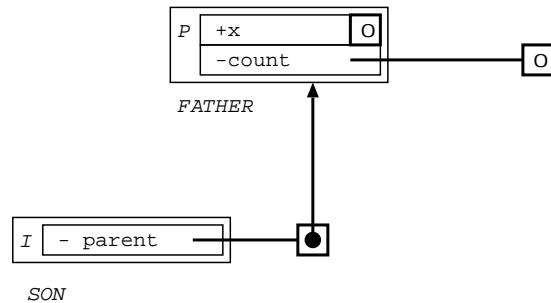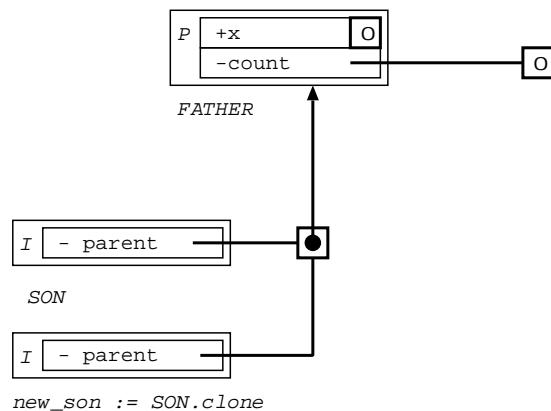
Section Inherit
```
 - parent:FATHER := FATHER;        // name of the slot doesn't matter
```
Section Public
```
 - change_parent p:FATHER <- ( parent := p; );
```
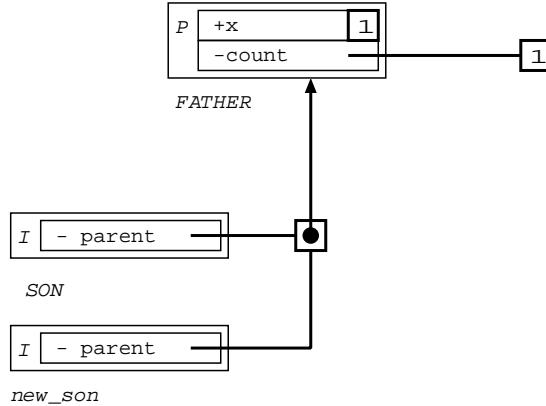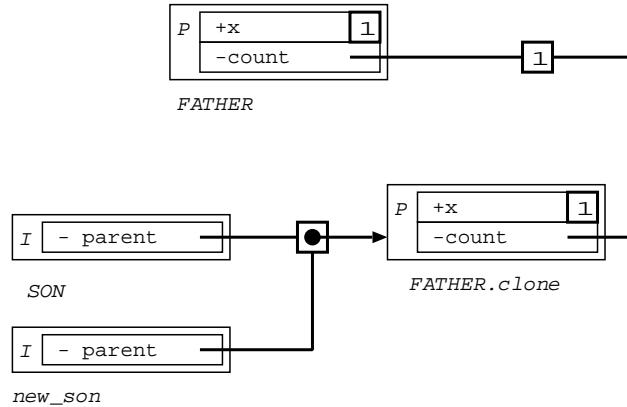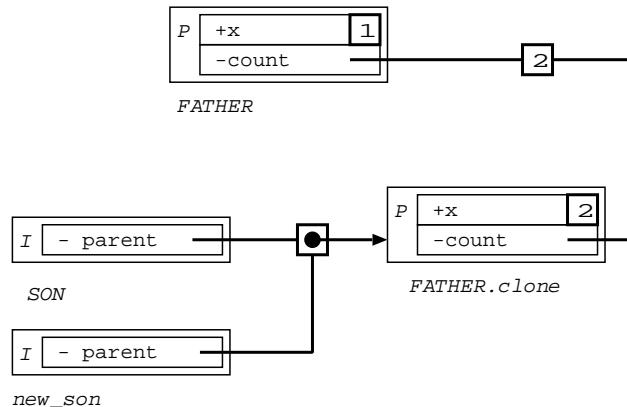


```
new_son := SON.clone;
```


new_son := SON.clone

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

```
P  +x            1
   -count              ──────────  1
```
FATHER

```
I  - parent        ●
```
SON

```
I  - parent
```
new_son

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

```
P  +x            1
   -count              ──────  1
```
FATHER

```
I  - parent   ●──►   P  +x            1
                        -count
```
SON   *FATHER.clone*

```
I  - parent
```
new_son

new_son.**inc_x**;
new_son.**inc_count**;

```
P  +x            1
   -count              ──────  2
```
FATHER

```
I  - parent   ●──►   P  +x            2
                        -count
```
SON   *FATHER.clone*

```
I  - parent
```
new_son

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

<div align="center">Object FATHER</div>

Section Header
  + **name**   := FATHER;

Section Public

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

Object SON

Section Header
```
+ name      := SON;
```

Section Inherit
```
+ parent:FATHER := FATHER;
```
Section Public
```
- change_parent p:FATHER <- ( parent := p; );
```


FATHER

SON

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


FATHER

SON

new_son := SON.clone

```
new_son.inc_x;
new_son.inc_count;
```
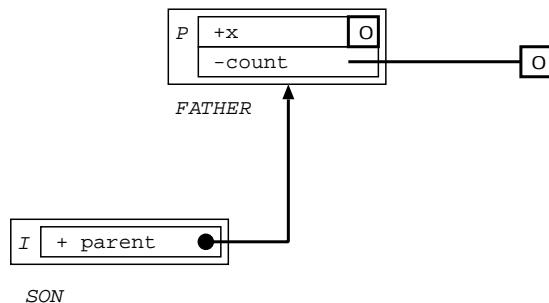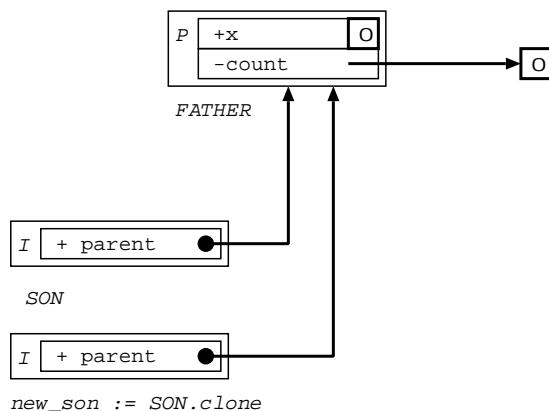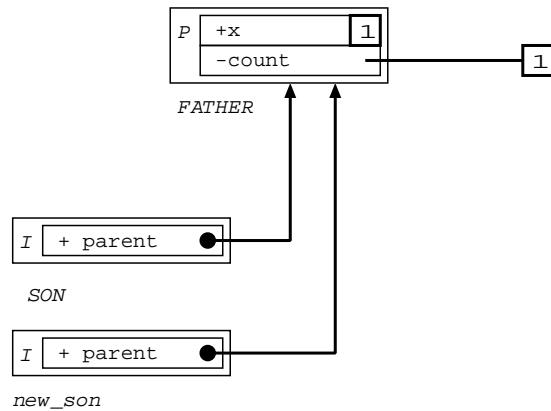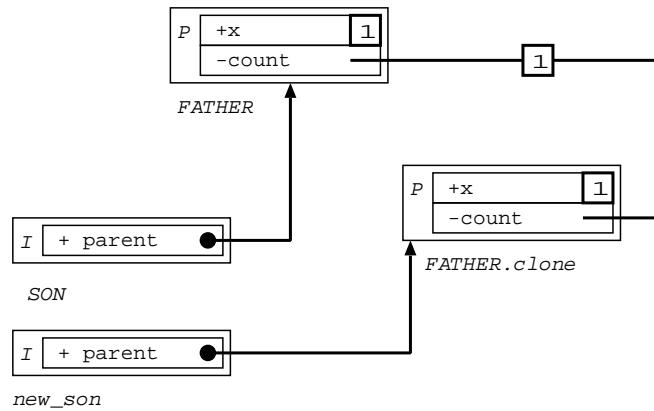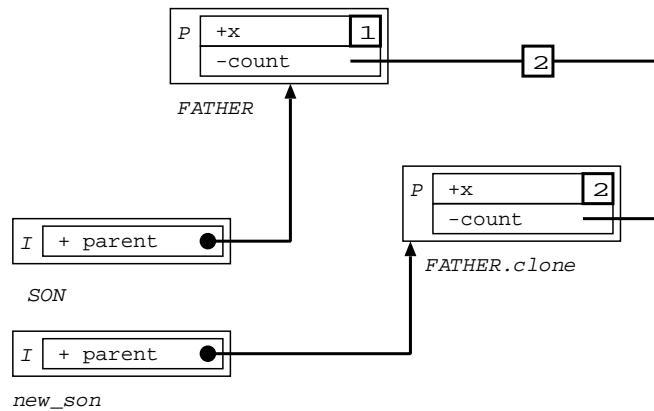
new_son.**change_parent** (FATHER.**clone**);

new_son.**inc_x**;
new_son.**inc_count**;

Chapter 0011b

Language Reference

3.1 Lip: Lisaac project file (Lisaac Makefile)

Before entering deeper with the Lisaac language, the first step is to explain how to compile a Lisaac program using Lip files. They are used to describe compiler options, paths, include external libs even do inheritance between projects.

3.1.1 Lip Grammar

The Lip grammar is a subset of the Lisaac grammar.

```
PROGRAM        -> { 'Section' ('Inherit' | 'Public' | 'Private') { SLOT ';' } }
SLOT           -> '+' identifier ':' TYPE [ ':=' EXPR_CONSTANT ]
               |  '-' identifier [ identifier ':' TYPE ] '<-' EXPR
TYPE           -> 'BOOLEAN' | 'STRING' | 'INTEGER' | 'LIP'
EXPR_AFFECT    -> [ identifier !!AMBIGU!! ':=' ] EXPR
EXPR           -> EXPR_CMP { ('|' | '&') EXPR_CMP }
EXPR_CMP       -> EXPR_BINARY { ('='|'!='|'>'|'<'|'>='|'<=') EXPR_BINARY }
EXPR_BINARY    -> EXPR_UNARY { ('-'|'+') EXPR_UNARY }
EXPR_UNARY     -> ( '-' | '!' ) EXPR_UNARY
               |  EXPR_BASE
EXPR_LIST      -> { EXPR_AFFECT ';' } [ EXPR_AFFECT ]
EXPR_BASE      -> EXPR_RECEIVER { '.' EXPR_MESSAGE }
EXPR_RECEIVER  -> EXPR_PRIMARY
               |  EXPR_MESSAGE
EXPR_MESSAGE   -> identifier [ EXPR_ARGUMENT ]
               |  'if' '{' EXPR_LIST '}' [ 'else' '{' EXPR_LIST '}' ]
EXPR_ARGUMENT  -> identifier
               |  EXPR_PRIMARY
EXPR_PRIMARY   -> EXPR_CONSTANT
               |  '(' EXPR_LIST ')'
EXPR_CONSTANT  -> integer
               |  string
               |  TRUE
               |  FALSE
```

3.1.2 Example

```
Section Inherit

  + parent:STRING;

Section Private

  + is_valid:BOOLEAN;
```

```
- src_path <-
(
  path (lisaac + "src/*");
);

- front_end <-
(
  general_front_end;
  ((input_file = "") | (input_file = "lisaac")).if {
    compiler_path;
    (is_valid).if {
      boost;
    };
  };
);

- back_end <-
(
  general_back_end;
  (is_valid).if {
    execute "cp lisaac.c ../bin/.";
    execute "cp lisaac ../bin/.";
  };
);

Section Public

- compiler <-
// Compile the Lisaac compiler.
(
  compiler_path;
);
```

3.1.3   Lip usage and features

The compiler always needs a Lip file to work. By default the compiler will look in the default
Lip file found in the Lisaac environnement variable path **LISAAC_DIRECTORY** or with the
old way in the path.h. This will appear if tou don't have a Lip file for your project, otherwise
the compiler will at first read the Lip file in the directory you are calling the compiler. The
compiler look in the current directory and in all parents to find a *make.lip* file. If no inheritance
is set, the *make.lip* file from the compiler is inherited. You can also add inheritance by hand
with this two ways:

```
+ parent : STRING;
+ parent := "../../../path../file";
```

The **+** is used for variables and **-** for methods.

In the *Private Section*, variables and functions will not appear to the compiler. You can then
use this Section to describe things you want to be done before the compiler is working and to
set options you want to call in the *Public Section* with the compiler. All compiler options are
thus described in the *Public Section*.

When you add a public function, It is possible to add a commentary down to the function
definition, then when calling the compiler this description and arguments will appear as show
this example:

```
Section Public

- debug_level:INTEGER <-
// Fix debug level (default: 15)
(
```

```
    // some code
  );
```

```
$lisaac

Usage:
  lisaac [<lip_file.lip>] [<input_file[.li]>] {<Options>}

Options:
  ...
  -debug <level:INTEGER> :
    Fix debug level (default: 15)
  ...
```

3.1.4 Advanced Lip usage

Some builtin methiods are part of the compiler:

- **exit**: exit from function as in C

- **path(STRING)**: add prototype path location

- **run(STRING)**: execute the string as a unix command

- **get_integer**: get command line integer

- **get_string**: get command line string

For describing a path you can use the ***** symbol as for example: *lisaac/personnal/*.*

Compiling steps:

1. The compiler look for a *make.lip* file

2. the front_end method is runned

3. compilation and C generation

4. the back_end method is runned

3.2 Lexical and syntax overview

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

The following syntax of Lisaac is described using "Extended Backus-Naur Form" (EBNF). Terminal symbols are enclosed between 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
→	production	separates the left and right hand sides of a production

3.2.1   Lexical overview

The following rules draw 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
real	constant of type REAL
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
style	clone comportement
type	unary typing operation
result	result identifier value

section	→	`"Header"` | `"Inherit"` | `"Private"` | `"Public"`
	|	`"Mapping"` | `"Interrupt"` | `"Directory"`
	|	`"External"` | `"Insert"`
identifier	→	`LOWER_CASE` { `LOWER_CASE` | `DECIMAL_DIGIT` | `'_'` }
operator	→	`OP_CHAR` { `OP_CHAR` } *except affect symbol and* `'='` *or* `'!='`
integer	→	`OCTAL_DIGIT` { `OCTAL_DIGIT` | `'_'` } `'o'`
	|	`DECIMAL_DIGIT` { `DECIMAL_DIGIT` | `'_'` } [`'d'`]
	|	`HEXA_DIGIT` { `HEXA_DIGIT` | `'_'` } `'h'`
real	→	`DECIMAL_DIGIT` { `DECIMAL_DIGIT` | `'_'` } `'.'` [{ `DECIMAL_DIGIT` }]
		[`'E'` [`'+'` | `'-'`] `DECIMAL_DIGIT` { `DECIMAL_DIGIT` }]
cap_identifier	→	`UPPER_CASE` { `UPPER_CASE` | `DECIMAL_DIGIT` | `'_'` }
characters	→	`'''` (`NORMAL_CHAR` | `ESCAPE_CHAR`) `'''`
string	→	`'"'` { `NORMAL_CHAR` | `ESCAPE_CHAR` } `'"'`
external	→	`` '`' `` { `NORMAL_CHAR` | `ESCAPE_CHAR` } `` '`' ``
affect	→	`":="` | `"<-"` | `"?="`
style	→	`'+'` | `'-'`
type	→	`"Expanded"` | `"Separate"` | `"Strict"`
result	→	`"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`	→	*any character except* `'\'` *and* `'''`
`ESCAPE_CHAR`	→	`'\t'` | `'\b'` | `'\n'` | `'\f'` | `'\r'` | `'\v'` | `'\a'`
	|	`'\0'` | `'\\'` | `'\''` | `'\"'` | `'\?'`
	|	`NUMERIC_ESCAPE`
`NUMERIC_ESCAPE`	→	`'\'` *integer* `'\'`

Numbers

Notation of integers: *12, 12d*: decimal value

1BAh, 0FFh: hexadecimal value

01010b, 10b: binary value

14o, 6o: octal value

10_000, 0FC4_0ABCh: reading facility

Notation of reals: *12.*: simple decimal value

12.5: simple real value

1.5E6: value with exposant

10_000.33: reading facility

Characters

Notations: *'a', 'Z', '4'*: simple character

'\n', '\t', '\r': escape character

'\10\', '\0Ah\': code character

The complete list of escape sequences 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 by enclosing it between backslashes. You can specify the type of the number (d or nothing for decimal, h for hexadecimal, o for byte or octal, b for binary)

For example: '\123\', '\123d\', '\4Ah\','\101o\', '\10010110b\'.

String

A STRING_CONSTANT is composed of 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 followed 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"

3.2.2   Syntax overview

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

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

3.3   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. 3.3.1)

- the mode of application of the *lookup* mechanism: inheritance slot (see 3.3.2) or normal
  message slot

- the exception mode (see 3.3.5)

- the data structure mapping mode (see 3.3.4)

- the link with C code mode (see 3.3.6)

- the classical code section (see 3.3.7)

3.3.1 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 complement prototype. The `category` slot indicates the category of the prototype in regard to its level of protection against the other prototypes. There are 3 levels of protection and a special level: KERNEL, DRIVER, APPLICATION and DOCILE.

- A KERNEL object can only use objects of KERNEL level.

- A DRIVER object can use objects of KERNEL or DRIVER 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 regarding 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 execution: imagine for example the consequences of modifying the **category** slot !

Slot name	Type	Description
'name'	PROTOTYPE	prototype's name (*mandatory*)
'category'	KERNEL, DRIVER	protection level
	APPLICATION,DOCILE	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'	*external*	C equivalent type (if any)
'default'	EXPRESSION	Default value of the prototype (see 3.6.1)
'external'	*external*	C code which will be included in the C compiled file
'lip'	*piece of code*	Include Lip code

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>';
- lip := ( add_lib "-lX11" );
```

Objects and clone

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

- Slots defined with the the + symbol,

  ```
  +  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,

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

  are reserved for parallel execution of prototypes. They give an "agent" prototype (**??**). This kind of prototype can be the entry point of an application or a prototype running concurrently (see 3.14).

- 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 cases expanded objects are simple objects, such as INTEGER, CHARACTER, BOOLEAN,... Usually you have to define the slot **default** and a **type** associated.

3.3.2   The `Inherit` section

This section describes the inheritance slots of the object. Like in Self, a prototype can have several parents 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 3.3.2 page 53).

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 3.3.2). If a slot called on an object is not found in this object, the *lookup* algorithm searches in the parents to find the correct slot and returns the first found.

Section Header
```
  + name   := FATHER1;
```
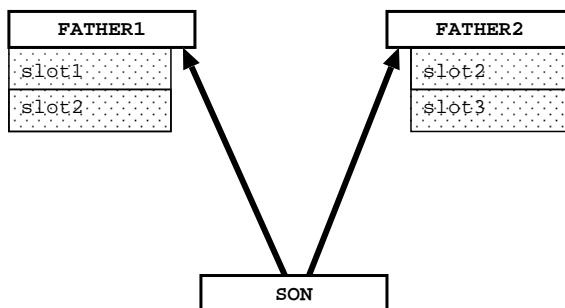
**Section Public**
```
  + slot1 <- /* ... */
  + slot2 <- /* ... */
```

**Section Header**
```
  + name   := FATHER2;
```

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



**Section Header**
```
  + name   := SON;
```

**Section Inherit**
```
  - parent1:FATHER1 := FATHER1;
  - parent2:FATHER2 := FATHER2;
```

**Section Header**
```
  + name   := TEST;
```

**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 3.4.2 page 65) but you can change the kind of slot (+, - and **Expanded**).

Section Header
```
+ name      := FATHER1;
```

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

Section Header
```
+ name      := FATHER2;
```

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

Section Header
```
+ name      := SON;
```

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;
```

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 slot doesn't matter. We often named it "parent" but it's for more visibility than anything, it's not a reserved keyword. But it's mandatory to be precise about 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;
```

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

Object SON

Section Header
```
  + name       := SON;
```

**Section Inherit**
```
  - parent:FATHER := FATHER;
```
Section Public
```
  - change_parent p:FATHER <- (parent := p;);
```


```
new_son := SON.clone;
```
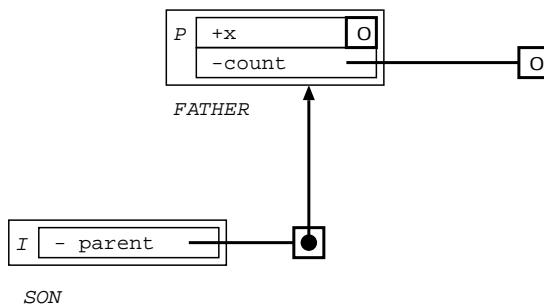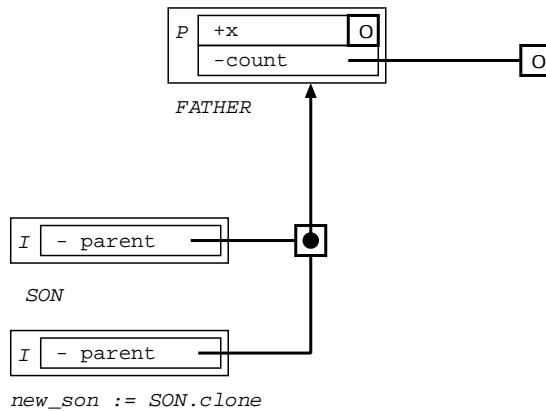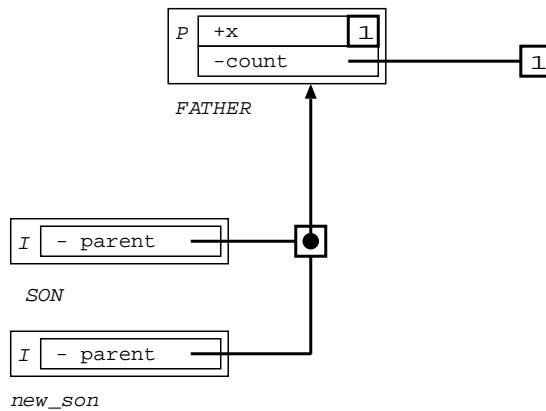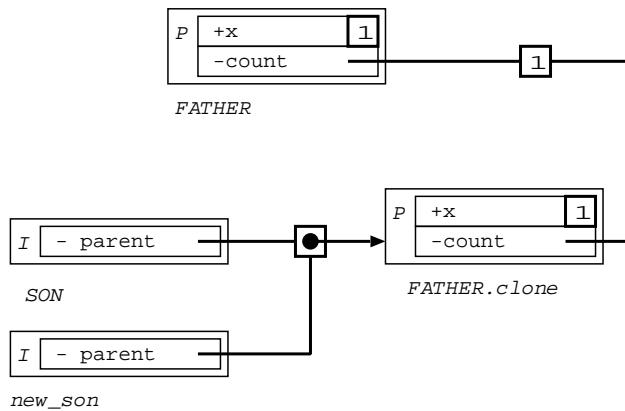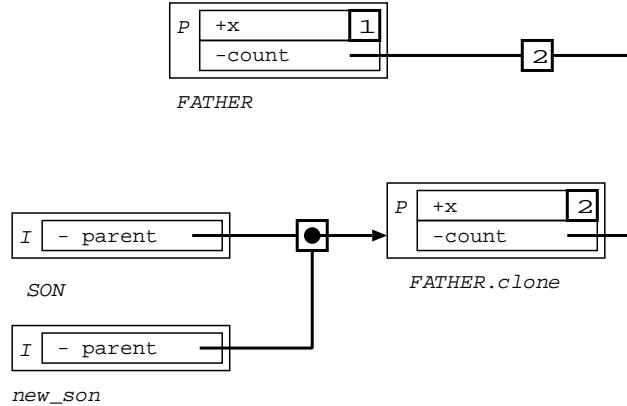
new_son.**inc_x**;
new_son.**inc_count**;

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 got their parents changed.

<div align="center">Object FATHER</div>

Section Header
   + **name**   := FATHER;

Section Public
   + **x**   :INTEGER;
   - **inc_x** <- (x := x + 1;);
   - **count** :INTEGER;
   - **inc_count** <- (count := count + 1;);

<div align="center">Object SON</div>

Section Header
   + **name**   := SON;

Section Inherit
   + **parent** :FATHER := FATHER;
Section Public
   - **change_parent** p:FATHER <- (parent := p;);

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


new_son := SON.clone

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



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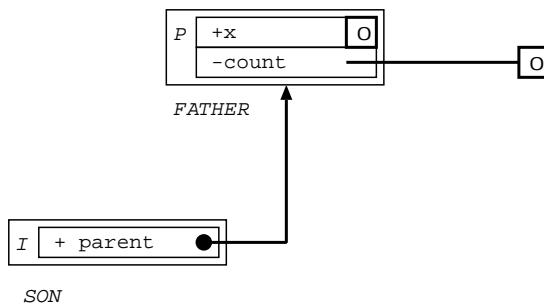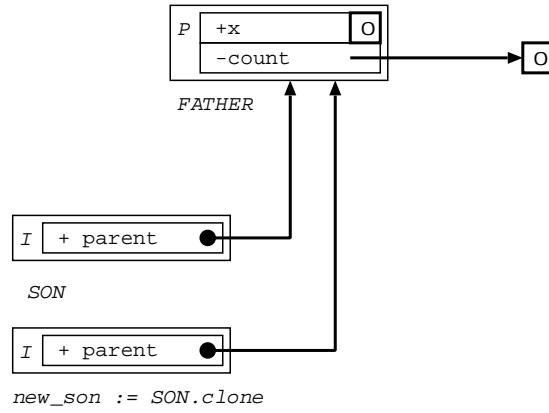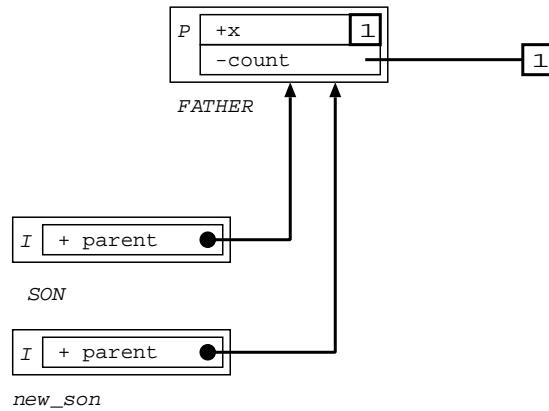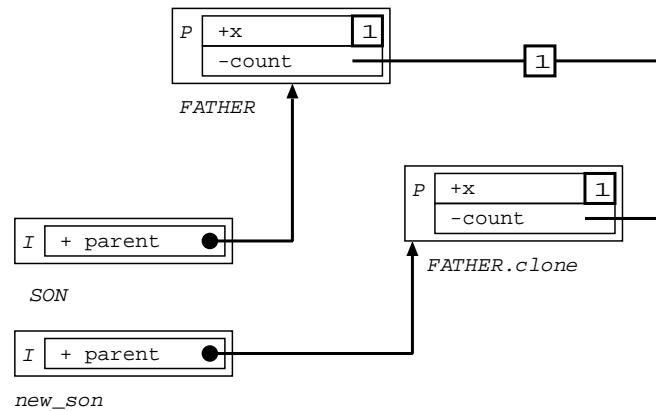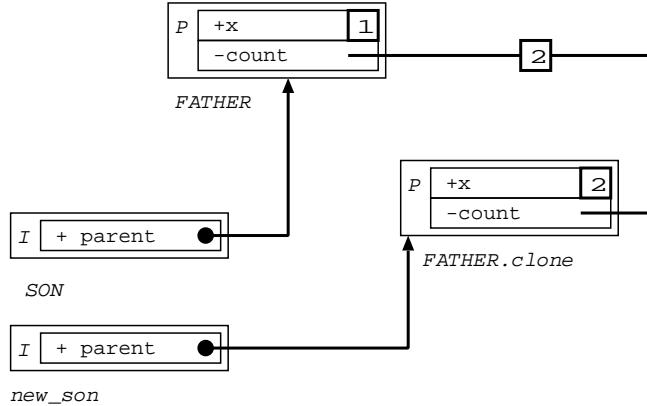


```
new_son.inc_x;
new_son.inc_count;
```

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.

<div align="center">Object FATHER</div>

Section Header
  + **name**   := FATHER;

Section Public
  + **x**   :INTEGER;
  - **inc_x** <- (x := x + 1;);
  - **count**:INTEGER;
  - **inc_count** <- (count := count + 1;);

<div align="center">Object SON</div>

Section Header
  + **name**   := SON;

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);
```
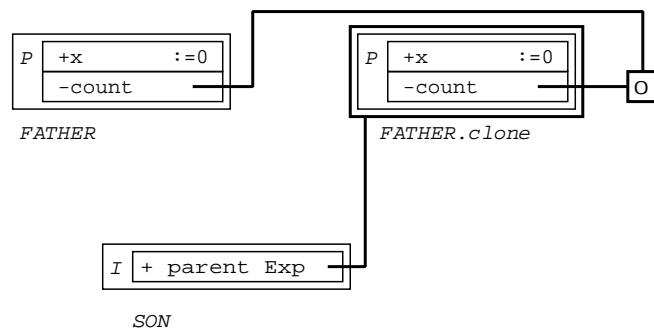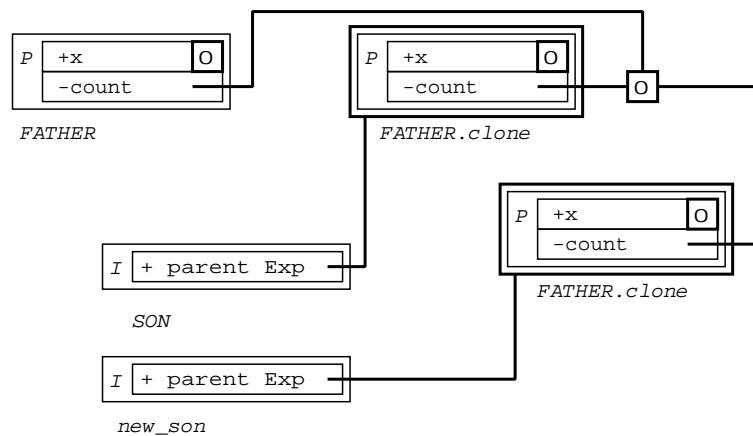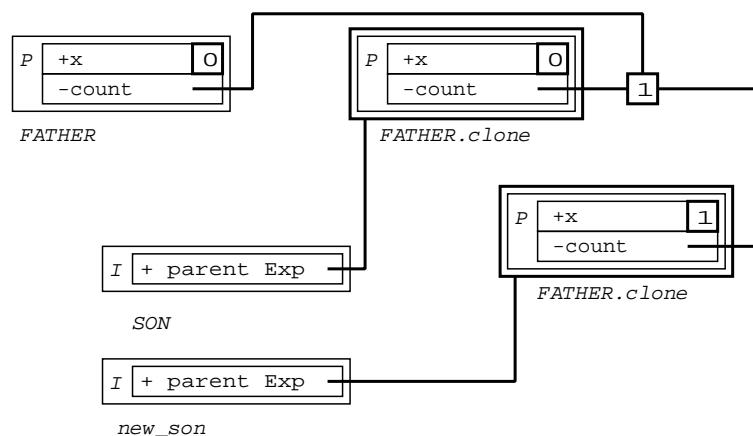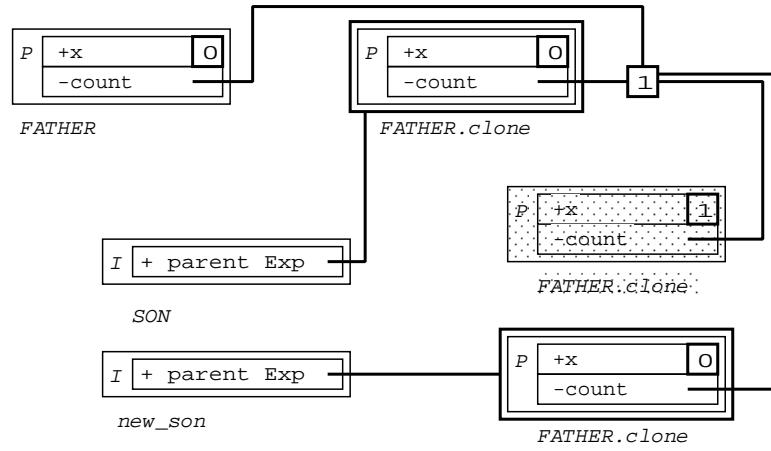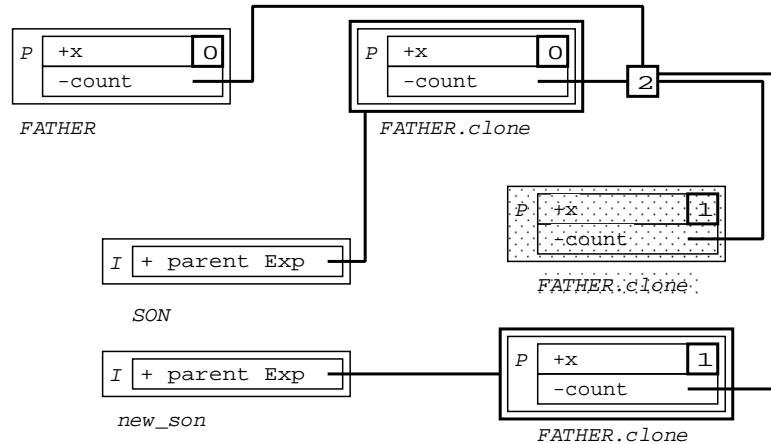
new_son.**inc_x**;
new_son.**inc_count**;

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.

<div align="center">Object FATHER</div>

Section Header
  + **name**   := FATHER;

Section Public
  + **x**   :INTEGER;
  - **inc_x** <- (x := x + 1;);
  - **count**:INTEGER;
  - **inc_count** <- (count := count + 1;);

<div align="center">Object SON</div>

Section Header
  + name   := SON;
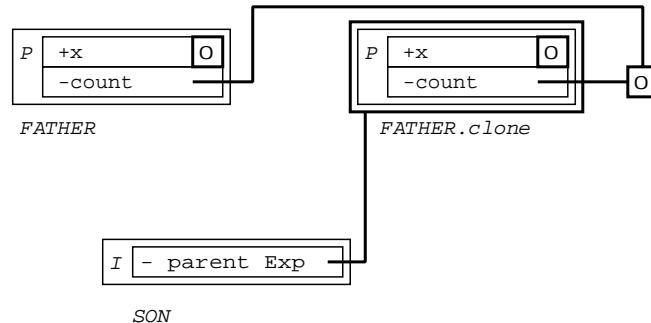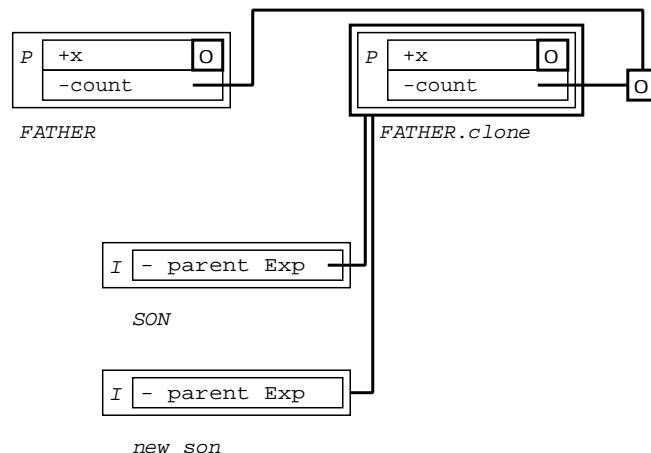
Section Inherit
  - parent:Expanded FATHER;
Section Public
  - change_parent p:FATHER <- (parent := p;);
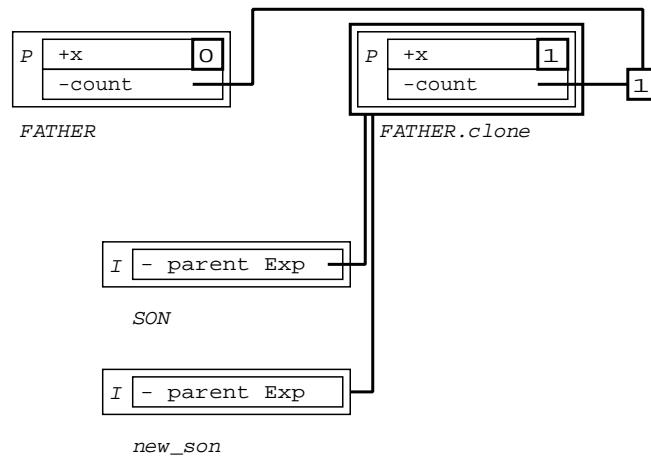
new_son := SON.**clone**;
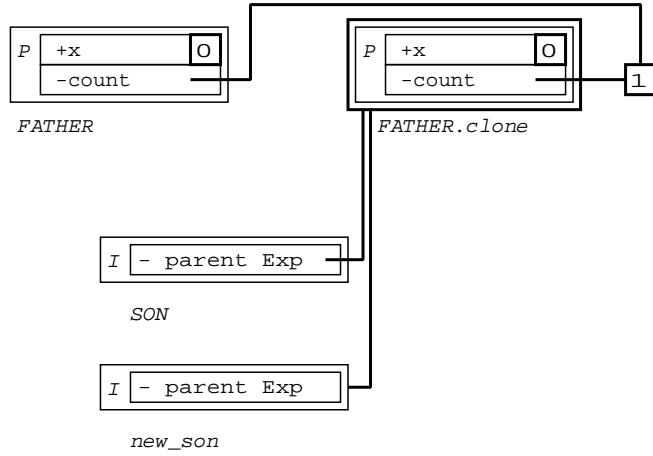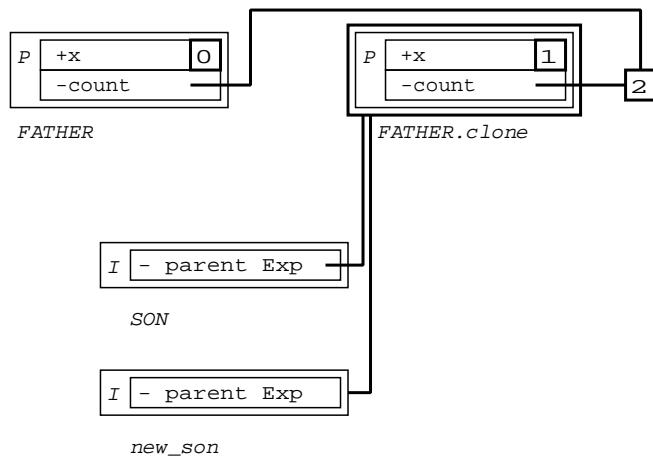
new_son.**inc_x**;
new_son.**inc_count**;

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;
```

Section Public
```
  + bar <- /* ... */
  + foo <- /* ... */
```

Section Header
```
  + name := B;
```

Section Inherit
```
  + parent:A := A;
```
Section Public
```
  + bar <- /* ... */
  + toto <- /* ... */
```

Section Header
```
  + name := C;
```

Section Inherit
```
  + parent:A := A;
```
Section Public
```
  + titi <- /* ... */
  + foo <- /* ... */
```

Section Header
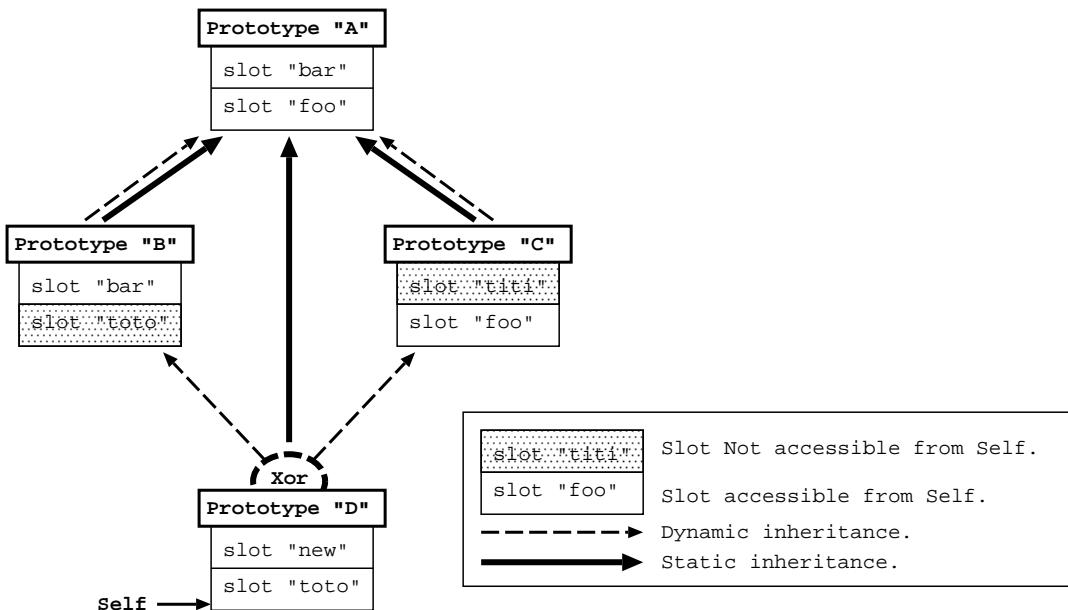
```
  + name := D;
```

Section Inherit
```
  + parent:A <-
  (
  ...              // code that can be dynamically B or C
  );
```
Section Public
```
  + new <- /* ... */
  + toto <- /* ... */
```


Section Header
```
  + name := TEST;
```

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 3.7 page 80). 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 arrived, 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 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. You can bypass the lookup algorithm by precising the parent on which you call the slot.

Section Header
  + **name** := FATHER1;

Section Public
  - **method** <- /* ... */

Section Header
  + **name** := FATHER2;

Section Public
  - **method** <- /* ... */
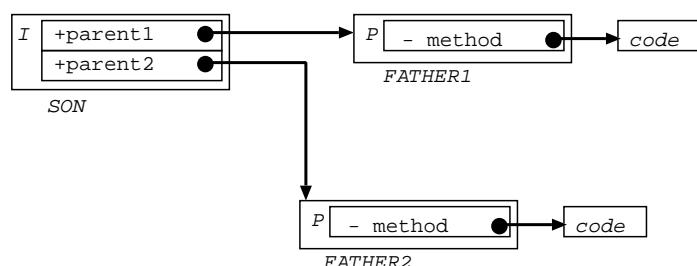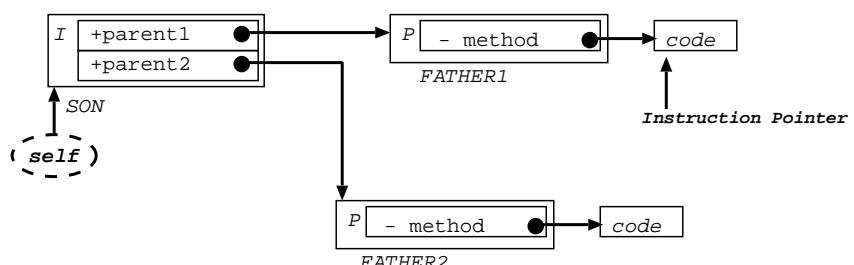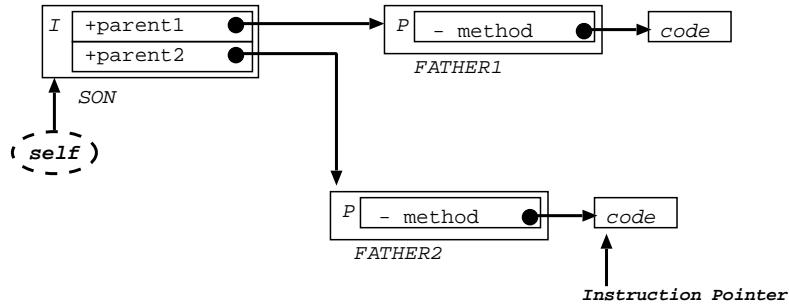
Section Header
  + **name** := SON
Section Inherit
  + **parent1** := FATHER1;
  + **parent2** := FATHER2;

method;

parent2.method;

3.3.3   The `Insert` section

Slots defined in an Insert section are almost equivalent to Inheritence slots. The difference resides in the fact that the return type is not a parent of your prototype. In other words, your prototype isn't a sub-type of a slot's return type residing in section Insert.

Note 1: Expanded prototypes can only use section Insert, not the Inherit section.

Note 2: The slots order is important regarding the search (lookup algorythm) of a message. You can put as many Inherit or Insert sections at the beginning of your prototype.

3.3.4   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 writting).
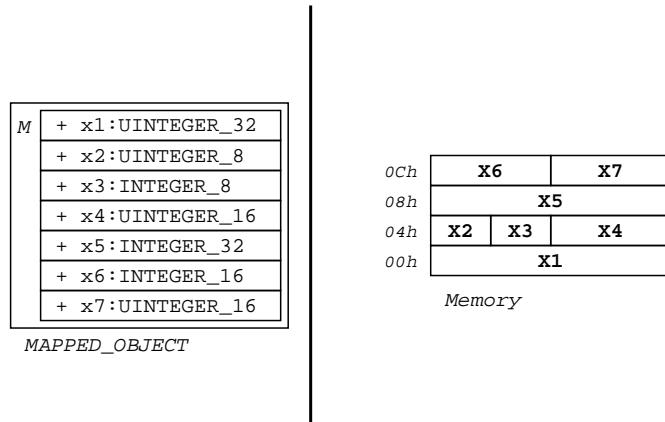
Section Mapping
```
  + x1:UINTEGER_32;   // 4 bytes, unsigned
  + x2:UINTEGER_8;    // 1 byte,  unsigned
  + x3:INTEGER_8;     // 1 byte,  signed
  + x4:UINTEGER_16;   // 2 bytes, unsigned
  + x5:INTEGER_32;    // 4 bytes, signed
  + x6:INTEGER_16;    // 2 bytes, signed
  + x7:UINTEGER_16;   // 2 bytes, unsigned
```

These prototype match exactly a 16 byte physical structure.

```
M │ + x1:UINTEGER_32
  │ + x2:UINTEGER_8
  │ + x3:INTEGER_8
  │ + x4:UINTEGER_16
  │ + x5:INTEGER_32
  │ + x6:INTEGER_16
  │ + x7:UINTEGER_16
```
MAPPED_OBJECT

0Ch	**X6**	**X7**
08h	**X5**	
04h	**X2** **X3**	**X4**
00h	**X1**	

Memory

⚠ 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:UINTEGER;   // Compiler will stop in error
Section Public
  + count:INTEGER := 3;   // Compiler will stop in error
  - slot:INTEGER <- /* ... */ // Ok

The mapping can also be used to represent files.

3.3.5   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 on 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 on the context and cannot be anticipated during compilation.
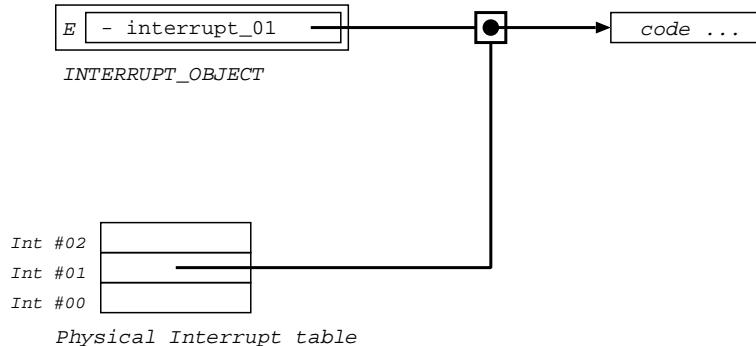
Section Interrupt
  - **interrupt_01** <- /* ... */;

You must define as C external (3.15 page 112) 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");

';
```

3.3.6 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 3.15 page 112.

3.3.7 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 4 kind of sections of this type: the **Private** section, the **Public** section, the **Directory** 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 has also the right to call it.

Section Directory
This type of section gives access to all prototypes contained in the same directory as your prototype. Prototypes contained in a sub-directory also have access to these slots. It permits easy access securisation while organizing your prototypes into directories and sub-directories.

Section Public
It's the most permitive 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;

Section Private
  + **slot_private** <- /* ... */
Section SELF
  + **slot_self** <- /* ... */
Section FIRST
  + **slot_list1** <- /* ... */
Section FIRST,SECOND
  + **slot_list2** <- /* ... */
Section Public
  + **slot_public** <- /* ... */

Object	**slot_private**	**slot_self**	**slot_list1**	**slot_list2**	**slot_public**
self object only, Not its descendants	OK	OK	OK	OK	OK
self 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;

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;
```

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;
```

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;
```

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;

Section Private
  + slot_private <- /* ... */
Section SELF
  + slot_self <- /* ... */
Section FATHER
  + slot_list1 <- /* ... */
Section FIRST
  + slot_list2 <- /* ... */


Section Header
  + name := SON;

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;
```

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;
```

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;
```

Section Private
```
   + slot_private <- /* ... */
```
Section FIRST
```
   + slot_list1 <- /* ... */
```

Section Header
```
   + name := SON;
```

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
  /* ... */

3.4   Type names

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

  + **color**:INTEGER;

3.4.1   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);

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

⚠ The name of the prototype is the entire name, with '(' and ')'.

Section Header
  + **name** := TEST;

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.

3.4.2 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;
```

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

Section Header
```
  + name := SON;
```

 Section Inherit
```
  - parent: FATHER:= FATHER;
```
Section Public
```
  + to_string arg:INTEGER :STRING <- /* ... */ // Ok, follow the same profile
```

Section Header
```
  + name := SON;
```

Section Inherit
```
  - parent:FATHER := FATHER;
```
Section Public
```
  + to_string arg:REAL :ARRAY(CHARACTER) <- // Error: not the same profile
  /* ... */
```

3.4.3 Particular type: SELF type

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

Section Header
```
  + name := EXAMPLE;
```

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

 Here the SELF type is exactly EXAMPLE.
 Another example using inheritance:

Section Header
```
  + name := FATHER;
```

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

```
  );
```

Section Header
  + **name** := SON;

Section Inherit
  - **parent**:FATHER := FATHER;

Section Header
  + **name** := TEST;

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 which returns SELF type is defined in a parent, it's the current object which 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;

Section Public
  - **a**:SELF;

Section Header
  + **name** := SON1;

Section Inherit
  - **parent**:FATHER := FATHER;
Section Public
  - **affect_a** <- (a := SELF;); // Here SELF is SON1

Section Header
  + **name** := SON2;

Section Inherit
  - **parent**:FATHER := FATHER;

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


Section Header
```
  + name := TEST;
```

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
);
```

3.4.4   Particular type: FIXED_ARRAY(E) type

The FIXED_ARRAY(E) type is the object representation of a values' vector.

```
  + my_vector:FIXED_ARRAY(INTEGER);
```

```
my_vector := (0, 1);       // It's one vector with 2 values.
my_vector := (0, 1, 2, 3); // It's one vector with 4 values.
```

The prototype FIXED_ARRAY(E) becomes a non-mutable collection (static)

3.5   Prefix of types

3.5.1   Expanded type

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 **-** (see 3.6.4). If the slot `name` of a prototype is followed by `Expanded`, slots of this type are automatically `Expanded`.
    For example :

Section Header
```
  + name := Expanded FOO;
```

Section Public
```
  - slot_foo:FOO; // <=> Expanded FOO
```

3.5.2   Strict type

If a slot use the keyword **Strict**, its value is exactly this type. You can't affect this slot with a son of the same type. It establish a strong restriction permitting exchange and manipulation between referenced and expanded objects (see 3.5.2). **Strict** can only be used on reference objects. **Strict Expanded** is therefore illegal.

If the slot `name` of a prototype is defined with the `Strict` word, slots of this type are automatically `Strict`.

For example :

Section Header
  + **name** := **Strict** FOO;

Section Public
  – **slot_foo**:FOO; // <=> Strict FOO

3.6   Slots

3.6.1   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;

  – **default**:= NULL;

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

3.6.2   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;
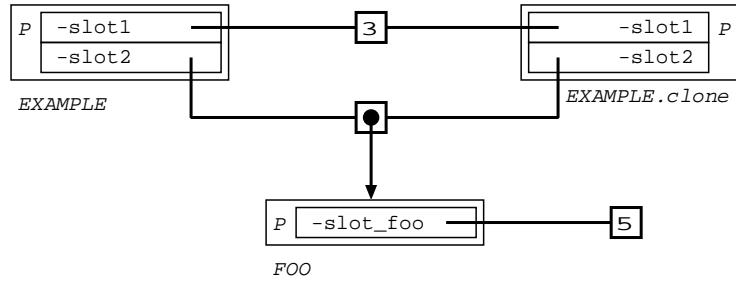
Section Public
  – **slot_foo**:INTEGER := 5;

Section Header
  + **name** := EXAMPLE;

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 3.6.4 page 74.

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;
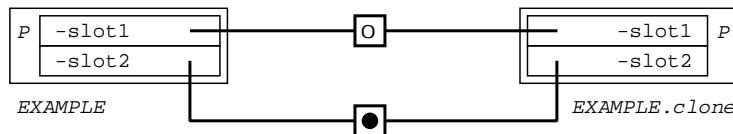
Section Public
    - **slot_foo**:INTEGER := 5;

Section Header
    + **name** := EXAMPLE;

Section Public
    - **slot1**:INTEGER;
    - **slot2**:FOO;

Assignment

Section Header
   + **name** := FOO;

Section Public
    - **slot_foo**:INTEGER := 5;

Section Header
    + **name** := EXAMPLE;

Section Public
    - **slot1**:INTEGER;
    - **slot2**:FOO;

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

Section Header
```
+ name := TEST;
```

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

```
P | -slot1 |──────────[ O ]──────────| -slot1 | P
  | -slot2 |                          | -slot2 |
  obj_example1 ──────[ ● ]────── obj_example2
```

```
  obj_example1.inc_slot1;
```

```
P | -slot1 |──────────[ 1 ]──────────| -slot1 | P
  | -slot2 |                          | -slot2 |
  obj_example1 ──────[ ● ]────── obj_example2
```

```
  obj_example2.inc_slot1;
```

```
P | -slot1 |──────────[ 2 ]──────────| -slot1 | P
  | -slot2 |                          | -slot2 |
  obj_example1 ──────[ ● ]────── obj_example2
```

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

```
P | -slot1 |──────────[ 2 ]──────────| -slot1 | P
  | -slot2 |                          | -slot2 |
  obj_example1 ──────[ ● ]────── obj_example2
                        │
                        ▼
              P | -slot_foo |──────[ 5 ]
                obj_foo1
```
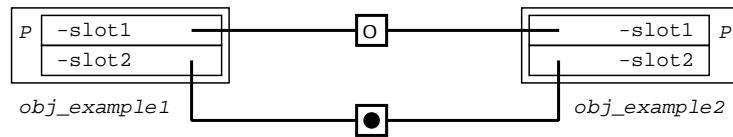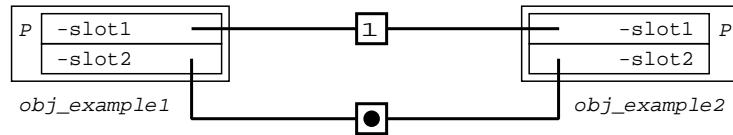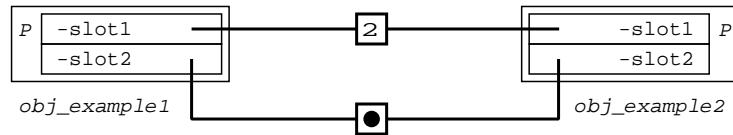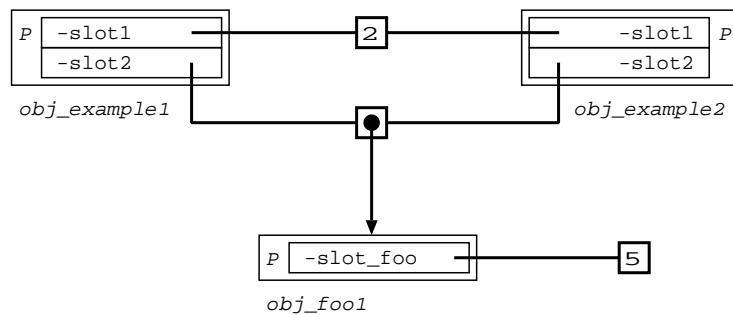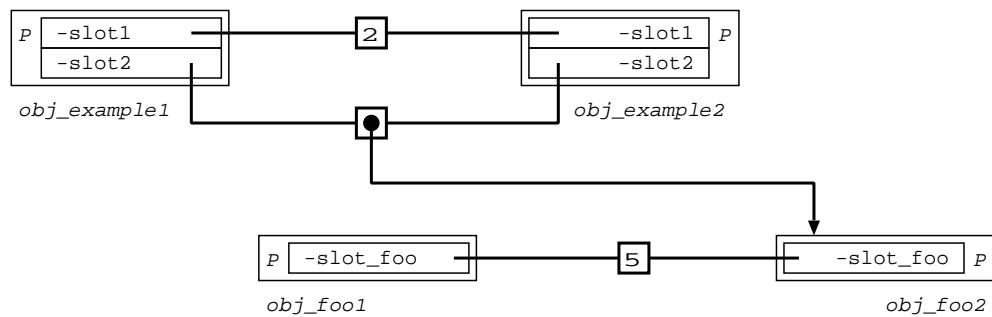
```
obj_foo2 := FOO.clone;
obj_example2.set_slot2 obj_foo2;
```


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

Section Header
  + **name** := FATHER;

Section Header
  + **name** := SON;

Section Inherit
  - **parent**:FATHER := FATHER;

Section Header
  + **name** := OBJECT_OTHER;

Section Header
  + **name** := TEST;

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 3.6.4 page 74).

### 3.6.3   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;

**Section Public**
  - **slot_foo**:INTEGER := 5;

**Section Header**
  + **name** := EXAMPLE;

**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 3.6.4 page 74.

⚠ 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;

**Section Public**
  - **slot_foo1**:INTEGER := 5;
  + **slot_foo2**:INTEGER := 3;

**Section Header**
  + **name** := EXAMPLE;

**Section Public**
   + **slot1**:INTEGER;
   + **slot2**:FOO;



*EXAMPLE*                                    *EXAMPLE.clone*

**Assignment**


**Section Header**
   + **name** := FOO;

**Section Public**
   − **slot_foo1**:INTEGER := 5;
   + **slot_foo2**:INTEGER := 3;


**Section Header**
   + **name** := EXAMPLE;

**Section Public**
   + **slot1**:INTEGER;
   + **slot2**:FOO;
   − **inc_slot1** <− ( slot1 := slot1 + 1; );
   − **set_slot2** f:FOO <− (slot2 := f; );


**Section Header**
   + **name** := TEST;

**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**;

obj_example2.**inc_slot1**;



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



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



### 3.6.4   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;

**Section Public**
  – **slot_foo1**:INTEGER := 5;
  + **slot_foo2**:INTEGER := 4;


**Section Header**
  + **name** := EXAMPLE;

**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 (3.6.2) (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;

**Section Public**
  – **slot_foo1**:INTEGER := 5;
  + **slot_foo2**:INTEGER := 4;


**Section Header**
  + **name** := EXAMPLE;

**Section Public**
  – **slot1**:INTEGER := 3;
  – **slot2**:FOO;

Let's now see with Non Sharable slots.

**Section Header**
  + **name** := FOO;

**Section Public**
  – **slot_foo1**:INTEGER := 5;
  + **slot_foo2**:INTEGER := 4;

**Section Header**
  + **name** := EXAMPLE;

**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), for it, use `Strict` type. See also 3.9.1 page 85.

**Section Header**
  + **name** := FATHER;

**Section Header**
  + **name** := SON;

**Section Inherit**
  – **parent**:FATHER := FATHER;

**Section Header**
  + **name** := TEST;

**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;

**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;

**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;

**Section Public**

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

obj_foo1 · P -slot_foo1 +slot_foo2 4
obj_foo2 · -slot_foo1 P 4 +slot_foo2

obj_example1 · P +slot1 Exp · P -slot_foo1 +slot_foo2 4 · -slot2 Exp
5
obj_example2 · -slot_foo1 P 4 +slot_foo2 · +slot1 Exp P · -slot2 Exp

-slot_foo1 P 4 +slot_foo2

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

obj_foo1 · P -slot_foo1 +slot_foo2 5
obj_foo2 · -slot_foo1 P 4 +slot_foo2

obj_example1 · P +slot1 Exp · P -slot_foo1 +slot_foo2 5 · -slot2 Exp
6
obj_example2 · -slot_foo1 P 4 +slot_foo2 · +slot1 Exp P · -slot2 Exp

-slot_foo1 P 4 +slot_foo2

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

obj_foo1 (top section)

P	-slot_foo1			-slot_foo1	P
	+slot_foo2	6	4	+slot_foo2	

obj_foo1   *obj_foo2*

P	+slot1 Exp						+slot1 Exp	P
	P	-slot_foo1			7	P	-slot_foo1		
		+slot_foo2	5			5	+slot_foo2		
	-slot2 Exp						-slot2 Exp	

obj_example1   *obj_example2*

	-slot_foo1	P
4	+slot_foo2	

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

P	-slot_foo1			-slot_foo1	P
	+slot_foo2	6	5	+slot_foo2	

obj_foo1   *obj_foo2*

P	+slot1 Exp						+slot1 Exp	P
	P	-slot_foo1			8	P	-slot_foo1		
		+slot_foo2	5			5	+slot_foo2		
	-slot2 Exp						-slot2 Exp	

obj_example1   *obj_example2*

	-slot_foo1	P
5	+slot_foo2	

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

P	-slot_foo1			-slot_foo1	P
	+slot_foo2	7	5	+slot_foo2	

obj_foo1   *obj_foo2*

P	+slot1 Exp						+slot1 Exp	P
	P	-slot_foo1			9	P	-slot_foo1		
		+slot_foo2	5			5	+slot_foo2		
	-slot2 Exp						-slot2 Exp	

obj_example1   *obj_example2*

	-slot_foo1	P
7	+slot_foo2	

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


3.7   Slot descriptors

An object may hold any number of slots which must be in the section codes (see 3.3.7 page 59).
Slots can contain data (values and references) or methods (code).

3.7.1   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 3.10 page 3.10.

```
- 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 by the type of their arguments or the type of return. You can't also have 2 slots which differ 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 !
```

3.7.2   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 limit[1]

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
  );
```

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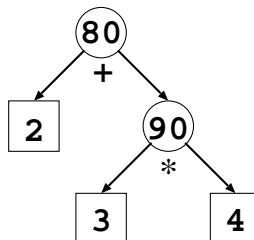
- '+' left 80 other:SELF :SELF <- (Self - -other);
- '*' left 90 other:SELF :SELF <- /* ... */
```

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 127).

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

---

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

3.7.3 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 3.13.2 page 102. 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 developper 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 developper can switch off these assertions in the final delivery version by using a simple compile-time option.

3.7.4   Variable-argument list

You can define an argument representing a variable sized vector of parameters of a given type. In this case, we use the particular type FIXED_ARRAY(E) 3.4.4.

```
- add list:FIXED_ARRAY(INTEGER) <-
// Append several integer in my collection.
(
  (list.lower).to (list.upper) do { j:INTEGER;
    add_last (list.item j);
);
```

Calls are then directly done by simple lists :

```
my_object.add (0,1);
my_object.add (0,1,2,3);
my_object.add (0);
```

3.8   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's declaration (see section 3.7.2 page 81).

⚠ 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 3.7.1 page 80.

3.9   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 3.3.2). 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 :

> **- 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 3.9.3 page 86.

- **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

3.9.1 Typing rules

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

A B	type	Self	Expanded type	Strict type
type	B.sub A	B.sub A	B.sub A	B.sub A
Self	*fail*	A = B	A = B	A = B
Expanded type	*fail*	A = B	A = B	A = B
Strict type	*fail*	A = B	A = B	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 type reference.

3.9.2 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;

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.

3.9.3   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 on 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.

The second use of '?=' concerns the affectation of `Strict` type slots with a non `Strict` type value.

```
( + foo:APPLE;
  + bar:Strict APPLE;
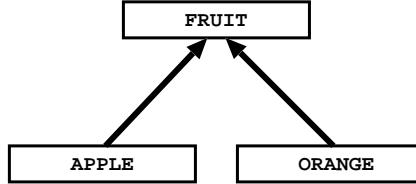
  (test).if {
    foo := APPLE.clone;
  } else {
    foo := APPLE_GREEN.clone;
  };

  bar ?= foo; // 'bar = foo', if 'foo' is extactly APPLE.
```

3.9.4 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**))

3.9.5 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};
```

3.10 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**
(*local*;	(*local*;	(*local*;
expr1;	*expr1*;	*expr1*;
expr2;	*expr2*;	*expr2*;
expr3;	*expr3*;	*result*,
expr4;	*result*	*result2*
)	)	)

A list is immediately evaluated when reached by the execution flow. Thus, a routine which argument is the (single-statement) list '(3 + 2)' receives as argument the result of the evaluation, 5, not the list itself[2].

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

You can also have code and return value for arguments:

```
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.

3.10.1   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
```

---

[2]This is the contrary for statement blocks, see section 3.11 page 93.

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 et the closing parenthesis (or an "empty statement" only), nothing is returned from the list after it has been evaluated.

3.10.2   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 3.7. 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 after having seen 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 list 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.

3.10.3   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 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 '-' preserve 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;
)
```

3.11 Statement blocks

Statement blocks, or simply "blocks", have a number of similarities with lists (see section 3.10 page 87).

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 use of blocks in Lisaac is to implement library-defined control structures (see section 3.11.3 page 97).

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
{ *local*;	{ *local*;	{ *local*;
expr1;	*expr1*;	*expr1*;
expr2;	*expr2*;	*expr2*;
expr3;	*expr3*;	*result*,
expr4;	*result*	*result2*
}	}	}

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
```

3.11.1   Return values of blocks

The value returned by a block is determined exactly like that of a list (see section 3.10.1 page 88).

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.

A block is not context sensitive, if it does not use local variables or of method parameters where it is declared.

1. A non context sensitive block can set any slot of type BLOCK ({ }) or be returned as result of a method.

2. A context sensitive block can set a local in '+' of type BLOCK ({ }) or a parameter of type BLOCK ({ }) when sending a message.

So it can not set a slot of an other BLOCK ({ }) object, neither a BLOCK ({ }) local in '-' and neiher be send as result of a method.

Example of non context sensitive block.

```
+ data_1:INTEGER;
- data_2:INTEGER;

- method_1 param_1:INTEGER <-
( + local_1:INTEGER;
  - local_2:INTEGER;
  + block_2:{ INTEGER; };

  block := { param_2:INTEGER; // Non context sensitive block !
    + local_3:INTEGER;
    + local_4:INTEGER;

    (data_1 + data_2 + local_2 + local_3 + local_4 + param2).print;
```

```
  };

  block_2 := { param_2:INTEGER; // context sensitive block !
    + local_3:INTEGER;
    + local_4:INTEGER;

    (
     data_1 + data_2 + local_2 + local_3 + local_4 + param2+
     param_1 + local_3
    ).print;
  };
);
```

A block not sensitive context can be evaluated anywhere, and at any times!

A block sensitive context must be evaluated with the context of the method always in stack (this guaranteed by rule 2).

3.11.2   Declaration of blocks

Without argument, without result (block with a just code) :

```
  b:{ };
```

Without argument and one boolean result :

```
  b:{ BOOLEAN };
```

Without argument with two results, one integer and one boolean:

```
  b:{ INTEGER, BOOLEAN };
```

One integer argument, without result:

```
  b:{ INTEGER; };
```

Two arguments (one integer and one boolean), without result:

```
  b:{ (INTEGER, BOOLEAN); };
```

Two integers arguments, with one boolean result:

```
  b:{ (INTEGER, INTEGER); BOOLEAN };
```

Complex example:

```
  b:{ (INTEGER, STRING); BOOLEAN, CHARACTER };
```

3.11.3   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:{ } <- /* ... */
```

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**:{ INTEGER };

- **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:{ };
  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:{ BOOLEAN } :BOOLEAN <- FALSE;

In the TRUE prototype:

- '**&&**' **left** 20 other:{ BOOLEAN } :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:{ } else false_block:{ } <-
(
  false_block.value;
);
```

In the TRUE prototype:

```
- if true_block:{ } else false_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:{ BOOLEAN } <-
(
  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:{ } <-
(
  (Self<=limit_up).if {
    blc.value Self;
    (Self + 1).to limit_up do blc;
  };
);
```

3.11.4 Argument and local variables in statement blocks

Locals in blocks are declared like locals in lists (see section 3.10.3 page 92):

```
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.

3.12   Export / Import automatic conversion object

3.12.1   Auto-Export object

Sometimes you want to transform an object in another object, especially for the numbers. This can be done with the "Auto-export" 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;

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;

Section Public
  – **main** :=
  (+ a:PROTO1;
    + b:PROTO2;
    /* ... */
     b := a;   // similar to: b := a.**to_proto2**;
    /* ... */
  );
```

3.12.2 Auto-Import object

In the Header section, in the slot **name**, you can define the prototypes in which the object can be "auto-imported" with the '<-' symbol.

Section Header
  + **name** := PROTO1 <- PROTO2,PROTO3;

Section Public
  - **from_proto2** elt:PROTO2 :PROTO1 <- (/* ... */);
  - **from_proto3** elt:PROTO3 :SELF   <- (/* ... */);

In the public section, you must define functions called **from_**_name_of_type_ (here **from_proto2** and **from_proto3**) which are automatically called when there is an auto-import. These function must return the corresponding SELF type.

Section Header
  + **name** := TEST;

Section Public
  - **main** :=
(+ a:PROTO1;
  + b:PROTO2;
  /* ... */
   a := b;   // similar to: a := PROTO1.**from_proto2** b;
  /* ... */
);

Auto-export (or Auto-Import) 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-Export or Auto-Import is not inherited.

3.12.3 Complex import / Export with vector object

You can also define vectors for an importation or exportation.

Section Header
  + **name** := PROTO <- (INTEGER, STRING_CONSTANT);

Section Private

  + count:INTEGER;

  + name:STRING_CONSTANT;

Section Public

  - **from_integer_string_constant** (n:INTEGER, s:STRING_CONSTANT) :PROTO <-
  (

```
    create (n,s)
  );


  - create (n:INTEGER, s:STRING_CONSTANT) :PROTO <-
  ( + result:PROTO;

    result := clone;
    result.make (n,s);
    result
  );


  - make (n:INTEGER, s:STRING) <-
  (
    count := n;
    name  := s;
  );
```

This can be used to simplify objects initilization :

Section Header
```
  + name := TEST;
```

Section Public
```
  - main :=
  ( + a:PROTO;

    /* ... */
     a := (1, "Benoit");  // similar to: a := PROTO.create (1, "Benoit");
    /* ... */
  );
```

3.13   Tools for programming by contract

Compiler furnishes 3 native functions :

```
  - debug_level:INTEGER           // Flag indicating the level of debug mode
  - top_runtime_stack:POINTER   // Give the top stack pointer.
  - print_runtime_stack_on ptr:POINTER // Print the stack from 'ptr'
```

Example: output with **print_runtime_stack**

```
============== BOTTOM ==============
Line #20 Column #9 in HELLO (./hello.li).
  - main <-
        ^
Line #39 Column #18 in HELLO (./hello.li).
    tab.put 3 to 5;
                 ^
Line #217 Column #22 in FAST_ARRAY (/lib/collection/fast_array.li).
    ? {valid_index i};
                    ^
```

```
=============== TOP ==============
User assertion violated.
```

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

```
- '?' <-
// User assertion without message.
( + ptr:POINTER;

  ptr := top_runtime_stack;
  ((debug_level >=10) && {! value}).if {
    crash_on ptr with_message "User assertion violated.";
  };
);
```

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

```
- crash_on ptr:POINTER with_message msg:ABSTRACT_STRING <-
(
  print_runtime_stack_on ptr;
  msg.print;
  '\ n'.print;
  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 on 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 3.15.2
```

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 on 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.
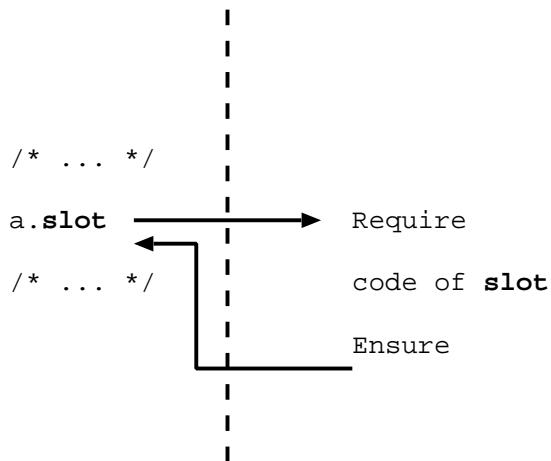
3.13.1   degrees of assertions

debug_level	Usage	Example
$\geq n$	*No define*	2 ? {**valid_index** i}
≥ 5	Require code	-? {**valid_index** i}
≥ 5	Require code with message	{**valid_index** i} -? "No valid index."
≥ 10	User code	? {**valid_index** i}
≥ 10	User code with message	{**valid_index** i} ? "No valid index."
≥ 15	Ensure code	+? {**valid_index** i}
≥ 15	Ensure code with message	{**valid_index** i} +? "No valid index."

Note: The definitions of those slots are located in the BLOCK prototype.

3.13.2   Requires and Ensures

To secure programming, you can put requires, ensure and invariant into the code: conditions which 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]. Require is written between the slot header and the code.

```
- slot <-
[
   // Require
]
(
   // Code
)
[
   // Ensure
];
```

In Require or Ensure section you can write your code as 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.

3.13.3   Invariant

You can define at the end of code invariant conditions, which 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};
 ];
```
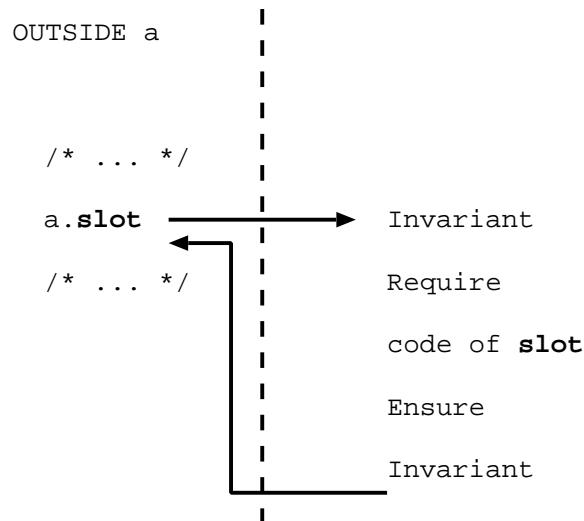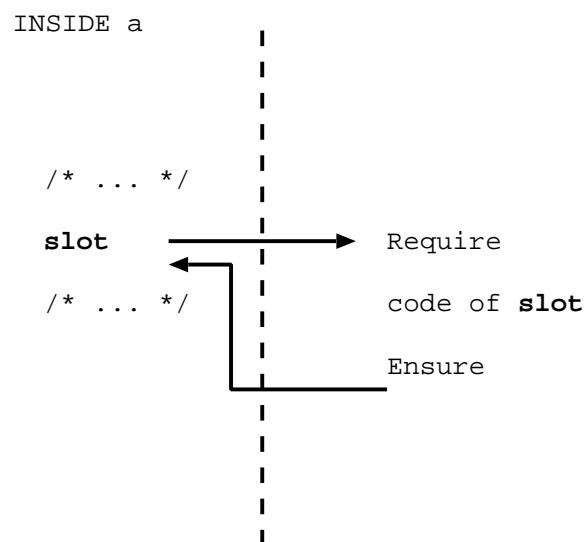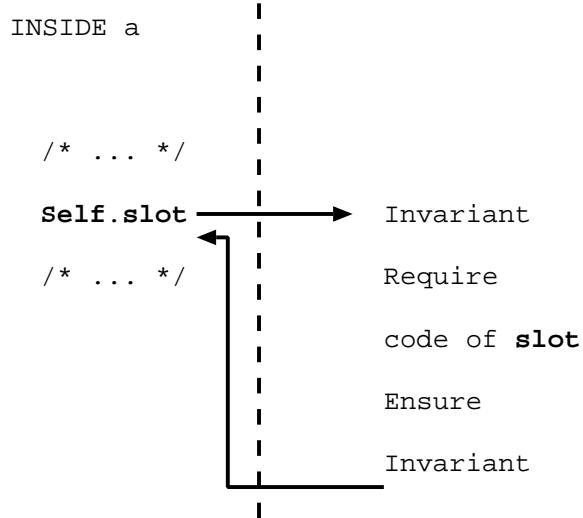
The invariant is verified each time you call a message with the explicit receiver, before and after the call.

```
        OUTSIDE a             ¦
                              ¦
                              ¦
                              ¦
        /* ... */             ¦
                              ¦
        a.slot  ─────────────────────►  Invariant
                ◄──────┐      ¦
        /* ... */      │      ¦          Require
                       │      ¦
                       │      ¦          code of slot
                       │      ¦
                       │      ¦          Ensure
                       │      ¦
                       │      ¦          Invariant
                       └──────────────
                              ¦
```

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

```
        INSIDE a              ¦
                              ¦
                              ¦
                              ¦
        /* ... */             ¦
                              ¦
        slot    ─────────────────────►  Require
                ◄──────┐      ¦
        /* ... */      │      ¦          code of slot
                       │      ¦
                       │      ¦          Ensure
                       └──────────────
                              ¦
                              ¦
                              ¦
```

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

```
INSIDE a                 ┊
                         ┊
                         ┊
    /* ... */            ┊
                         ┊
    Self.slot ───────────┊──────────▶  Invariant
                     ┌──◀┊
    /* ... */        │   ┊              Require
                     │   ┊
                     │   ┊              code of slot
                     │   ┊
                     │   ┊              Ensure
                     │   ┊
                     │   ┊              Invariant
                     └───────────────┘
                         ┊
```

3.13.4   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. It 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 only use **Old** with slots containing no arguments.

The keyword **Result**_x represents the result of the function and can only be used 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 };
];
```
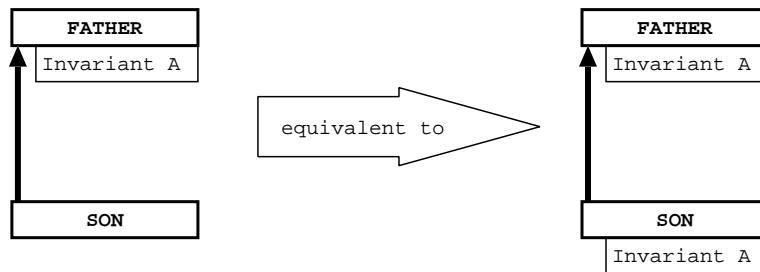
```
- slot2:INTEGER,INTEGER <-
 (
   count + 1,
   count - 1
 )
[
   ? { Result_1 >= 1 };
   ? { Result_2 > 0 };
];
```
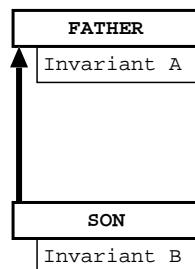
3.13.5   Inheritance

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

If an invariant is defined in an object, it replaces those of its static parent.
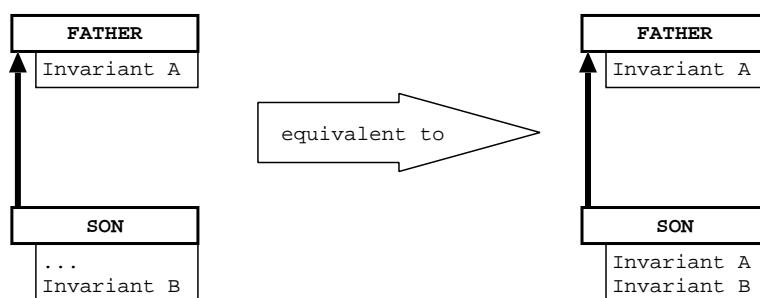
An object can also inherit invariant from its parent and add its own invariant. This is done by using dots (**...**). The invariant of the parent is inserted where the dots are written.

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

The same pattern is used with Require and Ensures. If an object has a slot with no Require or/and Ensure, this slot inherits the Require / Ensure of the corresponding parent's slot (if any). The Require / Ensure defined in a slot replace those of the object's parent. Dots are the way to append a new Require / Ensure without overwritting the conditions inherited from the parent.

3.14 COP: Concurrent Object Prototypes

3.14.1 Description

The COP Model

Version 0.3 of this specification introduces a concurrency model (COP) that let you run multiple execution paths in parallel, evolving in separate environments. This model is well suited for multi-processor or multi-core machines as well as multi-node clusters, thanks to the independence of the different execution environments. This model is implemented in a language-natural way, so no specific library or prototype needs to be used.

Concurrency in Lisaac is achieved using the scope sign of the 'name' slot. The meaning of the sign is kept: on one hand, you have '+', which stands for a slot private to an object's instance and for COP means that the object is specific to an execution environment without being accessible from the others when applied to the 'name' slot. On the other hand, you have the '-' scope which means that this slot is shared among every instance of the prototype; in the COP model, this scope is used for an object defining its own execution environment and which may be accessed from others.

We can see that only a '-' object can define an execution environment. In fact, there are as many environments as there are '-' objects. Inside each environment, '+' objects which are referenced in the '-' object slots (included inherited objects) are also present. Thus an environment contains :

- the '-' object that defines it, which we call the *interface object*;

- the '+' objects referenced through the slots of the interface object, including inherited ones (see the following sections for more details about inheritance);

- the objects created (cloned) after the environment creation;

- obviously, every object expanded in the aforementioned objects.

The Method Call Queue

To each '-' object is attached a *method call queue*. When a call is made to a '-' object, the call is actually stored in its call queue. The method is then executed when every previous call have been processed: there is no concurrency inside an environment. Once there is no pending call any more, the objects is put in a *sleep* state until in receives another call.

Following the implementations, the method call queue can be a shared memory segment using mutual exclusion, a network socket, a native facility. . .

In the case where a '-' object calls one of its methods, the call is directly executed; it doesn't go through the queue. This happens when the the call is done on *self* implicitely. When the receiver is explicit (`Self.method;`), the call is done via the queue.

Waiting for the Return Value
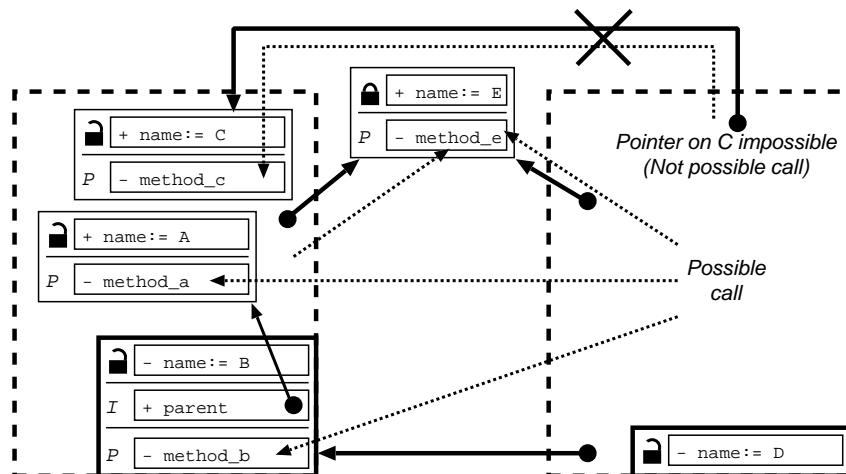
The calling object can chose to either wait for the result of the method call (its return value) or continue the execution in a parallel fashion. When the method returns no value, it is always executed concurrently without waiting for its termination. If it returns a value, then the calling object waits for its termination only if it makes use the this value; otherwise it behaves as if there was no such value, as in the previous case.

Advantages of the COP Model

The clear advantage of this model is the total lack of synchronization needs, even inside the environments. This comes from the fact that there can be at most one execution in an environment at a time.

Since '+' objects aren't shareable among execution environments and '-' ones define their own, the only thing that might be manipulated by several simultaneous executions is the method call queue. But as it is compiler's business to manage it, there is nothing to do from a programmer point of view. Moreover, creating a new environment is as simple as creating or cloning a '-' object.

3.14.2   Communication Between Environments

Non-Mutable Objects

For the purpose of inter-environment communication, we need a new kind of object that can pass beyond the bounds of an environment, beside '-' and expanded objects. These objects are referred to as *non-mutable objects*. Such an object has slots that are all non-mutable: they hold a value that cannot be changed or a reference to a non-mutable object that cannon be changed either; so this is a recursive property.

Any object can be turned into a non-mutable object, using the 'to_non_mutable' method. This method is recursive: it will be called onto every object contained in each slot. Once a non-mutable object has been created this way, it cannot be turned back into a mutable state.

However it is still possible to create a clone of a non-mutable object using its 'clone' method. But beware, the clone itself will be mutable, but not the objects referenced through its slots. So you have to manually call 'clone' on the slots that require it. You also have the possibility to use the 'deep_clone' method that will recursively clone the object and all the ones contained in its slots.

Object Sending

Only '-' objects may be accessed from other execution environments by calling their methods. This corresponds to the message-passing style of object-oriented programming. Given this property, the parameters that can be passed as method arguments are ones that can cross environment bounds. Namely they are one of the following:

- a '-' object;

- an Expanded object;

- an object containing no data;

- a non-mutable object or a '+' object *that has been made non-mutable.*

While the first three can be determined at compile time, the last one is checked at runtime. In case of violation, the program will either fail to compile or crash.

3.14.3 Typing Rules

Inheritance Rules

If a '-' object inherits from a '+' object, a clone of this object implies the creation of a new execution environment with the duplication of the '+' parent. This is actually the case for every slot of the object: the '`deep_clone`' method is thus called instead of the '`clone`' one.

A '-' parent object, being accessible from everywhere, doesn't get duplicated this way; it is useless since its purpose is to be shared among other environments. A '-' parent can then be inherited by any other object, being '-' or '+'. A '-' parent cannot be expanded.

It is worth noting that an object can inherit from several '-' objects; multiple different environments can then be present in a single inheritance tree, as in the following example:

```
Section Header
  - name := VIDEO;


Section Public
  - line (x1, y1: INTEGER) to (x2, y2: INTEGER) <- (...);



Section Header
  - name := WINDOW;


Section Inherit
  + parent_video:VIDEO := VIDEO;


Section Public
  - line (x1,y1:INTEGER) to (x2,y2:INTEGER) <-
  ( + new_x1, new_y1, new_x2, new_y2: INTEGER;
    // Clip and translate coordinates into new_*, then send back to VIDEO
    parent_video.line (new_x1, new_y1) to (new_x2, new_y2);
  );
```
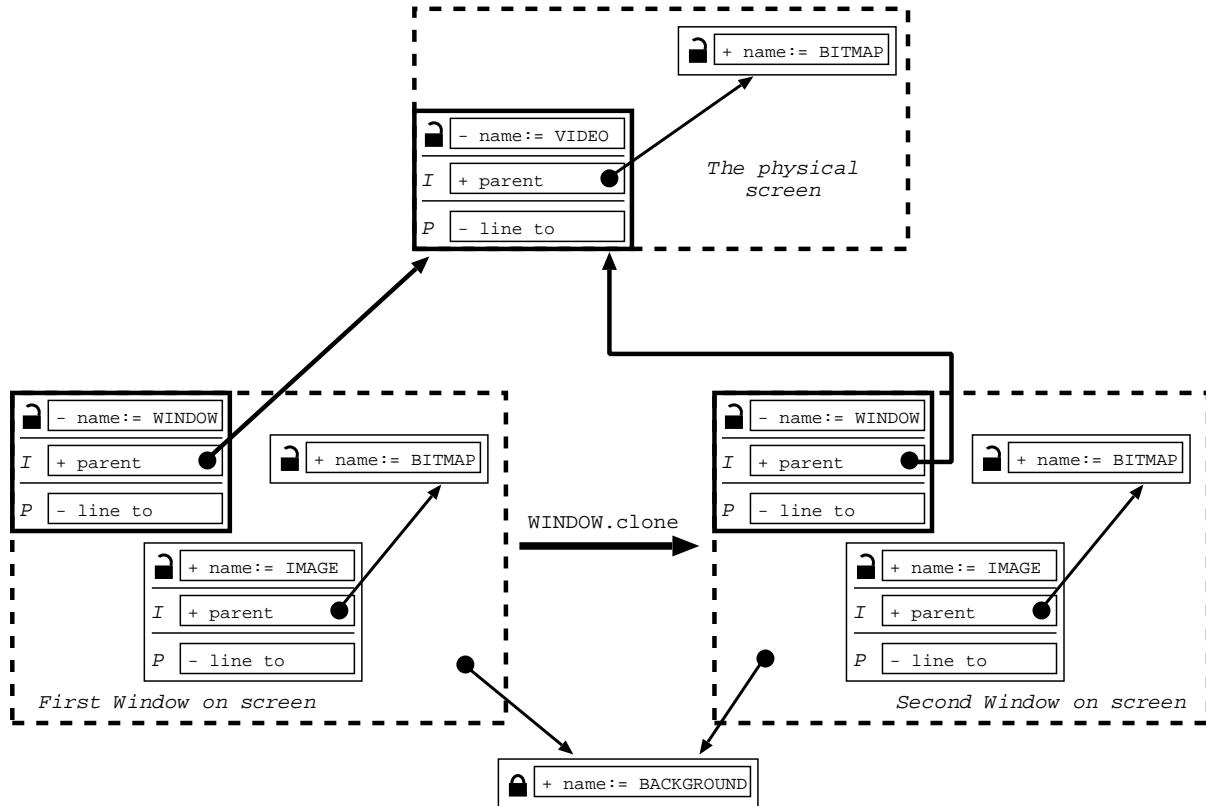
In this example, the VIDEO object and the WINDOW objects each run in a different environment due to the '-' scope of their '*name*' slot.

The last case, namely a '+' object inheriting from another '+' object, is just the "classic" inheritance, out of the scope of the COP specification.

Assignment Rules

In addition to type checking, assignments must deal with the scope of an object when it is being assigned to a variable. While a variable of one type can hold an object whose type is any of the descendents of the variable type (or the variable type itself), it can only receive an object whose type is of the same scope. A '-' variable can only hold a '-' object and a '+' variable can only hold a '+' object. For instance:

```
Section Header
  - name := A;


Section Header
  + name := B;

Section Inherit
  + parent:A := A;


Section Header
  - name := C;
```
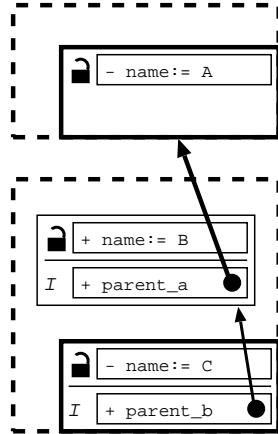
```
Section Inherit
  + parent:B := b;
```



```
var:A;
```

```
var := A; // OK.
var := B; // Denied: 'var' is '-' (A) whereas 'B' is '+'!
var := C; // OK.
```

In this example, *B* and *C* are in the same execution environment while *A* has its own.

3.14.4 Creating Execution Environments

Execution environments, which might be mapped onto threads depending on the operating system the program has been compiled for, are only created when '-' objects get used. There are however three ways to achieve creation of a new environment: declaring a '-' prototype, cloning a '-' prototype and using the SEPARATE generic prototype.

'-' Prototypes

Since a prototype is already a living object in prototype-based object-oriented languages, declaring a prototype with the '-' scope applied to the 'name' slot is sufficient to create a new environment.

Cloning a '-' Prototype

A '-' prototype defines a new execution environment. Then cloning a prototype whose 'name' slot has the '-' scope implies creating a new environment along in the process. This is the main method behind massive parallelism, for instance worker threads in a web server.
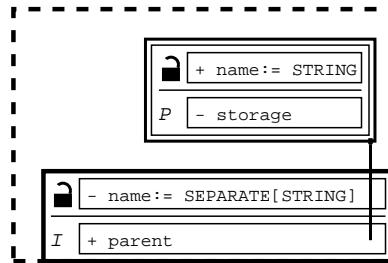
The SEPARATE Generic Prototype

Last but not least, a new environment can be created using the SEPARATE generic prototype. This one lets you manually create new environments "by hand" on a per-object basis. This may be used for example to create a shared object that can be accessed by several environments at once, by passing it through method parameters to objects in other environments since only constant and '-' objects can be sent this way.

Basically, a `SEPARATE[E]` object is just an enclosed expanded E object in a '-' container. It is defined this way:

```
Section Header
  - name := SEPARATE(E);

Section Inherit
  + parent:Expanded E;
```


3.15 Externals

There are two ways to include C code in Lisaac: the **external** slot in **Section Header** or directly in the Lisaac code. It is defined between '.

3.15.1 Slot external

C code defined in the **external** slot is directly included in the code. You can define includes, functions, macros, . . .

Section Header
```
  + name := EXAMPLE;

  - 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.

3.15.2 C code in Lisaac

C code can be inserted anywhere in the code (in the definition of a slot, even in Require / Ensure or Invariant).

```
  - slot <-
  ( + a:INTEGER;
    a := count;
    'fputc('Y',stdout)';
  );
```

You can use a Lisaac local variable or argument preceded by '@'.

```
- slot <-
( + a:CHARACTER;
  'fputc(@a,stdout)';
);
```

⚠ Global variables (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 indicate the return type after **:**.

```
- slot <-
( + a:INTEGER;
  a := '@a ++':INTEGER;
);
```

You can also indicate 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 optimizes 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 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. The compiler will not optimize the code
or delete the external, even if the result is not used.

3.15.3   Lisaac code in C

As explained in 3.3.6, 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).

3.15.4   Lisaac external

Externals composed of a simple integer are Lisaac externals (compiler native functions). Exam-
ple:

```
  '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
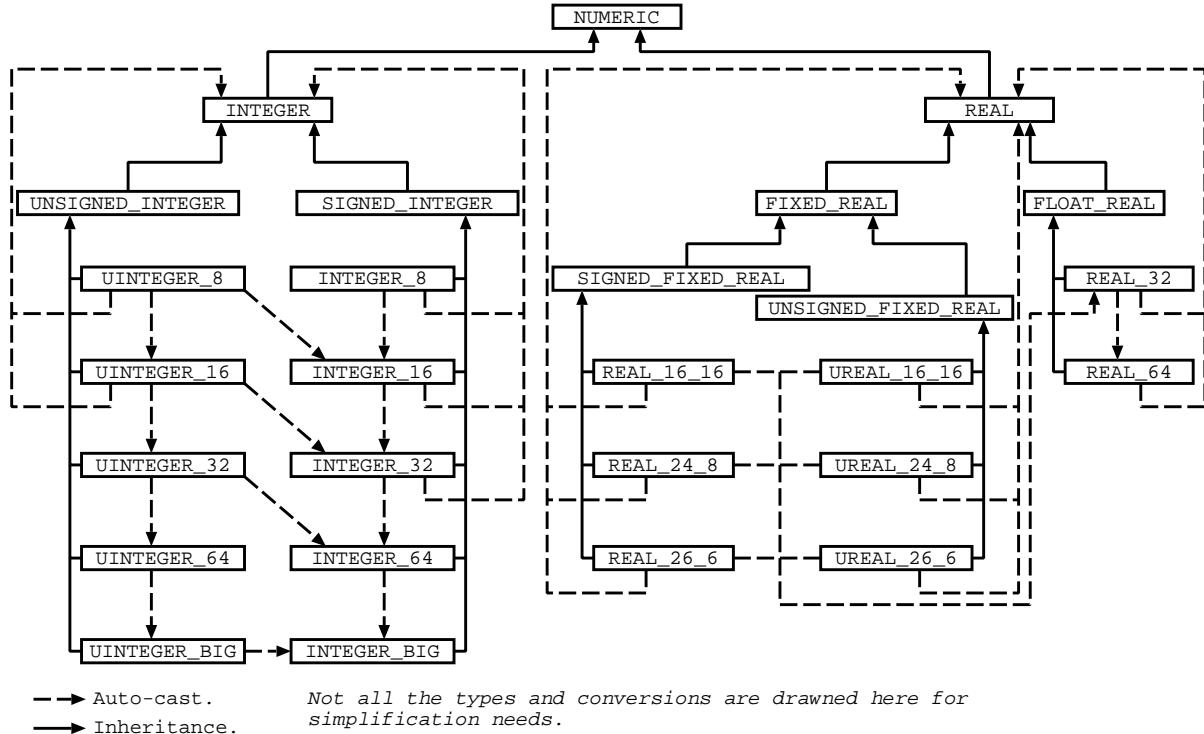
OBJECT is the base prototype which contains all the core functions needed to program efficiently. All the prototypes of the library inherit, directly or not, from this prototype. When defining your own prototypes, don't forget to inherit from OBJECT if you want to use its functions.

The most common 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.
//
- print_runtime_stack     // print stack as defined in 3.13
- 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 inherit from the NUMERIC prototype. There are conversion facilities between
the types, as you can see on the following figure.

Auto-cast.   *Not all the types and conversions are drawned here for*
Inheritance.   *simplification needs.*

For example, a UINTEGER_8 can be converted to a INTEGER_32 without range control, using
auto-export.

```
+ a:INTEGER_32;
+ b:UINTEGER_8;
/* ... */
a := b;
```

When you do not have an explicit value's type, INTEGER is chosen for integer constant, or REAL
for real constant. Any conversion is done after assignment.

```
+ a:INTEGER_8;
/* ... */
a := 3;    // 3 is INTEGER, auto-casted to INTEGER_8, with range control.
```

The most commonly used slots are:

```
//
// Arithmetic operations
//
- '+'  left 80  other:SELF :SELF          // add
- '-'  left 80  other:SELF :SELF          // substract
- '*'  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:{SELF;}         // iterate forward from Self to limit_up
- downto limit_down:SELF do blc:{SELF;} // iterate backward to limit_down

- to limit_up:SELF by step:SELF do blc:{SELF;}        // iterate with step
- downto limit_down:SELF by step:SELF do blc:{SELF;}  // iterate with step

- times blc:{}                            // run Self times blc


//
// Print
//
- to_hexadecimal:STRING      // returns a string with the hexadecimal value
- print                      // prints the value to the standard output


//
// Debug
//
- '?' b:{BOOLEAN}            // assertion, see 3.13
```

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 are a few commonly used slot:

```
//
// Conversions
//
- code:SMALLINT          // ASCII Code
- to_upper:CHARACTER     // returns the equivalent character in upper case
- to_lower:CHARACTER     // returns 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 inherit TRUE and FALSE. By default, a BOOLEAN is FALSE. Conditionnals methods are declared in BOOLEAN but their real definition appears in TRUE or FALSE, the dynamic types of BOOLEAN.

For example, the **if ... else** method is declared deferred (to be redefined) in BOOLEAN.

```
- if b_true:{ } else b_false:{ } <- deferred;
```

In TRUE the slot is redefine:

```
- if b_true:{ } else b_false:{ } <- b_true.value;
```

In FALSE the slot is redefine:

```
- if b_true:{ } else b_false:{ } <- 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 will finally return 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:{} :BOOLEAN
- if true_block:{} else false_block:{}
- elseif cond:{} then block:{} :BOOLEAN
- elseif cond:{} then block:{} else block_else:BLOCK
```

Here are 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 (syntax: {})

BLOCK is a particular prototype because it is implicitly constructed using braces **{** and **}**. For more informations about blocks see 3.11. Here are some functions defined in the BLOCK prototype.

```
//
// Conditional :
```

```
//
- '||' left 10 other:{BOOLEAN} :BOOLEAN // or else
- '&&' left 20 other:{BOOLEAN} :BOOLEAN // and then


//
// Loop :
//
- while_do body:{}                        // while Self is TRUE, evaluate body
- do_while test:{BOOLEAN}                  // evaluate Self while test is TRUE
- until_do body:{}                         // until Self is TRUE, evaluate body
- do_until test:{BOOLEAN}                  // evaluate Self until test is TRUE


//
// Debug
//
- '?'                                     // assertion, see 3.13
```

Example: using loops
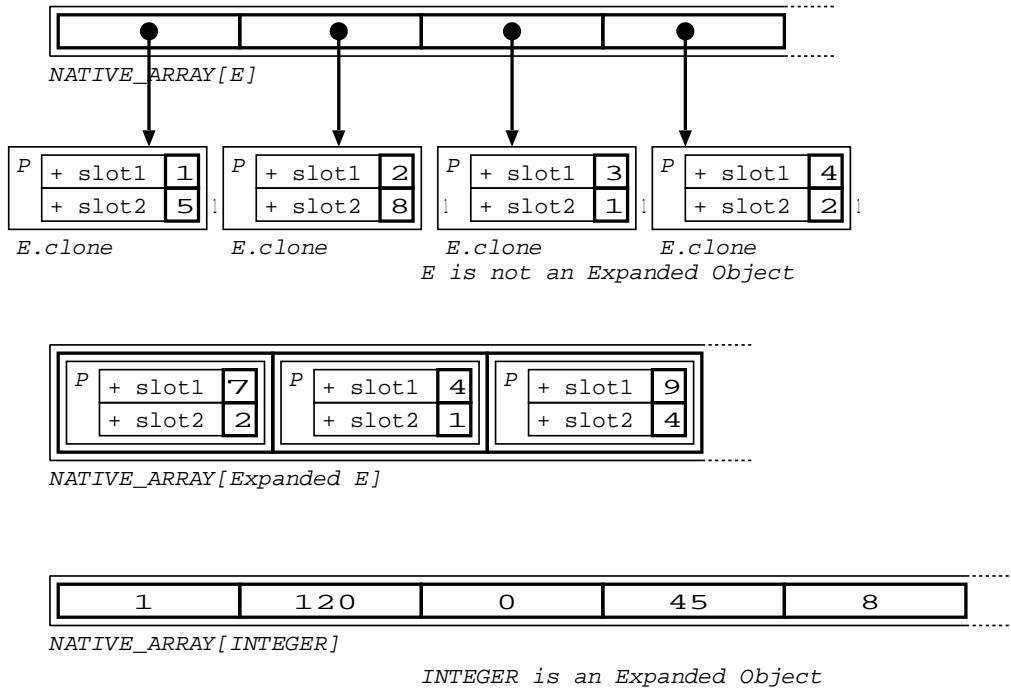
```
+ j:INTEGER;
{ j < 10 }.while_do { j.print; j := j + 1; };
j := 0;
{ j.print; j := j + 1; }.do_until { j >= 10 };
```

4.6   NATIVE_ARRAY

NATIVE_ARRAY is a particular collection prototype using genericity. It's an expanded type which have the particularity to be directly matched on memory data. 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 information about its use, watch the code of this prototype and how it is used in collections.

```
                    ●         ●         ●         ●
  NATIVE_ARRAY[E]

  P + slot1  1   P + slot1  2   P + slot1  3   P + slot1  4
    + slot2  5     + slot2  8     + slot2  1     + slot2  2
  E.clone        E.clone        E.clone        E.clone
                               E is not an Expanded Object


  P + slot1  7   P + slot1  4   P + slot1  9
    + slot2  2     + slot2  1     + slot2  4
  NATIVE_ARRAY[Expanded E]


      1      120       0       45       8
  NATIVE_ARRAY[INTEGER]
                        INTEGER is an Expanded Object
```

4.7 STRING

There are 3 type of string in the library: ABSTRACT_STRING, STRING_CONSTANT and STRING.

ABSTRACT_STRING is an abstract prototype, which defines 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.
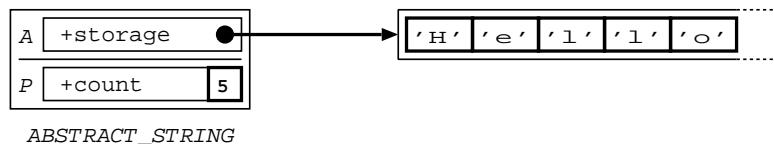
Those 3 prototypes are similar in their internal representation.

Section Header

```
+ name := ABSTRACT_STRING -> STRING;      // can be autocasted in STRING
/* ... */
```
Section ABSTRACT_STRING   // ABSTRACT_STRING and its descendants
```
+ storage:NATIVE_ARRAY(CHARACTER);
```

Section Public
```
/* ... */
```

```
  A  +storage    ●  ───────►  'H' 'e' 'l' 'l' 'o'
  P  +count      5
      ABSTRACT_STRING
```

In ABSTRACT_STRING you can find the following slots (visible from STRING and STRING_CONSTANT, because of the inheritance.

```
//
// Features
//
+ 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 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 with 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  // Appends other to Self
- prepend other:ABSTRACT_STRING // Prepends other to Self
- put ch:CHARACTER to index:INTEGER  // Puts ch at position index
- add_last ch:CHARACTER          // Appends ch to Self
- to_lower                       // Converts all the characters to lower case
```
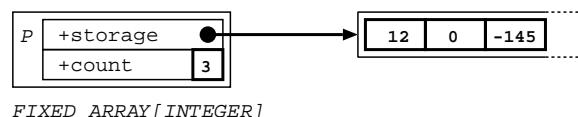
```
- to_upper                          // Converts all the characters to upper case
```

4.8 FAST_ARRAY
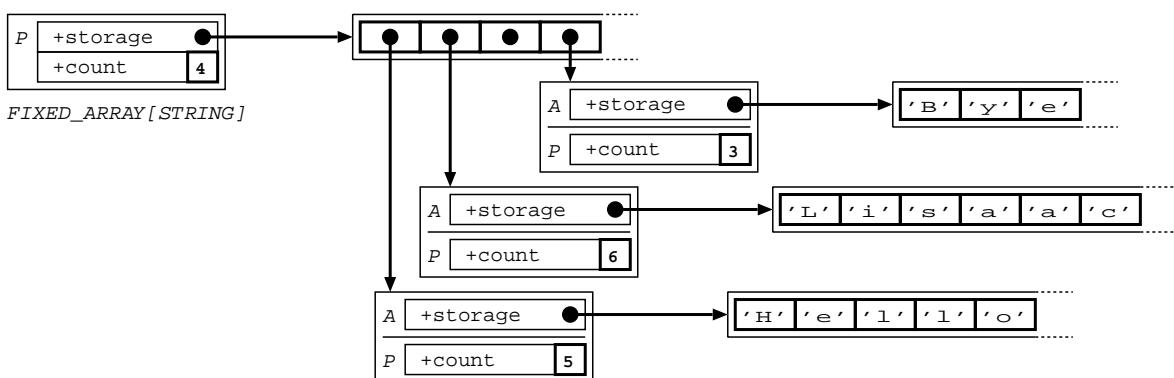
FAST_ARRAY is an array with a fixed lower bound using genericity. You can define a FAST_ARRAY of any object. As for STRING, FAST_ARRAY has a storage:

```
+ storage:NATIVE_ARRAY(E);   // Internal access to storage location
+ count:INTEGER;             // Number of elements of the array
```

Example: FAST_ARRAY of an expanded object

FIXED_ARRAY[INTEGER]

Example: FAST_ARRAY of a non expanded object

FIXED_ARRAY[STRING]

```
//
// 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


//
// 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  // Creates a subarray
- append other:SELF                          // Appends other array
- add_last element:E                         // Appends element
- put element:E to i:NUMERIC                 // Puts 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 multiple inputs.

```
- read_character:CHARACTER    // returns the character read
- read_line_in str:STRING     // puts 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 multiple outputs.

```
- put_character c:CHARACTER    // writes a single character on the output
- put_string s:ABSTRACT_STRING // writes a string
- put_new_line                 // writes 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 executable's call. If you have to get arguments from 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

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

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

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

5.1.2 Cloning

clone		create a clone

5.1.3 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

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

Conditonal *(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, 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:{ } <-
    (? {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
FAST_ARRAY : 1-dimension fixed array
FAST_ARRAY2: 2-dimension fixed array
FAST_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
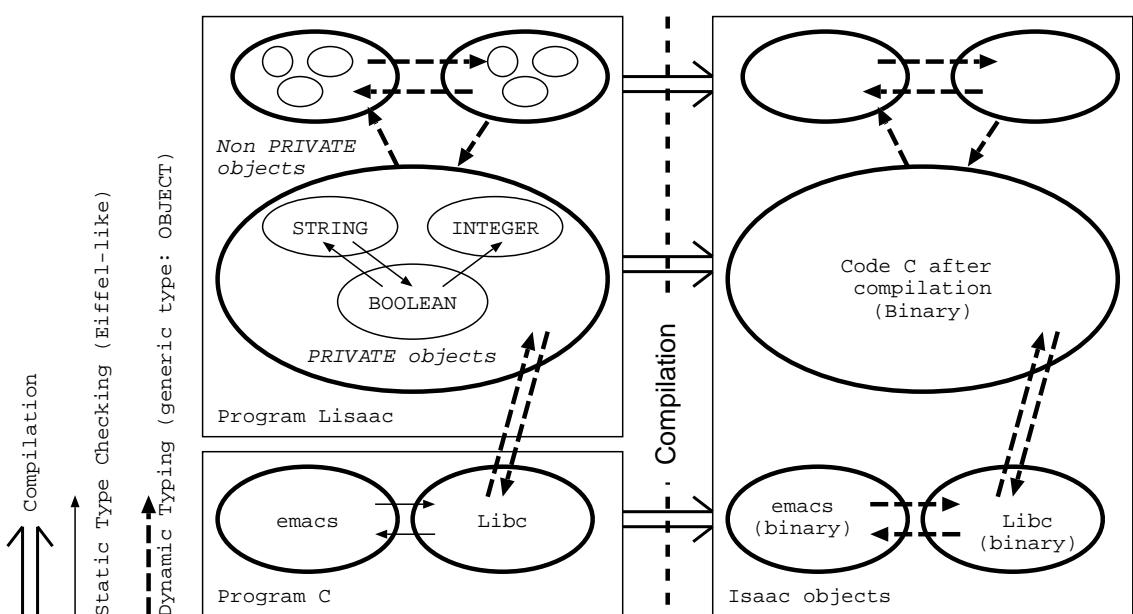
5.4.2 Example

```
+ a:FAST_ARRAY(INTEGER);
+ b:INTEGER;
a := FAST_ARRAY(INTEGER).create 10;
a.put 5 to 0;
a.put 2 to 1;
b := a.item 0;
```
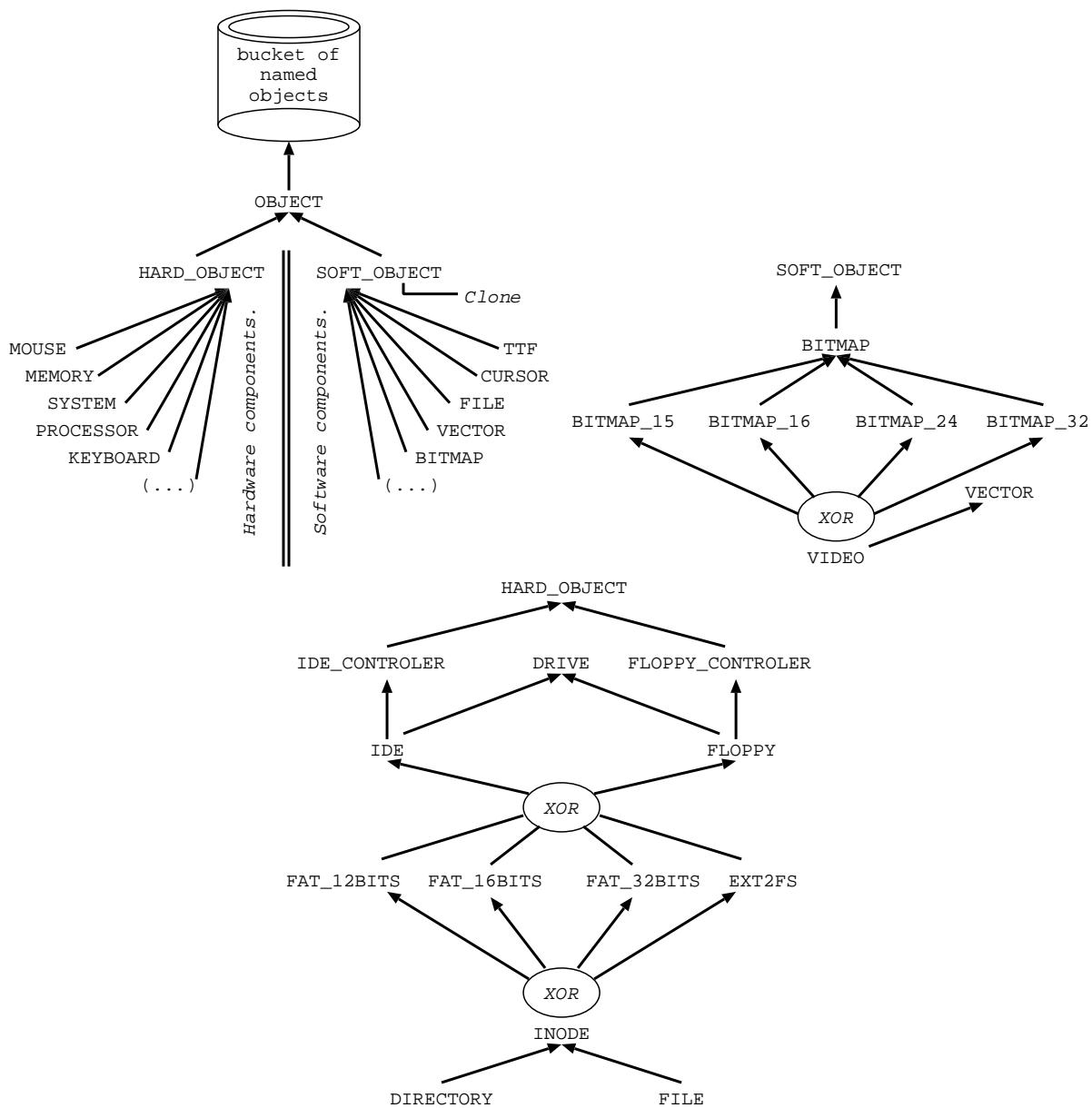
Chapter 0110b
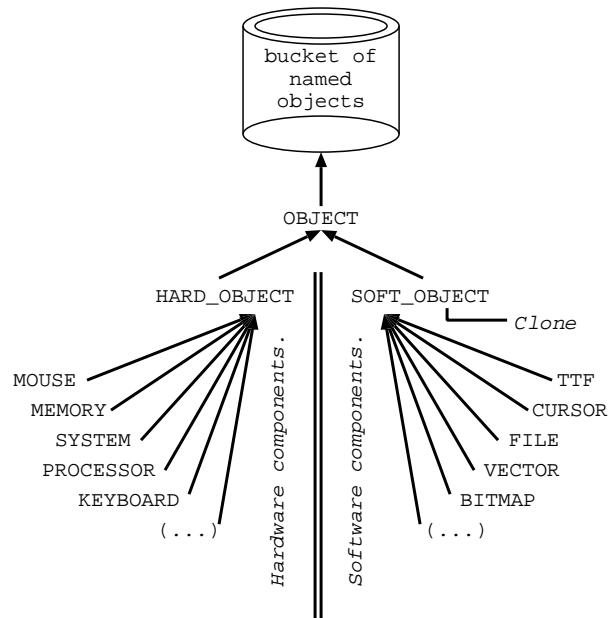
Isaac Operarting System structure

6.1 Compilation limit
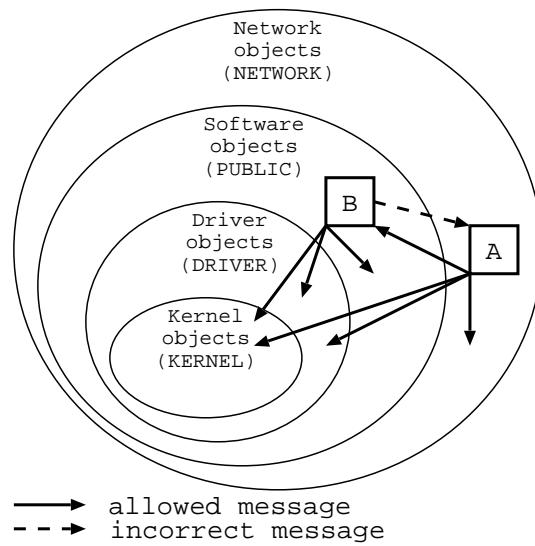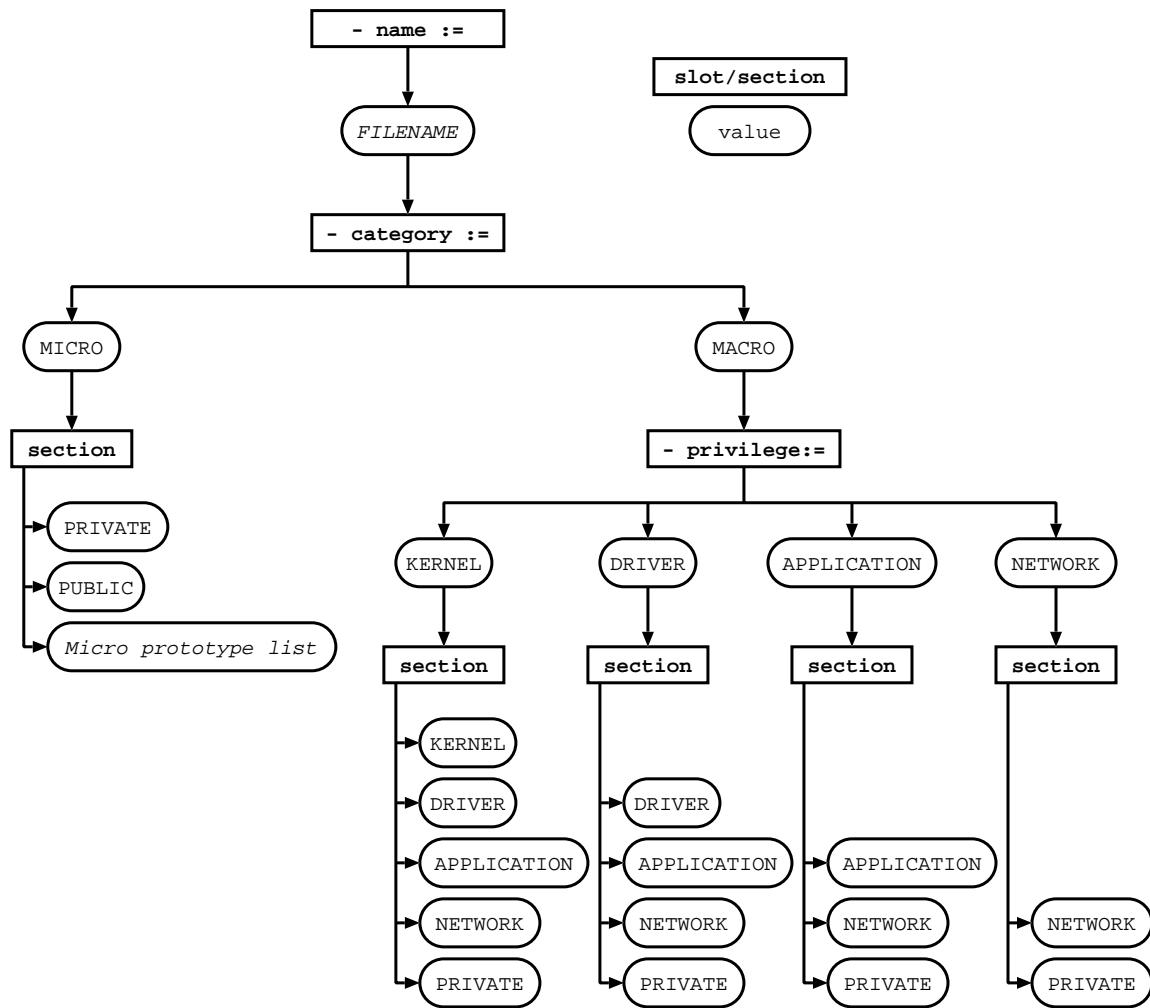
6.2   Hardware design

6.3   Security level

Bibliography

[AB72]   C.T. Clingen & R.C. Daley A. Bensoussan. The Multics Virtual Memory: Concepts
           and Design. In *Communications of ACM*, volume 15, number 5, pages 308–318, May
           1972.

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

[Col96]   D. Colnet. `http://SmallEiffel.loria.fr`. Site web : SmallEiffel The GNU Eiffel
           Compiler., 1996.

[Col00]   D. Colnet. `http://smalleiffel.loria.fr`. Site web: SmallEiffel, the GNU Eiffel
           compiler., 2000.

[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.

[htt]   http://www.mentor.com/embedded/papers/whitepapers/using80x86/.

[htt92a]   . 1992.

[htt92b]   . 1992.

[Hum90]   R. Hummel. Interruption and exception. In *Intel486 Microprocessor Family Program-
           mer's Reference Manual*, pages 83–104, 1990.

[Mey92]   B. Meyer. *Eiffel, The Language*. Prentice Hall, Englewood Cliffs, 1992. ISBN 0-13-
           247925-7.

[Mey94]   Bertrand Meyer. *Eiffel, The Language*. Prentice Hall, 1994.

[Mey97]   Bertrand Meyer. Prentice Hall, 2nd Edition. In *Object-oriented Software construction*,
           pages 311–323, 1997.

[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.

[Rif94]   J-M. Rifflet. Espace d'adressage virtuel. In *La programmations sous unix*, page 410, 1994.

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

[Son00]   B. Sonntag. `http://www.lisaac.org`. Site web: Lisaac., 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.

[Tan94a] A. Tanenbaum. La segmentation. In *Systmes d'exploitation, systmes centraliss, systmes distribus*, pages 144–159, 1994.

[Tan94b] A. Tanenbaum. Partage de mmoire. In *Systmes d'exploitation, systmes centraliss, systmes distribus*, pages 731–739, 1994.

[TcC99]   Prashant Pradhan Tzi-cker Chiueh, Ganesh Venkitachalam. Integrating segmentation and paging protection for safe, efficient and transparent software extensions. In *17th ACM Symposium on Operating Systems Principles*, pages 140–153. ACM Press, 1999.

[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.

Index