



SONNTAG Benoît
BOUTET Jérôme
CHABERT Alexandre
OSWALD Xavier

bsonntag@loria.fr
x.oswald@free.fr

Présentation de Lisaac

22 décembre 2009

Table des matières

1	Introduction	5
1.1	Motivation	5
1.2	Le compilateur Lisaac	6
1.3	Pourquoi utiliser Lisaac	8
1.4	Notations	8
2	Présentation rapide	11
2.1	Lisaac : un langage objet à prototypes	11
2.2	Notations	12
2.3	Les objets	13
2.4	Slots	13
2.4.1	Méthodes et fonctions	14
2.4.2	Variables locales	18
2.5	Compilation et exécution	19
2.6	Comment écrire	19
2.6.1	Types	19
2.6.2	Mon premier programme en Lisaac	20
2.6.3	Comment afficher	21
2.6.4	Comment lire	21
2.6.5	Conditionnelles : <i>if else</i>	21
2.6.6	Une boucle : <i>do_while</i>	22
2.7	Lisaac : un langage orienté objet	23
2.7.1	Cloner	24
2.7.2	Héritage	25

Chapitre 1

Introduction

Lisaac est le premier langage orienté objet à prototypes disposant d'outils de programmation système réellement compilé. Deux langages en sont à l'origine : Le langage Self pour sa flexibilité ainsi que le concept d'héritage dynamique et le langage Eiffel pour son typage statique et sa sécurité avancée grâce à la programmation par contrat. Le compilateur produisant actuellement du code C ANSI lui-même compilable par tout compilateur adéquat (GCC...), Lisaac apparaît donc intrinsèquement comme massivement multi-plateforme. De plus, les résultats observés sur les objets compilés montrent qu'il est tout à fait possible d'obtenir des exécutables combinant la vitesse du C au haut niveau des derniers langages objets.

1.1 Motivation

La conception ainsi que l'implémentation du système d'exploitation Isaac nous ont conduit à la création d'un nouveau langage de programmation nommé Lisaac.

Celui-ci intègre nativement des mécanismes de protection des communications, des interruptions système ainsi que le mappage mémoire pour les pilotes. Tout ceci associé à la mise en oeuvre des prototypes et de l'héritage dynamique font que ce langage s'accorde parfaitement à l'architecture du système d'exploitation.

Le but de ce projet est de s'affranchir de la rigidité interne des systèmes actuels, ceux-ci se trouvant à notre avis trop dépendant des langages de bas niveau utilisés pour leur écriture. De ce fait, le système d'exploitation Isaac a été lui même complètement écrit dans un langage de haut niveau.

L'évolution des langages de programmation actuels convient aux besoins en terme de processing de données ainsi qu'aux contraintes de conception et production de logiciels. Aucuns de ceux orientés objet ne sont pourtant parvenus à apporter de réelle alternative à leurs homologues procéduraux (C en tête) concernant le développement de systèmes d'exploitation modernes.

Jusqu'à aujourd'hui, durant la création de système d'exploitation, les contraintes relatives à la programmation matérielle se voient systématiquement adressées à l'aide d'un langage de programmation bas niveau, tel que le C. Ce choix conduit généralement à un manque de flexibilité se faisant ressentir à la couche applicative.

Ces observations nous ont conduit à concevoir et implémenter un nouveau langage orienté objet doté de fonctionnalités additionnelles, adaptées à l'écriture de systèmes d'exploitations.

Afin de réaliser ce but, nous avons commencé par rechercher des langages objets existants intégrant de puissantes caractéristiques de flexibilité ainsi que d'expressivité.

Ce langage provient aussi d'expérimentations concernant la création d'un système d'exploitation basé sur les objets dynamiques et dont les possibilités se trouvent être un mélange subtil de Self et Eiffel ainsi que du C pour certaines de ses capacités bas niveau.

Le langage Lisaac est le premier langage orienté objet à prototype compilé réellement utilisable. Les objets compilés restent des objets à part entière, avec leur capacités et leur expressivité totalement préservées. Les outils systèmes sont inclus nativement, tels le mappage et les interruptions.

1.2 Le compilateur Lisaac

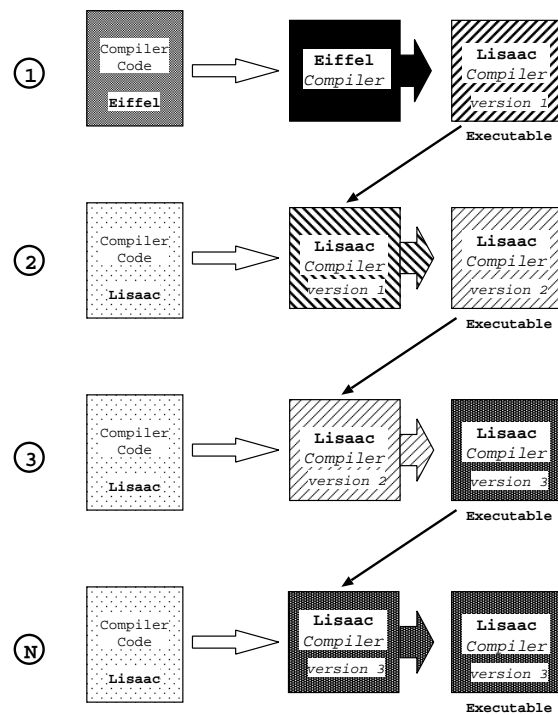
Le compilateur Lisaac est à ce jour entièrement écrit dans le langage du même nom et produit du code C ANSI optimisé compilable par tout compilateur adéquat (GCC...), sur toute architecture supportée par ce dernier. L'opération de bootstrap a été finalisée en Janvier 2004, comme expliqué ci-après.

Etape 1 : Le compilateur Lisaac est écrit dans un autre langage (ici Eiffel, qui fut choisi de par sa grande proximité avec Lisaac) avant d'être compilé. L'exécutable obtenu est la première version du compilateur Lisaac.

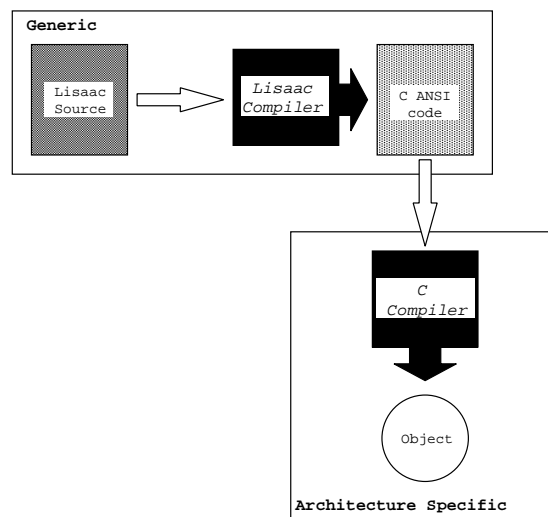
Etape 2 : Le code source du compilateur V1 est entièrement traduit en Lisaac et compilé à l'aide de l'exécutable V1. Le compilateur version 2 est produit. Si aucune erreur n'apparaît, les versions V1 et V2 opèrent de manière identique, la seule différence entre les deux étant la dépendance de la V1 envers le langage et le compilateur Eiffel.

Etape 3 : Le code source du compilateur Lisaac V2 est compilé par l'exécutable de même version. Le compilateur V3 est alors obtenu.

Toute itération ultérieure ne changeant pas l'exécutable produit, nous sommes à présent en état stable et donc totalement indépendant de tout autre langage. En effet, le compilateur étant écrit en Lisaac et compilé par lui-même, il est donc constitué exclusivement de technologie Lisaac.



Le compilateur s'exécute alors sur toute architecture disposant d'un compilateur C.



1.3 Pourquoi utiliser Lisaac

Lisaac a été premièrement développé afin d'implémenter le système d'exploitation Isaac mais est devenu un langage objet indépendant pouvant être utilisé pour écrire tout programme.

Celui-ci bénéficie de nombreux avantages : c'est un puissant langage de haut niveau, basé sur le concept de prototypes. La sécurité a été depuis le commencement un réel aspect de Lisaac, regroupant le typage statique, les assertions comme *Requires*, *Ensures* et *Invariant*, ainsi que de nombreuses vérifications lors de la compilation.

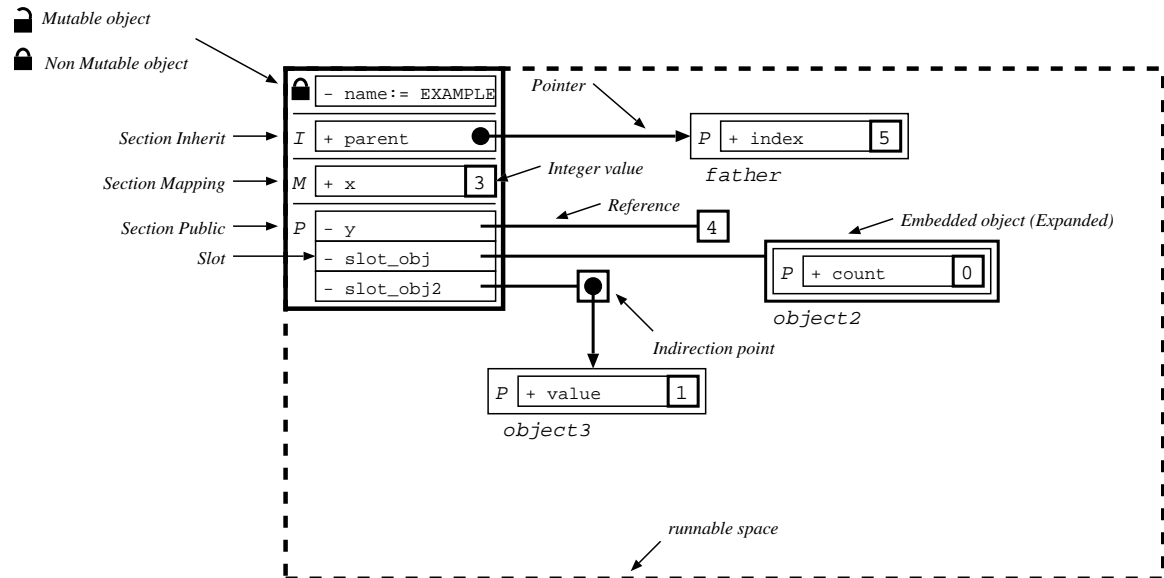
Le grand nombre d'optimisations de haut niveau fournissent quand à eux une efficacité ainsi qu'une vitesse élevée au code compilé.

Une librairie fournie, entièrement écrite en Lisaac, offre au programmeur un large panel de prototypes et fonctions intégrés, comme :

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

1.4 Notations

Vous trouverez dans cette section la représentation des objets en mémoire, comme indiqué dans la figure suivante.



Chapitre 2

Présentation rapide

Après la lecture de ce chapitre, vous serez en mesure d'écrire des programmes simple en Lisaac.

2.1 Lisaac : un langage objet à prototypes

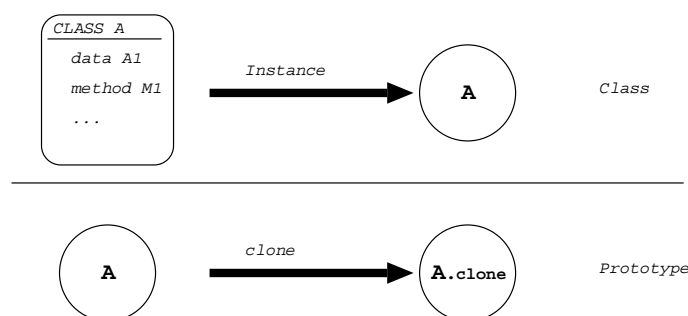
Lisaac est un langage orienté objet basé sur le concept des prototypes. Les langages à classe et prototypes diffèrent sur peu de points, bien qu'essentiels.

Dans un langage à classe, un objet doit être instancié depuis sa description pour être vivant.

Pour les langages à prototypes, la description de l'objet est déjà vivante. Dans Lisaac, l'objet maître ("master") est directement utilisable sans instanciation préalable.

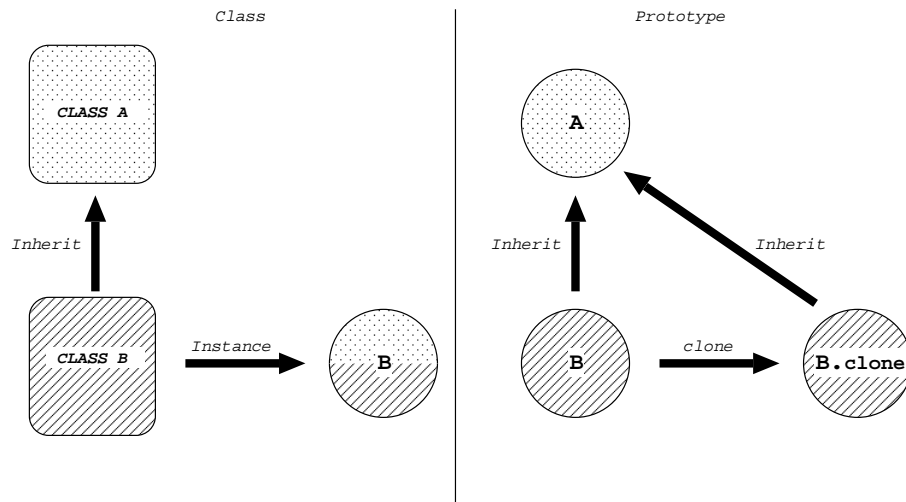
Cet objet particulier est écrit en lettres majuscules et peut être manipulé comme tout autre objet.

De nouveaux objets sont obtenus en clonant l'objet maître. La routine **clone** n'est pas une fonction codée en dur mais provenant d'une librairie.



Conscient de cette propriété, il est évident que l'héritage est particulier. Les objets héritent d'homologues vivant, déjà en service. Cela permet de nombreuses

variations en regard du type (+ or -), comme partager des parents entre deux objets ou l'héritage dynamique (par changement de la référence du parent), ce que nous verrons plus tard.



Les objets sont les entités fondamentales dans Lisaac. Toute entité est représentée par un ou plusieurs objets. Les contrôles eux même sont fournis par des objets : les blocs. Ceux-ci sont les emplacements utilisés pour contenir des structures de contrôle définies par l'utilisateur.

Un objet est composé d'un groupe de slots. Un slot est une paire nom-valeur. Tout slot peut contenir une ou plusieurs références à d'autres objets. Quand un slot est trouvé durant une recherche de message (voir section 2.4 page 13), l'objet contenu dans celui-ci est évalué.

2.2 Notations

Lisaac est sensible à la casse et respecte les contraintes suivantes :

- Les variables ainsi que les slots sont écrits en minuscules (*x*, *counter*, ...).
- Les types d'objets (ou noms principaux d'objets/prototype) sont en majuscules (INTEGER, BOOLEAN, ...).
- Les mots-clé commencent quand à eux par une majuscule suivie de minuscules (**Section**, **Header**, ...).

Le symbole `:=` est une affectation. Faites attention à ne pas utiliser le symbole `=` qui compare deux objets et renvoie un booléen. Vous verrez les symboles `+` ou `-` devant les slots. Ils définissent le type de slot et sont obligatoires! Leur rôle sera expliqué dans les prochaines pages.

Une séquence se termine par `;`. En son absence, le compilateur continue sur la ligne suivante.

Une liste de séquences peut être définie entre (et).

Les commentaires commençant par // s'arrêtent en fin de ligne.
Ceux se trouvant entre /* et */ peuvent prendre plusieurs lignes.

2.3 Les objets

En Lisaac, les objets sont les entités fondamentales. Tout est représenté par un ou plusieurs objets, d'un simple entier ou booléen à des formes plus complexes comme des tableaux et fenêtres.

Un objet est contenu dans un et un seul fichier, nommé d'après le nom de l'objet suivi de l'extension **.li**.

Par exemple, INTEGER.LI, BOOLEAN.LI, WINDOW.LI, ...

Le code source d'un objet est divisé en sections.

Section Header est obligatoire. Dans celle-ci sont définis le nom, le type et la catégorie de l'objet. Les définitions des autres catégories sont expliquées ultérieurement. Ensuite vient **Section Public**, dans laquelle le slot s'exécutant à l'appel de l'objet est écrite (voir la section suivante).

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

Example : fichier HELLO_WORLD.LI

Section Header

```
+ name := HELLO_WORLD; // Le nom est en majuscules
```

Section Public

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



Il n'y a pas de ; après **Section xxxx**.

2.4 Slots

Un objet est composé de slots, qui sont des services rendus par l'objet.

Un slot peut contenir des données ainsi que du code (fonction ou méthode).

Il est défini par un nom et peut posséder un type statique pour les données et fonctions.

Un slot est préfixé par le signe + ou -, qui donne le type du slot (pour faire simple, la valeur est partagée entre les objets avec - et locale à l'objet avec +).

Le type du slot est défini après le signe :.

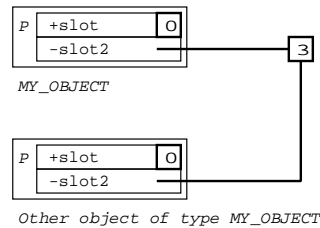
Section Header

```
+ name := MY_OBJECT; // Name en majuscules
```

Section Public

```
+ slot :INTEGER; // Valeur locale de type INTEGER
```

```
- slot2 :INTEGER := 3;           // Valeur partagée de valeur '3'
```



Tout ceci sera expliqué bientôt.

2.4.1 Méthodes et fonctions

Slots simples

Comme défini précédemment, les méthodes (ou routines) sont une notion fondamentale dans les langages orientés objet, accompagnées du concept compagnon de connexion retardée (ou envoi de message, appel de routine, appel de méthode, dispatche dynamique, ...).

Deux types de slots existent. Le premier est exécuté au chargement de l'objet. Il est défini comme valeur par défaut d'une variable et est représenté par le signe `:=`.

```
+ slot :INTEGER := 3 + 4;        // '3 + 4' est évaluée à l'initialisation
+ slot2:INTEGER := slot * 2;
```

Le second n'est exécuté qu'à l'appel du slot correspondant. Il est défini par le symbole `<-`.

```
+ slot :INTEGER := 3;
+ slot2:INTEGER <- ( 5 + slot; ) ; // '5 + slot' n'est évaluée
//qu'à l'appel du slot slot2
```

Du code plus complexe peut être défini entre parenthèses.

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


Notez qu'il est possible de tout écrire sur la même ligne, bien qu'il soit évidemment plus aisé d'aligner le code comme montré précédemment, ne serait-ce que pour la compréhension. La valeur de retour d'un code est la valeur du dernier code (sans `;`) avant `)`.

```
+ slot :INTEGER := 3;
+ slot2 :INTEGER <-
(
    slot                                // Valeur de retour, évaluée avec l'appel du code 5
);
+ slot3:INTEGER :=
(
    slot := slot + 6;
    slot := slot * 5;
    slot                                // Valeur de retour, évaluée durant le chargement de l'objet
);
```



Soyez prudent, le type de retour d'un slot doit être le même que celui de la valeur retournée.

```
+ slot :INTEGER := 3;
+ slot2 :INTEGER := 4;
+ slot3 :BOOLEAN <-
(
    slot = slot2                        // Renvoi TRUE si égal, FALSE sinon
);
+ slot4:INTEGER <-
(
    slot3                                // Erreur : les types sont différents
);
```

Appel d'un slot

L'appel d'un slot dépend de quand cela arrive.

File object1.li

Section Header

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

Section Public

```
+ slot :INTEGER := 3;
+ slot2 :INTEGER <-
(
    slot * 2 + 4                        // Appel de 'slot' depuis le même objet
);
```

Un objet étant initialisé avec la valeur NULL, un appel sur celui-ci provoquera une erreur. Dans un langage à prototypes (voir 2.1 page 11), l'objet 'maître' (écrit en lettres majuscules) est vivant sans instanciation préalable. Les autres objets sont créés par clonage de l'objet 'maître', en utilisant le slot **clone** (pour plus d'informations voir 2.7 page 23.)

File main_object.li

Section Header

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

Section Public

```
+ slot_object:OBJECT1;
+ slot_object2 :INTEGER <-
(
  slot_object := OBJECT1.clone;    // clone de l'objet OBJECT1
  slot_object.slot2 + 5            // Appel de 'slot2' sur 'slot_object' de type OBJECT1
);
```

Le symbole . définit l'appel d'un slot appartenant à un autre objet.

Slots avec arguments

Un slot peut aussi être appelé avec des paramètres.

```
+ slot a:INTEGER :INTEGER <-      // 1 paramètre. Aucune parenthèse n'est nécessaire
(
  a * 2
);

+ slot2 (a,b:INTEGER) :INTEGER <- // 2 paramètres du même type
(
  a + b
);

+ slot3 (a:INTEGER,b:CHARACTER) :INTEGER <- // 2 paramètres de type différent
(
  b.print;
  a * 3
);

+ slot4 :INTEGER <-
(
  slot 3 + slot2 (2,3) + slot3 (4,'y')      // Appel de slots
);
```

Vous pouvez définir vos propres mots-clé pour séparer des paramètres.

```

+ slot a:INTEGER value b:INTEGER :INTEGER <- // value est mon propre mot-clé
(
  a + b * 2
);


+ slot2 (a,b:INTEGER) write c:CHARACTER :INTEGER <-
(
  c.print;
  a * b
);

+ slot3 a:INTEGER multiply b:INTEGER add c:INTEGER:INTEGER <-
(
  a * (b + c)
);

+ slot4 :INTEGER <-
(
  slot 3 + (slot2 (2,3) write 'c') + (slot3 4 multiply 5 add 6) // Appel de slot
);

```

Assignement des slots

 Attention, vous ne pouvez pas assigner de valeur à un slot extérieur à l'objet, comme montré dans l'exemple ci-dessous.

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; // Le compilateur va stopper
);

```

Cette règle a été posée afin de protéger les slots des objets. Quand vous définissez un objet, vous devez spécifier les slots pouvant changer en créant des méthodes dédiées. Vous pouvez trouver ceci fastidieux mais cela vous assurera

un contrôle total sur toutes les opérations effectuées sur votre objet. Imaginez simplement un slot compteur pouvant être modifié par n'importe qui travaillant avec votre objet ... Vous pouvez aussi créer des conditions dans votre objet pour le protéger d'avantage.

Example : utilisation d'un 'setter'

Section Header

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

Section Public

```
+ value:INTEGER := 3;
- set_value v:INTEGER <- // Définissez votre propre 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 Variables locales

Vous pouvez définir des variables locales à l'intérieur de votre slot. La syntaxe est la même que pour un slot. Une variable locale sera souvent non-partagée (+) et est initialisée avec la valeur par défaut de son 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
);
```



Notez que vous devez définir toutes les variables dans les premières lignes de votre slot, sans code à l'intérieur de cette liste de définitions.

```
+ slot a:INTEGER :INTEGER <-
( + var1:INTEGER;
  + var2:INTEGER;
  + result:INTEGER;
  var1 := a * 2;
  var2 := a + 4;
  + var3:INTEGER;                // Le compilateur stoppera avec une erreur
  var3 := a - 5;
  result := var1 + var2 - var3;
  result
);
```

2.5 Compilation et exécution

Pour compiler vos programmes Lisaac, vous aurez simplement à taper :

```
lisaac my_object.li
```

Cette commande produit 2 fichiers : `my_object.c` et `my_object`, qui est un exécutable. Par défaut Lisaac utilise GCC pour la compilation du code C généré.

Running an object

Dans votre objet principal, **Section Public** doit contenir un slot nommé 'main'. Il sera exécuté à l'initialisation de votre fichier compilé.

Section Header

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

Section Public

```
+ value:INTEGER := 3;
- main <-
(
  value.print;
);
```

Pour plus d'informations sur les slots et appels de méthodes, voir section 2.4.1, page 14 et 15.

2.6 Comment écrire

2.6.1 Types

Il n'y a pas de type intégré dans Lisaac. Chaque type provient de la librairie (vous pouvez consulter le code source pour vous faire une idée de l'implémentation). Les types de bases utilisables sont :

- INTEGER avec les opérations arithmétiques et beaucoup d'autres (implémentés dans l'objet NUMERIC, parent de tous les types nombres)

Notations : *12*, *12d* : valeur décimale

1BAh, *0FFh* : valeur hexadécimale

01010b, *10b* : valeur binaire

14o, *6o* : valeur octale

10KB, *10MB*, *10GB* : facilités système

- BOOLEAN : vous avez 2 'valeurs' pour BOOLEAN : TRUE ou FALSE. Chacunes de ces valeurs sont aussi des objets.

- CHARACTER : un simple caractère

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

\n, *\t*, *\r* : caractère d'échappement

\10, *\0Ah* : code caractère

- STRING_CONSTANT : composé de multiples caractères; ne peut être modifié; défini entre " "

Notations : *"Hello World\n"* : simple chaîne

- STRING : chaîne pour construire avec des fonctions de la librairie

- FAST_ARRAY : un tableau à limite basse fixée compatible avec de nombreuses opérations

- BLOCK : un bloc de code, défini entre { et }

Référez vous au chapitre sur la librairie pour plus de détails.

2.6.2 Mon premier programme en Lisaac

Voici le classique programme "Hello World", dont le résultat est l'écriture de cette même phrase sur la sortie standard :

Edit File HELLO_WORLD.LI

Section Header

```
+ name := HELLO_WORLD;
```

Section Public

```
- main <-
(
  "Hello world !".print;
);
```

Compilez avec : `lisaac hello_world` ou `lisaac hello_world.li` Cela produit un fichier exécutable nommé `hello_world`.

Dans ce premier programme en Lisaac, **main** est la racine du système, ou début de l'exécution du programme (programme principal).

Seule cette instruction est évaluée (i.e. exécutée) immédiatement au départ du programme.

Tout est objet en Lisaac comme vous pouvez le voir dans cet exemple : le slot **print** est appelé sur l'objet String "Hello world!".

2.6.3 Comment afficher

Comme vu précédemment, **print** est une méthode provenant du prototype `STRING` de la librairie. Il existe aussi la même méthode pour le type `NUMERIC`.

```
"Hello World !".print;
3.print;
my_string.print;      // objet de type STRING (créé précédemment biensûr)
```

2.6.4 Comment lire

Maintenant, lisons depuis l'entrée standard :

Section Header

```
- name := HOW_TO_READ;
```

Section Public

```
- main :=          // un main sur plusieurs lignes
(
  "Enter your name : ".print;
  IO.read_string;
  ("Welcome, " + IO.last_string).print;
);
```

last_string renvoie une référence de la dernière chaîne (string) entrée depuis la sortie standard.

Notez l'utilisation du prototype initial prédéfini `IO` pour les entrées-sorties.

2.6.5 Conditionnelles : *if else*

Une structure de contrôle basique dans nombre de langages est le couple **if** - **then** - **else**. En Lisaac, le **then** est omis.¹

Comme vu précédemment, tout est objet dans le langage. Cette méthode conditionnelle suit donc la même construction : `condition.if block_true else block_false`

`condition` est un objet `BOOLEAN` (true ou false) sur lequel on appelle la méthode **if** suivie de 2 paramètres : `block_true` et `block_false` (objets de type `BLOCK`), séparés par le mot-clé **else**.

Section Header

```
- name := IF_ELSE;
```

Section Public

```
- main <-
( + gender:CHARACTER;    // variable locale
```

¹**if else** ne sont pas intégrés à Lisaac mais sont implémentés dans la librairie comme simples appels de méthodes.

```

IO.put_string "Entrez votre genre (M/F) : ";
IO.read_character;
gender := IO.last_character;

(gender == 'M').if {      // conditional
  IO.put_string "Bonjour monsieur !"; // partie then
} else {
  IO.put_string "Bonjour madame !"; // partie else
};
);

```

Vous pouvez utiliser indifféremment `"ma_chaine".print` ou `IO.put_string "ma_chaine"`. Cela aura le même effet.

Notez l'usage de la variable locale **gender** afin de contenir la réponse de l'utilisateur.

La conditionnelle consiste en une expression booléenne (`gender == 'M'`) vers laquelle le message **if else** est envoyé.

Une liste d'instructions et une expression sont de même construction syntaxique, entre parenthèses.

`{ /* ... */ }` définit une liste d'instructions comme une liste classique mais son type est `BLOCK` et son évaluation est retardée

2.6.6 Une boucle : *do_while*

Voici une boucle conditionnelle en Lisaac :

Section Header

```
- name := DO_WHILE;
```

Section Public

```

- main <-
  ( + gender:CHARACTER;

    IO.put_string "Entrez votre genre (M/F) : ";

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

    (gender == 'M').if {
      IO.put_string "Bonjour Monsieur !";
    } else {
      IO.put_string "Bonjour Madame !";
    };
  );

```


Le bloc d'entrée est exécuté au moins une fois, et continue aussi longtemps que la condition de boucle reste vraie.

2.7 Lisaac : un langage orienté objet

Lisaac est un langage orienté objet. Vous pouvez construire une application en utilisant plusieurs objets, ceci étant déjà le cas quand vous appelez des méthodes d'objets de la librairie. Le compilateur lie automatiquement tous les objets nécessaires à votre objet principal (voir le chapitre du compilateur pour plus d'informations).

Quand vous exécutez un programme, seuls les objets "maîtres" (écrits en majuscules) sont vivants. Les autres sont initialisés avec NULL et vous ne pouvez les utiliser (il y aura un arrêt du compilateur).

Section Header

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

Section Public

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

Section Header

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

Section Public

```
- main <-
( + my_object:OBJECT1;
  OBJECT1.slot;          // Aucun problème, vous utilisez l'objet "maître"
  my_object.slot;        // Le compilateur va s'arrêter avec l'erreur 'CALL ON NULL'
);
```

Si vous voulez vous servir d'un autre objet, vous devez utiliser l'opération clone de l'objet "maître" (voir 2.7.1).

The 'Self' object

Nous appellons *self* l'objet vivant actuel. Quand vous appelez un slot appartenant à plus d'un objet, c'est en fait le slot de l'objet *self* qui est appelé implicitement. Le mot-clé **Self** peut être utilisé pour appeler explicitement l'objet *self* (à l'instar de "this" en Java et C++ ou "Current" en Eiffel).

Section Header

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

Section Public

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

```
- main <-
(
  Self.slot_data.print;    // produit le même effet que slot_data.print;
);
```



Notez que *self* est différent entre objets, même s'ils sont de type identique, du fait que **Self** est un objet.

2.7.1 Cloner

Vous pouvez cloner un objet afin d'en créer un de même type. La méthode **clone** est définie dans le type **OBJECT** de la librairie.

Le nom du slot doit être précédé de "+" pour pouvoir être cloné. Comme vu précédemment, vous devez utiliser **clone** afin de travailler avec un objet.

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;      // Aucun problème ici, my_object n'est pas Null
);
```

Example : représentation mémoire (les slots "set_x" et "set_count" ne sont pas représentés afin de simplifier l'exemple. La représentation réelle est donnée ultérieurement)

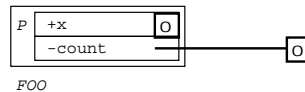
Section Header

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

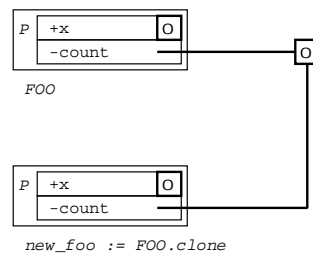
Section Public

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

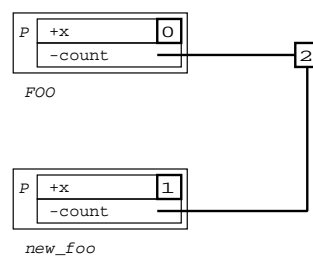
- count:INTEGER;
- set_count v:INTEGER <- ( count := v; );
```



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



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



2.7.2 Héritage

Vous pouvez définir l'héritage pour les objet, avec autant de parents que vous le souhaitez. Un parent est défini dans **Section Inherit** à l'aide d'un slot standard. Un parent est aussi un objet à part entière auquel vous pouvez envoyer des messages. Si un slot appelé sur un objet n'est pas trouvé dans celui-ci, l'algorithme de *lookup* recherche ce slot dans les parents. Pour ce faire, il exécute une recherche ordonnée depuis le premier slot déclaré dans la section d'héritage.

Exemple : Voyons un cas d'héritage avec un parent défini en '-'

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

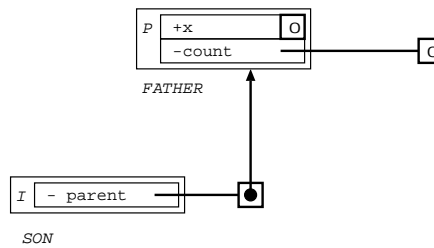
```
+ name      := SON;
```

Section Inherit

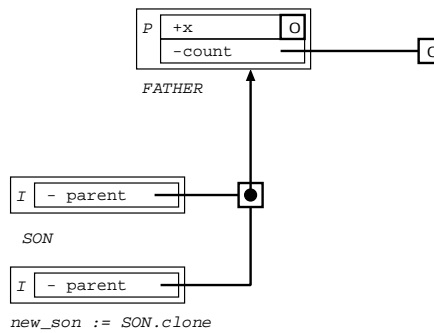
```

- parent:FATHER := FATHER;          // le nom du slot n'a aucune importance
Section Public
- change_parent p:FATHER <- ( parent := p; );

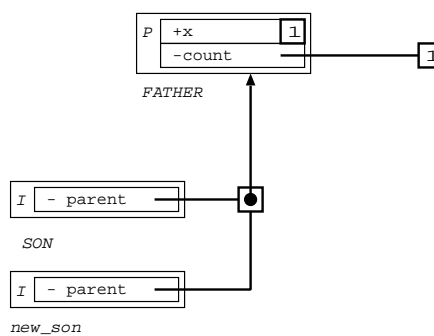
```



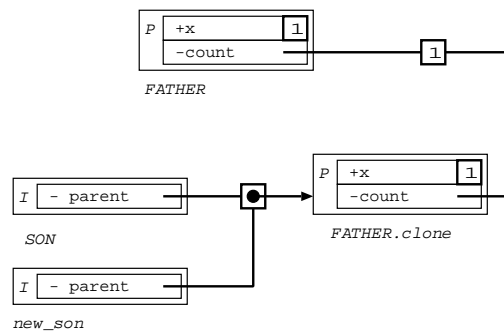
```
new_son := SON.clone;
```



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



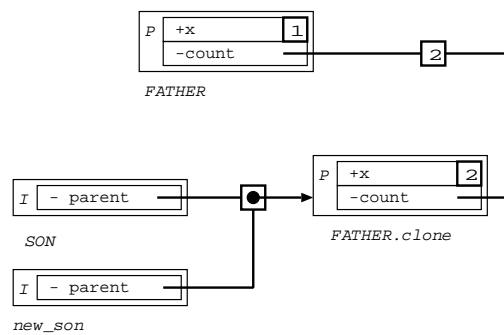
```
new_son.change_parent (FATHER.clone);
```



```

new_son.inc_x;
new_son.inc_count;

```



Exemple 2 : Voyons maintenant la même situation mais avec un parent défini en '+'

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

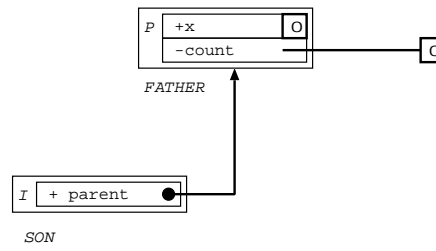
```
+ name      := SON;
```

Section Inherit

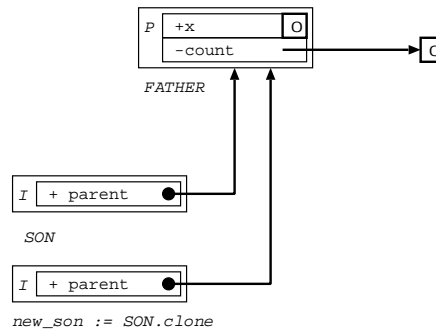
```
+ parent:FATHER := FATHER;
```

Section Public

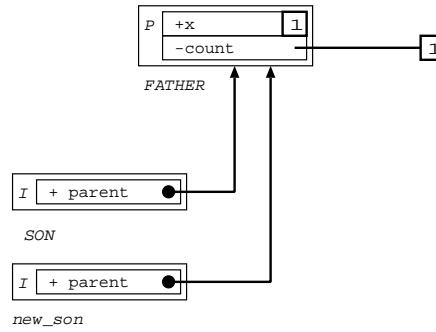
```
- change_parent p:FATHER <- ( parent := p; );
```



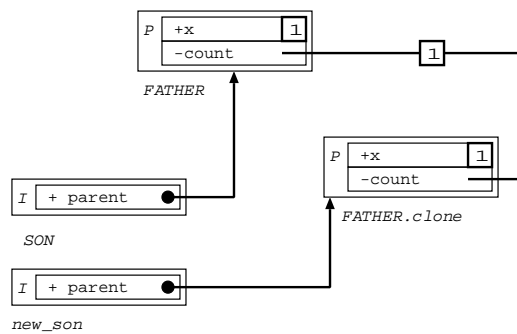
```
new_son := SON.clone;
```



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



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



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

