

# Lisaac

*The power of simplicity at work for you*

Benoît Sonntag – benoit.sonntag@lisaac.org  
and the Lisaac team



<http://www.lisaac.org>

December 22, 2009

# Partnerships

## Core partners



## External partners



# Why a new language ? (1/2)

- C language

## advantages

Memory mapping, interrupt management, ASM glue, multiple kinds of integer, compiled, very good performance

## inconveniences

Not high level language

- SmartEiffel language

## advantages

High level language, genericity, uniformity, static type, programming by contract, compiled, good performance

## inconveniences

Not prototype object oriented, lack of OS programming facility

# Why a new language ? (2/2)

- Self language

## advantages

Uniformity, expressivity, simplicity, prototype object oriented

## inconveniences

Not compiled, lack of protection (no type), lack of OS programming facility

- Java language

## advantages

C-like syntax, static type, internet facility

## inconveniences

Not prototype object oriented, lack of OS programming facility, not good performance, lack of uniformity and expressivity

# History : Lisaac for IsaacOOS Language

In the past...

**C** language



**Unix** system

The futur...

**Lisaac**

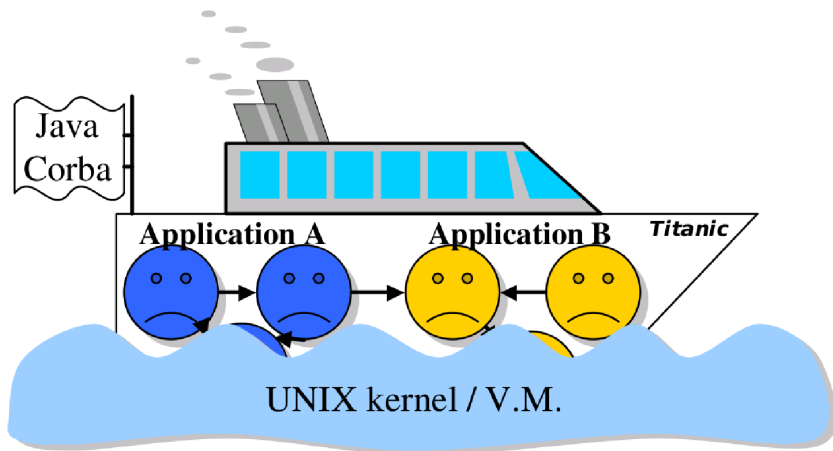
*Prototype based Object  
Oriented Language*



**IsaacOOS**

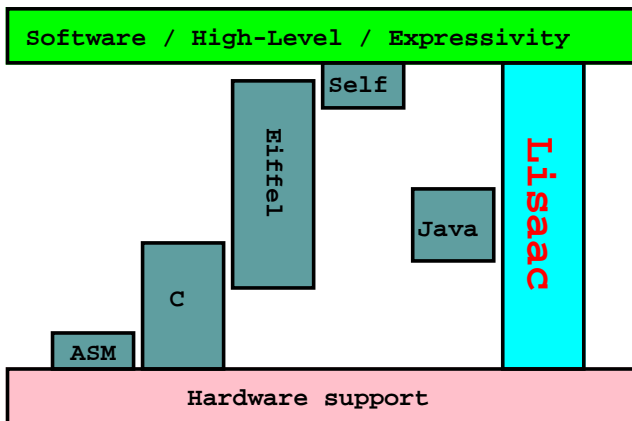
*Prototype Object Operating  
System*

# Let them sink in a bigger box ?



# High-level vs Hardware

## Object Oriented for Hardware

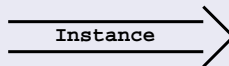


# Class vs Prototype (1/3)

## Class



1 Skeleton  
(=class)

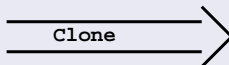


1 Object

## Prototype



1 Object prototype  
(=the One)



1 other Object



# Class vs Prototype (2/3)

## Class



Class A



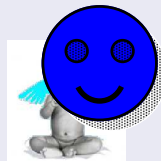
Class B

B Instance

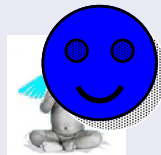


1 Object with  
A and B definition

## Prototype

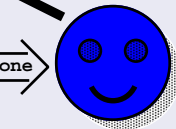


A object  
(Prototype or not)



B object  
(Prototype or not)

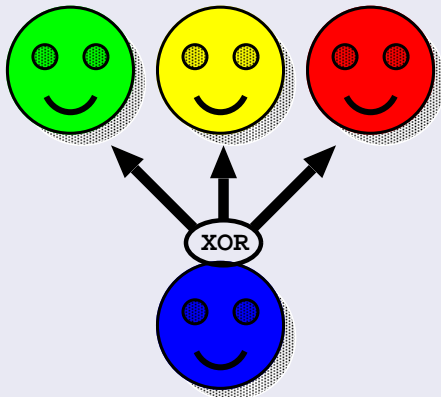
B.Clone




1 other Object

# Class vs Prototype (3/3)


## Dynamic inheritance



# Inherit Lisaac

-  **Self**: Flexibility, simplicity and prototype concept

+

-  **Eiffel**: Static type, programming by contract

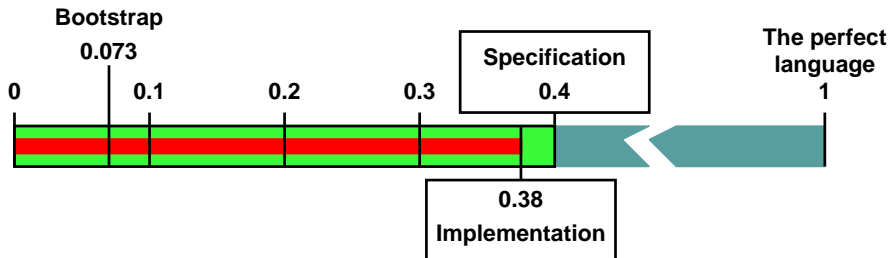
+

- **C**: Interrupt management, memory mapping

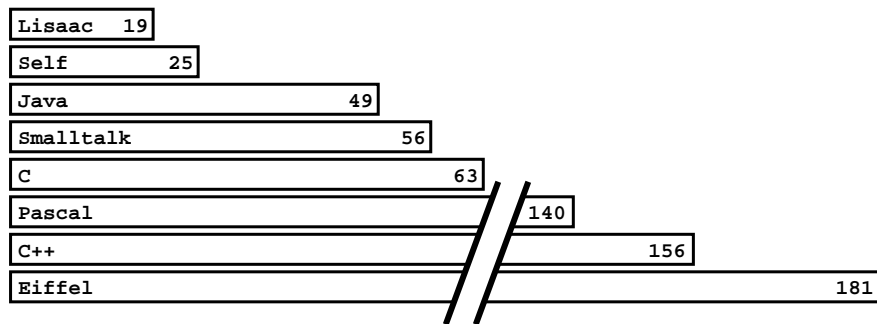


**Lisaac**: Full prototype object for hardware

# Progress...



# The grammar of Lisaac



*Number of gamatical rules*

# Syntax rules

## Identifier

Low case & mono space-name environment

*Example:* `main, factorial`

## Keyword

Upper case for a first character, low case else

*Example:* `Section, Old, Private, Header`

## Type/prototype

Upper case

*Example:* `STRING, CHARACTER, INTEGER`

## Comment

Like C++

*Example:* `/* Comment multiline */` or `// Comment line`

# Base type (1/2)

## INTEGER

- *Hexadecimal*: 0Bh  
0B80\_0000h
- *Decimal*: 12 12d 100\_000
- *Octal*: 14o 777o 7\_333o
- *Binary*: 01b 1101b  
1010\_1111b

## REAL

- *Simple*: 1.1 0.05
- *Scientific*: 5E-2

## CHARACTER

- *Simple*: 'a' 'k'
- *Escape*: '\n' '\t'
- *Code*: '\10\' '\0Ah\'

## STRING\_CONSTANT

- *Simple*: "Hello world\n"
- *Multiline*: "Hello \  
\world\n"

## BLOCK

- *Encapsulate code*: { ... }  
See after...

## Base type: Example (2/2)

### Warning

Even base types are full objects!

#### INTEGER

```
10.factorial.print;
```

#### REAL

```
2.7E-5.print;
```

#### CHARACTER

```
'a'.to_upper.print;
```

#### STRING\_CONSTANT

```
"Hello world\n".print;
```

#### BLOCK

```
{ ... }.value;
```



# Prototype

- One prototype = one file
- The name's prototype = the name's file

*Example :*

The file name `string.li` contains the **STRING** prototype.

- One prototype is a set of **Section**:
  - ① **Section Header** (*Mandatory*)
  - ②  $n \times$  **Section Inherit** or **Section Insert**
  - ③  $n \times$  **Section Public** or other sections...
- One section is a set of **slots** (*datas or functions*).

# Sections

## Inheritance sections after Header section

- **Inherit**: Inheritance definition (*Private*)
- **Insert**: Non-conforming inheritance (*Private*)

## Simple sections

- **Public**: Services with public access
- **Private**: Services with private access
- **Directory**: Services with prototype's directory access
- ***prototype list***: Services with selective access

## Specific sections

- **Mapping**: Mapping structure object
- **Interrupt**: Hardware interruption handler
- **External**: External of Lisaac's slot to C function

# Example: Hello world!

```
hello.li
```

```
Section Header
```

```
+ name := HELLO; // Mandatory
```

```
Section Public
```

```
- main < -
```

```
(
```

```
  'Hello world !\n'.print;
```

```
);
```

*Command line:* lisaac hello.li

*Executable result:* hello (ou hello.exe for windows)

# Slot identifier

```
— qsort tab:COLLECTION from low:INTEGER to high:INTEGER ←  
( + i,j:INTEGER;  
  + x,y:OBJECT;  
  i := low;  
  j := high;  
  x := tab.item ((i + j)>> 1);  
  { ...  
    (i <= j).if {  
      tab.swap j and i;  
      ...  
    };  
  }.do_while {i <= j};  
  (low < j).if { qsort tab from low to j; };  
  (i < high).if { qsort tab from i to high; };  
);
```

# Slot identifier

```
- qsort tab:COLLECTION from low:INTEGER to high:INTEGER ←  
(  
  + i,j:INTEGER;  
  + x,y:OBJECT;  
  i := low;  
  j := high;  
  x := tab.item ((i + j)>> 1);  
  { ...  
    (i <= j).if {  
      tab.swap j and i;  
      ...  
    };  
  }.do_while {i <= j};  
  (low < j).if { qsort tab from low to j; };  
  (i < high).if { qsort tab from i to high; };  
);
```

# Slot identifier: if

```
- qsort tab:COLLECTION from low:INTEGER to high:INTEGER ←  
( + i,j:INTEGER;  
  + x,y:OBJECT;  
  i := low;  
  j := high;  
  x := tab.item ((i + j)>> 1);  
  { ...  
    (i <= j).if {  
      tab.textcolorblueswap j and i;  
      ...  
    };  
  }.do_while {i <= j};  
  (low < j).if { qsort tab from low to j; };  
  (i < high).if { qsort tab from i to high; };  
);
```

# Slot identifier: loop

```
— qsort tab: COLLECTION from low: INTEGER to high: INTEGER ←  
( + i, j: INTEGER;  
  + x, y: OBJECT;  
  i := low;  
  j := high;  
  x := tab.item ((i + j) >> 1);  
  { ...  
    (i <= j).if {  
      tab.swap j and i;  
      ...  
    };  
  }.do_while {i <= j};  
  (low < j).if { qsort tab from low to j; };  
  (i < high).if { qsort tab from i to high; };  
)
```

# Arguments/results definition

## Argument

- Simple: `qsort` `tab:COLLECTION`
- Vector: `put_pixel` `(x,y:INTEGER)`

## Result

- Simple: `is_even:BOOLEAN`
- Vector: `get_coord:(INTEGER,INTEGER)`



# Operator slot: Unary (1/3)

## Prefix

```
- '-' Self:SELF :SELF ←  
zero - Self; // Self ≡ this
```

*Example:* (-3).print;

## Postfix

```
- Self:SELF '!' :SELF ←  
( + result:INTEGER;  
  (Self = 0).if { result := 1; }  
                else { result := (Self - 1) !; };  
  result  
);
```

*Example:* (10 !).print;;

## Operator slot: Binary (2/3)

### Infix associativity **left** priority **80**

```
- Self:SELF '+' Left 80 other:SELF :SELF ←  
Self - - other;
```

*Example:*  $2 + 3 + 4 = ((2 + 3) + 4)$

### Infix associativity **left** priority **90**

```
- Self:SELF '*' Left 90 other:SELF :SELF ← ...
```

*Example:*  $2 + 3 * 4 = (2 + (3 * 4))$

### Infix associativity **right** priority **90**

```
- Self:SELF '^' right 90 other:SELF :SELF ← ...
```

*Example:*  $2 \wedge 3 \wedge 4 = (2 \wedge (3 \wedge 4))$

# Operator slot (3/3)

## Priority

- 1 Classic message

*Example:*  $2 + 5.\text{factorial} \iff 2 + (5.\text{factorial})$

- 2 Postfix message

*Example:*  $- 5 ! \iff - (5 !)$

- 3 Prefix message

*Example:*  $2 + - 5 \iff 2 + (- 5)$

- 4 Infix message Depending priority

*Example:*  $2 + 3 * 5 \iff 2 + (3 * 5)$

## Character list for operator (*It's free style!*)

!	@	#	\$	%	^	&	<			(<-	!= impossible)
*	+	-	=	~	/	?	>	\			

# Style slot (1/3)

**+**: No shared, clonable or call sensitif

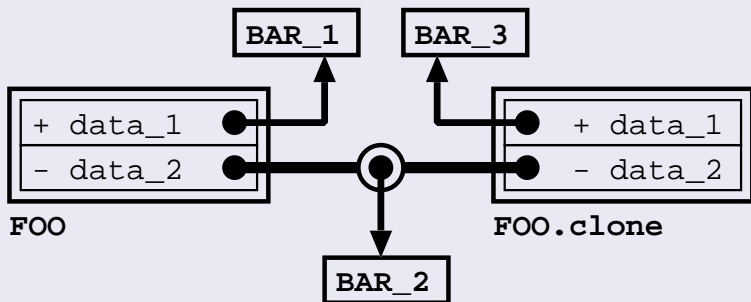
- Distinct for classic data slot
- Distinct for classic local slot (Local variable)

**-**: Shared (= **static** in java), persistant value

- For method slot
- For static data slot or local

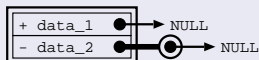
## Style slot (2/3)

No shared vs shared



# Style slot (3/3)

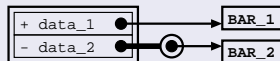
## Step #0



FOO

## Step #1

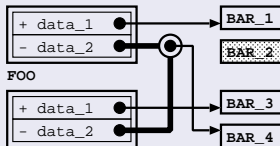
```
data_1 := BAR_1;
data_2 := BAR_2;
```



FOO

## Step #3

```
foo2.set_data_1 BAR_3;
foo2.set_data_2 BAR_4;
```

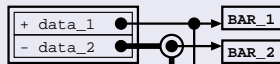


FOO

FOO.clone

## Step #2

```
foo2 := FOO.clone;
```



FOO

FOO.clone

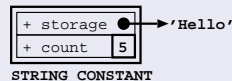
# Expanded attribute = Embedded object (1/2)

## Default attribute (in header declaration)

### Section Header

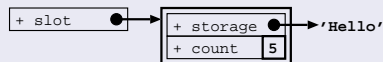
+ name := Expanded INTEGER;

Examples: All tiny objects like CHARACTER, REALs, INTEGERs

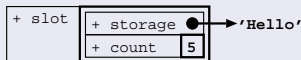


## Attribute type declaration

+ slot:STRING\_CONSTANT;



+ slot:Expanded STRING\_CONSTANT;



Expanded slot is never NULL!

# Expanded attribute & inheritance (2/2)

## Definition

Distinct & Expanded inheritance slot



Class inheritance system (= Java like)

### Section Header

```
+ name := DOG;
```

### Section Inherit

```
+ parent_animal:Expanded ANIMAL;
```

## Note

All other inheritance slot combinations  $\implies$  Prototype system only



# Strict attribute

Strict: Static type  $\implies$  dynamic type

```
+ data:Strict FRUIT;
```

```
...
```

```
data := APPLE.clone; // IMPOSSIBLE!!!
```

```
data := FRUIT.clone; // OK!
```

## Note

Expanded attribute  $\implies$  Strict attribut

# SELF type

SELF : Dynamic type  $\implies$  static type

In **FRUIT** :

```
— clone:SELF  $\leftarrow$  ...
```

With **APPLE** and **ORANGE** inherit **FRUIT** :

```
apple := APPLE.clone; // return Strict APPLE  
orange := ORANGE.clone; // return Strict ORANGE
```

## Note

- Self type  $\implies$  Strict attribute
- **Data slot** or **shared local** variable with SELF type is impossible!

# Genericity type

## Declaration in header

### Section Header

```
+ name := ARRAY(E);
```

## Note

E is parameter type. Syntax: [A..Z][0..9]\*

## Example

```
+ bucket:ARRAY(FRUIT);  
bucket := ARRAY(FRUIT).create 2;  
bucket.put ORANGE to 1;  
bucket.put APPLE to 2;
```

# Parameters' types used in the method (without genericity)

## Example

```
- max a:E and b:E :E ←  
( + result:E;  
  (a > b).if {  
    result := a;  
  } else {  
    result := b;  
  };  
  result  
);
```

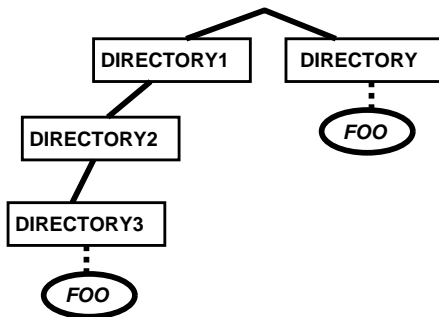
## Note

All parameter type must be defined in arguments function.

# Same prototype name

## Example

```
DIRECTORY.FOO.message;  
DIRECTORY1.DIRECTORY2.DIRECTORY3.FOO  
DIRECTORY1...FOO
```



# Assignment: data (1/3)

## Example

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

## Note

- Assignment is statically ok, if the static type is an identical or a sub-type.
- Simple data assignment ':= ' is the '=' in Java, C++, ...
- Warning with **Strict attribute** type (*see before ...*)

## Assignment : data, if possible (2/3)

### Example

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

### Note

- Assignment is dynamically ok, if the dynamic type is identical or sub-type.
- This mechanism replaces the cast of Java

# Assignment: code (3/3)

## Example

```
— color (r,g,b:INTEGER) < —  
(  
  true_color:=r<<16|g<<8|b;  
);  
...  
(  
  color < — (  
    gray_color := (r+g+b)/3;  
  );  
);
```





# Inheritance: Class like (1/6)

+ & Expanded = Class system

Section Inherit

+ parent\_animal:Expanded ANIMAL;

+ parent_animal	ANIMAL
-----------------	--------

PROTOTYPE

+ parent_animal	ANIMAL
-----------------	--------

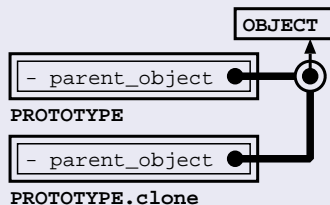
PROTOTYPE.clone

# Inheritance: Prototype “trait” (*Self like*) (2/6)

— = Full shared

## Section Inherit

— `parent_object:OBJECT := OBJECT;`



# Inheritance: No shared & dynamic (*Lisaac inside*) (3/6)

+ = Full dynamic

Section Inherit

+ parent\_object:OBJECT := OBJECT;

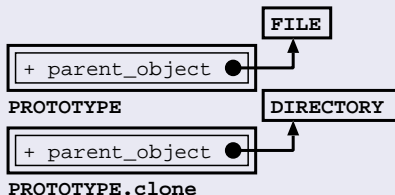
Section Public

...

parent\_object := FILE;

...

parent\_object := DIRECTORY;

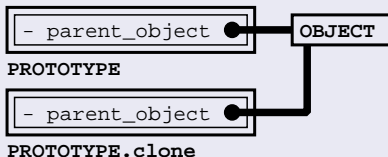


# Inheritance: Shared & Embedded (*Lisaac inside*) (4/6)

– & Expanded (*uniformity form*)

## Section Inherit

– `parent_object:Expanded OBJECT;`



# Inheritance: Dynamic compute parent (*Lisaac inside*)

## (5/6)

For each lookup

Section Inherit

```
+ parent:OBJECT ←  
( + result:OBJECT;  
  ...// compute my parent  
  result  
);
```

Warning

Endless Recursivity caused by the lookup algorithm.

# Inheritance: Dynamic once compute parent (*Lisaac inside*) (6/6)

Once execution dynamic parent evaluation

## Section Inherit

```
+ parent:OBJECT ←  
( + result:OBJECT;  
  ...// compute my parent  
  parent := result // my parent is a data now!!!  
);
```

## Note

- The first lookup, the parent is dynamically defined
- The next lookup, the parent is a simple data value

# Non-conforming inheritance

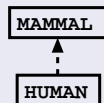
## Insert keyword

Section Header

```
+ name := HUMAN;
```

Section Insert

```
+ parent_mammal:Expanded MAMMAL;
```



## Example

```
+ a:MAMMAL;
```

```
a := HUMAN.clone; // Impossible!!!
```

## Warning

The **Expanded** default object has always non-conforming inheritance

# List: Set of Instructions & immediate evaluation (1/3)

## Without return value

```
( < Local >;  
  < Expr1 >;  
  < Expr2 >;  
  < Expr3 >;  
)
```

## With one return value

```
( < Local >;  
  < Expr1 >;  
  < Expr2 >;  
  < result >  
)
```

## With $n$ return value

```
( < Local >;  
  < Expr1 >;  
  < Expr2 >;  
  < result1 >,  
  < result2 >  
)
```



## List: Examples (2/3)

### For expressions

```
(2 + 4) * 7
```

### For procedures

```
- foo ←  
(  
  ''Hello''.print;  
);
```

### For functions

```
- zero:INTEGER ←  
(  
  ''Call zero''.print;  
  0  
);
```

## List : Examples (3/3)

For vector assignment

```
(a,b) := (3,7);
```

For functions with resultS

```
- coord:(INTEGER,INTEGER) ← ( x_current,y_current );
```

For vector argument

```
put_pixel (x,y) color 0;
```

Plugin of vectors

```
(x,y) := get_coord;  
put_pixel (x,y) color 0;  
≡  
put_pixel get_coord color 0;
```

# BLOCK: Set of instructions & late evaluation (1/4)

## Without return value

```
{ < Args >;  
  < Local >;  
  < Expr1 >;  
  < Expr2 >;  
}
```

## With one return value

```
{ < Args >;  
  < Local >;  
  < Expr1 >;  
  < Expr2 >;  
  < result >  
}
```

## With $n$ return value

```
{ < Args >;  
  < Local >;  
  < Expr1 >;  
  < Expr2 >;  
  < result1 >,  
  < result2 >  
}
```

# BLOCK vs List (2/4)

## List

```
( < Local >;  
  < Expr1 >;  
  < Expr2 >;  
  < Expr3 >;  
)
```

≡

## BLOCK.value

```
{ < Local >;  
  < Expr1 >;  
  < Expr2 >;  
  < Expr3 >;  
}.value
```

# BLOCK: Example (3/4)

## Embedded code in object

```
+ display: {(INTEGER,INTEGER); INTEGER};
display := { (x,y:INTEGER); // Vector parameter
             + sum:INTEGER; // One local variable
             x.print;
             ', '.print;
             y.print;
             sum := x + y;
             sum // The result block
             };
...
display.value (3,4) .print;
```

# BLOCK: Examples (4/4)

## For expressions

```
(a != NULL) && {a.value = 3}
```

## For conditionals

```
(a > b).if {  
  'y'.print;  
} else {  
  'n'.print;  
};
```

## For loops

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

## For iterations

```
1.to 10 do { j:INTEGER;  
  j.print;  
};
```

# C like Switch statement (1/3)

For vector assignment

```
foo.switch
.case 1 do {
    'Case 1'.print;
}.break
.case 2 do {
    'Case 2'.print;
}
.case 3 do {
    'Case 3'.print;
}
.default {
    'Default case'.print;
};
```

## C like Switch statement (2/3)

```
- Self:SELF.switch:(SELF,INTEGER_8) <- (Self, 0);

- (Self:SELF, stat:INTEGER_8).case
  value:SELF do body:{ } :(SELF,INTEGER_8) <-
( + new_stat:INTEGER_8;
  Self,
  (((stat = 0) && {value = Self}) || {stat = 1}).if {
    new_stat := 1;
    body.value;
  };
  new_stat
);
```



## C like Switch statement (3/3)

```
- (Self:SELF, stat:INTEGER_8).break:(SELF,INTEGER_8) <-  
( + new_stat:INTEGER_8;  
  Self,  
  (stat = 1).if {  
    new_stat := 2;  
  });  
new_stat  
);  
  
- (Self:SELF, stat:INTEGER_8).default body:{} <-  
(  
  (stat = 0).if body;  
);
```

# Auto-conversion : export (1/3)

## Example

### Section Header

```
+ name := Expanded CHARACTER;  
- export := INTEGER_8;
```

### Section Public

```
- to_integer_8:INTEGER_8 ← ...  
...  
( + a:CHARACTER;  
  + b:INTEGER_8;  
  ...  
  b := a; // ⇔ b := a.to_integer_8;
```

## Note

- export primitive is not transitivity
- ARRAY(INTEGER) type  $\implies$  to\_array\_of\_integer slot

# Auto-conversion : import (2/3)

## Example

### Section Header

```
+ name := Expanded CHARACTER;  
- import := INTEGER_8;
```

### Section Public

```
- from_integer_8 a:INTEGER_8 :SELF ← ...  
...  
( + a:CHARACTER;  
  + b:INTEGER_8;  
  ...  
  a := b; // ⇔ a := CHARACTER.from_integer_8 b;
```

# Auto-conversion : export/import (3/3)

## Priority for resolved confliting type

- 1 If source is a subtype of destination then OK, else
- 2 search an **export** in source static type to destination, else
- 3 search an **import** in destination static type for source, else
- 4 **Error** type mismatch!

# Default value of prototype

## Example

### Section Header

```
+ name := Expanded CHARACTER;  
- default := '\0';
```

### Section Header

```
+ name := STRING;  
- default := STRING.clone;
```

## Note

- By default, `NULL` is the default value for not Expanded prototype
- For Expanded prototype, the prototype is the default value

# Pattern code: pre-pattern (1/6)

## Definition Pre-pattern

The pattern code common at a set of the slot definition. This pattern code must be at the beginning of the code slot.

## Example in the parent

```
- my_slot ←  
[ // my pre-pattern  
  ''Call my_slot!'' .println;  
]  
( // my body  
  deferred; // abstract slot  
);
```

# Pattern code: pre-pattern (2/6)

## In two children redefinition

```
- my_slot ←  
( ''First!'' .print; );
```

```
- my_slot ←  
( ''Second!'' .print; );
```

## Result runtime

```
Call my_slot!  
First!
```

```
Call my_slot!  
Second!
```

# Pattern code: pre-pattern (3/6)

## In two children redefinition

```
- my_slot ←  
[ // redefine pattern  
  ''It's me!'' .println;  
]  
( ''First!'' .print; );  
  
- my_slot ←  
[ // recompose pattern  
  ''Old :'' .println;  
  ...  
  ''End!'' .println;  
]  
( ''Second!'' .print; );
```

## Result runtime

It's me!  
First!

Old :  
Call my\_slot!  
End!  
Second!



# Pattern code: post-pattern (4/6)

## Definition Post-pattern

The pattern code common at a set of the slot definition. This pattern code must be at the end of the code slot.

## Example

```
- my_slot ←  
( // my body  
  deferred; // abstract slot  
)  
[ // my post-pattern  
  ''End of call my_slot!'' .println;  
];
```

# Pattern code: out-pattern (5/6)

## Definition Out-pattern

The pattern code common at a set of all output slot definition.  
This pattern is common for all extern call slot prototype.

*Welcome in the Matrix!*

## Definition & note

- The out-pattern is define at the end of prototype/file
- The out-pattern is executing after the execution extern call.
- call of type `my_slot`: not execute out-pattern (not extern call)
- call of type `my_object.my_slot`: execute out-pattern
- call of type `Self.my_slot`: execute out-pattern

# Pattern code: in-pattern (6/6)

Progress. . .

Why not? In the future. . .

# Programming by contract : code level (1/5)

## Note

- The set of contract is tested during runtime.
- The violation of contract implies the crash of execution and to print of the stack runtime.
- The contract can be inhibited by the compiler option.

## Assertion in a list code

```
( // Source code ...  
  ? {j > 0}; // my assertion  
  // Source code ...  
)
```

# Programming by contract : Prototype level (2/5)

## Note

The **invariant** primitive uses the “**out-pattern**”

Invariant to end of prototype file

Section Header

+ name := COLLECTION(E);

Section Public

...

// The end of file :

[

? {lower <= upper + 1};

];

# Programming by contract : slot level (3/5)

## Note

- The **require** primitive use the “**pre-pattern**”
- The **ensure** primitive use the “**post-pattern**”

## Primitive additive for ensure

- **Old** : compute the expression value before the call slot. This primitive can be used in the body slot too.
- **Result** or **Result\_< n >** : send the result value of slot

*Example :*

```
? { Result = item upper } ;  
? { count = Old count } ;
```

# Programming by contract: Require/Ensure (4/5)

## Require / ensure on a slot

```
- swap idx1:INTEGER with idx2:INTEGER ←  
  // Swap item at index 'idx1' with item at index 'idx2'  
  [ // Require  
    ? {valid_index idx1};  
    ? {valid_index idx2};  
  ]  
  ( + tmp:E; // Body slot  
    tmp := item idx1;  
    put (item idx2) to idx1;    put tmp to idx2;  
  )  
  [ // Ensure  
    ? {item idx1 = Old item idx2};  
    ? {item idx2 = Old item idx1};  
  ];
```

# Programming by contract : Inheritance (5/5)

## Inheritance of contract

- By default, a prototype inherit all the contract of parent :
  - 1 Require on the slot
  - 2 Ensure on the slot
  - 3 Invariant on the prototype
- The redefine contract delete the old contract of parent
- In the redefine, you can paste the old contract with '...' primitive

## Note & resume...

- Require : test on arguments validity
- Ensure : test on results validity
- Invariant : test of the cohere on data set object
- Assertion : test a stat in the code (No inheritance primitive)



# Memory Mapping : hardware structure (1/3)

## Example for Global Descriptor Table on Intel x86

### Section Header

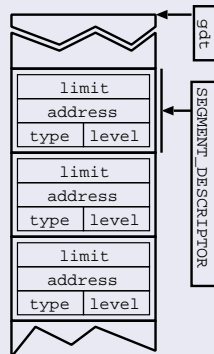
```
+ name := SEGMENT_DESCRIPTOR;
```

### Section Mapping

```
+ limit:INTEGER_32;  
+ address:INTEGER_32;  
+ type:INTEGER_16;  
+ level:INTEGER_16;
```

...

```
- gdt:NATIVE_ARRAY (Expanded SEGMENT_DESCRIPTOR);
```



# Memory Mapping: binary file structure (2/3)

## Section Header

```
+ name := MY_STRUCT;
```

## Section Mapping

```
+ coord_x:INTEGER_32;
```

```
+ coord_y:INTEGER_32;
```

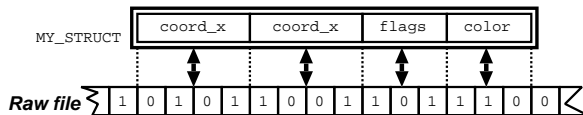
```
+ flags:INTEGER_16;
```

```
+ color:INTEGER_16;
```

## Section Public

```
- move ← ...
```

```
- set_color ← ...
```



# Memory Mapping: composite structure example (3/3)

## Section Header

```
+ name := STRUCT_1;
```

## Section Mapping

```
+ code:INTEGER_32;
```

```
+ stat:Expanded STRUCT_2;
```

```
+ type:INTEGER_16;
```

...

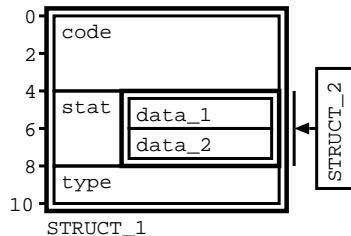
## Section Header

```
+ name := STRUCT_2;
```

## Section Mapping

```
+ data_1:INTEGER_16;
```

```
+ data_2:INTEGER_16;
```



# Interrupt hardware manager

## Example

### Section Interrupt

```
- my_interrupt ←  
( // Code Lisaac ...  
);
```

## Note

- Can't a call direct `my_interrupt` slot
- `my_interrupt` call send a `POINTER` address function. It's necessary for to put this address in Interrupt Descriptor Table.

## Restriction

- Parameter or result is prohibited
- The function should not be Self dependent

# External C to Lisaac (1/4)

## Example without result

```
- die_with_code code:INTEGER ← 'exit(@code)';
```

## Note

- @<identifier> for access to **local variable only**  
(or argument)
- This access is always **read only**.

# External C to Lisaac : with result (2/4)

## Example

- Persistent external :
  - `basic_getc` ← `'getchar()':(CHARACTER);`
- Non persistent external :
  - `Self:SELF '>>' other:SELF :SELF` ←  
`'@Self>>@other':SELF;`

## Note : Warning

- **Persistent** : The persistent external means that the code will remain present even if the return value is not used. Parentheses in the type of return shows that the return value is not important, is the execution of this external is important.
- **Non persistent** : If the result external is not used, then the external is deleted by the compiler.

# External C to Lisaac: dynamic type (3/4)

## Example

```
- Self:SELF '>' other:SELF :BOOLEAN ←  
'@Self>@other':BOOLEAN{TRUE,FALSE};
```

## Note

- This **static type** result is `BOOLEAN`
- The **dynamic type set** for this result is `TRUE` or `FALSE`
- Each dynamic type must be a sub type of static type

# External C to Lisaac : mapping C type (4/4)

## Example

### Section Header

- + name := Expanded CHARACTER;
- type := 'signed char';

## Note

The compiler translate the CHARACTER with C type signed char

## Warning

With Expanded or not and the C type :

- Expanded type  $\implies$  No pointer C type
- No Expanded type  $\implies$  Pointer C type



# External Lisaac to C

## examples

### Section External

```
- function_for_c (a,b:INTEGER) :INTEGER ←  
  ( // Code Lisaac ...  
  );
```

## Note

Here, we have a function `int function_for_c(int a,int b)` in C code product

## Restriction

- Several keywords for the name function is prohibited
- The function should not be Self dependent
- The vector result is prohibited

# External intern of Lisaac

## Definition

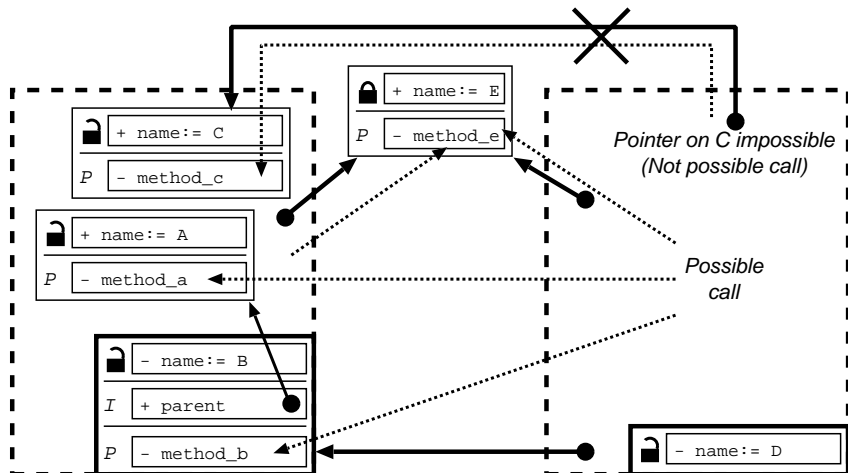
This is a fundamental external known and used by the compiler.

Syntax: '**<number>**' with  $number \in [0..31]$

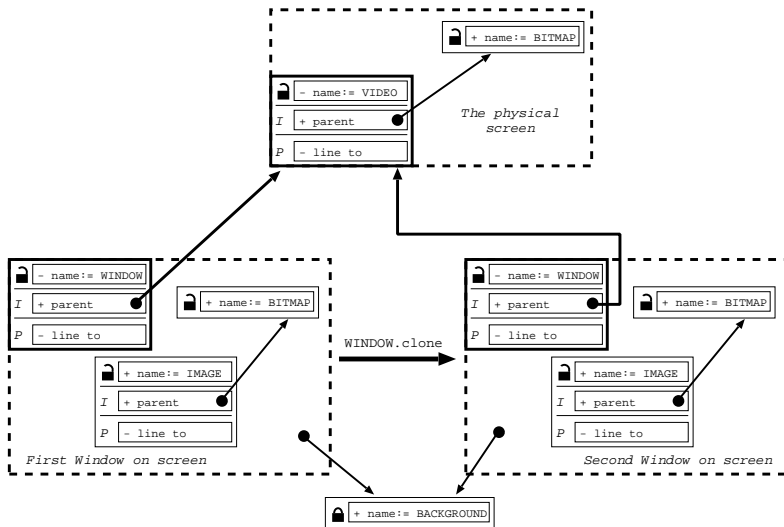
## examples

```
- Self:SELF '-' Left 80 other:SELF :SELF ← '1';  
- Self:SELF '*' Left 100 other:SELF :SELF ← '2';  
- Self:SELF '/' Left 100 other:SELF :SELF ← '3';  
- Self:SELF '&' Left 100 other:SELF :SELF ← '4';  
- Self:SELF '>' Left 100 other:SELF :BOOLEAN ← '5';
```

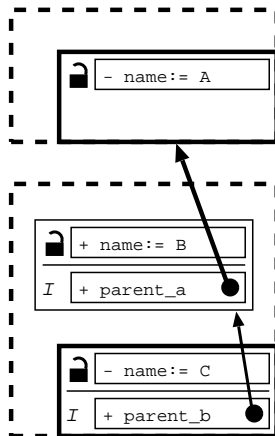
# COP : Concurrent Object Prototypes (1/4)



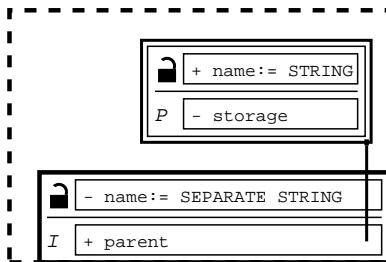
# COP : Concurrent Object Prototypes (2/4)



# COP : Concurrent Object Prototypes (3/4)



# COP : Concurrent Object Prototypes (4/4)



# LIP : Lisaac Project manager (1/11)



## One file = one project

By default: `lisaac/make.lip`

- Communication between Compiler and Lip file:  
*Via Intern variables*
- Full configuration of compiler options
- Subset Lisaac language Interpreter
- Dynamic description of paths directories
- Set of instructions before compilation pass (Front-end)
- Set of instructions after compilation pass (Back-end)
- Dynamic execution during compilation in live prototype context

# LIP : Lip file location (2/11)

## Explicite path for a Lip file

```
sonntag@isaac:~/slides/lisaac$ lisaac ../project/make.lip
```

## Implicite research

- 1 Search lip file in current directory.
- 2 if failed, search in parent of directory.
- 3 go to (2) until the root directory
- 4 Else, search lip file by default (`lisaac/make.lip`)



# Lip: Intern variables (3/11)

Compiler  $\Rightarrow$  Lip (*immediately*)

```
+ lisaac:STRING;
```

Example: /home/sonntag/lisaac/

- 1 Read LISAAC\_DIRECTORY environnement variable
- 2 if (1) failed, search #define LISAAC\_DIRECTORY in path.h

Compiler  $\Rightarrow$  Lip (*immediately*)

```
+ input_file:STRING;
```

Example: hello\_world (*Read command line argument*)

Compiler  $\Rightarrow$  Lip (*after compilation*)

```
+ is_cop:BOOLEAN;
```

# Lip: Intern variables (4/11)

Compiler  $\Leftarrow$  Lip (*Debug information*)

```
+ debug_level:INTEGER;  
+ debug_with_code:BOOLEAN;  
+ is_all_warning:BOOLEAN;
```

Compiler  $\Leftarrow$  Lip (*Optimization*)

```
+ is_optimization:BOOLEAN;  
+ inline_level:INTEGER;
```

# Lip: Intern variables (5/11)

Compiler  $\Leftarrow$  Lip (*Generate code*)

+ `is_java:BOOLEAN;`

Compiler  $\Leftarrow$  Lip (*Other*)

+ `is_statistic:BOOLEAN;`

+ `is_quiet:BOOLEAN;`

# Lip: Subset Lisaac language (6/11)

## Syntax

- Types: **BOOLEAN, STRING, INTEGER**

- Binary Operators:

|   &   +   -   <   >   ≤   ≥   =   !=

- Unary Operators: -   !

- Assignment: :=

- Style slot:

- + data slot
- - method slot

*(with 0 or 1 parameter and without return value)*

# Lip: Subset Lisaac language (7/11)

## Slot built-in

- `BOOLEAN.if { ... }`
- `BOOLEAN.if { ... } else { ... }`
- `BOOLEAN||STRING||INTEGER.print`
- `path text:STRING`
- `run cmd:STRING :INTEGER`
- `get_integer:INTEGER`
- `get_string:STRING`
- `exit`

# Lip: Option description (8/11)

In **Section Public**

```
- debug level:INTEGER < -  
// Fix debug level (default: 15)  
(  
  ((level < 1) | (level > 20)).if {  
    "Incorrect debug level.".print;  
    exit;  
  };  
  debug_level := level;  
);
```

Compiler Lisaac option

Options:

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

# Lip: Other Section (9/11)

## In **Section Private**

- Others code slots.
- Data slot intern and others data slots.

## In **Section Inherit** (*Multi-inheritance*)

- With lip path :  
+ `parent:STRING := '../my_project/linux/'`;
- Without path: Inheritance Lip file by default.  
+ `parent:STRING`;

## Inheritance

- Redefinition slot is authorized.
- Lookup algorithm is active.

# Lip: Particular method slot (10/11)

## front\_end

Executed by compiler, before compilation step.

- Detect operating system,
- Loading path set for a project,

## back\_end

Executed by compiler, after compilation step.

- Added gcc options, lib, ...
- Finalize the compilation with gcc or others

## Warning

`back_end` & `front_end` is mandatory in **Section Private**



# Lip: Dynamic execution during compilation (11/11)

## In the Section Header

### Section Header

```
+ name := VIDEO;  
- lip <- ( add_lib '-lX11' );
```

## In make.lip

```
- add_lib lib:STRING <-  
( run "echo \"int main(){ return(1); }\" > _t.c";  
  (run("gcc _t.c"+lib+" 2>/dev/null")=0).if {  
    lib_gcc := lib_gcc + " " + lib;  
  } else {  
    ("ERROR: '" + lib + "' lib not found.").print;  
    run "rm _t.c"; exit;  
  };  
)
```

# Question ?

## IRC

- Server: `irc.oftc.net`
- Channel: `#isaac`

## Information & contacts

- **Wiki**: <http://wiki.lisaac.org>
- **Mailing list**:  
<http://www.lisaac.org/community/contact>



*Good luck!*