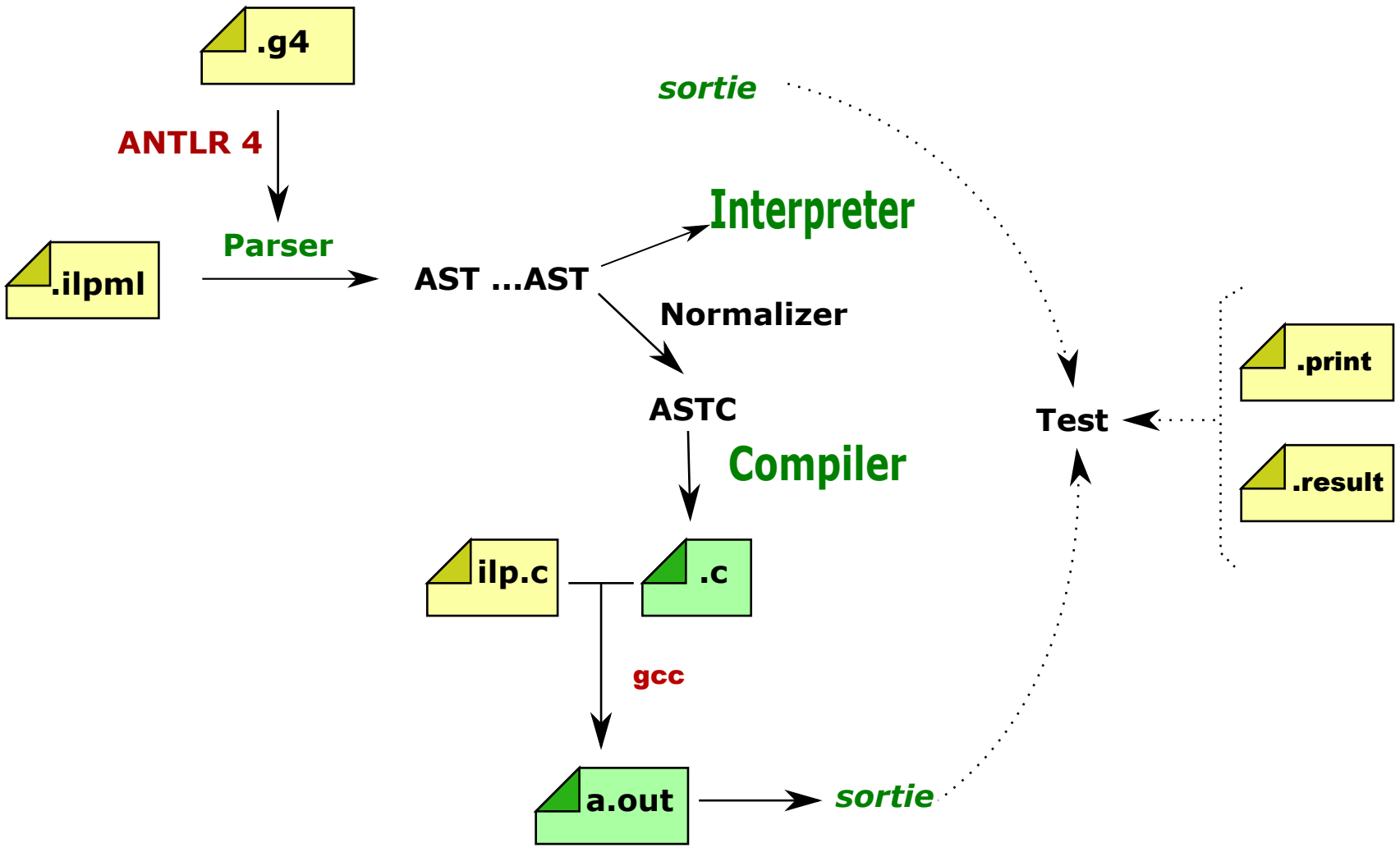
Sorbonne Université
Master Informatique 2020-2021
Spécialité STL
Développement des langages de programmation
DLP – MU4IN501

Carlos Agon
agonc@ircam.fr

Grand schéma



Plan du cours 3

- Compilation vers C
- Représentation des concepts en C
- Bibliothèque d'exécution

Compilation

Analyser la représentation du programme pour le transformer en un programme calculant sa valeur et son effet.
Un interprète fait, un compilateur fait faire.

- Programme : données $\Rightarrow$ résultat
- Interprète : programme $\times$ données $\Rightarrow$ résultat
- Compilateur : programme $\Rightarrow$ (données $\rightarrow$ résultat)

Une machine abstraite (super simple)

Le langage :

e:= N | e + e | e - e

Le jeu d'instructions de la machine :

CONST(N)	empiler l'entier N
ADD	dépiler deux entiers, empiler leur somme
SUB	dépiler deux entiers, empiler leur différence

Schéma de compilation :

C[N] = CONST(N)

C[a1 + a2] = C(a1); C[a2]; ADD

C[a1 - a2] = C(a1); C[a2]; SUB

Exemple :

C[3 – 1 + 2] = CONST(3); CONST(1) ; CONST(2);ADD ; SUB

Une machine abstraite arithmétique

Composants de la machine :

① Un pointeur de code
② Une pile

Transactions de la machine :

Etat avant		Etat après	
Code	Pile	Code	Pile
CONST(n);c	s	c	n.s
ADD;c	n2.n1;s	c	(n1 + n2).s
SUB;c	n2.n1;s	c	(n1 - n2).s

Evaluation

Etat initial code = C[exp] et pile = ε

Etat final code = ε et pile = v. ε v le résultat

Code	Pile
`CONST(3) ; CONST(1) ; CONST(2) ; ADD ; SUB`	ε
`CONST(1) ; CONST(2) ; ADD ; SUB`	3.ε
`CONST(2) ; ADD ; SUB`	1.3.ε
`ADD ; SUB`	2.1.3ε
`SUB`	3.3.ε
ε	0.ε

Exécution du code par interprétation

Interprète écrit en C ou assembler.

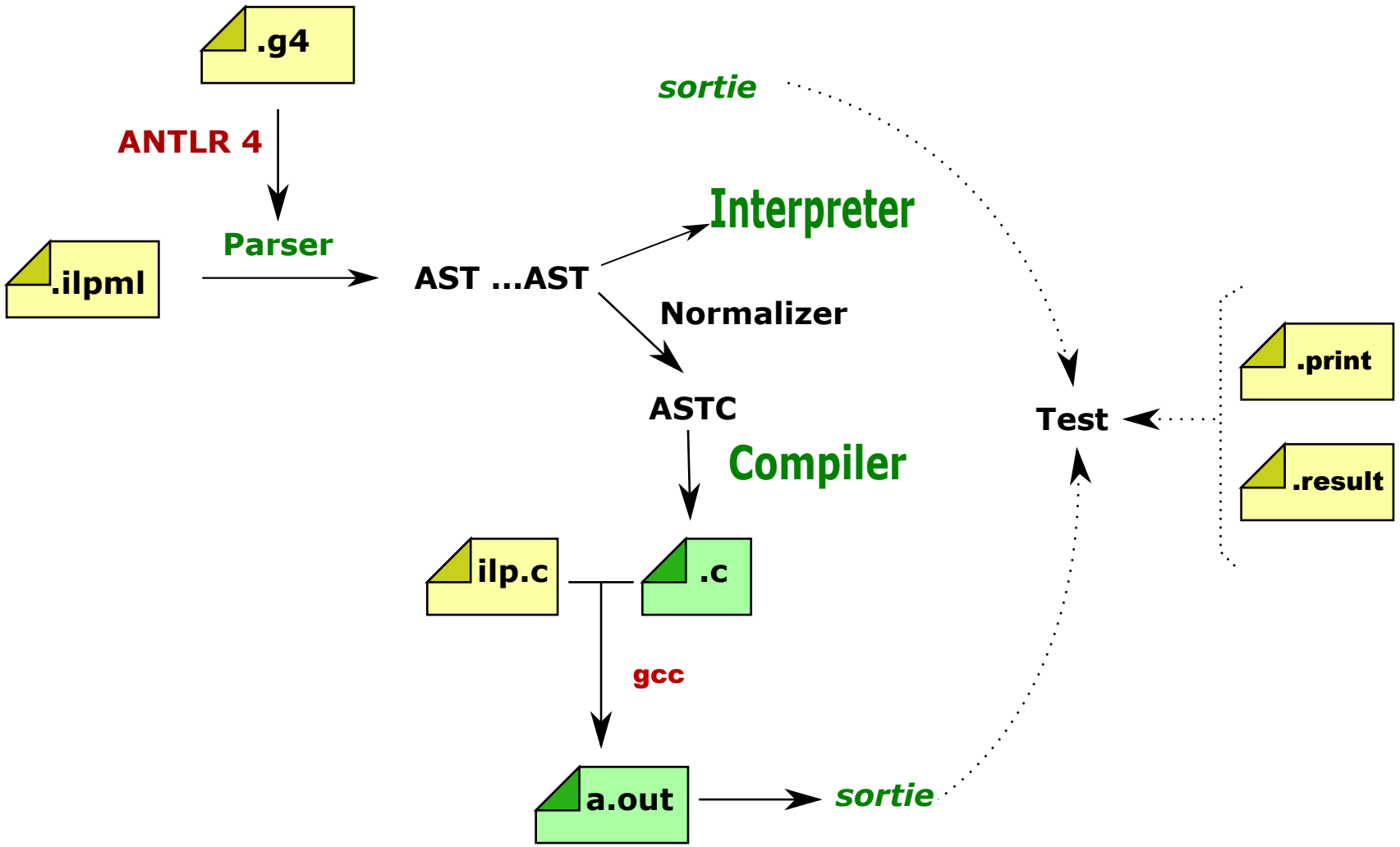
```
int interpreter(int * code)
{
int * s = bottom_of_stack;
while (1) {
 switch (*code++) {
  case CONST: *s++ = *code++; break;
  case ADD: s[-2] = s[-2] + s[-1]; s--; break;
  case SUB: s[-2] = s[-2] - s[-1]; s--; break;
  case EPSILON: return s[-1];
  }
 }
}
```

Exécution du code par expansion

Plus vite encore, convertir les instructions abstraites en séquences de code machine.

CONST(i)	--->	pushl $i
ADD	--->	popl %eax addl 0(%esp), %eax
SUB	--->	popl %eax subl 0(%esp), %eax
EPSILON	--->	popl %eax ret

Grand schéma



Compilation

Traduire : un programme vers du code en langage machine :

- 50's assembler du code machine textuel pour des langages de haut niveau (Fortran)
- 60's Les langages évoluent (la récursion) le langage machine suit (utilisation d'une pile), mais pas tant que ça.
- 80's Représentation automatique des données, le langage machine ne suit plus..

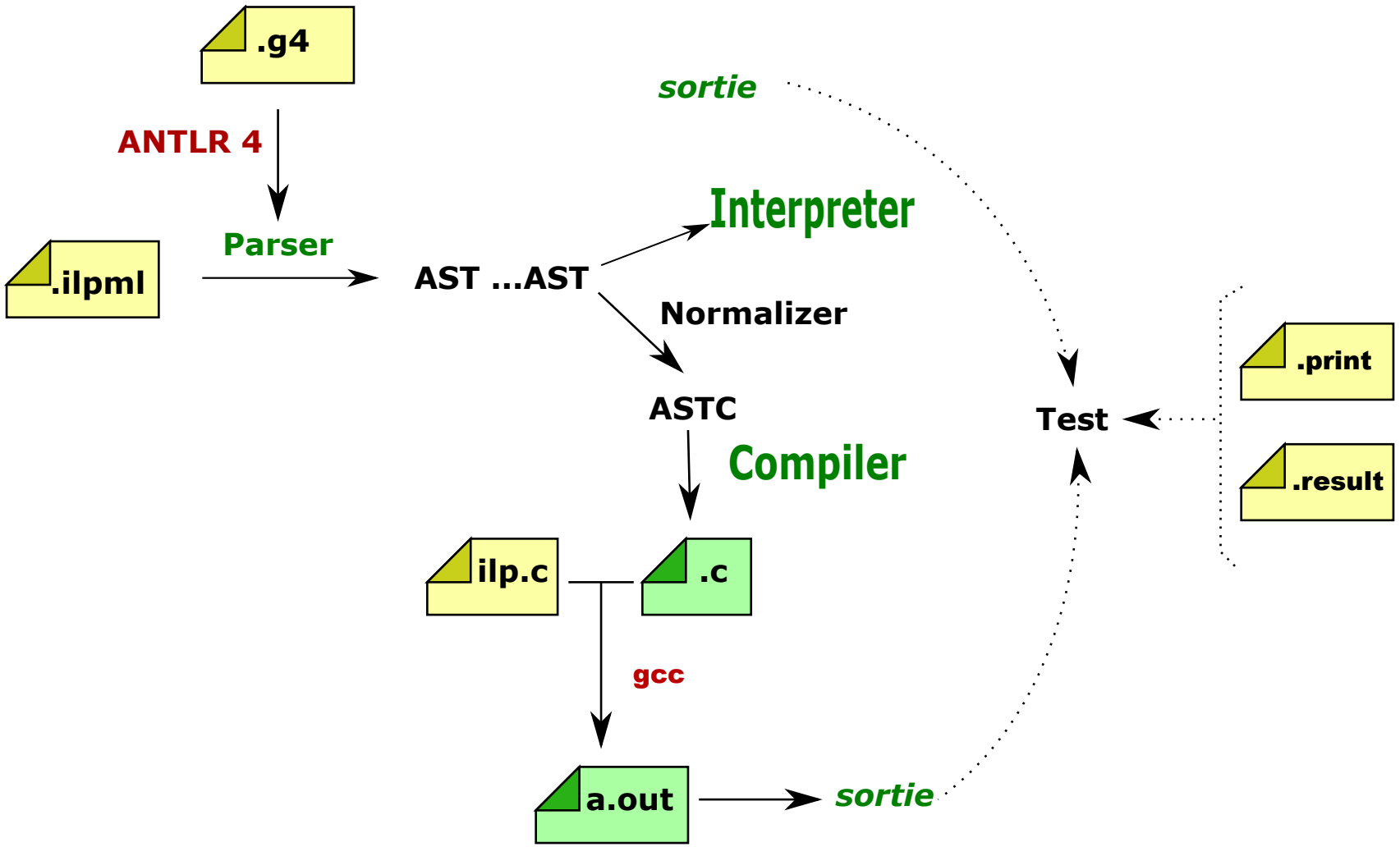
Il y a de plus en plus un écart entre l'expressivité des langages des haut niveau et celle du langage machine - le compilateur doit s'occuper de cette fracture.

Compilation - autres tâches

- **Link** : liaison des fichiers des unités de compilation (module, class, interface, package, etc.)
- **Runtime** : utilisation des bibliothèques d'exécution (i/o, gc, appels de méthodes, etc.)
- **VM** : production de bytecode et interprétation/compilation (jit) vers du code machine
- **Optimisations** : temps, espace, énergie
- **Sûreté** : typage
- ...

Notre choix : génération de code C

Grand schéma



Concepts présents dans ILP1

- Les structures de contrôle : alternative, séquence, bloc local
- les opérateurs : `+`, `-`, etc.
- les fonctions primitives : `print`, `newline`
- instruction, expression, variable, opération, invocation
- les valeurs : entiers, flottants, chaînes, booléens.

mais, en C, pas de typage dynamique, pas de gestion de la mémoire.
Par contre, C connaît la notion d'environnement.

Hypothèses

Le compilateur est écrit en Java.

1. Il prend un IAST,
2. le compile en C.

Il ne se soucie donc pas des problèmes syntaxiques d'ILP1 mais uniquement des problèmes sémantiques

- que ce soit lui qui le traite (propriété **statique**)
- ou le code engendré qui le traite (propriété **dynamique**).

Statique/dynamique

Est **dynamique** ce qui ne peut être résolu qu'à l'exécution.
Est **statique** ce qui peut être déterminé avant l'exécution.

Statique et dynamique

```
{ int x = round(2.78);
  print(y);                  // y variable inconnue!
  float z;
  print(z);                  // z non initialisee!
  if ( foo(x) ) {
    z = x/(3 - x);           //
    print(z);                // z est definie
  };
  print(z)                   // qu'est z ?
}
```

Statique et dynamique : examen 2014

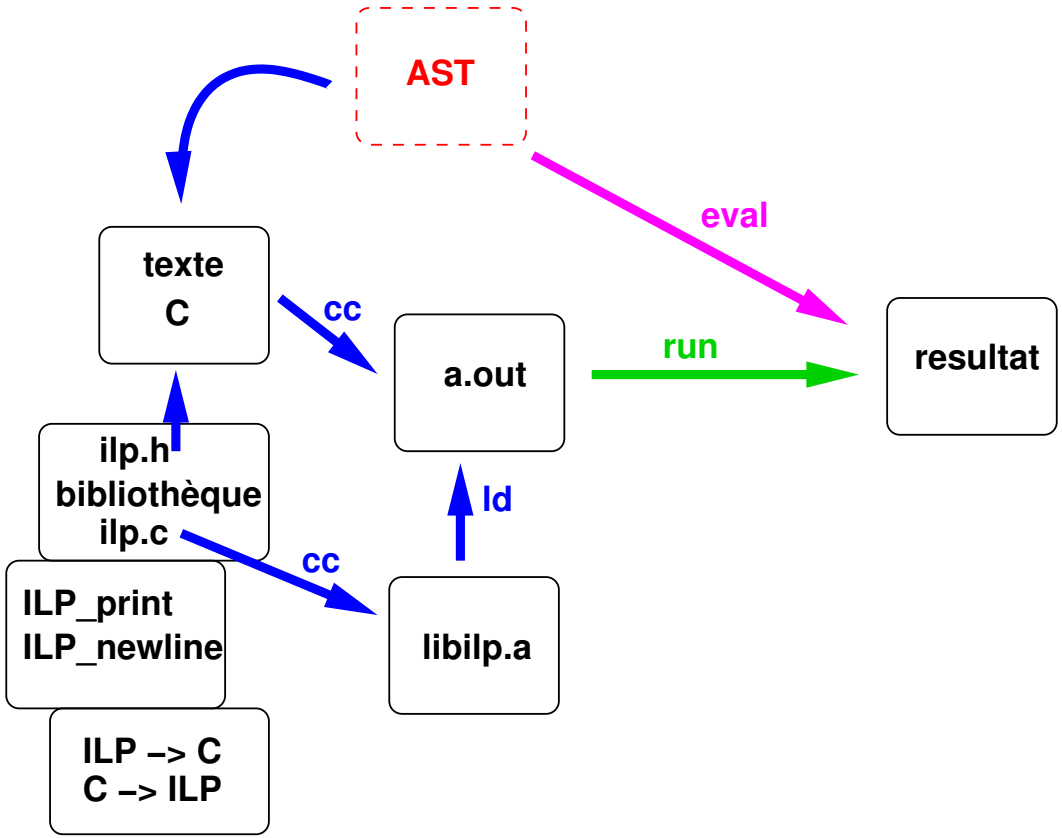
On souhaite étendre ILP2 avec l'introduction des fonctions auxiliaires *avant* et *après*. Lors de la définition d'une fonction vous pouvez ajouter de manière optionnelle un qualificateur `:before` ou `:after`, comme l'illustre le programme suivant :

```
function foo (i) {
  print (i); i}

function foo :before (i) {
  print ("Before foo...");}

function foo :after (i) {
  print ("...After foo");}
```

L'appel de fonction `foo (2)` imprimera `"Before foo...2...After foo"` et retournera 2.
Expliquez aussi si la gestion des erreurs se fait de manière dynamique ou statique.

Composantes

On souhaite que le compilateur ne dépende pas de la représentation exacte des données.

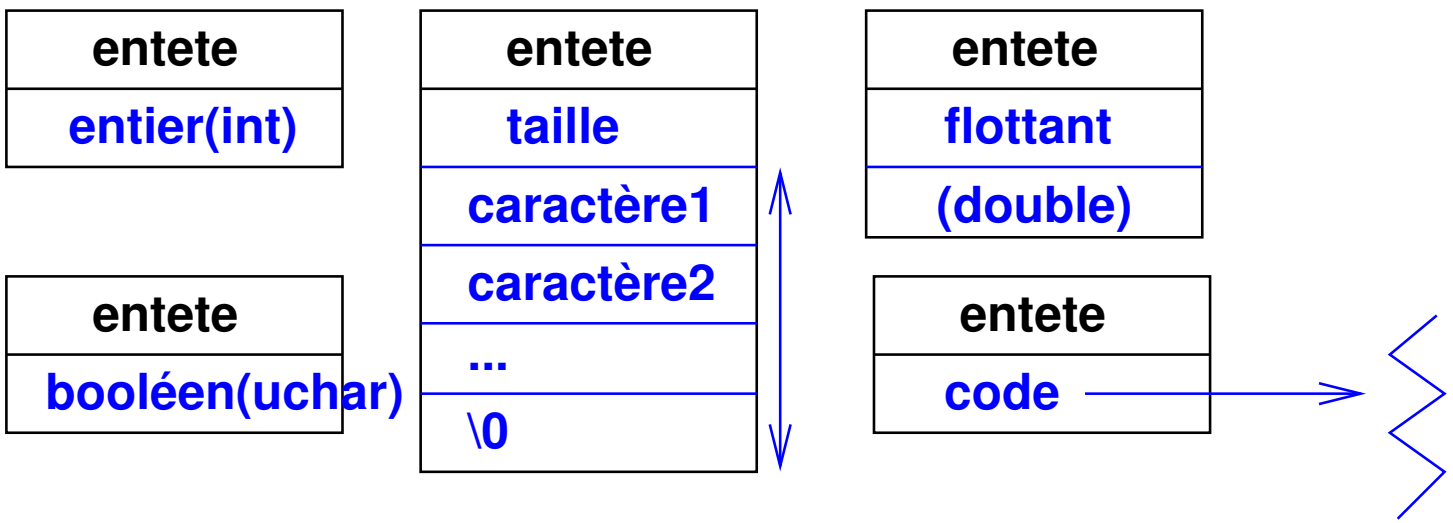


Représentation des valeurs

On s'appuie sur C :

Ressource: C/ilp.c

Afin de pouvoir identifier leur type à l'exécution (propriété dynamique), toute valeur est une structure allouée dotée d'un entête (indiquant son type) et d'un corps et manipulée par un pointeur.



```
typedef struct ILP_Object {
      struct ILP_Class*  _class;
      union {
            unsigned  char asBoolean;
            int            asInteger;
            double         asFloat;
            struct asString {
                  int      _size;
                  char     asCharacter[1];
            } asString;
          struct asClass {
                  struct ILP_Class*   super;
                  char*               name;
                  int                 fields_count;
                  struct ILP_Field*   last_field;
                  int                 methods_count;
                  ILP_general_function method[1];
            } asClass;
          ...
      }                   _content;
} *ILP_Object;
```

```
typedef struct ILP_Class {
      struct ILP_Class* _class;
      union {
            struct asClass_ {
                  struct ILP_Class*    super;
                  char*                name;
                  int                  fields_count;
                  struct ILP_Field*    last_field;
                  int                  methods_count;
                  ILP_general_function method[2];
            } asClass;
      }                    _content;
} *ILP_Class;
```

Exemple de classes

```
struct ILP_Class ILP_object_Integer_class = {
     &ILP_object_Class_class,
     { { &ILP_object_Object_class,
         "Integer",
         0,
         NULL,
         2,
         { ILP_print,
           ILP_classOf } } }
};
```

```
struct ILP_Class ILP_object_Boolean_class = {
     &ILP_object_Class_class,
     { { &ILP_object_Object_class,
         "Boolean",
         0,
         NULL,
         2,
         { ILP_print,
           ILP_classOf } } }
};
```

Structures

Pour chaque type de données d'ILP :

- constructeurs (allocateurs)
- reconnaisseur (grâce au type présent à l'exécution)
- accesseurs
- opérateurs divers

et, à chaque fois, les macros (l'interface) et les fonctions (l'implantation).

Autour des booléens

Fonctions ou macros d'appoint :

```
1  #define ILP_TRUE  (&ILP_object_true)
2  #define ILP_FALSE (&ILP_object_false)
3
4
5  #define ILP_Boolean2ILP(b) \
6    ILP_make_boolean(b)
7
8  #define ILP_isBoolean(o) \
9    ((o)->_class == &ILP_object_Boolean_class)
10
11 #define ILP_isTrue(o) \
12   (((o)->_class == &ILP_object_Boolean_class) && \
13    ((o)->_content.asBoolean))
14
15 #define ILP_isEquivalentToTrue(o) \
16   ((o) != ILP_FALSE)
17
18 #define ILP_CheckIfBoolean(o) \
19   if ( ! ILP_isBoolean(o) ) { \
20         ILP_domain_error("Not a boolean", o); \
21   };
```

Implantation

```
1  enum ILP_BOOLEAN_VALUE {
2       ILP_BOOLEAN_FALSE_VALUE = 0,
3       ILP_BOOLEAN_TRUE_VALUE  = 1
4  };
5
6  struct ILP_Object ILP_object_true = {
7       &ILP_object_Boolean_class,
8       { ILP_BOOLEAN_TRUE_VALUE }
9  };
10
11 struct ILP_Object ILP_object_false = {
12      &ILP_object_Boolean_class,
13      { ILP_BOOLEAN_FALSE_VALUE }
14 };
15
16 ILP_Object
17 ILP_make_boolean (int b)
18 {
19      if ( b ) {
20           return ILP_TRUE;
21      } else {
22           return ILP_FALSE;
23      }
24 }
```

Autour des entiers

Fonctions ou macros d'appoint :

```
#define ILP_Integer2ILP(i) \
  ILP_make_integer(i)

#define ILP_AllocateInteger() \
  ILP_malloc(sizeof(struct ILP_Object),
    &ILP_object_Integer_class)

#define ILP_isInteger(o) \
  ((o)->_class == &ILP_object_Integer_class)

#define ILP_CheckIfInteger(o) \
  if ( ! ILP_isInteger(o) ) { \
       ILP_domain_error("Not an integer", o);
  };
```

```
1  #define ILP_Plus(o1,o2) \
2    ILP_make_addition(o1, o2)
3
4  #define ILP_Minus(o1,o2) \
5    ILP_make_subtraction(o1, o2)
6
7  #define ILP_Times(o1,o2) \
8    ILP_make_multiplication(o1, o2)
9
10 #define ILP_Divide(o1,o2) \
11   ILP_make_division(o1, o2)
12
13 #define ILP_Modulo(o1,o2) \
14   ILP_make_modulo(o1, o2)
15
16 #define ILP_LessThan(o1,o2) \
```

Allocation de la mémoire

```
1  #ifdef WITH_GC
2      /* If Boehm's GC is present: */
3  #   include "include/gc.h"
4  #   define ILP_START_GC GC_init()
5  #   define ILP_MALLOC GC_malloc
6  #else
7  #   define ILP_START_GC
8  #   define ILP_MALLOC malloc
9  #endif
10
11 ILP_Object
12 ILP_malloc (int size, ILP_Class class)
13 {
14       ILP_Object result = ILP_MALLOC(size);
15       if ( result == NULL ) {
16           return ILP_die("Memory exhaustion");
17       };
18       result->_class = class;
```

```
1  ILP_Object
2  ILP_make_addition (ILP_Object o1, ILP_Object o2)
3  {
4    if ( ILP_isInteger(o1) ) {
5      if ( ILP_isInteger(o2) ) {
6        ILP_Object result = ILP_AllocateInteger();
7        result->_content.asInteger =
8            o1->_content.asInteger
9          + o2->_content.asInteger;
10       return result;
11     } else if ( ILP_isFloat(o2) ) {
12       ILP_Object result = ILP_AllocateFloat();
13       result->_content.asFloat =
14           o1->_content.asInteger
15         + o2->_content.asFloat;
16       return result;
17     } else {
18       return ILP_domain_error("Not a number", o2);
19     }
20     ...
```

Attention : l'addition consomme de la mémoire (comme en Java) !

```
#define DefineComparator(name,op) \
ILP_Object                        \
ILP_compare_##name (ILP_Object o1, ILP_Object o2) \
{                                 \
  if ( ILP_isInteger(o1) ) {      \
    if ( ILP_isInteger(o2) ) {    \
      return ILP_make_boolean(    \
          o1->_content.asInteger \
        op o2->_content.asInteger);\
    } else if ( ILP_isFloat(o2) ) {   \
      return ILP_make_boolean(        \
          o1->_content.asInteger \
        op o2->_content.asFloat); \
    } else {                            \
      return ILP_domain_error("Not a number", o2); \
    }
   ...
```

Primitives

```
1  ILP_Object
2  ILP_print (ILP_Object self)
3  {
4  if ( self->_class == &ILP_object_Integer_class ) {
5   fprintf(stdout, "%d", self->_content.asInteger);
6  } else if (self->_class == &ILP_object_Float_class ) {
7   fprintf(stdout, "%12.5g", self->_content.asFloat);
8  } else if (self->_class == &ILP_object_Boolean_class ) {
9   fprintf(stdout, "%s", (ILP_isTrue(self)?"true":"false"));
10 } else if (self->_class == &ILP_object_String_class ) {
11  fprintf(stdout, "%s", self->_content.asString.asCharacter);
12 } else if ...
13  }
14 return ILP_FALSE;
15 }
```

Mise en œuvre du compilateur

Ressource: ilp1.compiler

```
1  public class Compiler
2  implements IASTCvisitor<Void, Compiler.Context, CompilationException> {
3
4      public Compiler (IOperatorEnvironment ioe,
5                       IGlobalVariableEnvironment igve ) {
6          this.operatorEnvironment = ioe;
7          this.globalVariableEnvironment = igve;
8      }
9      protected final IOperatorEnvironment operatorEnvironment;
10     protected final IGlobalVariableEnvironment globalVariableEnvironment;
11
12 //
13
14     public String compile(IASTprogram program)
15             throws CompilationException {
16         ...
17         IASTCprogram newprogram = normalize(program);
18         newprogram = optimizer.transform(newprogram);
19         ...
20         Context context = new Context(NoDestination.NO_DESTINATION);
21         StringWriter sw = new StringWriter();
22         try {
23             out = new BufferedWriter(sw);
24             visit(newprogram, context);
25             out.flush();
26         } catch (IOException exc) {
27             throw new CompilationException(exc);
28         }
29         return sw.toString();
30     }
```

Récapitulation

- le compilateur fait faire !
- bibliothèque d'exécution
- représentation des données en C
  - constructeur, reconnaisseur, accesseur
- conversion ILP <-> C
- statique/dynamique
- transformation de l'AST