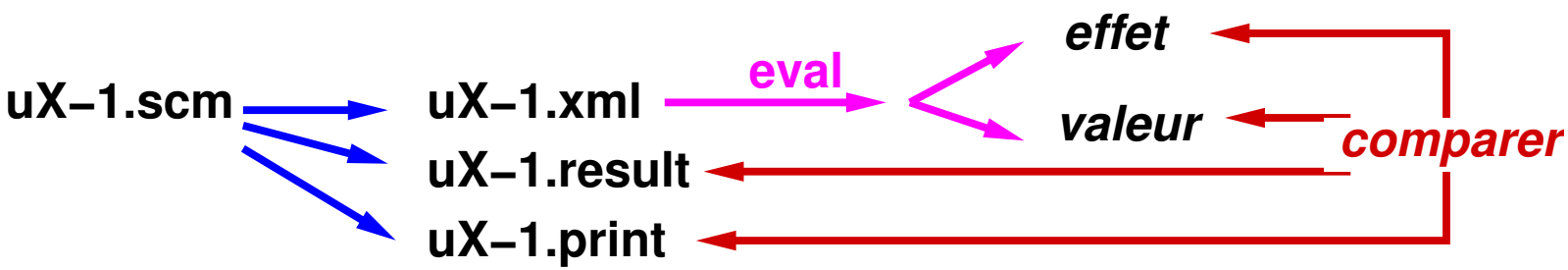
Master Informatique 2015-2016
Spécialité STL
Développement des langages de programmation
DLP – 4I501

Carlos Agon
agonc@ircam.fr

Batterie de tests

Ressource: Samples/u*-1.*



Ressource: com.paracamplus.ilp1.interpreter.test.InterpreterTest.java

Tests

Tests avec JUnit3 Cf. `http://www.junit.org/`

```
package com.paracamplus.ilp1.tools.test;

import junit.framework.TestCase;

import com.paracamplus.ilp1.tools.ProgramCaller;

public class ProgramCallerTest extends TestCase {

    public void testProgramCallerInexistentVerbose () {
        final String programName = "lasdljsdfousadfl lsjd";
        ProgramCaller pc =
                          new ProgramCaller(programName);
        assertNotNull(pc);
        pc.setVerbose();
        pc.run();
        assertTrue(pc.getExitValue() != 0);
    }
```

Séquencement JUnit3

Pour une classe de tests `SomeTest` :

1. charger la classe de test `SomeTest`
2. pour chaque méthode nommée `test`*X*,
   1. instancier la classe de test `SomeTest`
   2. tourner `setUp()`
   3. tourner `test`*X*
   4. tourner `tearDown()`

JUnit 4

Les tests ne sont plus déclarés par héritage mais par annotation (cf. aussi TestNG). Les annotations sont (sur les méthodes) :

```
@BeforeClass
@Before
@Test
@Test(expected = Exception.class)
@After
@AfterClass
```

et quelques autres comme (sur les classes) :

```
@RunWith   @SuiteClasses
           @Parameters
```

Séquencement JUnit4

Pour une classe de tests `Foobar` :

1. charger la classe `Foobar`
2. tourner toutes les methodes `@BeforeClass`
3. pour chaque méthode annotée `@Test`,
   1. instancier la classe `Foobar`
   2. tourner toutes les méthodes `@Before`
   3. tourner la méthode testée
   4. tourner toutes les méthodes `@After`
4. enfin, tourner toutes les methodes `@AfterClass`

Annotations

Les annotations sont des métadonnées dans le code source

- originalement en JAVA avec Javadoc
- annotations connues : @Deprecated, @Override, ...
- annotations multi paramétrées : @Annotation(arg1="val1", arg2="val2", ...)

Utilisations des annotations :

- par le compilateur pour détecter des erreurs
- pour la documentation
- pour la génération de code
- pour la génération de fichiers

Avec les annotations le code source est parcouru mais il n'est pas modifié.

Définition d'une annotation

```
public @interface MyAnnotation {
int arg1() default 4;
String arg2();
}
```

```
@MyAnnotation(arg1=0, arg2="valeur2")
public class UneCLasse {
...
}
```

Les annotations des annotations

```
@Target({ElementType.METHOD, ElementType.CONSTRUCTOR })
@Retention(RetentionPolicy.RUNTIME)
@Inherited
public @interface MyAnnotation {
int arg1() default 4;
String arg2();
}
```

Pour l'annotation @Retention :

- RetentionPolicy.SOURCE : dans le code source uniquement (ignorée par le compilateur)
- RetentionPolicy.CLASS : dans le code source et le bytecode (fichier .java et .class)
- RetentionPolicy.RUNTIME : dans le code source et le bytecode et pendant l'exécution par introspection

Les annotations pendant l'exécution

La pluspart des méta-objets implémentent `java.lang.reflect.AnnotatedElement` :

- `boolean isAnnotationPresent(Class<? extends Annotation>)` :
  True si le méta-objet est annoté avec le type du paramètre
- `<T extends Annotation> getAnnotation(Class<T>)` :
  renvoie l'annotation de type T ou null
- `Annotation[] getAnnotations()` :
  renvoie la liste des annotations
- `Annotation[] getDeclaredAnnotations()` :
  renvoie la liste des annotations directes (pas les heritées)

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)

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..

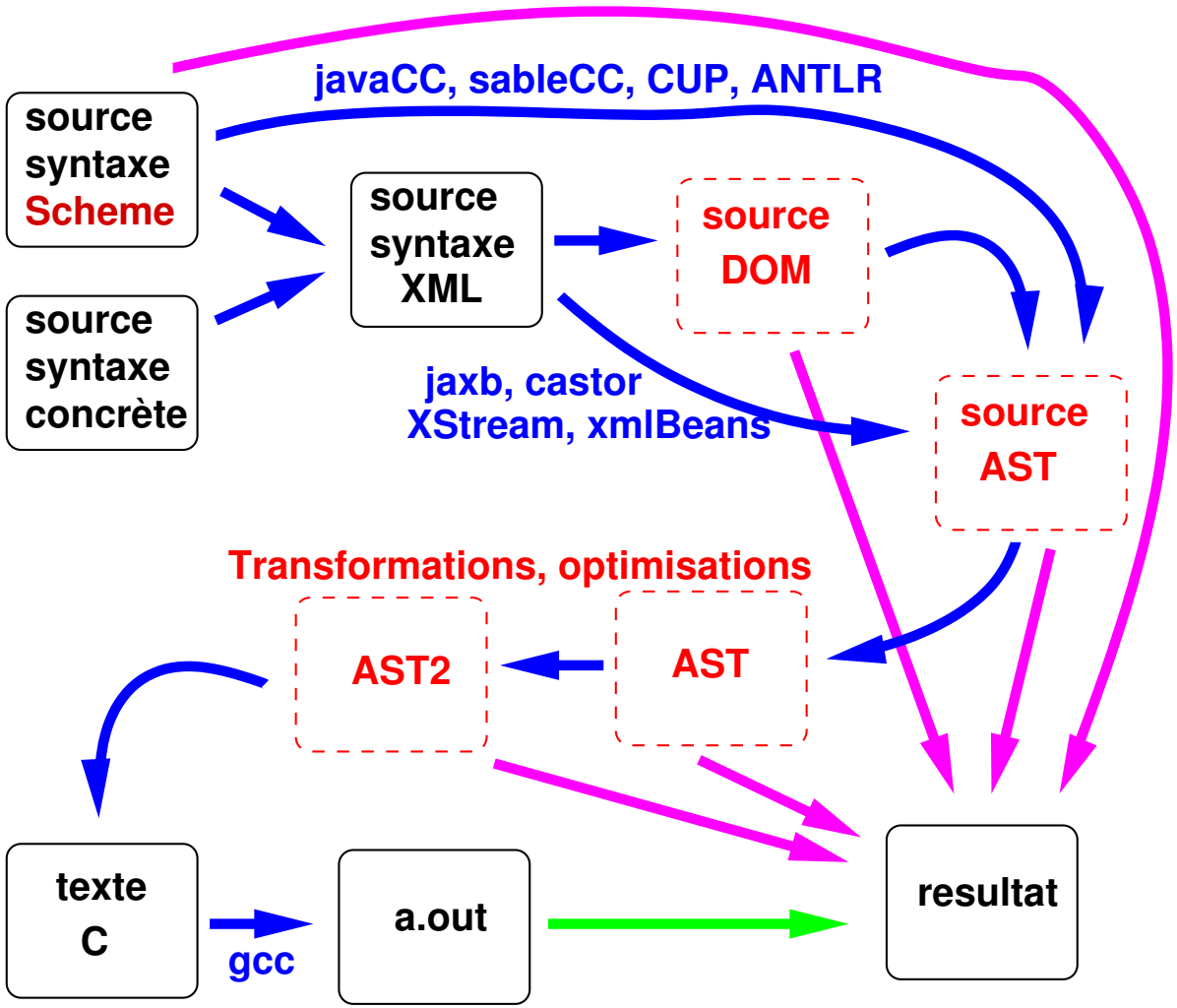
Il y a de plus en plus un écart entre l'expressivité des langages des haut niveau et celle du langage machine - le compilateur doit s'occuper de cette fracture.

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

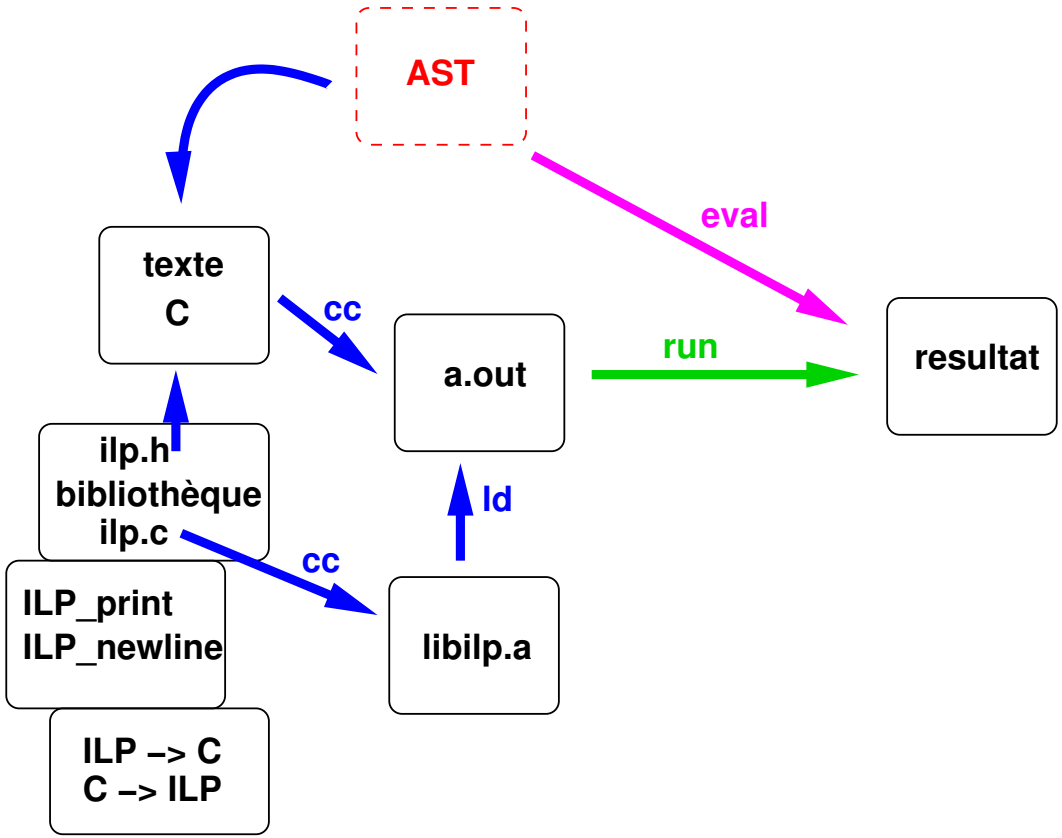
On souhaite étendre ILP2 avec l'introduction des fonctions auxiliaires *avant* et *après*. Lors de la définition d'une fonction vous pouvez ajouter de manière optionnelle un qualificateur `:before` ou `:after`, comme l'illustre le programme suivant :

```
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.

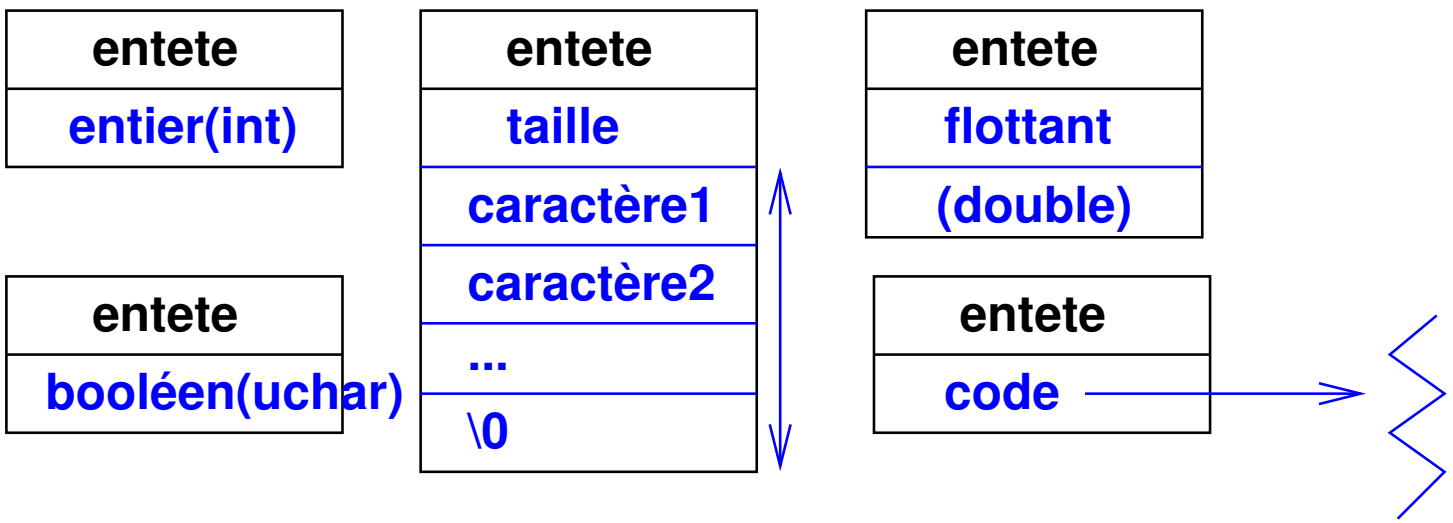


Représentation des valeurs

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 :

```
#define ILP_TRUE  (&ILP_object_true)
#define ILP_FALSE (&ILP_object_false)


#define ILP_Boolean2ILP(b) \
  ILP_make_boolean(b)

#define ILP_isBoolean(o) \
  ((o)->_class == &ILP_object_Boolean_class)

#define ILP_isTrue(o) \
  (((o)->_class == &ILP_object_Boolean_class) && \
   ((o)->_content.asBoolean))

#define ILP_isEquivalentToTrue(o) \
  ((o) != ILP_FALSE)

#define ILP_CheckIfBoolean(o) \
  if ( ! ILP_isBoolean(o) ) { \
       ILP_domain_error("Not a boolean", o); \
  };
```

Implantation

```
struct ILP_Object ILP_object_true = {
      &ILP_object_Boolean_class,
      { ILP_BOOLEAN_TRUE_VALUE }
};

struct ILP_Object ILP_object_false = {
      &ILP_object_Boolean_class,
      { ILP_BOOLEAN_FALSE_VALUE }
};

ILP_Object
ILP_make_boolean (int b)
{
      if ( b ) {
            return ILP_TRUE;
      } else {
            return ILP_FALSE;
      }
}
```

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); \
  };
```

```
#define ILP_Plus(o1,o2) \
  ILP_make_addition(o1, o2)

#define ILP_Minus(o1,o2) \
  ILP_make_subtraction(o1, o2)

#define ILP_Times(o1,o2) \
  ILP_make_multiplication(o1, o2)

#define ILP_Divide(o1,o2) \
  ILP_make_division(o1, o2)

#define ILP_Modulo(o1,o2) \
  ILP_make_modulo(o1, o2)

#define ILP_LessThan(o1,o2) \\end{lstlisting}
```

Allocation de la mémoire

```
#ifdef WITH_GC
    /* If Boehm's GC is present: */
#   include "include/gc.h"
#   define ILP_START_GC GC_init()
#   define ILP_MALLOC GC_malloc
#else
#   define ILP_START_GC
#   define ILP_MALLOC malloc
#endif

ILP_Object
ILP_malloc (int size, ILP_Class class)
{
      ILP_Object result = ILP_MALLOC(size);
      if ( result == NULL ) {
            return ILP_die("Memory exhaustion");
      };
      result->_class = class;
```

```
ILP_Object
ILP_make_addition (ILP_Object o1, ILP_Object o2)
{
  if ( ILP_isInteger(o1) ) {
    if ( ILP_isInteger(o2) ) {
      ILP_Object result = ILP_AllocateInteger();
      result->_content.asInteger =
          o1->_content.asInteger
        + o2->_content.asInteger;
      return result;
    } else if ( ILP_isFloat(o2) ) {
      ILP_Object result = ILP_AllocateFloat();
      result->_content.asFloat =
          o1->_content.asInteger
        + o2->_content.asFloat;
      return result;
    } else {
      return ILP_domain_error("Not a number", o2);
    }
    ...
```

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

```
ILP_Object
ILP_print (ILP_Object self)
{
if ( self->_class == &ILP_object_Integer_class ) {
 fprintf(stdout, "%d", self->_content.asInteger);
} else if (self->_class == &ILP_object_Float_class ) {
 fprintf(stdout, "%12.5g", self->_content.asFloat);
} else if (self->_class == &ILP_object_Boolean_class ) {
 fprintf(stdout, "%s", (ILP_isTrue(self)?"true":"false"));
} else if (self->_class == &ILP_object_String_class ) {
 fprintf(stdout, "%s", self->_content.asString.asCharacter
} else if ...
 }
return ILP_FALSE;
}
```

Mise en œuvre du compilateur

Ressource: ilp1.compiler

```
public class Compiler
implements IASTCvisitor<Void, Compiler.Context, CompilationException> {

    public Compiler (IOperatorEnvironment ioe,
                     IGlobalVariableEnvironment igve ) {
        this.operatorEnvironment = ioe;
        this.globalVariableEnvironment = igve;
    }
    protected final IOperatorEnvironment operatorEnvironment;
    protected final IGlobalVariableEnvironment globalVariableEnvironment;

//

    public String compile(IASTprogram program)
            throws CompilationException {
        ...
        IASTCprogram newprogram = normalize(program);
        newprogram = optimizer.transform(newprogram);
        ...
        Context context = new Context(NoDestination.NO_DESTINATION);
        StringWriter sw = new StringWriter();
        try {
            out = new BufferedWriter(sw);
            visit(newprogram, context);
            out.flush();
        } catch (IOException exc) {
            throw new CompilationException(exc);
        }
        return sw.toString();
    }
```

Mise en œuvre du compilateur

Ressource: ilp1.compiler

```
public static class Context {
    public Context (IDestination destination) {
        this.destination = destination;
    }
    public IDestination destination;
    public static AtomicInteger counter = new AtomicInteger(0);

    public IASTvariable newTemporaryVariable () {
        int i = counter.incrementAndGet();
        return new ASTvariable("ilptmp" + i);
    }

    public Context redirect (IDestination d) {
        if ( d == destination ) {
            return this;
        } else {
            return new Context(d);
        }
    }
}
```

Une optimisation

L'optimisation reine : 20% d'amélioration.
Transformation de programme. On remplace des appels de fonctions par leur corps, après substitution de leurs variables par les paramètres d'appel.

```
function f (x y) {
    let t = x + y
    in 2*t;
}

let z = 34 in
let y = f(1, z) in ...
```

```
let z = 34 in
let y = ({ let t = 1 + z;
           in 2*t;
         }) in ...
```

Intérêts

- Rapprocher des fragments de code indépendants (surtout avec précalculs statiques (*constant folding*) et suppression du code mort (*dead code elimination*).

```
function f(x, y) {
  if (x > 1 ) {
    x + y
  } else {
    x
  }
}
let z = 2 in
  print(3 + f(z, t));
```

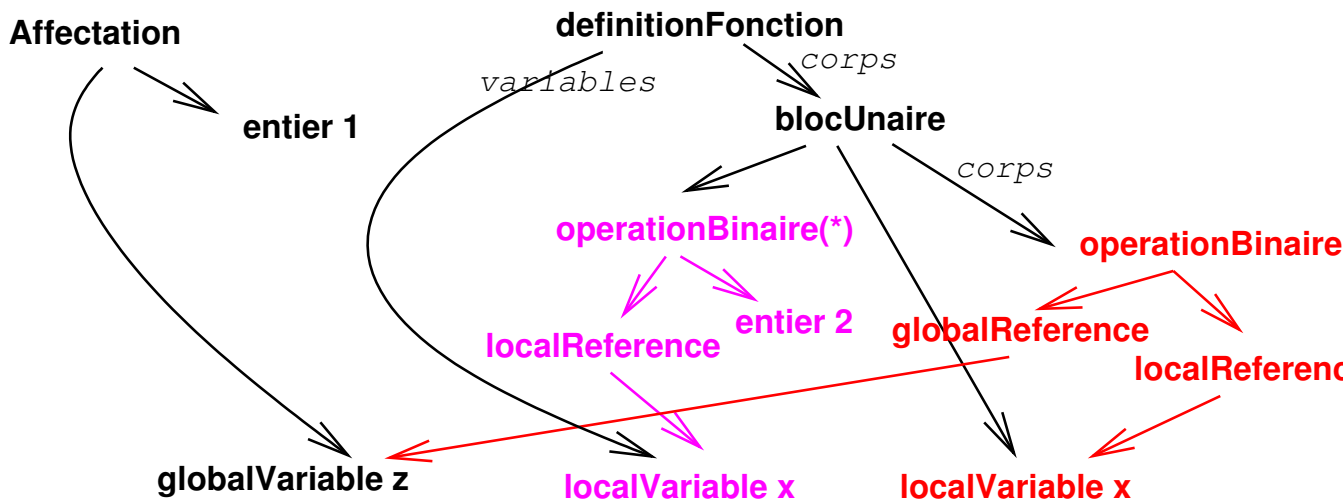
```
print(3 + (2 + t));
```

Mais attention aux variables ...

Normalisation

Partage physique des objets représentant les variables.
Taxonomie des variables locales, globales, globales fonctionnelles, prédéfinies.

```
z = 1;
function f(x) {
  let x = 2*x
  in z+x
}
```



L'identification des variables :

- améliore la comparaison (et notamment la vitesse de l'interprète)
- réalise l'alpha-conversion (l'adresse est le nom).

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