

PLAN DU COURS 8

Objets, classes et envoi de message

- ☐ Préliminaires
- ☐ Syntaxe
- ☐ Interprétation
- ☐ Compilation

Master d'informatique 2007-2008

Spécialité STL

« Implantation de langages »

ILP – MI016

Cours 8

C. Queinne^a

^a<http://www-spi.ilp6.fr/~queinnec/>

PAQUETAGE

super-paquetage *fr.upmc.ilp.ilp6* avec les paquetages usuels :

```
fr.upmc.ilp.ilp6.interfaces // 6 classes
.ast // 12 classes
.runtime // 5 classes
.cgen // 4 classes
```

Des ajouts à la grammaire précédente *Grammars/grammar6.mc*
et des programmes de tests en *Grammars/Samples/*-6.xml*

CARACTÉRISTIQUES

- ☐ classe
 - définition (globale)
 - héritage simple (champs et comportements)
 - allocation (sans initialisation) d'objets
- ☐ accès aux champs
- ☐ méthode
 - appartenance à une classe
 - moi
 - méthode héritée
 - super
- ☐ appel de méthode

```
creationObjet = element creationObjet { # new Classe(arguments)
  attribue classe { xsd:Name },
  expression *
}
lectureChamp = element lectureChamp { # objet.champ
  attribue champ { xsd:Name },
  element cible { expression }
}
ecritureChamp = element ecritureChamp { # objet.champ = expression
  attribue champ { xsd:Name },
  element cible { expression },
  element valeur { expression }
}
envoiMessage = element envoiMessage { # receveur.message(arguments)
  attribue message { xsd:Name },
  element receveur { expression },
  element arguments { expression } *
}
appelSuper = element appelSuper { # super()
  empty
}
```

SYNTAXE

5

On s'écarte de JavaScript qui est un langage à prototypes.

```
definitionClasse = element definitionClasse {
  attribue nom { xsd:Name },
  attribue parent { xsd:Name } ?,
  element champs {
    element champ {
      attribue nom { xsd:Name }
    } *
  } ?,
  element methodes {
    element methode {
      attribue nom { xsd:Name },
      element variables { variable * },
      element corps { expression + }
    } *
  } ?
}
```

8

EXEMPLES SYNTAXIQUES

class Point { // *extends Object // Ordre des définitions*

```
  field x; field y;
  method setXlike (p) { this.x = p.x; }
  method voir () { return "x=" + double(this)/2 + ", y=" + this.y; }
}

class ColoredPoint extends Point {
  field color; // pas de valeur d'initialisation
  method getColor () { return this.color; }
  method voir () { return this.color + super(); }
}

function double (x) { return x + x; }
let p1 = new ColoredPoint(11, 22, "blue")
and p2 = new Point(1, 9) in
  print(p1.voir());
  p2.setXlike(p1.truc());
}
```

```
}
moi = element moi {
  empty
}

#this
```

10

AST

```
// paquetage fr.upmc.ilp.ilp6.ast
CEAST6
CEAST6classDefinition // implante IAST6classDefinition
CEAST6methodDefinition // implante IAST6methodDefinition
CEAST6program // étend ilp4.ast.CEAST6program
CEAST6expression // étend ilp4.ast.CEAST6expression
CEAST6instantiate
CEAST6readField
CEAST6writeField
CEAST6self
CEAST6super
CEAST6send
```

○ accès aux champs

statique

$p.x$ doit vérifier que x est un champ de l'objet p
 $p["x"]$ interdit

Attention : pour simplifier en ILP6, deux classes non reliées par héritage ne peuvent avoir un champ ou une méthode homonyme ! Si C_1 et C_2 ont un champ ou une méthode de même nom alors $C_1 \subseteq C_2 \vee C_2 \subseteq C_1$.

9

INTERFACES

```
fr.upmc.ilp.ilp6.interfaces.IAST6classDefinition
IAST6methodDefinition
IAST6instantiation
IAST6readField
IAST6writeField
IAST6super
IAST6self
IAST6program
IAST6visitor
```

11

STATIQUE / DYNAMIQUE

Les choix d'implantation :

○ classe

▸ définition (globale)

statique

▸ héritage simple (champs et comportements)

statique

▸ allocation d'objets

statique

▸ super

statique

`new Point(1, 2)`

permis

`o.getClass().newInstance(1, 2)`

interdit

▸ pas de comportement réflexif.

MODÈLE DE DONNÉES

nom -> Classe *interprétation ou compilation*

- Classe
 - nom
 - nom superclasse
 - champs hérités
 - méthodes héritées
 - champs propres (liste de noms)
 - méthodes propres (liste de fonctions globales)
- Instance
 - classe
 - champs (nom -> valeur)
- Méthode
 - fonction globale
 - nom
 - variables (liste)
 - corps

○ méthode

▸ appartenance à une classe

▸ moi

▸ méthode héritée

// *La méthode voir de ColoredPoint*

method voir () {

return this.color //this *est un ColoredPoint au sens large*

+ super() ; //la méthode voir de Point

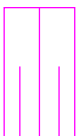
}

○ appel de méthode

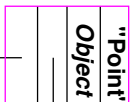
dynamique

un Point la classe Point

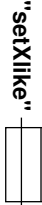
ILPInstance



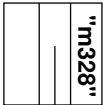
ILPClass



"voir"



UserGlobalFunction



Variable



INTERPRÉTATION

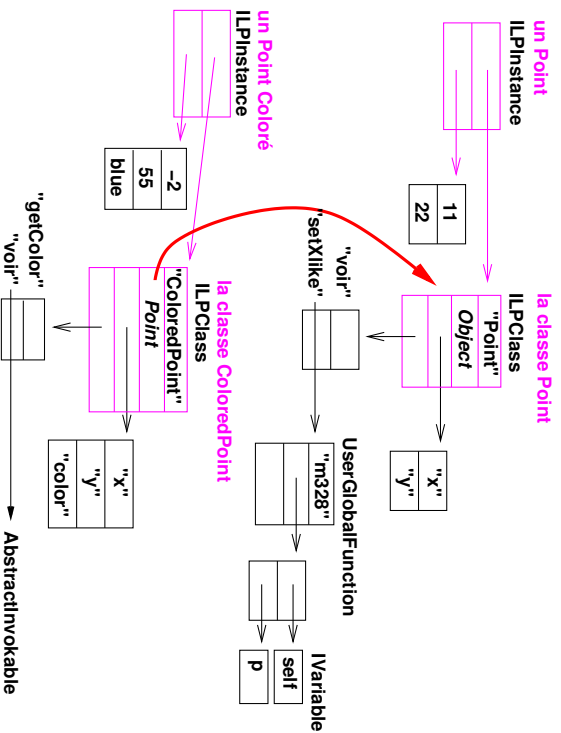
Les classes sont des ressources globales donc stockées dans l'environnement global *common*.

Les classes sont représentées par des instances d'ILPClass

```
String className
ILPClass superClass
String[] fieldName
Map method
```

*propres et hérités
propres seulement*

```
String getName()
String fieldName(int)
int fieldSize()
int fieldRank(fieldName)
Object send(self, message, argument[], common)
```



Les méthodes sont représentées par des `UserGlobalFunction` dont la première variable est `ilpSelf`. Par contre la définition des méthodes est l'apanage de `IASTmethodDefinition`.

`o.m(a, b)` \equiv `m_global(o, a, b)`

Les instances d'ILP sont représentées par des instances de `ILPInstance`

```
ILPClass clazz;
Object[] fields;

Object read(fieldName)
Object write(fieldName, object)
Object send(message, argument, common)
```

20

ALLOCATION

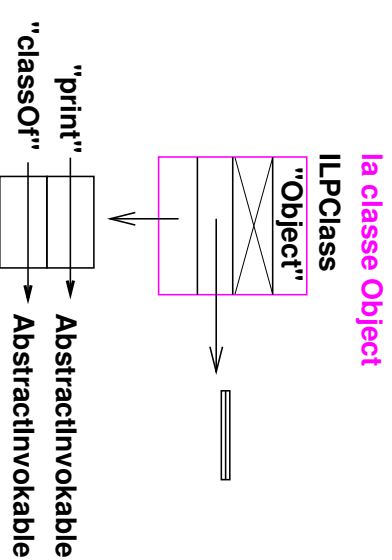
```
public class CEASTInstantiator extends CEASTExpression {
    public Object eval (final IlexicalEnvironment lexenv,
                       final ICommon common)
        throws EvaluationException {
        final ILPClass clazz = common.findClass(className);
        final Object[] value = new Object[argument.length];
        for ( int i = 0 ; i<argument.length ; i++ ) {
            value[i] = argument[i].eval(lexenv, common);
        }
        return new ILPInstance(clazz, value);
    }
}
```

19

DÉTAILS D'INTERPRÉTATION

La classe racine : `Object` qui pourrait avoir ses champs et méthodes propres. En fait, elle n'est pas représentée explicitement dans l'interprète.

`print(o)` \equiv `o.print()`



```
throws EvaluationException {
    return clazz.send(this, message, argument, common);
}
```

21

INSTANCE

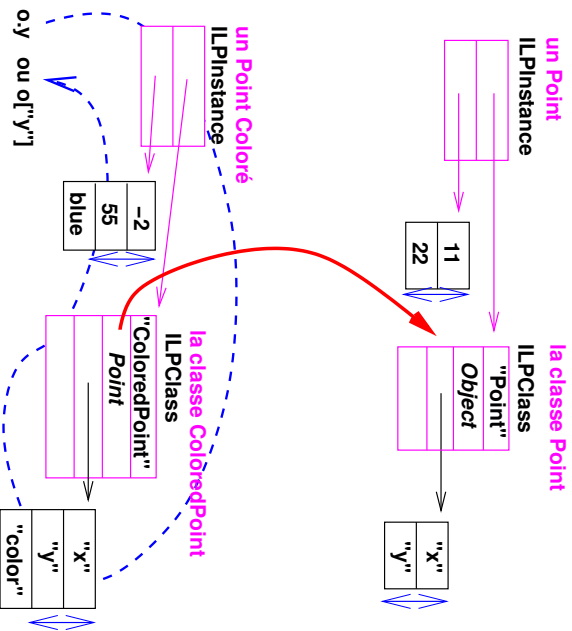
Les noms des champs sont convertis en des décalages.

```
public class ILPInstance {
    public ILPInstance (final ILPClass clazz, final Object[] argument) {
        this.clazz = clazz;
        this.field = argument;
    }
    private final ILPClass clazz;
    private final Object[] field;

    public Object read (final String fieldName)
        throws EvaluationException {
        return field[clazz.fieldRank(fieldName)];
    }

    public Object write (final String fieldName, final Object value)
        throws EvaluationException {
        field[clazz.fieldRank(fieldName)] = value;
        return Boolean.FALSE;
    }

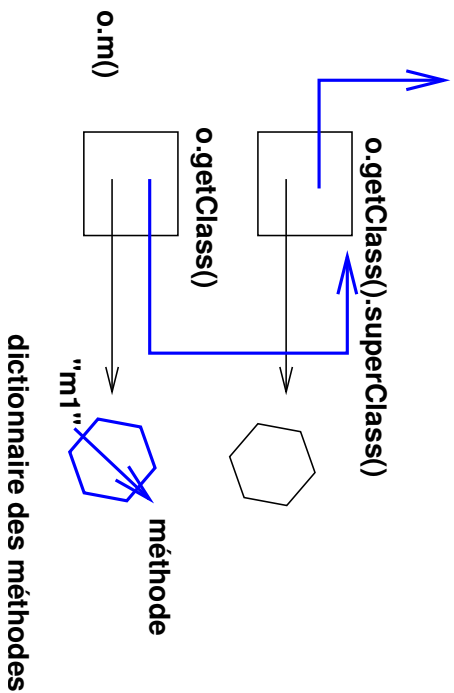
    public Object send (final String message,
        final Object[] argument,
        final ICommon common)
```



23

ACCÈS AUX CHAMPS

```
public class CEASTWriteField extends CEASTExpression {
    public Object eval (final IlexicalEnvironment lexenv,
        final ICommon common)
        throws EvaluationException {
        final Object target = object.eval(lexenv, common);
        final Object newValue = value.eval(lexenv, common);
        if ( target instanceof ILPInstance ) {
            return ((ILPInstance) target).write(fieldName, newValue);
        } else {
            throw new EvaluationException("Wrong class");
        }
    }
}
```



25

À la Smalltalk (*lookup*) :

```
public class CEASTsend extends CEASTExpression {
    public Object eval (final ILexicalEnvironment lexenv,
                      final ICommon common)
        throws EvaluationException {
    final Object target = receiver.eval(lexenv, common);
    final Object[] value = new Object[argument.length];
    for ( int i = 0 ; i<argument.length ; i++ ) {
        value[i] = argument[i].eval(lexenv, common);
    }
    if ( target instanceof ILPInstance ) {
        return ((ILPInstance) target).send(methodName, value, common);
    } else {
        throw new EvaluationException("No such method " + methodName);
    }
}
```

27

CLASSE

```
public class ILPClass {
    ...
    public String fieldName (int rank) {
        return this.fieldName[rank];
    }
    public int fieldSize () {
        return this.fieldName.length;
    }
    public int fieldRank (final String name)
        throws EvaluationException {
        for ( int i = 0 ; i<fieldName.length ; i++ ) {
            if ( fieldName[i].equals(name) ) {
                return i;
            }
        }
        throw new EvaluationException("No such field " + name);
    }
    public Object send (final ILPInstance self,
                       final String message,
                       final Object[] argument,
                       final ICommon common)
```

```
throws EvaluationException {
    UserGlobalFunction m = (UserGlobalFunction) this.method.get(message);
    if ( m != null ) {
        final Object[] newArgument = new Object[1 + argument.length];
        newArgument[0] = self;
        for ( int i = 0 ; i<argument.length ; i++ ) {
            newArgument[i+1] = argument[i];
        }
        return m.invoke(newArgument, common);
    } else {
        // Pas de méthode propre de ce nom!
        if ( superClass != null ) {
            return superClass.send(self, message, argument, common);
        } else {
            // On est sur Object
            throw new EvaluationException("No such method " + message);
        }
    }
}
```

```

    }
    // toutes les fonctions sous-jacentes aux methodes ont des noms differents.
    CEASTGlobalFunctionVariable gfv =
        new CEASTGlobalFunctionVariable("lpmethod_" + getCounter());
    this.delegate = new CEASTFunctionDefinition(
        gfv,
        varsPlusSelf,
        delegate.getBody() );
}
private final CEASTFunctionDefinition delegate;
private final IAST4Variable selfVariable;
private final String methodName;

```

MOI

C'est la valeur de la variable introduite en premier dans la liste des variables des méthodes.

```

// dans CEASTself
public Object eval6 (final ILexicalEnvironment lexenv,
                    final ICommon common)
    throws EvaluationException {
    return lexenv.lookup(this);
}

public class CEASTMethodDefinition
    extends CEAST6 implements IAST6MethodDefinition {
    public CEASTMethodDefinition (IAST4FunctionDefinition delegate) {
        this.methodName = delegate.getFunctionName();
        // Ajouter self en tete des variables:
        this.selfVariable = new CEASTself();
        IAST4Variable[] vars = delegate.getVariables();
        IAST4Variable[] varsPlusSelf = new IAST4Variable[1+vars.length];
        varsPlusSelf[0] = this.selfVariable;
        for ( int i=0 ; i<vars.length ; i++ ) {
            varsPlusSelf[i+1] = vars[i];
        }
    }
}

```

```

// Dans ILPmethod
public static IAST4LocalVariable cmv =
    new CEASTLocalVariable("ilp_CurrentMethod");
protected static IAST4LocalVariable cmargs =
    new CEASTLocalVariable("ilp_CurrentArguments");

```

```

public void setDefiningClass (ILPClass clazz) {
    this.definingClass = clazz;
}
private ILPClass definingClass;
public ILPClass getDefiningClass () {
    return this.definingClass;
}

```

```

public Object callSuper (final ILexicalEnvironment lexenv,
                        final ICommon common )
    throws EvaluationException {
    Object[] arguments = (Object[]) lexenv.lookup(cmargs);
    return getDefiningClass().getSuperClass()
        .send(getMethodName(), arguments, common);
}

```

SUPER

Différent de JavaScript et Java : **super()** désigne la valeur de la super-méthode invoquée avec les mêmes arguments que la méthode courante. Ainsi

```

class A extends B {
    method f (x) {
        return x
            + super() // en Java: super.f(x)
    }
}

```

La méthode invoquée est la méthode **f** valable en **B** : elle est connue dès la compilation car il n'y a pas de création dynamique de méthode.

```

// Dans CEASTsuper
public Object eval6 (final ILexicalEnvironment lexenv,
                    final ICommon common)
    throws EvaluationException {
    ILPmethod currentMethod = (ILPmethod) lexenv.lookup(ILPmethod.cmv);
    return currentMethod.callSuper(lexenv, common);
}

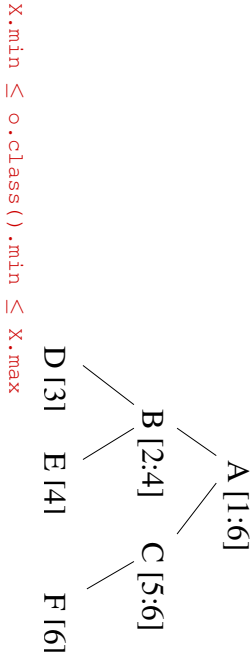
```


COMPILEATION

- ☐ pas d'objet en C
- ☐ pas de classe mais des **struct**
- ☐ accès aux champs statique
- ☐ déjà une solution pour **instanceof**
- ☐ pointeurs
- ☐ fonctions

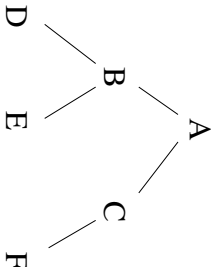
```
@Override
public Object invoke (final Object[] arguments,
                     final ICommon common)
    throws EvaluationException {
    IAST2variable[] variables = getVariables();
    if ( variables.length != arguments.length ) {
        final String msg = "Wrong arity";
        throw new EvaluationException(msg);
    };
    ILexicalEnvironment lexenv = getEnvironment()
        .extend(cmv, this)
        .extend(cmargs, arguments);
    for ( int i = 0 ; i<variables.length ; i++ ) {
        lexenv = lexenv.extend(variables[i], arguments[i]);
    }
    return getBody().eval(lexenv, common);
}
```

NUMÉROTATION PRÉFIXE

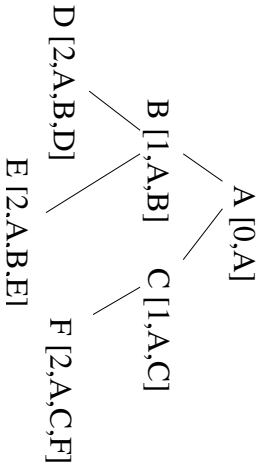


HÉRITAGE SIMPLE

class-of s'obtient en temps constant avec un schéma par en-tête. La question restante correspond à **instanceof**.

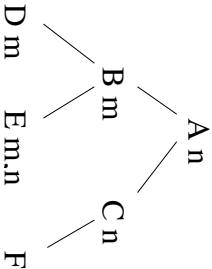


INDEXATION EN PROFONDEUR



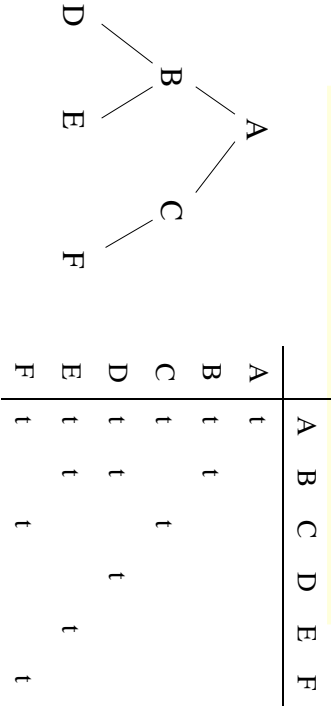
```
X.depth() <= o.class().depth()  
&& o.class().supers[X.depth()] = X
```

INVOCATION DE MÉTHODE



	m	n
A	error	nA
B	mB	nA
C	error	nC
D	mD	nA
E	mE	nE
F	error	nC

MATRICE CARACTÉRISTIQUE

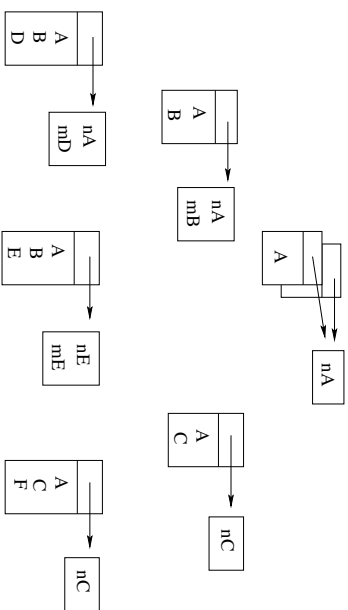


```
matrice[o.class().numero, X.numero]
```

PLACEMENT DES CHAMPS

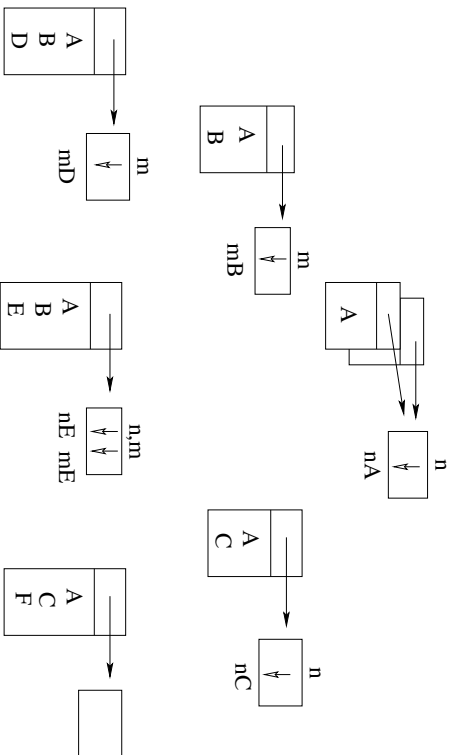
Invariant : Si $SC < C$ alors le contenu d'un SC débute par le contenu d'un C.
Ainsi tous les décalages sont conservés pour l'accès aux champs (dans le cas de l'héritage simple).

À LA JAVA

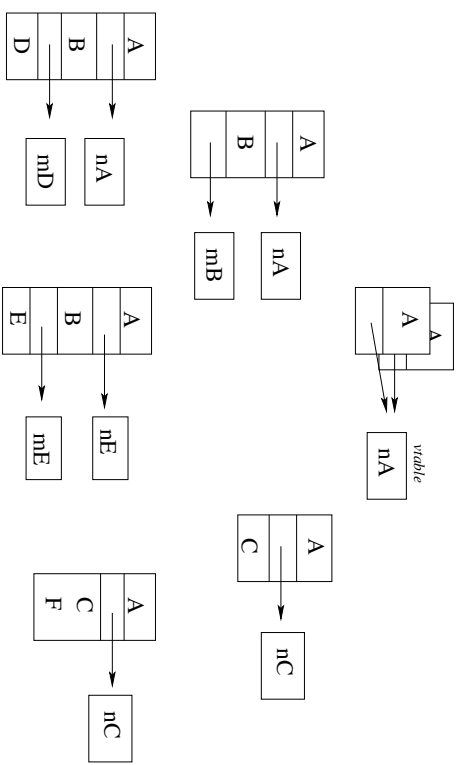


Si $SC < C$ alors les méthodes possibles sur C forment un préfixe des méthodes possibles sur SC (en héritage simple). Possible car l'ensemble des méthodes est connu statiquement.

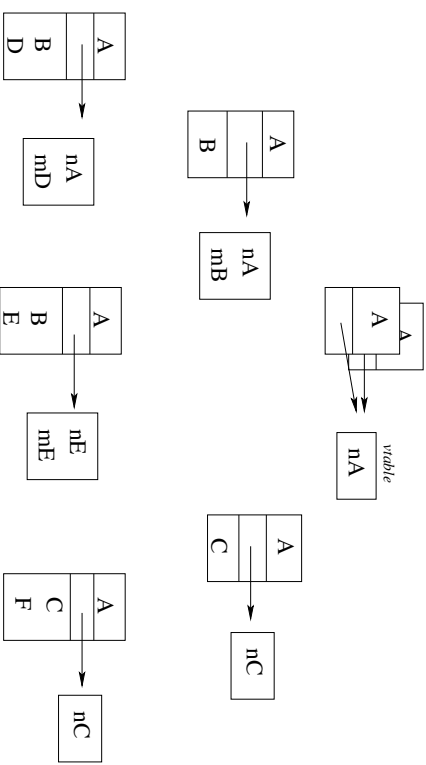
À LA SMALLTALK



ou encore



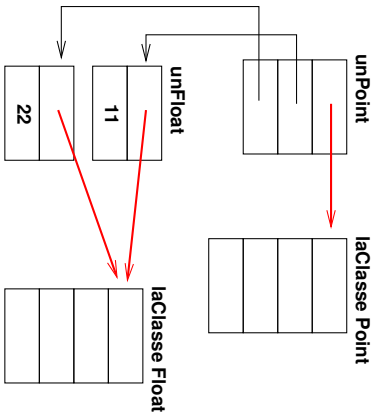
À LA C++



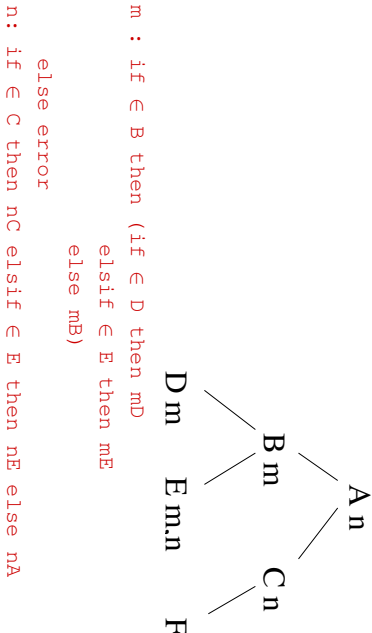
COMPILEATION

Ressource: C/itpObj.h

Nouvelle représentation des objets en C et matérialisation du lien d'héritage



ARBRE DE DÉCISION



```
ILP_general_function method[1];  
} asClass;  
struct asMethod {  
    char*      name;  
    short      arity;  
    short      index;  
} asMethod;  
struct asField {  
    struct ILP_Class*  defining_class;  
    struct ILP_Field*  previous_field;  
    char*              name;  
    short              offset;  
} asField;  
struct asInstance {  
    struct ILP_Object*  field[1];  
} asInstance;  
} *ILP_Object;  
  
#define ILP_AllocateInteger() \  

```

```
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 asException {  
            char      message[ILP_EXCEPTION_BUFFER_LENGTH];  
            culprit[ILP_EXCEPTION_CULPRIT_LENGTH];  
        } asException;  
    } asClass {  
        struct ILP_Class*  super;  
        char*              name;  
        int                fields_count;  
        struct ILP_Field*  last_field;  
        int                methods_count;  
    }  
}
```


ACCÈS AUX CHAMPS

Compiler 0.x

x donne (typage ou notre hypothèse additionnelle) **Point** qui donne le décalage.

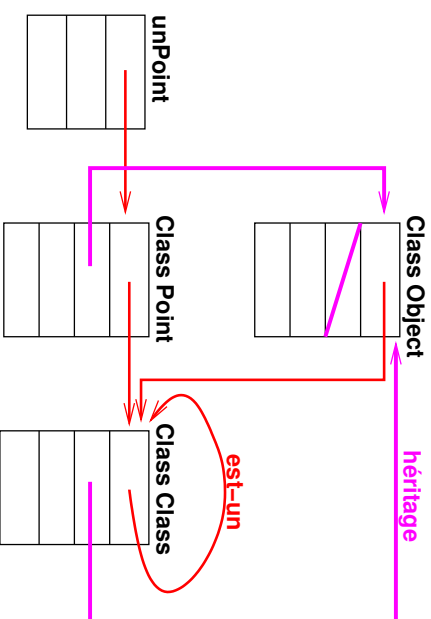
Efficacité ! On précalcule les décalages pour accéder aux champs mais on vérifie leur existence.

```

        if (ILP_IsA(o, (ILP_Class) & ILP_Object_Point_class)) {
            destination o->content.asinstance.field[0];
        } else {
            destination ILP_UnknownFieldError("x", o);
        }
    }
}

```

MODÈLE OBJVLISP



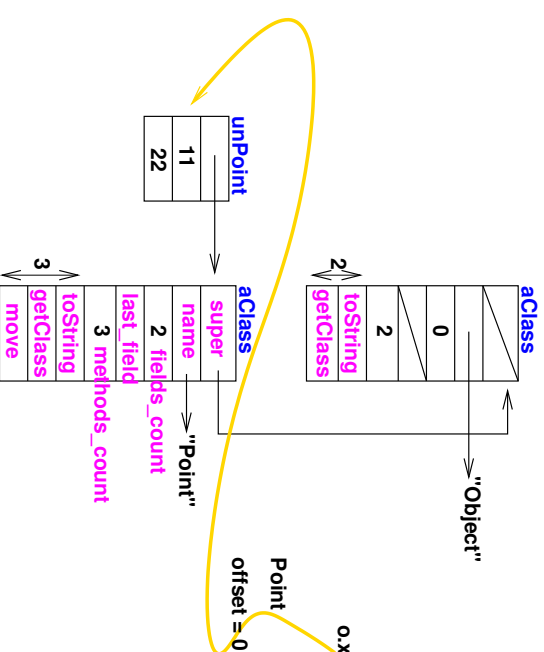
LIEN D'HÉRITAGE

On peut faire plus efficace que cette recherche ascendante linéaire :

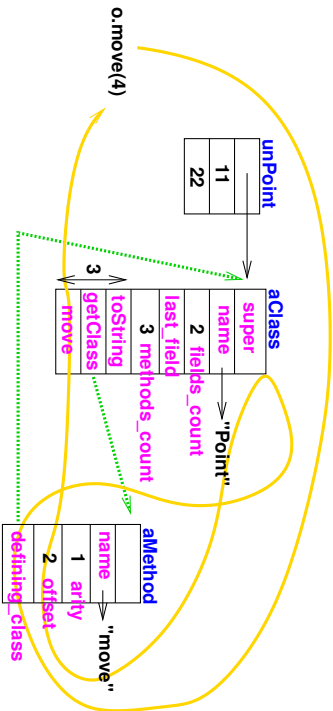
```

int /* boolean */
ILP_is_a (ILP_Object o, ILP_Class class)
{
    ILP_Class oclass = o->_class;
    if ( oclass == class ) {
        return 1;
    } else {
        oclass = oclass->_content.asClass.super;
        /* Object a NULL pour superclasse. */
        while ( oclass ) {
            if ( oclass == class ) {
                return 1;
            }
        }
        oclass = oclass->_content.asClass.super;
    }
    return 0;
}

```



ENVOI DE MÉTHODES



```
ilpTMP808 = ilp_find_method(receveur,
                             &ilp_object_move_method,
                             1);
destination ilpTMP808 (receveur, argument);
```

```
}
}
```

COMPILATION DE CLASSE

```
ilp_GenerateClass(4);
extern struct ilp_Class4 ilp_object_Point_class;
extern struct ilp_Field ilp_object_x_field;
extern struct ilp_Field ilp_object_y_field;
extern struct ilp_Method ilp_object_longueur_method;
extern struct ilp_Method ilp_object_move_method;
```

```
struct ilp_Field ilp_object_x_field = {
    &ilp_object_Field_class,
    {(ilp_Class) & ilp_object_Point_class,
     NULL,
     "x",
     0}
};
```

```
struct ilp_Field ilp_object_y_field = {
    &ilp_object_Field_class,
    {(ilp_Class) & ilp_object_Point_class,
```

```
ilp_general_function
ilp_find_method (ilp_Object receiver,
                 ilp_Method method,
                 int argc)
{
    ilp_Class oclass = receiver->_class;
    if ( ! ilp_is_subclass_of(oclass,
                             method->_content.asMethod.class_defining) ) {
        Signaler une absence de méthode
    } else {
        int index = method->_content.asMethod.index;
        if ( argc != method->_content.asMethod.arity ) {
            Signaler une erreur d'arité
        }
        return oclass->_content.asClass.method[index];
    }
}
```

```
        &ILP_object_Method_class,
        {{(struct ILP_Class *) &ILP_object_Point_class,
            "longueur",
            1,
            2
        }}
    };

    struct ILP_Method ILP_object_move_method = {
        &ILP_object_Method_class,
        {{(struct ILP_Class *) &ILP_object_Point_class,
            "move",
            3,
            3
        }}
    };
};
```

```
        &ILP_object_x_field,
        "y",
        1}}
    };
};
```

```
    struct ILP_Class4 ILP_object_Point_class = {
        &ILP_object_Class_class,
        {{(ILP_Class) & ILP_object_Object_class,
            "Point",
            2,
            &ILP_object_y_field,
            4,
            {ILP_print,
              ILP_classOf,
              ilpFUNC234,
              ilpFUNC235,
            }}}
    };
};
```

```
    struct ILP_Method ILP_object_longueur_method = {
```

```
        if ( currentMethod != null ) {
            destination.compile(buffer, lexenv, common);
            buffer.append("ILP_FindAndCallSuperMethod(");
            buffer.append(currentMethod.getRealAriety());
            buffer.append(");");
        } else {
            final String msg = "No supermethod!";
            throw new CgenerationException(msg);
        }
    }
};
```

63

SUPER

On précalcule (statiquement) quelle est la super-méthode. La compilation d'une méthode stocke la méthode courante dans l'environnement lexical.

```
// CEASTsuper
public void compile6 (final StringBuffer buffer,
    final ICgenLexicalEnvironment lexenv,
    final ICgenEnvironment common,
    final IDestination destination)
    throws CgenerationException {
    IAST6methodDefinition currentMethod = null;
    ICgenLexicalEnvironment le = lexenv;
    while ( ! le.isEmpty() ) {
        if ( le instanceof CgenMethodLexicalEnvironment ) {
            CgenMethodLexicalEnvironment cmle =
                (CgenMethodLexicalEnvironment) le;
            currentMethod = cmle.getMethodDefinition();
            break;
        } else {
            le = le.getNext();
        }
    }
};
```


Ajout à la bibliothèque d'exécution en C :

```
#define ILP_FindAndCallSuperMethod(i) \
(((ILP_SuperMethod != NULL) \
 ? (*ILP_find_and_call_super_method##i) \
 : (*ILP_dont_call_super_method) ) ( \
    ilp_self, ilp_CurrentMethod, ilp_SuperMethod, ilp_CurrentArguments))
extern ILP_Object ILP_find_and_call_super_method(
    ILP_Object self,
    ILP_Method current_method,
    ILP_general_function super_method,
    ILP_Object arguments[] );
#define DefinesSuperMethodCaller(i) \
ILP_Object \
ILP_find_and_call_super_method##i ( \
    ILP_Object self, \
    ILP_Method current_method, \
    ILP_general_function super_method, \
    ILP_Object arguments[] ) \
{ \
```

Un petit exemple :

```
/* class Point2D extends Point */
/* method print (x) { print "print@Point2D"; super() } */
ILP_Object
ilpMETHOD_5f80_3222(ILP_Object ilp_self, ILP_Object x)
{
    static ILP_Method ilp_CurrentMethod = &ilp_object_print_method;
    static ILP_general_function ilp_SuperMethod = ILP_print;
    ILP_Object ilp_CurrentArguments[2];
    ilp_CurrentArguments[1] = x;

    ILP_Object ilpLOCAL_3236;
    ilpLOCAL_3236 = ILP_String2ILP("print@Point2D");
    (void) ILP_print(ilpLOCAL_3236);
    return ILP_FindAndCallSuperMethod(1);
}
```

```
current_method->_content.asMethod.name, \
(void*) self ); \
/*DEBUG*/ \
fprintf(stderr, ILP_the_exception._content.asException.message); \
ILP_the_exception._content.asException.culprit[0] = self; \
ILP_the_exception._content.asException.culprit[1] = \
    (ILP_Object) current_method; \
ILP_the_exception._content.asException.culprit[2] = NULL; \
ILP_throw((ILP_Object) &ILP_the_exception); \
/* UNREACHED */ \
return NULL; \
} \
} \
}
DefinesSuperMethodCaller(0)
DefinesSuperMethodCaller(1)
DefinesSuperMethodCaller(2)
DefinesSuperMethodCaller(3)
ILP_Object
```

```
/* assert( super_method != NULL ); */ \
switch ( i ) { \
    case 0: { \
        return (*super_method)(self); \
    } \
    case 1: { \
        return (*super_method)(self, arguments[1]); \
    } \
    case 2: { \
        return (*super_method)(self, arguments[1], arguments[2]); \
    } \
    case 3: { \
        return (*super_method)(self, arguments[1], \
                                arguments[2], \
                                arguments[3]); \
    } \
    default: { \
        sprintf(ILP_the_exception._content.asException.message, \
                ILP_EXCEPTION_BUFFER_LENGTH, \
                "Cannot invoke supermethod %s\nCulprit: 0x%p\n", \
```

```
        return NULL;
    }
}
```

```
ILP_dont_call_super_method (
    ILP_Object self,
    ILP_Method current_method,
    ILP_general_function super_method,
    ILP_Object arguments[] )
{
    /* assert ( super_method == NULL ); */
    snprintf(ILP_the_exception._content.asException.message,
             ILP_EXCEPTION_BUFFER_LENGTH,
             "No supermethod %s\nCulprit: 0x%p\n",
             current_method->_content.asMethod.name,
             (void*) self );

    /*DEBUG*/
    fprintf(stderr, ILP_the_exception._content.asException.message);
    ILP_the_exception._content.asException.culprit[0] = self;
    ILP_the_exception._content.asException.culprit[1] =
        (ILP_Object) current_method;
    ILP_the_exception._content.asException.culprit[2] = NULL;
    ILP_throw((ILP_Object) &ILP_the_exception);
    /* UNREACHED */
}
```

72

MÉTHODES COMPILÉ6, EVAL6

```
// paquetage fr.upmc.ilp.ilp6.ast
CEAST6
CEAST6classDefinition // implante IAST6classDefinition
CEAST6methodDefinition // implante IAST6methodDefinition
CEAST6program // étend ilp4.ast.CEAST6program
CEAST6expression // étend ilp4.ast.CEAST6expression
CEAST6instantiate
CEAST6readfield
CEAST6writefield
CEAST6self
CEAST6super
CEAST6send
```

Toutes ces classes ont besoin de lire ou d'enrichir l'environnement des classes/méthodes. Or les **I*Environment** doivent être enrichis pour contenir cette nouvelle information en implantant :

71

NOUVEAUTÉS JAVA

- extension des environnements
- extension du visiteur

```
// dans CEAST6expression
@Override
public void compile (final StringBuffer buffer,
                    final ICGenLexicalEnvironment lexenv,
                    final fr.upmc.ilp.ilp2.interfaces.ICGenEnvironment common,
                    final IDestination destination)
    throws CGenerationException {
    compile6(buffer,
            lexenv,
            CEAST6.narrowToICGenEnvironment(common),
            destination );
}

// NOTE : on aurait pu prendre le même nom et laisser faire la surcharge :
public abstract void compile6 (final StringBuffer buffer,
                              final ICGenLexicalEnvironment lexenv, /*ilp6*/
                              final ICGenEnvironment common,
                              final IDestination destination )
    throws CGenerationException;
```

```
public interface IClassEnvironment {
    public void addClassDefinition (IAST6classDefinition cd);
    public IAST6classDefinition findClassDefinition (String className)
        throws RuntimeException;
}
```

Sont enrichis : **ICGenEnvironment**, **INormalizeGlobalEnvironment** et leurs implantations. Mais le code d'ILP4 ne sait pas fournir ces environnements enrichis aux nouvelles expressions d'ILP6.

Il faut alors écrire :

```
public void accept (IAST6visitor visitor) {
    visitor.visit(this);
}

public void accept (IAST4visitor visitor) {
    CEAST6.narrowToIAST6visitor(visitor).visit(this);
}
```

Non nécessaire si l'on écrit son discriminant (comme en `ilp1.cgen.analyse`)

75

VISITEUR

Il y a de nouvelles expressions :

```
public interface IAST6visitor extends IAST4visitor {
    void visit (IAST6classDefinition classDefinition);
    void visit (CEASTinstantiated expression);
    void visit (CEASTsend expression);
    void visit (CEASTreadfield expression);
    void visit (CEASTwritefield expression);
    void visit (CEASTself expression);
    void visit (CEASTsuper expression);
    // NOTE: utiliser plutôt des interfaces.
}
```

On peut, dans ILP4, fournir un IAST6visitor à la place d'un IAST4visitor mais quand ILP4 visite un nœud ILP6, il doit lui fournir un IAST6visitor.

On ne peut donc simplement écrire, dans une classe d'ILP6 :

```
//public void accept (IAST4visitor visitor) { // FAUX
//    visitor.visit(this); // FAUX
//}
```

SUGGESTIONS

- ☐ lire le code d'ILP6
- ☐ et regarder le code C engendré