# Projet « Concepts de programmation »

soutenance dans la semaine du 13 juin

Année 2004-2005

## 1 Présentation

Ce projet a pour but d'écrire quelques fonctions d'analyse et de transformation d'un mini langage impératif. Le code transformé sera affiché sous la forme d'un graphe représentant le flot de contrôle du programme. Deux analyses seront considérées: les variables vivantes et les expressions très utilisées.

- Variables vivantes: une variable vivante est une variable dont la valeur peut avoir une incidence sur le résultat du programme. A chaque état d'un programme est attaché un ensemble de variables dont la valeur peut avoir une influence sur le résultat final. Dans l'exemple suivant est attaché entre chaque instruction l'ensemble des variables vivantes en ce point:

  ```
  -- {y, z}
  x := x + 2;
  -- {y, z}
  x := y;
  -- {x, z}
  y := y + 1;
  -- {x, z}
  return x + z;
  -- { }
  ```

  L'analyse des variables vivantes (décrite au paragraphe 2.2.1) est exploitée pour simplifier le programme: l'affectation à une variable qui n'est pas vivante après l'instruction est inutile. Ainsi, le code précédent peut être simplifié en:

  ```
  x := y;
  return x + z;
  ```

- Expressions très utilisées: une expression est dite *expression très utilisée* si elle est utilisée dans tous les chemins d'exécution du programme, et non modifiée le long de ces chemins. Par exemple, dans le code suivant, `x + y` est très utilisée (utilisée dans les deux branches du `if`), tandis que `u + z` ne l'est pas, parce que `u + z` n'apparaît pas sur le chemin d'exécution qui ne traverse pas le corps de la boucle. Encore une fois, entre accolades, les résultats de l'analyse à ce point du graphe de flot de contrôle:

  ```
  { x+y, u>z }
  while (u > z) {
    { x+y, u+z, u+z > 1 }
     if (u + z > 1) {
       { x+y, u+z, (x+y) - (u+z) }
       u := (x + y) - (u + z)
  ```

```
      { x+y, u>z } }
    else {
      { x+y, u+z, (x+y) + (u+z), u>z }
      v := (x + y) + (u + z)
      { x+y, u>z } }
  { x+y, u>z } }
{ x+y }
return(x + y);
```

L'analyse des expressions très utilisées est décrite dans le paragraphe 2.2.2.

Une optimisation (non demandée dans ce projet !) s'appuyant sur cette analyse consiste à n'évaluer une expression très utilisée qu'une seule fois en introduisant des variables auxiliaires, si en plus on sait que l'expression est constante:

```
  aux := (x + y);
  while (u > z) {
    if (u + z > 1) {
      u := aux - (u + z);
    else {
      v := aux + (u + z);
    }
  }
return aux;
```

Le projet est découpé en deux parties: la partie analyse et transformation réalisée en Java et en Caml, et la partie affichage réalisée en C. Le module C devra être conçu pour faciliter son invocation depuis les deux langages.

# 2 Parties Caml et Java

Dans cette partie, il s'agit d'implanter l'analyse décrite dans la Section 1. La démarche à suivre sera la même pour les deux langages, Caml et Java. Ceci permettra de comparer les différents styles de programmation.

Les étapes sont les suivantes, expliquées en détail plus bas:

1. Définition d'une syntaxe abstraite pour représenter notre mini-langage.
2. Implantation des analyses: variables vivantes et expressions très utilisées. On verra que les analyses sont des instances d'un schéma général.
3. Implantation de l'optimisation de programme basée sur l'analyse des variables vivantes.

## 2.1 Syntaxe abstraite

Par *syntaxe abstraite*, on entend une représentation de la syntaxe d'un programme ou d'une expression qui ne contient que les éléments nécessaires pour comprendre sa structure, mais qui omet tous les éléments dont le seul but est d'augmenter la lisibilité (la *syntaxe concrète*).

Ainsi, les opérateurs binaires sont représentés en notation préfixe, leur priorité est rendue explicite: Par exemple, $e_1 + e_2 * e_3$ devient `Add(`$e_1$`, Mult(`$e_2$`, `$e_3$`))` en Caml. L'expression $(e_1 + (e_2 * e_3))$ a la même structure, donc la même syntaxe abstraite, tandis que $(e_1 + e_2) * e_3$ devient `Mult(Add(`$e_1$`, `$e_2$`), `$e_3$`)`.

Il en est de même pour les instructions: une boucle `while` avec une condition $e$ et un corps $c$ est représentée par un constructeur binaire: `While(`$e$`,`$c$`)`. Une sélection `if` $e$ `then` $c_1$ `else` $c_2$ se traduit en

`If(`$e$`,`$c_1$`, `$c_2$`)`. Une sélection a toujours deux branches, une sélection sans `else` est facilement représentable avec le constructeur `If` et `Skip` introduit plus bas. *(Comment?)*

En somme, notre mini-langage comporte les éléments suivants:

- *Expressions* $e$, avec:
  - des opérateurs binaires, au moins une addition binaire, soustraction, multiplication et égalité.
  - des variables
  - des constantes
- *Instructions* $c$, avec:
  - `Skip`, représentant une instruction vide
  - L'affectation d'une expression à une variable
  - La séquence de deux instructions: $c_1$`; `$c_2$
  - Une sélection, comme décrite plus haut
  - Une boucle `while`
  - Un "return", de la forme `Return` $e$.

Ainsi,

```
if (x == 3) then x = 2; else x = x + 2;
```

est représenté en Caml par:

```
If (Eq (Var "x", Const 3),
    Aff("x", Const 2),
    Aff("x", Add (Var "x", Const 2)))
```

et en Java par:

```
new If (new Eq (new Var("x"), new Const(3)),
        new Aff("x", new Const(2)),
        new Aff("x", new Add (new Var("x"), new Const(2))))
```

Développez les types de données nécessaires pour représenter ce genre de programmes !

**Remarque**

On notera qu'il n'est pas demandé de traiter un programme source exprimé dans le mini-langage, mais sa représentation par un terme Caml ou un objet Java.

## 2.2 Analyses

Les analyses se définissent par induction sur les constructeurs du langage selon un schéma proche de celui du calcul de plus faibles préconditions: l'information (ensemble de variables vivantes ou ensemble de sous-expressions) est propagée de la fin du programme vers le début du programme. Chaque type d'instruction transforme l'information d'une manière qui lui est spécifique.

### 2.2.1 Variables vivantes

La méthode de calcul des variables vivantes est illustrée sur l'exemple donné en Section 1. Puisque `x` et `z` sont utilisées dans l'instruction `return x + z` les variables vivantes avant cette instruction sont `{x,z}`. Avant l'affectation `y := y + 1`, les variables `x` et `z` restent vivantes, mais `y` ne devient pas vivante puisque sa valeur est inutilisée par la suite. Par contre, avant l'instruction `x := y`, `y` devient vivante à la place de `x`. D'une manière générale, les variables vivantes avant une affectation sont obtenues en remplaçant la variable affectée par l'ensemble des variables de sa partie droite. Enfin, la première instruction ne modifie pas l'ensemble des variables vivantes puisque la variable affectée n'est pas vivante après l'affectation.

D'une manière générale, l'algorithme de calcul des variables vivantes prend deux arguments: un ensemble de variables et une instruction et procède comme suit:

- Pour une affectation, si la variable est vivante, on la remplace par l'ensemble des variables de la partie droite de l'affectation.
- Pour un return, on retourne l'ensemble des variables de l'expression.
- Pour une séquence de deux instructions, on applique l'algorithme sur la deuxième instruction de la séquence, puis sur la première instruction et le résultat du traitement de la deuxième instruction.
- Pour une *sélection*, on calcule récursivement l'ensemble des variables vivantes de chaque branche, on prend leur union et on ajoute les variables de la condition de branchement.
- Pour une *boucles*, on exploite le fait qu'une boucle `while (`$e$`)` $c$ peut être dépliée indéfiniment, résultant en une suite de sélections `if (`$e$`) then` $c$ `else skip; if (`$e$`) then` $c$ `else Skip; ...`

  Le calcul de l'ensemble des variables vivantes s'appuie sur cette idée: on détermine l'ensemble des variables vivantes lors d'un seul, de deux, ..., de $n$ passages dans la boucle, et on s'arrête lorsque cet ensemble n'évolue plus. On s'assurera de la terminaison de l'algorithme.

### 2.2.2 Expressions très utilisées

L'algorithme de calcul des expressions très utilisées suit le même schéma, mais l'analyse ne renvoie pas un ensemble de variables, mais un ensemble d'expressions. Les traitements spécifiques concernent l'affectation et la sélection:

- Pour une affectation, on supprime de l'ensemble les expressions contenant la variable affectée et on ajoute toutes les sous-expressions de la partie droite.
- Pour une sélection, on prend l'intersection des résultats associés aux deux branches et on ajoute les sous-expressions de la condition.

Vous pouvez maintenant abstraire ce que ces deux algorithmes ont en commun. Par exemple, la traversée récursive des instructions et le calcul d'un point fixe sont identiques. Seules les opérations effectuées pour chaque type d'instruction sont différentes.

Ainsi, vous obtenez en Caml une fonction (d'ordre supérieur) paramétrée par les traitements spécifiques aux algorithmes qui en sont instance, voire un module paramétré. Cette abstraction est aussi possible en Java: une interface regroupe alors les paramètres de l'itérateur générique.

On notera que le calcul de plus faible précondition est aussi une instance de cet algorithme générique.

## 2.3 Optimisation

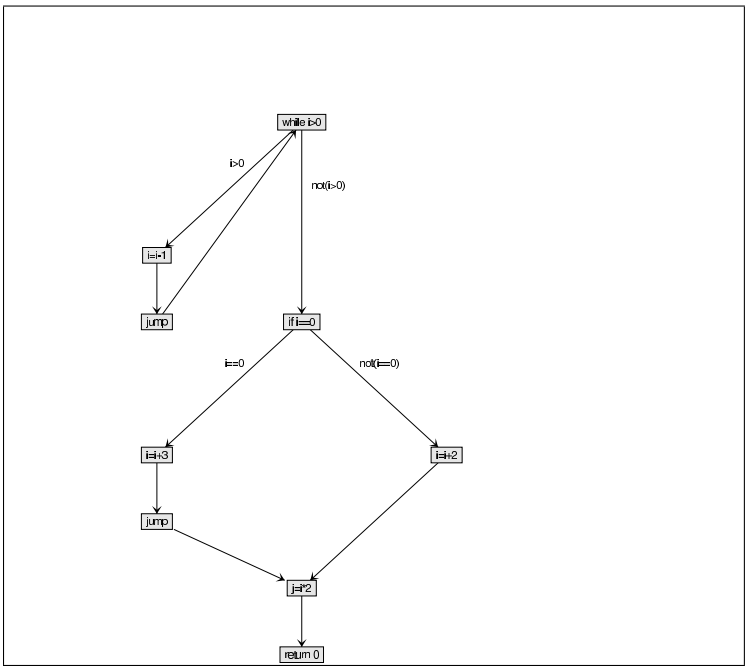
L'analyse des variables vivantes permet d'optimiser les programmes. Pour ceci, il suffit de traverser le programme et, pour chaque affectation, de déterminer l'ensemble des variables vivantes après cette instruction. Si la variable affectée ne figure pas dans cet ensemble, l'affectation peut être remplacée par `Skip`.

Les occurrences inutiles de `Skip` doivent ensuite être supprimées, comme par exemple dans une séquence de la forme `Skip; c;` ou `c; Skip;`.

# 3 Partie C

## 3.1 Vue d'ensemble

Il est demandé d'écrire des modules en C permettant d'afficher (et d'imprimer) le graphe d'un programme. Le résultat final aura la forme suivante :




```
while i > 0 do
  i = i - 1;
done;
if i == 0 then
  i = i + 3;
else
  i = i + 2;
fi
j = i * 2;
return 0;
```

La partie centrale du projet C consiste à calculer les positions des sommets du graphe. Cette partie sera expliquée dans la section 3.4. Mais on commence par définir (section 3.2) les fonctions définissant l'interface entre cette partie et OCaml/Java d'une part, et Postscript d'autre part.

## 3.2 Interface Postscript

Un module *noeud* est fourni pour effectuer les opérations d'affichage. Afin d'en assurer un fonctionnement correct, il faut :

- que la constante symbolique REPERTOIRE_PS soit définie et égale à un répertoire où le programme pourra trouver les fichiers "preambule.ps" et "fin.ps".
- que la constante symbolique CHEMIN_GV soit définie et égale au chemin complet d'un exécutable pouvant lire les fichiers postscript (tm) tels que *gv* ou *ghostview*.
- que la fonction *get_nombre_sommets* soit correctement définie (voir Section 3.3.3).

Ce module définit trois fonctions :

```
void  enregistre_noeud(valeur v, node_id id, int abscisse, int ordonnee);
void  enregistre_arc(node_id origine, node_id destination, valeur label);
void  ecris_vers(char *fichier);
```

Pour les utiliser il suffit d'enregistrer tous les noeuds et les arcs du graphe avec les deux premières fonctions. Une fois l'enregistrement terminé, on peut utiliser la fonction *ecris_vers* pour écrire le résultat dans un fichier Postscript.

## 3.3 Structures de données

### 3.3.1 Types de base

Pour la suite, on supposera que les alias de type suivants sont définis.

```
typedef char *          valeur;
typedef int             node_id;
typedef unsigned char   boolean;
```

Afin de représenter les différentes instructions on introduit une énumération:

```
typedef   enum {
          assignation = 0,
          fin_programme,
          instr_skip,
          instr_seq,
          goto_simple,
          instr_if,
          instr_while,
}         instruction;
```

### 3.3.2 La structure *sommet*

```
struct    sommet_s {
          node_id            id;
          instruction        sorte;
          struct sommet_s    **successeurs;
          valeur             label;
};

typedef   struct sommet_s    sommet;
```

### 3.3.3 Fonctions de créations

Il est demandé d'écrire un module implantant les fonctions suivantes. Il est possible (et recommandé !) d'écrire des fonctions annexes qui n'apparaîtront pas dans le fichier *.h*.

```
sommet *   cons_while(valeur condition, sommet * corps);
sommet *   cons_seq(sommet * instruction1, sommet * instruction2);
sommet *   cons_if(valeur condition, sommet * cas_vrai,
           sommet * cas_faux);
sommet *   cons_assignation(valeur assign);
sommet *   cons_return(valeur val);
sommet *   cons_skip();
int        get_nombre_sommets();
```

La fonction *get_nombre_sommets()* retourne le nombre actuel de sommets créés.

## 3.4 Calcul des positions des sommets

Les arcs sont positionnés automatiquement.

### 3.4.1 Principe du positionnement

L'abscisse et l'ordonnée d'un sommet sont calculées séparément. La page sur laquelle peuvent être placés les sommets est un rectangle dont le coin inférieur gauche a pour coordonnées (0,0) et le coin supérieur droit a pour coordonnées (612,792).

### 3.4.2 Calcul des abscisses

Il faut définir un module *placement_ abscisse* dont l'interface contient deux fonctions :

- *int get_ abscisse(node_ id i)* qui retourne l'abscisse du sommet d'identificateur $i$;
- *void place_ abscisse(sommet *s, int n)* qui calcule les abscisses à partir du sommet $s$ pour un graphe ayant au plus $n$ sommets.

**Algorithme général.** Il est conseillé d'utiliser une fonction auxiliaire récursive *pa_ aux* pour faire le calcul et d'utiliser un tableau d'entiers contenant les abscisses des différents sommets. L'algorithme est décomposé en 2 phases :

1. allocation de la place pour les structures de données;
2. calcul récursif de l'abscisse de chaque sommet.

**Calcul de l'abscisse.** Le principe général est le suivant.

- dès qu'on arrive sur un sommet dont l'abscisse est déjà calculée, on s'arrête en ne faisant rien.
- Sinon on positionne le noeud au milieu d'un intervalle (min,max) passé en paramètre.
- Cet intervalle change dans les cas suivants :
  - pour le corps d'une boucle et pour la partie *then* d'une condition, il devient (min, (min+max)/2);
  - pour la partie *else* d'une condition il devient ((min+max)/2, max);
  - pour le fils d'un *goto_ simple* il devient (min, 2*max-min).

  L'algorithme est correct pour le *goto_ simple* car dans le cas de la fin d'un corps de boucle on s'arrête immédiatement, et dans le cas de la fin d'une partie positive de condition le calcul permet de retrouver les anciennes valeurs de (min, max).

### 3.4.3 Calcul des ordonnées

Il faut définir un module *placement_ ordonnee* dont l'interface contient deux fonctions :

- *int get_ ordonnee(node_ id i)* qui retourne l'ordonnée du sommet d'identificateur $i$;
- *void place_ ordonnee(sommet *s, int n)* qui calcule les ordonnées à partir du sommet $s$ pour un graphe ayant au plus $n$ sommets.

**Algorithme général.** Il est conseillé d'utiliser une fonction auxiliaire récursive *po_ aux* pour faire le calcul et d'utiliser un tableau d'entier contenant les ordonnées des différents sommets. L'algorithme est décomposé en 3 phases :

1. allocation de la place pour les structures de données;
2. calcul du *niveau* (0 pour le sommet le plus haut, 1 pour ceux juste en dessous, etc.) de chaque sommet;
3. S'il y a $n$ niveaux, traduction du niveau $k$ en ordonnée $p$ avec la formule:

$$p = 750 - \frac{k \times 680}{n}$$

**Calcul des niveaux.** Le fils d'un sommet au niveau $k$ est au niveau $k+1$ sauf dans deux cas :

- le fils d'un *jump* qui termine le corps d'une boucle *while*, qui est déjà correctement placé;
- le fils commun d'un *jump* (au niveau $k_1$) de la partie *then* d'une condition *if* et de la dernière instruction (au niveau $k_2$) de la partie *else* de cette condition. Ce fils doit être placé au niveau $\max(k_1, k_2) + 1$.

Une solution peut être trouvée en supposant que la partie *then* d'une condition est toujours visitée avant la partie *else*. Dans ce cas il est possible de différencier le *jump* qui conclut un corps de boucle de celui qui conclut une partie *then* en regardant leur fils et en examinant si ce fils a déjà une ordonnée ou non.

- *fin de corps de boucle* :s'il est déjà placé, on s'arrête;
- *fin de partie then :* on met le niveau à $k$ et on s'arrête
- lorsqu'on entre dans un sommet, si le niveau est déjà positionné, on est sûr qu'on vient de sortir de la partie *else* d'une condition. Il faut mettre à jour le niveau en prenant le maximum de la valeur courante et de $k$, et continuer avec cette nouvelle valeur.

# 4 Interface avec C

L'affichage en C d'un programme représenté par un terme Caml ou un objet Java nécessite la reconstruction par C de la structure de données dénotant le programme. Le code C travaille sur une structure de graphe obtenue en ajoutant des arcs à une structure arborescente. La reconstruction en C de cette structure s'appuie sur un parcours récursif de la structure source. Pour cela, on associe à chaque classe Java ou constructeur Caml une fonction externe, implantée en C, et retournant l'adresse allouée par C de la copie de structure source. Considérons par exemple la construction `if_then_else` du mini langage.

## 4.1 En Caml

On introduit un type abstrait, c'est à dire non défini, `noeud` dont les valeurs sont des pointeurs sur des structures C, et une déclaration de fonction Caml implantée en C:

```
type noeud
external if_then_else: string -> noeud -> noeud -> noeud = "C_if"
```

Cette fonction reçoit en arguments une chaîne de caractères contenant le test et les résultats des allocations par C des deux branches du `if`. Elle appele la fonction C `C_if` utilisant la fonction `cons_if` pour allouer un noeud et l'initialiser avec les trois arguments fournis.

Les déclarations C correspondantes sont les suivantes:

```
#include "/usr/local/lib/ocaml/caml/memory.h"
#include "/usr/local/lib/ocaml/caml/mlvalues.h"

CAMLprim value C_ite(value vs, value ift, value iff)
{
  CAMLparam3(vs,ift,iff);
  char* s = String_val(vs); /* conversion en chaîne de caractères */

  /* allocation et initialisation de la structure */
  sommet *n = cons_if(s, (sommet*) ift, (sommet*) iff);

  /* transmission à Caml de l'adresse de la struture allouée */
  CAMLreturn ((value) n);
}
```

La compilation est l'exécution du projet C/Caml est réalisée par les deux instructions suivantes:

```
gcc -c partie_ecrite_en_C.c
ocamlc -custom -o projet partie_Caml.ml ... partie_ecrite_en_C.o ...
./projet   -- exécution du projet Caml
```

Les parties C et Caml peuvent bien sûr comporter plusiers fichiers.

Les informations nécessaires sont disponibles à l'adresse suivante:
`http://caml.inria.fr/pub/docs/manual-ocaml/manual032.html`

## 4.2 En Java

Les adresses des zones allouées par C seront vues en Java sous la forme d'entiers. L'interface Java/C est donc décrite par une classe Java comportant une méthode statique par fonction C. Le mot clé `native` indique que la méthode est en fait implantée en C:

```
class InterfaceAvecC {
  ...
  public static native int if_then_else(String s, int ift, int iff);
}
```

L'en-tête du fichier C implantant les méthodes est automatique généré en utilisant la commande `javah`:

```
  javac InterfaceAvecC.java
  javah -jni InterfaceAvecC
```

Avec l'exemple précédent, le profil généré (dans un .h) est le suivant:

```
JNIEXPORT jint JNICALL Java_InterfaceAvecC_if_1then_1else
  (JNIEnv *env, jclass, jstring js, jint ji, jint);
```

Les arguments de la méthode se retrouvent en position 3 à 5. Le fichier (.c) implantant les interfaces décrites dans le .h doit convertir les types Java non natifs en types C:

```
const char *s = (*env)->GetStringUTFChars(env, js, 0);
int i = ji;
```

Le fichier C doit ensuite être compilé en une libraririe partagée dont le nom doit commencer par `lib`:

```
-- Linux/gcc
gcc -I/usr/java/jdk1.5.0_01/include/ -I/usr/java/jdk1.5.0_01/include/linux -c InterfaceAvecC.c
gcc -shared -o libInterfaceAvecC.so InterfaceAvecC.o

-- solaris
cc -G -I/opt/JAVA/jdk1.5.0_01/include -I/opt/JAVA/jdk1.5.0_01/include/solaris InterfaceAvecC.c
   -o libInterfaceAvecC.so
```

Cette libraririe doit être chargée dans le code Java:

```
class InterfaceAvecC {
  static {
    System.loadLibrary("InterfaceAvecC");
  }
  public static native int if_then_else(String s, int ift, int iff);
}
```

Toutes les informations sont disponibles à l'adresse suivante:
`http://java.sun.com/j2se/1.4.2/docs/guide/jni/`

## 5 Travail demandé

Le jour de la soutenance, vous devrez rendre un dossier contenant les éléments suivants:

- Un (ou des) exemples de programme contenant tous les constructeurs et annotés par les variables vivantes et les expressions très utilisées.
- Le résultat de l'optimisation des programmes donnés en exemple.
- Une justification de la terminaison des algorithmes d'analyse.
- Un comparatif sur les styles de programmation Java et Caml.
- Un listing commenté de votre projet

Des références supplémentaires se trouvent sur les pages Web:
`http://www.irit.fr/recherches/TYPES/SVF/strecker//Teaching/L3_S6_Concepts_Programmation_2004/`
(partie Caml/Java) et
`http://www.irit.fr/recherches/LILAC/Pers/Chevalier/index.html#enseignement`
(partie C)