# Cours de compilation
# 2ème partie
# Génération de code

Jacques Ferber

## 1. Une approche fonctionnelle de la compilation

On utilisera la notion de schéma de compilation. Un schéma de compilation est une règle de réécriture de la forme:

```
Compil[<phrase1>] → <phrase2>
```

où `<phrase1>` est une énoncé du langage source et `<phrase2>` est une expression qui contient du code pour la machine cible. On notera les variables sous la forme `<x>` pour désigner une variable du schéma de compilation (qu'il faut bien évidemment différencier des variables du langage cible).

L'opération de concaténation des expressions s'exprimera à l'aide d'un point virgule ';'.

L'opération de compilation est une opération récursive.

On utilisera fortement le fait de passer les arguments sur la pile. On verra par la suite les problèmes d'optimisation que cela pose et on réfléchira à la manière de les régler.

*Note* : dans la définition de la compilation, on utilisera un pseudo-langage dont les arguments sont notés entre crochets [a1,a2,...,an] et qui utilise une notation infixe. L'opérateur @ dans la génération de code indique qu'il faut évaluer l'expression qui suit.

## 2. Les expressions constantes

Les expressions constantes seront empilées en plaçant les valeurs intermédiaires sur la pile en transformant les expressions sous une forme préfixée.

```
Compile[(<op> <e1> <e2>)] où op est une opération primitive →
    Compile[<e1>] ;
    (PUSH A0)
    Compile[<e2>] ;
    (PUSH A0)
    CompileOp[<op>]
```

On supposera que les résultats de toutes les opérations sont passées dans le registre `A0`.

La compilation d'un opérateur primitif passe généralement par l'appel à une bibliothèque de primitives (calculs de flottants, calculs d'entiers longs, ...) par l'intermédiaire de l'instruction d'appel ( `JSR <prim>`). Dans ce cas CompileOp a la forme suivante:

```
CompileOP[<op>] →
   (JSR @etiquetteDe [<op>])
```

Si `op` est en plus une fonction prédéfinie de la machine virtuelle (par exemple la fonction +, on peut la compiler ainsi:

```
CompileOp[+] →
      (POP A1) ;
      (POP A0) ;
      (ADD A0 A0 A1)
```

En fait, dans le cas de SCHEME on ne sait pas si l'opérateur est effectivement primitif ou non car on ne connaît pas le type des expressions. Cela changera avec l'introduction des types et donc on doit compiler les primitives en faisant appel à une bibliothèque de procédures primitives.

Les constantes sont directement placées dans le registre `A0`: `(MOV A0 <cste>)`

Les variables globales sont traitées par utilisation de l'accès direct à une case mémoire: `(MOV A0 (loc <var>))`

Ex: Compiler l'expressions suivante: (a + 4)*(2 - b)

```
(* (* (+ 4 a) 2) (+ 2 b))
```

en supposant que a et b sont des variables globales.

```
Compile[(* (+ 4 a) (+ 2 b))] →
(1) Compile[(* (+ 4 a) 2)];
      (PUSH A0)
(2) Compile[(+ 2 b)];
      (PUSH A0)
(3)   CompileOp[*]

(1)   Compile[(* (+ 4 a) 2)] →
(1.1) Compile[(+ 4 a)];
         (PUSH A0)
(1.2)    CompileCste[2];          → (MOV A0 2)
         (PUSH A0)
(1.3)    CompileOp[*]             → (JSR op_mult)

(1.1)    Compile[(+ 4 a)] →
(1.1.1)       CompileCste[4];     → (MOV A0 4)
                                    (PUSH A0)
(1.1)         CompileVarGlob[a]   → (MOV A0 (loc a))
                                    (PUSH A0)
(1.1.3)       CompileOp[+]        → (JSR op_plus)

(2)   Compile[(+ 2 b)]
(2.1) CompileCste[2];             → (MOV A0 2)
                                  → (PUSH A0)
(2.2) CompileVarGlob[b];          → (MOV A0 (loc b))
```

```
                                        → (PUSH A0)
  (2.3) CompileOp[-]                    → (JSR op_minus)
```

En remettant tout dans l'ordre, cela donne:

```
(MOV A0 4)
(PUSH A0)
(MOV A0 (loc a))
(PUSH A0)
(JSR op_plus)
(MOV A0 2)
(PUSH A0)
(MOV A0 (loc b))
(PUSH A0)
(JSR op_minus)
(JSR op_mult)
```

Lorsqu'on ne fait pas appel à des sous-programmes et qu'on compile directement dans la machine, cela donne:

```
(MOV A0 4)
(PUSH A0)
(MOV A0 (loc a))
(PUSH A0)
(POP A1)
(POP A0)
(ADD A0 A0 A1)
(PUSH A0)
(MOV A0 2)
(PUSH A0)
(MOV A0 (loc b))
(PUSH A0)
(POP A1)
(POP A0)
(SUB A0 A0 A1)
(PUSH A0)
(POP A1)
(POP A0)
(MULT A0 A0 A1)
```

# 3. Les fonctions de tests

En général, les procédures de tests se compilent comme les opérateurs arithmétiques (saut quand les tests peuvent être effectués par la machine cible et qu'il se situe dans une structure de contrôle telle qu'un 'if').

Ex: (a > 4)

```
Compile[(> a 4)] →
   CompileVarGlob[a];
   CompileCste[4]
```

```
(MOV A0 (loc a))
(PUSH A0)
(MOV A0 4)
(PUSH A0)
(JSR op_sup)
```

# 4. L'affectation de variables globales

```
Compile[(set! <var> <expr>) →
   Compile[<expr>];
   (MOV (loc <var>) A0)
```

# 5. Les structures de contrôles

## 5.1 Séquence

La structure de séquence s'exprime simplement comme la concaténation de la compilation des instructions se situant dans la séquence.

```
Compile[(begin <e1> <e2> ... <en>)] →
   Compile[<e1>];
   Compile[<e2>];
      ...
   Compile[<en>]
```

## 5.2 If et While

Les structures de contrôles classiques (conditionnelles et boucles) doivent gérer des étiquettes.

```
Compile[(if <test> <alors> <sinon>)] →
   soit etiq-sinon = generer-etiquette()
        etiq-fin = generer-etiquette()
      Compile[<test>];
      (JNI A0 @etiq-sinon);
      Compile[<alors>];
      (JMP @etiq-fin);
   @etiq-sinon ;
      Compile[<sinon>];
   @etiq-fin ;
```

Le ' while' utilise une technique semblable.

```
Compile[(while <test> <expr>)] →
   soit etiq-boucle = generer-etiquette()
        etiq-fin = generer-etiquette()
   @etiq-boucle
      Compile[<test>];
      (JNI A0 @etiq-fin);
      Compile[<expr>];
```

```
        (JMP @etiq-boucle)
    @etiq-fin
```

Le 'for' est plus complexe: (for (i 0 n) <      expr>)

Deux cas se présentent. Soit la valeur finale est une constante et on n'a besoin de la calculer qu'une fois (c'est le cas général), soit la valeur finale peut évoluer au cours de la boucle et il faut alors la recalculer systématiquement.

Cas 1:

```
Compile[(for (<var> <expr-init> <expr-fin>) <expr>)] →
    soit etiq-boucle = generer-etiquette()
          etiq-fin = generer-etiquette()
        Compile[<expr-init>];
        (MOV (loc <var>) A0);
        Compile[<expr-fin>]; // suppose que la valeur est une cste
        (PUSH A0);
@etiq-boucle
        (MOV A0 (loc <var>));
        (POP A1)      // on recupere la valeur finale de la boucle
        (JGE A0 A1 etiq-fin)
        Compile[<expr>];
        (JMP @etiq-boucle)
@etiq-fin
```

Cas 2:

```
Compile[(for (<var> <expr-init> <expr-fin>) <expr>)] →
    soit etiq-boucle = generer-etiquette()
          etiq-fin = generer-etiquette()
        Compile[<expr-init>];
        (MOV (loc <var>) A0);
@etiq-boucle
        Compile[<expr-fin>]; // recalcule la valeur de fin
        (MOV A1 A0);       // la valeur finale est dans A1
        (MOV A0 (loc <var>));
        (JGE A0 A1 etiq-fin)
        Compile[<expr>];
        (JMP @etiq-boucle)
@etiq-fin
```

# 6. Les appels de sous-programmes

Les sous-programmes se trouvent à deux endroits :

1. Dans la déclaration du sous-programme
2. Lors de son utilisation (appel)

## 6.1 Compilation d'une déclaration de sous-programme

*Remarque* : pour l'instant on compile les sous-programmes sans paramètres ni variables locales :

```
Compile[(define (<proc>) <expr>)] où <expr> est généralement de la
                                  forme (begin <e1> .. <en>)
 →
   <proc> ;   // l'etiquette du sous-programme
      (FENTRY <proc>) ; // ne sert à rien en fait
      Compile[expr] ;
      (RTN)
```

L'appel d'un sous-programme s'exprime ainsi :

```
Compile[(<proc>)] →
   (JSR <proc>)
```

*Remarque* : cela montre la simplicité de la définition des sous-programmes. En fait le problème principal se situe non pas dans les sous-programmes, mais dans les variables locales et les paramètres passés en argument.

# 7. Paramètres et variables locales

De nombreuses techniques peuvent être utilisées pour traiter les paramètres des procédures. La première, qui est celle employée initialement par Fortran, consiste à réserver des variables globales dans chaque procédure. Mais elle ne permet pas la récursivité. Les autres utilisent toutes la pile.

## 7.1 Passage de paramètres à la Fortran

```
Compile[(define (<proc> <x1> .. <xn>) <expr>)] →
   soit debut_proc = concat['debut_, <proc>]
        var_x1 = concat['var_, <proc>, _, <x1>]
          ...
        var_xn = concat['var_, <proc>, _, <xn>]
      <proc> ;// l'etiquette du sous-programme
         (JMP @debut_proc)
      @var_x1 0 ;
         ...
      @var_xn 0 ;
      @debut_proc
```

```
(FENTRY <proc>) ; // ne sert à rien en fait
(POP A2)   ; // récupère l'adresse de retour
(POP A0)   ; // récupère les variables
(MOV (loc @var_xn) A0) ; // et on les stocke en mémoire
...
(POP A0)
(MOV (loc @var_x1) A0)
(PUSH A2)  ; // remet l'adresse de retour sur la pile
Compile[expr] ;
(RTN)
```

Appel d'un sous-programme:

```
Compile[(<proc> <e1> .. <en>)] →
   Compile[<e1>] ;
   (PUSH A0)
   Compile[<en>] ;
   (PUSH A0)
   (JSR <proc>))
```

*Problème* : pas de récursivité possible : {{le montrer avec un exemple simple dans lequel on voit que les variables d'appel sont écrasées par les appels successifs}}

*Autre problème* : l'accès aux valeurs passées en argument lors de la compilation du corps de la procédure. Comment y accéder ?

Réponse : utiliser ce que l'on appelle un environnement. En informatique, un environnement est une suite de couples (symbole, valeur), que l'on notera :

```
{(s1,v1),..,(sk,vk)}
```

et que l'on manipulera avec les deux fonctions

```
at[env,si]→ vi si si est un symbole de l'environnement
            et une erreur sinon.
atput[env,sj,v'j]→ env' = env + {(sj,v'j)} si sj n'est pas
      un symbole de l'environnement et
      env - {(sj,vj)} + {(sj,v'j)} si (sj,v'j) était dans env.
```

De plus on ajoutera un argument à la fonction `Compile[<expr>, <env>]` qui exprime que l'expression se compile dans l'environnement `<env>`. Nous verrons par la suite que nous ajouterons d'autres variables à la fonction `Compile`. Dans toutes les expressions que nous avons compilé pour l'instant, cela ne change rien : on reporte simplement la variable d'environnement inchangée.

Ainsi, la compilation d'expression s'écrit maintenant ainsi :

```
Compile[(<op> <e1> <e2>), <env>] où op est une opération primitive
→
   Compile[<e1>,<env>] ;
   (PUSH A0)
   Compile[<e2>,<env>] ;
   (PUSH A0)
   CompileOp[<op>,<env>]
```

Lors de la compilation du corps d'une procédure, nous allons passer à la fois les noms des variables ainsi que les adresses (les étiquettes) des variables locales (je note en gras ce qui change) :

```
Compile[(define (<proc> <x1> .. <xn>) <expr>),<env>]
   soit debut_proc = concat['debut_, <proc>]
        var_x1 = concat['var_, <proc>, _, <x1>]
          ...
        var_xn = concat['var_, <proc>, _, <xn>]
         <nenv> = {(<x1>,var_x1), .., (<xn>,var_xn)}
   →
      <proc> ;// l'etiquette du sous-programme
         (JMP @debut_proc)
      @var_x1 0 ;
         ...
      @var_xn 0 ;
      @debut_proc
         (FENTRY <proc>) ; // ne sert à rien en fait
         (POP A2)
         (POP A0)
         (MOV (loc @var_xn) A0)
         ...
         (POP A0)
         (MOV (loc @var_x1) A0)
         (PUSH A2)
         Compile[<expr>,<nenv>] ;
         (RTN)
```

## 7.2 Accès aux variables locales

Lors de la compilation des variables, ce qui s'exprimait auparavant ainsi :

```
Compile[<ident>] → (MOV A0 (loc <ident>))
```

se compile maintenant à l'aide des environnements :

```
Compile[<ident>,<env>] → (MOV A0 (loc @at[<env>,<ident>]))
```

## 7.3 Valeur de retour

On notera qu'il n'y a strictement rien à faire pour que les procédures retournent un résultat. Le dernier résultat se trouve dans A0 lors de la sortie de la procédure. Si l'on désire avoir une instruction de type `return` comme en C, on écrira simplement :

```
Compile[(return <expr>), <env>] →
   Compile[<expr>, <env>] ;
   (RTN)
```

## 7.4 Passages de paramètres par la pile

Puisqu'on a utilisé une pile pour écrire le langage, continuons !

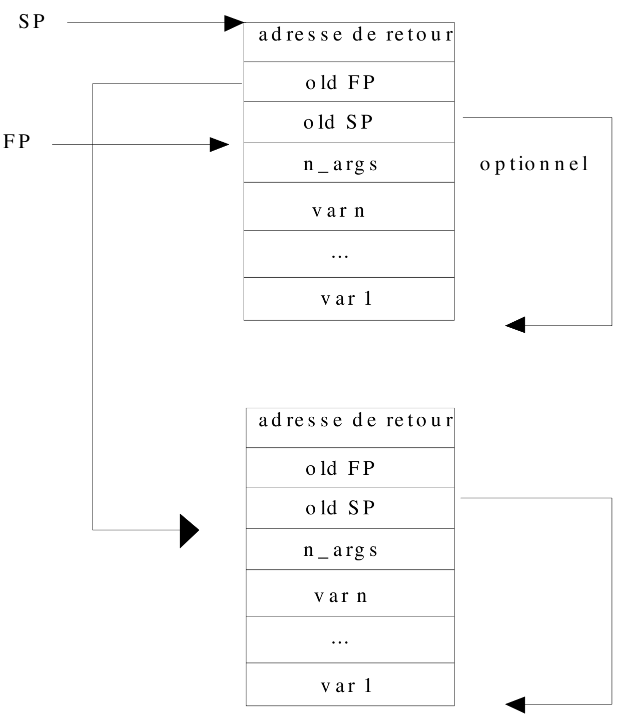
Principe : les variables sont simplement laissées sur la pile. Une procédure est alors décrite ainsi :

```
Compile[(define (<proc> <x1> .. <xn>) <expr>),<env>] →
      @debut_proc
```

```
(FENTRY <proc>) ; // ne sert à rien en fait
Compile[<expr>,<env>] ;
(RTN)
```

comme au début, et l'appel de procédure se fait simplement comme dans l'appel de type Fortran. Problème : comment accéder aux variables locales sur la pile ? Réponse : il faut marquer la pile lors de l'appel des sous-programmes ⇒ notion de bloc de pile ( stack frame).

Il existe de nombreuses formes de bloc de pile, mais elles se ramenent simplement à faire des variantes de celle qui est décrite ci-dessous.




Cela signifie que lors de la compilation d'un appel il faut générer le code suivant :

```
Compile[(<proc> <e1> .. <en>),<env>]
   soit n = nombre d'arguments
   →
   Compile[<e1>,<env>] ;
   (PUSH A0)
   Compile[<en>,<env>] ;
   (PUSH A0)
// debut de l'entete d'appel
   (PUSH @n)        // nombre d'arguments
// ce qui se passe ici pourrait être fait dans la procédure
// appelée (technique du link et unlink). Mais, dans ce cas,
// l'adresse de retour se situe juste au dessus de n-args
   (MOV A1 FP)      // on recupere l'ancien FP
   (MOV FP SP)      // le nouvel FP = SP
   (MOV A2 SP)      // on recupere l'ancien SP
```

```
    (DIFF A2 A2 @n)
    (DIFF A2 A2 1)   // A2 = SP - (n + 1)
    (PUSH A2)        // sauve l'ancien SP
    (PUSH A1)        // sauve l'ancien FP
 // fin de l'entete d'appel
    (JSR <proc>)
 // debut de la restauration du contexte d'exécution
    (POP A1)         // recupere l'ancien FP
    (POP A2)         // recupere l'ancien SP
    (MOV FP A1)      // FP est restaure a son ancienne valeur
    (MOV SP A2)      // on a supprime toutes les variables
 // fin de la restauration
```

On récupère les variables sur la pile à l'adresse suivante :

```
    vari → FP - (n + 1) + i // on compte i à partir de 0
```

Si on utilise l'instruction (local <reg><n>) qui va chercher le n ième élément sur la pile à partir de FP et le place dans <reg>

Ainsi la compilation d'une procédure avec passage de paramètres se compile ainsi:

```
 Compile[(define (<proc> <x1> .. <xn>) <expr>),<env>]
    soit n = nombre de paramètres
    <nenv> = {(<x1>,-(n+1)+1),..,
                    (<xi>,-(n+1)+i),..,
                    (<xn>,-2)} // - 2 car -(n+1)+(n-1) = -2
    →
    <proc> ;   // l'etiquette du sous-programme
          (FENTRY <proc>) ; // ne sert à rien en fait
          Compile[<expr>,<nenv>] ;
          (RTN)
```

et l'accès aux variables s'écrit ainsi :

```
 Compile[<ident>,<env>] → (LOCAL A0 @at[<env>,<ident>])
```

De même l'affectation est donnée par :

```
 Compile[(set! <ident> <expr>), <expr>] →
    Compile[<expr>, <env>];
    (SETLOCAL @at[<env>, <ident>] A0)
```

## 7.5 Compilation des variables locales

Principe: les variables locales sont sur la pile et "poussent" à l'envers des paramètres.

Ces variables sont introduites par le mot clé Var en Pascal au début d'une procédure où se situent au début d'un "bloc" en C. En SCHEME et Lisp ces variables sont introduites par la structure Let. On supposera que les variables locales sont définies une fois pour toute par le mot clé `vars` au début de la définition d'une procédure.

On supposera qu'il n'est pas possible de définir

```
 (define (f x)
```

```
(vars z y)
(set! z (1+ x))
(set! y 0)
(while (> y z)
   ...
   (set! y (1+ y))
))
```

```
Compile[(vars <x1> .. <xn>), <env>]
   pour <xi>
      atput[<env>,<xi>,i+2] // le +2 est utilisé pour sauter par
                 // dessus les infos laissées sur la pile
   → pour i = 1 to n do
      (PUSH 0)      // réserve de la place pour la variable locale
```

L'accès aux variables locales se fait ensuite comme pour les paramètres, c'est-à-dire par la règle de compilation suivante:

```
Compile[<ident>,<env>] → (LOCAL A0 @at[<env>,<ident>])
```

## 7.6 Procédures dans des variables

Certains compilateurs Pascal et C permettent d'avoir des pointeurs sur des procédures et d'utiliser dynamiquement ces procédures. Par exemple:

```
void f(int x) {..}

main() {
   ...
   m = f;
   ..
   (m*)(3);
   ..
}
```

De même en SCHEME on peut écrire:

```
(define (f x y)(* x y))

(set! a f)

(a 3 4) → 12
```

*Principe* : si l'on s'aperçoit que l'identificateur réfère à une procédure (mais qu'elle n'a pas été définie ainsi), on compile non pas un appel direct, mais un appel indirect via `A0`. Les modifications par rapport à un appel direct sont notées en gras:

```
Compile[(<expr> <e1> ... <en>), <env>] →
      soit n = nombre d'arguments
   →
   Compile[<e1>,<env>] ;
   (PUSH A0)
   Compile[<en>,<env>] ;
```

```
    (PUSH A0)
 // debut de l'entete d'appel
    (PUSH @n)       // nombre d'arguments
 // insertion de la compilation de <expr>
    Compile[<expr>, <env>]; // Attention: résultat dans A0
    (MOV A1 FP)     // on recupere l'ancien FP
    (MOV FP SP)     // le nouvel FP = SP
    (MOV A2 SP)     // on recupere l'ancien SP
    (DIFF A2 A2 @n)
    (DIFF A2 A2 1)  // A2 = SP - (n + 1)
    (PUSH A2)       // sauve l'ancien SP
    (PUSH A1)       // sauve l'ancien FP
 // fin de l'entete d'appel
    (JSR A0)        // appel indirect via A0
 // debut de la restauration du contexte d'exécution
    (POP A1)        // recupere l'ancien FP
    (POP A2)        // recupere l'ancien SP
    (MOV FP A1)     // FP est restaure a son ancienne valeur
    (MOV SP A2)     // on a supprime toutes les variables
 // fin de la restauration
```

*Remarque* : cette modification peut s'appliquer aussi à l'appel direct. C'est généralement ce qui se passe dans les langages faiblement typés, où il est difficile de savoir si la variable est un identificateur correspondant effectivement à la fonction où s'il s'agit d'une variable qui contient une procédure.

## 7.7 Procédures locales

Compilation des fonctions auxiliaires que l'on ne veut pas définir à l'extérieur d'une procédure. On rencontre ce type de procédure en PASCAL, en ADA et en SCHEME. Exemple en SCHEME :

```
(define (indice lst x)
  (define (indice1 lst x i)
      (cond
         ((null? lst) ())
         ((equal? (car lst) x) i)
         (else (indice1 (cdr lst) x (1+ i)))))
  (indice1 lst x 0))

(indice '(a b c d e) 'c) → 2

(define (f x)
  (define (g z)
     (set! x (1+ z))
     x)
  (+ x (g 3)))

(f 5) → 9
```

Principe:

1. On considère que la procédure interne (par exemple g) dans le deuxième exemple est une variable locale de la procédure.
2. On compile l'appel à la procédure locale comme s'il s'agissait d'une procédure passée dans une variable.
3. Les variables locales à la procédure interne sont récupérées normalement sur la pile dans un bloc de pile local.
4. Les variables globales à la procédure interne, mais locales à la procédure de définition sont accédées au travers du pointeur de frame et on remonte ensuite les blocs de piles jusqu'à aller au bloc de pile de bon niveau.
5. Lors de la compilation, on passe un environnement structuré en plusieurs parties. Chaque partie correspond à un niveau d'emboîtement.

La compilation des procédures les plus globales n'est pas modifiée!

Voici la nouvelle compilation des procédures:

```
Compile[(define (<proc> <x1> .. <xn>) <expr>),<env>] et <env> ≠ ()
   atput[<env>,<proc>,paramNumber[]] // on ajoute une variable
      // local à l'environnement englobant.
   soit n = nombre de paramètres
         <nenv> = {{(<x1>,-(n+1)+1),..,
                    (<xi>,-(n+1)+i),..,
                    (<xn>,-2)}
                    <env>}     // emboitement des environnement
   →
   <proc> ;   // l'etiquette du sous-programme
         (FENTRY <proc>) ; // ne sert à rien en fait
         Compile[<expr>,<nenv>] ;
         (RTN)
```

Le changement principal se situe dans l'accès aux variables:

```
Compile[<ident>,<env>]
   et <ident> ∈ 1er élément de <env> // variable locale
   →
         (LOCAL A0 @at[<env>,<ident>])

Compile[<ident>,<env>]
   et <ident> ∉ <env> // variable purement globale
   →
         (MOV A0 @<ident>)
```

Et voilà le code de recherche d'un élément dans les blocs de piles englobant!

```
Compile[<ident>,<env>]
   et <ident> ∉ 1er élément de <env> // variable locale/globale
   et <ident> ∈ <env> (élément suivant)
   soit var_x = at[<env>, <ident>] et
         n = niveau d'emboitement de <ident>
   →
      (MOV A1 (FP+1))    // on prend l'ancien FP (indirect indexé)
      (MOV A2 @n)
```

```
recherche
   (JEQ A2 0 trouve)
   (MOV A1 (A1+1))    // recupere les FP
   (SUB A2 A2 1)      // on decremente A2
   (JMP recherche)
trouve
   (MOV A2 FP)        // on sauve FP
   (MOV FP A1)        // FP prend la valeur du FP "global"
   (LOCAL A0 @var_x)
   (MOV FP A2)        // FP reprend sa valeur
```

# 8. Structures de données

Compilation des structures de données.

Plusieurs classes de données:

1. Les données statiques qui sont allouées par le compilateur.
2. Les données semi-dynamiques qui sont allouées sur la pile lors de l'appel d'une procédure.
3. Les données dynamiques qui sont allouées à la demande du programmeur (instruction new en Pascal et malloc en C, objets en Ada et Smalltalk).

## 8.1 Tableaux

### 8.1.1 Compilation statique

#### *8.1.1.1 Déclaration*

On réserve de la place dans une zone de données.

Var A : Array[1..6] of Integer;

sera transformé en:

```
A: word 0
   word 0
   ...
   word 0
```

On a donc:

```
Compile[<nom> : Array [1..<n>] of Integer, <env>]
   et <env> = nil // environnement global
   →
   <nom> word 0;
   for  i = 1 to <n>-1 do
         word 0
```

#### *8.1.1.2 Accès*

*En lecture:*

```
Compile[<tab>[<expr>],<env>] →
   Compile[<expr>, <env>];
   (MOV A1 <tab>); // recupere l'adresse de base
   (ADD A1 A1 A0); // ajoute l'offset
   (MOV A0 (A1)); // recherche indirecte de l'information.
```

*En écriture:*

```
Compile[<tab>[<expr1>]:= <expr2>,<env>] →
   Compile[<expr1>, <env>];
   (PUSH A0)
   Compile[<expr2>, <env>];
```

```
(MOV A1 <tab>); // recupere l'adresse de base
(POP A2);  // recupere la valeur de l'offset
(MOV A1 A1 A2); // ajoute l'offset
(MOV (A1) A0);  // sauvegarde indirecte
```

### 8.1.2 Compilation semi-dynamique

#### *8.1.2.1 Déclaration*

On réserve de la place sur la pile lors de l'allocation des variables locales.

```
Compile[<nom> : Array [1..<n>] of Integer, <env>] et <env> local
   →
   for i = 1 to <n> do
      (PUSH 0)
```

#### *8.1.2.2 Accès*

Les accès s'effectuent comme précédemment. Seul change l'adresse de base qui se trouve sur la pile.

*En lecture:*

```
Compile[<tab>[<expr>],<env>] →
   Compile[<expr>, <env>];
   (LOCAL A1 @at[<env>,<tab>]); // recupere l'adresse de base
   (ADD A1 A1 A0); // ajoute l'offset
   (MOV A0 (A1)); // recherche indirecte de l'information.
```

*En écriture:*

```
Compile[<tab>[<expr1>]:= <expr2>,<env>] →
   Compile[<expr1>, <env>];
   (PUSH A0)
   Compile[<expr2>, <env>];
   (LOCAL A1 @at[<env>,<tab>]); // recupere l'adresse de base
   (POP A2);  // recupere la valeur de l'offset
   (MOV A1 A1 A2); // ajoute l'offset
   (MOV (A1) A0);  // sauvegarde indirecte
```

### 8.1.3 Compilation dynamique

Var P: ^ Array[1..6] of Integer;

```
Compile[new(P), <env>]
   s = taille du type de P
    →
   (ALLOCATE @s);  // general. un appel a une routine systeme
   (MOV (loc <P>) <A0>) // si P est une variable globale.
```

#### *8.1.3.1 Accès*

Les accès s'effectuent comme précédemment. Seul change l'adresse de base qui se trouve dans la variable P. La différence se situe dans l'appel indirect de la variable (opérateur ^ en Pascal et * en C).

*En lecture:*

```
Compile[<p>^[<expr>],<env>] →
   Compile[<expr>, <env>];
   (MOV A1 <P>)
   (MOV A1 (A1)); // recupere l'adresse de base (via l'indirect.)
   (ADD A1 A1 A0); // ajoute l'offset
   (MOV A0 (A1)); // recherche indirecte de l'information.
```

*En écriture:*

```
Compile[<tab>[<expr1>]:= <expr2>,<env>] →
   Compile[<expr1>, <env>];
   (PUSH A0)
   Compile[<expr2>, <env>];
   (MOV A1 <P>)
   (MOV A1 (A1)); // recupere l'adresse de base (via l'indirect.)
   (POP A2);  // recupere la valeur de l'offset
   (MOV A1 A1 A2); // ajoute l'offset
   (MOV (A1) A0);  // sauvegarde indirecte
```

## 8.2 Record

On gère les record comme des tableaux. Seul change le fait que les valeurs sont de types différents et donc (normalement) de taille différente.

```
R: Record a: Integer; b: Integer end;
```

sera transformé en:

```
R: word 0      // A
   word 0      // B
```

C'est le compilateur qui a la charge de faire le lien entre les noms de champs et l'indice de la valeur

```
Compile[<rec>:Record <champ1>:<type1>;..;<champ1>:<type1> end,
        <env>] et <env> = nil
   soit taille = taille de <rec>
   →
   <nom> word 0;
   for  i = 1 to <taille>-1 do
         word 0
```

Suppose que la table des symbole dispose des informations nécessaires ensuite pour transformer les noms des champs en offsets.

## 8.3 Objets

Même structure que pour les records.

## 8.4 Allocation et récupération dynamique de mémoire

Allocation: pointeurs et doubles-pointeurs ( handle). Gestion dynamique de la mémoire dans un malloc.

Récupération de mémoire: le GC.

# 9. Un moteur de génération de code

```
;;-----------------------------------------------------------
;;
;;        la compilation d'un langage de type Scheme
;;        en utilisant la portee lexicale
;;
;;-----------------------------------------------------------
;; (C) J. Ferber - LIRMM
;;-----------------------------------------------------------

(define *LAP* nil) ;; la variable qui contient le code en cours de
generation
(define *INDEXVAR* 0)


(define (gencode x) ; pour engendrer un seul element (etiquette ou
instruction)
    (push *LAP* x)
    )

(define (gencodes . lx) ; pour engendrer une sequence
    (mapc (lambda (x) (gencode x)) lx))


(define (print-lap lp)
  (mapc (lambda (c)
      (if (atom? c) (print c) ; cas d'une etiquette, cadree a
gauche
      (print "      " c) ; cas d'une instruction, indentee
                 ))
        lp ))


;; autres utilitaires

(define (inCurrentEnv? var env)
   (assq var (car env)))

(define (inEnv? var env)
   (when env
      (if (assq var (car env))
     env
    (inEnv? var (cdr env)))))

(define (inGlobalEnv? var)
   (assq var (car *global-env*)))

(define (getVarAdress var env)
   (cassq var (car env)))
```

```
(define (incrindexvar)
   (set! *INDEXVAR* (1+ *INDEXVAR*)))

(define-method MiClosure 'compile
  (lambda(self)
     (comp-proc (getval self 'code)(getval self 'env))
     (setval self 'compiledCode (reverse *LAP*))
    #t))

(define-method MiClosure 'print-lap
  (lambda(self)
     (print-lap (getval self 'compiledCode))))

;;-----------------------------------------------------------------
----
;;
;;   le traitement de la compilation
;;
;;   la fonction qui fait le boulot
;;   env = ((var1 ... vark) ... (vark+1 ... varn))
;;   suite de frames contenant les environnements locaux
;;   le car de env contient l'environnement local,
;;   les autres doivent etre clos
;;
;;---------------------------------------------------------

;; pour compiler le code d'une lambda globale
(define (comp-proc expr env)
   (set! *LAP* nil)
   (gencode `(FENTRY))   ;; juste une etiquette. Ne sert en fait a
rien
   (set! *INDEXVAR* 0)
   (let ((env (cons (map (lambda(x)(cons x (incrindexvar)))
                         (car expr))
                    env)))
      (comp-begin (cdr expr) env)
      (gencode '(RTN))
    #t))

(define (comp-eval expr env)
    (cond
     ((null? expr) (gencode self `(MOV A0 ())))
     ((isConstant? expr) (comp-constant expr))
     ((atom? expr) (comp-var expr env))
     ((isSpecial? (car expr)) (comp-special expr env))
     (else
          (comp-apply (car expr) (cdr expr) env))))

(define (comp-constant expr)
```

```
  (if (isMNumber? expr)
    (gencode `(MOV A0 ,expr))
    (gencode `(MOV A0 ',expr))))

(define (comp-var var env)
 (cond
   ((inCurrentEnv? var env)
    (let ((i (getVarAdress var env)))
       (if (number? i)
         (gencode (list 'LOCAL 'A0 (1- i)))
         (gencode `(ENVIR A0 ,var)))))
   ((inEnv? var (cdr env))
        (gencode `(ENVIR A0 ,var)))
   (else (gencode `(GLOBAL A0 ,var)))))

(define (comp-apply fn args env)
  (if (and (symbol? fn)
           (not (inEnv? fn env))    ;; si c'est defini globalement
           (inGlobalEnv? fn))
     (let ((n (comp-evlis args env)))
        (=> (mval fn *global-env*) 'comp-apply fn n env))
    ;; pour l'instant pas de traitement des procedures non
globales
     (looerror 'comp-apply "ce n'est pas une procedure" fn)))

(define (comp-evlis args env)
   (comp-evlis1 args env 0))

(define (comp-evlis1 args env n)
  (ifn args
     (begin (gencode `(PUSH ,n)) n)
    (begin
      (comp-eval (car args) env)
      (gencode '(PUSH A0))
      (comp-evlis1 (cdr args) env (1+ n)))))

(define-method MiClosure 'comp-apply
 (lambda (self name n env)
   (let ((code (getval self 'code)))
    (gencode '(SAV))
    (gencode `(CALL_PROC name ,(length (car code))))
    (gencode '(REST)))))

(define-method MiSubr 'comp-apply
 (lambda (self name n env)
       (gencode              ;; le resultat de l'appel sera dans
A0
            (list 'CALL_PRIM name))
      ))
```

```
(define (comp-special expr env)
  (let ((fn (car expr))
        (args (cdr expr)))
   (case fn
      (if  (let ((et1  (gensym))
                 (et2  (gensym)))
             (comp-eval (car args) env)    ;; test
             (gencode `(JNI A0 ,et1))
             (comp-eval (cadr args) env)    ;; alors
             (gencode  `(JMP ,et2))
             (gencode  et1)
             (comp-eval  (caddr args) env)  ;; sinon
             (gencode  et2)))
      (begin (comp-begin (cdr expr) env))
      (set! (comp-set! (cdr expr) env))
)))

(define (comp-begin expr env)
  (if (or (null? expr)(null? (cdr expr)))
      (comp-eval (car expr) env)
      (begin
          (comp-eval (car expr) env)
          (comp-begin (cdr expr) env))))

(define (comp-set! expr env)
  (let ((var (car expr))
        (val (cadr expr)))
     (comp-eval val env)
     (cond
        ((inCurrentEnv? var env)
            (let ((i (getVarAdress var env)))
               (if (number? i)
                 (gencode (list 'SETLOCAL (1- i) A0))
                 (gencode `(SETENVIR ,var A0)))))
        ((inEnv? var (cdr env))
            (gencode  `(SETENVIR ,var A0)))
        (else (gencode `(SETGLOBAL ,var A0))))))

;;------------------------------------------------------------------
;;
;;    quelques exemples de fonctionnement
;;
;;------------------------------------------------------------------

;; carre
;(mdefine carre (Mlambda (x) (* x x)))
; (=> (mget 'carre) 'compile)
; (=> (mget 'carre) 'print-lap)
; (run '(carre 4))
```

```
;       (FENTRY 1)
;       (LOCAL A0 0)
;       (PUSH A0)
;       (LOCAL A0 0)
;       (PUSH A0)
;       (PUSH 2)
;       (CALL_PRIM *)
;       (RTN)

;; test avec le if
;; (mdefine testif (Mlambda (x) (if (= x 1003) t (1+ x))))
;  (=> (mget 'testif) 'compile)
;  (=> (mget 'testif) 'print-lap)
;       (FENTRY 1)
;       (LOCAL A0 0)
;       (PUSH A0)
;       (MOV A0 1003)
;       (PUSH A0)
;       (PUSH 2)
;       (CALL_PRIM =)
;       (JNI A0 g130)
;       (MOV A0 't)
;       (JMP g131)
;g130
;       (LOCAL A0 0)
;       (PUSH A0)
;       (PUSH 1)
;       (CALL_PRIM 1+)
;g131
;       (RTN)

; (mdefine fac (Mlambda(n)(if (= n 1000) 1001 (
;        * n (fac (1- n))))))
; (=> (mget 'fac) 'compile)
; (=> (mget 'fac) 'print-lap)
; (run '(fac 3))
;       (FENTRY)
;       (LOCAL A0 0)
;       (PUSH A0)
;       (MOV A0 1000)
;       (PUSH A0)
;       (PUSH 2)
;       (CALL_PRIM =)
;       (JNI A0 g128)
;       (MOV A0 1001)
;       (JMP g129)
;g128
;       (LOCAL A0 0)
;       (PUSH A0)
```

```
;       (LOCAL A0 0)
;       (PUSH A0)
;       (PUSH 1)
;       (CALL_PRIM 1-)
;       (PUSH A0)
;       (PUSH 1)
;       (SAV)
;       (CALL_PROC name 1)
;       (REST)
;       (PUSH A0)
;       (PUSH 2)
;       (CALL_PRIM *)
;g129
;       (RTN)
```

# 10. Problèmes avancés

## 10. 1 Fermetures

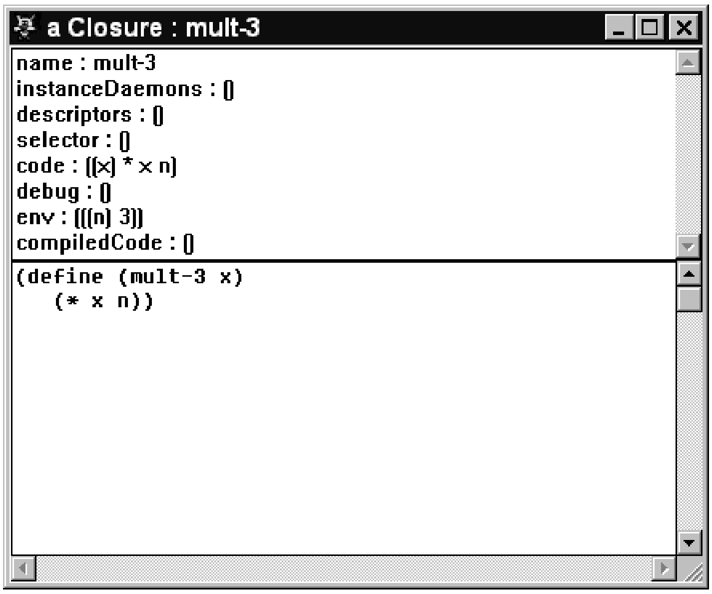
Bof!! Peut être un peu compliqué dans le cadre d'un cours normal.

Exemple:

```
(define (mult-n n)
    (lambda(x)(* x n)))

(define mult-3 (mult-n 3))

(mult-3 4) → 12
```




Principe:

1. On représente les fermetures sous la forme d'un couple < envglobal, code>

2. L'appel d'une fermeture consiste à appeler le code de la fermeture.

## 10. 2 Récursivité terminale

## 10. 3 Échappements et les exceptions ( ? ?)