Génération de code

Mathieu Lafourcade
LIRMM
version du 16/03/2003

Introduction

Nous aborderons la génération de code selon une approche fonctionelle. En effet, le générateur de code ayant besoin de la structure aborescente du programme, nous utiliserons à la place une forme équivalent préfixée (à la Lisp). La notion de schéma de compilation sera au cœur des transformations introduites.

Un schéma de compilation est une règle de réécriture de la forme:

```
Compile[<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 cibles 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.

Notre pseudo-langage Lisp dispose d'arguments notés entre crochets [a1 ,a2,..., an]. L'opérateur @ dans la génération de code indique qu'il faut évaluer l'expression qui suit.

La machine virtuelle - MV

La machine virtuelle que nous utilisons dispose d'une mémoire (fini, mais suffisement grande afin de ne pas être géné) et de trois registres R0, R1 et R2. La mémoire est un ensemble fini de N cellules. La première cellule a par définition le numéro 0, la dernière le numéro N-1. On parlera plutôt d'adresse mémoire que de numéro de cellule.

La pile est gérée dans la mémoire. La pile commence à l'adresse indique dans le registre spécial BP (base pointer). A priori, on ne touchera jamais à ce registre. le sommet de pile est indiqué dans le registre spécial SP (stack pointer). La pile est vide quand `SP = BP`. On a toujours `SP >= BP` (la pile est montante). Le compteur de programme est contenu dans un registre spécial PC (program counter).

Pour l'instant, nous avons donc les registres suivants : `R0, R1, R2, BP, SP, PC`

Nous introduirons dans la suite, un pointeur de cadre (frame pointer) : `FP`. Pour les comparaison et les sauts conditionnel, on dispose de trois registres booléens (1 bit) appelés drapeaux (flag) : `DPP, DE, DPG` (pour drapeau plus petit, drapeau égal, drapeau plus grand).

Modes d'adressage

Un adressage correspond soit à un source <src> d'où est lue une valeur, soit à une destination <dest> où une valeur est écrite.

Valeur constante

Une valeur constante s'écrit directement précédée par le symbole #. Par exemple :

```
#50
```

Correspond à la valeur 50.

une valeur constante ne peut être qu'une source (une telle destination n'aurait pas de sens).

Mode direct

D'une façon générale, les instructions qui ont besoin d'accéder à des données vont indiquer comme paramètres l'endroit ou se trouvent ces données. Les registres sont directement indiqués tel quels. Par exemple :

```
R0, R1 ou R2
```

S'il s'agit de sources <src>, ils correspondent respectivement à :

```
contenu(R0), contenu(R1) et contenue (R2)
```

Les adresses sont indiquées directement. Par exemple :

```
500
```

correspond pour ue source à la valeur contenue dans l'adresse 500 : `contenu(500)`

Mode indexé

Une adresse calculée par addition d'un déplacement c au contenu d'une registre R, s'écrit sous la forme c(R). Ainsi, l'instruction:

```
4(R0)
```

correspond à la valeur contenue à l'adresse : `4 + contenu(R0)` c'est-à-dire `contenu(4 + contenu(R0))`

Mode indirect

La forme indirecte est préfixée par *. Ainsi, l'instruction :

```
*R0
```

correspond à la valeur contenue à l'adresse `contenu(R0)` c'est-à-dire `contenu(contenu(R0))`. On a aussi :

```
*500
```

qui correspond à la valeur contenue à l'adresse contenue à l'adresse 500 c'est-à-dire `contenu(contenu(500))`

Mode indirect indexé

Il s'agit de la forme indirecte du mode indexée. Ainsi, l'instruction:

```
*4(R0)
```

correspond à la valeur contenue à l'adresse `contenu(4 + contenu(R0))` c'est-à-dire `contenu(contenu(4 + contenu(R0) + 4))`

Etiquettes et variables globales

Ce type symbole est inséré dans le code pour référencer un adresse particulière. On suppose que la machine virtuelle dispose d'une table <étiquette, adresse> qui permet de retrouver l'adresse correspondant à une étiquette donnée. En précédant une étiquette par le caractère @, on fait référence à son adresse. Par exemple :

```
@ETIQ
```

correspond à l'adresse ou se trouve l'étiquette `ETIQ`.

D'une façon similaire, les variables globales sont référencées comme des étiquettes. Il s'agit d'étiquettes qui se trouvent à un endroit définis à l'avance dans la mémoire. Par exemple, les variables peuvent commencer à l'adresse 0. Par exemple, nous avons les variables globales A, B et C :

```
0000       A   50          variable A dont le valeur est 50
0001       B   214         variable B dont le valeur est 214
0002       C   -250        variable C dont le valeur est -250
```

On a donc :

```
@C              l'adresse de C --> 0002
*@C             le contenu de l'adresse de C --> -250
```

Instructions

Les instructions ont soit 0, 1 ou 2 paramètres. Sauf indication contraire tous les modes d'adressage sont possibles. En général, il s'agit de la source `<src>` et de la destination `<dest>`.

Instructions de base

- `MOVE <src> <dest>`

  Charge `<src>` dans `<dest>`

  Les différents modes d'adressage présentés ci-dessus peuvent être utilisés. Par exemple :

  ```
  MOVE  #50 R0      charge la constante 50 dans R0
  MOVE  R0 R1       recopie le contenu de R0 dans R1
  MOVE  R0 500      charge le contenu de R0 à l'adresse 500
  MOVE  R0 *R1      charge le contenu de R0 en mémoire à l'adresse contenue dans R1
  MOVE  *R0 *R1     charge le contenu de la case mémoire contenue en R0 à l'adresse contenue dans R1
  ```

  Les accès aux registres et à la mémoire sont banalisés. Si on tient (par réalisme ?) à les distinguer, on réservera `MOVE` pour les registes, et on introduira `LEA` pour charger un registe depuis la mémoire et `STO` pour écrire le contenu d'un registe dans la mémoire.

- `ADD <src> <dest>`

  Ajoute `<src>` à `<dest>`, le resultat se trouve dans `<dest>`.

  Par exemple :

```
ADD  4(R0) *12(R1)
```

a pour effet de ranger la valeur `contenu(contenu(12 + contenu(R1))) + contenu(4 + contenu(R0))` dans la destination *12(R1).

- `SUB <src> <dest>`

  Soutrait `<src>` à `<dest>`, le resultat se trouve dans `<dest>`.

  Par exemple :

  ```
  SUB  4(R0) *12(R1)
  ```

  a pour effet de ranger la valeur `contenu(contenu(12+contenu(R1)))-contenu(4+contenu(R0))` dans la destination *12(R1).

- `MULT <src> <dest>`

  Multiplie `<src>` à `<dest>`, le resultat se trouve dans `<dest>`.

  Par exemple :

  ```
  MULT  4(R0) *12(R1)
  ```

  a pour effet de ranger la valeur `contenu(contenu(12 + contenu(R1))) * contenu(4 + contenu(R0))` dans la destination *12(R1).

- `DIV <src> <dest>`

  Divise `<dest>` par `<src>`, le resultat se trouve dans `<dest>`.

  Par exemple :

  ```
  DIV  4(R0) *12(R1)
  ```

  a pour effet de ranger la valeur `contenu(contenu(12 + contenu(R1))) / contenu(4 + contenu(R0))` dans la destination *12(R1).

- `INCR <dest>`

  Incrémente `<dest>` de 1

  Par exemple :

  ```
  INCR  *12(R1)
  ```

  a pour effet de ranger la valeur `contenu(contenu(12 + contenu(R1))) + 1` dans la destination *12(R1).

  Cette instruction est équivalente à (`ADD #1 <dest>`) qui suit, mais elle potentiellement plus efficace.

- `DECR <dest>`

  Décrémente `<dest>` de 1

  Par exemple :

  ```
  DECR  *12(R1)
  ```

  a pour effet de ranger la valeur `contenu(contenu(12+contenu(R1)))-1` dans la destination *12(R1).

  Cette instruction est équivalente à (`SUB #1 <dest>`) qui suit, mais elle potentiellement plus efficace.

Avec ces instructions, nous sommes capables d'écrire les instructions de pile. Toutefois, celles-ci peuvent raisonnablement être implémenté directement (code plus effiace et surtout plus clair).

Instructions de pile

- `PUSH <src>`

  Pousse sur la pile le contenu de `<src>`.

  Le code qui suit est équivalent à `PUSH <src>` :

  ```
  INCR  SP          on incrémente SP
  MOVE  <src> *SP   on met <src> à la nouvelle adresse de SP
  ```

- `POP <dest>`

  Depile le sommet de pile, et met l'information dans `<destination>`.

  Le code qui suit est équivalent à `POP <dest>` :

  ```
  MOVE  *SP <dest> on met le contenu de l'adresse de SP dans <dest>
  DECR  SP         on décrémente SP
  ```

Instructions de saut

- `JMP <dest>`

  Saute à l'adresse `<dest>`.

  Le code qui suit est équivalent à `JMP <dest>` :

  ```
  MOVE <dest> PC   on affecte le compteur de programme avec <dest>
  ```

- `JSR <etiq>`

  Saute à l'adresse de l'étiquette `<etiq>`. Empile l'adresse de retour sur la pile. On revient à l'instruction qui suit le saut avec l'instruction `RTN`

  Le code qui suit est équivalent à `JSR <etiq>` :

  ```
  PUSH  PC         on pousse l'adresse courante sur la pile
  JMP   @<etiq>    on saute
  ```

- `RTN`

  Saute à l'adresse contenue en sommet de pile. Supprime le sommet de pile.

  Le code qui suit est équivalent à `RTN` :

  ```
  MOVE *SP R0      adresse présente sur la pile --> R0
  DECR  SP         on décrémente SP
  JMP   *R0        saut à adresse contenue dans R0
  ```

`CMP <src1> <src2>`

Compare `<src1>` à `<src2>` et positionne les drapeaux `DPP`, `DE` et `DPG` de façon adéquate :

100 (si `<src1> < <src2>`),
010 (si `<src1> = <src2>`),
001 (si `<src1> > <src2>`),

- `JL, JEQ, JG, JLE, JGE, JNE <dest>`

  Saute à l'adresse `<dest>` si les drapeaux `DPP` sont respectivement à

  100 (plus petit),
  010 (égal),
  001 (plus grand),
  110 (plus petit ou égal),
  011 (plus grand ou égal),
  101 (non égal).

  On effectue un `AND` entre le schéma (ci-dessus) et l'état des drapeaux. Si le résultat est différent de 000 alors le saut est effectué. Les instructions de saut n'ont de sens que si elles sont précédées par un `CMP`. L'état initial des drapeaux est indéfini.

  Par exemples :
  On a 001 comme états de drapeaux et le test est `JGE : 001 and 011 = 001 (saut effectué)`
  On a 001 comme états de drapeaux et le test est `JLE : 001 and 110 = 000 (saut pas effectué)`

Instructions diverses

- `NOP`

  Instruction vide, ne fait rien

- `HALT`

  Termine l'execution de la machine virtuelle. Cette instruction doit au moins suivre la dernière ligne de code valide.

Schémas de compilation

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  R0 ;
Compile[<e2>] ;
PUSH  R0 ;
CompileOp[<op>]
```

On supposera que les résultats de toutes les opérations sont passées dans le registre `R0`. Il s'agit d'une convention ; une autre pourrait être choisie.

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  @<op>
```

Afin que ce mécanisme fontionne, il faut précéder chaque procédure (ou fonction) par un en-tête contenant son étiquette.

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

`CompileOP[<+>] -->`

```
POP  R1 ;
POP  R0 ;
ADD  R1 R0
```

Ce code remplace l'appel `JSR @+` .

En fait, dans le cas de Lisp 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 `R0` :

`CompileCste[<cste>] -->`

```
MOVE  #<cste> R0
```

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

`CompileVarg[<varg>] -->`

```
MOVE  *@<varg> R0
```

Exemple : Compiler l'expressions suivante: `((a + 4) * 2) * (3 - b)` et en supposant que a et b sont des variables globales. Soit soit forme préfixée `(* (* (+ 4 a) 2) (- 3 b))`

```
Compile[(* (* (+ 4 a) 2) (- 3 b))] -->

     Compile[(* (+ 4 a) 2)] ;
     PUSH R0 ;
     Compile[(- 2 b)] ;
     PUSH R0 ;
     CompileOp[*]

     Compile[(* (+ 4 a) 2)] -->

          Compile[(+ 4 a)];
          PUSH R0 ;
          CompileCste[2] ;
          PUSH R0 ;
          CompileOp[*] ;

          Compile[(+ 4 a)] -->

               CompileCste[4] ;
               PUSH R0 ;
               CompileVarg[a] ;
               PUSH R0 ;
               CompileOp[+] ;

               CompileCste[4] --> MOVE
               #4 R0

               CompileVarg[a] --> MOVE
               *@a R0

               CompileOp[+] --> JSR @+

          CompileCste[2] --> MOVE #2 R0

          CompileOp[*] --> JSR @*

     Compile[(- 3 b)] -->

          CompileCste[3] ;
          PUSH R0 ;
          CompileVarGlob[b] ;
          PUSH R0 ;
          CompileOp[-] ;

          CompileCste[3] --> MOVE #3 R0

          CompileVarg[b] --> MOVE *@b R0

          CompileOp[-] --> JSR @+

     CompileOp[*] --> JSR @*
```

Si on remet tout ça dans l'ordre on obtient : `Compile[(* (* (+ 4 a) 2) (- 3 b))] -->`

```
MOVE  #4 R0
PUSH  R0
MOVE  *@a R0
PUSH  R0
JSR   @+
PUSH  R0
MOVE  #2 R0
PUSH  R0
JSR   @*
PUSH  R0
MOVE  #3 R0
PUSH  R0
MOVE  *@b R0
PUSH  R0
JSR   @-
PUSH  R0
JSR   @*
```

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

```
MOVE  #4 R0
PUSH  R0
MOVE  *@a R0
PUSH  R0
POP   R1       à la place de JSR @+
POP   R0
ADD   R1 R0
PUSH  R0
MOVE  #2 R0
PUSH  R0
POP   R1       à la place de JSR @*
POP   R0
MULT  R1 R0
PUSH  R0
MOVE  #3 R0
PUSH  R0
MOVE  *@b R0
PUSH  R0
POP   R1       à la place de JSR @-
POP   R0
SUB   R1 R0
PUSH  R0
POP   R1       à la place de JSR @*
POP   R0
MULT  R1 R0
```

Fonctions de tests

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

Exemple : `(a > 4)`

```
Compile[(> a 4)] -->

     CompileVarg[a] --> MOVE  *@a R0
     PUSH  R0
     CompileCste[4] --> MOVE  #4 R0
     PUSH  R0
     CompileOp[>] --> JSR   @>
```

Soit :

```
      MOVE  *@a R0
      PUSH  R0 ;
      MOVE  #4 R0
      PUSH  R0 ;
      JSR   @>
```

Les fonction booléennes retournent 0 (faux) ou 1 (vrai) comme valeurs.

Affectation de variables globales

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

`Compile[(setf <var> <expr>)] -->`

```
  Compile[<expr>] ;
  MOVE R0 @<var>
```

Par exemple, nous avons pour `(setf a 533) -->`

```
        MOVE  #533 R0
        MOVE  R0 @a
```

L'état de la mémoire après exécution est :

```
0000       A   533
0001       B   214
0002       C   -250
```

Note : le code précédent qui pourrait s'optimiser :

```
        MOVE  #533 @a
```

Mais, une telle optimisation n'est immédiatement possible, si on a `(setf a (+ a 5)) -->`

```
        MOVE  *@a R0
        PUSH  R0
        MOVE  #5 R0
        PUSH  R0
        POP   R1
        POP   R0
        ADD   R1 R0
        MOVE  R0 @a
```

Structures de contrôles

Séquence

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

`Compile[(progn <expr1> <expr2> … <exprn>)] -->`

```
  Compile[<expr1>] ;
  Compile[<expr2>] ;
  …
  Compile[<exprn>] ;
```

Si alors sinon

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

`Compile[(if <test> <alors> … <sinon>)] -->`

```
soit
  etiq-sinon = etiq()
  etiq-fin = etiq()

  Compile[<test>]
  CMP R0 #0                    si R0 = 0 alors test faux
  JEQ @etiq-sinon
  Compile[<alors>]
  JMP @etiq-fin
etiq-sinon
  Compile[<sinon>
etiq-fin
```

Par exemple : `(if (< a 5) 1 2) -->`

```
01            MOVE  *@a R0    calcul de (< a 5)
02            PUSH  R0
03            MOVE  #4 R0
04            PUSH  R0
05            JSR   @>
06            CMP R0 1        le résultat de (< a 5) se trouve dans R0
07            JNE   @SINON1
08            MOVE  #1 R0
09    SINON1  JMP   @FIN1
10      FIN1  MOVE  #2 R0
```

Boucle "tant que"

Le 'tant que' utilise une technique semblable.

`CompileOp[(while <test> <expr>)] -->`

```
soit
  etiq-boucle = etiq()
  etiq-fin = etiq()

etiq-boucle
  Compile[<test>] ;
  JEQ R0 @etiq-fin ;
  Compile[<expr>] ;
  JMP @etiq-boucle ;
etiq-fin
```

Par exemple : `(while (< a 5) (setf a (+ a 2)))`

```
01    BOUCLE1  MOVE  *@a R0    Calcul de (< a 5)
02             PUSH  R0
03             MOVE  #5 R0
04             PUSH  R0
05             JSR   @<
06             JEQ R0 @FIN1
07             MOVE  *@a R0    Calcul de (setf a (+ a 2))
08             PUSH  R0
09             MOVE  #2 R0
10             PUSH  R0
11             POP   R1
12             POP   R0
13             ADD   R1 R0
14             MOVE  R0 @a     fin de (setf a (+ a 2))
15             JMP   @BOUCLE1
16       FIN1
```

Boucle "pour"

Le 'pour' u 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 - cas 1), soit la valeur finale peut évoluer au cours de la boucle et il faut alors la recalculer systématiquement (cas 2).

Cas 1 : `<expr-fin>` n'est calculée qu'une seule fois

`Compile[(for (<var> <expr-init> <expr-fin>) <expr>)] -->`

```
soit
  etiq-boucle = etiq()
  etiq-fin = etiq()

  Compile[<expr-init>] ;
  MOVE  R0 @<var> ;            <expr-init> --> <var>
  Compile[<expr-fin>] ;        <expr-fin> est une cste
  PUSH  R0 ;                   <expr-fin> --> pile
etiq-boucle
  MOVE @<var> R0 ;             <var> --> R0
  POP R1 ;                     <expr-fin> --> R1
  SUB R0 R1 ;                  R1 - R0 --> R1
  JLE R1 @etiq-fin ;
  Compile[<expr>] ;
  INCR @<var>                  on incrémente <var>
  JMP @etiq-boucle ;
etiq-fin
```

Par exemple : `(for (i 0 10) (setf a (/ a 2))) -->`

```
01    BOUCLE1  MOVE            a faire
02       FIN1
03
```

Cas 2 : `<expr-fin>` est calculée à chaque tour

```
Compile[(for (<var> <expr-init> <expr-fin>) <expr>)] -->
```

```
soit
  etiq-boucle = etiq()
  etiq-fin = etiq()

  Compile[<expr-init>] ;
  MOVE R0 @<var> ;              <expr-init> --> <var>
etiq-boucle
  Compile[<expr-fin>] ;         <expr-fin> est recalculé
  MOVE R0 R1 ;                  <expr-fin> --> R1
  MOVE  @<var> R0 ;
  SUB R0 R1 ;                   R1 - R0 --> R1
  JLE R1 @etiq-fin ;
  Compile[<expr>] ;
  INCR @<var> ;                 on incrémente <var>
  JMP @etiq-boucle ;
etiq-fin
```

Par exemple : `(for (i 0 (+ a 1)) (setf a (/ a 2)) ) -->`

```
01    BOUCLE1   MOVE            a faire
02        FIN1
03
```

Fonctions

Les fonctions se trouvent à deux endroits :

1. Dans la déclaration de la fonction
1. Lors de son utilisation (l'appel)

Compilation d'une déclaration d'une fonction

Pour l'instant on compile les fonctions sans paramètres ni variables locales. Une déclaration d'un sous-programme s'exprime ainsi :

```
Compile[(defun <proc> <expr>)] -->
```

où `<expr>` est généralement de la forme `(progn <expr1> .. <exprn>)`

```
<proc>                          l'etiquette du sous-programme
  Compile[<expr>] ;
  RTN
```

Compilation d'un appel de sous-programme

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.

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 la pile.

Passage de paramètres à la Fortran

Déclaration d'un sous-programme :

```
Compile[(defun <proc> (<x1> .. <xn>) <expr>)] -->
```

```
soit
  debut_proc = etiq(debut_,
<proc>)
  var_x1 = etiq(v, <proc>, _,
<x1>)
  ...
  var_xn = etiq(v, <proc>, _,   l'etiquette du sous-programme
<xn>)

<proc> ;
  JMP @debut_proc
var_x1   0 ;
  ...                           récupère l'adresse de retour
var_xn   0 ;                    récupère les variables
debut_proc                      on les stocke en mémoire
  POP R2 ;
  POP R0 ;
  MOVE R0 @var_xn ;             remet l'adresse de retour sur la pile
...
  POP R0
  MOVE R0 @var_x1 ;
  PUSH A2
  Compile[expr] ;
  RTN
```

Par exemple :

```
01    BOUCLE1   MOVE            à faire
02        FIN1
03
```

Appel d'un sous-programme :

```
Compile[(<proc> <expr1> .. <exprn>)] -->
```

```
  Compile[<expr1>] ;
  PUSH R0 ;
  …
  Compile[<exprn>] ;
  PUSH R0 ;
  JSR <proc>
```

Par exemple :

```
01    BOUCLE1   MOVE            à faire
02        FIN1
03
```

Problèmes

1) pas de récursivité possible ;

2) l'accès aux valeurs passées en argument lors de la compilation du corps de la procédure. Comment y accéder ?

Par exemple pour (1), considérons le code Lisp suivant :

```
(defun foo (x)
(if (= x 0)
1
(+ 1 (foo (- x 1))
))

(foo 5)
```

Réponse : pour le problème (2), on peut utiliser un **environnement**. 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, s] == v` si `s` 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.

On ajoutera donc 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> <expr1> <expr2>),<env>] -->
```
où `<op>` est une opération primitive

```
  Compile[<expr1>, <env>] ;
  PUSH R0 ;
  Compile[<expr2>, <env>] ;
  PUSH R0 ;
  CompileOp[<op>, <env>]
```

Par exemple : ?

```
01    BOUCLE1   MOVE            à faire
02        FIN1
03
```

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 :

```
Compile[(defun (<proc> <x1> .. <xn>) <expr>),<env>] -->
```

où `<op>` est une opération primitive

```
soit
  debut_proc = etiq(debut_, <proc>)
  var_x1 = etiq(v, <proc>, _, <x1>)
   ...
  var_xn = etiq(v, <proc>, _, <xn>)
  <nenv> = {(<x1>, @var_x1), .., (<xn>,
@var_xn)}

<proc> ;
  JMP @debut_proc
var_x1   0 ;
  ...
var_xn   0 ;
debut_proc
  POP R2 ;
  POP R0 ;
  MOVE R0 @var_xn ;
...
  POP R0
  MOVE R0 @var_x1 ;
  PUSH R2
  Compile[expr,<nenv>] ;
  RTN
```

Par exemple :

```
01    BOUCLE1   MOVE
02       FIN1
03
```

à faire

Accès aux variables locales

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

```
Compile[<ident>] -->

  MOVE @<ident> R0
```

se compile maintenant à l'aide des environnements :

```
Compile[<ident>, <env>] -->

  MOVE (loc @at[<env>,<ident>]) R0
```

Exemple :

```
01    BOUCLE1   MOVE
02       FIN1
03
```

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 R0 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
```

Passages de paramètres par la pile

On utilise la pile pour le passage de paramètres. Le principe consiste à laisser les variables sur la pile. On obtient alors :

```
Compile[(defun <proc> (<x1> … <xn>) <expr>), <env>] -->
```

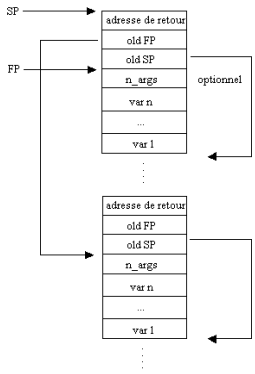
```
debut_proc
  Compile[<expr>, <env>] ;
  RTN
```

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. Il s'agit de la notion de bloc de pile (stack frame). on remarquera que ce n'est pas la seule solution possible, mais c'est une des plus simples et qui donne un code généré les plus lisibles.

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> <expr1> … <exprn>), <env>] -->
```

```
Soit n le nombre d'arguments
  Compile[<expr1>, <env>] ;
  PUSH R0
  Compile[<expr2>, <env>] ;
  PUSH R0
  …
  Compile[<exprn>, <env>] ;
  PUSH R0
  PUSH #<n>                    // on pousse le nb d'arguments

  MOVE FP R1                   // on recupère l'ancien FP
  MOVE SP FP                   // le nouvel FP = SP
  MOVE SP R2                   // on recupère l'ancien SP
  SUB  @n R2                   // R2 = SP - (n + 1)
  SUB  1 R2
  PUSH R2                      // on sauve l'ancien SP
  PUSH R1                      // on sauve l'ancien FP
                               // fin de l'entete d'appel

  JSR @<proc>                  // appel

                               // début de la restauration
    POP R1                     // on recupère l'ancien FP
  POP R2                       // on recupère l'ancien SP
  MOVE R1 FP                   // FP est restauré à son ancienne valeur
  MOVE R2 SP                   // on a supprimé toutes les variables

                               // fin de la restauration
```

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

$$\mathbf{var_i = FP - (n + 1) + i \ // \ on\ compte\ i\ à\ partir\ de\ 0}$$

On peut utiliser avec profit l'instruction `(local <n> <reg>)` qui va chercher le $n^{ième}$ élément sur la pile à partir de FP et le place dans le registre `<reg>`. Ainsi la compilation d'une procédure avec passage de paramètres se compile ainsi:

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

```
soit n le nombre de paramètres
<nenv> = {(<x1>,-(n+1)+1),..,
    (<xi>,-(n+1)+i),..,              // - 2 car -(n+1)+(n-1) = -2
    (<xn>,-2)}                       // l'etiquette de la fonction
                                     // ne sert à rien en fait
<proc>

  FENTRY <proc>
```

```
  Compile[<expr>,<nenv>]

  RTN
```

Maintenant, l'**accès aux variables** s'écrit ainsi :

```
        Compile[<ident>,<env>] -->

LOCAL @at[<env>,<ident>] R0
```

De même l'**affectation** est donnée par :

```
        Compile[(setq <ident> <expr>), <expr>] -->

Compile[<expr>, <env>]
SETLOCAL R0 @at[<env>, <ident>]
```

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 fonction 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 `let` au début de la définition d'une fonction.

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

```
        (define (f x)
        (let (z y)
        (setq z (1+ x)
        (setq y 0)
        (while (> y z)

        ...
        (setq y (1+ y)
        )
        ))
```

On a donc le schémas suivant :

```
        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 @at[<env>,<ident>] R0
```

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 fonction (mais qu'elle n'a pas été définie ainsi), on compile non pas un appel direct, mais un appel indirect via `R0`.

```
        Compile[(<expr> <e1> ... <en>), <env>] -->

soit n = nombre d'arguments

Compile[<e1>,<env>]
PUSH R0
…
Compile[<en>,<env>]               // début de l'entête d'appel
PUSH R0

PUSH #<n>                         // nombre d'arguments

Compile[<expr>, <env>]            // insertion de la compilation de <expr>
                                  // Attention: résultat dans A0

MOVE FP R1                        // on recupère l'ancien FP
MOVE SP FP                        // le nouvel FP = SP
MOV SP R2                         // on recupère l'ancien SP
DIFF #<n> R2
DECF R2                           // R2 = SP - (n + 1)

PUSH R2                           // on sauve l'ancien SP
PUSH R1                           // on sauve l'ancien FP

                                  // fin de l'entete d'appel

JSR R0                            // appel indirect via R0

                                  // début de la restauration
                                  // du contexte d'exécution

                                  // on recupère l'ancien FP
POP R1                            // on recupère l'ancien SP
POP R2                            // FP est restauré à son ancienne valeur
MOVE R1 FP                        // on a supprimé toutes les variables
MOVE R2 SP
                                  // 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.

Fonctions auxilliaires (labels)

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 Lisp:

à faire

```
(defun indice (lst x)
(labels ((indicel (lst x i)
(cond
((null? lst) ())
((equal? (car lst) x) i)
(else (indicel (cdr lst) x (1+ i)))))

) ;; du labels

(indicel lst x 0)
)

;; appel

(indice '(a b c d e) 'c) --> 2

;;
;; autre exemple

(define (f x)

(define (g z)

(set! x (1+ z))

x)

(+ x (g 3)))

 ;; appel

(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.

1. On compile l'appel à la procédure locale comme s'il s'agissait d'une procédure passée dans une variable.
1. Les variables locales à la procédure interne sont récupérées normalement sur la pile dans un bloc de pile local.
1. 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.
1. 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> ¹ ()
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 environnements

-->

<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> appartient au 1er élément de <env> // variable locale

-->

LOCAL @at[<env>,<ident>]) R0

Compile[<ident>,<env>]
et <ident> n'appartient pas à <env> // variable purement globale

-->

MOVE @<ident> R0
```

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

```
Compile[<ident>,<env>]
et <ident> n'appartient pas au 1er élément de <env> // variable locale/globale
et <ident> appartient à <env> (élément suivant)

soit var_x = at[<env>, <ident>]
et n = niveau d'emboitement de <ident>

-->

MOVE (FP+1) R1 // on prend l'ancien FP (indirect indexé)
MOVE @n R2

recherche

JEQ R2 trouve // R2 = 0

MOV (R1+1) R1 // recupere les FP

DECF R2 // on decremente R2

JMP recherche

trouve

MOV FP R2 // on sauve FP

MOVE R1 FP // FP prend la valeur du FP "global"

LOCAL @var_x R0 // on met le resultat dans R0

MOV R2 FP // FP reprend sa valeur
```

C'est tout !

Structures de données

Pour la compilation des structures de données, il existe 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).

Tableaux et Compilation statique

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
```

2. Accès en lecture:

```
Compile[<tab>[<expr>],<env>] -->

Compile[<expr>, <env>];

MOVE R1 <tab>); // recupere l'adresse de base

ADD R0 R1; // ajoute l'offset

(MOV A0 (A1)); // recherche indirecte de l'information.
```

3. Accès 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
```

Tableaux et Compilation semi-dynamique

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)
```

2. Accès en lecture

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

```
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.
```

3. Accès 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
```

Tableau et 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.
```

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
```

Record (ou struct)

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 à 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
```

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

Objets

Même structure que pour les records.