INFO703
TP - Compilation

Rapport
Génération de code pour un sous-ensemble du langage λ-ada.

Donat-Bouillud Vivien
Caballol Cédric

Introduction :

Ce TP a pour objectif de générer du code pour la machine à registres décrite dans le cours grâce à un arbre abstrait et aux outils JFlex et CUP. Il y a deux exercices complémentaires :

- Dans le premier, on génère du code pour les expressions arithmétiques sur les nombres entiers. Par exemple :

```
let prixHt = 200;
let prixTtc =  prixHt * 119 / 100 .
```

La génération de code correspondant à ces deux lignes est la suivante :

```
DATA SEGMENT
    prixHt DD
    prixTtc DD
DATA ENDS
CODE SEGMENT
    mov eax, 200
    mov prixHt, eax
    mov eax, prixHt
    push eax
    mov eax, 119
    pop ebx
    mul eax, ebx
    push eax
    mov eax, 100
    pop ebx
    div ebx, eax
    mov eax, ebx
    mov prixTtc, eax
CODE ENDS
```

- Dans le second, on étend la génération de code aux opérateurs booléens, de comparaison, aux boucles et aux conditionnelles correspondant au sous-ensemble du langage λ-ada utilisé pour le TP précédent. Par exemple :

```
let a = input;
let b = input;
while (0 < b)
do (let aux=(a mod b); let a=b; let b=aux );
output a
.
```

La génération de code correspondant à ces six lignes est la suivante :

```
DATA SEGMENT
    b DD
    a DD
    aux DD
DATA ENDS
CODE SEGMENT
    in eax
    mov a, eax
    in eax
    mov b, eax
debut_while_1:
    mov eax, 0
    push eax
    mov eax, b
    pop ebx
    sub eax,ebx
    jle faux_gt_1
    mov eax,1
    jmp sortie_gt_1
faux_gt_1:
    mov eax,0
sortie_gt_1:
    jz sortie_while_1
    mov eax, b
    push eax
    mov eax, a
    pop ebx
    mov ecx,eax
    div ecx,ebx
    mul ecx,ebx
    sub eax,ecx
    mov aux, eax
    mov eax, b
    mov a, eax
    mov eax, aux
    mov b, eax
    jmp debut_while_1
sortie_while_1:
    mov eax, a
    out eax
CODE ENDS
```

Analyseur syntaxique et génération de l'arbre :

Notre grammaire est capable comme demandé de reconnaître les expressions arithmétiques sur les nombres entiers, ainsi que les opérateurs booléens, ceux de comparaison, la boucle et la conditionnelle.

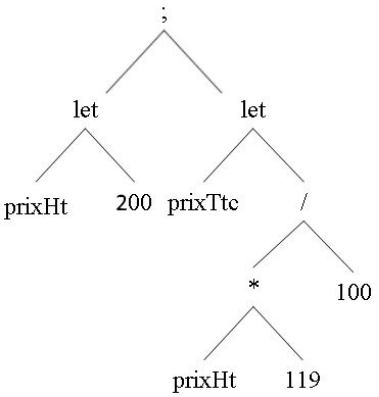
exemple de notre fichier .cup

```
/* grammaire */
// un pg est une sequence d'insctions terminee par point
program ::=
    sequence:s POINT  {: RESULT= s; :}
    ;

// une sequence est une suite d'instructions separes par des point virgules
sequence ::= expression:e1 SEMI sequence:e2            {: RESULT= new Arbre(";", e1, e2); :}
           | expression:e                              {: RESULT= e; :}
           |
           ;

// une expession est soit une affectation ,une
expression ::= expr:e   {: RESULT= e; :}
             | LET IDENT:nom EGAL expr:e           {: RESULT= new Arbre("let", new Arbre(nom,null,null), e); :}
             | WHILE expr:cond DO expression:e     {: RESULT= new Arbre("while", cond, e); :}
             | IF expr:cond THEN expression:a1 ELSE expression:a2 {: RESULT= new Arbre("if", cond, new Arbre("alt", a1, a2)); :}
             | error // reprise d'erreurs
             ;

//
expr ::= NOT:op expr:e          {: RESULT= new Arbre("not", e, null); :}
       | expr:e1 AND expr:e2    {: RESULT= new Arbre("and", e1, e2); :}
       | expr:e1 OR expr:e2     {: RESULT= new Arbre("or", e1, e2); :}
       | expr:e1 EGAL expr:e2   {: RESULT= new Arbre("=", e1, e2); :}
       | expr:e1 GT expr:e2     {: RESULT= new Arbre(">", e1, e2); :}
       | expr:e1 GTE expr:e2    {: RESULT= new Arbre(">=", e1, e2); :}
       | expr:e1 LT expr:e2     {: RESULT= new Arbre("<", e1, e2); :}
       | expr:e1 LTE expr:e2    {: RESULT= new Arbre("<=", e1, e2); :}
       | expr:e1 PLUS expr:e2   {: RESULT= new Arbre("+", e1, e2); :}
       | expr:e1 MOINS expr:e2  {: RESULT= new Arbre("-", e1, e2); :}
       | expr:e1 MUL expr:e2    {: RESULT= new Arbre("*", e1, e2); :}
       | expr:e1 DIV expr:e2    {: RESULT= new Arbre("/", e1, e2); :}
       | expr:e1 MOD expr:e2    {: RESULT= new Arbre("%", e1, e2); :}
       | MOINS expr:e           {: RESULT= new Arbre("-", e, null); :}  %prec MOINS_UNAIRE
       | OUTPUT expr:e          {: RESULT= new Arbre("output", e, null); :}
       | INPUT                  {: RESULT= new Arbre("input", null, null); :}
       | NIL                    {: RESULT= new Arbre("nil", null, null); :}
       | ENTIER:n               {: RESULT= new Arbre(n.toString(), null, null); :}
       | IDENT:id               {: RESULT= new Arbre(id, null, null); :}
       | PAR_G sequence:e PAR_D {: RESULT= e; :}
       ;
```

Les différents symboles remontent dans la séquence "program"
et celui-ci est parsé sous forme d'arbre dans le programme Main.

```
parser p = new parser (yy);
Symbol s = p.parse( );
Arbre arb = (Arbre)s.value;
```

On peut afficher ensuite cet arbre via un parcours préfixe.
Exemple: avec la séquence
let prixHT = 200;
let prixTtc = prixHT * 119 / 100 .
On obtient l'arbre suivant:



Qui va s’afficher ainsi :

```
/opt/jdk1.8.0_181/bin/java ...
;
let
prixHT
200
let
prixTtc
/
*
prixHT
119
100

Process finished with exit code 0
```

Génération de code :

Philosophie:

Premièrement, on effectue un premier parcours d’arbre pour détecter les identificateurs, grâce à la fonction *chercheLetsPrefixement()* , puis on les écrits dans la partie DATA SEGMENT :

```
/* génération de code */
FileWriter fw = new FileWriter(new File(args[1]));
try{

    // 1) DATA SEGMENT
    fw.write( str: "DATA SEGMENT");
    fw.write(System.lineSeparator()); //new line
    // on parcourt l'arbre pour detecter les let
    Set<String> listeIdentificateurs = new HashSet<~>();
    listeIdentificateurs = arb.chercheLetsPrefixement(listeIdentificateurs);
    for (String id : listeIdentificateurs){
        fw.write(String.format("\t%s DD",id));
        fw.write(System.lineSeparator()); //new line
    }
    fw.write( str: "DATA ENDS");
    fw.write(System.lineSeparator()); //new line
```

Puis, on génère le code associé dans la partie CODE SEGMENT :

```
    // 2) CODE SEGMENT
    fw.write( str: "CODE SEGMENT");
    fw.write(System.lineSeparator()); //new line
    /* CODE */
    arb.genereCode(fw);
    fw.write(System.lineSeparator());

    fw.write( str: "CODE ENDS");
    fw.write(System.lineSeparator()); //new line

    fw.close();
}catch (IOException ex){
    ex.printStackTrace();
}
```

pour ceci, on utilise la fonction *generecode()* qui pour chaque noeud de l'arbre, va générer le code associé en fonction de son type (grâce à une structure "switch-case")
remarque: La philosophie du "let" est d'effectuer l'affectation à l'identificateur, puis, de push cette valeur dans la pile:

```
case "let":
    if (this.droite != null) {
        this.droite.genereCode(fw);
        fw.write( str: "\tpop eax");
        fw.write(System.lineSeparator());
        fw.write( str: "\tmov " + this.gauche.valeur + ", eax");
        fw.write(System.lineSeparator());
        fw.write( str: "\tpush eax");
        fw.write(System.lineSeparator());
    }
    break;
```

Autre subtilitée, à chaque fois qu'on reconnaît une valeur, on la "push" dans la pile (cas par défaut du switch)

```
default:    // Si on est pas sur un identificateur (et donc une valeur)
    fw.write( str: "\tmov eax, " + this.valeur);
    fw.write(System.lineSeparator());
    fw.write( str: "\tpush eax"); // on la met dans la pile
    fw.write(System.lineSeparator());
    break;
```

ce qui à pour effet d'avoir une redondance de "push, pop" dans notre programme, mais qui a le mérite de marcher.

Exemples de génération de code

Utilisation : Pour générer le fichier generatedCode.asm, il suffit de lancer le main avec comme premier paramètre le nom de fichier contenant le code à analyser (entrée) et comme deuxième paramètre le nom du fichier dans lequel on veut écrire le code généré (sortie) par exemple:

avec test.txt :
let prixHT = 200;
let prixTtc = prixHT * 119 / 100 .

On obtient donc le fichier generatedCode.asm suivant:

```
DATA SEGMENT
  prixTtc DD
  prixHT DD
DATA ENDS
CODE SEGMENT
  mov eax, 200
  push eax
  pop eax
  mov prixHT, eax
  push eax
  pop eax
  mov eax, prixHT
  push eax
  mov eax, 119
  push eax
  pop eax
  pop ebx
  mul eax, ebx
  push eax
  mov eax, 100
  push eax
  pop eax
  pop ebx
  div ebx, eax
  mov eax, ebx
  push eax
  pop eax
  mov prixTtc, eax
  push eax
CODE ENDS
```

exemple d’utilisation du if:

avec test.txt :

```
let a = input;
let b = input;
if (3 <= 2) then (output a) else (output b);
.
```

On obtient le fichier generatedCode.asm suivant:

```
DATA SEGMENT
  a DD
  b DD
DATA ENDS
CODE SEGMENT
  in eax
  push eax
  pop eax
  mov a, eax
  push eax
  pop eax
  in eax
  push eax
  pop eax
  mov b, eax
  push eax
  pop eax
  mov eax, 2
  push eax
  mov eax, 3
  push eax
  pop eax
  pop ebx
  sub eax, ebx
  jle vrai_lt_1
  mov eax, 0
  jmp sortie_lt_1
vrai_lt_1:
  mov eax, 1
sortie_lt_1:
  jnz debut_then1
  jmp debut_else1
debut_then1:
  mov eax, a
  out eax
  jmp fin_if1
```

```
debut_else1:
  mov eax, b
  out eax
  jmp fin_if1
fin_if1:
CODE ENDS
```

exemple d'utilisation du while:

avec test.txt :

```
let a = input;
let b = input;
while (b > 0)
do (let aux=(a mod b); let a=b; let b =aux );
output a
.
```

On obtient le fichier generatedCode.asm suivant:

```
DATA SEGMENT
  a DD
  b DD
  aux DD
DATA ENDS
CODE SEGMENT
  in eax
  push eax
  pop eax
  mov a, eax
  push eax
  pop eax
  in eax
  push eax
  pop eax
  mov b, eax
  push eax
  pop eax
debut_while1:
  mov eax, 0
  push eax
  mov eax, b
  push eax
  pop eax
  pop ebx
  sub eax, ebx
  jg vrai_gt_1
```

```
  mov eax, 0
  jmp sortie_gt_1
vrai_gt_1:
  mov eax, 1
sortie_gt_1:
  jz sortie_while1
  mov eax, b
  push eax
  mov eax, a
  push eax
  pop eax
  pop ebx
  mov ecx, eax
  div ecx, ebx
  mul ebx, ecx
  sub eax, ebx
  push eax
  pop eax
  mov aux, eax
  push eax
  pop eax
  mov eax, b
  push eax
  pop eax
  mov a, eax
  push eax
  pop eax
  mov eax, aux
  push eax
  pop eax
  mov b, eax
  push eax
  jmp debut_while1
sortie_while1:
  pop eax
  mov eax, a
  out eax
CODE ENDS
```

NB : Vous pouvez trouver une version du Projet sur github:
https://github.com/Donatbov/Compilation_TP