Université Bordeaux
Licence, module Compilation, 2016/2017

TD5 – Déclarations/appels de fonctions

Exercice 5.1

La redondance de code est une des premières choses à éviter dans un programme. Pour cela, on utilise souvent des fonctions qui factorisent le code en portions réutilisables. Ici, on s'intéresse de nouveau aux expressions du calcul propositionnel. On introduit cette fois-ci des fonctions booléennes prenant en paramètre un certain nombre de variables et renvoyant le résultat d'une expression faisant uniquement intervenir les variables en paramètre.

Exemple : `def ma_fonction( a, b, c ) = not a or (b -> c <-> a)` est une fonction correcte qui pour les valeurs `a,b,c` en paramètre renvoie `not a or (b -> c <-> a)`. `def ma_fonction( a, b, c, d ) = a or e -> 1` est incorrecte car elle utilise la variable `e` qui n'est pas un argument.

Un programme est alors un suite d'affectations de variables et de définitions de fonctions, suivie d'une expression.

L'objectif de cet exercice est de réaliser un interpréteur sur ce langage qui sera donc amené à traiter un programme tel que :

```
p = 0;
q = 1;
def xor( a1, a2 )= (a1 and not a2) or (a2 and not a1);
def fon2 ( a, b, z )= (a or b) and not xor(b,z) and xor(a,z);
t = fon2( p, q, 1);
fon2( p and not q, q, not t) and not t
```

On supposera dans un premier temps que les noms des paramètres des fonctions sont distincts des noms des variables du programme.

```
start: Progr ;

Progr : Instr_List Expr

Instr_List : %empty
           | Instr_List Instr
Instr : Affect ';'
      | Func_Def ';'
Affect : Var '=' Expr
Func_Def : Def Var '(' Arg_L ')' '=' Expr
Arg_L : %empty
      | Arg_Ln
Arg_Ln : Var
       | Arg_L ',' Var
Expr_L : %empty
       | Expr_Ln
Expr_Ln :Expr
        | Expr_L ',' Expr
```

```
Expr : True | False
     | Var
     | Var '(' Expr_L ')'
     | Expr And Expr
     | Expr Or Expr
     | Expr Imp Expr
     | Expr Equiv Expr
     | Not Expr
     | '(' Expr ')'
```

Exercice 5.2

Dans un langage de programmation un peu plus complet, plusieurs variables différentes, en argument d'une fonction ou en variable globale par exemple, peuvent porter le même nom. Comment tenir compte de cette difficulté ?

Exercice 5.3

Dans l'ambition de se rapprocher d'un format de programme plus générique sans quitter la simplicité du calcul propositionnel, rajouter à la grammaire la possibilité d'avoir :
- une suite de déclarations de variables globales
- une séquence d'instructions dans le corps d'une fonction plutôt qu'une simple expression.

On se placera dans la grammaire suivante :

```
start: Progr ;

Progr : VarLoc FuncDef_L Instr_L Expr
VarLoc : %empty
       | Var Arg_L ';'
Instr_L : %empty
        | Instr_L Instr
Instr : Affect ';'
Affect : Id '=' Expr
FuncDef_L : %empty
          | FuncDef_L FuncDef
FuncDef : Def Id '(' Arg_L ')' VarLoc Instr_L Expr ';'
...
```

(elle se termine comme la grammaire de l'exercice 1). Un programme débutera par la déclaration de variables globales (éventuellement utilisables dans le corps de fonctions). Suivra la définition de fonctions puis une suite d'instructions (ici on a uniquement des affectations), puis l'expression à calculer.

Exemple :

```
var a, b, c;
def f1(a)
  var b;
  c = a and c;
```

```
  b = not a;
  c -> b;
def f2(x,y)=
  var u, v;
  u = x or not y;
  v = not x or y;
  u and v;
a = 1;
b = 0;
c = 0;
a = f1(a);
f2(b,c)
```

On pourra s'assurer qu'une variable est bien déclarée avant d'être utilisée.