TP n° 3
Projet de langage

Organisation

Le projet se fait par binôme.

Chaque groupe déposera sur Madoc son projet au plus tard le 1 décembre 2010.
Le projet contiendra un rapport et du code source intégral (fichiers C, flex et bison) de l'interpréteur développé lors du projet ainsi que quelques fichiers d'exemples.
Le rapport devra discuter les choix d'implémentation (grammaires, structures de données et algorithmes) et documenter l'utilisation de l'interpréteur. Le contenu de la section 2 devra obligatoirement être implémenté par l'interpréteur, mais il n'est pas interdit d'y ajouter des améliorations que vous pourriez imaginer, ou de traiter des parties de la section 3.

Vous avez quatre séances de TP pour réaliser ce projet.

1 Introduction

Le but de ce projet est de programmer un interpréteur pour un petit langage, complètement décrit dans la section 2. L'interpréteur est appelé en ligne de commande avec, en argument, le nom d'un fichier à interpréter : ./interpreteur monfichier.txt. Par exemple, si le fichier monfichier.txt contient le texte suivant,

```
print("Entrez un nombre : ") ;
X=input ; Y=1 ; Z=1 ;
while (Y<X) { Z=Z*Y ; Y=Y+1 ; }
print(X) ; print(" ! vaut ") ; print(Z) ; print("\n") ;
```

Le résultat d'une exécution possible sera (la partie en gras est celle entrée par l'utilisateur) :

```
Entrez un nombre : 6
6 ! vaut 720
```

Le projet doit être programmé en C en utilisant les logiciels flex et bison pour l'analyse lexicale et syntaxique. Les actions du fichier bison doivent construire une représentation mémoire du programme (on utilisera, par exemple, des listes chaînées pour représenter les blocs d'instructions et des arbres pour les expressions). Une fois l'analyse syntaxique terminée, une fonction C exécute le programme en parcourant cette représentation.
Remarquez que certaines parties de la représentation doivent être parcourues plusieurs fois en cas de boucle ***while***.

2 Le langage

Dans notre langage, les espaces, tabulations et retours à la ligne ne sont pas significatifs. De plus, la séquence // indique le début d'un commentaire qui se termine à la fin de la ligne.

2.1 Les instructions

Un programme se compose d'une séquence d'instructions. Certaines instructions (affectation, affichage sur l'écran) se terminent par un point-virgule ; tandis que d'autres (boucles, tests) se terminent par le marqueur de fin de bloc }.

Les affectations. L'instruction d'affectation s'écrit simplement *variable=expression* ;. Un nom de variable est composé d'un nombre arbitraire de lettres et de chiffres, le premier caractère étant forcément une lettre (pour ne pas confondre avec le début d'un nombre). Contrairement au C, les variables n'ont pas besoin d'être déclarées avant d'être utilisées. Elles ont toutes implicitement le type «nombre flottant» double du C.

Les Tests. Un test s'écrit **if** (*expr*) { $inst_1 \cdots inst_n$ } ou **if** (*expr*) { $inst_1 \cdots inst_n$ } **else** { $inst'_1 \cdots inst'_m$ }. Les instructions $inst_1$ à $inst_n$ sont exécutées si la valeur de l'expression *expr* est non nulle. Si cette valeur est nulle et qu'une branche **else** est spécifiée, alors les instructions $inst'_1$ à $inst'_m$ sont exécutées.

Les boucles. Une boucle s'écrit **while** (*expr*) { $inst_1 \cdots inst_n$ } et exécute les instructions $inst_1$ à $inst_n$ tant que la condition *expr* est vraie. Juste après l'exécution de $inst_n$, l'expression *expr* est à nouveau évaluée et, si le résultat est non nul, l'exécution reprend à $inst_1$. Si le résultat est 0, alors l'exécution continue à l'instruction suivant le bloc d'instructions. Si la condition est initialement fausse, le bloc n'est pas exécuté.

2.2 Les expressions

Une expression est composée de variables et de constantes numériques (avec ou sans virgule) reliées par des opérateurs. On peut également utiliser les parenthèses (et).
Pour simplifier, on suppose que la valeur d'une expression est de type flottant et correspond au type C double. Si une variable est utilisée dans une expression alors qu'elle n'a encore jamais été affectée, sa valeur sera 0.

Les opérateurs arithmétiques. On peut utiliser les opérateurs arithmétiques classiques +, -, *, /. Les règles de priorité et d'associativité classiques sont utilisées. L'opérateur - peut être utilisé avec un seul argument pour calculer l'opposé d'un nombre (comme dans -X, ou même 2*-X qui signifie 2*(-X)).

Les opérateurs de comparaison. On peut également utiliser les opérateurs de comparaison = = (=), != (≠), >, <, >= (≥) et <= (≤) dans les expressions. Le résultat d'une comparaison est 1 si la relation est vraie, 0 sinon. Ces opérateurs ont une priorité plus faible que celle des opérateurs arithmétiques : X+1>2*Y se lit (X+1)>(2*Y). Attention, l'expression X<Y<Z est tout à fait valide mais ne signifie pas que Y doit se trouver entre X et Z…Attention également à ne pas confondre l'opérateur de comparaison = = avec l'instruction d'affectation =.

Les opérateurs booléens. Enfin, les opérateurs &&, || et ! correspondent aux fonctions booléennes suivantes : && renvoie 1 si ses deux argument sont différent de 0, et 0 sinon ; ||

renvoie 1 si un au moins de ses arguments est différent de 0, et 0 sinon ; ! renvoie 1 si son argument s'évalue en 0, 0 sinon. L'opérateur || a la plus basse priorité, suivi de &&, puis de !, puis des opérateurs de comparaison vus dans le paragraphe précédent. Par exemple X>0 && Y>0 || X<0 && Y<0 se lit ((X>0) && (Y>0)) || ((X<0) && (Y<0)).

2.3 Interactions avec l'utilisateur

Un programme peut dialoguer avec l'utilisateur en utilisant :

- L'instruction print(*expr*); qui affiche à l'écran la valeur de l'expression *expr*.
- L'instruction print("texte"); qui affiche à l'écran la chaîne *texte*. Il est possible d'afficher des caractères spéciaux comme le retour à la ligne ou le caractère " en utilisant dans *texte* les séquences \n et \ ".
- L'expression **input** qui attend que l'utilisateur entre une valeur numérique au clavier et renvoie cette valeur. **input** peut être utilisé partout où une expression est attendue. Par exemple, X**=input;** stockera la valeur lue dans la variable X, tandis que **while** (**input**!=0) { · · · } exécutera un bloc d'instructions tant que l'utilisateur n'aura pas entré la valeur 0 au clavier.

2.4 Exemple

Cet exemple dessine sur l'écran un ensemble de Juila en ASCII-art. Essayez les paramètres CX=0.3 et CY=0.45 :

```
print("CX ? ") ; CX=input ;
print("CY ? ") ; CY=input ;
Y=-1.2 ;
while (Y<1.2) {
X=-1.2 ;
while (X<1.2) {
ZX=X ; ZY=Y ; I=0 ;
while (I<80 && ZX*ZX+ZY*ZY<40) {
T=ZX*ZX-ZY*ZY+CX ;
ZY=2*ZX*ZY+CY ;
ZX=T ;
I=I+1 ;
}
while (I>=6) { I=I-6 ; }
if (I==0) { print("≠") ; }
if (I==1) { print("-") ; }
if (I==2) { print("") ; }
if (I==3) { print(".") ; }
if (I==4) { print("*") ; }
if (I==5) { print(" ") ; }
X=X+0.03 ;
}
Y=Y+0.07 ;
print("\n") ;
}
```

3 Extensions

Cette partie propose quelques extensions que vous pouvez ajouter si vous avez le temps. Vous pouvez traiter une ou plusieurs extension(s) (pas forcément dans l'ordre proposé ici) mais elles ne sont pas obligatoires. Vous pouvez également proposer une ou des extension(s) de votre cru !
Il vaut mieux se limiter à un interpréteur sans extension mais qui marche bien que d'essayer de tout faire... mal !

3.1 Graphisme

Le langage est enrichi d'instructions permettant de dessiner à l'aide d'une «tortue». Comme il n'est pas pratique de dessiner directement sur l'écran, la sortie se fera dans un fichier dessin.ps au format PostScript, lisible avec des logiciels tels que gv ou gsview.

La tortue. La «tortue» possède une position courante (abscisse, ordonnée) et une direction courante (angle en degrés). Les instructions td(*expr*) ; «tourne à droite» et tg(*expr*); «tourne à gauche» font tourner la tortue sur elle-même d'un angle en degrés sans changer sa position. L'instruction av(*expr*); «avance» fait avancer la tortue d'une certaine distance dans la direction courante (ou reculer si la distance est négative). Comme la tortue est munie d'un crayon, elle trace un trait entre sa dernière et sa nouvelle position à chaque fois qu'elle avance.

Le format PostScript. Le fichier doit commencer par la ligne **%!postscript** et se terminer par la ligne **showpage**. Entre ces deux lignes, on utilisera autant de fois que nécessaire la commande x y **moveto** x' y' **lineto stroke** qui trace un trait de la position (x, y) à la position (x', y'), où x, y, x' et y' sont des nombres à virgule. Sur une page A4, la position (0,0) correspond au coin en inférieur gauche de la page et (594, 838) au coin supérieur droit.

Exemples. Le programme suivant :

```
X=0 ; while (X<4) { av(10) ; td(90) ; X=X+1 ; }
```

pourra générer le fichier PostScript dessin.ps suivant :

```
% !postscript
297.0 419.0 moveto 307.0 419.0 lineto stroke
307.0 419.0 moveto 307.0 409.0 lineto stroke
307.0 409.0 moveto 297.0 409.0 lineto stroke
297.0 409.0 moveto 297.0 419.0 lineto stroke
showpage
```

Voici un exemple de dessin un peu plus complexe :

```
X=0 ; while (X<360) { av(1) ; td(1) ; X=X+1 ; }
tg(80) ; X=0 ; while (X<40) { av(57) ; td(170) ; av(57) ; tg(160) ; X=X+1 ; }
```

Extensions. Il est possible de dessiner en couleurs également! La commande PostScript r g b **setrgbcolor** sectionnera pour nouvelle couleur courante la couleur (r, g, b) où r, g et b sont des flottants entre 0 et 1 représentant le taux de rouge, de vert et de bleu de la couleur (0 0 0 correspond au noir, 1 1 1 au blanc, 1 0.5 0.5 à un rouge clair, etc.). La couleur courante influencera toutes les opérations **stroke** suivantes. Pour profiter de la couleur dans votre interpréteur, vous pouvez ajouter à la tortue une notion de couleur courante et une instruction color*(expr,expr,expr*); permettant de la modifier.

De même, la commande PostScript e **setlinewidth** change l'épaisseur du trait courant et vous pouvez ajouter au langage une instruction **epaisseur**(*expr*); permettant de spécifier l'épaisseur du trait de la tortue.
Voici un programme qui utilise des couleurs :

```
X=1 ; Y=0 ; color(1,1,1) ; av(-100) ;
while (Y<28) {
while (X>0) { color(X,1-X,0.5) ; av(0.3) ; td(3*X+0.3) ; X=X-0.001 ; }
while (X<1) { color(X,1-X,0.5) ; av(0.3) ; tg(3*X+0.3) ; X=X+0.001 ; }
color(1,1,1) ; td(90) ; av(30) ; tg(80) ;
Y=Y+1 ;
}
```

3.2 Procédures

Procédures. L'instruction **proc** nom(var_1,…,var_n) { $inst_1$ · · · $inst_m$ } définit une nouvelle procédure de nom *nom*, ayant comme argument les noms de variables var_1 à var_n et pour corps le bloc d'instructions $inst_1$ à $inst_m$. Cette procédure est appelée par l'instruction nom($expr_1$,· · ·,$expr_n$) ; les expressions $expr_1$ à $expr_n$ sont alors évaluées et leur valeurs stockées dans les variables arguments var_1 à var_n puis le corps de la procédure est exécuté. Quand la procédure retourne, les variables argument var_1 à var_n reprennent la valeur qu'elles avaient avant l'appel (ces variables sont locales).

Voici un exemple :

```
proc carre(Y) { print(Y*Y) ; }
X=1 ; while (X<100) { carre(X) ; print("\n") ; X=X+1 ; }
```

Procédures récursives. Il doit être possible à une procédure d'appeler d'autres procédures, et même de s'appeler elle-même (récursivité), comme dans l'exemple suivant qui résout le problème des tours de Hanoi :

```
proc hanoi(A,B,C,N) {
        if (N>0) {
                hanoi(A,C,B,N-1) ;
                print("Deplacer      le
                plateau ") ;
                plateau(A) ;
                print(" sur le plateau
                ") ;
                plateau(C) ;
                print("\n") ;
                hanoi(B,A,C,N-1) ;
                }
                }

proc plateau(X) {
  if (X==0) { print("de gauche") ; }
  if (X==1) { print("du milieu") ; }
  if (X==2) { print("de droite") ; }
}
print("Nombre de plateaux ? ") ;
hanoi(0,1,2,input) ;
```

Procédures et graphisme. Enfin, voici un exemple qui mélange procédures (mutuellement) récursives et graphismes avec la tortue pour dessiner la «courbe du dragon» :

```
proc dragona(X) {
```

```
if (X>0) { td(45) ; dragona(X-1) ; tg(90) ; dragonb(X-1) ; td(45) ; }
else { av(2) ; }
}
proc dragonb(X) {
if (X>0) { tg(45) ; dragona(X-1) ; td(90) ; dragonb(X-1) ; tg(45) ; }
else { av(2) ; }
}
tg(90) ; dragona(14) ;
```