Projet de langage

Le projet de langage se fait par groupe de deux élèves.

Chaque groupe déposera au secrétariat de l'IUP un rapport accompagné d'une disquette au plus tard le 9 décembre au matin. Chaque jour de retard sera sanctionné d'un point.
La disquette contiendra le 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 des sections 1 et 2 devra obligatoirement être implémenté par l'interpréteur, mais il n'est pas interdit d'y ajouter des améliorations de votre cru, ou de traiter des parties de la section 3.

Lors d'une soutenance orale ultérieure (date à déterminer), chaque groupe fera la démonstration sur ordinateur de l'interpréteur et répondra aux questions des examinateurs. Une note pour l'ensemble du projet (rapport, disquette et soutenance) sera donnée à chaque élève.

Le but du projet est d'écrire un interpréteur pour un langage simple s'apparentant au `BASIC` et au `LOGO`.

1 Introduction

L'interpréteur lit un fichier d'entrée contenant le programme à exécuter. Il affiche le résultat de l'exécution à l'écran avant de quitter.
Voici un exemple de fichier `exemple.prog` :

```
PRINT 3/2,"\n" :
X=2 : PRINT "X vaut ",X,"\n" :
TRUC=1 : FOR I=1 TO 10 [ TRUC=TRUC*I ] : PRINT "10 ! = ",TRUC,"\n" :
INPUT "Entrez un nombre ",DATA : WHILE DATA<=0 [ INPUT "Un nombre positif ",DATA ] :
IF DATA MOD 2 = 0 [ PRINT "C'était un nombre pair\n" ] :
IF DATA MOD 2 = 1 [ PRINT "C'était un nombre impair\n" ]
```

La sortie correspondante sera (les nombres en italique ont été entrés par l'utilisateur) :

```
1.5
X vaut 2
10 ! = 3628800
Entrez un nombre -5
Un nombre positif 0
Un nombre positif 42
C'était un nombre pair
```

Les programmes peuvent dialoguer avec l'utilisateur interactivement en affichant des expressions ou des chaînes de caractères à l'écran (`PRINT`) et en demandant à l'utilisateur d'entrer des valeurs (`INPUT`).

Notre langage contient des boucles (`FOR`, `WHILE`). À cause de cela, il n'est pas possible d'exécuter le programme lors du *parsing*. La méthode à employer (vue en cours) consiste donc à construire lors du parsing une représentation mémoire du programme (généralement à base de listes chaînées pour les blocs d'instructions, et d'arbres pour les expressions). Lorsque le fichier entier a été parsé, l'exécution se fait en parcourant cet arbre forme représentation (certaines parties sont parcourues plusieurs fois dans le cas des boucles).

2 Le langage

Dans notre langage, les espaces, tabulations et retours à la ligne ne sont pas significatifs. Les instructions se suivent les unes à la suite des autres et sont séparées par le symbole `:` (`X=2 : Y=3` signifie exécuter `X=2` puis exécuter `Y=3`).

Les variables. Elle sont toutes de type *flottant*. 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).
Pour affecter une nouvelle valeur à une variable, il suffit d'utiliser l'instruction *variable=expression*.
Contrairement au `C`, les variables n'ont pas besoin d'être déclarées avant d'être utilisées.

Les expressions. Le expressions sont des expressions numériques flottantes qui peuvent contenir des variables, des nombres à virgule, les opérateurs arithmétiques classiques (+, -, *, /), l'opérateur modulo MOD (correspondant à la fonction fmod en C) et des parenthèses. 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)).
Si une variable est utilisée dans une expression sans avoir été affectée, on suppose que sa valeur est 0.

Les comparaisons. On peut également utiliser les opérateurs de comparaison =, <> (différent), >, <, >= et => ($\geq$), <= et =< ($\leq$) dans les expressions. Ces opérateurs ont une priorité plus faible que celle des opérateurs arithmétiques.
Le résultat d'une comparaison est 1 si la relation est vraie, 0 sinon. Ce résultat peut être utilisé ou stocké dans une variable comme n'importe quelle nombre. Par exemple, il est possible de simuler un *et logique* par la multiplication (X>0 *et* X<10 s'écrit alors (X>0) * (X<10)).

Encore plus d'expressions. Les fonctions mathématiques suivantes prédéfinies peuvent également apparaître dans une expression :
- ABS(*expression*) calcule la valeur absolue de l'expression entre parenthèses (équivalent de fabs en C) ;
- RAND(*expression*) renvoie une valeur aléatoire entière entre 0 et la valeur de l'expression spécifiée.

L'instruction PRINT. Elle est suivi d'un ou plusieurs objet(s) séparé(s) par des virgules. Chaque objet est soit une chaîne de caractères entre guillemet ("Coucou"), soit une expression (2*X+3). PRINT affiche les chaîne et les valeurs des expressions spécifiées à la suite les unes des autres sans ajouter d'espace ni revenir à la ligne. L'utilisateur force un retour à la ligne par la séquence \n dans une chaîne de caractères. De même, la séquence \" affiche le caractère ".

L'instruction INPUT. INPUT *chaîne*,*variable* affiche la chaîne de caractères sur l'écran et attend que l'utilisateur tape un nombre suivi d'un retour à la ligne. La variable prend pour nouvelle valeur le nombre tapé par l'utilisateur, ou 0 si ce n'est pas un nombre valide.

Tests. IF *expr* [$instr_1$: $\cdots$: $instr_n$] exécute les instructions $instr_1$ à $instr_n$ si et seulement si l'expression *expr* est vraie (s'évalue en 1).

Boucles. WHILE *expr* [$instr_1$: $\cdots$: $instr_n$] exécute $instr_1$ à $instr_n$ tant que l'expression est vraie (s'évalue en 1). Juste après l'exécution de $instr_n$, *expr* est à nouveau évaluée et, si elle est toujours vraie, l'exécution recommence à $instr_1$. Quand la condition est fausse, on passe à l'instruction qui suit la fin du bloc]. Si la condition est initialement fausse, le bloc n'est pas exécuté.

FOR *var*=$expr_1$ TO $expr_2$ [$instr_1$: $\cdots$: $instr_n$] exécute $instr_1$ $\cdots$ à $instr_n$ pour chaque valeur de *var* de $expr_1$ à $expr_2$. On peut considérer que c'est un équivalent concis de *var*=$expr_1$ WHILE *var*<=$expr_2$ [$instr_1$: $\cdots$: $instr_n$: *var*=*var*+1].

Commentaires. Tout texte entre accolades { } doit être ignoré par l'interpréteur.

Vous pouvez trouver sur la page http ://www.di.ens.fr/~mine/iup3/ des exemples de source pour tester votre interpréteur.

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 Sortie graphique

Afficher du texte, c'est bien joli, mais des graphismes, ce serait encore mieux. Comme il est difficile d'afficher des graphismes sur une station Unix, on demandera à l'interpréteur de générer un fichier portable au format PostScript (lisible avec gv sur les stations Unix).

3.1.1 Commandes graphiques

Le langage est enrichi de commandes graphiques à base de *tortue*. La tortue a une position et un angle (en degrés) courant. La commande `AV` *expr* fait avancer la tortue, `RE` *expr* fait reculer la tortue. `TD` *expr* et `TG` *expr* font tourner la tortue sur elle-même dans le sens des aiguilles d'une montre ou dans le sens inverse d'un certain angle en degrés.
La tortue transporte un crayon qui lui fait tracer un traît à chaque fois qu'elle avance (ou recule) de sa position courante à une nouvelle position par les commandes `AV` et `RE`. On peut toutefois demander à la tortue de ne pas dessiner en avançant grâce à la commande `UP`, ou au contraire de lui dire, avec `DOWN`, de recommencer à tracer des traits.

Voici quelques exemples amusants :
- `FOR I=1 TO 4 [ AV 100 : TG 90 ]` dessine un carré;
- `FOR I=1 TO 360 [ AV 1 : TD 1 ]` dessine un cercle.

Mais le plus amusant est de combiner la sortie graphique avec les procédures (extension suivante).

3.1.2 Format PostScript simplifié

Nous utiliserons un sous-ensemble restreint du format PostScript qui permet de dessiner des segments de droite.
Voici le squelette du fichier généré par l'interpréteur :

```
%!postscript
x1 y1 moveto x'1 y'1 lineto stroke
x2 y2 moveto x'2 y'2 lineto stroke
...
xn yn moveto x'n y'n lineto stroke
showpage
```

La première ligne indique le format du fichier. Les n lignes suivantes dessinent des segments de droite des coordonnées (x_i, y_i) à (x'_1, y'_i). La dernière ligne indique la fin de la page.
Pour information, les coordonnées $(0, 0)$ correspondent au coin en bas à gauche de la page et $(594, 838)$ correspondent au coin au haut à droite.

3.1.3 De la couleur

Il possible en PostScript de faire des dessins en couleur et de changer l'épaisseur du trait. La commande r g b `setrgbcolor` indique que la couleur courante (qui influencera toutes les opérations `stroke` suivantes) aura un taux r de rouge, g de vert et b de bleu (valeurs flottantes entre 0 et 1 : 0 0 0 correspond au noir, 1 1 1 au blanc, 1 0 0 au rouge, etc.) La commande e `setlinewidth` change l'épaisseur du trait (e est un flottant).

Si vous avez le courage, vous pouvez enrichir le langage en permettant à la tortue de dessiner en couleur et avec des épaisseurs de trait variées. Par exemple, la commande `COLOR` $expr_r$, $expr_g$, $expr_b$ permettra de changer la couleur du crayon de la tortue.

Pour tout complément, le manuel du langage PostScript est disponible gratuitement à l'adresse suivante `http ://partners.adobe.com :80/asn/developer/PDFS/TN/PLRM.pdf`

3.2 Procédures

Une procédure *nom* et à n arguments est définie par la commande `PROC` *nom* `(`var_1,...,var_n`)` `[` $instr_1$ `:` $\cdots$ `:` $instr_m$ `]`.
Ensuite, chaque occurrence de l'instruction *nom*`(`$expr_1$,...,$expr_n$`)` exécutera le bock d'instructions $instr_1$ à $instr_m$ où var_1,...,var_n prennent la valeur des expressions $expr_1$,...,$expr_n$.
Attention : les valeurs de var_1,...,var_n ne sont modifiés que pendant l'exécution de la procédure *nom*. À la fin de l'appel, ces variables reprennent la valeur qu'elles avaient juste avant l'appel (les arguments sont locaux au corps de la fonction, comme en `C`) !

Voici un exemple simple :

```
PROC CARRE (X) [ PRINT X," au carré vaut ",X*X,"\n" ] :
FOR I=2 TO 5 [ CARRE(I) ] :
X=10 : CARRE(12) : PRINT X
```

qui affiche à l'écran :

```
2 au carré vaut 4
3 au carré vaut 9
4 au carré vaut 16
5 au carré vaut 25
12 au carré vaut 144
10
```

Il est possible d'appeler une procédure dans une autre procédure. Il est possible à une procédure de s'appeler elle-même (récursivité).
Considérez l'exemple :

```
PROC FACT(X,Y) [ IF X=0 [ PRINT Y ] : IF X>0 [ FACT(X-1,Y*X) ] ] :
PROC PRINTFACT (X) [ PRINT X,"! = " : FACT(X,1) : PRINT "\n" ] :
PRINTFACT(5)
```

Le plus rigolo est de mélanger procédures récursives et tortues (section précédente) pour dessiner des fractals :

```
PROC FLOC2(X) [
  IF X>0 [ FLOC2(X-1) : TG 60 :
    FLOC2(X-1) : TD 120 :
    FLOC2(X-1) : TG 60 :
    FLOC2(X-1) ] :
  IF X=0 [ AV 1 ]
] :
PROC FLOC(X) [ FLOC2(X) : TD 120 : FLOC2(X) : TD 120 : FLOC2(X) ] :

FLOC(5)
```

qui donne le résultat suivant :