L2 Licence Sciences et Technologie **UGA**
Parcours Informatique **UE INF404**

Devoir Surveillé du 13 mars 2024

Durée : 1h - Document autorisé : une feuille A4 recto-verso

Les programmes peuvent être écrits en C et/ou en notation algorithmique. Le barème est indicatif.
Toutes les questions sont indépendantes, sauf **Q4** et **Q5** qui sont liées.

Introduction

On s'intéresse dans la suite à un langage de programmation très simple dans lequel :
- 10 **variables** entières sont disponibles, nommées `V0`, `V1`, ...`V9` ;
- les **instructions** sont :
  - `set Vi Exp`, qui affecte la valeur de l'expression `Exp` à la variable `Vi` ;
  - `seq I1 I2`, qui exécute successivement les instructions I1 et I2 ;
- enfin, les **expressions** sont :
  - une constant entière n ;
  - `add Vi Vj`, représentant la somme des variables `Vj` et `Vj` ;
  - `sub Vi Vj`, représentant la différence des variables `Vi` et `Vj`.

Dans ce langage les instructions sont systématiquement écrites entre parenthèses.
On donne ci-dessous un exemple de programme :

```
(seq
   (set V1 42)
   (seq
      (set V2 add V1 V1)
      (set V8 sub V1 V7)
   )
)
```

L'objectif de ce travail est d'écrire un interpréteur sur le même modèle que ce qui a été fait en TP pour la "calculatrice". Cet interpréteur prendra donc en entrée un programme lu au clavier et effectuera : une *analyse lexicale* (Partie 1), une *analyse syntaxique* (Partie 2) et produira *un résultat* à partir de l'arbre abstrait issu de l'analyse syntaxique (Partie 3).

Partie 1 : analyse lexicale [5 points]

Les lexèmes du langage sont les suivants :
- `PARO`, représente la parenthèse ouvrante : `(`
- `PARF`, représente la parenthèse fermante : `)`
- `ENTIER`, représente une suite non vide de chiffres décimaux
- `VAR`, représente un nom de variable : `V0`, `V1`, ..., `V9`
- `SET`, représente le mot-clé `set`
- `SEQ`, représente le mot-clé `seq`
- `ADD`, représente le mot-clé `add`
- `SUB`, représente le mot-clé `sub`

Les caractères séparateurs sont "espace" et "fin de ligne".

Q1. Dessinez un automate **déterministe** reconnaissant les lexèmes du langage. Cet automate lit en entrée une séquence de caractères et il atteint un état final lorsqu'un lexème a été reconnu. Les transitions seront étiquetées uniquement par le caractère courant. On ne fera pas apparaitre le traitement des erreurs lexicales.

Q2. Donnez un exemple d'erreur *lexicale* (cohérente avec votre réponse à la question **Q1**).

Partie 2 : analyse syntaxique [7 points]

La grammaire du langage est la suivante :

$$
\begin{aligned}
Pgm &\rightarrow Par\_Inst \\
Par\_Inst &\rightarrow \texttt{PARO}\ Inst\ \texttt{PARF} \\
Inst &\rightarrow \texttt{SET VAR}\ Exp \\
Inst &\rightarrow \texttt{SEQ}\ Par\_Inst\ Par\_Inst \\
Exp &\rightarrow \texttt{ENTIER} \\
Exp &\rightarrow \texttt{ADD VAR VAR} \\
Exp &\rightarrow \texttt{SUB VAR VAR}
\end{aligned}
$$

Q3. Donnez un exemple de programme sans *erreur lexicale* mais comportant une *erreur syntaxique*.

Q4. En utilisant les primitives de `analyse_lexicale.h`, fournies en Annexe A, écrivez le corps de la fonction `analyser` spécifiée ci-dessous. Vous **devez** écrire des fonctions auxilliaires (comme `Rec_Pgm`, `Rec_Par_Inst`, `Rec_Inst`, et `Rec_Exp`). Vous pouvez également utiliser la fonction `Erreur()` [1] pour indiquer qu'une erreur est détectée. Enfin, vous pourrez abréger `lexeme_courant()` par `LC()`.

```
void analyser() ;
// lit une sequence de lexemes et appelle la fonction Erreur()
// si cette sequence est syntaxiquement incorrecte
```

La réponse à cette question peut être goupée avec celle de la question Q5.

Partie 3 : Calcul des valeurs finales des variables [8 points]

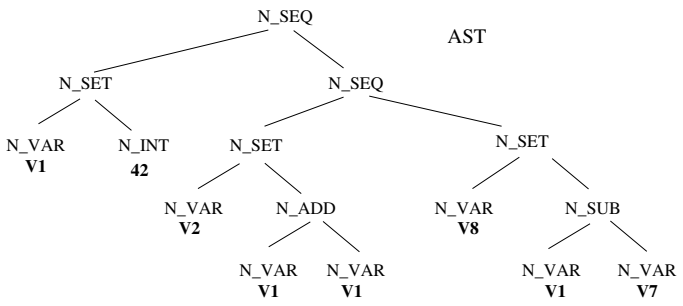

FIGURE 1 – Arbre abstrait du programme donné en introduction

1. qui arrête l'exécution

Q5. Complétez le code de la fonction `analyser()` de la question **Q4** (en ajoutant éventuellement des paramètres et résultats aux fonctions auxilliaires utilisées) pour que cette fonction fournisse en résultat l'arbre abstrait (AST) du programme analysé [2]. A titre d'exemple l'AST du programme donné en introduction est représenté Figure 1. Pour construire cet AST on pourra utiliser les types et fonctions fournis en Annexe B.

Q6. Ecrivez une fonction qui parcourt l'AST d'un programme et affiche **la valeur finale** de toutes les variables. On utilisera **(sans les écrire)** les fonctions fournies en Annexe C pour gérer la table des symboles nécessaire à la gestion de ces variables. On fera en sorte que toute les variables soit initialisées à 0 avant l'exécution du programme. On pourra utiliser des fonctions auxilliaires **(en les écrivant)**.

```
void executer(Ast A) ;
// execute le programme représenté par A et affiche la valeur finale de toute les variables
```

Annexe A : le fichier `analyse_lexicale.h`

```
typedef enum {PARO, PARF, ENTIER, VAR, SET, SEQ, ADD, SUB} Nature_Lexeme;

typedef struct {
   Nature_Lexeme nature;  // nature du lexeme
   char *chaine;          // chaine de caracteres correspondant au lexeme courant
   int  val;              // valeur d'un lexème ENTIER
} Lexeme;

void demarrer();
//   debute l'analyse lexicale
//   lexeme courant est le premier lexeme de la sequence

void avancer();
//   lit le lexeme suivant

Lexeme lexeme_courant() ;  // pourra etre abrege en LC
//   renvoie la valeur du lexeme courant

void arreter();
//  arrete l'analyse lexicale
```

Annexe B : les fichiers `type_ast.h` et `construction_ast.h`

```
typedef enum {N_ENTIER, N_VAR, N_SET, N_SEQ, N_ADD, N_SUB} TypeAst ;
     // indique si un noeud represente un scalaire, un tableau ou une virgule

typedef struct noeud { // un noeud de l'AST
    TypeAst nature ;
    struct noeud *gauche, *droite ; // pointeurs sur les fils gauches et droits
    char *chaine ;  // chaine de caractères associée aux noeuds N_VAR
    char val ;      // valeur associée aux noeuds N_ENTIER
} NoeudAst ;

typedef NoeudAst *Ast ; // un AST est un pointeur sur son noeud racine
```

2. si le fichier lu en entrée ne contient pas d'erreurs.

```
Ast creer_entier (int val) ;
// renvoie un arbre abstrait de nature N_ENTIER et de valeur val

Ast creer_var (char *str) ;
// renvoie un arbre abstrait de nature N_VAR et de chaine de caractéres str

Ast creer_set (Ast *fg, Ast *fd) ;
// renvoie un arbre abstrait de nature N_SET et de fils fg et fd

Ast creer_seq (Ast *fg, Ast *fd) ;
// renvoie un arbre abstrait de nature N_SEQ et de fils fg et fd

Ast creer_add (Ast *fg, Ast *fd) ;
// renvoie un arbre abstrait de nature N_ADD et de fils fg et fd

Ast creer_sub (Ast *fg, Ast *fd) ;
// renvoie un arbre abstrait de nature N_SUB et de fils fg et fd
```

Annexe C : le fichier `table_symbole.h`

```
void initTS () ;
// initialise la table des symboles en affectant toutes les variables à 0

void set (char *var, int val) ;
// affecte la variable de nom var avec la valeur val

int val (char *var) ;
// renvoie la valeur courante de la variable de nom var

void afficheTS() ;
// affiche les valeurs de toutes les variables
```