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

Examen du 15 mai 2017

Durée : 2h - une feuille A4 recto-verso autorisée - Barême indicatif

Indication importante :

Les programmes demandés peuvent être écrits en C et/ou en notation algorithmique. Le critère essentiel est qu'ils soient **lisibles**. De même vous pouvez utiliser certaines fonctions auxilliaires sans les écrire mais à condition de bien les **spécifier** (paramètres, effets de bord, etc.).

Partie 1 : Expressions Arithmétiques avec variables (6 points)

On considère le langage des **expressions arithmétiques avec variables**, noté *Eag*, similaire à celui étudié en cours de semestre. Les **lexèmes** utilisés sont rappelés en Annexe A. La **syntaxe** de ce langage est défini par la grammaire suivante :

Eag	$\rightarrow$	Terme X
X	$\rightarrow$	`PLUS` Terme X
X	$\rightarrow$	`MOINS` Terme X
X	$\rightarrow$	ε
Terme	$\rightarrow$	Facteur Y
Y	$\rightarrow$	`MULT` Facteur Y
Y	$\rightarrow$	`DIV` Facteur Y
Y	$\rightarrow$	ε
Facteur	$\rightarrow$	`PARO` Eag `PARF`
Facteur	$\rightarrow$	`ENTIER`
Facteur	$\rightarrow$	`IDF`

On donne un exemple d'expression correcte : `x + 5 * y - x`

Q1. Donnez un exemple d'expression comportant une erreur lexicale.

Q2. Donnez un exemple d'expression comportant une erreur syntaxique (mais sans erreur lexicale).

Q3. Dessinez l'**arbre de dérivation** de l'expression `x + 5 * y - x`

Q4. Dessinez l'**arbre abstrait** (ou Ast) de l'expression `x + 5 * y - x`

Q5. Le type `Ast` étant rappelé en Annexe B, écrivez le corps de la fonction suivante :

```
int nbIDF (Ast A) ;
/*
   A etant l'Ast d'une expression arithmetique avec variables,
   nbIDF(A) est le nombre de variables distinctes apparaissant dans A
*/
```

Indications :

- si `A` est l'arbre abstrait de `x + 5 * y - x`, alors `nbIDF(A)` vaut 2.
- vous pouvez utiliser une structure de données intermédiaire [1] pour répondre à cette question, mais il faudra alors donner le contenu du fichier ".h" correspondant (définition des types, et en-tête des fonctions d'accès à cette structure).

Partie 2 : Déclaration de fonction (7 points)

On étend maintenant le langage *Eag* pour permettre de déclarer une fonction avec paramètres. La syntaxe d'une déclaration de fonction est donnée par la grammaire suivante :

$$
\begin{aligned}
\textit{DecFonc} &\rightarrow \texttt{IDF}\ \textit{Liste\_ParamF}\ \texttt{EGAL}\ \textit{Eag}\ \texttt{PVIRG} \\
\textit{Liste\_ParamF} &\rightarrow \texttt{IDF}\ \textit{Liste\_ParamF} \\
\textit{Liste\_ParamF} &\rightarrow \varepsilon
\end{aligned}
$$

On donne ci-dessous un exemple de déclaration d'une fonction `f` avec deux paramètres formels de noms `x` et `y` :

```
f x y = x + 5 * y - x ;
```

Q6. Cette question pourra être groupée avec la question Q8.

Ecrivez le corps de la fonction suivante qui effectue l'analyse syntaxique d'une déclaration de fonction :

```
void Rec_DecFonc() ;
/*
   lit une sequence de lexemes et renvoie un message d'erreur si cette sequence
   ne correspond pas a une declaration de fonction syntaxiquement correcte
*/
```

Les types et primitives de l'analyseur lexical sont rappelés en Annexe A. Vous pouvez écrire des fonctions intermédiaires comme `Rec_Liste_ParamF`. Vous pouvez également utiliser (sans la ré-écrire) la fonction `Rec_Eag` suivante :

```
void Rec_Eag(Ast *A) ;
/*
   reconnait une sequence de lexemes correspondant a une expressions arithmetique
   avec variables et construit son arbre abstrait *A
*/
```

1. comme une table des symboles ...

Q7. Proposez une structure de données de nom `FoncDec` permettant de mémoriser les informations relatives à une déclaration de fonctions : le nom de la fonction, la liste de ses paramètres, le corps de la fonction.

Vous pouvez utiliser au choix des arbres et/ou tableaux et/ou listes chaînées, etc. Vous pouvez également supposer que le nombre maximal de paramètres d'une fonction est borné. **Pour répondre à cette question vous devez :**

1. dessiner le contenu de votre structure de donnée pour la déclaration :
   `f x y = x + 5 * y - x ;`
2. donner le contenu d'un fichier `fonction.h` contenant :
   — les déclarations de type décrivant cette structure de données `FoncDec` ;
   — les en-têtes des fonctions nécessaires pour construire et consulter cette structure ; il n'est pas nécessaire d'**écrire** le corps de ces fonctions, mais de **spécifier** leur comportement en quelques phrases ...

Q8. Ecrivez le corps de la fonction suivante :

```
void Rec_DecFonc(FoncDec *DF) ;
/*
   lit une sequence de lexemes et renvoie un message d'erreur si cette sequence ne
   correspond pas a une declaration de fonction syntaxiquement correcte,
   sinon construit la structure DF contenant les informations relatives a cette declaration.
*/
```

Q9. Ecrivez le corps de la fonction suivante :

```
int verifParam (FoncDec DF) ;
/*
   verifie qu'il n'y a pas d'erreurs de parametres dans la declaration de fonction DF :
   les parametres utilises dans le corps de la fonctions correspondent aux parametres declares.
*/
```

Indication : une "erreur de paramètre" est une situation dans laquelle
— soit le nombre de paramètres déclarés ne correspond pas au nombre de paramètres utilisés dans la fonction ;
— soit les paramètres déclarés n'ont pas les mêmes noms que ceux utilisés dans la fonction.

Exemples :
— la déclaration `f x y = x + 5*y - x ;` ne comporte pas d'erreur de paramètre ;
— la déclaration `g x z = x + 5*y - z ;` comporte une erreur de paramètre.
— la déclaration `g x = x + 5*y - x ;` comporte une erreur de paramètre.
— la déclaration `g x t z = x + 5*y - z ;` comporte une erreur de paramètre.

Partie 3 : Exécuter un programme (7 points)

Dans la suite, un "programme" est défini comme une séquence de déclarations de fonctions suivie d'un appel à l'une de ces fonctions. Chaque fonction déclarée peut également appeler une fonction déclarée avant elle dans le programme.

On donne à titre d'exemple le programme **P1** suivant :

```
f1 x = 3 + x ;
f2 x y = x * [f1 y] ;
f3 t = [f2 t [f1 t]] ;
[f3 (3 + 2)]
```

La syntaxe d'un programme est donnée par la grammaire suivante :

$$\begin{aligned} Pgm &\rightarrow Liste\_DecFonc\ \ CallFonc\ \ \texttt{FSEQ} \\ Liste\_DecFonc &\rightarrow DecFonc\ \ \ Liste\_DecFonc \\ Liste\_DefFonc &\rightarrow \varepsilon \end{aligned}$$

La syntaxe d'un appel de fonction est :

$$\begin{aligned} CallFonc &\rightarrow \texttt{CROCHO IDF}\ \ Liste\_ParamE\ \ \texttt{CROCHF} \\ Liste\_ParamE &\rightarrow Eag\ \ Liste\_ParamE \\ Liste\_ParamE &\rightarrow \varepsilon \end{aligned}$$

Enfin, la syntaxe des expressions arithmétiques est modifiée pour permettre des appels de fonctions dans le corps d'une fonction. On ajoute pour cela une nouvelle règle :

$$Facteur \rightarrow CallFonc$$

Q10. Donnez le résultat obtenu lors de l'exécution du programme **P1**, c'est-à-dire le résultat de l'appel à la fonction `f3` avec pour paramètre effectif la valeur `3+2`.

Q11. Complétez le contenu du fichier `fonctions.h` en décrivant les structures de données permettant de représenter :
- un ensemble de déclaration de fonction (type `EnsFoncDec`) [2] ;
- un appel de fonction (type `FoncCall`) ;

Q12. Dessinez le contenu de ces structures dans le cas du programme **P1**.

Q13. En utilisant le cas échéant des fonctions auxilliaires correctement spécifiées, écrivez le code de la fonction suivante :

```
int  executerPgm (EnsFoncDec LFonc,  FoncCall fcall) ;
/*
 renvoie le resultat de l'appel fcall lorsque :
     1. la fonction appellee par fcall est definie dans LFonc
     2. toute fonction appellee par une fonction definie dans LFonc est
         elle-meme definie precedement dans LFonc
     3. il n'y a pas d'erreur de parametre dans les fonctions definies dans LFonc
  sinon affiche un message d'erreur approprie  ...
*/
```

2. vous pouvez bien sur ré-utiliser le type `FoncDec` de la question **Q7**.

Annexe A : le fichier `analyse_lexicale.h`

```
   typedef enum {
    ENTIER,      /* une suite de chiffre */
    IDF,         /* une suite de lettres */
    PLUS,        /* '+' */
    MOINS,       /* '-' */
    MULT,        /* '*' */
    DIV,         /* '/' */
    PARO,        /* '(' */
    PARF,        /* ')' */
    CROCHO,      /* '[' */
    CROCHF,      /* ']' */
    PVIRG,       /* ';' */
    EGAL         /* '=' */
  } Nature_Lexeme ;

   typedef struct {
      Nature_Lexeme nature;   // nature du lexeme
      char chaine[256];       // chaine de caracteres
      int  val;               // valeur d'un lexeme ENTIER
   } Lexeme ;

   void demarrer(char *nom_fichier);
   // initialise l'analyse lexicale

   void avancer();
   // lit le lexeme suivant

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

   int fin_de_sequence();
   // vrai ssi la fin de sequence est atteinte
```

Annexe B : le fichier `type_ast.h` (modifiable si besoin)

```
typedef enum {N_ENTIER, N_IDF, N_PLUS, N_MOINS, N_MULT, N_DIV} TypeAst ;

typedef struct noeud {
TypeAst nature ;
struct noeud *gauche, *droit ;
    char chaine[256];       // chaine de caracteres pour un N_IDF
    int  val;               // valeur pour un N_ENTIER
} NoeudAst ;

typedef NoeudAst *Ast ;
```