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

Examen du 23 mai 2024

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 paramètres (6 points)

On considère le langage des **expressions arithmétiques avec *paramètres***, noté *Eag*. Ce langage est similaire à celui étudié en cours de semestre, à la différence près que les expressions peuvent comporter des paramètres entiers notés `$1`, `$2`, etc. Une liste complète des **lexèmes** est rappelée 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	$\rightarrow$	`PARAM`

On donne un exemple d'expression arithmétique paramétrée correcte : `$1 + 5 * $2 - $1`

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. Le type `Ast` étant rappelé en Annexe B, dessinez l'**arbre abstrait** de l'expression `$1 + 5 * $2 - $1`

Partie 2 : Déclaration de fonction (7 points)

On étend maintenant le langage *Eag* pour permettre de déclarer une **fonction avec paramètres**. On donne ci-dessous un exemple de déclaration d'une telle fonction :

`{f 2 = $1 + 5 * $2 - $1}`

Dans cette déclaration `f` est le nom de la fonction, l'entier `2` désigne le nombre de paramètres formels de `f` et l'expression `$1 + 5 * $2 - $1` est le corps de la fonction.

La syntaxe d'une déclaration de fonction est donnée par la grammaire suivante :

$$DecFonc \rightarrow \texttt{ACCO FUNC ENTIER EGAL } Eag \texttt{ ACCF}$$

Q4. Proposez une structure de données de nom `FoncDec` permettant de mémoriser les informations relatives à une déclaration de fonctions : le nom de la fonction, le nombre de ses paramètres formels, le corps de la fonction.

Vous pouvez utiliser au choix des arbres et/ou tableaux et/ou listes chaînées, etc. **Pour répondre à cette question vous devez :**

1. dessiner le contenu de votre structure de donnée pour la déclaration :
   `{f 2 = $1 + 5 * $2 - $1}`
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 ...

Q5. Ecrivez le corps de la fonction suivante qui effectue l'analyse syntaxique d'une déclaration de fonction et construit la structure de type `FoncDec` correspondante :

```
void Rec_DecFonc(FoncDec *df) ;
/*
lit une sequence de lexemes correpondant à une déclaration de fonction et
initialise la structure fd avec les informations correspondantes s'il n'y
a pas d'erreur de syntaxe dans cette déclaration.
*/
```

Les types et primitives de l'analyseur lexical sont rappelés en Annexe A. Vous pouvez utiliser **(sans la ré-écrire)** les fonction suivantes :

```
void Rec_Eag(Ast *A) ;
/*
   reconnait une séquence de lexemes correspondant à une expressions arithmétique
   paramétrée et construit son arbre abstrait *A
*/

void Erreur();
/* stoppe l'exécution du programme avec un message d'erreur */
```

Q6. On souhaite vérifier que la définition d'une fonction est correcte dans le sens où le nombre de paramètres formels déclarés correspond au nombres de paramètres effectifs utilisés dans la fonction. Par exemple, la déclaration `{g 3 = $1 + 5*$2 - $1}` n'est pas correcte (3 paramètres déclarés, 2 utilisés dans le corps de la fonction).

Ecrivez le corps de la fonction suivante :

```
int verifParam (FoncDec df) ;
/*
   vaut vrai ssi, pour la fonction représenté par DF, le nombre de paramètres formels déclarés correspo
   au nombre de paramètres utilisés.
*/
```

Partie 3 : Déclaration et exécution de 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 des fonctions **déclarées avant elle** dans le programme.

On donne à titre d'exemple le programme **P1** suivant, qui déclare trois fonction `f1`, `f2` et `f3` et qui appelle la fonction `f3` en lui passant en paramètre l'expression `3+2`.

```
{f1 1 = 3 + $1}
{f2 2 = $1 * [f1 $2]}
{f3 1 = [f2 $1 [f1 $1]]}
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 :

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

Lorsque l'on exécute un appel de fonction, les paramètres effectifs sont substitués aux paramètres formels selon le principe suivant : le premier paramètre effectif correspond au paramètre formel `$1`, le deuxième au paramètre `$2`, etc.

Exemple : l'appel `f (12 42)` de la fonction {`f 2 = $1 + 5 * $2 - $1`} consiste à évaluer l'expression `12 + 5 * 42 - 12`.

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 des fichiers `fonctions.h` et `ast.h` en décrivant les structures de données permettant de représenter :

— un ensemble de déclaration de fonction (type `EnsFoncDec`) [1] ;

— un appel de fonction dans un arbre abstrait.

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

Q13. On suppose que les fonctions ont au plus 5 paramètres, et on définit le type `SeqParam` qui représente la séquence des valeurs des paramètres effectifs d'une fonction comme un tableau de 5 entiers :

```
typedef int SeqParam[5] ; // valeurs des paramètres effectifs d'un appel de fonction
```

Ecrivez le code de la fonction suivante :

```
int execAppel(FoncDec df, SeqParam params) ;
/*
   execute la fonction declaree dans df en lui transmettant la liste des
   valeurs des parametres effectifs params et renvoie la valeur du resulat.
*/
```

Indication : on supposera ici que le corps de la fonction `df` ne **contient pas** d'appel de fonction.

Q14. Ecrivez le code de la fonction suivante qui permet d'exécuter un programme :

```
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
*/
```

1. vous pouvez bien sur ré-utiliser le type `FoncDec` de la question **Q4**.

Annexe A : le fichier `analyse_lexicale.h`

```
   typedef enum {
    IDF          /* une suite non vide de lettres */
    ENTIER,      /* une suite non vide de chiffres */
    PARAM,       /* paramètre formel: le caractère $ suivi d'une suite de chiffres */
    PLUS,        /* '+' */
    MOINS,       /* '-' */
    MULT,        /* '*' */
    DIV,         /* '/' */
    PARO,        /* '(' */
    PARF,        /* ')' */
    ACCO,        /* '{' */
    ACCF,        /* '}' */
    EGAL         /* '=' */
  } Nature_Lexeme ;

   typedef struct {
      Nature_Lexeme nature;   // nature du lexeme
      char chaine[256];       // chaine de caracteres
      int  val;               // valeur d'un lexeme ENTIER
  int nparam;             // numéro de paramètre (1 pour $1, 2 pour $2, etc.)
   } 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
int nparam;         // numéro de paramètre (1 pour $1, 2 pour $2, etc.)
} NoeudAst ;

typedef NoeudAst *Ast ;
```