

Examen du 19 mai 2016

Durée : 2h - Document autorisé : une feuille A4 manuscrite

Introduction

Ce sujet porte sur un langage d'expression, inspiré des langages fonctionnels comme Caml¹. On trouvera donc différentes étapes :

- analyse lexicale du langage (partie 1) ;
- analyse syntaxique du langage (partie 2) ;
- interprétation (partie 3) ;
- et enfin, une extension (partie 4).

Indication importante :

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

Définition du langage F

Intuitivement, le langage F est formé de trois types d'expressions :

- des **expressions simples**, dont la valeur peut être un entier ou un booléen, et qui correspondent aux expressions arithmétiques ou booléennes utilisées dans le projet, comme par exemple :
 $x + 12$ ou $(x \leq 3 \text{ et not } (y = 5))$;
- des **expressions conditionnelles**, comme par exemple :
 $\text{si } x > 0 \text{ alors } x+1 \text{ sinon } 12$;
- des **expressions avec déclarations**, comme par exemple :
 $\text{soit } x = 12 \text{ dans } (x+3)$.

On donne ci-dessous 4 exemples d'expressions correctes (e1, e2, e3 et e4) du langage F et leurs valeurs :

- e1** : soit $x = 30$ dans $(x+12)$, de valeur 42
- e2** : soit $x = 10$ dans (soit $y = x+5$ dans $(x*y+1)$), de valeur 151
- e3** : soit $x = 5$ dans (si $x > 10$ alors x sinon $x+2$), de valeur 7
- e4** : soit $x = 2$ dans (soit $x = x+1$ dans $(x+2) * x$), de valeur 15

1. Il n'est toutefois pas nécessaire de connaître CAML pour faire cet examen !

Partie 1 : analyse lexicale (3 points)

En plus des lexèmes utilisés dans le projet pour les expressions arithmétiques et booléennes (ENTIER, PLUS, EGAL, ET, PARO, PARF, etc.) que l'on détaillera pas ici, le langage F nécessite les lexèmes suivants :

- les *identificateurs*, IDF, représentés par des lettres minuscules : a, b, ..., z
- les *mots-clés* si, alors, sinon, soit, et dans représentés par SI, ALORS, SINON, SOIT, et DANS.

Le type `Lexeme` complet est donné en Annexe A.

Q1. (1pt) Donnez un exemple d'erreur lexicale pour le langage F .

Q2. (2pt) Expliquez précisément comment programmer la partie de l'analyseur syntaxique qui reconnaît les lexèmes de type IDF ainsi que les mots-clés du langage (SI, ALORS, SINON, etc.). Pour cela vous dessinerez un fragment d'automate et vous donnerez la portion de code correspondante de la procédure `reconnaitre_lexeme` :

```
procedure reconnaitre_lexeme ;
-- etat initial :
-- le caractere courant est un espace ou le 1er caractere du lexeme courant
-- etat final :
-- le caractere courant est un espace ou le 1er caractere du lexeme suivant
-- la variable Lexeme_Courant (de type Lexeme) contient le lexeme courant
```

Partie 2 : analyse syntaxique et arbre abstrait (7 points)

La **syntaxe** de ce langage est défini par la grammaire suivante² :

$$\begin{aligned} Pgm &\rightarrow Exp \text{ FSEQ} \\ Exp &\rightarrow SI \ Exp_Bool \ ALORS \ Exp \ SINON \ Exp \\ Exp &\rightarrow SOIT \ IDF \ EGAL \ Exp_Arith \ DANS \ PARO \ Exp \ PARF \\ Exp &\rightarrow PARO \ Exp \ PARF \\ Exp &\rightarrow Exp_Arith \\ Exp_Bool &\rightarrow \dots \\ Exp_Arith &\rightarrow \dots \end{aligned}$$

Q3. (1pt) Donnez un exemple de programme F comportant une erreur syntaxique (mais sans erreurs lexicales).

Q4. (1pt) Dessinez des arbres abstraits associés aux 4 expressions (e1, e2, e3 et e4) données en introduction.

Q5. (1pt) Définissez un type `Ast` permettant de représenter l'arbre abstrait d'une expression `Exp`. On pourra compléter le type `Ast` utilisé dans le projet (et fourni en Annexe B), on pourra également faire des schémas.

Q6. (4pt) En utilisant le type `Lexeme` fourni en Annexe A, les primitives de `analyse_lexicale.ads` utilisées en projet (`Demarrer`, `Avancer`, `Lexeme_Courant`, etc) et votre type `Ast` de la question **Q5** écrivez le corps de la procédure `analyser` spécifiée ci-dessous, sans détailler la reconnaissance des **expressions simples**. Concrètement, vous devez donc écrire la fonction `rec_Pgm` et la fonction `rec_Exp` mais sans écrire `rec_Exp_Bool` ni `rec_Exp_Arith`. Il n'est pas nécessaire non plus de détailler les opérations de construction de l'arbre abstrait, que vous utilisez mais il faut les **spécifier** !

```
procedure analyser (A : out Ast);
-- e.i. : indifferent
-- e.f. : une sequence de lexemes a ete lue, A designe son arbre abstrait,
--      l'exception Erreur_Syntaxique est levee en cas d'erreur de syntaxe.
```

2. La syntaxe des **expressions simples** (`Exp_Bool` et `Exp_Arith`) n'a pas besoin d'être détaillée ici

Partie 3 : interprétation (6 points)

On s'intéresse maintenant au calcul de la **valeur** d'une expression du langage F . On suppose ici que cette valeur est toujours de type **entier** (les expressions booléennes ne sont utilisées que comme condition des expressions conditionnelles).

Pour cela, il faut tout d'abord définir une *table des symboles* permettant de mémoriser les valeurs de chaque identificateur. Il y a **plusieurs différences importantes** avec ce qui a été fait dans le projet :

- les identificateurs sont représentés par une lettre minuscule, il y a donc au plus 26 noms d'identificateurs différents, et on peut utiliser ces noms pour indexer un tableau ($T('a')$, $T('b')$, etc.);
- un identificateur ne change jamais de valeur (il n'y a pas d'affectation dans ce langage);
- un même identificateur peut apparaître dans des blocs imbriqués différents, il doit donc être représenté **plusieurs fois** dans la table des symboles (comme dans l'expression `soit x = 3 dans (soit x = x+1 dans (...))`)

Q7. (3pt) Donnez le contenu du paquetage interface `table_symbole.ads`. Vous devez donc donner :

- La définition des types représentant la structure de données de la “table des symboles”; vous pouvez utiliser pour cela du code Ada, une notation algorithmique et/ou faire un schéma.
- La spécification des fonctions/procédures fournies par ce paquetage.

Q8 (3pt). Ecrivez une fonction `valeur` qui prend en paramètre l'arbre abstrait d'une expression et renvoie sa valeur :

```
function valeur (A : Ast) : integer ;  
--    valeur (A) est la valeur de l'expression d'Ast A
```

On pourra pour cette question utiliser sans lécrire une fonction `valeur_bool (A : Ast) : bool` qui renvoie la valeur de l'expression booléenne d'arbre abstrait A.

Partie 4 : extension (4 points)

On propose dans cette partie de rajouter au langage la possibilité de définir et d'appeler des **fonctions** à un seul paramètre et à résultat entier, comme par exemple :

```
soit fun F (x) = x + 1 dans (call F (12) + 5)
```

Pour traiter cette extension, on suppose que :

- les noms des fonctions sont des lettres majuscules;
- l'analyse syntaxique construit un arbre abstrait pour chaque fonction déclarée et les stocke dans une table des fonctions, par exemple : `type Tab_Func = array ('A' .. 'Z') of Ast`

Q9 (2pt) Complétez le type `Ast` pour représenter :

- une déclaration de fonction (et son paramètre !)
- un appel de fonction

Q10. (1pt) Complétez la fonction `valeur` de la question **Q8** pour traiter le cas d'un appel de fonction.

Q11. (1pt) Votre solution proposée à la question **Q10** permet-elle de traiter le cas de fonctions récursives ? Justifiez ...

Annexe A : le paquetage machine_lexemes.ads

```
type Nature_Lexeme is (
  ENTIER,          -- sequence de chiffres
  PLUS,            -- +
  MOINS,           -- -
  MUL,             -- *
  INF,             -- <
  SUP,            -- >
  EGAL,           -- =
  PARO,           -- (
  PARF,           -- )
  SI,             -- si
  ALORS,          -- alors
  SINON,          -- sinon
  SOIT,           -- soit
  DANS,           -- dans
  FSEQ,           -- pseudo lexeme ajoute en fin de sequence
);

type Pointeur_Chaine is access String;

type Lexeme is record
  nature : Nature_Lexeme;    -- nature du lexeme
  chaine : Pointeur_Chaine;  -- chaine de caracteres
  valeur : Natural;         -- valeur d'un entier
end record;
```

Annexe B : le paquetage arbre_abstrait.ads (à compléter)

```
type Ast is private;

type TypeAst is (OPERATION, VALEUR);
type TypeOperateur is (PLUS, MUL, MOINS, INF, SUP, EGAL, ...);

private

type NoeudAst;
type Ast is access NoeudAst;

type NoeudAst is record
  nature : TypeAst;
  operateur : TypeOperateur;
  gauche, droite : Ast;
  valeur : Integer;
end record;
```