

PROJET DE COMPILATION - ANALYSE LEXICALE

1. OBJECTIF

L'objectif de ce TP est de programmer un analyseur lexical pour le langage L . L'analyseur lexical se présente comme une fonction, `int yylex()` qui, à chaque appel, renvoie comme résultat le code de l'unité lexicale dont c'est le tour. De plus, cette fonction met dans la variable `yytext` la valeur de l'unité lexicale reconnue. Ainsi, lorsque `yylex` reconnaît la chaîne de caractères `123`, elle renverra le type `NOMBRE` et mettra dans la variable `yytext` la chaîne de caractères `"123"`.

2. LE LANGAGE L

Le langage L est un langage de programmation minimaliste, inspiré du langage C.

Types: Le langage L connaît deux types de variables :

- Un type simple : le type `entier`.
- Un type dérivé : les tableaux d'entiers, déclarés avec des crochets `[...]`.

Opérateurs: Le langage L connaît les opérateurs suivants :

- arithmétiques : `+`, `-`, `*`, `/`
- comparaison : `<`, `=`
- logiques : `&` (et), `|` (ou), `!` (non)

Instructions: Le langage L connaît les instructions suivantes :

- Instruction vide ;
- Bloc d'instructions, délimité par des accolades `{ ... }`
- Affectation `$a = $b + 1;`
- Instruction `si expression alors { ... }` et `si expression alors { ... } sinon { ... }`
- Instruction `tantque expression faire { ... }`
- Instruction `retour expression ;`
- Instruction d'appel à fonction `fonction(liste d'expressions);`

Fonctions: un programme L est une suite de fonctions, dont `main`

- Ce sont des fonctions à résultat entier.
- Le passage des arguments se fait par valeur.
- Les fonctions possèdent des variables locales.
- Une fonction ne peut pas être déclarée à l'intérieur d'une autre.
- On peut ignorer le résultat rendu par une fonction.

Fonctions prédéfinies: Les entrées-sorties de valeurs entières se font à l'aide de deux fonctions prédéfinies, `$a = lire();` et `ecrire($a);`.

Voici un exemple d'un programme en L :

```
f(entier $a, entier $b) # déclaration d'une fonction à deux arguments
entier $c, entier $k;  # déclaration de deux variables locales
{
    $k = $a + $b;      # affectation et expression arithmétique
    retour $k;        # valeur de retour de la fonction
}
# fin du bloc d'instruction
main()                # point d'entrée dans le programme
entier $d;
{
    $d = f($d, 2);    # affectation et appel de fonction
    ecrire($d + 1);   # appel de la fonction prédéfinie ecrire
}
```

3. UNITÉS LEXICALES

On distingue 5 types d'unités lexicales :

- les symboles simples (+, ;, (, {, ...),
- les mots-clefs, dont :
 - les instructions de contrôle (**si**, **alors**, **retour**, ...),
 - les types (**entier**),
 - les fonctions spéciales **lire** et **ecrire**,
- les noms de variables,
- les noms de fonctions et
- les nombres, uniquement entiers.

Les unités lexicales de type symboles simples et mots-clefs constituent des **classes fermées**, on peut en faire l'inventaire exhaustif.

Les unités lexicales de type noms de variables et de fonction ainsi que les nombres constituent des **classes ouvertes**, on ne peut pas en faire l'inventaire. Cependant, on peut les décrire à l'aide de contraintes de forme (par exemple : un nombre entier est une suite de chiffres).

Les unités lexicales des deux différentes classes distinguées ci-dessus ne sont pas traitées de la même manière. Dans le cas d'une classe fermée, c'est l'unité lexicale elle-même qui est retournée par `yylex` (en fait un code correspondant à l'unité lexicale, comme expliqué ci-dessous). Dans le cas d'une classe ouverte, c'est le (code du) *type d'unité lexicale* qui est renvoyé ; la valeur de l'unité lexicale est copiée dans `yytext`.

Chaque unité lexicale d'une classe fermée ou type d'unité lexicale d'une classe ouverte est représentée par une constante symbolique, définie comme ceci :

```
#define TANTQUE 20
```

Les déclarations des constantes symboliques sont regroupées dans le fichier `symboles.h` qui vous est fourni et qui sera partagé avec l'analyseur syntaxique. Il convient d'utiliser les constantes données dans ce fichier pour pouvoir tester facilement le programme.

Le dernier code renvoyé par l'analyseur lexical doit être le symbole (`FIN`). C'est-à-dire que, quand l'analyseur lexical arrive à la fin du fichier (EOF), il doit encore renvoyer une dernière unité lexicale "FIN" pour signaler qu'il est arrivé à la fin du fichier. Ceci sera pratique lors de l'analyse syntaxique.

L'ensemble de tous les mots-clefs est regroupé au sein d'un tableau de mots-clefs qui associe à chaque unité lexicale de type mot-clef, le code du mot-clef correspondant.

4. IDENTIFICATEURS ET NOMBRES

Un identificateur est une suite de caractères qui n'est pas dans le tableau des mots-clefs. Il s'agit d'une suite de lettres non accentuées majuscules ou minuscules, et les symboles dollar (\$) ou souligné (_). Un identificateur de variable commence toujours par un symbole dollar tandis qu'un identifiant de fonction commence toujours par un caractère différent d'un chiffre ou d'un dollar (c'est-à-dire une lettre majuscule, minuscule ou souligné). De plus, un identificateur de fonction ne peut pas être identique à un mot-clef du langage *L*. Par exemple, on ne peut pas déclarer une fonction qui s'appelle `tantque` ou `ecrire`. En utilisant la notation d'expressions régulières UNIX (comme dans `grep`) :

- `ID_VAR` → `\$[0-9a-zA-Z_\$]*`
- `ID_FCT` → `[a-zA-Z_][0-9a-zA-Z_\$]*`
- `NOMBRE` → `[0-9]+`

Un identificateur de fonction ou de variable peut avoir au plus 99 caractères (y compris le dollar qui précède le nom d'une variable). Le langage *L* est sensible à la casse.

5. LECTURE DU TEXTE SOURCE

L'analyseur lexical accède au texte source caractère par caractère. Pour la clarté des idées, on s'interdira de lire le texte source par lignes.

Un petit problème est celui des unités dont la reconnaissance implique la lecture d'un caractère « en trop » : les nombres et les identificateurs. On pourra utiliser la fonction `ungetc` de la bibliothèque standard.

La lecture du fichier contenant le programme source se fera à travers un flux appelé `yyin`.

6. COMMENTAIRES ET BLANCS

Les unités lexicales peuvent être séparées les unes des autres dans le programme source par un nombre quelconque de blancs, caractères de tabulation (`\t`) et retours chariot (`\n`). Tous ces caractères doivent être ignorés. C'est l'analyseur lexical qui doit s'occuper de supprimer les espaces. Cependant, l'analyseur doit s'en servir pour séparer les unités lexicales. Par exemple, il est obligatoire de mettre un (ou plusieurs) espaces dans la déclaration d'une variable, entre le mot-clef et le nom, par exemple, `entier $max`. Cependant, dans une expression comme `$max = 1`; les espaces sont optionnels car le caractère `=` ne peut pas faire partie de l'identificateur.

Il est possible de rajouter des commentaires d'une ligne. Ces commentaires commencent par un caractère dièse (`#`) et terminent à la fin de la ligne. Le caractère dièse et tout ce qui suit jusqu'à la fin de la ligne (`\n`) est ignoré par le compilateur.

Il vous est conseillé, pour commencer, de ne pas permettre des commentaires dans les programmes écrits en *L*. En tout cas, cette question ne doit concerner que l'analyseur lexical, et il ne doit rester aucune trace des commentaires au niveau de l'analyseur syntaxique. Notamment, les commentaires ne sont mentionnés nulle part dans la grammaire.

7. EXEMPLE

Voici un exemple possible de résultat de l'analyse syntaxique d'une instruction *si ... alors* :

```
source  si  $n  <  1  alors  $n  =  $n  +  1
retour  SI  ID_VAR  INFERIEUR  NOMBRE  ALORS  ID_VAR  EGAL  ID_VAR  PLUS  NOMBRE
```

8. PROGRAMME DE TEST

La fonction `yylex` sera appelée par l'analyseur syntaxique qui ne sera programmé que lors des prochains TP. Il est fortement conseillé de valider l'analyseur lexical à partir d'un programme de test dans le genre de celui-ci.

```
#include <stdio.h>
#include <stdlib.h>
#include "lexique.h"
#include "symboles.h"

char yytext[100];
FILE *yyin;

int main(int argc, char **argv) {
    uniteCourante;
    yyin = fopen(argv[1], "r");
    if(yyin == NULL){
        fprintf(stderr, "impossible d'ouvrir le fichier %s\n", argv[1]);
        exit(1);
    }
    uniteCourante = yylex();
    while (uniteCourante != FIN) {
        printf("%s, %d\n", yytext, uniteCourante);
        uniteCourante = yylex();
    }
    return 0;
}
```