DEUG MIAS2. Institut Galilée
Option : Langages formels et compilation. 1999–2000
TP 2.

Des renseignements sur LEX vous sont donnés dans les pages ci–jointes mais la véritable référence reste le 'TP LEX and YACC User Manual'.

Exercice 1.

Editer le programme suivant dans un fichier `ESSAI1.L`

```
%{
           uses lexlib ;
           (*
           PRESQUE RIEN DANS LE PREMIER SECTEUR
           SINON uses lexlib ; ET CE COMMENTAIRE :
           CE SERA RECOPIE DANS ESSAI1.PAS
           *)
%}
%%
[\t ]+    ;   (* instruction vide *)
Q   |
q   begin writeln('fin du programme ESSAI1') ; halt end ;

suis  |
es  |
est  |
sommes  |
etes  |
sont  writeln(yytext, ' : present du verbe ETRE') ;

ai  |
as  |
a  |
avons  |
avez  |
ont  writeln(yytext, ' : present du verbe AVOIR') ;

[A-Za-z]+  writeln(yytext, ' n''est pas dans notre liste') ;

.|\n  writeln(yytext) ;
%%
begin

   yylex

end.
```

- Compiler ce programme en LEX avec l'option `/v` et consulter les fichiers `ESSAI1.PAS` (le programme produit par LEX) et `ESSAI1.LST` (une description de l'automate fini correspondant aux expressions régulières de votre programme LEX).
- Compiler `ESSAI1.PAS` par TPC puis tester le programme `ESSAI1.EXE` : celui–ci est censé analyser la chaîne de caractères que vous entrerez au clavier. Faites plusieurs essais!

Exercice 2.

Enrichir le programme précédent en lui ajoutant d'autres "tribus" linguistiques. Tester le programme `.EXE` correspondant.

Par la suite, le texte à analyser se trouvera dans un fichier ainsi que le résultat de l'analyse : LEX met à notre disposition deux variables

```
yyinput : text ;
yyoutput : text ;
```

auxquelles on peut assigner respectivement le nom d'un "fichier d'entrée" (à analyser) et celui d'un "fichier de sortie" (résultat de l'analyse). Les exercices qui suivent supposent donc que vous savez utiliser les primitives Turbo Pascal relatives aux fichiers.

Exercice 3.

Ecrire un programme LEX qui compte le nombre de lettres, de mots et de lignes d'un fichier (la sortie pourra se faire à l'écran). Exécuter le programme `.EXE` correspondant sur un fichier de votre choix (pas trop précieux!)

Exercice 4.

Ecrire un programme LEX qui recopie un programme en Turbo Pascal en mettant tous ses mots réservés en MAJUSCULES (par exemple le mot BegIN devra être recopié BEGIN). Exécuter le programme `.EXE` correspondant sur un fichier de votre choix (en Turbo Pascal mais pas trop précieux!)

Présentation succinte de LEX.

La version de LEX que nous utilisons est adaptée au langage Turbo Pascal. La lecture de la présentation succinte qui suit n'est pas suffisante pour comprendre toutes les possiblilités qui sont offertes par LEX : il faut aussi consulter le *User Manual* et des fichiers comme, par exemple, `LEXLIB.PAS`.

LEX est un programme qui à partir d'un fichier source `FICHIER.L`, dont on va voir la syntaxe par la suite, produit un programme en turbo Pascal, dont le nom par défaut est `FICHIER.PAS`. Ce dernier peut à son tour être compilé pour produire un programme exécutable `FICHIER.EXE`, qui n'est rien d'autre qu'un analyseur lexical.

Les deux commandes utiles sont respectivement :

```
LEX [options] fichier[.L] [fichier-sortie[.PAS]]
TPC fichier-sortie[.PAS]
```

Lorsqu'on utilise l'option bavarde `/v` (v pour *verbeux*), on obtient un fichier dont le nom a le suffixe `.LST` et qui contient la description "en clair" de l'instrument essentiel de l'analyse lexicale : un automate fini ! Il faut consulter un tel fichier au moins une fois dans sa vie.

Un fichier source LEX se divise en trois sections, séparées par `%%` :

```
%{
     Déclarations en Turbo Pascal
%}
Définitions LEX
%%
règles
%%
Procédures Turbo Pascal auxiliaires
```

Chacune de ces sections peut être vide ! Voyons ce qu'il est cependant possible d'y mettre.

Première section.

Les déclarations en Turbo Pascal seront recopiées, sans modification dans le programme `.PAS` qui est produit par LEX ; ceci se fait au tout début dudit programme, c'est–à–dire au niveau global : on peut y déclarer des constantes, des variables, des procédures et des fonctions, des commentaires ...

Définitions LEX. Le premier type de définition se présente sous la forme

```
nom   expression
```

où `nom` est un identificateur (de type Turbo Pascal) qui pourra être utilisé, dans la suite de la première section et dans la seconde pour représenter l'expression régulière* `expression` ; afin de distinguer cet identificateur de la chaîne des caractères qui sert à le désigner, on l'utilisera entre `{ }`, par exemple, on peut avoir successivement :

```
alpha [a-zA-Z]
alphanum {alpha}|[0-9]
ident {alpha}{alphanum}*
```

La deuxième section : les règles.

Cette section donne à LEX les instructions qui lui sont nécessaires pour définir la fonction

```
function yylex : integer ;
```

La forme la plus simple d'une instruction LEX est

```
expression   instruction_simple ;
```

* cf. ci–dessous les définitions précises en LEX

où `instruction_simple` est une instruction simple de Turbo Pascal, qui peut être vide.

La fonction `yylex` parcourt le fichier à analyser : lorsque qu'une chaîne de caractères `s` de ce fichier répond au modèle donné par `expression`, la fonction `yylex` a pour premier effet l'exécution de l'action définie par l'`instruction_simple` et affecte la valeur `s` à une variable

```
yytext : string ;
```

déclarée par LEX lui–même.

La forme génerale d'une instruction LEX est la suivante :

```
r1  |
r2  |
...
rn  instruction
```

où

- le symbole `|` indique que l'instruction est la même que pour l'expression suivante,
- `instruction` peut être une instruction simple ou composée (dans ce cas, il faudra utiliser la syntaxe habituelle `begin` ... `end`, de plus, si l'instruction en question comporte plusieurs lignes, il faut veiller ne jamais commencer au début d'une ligne car LEX tenterait de la prendre à son propre compte, comme il est dit un peu plus loin).

Lorsqu'une instruction comporte plusieurs expressions, il peut se présenter des cas d'ambiguïté : c'est alors la chaîne la plus longue qui sera choisie : par exemple, si `'a'` répond à `ri` et `'avoir'` à `rj`, cette dernière chaîne sera reconnue et non pas découpée en `'a'` suivie de `'voir'`.

Il faut penser à la fonction `yylex` comme étant essentiellement constituée d'une instruction `case expression of` dont les cas sont

```
      r1, r2, ... , rn : instruction
```

Dernière section.

Lorsque cette section est vide, le programme `.PAS` se termine simplement par

```
      begin
            yylex
      end.
```

Sinon, vous écrivez ici, en Turbo Pascal, la fin de programme que vous souhaitez !

Attention. Un programme LEX est traité ligne par ligne, avec la convention suivante, valable dans les deux premières sections :

une ligne commençant par un blanc est simplement recopiée dans le fichier `.PAS`

une instruction LEX ne doit donc jamais être précédée d'un blanc.

Ainsi se termine la description d'un fichier Lex `.L`.

Voici maintenant ce qu'il est utile de savoir pour écrire des expressions "bien" régulières.

Caractères réservés.

Ce sont les caractères significatifs pour LEX lui–même, c'est–à–dire : `%`, `\`, `|`, `{` et `}`, ainsi que les caractères servant à coder les expressions régulières.

Lorsqu'on désire utiliser un caractère réservé `c` pour sa valeur littérale, on peut utiliser l'une des notations `\c` ou `'c'`, par exemple : `\{`, `'{'`, `'+'`, ...

Les caractères en général.

En dehors des caractères réservés, tout caractère "imprimable" prend toujours sa valeur littérale. Le tableau qui suit permet de résoudre des cas plus spéciaux :

Expression	Description
`\n`	passage à la ligne
`\r`	retour de chariot
`\t`	tabulation
`\b`	espace arrière
`\f`	passage à la page
`\nnn`	caractère de code ASCII `nnn` en octal

Les expressions régulières.

Une expression régulière caractérise un langage régulier ; en fait, une telle expression est le modèle auquel doit correspondre une chaîne de caratères pour appartenir au langage en question. Les tableaux suivants décrivent la façon d'écrire les expressions régulières qui est compréhensible par un programme LEX. Cependant, les expressions ainsi définies ne sont entièrement définies que si on applique les conventions de "précédence" suivantes :

- Les opérateurs `*`, `+`, `?` et `{ }` ont le plus grand degré de précédence,
- vient ensuite la `concaténation`, qui est notée par simple juxtaposion,
- l'opérateur de mise en alternative `|` a le plus bas degré de précédence.

Les parenthèses `( )` permettent de contrevenir volontairement aux conventions précédentes (par exemple `a|b*` est le modèle des chaînes qui commencent par une occurrence de `a` et se poursuivent par un nombre quelconque d'occurrences de `b`, alors que `(a|b)*` est celui de toutes les chaînes que l'on peut écrire avec les caractères `a` et `b`).

Enfin, `< >` et `/` ne peuvent apparaître qu'une fois dans une expression régulière.

Expression	Description	Exemple
`c`	tout caractère `c` non réservé	`a`
`\c`	le caractère réservé `c`, littéralement	`\*`
`'s'`	la chaîne `s`, littéralement	`'*a*'`
`[s]`	tout caractère apparaissant dans `s`	`[abc]`

Expressions de base.

Remarque : Dans une expression du type `[s]`, on peut écrire `s` comme une juxtaposition de "domaines" (des intervalles de la table ASCII) par exemple :

`[a-z]` est l'expression de toute lettre minuscule,
`[A-Za-z0-9]` est l'expression de tout caractère alphanumérique.

<table>
<tr><th>Expression</th><th>Description</th><th>Exemple</th></tr>
<tr><td><code>r1r2</code></td><td><code>r1</code> puis <code>r2</code> (concaténation)</td><td><code>[abc]ef</code></td></tr>
<tr><td><code>r1|r2</code></td><td><code>r1</code> ou <code>r2</code></td><td><code>a|b</code></td></tr>
<tr><td><code>r{m}</code></td><td><code>m</code> occurrences de <code>r</code></td><td><code>r{3}</code></td></tr>
<tr><td><code>r*</code></td><td>un nombre quelconque d'occurrences de <code>r</code></td><td><code>[abc]*</code></td></tr>
<tr><td><code>(r)</code></td><td><code>r</code> elle–même</td><td><code>(a|b)</code></td></tr>
</table>

Opérations de base.

Expression	Description	Exemple
`r?`	zéro ou une occurrence de `r`	`[abc]?`
`r{m,n}`	de `m` à `n` occurrences de `r`	`r{1,5}`
`r+`	un nombre non nul d'occurrences de `r`	`[abc]+`
`.`	tout caractère sauf `\n`	`a.\*`
`[^s]`	tout caractère n'apparaissant pas dans `s`	`[^abc]`

Autres opérations.

Expression	Description	Exemple
`^`	début de ligne	`^abc`
`<x>r`	`r` sous la condition initiale `x`	`<x>abc`
`$`	fin de ligne	`abc$`
`r1/r2`	`r1` si elle est suivie par `r2`	`a/.*b`

Opérations à "contexte".