Maîtrise d'Informatique **J.Bétréma**
Compilation (tronc commun) **Travaux dirigés**

Construction d'analyseurs lexicaux avec `lex`

1. (a) Ecrire un analyseur lexical *lwc* qui simule *wc*, c'est à dire qui compte les lignes, les mots, et les caractères d'un texte. Essayer plusieurs définitions raisonnables pour la notion de *mot*.

   Appliquer *lwc* au fichier source *lwc.l*, et comparer avec les résultats du programme standard *wc*.

   (b) L'espace, le tabulateur et le saut de ligne sont les caractères *blancs*; on appelle *mot* toute suite de caractères alphanumériques, et *paquet* toute suite de caractères non blancs. Modifier *lwc* pour qu'il compte simultanément les lignes, les paquets, les mots, et les caractères.

   Appliquer *lwc* au fichier source *lwc.l*, et comparer avec les résultats du programme standard *wc*.

2. En langage C, une chaîne de caractères littérale est délimitée par des apostrophes doubles (caractère "); à l'intérieur d'une chaîne, $\backslash x$ désigne un caractère spécial, pour $x = t, n, \ldots$ et désigne 'x' lui-même dans les autres cas. Ecrire un analyseur lexical qui isole les chaînes de caractères littérales dans un programme C.

3. Ecrire un analyseur lexical qui élimine les commentaires d'un programme C.

4. Ecrire un programme C qui évalue des expressions postfixées, en *incluant* un analyseur lexical qui identifie les constantes numériques et les opérateurs. Chaque nombre doit être empilé, et chaque opération doit être exécutée, en prenant pour opérandes les deux nombres en sommet de pile; le résultat d'une opération remplace, sur la pile, les opérandes.

   Note : séparer le programme C et l'analyseur lexical.

5. (Février 92)

   Voici la spécification d'un analyseur lexical, dans le format reconnu par `lex`:

```
%%
[A-Z][a-z]*      printf ("IDENT: %s\n", yytext);
\'[a-z]*\'       printf ("CHAINE: %s\n", yytext);
:=               printf ("AFFECTATION\n");
\;               printf ("SEPARATION\n");
.                ; /* ignorer le reste */
%%
```

   On suppose cette spécification compilée, et on appelle `lexico` l'analyseur obtenu. Lorsque le texte fourni en entrée est:

```
Alpha := 'alpha'; Beta := 'beta';
```

voici le résultat de l'exécution de `lexico`:

```
IDENT: Alpha
AFFECTATION
CHAINE: 'alpha'
SEPARATION
IDENT: Beta
AFFECTATION
CHAINE: 'beta'
SEPARATION
```

(a) Donner le résultat de l'exécution de `lexico` lorsque le texte fourni en entrée est:

```
Alpha := 'Alpha'; Beta := 'Beta';
```

(b) Même question lorsque le texte fourni en entrée est:

```
ALPHA := 'alpha'; BETA := 'beta';
```

(c) Dans la spécification, on remplace la ligne:

```
\'[a-z]*\'       printf ("CHAINE: %s\n", yytext);
```

par:

```
\'.*\'           printf ("CHAINE: %s\n", yytext);
```

Quel est maintenant le résultat de l'exécution lorsque le texte fourni en entrée est celui du préambule, à savoir:

```
Alpha := 'alpha'; Beta := 'beta';          ?
```

6. (Janvier 96)

   Pour cet exercice, on définit un *mot* comme une suite de une ou plusieurs lettres minuscules, à l'exclusion de tout autre caractère. On partitionne les mots en deux ensembles :

   - T est l'ensemble des mots qui se terminent par "tant"; par exemple *autant* et *constatant* font partie de T;
   - U dénote l'ensemble complémentaire de T.

   Ecrire un analyseur lexical, dans le format reconnu par *lex*, qui, dans un texte, affiche uniquement les mots de U (un par ligne).

7. (Septembre 96)

   Voici un programme écrit dans le langage reconnu par *lex* :

```
  int banniere = 0;

%%
.*"<BODY".*             {printf ("<BODY>"); banniere = 1;}
.*"</BANNER>".*         banniere = 0;
^"<IMG".*               ;
.*"cgi-bin".*           ;
.*"</HTMLPLUS>".*       printf("</BODY></HTMLPLUS>");
.*                      if (!banniere) printf ("%s", yytext);
\n                      if (!banniere) printf ("\n");
%%
```

Décrire exactement les transformations subies par un texte qu'on soumet à l'action de ce programme (une fois celui-ci compilé, bien entendu).

Corrigé

1. Une solution correcte (j'espère) est la suivante; bien entendu de nombreuses solutions partielles, mais plus simples, sont intéressantes.

```
%{
int
  caractere = 0,
  ligne = 0,
  mot = 0,
  paquet = 0,
  separateur = 1;
%}
mot         [A-Za-z0-9]+
blanc       [ \t]+

%%
{mot}       {
              mot++;
              caractere += strlen (yytext);
              separateur = 0;
            }
{blanc}     {
              if (! separateur)
                paquet++;
              caractere += strlen (yytext);
              separateur = 1;
            }
.           {
              caractere++;
              separateur = 0;
            }
\n          {
              if (! separateur)
                paquet++;
              caractere++;
              ligne ++;
              separateur = 1;
            }
%%

void main () {
  yylex ();
  if (! separateur)
    paquet++;
  printf ("%d ligne(s), %d paquet(s), %d mot(s), %d caractere(s)\n",
    ligne, paquet, mot, caractere);
}
```

2. Cet analyseur isole les chaînes de caractères, même écrites sur plusieurs lignes (avec la contre-oblique de continuation) :

```
  void ecrire (char *s);

ordinaire    [^\"\n\\]
special      \\(.|\n)
caractere    {ordinaire}|{special}
%%
\"{caractere}*\"     ecrire (yytext);
.|\n                 ;
%%
void ecrire (char *s) {
  int n = strlen (s);
  s [n - 1] = '\0';
  printf ("%s\n", s + 1);
}
```

3. La solution :

```
    int com = 0;
%%
"/*"    com = 1;
"*/"    com = 0;
.|\n    if (!com) putchar (yytext [0]);
```

est lente, et fausse si le commentaire se termine par `/*/`. La solution correcte est (attention, l'oblique `/` est un caractère spécial pour *lex*) :

```
a    "*"
b    "/"
x    [^/*]
%%
"/*"({b}|{a}*{x})*{a}+{b}    ;
```

4. Analyseur lexical (`hplex.l`, compilé en `hplex.c`) :

```
signe       [\+\-]
point       [\.]
chiffre     [0-9]
entier      {chiffre}+
decimales   ({point}{chiffre}*)
mantisse    ({entier}{decimales}?|{point}{entier})
exposant    ((e|E){signe}?{entier})
nombre      {signe}?{mantisse}{exposant}?

%%

{nombre}    return NOMBRE;

"+"         { operateur = PLUS; return OPERATEUR; }
"-"         { operateur = MOINS; return OPERATEUR; }
"*"         { operateur = MULT; return OPERATEUR; }
"/"         { operateur = DIV; return OPERATEUR; }

[ \t]+      ;
.           return IMPREVU;
\n          return FIN;
```

```
/* Calculatrice postfixee Hewlett Packard */

#define NOMBRE 1
#define OPERATEUR 2
#define FIN 3
#define IMPREVU 9

#define PLUS 11
#define MOINS 12
#define MULT 13
#define DIV 14

#define MAX 8
#define VIDE 1
#define PLEINE 2

int operateur;

#include <stdio.h>
#include "hplex.c"

/* extern double atof (char *s); */

static int
  erreur_pile,
  sp = 0;

static double
  pile [MAX];

void empiler (double x) {
  erreur_pile = 0;
  if (sp < MAX)
    pile [sp ++] = x;
  else
    erreur_pile = PLEINE;
}

double depiler () {
  if (sp > 0) {
    erreur_pile = 0;
    return pile [-- sp];
  }
  else {
    erreur_pile = VIDE;
    return 0.0;
  }
}
```

```
void afficher_pile () {
  int i;
  for (i = 0; i < sp; i++)
    printf ("%12.4lf\n", pile [i]);
}

void main () {
  double x, y;
  int token, sp0;

  while (token = yylex () )
    switch (token) {
      case NOMBRE:
        empiler ( atof (yytext) );
        if (erreur_pile)
          printf ("Nombre %s ignore (pile pleine)\n", yytext);
        break;
      case OPERATEUR:
        sp0 = sp;
        y = depiler ();
        x = depiler ();
        if (erreur_pile) {
          printf ("Operateur %s ignore\n", yytext);
          sp = sp0;
        }
        else switch (operateur) {
          case PLUS:
            empiler (x + y);
            break;
          case MOINS:
            empiler (x - y);
            break;
          case MULT:
            empiler (x * y);
            break;
          case DIV:
            empiler (x / y);
            break;
        }
        break;
      case FIN:
        afficher_pile ();
        break;
      case IMPREVU:
        printf ("%s : erreur lexicale\n", yytext);
        break;
    }
}
```

5. (a) `Alpha := 'Alpha'; Beta := 'Beta';`

```
IDENT: Alpha
AFFECTATION
IDENT: Alpha
SEPARATION
IDENT: Beta
AFFECTATION
IDENT: Beta
SEPARATION
```

   (b) `ALPHA := 'alpha'; BETA := 'beta';`

```
IDENT: A
IDENT: L
IDENT: P
IDENT: H
IDENT: A
AFFECTATION
CHAINE: 'alpha'
SEPARATION
IDENT: B
IDENT: E
IDENT: T
IDENT: A
AFFECTATION
CHAINE: 'beta'
SEPARATION
```

   (c) Avec la nouvelle définition d'une chaîne, le résultat de l'exécution de `lexico` est :

```
IDENT: Alpha
AFFECTATION
CHAINE: 'alpha'; Beta := 'beta'
SEPARATION
```

6. Cet analyseur lexical isole les mots qui ne se terminent pas par *tant* :

```
%%
[a-z]*tant      ;
[a-z]+          printf ("%s\n", yytext);
.|\n            ;
%%
```

7.
   - La bannière, constituée des lignes situées entre celle qui contient `<BODY` (début du corps du document) et celle qui contient `</BANNER>`, est supprimée ; on ne garde que la balise `<BODY>`.
   - Les lignes qui commencent par `<IMG` (balise d'une image) sont supprimées.
   - Les lignes qui contiennent `cgi-bin` (appel à un exécutable) sont supprimées.