Compilation : analyse lexicale avec lex

Pierre Wodey
ISIMA/LIMOS
Campus des Cézeaux
BP 10125
63173 Aubière Cedex

8 octobre 2008

Table des matières

Chapitre 1

Organisation

Le tp d'analyse lexicale avec lex est organisé en différents fichiers :
- analex.l fichier qui sera traité par 'lex' pour générer le fichier lex.yy.c ;
- le fichier files.h qui déclare les fichiers de trace ;
- le fichier main.c, programme principal ;
- le fichier y.tab.h normalement, généré par yacc ;
- les fichiers synfr.h et syngb.h qui détermine la langue pour les mots-clés ;
- les fichiers table.h et table.c qui contiennent la déclaration de la table des symboles.

Il n'y a pas d'analyse syntaxique, la fonction "main", se contentant d'appeler la fonction "yylex".

1.1 Fichier files.h

"files.h" 2 ≡

```
/* --------------- files.h -------------------------------- */
/*                                                          */
/*  Fichier d'entete qui contient les definitions des       */
/* pointeurs sur les fichiers qui sont definis dans le      */
/* fichier main.c                                           */
/*                                                          */
/*  Fabrice BARAY - ISIMA - 1996-1997                       */
/* -------------------------------------------------------- */

#ifndef FILES_H
#define FILES_H

#include <stdio.h>

extern FILE *lexi;  /* fichier contenant le resultat de l'analyse lexicale */

#endif

/* --------------- fin de files.h ------------------------- */
```

◊

1.2 Fichiers table.c et table.h

"table.h" 3a ≡

```
/* ---------------------------------------------------------------------- */
/* table.h                                                                */
/*                                                                        */
/*  Definition de la table des symboles                                   */
/*                                                                        */
/*  Fabrice BARAY - ISIMA - 1996-1997                                     */
/* ---------------------------------------------------------------------- */

#ifndef TABLE_H
#define TABLE_H

#define MAX_LEX 100

/* Cette table contient tous les lexemes qui apparaissent dans
  le fichier source que traite le compilateur. */
char *tab_symb[MAX_LEX];

/* Fonction d'initialisation de la table des symboles */
void init_table();

#endif
/* ---------------------------------------------------------------------- */
```

◊

"table.c" 3b ≡

```
/* ---------------------------------------------------------------------- */
/*  table.c                                                               */
/*                                                                        */
/*  Fonctions utilisant la table des symboles                             */
/*                                                                        */
/*  Fabrice BARAY - ISIMA - 1996-1997                                     */
/* ---------------------------------------------------------------------- */


#include "table.h"
#include <stdlib.h>
#include <string.h>

/* Fonction d'initialisation de la table des symboles */
void init_table() {
   int i;

   for(i=0;i<MAX_LEX;tab_symb[i++]=NULL);
}

/* ---------------------------------------------------------------------- */
// fonction ChercheIdent()
int ChercheIdent(char *mot) {
   int i=0;

   while (tab_symb[i]&&strcasecmp(tab_symb[i],mot)) i++;

   if (!tab_symb[i])
      tab_symb[i]=strdup(mot);
   return i;
}
```

◊

1.3 Fichier y.tab.h

Ce fichier a été généré par l'outil yacc, il déclare YYSTYPE le type de yylval, la variable yylval et les constantes pour les lexèmes déclarés en token.

"y.tab.h" 4 ≡

```
typedef union  {
   int   entier;
   int   no_ident;
   char *chaine;
} YYSTYPE;
YYSTYPE yylval;
# define CSTE_ENTIERE 257
# define CSTE_CHAINE 258
# define IDENT 259
# define LE 260
# define GE 261
# define NE 262
# define AFFECT 263
# define AND 264
# define ARRAY 265
# define FBEGIN 266
# define CONST 267
# define ELSE 268
# define END 269
# define FUNCTION 270
# define IF 271
# define NIL 272
# define NOT 273
# define OF 274
# define OR 275
# define PROGRAM 276
# define RECORD 277
# define RETURN 278
# define STRING 279
# define THEN 280
# define TYPE 281
# define VAR 282
# define WHILE 283
# define XOR 284
# define INTEGER 285
```

◊

1.4 Fichiers synfr.h et syngb.h

Il suffit de choisir l'un des deux pour choisir la langue. Ces fichiers contiennent un tableau qui associe à une chaîne de caratères en français ou anglais le numéro du token défini dans y.tab.h.

"syngb.h" 5a ≡

```
/* ----------------------- syngb.h ---------------------------------- */
/* Tableau contenant le syntaxe du langage                            */
/*    en anglais                                                      */
/*                                                                    */
/*  Fabrice BARAY - ISIMA - 1996-1997                                 */
/* ------------------------------------------------------------------ */

#ifndef SYNGB_H
#define SYNGB_H

static const struct {
 char *nom;
 int unitlex;
} MotsRes[] = {
 { "and",            AND        },
 { "array",          ARRAY      },
 { "begin",          FBEGIN     },
 { "const",          CONST      },
 { "else",           ELSE       },
 { "end",            END        },
 { "function",       FUNCTION   },
 { "if",             IF         },
 { "integer",        INTEGER    },
 { "nil",            NIL        },
 { "not",            NOT        },
 { "of",             OF         },
 { "or",             OR         },
 { "program",        PROGRAM    },
 { "record",         RECORD     },
 { "return",         RETURN     },
 { "string",         STRING     },
 { "then",           THEN       },
 { "type",           TYPE       },
 { "var",            VAR        },
 { "while",          WHILE      },
 { "xor",            XOR        },
 { NULL,             IDENT      }
};

#endif
/* ----------------------- syngb.h ---------------------------------- */
```

◊

"synfr.h" 5b ≡

```
/* --------------------- synfr.h ------------------------------------ */
/* Tableau contenant la syntaxe du langage                            */
/*    en francais                                                     */
/*                                                                    */
/*  Fabrice BARAY - ISIMA - 1996-1997                                 */
/* ------------------------------------------------------------------ */

#ifndef SYNFR_H
#define SYNFR_H

static const struct {
 char *nom;
 int unitlex;
} MotsRes[] = {
 { "et",              AND       },
 { "tableau",         ARRAY     },
 { "debut",           FBEGIN    },
 { "constante",       CONST     },
 { "sinon",           ELSE      },
 { "fin",             END       },
 { "fonction",        FUNCTION  },
 { "si",              IF        },
 { "entier",          INTEGER   },
 { "vide",            NIL       },
 { "non",             NOT       },
 { "de",              OF        },
 { "ou",              OR        },
 { "programme",       PROGRAM   },
 { "enreg",           RECORD    },
 { "retourne",        RETURN    },
 { "chaine",          STRING    },
 { "alors",           THEN      },
 { "type",            TYPE      },
 { "var",             VAR       },
 { "tantque",         WHILE     },
 { "oux",             XOR       },
 { NULL,              IDENT     }
};

#endif

/* --------------------- synfr.h ------------------------------------ */
```

◊

Chapitre 2

Le fichier lex

Ce fichier à été construit par Fabrice Baray, j'y ai apporté quelques modifications.

Le fichier "analex.l" contient la description du traitement de l'analyse lexicale pour l'outil lex qui génère le code c de la fonction d'analyse lexicale. L'outil lex génère le fichier lex.yy.c.

Un fichier lex décrit les actions et les valeurs retournées (lexèmes) à l'analyse syntaxique en passant par l'identification de motifs.

2.1 Structure globale du fichier

Un fichier lex comporte 5 parties.La première permet de fixer un certain nombre d'otpions.

La seconde contient du code C qui sera directement placé dans le fichier résultant lex.yy.c. Ce code est placé entre le `%{` et le `%}`.

Les motifs peuvent être définis par des expressions régulières qui portent un nom. Dans ce cas leur définition est faite dans la partie "les motifs nommés", qui est la troisième partie.

La quatrième partie décrit les traitements associés aux motifs, qu'ils soient définit par des motifs nommés ou directement par des chaînes de caractères. Cette partie du fichier est encadrée par les symboles `%%`.

Enfin on peut mettre à la fin du fichier du code C qui sera placé en fin du fichier résultant.

`"analex.l"` 7 ≡

⟨les options 8a⟩
`%{`
⟨code C pour les déclarations 8b⟩
`%}`
⟨les motifs nommés 9b⟩
`%%`
⟨le traitement associé aux motifs 10⟩
`%%`
⟨code C supplémentaire 13c⟩
◊

2.2 Les options

Il est possible de placer des options dans le fichier, sous le format "%option nom". Ici nous choissons l'option "yylineno" qui pertmet d'avoir dans la variable "yylineno" le numéro de la ligne courante, utile pour les messages d'erreurs.

⟨les options 8a⟩ ≡

```
%option yylineno
```

◊

Macro référencée dans fragment 7.

2.3 Le code C en début de fichier

2.3.1 Structure du code C

La première partie de code C contient les inclusions de fichiers et les déclarations de variables et fonctions. Ce code se retrouvera tel quel dans le fichier résultant "lex.yy.c".

Il est constitué d'un ensemble d'inclusions de fichiers, de la déclaration de l'option DEBUG_L ou non (pour les traces en dans le fichier lexi) et d'un ensemble de fonctions spécifiques dont le code est placé en fin de fichier.

⟨code C pour les déclarations 8b⟩ ≡

⟨les inclusions de fichiers 8c⟩

```
/* Definition d'une constante pour valider ou inhiber les ecritures
  des lexemes reconnus dans un fichier d'extension ".fbll"             */
#undef DEBUG_L
#define DEBUG_L


/* ------------------------------------------------------------------ */
/* Definition des prototypes de quelques methodes utiles.             */
```

⟨les fonctions et structures locales ou externes 9a⟩

◊

Macro référencée dans fragment 7.

2.3.2 Les fichiers inclus

Outre les trois fichiers standards, il faut toujours inclure le fichier "y.tab.h" construit par l'outil yacc. Ce fichier contient les définitions des numéros de lexèmes (token dans yacc) sous la forme de constantes, mais aussi la structure de données des lexèmes qui portent des valeurs ("YYSTYPE") et la variable "yylval" qui sert à passer la valeur de tels lexèmes.

Pour notre application il faut également inclure le fichiers "files.h" qui contient la déclaration du fichier de trace (variable "lexi") et le fichier "table.h" qui contient les déclarations de la table des symboles.

⟨les inclusions de fichiers 8c⟩ ≡

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "y.tab.h"

#include "files.h"
/*------------------------------------------------------------------ */
/*  Inclusion du fichier contenant la definition du tableau pour la
   definition des la syntaxe des mots cles du langage. On peut ainsi
   gerer differents langages sources, comme par exemple le francais
   (synfr.h) et l'anglais (syngb.h).                                  */

#include "synfr.h"◊
```

Macro référencée dans fragment 8b.

2.3.3 Les fonctions et structures

Nous définissons quelques fonctions utiliséees dans le code des traitements liés aux motifs :

- "ChercheMotRes()", qui cherche si le mot passé en paramètre est dans la table des mots réservés. S'il y est, cette fonction retourne le numéro du token du mot réservé, sinon celui du token IDENT ;
- "ChercheIdent()", qui cherche dans la table des symboles l'existance du mot passée en paramètre. S'il y est elle renvoie son numéro, sinon elle l'insère dans la table et renvoie son numéro. Elle est externe.

⟨les fonctions et structures locales ou externes 9a⟩ ≡

```
int ChercheMotRes(char *);
extern int ChercheIdent(char *);
extern char * tab_symb[];
◊
```

Macro référencée dans fragment 8b.

2.4 Les motifs

Les motifs définis par des expressions régulières :

- un délimiteur "delim" est soit un espace, soit une tabulation, soit un retour chariot ;
- un blanc est une suite de délimiteurs de longueur supérieure à 1 ;
- une lettre est un caractère alphabétique ;
- un chiffre est un caractère numérique ;
- un symbole est une lettre, un chiffre ou le caractère '_' ;
- un identificateur est une lettre suivie d'un nombre quelconque de symboles. Le motif "ident" dénotera aussi bien un mot réservé du langage qu'un identificateur ;
- un entier est une suite de longueur supérieure ou égale à 1 de chiffres ;
- le motif autre désigne tout ce qui ne sera pas reconnu comme constituant un lexème.

⟨les motifs nommés 9b⟩ ≡

```
/* ------------------------------------------------------------------ */
/*  Description des caracteres constituant un programme               */
delim         [ \t\n]
blanc         {delim}+
lettre        [a-zA-Z]
chiffre       [0-9]
symbole       {lettre}|{chiffre}|"_"
ident         {lettre}{symbole}*
entier        {chiffre}+
BadId         {entier}{ident}
comm           \{[^\}]*\}
dbquote        \"\"
str            \"([^\n\"]|{dbquote})*\"
autre         .
```

◊

Macro référencée dans fragment 7.

2.5 Les motifs et les traitements

Cette partie du fichier décrit, pour chaque motif les traitements qui lui sont associés. Les traitements seront exécutés après reconnaissance d'un motifs dont la chaîne de caractères reconnue est toujours présente dans la variable prédéfinie "yytext".

Le format de description est le suivant :

```
motifs traitement;
```

La partie "motifs" est constituée d'un ou plusieurs motifs (dans ce cas séparés par le symbole |) pour lesquels le même traitement s'applique. Un motif est soit une chaîne de caractères soit un nom de motif entre accolades.

Attention, dans cette partie du fichier, il n'est pas possible de mettre des commentaires hors des zones de traitement.

S'il n'y a rien à faire, le traitement est vide, sinon on décrit le code C à exécuter (entre accolades). Remarque, s'il n'y a pas d'instruction "return" dans le traitement, on ne sort pas de l'exécution de la fonction d'analyse lexicale, qui passe donc au motif suivant. Nous distingunons les cas suivants :

- les blancs, pour lesquels nous ne faisons rien, ce qui signifie que l'analyseur lexical cherchera le lexème suivant ;
- le motif "ident", mot réservé ou identificateur ;
- les motifs sur deux caractères ;
- les motifs sur un caractère ;
- les entiers ;
- les commentaires ;
- les chaînes de caractères ;
- les autres cas.

⟨le traitement associé aux motifs 10⟩ ≡

```
{blanc}        ;
```

⟨expression "ident" 11a⟩

⟨lexèmes sur deux caractères 11b⟩

⟨lexèmes sur un caractère 11c⟩

⟨expression "entier" 12a⟩

⟨commentaire 12b⟩

⟨chaîne de caractères 13a⟩

⟨autres cas 13b⟩

◊

Macro référencée dans fragment 7.

Un motif correspondant à la règle "ident" est soit un mot réservé soit un identificateur dont la chaîne de caractères est dans "yytext". Dans le premier cas, il faut retourner comme lexème le numéro du mot réservé, dans le second la valeur IDENT. C'est ce que détermine la fonction "ChercheMotRes()", cette valeur est placée dans la variable "i". Dans le cas d'un identificateur, on place le numéro de l'identificateur, obtenu par l'appel à la fonction "ChercheIdent()", dans la variable prédéfinie "yylval" via l'entier "j".

⟨expression "ident" 11a⟩ ≡

```
{ident}        { int i,j;
                 i=ChercheMotRes(yytext);
                 if (i==IDENT) {
                   j=ChercheIdent(yytext);
                   yylval.no_ident=j;
                  }
                  #ifdef DEBUG_L
                    if (i==IDENT)
                      fprintf(lexi,"AL ident  : %i %s\n",j,tab_symb[j]);
                    else
                       fprintf(lexi,"AL motcle : %i\n",i);
                   #endif
                 return i;
               }
```

◊

Macro référencée dans fragment 10.

Pour les motifs sur deux caractères, nous n'avons pas défini d'expression régulière mais donnons directement la chaîne caractère correspondante. Toutefois, nous renvoyons un entier défini comme token dans l'analyse syntaxique. Ainsi nous distinguons chacun des cas.

⟨lexèmes sur deux caractères 11b⟩ ≡

```
"<="                { return LE; }
">="           { return GE; }
"<>"           { return NE; }
":="           { return AFFECT; }
```

◊

Macro référencée dans fragment 10.

Pour les motifs sur un caractère, nous renvoyons simplement le caractère correspondant "yytext[Ø]" comme identification du lexème. Ceci signifie que ces lexèmes ne sont pas définis en tant que token dans l'analyse lexico-syntaxique.

⟨lexèmes sur un caractère 11c⟩ ≡

```
"-"            |
"+"            |
"*"            |
"/"            |
"="            |
"<"            |
">"            |
"("            |
")"            |
"["            |
"]"            |
"."            |
","            |
";"            |
":"            |
"@"            |
"^"            {
                 #ifdef DEBUG_L
                    fprintf(lexi,"AL symb   : %c\n",yytext[0]);
                 #endif
                 return yytext[0];
               }
```

◊

Macro référencée dans fragment 10.

Pour les entiers, nous retournons le lexème CSTE_ENTIERE et, dans la variable "yylval", nous palçons la valeur de l'entier en utilisant la fonction de conversion "atol" qui convertit la chaîne de caractères de l'entier en une valeur de type "int".

⟨expression "entier" 12a⟩ ≡

```
{entier}       {
                yylval.entier = atol(yytext);
                #ifdef DEBUG_L
                  fprintf(lexi,"AL entier : %s\n",yytext);
                #endif
                return CSTE_ENTIERE;
               }
```

◊

Macro référencée dans fragment 10.

Les commentaires commencent par le caractère { et finissent par le caractère }. Sur l'apparition de l'accolade ouvrante nous appelons la fonction "comment())" qui saute tous les caractères jusqu'à ce quelle rencontre l'accolade fermante. Rien n'est retourné, l'analyse lexicale cherchera donc le motif suivant. Les commentaires sont donc complètement ignorés par l'analyse synbtaxique.

⟨commentaire 12b⟩ ≡
```
{comm}        {
               #ifdef DEBUG_L
                 fprintf(lexi,"Commentaire : %s\nFin du commentaire\n",yytext);
               #endif
              }◊
```
Macro référencée dans fragment 10.

Pour les chaînes de caractères, entourées par le symbole ", nous appelons, à l'apparition du premier ", la fonction "string" qui vérifie et récupère les caractères et les place dans la variable "yylval". La valeur retournée est le token CSTE_CHAINE.

⟨chaîne de caractères 13a⟩ ≡
```
{str}       {
              #ifdef DEBUG_L
                 fprintf(lexi,"chaine de car : %s\n", yytext);
              #endif
                 yylval.chaine = strdup(yytext); /* On duplique la chaine */
                 return CSTE_CHAINE; }
◊
```
Macro référencée dans fragment 10.

Tous les autres cas, donc autres caractères ou motifs, sont refusés par l'analyse lexicale. Nous générons un message d'erreur qui utilise la variable "yylineno".

⟨autres cas 13b⟩ ≡
```
{autre}|{BadId}  {
             fprintf(stderr, "Ligne %d : motif illegal %s, 1er car (ASCII = %d)\n",
                      yylineno, yytext, (int)yytext[0]);
             //exit(1);
           }
◊
```
Macro référencée dans fragment 10.

2.6 Code C en fin de fichier

⟨code C supplémentaire 13c⟩ ≡

```
/* ------------------------------------------------------------------ */
/* Cherche si mot est un mot reserve.
   Si c'est le cas, retourne son unite lexicale.
   Sinon retourne IDENT.
                          */
int ChercheMotRes(char *mot) {
   int i = 0, trouve = 0;

   while (!trouve && MotsRes[i].nom)
      trouve = !strcasecmp(MotsRes[i++].nom, mot);
   return (MotsRes[trouve?--i:i].unitlex);
}
```

◊

Macro référencée dans fragment 7.