# Langages et Automates

Lex & Accent

# Plan

1. Introduction
2. Lex
3. Accent

# Introduction

1. Lex et Accent sont des outils Unix basés sur le langage de programmation C
2. Lex construit automatiquement un analyseur lexical ou lexer à partir d'expressions régulières définissant les motifs des unités lexicales
3. Accent permet de produire un analyseur syntaxique ou parser à partir de toute grammaire non contextuelle
4. Au cours de l'analyse, Accent est capable d'exécuter des actions sémantiques
   Accent permet ainsi de produire des interpréteurs ou des compilateurs complets c'est-à-dire Accent peut effectuer les phases d'analyse sémantique, d'interprétation et/ou de génération de code
5. Accent s'utilise la plupart du temps conjointement à Lex. Lex fournit les unités lexicales à Accent qui se charge d'analyser leur organisation syntaxique
   Une erreur de syntaxe est détectée toutes les fois où le texte d'entrée ne satisfait aucune des règles de la grammaire

# Lex (Compilation)

A. Compilation

La compilation d'un programme source écrit dans le langage Lex s'effectue en deux étapes :

1) Le compilateur *Lex* transforme les expressions régulières en un automate fini, et produit, dans un fichier de nom `lex.yy.c`, du code qui simule cet automate.

   `nomAnalyseur.lex` → `compilateur lex` → `lex.yy.c`

   Ligne de commande : `lex nomAnalyseur.lex`

# Lex (Compilation)

2) Le fichier `lex.yy.c` est compilé par un compilateur C en un analyseur lexical

`lex.yy.c` → `compilateur C` → `a.out`

Ligne de commande : `gcc [-o nom exécutable] lex.yy.c -ll`
L'option `-ll` est utilisé pour lier le fichier objet créé par le compilateur avec la librairie Lex
**`-o nom exécutable`** est optionnel et permet de donner un nom précis à l'analyseur généré. Sans cette option, l'analyseur aura pour nom a.out

L'analyseur obtenu prend en entrée un flot de caractères et produit en sortie un flot d'unités lexicales

Flux d'entrée → nom exécutable → Suite d'unités lexicales

# Lex (Structure du programme)

A. Compilation
B. Structure d'un programme Lex

Un programme Lex est composée de trois sections séparées par %% :

Déclarations {voir exemple transparent 6}

%%

Règles de traduction voir exemple transparent 8}

%%

Fonctions auxiliaires {voir exemple transparent 10}

**Remarque** : Les sections Déclarations et Fonctions auxiliaires sont optionnelle

# Lex (Section d´eclarations - Exemple)

```
%{
/* Fichier exemple.lex */
/* Section en C : Declarations de constantes et de variables Insertions de
bibliotheques */
/* Declaration de l action d affichage des entiers */
void afficher_entier(char *ent);

/* Declaration de l action d affichage des reels */
void afficher_reel(char *reel);

/* Declaration de l action d affichage des identificateurs */
void afficher_id(char *id); %}

/* Definition de macros */
separateur [ \t\n]
chiffre [0-9]
lettre [a-zA-Z]
```

# Lex (Section déclarations)

- Contient des déclarations en C de variables et de constantes ainsi que des insertions de bibliothèques C servant aux sections suivantes
  Ces déclarations et ces insertions sont introduites entre les délimiteurs %{ et %}
  **Remarque** : Tout ce qui se trouve entre ces délimiteurs est copié dans le fichier yy.lex.c lors de la phase de compilation Lex
- Contient des déclarations de macros. Une macro est un nom donné à un ensemble de caractères, nom pouvant être utilisé dans la deuxième section
  **Remarque** : Les d´eclarations de macros commencent en 1 ère colonne du fichier
- Exemple transparent 6 : La partie d´eclarations contient la d´eclaration C de 3 actions utilisées dans la deuxième section et la déclaration des macros suivantes :
  - **Séparateur** désignant l'espace, la tabulation ou le retour à la ligne
  - **Chiffre** désignant n'importe quel chiffre
  - **Lettre** désignant n'importe quelle lettre minuscule ou majuscule

# Lex (Section règles de traduction - Exemple)

```
%%
"+"                          { printf ("Addition\n"); }
"-"                          { printf ("Soustraction\n"); }
":="                         { printf ("Affectation\n"); }
{chiffre}+                   { /* Une suite de chiffres est rencontree
                                  Elle est stockee dans la variable predefinie yytext */
                                  afficher_entier(yytext); }


{chiffre}*\.{chiffre}+               { /* Une suite de chiffres suivie d un point et d une suite de chiffres est
rencontree

                                      Elle est stockee dans la variable predefinie yytext */
afficher_reel(yytext); }
id                           { printf ("Mot id trouve\n"); }
{lettre}+                    { /* Une suite de lettres est rencontree
                                  Elle est stockee dans la variable predefinie yytext */ afficher_id(yytext);
}
{separateur}+                ;/* Suppression des espaces, tabulations et retours a la ligne : Pas d'action */

.                            { /* Caractere . designe n importe quel caractere excepte le retour a la ligne */
                                  printf("Mauvais caractere : %c\n", yytext[0]); }
%%
```

# Lex (Section règles de traduction)

- Chaque règle de traduction est formée de deux parties séparées par des espaces :
  - Un motif $M_i$ : Expression régulière commençant en 1ère colonne du fichier
    Cette expression peut utiliser les macros de la section Déclarations
  - Une action $A_i$ : Fragment de code C

- L'analyseur lexical créé par Lex lit l'entrée, caractère par caractère, jusqu'à ce qu'il trouve le plus long préfixe de l'entrée reconnu par l'un des motifs d´efinis
  Si ce motif est $M_i$, l'analyseur exécute l'action associé $A_i$

- Lors de la phase de compilation Lex, l'ensemble des règles de traduction est traduite dans une fonction d'analyse lexicale appelée yylex() du fichier `yy.lex.c`

- Exemple transparent 8 : Lorsqu'une suite non vide de chiffres est rencontrée (expression régulière `{chiffre}+`), l'analyseur la range dans la variable `yytext` (transparent 12) et appelle l'action afficher entier située dans la troisième partie de l'analyseur

# Lex (Section fonctions auxiliaires - Exemple)

```
/* Section des fonctions auxiliaires C */

void afficher_entier(char *ent){
/* La chaine de caracteres ent contient une suite de chiffres L'instruction atoi
convertit cette chaine en un entier */
     printf ("Un entier est reconnu : %d\n",atoi(ent));
}

void afficher_reel(char *reel){
/* La chaine de caracteres reel contient une suite de chiffres suivie d un point
suivi d une suite de chiffres L'instruction atof convertit cette chaine en un reel */
     printf ("Reel : %.2f\n",atof(reel));
}

void afficher_id(char *id){
/* La chaine de caracteres id contient une suite de lettres */
     printf ("Identificateur : %s\n",id);
}
```

# Lex (Section fonctions auxiliaires)

- Cette section contient toutes les fonctions additionnelles C utilisées dans les actions de la section Règles de traduction

- Cette section peut éventuellement contenir un programme principal `main()` permettant de spécifier l'utilisation globale de l'analyseur lexical
  **Remarques** :
  - Dans le cas où cette section ne contient pas de `main()`, l'appel à la fonction d'analyse `yylex()` est fait automatiquement
  - Dans le cas contraire, le `main()` doit posséder un appel à `yylex()` afin de lancer l'analyse

- Lors de la phase de compilation Lex, cette section est recopiée dans le fichier `yy.lex.c` à la suite de la fonction d'analyse `yylex()`

- Exemple transparent 10 : L'action afficher entier est exécutée lorsqu'une suite non vide de chiffres est rencontrée (expression régulière `{chiffre}+` de la partie règles de traduction)

# Lex (Fonctionnement de l’analyseur)

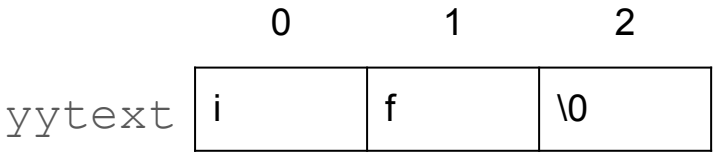
A. Compilation
B. Structure d’un programme Lex
C. Fonctionnement d’un analyseur lexical Lex

L’analyseur :

1) Lit l’entrée caractère par caractère et sélectionne le plus long préfixe de l’entrée reconnu par l’un des motifs définis
2) Exécute l’action associée à ce motif
3) Recommence le traitement sur le morceau de texte suivant

A tout moment, la partie du texte reconnu est stockée dans une variable prédéfinie de type chaîne de caractères, appelée `yytext`
La longueur de la partie du texte reconnu est stockée dans la variable prédéfinie `yyleng`

Exemple : Si le texte reconnu est `if`, la variable `yyleng` a pour valeur 2, et :



# Lex (Résolution des conflits)

A. Compilation
B. Structure d'un programme Lex
C. Fonctionnement d'un analyseur lexical Lex
D. Résolution des conflits

Lorsque plusieurs préfixes de l'entrée sont reconnus par un ou plusieurs motifs :

- Le motif reconnaissant le plus long préfixe est sélectionné
- Si le plus long préfixe est reconnu par plusieurs motifs, le motif listé en premier dans le programme Lex est sélectionné

Exemple : Soient les 3 motifs suivants :

a
aab
a*b+

- Le mot **abbba** a plusieurs préfixes reconnus : **a** par le 1$^{er}$ motif, **abb** par le 2$^{ème}$ motif, **ab**, **abb** et **abbb** par le 3$^{ème}$ motif
  Le 3$^{ème}$ motif est sélectionné car il reconnait le plus long préfixe et le préfixe reconnu est **abbb**
- Le plus long préfixe reconnu du mot **abb** est **abb**. Il est reconnu par les 2$^{ème}$ et 3$^{ème}$ motifs mais c'est le 2$^{ème}$ motif qui est sélectionné car il est listé en premier

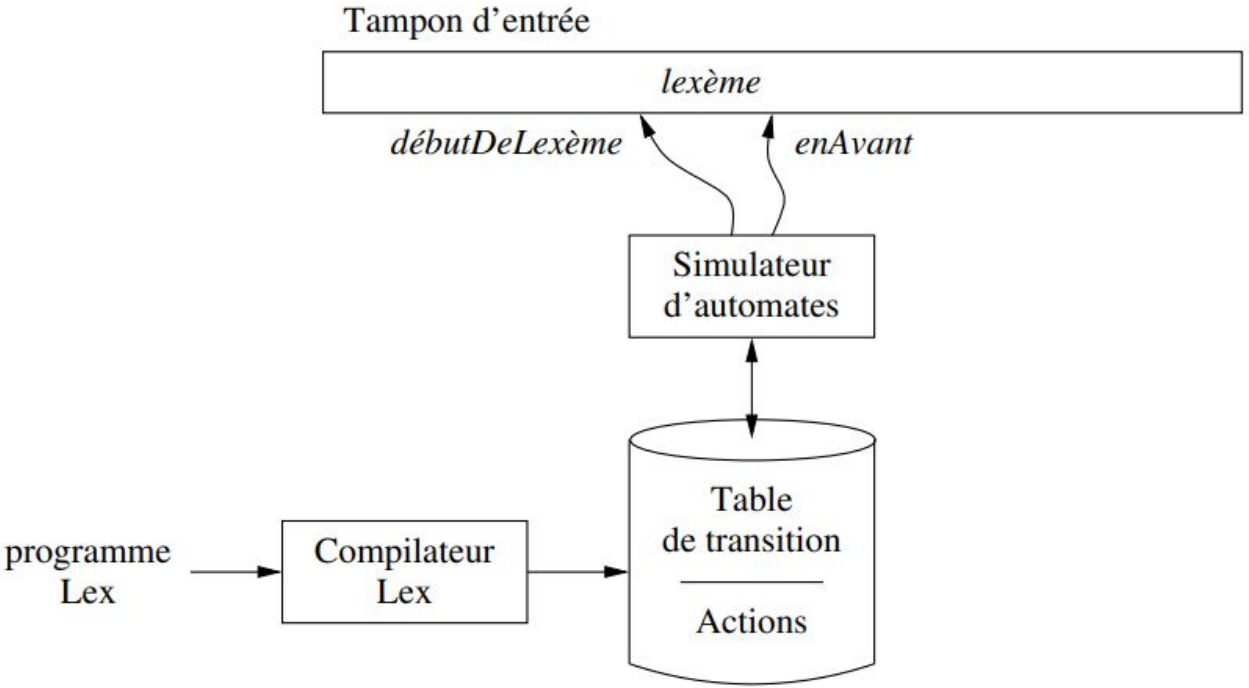
# Lex (Structure de l’analyseur)

A. Compilation
B. Structure d’un programme Lex
C. Fonctionnement d’un analyseur lexical Lex
D. Résolution des conflits
E. Structure de l’analyseur

L’analyseur lexical produit par Lex est constitué :

- D’un programme fixe qui simule un automate
- De composants créés à partir du programme Lex par le compilateur Lex :
    - Une table de transition représentant l’automate
    - Les fonctions auxiliaires
    - Les actions du programme Lex qui apparaissent sous forme de fragments de code appelés au moment adéquat par le simulateur d’automates

# Lex (Structure de l'analyseur)



# Lex (Structure de l’analyseur)

Construction de l’automate

1ère étape : Construire un AFN pour chacune des expressions régulières du programme Lex à l’aide de l’algorithme de construction de Thompson

2ème étape : Combiner tous les AFN en un seul en introduisant un nouvel état initial relié par ε-transitions aux états initiaux des AFN

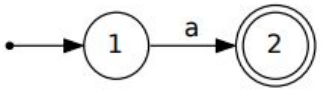
3ème étape : Déterminiser l’AFN

# Lex (Structure de l'analyseur - Exemple)

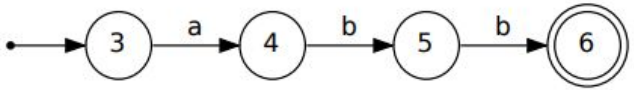
Exemple : Soient les motifs **a**, **abb** et **a*b+**

1ère étape :

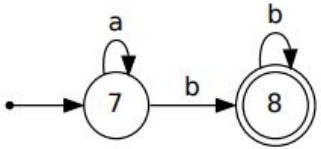
AFN pour **a** :




AFN pour **abb** :

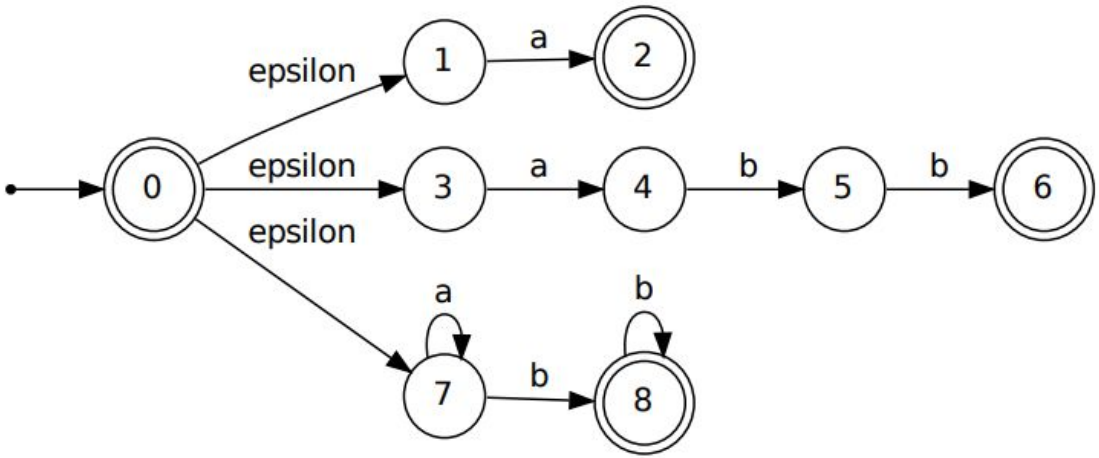



AFN pour **a*b+** :



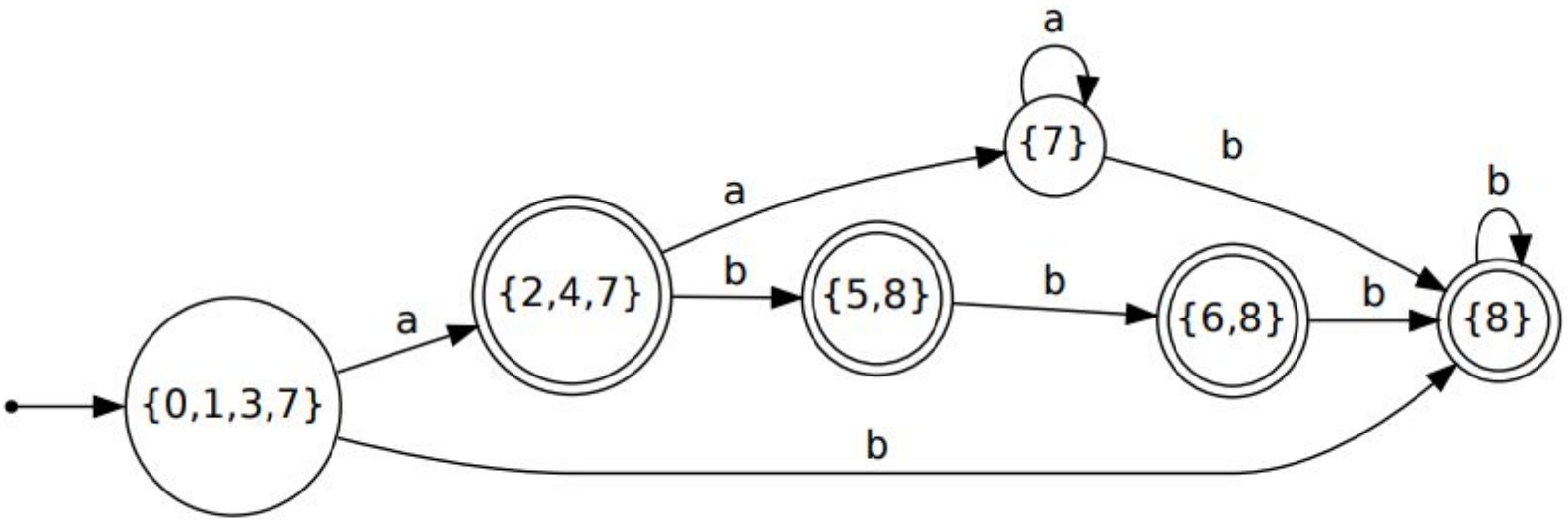
# Lex (Structure de l'analyseur - Exemple)

$2^{ème}$ étape :



# Lex (Structure de l'analyseur - Exemple)

$3^{ème}$ étape : Automate fini déterministe équivalent



# Lex (Structure de l'analyseur)

Reconnaissance de motifs

L'analyseur lexical lit l'entrée caractère par caractère à partir du caractère repéré par le pointeur `débutDeLexème`, le caractère courant étant repéré par le pointeur `enAvant`

A chaque avancé du pointeur `enAvant`, l'analyseur lexical détermine l'état de l'AFD dans lequel il se trouve

Lorsque l'analyseur atteint un caractère pour lequel il n'y a plus d'état, il n'est plus possible de trouver un préfixe plus long de l'entrée reconnu par l'un des motifs

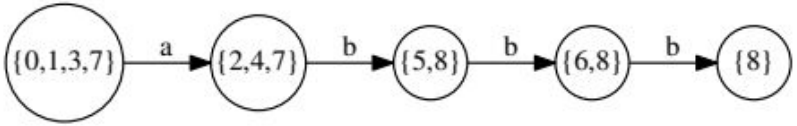
L'analyseur parcourt alors en arrière la suite des états de l'AFD jusqu'à trouver un état de l'AFD contenant au moins un état final de l'AFN. Si cet état contient plusieurs états finaux de l'AFN, c'est le premier motif dans le programme Lex associé à l'un de ces états finaux qui est choisi

L'action associée au motif sélectionné est effectuée et le pointeur `débutDeLexème` est positionné à la fin du préfixe reconnu

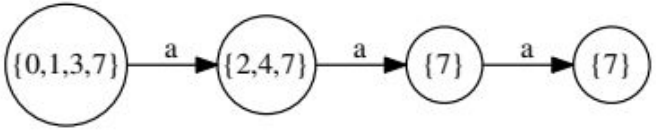
# Lex (Structure de l'analyseur)

Exemples :

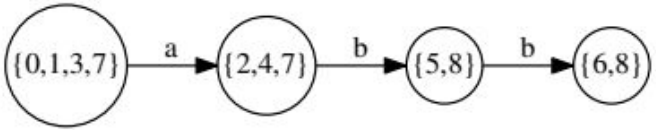
Entrée **abbba** :




Entrée **aaa** :



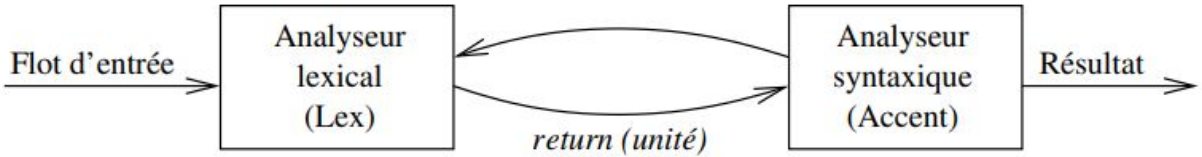

Entrée **abba** :



# Accent (Utilisation)

A. Utilisation

- Accent s'utilise conjointement à Lex. l'analyseur lexical Lex fournit les unités lexicales à l'analyseur syntaxique Accent qui se charge d'analyser leur organisation syntaxique
  Une erreur de syntaxe est détectée toutes les fois où le texte d'entrée ne satisfait aucune des règles de la grammaire




- L'analyseur syntaxique Accent appelle la fonction `yylex()` de l'analyseur lexical Lex toutes les fois où il a besoin d'une unité lexicale provenant du flot d'entrée
  Dans l'analyseur lexical, l'instruction **return(unité)** permet de renvoyer à l'analyseur syntaxique le nom de l'unité lexicale reconnue

# Accent (Compilation)

A. Utilisation
B. Compilation

La création d'un analyseur syntaxique à partir d'Accent s'effectue en trois étapes :

1) Le compilateur Accent produit un fichier `yy.grammar.c` contenant l'encodage en C de la grammaire d´ecrite dans le programme Accent

   `nomAnalyseur.acc` → `compilateur Accent` → `yy.grammar.c`

2) Le compilateur Lex génère le fichier `lex.yy.c`

   `nomAnalyseur.lex` → `compilateur Lex` → `lex.yy.c`

3) Le fichier `yy.grammar.c` est compilé par un compilateur C avec le fichier `lex.yy.c` et 2 fichiers auxiliaires `auxil.c` (transparent 34) et `entire.c` en un analyseur syntaxique

   `yy.grammar.c`, `lex.yy.c`, `auxil.c`, `entire.c` → `compilateur C` → `a.out`

   entire est un analyseur syntaxique g´en´erique qui peut analyser un fichier source à partir de n'importe quelle grammaire non contextuelle

# Accent (Méthode)

Production des fichiers Lex et Accent pour une grammaire donnée

1ère étape : Associer à chaque terminal de la grammaire un token

Un token est un nom donné à un terminal permettant de renvoyer l'unité lexicale reconnue de l'analyseur lexical (fichier Lex) à l'analyseur syntaxique (fichier Accent)

2ème étape : Ecrire le fichier Lex reconnaissant les motifs des terminaux de la grammaire

Dans la partie règles de traduction du fichier Lex, pour chaque motif d'un terminal de la grammaire reconnu, l'analyseur lexical renvoie le nom du token associé au terminal à l'analyseur syntaxique

3ème étape : Ecrire le fichier Accent

# Accent (Méthode - Exemple)

Soit la grammaire suivante :

$$
\begin{array}{rcl}
\mathit{Exp} & \rightarrow & \mathit{Exp} + \mathit{Term} \mid \mathit{Exp} - \mathit{Term} \mid \mathit{Term} \\
\mathit{Term} & \rightarrow & \mathit{Term} * \mathit{Fact} \mid \mathit{Term} \,/\, \mathit{Fact} \mid \mathit{Fact} \\
\mathit{Fact} & \rightarrow & \textbf{entier} \mid (\mathit{Exp}) \mid - \mathit{Exp}
\end{array}
$$

1ère **étape :** Associer à chaque terminal de la grammaire un token

| **Terminaux** | **Tokens** |
|---|---|
| + | PLUS |
| − | MOINS |
| * | MULT |
| / | DIV |

| **Terminaux** | **Tokens** |
|---|---|
| **entier** | ENTIER |
| ( | PAROUV |
| ) | PARFER |

2ème **étape :** Ecrire le fichier *Lex* (voir transparent suivant)

```
%{
/* Fichier exemple.lex */
#include "yygrammar.h" /* Fichier genere lors de la compilation Accent */ char
err[20]; /* Chaine de caracteres pour les erreurs de syntaxe */ %}
/* Definition de macros */
separateur          [ \t]
chiffre             [0-9]

    %%
"+"                 return PLUS; /* Indique au parser que + est reconnu */
"-"                 return MOINS; /* Indique au parser que - est reconnu */
"*"                 return MULT; /* Indique au parser que * est reconnu */
"/"                 return DIV; /* Indique au parser que / est reconnu */
"("                 return PAROUV; /* Indique au parser que ( est reconnu */
")"                 return PARFER; /* Indique au parser que ) est reconnu */
{chiffre}+      return ENTIER; /* Indique au parser qu'un entier est reconnu */
{separateur}+       ;               /* Elimination des espaces */
\n                  yypos++; /* Compte le nombre de lignes du fichier source */
.                   { sprintf(err,"Mauvais caractere %c",yytext[0]);
                        yyerror(err); /* Generation d'une erreur de syntaxe */
                    }
%%
```

# Accent (Structure du programme)

Structure d’un programme Accent

Un programme Accent comprend trois parties :

- Déclarations C {voir exemple transparent 28}
- Déclaration des tokens {voir exemple transparent 30}
- Règles de traduction {voir exemple transparent 32}

# Accent (Section déclarations C - Exemple)

```
/* Fichier exemple.acc */

%prelude { /* Code C */
    /* Inclusion de bibliotheques C */
    #include <stdio.h>
    #include <malloc.h>

    /* Action de fin d analyse */
    void fin_analyse(){
        printf("Fin de traitement \n");
        printf("Syntaxe correcte \n");
    }
}
```

# Accent (Section déclarations C)

- Section délimitée par `%prelude{` et `}` contenant du code C : Insertion de bibliothèques et déclarations de constantes, variables, fonctions
- Lors de la phase de compilation Accent, cette partie est recopiée dans le fichier `yygrammar.c`
- Exemple transparent 28 :
  - Inclusion des bibliothèques `stdio.h` et `malloc.h`
  - Définition de l'action fin analyse qui sera appelée dans la partie règles de traduction de l'analyseur syntaxique (transparent 32)

# Accent (Section déclaration des tokens - Exemple)

/* Declaration des tokens */

```
%token PLUS, MOINS, MULT, DIV, PAROUV, PARFER, ENTIER;
```

# Accent (Section déclarations des tokens)

- Section contenant la d´eclaration des tokens associés aux terminaux de la grammaire
- Ces tokens sont introduits par le mot clé `%token` suivi par une suite de noms séparés par des virgules
- Exemple transparent 30 : Liste des tokens définis transparent 25

# Accent (Section règles de traduction - Exemple)

```
/* Grammaire */
Root : Exp {fin_analyse();}
;

Exp : Exp PLUS Term
      | Exp MOINS Term
      | Term
;

Term : Term MULT Fact
       | Term DIV Fact
       | Fact
;

Fact : ENTIER
       | PAROUV Exp PARFER
       | MOINS Exp
;
```

# Accent (Section règles de traductions)

- Section d´ecrivant les différentes productions de la grammaire
- Les productions de même non-terminal en partie gauche sont séparées par le symbole | et le symbole ; sépare les déclarations de productions
- Des actions sémantiques peuvent être insérées à n'importe quel emplacement dans les parties droites de productions

$$
\begin{array}{lcl}
\langle \text{partie gauche} \rangle & : & \langle \text{partie droite} \rangle_1 \quad \{ \langle \text{action sémantique} \rangle \}_1 \\
 & & \quad \ldots \\
 & | & \langle \text{partie droite} \rangle_n \quad \{ \langle \text{action sémantique} \rangle \}_n \\
 & ; &
\end{array}
$$

- Exemple transparent 32 : La production `Root` $\longrightarrow$ `Exp` a été ajouté pour avoir une production unique pour le starter de la grammaire (`Root`)
  En fin d'analyse, l'action `fin_analyse` est appelée (transparent 28)

# Accent (Structure du programme)

Le fichier `auxil.c` contient les fonctions C suivantes :

- La fonction principale `main()` qui appelle l'analyseur syntaxique `yyparse()` contenu dans le fichier entire.c

- La fonction d'affichage des erreurs `yyerror()`
  - Cette fonction sert à afficher les erreurs syntaxiques détectés par l'analyseur syntaxique
  - Elle peut être utilisée par l'analyseur lexical pour afficher les erreurs lexicales (transparent 26)

- La fonction `yywrap()` qui est appelé par la fonction `yylex()` lorsqu'elle atteint la fin du fichier d'entrée. Cette fonction retourne :
  - 1 si l'analyse est terminée
  - 0 si l'analyse continue sur un nouveau fichier à analyser (fichier se trouvant sur la ligne de commande)

```
/* Fichier auxil.c */
#include <stdio.h>
#include <stdlib.h>
/* Programme principal */
int main(void) {
    yyparse(); /* Appel de l'analyseur syntaxique */
    return 0;
}
/* Fonction d'affichage des erreurs */
int yyerror(char *msg){
    extern long yypos; /* Numero de la ligne courante du fichier d
entree (voir fichier lex) */
    printf("Ligne %li: %s\n", yypos, msg);
    exit(1);
}
/* Retourne 1 si pas d autre fichier a analyser */
int yywrap() {
    return 1;
}
```