

lex et yacc

lex et yacc

lex : générateur d'analyseur lexical.

- Prend en entrée la définition des unités lexicales.
- Produit un automate fini déterministe minimal permettant de reconnaître les unités lexicales.
- L'automate est produit sous la forme d'un programme C.
- Il existe plusieurs versions de **lex**, nous utiliserons ici **flex**.

yacc : générateur d'analyseur syntaxique.

- Prend en entrée la définition d'un schéma de traduction (grammaire + actions sémantiques).
- Produit un analyseur syntaxique pour le schéma de traduction.
- L'analyseur est produit sous la forme d'un programme C.
- Il existe plusieurs versions de **yacc**, nous utiliserons ici **bison**.

Grammaire en yacc

Expressions arithmétiques :

```
/* fichier calc.y */
%token NOMBRE /* liste des terminaux */
%%
expression: expression '+' terme
           | expression '-' terme
           | terme
           ;
terme: terme '*' facteur
     | terme '/' facteur
     | facteur
     ;
facteur: '(' expression ')'
       | '-' facteur
       | NOMBRE
       ;
%%
int yyerror(void)
{ fprintf(stderr, "erreur de syntaxe\n"); return 1;}
```

Analyseur lexical en lex

```
/* fichier calc.l */
%{
/* fichier dans lequel est defini la macro constante NOMBRE */
#include "calc.tab.h"
%}

%%
[0-9]+ {return NOMBRE;}
[ \t] ; /* ignore les blancs et tabulations */
\n      return 0;
.        return yytext[0];
%%
```

La fonction principale

```
/*fichier calc.c */  
int main(void)  
{  
    yyparse();  
}
```

Compilation

- `$ bison -d calc.y`
produit les fichiers :
 - `calc.tab.c` qui contient le code en c de l'analyseur
 - et `calc.tab.h` qui contient la définition des codes des unités lexicales, afin qu'elles puissent être partagées par l'analyseur syntaxique et l'analyseur lexical.
- `$ flex calc.l`
produit le fichier :
 - `lex.yy.c` qui contient le code en c de l'analyseur lexical.
- `$ gcc -o calc calc.c calc.tab.c lex.yy.c`
produit l'exécutable :
 - `calc` qui permet d'analyser des expressions arithmétiques.

Exécution

- Analyse d'une expression correcte

```
$ ./calc
```

```
1 + 2
```

```
$
```

- Analyse d'une expression incorrecte

```
$ ./calc
```

```
1 + A
```

```
erreur de syntaxe
```

```
$
```

Communication

Entre les analyseurs syntaxique et lexical :

- Le fichier `lex.yy.c` définit la fonction `yylex()` qui analyse le contenu du flux `yyin` jusqu'à détecter une unité lexicale. Elle renvoie alors le code de l'unité reconnue.
- L'analyseur syntaxique (`yyparse()`) appelle la fonction `yylex()` pour connaître la prochaine unité lexicale à traiter.
- Le fichier `calc.tab.h` associe un entier à tout symbole terminal.
`#define NOMBRE 258`

Ajout d'actions aux règles

```
%token NOMBRE
%%
calcul: expression {printf("%d\n",$1);}
      ;
expression: expression '+' terme {$$ = $1 + $3;}
          | expression '-' terme {$$ = $1 - $3;}
          | terme                 {$$ = $1;}
      ;
terme: terme '*' facteur {$$ = $1 * $3;}
     | terme '/' facteur {$$ = $1 / $3;}
     | facteur           {$$ = $1;}
     ;
facteur: '(' expression ')' {$$ = $2;}
       | '-' facteur       {$$ = -$2;}
       | NOMBRE            {$$ = $1;}
       ;
%%
```

L'analyseur lexical associe des valeurs aux unités lexicales.

```
%{  
#include "calc.tab.h"  
%}  
  
%%  
[0-9]+ {yyval=atoi(yytext);return NOMBRE;}  
[ \t] ; /* ignore les blancs et tabulations */  
\n      return 0;  
.      return yytext[0];  
%%
```

Exécution

- exemple 1

```
$ ./calc  
1 + 6 * 4  
25  
$
```

- exemple 2

```
$ ./calc  
1 / 2  
0  
$
```

Par défaut terminaux et non terminaux sont associés à un attribut à valeur entière.

Typage des symboles

```
%union {double dval; int ival;}
%token <dval> NOMBRE
%type <dval> expression terme facteur
%%
calcul: expression {printf("%f\n",$1);};
expression: expression '+' terme {$$ = $1 + $3;}
           | expression '-' terme {$$ = $1 - $3;}
           | terme                {$$ = $1;}
           ;
terme: terme '*' facteur {$$ = $1 * $3;}
      | terme '/' facteur {$$ = $1 / $3;}
      | facteur           {$$ = $1;}
      ;
facteur: '(' expression ')' {$$ = $2;}
        | '-' facteur       {$$ = -$2;}
        | NOMBRE            {$$ = $1;}
        ;
%%
```

Analyse lexicale des nombres

```
%{  
#include "calc.tab.h"  
%}  
  
%%  
[0-9]+|[0-9]*\.[0-9]+ {yyval.dval=atof(yytext);return NOMBRE;}  
[ \t] ; /* ignore les blancs et tabulations */  
\n      return 0;  
.  
%%
```

Règles de priorité

```
%union {double dval; int ival;}
%token <dval> NOMBRE
%type <dval> expression
%left '+' '-'
%left '*' '/'
%nonassoc MOINSU
%%
calcul: expression {printf("%f\n",$1);};
expression: expression '+' expression    {$$ = $1 + $3;}
           | expression '-' expression    {$$ = $1 - $3;}
           | expression '*' expression    {$$ = $1 * $3;}
           | expression '/' expression    {$$ = $1 / $3;}
           | '(' expression ')'           {$$ = $2;}
           | '-' expression %prec MOINSU {$$ = -$2;}
           | NOMBRE                       {$$ = $1;}
;
%%
```

Règles de priorité

```
%left '+' '-'  
%left '*' '/'  
%nonassoc MOINSU
```

- Chacune de ces déclarations définit un niveau de priorité.
- Elles indiquent que + et - sont associative à gauche ($1 - 3 - 4 = (1 - 3) - 4$) et qu'ils sont moins prioritaires que * et /.
- MOINSU est une pseudo unité lexicale qui permet d'associer un niveau de priorité au - unaire :

```
|      '-' expression %prec MOINSU {$$ = -$2;}
```

L'indispensable Makefile

```
LEX = flex
```

```
YACC = bison -d
```

```
CC = gcc
```

```
calc: calc.c calc.tab.c lex.yy.c
```

```
    $(CC) -o calc calc.tab.c lex.yy.c calc.c
```

```
calc.tab.c: calc.y
```

```
    $(YACC) calc.y
```

```
lex.yy.c: calc.l
```

```
    $(LEX) calc.l
```


lex

lex est un langage de spécification d'analyseurs lexicaux.

- Un programme en lex définit un ensemble de schémas qui sont appliqués à un flux textuel.
- Chaque schéma est représenté sous la forme d'une expression régulière.
- Lorsque l'application d'un schéma réussit, les actions qui lui sont associées sont exécutées.
- L'exécutable lex permet de transformer un programme en lex en un programme en C qui définit la fonction `yylex(void)` qui constitue l'analyseur lexical.

Syntaxe des expressions régulières de lex

Concaténation

ab

| Disjonction

$a \mid b$

* Etoile de Kleene

$(a \mid b)^*$

{ n } Répétition n fois

$(a \mid b)\{3\}$

? Optionnalité (0 ou 1 fois)

$a?$

+ Une fois ou plus

a^+

() Groupement d'expressions régulières

Modification de la priorité dans une expression

$a(b \mid c)$

Syntaxe des expressions régulières de lex

- .
 - []
 - [abc]
 - [a-z]
 - [a-zA-Z]
 - [^]
 - [^abc]
 - \
- Tout caractère excepté le retour chariot \n
- Ensemble de caractères.
- définit l'ensemble {a,b,c}
- définit l'ensemble {a,b,c, ..., z}
- définit l'ensemble {a,b,c, ..., z,A,B,C, ..., Z}
- Complémentaire d'un ensemble de caractères.
- définit le complémentaire de l'ensemble {a,b,c}
- Caractère d'échappement

Syntaxe des expressions régulières de lex

- " . . . " interprète tout ce qui se trouve entre les guillemets de manière littérale
 - "a*\$"
- ^ Début de ligne
 - ^abc
- \$ Fin de ligne
 - abc\$
- / Reconnaît un motif appartenant à l'expression régulière de gauche s'il est suivi par un motif reconnu par l'expression régulière de droite
 - 0/1 reconnaît un 0 s'il est suivi par un 1

Structure d'un fichier lex

```
%{  
    Partie 1 : déclarations pour le compilateur C  
}%  
    Partie 2 : définitions régulières  
%%  
    Partie 3 : règles  
%%  
    Partie 4 : fonctions C supplémentaires
```

Structure d'un fichier lex

- **La partie 1** se compose de déclarations qui seront simplement recopiées au début du fichier produit.
On y trouve souvent une directive `#include` qui produit l'inclusion du fichier d'en tête contenant les définitions des codes des unités lexicales.
Cette partie et les symboles `%{` et `%}` qui l'encadrent peuvent être omis.
- **La partie 4** se compose de fonctions C qui seront simplement recopiées à la fin du fichier produit.
Cette partie peut être absente également (les symboles `%%` qui la séparent de la troisième partie peuvent alors être omis).

Définitions régulières

- Les définitions régulières sont de la forme

identificateur expressionRégulière

où *identificateur* est écrit au début de la ligne et séparé de *expressionRégulière* par des blancs.

lettre [A-Za-z]

chiffre [0-9]

- Les identificateurs ainsi définis peuvent être utilisés dans les règles et dans les définitions suivantes ; il faut alors les encadrer par des accolades.

lettre [A-Za-z]

chiffre [0-9]

alphanum {lettre}|{chiffre}

Règles

Les règles sont de la forme

expressionRégulière { *action* }

où

- *expressionRégulière* est écrit au début de la ligne
- *action* est un morceau de code C, qui sera recopié tel quel, au bon endroit, dans la fonction `yylex`.

```
if                {return SI;}  
then              {return ALORS;}  
{lettre}{alphanum}* {return IDENTIF;}
```

A la fin de la reconnaissance d'une unité lexicale, la chaîne reconnue est la valeur de la variable `yytext` de type `char *`.

La variable `yylen` indique la longueur de l'unité lexicale contenue dans `yytext`.

```
(+|-)?[0-9]+ {yyval = atoi(yytext); return NOMBRE;}
```


Ordre d'application des règles

- lex essaye les règles dans leur ordre d'apparition dans la fichier.
- La règle reconnaissant la séquence la plus longue est appliquée.

```
a {printf("1");}
```

```
aa {printf("2");}
```

l'entrée aaa provoquera la sortie 21

- Lorsque deux règles reconnaissent la séquence la plus longue, la première est appliquée.

```
aa {printf("1");}
```

```
aa {printf("2");}
```

l'entrée aa provoquera la sortie 1

- Les caractères qui ne sont reconnus par aucune règle sont copiés sur la sortie standard.

```
aa {printf("1");}
```

l'entrée aaa provoquera la sortie 1a

La fonction `yywrap(void)`

- Lorsque `yylex()` atteint la fin du flux `yyin`, il appelle la fonction `yywrap` qui renvoie 0 ou 1.
- Si la valeur renvoyée est 1, le programme s'arrête.
- Si la valeur renvoyée est 0, l'analyseur lexical suppose que `yywrap` a ouvert un nouveau fichier en lecture et le traitement de `yyin` continue.
- Version minimaliste de `yywrap`

```
int yywrap(void)
{
    return 1;
}
```

Les états

- Certaines règles peuvent être associées à des **états**.
- Elles ne s'appliquent que lorsque l'analyseur se trouve dans l'état spécifié.

```
%s COMM                      /* on définit l'état COMM */
%%
"/*" BEGIN COMM;             /* on rentre dans l'état COMM */
<COMM>. ;                    /* on ignore tout ce qui se trouve*/
<COMM>\n ;                   /* dans les commentaires */
<COMM>"*/" BEGIN INITIAL /* on revient à l'état standard */
```

- Les règles qui ne sont pas associées à un état s'appliquent dans tous les états.

Les états exclusifs

- Dans l'exemple précédent, toutes les règles qui ne sont pas liées à l'état COMM peuvent aussi s'appliquer dans l'état COMM
- Pour éviter ce comportement, on utilise des états **exclusifs** : une règle qui n'est liée à aucun état ne **s'applique pas** dans un état exclusif.

```
%x COMM                                /* l'état COMM est exclusif */
%%
"/" BEGIN COMM;                        /* on rentre dans l'état COMM */
<COMM>. ;                               /* on ignore tout ce qui se trouve */
<COMM>\n ;                             /* dans les commentaires */
<COMM>"/" BEGIN INITIAL /* on revient à l'état standard */
```

Un exemple

```
%{  
#include "syntaxe.tab.h"  
union {int ival; char *sval; double fval;} yylval;  
%}  
%%  
si          {return SI;}  
alors       {return ALORS;}  
sinon       {return SINON;}  
\[A-Za-z]+\ {yylval.sval=strdup(yytext);return ID_VAR;}  
[0-9]+\     {yylval.ival=atoi(yytext);return NOMBRE;}  
[A-Za-z]+\ {return ID_FCT;}  
.          {return yytext[0];}  
%%
```

Structure d'un fichier yacc

```
%{  
    Partie 1 : déclarations pour le compilateur C  
}%  
    Partie 2 : déclarations pour \yacc  
%%  
    Partie 3 : schémas de traduction  
              (productions + actions sémantiques)  
%%  
    Partie 4 : fonctions C supplémentaires
```

Les parties 1 et 4 sont simplement recopiées dans le fichier produit, respectivement à la fin et au début de ce dernier. Chacune des deux parties peut être absente.

Déclarations pour yacc

- Une directive union décrivant les types des valeurs associées aux symboles terminaux et non terminaux.
- Les déclarations des symboles non terminaux, enrichis de leurs types.
- Les symboles terminaux (à l'exception des symboles composés d'un caractère) enrichis de leurs types.
- Les priorités des opérateurs ainsi que leur associativité.

```
%union {char *sval; int ival; double dval;}  
%token <sval> VAR  
%token <dval> NOMBRE  
%type  <dval> expression terme facteur  
%left '+' '-'  
%left '*' '/'  
%nonassoc MOINSU
```

Schémas de traduction

Un schéma de traduction est un ensemble de productions ayant la même partie gauche, chacune associée à une action sémantique.

```
expression: expression '+' expression    {$$ = $1 + $3;}
           | expression '-' expression    {$$ = $1 - $3;}
           | expression '*' expression    {$$ = $1 * $3;}
           | expression '/' expression    {$$ = $1 / $3;}
           | '(' expression ')'           {$$ = $2;}
           | '-' expression %prec MOINSU {$$ = -$2;}
           | NOMBRE                       {$$ = $1;}
           ;
```

Le symbole qui constitue la partie gauche de la première production est l'axiome de la grammaire.

La fonction `int yyparse(void)`

- L'analyseur syntaxique se présente comme une fonction `int yyparse(void)`, qui retourne 0 lorsque la chaîne d'entrée est acceptée, une valeur non nulle dans le cas contraire.
- Pour obtenir un analyseur syntaxique autonome, il suffit d'ajouter en partie 3 du fichier `yacc`

```
int main(void){  
    if(yyparse() == 0)  
        printf("Analyse réussie\n");  
}
```

La fonction `yyerror(char *message)`

- Lorsque la fonction `yparse()` échoue, elle appelle la fonction `yyerror` qu'il faut définir.
- version rudimentaire de `yyerror`

```
int yyerror(void)
{
    fprintf(stderr, "erreur de syntaxe\n");
    return 1;
}
```

Actions sémantiques et valeurs des attributs

Une action sémantique est une séquence d'instructions C écrite, entre accolades à droite d'une production.

- Cette séquence est copiée par yacc dans l'analyseur produit, de telle manière qu'elle sera exécutée pendant l'analyse lorsque la production correspondante aura été reconnue.
- Les variables \$1, \$2, ... désignent les valeurs des attributs des symboles constituant le premier, second ... symbole de la partie droite de la production concernée.
- \$\$ désigne la valeur de l'attribut du symbole qui est la partie gauche de cette production.
- L'action sémantique { \$\$ = \$1; } est implicite, il n'est pas nécessaire de l'écrire.

Analyse par décalage-réduction

L'analyseur produit par yacc est un analyseur ascendant, par décalage-réduction.

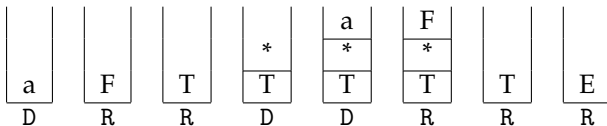
- Principe général : on maintient une pile de symboles dans laquelle sont empilés les terminaux au fur et à mesure qu'ils sont lus.
- L'opération qui consiste à empiler un terminal est appelée **décalage**.
- lorsque les k symboles au sommet de la pile constituent la partie droite d'une production, ils peuvent être dépilés et remplacés par la partie gauche de la production.
- Cette opération s'appelle **réduction**.
- Lorsque la pile ne comporte que l'axiome et que tous les symboles de la chaîne d'entrée ont été lus, l'analyse a réussi.

Exemple

■ Grammaire :
$$\left\{ \begin{array}{lcl} E & \rightarrow & E + T \\ E & \rightarrow & T \\ T & \rightarrow & T * F \\ T & \rightarrow & F \\ F & \rightarrow & a \end{array} \right.$$

■ Chaîne à analyser : $a * a$

■ Simulation de l'analyse par décalage-réduction :



Problème majeur

- Si les symboles au sommet de la pile constituent la partie droite de deux productions distinctes, laquelle utiliser pour effectuer la réduction ?
S'il n'est pas possible de décider, on parle d'un **conflit réduction / réduction**.
- Lorsque les symboles au sommet de la pile constituent la partie droite d'une ou plusieurs productions, faut-il réduire tout de suite, ou bien continuer à décaler, afin de permettre ultérieurement une réduction plus juste ?
S'il n'est pas possible de décider, on parle d'un **conflit décalage / réduction**.

Conflit de réduction / réduction

$$1: S \rightarrow X$$

$$2: S \rightarrow Y$$

$$3: X \rightarrow a$$

$$4: Y \rightarrow a$$

On ne sait pas s'il faut réduire par 3 ou 4.

Conflit de réduction / réduction

$$\begin{array}{llll} 1 : & S & \rightarrow & X \ b \\ 2 : & S & \rightarrow & Y \ c \\ 3 : & X & \rightarrow & a \\ 4 : & Y & \rightarrow & a \end{array}$$

Si on sait que a est suivi de b (ou de c), il n'y a pas conflit !

Conflit de décalage / réduction

$$\begin{array}{lclcl} 1: & S & \rightarrow & X & \\ 2: & S & \rightarrow & Y & r \\ 3: & X & \rightarrow & a & r \\ 4: & Y & \rightarrow & a & \end{array}$$

Après un décalage d'un a , on ne sait pas s'il faut décaler ou réduire.

Grammaires $LR(k)$

- Une grammaire est $LR(k)$ s'il est possible d'effectuer une analyse par décalage-réduction sans conflit en s'autorisant à lire les k symboles suivant le symbole courant.
- La grammaire suivante n'est pas $LR(1)$ mais elle est $LR(2)$:

$$\begin{array}{lcl} 1: & S & \rightarrow X \ b \ c \\ 2: & S & \rightarrow Y \ b \ d \\ 3: & X & \rightarrow a \\ 4: & Y & \rightarrow a \end{array}$$

Que faire en cas de conflit ?

- Vérifier que la grammaire n'est pas ambiguë.
- Une grammaire non ambiguë peut ne pas être $LR(k)$; il faut alors réécrire la grammaire.

Construire une table $LR(0)$

Soit la grammaire suivante :

$$\begin{array}{lcl} E & \rightarrow & E * B \mid E + B \mid B \\ B & \rightarrow & 0 \mid 1 \end{array}$$

Construire une table $LR(0)$

Soit la grammaire suivante :

$$\begin{aligned} E &\rightarrow E * B \mid E + B \mid B \\ B &\rightarrow 0 \mid 1 \end{aligned}$$

On ajoute un non-terminal de départ $S \rightarrow E$, puis on introduit un nouveau symbole \bullet et on le place partout :

$$\begin{aligned} S &\rightarrow \bullet E \mid E \bullet \\ E &\rightarrow \bullet E * B \mid E \bullet * B \mid E * \bullet B \mid E * B \bullet \\ &\quad \bullet E + B \mid E \bullet + B \mid E + \bullet B \mid E + B \bullet \\ &\quad \bullet B \mid B \bullet \\ B &\rightarrow \bullet 0 \mid \bullet 1 \mid 0 \bullet \mid 1 \bullet \end{aligned}$$

On appelle cet ensemble de règles des **items**.

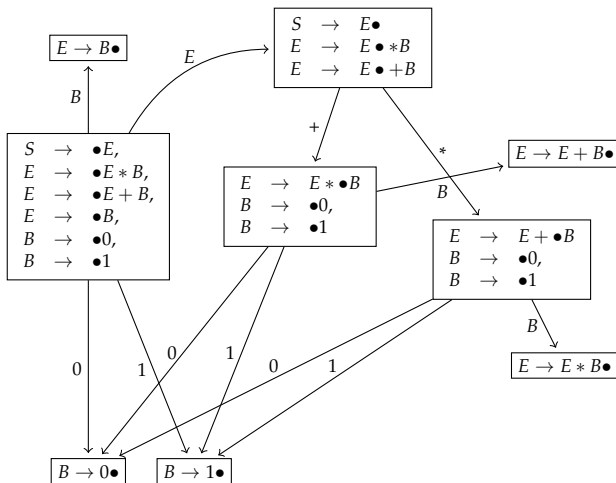
Construire une table $LR(0)$

La fonction $FERMETURE(I)$ prend une liste d'items et pour tout item $A \rightarrow \alpha \bullet B\gamma$ dans $FERMETURE(I)$, ajoute $B \rightarrow \bullet\beta$ pour toute règle $B \rightarrow \beta$.

$$\begin{aligned} FERMETURE(S \rightarrow \bullet E) = \{ & S \rightarrow \bullet E, \\ & E \rightarrow \bullet E * B, E \rightarrow \bullet E + B, E \rightarrow \bullet B, \\ & B \rightarrow \bullet 0, B \rightarrow \bullet 1 \} \end{aligned}$$

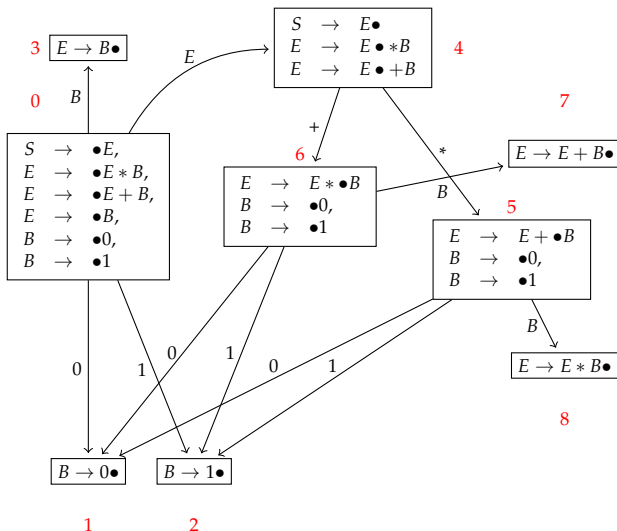
Automate

Les fermetures sont les états.

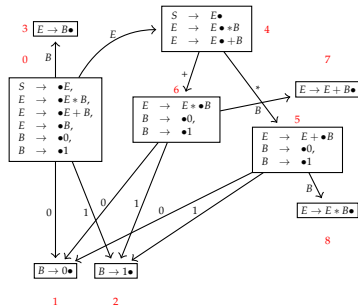


Automate

Les fermetures sont les états.

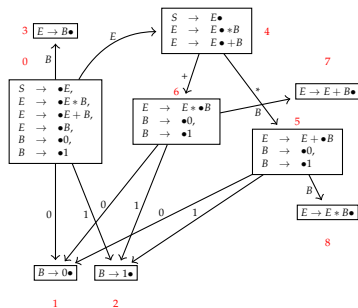


Construire une table $LR(0)$, suite



Construire une table LR(0), suite

d = décalage, r = réduction, acc = acceptation



état	action					goto	
	0	1	*	+	\$	E	B
0	d1	d2				4	3
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	r3	r3	r3	r3	r3		
4			d5	d6	acc		
5	d1	d2					8
6	d1	d2					7
7	r2	r2	r2	r2	r2		
8	r1	r1	r1	r1	r1		

1 : $E \rightarrow E * B$ 4 : $B \rightarrow 0$
 2 : $E \rightarrow E + B$ 5 : $B \rightarrow 1$
 3 : $E \rightarrow B$

Utiliser la table

On lit le mot :

1 + 0\$

état	action					goto	
	0	1	*	+	\$	E	B
0	d1	d2				4	3
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	r3	r3	r3	r3	r3		
4			d5	d6	acc		
5	d1	d2					8
6	d1	d2					7
7	r2	r2	r2	r2	r2		
8	r1	r1	r1	r1	r1		

0
⊥

1 : $E \rightarrow E * B$ 4 : $B \rightarrow 0$
 2 : $E \rightarrow E + B$ 5 : $B \rightarrow 1$
 3 : $E \rightarrow B$

Utiliser la table

On lit le mot :

1 + 0\$

état	action					goto	
	0	1	*	+	\$	E	B
0	d1	d2				4	3
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	r3	r3	r3	r3	r3		
4			d5	d6	acc		
5	d1	d2					8
6	d1	d2					7
7	r2	r2	r2	r2	r2		
8	r1	r1	r1	r1	r1		

0
⊥

1 : $E \rightarrow E * B$ 4 : $B \rightarrow 0$
 2 : $E \rightarrow E + B$ 5 : $B \rightarrow 1$
 3 : $E \rightarrow B$

Utiliser la table

On lit le mot :

1 + 0\$

état	action					goto	
	0	1	*	+	\$	E	B
0	d1	d2				4	3
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	r3	r3	r3	r3	r3		
4			d5	d6	acc		
5	d1	d2					8
6	d1	d2					7
7	r2	r2	r2	r2	r2		
8	r1	r1	r1	r1	r1		

2
0
⊥

1 : $E \rightarrow E * B$ 4 : $B \rightarrow 0$
 2 : $E \rightarrow E + B$ 5 : $B \rightarrow 1$
 3 : $E \rightarrow B$

Utiliser la table

On lit le mot :

1 + 0\$

état	action					goto	
	0	1	*	+	\$	E	B
0	d1	d2				4	3
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	r3	r3	r3	r3	r3		
4			d5	d6	acc		
5	d1	d2					8
6	d1	d2					7
7	r2	r2	r2	r2	r2		
8	r1	r1	r1	r1	r1		

0
⊥

1 : $E \rightarrow E * B$ 4 : $B \rightarrow 0$
 2 : $E \rightarrow E + B$ 5 : $B \rightarrow 1$
 3 : $E \rightarrow B$

Utiliser la table

On lit le mot :

1 + 0\$

état	action					goto	
	0	1	*	+	\$	E	B
0	d1	d2				4	3
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	r3	r3	r3	r3	r3		
4			d5	d6	acc		
5	d1	d2					8
6	d1	d2					7
7	r2	r2	r2	r2	r2		
8	r1	r1	r1	r1	r1		

3
0
⊥

1 : $E \rightarrow E * B$ 4 : $B \rightarrow 0$
 2 : $E \rightarrow E + B$ 5 : $B \rightarrow 1$
 3 : $E \rightarrow B$

Utiliser la table

On lit le mot :

1 + 0\$

état	action					goto	
	0	1	*	+	\$	E	B
0	d1	d2				4	3
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	r3	r3	r3	r3	r3		
4			d5	d6	acc		
5	d1	d2					8
6	d1	d2					7
7	r2	r2	r2	r2	r2		
8	r1	r1	r1	r1	r1		

0
⊥

1 : $E \rightarrow E * B$ 4 : $B \rightarrow 0$
 2 : $E \rightarrow E + B$ 5 : $B \rightarrow 1$
 3 : $E \rightarrow B$

Utiliser la table

On lit le mot :

1 + 0\$

état	action					goto	
	0	1	*	+	\$	E	B
0	d1	d2				4	3
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	r3	r3	r3	r3	r3		
4			d5	d6	acc		
5	d1	d2					8
6	d1	d2					7
7	r2	r2	r2	r2	r2		
8	r1	r1	r1	r1	r1		

4
0
⊥

1 : $E \rightarrow E * B$ 4 : $B \rightarrow 0$
 2 : $E \rightarrow E + B$ 5 : $B \rightarrow 1$
 3 : $E \rightarrow B$

Utiliser la table

On lit le mot :

1 + 0\$

état	action					goto	
	0	1	*	+	\$	E	B
0	d1	d2				4	3
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	r3	r3	r3	r3	r3		
4			d5	d6	acc		
5	d1	d2				8	
6	d1	d2				7	
7	r2	r2	r2	r2	r2		
8	r1	r1	r1	r1	r1		

6
4
0
⊥

1 : $E \rightarrow E * B$ 4 : $B \rightarrow 0$
 2 : $E \rightarrow E + B$ 5 : $B \rightarrow 1$
 3 : $E \rightarrow B$

Utiliser la table

On lit le mot :

1 + 0\$

état	action					goto	
	0	1	*	+	\$	E	B
0	d1	d2				4	3
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	r3	r3	r3	r3	r3		
4			d5	d6	acc		
5	d1	d2					8
6	d1	d2					7
7	r2	r2	r2	r2	r2		
8	r1	r1	r1	r1	r1		

1
6
4
0
⊥

1 : $E \rightarrow E * B$ 4 : $B \rightarrow 0$
 2 : $E \rightarrow E + B$ 5 : $B \rightarrow 1$
 3 : $E \rightarrow B$

Utiliser la table

On lit le mot :

1 + 0\$

état	action					goto	
	0	1	*	+	\$	E	B
0	d1	d2				4	3
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	r3	r3	r3	r3	r3		
4			d5	d6	acc		
5	d1	d2				8	
6	d1	d2				7	
7	r2	r2	r2	r2	r2		
8	r1	r1	r1	r1	r1		

6
4
0
⊥

$1: E \rightarrow E * B$ $4: B \rightarrow 0$
 $2: E \rightarrow E + B$ $5: B \rightarrow 1$
 $3: E \rightarrow B$

Utiliser la table

On lit le mot :

1 + 0\$

état	action					goto	
	0	1	*	+	\$	E	B
0	d1	d2				4	3
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	r3	r3	r3	r3	r3		
4			d5	d6	acc		
5	d1	d2					8
6	d1	d2					7
7	r2	r2	r2	r2	r2		
8	r1	r1	r1	r1	r1		

7
6
4
0
⊥

1 : $E \rightarrow E * B$ 4 : $B \rightarrow 0$
 2 : $E \rightarrow E + B$ 5 : $B \rightarrow 1$
 3 : $E \rightarrow B$

Utiliser la table

On lit le mot :

1 + 0\$

état	action					goto	
	0	1	*	+	\$	E	B
0	d1	d2				4	3
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	r3	r3	r3	r3	r3		
4			d5	d6	acc		
5	d1	d2					8
6	d1	d2					7
7	r2	r2	r2	r2	r2		
8	r1	r1	r1	r1	r1		

0
⊥

1 : $E \rightarrow E * B$ 4 : $B \rightarrow 0$
 2 : $E \rightarrow E + B$ 5 : $B \rightarrow 1$
 3 : $E \rightarrow B$

Utiliser la table

On lit le mot :

1 + 0\$

état	action					goto	
	0	1	*	+	\$	E	B
0	d1	d2				4	3
1	r4	r4	r4	r4	r4		
2	r5	r5	r5	r5	r5		
3	r3	r3	r3	r3	r3		
4			d5	d6	acc		
5	d1	d2					8
6	d1	d2					7
7	r2	r2	r2	r2	r2		
8	r1	r1	r1	r1	r1		

4
0
⊥

1 : $E \rightarrow E * B$ 4 : $B \rightarrow 0$
 2 : $E \rightarrow E + B$ 5 : $B \rightarrow 1$
 3 : $E \rightarrow B$

Passer à $LR(1)$

- Même principe, plus d'états.
- Items de la forme

$$S \rightarrow \alpha \bullet \gamma, \quad a$$

où

$$a \text{ est dans } \begin{cases} \text{PREMIER}(\gamma) \\ \text{SUIVANT}(S) \text{ si } \gamma = \varepsilon. \end{cases}$$

Sources

- John Levine, Tony Mason, Doug Brown, *lex & yacc*, O'Reilly & Associates, Inc.
- Henri Garreta, *Polycopié du cours de compilation*.
- Andrew Appel, *Modern compiler implementation in C*. Cambridge University Press, 1998