Trad 1

C3

chapitre 3 : Analyse Syntaxique

① Rôle

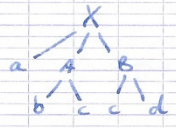
- vérifier que la phrase en entrée vérifie la syntaxe du langage définie par une grammaire
  - ↳ reconnaître les unités syntaxiques décrites par la grammaire
    - déclarations
    - instructions
    - variables, expressions
    - fonctions, procédures.

- Signaler les erreurs syntaxiques
  - ↳ on attend aussi que l'analyseur :
    - annonce clairement et localise l'erreur
    - corrige l'erreur
    - continue l'analyse.
    - ne doit pas ralentir l'analyse du programme correcte.

- suite unités lexicales → Analyseur Syntaxique → erreur
  Analyseur Syntaxique → représentation d'arbre abstrait (structure de données).

② Analyse Syntaxique Descendante (LL(k))

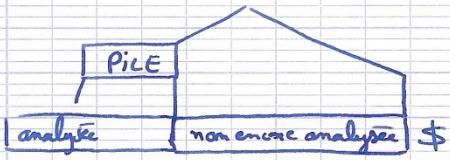
cf Maths Discrètes + TD



$X \to aAB$
dérivations à faire

ε, le mot vide n'est pas un token !!

③ Analyse Syntaxique Ascendante



Principe :

- Part du mot de la phrase à analyser.
- On remplace itérativement des fragments du mot courant qui correspondent à des membres droites d'une production par le membre gauche de cette production.
- L'analyse réussit si le mot courant final est l'axiome

Analyse Ascendant Prédicatif LR(k)

Left to right scanning
Right most derivation : on construit une dérivation droite en partant du mot à analyser.
k unité lex d'avance pour faire la dérivation.

famille d'algos.

LR(0) SLR(1) LR(1) LALR(1)
↳ basée sur des opérations de :
- décalage : on lit des terminaux dans le mot d'entrée (shift) ;
- réduction : on remplace une partie droite de production par sa partie gauche (reduce).

TRad 1

C4

Analyse LR : exemple :

On veut analyser le mot $w = a_1 a_2 \dots a_m$
où à chaque instant on a :
- un suffixe w' du mot d'entrée
- une pile contenant terminaux et non terminaux

L'analyse réussit si w a été entièrement lu et si la pile contient l'axiome.

$S' \to S$
$S \to Ac$ mot : dadbc \$
$A \to AaAb \mid d$

- décalage : le 1er terminal du mot lu est placé dans la pile.

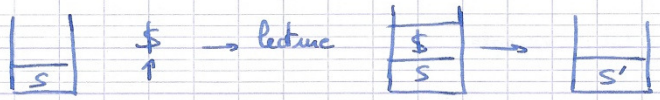
- réduction : les "n" premiers symboles de la pile forment un membre droit de prod
  ↳ on les dépile → on empile la partie gauche
- on ne peut plus réduire ⇒ on va lire (shift)

dbc \$
- pas de réduction possible ⇒ décalage (shift)

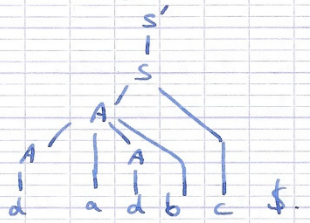
bc \$ ⇒ réduction par $A \to d$
↳ on ne peut plus réduire ⇒ shift

c \$ ⇒ réduction par $A \to AaAb$ ⇒ on shift

\$ réduction par $S \to Ac$



donc le mot est syntaxiquement correct.



Comment, par algo, décider à chaque étape si on réduit ou si on décale ?

⇒ construire un automate déterministe LR(0) qui dira ce qu'il faut faire en fonction :

- du contenu de la pile ;
- de l'unité lex d'entrée.

I) Automate LR(0).

① Définition

item : un item d'une grammaire G est une production de G avec un point (.) repérant une position de sa partie droite

[A → BCa] forme 4 items
A → .BCa
B.Ca
BC.a
BCa.

Trad 1

CS

fermeture : Soit I un ensemble d'items de G.

fermeture (I) = ensemble d'items construits à partir de I, tels que
- placer chaque item de I dans fermeture (I).
- si $A \to \alpha . B\beta \in$ fermeture (I) et si $B \to \gamma$
  Alors ajouter $B \to . \gamma$ à fermeture (I)
  (sauf si déjà dedans).

Itérer ce processus jusqu'à ne plus rien avoir à ajouter.

$S' \to .S$ r_0

$S \to .Ac$ r_1

$A \to .AaAb$ r_2 | $.d$ r_3

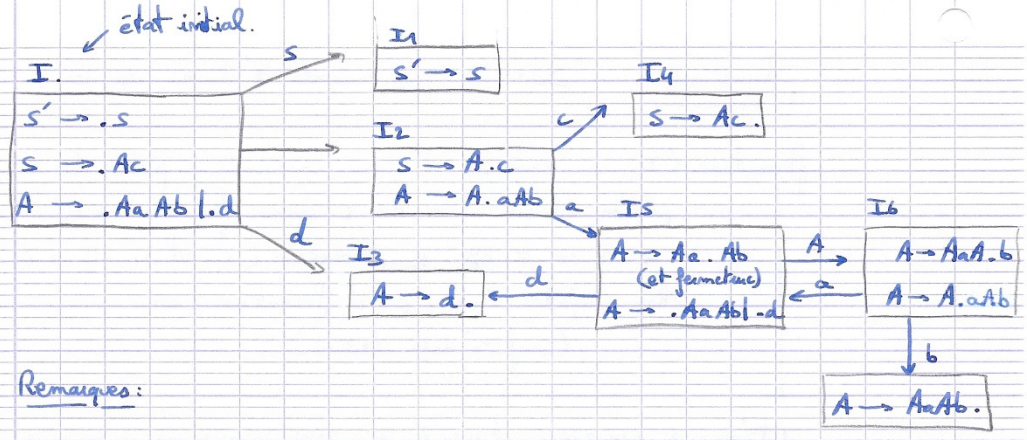
② Etats de l'automate.

Les états de l'automate LR(0) sont des ensembles d'items obtenus par fermeture ($S' \to .S$)

③ Transitions de l'automate.

Soient : $\begin{cases} I : \text{ens d'items} \\ X : \text{symbole de la grammaire } (\in NUT) \end{cases}$

On définit Transition (I, X) comme la fermeture de l'ens des items $A \to \alpha X . \beta$ sachant que $A \to \alpha X \beta \in I$.



Remarques :

⇒ La construction de l'automate est indépendante du mot d'entrée (pas de symbole de prévision)

⇒ Pour tout état i de l'automate LR(0) :

- i ne contient au plus qu'une seule production pointée complète (le "." à la fin). (pas de conflit réduction-réduction).
- si i contient une production pointée complète, alors pour aucun a il n'existe de transition trans(i,a)=i'

Table d'analyse LR(0) : automate + pile ⇒ on sait si on réduit ou lit.

Init : table contient I0. Lors d'une étape de calcul :

- si on "décale" un terminal a, on empile a et l'état i' = trans(i,a).
- si on "réduit" par $X \to \alpha_1 \dots \alpha_m$ on dépile $i_m \alpha_m i_{m-1} \alpha_{m-1} \dots i_1 \alpha_1 i$ et on empile X

On empile donc alternativement [état-symbole].

La table LR(0) est en fait codée en deux parties

table action (4 possibles) :	table transition :
- décaler	
- réduire	C'est la fonction des transitions restreinte aux non-terminaux
- erreur	état + non terminal = état.
- accepter le texte	

Trad 1

C6

Remplissage de la table LR(0)

partie action :

- Si $[S' \to S.]$ appartient à l'état i
  Alors action $(i, \$)$ = " accepter texte ".

- Si $[X \to \beta.]$ appartient à l'état i, $(X \neq S')$
  Alors action (i, a) = " réduire par $X \to \beta$ ", $\forall a \in \{T \cup \$\}$.

- Si $[X \to \alpha . a\beta]$ appartient à l'état i,
  Alors action (i, a) = " décaler en j " (parce que shift au lecture et aller en j)

- " erreur " dans toutes les autres cases.

partie transition :

- Si transition $(i, X) = j$
  Alors mettre j dans la case (i, X) de la table.

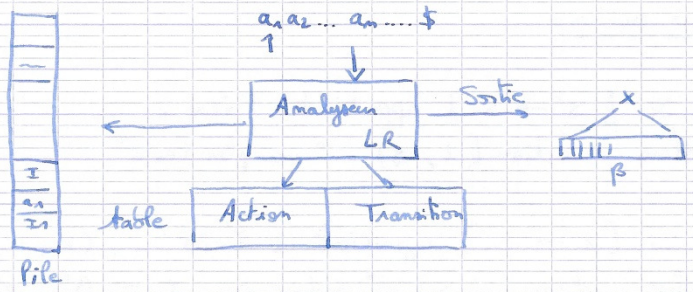
On reprend l'exemple :

je lis et je vais en 3

	a	b	c	d	$	S	A
I_0	erreur	erreur	erreur	d_3	erreur	1	2
I_1	erreur	erreur	erreur	erreur	" accepter le texte "		
I_2	d_5	erreur	d_4	erreur	erreur		
I_3	r_3	r_3	r_3	r_3	r_3		
I_4	r_1	r_1	r_1	r_1	r_1		
I_5	erreur	erreur	erreur	d_3	erreur		6
I_6	d_5	d_7	erreur	erreur	erreur		
I_7	r_2	r_2	r_2	r_2	r_2.		

réduire par la règle r_3 $A \to d$

Fonctionnement :



II) Analyseur SLR (1).

Soit la grammaire G, telle que :

$$G : \begin{cases} E' \to E \\ E \to E+T \mid T \\ T \to T*F \mid F \\ F \to idf. \end{cases}$$

I_0 :
- $E' \to .E.$
- $E \to .E+T \mid .T$
- $T \to .T*F \mid .F$
- $F \to .idf.$

$I_1 =$ AV (I_0, E)
- $E' \to E.$
- $E \to E.+T$

$I_2 =$ AV (I_0, T)
- $E \to T.$
- $T \to T.*F$

$I_3 =$ AV (I_0, F)
- $T \to F.$

$I_4 =$ AV (I_0, idf)
- $F \to idf.$

Trad 1

C7

Dérouler sur i + i * i $

état I_0	Pile		Texte Source
	0	→ décalage en I_4	i + i * i $
	0 i 4	→ réduction par F → id pour tous les terminaux	+ i * i $
		dépiler de la longueur de la partie droite.	
	0 F 3	→ réduire par T → F	+ i * i $
	0 T 2	→ réduire par E → T	
	0 E 1	→ Conflit lecture ? ou réduction ?	+ i * i $

Morralité : ↳ On a levé le conflit lecture / reduct° en regardant si le terminal présent en tête du mot à analyser était dans l'ensemble des suivants du non-terminal en partie gauche de la production utilisable [ici, c'était E' → E et + ∉ Suivant (E')] pour la réduction.

↳ Comme + ∉ Suivant (E'), on a choisit la lecture (ou décalage) ⇒ Seule opération possible

⇒ Ceci est formalisé par les analyseurs SLR(1) basés sur l'automate LR(0).

Remplissage de la table SLR(1) à partir de l'automate LR(0).

Partie transition : idem à la table d'analyse LR(0).

Partie action :

- Si $[S' \to S.] \in I_i$
  Alors Action$(i, \$)$ = " accepte texte".

- Si $[A \to \alpha . a \beta] \in I_i$ et si transition$(i,a) = j$.
  Alors Action(i,a) = décaler en j.

- Si $[A \to \alpha .] \in I_i$, $(A \neq S')$
  Alors action(i,a) = " réduire par $A \to \alpha$." pour tout $a \in$ Suivant(A)

- tout le reste = "erreur"

$\Rightarrow$ Toute grammaire LR(0) est SLR(1)
Toute grammaire SLR(1) est non-ambigüe
Beaucoup de grammaires non ambigües ne sont pas SLR(1).
Toute grammaire LL(1) est non ambigüe.

Trad 1

C8

III] Analyseurs LR(1)

Soit une grammaire telle que :

I_i : $\left[\begin{array}{l} A \to B.aD \\ D \to B. \\ \dots \end{array}\right.$ et I_j : $\left[\begin{array}{l} A \to Ba.D \\ \dots \end{array}\right.$

Conflit ! $\left[\begin{array}{ll} \text{Si on lit } a : & \text{lire ou shift ou décaler : action } (i,a) = \text{"décaler en } j\text{"} \\ \text{on a aussi :} & \text{action } (i,a) = \text{"réduire par } D \to B\text{"} \\ & \text{car } a \in \text{Suivant}(D). \end{array}\right.$

La méthode SLR(1) n'est pas assez puissante pour se rappeler suffisamment de contexte pour décider de l'action à effectuer (lecture ou réduction).

⇒ On va donc ajouter une information supplémentaire dans chacun des items de l'automate. Un item est : $[A \to \alpha.\beta, a]$ où $A \to \alpha\beta \in G$ et a est un terminal ou \$

Analyseur LR(1) longueur = 1.

a est appelé $\left|\begin{array}{l} \text{contexte à droite.} \\ \text{symbole de prévision.} \end{array}\right.$

La prévision n'aura d'effet que sur les items de la forme $[A \to \alpha., a]$ où l'action devient

"réduire par $A \to \alpha$ uniquement lorsque le prochain symbole d'entrée est le contexte a"

III) Analyseurs LR(1) (suite)

① Définitions et constructions | des items LR(1).
de l'automate LR(1).

Un item LR(1) : Une production avec un repère dans un contexte particulier

$S \to .\alpha, \beta$ → contexte

Un état LR(1) : ensemble d'items LR(1).

État initial : fermeture $(S' \to .S, \$)$

ⓐ Calcul des fermetures : fermeture $(I) =$

- placer tous les items de I dans fermeture (I).
- si $[A \to \alpha.B\beta, a] \in I$
  si $[B \to \gamma] \in G$ augmentée
  si b est premier (βa)
  Alors : ajouter $[B \to .\gamma, b]$ dans fermeture (I)
  [sauf si déjà dedans]

itérer l'opération jusqu'à ne plus rien avoir à ajouter.

ⓑ Calcul des transitions :

idem SLR(1)

transition $(I, X) = j$ si j est l'ensemble des items $[A \to \alpha X.\beta, a]$ tel que $[A \to \alpha.X\beta, a] \in I$

Trad 1
C9

② Calcul des actions :

[• Construire l'automate LR(1)]

- si $[A \to \alpha . a\beta, b] \in I_i$
  Alors action (i, a) = "décaler en j" où transition $(I_i, a) = I_j$
  (idem SLR(1))

- si $[S' \to S., \$] \in I_i$
  Alors action $(i, \$)$ = "accepter texte". (idem SLR(1)).

- Si $[A \to \alpha., b] \in I_i$ $(A \neq S')$
  Alors action (i, b) = "réduire par $A \to \alpha$"

- Autre : "erreur".

Remarque : Toute grammaire SLR(1) est LR(1).

Exemple : Soit G une grammaire telle que :

$S' \to S$ r_0
$S \to CC$ r_1
$C \to aC \mid d.$
r_2 r_3

est-elle LR(1) ?

Construire l'ensemble des états LR(1), c.à.d avec contexte.

Ensemble des états LR(1).

I_0 : $S' \to .S$, $\$$

puis fermeture de S

$S \to .CC$, $\$$

premier { rien , $\$ } = $\$$

puis fermeture de C.

$C \to .aC \mid .d$

premier (C) = { a, d } ⇒ permet d'avoir les contextes

donc $C \to .aC, a$ et $C \to .aC, d$; $C \to .d, a$; $C \to .d, d$ } ⇒ $C \to .aC, [a,d]$; $C \to .d, [a,d]$

Pour résumer : I_0 : $S' \to .S, \$$

$S \to .CC, \$$

$C \to .aC, [a,d] \mid .d, [a,d]$

I_1 : $AV(I_0, S)$

$S' \to S., \$$

propagation du contexte (pas de calcul).

I_2 = $AV(I_0, C)$.

$S \to C.C, \$$

fermeture de C : $C \to .aC, \$$

calcul des contextes $C \to .d, \$$

on trouve quoi derrière C ? rien et $\$$

I_3 : $AV(I_0, a)$

$C \to a.C, [a,d]$

fermeture de C : $C \to .aC, [a,d]$

$C \to .d, [a,d]$

I_4 = $AV(I_0, d)$

$C \to d., [a,d]$

pas de fermeture

I_5 : $AV(I_2, C)$.

$S \to CC., \$$

rien à faire.

I_6 : $AV(I_2, a)$

$C \to a.C, \$$

fermeture de C :

$C \to .aC, \$$

$C \to .d, \$$

I_7 : $AV(I_2, d)$

$C \to d., \$$.

I_8 : $AV(I_3, C)$

$C \to aC., [a,d]$

I_9 : $AV(I_6, C)$

$C \to aC., \$$.

$I_{\cancel{4}}$: $AV(I_3, a)$ = I_3

$C \to a.C, [a,d]$.

fermeture de C :

$C \to .aC, [a,d]$

$C \to .d, [a,d]$

$I_{\cancel{5}}$ = $AV(I_3, d)$ = I_4

$C \to d., [a,d]$

I_{10}. $AV(I_6, a)$ = I_6

$C \to a.C, \$$

fermeture de C...

$I_{\cancel{7}}$ = $AV(I_6, d)$ = I_7

Trad 1

C 10

Table d'analyse LR(1).

	a	d	$	S	C
I_0	d_{36}	d_{47}		1	2
I_1			(accepter) r_0		
I_2	d_6 d_{36}	d_7 d_{47}			5
I_{36}	d_{36}	d_{47}			89
I_{47}	r_3	r_3	r_3		
I_5			r_1		
~~I_6~~	d_6 d_{36}	d_7 d_{47}			89
~~I_7~~	r_3	r_3	r_3		
I_{89}	r_2	r_2	r_2		
~~I_9~~	r_2	r_2	r_2		

suppression de la ligne I_{36}, I_{47}, I_{89}

I_1 état de réduction (à cause de $S' \to S.$, $\$$.) réduire par $S' \to S$ (r_0) $\Rightarrow$ accepter. pour $\$$.

I_4 état de réduction réduire par r_3 dans $[a, d]$.

I_5 état de réduction réduire par r_1 dans $\$$

I_7 état de réduction réduire par r_3 dans $\$$

I_8 état de réduction réduire par r_2 dans $[a, d]$

I_9 état de réduction, réduire par r_2 pour $\$$

$\Rightarrow$ ici, pas de conflit $\Rightarrow$ l'analyseur est LR(1).

- lecture/réduction
- réduction/réduction.

Analyseur : LALR(1).

Regrouper des états LR(1) qui ont même noyau, mais des contextes différents.

exemple :
- I_3 et I_6 : I_{36} : $C \to a.C, [a, d, \$] \mid aC., [a, d, \$] \mid .d\,[a, d, \$]$.
- I_4 et I_7 : I_{47} : $C \to d., [a, d, \$]$.
- I_8 et I_9 : I_{89} : $C \to aC., [a, d, \$]$.

$\Rightarrow$ Toujours pas de conflit dans la table

$\Rightarrow$ l'analyseur est LALR(1).

Remarque : Toute grammaire LALR(1) est LR(1).

et pour finir sur l'ASA

La classe LR(0) trop faible pour traiter les langages de programmation.
Les classes [LR(1) ... LR(k)] → tables d'analyses très grandes à cause des contextes

→ trouver sa place entre LR(0) et LR(1)
LR(1) largement suffisant. (1 colonne par unité lexicale)
mais le nombre d'états grandit vite.

En pratique, on utilisera deux types d'analyseurs :
→ la puissance est comprise entre LR(0) et LR(1)
ce sont les SLR(1) et LALR(1).

Analyseur Simple LR (SLR)

Son automate est celui de LR(0)
Sa partie "transition" est la même que LR(0)
"action" shift la même que LR(0).
reduce : réduire par $A \to \alpha$ pour les Suivant(A).

Analyseur Look Ahead LR (LALR)

Son automate est celui de LR(1) dans lequel on fusionne les états ayant le même noyau. (ce sont des items LR(0)).

Un analyseur LALR(1) a autant d'états qu'un LR(0) ou SLR(1) [donc ≤ LR(1)]

cf TD [Une grammaire ambigüe n'est jamais LR(k), ∀k. Pourtant on a intérêt à utiliser des gram. ambigüe quitte à les trafiquer l'automate (si on peut !).

Trad 1
CM

Analyse ascendante et fonctions sémantiques.

X → {code 1} Y {code 2} bwT {code 3} Z {code 4}.

{code i} est du code pouvant faire un calcul ou réaliser des effets de bord.

① Au cours d'une ASD récursive : très facile !

```
Procedure analyse_X()
  cas unité_lex : {code 1}
                  analyse_Y(); {code 2}.
                  si code_unité = code_'b'
                  alors analyse_w();
                        analyse_T(); {code 3}
                        analyse_Z(); {code 4}.
```

② en ASA : (SLR(), LR() par exemple).

X → {code 1} Y {code 2} bwT {code 3} Z {code 4}

?? — exécuté lors de la réduction par X → YbwTZ.

Yacc procède ainsi ; transforme

X → X_1 Y X_2 bwT X_3 Z {code 4}

X_1 → ^ {code 1} X_2 → ^ {code 2} X_3 → ^ {code 3}.

des actions sémantiques sont toujours exécutées lors d'une réduction

X_i → . {code i}

⚠ Cela peut modifier le caractère LR() de la grammaire.

Exemple : $S' \to S$

$S \to \{\text{code } 1\}\ ab \mid \{\text{code } 2\}\ ac.$

LR(0) donc LR(1).

Par transformation :

$S' \to S$ $\qquad$ $X_1 \to \ ^{\wedge}\ \{\text{code } 1\}$

$S \to X_1\ ab \mid X_2\ ac$ $\qquad$ $X_2 \to \ ^{\wedge}\ \{\text{code } 2\}$

I_0 : $S' \to S, \$$

$S \to .\ X_1\ ab, \$ \mid .\ X_2\ ac, \$$

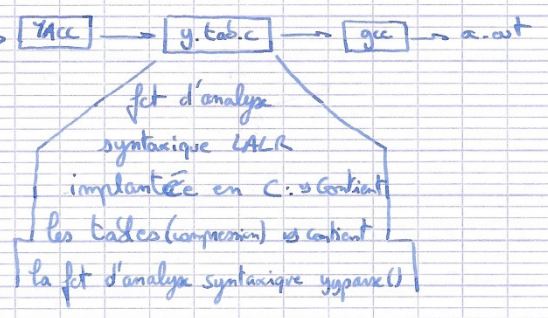
$X_1 \to .\ , a$
$X_2 \to .\ , a$) Conflit réduction/réduction.

Donc grammaire non LR(1) à cause des $\{\text{code } 1\}$ et $\{\text{code } 2\}$.
(Yacc réduira par la 1ère production rencontrée).

YACC : Yet Another Compiler Compiler

spécif yacc d'un langage algébrique

grammaire .y



Trad 1
C 12

Source Yacc

```
[ définitions en C
  (#define, include, var. globale, ...)
%%
  grammaire
%%
  routines C
```

```
        $1   $2   $3
expr :  expr '+' expr      } actions sémantiques {
     |  expr '-' expr      } langage C }
     |
     ;
```

actions exécutées uniquement lors des réductions.

→ le symbole $i référence la valeur associée au ième symbole de la partie droite (terminal ou non terminal).

→ le symbole $$ référence la valeur de l'attribut associé au non-terminal de la partie gauche

voir les poly :

```
lex   calc.lex
yacc  -vd  calc.yacc
gcc   -o  calc  y.tab.c  -ly  -ll
```

-v : verbose : produit calc.output

Pour résumer :

- Un analyseur lexical de nom yylex() doit être fourni à Yacc
- l'analyseur lexical yylex() produit des couples (unité lexicale, valeur associée)
  cette valeur est communiquée à l'analyse syntax. via la variable yylval
  code unité lexicale de type int par défaut.

- Aux lexèmes nommés sont associés les codes entiers (successifs) à partir de 257.
  → données dans | y.tab.h fourni par l'option -d de yacc.
  | calc.h

ex : # define REAL 257
  notre choix — auto déterminé.

(faire figurer ce.h dans le fichier lex)

- Aux caractères terminaux "simples", c'est-à-dire définis par un seul caractère est associé le code ASCII du caractère.

Gestion des piles dans Yacc.

2 piles | pile des états
| pile des valeurs

cf TD.

Yacc fait évoluer les deux en parallèle.

Trad 1

C 13

ⓐ Pile des valeurs.

type de ses éléments YYSTYPE

→ à chaque réduction par une production de longueur N, la pile des valeurs est dépilée N fois.
Les valeurs dépilées sont accessibles dans les actions sémantiques à l'aide des notations $i

→ Après l'exécution de l'action sémantique le contenu d'une variable (de yyparse()) est empilé dans la pile des valeurs
cette variable est accessible sous le nom $$.

→ Lors d'un "shift", yyparse() empile dans la pile des valeurs le contenu de la variable (externe) yylval.
Autrement dit : yylval est affectée dans yylex() afin de transmettre cette valeur (l'information lexicale) à l'analyseur syntaxique.

YYSTYPE permet de changer le type de yylval. (#define YYSTYPE double)
ou union si plusieurs types possibles.

```
%{
#include "typeCalc.h"
#include "calc.h"      /*  au lieu de  y.tab.h  dans lex-yacc */
extern YYSTYPE yylval;
%}

integer [0-9]+
real    ([0-9]*\.[0-9]+)
nl      \n


%%

[ \t]+      ; /* rien, on saute les blancs et TAB */
{integer}   { sscanf(yytext, "%d", &yylval.integer);
              return INTEGER;}
{real}      { sscanf(yytext, "%lf", &yylval.real);
              return REAL;}
{nl}        { extern lineno; lineno++;
              return('\n');}
quit        { return 0;}
.           { return(yytext[0]);}     /* le code ASCII du car. */

%%
```

calc. lex
calc. flex

```
#define REAL    257
#define INTEGER 258

extern YYSTYPE yylval;
```

calc.h

```
typedef union {
  double real;
  int integer;
}YYSTYPE;
```

↑ type des éléments de la pile des valeurs

typeCalc.h

```
%{
#include <stdio.h>
#include "typeCalc.h"
%}


/*  si pas de definition de type.h :
  %union{
      double real;
      int integer;
      }

      */
%start lines
```

↑ terminal

```
%token <real> REAL
%token <integer> INTEGER
```

↑ unité lexicale retournée par lex

```
%type <real> rexpr
%type <integer> iexpr
```

↑ terminal

```
%left '+','-'
%left '*','/'

%%

lines   : /* rien */
        | lines exp
        ;
exp     : '\n'
        | iexpr '\n' { printf("%d\n", $1);}
        | rexpr '\n' { printf("%15.8lf\n", $1);}
        ;
iexpr   : INTEGER
        | iexpr '+' iexpr { $$ = $1 + $3; }
        | iexpr '-' iexpr { $$ = $1 - $3; }
        | iexpr '*' iexpr { $$ = $1 * $3; }
        | iexpr '/' iexpr {  if ($3) $$ = $1 / $3;
                             else { fprintf(stderr, "division par zero\n");
                                    yyerror();  /* au lieu de yyerror;
                                                   dans lex-yacc      */}
                           }
        | '(' iexpr ')' { $$ = $2; }
        ;
rexpr   :  REAL
        | rexpr '+' rexpr { $$ = $1 + $3; }
        | rexpr '-' rexpr { $$ = $1 - $3; }
        | rexpr '*' rexpr { $$ = $1 * $3; }
        | rexpr '/' rexpr {  if ($3) $$ = $1 / $3;
                             else { fprintf(stderr, "division par zero\n");
                                    yyerror(); }
                           }
        | '(' rexpr ')' { $$ = $2; }
        | iexpr '+' rexpr { $$ = (double)$1 + $3; }
        | iexpr '-' rexpr { $$ = (double)$1 - $3; }
        | iexpr '*' rexpr { $$ = (double)$1 * $3; }
        | iexpr '/' rexpr {  if ($3) $$ = (double)$1 / $3;
                             else { fprintf(stderr, "division par zero\n");
                                    yyerror(); }
                          }
```

```
        ;

%%

char *progname;
int lineno;

int main(int argc, char **argv)
{
 progname = argv[0];

 yyparse();
}


int yyerror(char *s)
{

 fprintf(stderr, "%s: %s\n", progname, s);
 fprintf(stderr, " ligne numero %d\n", lineno);
}
```

{ calc.yacc
  calc.bison

calc.output

```
Conflict in state 23 between rule 7 and token '+' resolved as reduce.
Conflict in state 23 between rule 7 and token '-' resolved as reduce.
Conflict in state 23 between rule 7 and token '*' resolved as shift.
Conflict in state 23 between rule 7 and token '/' resolved as shift.
Conflict in state 24 between rule 18 and token '+' resolved as reduce.
Conflict in state 24 between rule 18 and token '-' resolved as reduce.
Conflict in state 24 between rule 18 and token '*' resolved as shift.
Conflict in state 24 between rule 18 and token '/' resolved as shift.
Conflict in state 25 between rule 8 and token '+' resolved as reduce.
Conflict in state 25 between rule 8 and token '-' resolved as reduce.
Conflict in state 25 between rule 8 and token '*' resolved as shift.
Conflict in state 25 between rule 8 and token '/' resolved as shift.
Conflict in state 26 between rule 19 and token '+' resolved as reduce.
Conflict in state 26 between rule 19 and token '-' resolved as reduce.
Conflict in state 26 between rule 19 and token '*' resolved as shift.
Conflict in state 26 between rule 19 and token '/' resolved as shift.
Conflict in state 27 between rule 9 and token '+' resolved as reduce.
Conflict in state 27 between rule 9 and token '-' resolved as reduce.
Conflict in state 27 between rule 9 and token '*' resolved as reduce.
Conflict in state 27 between rule 9 and token '/' resolved as reduce.
Conflict in state 28 between rule 20 and token '+' resolved as reduce.
Conflict in state 28 between rule 20 and token '-' resolved as reduce.
Conflict in state 28 between rule 20 and token '*' resolved as reduce.
Conflict in state 28 between rule 20 and token '/' resolved as reduce.
Conflict in state 29 between rule 10 and token '+' resolved as reduce.
Conflict in state 29 between rule 10 and token '-' resolved as reduce.
Conflict in state 29 between rule 10 and token '*' resolved as reduce.
Conflict in state 29 between rule 10 and token '/' resolved as reduce.
Conflict in state 30 between rule 21 and token '+' resolved as reduce.
Conflict in state 30 between rule 21 and token '-' resolved as reduce.
Conflict in state 30 between rule 21 and token '*' resolved as reduce.
Conflict in state 30 between rule 21 and token '/' resolved as reduce.
Conflict in state 32 between rule 13 and token '+' resolved as reduce.
Conflict in state 32 between rule 13 and token '-' resolved as reduce.
Conflict in state 32 between rule 13 and token '*' resolved as shift.
Conflict in state 32 between rule 13 and token '/' resolved as shift.
Conflict in state 33 between rule 14 and token '+' resolved as reduce.
Conflict in state 33 between rule 14 and token '-' resolved as reduce.
Conflict in state 33 between rule 14 and token '*' resolved as shift.
Conflict in state 33 between rule 14 and token '/' resolved as shift.
Conflict in state 34 between rule 15 and token '+' resolved as reduce.
Conflict in state 34 between rule 15 and token '-' resolved as reduce.
Conflict in state 34 between rule 15 and token '*' resolved as reduce.
Conflict in state 34 between rule 15 and token '/' resolved as reduce.
Conflict in state 35 between rule 16 and token '+' resolved as reduce.
Conflict in state 35 between rule 16 and token '-' resolved as reduce.
Conflict in state 35 between rule 16 and token '*' resolved as reduce.
Conflict in state 35 between rule 16 and token '/' resolved as reduce.

Grammar
rule 1    lines ->                /* empty */
rule 2    lines -> lines exp
rule 3    exp -> '\n'
rule 4    exp -> iexpr '\n'
rule 5    exp -> rexpr '\n'
rule 6    iexpr -> INTEGER
rule 7    iexpr -> iexpr '+' iexpr
rule 8    iexpr -> iexpr '-' iexpr
rule 9    iexpr -> iexpr '*' iexpr
rule 10   iexpr -> iexpr '/' iexpr
rule 11   iexpr -> '(' iexpr ')'
rule 12   rexpr -> REAL
rule 13   rexpr -> rexpr '+' rexpr
rule 14   rexpr -> rexpr '-' rexpr
rule 15   rexpr -> rexpr '*' rexpr
rule 16   rexpr -> rexpr '/' rexpr
rule 17   rexpr -> '(' rexpr ')'
rule 18   rexpr -> iexpr '+' rexpr
rule 19   rexpr -> iexpr '-' rexpr
rule 20   rexpr -> iexpr '*' rexpr
rule 21   rexpr -> iexpr '/' rexpr

Terminals, with rules where they appear

$ (-1)
'\n' (10) 3 4 5
'(' (40) 11 17
')' (41) 11 17
'*' (42) 9 15 20
'+' (43) 7 13 18
'-' (45) 8 14 19
'/' (47) 10 16 21
error (256)
REAL (257) 12
INTEGER (258) 6

Nonterminals, with rules where they appear

lines (12)
    on left: 1 2, on right: 2
exp (13)
    on left: 3 4 5, on right: 2
iexpr (14)
    on left: 6 7 8 9 10 11, on right: 4 7 8 9 10 11 18 19 20 21
rexpr (15)
    on left: 12 13 14 15 16 17 18 19 20 21, on right: 5 13 14 15 16
    17 18 19 20 21


state 0

    $default    reduce using rule 1 (lines)

    lines       go to state 1


state 1

    lines  ->  lines . exp   (rule 2)

    $           go to state 36
    REAL        shift, and go to state 2
    INTEGER     shift, and go to state 3
    '\n'        shift, and go to state 4
    '('         shift, and go to state 5

    exp         go to state 6
    iexpr       go to state 7
    rexpr       go to state 8
```

```
state 2

    rexpr  ->  REAL .    (rule 12)

    $default    reduce using rule 12 (rexpr)


state 3

    iexpr  ->  INTEGER .    (rule 6)

    $default    reduce using rule 6 (iexpr)


state 4

    exp  ->  '\n' .    (rule 3)

    $default    reduce using rule 3 (exp)


state 5

    iexpr  ->  '(' . iexpr ')'    (rule 11)
    rexpr  ->  '(' . rexpr ')'    (rule 17)

    REAL        shift, and go to state 2
    INTEGER     shift, and go to state 3
    '('         shift, and go to state 5

    iexpr       go to state 9
    rexpr       go to state 10


state 6

    lines  ->  lines exp .    (rule 2)

    $default    reduce using rule 2 (lines)


state 7

    exp  ->  iexpr . '\n'    (rule 4)
    iexpr  ->  iexpr . '+' iexpr    (rule 7)
    iexpr  ->  iexpr . '-' iexpr    (rule 8)
    iexpr  ->  iexpr . '*' iexpr    (rule 9)
    iexpr  ->  iexpr . '/' iexpr    (rule 10)
    rexpr  ->  iexpr . '+' rexpr    (rule 18)
    rexpr  ->  iexpr . '-' rexpr    (rule 19)
    rexpr  ->  iexpr . '*' rexpr    (rule 20)
    rexpr  ->  iexpr . '/' rexpr    (rule 21)

    '+'         shift, and go to state 11
    '-'         shift, and go to state 12
    '*'         shift, and go to state 13
    '/'         shift, and go to state 14
    '\n'        shift, and go to state 15


state 8

    exp  ->  rexpr . '\n'    (rule 5)
    rexpr  ->  rexpr . '+' rexpr    (rule 13)
    rexpr  ->  rexpr . '-' rexpr    (rule 14)
    rexpr  ->  rexpr . '*' rexpr    (rule 15)
    rexpr  ->  rexpr . '/' rexpr    (rule 16)

    '+'         shift, and go to state 16
    '-'         shift, and go to state 17
    '*'         shift, and go to state 18
    '/'         shift, and go to state 19
    '\n'        shift, and go to state 20


state 9

    iexpr  ->  iexpr . '+' iexpr    (rule 7)
    iexpr  ->  iexpr . '-' iexpr    (rule 8)
    iexpr  ->  iexpr . '*' iexpr    (rule 9)
    iexpr  ->  iexpr . '/' iexpr    (rule 10)
    iexpr  ->  '(' iexpr . ')'    (rule 11)
    rexpr  ->  iexpr . '+' rexpr    (rule 18)
    rexpr  ->  iexpr . '-' rexpr    (rule 19)
    rexpr  ->  iexpr . '*' rexpr    (rule 20)
    rexpr  ->  iexpr . '/' rexpr    (rule 21)

    '+'         shift, and go to state 11
    '-'         shift, and go to state 12
    '*'         shift, and go to state 13
    '/'         shift, and go to state 14
    ')'         shift, and go to state 21


state 10

    rexpr  ->  rexpr . '+' rexpr    (rule 13)
    rexpr  ->  rexpr . '-' rexpr    (rule 14)
    rexpr  ->  rexpr . '*' rexpr    (rule 15)
    rexpr  ->  rexpr . '/' rexpr    (rule 16)
    rexpr  ->  '(' rexpr . ')'    (rule 17)

    '+'         shift, and go to state 16
    '-'         shift, and go to state 17
    '*'         shift, and go to state 18
    '/'         shift, and go to state 19
    ')'         shift, and go to state 22
```