Langages, Compilation, Automates. Partie 5: Analyse syntaxique Grammaires LR, yacc

Florian Bridoux

Polytech Nice Sophia

2022-2023

Table des matières

Structure du compilateur

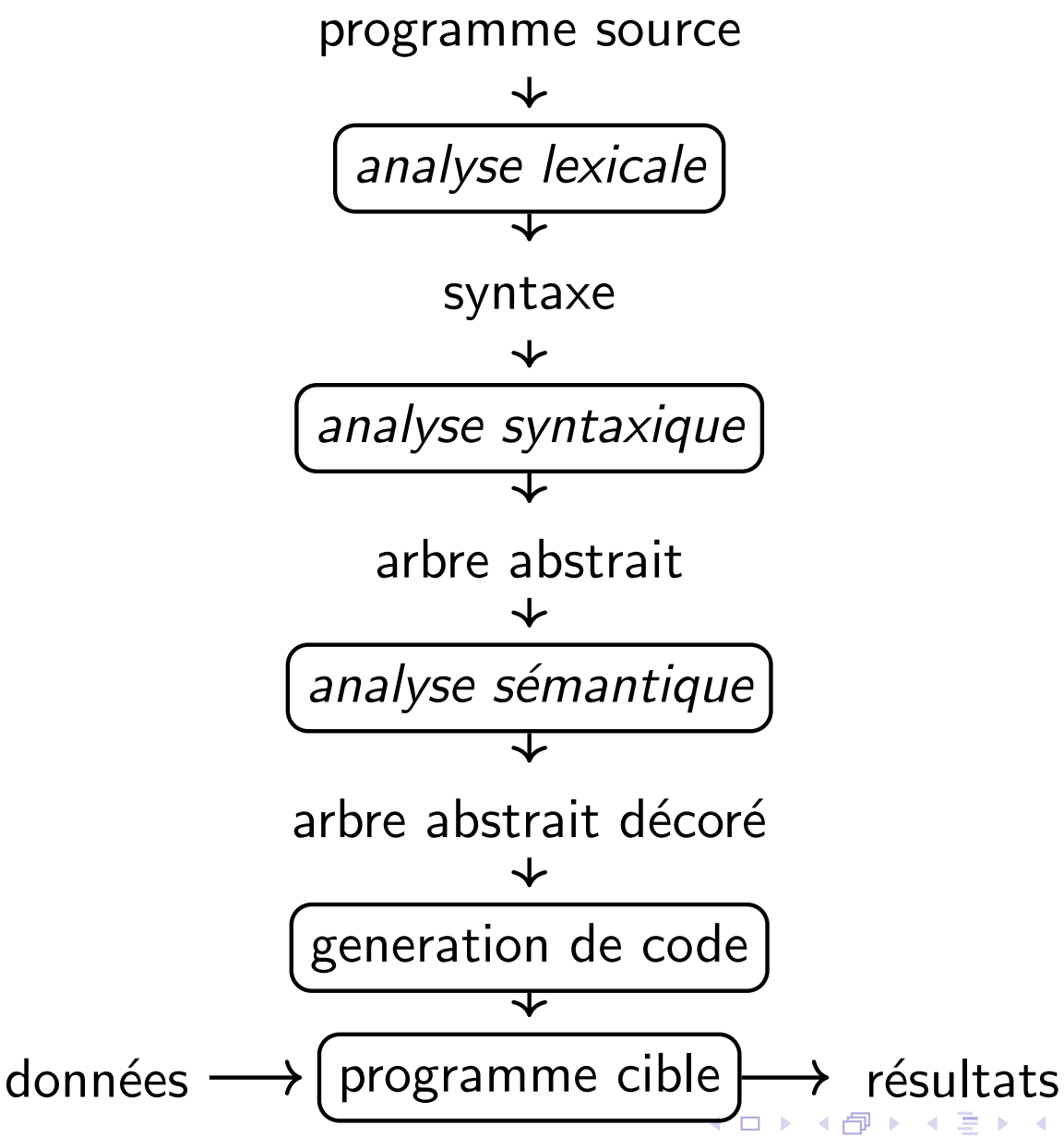
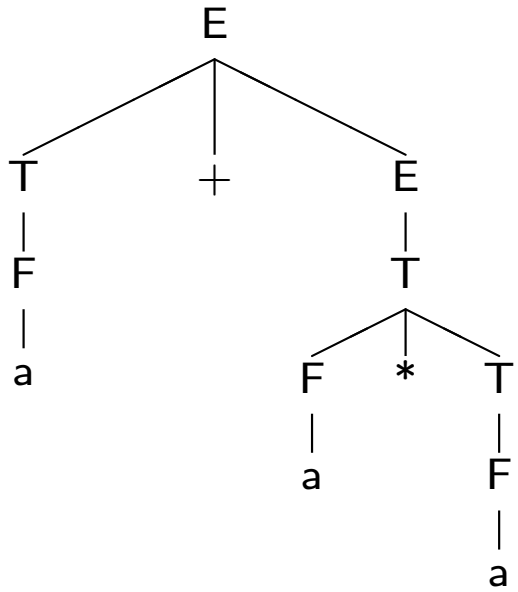


Table des matières

Analyse syntaxique

Dans le contexte d'une grammaire donnée, analyser m consiste à trouver son arbre de dérivation (aussi appelé *arbre de syntaxe*).

$$
\begin{array}{lcl}
E & \rightarrow & T + E \mid T \\
T & \rightarrow & F * T \mid F \\
F & \rightarrow & (E) \mid a
\end{array}
$$



Sens d'analyse

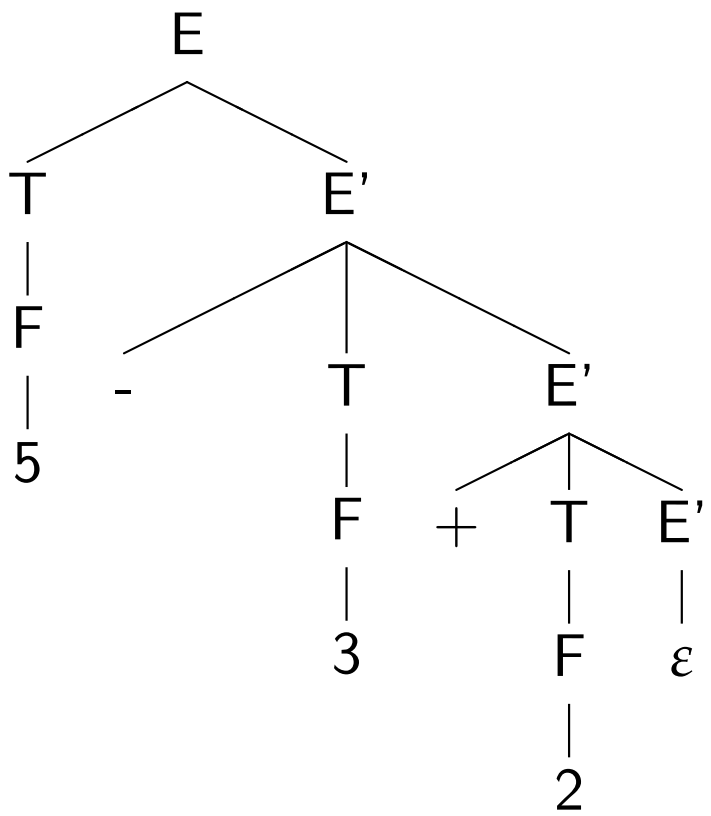
$$E \rightarrow T + E | T$$
$$T \rightarrow F * T | F$$
$$F \rightarrow (E) | a$$

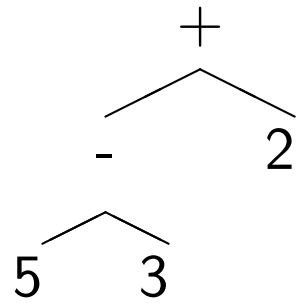
- **Analyse descendante:**
  L'arbre de dérivation est construit depuis la racine vers les feuilles
  Séquence de dérivations gauches à partir de l'axiome
  $E \Rightarrow T + E \Rightarrow F + E \Rightarrow a + E \Rightarrow a + T \Rightarrow a + F * T \Rightarrow a + a * T \Rightarrow a + a * F \Rightarrow a + a * a$
- **Analyse ascendante:**
  L'arbre de dérivation est construit des feuilles vers la racine
  Séquence de dérivation telle que la séquence inverse soit une dérivation droite de m.
  $a + a * a \Leftarrow F + a * a \Leftarrow T + a * a \Leftarrow T + F * a \Leftarrow T + F * F \Leftarrow T + F * T \Leftarrow T + E \Leftarrow E$

Arbre de dérivation v/s arbre abstrait

- L'arbre de dérivation produit par l'analyse syntaxique possède de nombreux nœuds superflus, qui ne véhiculent pas d'information.
- De plus, la mise au point d'une grammaire (élimination de l'ambiguïté, priorisation de certains opérateurs, factorisation,...) nécessite souvent l'introduction de règles dont le seul but est de simplifier l'analyse syntaxique.
- Un **arbre abstrait** constitue une interface plus naturelle entre l'analyse syntaxique et l'analyse sémantique, elle ne garde de la structure syntaxique que les parties nécessaires à l'analyse sémantique et à la production de code.

Arbre de dérivation v/s arbre abstrait





Construction de l'arbre abstrait

L'arbre abstrait est construit lors de l'analyse syntaxique, en associant à toute règle de grammaire une **action sémantique** et à chaque symbole de la grammaire des attribues.

- Un **attribut** est une information quelconque associée aux nœuds de l'arbre de dérivation.
- Exemples :
  - le type d'une expression
  - la ligne du programme (pour débuggage)
- Les nœuds de l'arbre étant représentées par les symboles de la grammaire, on associe les attributs à ces derniers.
- Notations : $A.t$ est l'attribut t associé au symbole A.

Grammaires attribuées

- Une **grammaire attribuée** est une grammaire dont les règles sont attachées à des fragments de programme appelé **actions sémantiques**.
- Les actions sémantiques sont exécutées quand la règle associée est utilisée lors de l'analyse syntaxique.
- Les actions sémantiques servent à:
  - créer les nœuds de l'arbre abstrait
  - monter (synthétiser) ou descendre (hériter) des informations via ses attributs à ses enfants (analyse descendante) ou de son père (analyse ascendante).
  - (éventuellement) détecter des erreurs dans le programme compilé non prise en compte par la grammaire
  - (éventuellement) afficher des informations sur l'arbre de dérivation (débuggage)

Construction de l'arbre abstrait

Objectif : construire l'arbre abstrait pendant la dérivation (ascendante)
Attribut : *n* noeud de l'arbre abstrait
Constructeur : *noeud(op, g, d)*

- *op* $\rightarrow$ opérateur ou valeur
- *g* $\rightarrow$ fils gauche
- *d* $\rightarrow$ fils droit

règle			action sémantique		
E	$\rightarrow$	E + T	E.n	=	noeud(+, E_1.n, T.n)
E	$\rightarrow$	E - T	E.n	=	noeud($-$, E_1.n, T.n)
E	$\rightarrow$	T	E.n	=	T.n
T	$\rightarrow$	T * F	T.n	=	noeud($\times$, T_1.n, F.n)
T	$\rightarrow$	T / F	T.n	=	noeud($\div$, T_1.n, F.n)
T	$\rightarrow$	F	T.n	=	F.n
F	$\rightarrow$	(E)	F.n	=	E.n
F	$\rightarrow$	entier	F.n	=	noeud(entier, NULL , NULL)

Table des matières

Analyse par décalage-réduction

L'analyse faite par `yacc` est une analyse ascendante, 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**.
- S'il existe une règle $(i) : P \longrightarrow \alpha_1 \dots \alpha_k$ et que les k symboles au sommet de la pile sont justement $\alpha_1 \dots \alpha_k$, alors on dépile les k symboles et on empile P.
- Cette opération s'appelle **réduction (i)**.
- 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.

Analyse par décalage-réduction

Grammaire à 2 règles:

(1) $S \longrightarrow aSb$

(2) $S \longrightarrow ab$

Analyse de *aabb*

Analyse par décalage-réduction

Grammaire à 2 règles:

(1) $S \longrightarrow aSb$

(2) $S \longrightarrow ab$

Analyse de *aabb*

pile de symbole	terminaux non décalés	opération
	aabb	

Analyse par décalage-réduction

Grammaire à 2 règles:

(1) $S \longrightarrow aSb$

(2) $S \longrightarrow ab$

Analyse de *aabb*

pile de symbole	terminaux non décalés	opération
	aabb	décalage
a	*abb*	

Analyse par décalage-réduction

Grammaire à 2 règles:

(1) $S \longrightarrow aSb$

(2) $S \longrightarrow ab$

Analyse de *aabb*

pile de symbole	terminaux non décalés	opération
	aabb	décalage
a	*abb*	décalage
aa	*bb*	

Analyse par décalage-réduction

Grammaire à 2 règles:

(1) $S \longrightarrow aSb$

(2) $S \longrightarrow ab$

Analyse de *aabb*

pile de symbole	terminaux non décalés	opération
	aabb	décalage
a	*abb*	décalage
aa	*bb*	décalage
aab	*b*	

Analyse par décalage-réduction

Grammaire à 2 règles:

(1) $S \longrightarrow aSb$

(2) $S \longrightarrow ab$

Analyse de $aabb$

pile de symbole	terminaux non décalés	opération
	$aabb$	décalage
a	abb	décalage
aa	bb	décalage
aab	b	réduction (2)
aS	b	

Analyse par décalage-réduction

Grammaire à 2 règles:

(1) $S \longrightarrow aSb$

(2) $S \longrightarrow ab$

Analyse de *aabb*

pile de symbole	terminaux non décalés	opération
	aabb	décalage
a	*abb*	décalage
aa	*bb*	décalage
aab	*b*	réduction (2)
aS	*b*	décalage
aSb		

Analyse par décalage-réduction

Grammaire à 2 règles:

(1) $S \longrightarrow aSb$

(2) $S \longrightarrow ab$

Analyse de *aabb*

pile de symbole	terminaux non décalés	opération
	aabb	décalage
a	*abb*	décalage
aa	*bb*	décalage
aab	*b*	réduction (2)
aS	*b*	décalage
aSb		réduction (1)
S		

La pile ne comporte que S et tous les terminaux ont été décalés:

$\Rightarrow$ l'analyse a réussi.

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) $S \longrightarrow X$

(2) $S \longrightarrow Y$

(3) $X \longrightarrow a$

(4) $Y \longrightarrow a$

pile de symbole	terminaux non décalés	opération
	a	décalage
a		réduction(3)/réduction(4)

On ne sait pas s'il faut réduire en utilisant la règle 3 ou 4.

Conflit de réduction / réduction

(1) $S \longrightarrow Xb$

(2) $S \longrightarrow Yc$

(3) $X \longrightarrow a$

(4) $Y \longrightarrow a$

pile de symbole	terminaux non décalés	opération
	ab	décalage
a	b	réduction(3)
X	b	décalage
Xb		réduction(1)
S		

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

Conflit de décalage / réduction

(1) $S \longrightarrow X$

(2) $S \longrightarrow Yr$

(3) $X \longrightarrow ar$

(4) $Y \longrightarrow a$

pile de symbole	terminaux non décalés	opération
	ar	décalage
a	r	réduction(3)/décalage

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.
- Le premier L signifie *left*: on lit le mot de gauche à droite. Le second R signifie *right*: on remplace les symboles les plus à droites.
- Une grammaire LR est une grammaire $LR(1)$.
- `yacc` nécessite une grammaires LR.
- La grammaire suivante n'est pas $LR(1)$ mais elle est $LR(2)$:
  (1) $S \longrightarrow Xbc$
  (2) $S \longrightarrow Ybd$
  (3) $X \longrightarrow a$
  (4) $Y \longrightarrow a$

Que faire en cas de conflit ?

- Vérifier que la grammaire n'est pas ambiguë : une grammaire non ambiguë peut ne pas être $LR(k)$;
- Selon la source du conflit il sera parfois nécessaire de factoriser la grammaire, de supprimer des productions vides,...
- Dans certain cas, il faut simplifier la grammaire, quitte à accepter des programmes non désirés et repérer les erreurs plus tard dans la compilation (analyse sémantique).

Table des matières

Structure d'un fichier bison

```
%{
  Partie 1 : déclarations pour le compilateur C
%}
  Partie 2 : déclarations pour {\tt bison}
%%
  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 bison

- Une directive `union` décrivant les types des valeurs associées aux symboles terminaux et non terminaux.
- Les symboles terminaux (et leurs types quand nécessaire).
- Les déclarations des symboles non terminaux, enrichis de leurs types.

```
%union {char* sval; int entier; n_exp* exp;}

%token <sval> IDENTIFIANT
%token PLUS
%token FOIS
%token <entier> ENTIER
%token FIN 0

%type <exp> expr
```

Schémas de traduction

Un schéma de traduction est un ensemble de règles, chacune associée à une action sémantique.

```
expr: expr PLUS expr { $$ =creer_n_op('+', $1, $3); }

expr: expr FOIS expr{ $$ =creer_n_op('*', $1 , $3);}

expr: ENTIER { $$ = creer_n_entier($1); }
```

Le symbole qui constitue la partie gauche de la première règle 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 `bison`

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

La fonction yyerror(char *message)

- Lorsque la fonction yyparse() 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 règle.

- Cette séquence est recopiée par `bison` 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.

Table des matières

Syntaxe sly.Parser

L'analyseur prend la forme d'une classe sly.Parser dont on peut hériter.

```
class FloParser(Parser):
 tokens = FloLexer.tokens #tokens de l'analyse lexicale

 @_('expr "+" expr')
  def expr(self, p):
   return arbre_abstrait.Operation('+',p[0],p[2])

 @_('expr "*" expr')
 def expr(self, p):
  return arbre_abstrait.Operation('*',p[0],p[2])

 @_('ENTIER')
 def expr(self, p):
  return arbre_abstrait.Entier(p.ENTIER)
```

Schémas de traduction

On encode une règle du type *expr* $\longrightarrow$ `ENTIER` en mettant ENTIER dans une string en décoration d'une fonction nommé `expr`. L'action sémantique associée à cette règle est le corps de la fonction.

```
@_('ENTIER')
def expr(self, p):
    return arbre_abstrait.Entier(p.ENTIER)
```

ou

```
@_('ENTIER')
def expr(self, p):
    return arbre_abstrait.Entier(p[0])
```

La valeur envoyé par un terminal est celle qui a été donnée dans le Lexer (par défaut la chaîne de caractère).

Schémas de traduction

Le symbole qui constitue la partie gauche de la première règle est l'axiome de la grammaire.
On peut (doit) lancer l'analyse syntaxique en s'appuyant sur l'analyse lexicale.

```
with open(sys.argv[1],"r") as f:
 data = f.read()
 try:
  arbre = parser.parse(lexer.tokenize(data))
  arbre.afficher()
  except EOFError:
   exit()
```

Le retour de l'analyse syntaxique est le retour de la règle (ou l'une des règles) associé à l'axiome.