IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

TP4 – Compilation

L'analyse syntaxique LR(1) et les conflits

Ingénierie du développement logiciel – F2B304

Correction

1 Introduction

L'analyse syntaxique consiste à regrouper les *unités lexicales* (les mots) en *unités syntaxiques* (les phrases). Les unités syntaxiques sont généralement représentées sous forme d'arbre : l'*arbre syntaxique abstrait (AST)*.

La forme de la grammaire du langage guidera alors le choix du type d'algorithme de reconnaissance. Pour les grammaires hors contexte, il existe deux principales classes d'analyse syntaxique : l'analyse descendante, dite LL, qui permet de produire la dérivation à gauche ou prefixe et l'analyse ascendante, dite LR, qui produit la dérivation à droite ou postfixe.

Dans ce module, on se concentre sur l'analyse LR et plus précisément sur l'analyse LR(1) [1] car l'outil menhir l'utilise. Commençons par un exemple d'analyse ascendante. Soit la grammaire suivante, où E, T et F sont des non terminaux, `+`, `*` et *id* sont des terminaux et chaque règle est suivie d'une action :

(1)	$E \rightarrow E$ `+` T	`+`(\$1,\$3)
(2)	$E \rightarrow T$	\$1
(3)	$T \rightarrow T$ `*` F	`*`(\$1,\$3)
(4)	$T \rightarrow F$	\$1
(5)	$F \rightarrow id$	\$1

L'analyse de la suite d'unités lexicales `a1 * a2` commence de cette suite de terminaux et va remonter jusqu'à reconnaître le non-terminal initial (ici E). Elle est décrite dans la figure 1 par une suite d'éléments reconnus, la transition étant étiquetée par le numéro de la règle de la grammaire correspondant. Ainsi, on obtient la suite des règles qui ont été appliquées (ici 5, 4, 5, 3 puis 2). Les actions correspondant à ces règles seront donc exécutées dans cet ordre. Ici par exemple, les seules construisant réellement l'AST sont 1, 3 et 5. On obtient l'AST `*(a1,a2)`.

1. Une analyse LR(k) utilise k symboles en avance pour guider la reconnaissance.

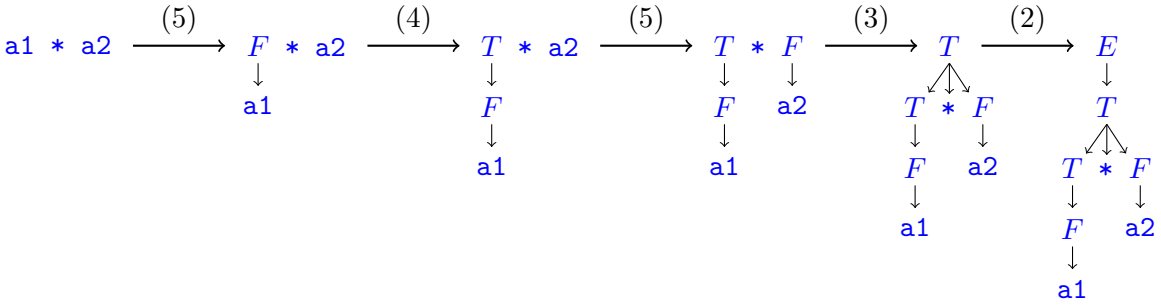

FIGURE 1 – Exemple de reconnaissance ascendante

L'analyse LR repose sur l'utilisation d'un automate à pile et sur une table qui va guider les actions de reconnaissance à chaque pas. Toutes les *analyses LR*[2] utilisent le même schéma algorithmique. Elles ne se différencient que par le calcul des tables et donc leur forme. Ce schéma général se place dans le cadre de la figure 2. La boîte analyseur s'occupe de la reconnaissance en utilisant la chaîne d'entrée (une suite d'unités lexicales notées t_j), une pile d'états (notés s_j) et deux tables, celle des actions et celle des successeurs.

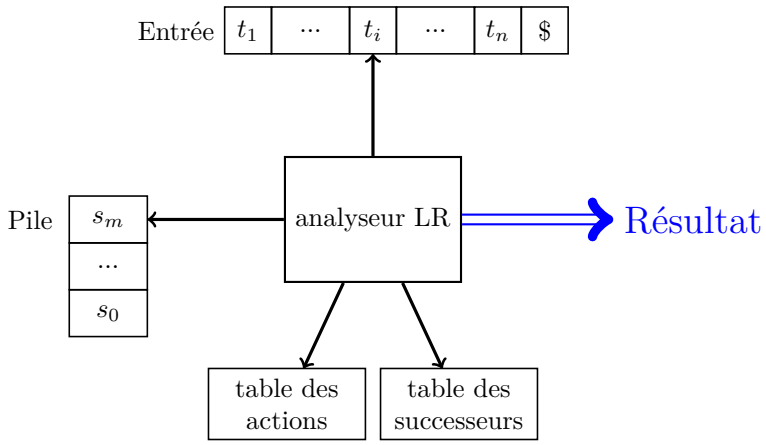

FIGURE 2 – Modèle d'un analyseur syntaxique LR

On décrira un analyseur LR dans l'état de celui de la figure par $s_0...s_m,\ t_i...t_n\$$. La table des actions est notée A, elle associe un couple état courant et symbole lu à une action. Celle des successeurs est notée S et elle associe un état et un non-terminal à un autre état. Il existe quatre actions :

1. un décalage vers un état s, *shift* en anglais, noté `shift` s ;
2. une réduction par la règle r, *reduce* en anglais, notée `reduce` r, une règle a la forme $A \to \beta$, sa longueur notée $\mathsf{len}(r)$ correspond au nombre d'éléments de β, son non-terminal noté $\mathsf{nt}(r)$ correspond à A ;
3. une acceptation notée `Accept` qui marque la fin de la reconnaissance ;
4. une erreur notée `error` qui interrompt la reconnaissance.

Initialement, la configuration est $s_0,\ t_1...t_n\$$ pour reconnaître la suite d'unité lexicales $t_1...t_n$. On décrira l'algorithme par une suite de couple configuration courante et suite des règles résultat de

2. Par exemple, les analyses SLR, LALR, LR(0), LR(1)...

la reconnaissance ($\perp$ exprime l'erreur et $\varnothing$ la fin). Les parties modifiées en partie gauche sont mises en orange pour faciliter la lecture.

$$s_0 ... s_m,\ t_i ... t_n\$ \quad r_0 ... r_k \longrightarrow$$
$$\begin{cases} s_0 ... s_m\ s,\ t_{i+1} ... t_n\$ & r_0 ... r_k & \text{si } \mathrm{A}(s_m, t_i) = \texttt{shift}\ s \\ s_0 ... s_l\ \mathrm{S}(s_l, \mathsf{nt}(r)),\ t_i ... t_n\$ & r_0 ... r_k\ r & \text{si } \mathrm{A}(s_m, t_i) = \texttt{reduce}\ r \text{ et } l = m - \mathsf{len}(r) \\ \varnothing & r_0 ... r_k & \text{si } \mathrm{A}(s_m, t_i) = \texttt{accept} \\ \varnothing & \perp & \text{si } \mathrm{A}(s_m, t_i) = \texttt{error} \end{cases}$$

Par un algorithme pas encore présenté, on construirait les tables d'actions et de successeurs suivantes pour la grammaire de la page précédente :

état	actions				successeurs		
	id	+	*	$	*E*	*T*	*F*
s_0	shift s_4	error	error	error	s_1	s_2	s_3
s_1	error	shift s_5	error	accept			
s_2	error	reduce 2	shift s_6	reduce 2			
s_3	error	reduce 4	reduce 4	reduce 4			
s_4	error	reduce 5	reduce 5	reduce 5			
s_5	shift s_4	error	error	error		s_7	s_3
s_6	shift s_4	error	error	error			s_8
s_7	error	reduce 1	shift s_6	reduce 1			
s_8	error	reduce 3	reduce 3	reduce 3			

Ce qui, appliqué à l'exemple, donne la suite de configuration suivante :

$$s_0,\ \texttt{a1*a2}\$ \longrightarrow s_0\, s_4,\ \texttt{*a2}\$ \longrightarrow s_0\, s_3,\ \texttt{*a2}\$\ (5) \longrightarrow s_0\, s_2,\ \texttt{*a2}\$\ (5)\,(4) \longrightarrow$$
$$s_0\, s_2\, s_6,\ \texttt{a2}\$\ (5)\,(4) \longrightarrow s_0\, s_2\, s_6\, s_4,\ \$\ (5)\,(4) \longrightarrow s_0\, s_2\, s_6\, s_8,\ \$\ (5)\,(4)\,(5) \longrightarrow$$
$$s_0\, s_2,\ \$\ (5)\,(4)\,(5)\,(3) \longrightarrow s_0\, s_1,\ \$\ (5)\,(4)\,(5)\,(3)\,(2) \longrightarrow \varnothing\ (5)\,(4)\,(5)\,(3)\,(2)$$

On retrouve la suite décrite intuitivement dans la figure 1. Remarquez que dans la pratique de l'outil `menhir`, la suite de règle correspondra à l'enchaînement de leurs actions qui sont censées produire l'AST par exemple.

Exercice 1

▷ **Question 1.1 :**
Implanter la grammaire précédente en utilisant `menhir`.

```
%{
  open Expr
%}
```

```
%token EOF PLUS TIMES
%token <string> IDENT
%start expression
%type < Expr.expression > expression
%%
expression:
 | e EOF     { $1 }
e:
 | e PLUS t   { Plus($1,$3)}
 | t       { $1 }
t:
 | t TIMES f   { Times($1,$3)}
 | f       { $1 }
f:
 | IDENT     { Var $1 }
%%
```

En utilisant l'option `--dump`[3], `menhir` peut vous générer l'automate de reconnaissance.

▷ **Question 1.2 :**
Comparer l'automate généré par `menhir` avec celui qui vous est fourni.

L'automate est identique à un renommage des états près et moyennant l'ajout de la règle expression pour prendre en compte la fin du flux d'unités lexicales (cela rajoute 2 états et un non-terminal).

2 Les conflits

Lors de la construction des tables d'analyse LR, il peut arriver qu'une case contienne plusieurs actions. Cette situation indique que la grammaire n'est pas LR et contient des ambiguïtés. Il va alors falloir résoudre le problème causé par cette ambiguïté. Trois actions sont possibles pour supprimer un conflit :

1. changer le langage,
2. modifier la grammaire,
3. regrouper les bouts d'AST en conflit en une seule structure et déléguer la résolution de l'ambiguïté à l'étape de traitement sémantique.

Ici, on va se limiter au cas 2.

2.1 Donner des priorités

Exercice 2

Soit la grammaire suivante :

3. Si vous utilisez `ocamlbuild`, il faut utiliser `-yaccflag --dump` pour qu'il passe l'option `--dump` à `menhir`.

```
%{
 open Expr
%}
%token EOF PLUS TIMES
%token <string> IDENT
%start expression
%type < Expr.expression > expression
%%
expression:
 | e EOF     { $1 }
e:
  | e PLUS e   { Plus($1,$3)}
  | e TIMES e  { Times($1,$3)}
  | IDENT     { Var $1 }
%%
```

▷ **Question 2.1 :**
Analyser l'automate produit par `menhir`. Identifier les conflits et donner des exemples d'expressions ambigües.

Les états 5 et 7 présentent tous les deux des conflits *shift/reduce*. Ces deux états sont similaires, ils correspondent à une situation où l'on vient de lire un e soit comme première sous-expression d'une opération $+$ ou $*$ ($e \to e \bullet + e$ ou $e \to e \bullet * e$), soit après une opération ($e \to e * e \bullet$ pour 5, $e \to e + e \bullet$ pour 7). Il y un conflit pour la lecture de $+$ et $*$ entre décalage ou réduction de par $e \to e * e$ ou $e \to e + e$.
`a1 * a2 + a3` est une expression ambigüe, elle est reconnue comme `a1 * (a2 + a3)` car lors du conflit dans la configuration $s_0\, s_3\, s_4\, s_5$, `+ a3`, le *shift* vers s_6 est privilégié. Si la réduction avait été privilégiée, on aurait obtenu `(a1 * a2) + a3`.

En `menhir`, on dispose des mots clefs `%left` et `%right` pour indiquer en cas de conflit la dérivation qui doit être privilégiée (gauche ou droite). En pratique, certains terminaux se voit associer une précédence, avec éventuellement une indication d'associativité à gauche ou à droite. Deux terminaux avec la même précédence ont la même associativité (ou absence d'associativité). La précédence et l'associativité d'une production est alors celle de son dernier terminal (mais elle peut être surchargée à l'aide du mot clef `%prec`). Lors d'un conflit décaler/réduire, on essaie de comparer la précédence de la production à utiliser pour réduire avec celle du lexème à décaler. Si l'une des précédences est plus grande que l'autre, on choisit l'action correspondante. Ainsi, on réduit si la précédence de la règle est plus grande, et on décale si celle du terminal est plus grande. Si les précédences sont égales, on utilise l'associativité, en réduisant si l'associativité est à gauche, en décalant si elle est à droite. Si l'associativité ou la précédence ne sont pas définies, le conflit est maintenu.

▷ **Question 2.2 :**
Modifier la grammaire à l'aide priorité pour obtenir la reconnaissance usuelle des mathématiques (la multiplication d'abord).

```
%{
```

```
 open Expr
%}
%token EOF PLUS TIMES
%token <string> IDENT
%start expression
%type < Expr.expression > expression
%left PLUS
%left TIMES
%%
expression:
 | e EOF     { $1 }
e:
  | e PLUS e   { Plus($1,$3)}
  | e TIMES e  { Times($1,$3)}
  | IDENT     { Var $1 }
%%
```

2.2 Modification de la grammaire

De façon plus générale, il est possible de guider le choix en cas d'ambigüité en modifiant la grammaire. Par exemple, la grammaire de l'exercice 1 permet également de guider la reconnaissance et donc de résoudre les conflits.

Pour résoudre les conflits réduction / réduction la méthode des précédences ne peut pas être utilisée, il faut donc revoir la grammaire.

To be continued...