École Normale Supérieure

Langages de programmation et compilation

Jean-Christophe Filliâtre

analyse syntaxique (2/2)

résumé de l'épisode précédent

- notion de grammaire
  - dérivation
  - grammaire ambiguë
  - NULL, FIRST, FOLLOW
    - calcul de point fixe

$$
\begin{array}{rcl}
E & \rightarrow & E + E \\
 & | & E * E \\
 & | & (E) \\
 & | & \texttt{int}
\end{array}
$$

- analyse ascendante
  - une table nous permet de choisir entre lecture et réduction
  - grammaires LR(0), SLR(1), LALR(1), LR(1)
  - outils de type `yacc`

plan

1. localisations
2. analyse syntaxique élémentaire
3. analyse descendante
4. indentation comme syntaxe

l'outil `ocamllex` maintient, dans la structure de type `Lexing.lexbuf`, la **position** courante dans le texte source qui est analysé (nom du fichier, ligne, colonne)

```
type position = ...
```

on peut obtenir la localisation de la dernière chaîne reconnue par `ocamllex`

```
val lexeme_start_p: lexbuf -> position
val lexeme_end_p  : lexbuf -> position
```

on utilise cette information pour localiser une erreur de syntaxe

```
let lb = Lexing.from_channel c in
try
  let f = Parser.file Lexer.next_token lb in
  ...
with
| Parser.Error ->
    let pos = Lexing.lexeme_start_p lb in
    eprintf "%d: syntax error" pos.pos_lnum;
    exit 1
```

(voir le code fourni au TD 2)

mais aussi possiblement des erreurs lexicales comme une chaîne ou un commentaire non fermé

l'outil `menhir` récupère cette information et la fournit dans deux valeurs `$startpos` et `$endpos` du type `Lexing.position`

dans une action sémantique, elles correspondent au début et à la fin du texte qui a été reconnu par la règle de grammaire

on peut stocker cette information dans l'arbre de syntaxe abstraite

```
expression:
| e1 = expression; PLUS; e2 = expression
    { Add ($startpos, $endpos, e1, e2) }
```

(cf cours 4 pour une solution plus élégante encore)

analyse syntaxique élémentaire

un bon exercice

écrivons, de la façon la plus élémentaire possible, un analyseur syntaxique pour des expressions arithmétiques incluant

- des constantes
- des additions
- des multiplications
- des parenthèses

plus précisément

point de départ : un analyseur lexical (par exemple écrit avec `ocamllex`)

```
type token =
  | CONST of int
  | PLUS
  | TIMES
  | LEFTPAR
  | RIGHTPAR
  | EOF
```

point d'arrivée : un arbre de syntaxe abstraite

```
type expr =
  | Const of int
  | Add   of expr * expr
  | Mul   of expr * expr
```

commencer par écrire une **fonction d'affichage** (*pretty-printer*)

```
let rec print fmt = function
| Add (e1, e2) -> fprintf fmt "%a +@ %a" print e1 print e2
| e            -> print2 fmt e

and print2 fmt = function
| Mul (e1, e2) -> fprintf fmt "%a *@ %a" print2 e1 print2 e2
| e            -> print3 fmt e

and print3 fmt = function
| Const n -> fprintf fmt "%d" n
| e       -> fprintf fmt "(@[%a@])" print e
```

(ici avec la bibliothèque `Format`)

l'analyseur syntaxique suit la même structure que la fonction d'affichage

```
let rec parse_expr () =
  let e = parse_term () in
  if !t = PLUS then (next (); Add (e, parse_expr ())) else e
and parse_term () =
  let e = parse_factor () in
  if !t = TIMES then (next (); Mul (e, parse_term ())) else e
and parse_factor () = match !t with
  | CONST n -> next (); Const n
  | LEFTPAR -> next ();
      let e = parse_expr () in
      if !t <> RIGHTPAR then error ();
      next (); e
  | _ -> error ()
```

(code complet sur la page du cours)

- on pourrait inclure l'analyse lexicale dans un tel code, avec d'autres fonctions récursives pour lire les constantes entières, ignorer les blancs, etc.

- pour des opérateurs associatifs **à gauche** (par ex. la soustraction), le code sera légèrement différent mais le principe reste le même

analyse descendante

idée : procéder par expansions successives du non terminal le plus à gauche (on construit donc une dérivation gauche) en partant de S et en se servant d'une **table** indiquant, pour un non terminal X à expanser et les k premiers caractères de l'entrée, l'expansion $X \to \beta$ à effectuer
(en anglais on parle de *top-down parsing*)

supposons $k = 1$ par la suite et notons $T(X, c)$ cette table

en pratique on suppose qu'un symbole terminal $\#$ dénote la fin de l'entrée, et la table indique donc également l'expansion de X lorsque l'on atteint la fin de l'entrée

on utilise une pile qui est un mot de $(N \cup T)^\star$; initialement la pile est réduite au symbole de départ

à chaque instant, on examine le sommet de la pile et le premier caractère c de l'entrée

- si la pile est vide, on s'arrête ; il y a succès si et seulement si c est $\#$
- si le sommet de la pile est un terminal a, alors a doit être égal à c, on dépile a et on consomme c ; sinon on échoue
- si le sommet de la pile est un non terminal X, alors on remplace X par le mot $\beta = T(X, c)$ en sommet de pile, le cas échéant, en empilant les caractères de β en partant du dernier ;
  sinon, on échoue

prenons cette grammaire des expressions arithmétiques
et considérons la table d'expansion suivante

$$
\begin{array}{rcl}
E & \to & T\ E' \\
E' & \to & \texttt{+}\ T\ E' \\
 & | & \epsilon \\
T & \to & F\ T' \\
T' & \to & \texttt{*}\ F\ T' \\
 & | & \epsilon \\
F & \to & \texttt{(}\ E\ \texttt{)} \\
 & | & \texttt{int}
\end{array}
$$

	+	*	(	)	int	#
E			TE'		TE'	
E'	$+TE'$			ϵ		ϵ
T			FT'		FT'	
T'	ϵ	$*FT'$		ϵ		ϵ
F			(E)		int	

(on verra plus loin comment construire cette table)

illustrons l'analyse descendante du mot

`int + int * int`

	+	*	(	)	int	#
E			TE'		TE'	
E'	$+TE'$			ϵ		ϵ
T			FT'		FT'	
T'	ϵ	$*FT'$		ϵ		ϵ
F			(E)		int	

pile	entrée
E	int+int*int#
$E'T$	int+int*int#
$E'T'F$	int+int*int#
$E'T'$int	int+int*int#
$E'T'$	+int*int#
E'	+int*int#
$E'T+$	+int*int#
$E'T$	int*int#
$E'T'F$	int*int#
$E'T'$int	int*int#
$E'T'$	*int#
$E'T'F*$	*int#
$E'T'F$	int#
$E'T'$int	int#
$E'T'$	#
E'	#
ϵ	#

programmation d'un analyseur descendant

un analyseur descendant se programme très facilement en introduisant une fonction pour chaque non terminal de la grammaire

chaque fonction examine l'entrée et, selon le cas, la consomme ou appelle récursivement les fonctions correspondant à d'autres non terminaux, selon la table d'expansion

(c'est ce que nous avons fait au début du cours)

programmation d'un analyseur descendant

faisons le choix d'une programmation purement applicative, pour changer, où l'entrée est une liste de lexèmes du type

```
type token = Tplus | Tmult | Tleft | Tright | Tint | Teof
```

on va donc construire cinq fonctions qui ≪ consomment ≫ la liste d'entrée

```
val e : token list -> token list
val e': token list -> token list
val t : token list -> token list
val t': token list -> token list
val f : token list -> token list
```

et la reconnaissance d'une entrée pourra alors se faire ainsi

```
let accept l =
  e l = [Teof]
```

les fonctions procèdent par filtrage sur l'entrée et suivent la table

	+	*	(	)	int	#
E			TE'		TE'	

```
let rec e = function
  | (Tleft | Tint) :: _ as m -> e' (t m)
  | _ -> error ()
```

	+	*	(	)	int	#
E'	$+TE'$			ϵ		ϵ

```
and e' = function
  | Tplus :: m -> e' (t m)
  | (Tright | Teof) :: _ as m -> m
  | _ -> error ()
```

	+	*	(	)	int	#
T			FT'		FT'	

```
and t = function
  | (Tleft | Tint) :: _ as m -> t' (f m)
  | _ -> error ()
```

	+	*	(	)	int	#
T'	ϵ	$*FT'$		ϵ		ϵ

```
and t' = function
  | (Tplus | Tright | Teof) :: _ as m -> m
  | Tmult :: m -> t' (f m)
  | _ -> error ()
```

	+	*	(	)	int	#
F			(E)		int	

```
and f = function
  | Tint :: m -> m
  | Tleft :: m -> begin match e m with
      | Tright :: m -> m
      | _ -> error ()
    end
  | _ -> error ()
```

programmation d'un analyseur descendant

remarques

- la table d'expansion n'est pas explicite : elle est dans le code de chaque fonction
- la pile non plus n'est pas explicite : elle est réalisée par la pile d'appels
- on aurait pu les rendre explicites

- en pratique, il faut aussi construire l'arbre de syntaxe abstraite

reste une question d'importance : comment construire la table ?

l'idée est simple : pour décider si on réalise l'expansion $X \to \beta$ lorsque le premier caractère de l'entrée est c, on va chercher à déterminer si c fait partie des **premiers** caractères des mots reconnus par β

une difficulté se pose pour une production telle que $X \to \epsilon$, et il faut alors considérer aussi l'ensemble des caractères qui peuvent **suivre** X

on a justement vu la semaine dernière comment calculer FIRST et FOLLOW

à l'aide des premiers et des suivants, on construit la table d'expansion $T(X, a)$ de la manière suivante

pour chaque production $X \to \beta$,

- on pose $T(X, a) = \beta$ pour tout $a \in \text{FIRST}(\beta)$
- si $\text{NULL}(\beta)$, on pose aussi $T(X, a) = \beta$ pour tout $a \in \text{FOLLOW}(X)$

$$
\begin{array}{lcl}
E & \rightarrow & T\ E' \\
E' & \rightarrow & \texttt{+}\ T\ E' \\
 & | & \epsilon \\
T & \rightarrow & F\ T' \\
T' & \rightarrow & \texttt{*}\ F\ T' \\
 & | & \epsilon \\
F & \rightarrow & \texttt{(}\ E\ \texttt{)} \\
 & | & \texttt{int}
\end{array}
$$

FIRST

E	E'	T	T'	F
{(, int}	{+}	{(, int}	{*}	{(, int}

FOLLOW

E	E'	T	T'	F
{#,)}	{#,)}	{+, #,)}	{+, #,)}	{*, +, #,)}

	+	*	(	)	int	#
E			TE'		TE'	
E'	$+TE'$			ϵ		ϵ
T			FT'		FT'	
T'	ϵ	$*FT'$		ϵ		ϵ
F			(E)		int	

Définition (grammaire LL(1))

Une grammaire est dite LL(1) si, dans la table précédente, il y a au plus une production dans chaque case.

LL signifie « **L**eft to right scanning, **L**eftmost derivation »

exemple : la grammaire du transparent précédent est LL(1)

$$
\begin{array}{lcl}
E & \rightarrow & E + T \\
 & | & T \\
T & \rightarrow & T * F \\
 & | & F \\
F & \rightarrow & (E) \\
 & | & \texttt{int}
\end{array}
$$

FIRST

E	T	F
$\{(, \texttt{int}\}$	$\{(, \texttt{int}\}$	$\{(, \texttt{int}\}$

	+	*	(	)	int	#
E			$E+T/T$		$E+T/T$	
T			$T*F/F$		$T*F/F$	
F			(E)		int	

une grammaire **récursive gauche**, *i.e.* contenant des productions de la forme

$$\begin{array}{rcl} X & \rightarrow & X\alpha \\ & | & \beta \end{array}$$

ne sera jamais LL(1)

en effet, les FIRST seront les mêmes pour ces deux productions (quel que soit le mot β)

il faut supprimer la récursion gauche, par exemple comme ceci

$$\begin{array}{rcl} X & \rightarrow & \beta\, X' \\ X' & \rightarrow & \alpha\, X' \\ & | & \epsilon \end{array}$$

si une grammaire contient

$$\begin{array}{rcl} X & \rightarrow & \mathtt{a}\,\alpha \\ & | & \mathtt{a}\,\beta \end{array}$$

elle ne sera pas LL(1)

il faut factoriser les productions qui commencent par le même terminal (factorisation gauche)

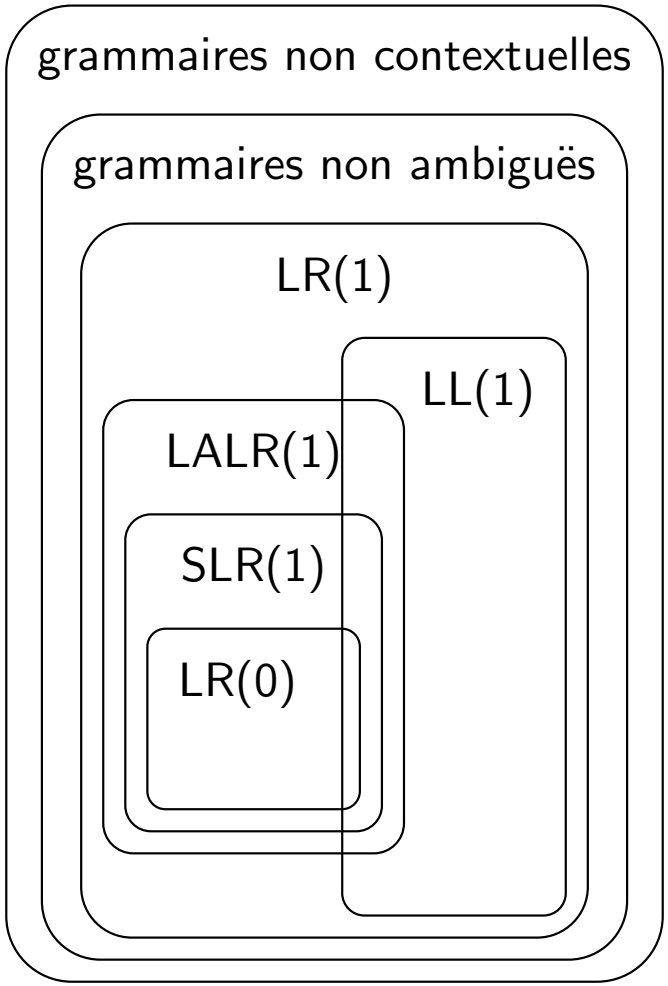
$$\begin{array}{rcl} X & \rightarrow & \mathtt{a}\, X' \\ X' & \rightarrow & \alpha \\ & | & \beta \end{array}$$

la grammaire « de LISP »

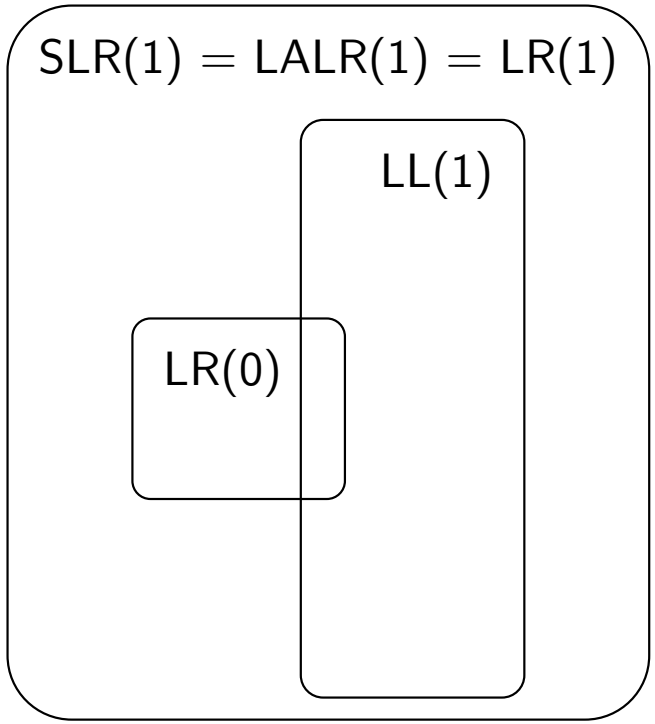
$$
\begin{array}{lcl}
S & \rightarrow & E\ \# \\
E & \rightarrow & \texttt{sym} \\
 & | & (\ L\ ) \\
L & \rightarrow & \epsilon \\
 & | & E\ L
\end{array}
$$

est-elle LL(1) ?

grammaires



langages



les analyseurs LL(1) sont relativement simples à écrire
(cf TD de cette semaine)

mais ils nécessitent d'écrire des grammaires peu naturelles

beaucoup de compilateurs utilisent une analyse descendante écrite à la main

exemples :

- `javac` ($\approx$ 3 kloc de code Java)
- `rustc` ($\approx$ 16 kloc de code Rust)
- `gcc` ($\approx$ 25 kloc de code C++)

indentation comme syntaxe

dans certains langages, l'**indentation** (blancs de début de ligne, alignement vertical) est utilisée pour définir la syntaxe

exemples :

- Python
- Haskell
- Purescript

l'idée n'est pas neuve (Landin, 1966)

en Python, l'indentation définit la structure de bloc

```
while q > 0:
    if q % 2 == 1:
        r += p
    p *= 2
    q //= 2
```

l'analyseur lexical introduit des lexèmes `NEWLINE` (fin de ligne), `INDENT` (quand l'indentation augmente) et `DEDENT` (quand elle diminue)

et la grammaire du langage les utilise

```
statement:
| IF expression COLON suite
| WHILE expression COLON suite
...
suite:
| statement NEWLINE
| NEWLINE INDENT statement+ DEDENT
...
```

en particulier, elle peut être écrite avec un outil de type yacc (cf TD 2)

la suite

- TD 5
  - analyseur LL(1)
  - aujourd'hui et **jeudi 9 novembre**

- prochain cours **vendredi 10 novembre**
  - typage