

École Normale Supérieure

Langages de programmation et compilation

Jean-Christophe Filliâtre

analyse syntaxique (2/2)

- notion de grammaire

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

$$\begin{array}{l} E \rightarrow E + E \\ \quad | E * E \\ \quad | (E) \\ \quad | \text{int} \end{array}$$

- 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

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 $X \rightarrow \beta$ 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 $(N \cup T)^*$; 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

$$\begin{array}{l}
 E \rightarrow T E' \\
 E' \rightarrow + T E' \\
 \quad | \epsilon \\
 T \rightarrow F T' \\
 T' \rightarrow * F T' \\
 \quad | \epsilon \\
 F \rightarrow (E) \\
 \quad | \text{int}
 \end{array}$$

| | + | * | (|) | int | # |
|-----------|--------------|--------------|--------------|------------|------------|------------|
| <i>E</i> | | | <i>TE'</i> | | <i>TE'</i> | |
| <i>E'</i> | + <i>TE'</i> | | | ϵ | | ϵ |
| <i>T</i> | | | <i>FT'</i> | | <i>FT'</i> | |
| <i>T'</i> | ϵ | * <i>FT'</i> | | ϵ | | ϵ |
| <i>F</i> | | | (<i>E</i>) | | 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 | # |

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]
```

programmation d'un analyseur descendant

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

| | + | * | (|) | int | # |
|----------|---|---|------------|---|------------|---|
| <i>E</i> | | | <i>TE'</i> | | <i>TE'</i> | |

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

| | + | * | (|) | int | # |
|-----------|--------------|---|---|---|-----|---|
| <i>E'</i> | + <i>TE'</i> | | | ε | | ε |

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

programmation d'un analyseur descendant

| | + | * | (|) | int | # |
|-----|---|---|-------|---|-------|---|
| T | | | FT' | | FT' | |

and $t = \text{function}$

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

| | + | * | (|) | int | # |
|------|------------|--------|---|------------|-----|------------|
| T' | ϵ | $*FT'$ | | ϵ | | ϵ |

and $t' = \text{function}$

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

programmation d'un analyseur descendant

| | | | | | | |
|----------|---|---|--------------|---|-----|---|
| | + | * | (|) | int | # |
| <i>F</i> | | | (<i>E</i>) | | int | |

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

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 $X \rightarrow \beta$ 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 $X \rightarrow \epsilon$, 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 $X \rightarrow \beta$,

- 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 $a \in \text{FOLLOW}(X)$

$E \rightarrow TE'$
 $E' \rightarrow +TE'$
 $\quad \mid \epsilon$
 $T \rightarrow FT'$
 $T' \rightarrow *FT'$
 $\quad \mid \epsilon$
 $F \rightarrow (E)$
 $\quad \mid \text{int}$

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)

| | | | | | | |
|-----|---------------|--------------|-------|-----------------------|-----------------------|-----------------------|
| E | \rightarrow | $E + T$ | FIRST | E | T | F |
| | | T | | $\{ (, \text{int} \}$ | $\{ (, \text{int} \}$ | $\{ (, \text{int} \}$ |
| T | \rightarrow | $T * F$ | | $\{ (, \text{int} \}$ | $\{ (, \text{int} \}$ | $\{ (, \text{int} \}$ |
| | | F | | $\{ (, \text{int} \}$ | $\{ (, \text{int} \}$ | $\{ (, \text{int} \}$ |
| F | \rightarrow | (E) | | | | |
| | | 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

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

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

$$\begin{array}{l} X \rightarrow \beta X' \\ X' \rightarrow \alpha X' \\ \quad | \quad \epsilon \end{array}$$

si une grammaire contient

$$\begin{array}{l} X \rightarrow a\alpha \\ \quad | a\beta \end{array}$$

elle ne sera pas LL(1)

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

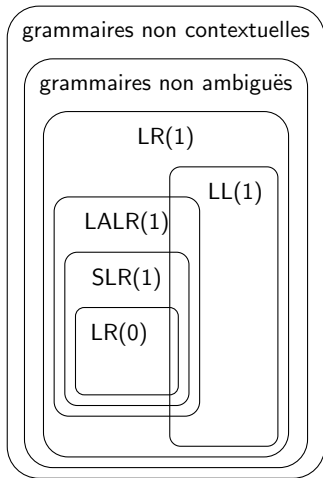
$$\begin{array}{l} X \rightarrow aX' \\ X' \rightarrow \alpha \\ \quad | \beta \end{array}$$

la grammaire « de LISP »

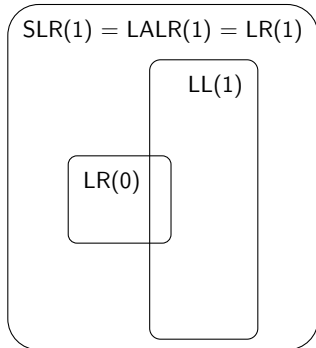
$$\begin{array}{l} S \rightarrow E \# \\ E \rightarrow \text{sym} \\ \quad | (L) \\ L \rightarrow \epsilon \\ \quad | E L \end{array}$$

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

- TD 5
 - analyseur LL(1)
 - aujourd'hui et **jeudi 10 novembre**

-  pas de cours vendredi 11 novembre
-  pas de TD vendredi 11 novembre et jeudi 17 novembre

- prochain cours **vendredi 18 novembre**
 - typage