

CS321 F'04 Lecture Notes
Lecture 8

Algorithm

(assuming \$ = EOF; S = start symbol)

```

push($); push(S);
repeat
  a ← input
  if top is a terminal or $ then
    if top = a then
      pop(); advance();
    else error();
  else if M[top,a] is X → Y1 Y2 ... Yk then
    pop();
    push(Yk); push(Yk-1); ...; push(Y1);
    /* do “semantic action” here */
  else error();
until top = $

```

“Semantic action” code is executed once for each step in the **left-most derivation** of an input sentence.

Table-driven Top-down Parsing

Recursive-descent parsers are highly stylized.

Can use single table-driven program instead

“Parsing table” is 2-dimensional table $M[X, a]$

- One entry for every non-terminal X and terminal a .
- Entries are productions or error indicators.
- Entry $M[X, a]$ says “what to do” when looking for non-terminal X while next input symbol is a .

“Parsing stack” handles recursion explicitly

- Holds “what’s left to match” (in reverse order)

Example

	id	+	*	()	\$
E	$E \rightarrow TE'$				$E \rightarrow TE'$	
E'		$E' \rightarrow +TE'$			$E' \rightarrow \epsilon$	$E' \rightarrow \epsilon$
T	$T \rightarrow FT'$				$T \rightarrow FT'$	
T'		$T' \rightarrow \epsilon$	$T' \rightarrow *FT'$		$T' \rightarrow \epsilon$	$T' \rightarrow \epsilon$
F	$F \rightarrow id$				$F \rightarrow (E)$	

Sample Execution:

Stack	Input	“Output”
$\$E$	id+id*id\$	
$\$E'T$	id+id*id\$	$E \rightarrow TE'$
$\$E'T'F$	id+id*id\$	$T \rightarrow FT'$
$\$E'T'id$	id+id*id\$	$F \rightarrow id$
$\$E'T'$	+id*id\$	
$\$E'$	+id*id\$	$T' \rightarrow \epsilon$
$\$E'T+$	+id*id\$	$E' \rightarrow +TE'$
$\$E'T$	id*id\$	
$\$E'T'F$	id*id\$	$T \rightarrow FT'$
$\$E'T'id$	id*id\$	$F \rightarrow id$
$\$E'T'$	*id\$	
$\$E'T'F*$	*id\$	$T' \rightarrow *FT'$
$\$E'T'F$	id\$	
$\$E'T'id$	id\$	$F \rightarrow id$
$\$E'T'$	\$	
$\$E'$	\$	$T' \rightarrow \epsilon$
$\$$	\$	$E' \rightarrow \epsilon$

Parsing Table Construction

$FIRST(\alpha)$ is the set of **terminals** (and possibly ϵ) that **begin** strings derived from α , where α is any string of grammar symbols (terminals or non-terminals). (Book defines $FIRST()$ only on individual symbols rather than strings of symbols; our definition is a consistent extension of the book's.)

$FOLLOW(A)$ is the set of **terminals** (possibly including $\$$) that can **follow** the **non-terminal** A in some **sentential form** (intermediate phrase in a derivation), i.e., the set of terminals

$$\{a \mid S \xRightarrow{*} \alpha A a \beta \text{ for some } \alpha, \beta\}$$

(This definition is equivalent to the book's. Note there is an erratum for Figure 3.5.)

Table Construction Algorithm

```

for each production  $A \rightarrow \alpha$  do
  for each  $a \in FIRST(\alpha)$  do
    add  $A \rightarrow \alpha$  to  $M[A, a]$ 
  if  $\epsilon \in FIRST(\alpha)$  then
    for each  $b \in FOLLOW(A)$  do
      add  $A \rightarrow \alpha$  to  $M[A, b]$ 
set any empty elements of  $M$  to error
    
```

Computing FOLLOW

Must compute simultaneously for all non-terminals A .

FOLLOW sets are **smallest** sets obeying these rules:

- $\$$ is in $FOLLOW(S)$
- **If** there is a production $A \rightarrow \alpha B \beta$, **then** everything in $FIRST(\beta) - \{\epsilon\}$ is in $FOLLOW(B)$.
- **If** there is a production $A \rightarrow \alpha B \beta$ where $\beta = \epsilon$ or $\epsilon \in FIRST(\beta)$, **then** everything in $FOLLOW(A)$ is in $FOLLOW(B)$.

Example

Computation	Relevant Production
$FOLLOW(E) = \{\$\}$	
$= \{\$\)\}$	$F \rightarrow (E)$
$FOLLOW(E') = FOLLOW(E) = \{\$\)\}$	$E \rightarrow TE'$
	(what about $E' \rightarrow +TE'$?)
$FOLLOW(T) = (FIRST(E') - \{\epsilon\})$	
$\cup FOLLOW(E)$	$E \rightarrow TE'$
$\cup FOLLOW(E')$	$E' \rightarrow +TE'$
$= \{+) \$\}$	
$FOLLOW(T') = FOLLOW(T) = \{+) \$\}$	$T \rightarrow FT'$
$FOLLOW(F) = (FIRST(T') - \{\epsilon\})$	
$\cup FOLLOW(T)$	$T \rightarrow FT'$
$\cup FOLLOW(T')$	$T' \rightarrow *FT'$
$= \{* +) \$\}$	

Computing FIRST

For any string of symbols α , $FIRST(\alpha)$ is the **smallest** set of terminals (and ϵ) obeying these rules:

$FIRST(a\alpha) = \{a\}$ for any terminal a
and **any** α (empty or non-empty)

$FIRST(\epsilon) = \{\epsilon\}$

$FIRST(A) = FIRST(\alpha_1) \cup FIRST(\alpha_2) \cup \dots \cup FIRST(\alpha_n)$
where $A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$
are all the productions for A

$FIRST(A\alpha) =$ if $\epsilon \notin FIRST(A)$ then $FIRST(A)$
else $(FIRST(A) - \{\epsilon\}) \cup FIRST(\alpha)$

Example

$FIRST(F) = FIRST((E)) \cup FIRST(id) = \{(id\}$
 $FIRST(T') = FIRST(*FT') \cup FIRST(\epsilon) = \{*\ \epsilon\}$
 $FIRST(T) = FIRST(FT') = FIRST(F) = \{(id\}$
 $FIRST(E') = FIRST(+TE') \cup FIRST(\epsilon) = \{+\ \epsilon\}$
 $FIRST(E) = FIRST(TE') = FIRST(T) = \{(id\}$

LL(1) Grammars

A grammar can be used to build a predictive table-driven parser \Leftrightarrow parsing table M has no duplicate entries.

In terms of FIRST and FOLLOW sets, this means that, for each production

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \dots \mid \alpha_n$$

- All $FIRST(\alpha_i)$ are disjoint, and
- There is at most **one** i such that $\epsilon \in FIRST(\alpha_i)$, and, if there is such an i , $FOLLOW(A) \cap FIRST(\alpha_j) = \emptyset$ for all $j \neq i$.

Such grammars are called **LL(1)**.

- the first **L** stands for “**L**eft-to-right scan of input.”
- the second **L** stands for “**L**eftmost derivation.”
- the **1** stands for “**1** token of lookahead.”

No LL(1) grammar can be ambiguous or left-recursive.