CS301 W'99 Lecture Notes
Lecture 8

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)

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 → Y_1 Y_2 ... Y_k then
    pop();
    push(Y_k); push(Y_{k-1}); ...; push(Y_1);
    /* do ``semantic action'' here */
  else error();
until top = $
```

"Semantic action" code is executed once for each step in the **leftmost derivation** of an input sentence.

Example

	`id`	`+`	`*`	`(`	`)`	`$`
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to \mathtt{id}$			$F \to (E)$		

Sample Execution:

Stack	Input	"Output"
$\$E$	`id+id*id$`	
$\$E'T$	`id+id*id$`	$E \to TE'$
$\$E'T'F$	`id+id*id$`	$T \to FT'$
$\$E'T'\mathtt{id}$	`id+id*id$`	$F \to \mathtt{id}$
$\$E'T'$	`+id*id$`	
$\$E'$	`+id*id$`	$T' \to \epsilon$
$\$E'T+$	`+id*id$`	$E' \to +TE'$
$\$E'T$	`id*id$`	
$\$E'T'F$	`id*id$`	$T \to FT'$
$\$E'T'\mathtt{id}$	`id*id$`	$F \to \mathtt{id}$
$\$E'T'$	`*id$`	
$\$E'T'F*$	`*id$`	$T' \to *FT'$
$\$E'T'F$	`id$`	
$\$E'T'\mathtt{id}$	`id$`	$F \to \mathtt{id}$
$\$E'T'$	`$`	
$\$E'$	`$`	$T' \to \epsilon$
$\$$	`$`	$E' \to \epsilon$

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).

$FOLLOW(A)$ 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

$\{a \mid S \stackrel{*}{\Rightarrow} \alpha A a \beta \quad \text{for some } \alpha,\beta\ \}$

Table Construction Algorithm

```
for each production A → α do
  for each a ∈ FIRST(α) do
    add A → α to M[A, a]
  if ε ∈ FIRST(α) then
    for each b ∈ FOLLOW(A) do
      add A → α to M[A, b]
set any empty elements of M to error
```


Computing FIRST

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

$$
\begin{aligned}
FIRST(a\alpha) &= \{a\} \text{ for any terminal } a \\
&\quad \text{and } \textbf{any } \alpha \text{ (empty or non-empty)} \\
FIRST(\epsilon) &= \{\epsilon\} \\
FIRST(A) &= FIRST(\alpha_1) \cup FIRST(\alpha_2) \cup \ldots \cup FIRST(\alpha_n) \\
&\quad \text{where } A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \\
&\quad \text{are all the productions for } A \\
FIRST(A\alpha) &= \text{if } \epsilon \notin FIRST(A) \text{ then } FIRST(A) \\
&\quad \text{else } (FIRST(A) - \{\epsilon\}) \cup FIRST(\alpha)
\end{aligned}
$$

Example

$$
\begin{aligned}
FIRST(F) &= FIRST(\texttt{(}E\texttt{)}) \cup FIRST(\texttt{id}) &= \{\texttt{(id}\} \\
FIRST(T') &= FIRST(\texttt{*}FT') \cup FIRST(\epsilon) &= \{\texttt{*}\ \epsilon\} \\
FIRST(T) &= FIRST(FT') = FIRST(F) &= \{\texttt{(id}\} \\
FIRST(E') &= FIRST(\texttt{+}TE') \cup FIRST(\epsilon) &= \{\texttt{+}\ \epsilon\} \\
FIRST(E) &= FIRST(TE') = FIRST(T) &= \{\texttt{(id}\}
\end{aligned}
$$

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

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

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

- 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.