CS321 Languages and Compiler Design I

Winter 2012

Lecture 8

Table-driven Top-down Parsing

Recursive-descent parsers are highly stylized.

Can use single table-driven program instead, using two data structures:

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 the input (in reverse order)

Table-driven Parsing 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 Table and Execution

Recall arithmetic expression grammar (after left-recursion removal):

$$
\begin{array}{lcl}
E & \rightarrow & TE' \\
E' & \rightarrow & \texttt{+}\ TE' \mid \epsilon \\
T & \rightarrow & FT' \\
T' & \rightarrow & \texttt{*}\ FT' \mid \epsilon \\
F & \rightarrow & \texttt{(}E\texttt{)} \mid \texttt{id}
\end{array}
$$

The corresponding parsing table is:

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

and a sample execution is...

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

PARSING TABLE CONSTRUCTION

FIRST (α) 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 1st ed. defines *FIRST* $()$ only on individual symbols rather than strings of symbols; our definition is a consistent extension of this.)

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

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

(This definition is equivalent to the book's. Note that there is an erratum for 1st edition Figure 3.5.)

TABLE CONSTRUCTION ALGORITHM

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

COMPUTING FIRST

For any string of symbols α, *FIRST* (α) 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 (ϵ) = $\{\epsilon\}$

FIRST (A) = *FIRST* $(\alpha_1) \cup$ *FIRST* $(\alpha_2) \cup \ldots \cup$ *FIRST* (α_n)
where $A \to \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n$
are all the productions for A

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

Example FIRST computation

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

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*(β), **then** everything in *FOLLOW*(A) is in *FOLLOW*(B).

Example FOLLOW computation

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*(α_i) are disjoint, and
- There is at most **one** i such that $\epsilon \in$ *FIRST*(α_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.