Parsers (cont.)

Depth First Traversal is suitable for top down parsing, but left recursive grammar can create infinite loop.

Left Recursion

- A grammar is ***left recursive*** if it has a non-terminal A such that there is a derivation.

  $A \overset{+}{\Rightarrow} A\alpha$ for some string α

- Top-down parsing techniques **cannot** handle left-recursive grammars.
- So, we have to convert our left-recursive grammar into an equivalent grammar which is not left-recursive.
- The left-recursion may appear in a single step of the derivation (*immediate left-recursion*), or may appear in more than one step of the derivation.

Immediate Left-Recursion

$A \rightarrow A\,\alpha \mid \beta$ where β does not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta\,A'$

$A' \rightarrow \alpha\,A' \mid \varepsilon$ an equivalent grammar

In general,

$A \rightarrow A\,\alpha_1 \mid \ldots \mid A\,\alpha_m \mid \beta_1 \mid \ldots \mid \beta_n$ where $\beta_1 \ldots \beta_n$ do not start with A

$\Downarrow$ eliminate immediate left recursion

$A \rightarrow \beta_1\,A' \mid \ldots \mid \beta_n\,A'$

$A' \rightarrow \alpha_1\,A' \mid \ldots \mid \alpha_m\,A' \mid \varepsilon$ an equivalent grammar

Immediate Left-Recursion -- Example

$E \rightarrow E+T \mid T$

$T \rightarrow T*F \mid F$

$F \rightarrow id \mid (E)$

$\Downarrow$ eliminate immediate left recursion

$E \rightarrow T E'$

$E' \rightarrow +T E' \mid \varepsilon$

$T \rightarrow F T'$

$T' \rightarrow *F T' \mid \varepsilon$

$F \rightarrow id \mid (E)$

Left-Recursion -- Problem

- A grammar cannot be immediately left-recursive, but it still can be left-recursive.
- By just eliminating the immediate left-recursion, we may not get a grammar which is not left-recursive.

$S \rightarrow Aa \mid b$

$A \rightarrow Sc \mid d$ This grammar is not immediately left-recursive, but it is still left-recursive.

$\underline{S} \Rightarrow Aa \Rightarrow \underline{S}ca$ or

$\underline{A} \Rightarrow Sc \Rightarrow \underline{A}ac$ causes to a left-recursion

- So, we have to eliminate all left-recursions from our grammar

Eliminate Left-Recursion -- Algorithm

- Arrange non-terminals in some order: $A_1 \dots A_n$

- for i **from** 1 **to** n **do** {

  - for j **from** 1 **to** i-1 **do** {

    replace each production

    $$A_i \rightarrow A_j \gamma$$

    by

    $$A_i \rightarrow \alpha_1 \gamma \mid \dots \mid \alpha_k \gamma$$

    where $A_j \rightarrow \alpha_1 \mid \dots \mid \alpha_k$

  }

  - eliminate immediate left-recursions among A_i productions

}

Eliminate Left-Recursion -- Example

$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid f$

- Order of non-terminals: S, A

for S:

- we do not enter the inner loop.
- there is no immediate left recursion in S.

for A:

- Replace $A \rightarrow Sd$ with $A \rightarrow Aad \mid bd$
  So, we will have $A \rightarrow Ac \mid Aad \mid bd \mid f$
- Eliminate the immediate left-recursion in A

$$A \rightarrow bdA' \mid fA'$$
$$A' \rightarrow cA' \mid adA' \mid \varepsilon$$

So, the resulting equivalent grammar which is not left-recursive is:

$S \rightarrow Aa \mid b$
$A \rightarrow bdA' \mid fA'$
$A' \rightarrow cA' \mid adA' \mid \varepsilon$

Eliminate Left-Recursion – Example2

$S \rightarrow Aa \mid b$
$A \rightarrow Ac \mid Sd \mid f$

- Order of non-terminals: A, S

for A:

- we do not enter the inner loop.
- Eliminate the immediate left-recursion in A

$$A \rightarrow SdA' \mid fA'$$
$$A' \rightarrow cA' \mid \varepsilon$$

for S:

- Replace $S \rightarrow Aa$ with $S \rightarrow SdA'a \mid fA'a$
  So, we will have $S \rightarrow SdA'a \mid fA'a \mid b$
- Eliminate the immediate left-recursion in S

$$S \rightarrow fA'aS' \mid bS'$$
$$S' \rightarrow dA'aS' \mid \varepsilon$$

So, the resulting equivalent grammar which is not left-recursive is:

$$S \rightarrow fA'aS' \mid bS'$$
$$S' \rightarrow dA'aS' \mid \varepsilon$$
$$A \rightarrow SdA' \mid fA'$$
$$A' \rightarrow cA' \mid \varepsilon$$

Left-Factoring

- A predictive parser (a top-down parser without backtracking) insists that the grammar must be *left-factored*.

  grammar ➔ a new equivalent grammar suitable for predictive parsing

stmt → `if` expr `then` stmt `else` stmt |
`if` expr `then` stmt

- when we see `if`, we cannot now which production rule to choose to re-write *stmt* in the derivation.

Left-Factoring (cont.)

- In general,

  $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ where α is non-empty and the first symbols of β_1 and β_2 (if they have one)are different.

- when processing α we cannot know whether expand

  A to $\alpha\beta_1$ or

  A to $\alpha\beta_2$

- But, if we re-write the grammar as follows

  $A \rightarrow \alpha A'$

  $A' \rightarrow \beta_1 \mid \beta_2$ so, we can immediately expand A to $\alpha A'$

Left-Factoring -- Algorithm

- For each non-terminal A with two or more alternatives (production rules) with a common non-empty prefix, let say

$$A \rightarrow \alpha\beta_1 \mid ... \mid \alpha\beta_n \mid \gamma_1 \mid ... \mid \gamma_m$$

convert it into

$$A \rightarrow \alpha A' \mid \gamma_1 \mid ... \mid \gamma_m$$
$$A' \rightarrow \beta_1 \mid ... \mid \beta_n$$

Left-Factoring – Example1

$A \rightarrow \underline{a}bB \mid \underline{a}B \mid cdg \mid cdeB \mid cdfB$

$\Downarrow$

$A \rightarrow aA' \mid \underline{cd}g \mid \underline{cd}eB \mid \underline{cd}fB$

$A' \rightarrow bB \mid B$

$\Downarrow$

$A \rightarrow aA' \mid cdA''$

$A' \rightarrow bB \mid B$

$A'' \rightarrow g \mid eB \mid fB$

Left-Factoring – Example2

$A \rightarrow ad \mid a \mid ab \mid abc \mid b$

$$\Downarrow$$

$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid b \mid bc$

$$\Downarrow$$

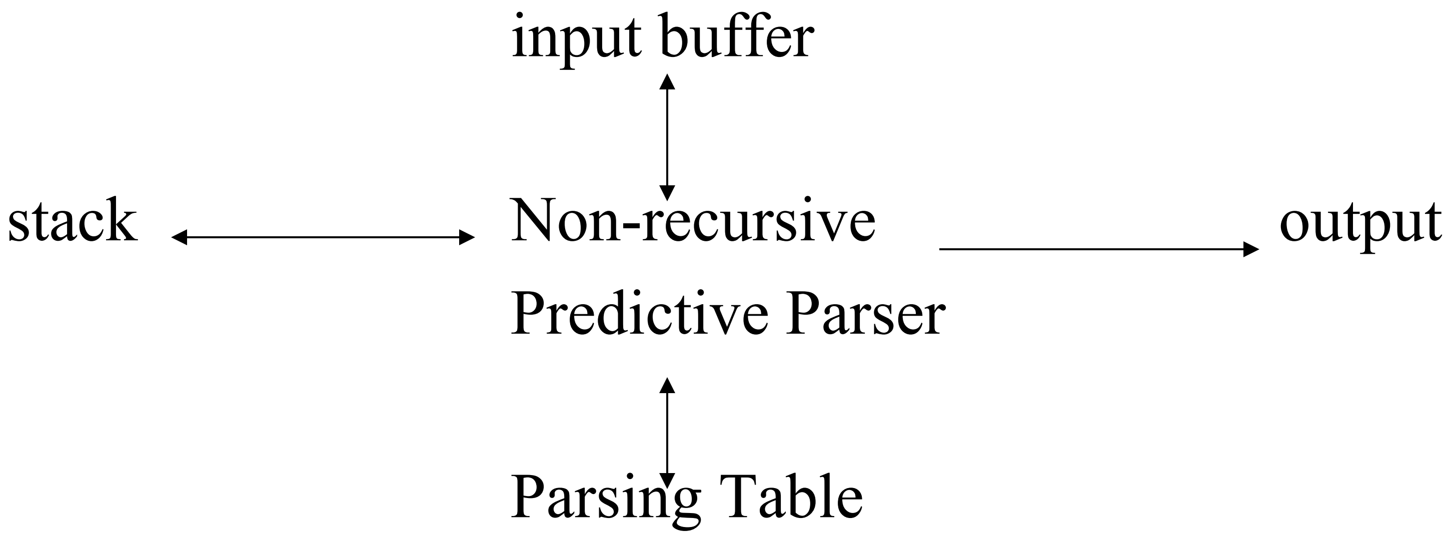
$A \rightarrow aA' \mid b$

$A' \rightarrow d \mid \varepsilon \mid bA''$

$A'' \rightarrow \varepsilon \mid c$

A Simple Non-Recursive Predictive Parser: **LL(1)**

- Top-down, predictive parsing: a table-driven parser.
  - **L**: Left-to-right scan of the tokens
  - **L**: Leftmost derivation.
  - **(1)**: One token of lookahead



LL(1) Parser

input buffer

- our string to be parsed. We will assume that its end is marked with a special symbol $.

output

- a production rule representing a step of the derivation sequence (left-most derivation) of the string in the input buffer.

stack

- contains the grammar symbols
- at the bottom of the stack, there is a special end marker symbol $.
- initially the stack contains only the symbol $ and the starting symbol S. $S ← initial stack
- when the stack is emptied (ie. only $ left in the stack), the parsing is completed.

parsing table

- a two-dimensional array M[A,a]
- each row is a non-terminal symbol
- each column is a terminal symbol or the special symbol $
- each entry holds a production rule.

LL(1) Parser – Parser Actions

- The symbol at the top of the stack (say X) and the current symbol in the input string (say a) determine the parser action.
- There are four possible parser actions.

1. If X and a are \$ ➔ parser halts (successful completion)

2. If X and a are the same terminal symbol (different from \$)

   ➔ parser pops X from the stack, and moves the next symbol in the input buffer.

3. If X is a non-terminal

   ➔ parser looks at the parsing table entry M[X,a]. If M[X,a] holds a production rule $X \rightarrow Y_1 Y_2 \ldots Y_k$, it pops X from the stack and pushes $Y_k, Y_{k-1}, \ldots, Y_1$ into the stack. The parser also outputs the production rule $X \rightarrow Y_1 Y_2 \ldots Y_k$ to represent a step of the derivation.

4. none of the above ➔ error
   - all empty entries in the parsing table are errors.
   - If X is a terminal symbol different from a, this is also an error case.

LL(1) Parser – Example1

$S \rightarrow aBa$

$B \rightarrow bB \mid \varepsilon$

	a	b	$
S	$S \rightarrow aBa$		
B	$B \rightarrow \varepsilon$	$B \rightarrow bB$	

LL(1) Parsing Table

stack	input	output
$S	abba$	$S \rightarrow aBa$
$aBa	abba$	
$aB	bba$	$B \rightarrow bB$
$aBb	bba$	
$aB	ba$	$B \rightarrow bB$
$aBb	ba$	
$aB	a$	$B \rightarrow \varepsilon$
$a	a$	
$	$	accept, successful completion

LL(1) Parser – Example2

$E \rightarrow TE'$
$E' \rightarrow +TE' \;|\; \varepsilon$
$T \rightarrow FT'$
$T' \rightarrow *FT' \;|\; \varepsilon$
$F \rightarrow (E) \;|\; id$

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

LL(1) Parser – Example2

stack	input	output
\$E	id+id\$	$E \to TE'$
\$E'T	id+id\$	$T \to FT'$
\$E' T'F	id+id\$	$F \to id$
\$ E' T'id	id+id\$	
\$ E' T'	+id\$	$T' \to \varepsilon$
\$ E'	+id\$	$E' \to +TE'$
\$ E' T+	+id\$	
\$ E' T	id\$	$T \to FT'$
\$ E' T' F	id\$	$F \to id$
\$ E' T'id	id\$	
\$ E' T'	\$	$T' \to \varepsilon$
\$ E'	\$	$E' \to \varepsilon$
\$	\$	accept