Unit 7
Predictive Parsing

Predictive Parsers

- Parser can “predict” which production to use
  - By looking at the next few tokens
  - No backtracking
- Predictive parsers accept LL(k) grammars
  - L means “left-to-right” scan of input
  - L means “leftmost derivation”
  - k means “predict based on k tokens of lookahead”
- In practice, LL(1) is used

- accomplished using a *predictive parsing table* M and a stack.

A stringent condition

The grammar must not be left recursive and no two right sides of a production have a common prefix.

Left Recursion

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

$A \Rightarrow A\alpha$ 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

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

$\Downarrow$ eliminate immediate left recursion

$A \to \beta\, A'$

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

In general,

$A \to A\,\alpha_1 \mid ... \mid A\,\alpha_m \mid \beta_1 \mid ... \mid \beta_n$ where $\beta_1 ... \beta_n$ do not start with A

$\Downarrow$ eliminate immediate left recursion

$A \to \beta_1\, A' \mid ... \mid \beta_n\, A'$

$A' \to \alpha_1\, A' \mid ... \mid \alpha_m\, A' \mid \varepsilon$ an equivalent grammar

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: $A_1 \ldots A_n$

- 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

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

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

}

- eliminate immediate left-recursions among A_i productions

}

Immediate Left-Recursion -- Example

$E \to E{+}T \mid T$

$T \to T{*}F \mid F$

$F \to id \mid (E)$

$\Downarrow$ eliminate immediate left recursion

$E \to T\,E'$

$E' \to +T\,E' \mid \varepsilon$

$T \to F\,T'$

$T' \to *F\,T' \mid \varepsilon$

$F \to id \mid (E)$

Predictive Parsing and Left Factoring

- In the grammar

  $E \to T + E \mid T$

  $T \to \text{int} \mid \text{int} * T \mid (E)$

- Hard to predict because
  - For T two productions start with int
  - For E it is not clear how to predict

- A grammar must be <u>left-factored</u> before use for predictive parsing

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

If_stmt $\rightarrow$ `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 (con'd)

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

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

convert it into

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

Left Factoring example

S --> if E then S | if E then S else S

can be rewritten as

S --> if E then S S’

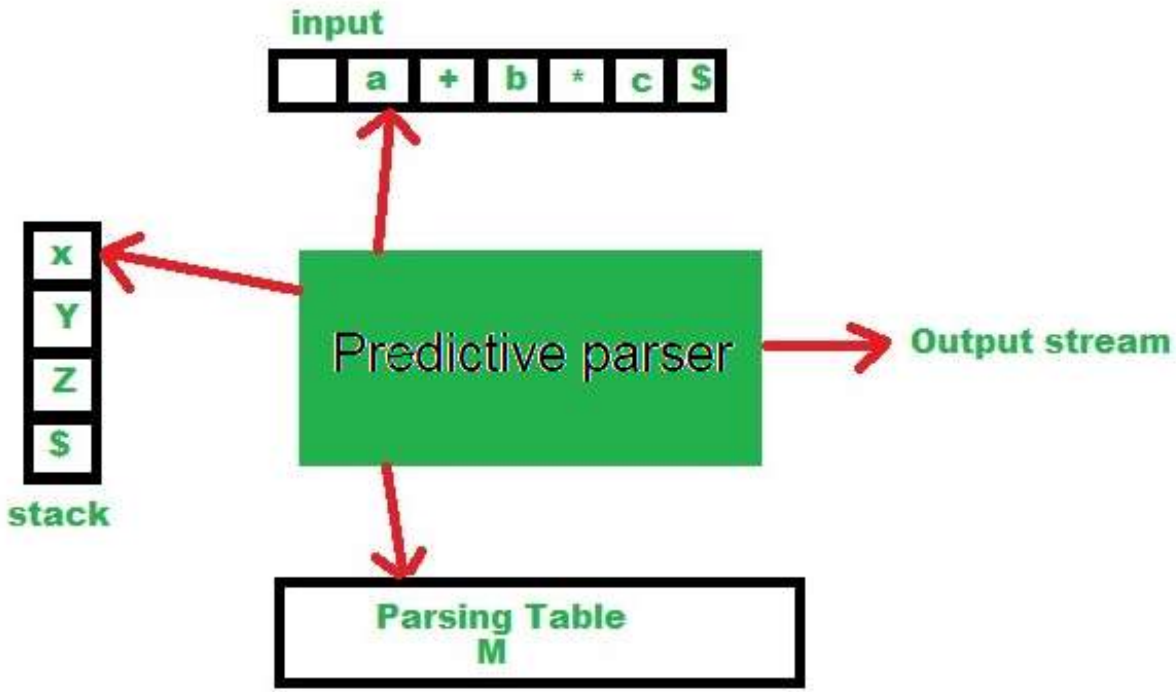
S’ --> else S | ε

In KPL

```
IfSt ::= KW_IF Condition KW_THEN Statement
 ElseSt

ElseSt ::= KW_ELSE Statement
ElseSt ::= ε
```

(Non recursive) Predictive parser



Parsing table M

- *M*[*X*, *token*] indicates which production to use if the top of the stack is a nonterminal *X* and the current token is equal to *token*;
- in that case we pop *X* from the stack and we push all the rhs symbols of the production *M*[*X*, *token*] in reverse order.
- We use a special symbol $ to denote the end of file. Let *S* be the start symbol

Non recursive Predictive Parser

The input contains the string to be parsed, followed by $ (EOF)
The stack contains a sequence of grammar symbols, preceded by #, the bottom-of-stack marker.
Initially the stack contains the start symbol of the grammar preceded by $.
The parsing table is a two dimensional array M[A,a], where A is a nonterminal, and a is a terminal or the symbol $.

- The parser is controlled by a program that behaves as follows:
- The program determines *X*, the symbol on top of the stack, and a, the current input symbol.
- These two symbols determine the action of the parser.

There are three possibilities:

1. If X = a = $, the parser halts and announces successful completion of parsing.
2. If X = a ≠ $, the parser pops X off the stack and advances the input pointer to the next input symbol.
3. If X is a nonterminal, the program consults entry M[X,a] of the parsing table M. This entry will be either an X-production of the grammar or an error entry.

If M[X,a] = {$X \rightarrow UVW$}, the parser replaces X on top of the stack by WVU (with U on top).
If M[X,a] = error, the parser calls an error recovery routine.

Parsing table for grammar $S \rightarrow aSb|c$

	a	b	c	$
S	$S \rightarrow aSb$	Error	$S \rightarrow c$	Error
a	Push	Error	Error	Error
b	Error	Push	Error	Error
c	Error	Error	Push	Error
#	Error	Error	Error	Accept

LL(1) Parsing Tables. Errors

- Yellow entries indicate error situations
  - Consider the [S,b] entry
  - “There is no way to derive a string starting with b from non-terminal S

Using Parsing Tables

- Method similar to recursive descent, except
  - For each non-terminal S
  - We look at the next token a
  - And chose the production shown at [S,a]
- We use a stack to keep track of pending non-terminals
- We reject when we encounter an error state
- We accept when we encounter end-of-input

LL(1) Parsing Algorithm

```
initialize stack = <S #>
repeat
   case stack of
      <X, rest>  : if T[X,*next] = T → Y1…Yn
                             then stack ← <Y1… Yn rest>;
                             else  error ();   //X-nonterminal
      <t, rest>   : if t == *next ++
                             then  stack ← <rest>;
                             else error (); //X-nonterminal
until stack == < >
//*next refers to the symbol to be checked
```

LL(1) Parsing Example for aacbb

Stack	Input	Action
S#	aacbb$	$S \to aSb$
aSb#	aacbb$	push
Sb#	acbb$	$S \to aSb$
aSbb#	acbb$	push
Sbb#	cbb$	$S \to c$
cbb#	cbb$	push
bb#	bb$	push
b#	b$	push
#	$	ACCEPT

Constructing Parsing Tables

- LL(1) languages are those defined by a parsing table for the LL(1) algorithm
- No table entry can be multiply defined
- We want to generate parsing tables from CFG

Constructing Parsing Tables (Cont.)

- If $A \rightarrow \alpha$, where in the line of A we place α ?
- In the column of t (t is a terminal) where t can start a string derived from α
  - $\alpha \rightarrow^{*} t \, \beta$
  - We say that $t \in First(\alpha)$
- In the column of t if α is ε and t can follow an A
  - $S \rightarrow^{*} \beta \, A \, t \, \delta$
  - We say $t \in Follow(A)$

Computing First Sets

Definition: $\text{First}(X) = \{ t \mid X \to^{*} t\alpha\} \cup \{\varepsilon \mid X \to^{*} \varepsilon\}$

Algorithm sketch :

1. for all terminals t do $\text{First}(X) \leftarrow \{ t \}$ //if X is terminal t
2. for each production $X \to \varepsilon$ do $\text{First}(X) \leftarrow \{ \varepsilon \}$
3. if $X \to A_1 \ldots A_n\ \alpha$ and $\varepsilon \in \text{First}(A_i)$, $1 \le i \le n$ do
   - add $\text{First}(\alpha)$ to $\text{First}(X)$
4. for each $X \to A_1 \ldots A_n$ s.t. $\varepsilon \in \text{First}(A_i)$, $1 \le i \le n$ do
   - add ε to $\text{First}(X)$
5. repeat steps 4 & 5 until no First set can be grown

First Sets. Example

- Recall the grammar

$E \rightarrow T E'$ $\quad T \rightarrow F T'$

$E' \rightarrow + E \mid \varepsilon$ $\quad T' \rightarrow * F \mid \varepsilon$

$F \rightarrow (E) \mid int$

First sets

First(( ) = { (}

First( )) = {) }

First(int) = { int }

First(+) = { + }

First(*) = { * }

First(T) = First (F) = {int, (}

First(E) = {int, (}

First(E') = {+, ε }

First(T') = {*, ε }

Computing Follow Sets

- Definition:

  $$\text{Follow}(X) = \{ t \mid S \to^* \beta\ X\ t\ \delta \}$$

- Intuition
  - If S is the start symbol then $\$ \in \text{Follow}(S)$
  - If $X \to A\ B$ then $\text{First}(B) \subseteq \text{Follow}(A)$ and
    $\text{Follow}(X) \subseteq \text{Follow}(B)$
  - Also if $B \to^* \varepsilon$ then $\text{Follow}(X) \subseteq \text{Follow}(A)$

Computing Follow Sets (Cont.)

Algorithm sketch:

1. Follow(S) $\leftarrow$ { \$ }
2. For each production $A \rightarrow \alpha\ X\ \beta$
   - add First(β) - {ε} to Follow(X)
3. For each $A \rightarrow \alpha\ X\ \beta$ where $\varepsilon \in$ First(β)
   - add Follow(A) to Follow(X)

- repeat step(s) 2, 3 until no Follow set grows

Follow Sets. Example

- Recall the grammar

$E \to T E'$ $\quad$ $T \to FT'$

$E' \to + E \mid \varepsilon$ $\quad$ $T' \to *F \mid \varepsilon$

$F \to (E) \mid int$

- Follow sets

Follow(+) = { int, (} $\quad$ Follow(*) = { int, (}

Follow(( ) = { int, (} $\quad$ Follow(E) = {), $}

Follow(E') = {$,) } $\quad$ Follow(T) = {+,) , $}

Follow( )) = {+,) , $} $\quad$ Follow(T') = {+,) , $}

Follow(int) = {*, +,) , $}

Constructing LL(1) Parsing Tables

- Construct a parsing table T for CFG G
- For each production $A \rightarrow \alpha$ in G do:
  - For each terminal $t \in First(\alpha)$ do
    - $T[A, t] = \alpha$
  - If $\varepsilon \in First(\alpha)$, for each $t \in Follow(A)$ do
    - $T[A, t] = \varepsilon$
  - If $\varepsilon \in First(\alpha)$ and $\$ \in Follow(A)$ do
    - $T[A, \$] = \varepsilon$

Example

- Grammar G:

$E \to TE'$
$E' \to +TE' | \varepsilon$
$T \to FT'$
$T' \to *FT' | \varepsilon$
$F \to (E) | \textbf{int}$

It's possible to implement a predictive parser for G

Parsing table

	+	*****	**(**	**)**	**int**	**$**
E			E→TE'		E→TE'	
E'	E'→+TE'			E'→ε		E'→ε
T			T→FT'		T→FT'	
T'	T'→ε	T'→*****FT’		T'→ε		T'→ε
F			F→**(**E**)**		F→**int**	
+	Push					
*		Push				
(			Push			
)				Push		
int					Push	
#						Accept

Notes on LL(1) Parsing Tables

- If any entry is multiply defined then G is not LL(1)
  - If G is ambiguous
  - If G is left recursive
  - If G is not left-factored
- Most programming language grammars are not LL(1)
- There are tools that build LL(1) tables