Chapter 5: Top-Down Parsing

陳奇業 成功大學資訊工程系

Objectives of Top-Down Parsing

- an attempt to find a **leftmost** derivation for an input string.
- an attempt to construct a parse tree for the input string starting from the root and creating the nodes of the parse tree in **preorder**.

Objectives of Top-Down Parsing

- In this chapter, we study the following two forms of top-down parsers:
  - Recursive-descent parsers contain a set of mutually recursive procedures that cooperate to parse a string. Code for these procedures can be written directly from a suitable grammar.
  - Table-driven LL parsers use a generic LL(k) parsing engine and a parse table that directs the activity of the engine. The entries for the parse table are determined by the particular LL(k) grammar. The notation LL(k) is explained below.

The Basic Method of Recursive-Descent

exp → *exp addop term* | *term*
addop → + | -
term → *term mulop factor* | *factor*
mulop → *
factor → (*exp*) | **number**

```
procedure factor ;
begin
  case token of
  ( :    match( ( ) ;
         exp ;
         match( ) ) ;
  number :
         match(number) ;
  else error ;
  end case ;
end factor ;
```

Using EBNF

- Consider now the case of an exp in the grammar for simple arithmetic expressions in BNF:
  $exp \rightarrow \exp addop\ term \mid term$
- The solution is to use the EBNF rule
  $exp \rightarrow term\{addop\ term\}$

```
procedure exp ;
begin
    term ;
    while token = + or token = - do
        match (token) ;
        term ;
    end while ;
end exp ;
```

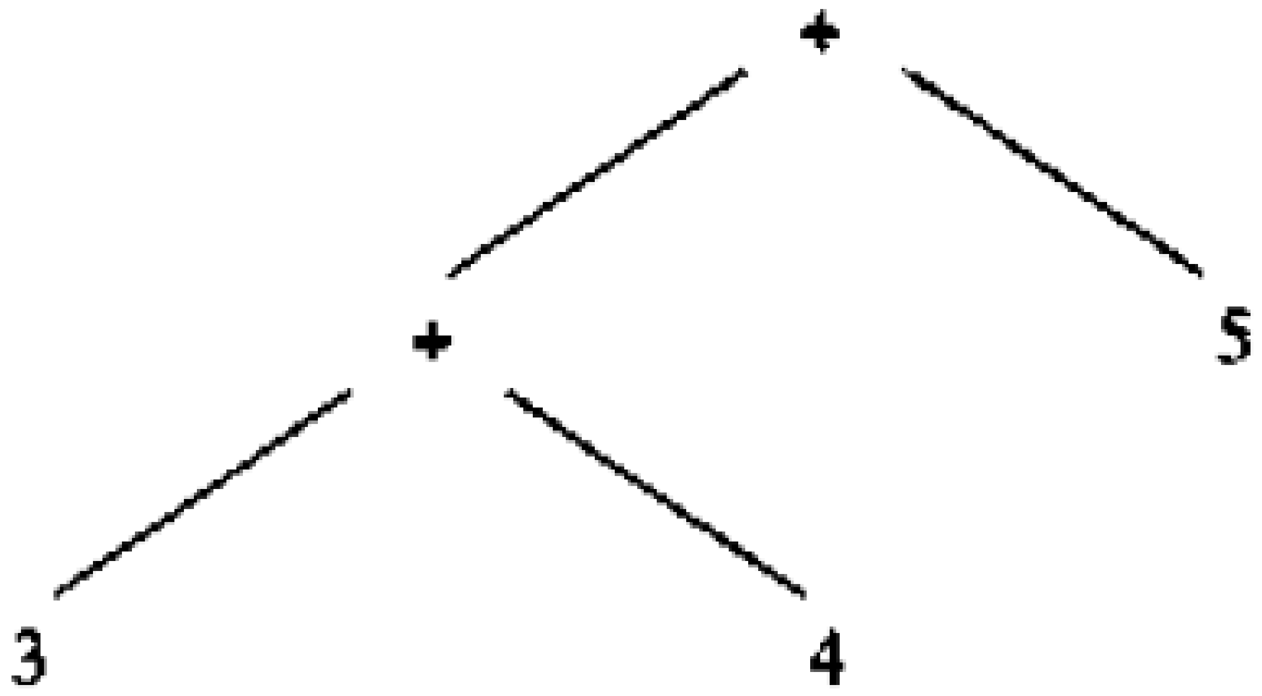
Syntax Tree

```
function exp : integer ;
var temp : integer ;
begin
  temp := term ;
  while token = + or token = - do
    case token of
     + : match (+) ;
         temp := temp + term ;
     - : match (-) ;
         temp := temp - term ;
    end case ;
  end while ;
  return temp ;
end exp ;
```

```
function exp : syntaxTree ;
var temp, newtemp : syntaxTree ;
begin
  temp := term ;
  while token = + or token = - do
    newtemp := makeOpNode(token) ;
    match (token) ;
    leftChild(newtemp) := temp ;
    rightChild(newtemp) := term ;
    temp := newtemp ;
  end while ;
  return temp ;
end exp ;
```

Syntax Tree

- We consider the expression 3+4+5



Approaches of Top-Down Parsing

with backtracking (making repeated scans of the input, a general form of top-down parsing)

Methods: To create a procedure for each nonterminal.

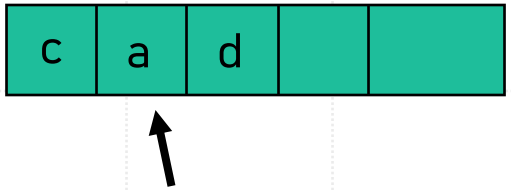
Problems for top-down parsing with backtracking

```
void A() {
    Choose an A-production, A → X_1 X_2 ··· X_k;
    for ( i = 1 to k ) {
        if ( X_i is a nonterminal )
            call procedure X_i();
        else if ( X_i equals the current input symbol a )
            advance the input to the next symbol;
        else /* an error has occurred */;
    }
}
```

e.g. S -> cAd A -> ab | a L = { cabd, cad }

```
S( ) { if input symbol == 'c'
        { Advance();
          if A()
            if input-symbol == 'd'
              {   Advance();
                  return true;
              }
        }
        return false;
      }
```

```
A( ) {  isave= input-pointer;
        if input-symbol == 'a'
          { Advance();
            if input-symbol == 'b'
              {  Advance();
                 return true;
              }
          }
        input-pointer = isave;
        if input-symbol == 'a'
          { Advance();
            return true; }
        else
            return false;
      }
```



Problems for top-down parsing with backtracking

- left-recursion (can cause a top-down parser to go into an infinite loop)
  - Def. A grammar is said to be left-recursive if it has a nonterminal A s.t. there is a derivation $A \Longrightarrow A\delta$ for some δ.

- backtracking - undo not only the movement but also the semantics entering in symbol table.

- the order the alternatives are tried (For the grammar shown above, try $w = cabd$ where $A \rightarrow a$ is applied first)

The LL(1) Predict Function

- Given the productions

  $A \rightarrow \alpha_1$

  $A \rightarrow \alpha_2$

  $\cdots$

  $A \rightarrow \alpha_n$

- During a (leftmost) derivation

  $\cdots A \cdots \Rightarrow \cdots \alpha_1 \cdots$ ***or***

  $\cdots \alpha_2 \cdots$ ***or***

  $\cdots$ ***or***

  $\cdots \alpha_n \cdots$

- Deciding which production to match
  - Using lookahead symbols

The LL(1) Predict Function

Single Symbol Lookahead

$$\text{Predict}(A \to X_1 \cdots X_m) = \begin{cases} (\text{First}(X_1 \cdots X_m) - \lambda) \cup \text{Follow}(A), \text{if } \lambda \in \text{First}(X_1 \cdots X_m) \\ \qquad\qquad \text{First}(X_1 \cdots X_m) \qquad\qquad , \text{otherwise} \end{cases}$$

- The limitation of LL(1)
  - LL(1) contains exactly those grammars that have disjoint predict sets for productions that share a common left-hand side

A grammar G is LL(1) if and only if whenever $A \to \alpha | \beta$ are two distinct productions of G, the following conditions hold:

- 1. For no terminal $\boldsymbol{a}$ do both α and β derive strings beginning with $\boldsymbol{a}$.
- 2. At most one of α and β can derive the empty string.
- 3. If $\beta \Rightarrow^* \epsilon$ then α does not derive any string beginning with a terminal in FOLLOW(A). Likewise, if $\alpha \Rightarrow^* \epsilon$, then β does not derive any string beginning with a terminal in FOLLOW(A)

The LL(1) Predict Function

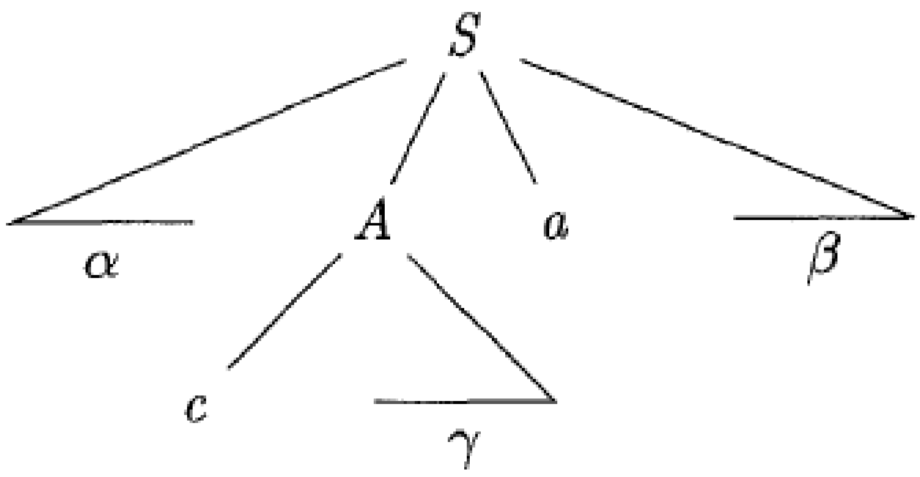


Figure 4.15: Terminal c is in FIRST(A) and a is in FOLLOW(A)

First set

- To compute $\mathrm{First}(X)$ for all grammar symbols X, apply the following rules until no more terminals or λ can be added to any First set.

1. If X is a terminal, then $\mathrm{First}(X) = \{X\}$.
2. If X in a nonterminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in $\mathrm{First}(X)$ if for some i, a is in $\mathrm{First}(Y_i)$, and λ is in all of $\mathrm{First}(Y_1), \dots, \mathrm{First}(Y_{i-1})$; that is $Y_1 \cdots Y_{i-1} \Longrightarrow^* \lambda$. If λ is in $\mathrm{First}(Y_j)$ for all $j = 1, 2, \dots, k$, then add λ to $\mathrm{First}(X)$.
3. If $X \rightarrow \lambda$ is a production, then add λ to $\mathrm{First}(X)$.

An Example

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

$$First(E) = First(T) = First(F) = \{(, id\}$$
$$First(E') = \{+, \lambda\}$$
$$First(T') = \{*, \lambda\}$$

Consider our simple integer expression grammar:[2]

$$
\begin{aligned}
&exp \rightarrow exp\ addop\ term \mid term \\
&addop \rightarrow + \mid - \\
&term \rightarrow term\ mulop\ factor \mid factor \\
&mulop \rightarrow * \\
&factor \rightarrow (\ exp\ ) \mid \mathbf{number}
\end{aligned}
$$

We write out each choice separately so that we may consider them in order (we also number them for reference):

$$
\begin{aligned}
&(1)\quad exp \rightarrow exp\ addop\ term \\
&(2)\quad exp \rightarrow term \\
&(3)\quad addop \rightarrow + \\
&(4)\quad addop \rightarrow - \\
&(5)\quad term \rightarrow term\ mulop\ factor \\
&(6)\quad term \rightarrow factor \\
&(7)\quad mulop \rightarrow * \\
&(8)\quad factor \rightarrow (\ exp\ ) \\
&(9)\quad factor \rightarrow \mathbf{number}
\end{aligned}
$$

$$
\begin{aligned}
&\text{First}(exp) = \{\ (,\ \mathbf{number}\ \} \\
&\text{First}(term) = \{\ (,\ \mathbf{number}\ \} \\
&\text{First}(factor) = \{\ (,\ \mathbf{number}\ \} \\
&\text{First}(addop) = \{+, -\} \\
&\text{First}(mulop) = \{*\}
\end{aligned}
$$

Follow set

- To compute $\mathrm{Follow}(A)$ for all nonterminals A, apply the following rules until nothing can be added to any Follow set.

1. Place λ in $\mathrm{Follow}(S)$, where S is the start symbol.
2. If there is a production $A \to \alpha B \beta$, then everything in $\mathrm{First}(\beta)$ except λ is in $\mathrm{Follow}(B)$.
3. If there is a production $A \to \alpha B$, or a production $A \to \alpha B \beta$, where $\mathrm{First}(\beta)$ contains λ, then everything in $\mathrm{Follow}(A)$ is in $\mathrm{Follow}(B)$.

An Example

$$E \rightarrow TE'$$
$$E' \rightarrow +TE'|\lambda$$
$$T \rightarrow FT'$$
$$T' \rightarrow * FT' |\lambda$$
$$F \rightarrow (E) \mid id$$

/* E is the start symbol */

$\text{Follow}(E) = \{\lambda,)\}$ // rules 1 & 2

$\text{Follow}(E') = \{\lambda,)\}$ // rule 3

$\text{Follow}(T) = \{+, \lambda,)\}$ // rules 2 & 3

$\text{Follow}(T') = \{+, \lambda,)\}$ // rule 3

$\text{Follow}(F) = \{*, +, \lambda,)\}$ // rules 2 & 3

$$\text{First}(E) = \text{First}(T) = \text{First}(F) = \{(, id\}$$
$$\text{First}(E') = \{+, \lambda\}$$
$$\text{First}(T') = \{*, \lambda\}$$

The LL(1) Predict Function

- A grammar G is LL(1) if and only if whenever $A \rightarrow \alpha | \beta$ are two distinct productions of G, the following conditions hold:

common prefixes

1. For no terminal a do both α and β derive strings beginning with a.
   $\text{First}(\alpha) \cap \text{First}(\beta) = \phi$
2. At most one of α and β can derive the empty string.
3. If $\beta \Longrightarrow^* \lambda$, then α does not derive any string beginning with a terminal in $\text{Follow}(A)$. Likewise, if $\alpha \Longrightarrow^* \lambda$, then β does not derive any string beginning with a terminal in $\text{Follow}(A)$.
   $\text{First}(\alpha) \cap \text{Follow}(A) = \phi$ (i.e. If $\text{First}(\alpha)$ contains λ then $\text{First}(\beta) \cap \text{Follow}(A) = \phi$)

1	<program>	→	**begin** <statement list> **end**
2	<statement list>	→	<statement> <statement tail>
3	<statement tail>	→	<statement> <statement tail>
4	<statement tail>	→	λ
5	<statement>	→	ID := <expression> ;
6	<statement>	→	**read** (<id list>) ;
7	<statement>	→	**write** (<expr list>) ;
8	<id list>	→	ID <id tail>
9	<id tail>	→	, ID <id tail>
10	<id tail>	→	λ
11	<expr list>	→	<expression> <expr tail>
12	<expr tail>	→	, <expression> <expr tail>
13	<expr tail>	→	λ
14	<expression>	→	<primary> <primary tail>
15	<primary tail>	→	<add op> <primary> <primary tail>
16	<primary tail>	→	λ
17	<primary>	→	(<expression>)
18	<primary>	→	ID
19	<primary>	→	INTLIT
20	<add op>	→	+
21	<add op>	→	−
22	<system goal>	→	<program> $

Not extended BNF form

$: end of file token

Figure 5.1 A Micro Grammar in Standard Form

The LL(1) Parse Table

- An LL(1) parse table
  $$T: V_n \times V_t \to P \cup \{\text{Error}\}$$
- The definition of T

$T[A][t] = A \to X_1 \cdots X_m$ if $t \in \text{Prediction}(A \to X_1 \cdots X_m)$;

$T[A][t] = \text{Error}$, otherwise

	ID	INTLIT	:=	,	;	+	−	(	)	begin	end	read	write	$
<program>										1				
<statement list>	2											2	2	
<statement>	5											6	7	
<statement tail>	3										4	3	3	
<expression>	14	14						14						
<id list>	8													
<expr list>	11	11						11						
<id tail>				9					10					
<expr tail>				12					13					
<primary>	18	19						17						
<primary tail>				16	16	15	15		16					
<add op>						20	21							
<system goal>										22				

Figure 5.5 The LL(1) Table for Micro

1	<program>	→	**begin** <statement list> **end**
2	<statement list>	→	<statement> <statement tail>
3	<statement tail>	→	<statement> <statement tail>
4	<statement tail>	→	λ
5	<statement>	→	ID := <expression> ;
6	<statement>	→	**read** (<id list>) ;
7	<statement>	→	**write** (<expr list>) ;
8	<id list>	→	ID <id tail>
9	<id tail>	→	, ID <id tail>
10	<id tail>	→	λ
11	<expr list>	→	<expression> <expr tail>
12	<expr tail>	→	, <expression> <expr tail>
13	<expr tail>	→	λ
14	<expression>	→	<primary> <primary tail>
15	<primary tail>	→	<add op> <primary> <primary tail>
16	<primary tail>	→	λ
17	<primary>	→	(<expression>)
18	<primary>	→	ID
19	<primary>	→	INTLIT
20	<add op>	→	+
21	<add op>	→	–
22	<system goal>	→	<program> $

Figure 5.1 A Micro Grammar in Standard Form

Nonterminal	First Set
<program>	{**begin**}
<statement list>	{ID, **read**, **write**}
<statement>	{ID, **read**, **write**}
<statement tail>	{ID, **read**, **write**, λ}
<expression>	{ID, INTLIT, (}
<id list>	{ID}
<expr list>	{ID, INTLIT, (}
<id tail>	{COMMA ,λ}
<expr tail>	{COMMA ,λ}
<primary>	{ID, INTLIT, (}
<primary tail>	{+, –, λ}
<add op>	{+, –}
<system goal>	{**begin**}

Figure 5.2 First Sets for Micro

NonTerminal	Follow Set
<program>	{$}
<statement list>	{**end**}
<statement>	{ID, **read**, **write**, **end**}
<statement tail>	{**end**}
<expression>	{COMMA, SEMICOLON,)}
<id list>	{)}
<expr list>	{)}
<id tail>	{)}
<expr tail>	{)}
<primary>	{COMMA, SEMICOLON, +, –,)}
<primary tail>	{COMMA, SEMICOLON,)}
<add op>	{ID, INTLIT, (}
<system goal>	{λ}

Figure 5.3 Follow Sets for Micro

1 <program> → **begin** <statement list> **end**
2 <statement list> → <statement> <statement tail>
3 <statement tail> → <statement> <statement tail>
4 <statement tail> → λ
5 <statement> → ID := <expression> ;
6 <statement> → **read** (<id list>) ;
7 <statement> → **write** (<expr list>) ;
8 <id list> → ID <id tail>
9 <id tail> → , ID <id tail>
10 <id tail> → λ
11 <expr list> → <expression> <expr tail>
12 <expr tail> → , <expression> <expr tail>
13 <expr tail> → λ
14 <expression> → <primary> <primary tail>
15 <primary tail> → <add op> <primary> <primary tail>
16 <primary tail> → λ
17 <primary> → (<expression>)
18 <primary> → ID
19 <p
20 <a
21 <a
22 <s

Figure 5

Nonterminal	First Set
<program>	{**begin**}
<statement list>	{ID, **read**, **write**}
<statement>	{ID, **read**, **write**}
<statement tail>	{ID, **read**, **write**, λ}
<expression>	{ID, INTLIT, (}
<id list>	{ID}
<expr list>	{ID, INTLIT, (}
<id tail>	{COMMA ,λ}
<expr tail>	{COMMA ,λ}
<primary>	{ID, INTLIT, (}
<primary tail>	{+, –, λ}
<add op>	{+, –}
<system goal>	{**begin**}

Figure 5.2 First Sets for Micro

Prod	Predict Set		
1	First(**begin** <statement list> **end**) =	First(**begin**) =	{**begin**}
2	First(<statement> <statement tail>) =	First(<statement>) =	{ID, **read**, **write**}
3	First(<statement> <statement tail>) =	First(<statement>) =	{ID, **read**, **write**}
4	(First(λ)–λ) ∪ Follow(<statement tail>) =	Follow(<statement tail>) =	{**end**}
5	First(ID := <expression> ;) =	First(ID) =	{ID}
6	First(**read** (<id list>) ;) =	First(**read**) =	{**read**}
7	First(**write** (<expr list>) ;) =	First(**write**) =	{**write**}
8	First(ID <id tail>) =	First(ID) =	{ID}
9	First(, ID <id tail>) =	First(,) =	{,}
10	(First(λ)–λ) ∪ Follow(<id tail>) =	Follow(<id tail>) =	{)}
11	First(<expression> <expr tail>) =	First(<expression>) =	{ID, INTLIT, (}
12	First(, <expression> <expr tail>) =	First(,) =	{,}
13	(First(λ)–λ) ∪ Follow(<expr tail>) =	Follow(<expr tail>) =	{)}
14	First(<primary> <primary tail>) =	First(<primary>) =	{ID, INTLIT, (}
15	First(<add op> <primary> <primary tail>) =	First(<add op>) =	{+, –}
16	(First(λ)–λ) ∪ Follow(<primary tail>) =	Follow(<primary tail>) =	{COMMA, ;,)}
17	First((<expression>)) =	First(() =	{(}
18	First(ID) =		{ID}
19	First(INTLIT) =		{INTLIT}
20	First(+) =		{+}
21	First(–) =		{–}
22	First(<program> $) =	First(<program>) =	{**begin**}

Figure 5.4 Calculation of Predict Sets for Micro

Building Recursive Descent Parsers from LL(1) Tables

- The form of parsing procedure:

```
void non_term(void)
{
    token tok = next_token();
    switch (tok) {
    case TERMINAL_LIST:
        parsing_actions();
        break;
      . . .
    default:
        syntax_error(tok);
        break;
    }
}
```

Building Recursive Descent Parsers from LL(1) Tables

- E.g. of an parsing procedure for <statement> in Micro
- An algorithm that automatically creates parsing procedures like the one in Figure 5.6 from LL(1) table

```
void statement(void)
{
    token tok;

    tok = next_token();
    switch (tok) {
    case ID:
        match(ID); match(ASSIGNOP); expression();
        match(SEMICOLON);
        break;

    case READ:
        match(READ); match(LPAREN); id_list();
        match(RPAREN); match(SEMICOLON);
        break;

    case WRITE:
        match(WRITE); match(LPAREN); expr_list();
        match(RPAREN); match(SEMICOLON);
        break;

    default:
        syntax_error(tok);
        break;
    }
}
```

Figure 5.6 Parsing Procedure for <statement>

Building Recursive Descent Parsers from LL(1) Tables

- The data structure for describing grammars

```
typedef int symbol;    /* a symbol in the grammar */

#define VOCABULARY  (NUM_NONTERMINALS + NUM_TERMINALS)

typedef struct gram {
    symbol terminals[NUM_TERMINALS];
    symbol nonterminals[NUM_NONTERMINALS];
    symbol start_symbol;
    int num_productions;
    struct prod {
        symbol lhs;
        int rhs_length;
        symbol rhs[MAX_RHS_LENGTH];
    } productions[NUM_PRODUCTIONS];
    symbol vocabulary[VOCABULARY];
    char *names[VOCABULARY];
} grammar;

typedef struct prod production;

typedef symbol terminal;
typedef symbol nonterminal;
```

Building Recursive Descent Parsers from LL(1) Tables

- gen_actions()
  - Takes the grammar symbols and generates the actions necessary to match them in a recursive descent parse

```
extern char *make_id(char *);

void gen_actions(symbol x[], int x_length);
{
   int i;
   char *id;

   /*
    * Generate recursive descent
    * actions needed to match x.
    */
   if (x_length == 0)
      printf ("; /* null */\n");
   else {
      for (i = 0; i < x_length; i++) {
         id = make_id(g.names[x[i]]);
         if (is_terminal(x[i]))
            printf("\t\tmatch(%s);\n", id);
         else
            printf("\t\t%s();\n", id);
      }
   }
}
```

Figure 5.7 Algorithm to Generate Recursive Descent Actions

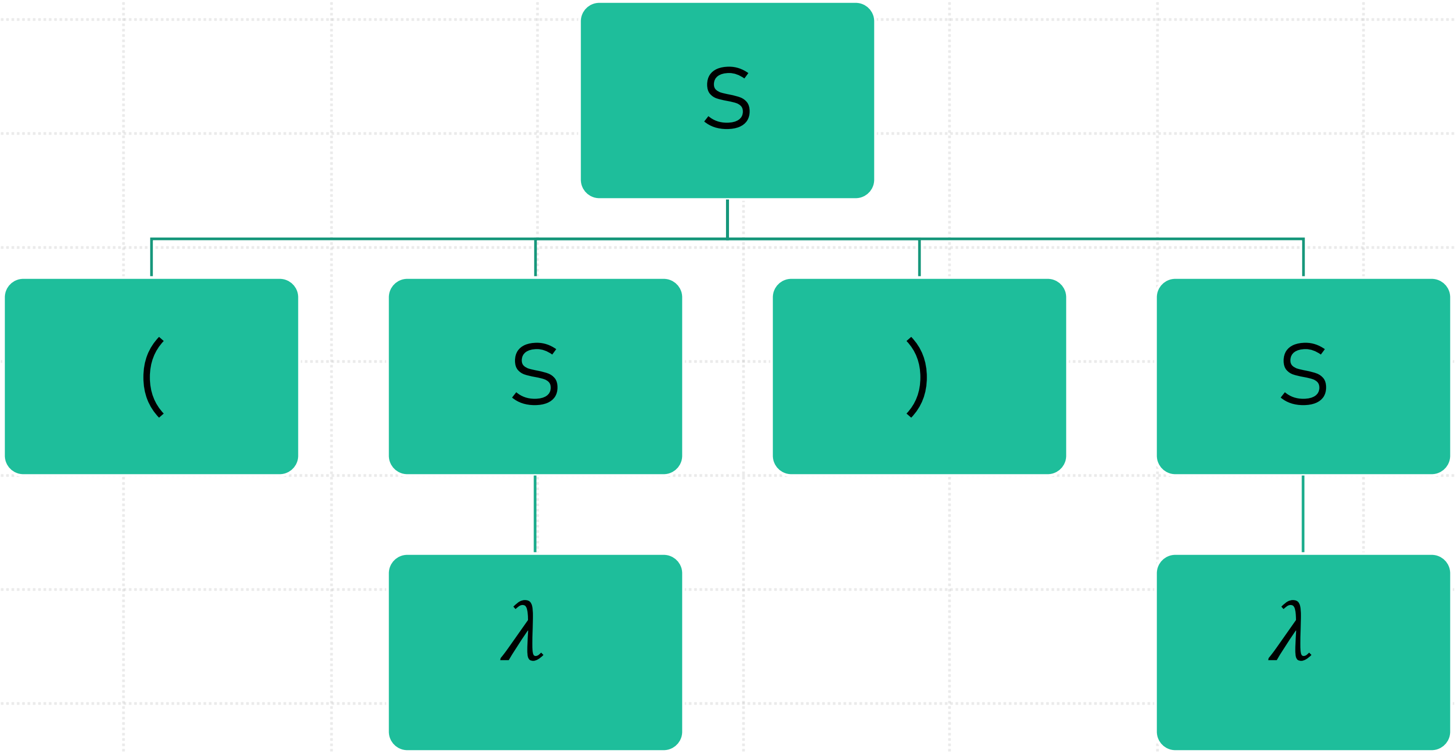
```
void make_parsing_proc(const nonterminal A,
                       const lltable T)
{
   /*
    * Generate recursive descent
    * parsing procedure for A.
    */
   extern grammar g;
   production p;
   terminal x;
   int i, j;

   printf("void %s(void)\n{\n", make_id(g.names[A]));
   printf("\ttoken tok = next_token()\n");
   printf("\tswitch (tok) {\n");

   /* for each production where A is the LHS */
   for (i = 0; i < g.num_productions; i++) {
      if (g.productions[i].lhs != A)
        continue;
      p = g.productions[i];
      /* for each terminal in the grammar */
      for (j = 0; j < NUM_TERMINALS; j++) {
        x = g.terminals[j];
        if (T[A][x] == i)  /* this production */
          printf("\tcase %s:\n", make_id(g.names[x]));
      }
      gen_actions(p.rhs, p.rhs_length);
      printf("\t\tbreak;\n");
   }
   printf("\tdefault:\n");
   printf("\t\tsyntax_error(tok);\n");
   printf("\tbreak;\n\t}\n}\n");
}
```

Figure 5.8 Algorithm to Generate Parsing Procedures

LL(1) Parsing

- $S \to (S)S$
- $S \to \lambda$
- Input String: ()

- $S \Rightarrow_{\mathrm{lm}} (S)S \Rightarrow_{\mathrm{lm}} ()S \Rightarrow_{\mathrm{lm}} ()$

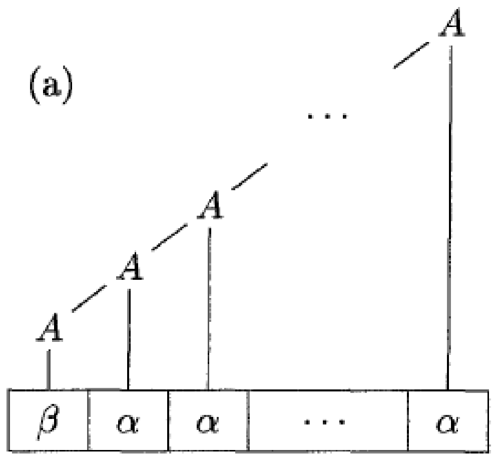


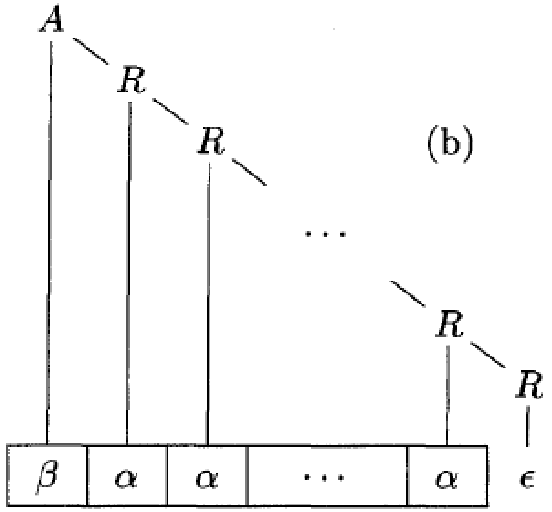
Elimination of Left Recursion

- It is possible for a recursive-descent parser to loop forever. A problem arises with "left-recursive" productions like
  expr $\rightarrow$ expr + term
- A left-recursive production can be eliminated by rewriting the offending production. Consider a nonterminal A with two productions
  A $\rightarrow$ A$\alpha|\beta$
- For example, A= expr, α= + term, β= term

Elimination of Left Recursion

- We can convert left recursion to right recursion in the following manner, using a new nonterminal R:
  A $\rightarrow \beta$R
  R $\rightarrow \alpha$R|ϵ





Elimination of Immediate Left Recursion

- Immediate left recursion can be eliminated by the following technique, which works for any number of A-productions. First, group the productions as

$$A \to A\alpha_1 | A\alpha_2 | \cdots | A\alpha_m | \beta_1 | \beta_2 | \cdots | \beta_n$$

where no β_i begins with an A. Then, replace the A-productions by

$$A \to \beta_1 A' | \beta_2 A' | \cdots | \beta_n A'$$
$$A' \to \alpha_1 A' | \alpha_2 A' | \cdots | \alpha_m A' | \lambda$$

e.g.

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid id$$

After transformation:

$$E \to TE' \quad E' \to +TE' \mid \lambda$$
$$T \to FT' \quad T' \to * FT' \mid \lambda$$
$$F \to (E) \mid id$$

How about left recursion occurred for derivation with more than two steps?

e.g., $S \rightarrow Aa \mid b \quad A \rightarrow Ac \mid Sd \mid e$

where $\boldsymbol{S} \Longrightarrow Aa \Longrightarrow \boldsymbol{S}da$

Algorithm: Eliminating left recursion

Input: Context-free Grammar G with no cycles or λ-production

Methods:

1. Arrange the nonterminals in some order $A_1, A_2, \dots, A_n$
2. for $i = 1$ to n do

    {

        for $j = 1$ to $i - 1$ do

        {

            replace each production of the form $A_i \to A_j\gamma$ by the production $A_i \to \delta_1\gamma|\delta_2\gamma|\cdots|\delta_k\gamma$, where $A_j \to \delta_1|\delta_2|\cdots|\delta_k$ are all <u>current</u> A_j-production;

        }

        eliminate the immediate left-recursion among the A_i-production;

    }

An Example

e.g.

$S \rightarrow Aa \mid b$

$A \rightarrow Ac \mid Sd \mid e$

Step 1: ==> $S \rightarrow Aa \mid b$

Step 2: ==> $A \rightarrow Ac \mid Aad \mid bd \mid e$

Step 3: ==> $A \rightarrow bdA' \mid eA' \quad A' \rightarrow cA' \mid adA' \mid \varepsilon$

Non-backtracking (recursive-descent) parsing

recursive descent : use a collection of mutually recursive routines to perform the syntax analysis.

Left Factoring : $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ ==> $A \rightarrow \alpha A' \quad A' \rightarrow \beta_1|\beta_2$

Methods:

1. For each nonterminal A find the **longest prefix α** common to two or more of its alternatives. If $\alpha \neq \lambda$ replace all the A productions

   $A \rightarrow \alpha\beta_1 | \alpha\beta_2 | \ \ldots \ | \alpha\beta_n | \, others$ by $A \rightarrow \alpha A' \mid others \quad A' \rightarrow \beta_1 | \beta_2 | \ \ldots \ | \beta_n |$

2. Repeat the transformation until no more found

e.g. $S \rightarrow iCtS \mid iCtSeS \mid a \quad C \rightarrow b$

==> $S \rightarrow iCtSS' \mid a \quad S' \rightarrow eS \mid \lambda \quad C \rightarrow b$

Predicative Parsing

Features:

- maintains a stack rather than recursive calls
- table-driven

Components:

1. An input buffer with end marker ($)
2. A stack with endmarker ($) on the bottom
3. A parsing table, a two-dimensional array $M[A, a]$, where 'A' is a nonterminal symbol and 'a' is the current input symbol (terminal/token).

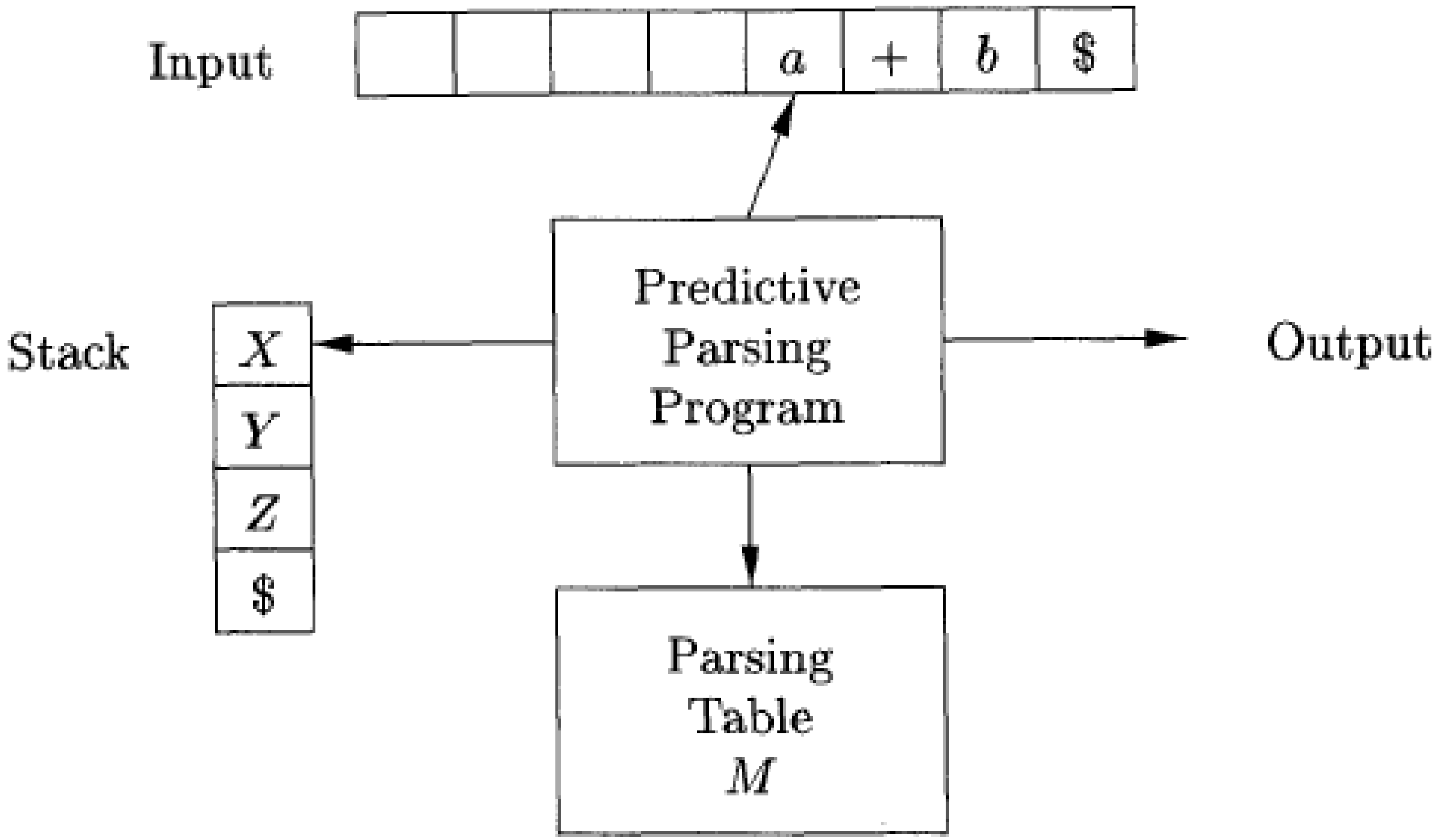


Figure 4.19: Model of a table-driven predictive parser

Parsing Table

$M[A, a]$	(	)	$
S	$S \rightarrow (S)S$	$S \rightarrow \lambda$	$S \rightarrow \lambda$

Algorithm:

Input: An input string w and a parsing table M for grammar G.

Output: A leftmost derivation of w or an error indication.

Initially $\underline{w\$}$ is in input buffer and $\underline{S\$}$ is in the stack.

Starting Symbol of the grammar

Method:

do { Let a of w be the next input symbol and X be the top stack symbol;

if X is a terminal

{ if $X == \boldsymbol{a}$ then pop X from stack and remove $\boldsymbol{a}$ from input;

else ERROR();}

else

{ if $M[X, a] = X \rightarrow Y_1 Y_2 \cdots Y_n$ then

1. pop X from the stack;
2. push $Y_n Y_{n-1} \cdots Y_1$ onto the stack with Y_1 on top;

else

ERROR();

}

} while ($X \neq \$$)

if ($X == \$$) and (the next input symbol $== \$$) then accept else error();

Figure 4.2

Table-based LL(1) parsing algorithm

```
(* assumes $ marks the bottom of the stack and the end of the input *)
push the start symbol onto the top of the parsing stack ;
while the top of the parsing stack ≠ $ and the next input token ≠ $ do
  if the top of the parsing stack is terminal a
        and the next input token = a
  then (* match *)
    pop the parsing stack ;
    advance the input ;
  else if the top of the parsing is nonterminal A
        and the next input token is terminal a
        and parsing table entry M[A, a] contains
              production A → X1X2 . . . Xn
  then (* generate *)
    pop the parsing stack ;
    for i := n downto 1 do
      push Xi onto the parsing stack ;
  else error ;
if the top of the parsing stack = $
        and the next input token = $
then accept
else error ;
```

Table 4.1

Parsing actions of a top-down parser

	Parsing stack	Input	Action
1	$\$\ S$	( ) $	$S \rightarrow (\ S\ )S$
2	$\$\ S\ )\ S\ ($	( ) $	match
3	$\$\ S\ )\ S$	) $	$S \rightarrow \varepsilon$
4	$\$\ S\ )$	) $	match
5	$\$\ S$	$	$S \rightarrow \varepsilon$
6	$	$	accept

$M[A, a]$	(	)	$
$\boldsymbol{S}$	$\boldsymbol{S} \rightarrow (\boldsymbol{S})\boldsymbol{S}$	$\boldsymbol{S} \rightarrow \boldsymbol{\lambda}$	$\boldsymbol{S} \rightarrow \boldsymbol{\lambda}$

Table 4.2

LL(1) parsing table for (ambiguous) if-statements

M[N, T]	**if**	**other**	**else**	0	1	$
statement	*statement* → *if-stmt*	*statement* → **other**				
if-stmt	*if-stmt* → **if** (*exp*) *statement* *else-part*					
else-part			*else-part* → **else** *statement* *else-part* → ε			*else-part* → ε
exp				*exp* → **0**	*exp* → **1**	

First(state) ={if, other}
First(if-stmt)={if}
First(else-part)={else, e}
First(exp)={0.1}

Follow(state)={$, else}
Follow(if-stmt)={$, else}
Follow(else-part)={$, else}
Follow(exp)={)}

Construct a Predicative Parsing Table

1. For each production $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.
2. For each terminal $\boldsymbol{a}$ in $\mathrm{First}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, a]$.
3. If λ is in $\mathrm{First}(\alpha)$, add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal b in $\mathrm{Follow}(A)$.
4. Make each undefined entry of M be error.

LL(1) grammar

A grammar whose parsing table has no multiply-defined entries is said to be LL(1).

First 'L' : scan the input from left to right.
Second ‘L’ : produce a leftmost derivation.
'1' : use one input symbol to determine parsing action.

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

Def. for Multiply-defined entry

If G is left-recursive or ambiguous, then M will have at least one multiply-defined entry.

e.g.

$\boldsymbol{S \rightarrow iCtSS' \mid a \quad S' \rightarrow eS \mid \lambda \quad C \rightarrow b}$

generates:

$M[S', e] = \{ S \rightarrow \lambda, S' \rightarrow eS \}$ with multiply- defined entry.

Parsing table with multiply-defined entry

	a	b	e	i	t	$\$$
S	$S \to a$			$S \to iCtSS'$		
S'			$S' \to \lambda$ $S' \to eS$			$S' \to \lambda$
C		$C \to b$				