Chapter 5 Grammars and Parsers

The LL(1) Predict Function

- Given the productions

  $A \rightarrow \alpha_1$

  $A \rightarrow \alpha_2$

  …

  $A \rightarrow \alpha_n$

- During a (leftmost) derivation

  $\ldots A \ldots \Rightarrow \ldots \alpha_1 \ldots$ ***or***

  $\ldots \alpha_2 \ldots$ ***or***

  $\ldots \alpha_n \ldots$

- Deciding which production to match
  - Using lookahead symbols

The LL(1) Predict Function

Single Symbol Lookahead

$$\text{Predict}(A \rightarrow X_1 \cdots X_m) = \text{if } \lambda \in \text{First}(X_1 \cdots X_m) \text{ then } (\text{First}(X_1 \cdots X_m) - \lambda) \cup \text{Follow}(A) \text{ else } \text{First}(X_1 \cdots X_m)$$

- 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

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

  $\mathbf{T}: V_n \times V_t \rightarrow P \cup \{\text{Error}\}$

- The definition of **T**

  $T[A][t] = \mathbf{A \rightarrow X_1 \ldots X_m}$ if $t \in$ Prediction($\mathbf{A \rightarrow X_1 \ldots X_m}$);

  T[A][t]=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

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

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) $\cup$ 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) $\cup$ 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) $\cup$ 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) $\cup$ 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

- Similar the implementation of a scanner, there are two kinds of parsers
  - Build in
    - The parsing decisions recorded in LL(1) tables can be ***hardwired*** into the parsing procedures used by recursive descent parsers
  - Table-driven

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

```
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

- An algorithm that automatically creates parsing procedures like the one in Figure 5.6 from LL(1) table

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

An LL(1) Parser Driver

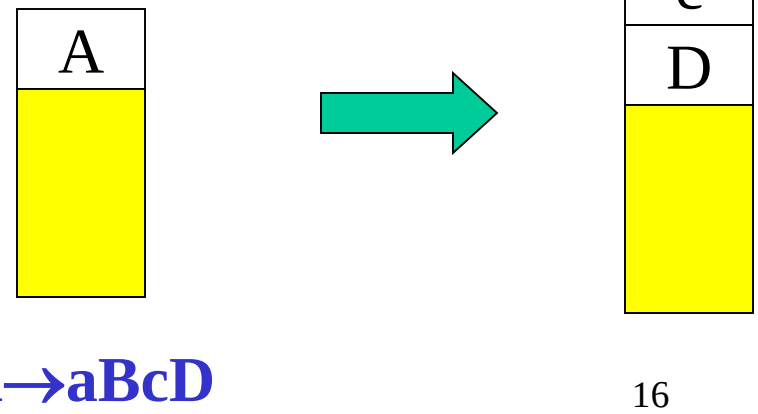
- Rather than using the LL(1) table to build parsing procedures, it is possible to use the table in conjunction with a driver program to form an LL(1) parser
- Smaller and faster than a corresponding recursive descent parser
- Changing a grammar and building a new parser is easy
  - New LL(1) driver are computed and substituted for the old tables

```
void lldriver(void)
{
   /* Push the Start Symbol onto an empty stack. */
   push(s);

   while (! stack_empty() ) {
      /* Let X be the top stack symbol; */
      /* let a be the current input token. */

      if (is_nonterminal(X)
            && T[X][a] == X → Y1 · · · Ym) {
         /* Expand non-terminal */
         pop(1);
         Push Ym, Ym-1, · · · Y1 onto the stack;
      } else if (X == a) {    /* X in terminals */
         pop(1);              /* Match of X worked */
         scanner(& a);        /* Get next token */
      } else
         /* Process syntax error. */
   }
}
```

Figure 5.9 An LL(1) Parser Driver



A→aBcD

LL(1) Action Symbols

- During parsing, the appearance of an action symbol in a production will server to initiate the corresponding semantic action – a call to the corresponding semantic routine.

gen_action("ID:=<expression> #gen_assign;")

```
match(ID);
match(ASSIGN);
exp();
assign();
match(semicolon);
```

LL(1) Action Symbols

- The semantic routine calls pass no explicit parameters
  - Necessary parameters are transmitted through a *semantic stack*
  - *Semantic stack ≠ parse stack*
- Semantic stack is a stack of semantic records.
- Action symbols are pushed to the parse stack
  - See figure 5.11

```
void lldriver(void)
{
   /* Push the Start Symbol onto an empty stack */
   push(s);

   while (! stack_empty() ) {
      /* Let X be the top stack symbol; */
      /* let a be the current input token */

      if (is_nonterminal(X)
            && T[X][a] == X → Y1 · · · Ym) {
         /* Expand nonterminal */
         Replace X with Y1 · · · Ym on the stack;
      } else if (is_terminal(X) && X == a) {
         pop(1);              /* Match of X worked */
         scanner(& a);        /* Get next token */
      } else if (is_action_symbol(X)) {
         pop(1);
         Call Semantic Routine corresponding to X;
      } else
         /* Process syntax error */
   }
}
```

Difference between Fig. 5.9 and Fig 5.11

Figure 5.11 An LL(1) Driver that Handles Action Symbols

Making Grammars LL(1)

- Not all grammars are LL(1). However, some non-LL(1) grammars can be made LL(1) by simple modifications.
- When a grammar is not LL(1) ?

	ID
<stmt>	2,5

- This is called a conflict, which means we do not know which production to use when <stmt> is on stack top and ID is the next input token.

Making Grammars LL(1)

- Major LL(1) prediction conflicts
  - Common prefixes
  - Left recursion
- Common prefixes

```
<stmt> → if <exp> then <stmt>
<stmt> → if <exp> then <stmt> else <stmt>
```

- Solution: *factoring transform*
  - See figure 5.12

Making Grammars LL(1)

```
void factor(grammar *G)
{
   while (G has 2 or more productions with the same
           LHS and a common prefix) {
      Let S = {A → αβ , . . . , A → αζ}
           be the set of productions with the same
           left-hand side, A, and a common prefix, α

      Create a new nonterminal, N;

      Replace S with the production set
           SET_OF( A → α N, N → β , . . . , N → ζ )
   }
}
```

Figure 5.12 An Algorithm that Factors Common Prefixes

<stmt> → if <exp> then <stmt list> <if suffix>

<if suffix> → end if;

<if suffix> → else <stmt list> end if;

Making Grammars LL(1)

- Grammars with left-recursive production can never be LL(1)

  $A \rightarrow A\ \beta$

  – Why? A will be the top stack symbol, and hence the same production would be predicted forever

Making Grammars LL(1)

- Solution: Figure 5.13

```
void remove_left_recursion(grammar *G)
{
   while (G contains a left-recursive nonterminal) {
      Let S = {A → Aα, A → β , . . . , A → ζ}
          be the set of productions with the same
          left-hand side, A, where A is
          left-recursive.

      Create two new nonterminals, T and N;

      Replace S with the production set
          SET_OF( A → N T, N → β , . . . , N → ζ,
              T → αT, T → λ )
   }
}
```

Figure 5.13 An Algorithm to Remove Left Recursion

$$\begin{cases} A \to A\alpha \\ A \to \beta \\ \dots \\ A \to \zeta \end{cases} \Rightarrow \begin{cases} A \to NT \\ N \to \beta \\ \dots \\ N \to \zeta \\ T \to \alpha T \\ T \to \lambda \end{cases}$$

Making Grammars LL(1)

- Other transformation may be needed
  - No common prefixes, no left recursion

1 <stmt> → <label> <unlabeled stmt>

2 <label> → ID :

3 <label> → λ

4 <unlabeled stmt> → ID := <exp> ;

	ID
<stmt>	2,3

Making Grammars LL(1)

```
<stmt> → ID <suffix>
<suffix> → : <unlabeled stmt>
<suffix> → := <exp> ;
<unlabeled stmt> → ID := <exp> ;
```

look ahead 2 tokens

- Example

```
A:    B := C ;

      B := C ;
```

Making Grammars LL(1)

- In Ada, we may declare arrays as

```
A:  array(I .. J, BOOLEAN)
```

- A straightforward grammar for array bound

```
<array bound> → <expr> .. <expr>
<array bound> → ID
```

- Solution

```
<array bound> → <expr> <bound tail>
<bound tail> → .. <expr>
<bound tail> → λ
```

Making Grammars LL(1)

- Greibach Normal Form
  - Every production is of the form $A \rightarrow a\alpha$
    - A is a terminal and is a α (possible empty) string of variables
  - Every context-free language L without λ can be generated by a grammar in Greibach Normal Form
  - Factoring of common prefixes is easy
- Given a grammar G, we can
  - G $\Rightarrow$ GNF $\Rightarrow$ No common prefixes, no left recursion *(but may be still not LL(1)*)

The If-Then-Else Problem in LL(1) Parsing

- "Dangling else" problem in Algo60, Pascal, and C
  - **else** clause is optional
- BL={$[^i]^j \mid i \geq j \geq 0$}

  [$\Rightarrow$ if <expr> then <stmt>

  ] $\Rightarrow$ else <stmt>
- BL is not LL(1) and in fact not LL(k) for any k

The If-Then-Else Problem in LL(1)

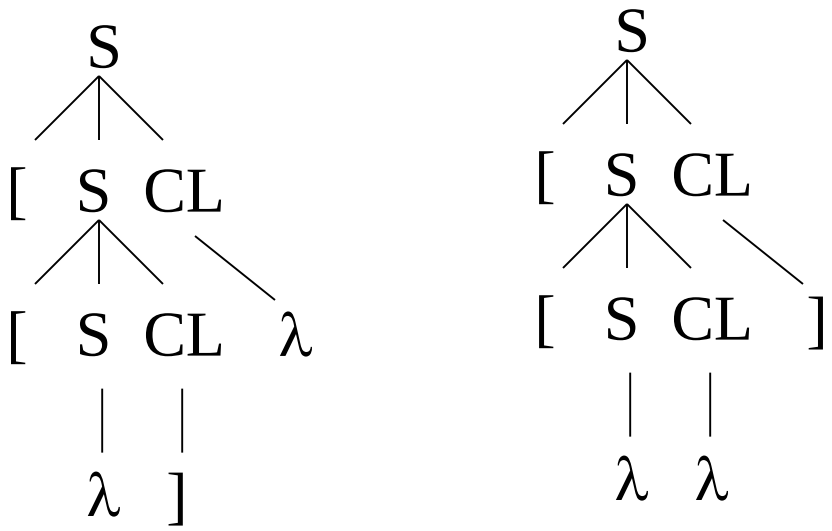
- First try
  - G_1:

    S → [S CL

    S → λ

    CL →]

    S → λ
- G_1 is ambiguous: E.g., [[]



The If-Then-Else Problem in LL(1)

- Second try
  - G_2:

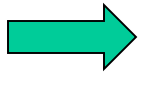
    S → [S

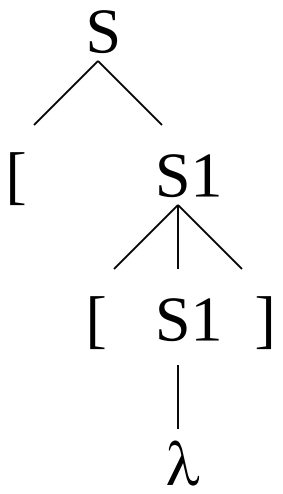
    S → S1

    S1 → [S1]

    S1 → λ

- G_2 is not ambiguous: E.g., [[]





- The problem is

  [∈ First([S) and [∈ First(S1)

  [[∈ First$_2$([S) and [[∈ First$_2$ (S1)

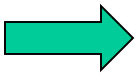
  - G_2 is not LL(1), nor is it LL(k) for any k.

The If-Then-Else Problem in LL(1)

- Solution: conflicts + special rules
  - G_3:

    G → S;
    S → if S E
    S → Other
    E → else S
    E → λ

- G_3 is ambiguous



	if	else	Other	;
S	Predict 2		Predict 3	
E		Predict 4 Predict 5		Predict 5
G	Predict 1		Predict 1	

- We can enforce that T[E, else] = 4th rule.
- This essentially forces "***else*** " to be matched with the nearest unpaired " ***if*** ".

The If-Then-Else Problem in LL(1)

- If all **if** statements are terminated with an **end if**, or some equivalent symbol, the problem disappears.

```
S → if S E
S → Other
E → else S end if
E → end if
```

- An alternative solution
  - Change the language