Compiler Technology of Programming Languages

Chapter 5

Top-Down Parsing

Prof. Farn Wang

The LL(k) Predict Function

- Given the productions

  $A \rightarrow \alpha_1$

  $A \rightarrow \alpha_2$

  …

  $A \rightarrow \alpha_n$

- During a (leftmost) derivation

  … A … $\Rightarrow$ … α_1 … ***or***

  … α_2 … ***or***

  … α_n …

- Deciding which production to match
  - Using k lookahead symbols

The LL(1) Predict Function

$$\begin{aligned}\text{Predict}(A \rightarrow X_1 \cdots X_m) = & \\ & \text{if } \lambda \in \text{First}(X_1 \cdots X_m) \\ & \qquad (\text{First}(X_1 \cdots X_m) - \lambda) \cup \text{Follow}(A) \\ & \text{else} \\ & \qquad \text{First}(X_1 \cdots X_m)\end{aligned}$$

- 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

```
function Predict(p : A → X1 ... Xm) : Set
    ans ← First(X1 ... Xm)                    (1)
    if RuleDerivesEmpty(p)                    (2)
    then
        ans ← ans ∪ Follow(A)                 (3)
    return (ans)
end
```

Figure 5.1: Computation of Predict sets.

The LL(1) Parse Table

- An LL(1) parse table

  T:V_n x $V_t \rightarrow P \cup \{Error\}$

- 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

1 S → A C $
2 C → c
3 | λ
4 A → a B C d
5 | B Q
6 B → b B
7 | λ
8 Q → q
9 | λ

Figure 5.2: A CFGs.

Example:

FIRST(AC)
→compute FIRST(A)
→{a, FIRST(BQ)}
→{a,b,FIRST(Q)}
→{a,b,q, λ}
→{a,b,q} + FIRST(C)
→{a,b,q}+{c, λ}
→{a,b,q,c, λ} + {$}
→{a,b,q,c,$}

1 S → A C $
2 C → c
3 | λ
4 A → a B C d
5 | B Q
6 B → b B
7 | λ
8 Q → q
9 | λ

Figure 5.2: A CFGs.

Predict Set

{a,b,q,c,$}	1 S → A C $
{c}	2 C → c
	3 | λ
{a}	4 A → a B C d
	5 | B Q
{b}	6 B → b B
	7 | λ
{q}	8 Q → q
{c,$}	9 | λ

Figure 5.2: A CFGs.

Predict Set

{a,b,q,c,$}
{c}
{$,d}
{a}
{b,q,c,$}
{b}
{q,c,d,$}
{q}
{c,$}

1 S → A C $
2 C → c
3 | λ
4 A → a B C d
5 | B Q
6 B → b B
7 | λ
8 Q → q
9 | λ

Figure 5.2: A CFGs.

Rule Number	A	$X_1 \ldots X_m$	First($X_1 \ldots X_m$)	Derives Empty?	Follow(A)	Answer
1	S	A C $	a,b,q,c,$	No		a,b,q,c,$
2	C	c	c	No		c
3		λ		Yes	d,$	d,$
4	A	a B C d	a	No		a
5		B Q	b,q	Yes	c,$	b,q,c,$
6	B	b B	b	No		b
7		λ		Yes	q,c,d,$	q,c,d,$
8	Q	q	q	No		q
9		λ		Yes	c,$	c,$

Figure 5.3: Predict calculation for the grammar of Figure 5.2.

```
procedure FillTable(LLtable)
    foreach A ∈ N do
        foreach a ∈ Σ do LLtable[A][a] ← 0
    foreach A ∈ N do
        foreach p ∈ ProductionsFor(A) do
            foreach a ∈ Predict(p) do LLtable[A][a] ← p
end
```

Figure 5.9: Construction of an LL(1) parse table.

Nonterminal	Lookahead a	b	c	d	q	$
S	1	1	1		1	1
C			2	3		3
A	4	5	5		5	5
B		6	7	7	7	7
Q			9		8	9

Figure 5.10: LL(1) table. The blank entries should trigger error actions in the parser.

Building Recursive Descent Parsers from LL(1) Tables

- Similar to 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

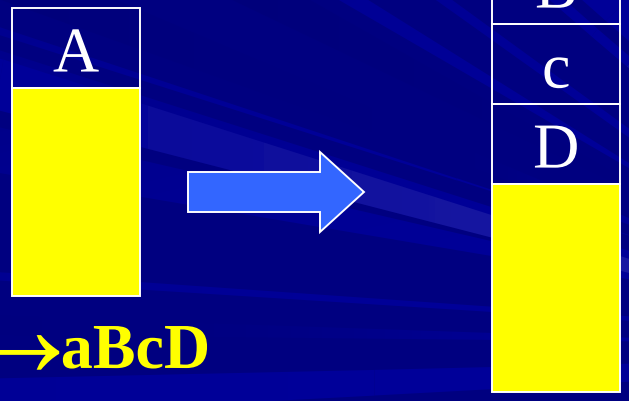
An LL(1) Parser Driver

- 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. */
   }
}
```



A→aBcD

LL(1) Action Symbols

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

ID := <expression> #gen_assign ;

```
match(ID);
match(ASSIGN);
exp();
gen_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

```
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 */
   }
}
```

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) ?

	a
<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 a 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: *left factoring*

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

FAQ

- How is First(A→P) different from Predict(A→P)?

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

- Why do we need the stack for LL Parsing?
  Recursion
- How is First(A) different from First(A→P)?

Making Grammars LL(1)

- Other transformations may be needed
  - No common prefixes, no left recursion, but conflicts still exist.

```
1   <stmt> → <label> <unlabeled st
2   <label> → ID :
3   <label> → λ
4   <unlabeled stmt> → ID := <exp> ;
```

```
A:    B := C ;

      B := C ;
```

	ID
<label>	2,3

LL(2) will work

Making Grammars LL(1)

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

- Example

```
A:    B := C ;

      B := C ;
```

Dangling else

<stmt> → if <expr> then stmt

<stmt> → if <expr> then stmt else stmt

<stmt> → while (<expr>) stmt

<stmt> → for (expr; expr; expr) stmt

<stmt> → if <expr> then stmt V

<V> → else stmt

→ λ

Left factoring may fail to solve the problem !

1 S → Stmt $
2 Stmt → if expr then Stmt V
3 | other
4 V → else Stmt
5 | λ

e.g. after left factoring, the dangling IF's problem remains…

Nonterminal	Lookahead					
	if	expr	then	else	other	$
S	1				1	
Stmt	2				3	
V				4,5		5

Figure 5.19: Ambiguous grammar for if-then-else and its LL(1) table. The ambiguity is resolved by favoring Rule 4 over Rule 5 in the boxed entry.

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

- Solution: conflicts + special rules

  In Figure 5.19, T[V, else] = 4 and 5

- We can enforce that T[V, 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