

COMPILER CONSTRUCTION

Top-Down Parsing

Chia-Heng Tu
Dept. of Computer Science and Information Engineering
National Cheng Kung University
Spring 2017

NCKU
National Cheng Kung University



Chapter 5

Top-Down Parsing



Overview

- This chapter discusses the principles for **automatic construction of the parsing phase of a compiler**
  - Ch. 2 presents a recursive-descent parser for the syntax analysis phase of a small compiler
  - **Recursive-descent parsers** belong to the more general class of **top-down** (also called **LL**) **parsers**, which were introduced in Ch. 4
- Discuss **top-down parsers** in greater detail
  - Analyze the conditions under which such parsers can be reliably and
  - construct from grammars automatically
  - The analysis builds on the algorithms and grammar-processing concepts presented in Ch. 4



Top-Down and Bottom-Up Parsers

- Top-down parsers are in theory not as powerful as the bottom-up parsers (Ch. 6)
- However, **top-down parsers have been constructed for many programming languages**
  - because of their simplicity, performance, and excellent error diagnostics
  - They are also convenient for prototyping relatively simple front-ends of larger systems that require a rigorous definition and treatment of the system's input

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



Recap: Mutual Recursion

- **Mutual recursion** is a form of recursion
  - where two mathematical or computational objects are **defined in terms of each other**
  - such as functions or data types
- Example
  - Determine whether a non-negative number is even or odd?
  - It is done by defining two separate functions that call each other, decrementing each time

```
bool is_even(unsigned int n)
{
    if (n == 0)
        return true;
    else
        return is_odd(n - 1);
}
bool is_odd(unsigned int n) {
    if (n == 0)
        return false;
    else
        return is_even(n - 1);
}
```

Parsing Problem

- Every string in a grammar's language
  - can be generated by a derivation
  - that begins with the grammar's start symbol
  - We learn from previous chapters
- While it is relatively straightforward to use a grammar's productions to generate sample strings in its language,
  - reversing the process does not seem as simple

Parsing Problem (Cont'd)

- The parsing problem:
  - Given an input string, how can we show why the string is or is not in the grammar's language
  - in this chapter, we consider **a parsing technique** that is successful with many context-free grammars
- This parsing technique is known by the following names:
  - **Top-down**, because the parser begins with the grammar's start symbol and **grows a parse tree from its root to its leaves**
  - **Predictive**, because the parser must predict at each step in the derivation **which grammar rule is to be applied next**
  - **LL(k)**, because these techniques **scan the input from left to right** (the first "L" of LL), produce a **leftmost derivation** (the second "L" of LL), and use **k symbols of lookahead**
  - **Recursive descent**, because this kind of parser can be implemented by a collection of **mutually recursive procedures**



Reprise: Recursive-Descent Parsing

- A **parsing procedure** is associated with each nonterminal A
- The procedure associated with A is charged with accomplishing one step of a derivation by <u>choosing and applying one of A's productions</u>
- The parser chooses the appropriate production for A by <u>inspecting the next k tokens</u> (terminal symbols) in the input stream
- The **Predict set** for production A ⇒ α is the set of tokens that trigger application of that production
- The **Predict set** for A ⇒ α is determined primarily by
  - the detail in α — the **right-hand side** (RHS) of the production
  - Other CFGs productions may participate in the computation of a production's Predict set

LL(k) Grammars

- The CFGs is an **LL(k) grammar,**
  - if it is possible to construct an LL(k) parser for the CFGs such that the parser recognizes the CFGs's language
- With the LL(k) parser,
  - the choice of production can be predicated on the next k tokens of input, where
    - the constant k is chosen before the parser takes inputs
  - The first "L" stands for scanning input from left to right
  - The second "L" for producing a leftmost derivation
  - The "k" for using k input symbol of lookahead at each step to make parsing decisions



Predict Set of an LL(k) Parser

- In other words,
  - an LL(k) parser can **peek** at the next k tokens to decide which production to apply
- The ***strategy*** for choosing productions must be established when the parser is constructed
  - The strategy is formalized by defining a function called **Predictk(p)**
  - This function considers the **grammar production p** and computes the set of **length-k token strings** that predict the application of rule p



Predict Strategy of an LL(1) Parser

- Consider the input string $\alpha a \beta \in \Sigma^*$
- Suppose the parser has constructed the derivation $S \Rightarrow^*_{lm} \alpha A Y_1 \ldots Y_n$, where
  - α has been matched and
  - A is the leftmost nonterminal in the derived sentential form
- To continue the leftmost derivation, **some production for A must be applied**
  - Because the input string contains an `a' as the next input token, the parse must continue with a production for A that derives `a' as its first terminal symbol



Predict Strategy of an LL(1) Parser (Cont'd)

- We use the following to find the set P
  - P = { p ∈ ProductionsFor(A) | a ∈ Predict(p) }

  p ∈ ProductionsFor(A)

  1. p refers to the productions that could be derived from A

  a ∈ Predict(p)

  2. a refers to the FIRST and FOLLOW sets for each given p

  - **ProductionsFor(A)**
    - returns an iterator that visits each production for nonterminal A
    - ProductionsFor(A) is defined in Sec. 4.5.1 on page 127

Predict Strategy of an LL(1) Parser (Cont'd)

- One of the following conditions must be true of the set P and the next input token a:
  1. P is the empty set
  2. P contains more than one production
  3. P contains exactly one production



Predict Strategy of an LL(1) Parser (Cont'd)

1. P is the empty set
   - In this case, **no production for** A can cause the next input token to be matched
   - **The parse cannot continue and a syntax error is issued**, with a as the offending token
   - The productions for A can be helpful in issuing error messages that indicate which terminal symbols could be processed at this point in the parse
   - Sec. 5.9 considers error recovery and repair in greater detail



Predict Strategy of an LL(1) Parser (Cont'd)

2. P contains more than one production
   - In this case, **the parse could continue**, but **nondeterminism** would be required to pursue the independent application of each production in P
   - For efficiency, we require that **our parsers operate deterministically**
   - Thus parser construction must ensure that this case cannot arise

Predict Strategy of an LL(1) Parser (Cont'd)

3. P contains exactly one production
   - In this case, **the leftmost parse can proceed deterministically** by applying the only production in set P

Compute Predict Set

```
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.

- Now, we show how to compute **Predict(p)**
- Consider a production
  p: A ⇒ $X_1 \ldots X_m$, m ≥ 0
  - When m = 0, it means A ⇒ λ (there are no symbols on A's RHS)
- From Fig. 5.1, the symbols included in the predict set are drawn from **one or both of the following**:
  - The set of possible terminal symbols that are **first produced in some derivation** from **$X_1 \ldots X_m$** (Marker 1 in Fig. 5.1)
  - The terminal symbols that can **follow A** in some sentential form (Marker 3 in Fig. 5.1)

Compute Predict Set (Cont'd)

```
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.

Marker 1

- The Predict procedure initializes ***ans*** to **FIRST($X_1 \ldots X_m$)**
  - that is the set of terminal symbols that can appear first (leftmost) in any derivation of $X_1 \ldots X_m$
  - Refer to Fig. 4.8 for computing FIRST set

Marker 2

- It detects if $X_1 \ldots X_m \Rightarrow \lambda$ with the procedure RuleDerivesEmpty(p),
  - which is true if, and only if, **production p can derive λ** (Refer to Fig. 4.7 for symbols and productions deriving λ)

Marker 3

- The symbols in **FOLLOW(A)** are further computed and included in ***ans***,
  - **FOLLOW(A) symbols** refer to the set of symbols that follow A when $A \Rightarrow \lambda$ (A derives λ); FOLLOW is defined in Fig. 4.11
  - Thus, the function shown in Fig. 5.1 computes the set of length-1 token strings that predict rule p
  - NOTE: λ is not a terminal symbol, so it does not participate in any Predict set



Check if Grammar G is LL(1)

- *Given $A \Rightarrow \alpha \,|\, \beta$,
  - which is two distinct productions of a grammar G
  - The grammar G is LL(1) if and only if the following conditions hold:
    1. FIRST(α) cannot contain any terminal in FIRST(β)
    2. At most one of α and β can derive λ
    3. if $\beta \rightarrow^* \lambda$, FIRST($\alpha$) cannot contain any terminal in FOLLOW(A)
       if $\alpha \rightarrow^* \lambda$, FIRST($\beta$) cannot contain any terminal in FOLLOW(A)

*Refer to Sec. 4.4.3 from the second textbook: Compilers Principles Techniques and Tools. 2nd. 2006.Addison-Wesley



Check if Grammar G is LL(1) (Cont'd)

1. FIRST(α) cannot contain any terminal in FIRST(β)
2. At most one of α and β can derive λ

→ In other words, the above conditions are equivalent to the statement, "**FIRST(α) and FIRST(β) are disjoint sets.**"

3. if $\beta \rightarrow^* \lambda$, FIRST(α) cannot contain any terminal in FOLLOW(A)
   if $\alpha \rightarrow^* \lambda$, FIRST(β) cannot contain any terminal in FOLLOW(A)

→ The above condition is equivalent to the statement that "if λ is in FIRST(β), then **FIRST(α) and FOLLOW(A) are disjoint sets**, and likewise if if λ is in FIRST(α)."

- Or, you can say grammar G is in an LL(1) grammar,
  - if the productions for each nonterminal A in G must have **disjoint predict sets**, as computed with one symbol of lookahead



Check if Grammar G is LL(1) (Cont'd)

- The procedure shown in Fig. 5.4 determines
  - whether a grammar is LL(1) based on the grammar's Predict sets
  - The Predict sets for each nonterminal A are checked for intersection
  - **If no two rules for A have any prediction symbols in common, then the grammar is LL(1)**

```
function IsLL1(G) returns Boolean
    foreach A ∈ N do
        PredictSet ← ∅
        foreach p ∈ ProductionsFor(A) do
            if Predict(p) ∩ PredictSet ≠ ∅                          4
            then return (false)
            PredictSet ← PredictSet ∪ Predict(p)
    return (true)
end
```

Predcit set for p is also in the *PredictSet* for the visited nonterminals

PredictSet keeps the Predcit set of all the visited nonterminals

Figure 5.4: Algorithm to determine if a grammar G is LL(1).



Example: Check if the Grammar is LL(1) Find Predict Sets

```
function Predict(p : A→X_1...X_m) : Set
    ans ← First(X_1...X_m)                    ①
    if RuleDerivesEmpty(p)                    ②
    then
        ans ← ans ∪ Follow(A)                 ③
    return (ans)
end
```

Figure 5.1: Computation of Predict sets.

- N = {S, C, A, B, Q}
- **ProductionsFor(C)** // $C \Rightarrow$ c | λ
  - c
  - λ
- **Predict(C)**
  - **First(C)** = {c, λ}

  // Compute Follow(C) since we have λ in the *First set* of *C*
  - **Follow(C)** = {d, $}
- NOTE: Predict set does not contain λ

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.



Example: Check if the Grammar is LL(1)

- $N = \{S, C, A, B, Q\}$
- **ProductionsFor(A)**
  - a B C d
  - B Q
- **Predict(**a B C d**)** $\cap$ **Predict(**B Q**)** $= \emptyset$ (**empty set**)
  - **Predict(**a B C d**)** = {a}
  - **Predict(**B Q**)** = {b, q, c, \$}
- The grammar listed in Fig. 5.2 is LL(1)

```
function IsLL1(G) returns Boolean
    foreach A ∈ N do
        PredictSet ← ∅
        foreach p ∈ ProductionsFor(A) do
            if Predict(p) ∩ PredictSet ≠ ∅
            then return (false)
            PredictSet ← PredictSet ∪ Predict(p)
    return (true)
end
```

Figure 5.4: Algorithm to determine if a grammar G is LL(1).

```
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.



Recursive-Descent LL(1) Parsers

- Now, we show the procedures of constructing a **recursive-descent parser for an LL(1) grammar**
  - The parser's input is **a sequence of tokens** provided by the stream *ts*
  - *ts* offers the following methods:
  1. **peek**, which examines the next input token without advancing the input
  2. **advance**, which advances the input by one token
- The parsers we construct rely on the **match** method shown in Fig. 5.5
  - This method checks the token stream *ts* for the presence of a particular token



Recursive-Descent Procedure in the Parser

- A separate procedure for each nonterminal *A* is illustrated in Fig. 5.6,
  - where *A* has rules $p_1, p_2, \ldots, p_n$
  - The code constructed for each p_i is obtained by **scanning the RHS of rule p_i** from left to right
  - **In other words, the above means** $A \Rightarrow p_1 \mid p_2 \mid \ldots \mid p_n$, and $p_i = X_1 \ldots X_m$
  - **ts.peek() ∈ Predict(p_i)** means the Predict set of p_i is used to see if the next input matches the rule $\boldsymbol{p_i}$

```
procedure A(ts)
    switch (...)
        case ts.PEEK( ) ∈ Predict(p1)
            /*    Code for p1                */
        case ts.PEEK( ) ∈ Predict(pi)
            /*    Code for p2                */
        /*    .                              */
        /*    .                              */
        /*    .                              */
        case ts.PEEK( ) ∈ Predict(pn)
            /*    Code for pn                */
        case default
            /*    Syntax error               */
end
```

Figure 5.6: A typical recursive-descent procedure. Successful LL(1) analysis ensures that only one of the **case** predicates is **true**.

Recursive-Descent Procedure in the Parser (Cont'd)

```
procedure A(ts)
    switch (...)
        case ts.PEEK() ∈ Predict(p1)
            /*    Code for p1                         */
        case ts.PEEK() ∈ Predict(pi)
            /*    Code for p2                         */
        /*    .                                       */
        /*    .                                       */
        /*    .                                       */
        case ts.PEEK() ∈ Predict(pn)
            /*    Code for pn                         */
        case default
            /*    Syntax error                        */
end
```

Figure 5.6: A typical recursive-descent procedure. Successful LL(1) analysis ensures that only one of the **case** predicates is **true**.

- If $A \Rightarrow \lambda$

  $A \Rightarrow p_1 \mid p_2 \mid \ldots \mid p_n$, where $p_i = \lambda$

  - Since m = 0, there are no symbols to visit
  - In such cases, the parsing procedure simply **returns immediately**
- Otherwise,

  $A \Rightarrow p_1 \mid p_2 \mid \ldots \mid p_n$ and $p_j = X_1 \ldots X_m$, where each X_i could be terminal or nonterminal

  - Considering each X_i, there are two possible cases, as follows:

  1. **X_i is a terminal symbol**

  - In this case, **a call to match(**ts, X_i**) is written into the parser** to insist that X_i is the next symbol in the token stream
    - If the token is successfully **matched**, then the token stream is advanced
    - Otherwise, the input string cannot be in the grammar's language and an **error** message is issued

  2. **X_i is a nonterminal symbol**

  - **A call to X_i(**ts**) is written into the parser**
    - In this case, there is a procedure responsible for continuing the parse by choosing an appropriate production for X_i

Example: Recursive-Descent Procedure

- Fig. 5.7 shows the parsing procedures created for the LL(1) grammar shown in Fig. 5.2
  - For presentation purposes, the default case (representing a syntax error) is not shown

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.

```
procedure S( )
    switch (...)
        case ts.PEEK( ) ∈ { a, b, q, c, $ }
            call A( )
            call C( )
            call MATCH($)
end
procedure C( )
    switch (...)
        case ts.PEEK( ) ∈ { c }
            call MATCH(c)
        case ts.PEEK( ) ∈ { d, $ }
            return ()
end
procedure A( )
    switch (...)
        case ts.PEEK( ) ∈ { a }
            call MATCH(a)
            call B( )
            call C( )
            call MATCH(d)
        case ts.PEEK( ) ∈ { b, q, c, $ }
            call B( )
            call Q( )
end
procedure B( )
    switch (...)
        case ts.PEEK( ) ∈ { b }
            call MATCH(b)
            call B( )
        case ts.PEEK( ) ∈ { q, c, d, $ }
            return ()
end
procedure Q( )
    switch (...)
        case ts.PEEK( ) ∈ { q }
            call MATCH(q)
        case ts.PEEK( ) ∈ { c, $ }
            return ()
end
```

Figure 5.7: Recursive-descent code for the grammar shown in Figure 5.2. The variable *ts* denotes the token stream produced by the scanner.

Example: Recursive-Descent Procedure (Con’td)

1. **X_i is a terminal symbol**
   - In this case, **a call to match(**ts, X_i**) is written into the parser** to insist that X_i is the next symbol in the token stream
     - If the token is successfully **matched**, then the token stream is advanced
2. **X_i is a nonterminal symbol**
   - **A call to *X_i*(**ts**) is written into the parser**
     - In this case, there is a procedure responsible for continuing the parse by choosing an appropriate production for X_i

Rule Number	A	$X_1 \ldots X_m$	First($X_1 \ldots X_m$)	Derives Empty?	Follow(A)	Answe
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 S( )
    switch (...)
        case ts.PEEK( ) ∈ { a, b, q, c, $ }
            call A( )
            call C( )
            call MATCH( $ )
end
procedure C( )
    switch (...)
        case ts.PEEK( ) ∈ { c }
            call MATCH( c )
        case ts.PEEK( ) ∈ { d, $ }
            return ()
end
procedure A( )
    switch (...)
        case ts.PEEK( ) ∈ { a }          ← Predict(A⇒ a B C d)
            call MATCH( a )              ← Terminal
            call B( )
            call C( )                    ← Nonterminal
            call MATCH( d )
        case ts.PEEK( ) ∈ { b, q, c, $ } ← Predict(A⇒B Q)
            call B( )
            call Q( )
end
procedure B( )
    switch (...)
        case ts.PEEK( ) ∈ { b }
            call MATCH( b )
            call B( )
        case ts.PEEK( ) ∈ { q, c, d, $ }
            return ()
end
procedure Q( )
    switch (...)
        case ts.PEEK( ) ∈ { q }
            call MATCH( q )
        case ts.PEEK( ) ∈ { c, $ }
            return ()
end
```

Figure 5.7: Recursive-descent code for the grammar shown in Figure 5.2. The variable *ts* denotes the token stream produced by the scanner.

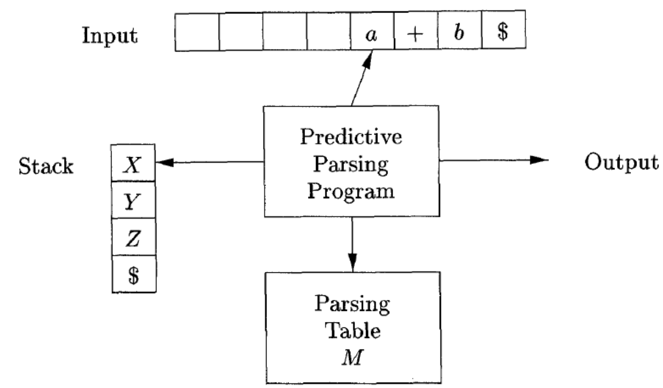


Why Table-Driven LL(1) Parsers?

- The task of creating recursive-descent parsers is mechanical and can be automated
- However, **the size of the parser's code grows with the size of the grammar**
  - Moreover, the overhead of **method calls and returns** can be a source of inefficiency
- The **table-driven LL(1) parsers** are developed to tackle the above issues
  - The parser itself is standard across all grammars, so we need only provide an **adequate** **parse table**
  - The parser mimics a leftmost derivation
  - It is also known as **nonrecursive predictive parser**

Facilities of Table-Driven LL(1) Parsers

- A **parsing table**
  - to describe the relationships among the **nonterminals** and the **input tokens**
  - generated from the given LL(1) grammar
- A **stack**
  - keeps the derived **nonterminals** during parsing
  - is used to simulate the actions performed by **match** and by the calls to the **nonterminals' procedures**
  - The stack is used to make the transition from explicit code to table-driven processing
- **Methods for the stack**
  - Typical methods: push and pop
  - Obtaining the top-of-stack contents method: TOS
    - The value is obtained without popping the stack



Overall architecture of the table-driven LL(1) parsers

*Refer to Sec. 4.4.4 from the second textbook: Compilers Principles Techniques and Tools. 2nd. 2006.Addison-Wesley

The LL(1) Parse Table

- We first show how to build **the LL(1) parse table**
  - Note that the given CFGs is the LL(1) grammar, which means the CFGs should pass the **IsLL1 test** in Fig. 5.4
- Its **rows** and **columns**
  - are labeled by the **nonterminals** and **terminals** of the CFGs, respectively
- It is **indexed**
  - by the **top-of-stack symbol** (obtained by the **TOS**( ) call) and
  - by the **next input token** (obtained by the *ts*.**peek**( ) call)

Example: The LL(1) Parse Table

- **Each nonblank entry** in a row is a production that
  - has the **row's nonterminal** as its **left-hand side** (LHS) symbol
  - is typically represented by its **rule number** in the grammar
- The table is used as follows:
  - The nonterminal symbol at the top-of-stack determines which row is chosen
  - The next input token (i.e., the **lookahead**) determines which column is chosen
- Example:
  - "*LLtable*[S, a]" means that top-of-stack is S and the next input token is a
  - $\boxed{\text{"}\mathit{LLtable}\text{[S, a] = 1"}}$ means that ***when*** top-of-stack is S and the next input token is a, we apply the 1st rule in Fig. 5.2; that is, the stack contents become "A 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.

Lookahead (Next input token)

Nonterminal symbol	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.



The LL(1) Parse Table Construction

- The procedure itself is shown in Fig. 5.9
- Input:
  - The target **CFGs**, *G*; we use the CFGs in Fig. 5.2 as example
  - The **productions** *p* for all the nonterminals defined in *G*
  - The **Predict set** of all the productions *p* as in Fig. 5.3
  - The two-dimensional **parsing table**, *LLtable*
- Output: the parsing table shown in Fig. 5.10
  - Upon the procedure's completion, any entry with 0 represents an error since it means a terminal symbol does not predict any production for the associated nonterminal

Rule Number	A	$X_1 \ldots X_m$	Predict Set
1	S	A C $	a,b,q,c,$
2	C	c	c
3		λ	d,$
4	A	a B C d	a
5		B Q	b,q,c,$
6	B	b B	b
7		λ	q,c,d,$
8	Q	q	q
9		λ	c,$

Modified Fig. 5.3: Predict calculation for the grammar of Fig. 5.2

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

Fig. 5.10: LL(1) table for grammar in Fig. 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.

The LL(1) Parse Table Construction (Cont'd)

- The procedure visits all of the productions p in G and every terminal a in **Predict(p)**

> for each nonterminal A in G
>   for each production p of the nonterminal A
>   for each terminal a in the predict set of p
> $LLtable[A][a]$ = **RULE_NUM(p)**

- NOTE:
  - In Fig. 5.3, **RULE_NUM(p)** = {1, …, 9}
  - p is a production ($X_1 \ldots X_m$) of nonterminal A
  - E.g., p could be (A C $), (a B C d), or (b B)

RULE_NUM(p) p

Rule Number	A	$X_1 \ldots X_m$	Predict Set
1	S	A C $	a,b,q,c,$
2	C	c	c
3		λ	d,$
4	A	a B C d	a
5		B Q	b,q,c,$
6	B	b B	b
7		λ	q,c,d,$
8	Q	q	q
9		λ	c,$

Modified Fig. 5.3: Predict calculation for the grammar of Fig. 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	a	b	c	d	q	$
	Lookahead					
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

Fig. 5.10: LL(1) table for grammar in Fig. 5.2

The LL(1) Parse Table Construction (Cont'd)

A how-to example

- When p is (B ⇒ b B)
  - Predict(p) = {b}
  - RULE_NUM(p) = 6
- We set LLtable[B][b] = 6
  - Be ware of the above **flow**

Nonterminal	a	b	c	d	q	$
	Lookahead					
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

Fig. 5.10: LL(1) table for grammar in Fig. 5.2

RULE_NUM(p)

Rule Number	A	$X_1 \ldots X_m$	Predict Set
1	S	A C $	a,b,q,c,$
2	C	c	c
3		λ	d,$
4	A	a B C d	a
5		B Q	b,q,c,$
6	B	b B	b
7		λ	q,c,d,$
8	Q	q	q
9		λ	c,$

Modified Fig. 5.3: Predict calculation for the grammar of Fig. 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.

Parsing Procedure for Generic LL(1) Parser

- To start the parsing procedure, we **call push(*S*)**
- Next, we do the following iteratively until **TOS() == $** (**Marker 8**)
  - If **TOS()** is a terminal symbol (**Marker 6**)
    - Call **match(*ts*, TOS())** to check if the symbols of *ts*.**peek()** and **TOS()** are the same; if so, **pop()** the top of the stack (**Marker 9**)
  - If **TOS()** is a nonterminal symbol (**Marker 10**)
    - Consult the parsing table and find the corresponding rule at the table entry (i.e., *LLtable*[**TOS()**, *ts*.**peek()**])
    - If the table entry is 0, raise Error
    - If the table entry is not 0, apply the rule

```
procedure LLPARSER( ts )
    call PUSH( S )
    accepted ← false
    while not accepted do                                          (5)
        if TOS( ) ∈ Σ                                              (6)
        then
            call MATCH( ts, TOS( ) )                               (7)
            if TOS( ) = $                                          (8)
            then  accepted ← true
            call POP( )                                            (9)
        else
            p ← LLtable[TOS( ), ts.PEEK( )]                        (10)
            if p = 0
            then
                call ERROR( Syntax error—no production applicable )
            else   call APPLY( p )
end
procedure APPLY( p : A → X_1 ... X_m )
    call POP( )                                                    (11)
    for i = m downto 1 do                                          (12)
        call PUSH( X_i )
end
```

Figure 5.8: Generic LL(1) parser.

Nonterminal symbol (TOS())	Lookahead (*ts*.peek()): 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

Fig. 5.10: LL(1) table for grammar in Fig. 5.2

match(*ts*, TOS()) is defined in Fig. 5.5

Example: Execution Trace of an LL(1) Parse

- Parse the input string: **a b b d c $**
  - Use the parser generated by the grammar in Fig. 5.2,
  - the Predict set in Fig. 5.3, and
  - the LL(1) parsing table in Fig. 5.10
  - Fig. 5.11 shows parsing trace (stack contents and applied rules)

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.

Rule Number	A	$X_1 \ldots X_m$	Predict Set
1	S	A C $	a,b,q,c,$
2	C	c	c
3		λ	d,$
4	A	a B C d	a
5		B Q	b,q,c,$
6	B	b B	b
7		λ	q,c,d,$
8	Q	q	q
9		λ	c,$

Modified Fig. 5.3: Predict calculation for the grammar of Fig. 5.2

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.

Parse Stack	Action	Remaining Input
S		abbdc$
	Apply 1: S→AC$	
$CA		abbdc$
	Apply 4: A→aBCd	
$CdCBa		abbdc$
	Match	
$CdCB		bbdc$
	Apply 6: B→bB	
$CdCBb		bbdc$
	Match	
$CdCB		bdc$
	Apply 6: B→bB	
$CdCBb		bdc$
	Match	
$CdCB		dc$
	Apply 7: B→λ	
$CdC		dc$
	Apply 3: C→λ	
$Cd		dc$
	Match	
$C		c$
	Apply 2: C→c	
$c		c$
	Match	
$		$
	Accept	

Figure 5.11: Trace of an LL(1) parse. The stack is shown in the left column, with top-of-stack as the rightmost character. The input string is shown in the right column, processed from left to right.

- **TOS() =** S, ts.**peek( ) = a**
  1. Apply Rule 1 ($LLtable[S, a] = 1$)
     Marker 10 in Fig. 5.8
  2. Do the actions on stack: **pop()** and **push(A C $)**

Step 1

Nonterminal	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

(Lookahead: columns a, b, c, d, q, $)

Fig. 5.10: LL(1) table for grammar in Fig. 5.2

Step 2

Rule Number	A	$X_1 \dots X_m$	Predict Set
1	S	→ A C $	a,b,q,c,$
2	C	c	c
3		λ	d,$
4	A	a B C d	a
5		B Q	b,q,c,$
6	B	b B	b
7		λ	q,c,d,$
8	Q	q	q
9		λ	c,$

Modified Fig. 5.3: Predict calculation for the grammar of Fig. 5.2

Parse Stack	Action	Remaining Input
S		abbdc$
	Apply 1: S→AC$	
$CA		abbdc$
	Apply 4: A→aBCd	
$CdCBa		abbdc$
	Match	
$CdCB		bbdc$
	Apply 6: B→bB	
$CdCBb		bbdc$
	Match	
$CdCB		bdc$
	Apply 6: B→bB	
$CdCBb		bdc$
	Match	
$CdCB		dc$
	Apply 7: B→λ	
$CdC		dc$
	Apply 3: C→λ	
$Cd		dc$
	Match	
$C		c$
	Apply 2: C→c	
$c		c$
	Match	
$		$
	Accept	

Figure 5.11: Trace of an LL(1) parse. The stack is shown in the left column, with top-of-stack as the rightmost character. The input string is shown in the right column, processed from left to right.

- **TOS() = A, ts.peek( ) = a**
  1. Apply Rule 4 ($LLtable[A, \text{a}] = 4$)
     Marker 10 in Fig. 5.8
  2. Do the actions on stack:
     pop() and **push(a B C d)**

Step 1

Nonterminal	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

(Lookahead: a b c d q $)

Fig. 5.10: LL(1) table for grammar in Fig. 5.2

Rule Number	A	$X_1 \dots X_m$	Predict Set
1	S	A C $	a,b,q,c,$
2	C	c	c
3		λ	d,$
4	A	a B C d	a
5		B Q	b,q,c,$
6	B	b B	b
7		λ	q,c,d,$
8	Q	q	q
9		λ	c,$

Modified Fig. 5.3: Predict calculation for the grammar of Fig. 5.2

Step 2

Parse Stack	Action	Remaining Input
S		abbdc$
	Apply 1: S$\to$AC$	
$CA		abbdc$
	Apply 4: A$\to$aBCd	
$CdCBa		abbdc$
	Match	
$CdCB		bbdc$
	Apply 6: B$\to$bB	
$CdCBb		bbdc$
	Match	
$CdCB		bdc$
	Apply 6: B$\to$bB	
$CdCBb		bdc$
	Match	
$CdCB		dc$
	Apply 7: B$\to\lambda$	
$CdC		dc$
	Apply 3: C$\to\lambda$	
$Cd		dc$
	Match	
$C		c$
	Apply 2: C$\to$c	
$c		c$
	Match	
$		$
	Accept	

Figure 5.11: Trace of an LL(1) parse. The stack is shown in the left column, with top-of-stack as the rightmost character. The input string is shown in the right column, processed from left to right.

- **TOS()** = a, *ts*.**peek**( ) = a
  - Call **match(** *ts*, **TOS())** Marker 7 in Fig. 5.8
  - Matched!!! **match()** will **advance** the *ts* once matched, which is defined in Fig. 5.5
  - Call pop()
  - You may parse the following input by yourself

Nonterminal	a	b	c	d	q	$
	Lookahead					
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

Fig. 5.10: LL(1) table for grammar in Fig. 5.2

Rule Number	A	$X_1 \ldots X_m$	Predict Set
1	S	A C $	a,b,q,c,$
2	C	c	c
3		λ	d,$
4	A	a B C d	a
5		B Q	b,q,c,$
6	B	b B	b
7		λ	q,c,d,$
8	Q	q	q
9		λ	c,$

Modified Fig. 5.3: Predict calculation for the grammar of Fig. 5.2

Parse Stack	Action	Remaining Input
S		abbdc$
	Apply 1: S→AC$	
$CA		abbdc$
	Apply 4: A→aBCd	
$CdCBa		abbdc$
	Match	
$CdCB		bbdc$
	Apply 6: B→bB	
$CdCBb		bbdc$
	Match	
$CdCB		bdc$
	Apply 6: B→bB	
$CdCBb		bdc$
	Match	
$CdCB		dc$
	Apply 7: B→λ	
$CdC		dc$
	Apply 3: C→λ	
$Cd		dc$
	Match	
$C		c$
	Apply 2: C→c	
$c		c$
	Match	
$		$
	Accept	

Figure 5.11: Trace of an LL(1) parse. The stack is shown in the left column, with top-of-stack as the rightmost character. The input string is shown in the right column, processed from left to right.



Obtaining LL(1) Grammars

- It can be difficult for inexperienced compiler writers to create LL(1) grammars
  - because LL(1) requires **a unique prediction** for each combination of nonterminal and lookahead symbols
  - It is easy to write productions that violate this requirement
- Two common types for LL(1) prediction conflicts
  - **common prefixes** and **left recursion**
- We introduce simple grammar transformations
  - that **eliminate ambiguity** caused by common prefixes and left recursion, and
  - these transformations allow us to obtain LL(1) form for most CFGss
  - However, there are some languages of interest for which no LL(1) grammar can be constructed; refer to Sec. 5.6 for more information

Common Prefixes

- Two productions for the same nonterminal share a **common prefix**
  - if the productions' RHSs **begin with the same string of grammar symbols**

Taking the grammar in Fig. 5.12 as an example

- Both **Stmt** productions are predicted by the **if** token (ambiguous!)
- Even if we allow greater lookahead, the ***else*** that distinguishes the two productions can lie arbitrarily far ahead in the input
  - I.e., Expr and StmtList can each generate a terminal string larger than any constant k

→ Grammar shown in Fig. 5.12 is not LL(k) for any k

1	Stmt	→	if Expr then StmtList endif
2		|	if Expr then StmtList else StmtList endif
3	StmtList	→	StmtList ; Stmt
4		|	Stmt
5	Expr	→	var + Expr
6		|	var

Figure 5.12: A grammar with common prefixes.

Eliminating Common Prefixes w/ Factoring

- Simplified steps for Fig. 5.13:
  - Find the **longest common prefixes** α of the productions **A**,
  - expand the productions **A** to **A’**, which is a new nonterminal, &
  - replace what follows α of original productions with **A’**

An simple example:

$A \Rightarrow \alpha\beta_1 \mid \alpha\beta_2$

Written productions:

$A \Rightarrow \alpha A'$

$A' \Rightarrow \beta_1 \mid \beta_2$

1	Stmt	→	if Expr then StmtList endif
2		|	if Expr then StmtList else StmtList endif
3	StmtList	→	StmtList ; Stmt
4		|	Stmt
5	Expr	→	var + Expr
6		|	var

Figure 5.12: A grammar with common prefixes.

```
procedure FACTOR( )
    foreach A ∈ N do
        α ← LongestCommonPrefix(ProductionsFor(A))
        while |α| > 0 do
            V ← new NonTerminal( )
            Productions ← Productions ∪ { A→αV }
            foreach p ∈ ProductionsFor(A) | RHS(p) = αβp do        (13)
                Productions ← Productions − { p }
                Productions ← Productions ∪ { V→βp }
            α ← LongestCommonPrefix(ProductionsFor(A))
end
```

Figure 5.13: Factoring common prefixes.



Results of Factoring

- **Rewrite the productions** with common prefixes
  - to defer the decision until enough of the input has bee seen that we can make the right choice
  - Fig. 5.14 shows the rewritten grammar for Fig. 5.12

1	Stmt	$\rightarrow$	if Expr then StmtList endif
2		$\mid$	if Expr then StmtList else StmtList endif
3	StmtList	$\rightarrow$	StmtList ; Stmt
4		$\mid$	Stmt
5	Expr	$\rightarrow$	var + Expr
6		$\mid$	var

Figure 5.12: A grammar with common prefixes.

1	Stmt	$\rightarrow$	if Expr then StmtList V_1
2	V_1	$\rightarrow$	endif
3		$\mid$	else StmtList endif
4	StmtList	$\rightarrow$	StmtList ; Stmt
5		$\mid$	Stmt
6	Expr	$\rightarrow$	var V_2
7	V_2	$\rightarrow$	+ Expr
8		$\mid$	λ

Figure 5.14: Factored version of the grammar in Figure 5.12.



Left Recursion

- A production is **left recursive**
  - if its **LHS symbol** is also the **first symbol of its RHS**
  - Example: **A** ⇒ **A**α | β
  - Also, in Figure 5.14, the production **StmtList→StmtList** ; **Stmt** is left-recursive

Left Recursive Grammar

- Grammars with **left-recursive productions** can never be LL(1)
  - With recursive-descent parsing, the application of this production **A $\Rightarrow$ Aα** will cause **procedure A to be invoked repeatedly** without advancing the input
    - With the state of the parse unchanged, this behavior will continue indefinitely
  - Similarly, with table-driven parsing, application of this production will **repeatedly push Aα on the stack without advancing the input**



Left Recursion Example

- Consider the following left-recursive rules
  1. A → A α
  2. | β

- Observations:
  - The rules produce strings like β α α
  - Each time Rule 1 is applied, an α is generated
  - The recursion ends when Rule 2 prepends a β to the string of α symbols
  - Using the regular-expression notation, the grammar generates βα*

→ That is, the β is generated first, and α symbols are then generated via right recursion



Left Recursion Example

- Consider the following left-recursive rules
  1. $A \rightarrow A\ \alpha$
  2. $\quad\quad |\ \beta$

- We can rewrite the grammar to:
  1. $A \rightarrow X\ Y$
  2. $X \rightarrow \beta$
  3. $Y \rightarrow \alpha\ Y$
  4. $\quad\quad |\ \lambda$

- Furthermore,
  - The rules also produce strings like $\beta\ \alpha\ \alpha$
  - The EliminateLeftRecursion algorithm is shown in Fig. 5.15
  - Applying it to the grammar in Fig. 5.14 results in Fig. 5.16



Another Left Recursion Example

- We trace the algorithm in Fig. 5.15 with producitons:

  (4) StmtList → StmtList ; Stmt

  (5) | Stmt

- Consider the nonterminal **StmtList**, we have **(4) StmtList → StmtList ; Stmt**
- Because RHS(**(4)**) = **StmtList** α, Rule **(4)** (Marker 1) is left-recursive production (Note: α is "**; Stmt**", r is **(4)**)
- Create two non-terminals X and Y
- Select Rule **(4)** (Note: p is **(4)**) (Marker 2)
  - As p == r == **(4)**, (Marker 3)
    Create the production **StmtList → X Y** (Marker 4)
- Select Rule **(5)** (Note: p is **(5)**) (Marker 2)
  - As p != r, (Marker 3)
    Create the production **X → Stmt** (Marker 5)
- Finally, run out of production rules (Marker 6)
  - Create **Y → ; Stmt Y** and **Y → λ**

```
procedure EliminateLeftRecursion( )
    foreach A ∈ N do
  ①  if ∃ r ∈ ProductionsFor(A) | RHS(r) = Aα
      then
          X ← new NonTerminal( )
          Y ← new NonTerminal( )
  ②      foreach p ∈ ProductionsFor(A) do
  ③          if p = r
  ④          then Productions ← Productions ∪ { A → X Y }
  ⑤          else Productions ← Productions ∪ { X → RHS(p) }
  ⑥      Productions ← Productions ∪ { Y → αY, Y → λ }
end
```

Figure 5.15: Eliminating left recursion.

Another Left Recursion Example (Cont'd)

- We trace the algorithm in Fig. 5.15 with producitons:
  - **(4) StmtList → StmtList ; Stmt**
  - **(5) | Stmt**
- Consider the nonterminal **StmtList**, we have **(4) StmtList → StmtList ; Stmt**
- Because RHS(**(4)**) = **StmtList** α, Rule **(4)** is left-recursive production (Note: α is "**; Stmt**", ***r*** is **(4)**)
- Create two non-terminals X and Y
- Select Rule **(4)** (Note: ***p*** is **(4)**)
  - As ***p*** == ***r*** == **(4)**, Create the production **StmtList → X Y**
- Select Rule **(5)** (Note: ***p*** is **(5)**)
  - As ***p*** != ***r***, Create the production **X → Stmt**
- Finally, run out of production rules
  - Create **Y → ; Stmt Y** and **Y → λ**

```
procedure EliminateLeftRecursion( )
  foreach A ∈ N do
  1 if ∃ r ∈ ProductionsFor(A) | RHS(r) = Aα
    then
       X ← new NonTerminal( )
       Y ← new NonTerminal( )
  2    foreach p ∈ ProductionsFor(A) do
  3       if p = r
  4       then Productions ← Productions ∪ { A → X Y }
  5       else Productions ← Productions ∪ { X → RHS(p) }
  6    Productions ← Productions ∪ { Y → αY, Y → λ }
end
```

Figure 5.15: Eliminating left recursion.

1	Stmt	→	if Expr then StmtList V_1
2	V_1	→	endif
3		|	else StmtList endif
4	StmtList	→	X Y
5	X	→	Stmt
6	Y	→	; Stmt Y
7		|	λ
8	Expr	→	var V_2
9	V_2	→	+ Expr
10		|	λ

Figure 5.16: LL(1) version of the grammar in Figure 5.14.



You Are Suggested to …

- Read Sec. 5.6 if you would like to know why the grammar in Fig. 5.12 is not LL(k)
- Read Sec. 5.7 for more about the properties of LL(1) parsers;
  Hint: a good summary of what we learn in this chapter
- Read Section 5.8 for more about the parser table
- Read Sec. 5.9 for error recorvery



QUESTIONS?