Chapter 4 Grammars and Parsing

Context-Free Grammars: Concepts and Notation

- A context-free grammar G = (Vt, Vn, S, P)
  - A finite terminal vocabulary Vt
    - The token set produced by scanner
  - A finite set of nonterminal vacabulary Vn
    - Intermediate symbols
  - A start symbol S $\in$Vn that starts all derivations
    - Also called goal symbol
  - P, a finite set of productions (rewriting rules) of the form $A \rightarrow X_1X_2 \ldots X_m$
    - $A \in Vn$, $X_i \in Vn \cup Vt$, $1 \leq i \leq m$
    - $A \rightarrow \lambda$ is a valid production

Context-Free Grammars: Concepts and Notation (Cont'd)

- Other notations
  - Vacabulary V of G,
    - $V = V_n \cup V_t$
  - L(G), the set of string s derivable from S
    - Context-free language of grammar G
  - Notational conventions
    - a,b,c, … denote symbols in V_t
    - A,B,C, … denote symbols in V_n
    - U,V,W, … denote symbols in V
    - $\alpha,\beta,\gamma,\ldots$ denote strings in V^*
    - u,v,w, … denote strings in V_t^*

Context-Free Grammars: Concepts and Notation (Cont'd)

- Derivation
  - One step derivation
    - If $A \rightarrow \gamma$, then $\alpha A \beta \Rightarrow \alpha \gamma \beta$
  - One or more steps derivation $\Rightarrow^+$
  - Zero or more steps derivation $\Rightarrow^*$
- If $S \Rightarrow^* \beta$, then β is said to be sentential form of the CFG
  - SF(G) is the set of sentential forms of grammar G
- $L(G) = \{x \in V_t^* | S \Rightarrow^+ x\}$
  - $L(G) = SF(G) \cap V_t^*$

Context-Free Grammars: Concepts and Notation (Cont'd)

- Left-most derivation, a top-down parsers
  - $\Rightarrow_{lm}$, $\Rightarrow_{lm}$, $\overset{+}{\Rightarrow}_{lm}$ $\overset{*}{}$
  - E.g. of leftmost derivation of F(V+V)

G_0

$E \rightarrow Prefix(E)$
$E \rightarrow V\ Tail$
$Prefix \rightarrow F$
$Prefix \rightarrow \lambda$
$Tail \rightarrow +E$
$Tail \rightarrow \lambda$

$E \Rightarrow_{lm} Prefix(E)$
$\Rightarrow_{lm} F(E)$
$\Rightarrow_{lm} F(V\ Tail)$
$\Rightarrow_{lm} F(V+E)$
$\Rightarrow_{lm} F(V+V\ Tail)$
$\Rightarrow_{lm} F(V+V)$

Context-Free Grammars: Concepts and Notation (Cont'd)

- Right-most derivation (canonical derivation)
  - $\Rightarrow_{rm}$, $\Rightarrow_{rm}$, $\overset{+}{\Rightarrow}_{rm}$ *
  - Buttom-up parsers
  - E.g. of leftmost derivation of F(V+V)

G_0

$E \rightarrow Prefix(E)$
$E \rightarrow V\ Tail$
$Prefix \rightarrow F$
$Prefix \rightarrow \lambda$
$Tail \rightarrow +E$
$Tail \rightarrow \lambda$

$E \Rightarrow_{rm} Prefix(E)$
$\Rightarrow_{rm} Prefix(V\ Tail)$
$\Rightarrow_{rm} Prefix(V+E)$
$\Rightarrow_{rm} Prefix(V+V\ Tail)$
$\Rightarrow_{rm} Prefix(V+V)$
$\Rightarrow_{rm} F(V+V)$

Same # of steps, but different order

Context-Free Grammars: Concepts and Notation (Cont'd)

- A parse tree
  - rooted by the start symbol
  - Its leaves are grammar symbols or λ

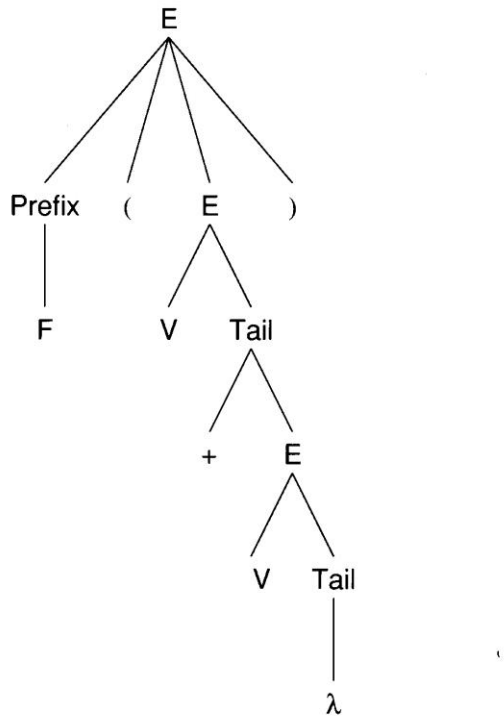


Figure 4.1 The Parse Tree Corresponding to F(V+V)

Context-Free Grammars: Concepts and Notation (Cont'd)

- A ***phase*** of a sentential form is a sequence of symbols descended from a single nonterminal in the parse tree
  - Simple or prime phrase
- The ***handle*** of a sentential form is the left-most simple phrase

Context-Free Grammars: Concepts and Notation (Cont'd)

- ***Regular grammars***
  - is of CFGs
  - Limited to productions of the form

    $A \rightarrow aB$

    $C \rightarrow \lambda$
  - See exercise 6
- The ***handle*** of a sentential form is the left-most simple phrase

Errors in Context-Free Grammars

- CFGs are a definitional mechanism. They may have errors, just as programs may.
- Flawed CFG
  1. Useless nonterminals
     - Unreachable
     - Derive no terminal string

$S \rightarrow A|B$
$A \rightarrow a$
$B \rightarrow Bb$
$C \rightarrow c$

Nonterminal C cannot be reached form S
Nonterminal B derives no terminal string

S is the start symbol.

Do exercise 7.

Errors in Context-Free Grammars

- Ambiguous:
  - Grammars that allow different parse trees for the same terminal string
- It is impossible to decide whether a given CFG is ambiguous

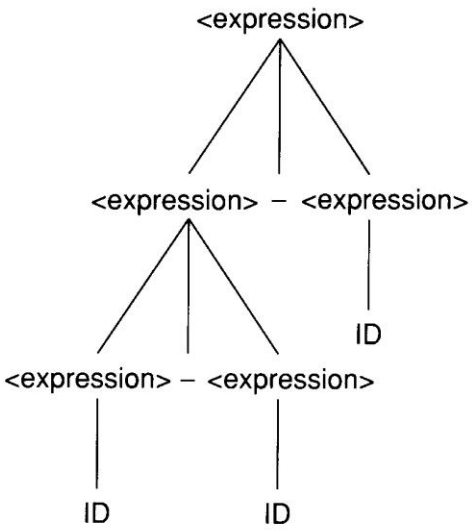


Figure 4.2 A Parse Tree for ID–ID–ID

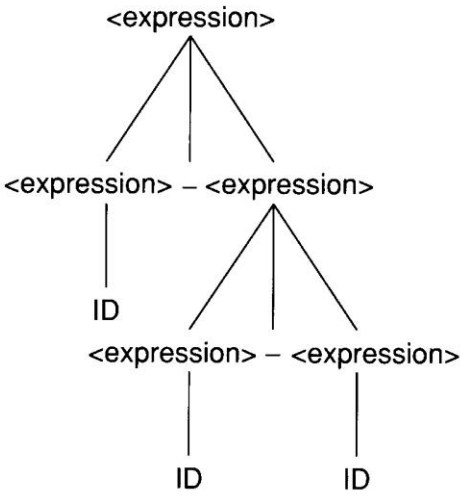


Figure 4.3 An Alternate Parse Tree for ID–ID–ID

Errors in Context-Free Grammars

- It is impossible to decide whether a given CFG is ambiguous
  - For certain grammar classes, we can prove that constituent grammars are unambiguous
- Wrong language
- A general comparison algorithm applicable to all CFGs is known to be impossible

Transforming Extened BNF Grammars

- Extended BNF≡ BNF
  - Extended BNF allows
    - Square bracket []
    - Optional list {}

```
for (each production P = A → α [X1 ... Xn] β) {
    Create a new nonterminal, N.
    Replace production P with P′ = A → α N β
    Add the productions: N → X1 ... Xn and N → λ
}

for (each production Q = B → γ {Y1 ... Ym} δ) {
    Create a new nonterminal, M.
    Replace production Q with Q′ = B → γ M δ
    Add the productions: M → Y1 ... Ym M and
                         M → λ
}
```

Figure 4.4 Algorithm to Transform Extended BNF Grammars into Standard Form

Parsers and Recognizers

- Recognizer
  - An algorithm that does boolean-valued test
    - “Is this input syntactically valid?
- Parser
  - Answers more general questions
    - Is this input valid?
    - And, if it is, what is its structure (parse tree)?

Parsers and Recognizers (Cont'd)

- Two general approaches to parsing
  - Top-down parser
    - Expanding the parse tree (via predictions) in a depth-first manner
    - Preorder traversal of the parse tree
    - ***Predictive*** in nature
    - lm
    - LL

Parsers and Recognizers (Cont'd)

- Buttom-down parser
  - Beginning at its bottom (the leaves of the tree, which are terminal symbols) and determining the productions used to generate the leaves
  - Postorder traversal of the parse tree
  - rm
  - LR

Parsers and Recognizers (Cont'd)

<Program> → **begin** <Stmts> **end** $
<Stmts> → <Stmt> ; <Stmts>
<Stmts> → λ
<Stmt> → SimpleStmt
<Stmt> → **begin** <Stmts> **end**

Grammar G_3

To parse

begin SimpleStmt; SimpleStmt; **end** $

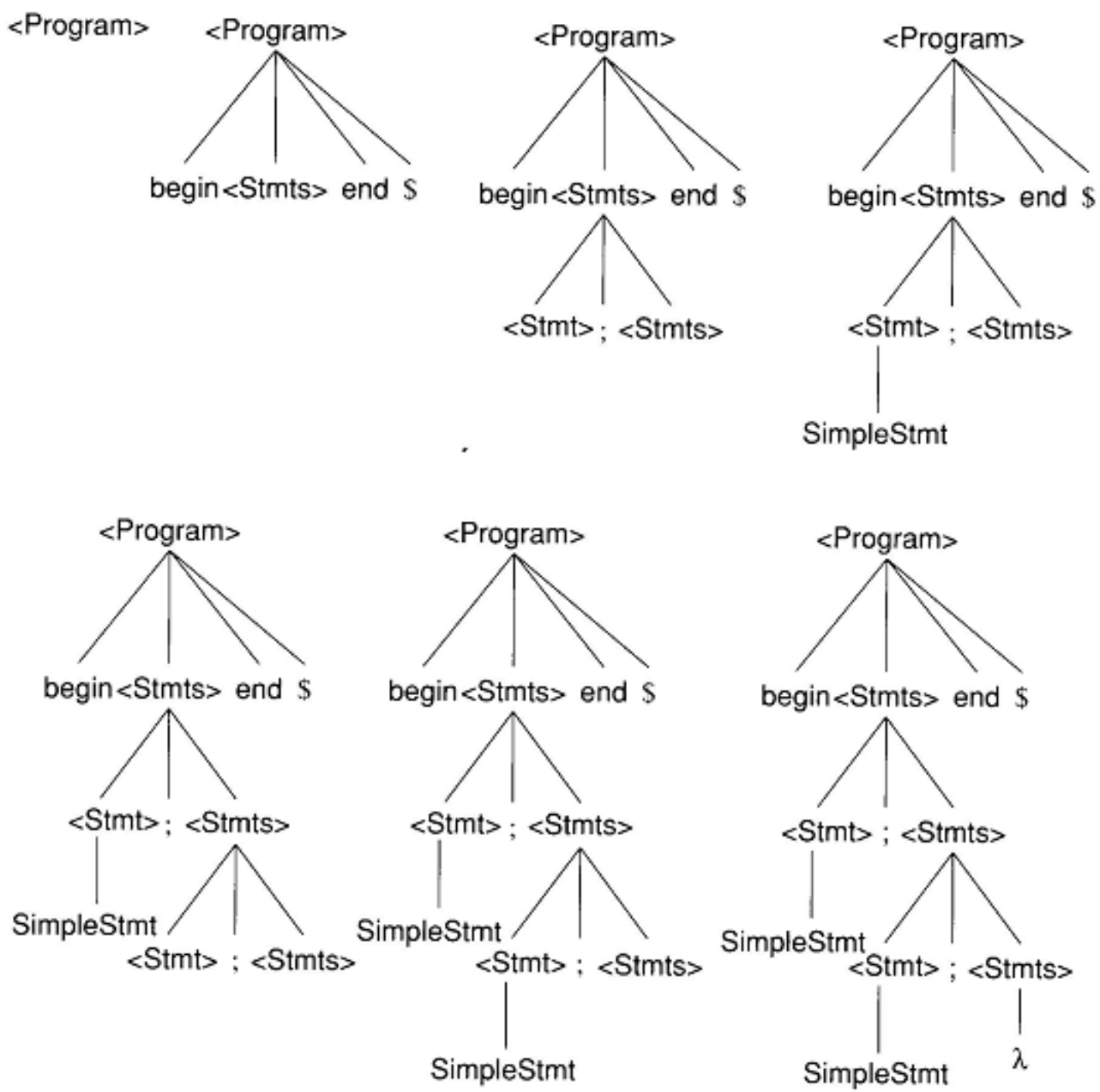


Figure 4.5 A Top-Down Parse

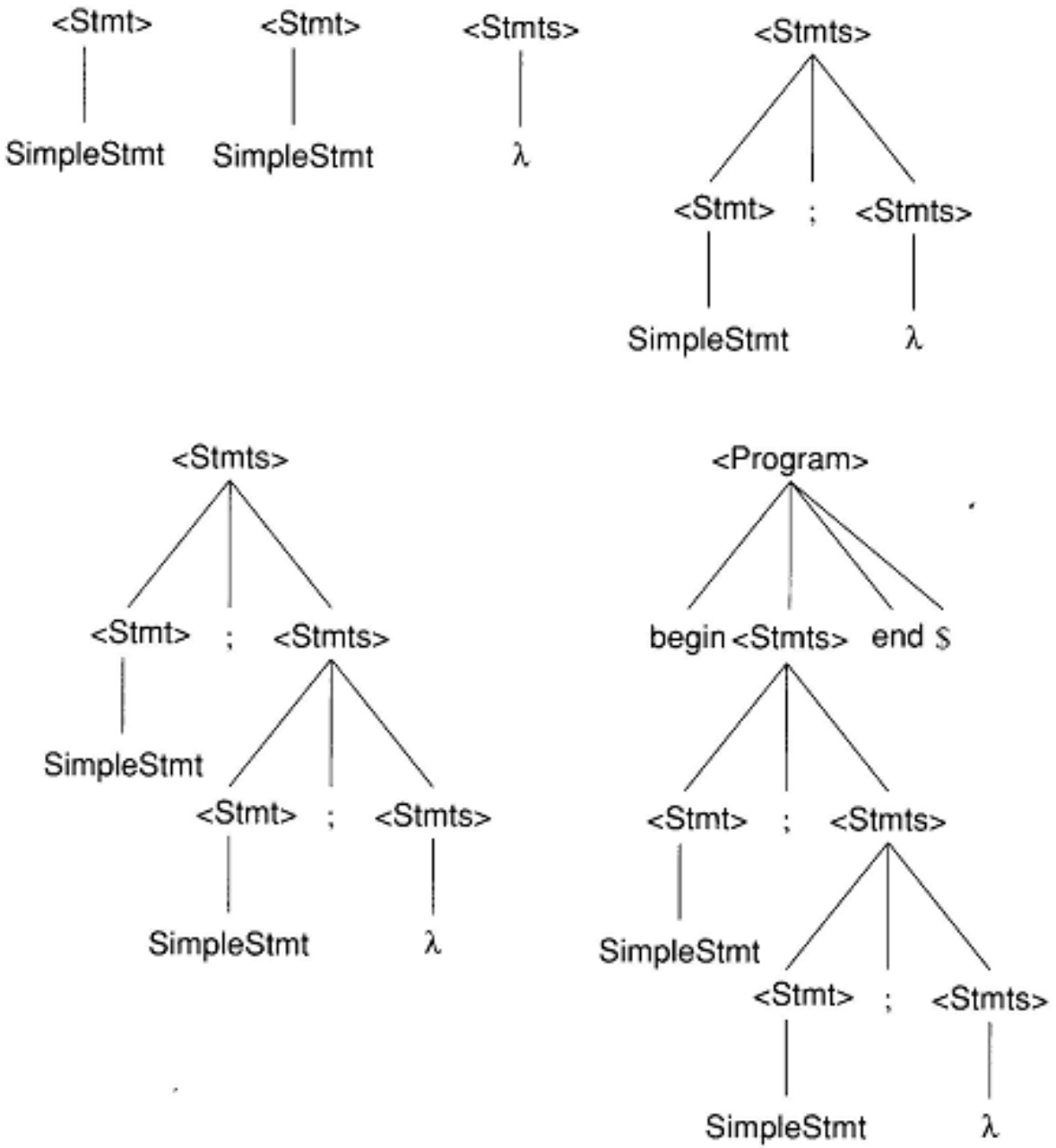
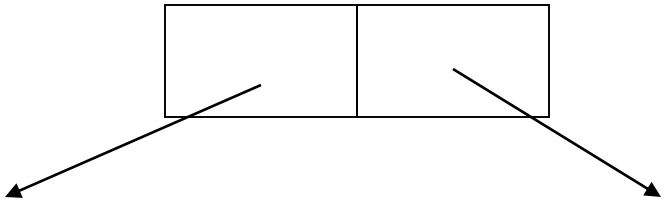


Figure 4.6 A Bottom-Up Parse

Parsers and Recognizers (Cont'd)

- Naming of parsing techniques



The way to parse token sequence

L: Leftmost
R: Righmost

- Top-down
  - LL
- Bottom-up
  - LR

Grammar Analysis Algorithms

- Goal of this section:
  - Discuss a number of important analysis algorithms for Grammars

Grammar Analysis Algorithms (Cont'd)

- The data structure of a grammar G

```
typedef int symbol;   /* a symbol in the grammar */
/*
 * The symbolic constants used below, NUM_TERMINALS,
 * NUM_NONTERMINALS, and NUM_PRODUCTIONS are
 * determined by the grammar.  MAX RHS LENGTH should
 * simply be "big enough."
 */
#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];
} grammar;

typedef struct prod production;

typedef symbol terminal;
typedef symbol nonterminal;
```

Grammar Analysis Algorithms (Cont'd)

- What nonterminals can derive λ?

  $A \Rightarrow BCD \Rightarrow BC \Rightarrow B \Rightarrow \lambda$

  – An iterative marking algorithm

```
typedef short boolean;
typedef boolean marked_vocabulary[VOCABULARY];

/*
 * Mark those vocabulary symbols found to
 * derive λ (directly or indirectly).
 */
marked_vocabulary mark_lambda(const grammar g)
{
   static marked_vocabulary derives_lambda;
   boolean changes;
                  /* any changes during last iteration? */
   boolean rhs_derives_lambda;
                  /* does the RHS derive λ? */
   symbol v;      /* a word in the vocabulary */
   production p;  /* a production in the grammar */
   int i, j;      /* loop variables */

   for (v = 0; v < VOCABULARY; v++)
      derives_lambda[v] = FALSE;
      /* initially, nothing is marked */

   do {
      changes = FALSE;
      for (i = 0; i < g.num_productions; i++) {
         p = g.productions[i];
         if (! derives_lambda[p.lhs]) {
            if (p.rhs_length == 0) {
               /* derives λ directly */
               changes = derives_lambda[p.lhs] = TRUE;
               continue;
            }
            /* does each part of RHS derive λ? */
            rhs_derives_lambda = derives_lambda[p.rhs[0]];
            for (j = 1; j < p.rhs_length; j++)
               rhs_derives_lambda = rhs_derives_lambda
                           && derives_lambda[p.rhs[j]];

            if (rhs_derives_lambda)
               changes = TRUE;
               derives_lambda[p.lhs] = TRUE;
         }
      }
   } while (changes);
   return derives_lambda;
}
```


Figure 4.7 Algorithm for Determining If a Nonterminal Can Derive λ

Grammar Analysis Algorithms (Cont'd)

- Follow(A)
  - A is any nonterminal
  - Follow(A) is the set of terminals that my follow A in some sentential form

  $\text{Follow}(A)=\{a \in V_t \mid S \Rightarrow^* \ldots Aa \ldots \} \cup$

  $\{\text{if } S \Rightarrow^+ \alpha A \text{ then } \{\lambda\} \text{ else } \varnothing\}$

- First(α)
  - The set of all the terminal symbols that can begin a sentential form derivable from α
  - If α is the right-hand side of a production, then First(α) contains terminal symbols that begin strings derivable from α

  $\text{First}(\alpha)=\{a \in V_t \mid \alpha \Rightarrow^* a\beta\} \cup$

  $\{\text{if } \alpha \Rightarrow^* \lambda \text{ then } \{\lambda\} \text{ else } \varnothing\}$

Grammar Analysis Algorithms (Cont'd)

- Definition of C data structures and subroutines
  - first_set[X]
    - contains terminal symbols and λ
    - X is any single vocabulary symbol
  - follow_set[A]
    - contains terminal symbols and λ
    - A is a nonterminal symbol

```
typedef set_of_terminal_or_lambda termset;
termset follow_set[NUM_NONTERMINAL];
termset first_set[SYMBOL];
marked_vocabulary derives_lambda = mark_lambda(g);
/* mark_lambda(g) as defined above */

termset compute_first(string_of_symbols alpha)
{
 int i, k;
 termset result;

 k = length(alpha);
 if (k == 0)
    result = SET_OF( λ );
 else {
   result = first_set[alpha[0]];
   for (i = 1; i < k && λ ∈ first_set[alpha[i-1]]; i++)
     result = result ∪ (first_set[alpha[i]] - SET_OF( λ ));

   if (i == k && λ ∈ first_set[alpha[k - 1]])
     result = result ∪ SET_OF( λ );
 }
 return result;
}
```

It is a subroutine of fill_first_set()

Figure 4.8 Algorithm to Compute First(alpha)

```
extern grammar g;

void fill_first_set(void)
{
   nonterminal A;
   terminal    a;
   production  p;
   boolean     changes;
   int         i, j;

   for (i = 0; i < NUM_NONTERMINAL; i++) {
      A = g.nonterminals[i];
      if (derives_lambda[A])
         first_set[A] = SET_OF( λ );
      else
         first_set[A] = ∅;
   }
   for (i = 0; i < NUM_TERMINAL; i++) {
      a = g.terminals[i];
      first_set[a] = SET_OF( a );
      for (j = 0; j < NUM_NONTERMINAL; j++) {
         A = g.nonterminals[j];
         if (there exists a production A → aβ)
            first_set[A] = first_set[A] ∪ SET_OF( a );
      }
   }
   do {
      changes = FALSE;
      for (i = 0; i < g.num_productions; i++) {
         p = g.productions[i];
         first_set[p.lhs] = first_set[p.lhs] ∪
            compute_first(p.rhs);
         if ( first_set changed )
            changes = TRUE;
      }
   } while (changes);
}
```


Figure 4.9 Algorithm to Compute First Sets for V

$$G_0 \begin{cases} E \rightarrow Prefix(E) \\ E \rightarrow V\ Tail \\ Prefix \rightarrow F \\ Prefix \rightarrow \lambda \\ Tail \rightarrow +E \\ Tail \rightarrow \lambda \end{cases}$$

The execution of fill_first_set() using grammar G_0

Step	first_set							
	E	Prefix	Tail	(	)	V	F	+
(1) First loop	∅	{λ}	{λ}					
(2) Second (nested) loop	{V}	{F,λ}	{+,λ}	{(}	{)}	{V}	{F}	{+}
(3) Third loop, production 1	{V,F,(}	{F,λ}	{+,λ}	{(}	{)}	{V}	{F}	{+}

```
void fill_follow_set(void)
{
   nonterminal A, B;
   int i;
   boolean     changes;

   for (i = 0; i < NUM_NONTERMINAL; i++) {
      A = g.nonterminals[i];
      follow_set[A] = ∅;
   }
   follow_set[g.start_symbol] = SET_OF( λ );

   do {
      changes = FALSE;
      for (each production A → αBβ) {
         /*
          * I.e. for each production and each occurrence
          * of a nonterminal in its right-hand side.
          */
         follow_set[B] = follow_set[B] ∪
                (compute_first(β) - SET_OF( λ ));
         if ( λ ∈ compute_first(β) )
            follow_set[B] = follow_set[B] ∪ follow_set[A];
         if ( follow_set[B] changed )
            changes = TRUE;
      }
   } while (changes);
}
```

Figure 4.10 Algorithm to Compute Follow Sets for All Nonterminals

$$G_0 \begin{cases} E \rightarrow Prefix(E) \\ E \rightarrow V\ Tail \\ Prefix \rightarrow F \\ Prefix \rightarrow \lambda \\ Tail \rightarrow +E \\ Tail \rightarrow \lambda \end{cases}$$

The execution of fill_follow_set() using grammar G_0

Step	follow_set		
	E	Prefix	Tail
(1) Initialization	{λ}	∅	∅
(2) Process Prefix in production 1	{λ}	{(}	∅
(3) Process E in production 1	{λ,)}	{(}	∅
(4) Process Tail in production 2	{λ,)}	{(}	{λ,)}

More examples

$S \rightarrow aSe$
$S \rightarrow B$
$B \rightarrow bBe$
$B \rightarrow C$
$C \rightarrow cCe$
$C \rightarrow d$

The execution of fill_first_set()

Step	first_set							
	S	B	C	a	b	c	d	e
(1) First loop	∅	∅	∅					
(2) Second (nested) loop	{a}	{b}	{c,d}	{a}	{b}	{c}	{d}	{e}
(3) Third loop, production 2	{a,b}	{b}	{c,d}	{a}	{b}	{c}	{d}	{e}
(4) Third loop, production 4	{a,b}	{b,c,d}	{c,d}	{a}	{b}	{c}	{d}	{e}
(5) Third loop, production 2	{a,b,c,d}	{b,c,d}	{c,d}	{a}	{b}	{c}	{d}	{e}

The execution of fill_follow_set()

Step	follow_set		
	S	B	C
(1) Initialization	{λ}	∅	∅
(2) Process S in production 1	{e,λ}	∅	∅
(3) Process B in production 2	{e,λ}	{e,λ}	∅
(4) Process B in production 3	no changes		
(5) Process C in production 4	{e,λ}	{e,λ}	{e,λ}
(6) Process C in production 5	no changes		

More examples

$S \rightarrow ABc$
$A \rightarrow a$
$A \rightarrow \lambda$
$B \rightarrow b$
$B \rightarrow \lambda$

The execution of fill_first_set()

Step	first_set					
	S	A	B	a	b	c
(1) First loop	∅	{λ}	{λ}			
(2) Second (nested) loop	∅	{a,λ}	{b,λ}	{a}	{b}	{c}
(3) Third loop, production 1	{a,b,c}	{a,λ}	{b,λ}	{a}	{b}	{c}

The execution of fill_follow_set()

Step	follow_set		
	S	A	B
(1) Initialization	{λ}	∅	∅
(2) Process A in production 1	{λ}	{b,c}	∅
(3) Process B in production 1	{λ}	{b,c}	{c}