Chapter 4:

Grammars and Parsing

陳奇業 成功大學資訊工程系

Parsing: Syntax Analysis

- decides which part of the incoming token stream should be grouped together.
- the output of parsing is some representation of a parse tree.
- intermediate code generator transforms the parse tree into an intermediate language.

Comparisons between regular expressions and context-free grammars

- A context-free grammar:

$$exp \rightarrow \text{exp}\, op\, \text{exp} \,|\, (exp) \,|\, number$$
$$op \rightarrow +|-|*$$

- A regular expression:

$$number = digit\ digit^*$$
$$digit = 0|1|2|3|4|5|6|7|8|9$$

The major difference is that the rules of a context-free grammar are **recursive**.

Rules from F.A.(r.e.) to CFG

1. For each state there is a nonterminal symbol.
2. If state A has a transition to state B on symbol a, introduce $A \to aB$.
3. If A goes to B on input λ, introduce $A \to B$.
4. If A is an accepting state, introduce $A \to \lambda$.
5. Make the start state of the NFA be the start symbol of the grammar.

Examples

(1) r.e.: $(a|b)(a|b|0|1)^*$

c.f.g.: $S \to aA|bA \quad A \to aA|bA|0A|1A|\lambda$

(2) r.e.: $(a|b)^*abb$

c.f.g.: $S \to aS \mid bS \mid aA$

$A \to bB$

$B \to bC$

$C \to \lambda$

Why don’t we use c.f.g. to replace r.e. ?

- r.e. => easy & clear description for token.
- r.e. => efficient token recognizer
- modularizing the components (The grammar rules use regular expressions as components)

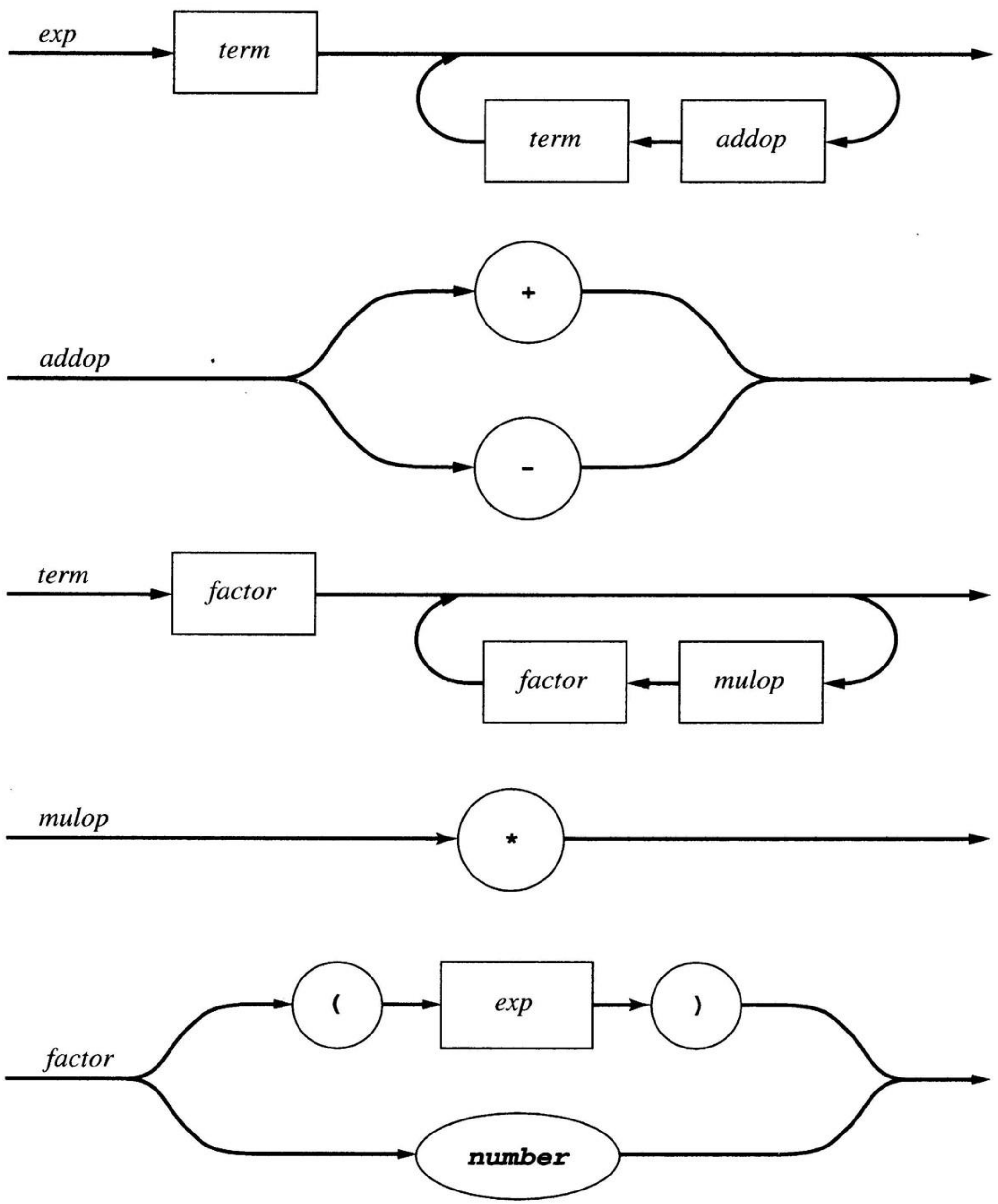
Features of programming languages

- contents:
  - declarations
  - sequential statements
  - iterative statements
  - conditional statements

Description of programming languages

- Syntax Diagrams
- Context Free Grammars (CFG)

Figure 3.4
Syntax diagrams for the grammar of Example 3.10



§

Contex Free Grammar (in BNF)

exp → exp addop term | term

addop → + | -

term → term mulop factor | factor

mulop → *

factor → (exp) | number

History

- In 1956 BNF (Backus Naur Form：巴科斯-諾爾範式) is used for description of natural language.
- The Syntactic Specification of Programming Languages - CFG (a BNF description)

Capabilities of Context-free grammars

- give precise syntactic specification of programming languages
- a parser can be constructed automatically by CFG
- the syntax entity specified in CFG can be used for translating into object code.
- useful for describing nested structures such as balanced parentheses, matching begin-end's, corresponding if-then-else, etc.

Context-Free Grammars: Concepts and Notation

- A context-free grammar $G = (V_t, V_n, S, P)$
  - A finite terminal vocabulary V_t
    - The token set produced by scanner
  - A finite set of nonterminal vocabulary V_n
    - Intermediate symbols
- A start symbol $S \in V_n$ that starts all derivations
  - Also called goal symbol
- P, a finite set of productions (rewriting rules) of the form $A \rightarrow X_1 X_2 \ldots X_m$
  - $A \in V_n, X_i \in V_n \cup V_t, 1 \leq i \leq m$
  - $A \rightarrow \lambda$ is a valid production

Context-Free Grammars: Concepts and Notation

- $G = (\{+, *, (,), number\}, \{\exp, op\}, \exp, P)$
- P: {exp → exp op exp | (exp) | number, op → + | − | ∗}

Figure 3.1
A derivation for the arithmetic expression
(34-3)*42

(number-number)*number

(1) $exp \Rightarrow exp\ op\ exp$		$[exp \rightarrow exp\ op\ exp]$
(2)	$\Rightarrow exp\ op\ \mathbf{number}$	$[exp \rightarrow \mathbf{number}]$
(3)	$\Rightarrow exp\ *\ \mathbf{number}$	$[op \rightarrow \mathbf{*}\ ]$
(4)	$\Rightarrow \mathbf{(}\ exp\ \mathbf{)}\ \mathbf{*}\ \mathbf{number}$	$[exp \rightarrow \mathbf{(}\ exp\ \mathbf{)}]$
(5)	$\Rightarrow \mathbf{(}\ exp\ op\ exp\ \mathbf{)}\ \mathbf{*}\ \mathbf{number}$	$[exp \rightarrow exp\ op\ exp]$
(6)	$\Rightarrow \mathbf{(}exp\ op\ \mathbf{number)}\ \mathbf{*}\ \mathbf{number}$	$[exp \rightarrow \mathbf{number}]$
(7)	$\Rightarrow \mathbf{(}exp\ \mathbf{-}\ \mathbf{number)}\ \mathbf{*}\ \mathbf{number}$	$[op \rightarrow\ \mathbf{-}\ ]$
(8)	$\Rightarrow \mathbf{(number - number)*number}$	$[exp \rightarrow \mathbf{number}]$

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

- Other notations
  - Vocabulary 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
  - $a, b, c, \ldots$ denote symbols in V_t
  - $A, B, C, \ldots$ denote symbols in V_n
  - $U, V, W, \ldots$ denote symbols in V
  - $\alpha, \beta, \gamma, \ldots$ denote strings in V^*
  - $u, v, w, \ldots$ 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 $\Longrightarrow^{+}$
  - Zero or more steps derivation $\Longrightarrow^{*}$
- If $S \Longrightarrow^{*} \beta$, then β is said to be sentential form of the CFG
  - SF(G) is the set of sentential forms of grammar G (may contain nonterminal vocabulary)
- $L(G) = \{x \in V_t^* | S \Longrightarrow^{+} x\}$
- $L(G) = \mathrm{SF(G)} \cap V_t^*$

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

- Left-most derivation, a top-down parsers
  - $\Rightarrow_{\mathrm{lm}}, \Rightarrow^{+}_{\mathrm{lm}}, \Rightarrow^{*}_{\mathrm{lm}}$
  - E.g. of leftmost derivation of f(v+v)

1	E	$\rightarrow$	Prefix (E)
2		|	v Tail
3	Prefix	$\rightarrow$	f
4		|	λ
5	Tail	$\rightarrow$	+ E
6		|	λ

$$
\begin{aligned}
\text{E} &\Rightarrow_{\mathrm{lm}} \text{Prefix (E)} \\
&\Rightarrow_{\mathrm{lm}} \text{f (E)} \\
&\Rightarrow_{\mathrm{lm}} \text{f (v Tail)} \\
&\Rightarrow_{\mathrm{lm}} \text{f (v + E)} \\
&\Rightarrow_{\mathrm{lm}} \text{f (v + v Tail)} \\
&\Rightarrow_{\mathrm{lm}} \text{f (v + v)}
\end{aligned}
$$

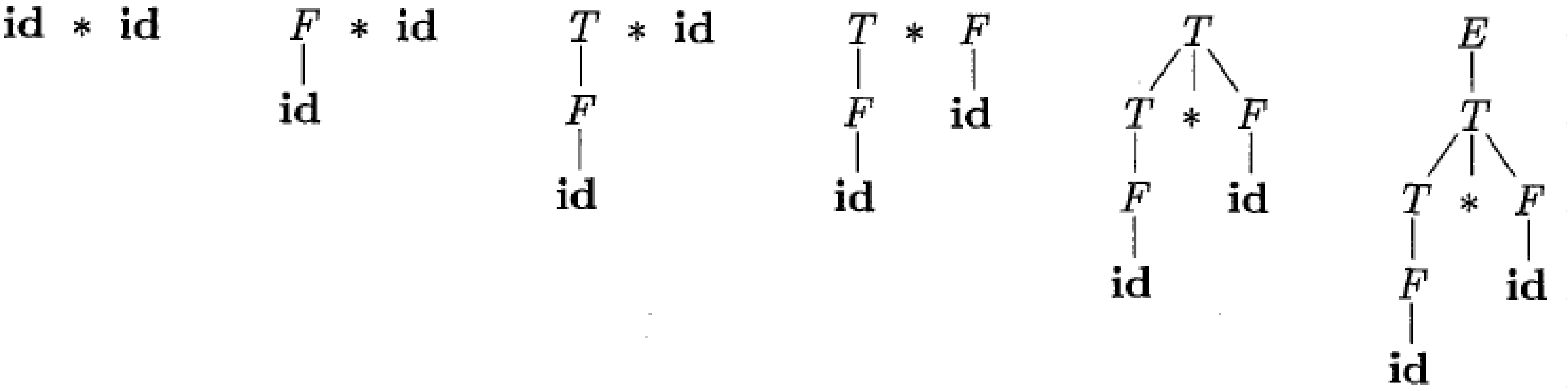
Context-Free Grammars: Concepts and Notation (Cont'd)

- Right-most derivation (canonical derivation)
  - $\Rightarrow_{\text{rm}}, \Rightarrow^{+}_{\text{rm}}, \Rightarrow^{*}_{\text{rm}}$
  - E.g. of rightmost derivation of f(v+v)

1	E	$\rightarrow$	Prefix (E)
2		$\mid$	v Tail
3	Prefix	$\rightarrow$	f
4		$\mid$	λ
5	Tail	$\rightarrow$	+ E
6		$\mid$	λ

$$
\begin{aligned}
E &\Rightarrow_{\text{rm}} \text{Prefix (E)} \\
&\Rightarrow_{\text{rm}} \text{Prefix (v Tail)} \\
&\Rightarrow_{\text{rm}} \text{Prefix (v + E)} \\
&\Rightarrow_{\text{rm}} \text{Prefix (v + v Tail)} \\
&\Rightarrow_{\text{rm}} \text{Prefix (v + v)} \\
&\Rightarrow_{\text{rm}} \text{f (v + v)}
\end{aligned}
$$

$$
\begin{aligned}
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow (E) \mid \mathbf{id}
\end{aligned}
$$

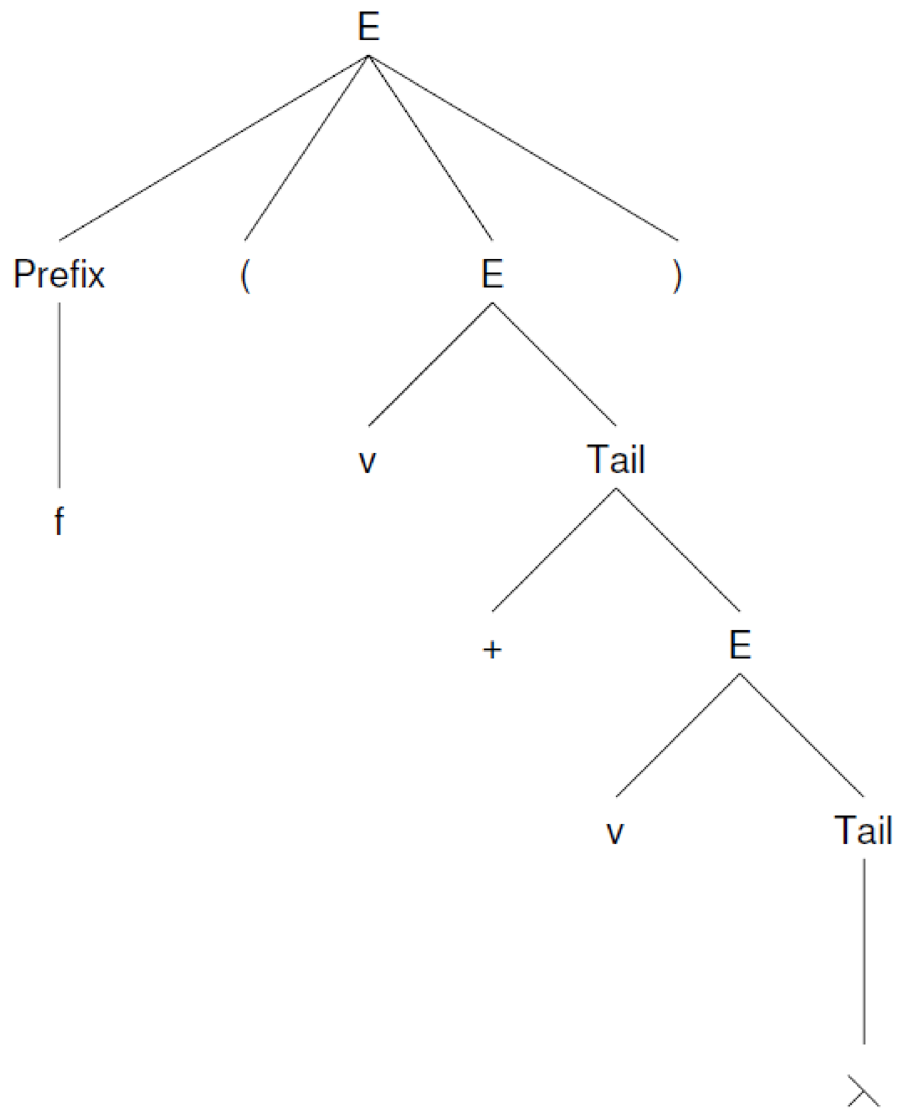


$E \rightarrow T \rightarrow T * F \rightarrow T * id \rightarrow F * id \rightarrow id * id$

Method	**classic approach**	**modern approach**
top-down	recursive descent	LL parsing (produce leftmost derivation)
bottom-up	operator precedence	LR parsing (shift-reduce parsing; produce rightmost derivation in reverse order)

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

- A parse tree
  - rooted by the start symbol
  - Its leaves are grammar symbols or λ
  - a graphical representation for derivations.
    - (Note the difference between parse tree and syntax tree.)
  - Often the parse tree is produced in only a figurative sense; in reality, the parse tree exists only as a sequence of actions made by stepping through the tree construction process.



Errors in Context-Free Grammars

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

$$\boxed{\begin{array}{l} S \rightarrow A|B \\ A \rightarrow a \\ B \rightarrow Bb \\ C \rightarrow c \end{array}}$$

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

S is the start symbol.

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

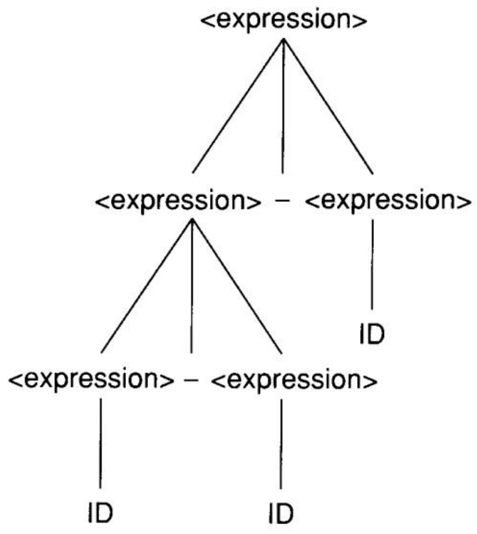


Figure 4.2 A Parse Tree for ID–ID–ID

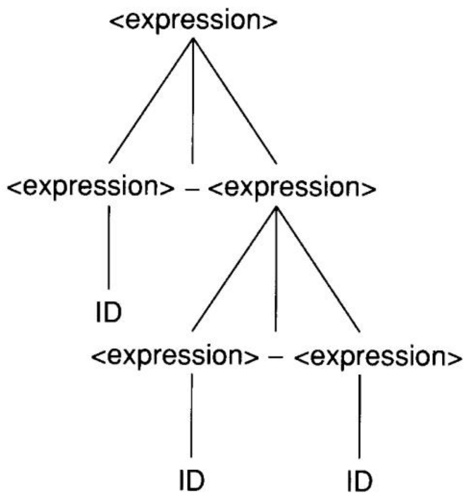


Figure 4.3 An Alternate Parse Tree for ID–ID–ID

Ambiguity

<u>Ambiguous Grammars</u>

- Def.: A context-free grammar that can produce more than one parse tree for some sentence.
- The ways to disambiguate a grammar: (1) specifying the intention (e.g. associativity and precedence for arithmetic operators, other) (2) rewrite a grammar to incorporate the intention into the grammar itself.

For (1) Precedence: ( )>negate > exponent > * / > + −

Associativity: exponent → right associativity

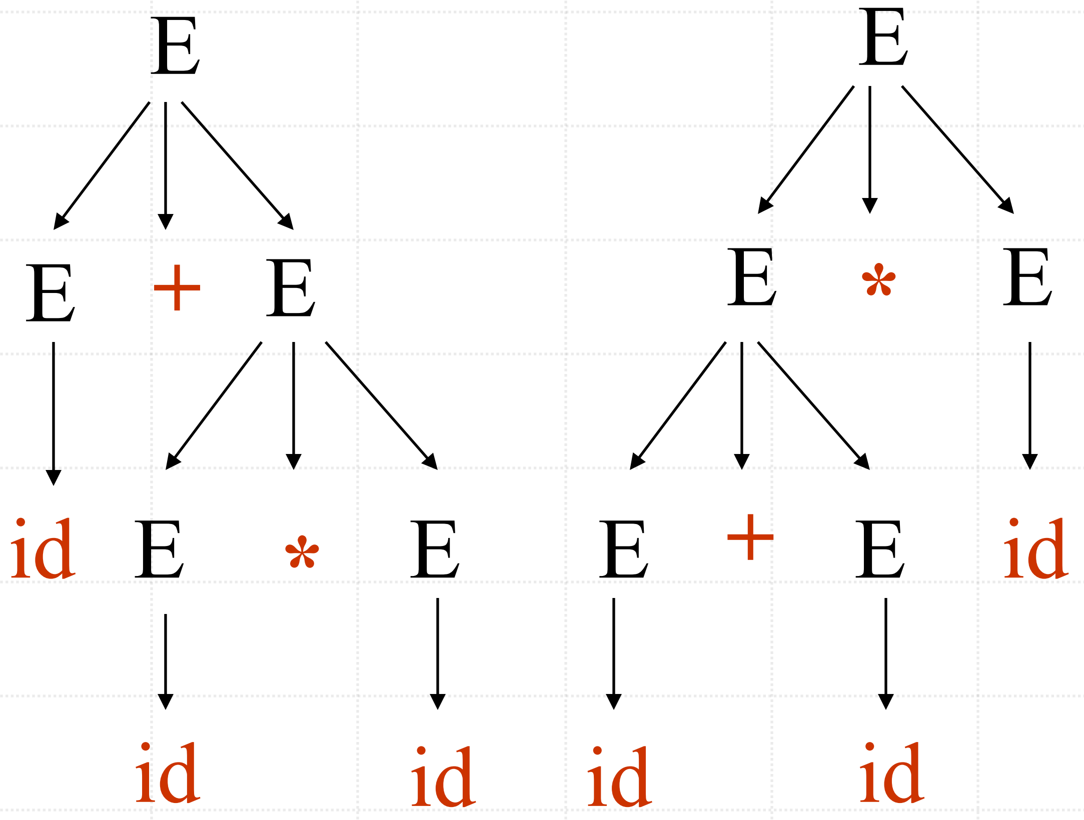
others → left associativity

For (2) 1. introducing one nonterminal for each precedence level.

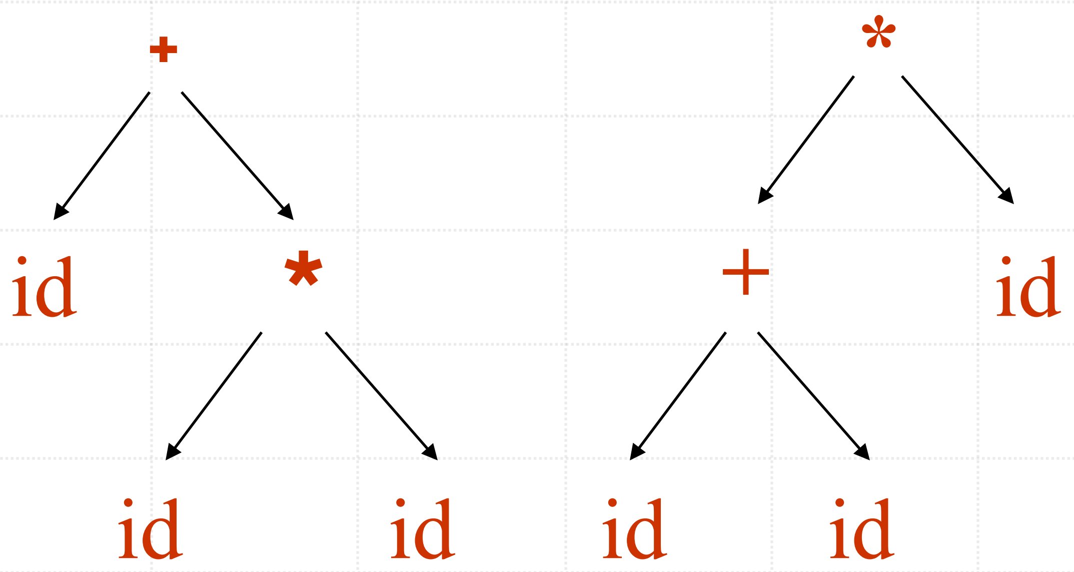
Example 1

$E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid E \uparrow E \mid (E) \mid - E \mid id$

is ambiguous ($\uparrow$ is exponent operator with right associativity.)



More than one parse tree for the sentence <u>id + id * id</u>



More than one syntax tree for the sentence id + id * id

The corresponding grammar shown below is unambiguous

element → (expression) | id   /*((expression) 括號內的最優先做之故) */

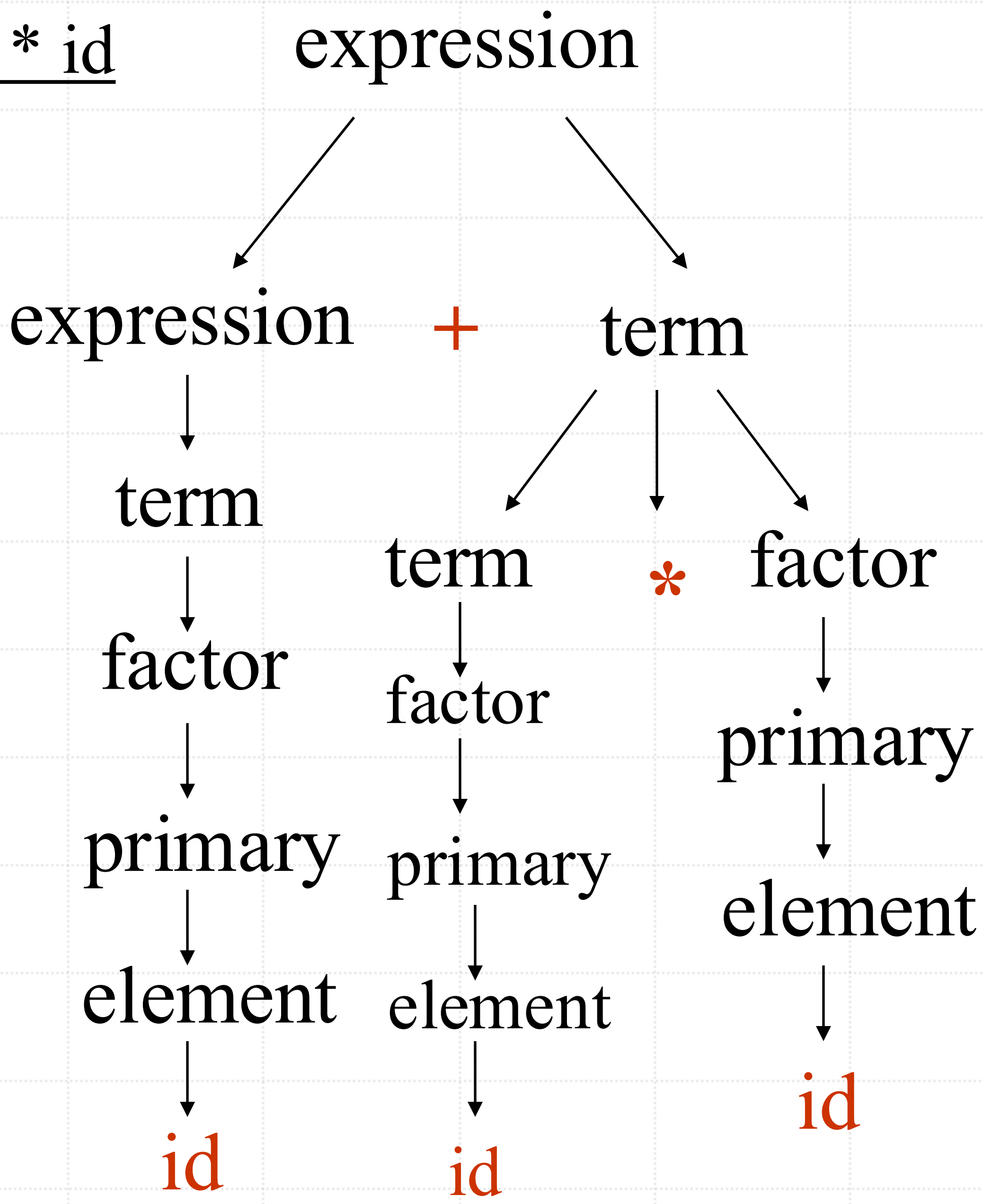
primary → -primary | element

factor → primary ↑ factor | primary   /*has right associativity */

term → term * factor | term / factor | factor

expression → expression + term | expression – term | term

Ex: id + id * id


expression
expression
+
term
term
term
*
factor
factor
factor
primary
primary
primary
element
element
element
id
id
id

Example 3.10 Consider our running example of simple arithmetic expressions. This has the BNF (including associativity and precedence).

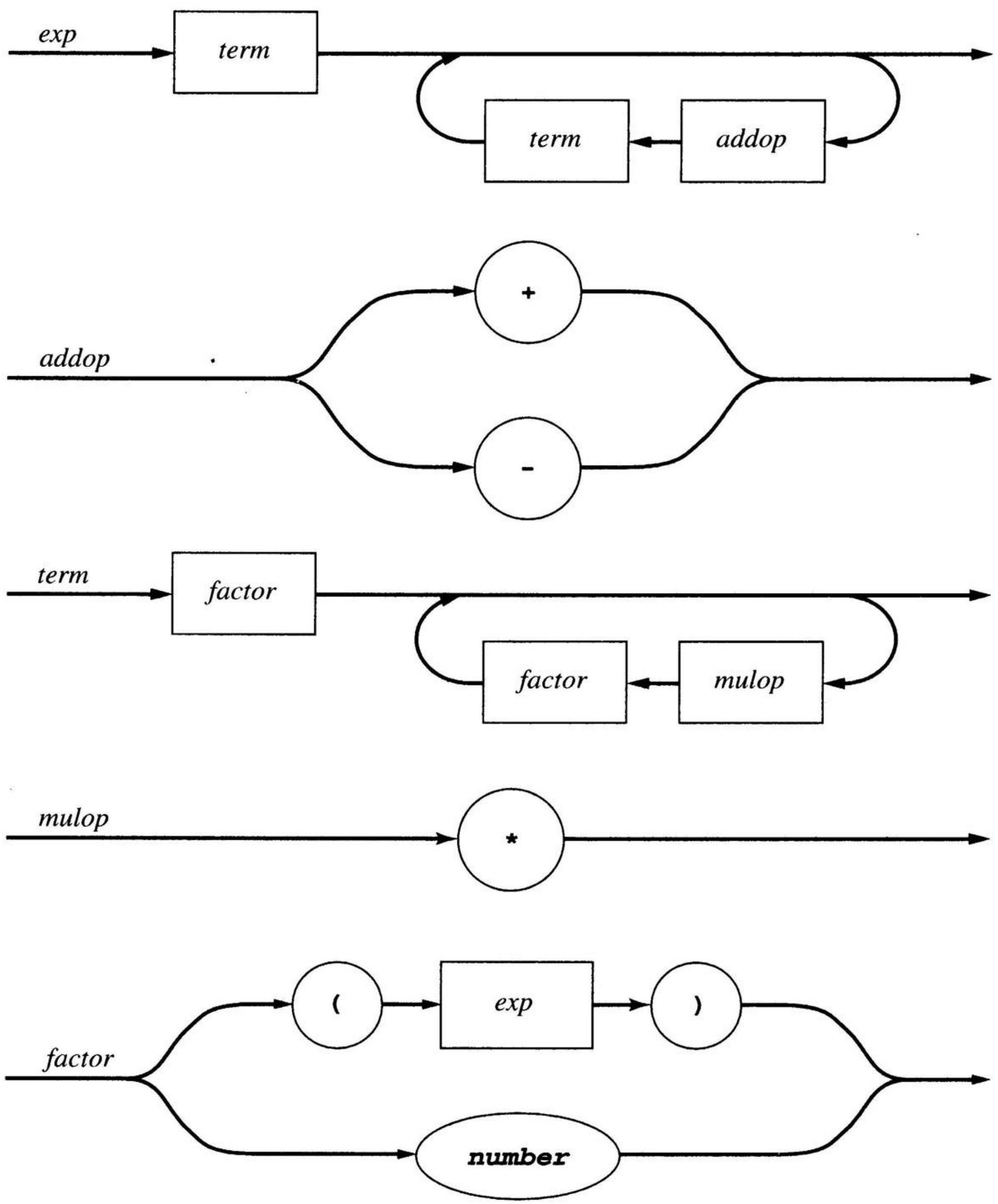
$$
\begin{aligned}
&exp \rightarrow exp\ addop\ term \mid term\\
&addop \rightarrow \texttt{+} \mid \texttt{-}\\
&term \rightarrow term\ mulop\ factor \mid factor\\
&mulop \rightarrow \texttt{*}\\
&factor \rightarrow \texttt{(}\ exp\ \texttt{)} \mid \textbf{number}
\end{aligned}
$$

The corresponding EBNF is

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

The corresponding syntax diagrams are given in Figure 3.4 (the syntax diagram for *factor* was given previously).

Figure 3.4
Syntax diagrams for the grammar of Example 3.10



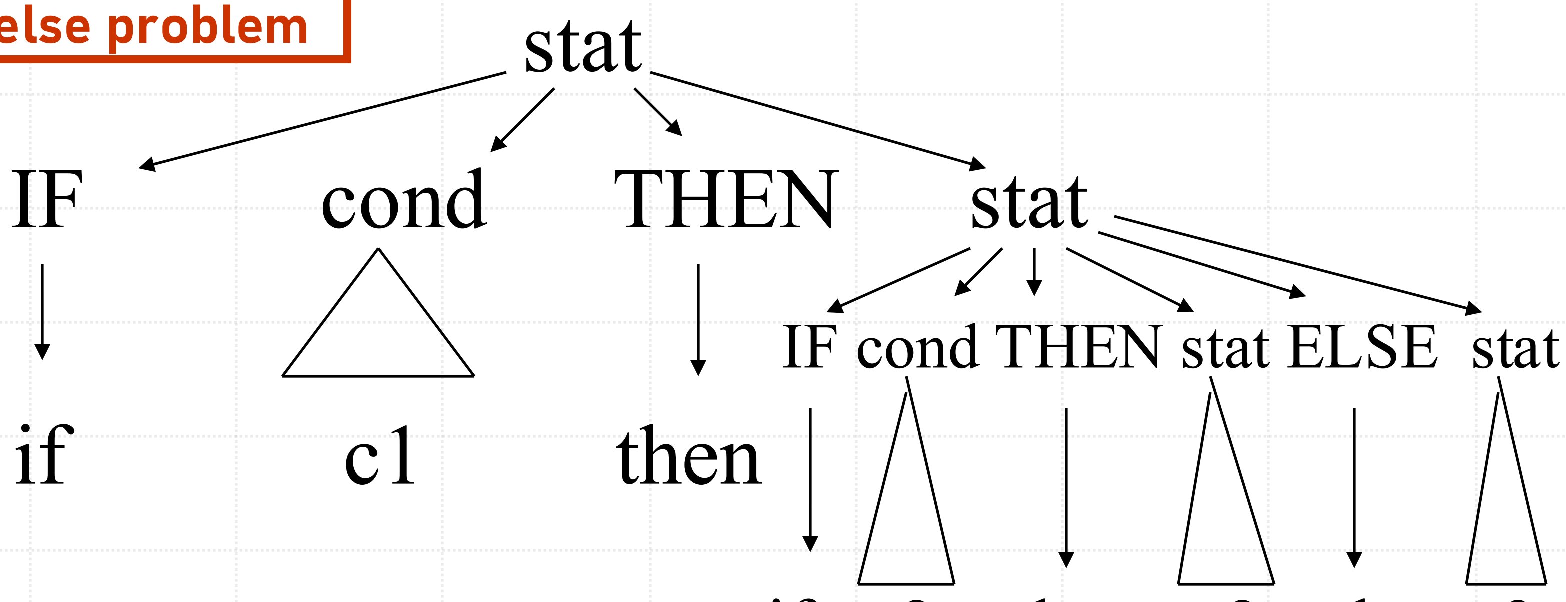
§

Example 2

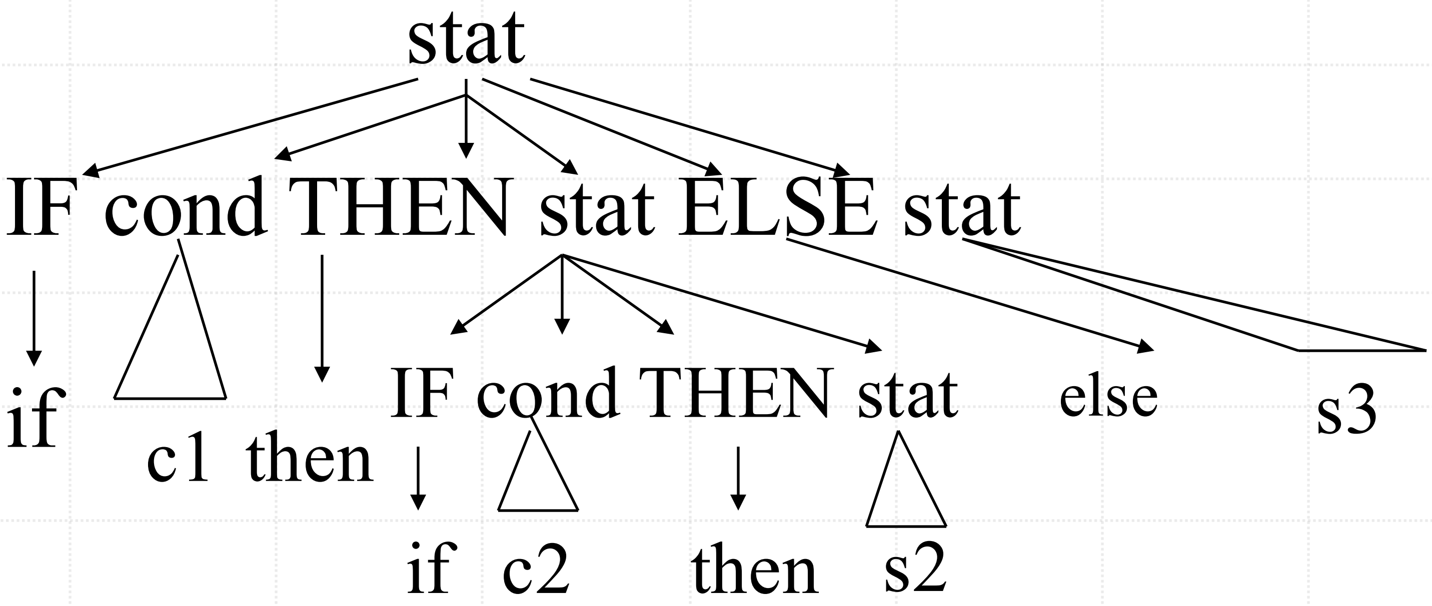
- stat $\rightarrow$ IF cond THEN stat | IF cond THEN stat ELSE stat | other stat

is an ambiguous grammar

Dangling else problem



If c1 then if c2 then s2 else s3

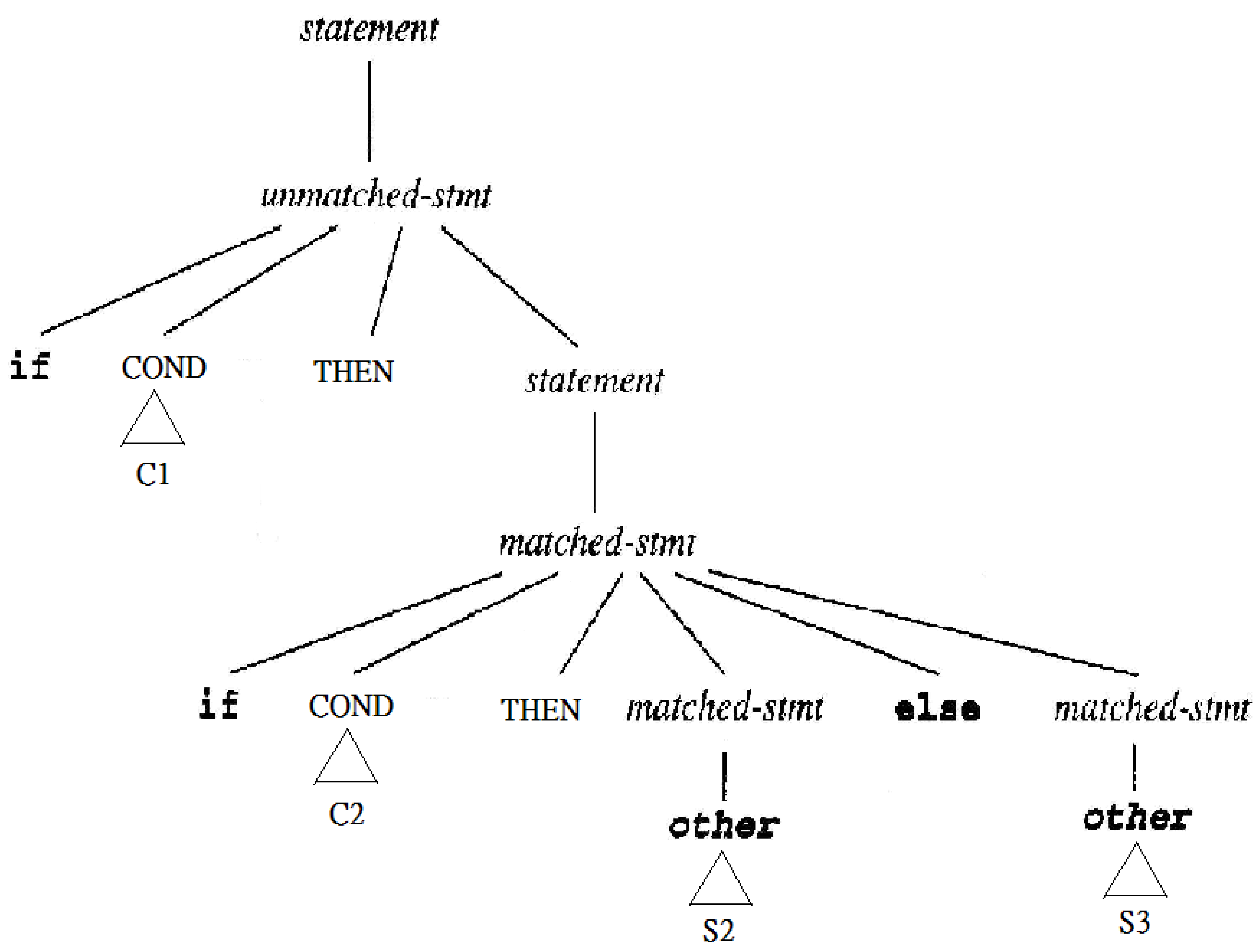


The corresponding grammar shown below is unambiguous.

stat → matched-stat | unmatched-stat

matched-stat → IF cond THEN matched-stat ELSE matched-stat | other-stat

unmatched-stat → IF cond THEN stat | IF cond THEN matched-stat ELSE unmatched-stat


statement
unmatched-stmt
if
COND
C1
THEN
statement
matched-stmt
if
COND
C2
THEN
matched-stmt
other
S2
else
matched-stmt
other
S3

Transforming Extened BNF Grammars

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

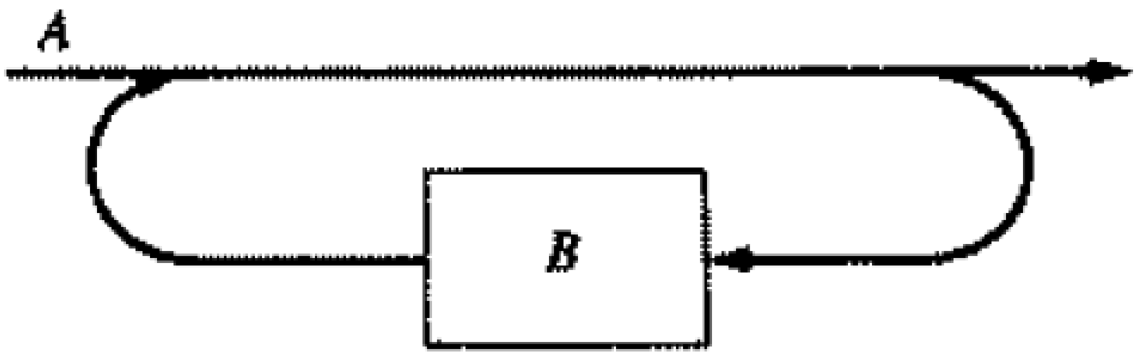
```
for (each production P = A → α [X₁ ... Xₙ] β) {
    Create a new nonterminal, N.
    Replace production P with P′ = A → α N β
    Add the productions: N → X₁ ... Xₙ and N → λ
}

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

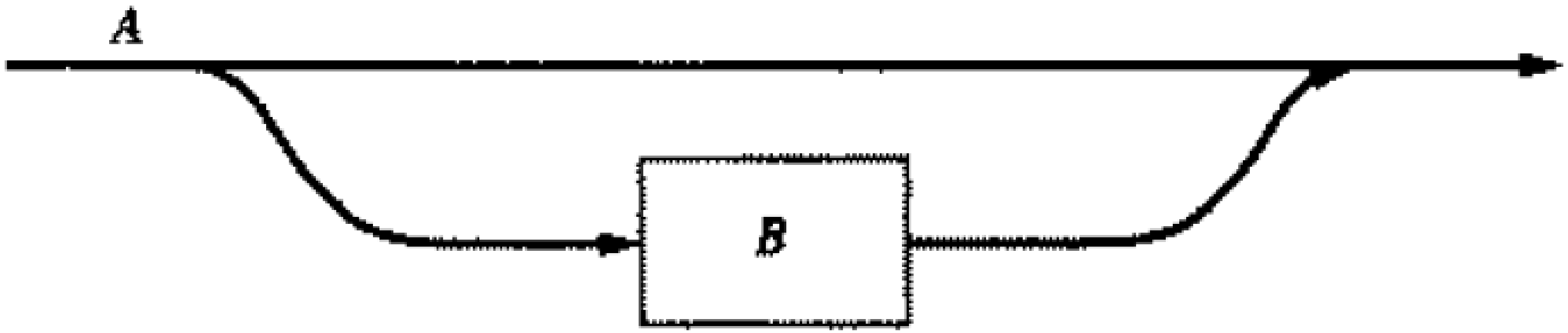
$$A \rightarrow \{ B \}$$

the corresponding syntax diagram is usually drawn as follows:



$$A \longrightarrow [\ B\ ]$$

is drawn as



Parsers and Recognizers

- Recognizer
  - An algorithm that does boolean-valued test
    - "Is this input syntactically valid?
- Parser
  - Answers more general questions
    - Is this input syntactically 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>	$\rightarrow$	**begin** <Stmts> **end** $
<Stmts>	$\rightarrow$	<Stmt> ; <Stmts>
<Stmts>	$\rightarrow$	λ
<Stmt>	$\rightarrow$	SimpleStmt
<Stmt>	$\rightarrow$	**begin** <Stmts> **end**

Grammar G_3

To parse

begin SimpleStmt; SimpleStmt; **end** $

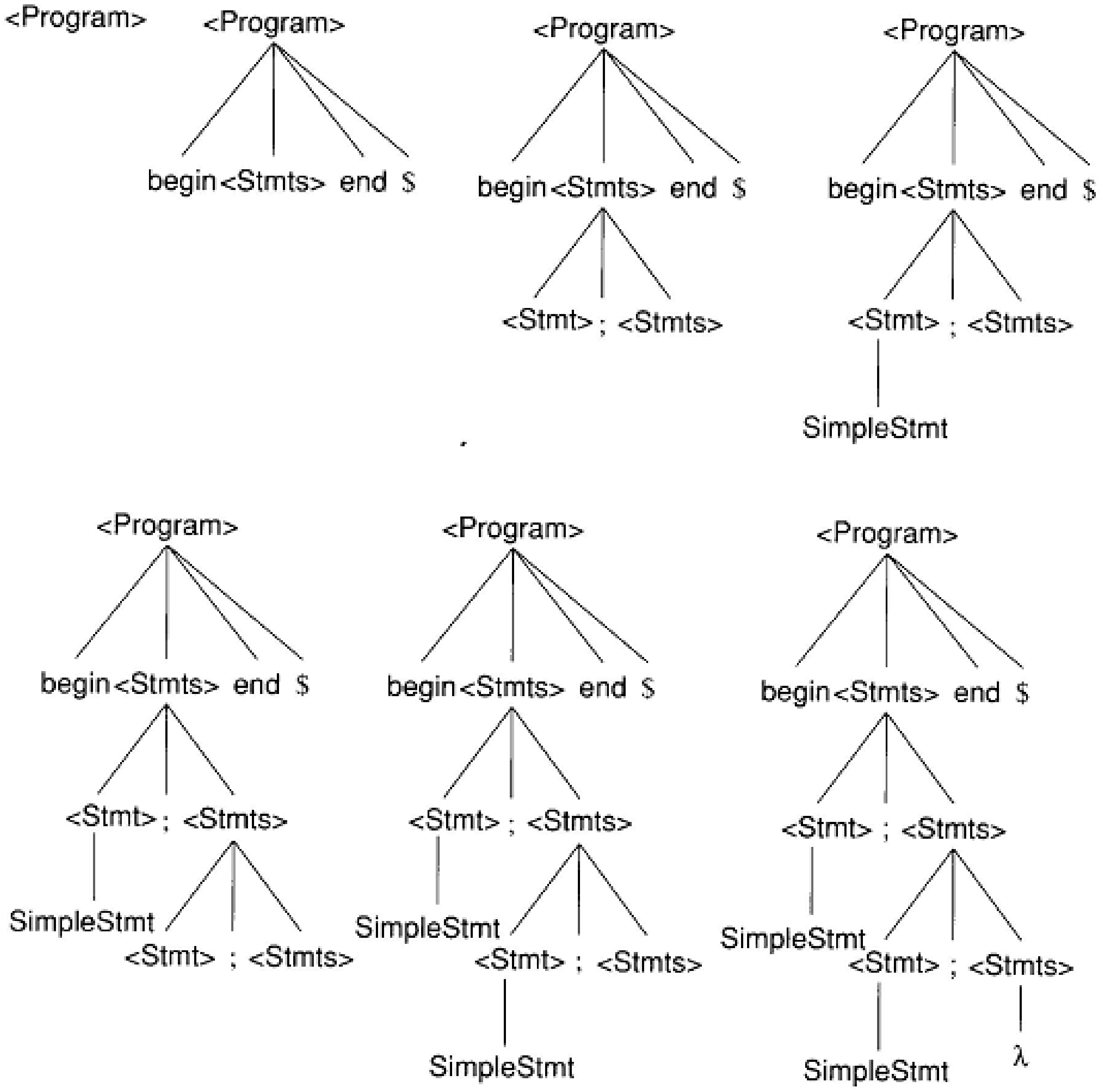


Figure 4.5 A Top-Down Parse

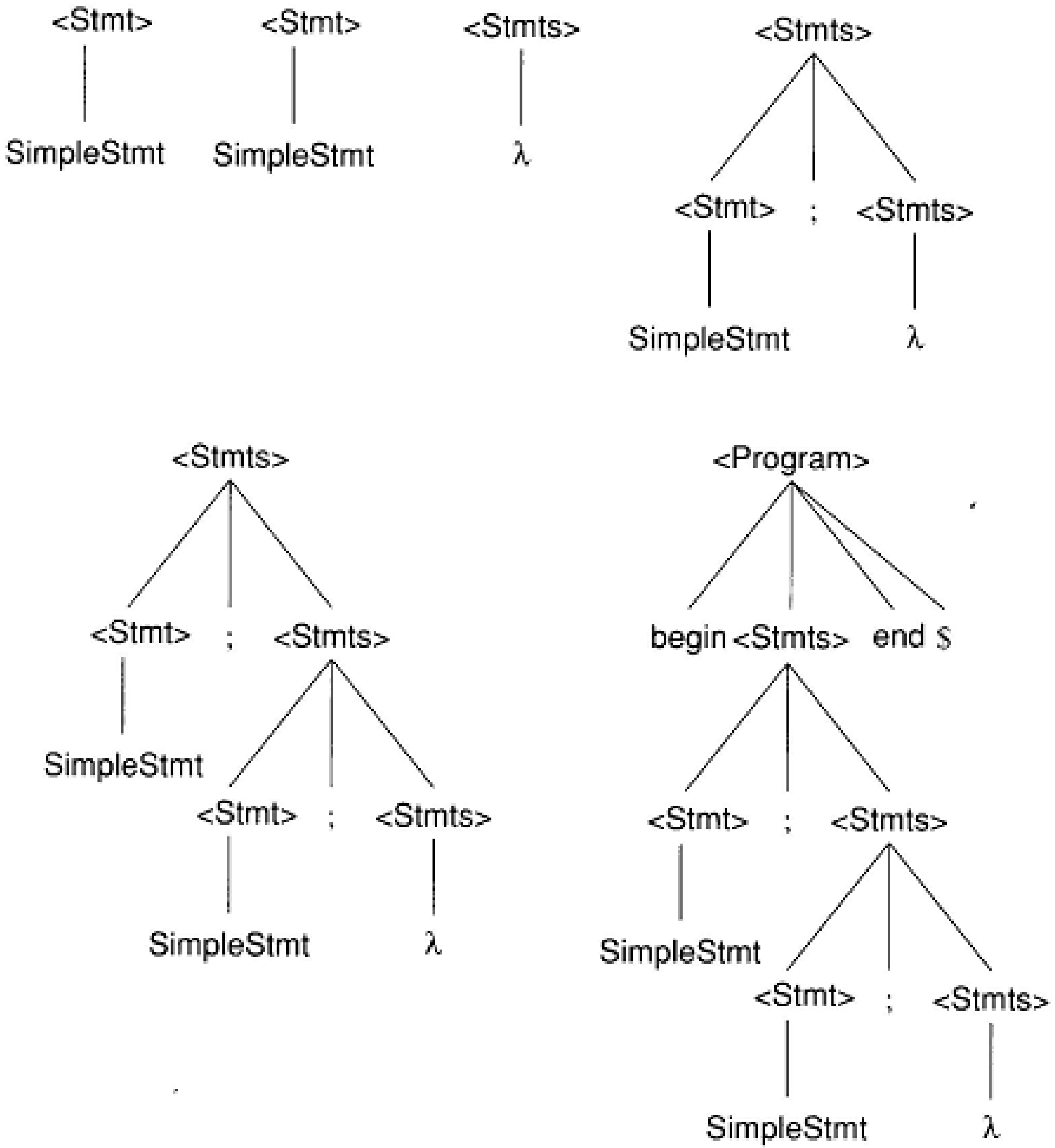
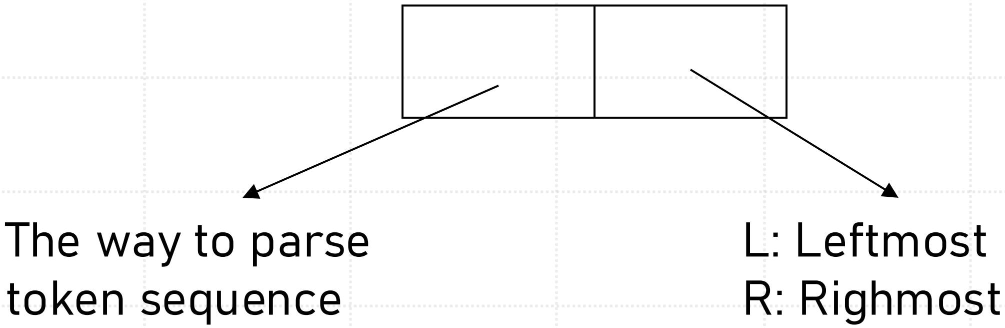


Figure 4.6 A Bottom-Up Parse

Parsers and Recognizers (Cont'd)

- Naming of parsing techniques



- 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)

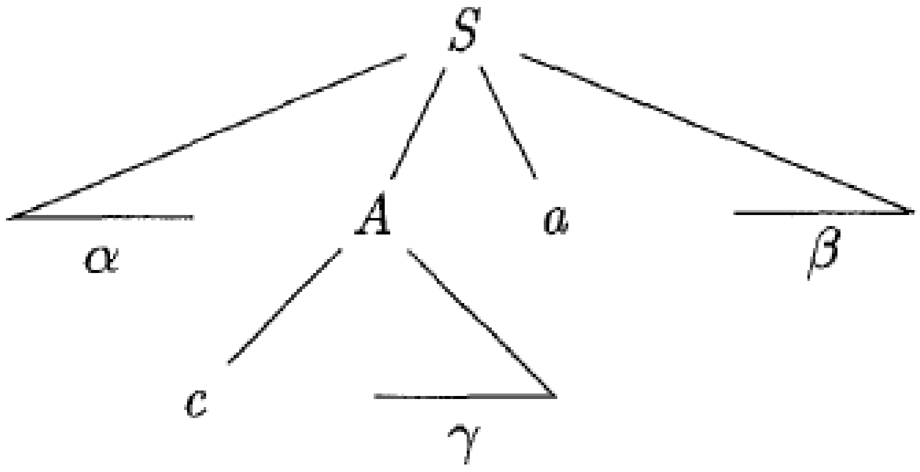


Figure 4.15: Terminal c is in FIRST(A) and a is in FOLLOW(A)

Grammar Analysis Algorithms (Cont'd)

- $\mathrm{Follow}(A)$
  - A is any nonterminal
  - $\mathrm{Follow}(A)$ is the set of terminals that my follow A in some sentential form
    - $\mathrm{Follow}(A) = \{a \in V_t | S \Longrightarrow^* \ldots Aa \ldots\} \cup \{\text{if } S \Longrightarrow^+ \alpha A \text{ then } \{\lambda\} \text{ else } \phi\}$
- $\mathrm{First}(\alpha)$
  - 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 $\mathrm{First}(\alpha)$ contains terminal symbols that begin strings derivable from α
    - $\mathrm{First}(\alpha) = \{a \in V_t | \alpha \Longrightarrow^* a\beta\} \cup \{\text{if } \alpha \Longrightarrow^* \lambda \text{ then } \{\lambda\} \text{ else } \phi\}$

Grammar Analysis Algorithms (Cont'd)

- Definition of C data structures and subroutines
  - $\text{first\_set}[X]$
    - contains terminal symbols and λ
    - X is any single vocabulary symbol
  - $\text{follow\_set}[A]$
    - 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

$$
\begin{array}{llll}
1 & E & \rightarrow & \text{Prefix} \; (\; E \;) \\
2 & & | & v \;\; \text{Tail} \\
3 & \text{Prefix} & \rightarrow & f \\
4 & & | & \lambda \\
5 & \text{Tail} & \rightarrow & + \;\; E \\
6 & & | & \lambda
\end{array}
$$

The execution of fill_first_set() using grammar G_0

Step	first_set							
	E	Prefix	Tail	(	)	v	f	+
(1) First loop	ϕ	$\{\lambda\}$	$\{\lambda\}$					
(2) Second (nested) loop	$\{v\}$	$\{f, \lambda\}$	$\{+, \lambda\}$	$\{(\}$	$\{)\}$	$\{v\}$	$\{f\}$	$\{+\}$
(3) Third loop, production 1	$\{v, f, (\}$	$\{f, \lambda\}$	$\{+, \lambda\}$	$\{(\}$	$\{)\}$	$\{v\}$	$\{f\}$	$\{+\}$

```
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

$$
\begin{array}{llll}
1 & E & \rightarrow & \text{Prefix} \ (\ E \ ) \\
2 & & | & v \ \text{Tail} \\
3 & \text{Prefix} & \rightarrow & f \\
4 & & | & \lambda \\
5 & \text{Tail} & \rightarrow & + \ E \\
6 & & | & \lambda
\end{array}
$$

The execution of fill_follow_set() using grammar G_0

Step	follow_set		
	E	Prefix	Tail
(1) Initialization	$\{\lambda\}$	ϕ	ϕ
(2) Process Prefix in production 1	$\{\lambda\}$	$\{(\}$	ϕ
(3) Process E in production 1	$\{\lambda,)\}$	$\{(\}$	ϕ
(4) Process Tail in production 1	$\{\lambda,)\}$	$\{(\}$	$\{\lambda,)\}$

More examples

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

Step	first_set							
	S	B	C	a	b	c	d	e
(1) First loop	ϕ	ϕ	ϕ					
(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}

More examples

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

Step	follow_set		
	S	B	C
(1) Initialization	$\{\lambda\}$	ϕ	ϕ
(2) Process S in production 1	$\{e, \lambda\}$	ϕ	ϕ
(3) Process B in production 2	$\{e, \lambda\}$	$\{e, \lambda\}$	ϕ
(4) Process B in production 3	No changes		
(5) Process C in production 4	$\{e, \lambda\}$	$\{e, \lambda\}$	$\{e, \lambda\}$
(6) Process C in production 5	No changes		

More examples

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

Step	first_set					
	S	A	B	a	b	c
(1) First loop	ϕ	$\{\lambda\}$	$\{\lambda\}$			
(2) Second (nested) loop	ϕ	$\{a, \lambda\}$	$\{b, \lambda\}$	$\{a\}$	$\{b\}$	$\{c\}$
(3) Third loop, production 1	$\{a, b, c\}$	$\{a, \lambda\}$	$\{b, \lambda\}$	$\{a\}$	$\{b\}$	$\{c\}$

More examples

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

Step	follow_set		
	S	A	B
(1) Initialization	$\{\lambda\}$	ϕ	ϕ
(2) Process A in production 1	$\{\lambda\}$	$\{b, c\}$	ϕ
(3) Process B in production 1	$\{\lambda\}$	$\{b, c\}$	$\{c\}$