

COMPILER CONSTRUCTION

Grammars and Parsing

Chia-Heng Tu
Dept. of Computer Science and Information Engineering
National Cheng Kung University
Spring 2020



Chapter 4

Grammars and Parsing



Outline

- Introduction to Grammar
- Context-Free Grammar
- Backus-Naur Form
- Parsing Techniques
- First and Follow



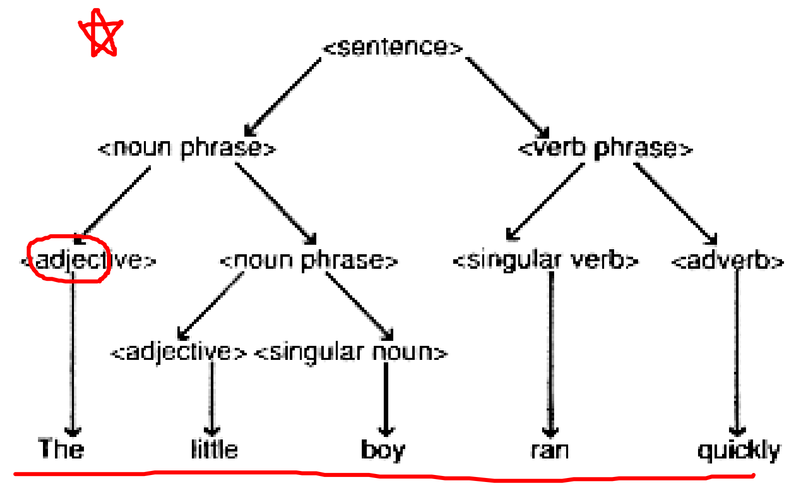
Why Grammar?

- Grammar rules of **natural languages**, such as English or Chinese
  - **Define proper sentence structure**, e.g., defining phrases in terms of subjects, verbs, and objects; and phrases and conjunctions
  - Served as a tool for **diagnosing malformed sentences** (**validity check**)
- It is possible to construct sentences in a natural language
  - that are grammatically correct
  - but still make no sense

Why Grammar? (Cont'd)

- A structured sentence can be diagrammed
  - to show how its components conform to a language's grammar
  - Grammars can also explain what is absent or superfluous in a malformed sentence
- **Ambiguity** of a sentence
  - Can often be explained by providing **multiple diagrams for the same sentence**

``The little boy ran quickly"
can be diagrammed as:



Grammars for Programming Languages

- Modern programming languages
  - contain a **grammar in their specification** as a guide to those who teach, study, or use the language

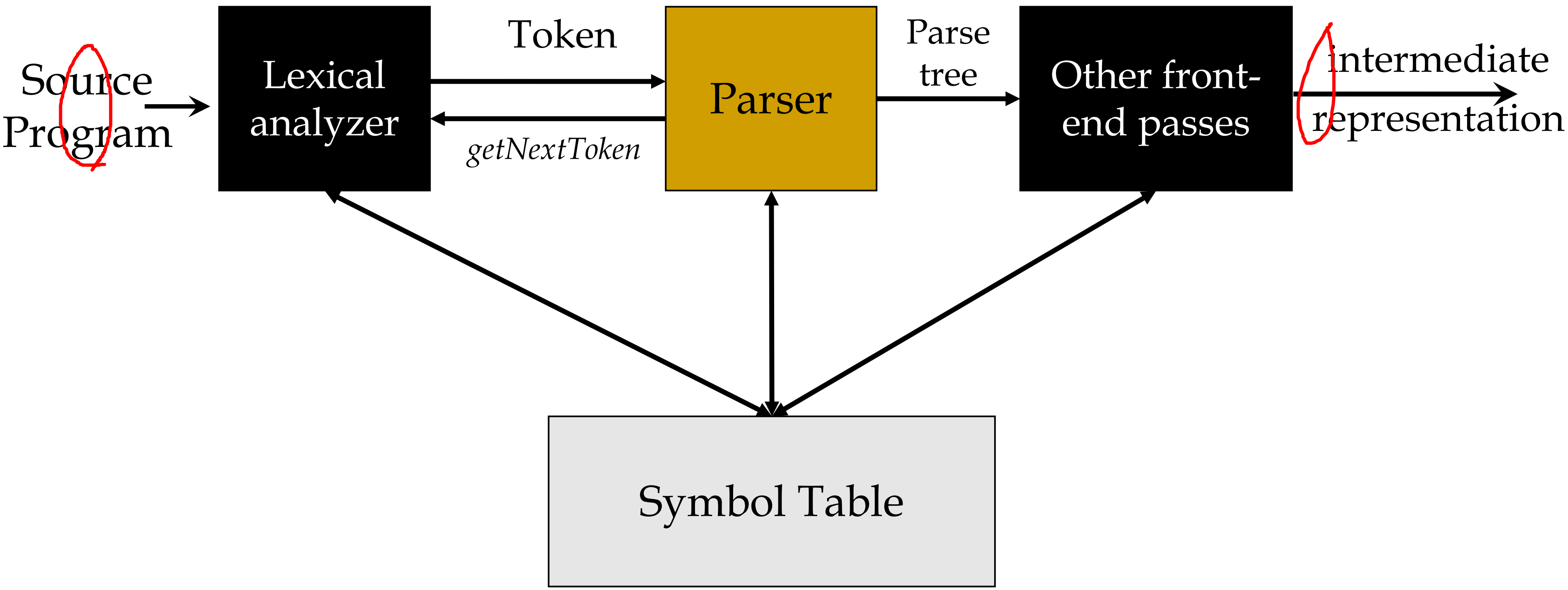
- A compiler front-end for the language
  - Scans for tokens in input stream based on the **regular sets**
  - Parses the structures formed by the tokens using the **grammar** that specifies a programming language's syntax



This Chapter is for CFGs

- We discuss the basics of **context-free grammars** (CFGs) in Ch. 2
- In Ch. 4 we
  - **formalize the definition and notation for CFGs** and
  - present algorithms that analyze such grammars in preparation for the parsing techniques covered in Ch. 5 and 6

The Role of the Parser



Context-Free Grammar

- Context-free grammar is a 4-tuple
  G = < Σ, N, P, S> where
  - Σ is a finite set of **terminal alphabet**, which is the set of tokens produced by the scanner
  - N is a finite set of **nonterminal alphabet**
  - P is a finite set of **productions** of the form
    $A \rightarrow \beta$
    where $A \in N$ and $\beta \in (N \cup \Sigma)^*$
  - $S \in N$ is a designated **start symbol**, which initiates all derivations



Conventions

- Terminals
  - $a, b, c, \dots \in \Sigma$
  - More example: **0**, **1**, **+**, *****, **id**, **if**
- Nonterminals
  - $A, B, C, \dots \in N$
  - More example: *expr*, *term*, *stmt*
- Grammar symbols
  - $X, Y, Z \in (N \cup \Sigma)$
- Strings of grammar symbols (sentential form)
  - $\alpha, \beta, \gamma \in (N \cup \Sigma)^*$
- The head of the first production is often the start symbol

Production Rules

- A **grammar** consists of a set of **production rules** and **a start symbol** (left symbol of first rule)

- A **production rule** consists of two parts: a left-hand side and a right-hand side
  - ex: expression → expression '+' term

left-hand side

right-hand side

Production Rules (Cont.)

- The **left-hand side** (LHS) is the **name** of the syntactic construct
- The **right-hand side** (RHS) shows a **possible form** of the syntactic construct

- E.g., there are **two possible forms** (rules) derived by **the name** "expression":

  expression → expression '+' term (rule 1)

  expression → expression '-' term (rule 2)

Production Rules (Cont.)

- LHS must be a single **non-terminal** symbol (or *non-terminal*)
- RHS of a production rule can contain zero or more **terminals and non-terminals**

- A **terminal symbol** (or *terminal*) is a grammar symbol
  - that cannot be rewritten
  - Is also an end point of the production process, also called **token**
  - Use lower-case letters such as a and b
- A **non-terminal symbol** (or *non-terminal*) is able to be rewritten
  - Use upper-case letters such as A, B, and S
- Non-terminal and terminal together are called **grammar symbols (or vocabulary)**

Different Forms of Production Rules

- There is often more than one way to rewrite a given nonterminal
- For example, when multiple productions share the same LHS symbol, it could be presented in one of the two forms

$$
\begin{aligned}
&A \rightarrow \alpha \\
&A \rightarrow \beta \\
&\quad \cdot \quad \cdot \quad \cdot \\
&A \rightarrow \zeta \ \text{(Zeta)}
\end{aligned}
\qquad\qquad
\begin{aligned}
&A \rightarrow \alpha \\
&\quad\ \mid \beta \\
&\qquad \cdot \quad \cdot \quad \cdot \\
&\quad\ \mid \zeta
\end{aligned}
$$



Derivations of Production Rules

- Derivation
  - The rewriting step replaces a **nonterminal** by the RHS of one of **its** production rules
- Example:
  - If $A \rightarrow \zeta$ is a production, then $\alpha A \beta \Rightarrow \alpha \zeta \beta$ denotes one step of a derivation using this production
  - The one-step derivation can be denoted as $\alpha A \beta \Rightarrow \alpha \zeta \beta$
  - The sequence of replacement a derivation of ζ from A

Notation for Derivations

- $\Rightarrow$ derives in one step
- $\Rightarrow^{+}$ derives in one or more steps
- $\Rightarrow^{*}$ derives in zero or more steps
- $\Rightarrow_{lm}$ refers to **leftmost derivation**, which expands nonterminals left to right
- $\Rightarrow_{rm}$ refers to **right most derivation**, which expands nonterminals right to left

- Example:

  $\alpha A \beta \Rightarrow \alpha \zeta \beta$

  – is leftmost derivation, if α does not contain a nonterminal

  – is rightmost derivation, if β does not contain a nonterminal

More about the Grammar

- We say that α is a **sentential form** of G
  - if $S \Rightarrow^{*} \alpha$, where S is the start symbol of a grammar G
  - Note that a sentential form may contain both terminals and nonterminals, and may be empty

- A **sentence** of *G* is a sentential form with no nonterminals
- The language generated by a grammar is its set of sentences
  - Hence, a string of terminals *w* is in L(G), the language generated by G, if and only if *w* is a sentence of G (or $S \Rightarrow^{*} w$)



Example of Rule Derivations

- Given the production rules:
  - expr → ‘(‘expr op expr ‘)’
  - expr → ‘1’
  - op → ‘+’
  - op → ‘*’

Example of Rule Derivations (Cont'd)

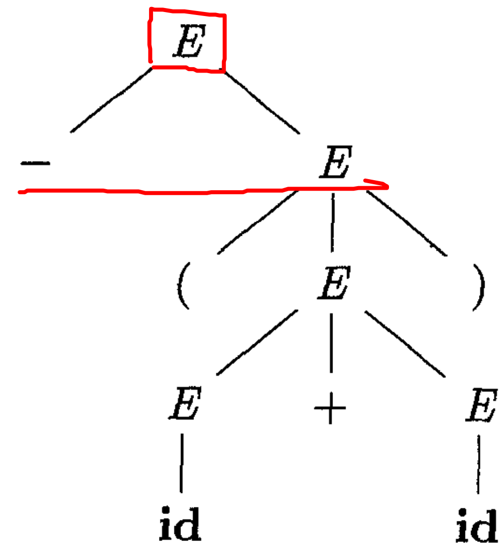
- Derivation of the string (1*(1+1))
  - expr
  - '('expr op expr ')'
  - '(' '1' op expr ')'
  - '(' '1' '*' expr ')'
  - '(' '1' '*' '('expr op expr ')' ')'
  - '(' '1' '*' '(' '1' op expr ')' ')'
  - '(' '1' '*' '(' '1' '+' expr ')' ')'
  - '(' '1' '*' '(' '1' '+' '1' ')' ')'

expr → '('expr op expr ')'
expr → '1'
op → '+'
op → '*'

  - Each of the strings is a **sentential form**
  - The *strings* refer to expr, '('expr op expr ')', '(' '1' op expr ')', '(' '1' '*' expr ')', etc.
- It forms a **leftmost derivation**, in which the leftmost nonterminal is always rewritten in each sentential form

A CFG Example

Parse tree for –(id + id)



- G = < Σ, N, P, S >
  - Σ = { **+**, *****, **(**, **)**, **-**, **id** }
  - N = { E }
  - $P = \; E \rightarrow E + E$
    $E \rightarrow E * E$
    $E \rightarrow (E)$
    $E \rightarrow - E$
    $E \rightarrow \mathbf{id}$
  - $S = \; E$
- Is the string –(**id** + **id**) a sentence of G?

  Yes, there is a derivation of the given string

  → $E \Rightarrow_{\text{lm}} -E \Rightarrow_{\text{lm}} -(E) \Rightarrow_{\text{lm}} -(E+E) \Rightarrow_{\text{lm}} -(\mathbf{id}+E) \Rightarrow_{\text{lm}} -(\mathbf{id}+\mathbf{id})$
- The strings *E*, - *E*, - *(E)* , . . . , - *(id* + *id)* are all sentential forms of this grammar
  - We write $E \Rightarrow^{*} -(\mathbf{id}+\mathbf{id})$ to indicate that - (**id** + **id**) can be derived from E



You would …

- Refer to Section 4.1 for more information
  - Examples of leftmost and right most derivations in Section 4.1.1 and 4.1.2
  - Parse Trees in Section 4.1.3



Properties of CFGs

- Some grammars have one or more of the following problems that preclude their use
  - (**Reduced Grammar**) The grammar may include useless symbols
  - (**Ambiguity**) The grammar may allow multiple, distinct derivations (parse trees) for some input string
  - (**Faulty Language Definition**) The grammar may include strings that do not belong in the language, or
    - the grammar may exclude strings that are in the language



We are …

- going to know more about the ambiguity next
- You would …
  - read Section 4.2.1 and 4.2.3 by yourself



Ambiguity

- A grammar that produces more than one parse tree for some sentence is said to be ***ambiguous***
- Hence, an ambiguous grammar is one that produces
  - more than one leftmost derivation or
  - more than one rightmost derivation for the same sentence



Example of Ambiguous Grammar

- Given the sentence: **id + id * id**
- Production rules:

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

- Two possible derviations:

$$\begin{aligned} E &\rightarrow E + E \\ &\rightarrow \mathbf{id} + E \\ &\rightarrow \mathbf{id} + E * E \\ &\rightarrow \mathbf{id} + \mathbf{id} * E \\ &\rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \end{aligned}$$

$$\begin{aligned} E &\rightarrow E * E \\ &\rightarrow E + E * E \\ &\rightarrow \mathbf{id} + E * E \\ &\rightarrow \mathbf{id} + \mathbf{id} * E \\ &\rightarrow \mathbf{id} + \mathbf{id} * \mathbf{id} \end{aligned}$$

Another Example

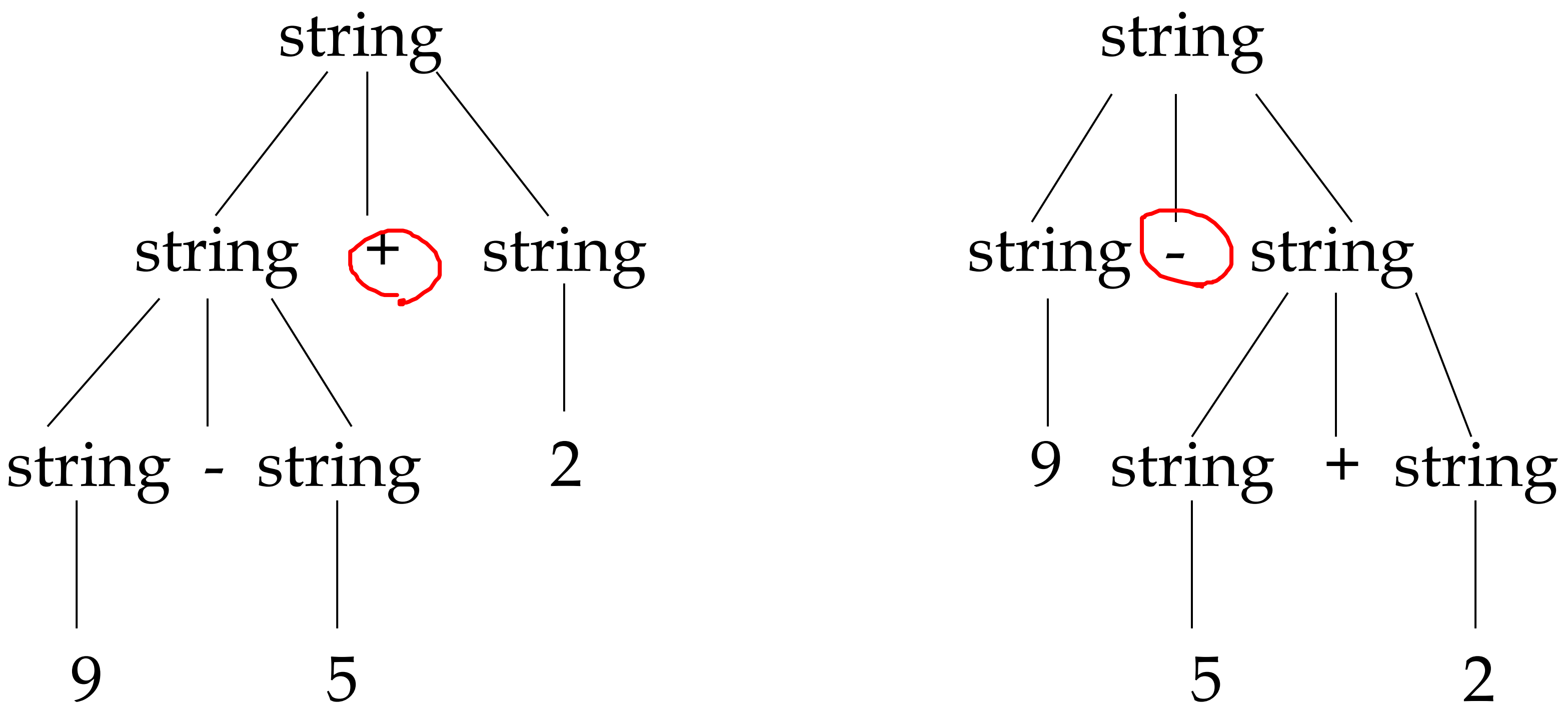
- Given the sentence: **9-5+2**
- Production rules:

string → string + string

| string – string

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Another Example (Con’td)





To Deal with Ambiguity

- Disambiguating rules are used to throw away undesirable parse trees

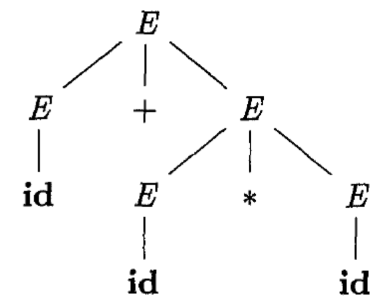
- Two options to deal with ambiguity:
  1. Enforce precedence and associativity of the existing rules
  2. Rewrite the grammar (rules)

Precedence and Associativity to Resolve Conflicts

- The grammar G with the rules

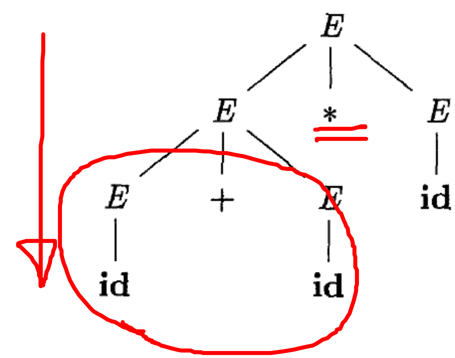
$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$

- G is **ambiguous** because
  - it does not specify the associativity or precedence of the operators + and *



- The following grammar G′ with the rules

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \mathbf{id}$



- G′ is **unambiguous** grammar that generates the same language
  - but gives + lower precedence than * (as the top figure), and makes both operators left associative
- There are parsers for both handling unambiguous and ambiguous grammars respectively

Precedence in the Grammar Rules

- The grammar G′ with the rules

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow (E) \mid \mathbf{id}$$

- In G′, it gives + lower precedence than *
  - The farther from the starting symbol is the production rule, the deeper its nodes will be nested in the derivation tree
  - Consequently, operators that are generated by production rules that are **more distant from the starting symbol** of the grammar tend to have **higher precedence**
  - This, of course, only applies if our evaluation algorithm starts by computing values **from the leaves of the derivation tree towards its root**

Curtesy of https://en.wikibooks.org/wiki/Introduction_to_Programming_Languages/Precedence_and_Associativity



Example

- Given the sentence: **id+id*id**
- The following grammar G′ with the rules

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

- The corresponding derivations:

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



Left Associativity Grammar

- Consider the following grammar (productions) for numerical expressions constructed with the - operation:

  $Exp \rightarrow Exp\ \text{-}\ Exp \mid Num$

  $Term \rightarrow Term * Term \mid Num$

  - This grammar is ambiguous since it allows both the interpretations (5 - 3) - 2 and 5 - (3 - 2)

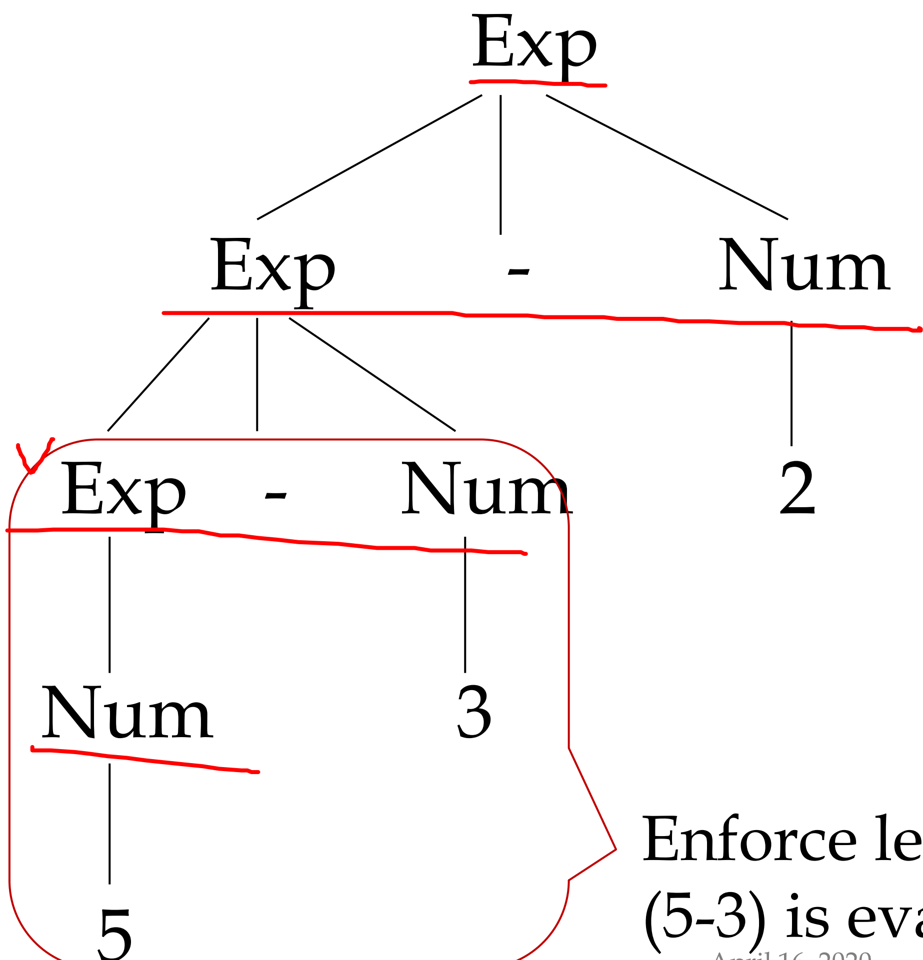
- If we want to impose the left-associativity (following the mathematical convention), it is sufficient to modify the productions in the following way:
  - $Exp \rightarrow Exp\ \text{-}\ Num \mid Num$

Curtesy of http://www.lix.polytechnique.fr/~catuscia/teaching/cg428/02Spring/lecture_notes/L03.html#Associativity

Parse Tree of Left-Associative Grammar

- Parse tree for **5-3-2**

Exp → Exp – Num | Num



Please go figure the right-associative grammar!!!

Enforce left-associativity
(5-3) is evaluated first



Extended Grammars

- Backus-Naur Form (BNF)
  - is a **formal grammar** for expressing context-free grammars

- The single grammar rule format:

  => Non-terminal → zero or more grammar symbols

- Actually, BNF **extends** the grammar notation defined above, with syntax for defining **optional** and **repeated symbols**



Optional Symbols

- Optional symbols are enclosed in square brackets
- In the production rule

  $A \rightarrow \alpha\ [\ X1 \ldots Xn\ ]\ \beta$

  – the symbols X1 . . .Xn are **entirely present or absent** between the symbols of α and β

  – Refer to Fig. 4.4 for the algorithm to transform a BNF grammar into standard form



Repeated Symbols

- Repeated symbols are enclosed in braces
- In the production rule

  $B \rightarrow \gamma \{ X1 \ldots Xm \} \delta$

  - the entire sequence of symbols X1 . . .Xm can be **repeated zero or more times**
  - Refer to Fig. 4.4 for the algorithm to transform a BNF grammar into standard form, which is accepted by parsers

Example of BNF

- The extensions are useful in representing many programming language constructs
- For example, in Java
  - Declarations can optionally include modifiers, such as **final**, **static**, and **const**, and
  - each declaration can include a ***list*** of identifiers
  - A production specifying a Java-like declaration could be as follows:
    Declaration→[***final*** *] [* ***static*** *] [* ***const*** *] Type identifier {,* ***identifier****}*
  - Possible declarations:
    int a
    static int a, b, c
    final static int a
- This declaration insists that the modifiers be ordered as shown

Parsers and Recognizers

- Compilers are expected to
  - verify the **syntactic validity** of their **inputs** with respect to a **grammar** that
  - defines the programming language's syntax

- **A recognizer** is an algorithm determining if $x \in L(G)$, given a grammar G and an input string x
- **A parser** is an algorithm that determines both the string's validity **and** its *structure* (or *parse tree*)
  - The process of finding the structure (or building the parse tree) in the flat stream of tokens is called **parsing**



Two Parsing Approaches

- Top-down parsers
  - Left-scan, Leftmost derivation
  - Best-known parser in this category, called **LL** parsers
- Bottom-up parsers
  - Left-scan, Rightmost derivation in reverse
  - Best-known parser in this category, called **LR** parsers

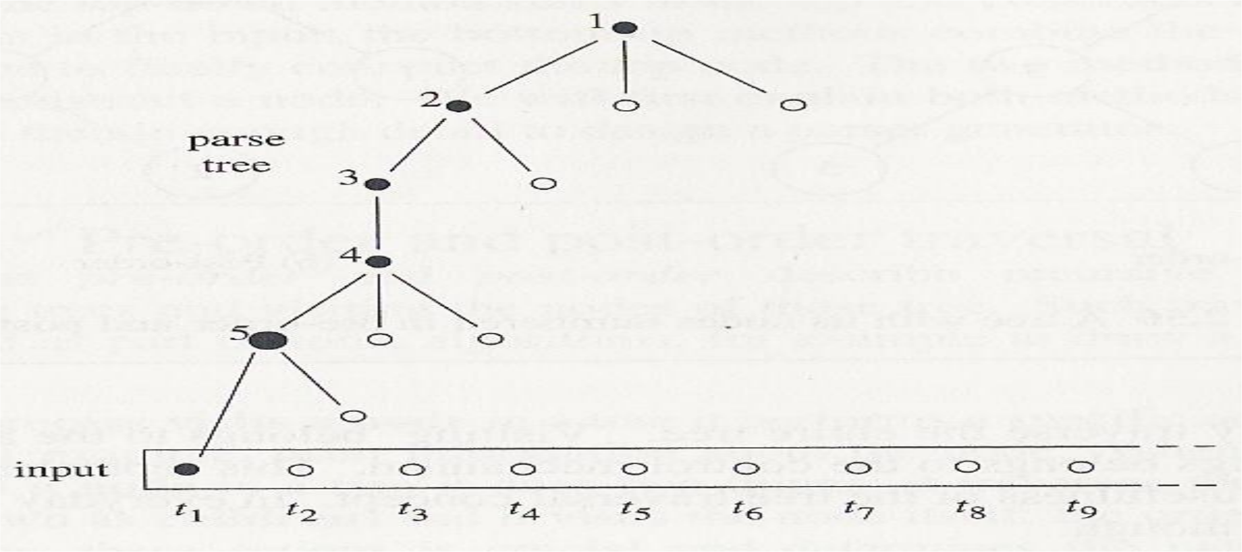


Top-Down Parsers

- A parser is considered **top-down**
  - if it generates a parse tree by starting at the root of the tree (the start symbol),
  - expanding the tree by applying productions in a **depth-first** manner
  - It corresponds to a **preorder traversal** of the parse tree
- Top-down parsing techniques are **predictive**
  - because they always predict the production that is to be matched before matching actually begins

Illustration of Top-Down Parsing

- A top-down parser begins by constructing the top node of the parse tree, which is the start symbol



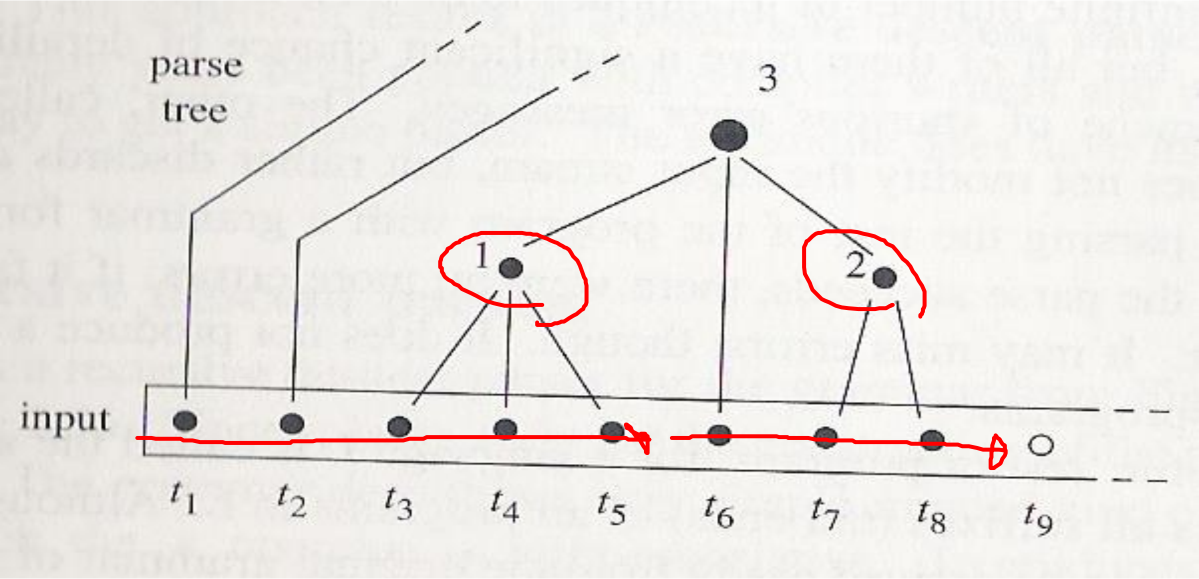


Bottom-Up Parsers

- The **bottom-up** parsers generate a parse tree by
  - starting at the tree's leaves and working toward its root
  - A node is inserted in the tree only after its children have been inserted
- A bottom-up parse corresponds to a **postorder traversal** of the parse tree

Illustration of Bottom-Up Parsing

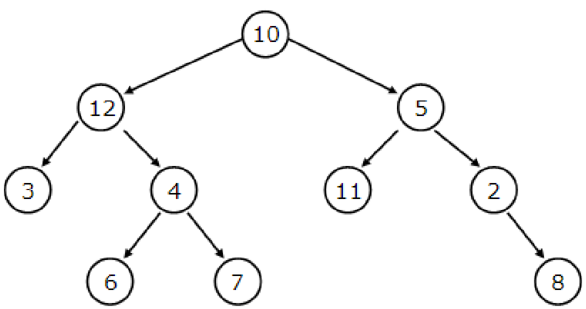
- The bottom-up parsing method constructs the nodes in the parse tree in post-order





To Refresh Your (and My) Memory

- Several ways for tree traversals



Levelorder tree traversal
10, 12, 5, 3, 4, 11, 2, 6, 7, 8

Inorder tree traversal
3, 12, 6, 4, 7, 10, 11, 5, 2, 8

Preorder tree traversal
10, 12, 3, 4, 6, 7, 5, 11, 2, 8

Postorder tree traversal
3, 6, 7, 4, 12, 11, 8, 2, 5, 10

Top-down parsers

Bottom-up parsers

- When traversing a node N in pre-order
  - the process first visits the node N and then traverses N's subtrees in left-to-right order
- When traversing a node N in post-order
  - the process first traverses N's subtrees in left-to-right order and then visits the node N

Quick note
Preorder: Root, Left, Right
Postorder: Left, Right, Root

Courtesy of http://www.namhpham.com/2015/10/post-order-tree-traversal-non-recursive.html



Example of the Parsers

- An example grammar generates the skeletal block structure of a hypothetical programming language

1 Program $\rightarrow$ begin Stmts end \$
2 Stmts $\rightarrow$ Stmt ; Stmts
3 | λ
4 Stmt $\rightarrow$ simplestmt
5 | begin Stmts end

- The Fig. 4.5 and 4.6 illustrate a top-down and bottom-up parse of the given string:

begin simplestmt ; simplestmt ; end \$

Top-Down Parsing Example

Legend:

1. Each box shows one step of the parse, with the particular rule denoted by bold lines between a parent (the rule's LHS) and its children (the rule's RHS)
2. Solid, non-bold lines indicate rules that have already been applied
3. Dashed lines indicate rules that have not yet been applied

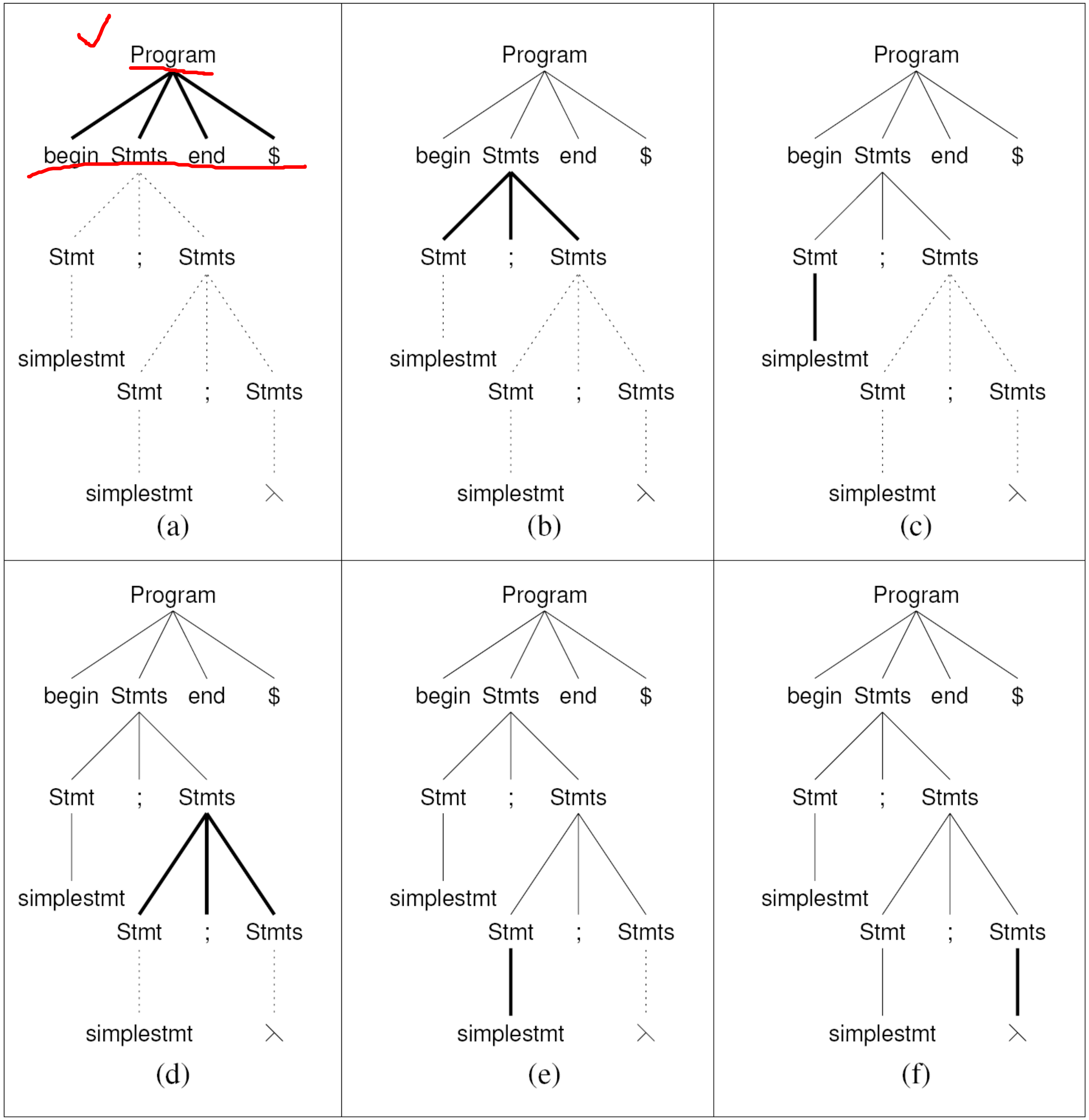


Figure 4.5: Parse of "begin simplestmt ; simplestmt ; end \$" using the top-down technique. Legend explained on page 126.

Top-Down Parsing Example'

- Fig. 4.5(a) shows the rule **Program→begin Stmts end $** applied as the first step of a top-down parse
- The red line indicates the next left-most non-terminal

Legend:
1. Each box shows one step of the parse, with the particular rule denoted by bold lines between a parent (the rule's LHS) and its children (the rule's RHS)
2. Solid, non-bold lines indicate rules that have already been applied
3. Dashed lines indicate rules that have not yet been applied

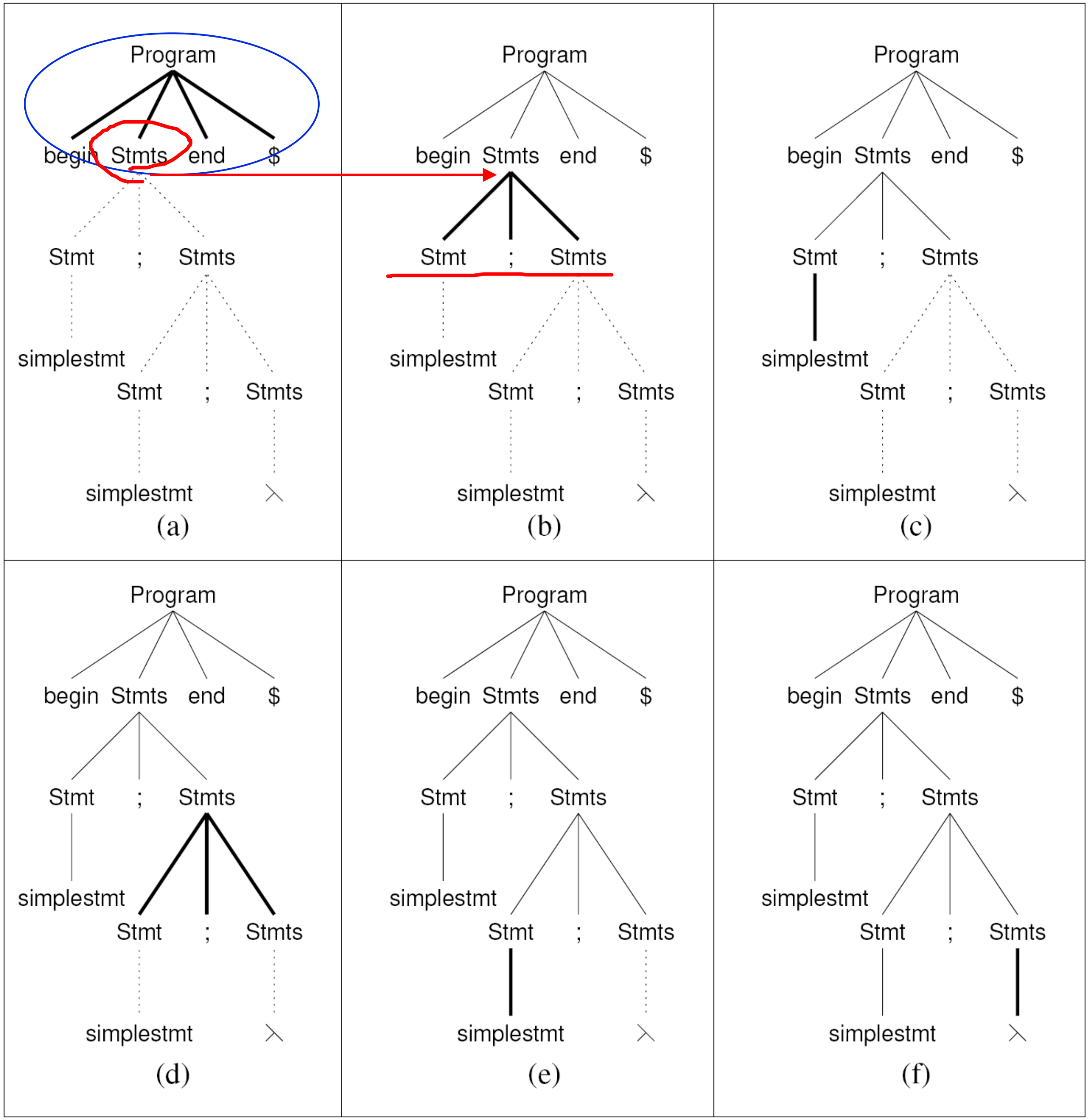


Figure 4.5: Parse of "begin simplestmt ; simplestmt ; end $" using the top-down technique. Legend explained on page 126.

Top-Down Parsing Example'

1 Program→ begin Stmts end $
2 Stmts → Stmt ; Stmts
3 | λ
4 Stmt → simplestmt
5 | begin Stmts end

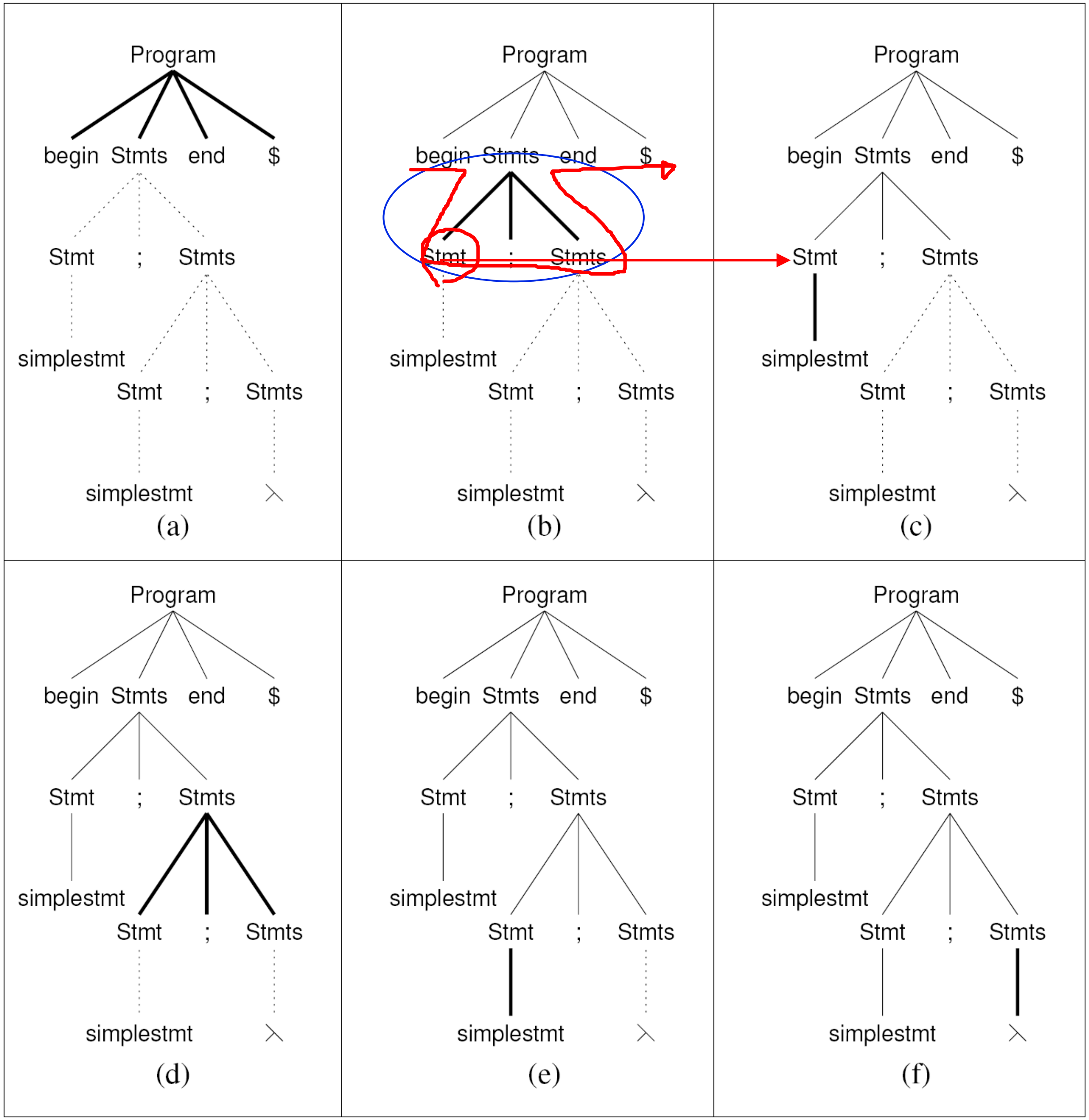


Figure 4.5: Parse of "begin simplestmt ; simplestmt ; end $" using the top-down technique. Legend explained on page 126.

Top-Down Parsing Example'

1 Program→ begin Stmts end $
2 Stmts → Stmt ; Stmts
3 | λ
4 Stmt → simplestmt
5 | begin Stmts end

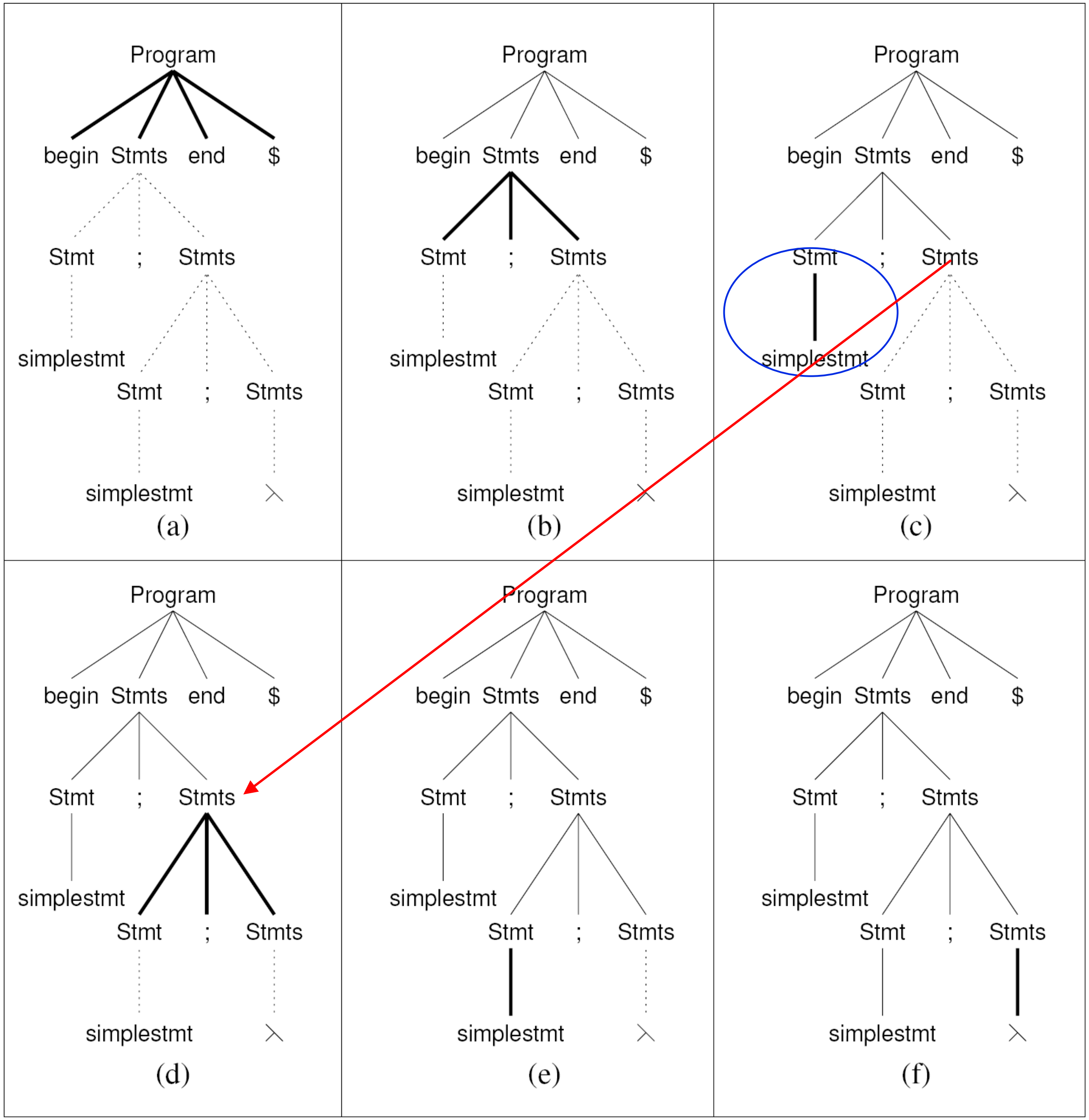


Figure 4.5: Parse of "begin simplestmt ; simplestmt ; end $" using the top-down technique. Legend explained on page 126.

Top-Down Parsing Example'

1 Program → begin Stmts end $
2 Stmts → Stmt ; Stmts
3 | λ
4 Stmt → simplestmt
5 | begin Stmts end

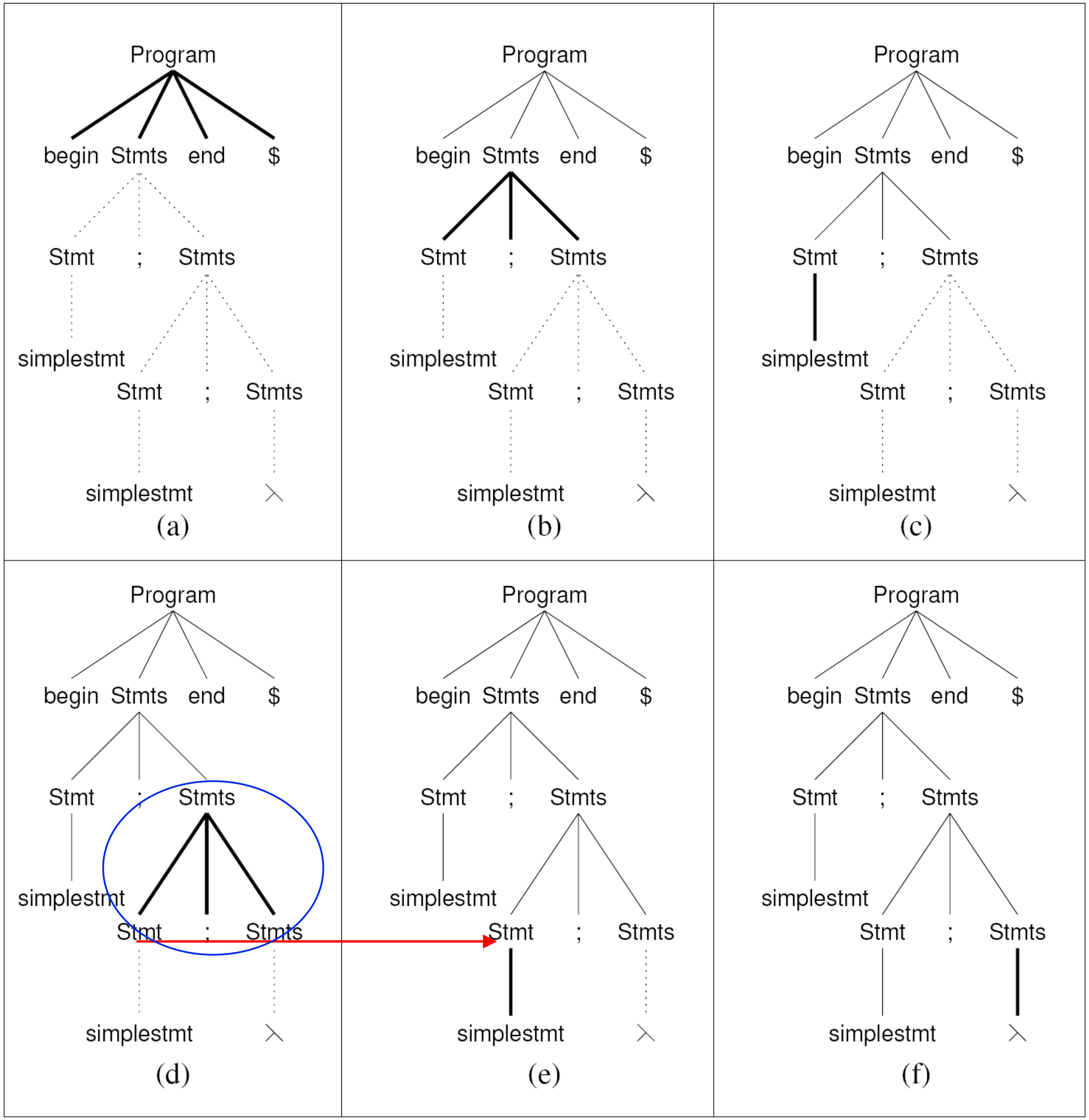


Figure 4.5: Parse of "begin simplestmt ; simplestmt ; end $" using the top-down technique. Legend explained on page 126.

Top-Down Parsing Example'

1 Program → begin Stmts end $
2 Stmts → Stmt ; Stmts
3 | λ
4 Stmt → simplestmt
5 | begin Stmts end

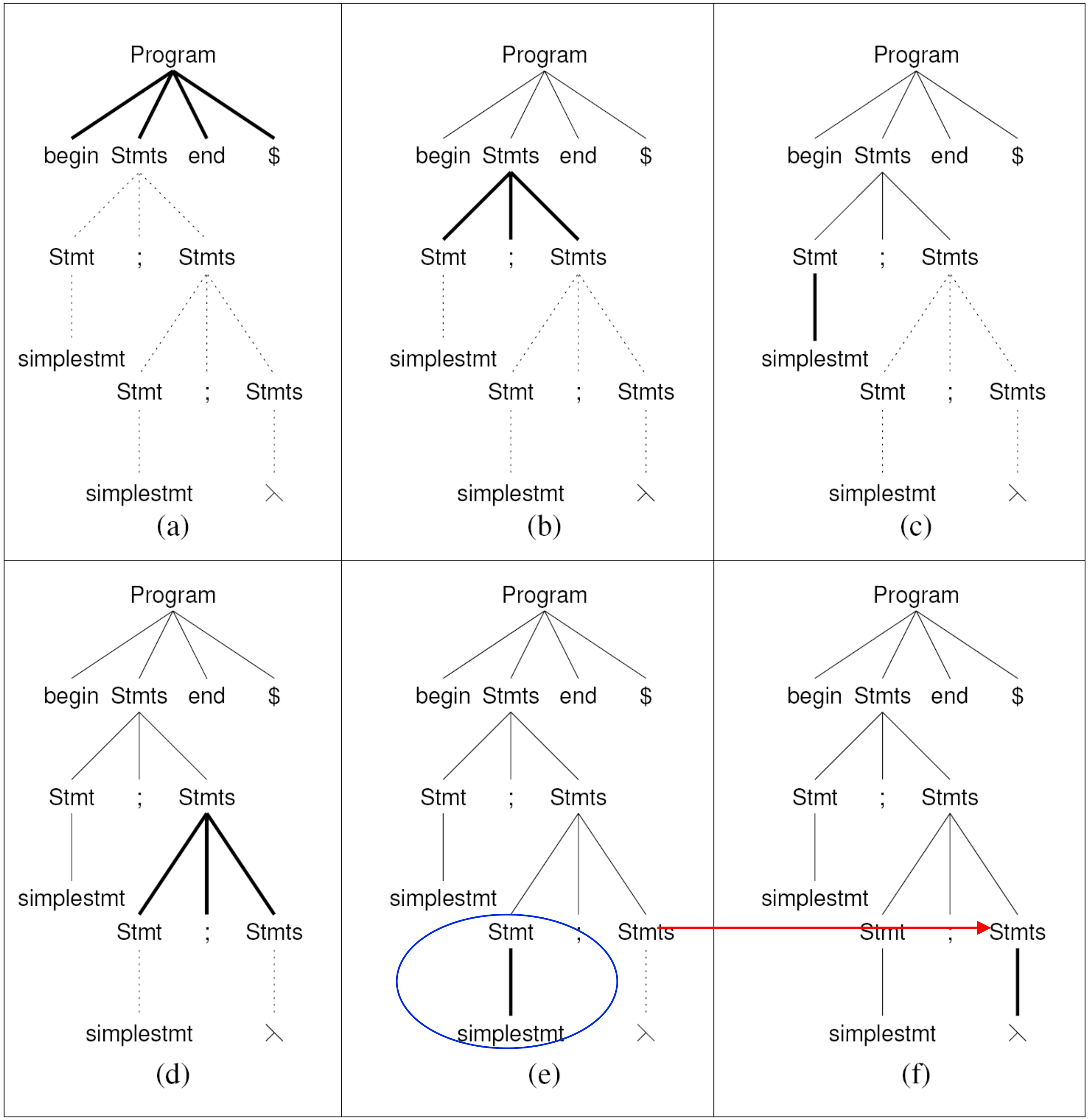


Figure 4.5: Parse of “begin simplestmt ; simplestmt ; end $” using the top-down technique. Legend explained on page 126.

Top-Down Parsing Example'

1 Program → begin Stmts end $
2 Stmts → Stmt ; Stmts
3 | λ
4 Stmt → simplestmt
5 | begin Stmts end

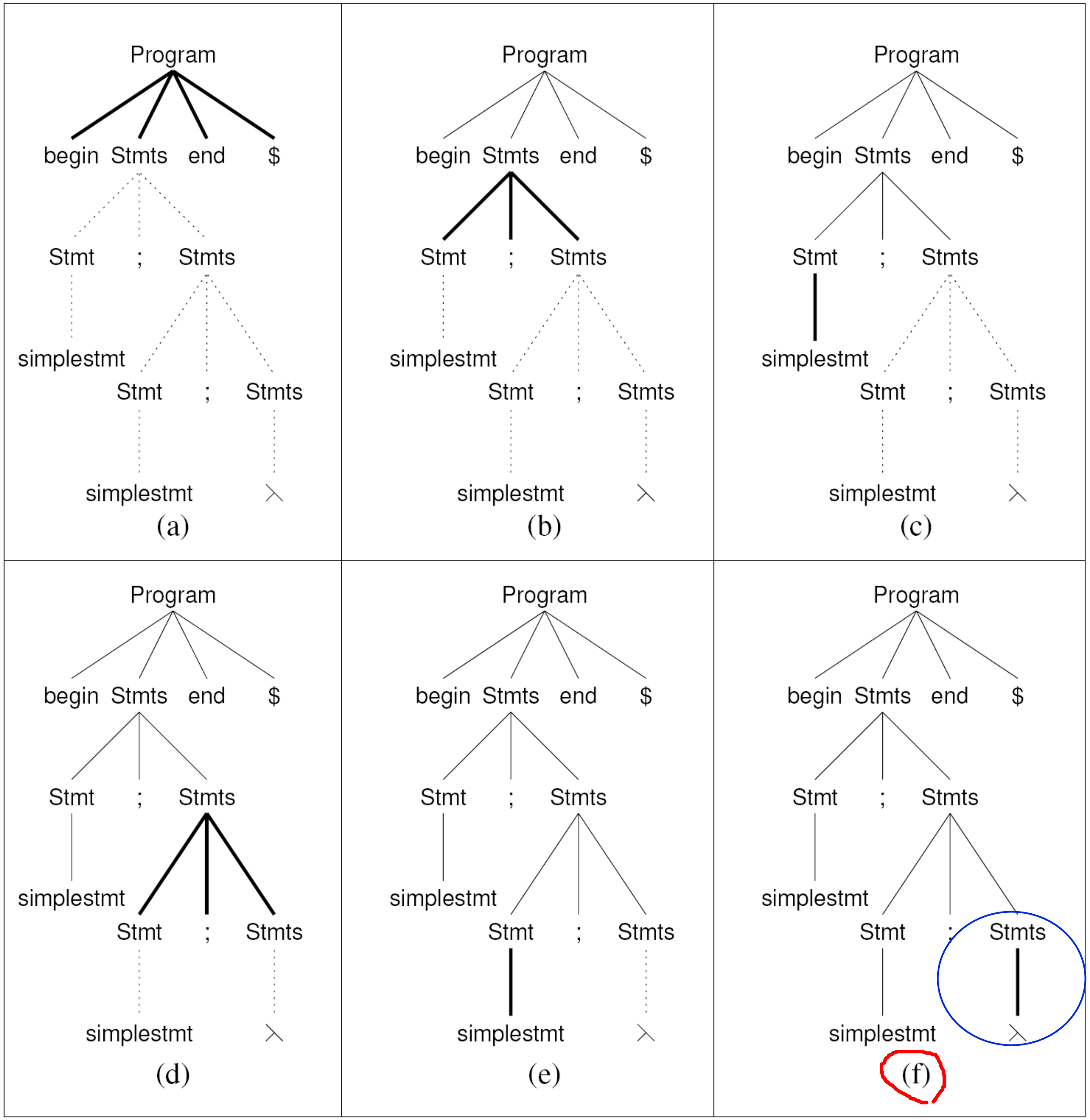


Figure 4.5: Parse of "begin simplestmt ; simplestmt ; end $" using the top-down technique. Legend explained on page 126.



You would …

- Practice the bottom-up parsing in Fig. 4.6
- If you get stuck
  - I assume you forget to read Section 4.1.1 and 4.1.2 ☺
  - Please carefully read the sections first and try it again

LL and LR Parsers

- LL and LR reflect
  - how the input is processed and which kind of parse is produced
  - The first character (L) states that the token sequence is processed from left to right
  - The second letter (L or R) indicates whether a leftmost or rightmost parse is produced
- The parsing technique can be further characterized by the **number of lookahead symbols**
  - i.e., symbols beyond the current token that the parser may consult to make
  - parsing choices
  - LL(1) and LR(1) parsers are the most common, requiring only one symbol of lookahead

Summary

- There is a many-to-one relationship between derivations and parse trees
  - A parse tree ignores variations in the order in which symbols in sentential forms are replaced
- While the parsing sequences of top-down and bottom-up parsing are different, two parsing techniques construct the same parse tree, as shown in Fig. 4.5 and 4.6



Why FIRST and FOLLOW?

str a

$A \rightarrow Ba$

$B \rightarrow a$

- FIRST and FOLLOW
  - are the two important functions for the construction of top-down and bottom-up parsers
- FIRST and FOLLOW allow the parsers to choose which production to apply
- During panic-mode error recovery, sets of tokens produced by FOLLOW can be used as synchronizing tokens

Why FIRST and FOLLOW? (Cont'd)

A very conceptual example

- Given a grammar, G

  1 S → B a

  2 B → b

  3 | λ

  - FIRST is to find **b**, which may be in the leading character for input string
  - FOLLOW is to find what is the next possible character when **B** is λ
  - The possible strings for L(G): ba, a

FIRST

- FIRST(α)
  - refers to the set of **terminals** that begin strings derived from α
  - where α is any string of grammar symbols

  Example:
    - If $A \Rightarrow^{*} cy$, then c is in FIRST(A)
    - If $\alpha \Rightarrow^{*} \lambda$, then then λ is also in FIRST (α)
- Given $A \Rightarrow a \mid b$
  - FIRST(A) is the union of FIRST (a) and FIRST(b),
  - where FIRST (a) and FIRST(b) are disjoint sets



FIRST (Cont'd)

- *Compute FIRST(X)
  - X is **grammar symbol**
  - We apply the following rules until no more terminals or λ can be added to it

1. If X is a terminal (i.e., $X \in \Sigma$), then FIRST(X)={X}
2. If $X \Rightarrow \lambda$ is a production (i.e., $X \in N$ and $X \Rightarrow \lambda$), then add λ to FIRST(X)
3. If X is a non-terminal (i.e., $X \in N$) and $X \Rightarrow Y_1 Y_2 \ldots Y_k$ is a production for some k>=1, then place α in FIRST(X) if for some i, α is in FIRST(Y_i), and λ is in all of FIRST(Y_1),…,FIRST(Y_{i-1})

If λ is in FIRST(Y_j) for all j=1,2,…,k, then add λ to FIRST(X)

*Courtesy of Compilers: Principles, Techniques, & Tools. Alfred V. Aho and et al. Pearson/Addison Wesley, 2007 (Section 4.4.2)



FIRST (Cont'd)

- More about the Rule 3 in the previous page

3. If X is a non-terminal (i.e., $X \in N$) and $X \Rightarrow Y_1 Y_2 \ldots Y_k$ is a production for some k>=1, then place α in FIRST(X) if for some i, α is in FIRST(Y_i), and λ is in all of FIRST(Y_1),…,FIRST(Y_{i-1})

- Examples:
  - Everything in FIRST(Y_1) is surely in FIRST(X)
    Hence, in normal case, we have: FIRST(X) = FIRST(Y_1)
  - If Y_1 does not derive λ, then we add nothing more to FIRST(X)
  - If $Y_1 \Rightarrow^* \lambda$, then we add FIRST($Y_2$) into FIRST(X), and so on

FIRST(α) Function

- **SymbolDerivesEmpty(A)** indicates whether or not the nonterminal A can derive λ
- **VisitedFirst(X)** is to indicate that the productions of X is already participate in the computation of FIRST(α)
- The argument in **InternalFirst(*Xβ*)** could be grammar symbol(s) in the LHS or RHS of a production rule
  - Given A ⇒ B, ***Xβ*** could be either A or B

```
function FIRST( α ) returns Set
    foreach A ∈ NONTERMINALS( ) do VisitedFirst(A) ← false          9
    ans ← INTERNALFIRST( α )
    return (ans)
end
function INTERNALFIRST( Xβ ) returns Set
    if Xβ = ⊥  ──────────────→ Rule 2                                10
    then return (∅)
    if X ∈ Σ  ──────────────→ Rule 1                                 11
    then return ({ X })
    /*    X is a nonterminal. ──────→ Rule 3                     */  12
    ans ← ∅
    if not VisitedFirst(X)
    then
        VisitedFirst(X) ← true                                       13
        foreach rhs ∈ ProductionsFor(X) do
            ans ← ans ∪ INTERNALFIRST( rhs )                         14
    if SymbolDerivesEmpty(X)                                         15
    then ans ← ans ∪ INTERNALFIRST( β )
    return (ans)                                                     16
end
```

Figure 4.8: Algorithm for computing First(α).



Handling Endless Recursion

$$A \Rightarrow B$$
$$\cdot \quad \cdot \quad \cdot$$
$$B \Rightarrow C$$
$$\cdot \quad \cdot \quad \cdot$$
$$C \Rightarrow A$$

```
function First(α) returns Set
    foreach A ∈ NonTerminals( ) do VisitedFirst(A) ← false      9
    ans ← InternalFirst(α)
    return (ans)
end
function InternalFirst(Xβ) returns Set
    if Xβ = ⊥                                                    10
    then return (∅)
    if X ∈ Σ                                                     11
    then return ({X})
    /*    X is a nonterminal.                                */  12
    ans ← ∅
    if not VisitedFirst(X)
    then
        VisitedFirst(X) ← true                                   13
        foreach rhs ∈ ProductionsFor(X) do
            ans ← ans ∪ InternalFirst(rhs)                       14
    if SymbolDerivesEmpty(X)                                     15
    then ans ← ans ∪ InternalFirst(β)
    return (ans)                                                 16
end
```

Figure 4.8: Algorithm for computing First(α).

- Termination of FIRST(α) must be handled properly in grammars
  - where **the computation of FIRST(α) appears to depend on FIRST(α)**
- **VisitedFirst(X)** is to indicate that the productions of X is already participate in the computation of FIRST(α)



Example

Given the grammar,

- $E \Rightarrow TE'$
- $E' \Rightarrow +TE' \mid \lambda$
- $T \Rightarrow FT'$
- $T' \Rightarrow *FT' \mid \lambda$
- $F \Rightarrow (E) \mid id$

where E is for expression, T is for term, and F is for factor.
Please find the FIRST set of each symbol

- FIRST sets for E, T, F, E′, and T′:
  - FIRST(E) = FIRST(T) = FIRST(F) = {(, id }
  - FIRST(E′) = {+, λ}
  - FIRST(T′) = {*, λ}

Example (Cont'd)

Compute FIRST(X)

- X is **grammar symbol**
- We apply the following rules until no more terminals or λ can be added to it

1. If X is a terminal (i.e., $X \in \Sigma$), then FIRST(X)={X}
2. If $X \Rightarrow \lambda$ is a production (i.e., $X \in N$ and $X \Rightarrow \lambda$), then add λ to FIRST(X)
3. If X is a non-terminal (i.e., $X \in N$) and $X \Rightarrow Y_1 Y_2 \ldots Y_k$ is a production for some k>=1, then place α in FIRST(X) if for some i, α is in FIRST(Y_i), and λ is in all of FIRST(Y_1),…,FIRST(Y_{i-1})

   If λ is in FIRST(Y_j) for all j=1,2,…,k, then add λ to FIRST(X)

Given the grammar,

- $E \Rightarrow TE'$
- $E' \Rightarrow +TE' \mid \lambda$
- $T \Rightarrow FT'$
- $T' \Rightarrow *FT' \mid \lambda$
- $F \Rightarrow (E) \mid id$

where E is for expression, T is for term, and F is for factor. Please find the FIRST set of each symbol

- FIRST sets for E, T, F, E', and T':
  - FIRST(E) $\overset{3}{=}$ FIRST(T) $\overset{3}{=}$ FIRST(F) $\overset{3,1,1}{=}$ {(, id }
  - FIRST(E') = {+, λ}
  - FIRST(T') = {*, λ}

Given: $F \Rightarrow (E) \mid id$

FIRST(F) $\overset{3}{=}$ FIRST(() ∪ FIRST (id)
FIRST(() $\overset{1}{=}$ {(}
FIRST(id) $\overset{1}{=}$ {id}

Example (Cont'd)

Compute FIRST(X)

- X is **grammar symbol**
- We apply the following rules until no more terminals or λ can be added to it

1. If X is a terminal (i.e., $X \in \Sigma$), then FIRST(X)={X}
2. If $X \Rightarrow \lambda$ is a production (i.e., $X \in N$ and $X \Rightarrow \lambda$), then add λ to FIRST(X)
3. If X is a non-terminal (i.e., $X \in N$) and $X \Rightarrow Y_1Y_2\ldots Y_k$ is a production for some k>=1, then place α in FIRST(X) if for some i, α is in FIRST(Y_i), and λ is in all of FIRST(Y_1),…,FIRST(Y_{i-1})

   If λ is in FIRST(Y_j) for all j=1,2,…,k, then add λ to FIRST(X)

Given the grammar,

- $E \Rightarrow TE'$
- $E' \Rightarrow +TE' \mid \lambda$
- $T \Rightarrow FT'$
- $T' \Rightarrow *FT' \mid \lambda$
- $F \Rightarrow (E) \mid id$

where E is for expression, T is for term, and F is for factor. Please find the FIRST set of each symbol

- FIRST sets for E, T, F, E′, and T′:
  - FIRST(E) = FIRST(T) = FIRST(F) = {(, id }
  - FIRST(E′) = {+, λ}
  - FIRST(T′) = {*, λ}

Given: $E' \Rightarrow +TE' \mid \lambda$

FIRST(E') = FIRST ($+TE'$) ∪ FIRST (λ)

FIRST($+TE'$) = {+}

FIRST(λ) = {λ}



Another Example for FIRST

Given the grammar,

- *input* $\Rightarrow$ *expression*
- *expression* $\Rightarrow$ *term rest_expression*
- *term* $\Rightarrow$ *ID* | *parenthesized_expression*
- *parenthesized_expression* $\Rightarrow$ '(' *expression* ')'
- *rest_expression* $\Rightarrow$ '+' *expression* | λ

FIRST sets for *input*, *expression*, *term*, *parenthesized_expression*, and *rest_expression*:

- FIRST (*input*) = *FIRST*(*expression*) =FIRST (*term*) ={ ID, (}
- FIRST (*parenthesized_expression*) = { (}
- FIRST (*rest_expression*) = { +, λ}

You Should …

- Refer to Section 4.5.3 to do the exercise in Fig. 4.9 and 4.10

Textbook Exercise

- Note the *ans* does not contain λ, which will cause trouble for you to derive the results

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

Figure 4.1: A simple expression grammar.

```
function FIRST(α) returns Set
    foreach A ∈ NONTERMINALS( ) do VisitedFirst(A) ← false        (9)
    ans ← INTERNALFIRST(α)
    return (ans)
end
function INTERNALFIRST(Xβ) returns Set
    if Xβ = ⊥                                                      (10)
    then return (∅)
    if X ∈ Σ                                                       (11)
    then return ({X})
    /*  X is a nonterminal.                                     */ (12)
    ans ← ∅
    if not VisitedFirst(X)
    then
        VisitedFirst(X) ← true                                     (13)
        foreach rhs ∈ ProductionsFor(X) do
            ans ← ans ∪ INTERNALFIRST(rhs)                         (14)
    if SymbolDerivesEmpty(X)                                       (15)
    then ans ← ans ∪ INTERNALFIRST(β)
    return (ans)                                                   (16)
end
```

Figure 4.8: Algorithm for computing First(α).

Level	First $\mathcal{X}$	First β	*ans*	Marker	Done? (★=Yes)	Comment
			FIRST(Tail)			
0	Tail	⊥	{ }	12		
1	+	E	{ + }	11	★	Tail → +E
1	⊥	⊥	{ }	10	★	Tail → λ
0			{ + }	14		After all rules for Tail
1	⊥	⊥	{ }	10	★	Since $\beta = \perp$
0			{ + }	15	★	Final answer
			FIRST(Prefix)			
0	Prefix	⊥	{ }	12		
1	f	⊥	{ f }	11	★	Prefix → f
1	⊥	⊥	{ }	10	★	Prefix → λ
0			{ f }	14		After all rules for Prefix
1	⊥	⊥	{ }	10	★	Since $\beta = \perp$
0			{ f }	15	★	Final answer
			FIRST(E)			
0	E	⊥	{ }	12		
1	Prefix	(E)	{ }	12		E → Prefix (E)
1			{ f }	16		Computation shown above
2	(	E)	{ (}	11	★	Since Prefix $\Rightarrow^\star$ λ
1			{ f,(}	15	★	Results due to E → Prefix (E)
1	v	Tail	{ v }	11	★	E → v Tail
1	⊥	⊥	{ }	10		Since $\beta = \perp$
0			{ f,(,v }	15	★	Final answer

Figure 4.9: First sets for the nonterminals of Figure 4.1.

FOLLOW

- FOLLOW(β)
  - refers to the set of terminals **a** that can appear immediately to the right of non-terminal β in some sentential form

Example:

α

- If $S \Rightarrow \alpha A a \beta$, then **a** is in the set of FOLLOW(A)

FOLLOW (Cont'd)

- *FOLLOW(B)
  - where B is non-terminal,
  - *S* is the start symbol for the grammar, and
  - $ is the input right end-marker
  - We apply all the following rules for all nonterminals B until nothing can be added

$S \rightarrow \cdots \alpha B \beta a$

$\alpha B \beta$

1. Place $ in FOLLOW(*S*)
2. If there is a production A ⇒ αBβ, then everything in **FIRST(β)** except λ is added to FOLLOW(B)
3. If there is a production
   (a) A ⇒ αB, or
   (b) A ⇒ αBβ, where FIRST(β) contains λ,
   then everything in **FOLLOW(A)** is added to FOLLOW(B)

$S \rightarrow \cdots Aa$

$A \rightarrow \alpha B \beta$

$\beta \rightarrow \lambda$

*Courtesy of Compilers: Principles, Techniques, & Tools. Alfred V. Aho and et al. Pearson/Addison Wesley, 2007 (Section 4.4.2)

FOLLOW (A) Function I

```
function FOLLOW(A) returns Set
    foreach A ∈ NONTERMINALS( ) do
        VisitedFollow(A) ← false                              (17)
    ans ← INTERNALFOLLOW(A)
    return (ans)
end
function INTERNALFOLLOW(A) returns Set
    ans ← ∅
    if not VisitedFollow(A)                                   (18)
    then
        VisitedFollow(A) ← true                               (19)
        foreach a ∈ OCCURRENCES(A) do                         (20)
            ans ← ans ∪ FIRST(TAIL(a))                        (21) → Rule 2
            if ALLDERIVEEMPTY(TAIL(a))                        (22)
            then
                targ ← LHS(PRODUCTION(a))
                ans ← ans ∪ INTERNALFOLLOW(targ)              (23) → Rule 3
    return (ans)                                              (24)
end
function ALLDERIVEEMPTY(γ) returns Boolean
    foreach X ∈ γ do
        if not SymbolDerivesEmpty(X) or X ∈ Σ
        then return (false)
    return (true)                                                  → Rule 3
end
```

(when argument γ is empty set (3a) or the following non-terminals in γ all contain λ (3b))

Figure 4.11: Algorithm for computing Follow(A).

- **SymbolDerivesEmpty(A)** indicates whether or not the nonterminal A can derive λ
- **VisitedFollow(X)** is to indicate that the productions of X is already participate in the computation of FOLLOW(A)
- **Occurrences(A)** finds and lists all the appearances of A in the production rules of the given Grammar (**Marker 21**)
- **What happens to \`$'?**
  It is fine. Rule 1 is not defined in the primary textbook

FOLLOW (A) Function II

```
function FOLLOW(A) returns Set
    foreach A ∈ NONTERMINALS( ) do
        VisitedFollow(A) ← false                              (17)
    ans ← INTERNALFOLLOW(A)
    return (ans)
end
function INTERNALFOLLOW(A) returns Set
    ans ← ∅
    if not VisitedFolow(A)                                    (18)
    then
        VisitedFollow(A) ← true                               (19)
        foreach a ∈ OCCURRENCES(A) do                         (20)
            ans ← ans ∪ FIRST(TAIL(a))                        (21) → Rule 2
            if ALLDERIVEEMPTY(TAIL(a))                        (22)
            then
                targ ← LHS(PRODUCTION(a))
                ans ← ans ∪ INTERNALFOLLOW(targ)              (23) → Rule 3
    return (ans)                                              (24)
end
function ALLDERIVEEMPTY(γ) returns Boolean
    foreach X ∈ γ do
        if not SymbolDerivesEmpty(X) or X ∈ Σ
        then return (false)
    return (true)                                                  → Rule 3
end
```

(when argument γ is empty set (3a) or the following non-terminals in γ all contain λ (3b))

Figure 4.11: Algorithm for computing Follow(A).

- (**Marker 21**) **Tail(a)** is the list of symbols immediately following the occurrence of **A**
  - S ⇒ ABC
  - Tail(a) is BC
- (**Marker 22**) detects if all of the symbols in **Tail(a)** could derive λ
  - This is different from the FOLLOW definition of Dragon Book, which considers S ⇒ αAβ with one symbol at A's tail
  - Fig. 4.11 considers a long tail: more than one symbols after A, e.g., S ⇒ **A**BC
  - Done by **AllDeriveEmpty(γ)**, where **γ==Tail(a)==BC**
- (**Marker 23**) if **Tail(a)** could be λ, we include FOLLOW of **LHS(CurrentOccurrence(A))**
  - If S ⇒ **A**BC and Tail(a) could be λ,
  - we add FOLLOW(S) to FOLLOW(**A**)
- **SymbolDerivesEmpty(A)** indicates whether or not the nonterminal A can derive λ



Example

Given the grammar,

- $E \Rightarrow TE'$
- $E' \Rightarrow +TE' \mid \lambda$
- $T \Rightarrow FT'$
- $T' \Rightarrow *FT' \mid \lambda$
- $F \Rightarrow (E) \mid id$

where E is for expression, T is for term, and F is for factor

Please find the FOLLOW set of each symbol

Example (Cont’d)

Given the grammar,

- $E \Rightarrow TE'$
- $E' \Rightarrow +TE' \mid \lambda$
- $T \Rightarrow FT'$
- $T' \Rightarrow *FT' \mid \lambda$
- $F \Rightarrow (E) \mid id$

E is start symbol

- FIRST sets for E, T, F, E’, and T’:
  - FIRST(E) = FIRST(T) = FIRST(F) = {(, id }
  - FIRST(E’) = {+, λ}
  - FIRST(T’) = {*, λ}

- FOLLOW sets for E, T, F, E’, and T’:
  - FOLLOW(E) = {\$,)}
  - FOLLOW(E’) = FOLLOW(E) = {\$,)}
  - FOLLOW(T) = FIRST(E’)+FOLLOW(E’) = {+, \$,) }
  - FOLLOW(T’) = FOLLOW(T)= {+, \$,) }
  - FOLLOW(F) = FIRST(T’)+FOLLOW(T) + FOLLOW(T’)= {*, +, \$,)}

FOLLOW(B)

1. Place \$ in FOLLOW(S)
2. If there is a production A ⇒ αBβ, then everything in FIRST(β) except λ is added to FOLLOW(B)
3. If there is a production A ⇒ αB, or A ⇒ αBβ, where FIRST(β) contains λ, then everything in FOLLOW(A) is added to FOLLOW(B)

Example (Cont'd)

Given the grammar,

- $E \Rightarrow TE'$
- $E' \Rightarrow +TE' \mid \lambda$
- $T \Rightarrow FT'$
- $T' \Rightarrow *FT' \mid \lambda$
- $F \Rightarrow (E) \mid id$

E is start symbol

- FIRST sets for E, T, F, E', and T':
  - FIRST(E) = FIRST(T) = FIRST(F) = {(, id }
  - FIRST(E') = {+, λ}
  - FIRST(T') = {*, λ}
- FOLLOW sets for E, T, F, E', and T':
  - FOLLOW(E) = {$,)}
  - FOLLOW(E') = FOLLOW(E) = {$,)}
  - FOLLOW(T) = FIRST(E')+FOLLOW(E') = {+, $,) }
  - FOLLOW(T') = FOLLOW(T)= {+, $,) }
  - FOLLOW(F) = FIRST(T')+FOLLOW(T) + FOLLOW(T')= {*, +, $,)}

FOLLOW(B)

1. Place $ in FOLLOW(S)
2. If there is a production A ⇒ αBβ, then everything in FIRST(β) except λ is added to FOLLOW(B)
3. If there is a production A ⇒ αB, or A ⇒ αBβ, where FIRST(β) contains λ, then everything in FOLLOW(A) is added to FOLLOW(B)

E is used in two productions.

Given: $E \Rightarrow TE'$ and *E* is start symbol

FOLLOW(*E*) $\stackrel{1}{=}$ {$}

Given: $F \Rightarrow (E) \mid id$

FOLLOW(*E*) $\stackrel{2}{=}$ FIRST()) = {)}

➔ FOLLOW(*E*) = {$} ∪ {)}

Given: $E \Rightarrow TE'$

FOLLOW(*E'*) $\stackrel{3a}{=}$ FOLLOW(*E*)

Example (Cont'd)

Given the grammar,

- $E \Rightarrow TE'$
- $E' \Rightarrow +TE' \mid \lambda$
- $T \Rightarrow FT'$
- $T' \Rightarrow *FT' \mid \lambda$
- $F \Rightarrow (E) \mid id$

E is start symbol

- FIRST sets for E, T, F, E′, and T′:
  - FIRST(E) = FIRST(T) = FIRST(F) = {(, id }
  - FIRST(E′) = {+, λ}
  - FIRST(T′) = {*, λ}

- FOLLOW sets for E, T, F, E′, and T′:
  - FOLLOW(E) = {$,)}
  - FOLLOW(E′) = FOLLOW(E) = {$,)}
  - FOLLOW(T) = FIRST(E′)+FOLLOW(E′) = {+, $,) }
  - FOLLOW(T′) = FOLLOW(T)= {+, $,) }
  - FOLLOW(F) = FIRST(T′)+FOLLOW(T) + FOLLOW(T′)= {*, +, $,)}

FOLLOW(B)

1. Place $ in FOLLOW(S)
2. If there is a production A ⇒ αBβ, then everything in FIRST(β) except λ is added to FOLLOW(B)
3. If there is a production A ⇒ αB, or A ⇒ αBβ, where FIRST(β) contains λ, then everything in FOLLOW(A) is added to FOLLOW(B)

Given:

$E \Rightarrow TE'$

$E' \Rightarrow +TE' \mid \lambda$

$FOLLOW(T) \overset{2}{=} FIRST(E') = \{+\}$

$FOLLOW(T) \overset{3b}{=} FOLLOW(E') = \{\$,)\}$

➔ $FOLLOW(T) = \{+\} \cup \{\$,)\}$

Another Example for FOLLOW

Given the grammar,

- *input* ⇒ *expression*
- *expression* ⇒ *term rest_expression*
- *term* ⇒ *ID* | *parenthesized_expression*
- *parenthesized_expression* ⇒ ‘(‘ *expression* ‘)’
- *rest_expression* ⇒ ‘+’ *expression* | *λ*

FOLLOW(B)

1. Place $ in FOLLOW(S)
2. If there is a production A ⇒ αBβ, then everything in FIRST(β) except λ is added to FOLLOW(B)
3. If there is a production (a) A ⇒ αB, or (b) A ⇒ αBβ, where FIRST(β) contains λ, then everything in FOLLOW(A) is added to FOLLOW(B)

FOLLOW sets for input, expression, term, parenthesized_expression, and rest_expression

- FOLLOW (input) = {$ } Rule 1
- FOLLOW (expression) = {$,)} Rule 3(a) got $; Rule 2 got)
- FOLLOW (term) = FOLLOW (parenthesized_expression) Rule3(a)
  = {+, $,) } Rule 2 got +; Rule 3(b) got $)
- FOLLOW (rest_expression) = { $,)} Rule 3(a)



You Should …

- Refer to Section 4.5.4 to do the exercise in Fig. 4.12 and 4.13



Just Another Example for FOLLOW

- Fig. 4.10 grammar
- Can you derive the FOLLOW sets for A, B?
- Because of different definition used in the book, S does not contain $

```
1  S → A B c
2  A → a
3    | λ
4  B → b
5    | λ
```

```
function Follow(A) returns Set
    foreach A ∈ NonTerminals() do
        VisitedFollow(A) ← false                          (17)
    ans ← InternalFollow(A)
    return (ans)
end
function InternalFollow(A) returns Set
    ans ← ∅
    if not VisitedFolow(A)                                (18)
    then
        VisitedFollow(A) ← true                           (19)
        foreach a ∈ Occurrences(A) do                     (20)
            ans ← ans ∪ First(Tail(a))                    (21)
            if AllDeriveEmpty(Tail(a))                    (22)
            then
                targ ← LHS(Production(a))
                ans ← ans ∪ InternalFollow(targ)          (23)
    return (ans)                                          (24)
end
function AllDeriveEmpty(γ) returns Boolean
    foreach X ∈ γ do
        if not SymbolDerivesEmpty(X) or X ∈ Σ
        then return (false)
    return (true)
end
```

Figure 4.11: Algorithm for computing Follow(A).

Level	Rule	Marker	Result	Comment
			Follow(B)	
0			Follow(B)	
0	S → A <u>B</u> c	21	{ c }	
0		24	{ c }	Returns
			Follow(A)	
0			Follow(A)	
0	S → <u>A</u> B c	21	{ b,c }	
0		24	{ b,c }	Returns
			Follow(S)	
0			Follow(S)	
0		24	{ }	Returns

Figure 4.12: Follow sets for the grammar in Figure 4.10. Note that Follow(S) = { } because S does not appear on the RHS of any production.



NOTE: By default, we use the rules defined in FIRST(X) and FOLLOW(B) in our examinations, which mean that your answers should include λ and $, when necessary.

QUESTIONS?



Functions Definitions

- Grammar(S)
  - Creates a new grammar with start symbol S
  - The grammar does not yet contain any productions
- Production(A, rhs)
  - Creates a newproduction for nonterminal A and returns a descriptor for the production
  - The iterator rhs supplies the symbols for the production's RHS
- Productions( )
  - Returns an iterator that visits each of the grammar's productions in no particular order
- Nonterminal(A)
  - Adds A to the set of nonterminals. An error occurs if A is already a terminal symbol
  - The function returns a descriptor for the nonterminal
- Terminal(x)
  - Adds x to the set of terminals
  - An error occurs if x is already a nonterminal symbol. The function returns a descriptor for the terminal
- NonTerminals( )
  - Returns an iterator for the set of nonterminals



Functions Definitions (Cont'd)

- Terminals( )
  - Returns an iterator for the set of terminal symbols
- IsTerminal(X)
  - Returns true if X is a terminal; otherwise, returns false
- RHS(p)
  - Returns an iterator for the symbols on the RHS of production p
- LHS(p)
  - Returns the nonterminal defined by production p
- ProductionsFor(A)
  - Returns an iterator that visits each production for nonterminal A
- Occurrences(X)
  - Returns an iterator that visits each occurrence of X in the RHS of all rules
- Production(y)
  - Returns a descriptor for the production $A \rightarrow \alpha$ where α contains the occurrence y of some vocabulary symbol
- Tail(y)
  - Accesses the symbols appearing after an occurrence
  - Given a symbol occurrence y in the rule $A \rightarrow \alpha\ y\ \beta$, Tail(y) returns an iterator for the symbols in β