Chapter 5 Context-Free Languages

Context-Free Grammars (CFG)

Parsing and Ambiguity

CFG and Programming Languages

Needs for CFG

- RL is effective in describing certain simple pattern
- Regular expressions were inadequate to define all interesting languages, such as (()), nested structure in programming languages
- CFL has important applications in the design of programming languages as well as in the construction of efficient compilers

5.1 Context-Free Grammars

Definition 5.1

- A grammar $G = (V, T, S, P)$ is said to be **context-free** if all productions in P have the form
  - $A \rightarrow x$
  - where $A \in V$ and $x \in (V \cup T)^*$.
- A language L is said to be context-free if and only if there is a CFG G such that $L = L(G)$

Grammar Set Notations:
a concise way of design description

G = (V, T, S, P)

T – a set of terminal symbols
V – a set of nonterminal symbols
S – a element of V, starting symbol
P – a set of production rules with certain format restrictions

Write out the production rule format in set notation for each of the following grammars :

- RG
- CFG

Context Free Grammar (CFG): the representation defining CFL

- A grammar is said to be context-free if all productions in P have the form
  - $A \rightarrow x$
  - English description:
    - where A is a single nonterminal and x can be any string of terminals or/and nonterminals
    - Single nonterminal of the left handside of the production rule makes the grammar context free
  - Set notation:
    - where $A \in V$ and $x \in (V \cup T)^*$

Regular Grammars (RG):
one of 4 representations of RL

- A subset of CFG
- A CFG is said to be regular grammar if all productions in P have the form
  - $A \rightarrow xB$, $A \rightarrow x$; or $A \rightarrow Bx$, $A \rightarrow x$
  - English:
    - Where A is a single nonterminal and x can be word (string of terminals) or semiword (a string of terminals ended or starting with a nonterminals
  - Set notation:
    - Where $A, B \in V$, and $x \in T^*$

Example 5.1
$G = (\{S\}, \{a, b\}, S, P\})$

- is context-free with productions
  - $S \to aSa$
  - $S \to bSb$
  - $S \to \lambda$
- A derivation in this grammar is
  - $S \Rightarrow aSa \Rightarrow aaSaa \Rightarrow aabSbaa \Rightarrow aabbaa$.
  - We can derive that $L(G) = \{ww^R : w \in \{a, b\}^*\}$.
  - The language is context-free, but is not regular (as shown in Example 4.8)

Example 5.2

- The grammar G is context-free with productions
  - $S \to abB$
  - $A \to aaBb$
  - $B \to bbAa$
  - $A \to \lambda$
- Show that
  - $L(G) = \{ab(bbaa)^n bba(ba)^n : n \geq 0\}$.

Example 5.3 -- 1

To show L = $\{a^n b^m : n \neq m\}$ is context-free

- We need to produce a CFG for L
- We know how to produce a CFG with n = m
  - Write out
- Now we need to add 2 sets of productions
  - For the case n > m we add the first set
    - $S \to AS_1$
    - $S_1 \to a\ S_1 b | \lambda$
    - $A \to aA|a$

Example 5.3 -- 2

To show $L = \{a^n b^m : n \neq m\}$ is context-free

- For the case $n < m$, we add another similar set, and we get the answer
  - $S \rightarrow AS_1 | S_1B$
  - $S_1 \rightarrow a S_1 b | \lambda$
  - $A \rightarrow aA | a$
  - $B \rightarrow bB | b$
- The resulting grammar is context-free, hence L is CFL.

Example 5.4

programming language L_2 include strings such as (()) and ()()()

- CFL L can be generated by CFG with productions
  - $S \rightarrow aSb|SS|\lambda$
  - Substitute a with (and b with) in the productions, we can generate L_2 above
  - $L = \{w \in \{a, b\}^* : n_a(w) = n_b(w) \text{ and } n_a(v) \geq n_b(v)$, where v is any prefix of w$\}$

Leftmost and Rightmost Derivations

- Definition 5.2
  - A derivation is said to be **leftmost** if in each step the leftmost variable in the sentential form is replaced. If in each step the right most variable is replaced, we call the derivation **rightmost**.

Example 5.5

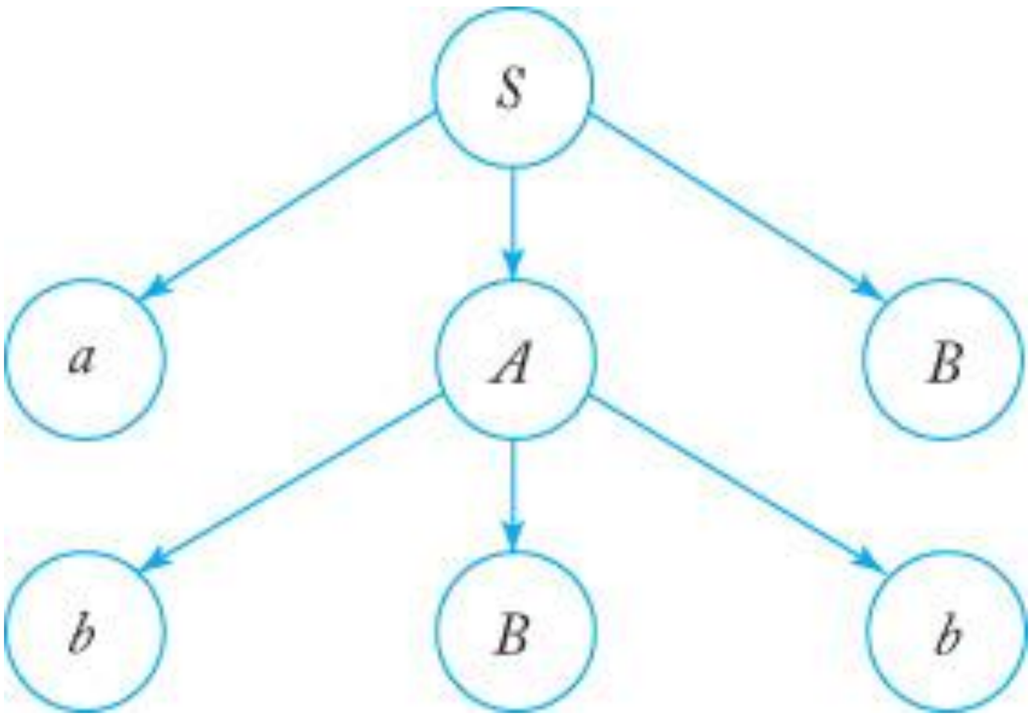
Leftmost and Rightmost Derivations

- $S \rightarrow aAB$
- $A \rightarrow bBb$
- $B \rightarrow A|\lambda$
  - Leftmost derivation of abbbb
    - $S \Rightarrow aAB \Rightarrow abBbB \Rightarrow abAbB \Rightarrow abbBbbB \Rightarrow abbbbB \Rightarrow abbbb$
  - Rightmost derivation of abbbb
    - $S \Rightarrow aAB \Rightarrow aA \Rightarrow abBb \Rightarrow abAb \Rightarrow abbBbb \Rightarrow abbbb$

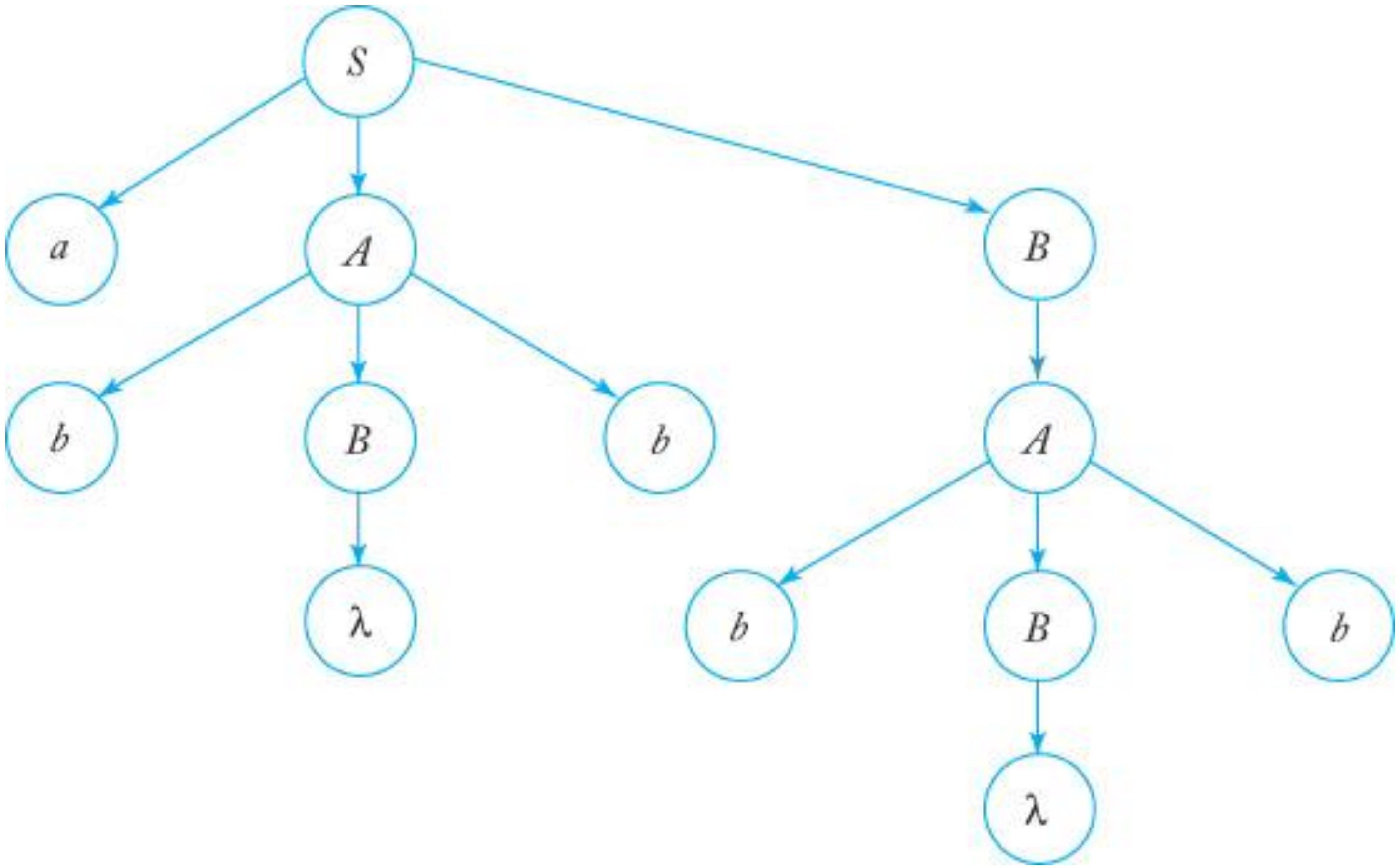
5.2 Derivation Trees/Parse Trees

- A second way of showing derivations, independent of the order in which productions are used
- Definition 5.3 (page 130, a formal definition of **derivation tree** and **partial derivation tree**)

Example 5.6: Partial derivation tree $S \rightarrow aAB$, $A \rightarrow bBb$, $B \rightarrow A|\lambda$



Example 5.6: a complete Derivation tree $S \rightarrow aAB,\ A \rightarrow bBb,\ B \rightarrow A|\lambda$

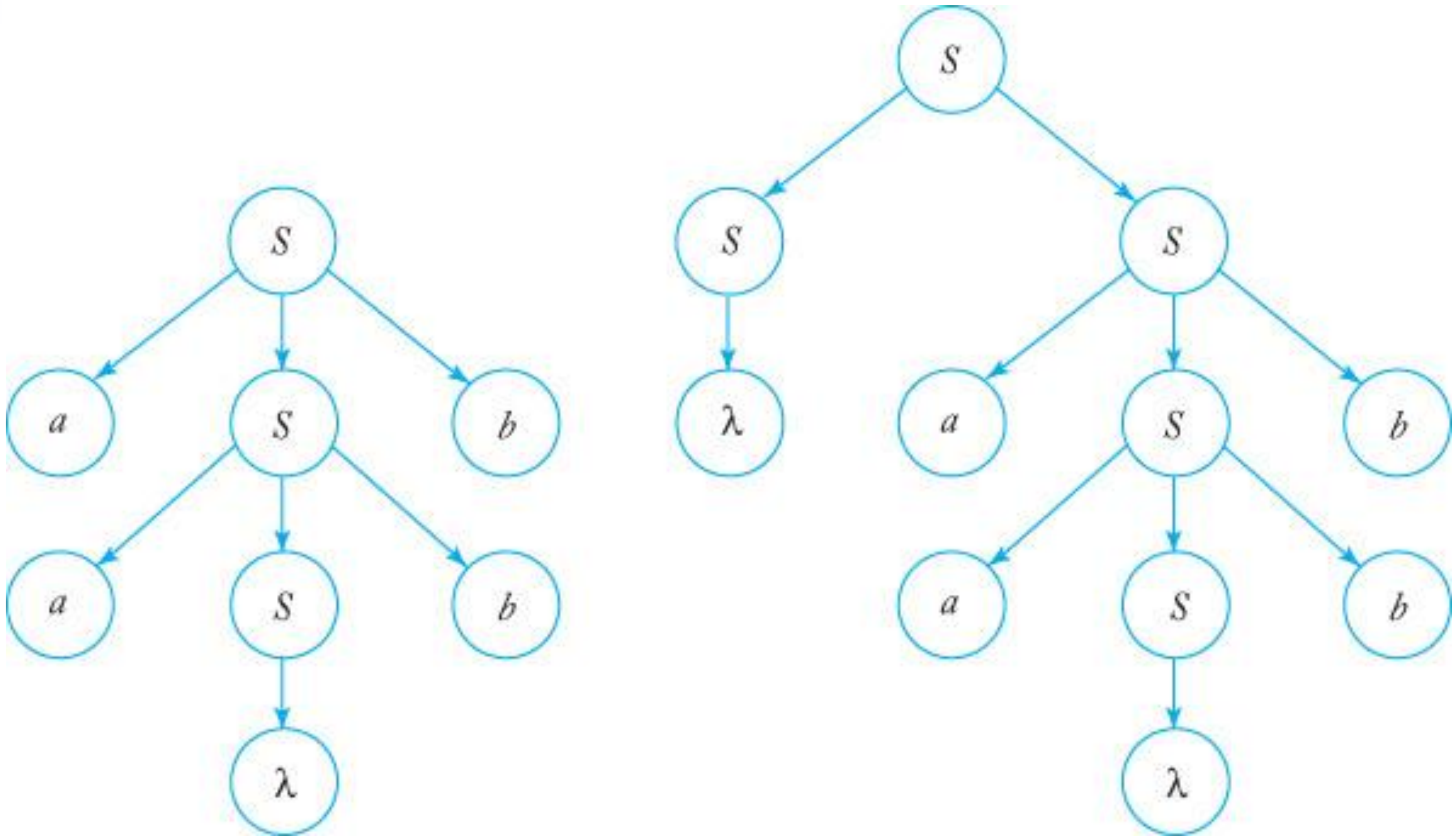


Parsing and Ambiguity

- **Parsing** – finding a sequence of productions by which a $w \in L(G)$ is derived
- Definition 5.5
  - A CFG G is said to be **ambiguous** if there exists some $w \in L(G)$ that has at least two distinct derivation trees. Alternatively ambiguity implies the existence of two or more leftmost or rightmost derivations

Example 5.10

Grammar $S \to aSb|SS|\lambda$, is ambiguous since aabb has the two derivation trees shown



Example 5.11 -- 1

G= (V, T, E, P) with

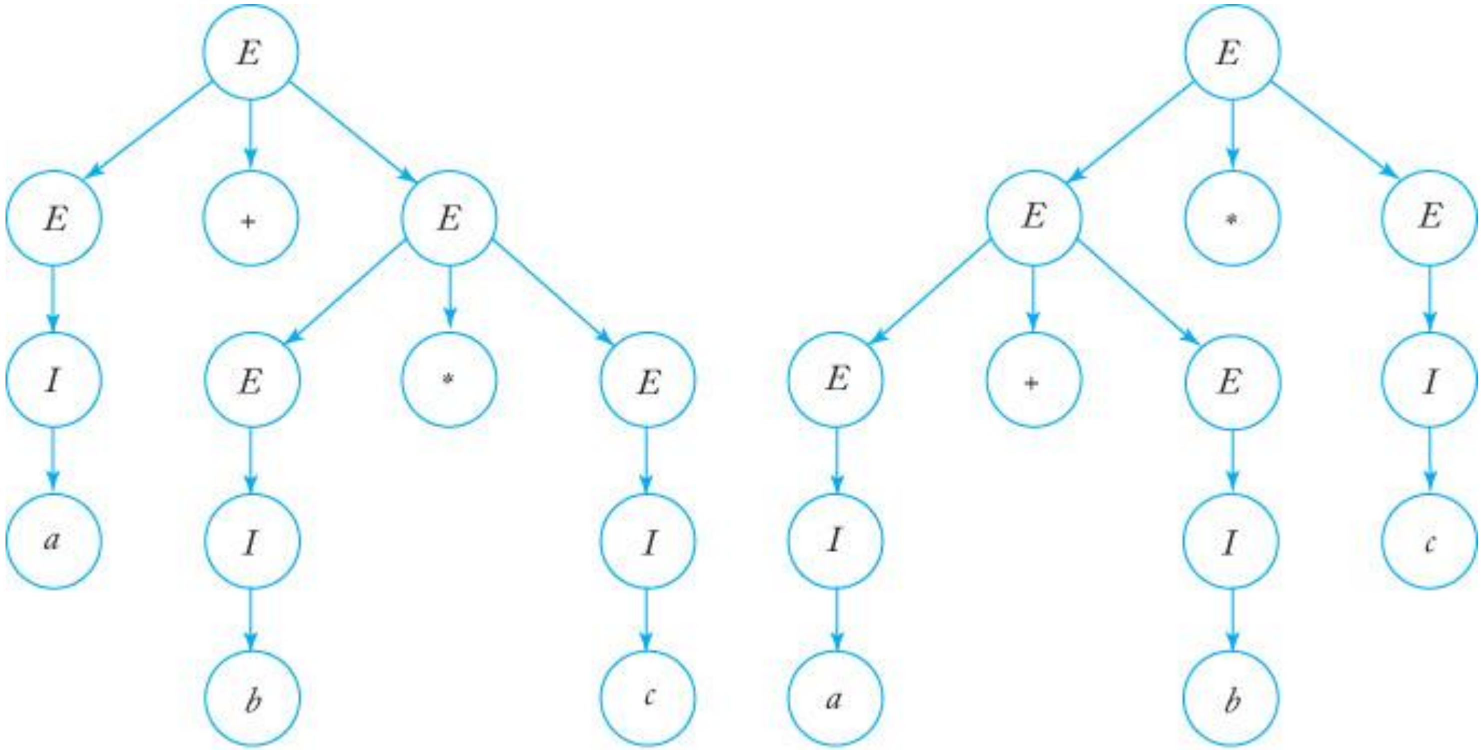
V= {E, I}, T = {a, b, c, +, *, (,)}

- and productions
  - $E \rightarrow I \mid E+E \mid E*E \mid (E)$
  - $I \rightarrow a \mid b \mid c$
- The grammar is ambiguous since a+b*c has two different derivation trees
  - Try to draw the two different derivation trees

Example 5.11 – 2

$E \to I \mid E+E \mid E*E \mid (E)$, $I \to a \mid b \mid c$

a+b*c has two different derivation trees



(a) (b)

Example 5.12 -- 1 Rewrite the grammar

- One way to resolve the ambiguity is to associate precedence rules with the operators + and *
- * has higher precedence than +
  - + is one level closer to root of the derivation tree, E
- We introduce new variables
  - V = {E, T, F, I}
  - Replace productions in such a way that + is closer to E than * in the derivation tree

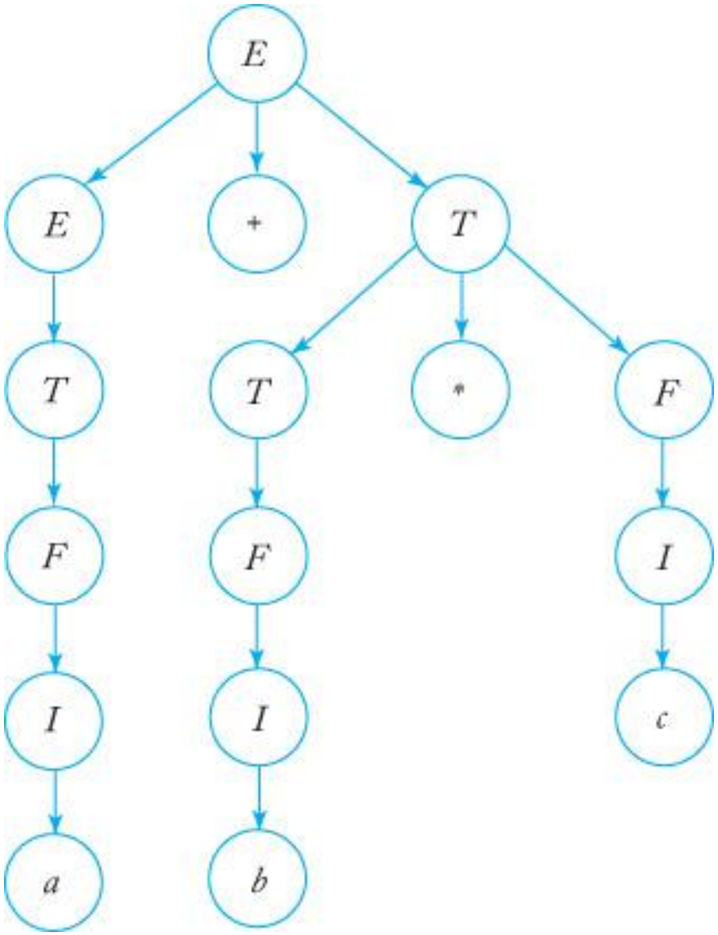
Example 5.12 – 2 Rewrite the productions

- ambiguous
- $E \rightarrow I \mid E+E \mid E*E \mid (E)$
- $I \rightarrow a \mid b \mid c$

Rewrite to remove ambiguity

- $E \rightarrow E + T \mid T$
- $T \rightarrow T * F \mid F$
- $F \rightarrow (E) \mid I$
- $I \rightarrow a|b|c$

Example 5.12 – 3:
a unique derivation tree of a+b*c (using new grammar)



A Practical Problem
Assume the following rules of associativity and precedence for expressions

- Precedence:
  - Highest *, /, **not**
  - +, -, &, **mod**
  - - (unary)
  - =, ≠, <, ≤, ≥, >
  - **and**
  - Lowest **or**, **xor**
- Associativity: left to right
- Write a CFG for the expression. Assume the only operands are the names a, b, c, d, and e.

Recursion to specify associativity Precedence cascade to specify precedence

<expr> ::= <expr> or <e1> | <expr> xor <e1> | <e1>

<e1> ::= <e1> and <e2> | <e2>

<e2> ::= <e2> = <e3> | <e2> /= <e3> | <e2> < <e3>
| <e2> <= <e3> | <e2> > <e3> | <e2> >= <e3> | <e3>

<e3> ::= <e4> | -<e4>

<e4> ::= <e4> + <e5> | <e4> - <e5> | <e4> & <e5> | <e4> mod <e5> | <e5>

<e5> ::= <e5> * <e6> | <e5> / <e6> | not <e5> | <e6>

<e6> ::= a | b | c | d | e | const | (<expr>)

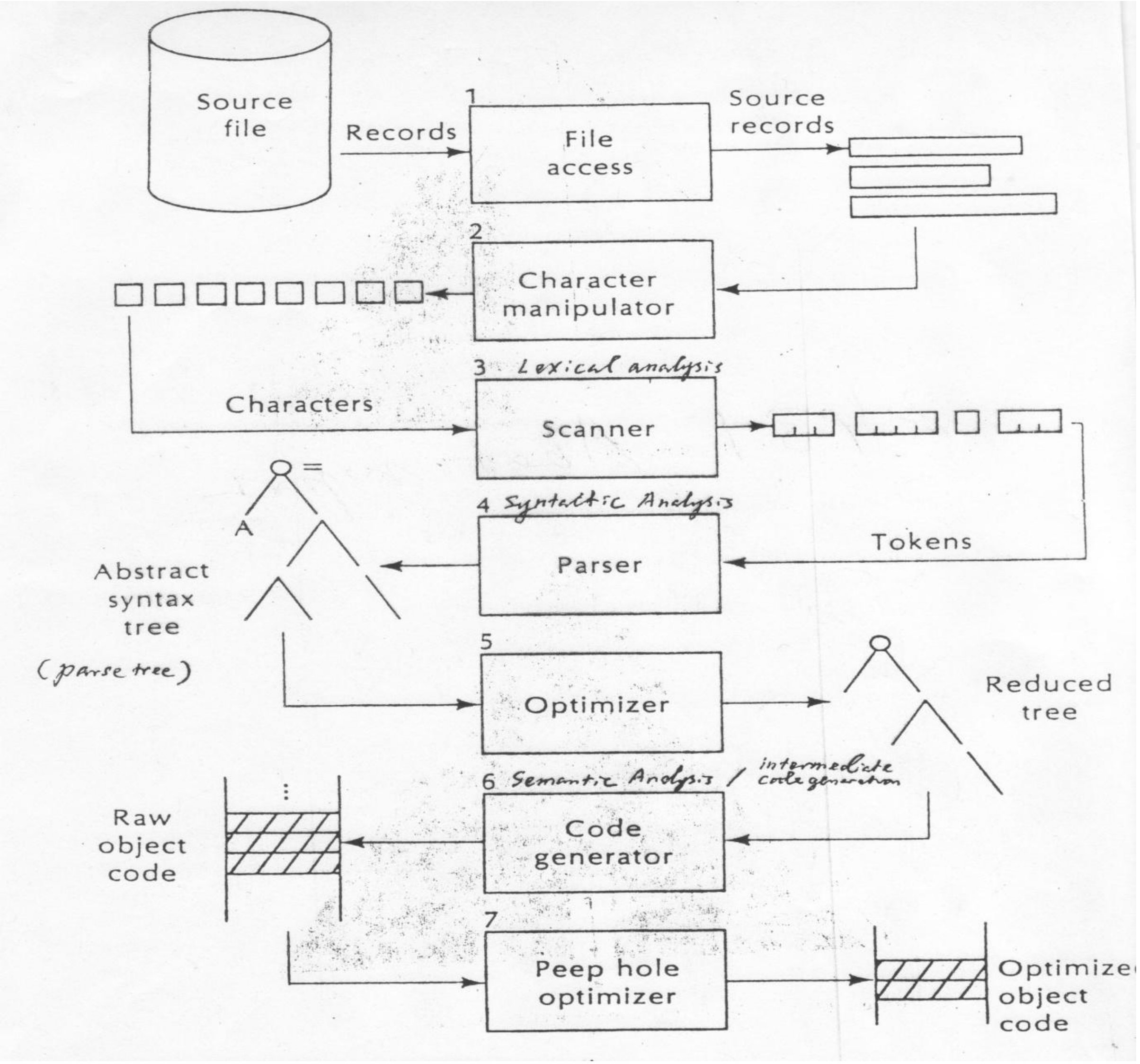
CFG and Programming Languages

- One of the most important uses of the computing theory is in
  - the definition of programming language
  - the construction of interpreters/compilers
  - Both RL and CFL are used in model all aspects
  - We can define a programming language by a grammar

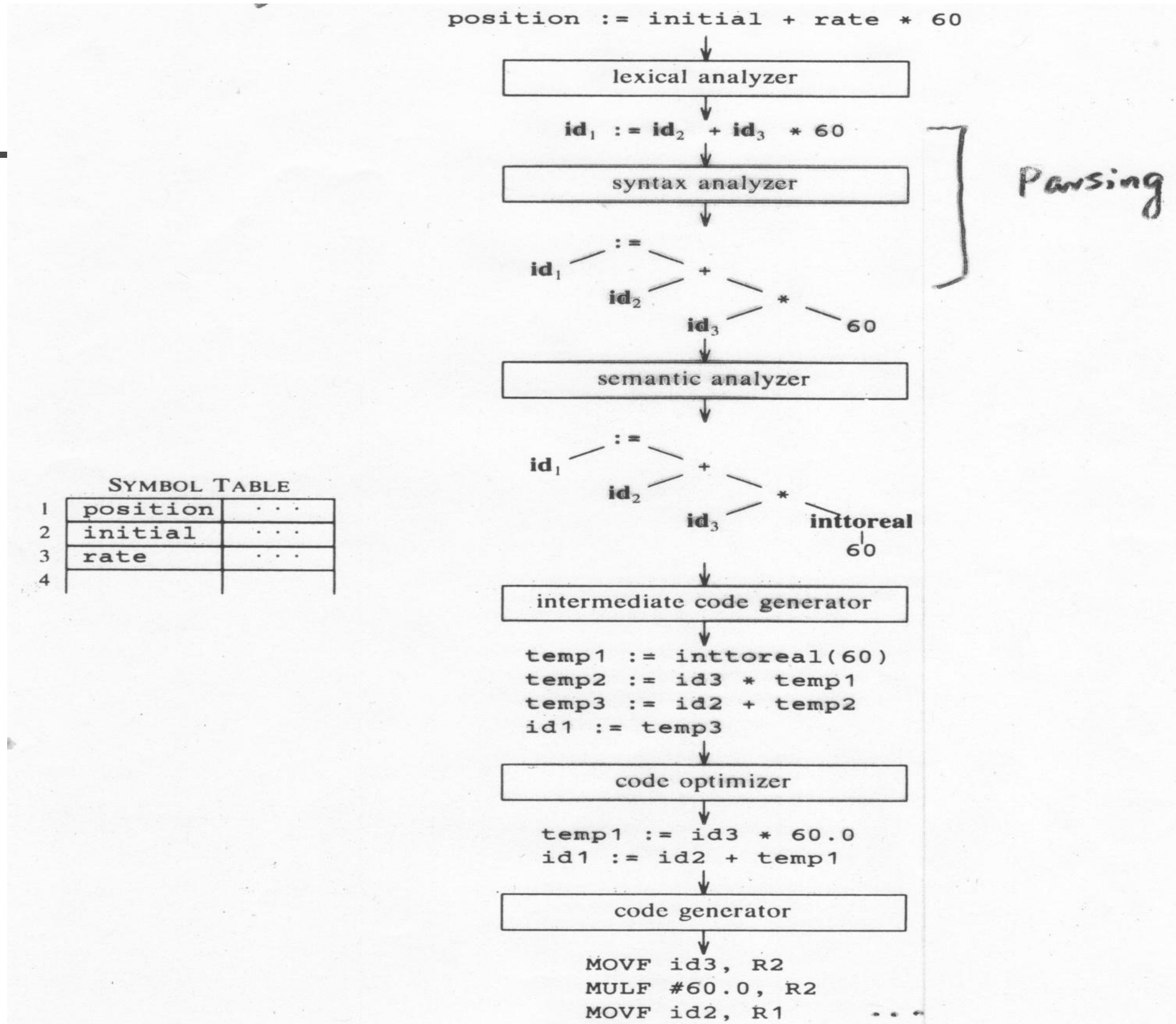
BNF and CFG

- A traditional notation in writing on programming languages is called the Backus-Naur form or BNF
- BNF is in essence the same as notation we use for CFG here
- <expression>::= <term>|<expression>+<term>
- <term>::= <factor>|<term>*<factor>
- <while_statement>::=while<expression><statement>

Major Operations in a Compiler



The Phases of a Compiler



In-class exercise: for given grammar, show BNF, EBNF, Syntax Diagram, first and follow sets

- Write G_3 in EBNF
- Draw syntax diagrams
- What are the 2 requirements for a predictive parser to be able to distinguish between choices in the grammar rule?
- Give the first sets/follow sets of G_3.
- Do we need to compute follow set for G_3?

G_3 in BNF → EBNF -- 1

- To remove recursion and ε
- V → SR$
- S → + | - | ε
- R → .dN | dN.N
- N → dN | ε

BNF → EBNF -- 2

- To remove recursion and ε
- V → SR$ V ::= SR$
- S → + | - | ε S ::= [+ | -]
- R → .dN | dN.N   R ::= .dN | dN.N
- N → dN | ε N ::= {d}

- Draw syntax diagrams for each variable
- Can we simplify EBNF more?

What are the 2 requirements for a predictive parser to be able to distinguish between choices in the grammar rule? (Hint: use first and follow sets)

1. Given A → B | C
2. Given A → D[E]F

Two Requirements

(1) Given $A \rightarrow B \mid C$

First (B) $\cap$ first (C) = $\varnothing$

(2) Given $A \rightarrow D[E]F$

First (E) $\cap$ follow (E) = $\varnothing$

EBNF to Syntax Diagram first sets and follow sets -1

- V ::= SR$
- S ::= [+ | -]
- R ::= .dN | dN.N
- N ::= {d}

first sets and follow sets - 2

- V ::= SR$ — first (V)=first (S) ∪ first (R) = {+, -, ., d}
- S ::= [+ | -] — first (S) = {+, -}
- R ::= .dN | dN.N — first (R) ={. , d}
- N ::= {d} — first (N) ={d}

- Follow (V) = {}, Follow (R) ={$}
- Follow (S) = first (R) = {., d}
- Follow (N) = {.} ∪ follow (R) = {., $}

EBNF
using substitution to simplify

- V ::= SR$
- S ::= [+ | -]
- R ::= .dN | dN.N
- N ::= {d}

- V ::= [+|-] R $
- R ::= .d{d} |d{d}.{d}

A Top-down Parser - 1

$G_3 = (\{V, S, R, N\}, \{+, -, ., d, \perp\}, P, V)$, where P is:

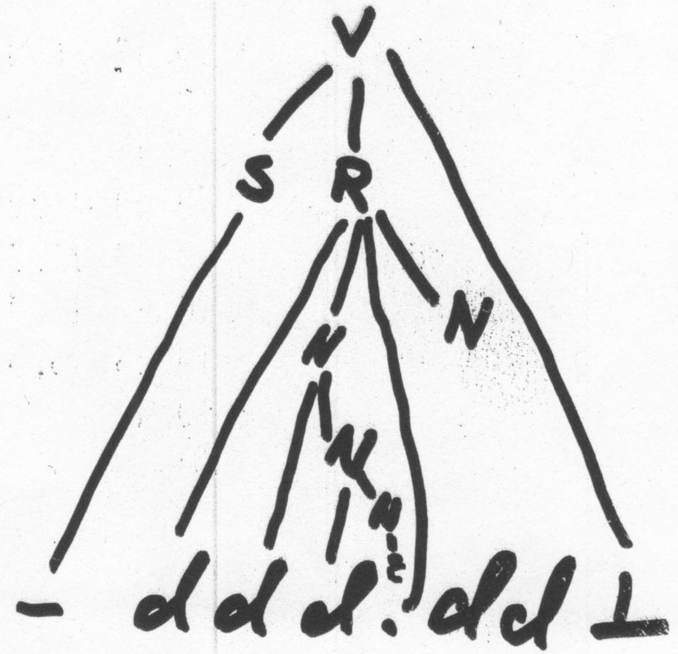
```
1.    V → SR⊥        {⊥ is a stop symbol}
2.    S → +
3.    S → -
4.    S → ε          {ε is the empty string}
5.    R → .dN        {d is a decimal digit}
6.    R → dN.N
7.    N → dN
8.    N → ε
```

		Next token				
		+	−	.	d	⊥
Left-most Exposed Nonterminal	V	1	1	1	1	×
	S	2	3	4	4	×
	R	×	×	5	6	×
	N	×	×	8	7	8

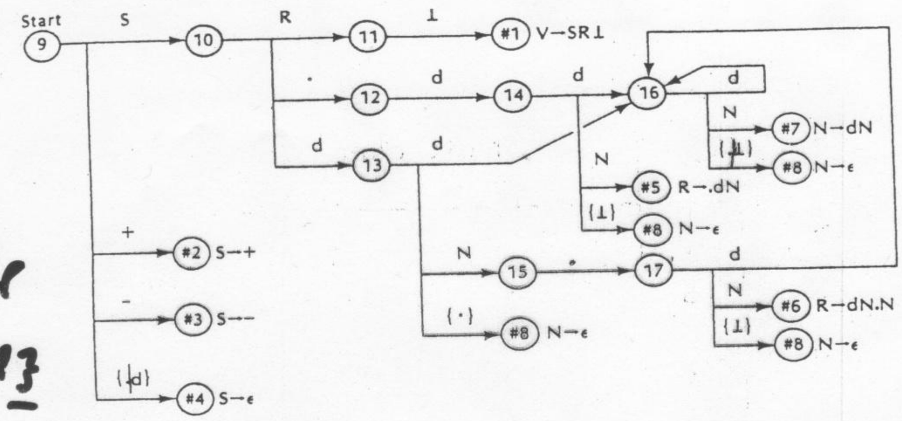
Figure 2.16. A top-down LL(1) parsing table for grammar G_3.

A Top-down Parser - 2

Frontier	Remaining Input	Production
V	$-$ddd.dd$\perp$	1
SR$\perp$	$-$ddd.dd$\perp$	3
$-$R$\perp$	$-$ddd.dd$\perp$	(match, drop $-$)
R$\perp$	ddd.dd$\perp$	6
dN.N$\perp$	ddd.dd$\perp$	(match)
N.N$\perp$	dd.dd$\perp$	7
dN.N$\perp$	dd.dd$\perp$	(match)
N.N$\perp$	d.dd$\perp$	7
dN.N$\perp$	d.dd$\perp$	(match)
N.N$\perp$	.dd$\perp$	8
.N$\perp$	.dd$\perp$	(match)
N$\perp$	dd$\perp$	7
dN$\perp$	dd$\perp$	(match)
N$\perp$	d$\perp$	7
dN$\perp$	d$\perp$	(match)
N$\perp$	$\perp$	8
$\perp$	$\perp$	(match and halt)



A Bottom-up Parser



Look ahead
{.|d}

{.|⊥}

Figure 2.17. A bottom-up parsing machine for grammar G_3. LR(1)

Rootline	State Path	Production
ddd.dd⊥	9	4, S → ε
Sddd.dd⊥	9, 10, 13, 16, 16	8, N → ε
SdddN.dd⊥	9, 10, 13, 16, 16	7, N → dN
SddN.dd⊥	9, 10, 13, 16	7, N → dN
SdN.dd⊥	9, 10, 13, 15, 17, 16, 16	8, N → ε
SdN.ddN⊥	9, 10, 13, 15, 17, 16, 16	7, N → dN
SdN.dN⊥	9, 10, 13, 15, 17, 16	7, N → dN
SdN.N⊥	9, 10, 13, 15, 17	6, R → dN.N
SR⊥	9, 10, 11	1, V → SR⊥
V	(halt)	

It should be clear that we have reproduced a right-most derivation of the sentence, in reverse order.

V R N N S ε ddd ε . dd⊥

HW
−d.d⊥ , +d.dd⊥