Chapter 3 Describing Syntax and Semantics

3.1 Introduction

- Providing a concise yet understandable description of a programming language is difficult but essential to the language's success.
  - ALGOL 60 and ALGOL 68 were first.
- For programming language implementors
- Language reference manual

3.1 Introduction (Cont'd)

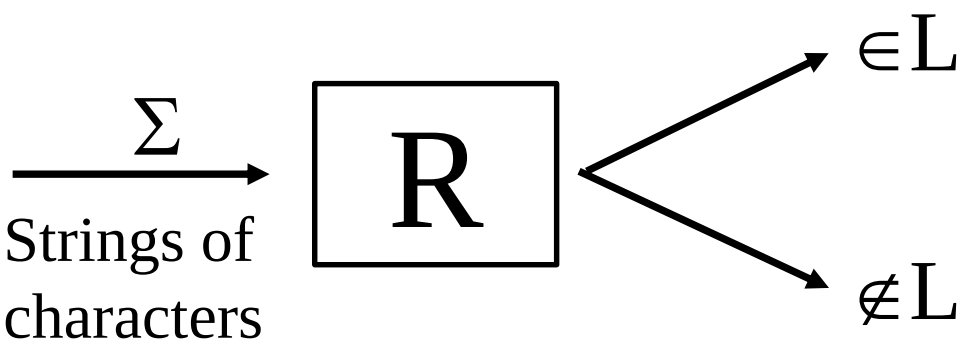
- Definition of Syntax and Semantics
  - Syntax:
    - Form, context-free
  - Semantics:
    - Meaning, context-sensitive

3.2 The General Problem of Describing Syntax

- Alphabet, Strings, Sentences, Language
- Lexemes and Tokens

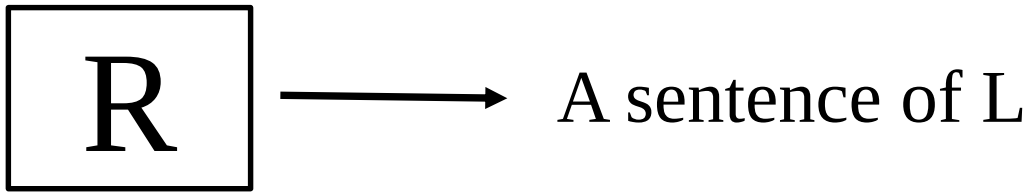
3.2.1 Language Recognizers

- R: Recognition device



3.2.1 Language Recognizers

- G: Language generator



3.3 Formal Methods of Describing Syntax

- Backus-Naur Form and Context-free Grammars
  - Appeared in late 1950s
  - Context-free Grammers
    - Chomsky, a noted linguist, advised it.
      - Context-free and regular grammars turned out to be useful for describing the syntax of programming languages
        - » Context-free grammar: Syntax
        - » Regular grammar: Token

3.3 Formal Methods of Describing Syntax

- Backus-Naur Form
  - Shortly after Chomsky's work
  - John Backus and Peter Naur
  - Describing the syntax of ALGOL 58
  - BNF

Backus-Naur Form and Context-free Grammars

- Fundamentals
  - A metalanguage is a language that is used to describe another language.
    - E.g., BNF

Context-free grammar (CFG)

- CFG consists of a set of production rules,

$$A \rightarrow B\ C\ D \ldots\ Z$$

LHS must be a single **nonterminal**

RHS consists 0 or more **terminals** or **nonterminals**

Context-free grammar (CFG)

- Two kinds of symbols
  - Nonterminals
    - Delimited by < and >
    - Represent syntactic structures
  - Terminals
    - Represent tokens
- E.g.

  <program> → **begin** <statement list> **end**

- Start or goal symbol
- λ : empty or null string

Context-free grammar (Cont'd)

- E.g.

<statement list> → <statement><statement tail>
<statement tail> → λ
<statement tail> → <statement><statement tail>

Context-free grammar (Cont'd)

- Extended BNF: some abbreviations

1. optional: [ ] 0 or 1

   <stmt> → if <exp> then <stmt>

   <stmt> → if <exp> then <stmt> else <stmt>

   can be written as

   <stmt> → if <exp> then <stmt> [else <stmt>]

2. repetition: { } 0 or more

   <stmt list> → <stmt> <tail>

   <tail> → λ

   <tail> → <stmt> <tail>

   can be written as

   <stmt list> → <stmt> { <stmt> }

Context-free grammar (Cont'd)

- Extended BNF: some abbreviations

  3. alternative: | or

     <stmt> → <assign>

     <stmt> → <if stmt>

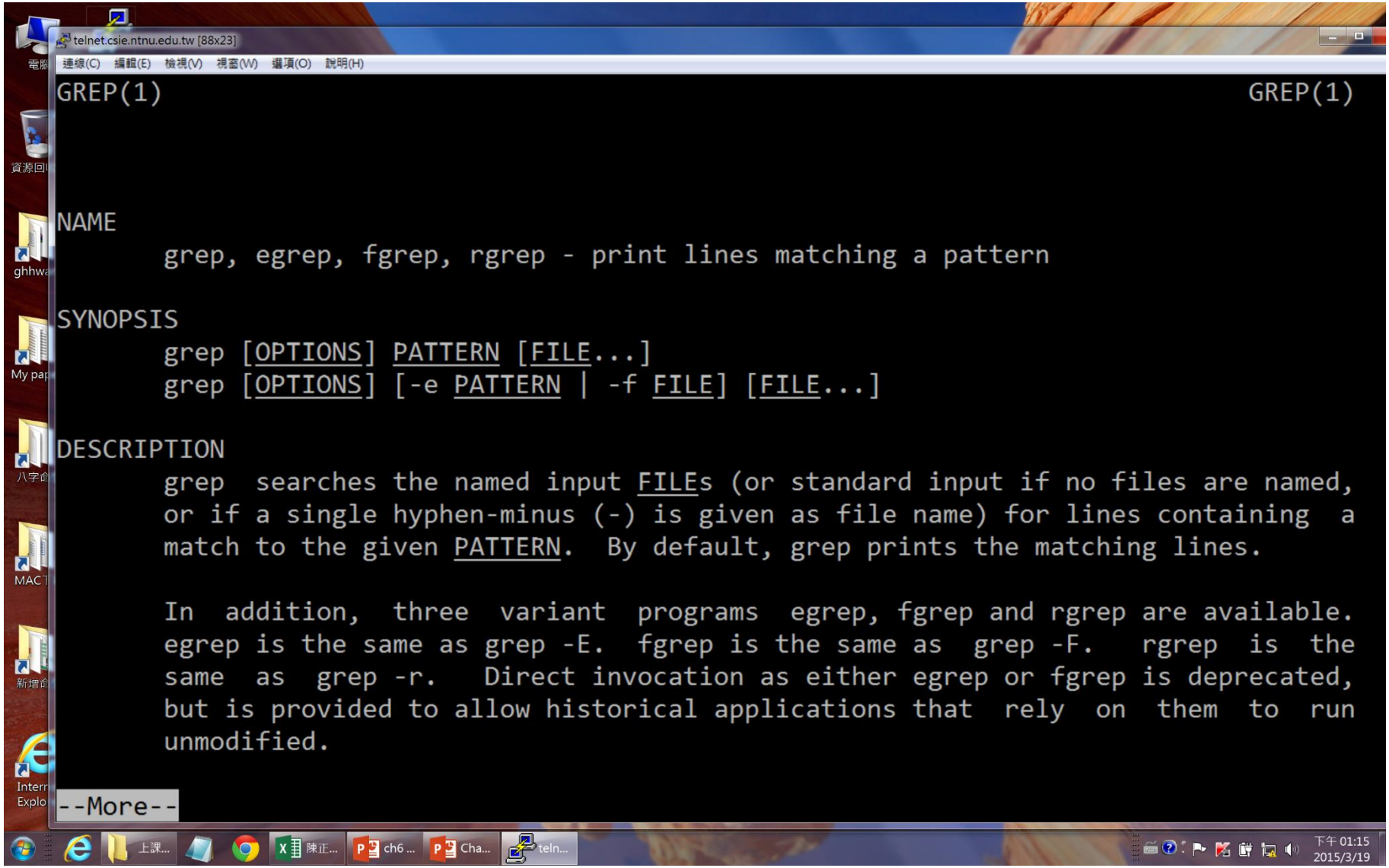
     can be written as

     <stmt> → <assign> | <if stmt>

- Extended BNF ≡ BNF

  – Either can be transformed to the other.

  – Extended BNF is more compact and readable


telnet.csie.ntnu.edu.tw [88x23]
連線(C) 編輯(E) 檢視(V) 視窗(W) 選項(O) 說明(H)
GREP(1) GREP(1)
NAME
grep, egrep, fgrep, rgrep - print lines matching a pattern
SYNOPSIS
grep [OPTIONS] PATTERN [FILE...]
grep [OPTIONS] [-e PATTERN | -f FILE] [FILE...]
DESCRIPTION
grep searches the named input FILEs (or standard input if no files are named, or if a single hyphen-minus (-) is given as file name) for lines containing a match to the given PATTERN. By default, grep prints the matching lines.
In addition, three variant programs egrep, fgrep and rgrep are available. egrep is the same as grep -E. fgrep is the same as grep -F. rgrep is the same as grep -r. Direct invocation as either egrep or fgrep is deprecated, but is provided to allow historical applications that rely on them to run unmodified.
--More--

The Syntax of Micro (Cont'd)

1. <program> → **begin** <statement list> **end**
2. <statement list> → <statement> {<statement>}
3. <statement> → ID := <expression> ;
4. <statement> → **read** (<id list>) ;
5. <statement> → **write** (<expr list>) ;
6. <id list> → ID {, ID}
7. <expr list> → <expression> {, <expression>}
8. <expression> → <primary> {<add op> <primary>}
9. <primary> → (<expression>)
10. <primary> → ID
11. <primary> → INTLITERAL
12. <add op> → PLUSOP
13. <add op> → MINUSOP
14. <system goal> → <program> SCANEOF

Figure 2.4 Extended CFG Defining Micro

- The **derivation** of

begin ID:= ID + (INTLITERAL – ID); end

<program>	
⇒ **begin** <statement list> **end**	(Apply rule 1)
⇒ **begin** <statement> {<statement>} **end**	(Apply rule 2)
⇒ **begin** <statement> **end**	(Choose 0 repetitions)
⇒ **begin** ID := <expression> ; **end**	(Apply rule 3)
⇒ **begin** ID := <primary> {<add op> <primary>} ; **end**	(Apply rule 8)
⇒ **begin** ID := <primary> <add op> <primary> ; **end**	(Choose 1 repetition)
⇒ **begin** ID := <primary> + <primary> ; **end**	(Apply rule 12)
⇒ **begin** ID := ID + <primary> ; **end**	(Apply rule 10)
⇒ **begin** ID := ID + (<expression>) ; **end**	(Apply rule 9)
⇒ **begin** ID := ID + (<primary> {<add op> <primary>}) ; **end**	(Apply rule 8)
⇒ **begin** ID := ID + (<primary> <add op> <primary>) ; **end**	(Choose 1 repetition)
⇒ **begin** ID := ID + (<primary> – <primary>) ; **end**	(Apply rule 13)
⇒ **begin** ID := ID + (INTLITERAL – <primary>) ; **end**	(Apply rule 11)
⇒ **begin** ID := ID + (INTLITERAL – ID) ; **end**	(Apply rule 10)

Backus-Naur Form and Context-free Grammars

- Describing Lists
  - Variable-length lists in mathematics are often written using an ellipsis (…)
  - For BNF, the alternative is recursion

```
<id_list> → id
          | id, <id_list>
```

Backus-Naur Form and Context-free Grammars

- Grammars and Derivations
  - start symbol
  - derivation
  - sentential form & leftmost derivations
    - See next slice

- A grammar

```
<program> → <stmts>
   <stmts> → <stmt> | <stmt> ; <stmts>
   <stmt> → <var> = <expr>
   <var> → a | b | c | d
   <expr> → <term> + <term> | <term> - <term>
   <term> → <var> | const
```

- A derivation

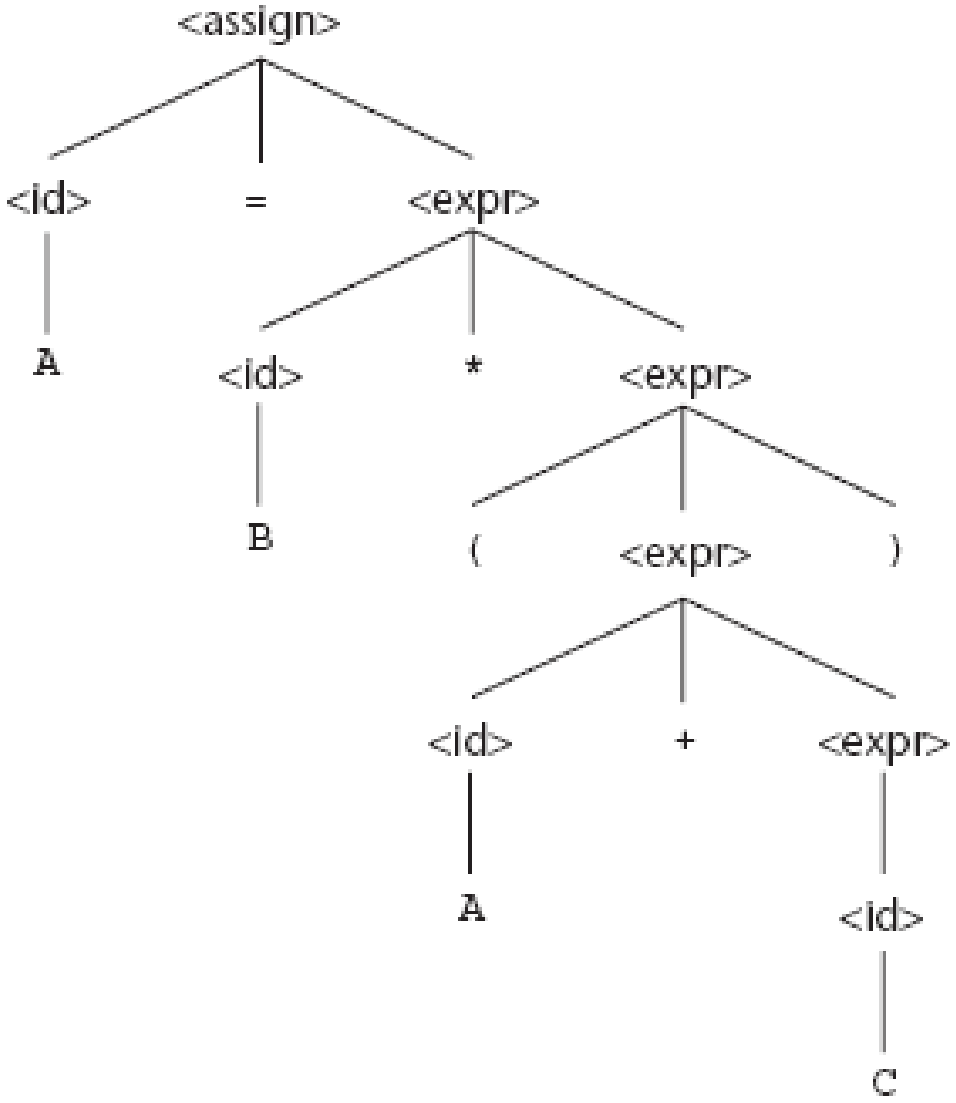
```
<program> => <stmts> => <stmt>
                     => <var> = <expr>
                     => a = <expr>
                     => a = <term> + <term>
                     => a = <var> + <term>
                     => a = b + <term>
                     => a = b + const
```

Backus-Naur Form and Context-free Grammars

- Parse trees
  - One of the most attractive features of grammars is that they naturally describe the hierarchical syntactic structure of the sentences of the languages they define.
    - Internal node: nonterminal symbol
    - Leaf node: terminal symbol

Figure 3.1

A parse tree for the simple statement
`A = B * (A + C)`

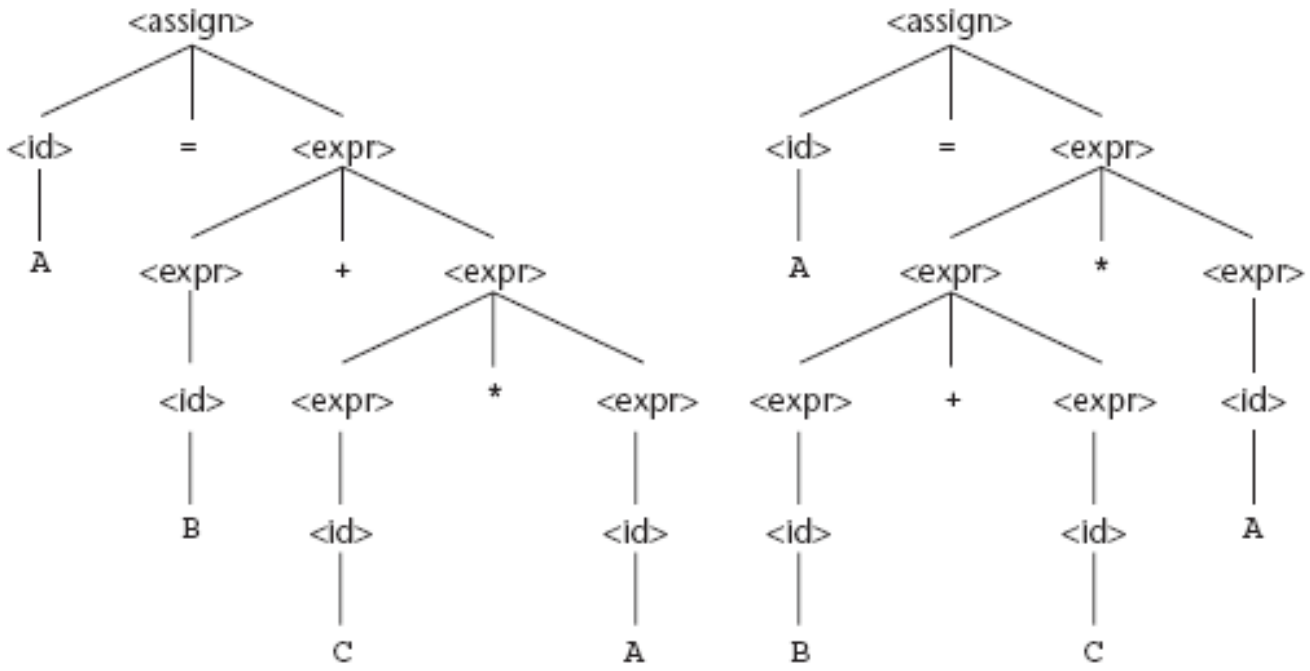


Backus-Naur Form and Context-free Grammars

- Ambiguity
  - A grammar that generates a sentential form for which there are two or more distinct parse tree is said to be ambiguous
    - See next slice
  - Why may ambiguity cause problem?
    - Code generation for compilers

Figure 3.2

Two distinct parse trees for the same sentence, `A = B + C * A`

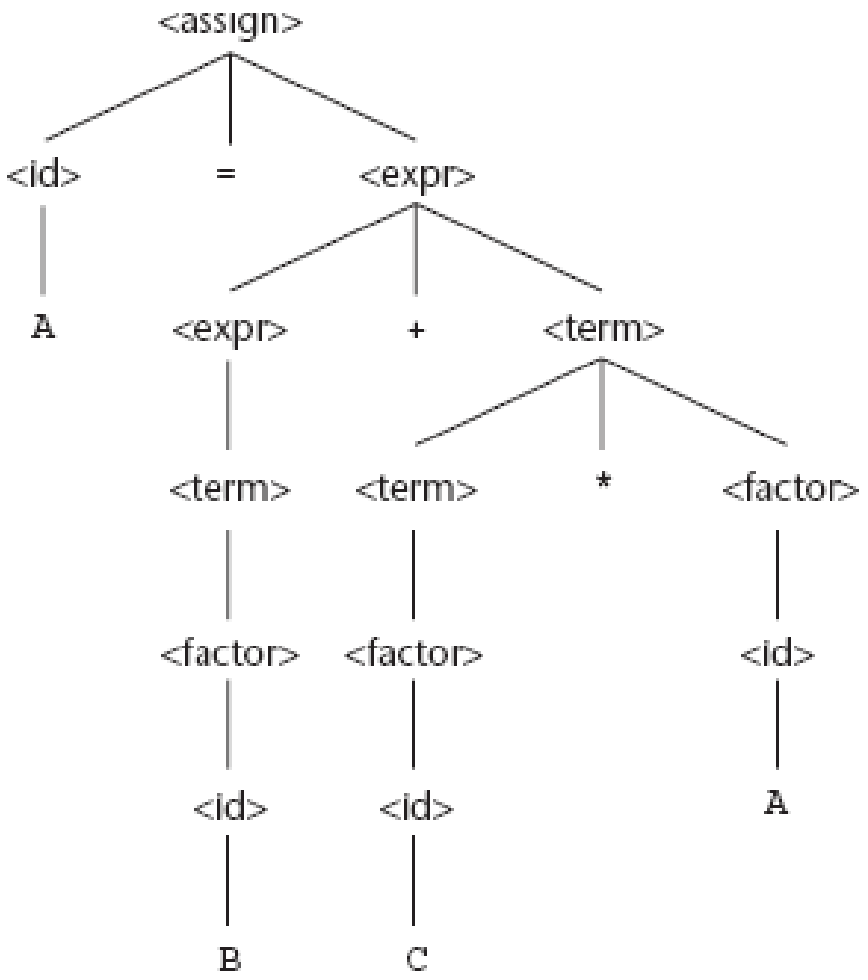


Backus-Naur Form and Context-free Grammars

- Operator precedence
  - The order of evaluation of operators
  - Can the BNF demonstrate the operator precedence?
    - See EXAMPLE 3.4 and the derivation of A=B+C ∗A

Figure 3.3

The unique parse tree for `A = B + C * A` using an unambiguous grammar

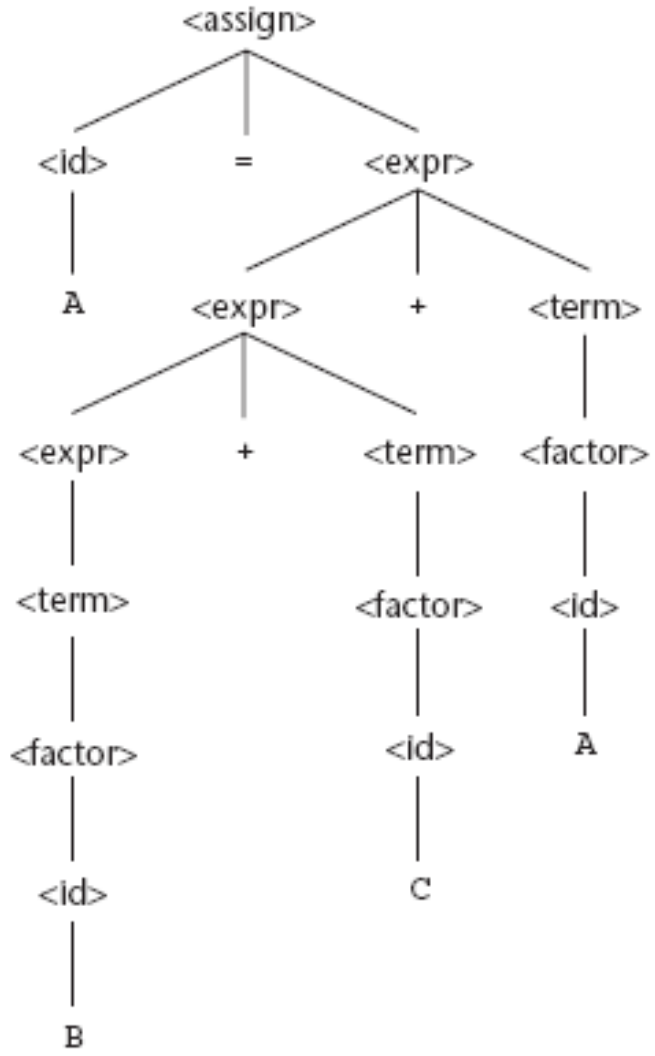


Backus-Naur Form and Context-free Grammars

- Associativity of operators
  - When an expression includes two operators that have the same precedence, a semantic rule is required to specify which should have precedence
  - The left recursion specifies left associativity

Figure 3.4

A parse tree for `A = B + C + A` illustrating the associativity of addition



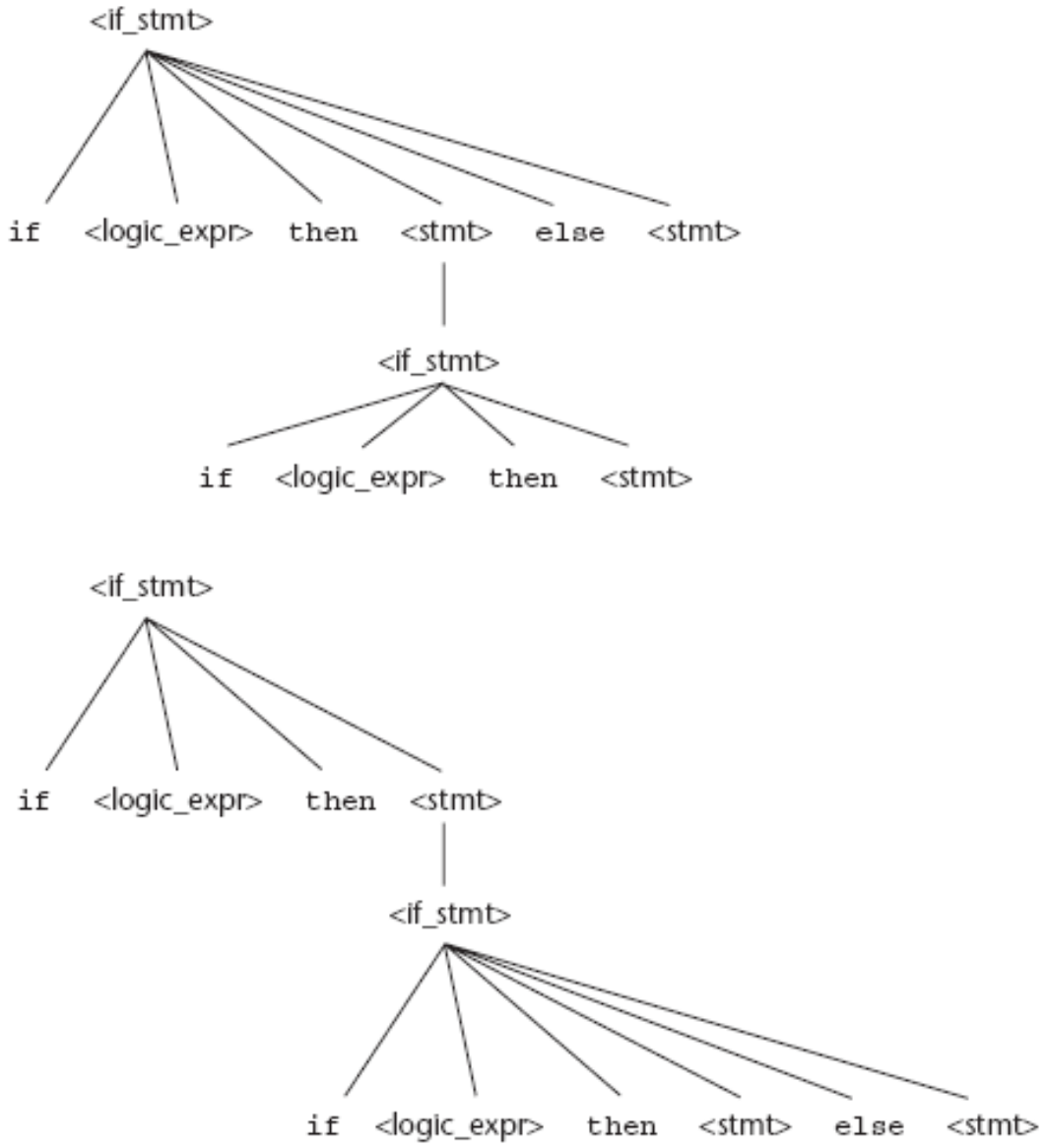
An unambiguous grammar for if-then-else

- An intuitive BNF for if-then-else

```
<if_stmt> → if <logic_expr> then <stmt>
          | if <logic_expr> then <stmt> else <stmt>
```

Figure 3.5

Two distinct parse trees for the same sentential form



An unambiguous grammar for if-then-else

- The unambiguous grammar

```
<stmt> → <matched> | <unmatched>
<matched>→ if <logic_expr> then <matched> else <matched>
           | <any_non-if_statement>
<unmatched>→ if <logic_expr> then <stmt>
             | if <logic_expr> then <matched> else <unmatched>
```