Lecture 2
Lexical and Syntax Analysis

Maher Sarem

ME CSE(IT)

Topics

- Introduction
- Lexical Analysis
- The Parsing Problem
- Recursive-Descent Parsing
- Bottom-Up Parsing

Introduction

- Language implementation systems must analyze source code, regardless of the specific implementation approach
- Nearly all syntax analysis is based on a formal description of the syntax of the source language (BNF)

Syntax Analysis

- The syntax analysis portion of a language processor nearly always consists of two parts:
  - A low-level part called a *lexical analyzer* (mathematically, a finite automaton based on a regular grammar)
  - A high-level part called a *syntax analyzer*, or parser (mathematically, a push-down automaton based on a context-free grammar, or BNF)

Advantages of Using BNF to Describe Syntax

- Provides a clear and concise syntax description
- The parser can be based directly on the BNF
- Parsers based on BNF are easy to maintain

Reasons to Separate Lexical and Syntax Analysis

- *Simplicity* - less complex approaches can be used for lexical analysis; separating them simplifies the parser
- *Efficiency* - separation allows optimization of the lexical analyzer
- *Portability* - parts of the lexical analyzer may not be portable, but the parser always is portable

Lexical Analysis

- A lexical analyzer is a pattern matcher for character strings
- A lexical analyzer is a "front-end" for the parser
- Identifies substrings of the source program that belong together - *lexemes*
  - Lexemes match a character pattern, which is associated with a lexical category called a *token*
  - `sum` is a lexeme; its token may be `IDENT`

Lexical Analysis (continued)

- The lexical analyzer is usually a function that is called by the parser when it needs the next token
- Three approaches to building a lexical analyzer:
  - Write a formal description of the tokens and use a software tool that constructs a table-driven lexical analyzer from such a description
  - Design a state diagram that describes the tokens and write a program that implements the state diagram
  - Design a state diagram that describes the tokens and hand-construct a table-driven implementation of the state diagram

Lexical Analysis (continued)

- In many cases, transitions can be combined to simplify the state diagram
  - When recognizing an identifier, all uppercase and lowercase letters are equivalent
    - Use a character class that includes all letters
  - When recognizing an integer literal, all digits are equivalent - use a digit class

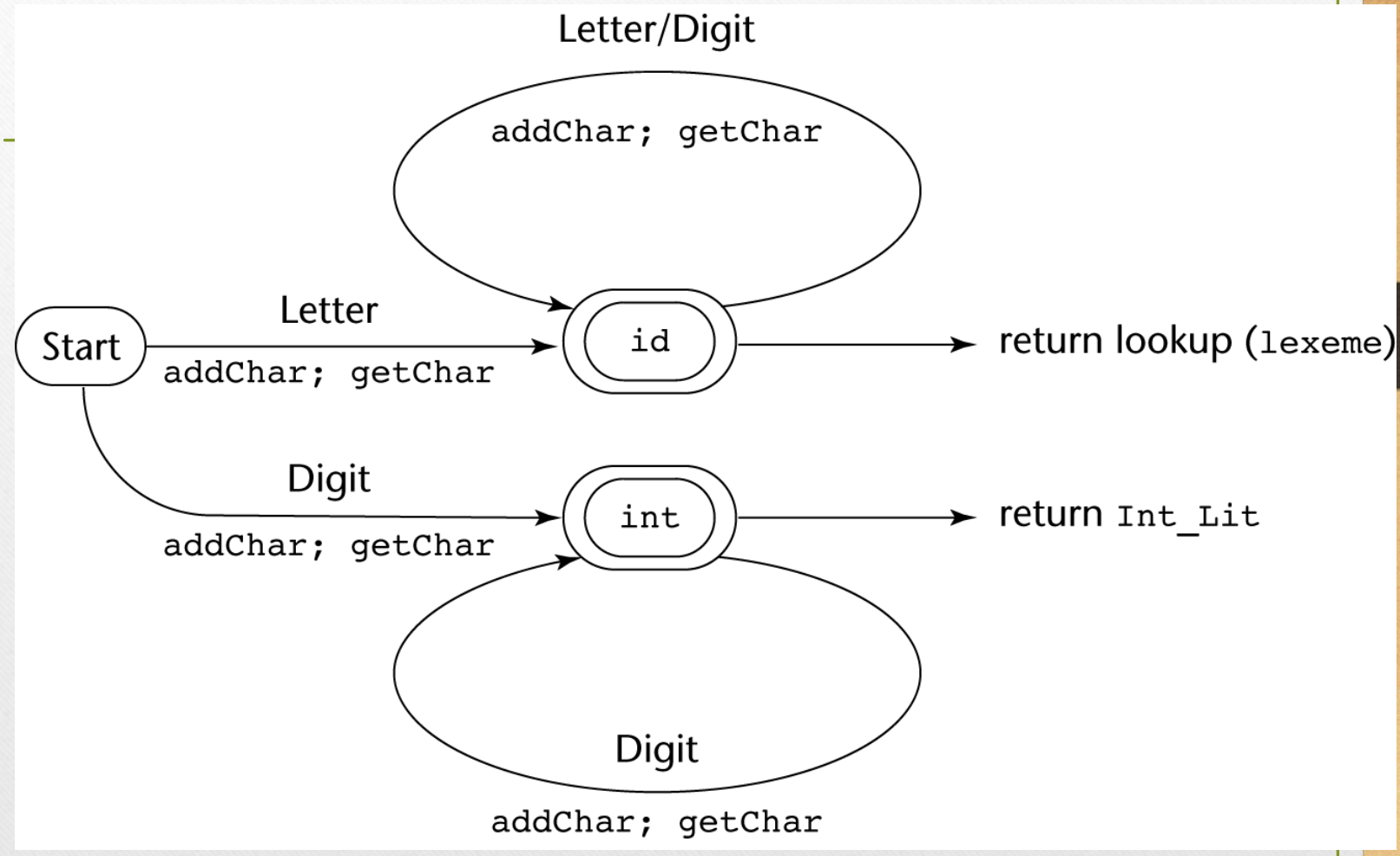
Lexical Analysis (continued)

- Reserved words and identifiers can be recognized together (rather than having a part of the diagram for each reserved word)
  - Use a table lookup to determine whether a possible identifier is in fact a reserved word

Lexical Analysis (continued)

- Convenient utility subprograms:
  - **`getChar`** - gets the next character of input, puts it in **`nextChar`**, determines its class and puts the class in **`charClass`**
  - **`addChar`** - puts the character from **`nextChar`** into the place the lexeme is being accumulated, **`lexeme`**
  - **`lookup`** - determines whether the string in **`lexeme`** is a reserved word (returns a code)

State Diagram



The Parsing Problem

- Goals of the parser, given an input program:
  - Find all syntax errors; for each, produce an appropriate diagnostic message and recover quickly
  - Produce the parse tree, or at least a trace of the parse tree, for the program

The Parsing Problem (continued)

- Two categories of parsers
  - *Top down* - produce the parse tree, beginning at the root
    - Order is that of a leftmost derivation
    - Traces or builds the parse tree in preorder
  - *Bottom up* - produce the parse tree, beginning at the leaves
    - Order is that of the reverse of a rightmost derivation
- Useful parsers look only one token ahead in the input

The Parsing Problem (continued)

- Top-down Parsers
  - Given a sentential form, xAα , the parser must choose the correct A-rule to get the next sentential form in the leftmost derivation, using only the first token produced by A
- The most common top-down parsing algorithms:
  - Recursive descent - a coded implementation
  - LL parsers - table driven implementation

The Parsing Problem (continued)

- Bottom-up parsers
  - Given a right sentential form, α, determine what substring of α is the right-hand side of the rule in the grammar that must be reduced to produce the previous sentential form in the right derivation
  - The most common bottom-up parsing algorithms are in the LR family

The Parsing Problem (continued)

- The Complexity of Parsing
  - Parsers that work for any unambiguous grammar are complex and inefficient ($O(n^3)$, where n is the length of the input)
  - Compilers use parsers that only work for a subset of all unambiguous grammars, but do it in linear time (O(n), where n is the length of the input)

Recursive-Descent Parsing

- There is a subprogram for each nonterminal in the grammar, which can parse sentences that can be generated by that nonterminal
- EBNF is ideally suited for being the basis for a recursive-descent parser, because EBNF minimizes the number of nonterminals

Recursive-Descent Parsing
(continued)

- A grammar for simple expressions:

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → id | int_constant | ( <expr> )
```

Recursive-Descent Parsing
(continued)

- Assume we have a lexical analyzer named `lex`, which puts the next token code in `nextToken`
- The coding process when there is only one RHS:
  - For each terminal symbol in the RHS, compare it with the next input token; if they match, continue, else there is an error
  - For each nonterminal symbol in the RHS, call its associated parsing subprogram

Recursive-Descent Parsing
(continued)

- This particular routine does not detect errors
- Convention: Every parsing routine leaves the next token in **`nextToken`**

Recursive-Descent Parsing
(continued)

- A nonterminal that has more than one RHS requires an initial process to determine which RHS it is to parse
  - The correct RHS is chosen on the basis of the next token of input (the lookahead)
  - The next token is compared with the first token that can be generated by each RHS until a match is found
  - If no match is found, it is a syntax error

Bottom-up Parsing

- The parsing problem is finding the correct RHS in a right-sentential form to reduce to get the previous right-sentential form in the derivation

Bottom-up Parsing (continued)

- Shift-Reduce Algorithms
  - Reduce is the action of replacing the handle on the top of the parse stack with its corresponding LHS
  - Shift is the action of moving the next token to the top of the parse stack

Bottom-up Parsing (continued)

- Advantages of LR parsers:
  - They will work for nearly all grammars that describe programming languages.
  - They work on a larger class of grammars than other bottom-up algorithms, but are as efficient as any other bottom-up parser.
  - They can detect syntax errors as soon as it is possible.
  - The LR class of grammars is a superset of the class parsable by LL parsers.

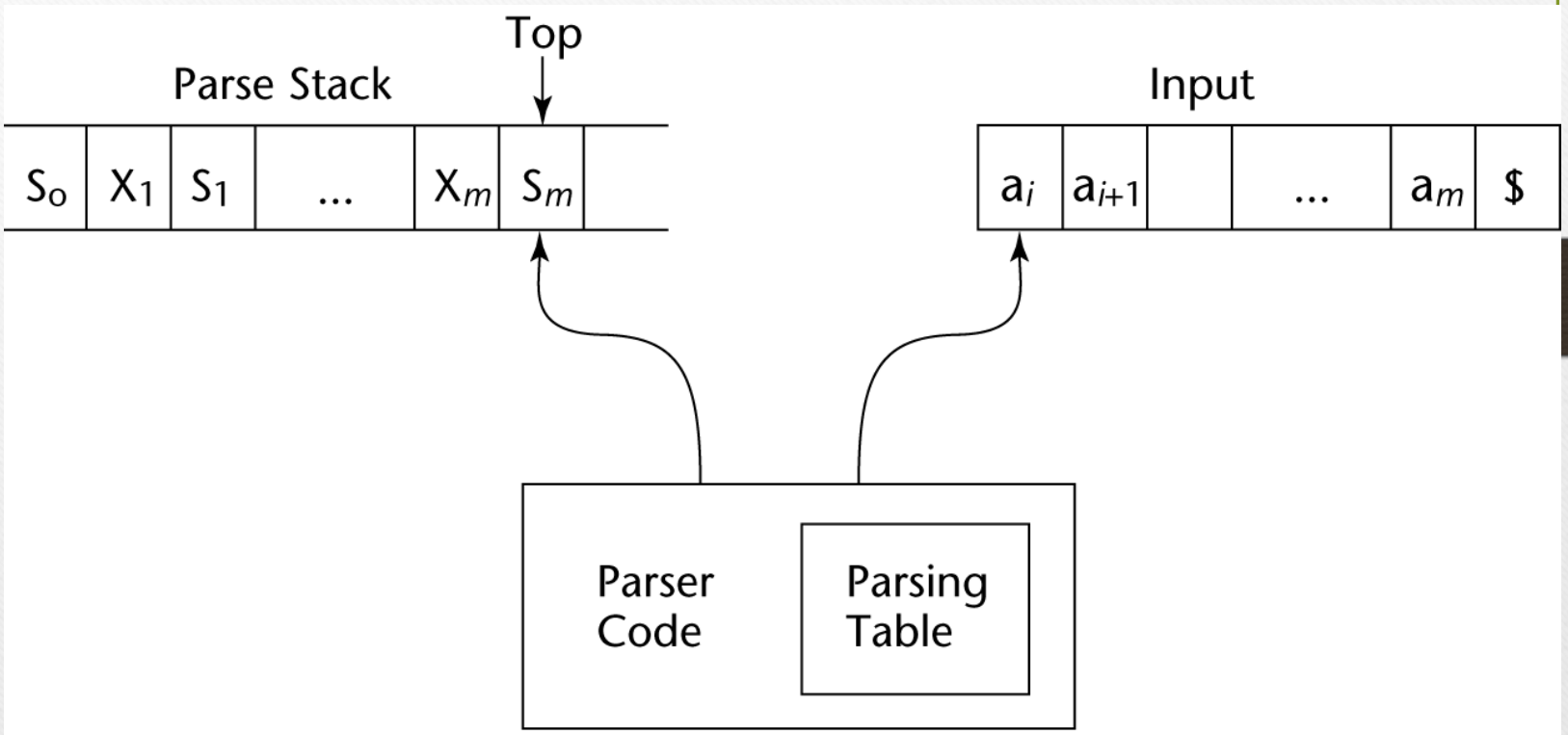
Bottom-up Parsing (continued)

- LR parsers must be constructed with a tool
- Knuth's insight: A bottom-up parser could use the entire history of the parse, up to the current point, to make parsing decisions
  - There are only a finite and relatively small number of different parse situations that could have occurred, so the history could be stored in a parser state, on the parse stack

Bottom-up Parsing (continued)

- LR parsers are table driven, where the table has two components, an ACTION table and a GOTO table
  - The ACTION table specifies the action of the parser, given the parser state and the next token
    - Rows are state names; columns are terminals
  - The GOTO table specifies which state to put on top of the parse stack after a reduction action is done
    - Rows are state names; columns are nonterminals

Structure of An LR Parser



Bottom-up Parsing (continued)

- Initial configuration: (S_0, $a_1 \ldots a_n\$$)
- Parser actions:
  - For a Shift, the next symbol of input is pushed onto the stack, along with the state symbol that is part of the Shift specification in the Action table
  - For a Reduce, remove the handle from the stack, along with its state symbols. Push the LHS of the rule. Push the state symbol from the GOTO table, using the state symbol just below the new LHS in the stack and the LHS of the new rule as the row and column into the GOTO table

Bottom-up Parsing (continued)

- Parser actions (continued):
  - For an Accept, the parse is complete and no errors were found.
  - For an Error, the parser calls an error-handling routine.

LR Parsing Table

	Action						Goto		
State	id	+	*	(	)	$	E	T	F
0	S5		S4				1	2	3
1		S6				accept			
2		R2	S7		R2	R2			
3		R4	R4		R4	R4			
4	S5			S4			8	2	3
5		R6	R6		R6	R6			
6	S5			S4				9	3
7	S5			S4					10
8		S6			S11				
9		R1	S7		R1	R1			
10		R3	R3		R3	R3			
11		R5	R5		R5	R5			

Bottom-up Parsing (continued)

- A parser table can be generated from a given grammar with a tool, e.g., **`yacc`** or **`bison`**

Describing Syntax and Semantics

Maher Sarem

ME CSE(IT)

Topics

- Introduction
- The General Problem of Describing Syntax
- Formal Methods of Describing Syntax

Introduction

- "In a well-designed programming language, semantics should follow directly from syntax; that is, the appearance of a statement should strongly suggest what the statement is meant to accomplish."
  - Robert W. Sebesta
  - A = A + 1;
  - A + 1 ➔ A

Describing Syntax: Terminology

- Language
  - Sentence
    - Lexeme – Token

- Example: a = 2 * c + 3;

Lexemes	Tokens
a	identifier
=	equal_sign
2	int_literal
*	mult_op
c	identifier
+	plus_op
3	int_literal
;	semicolon

- **Recognizer**
- **Generator**

Backus-Naur Form (1959)

- BNF is a meta-language for programming languages;
  - A description of the syntax rules of the languages

BNF Rule Examples

- `<ident_list> → identifier | identifier, <ident_list>`
- `<if_stmt> → if <logic_expr> then <stmt>`
- <assign> -> <var> = <expression>
- `<stmt> → <single_stmt>`
  `| begin <stmt_list> end`

- *Note: Backus' grammar similar to Chomsky's Context-free and regulart grammars*

EXAMPLE 3.1 A Grammar for a Small Language

<program> → **begin** <stmt_list> **end**
<stmt_list> → <stmt>
| <stmt> ; <stmt_list>
<stmt> → <var> = <expression>
<var> → A | B | C
<expression> → <var> + <var>
| <var> – <var>
| <var>

An example of a leftmost derivation of a program in the Small Language

<program> ➔ begin <stmt_list> end

➔ begin <stmt> ; <stmt_list> end

➔ begin <var> = <expression> ; <stmt_list> end

➔ begin A = <expression> ; <stmt_list> end

➔ begin A = <var> + <var> ; <stmt_list> end

➔ begin A = B + <var> ; <stmt_list> end

➔ begin A = B + C ; <stmt_list> end

➔ begin A = B + C ; <stmt> end

➔ begin A = B + C ; <var>= <expression> end

➔ begin A = B + C ; B = <expression> end

➔ begin A = B + C ; B = <var> end

➔ begin A = B + C ; B = C end

BNF Grammar Example 3.2 for a Simple Assignment Statement and example derivation

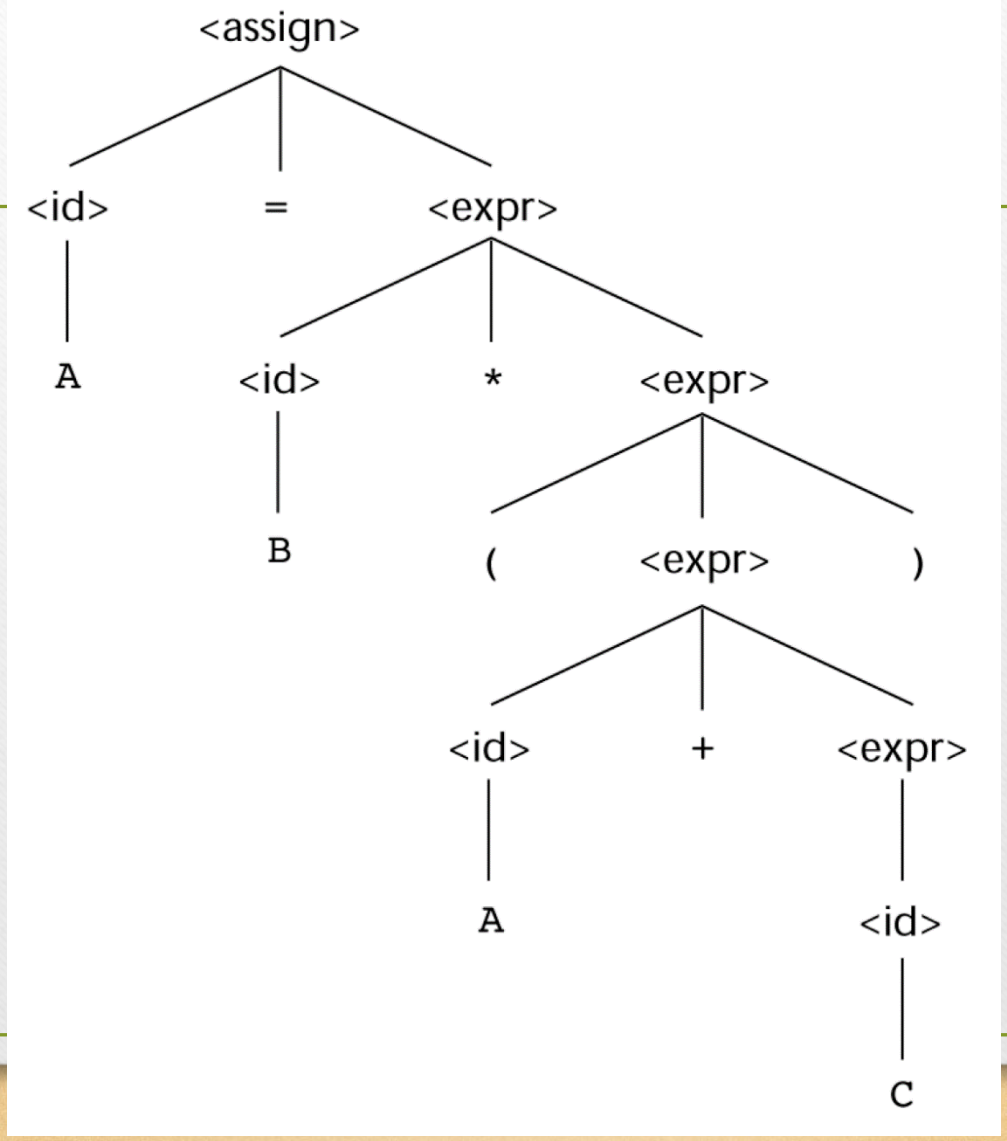
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

An example leftmost derivation of this grammar:

<assign> ➔ <id> = <expr>
. ➔ A = <expr>
. ➔ A = <id> * <expr>
. ➔ A = B * <expr>
. ➔ A = B * (<expr>)
. ➔ A = B * (<id> + <expr>)
. ➔ A = B * (A + <expr>)
. ➔ A = B * (A + <id>)
. ➔ A = B * (A + C)

Figure 3.1

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



An Example Grammar

<program> → <stmts>

<stmts> → <stmt> | <stmt> ; <stmts>

<stmt> → <var> = <expr>

<var> → a | b | c | d

<expr> → <term> + <term> | <term> - <term>

<term> → <var> | const

An Example Derivation

Ambiguity

A grammar is ambiguous if it can produce two distinct parse trees for the same expression

EXAMPLE 3.3 An Ambiguous Grammar for Simple Assignment Statements

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

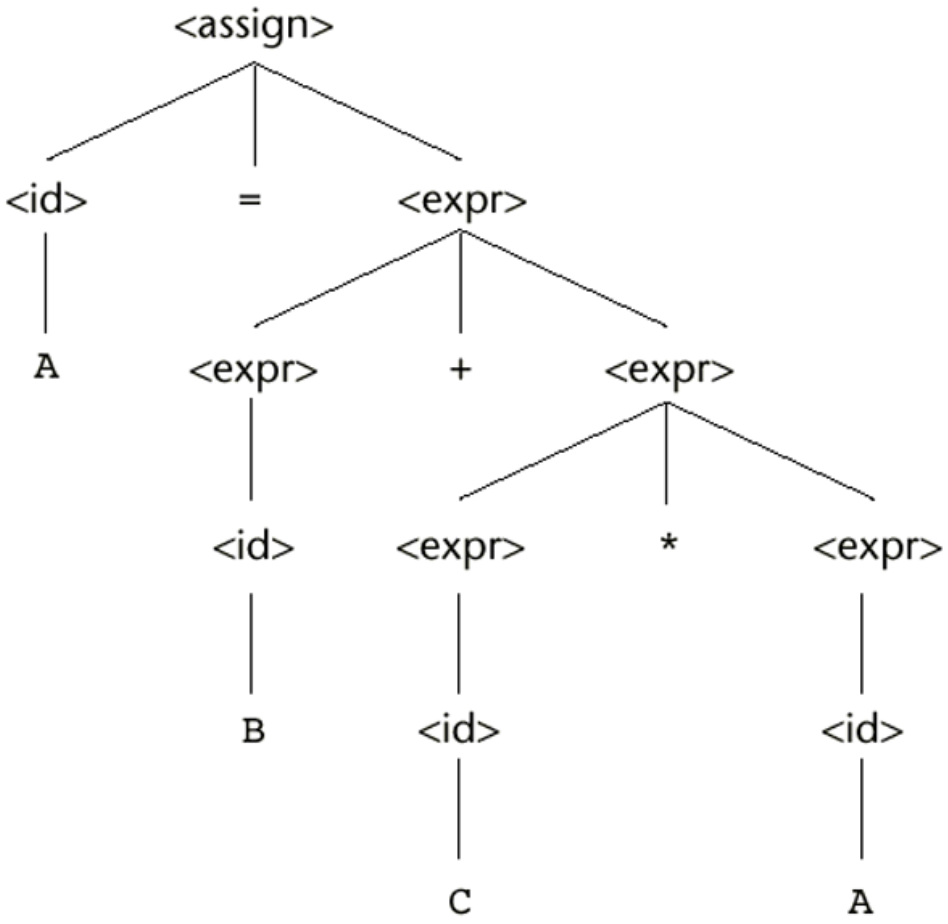
Compare to Example 3.2 which is not ambiguous:

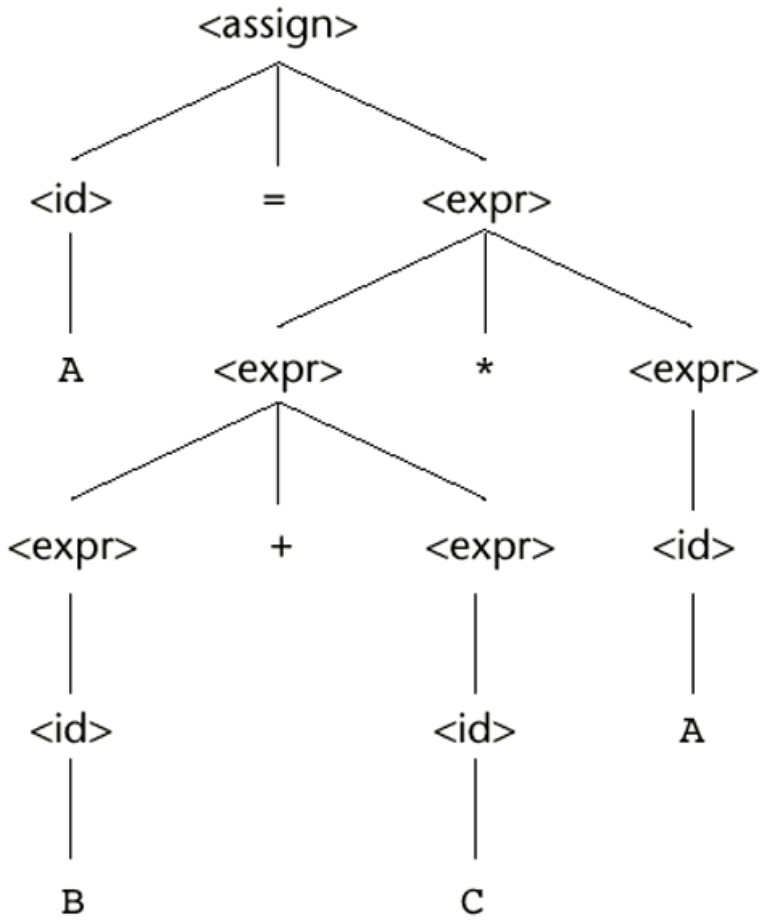
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

Ambiguous Grammars

Figure 3.2

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



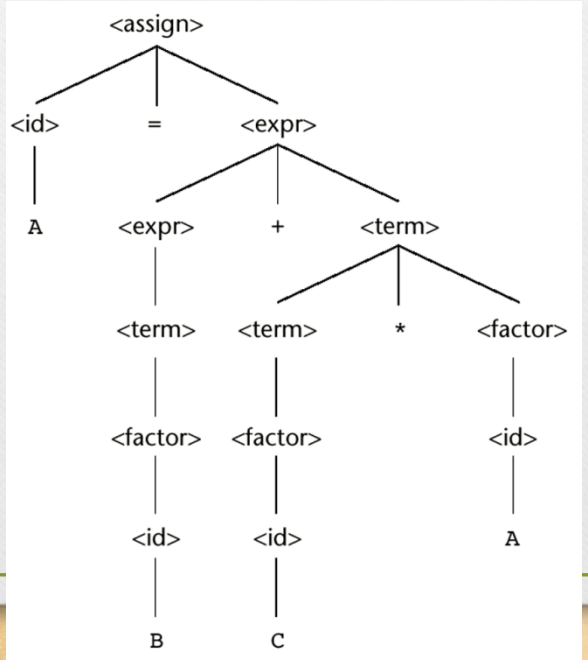


EXAMPLE 3.4 An Unambiguous Grammar for Expressions

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <expr> + <term>
| <term>
<term> → <term> * <factor>
| <factor>
<factor> → (<expr>)
| <id>

Parse tree for:

A = B + C * A



Note that rightmost and leftmost derivations produce the same parse tree.

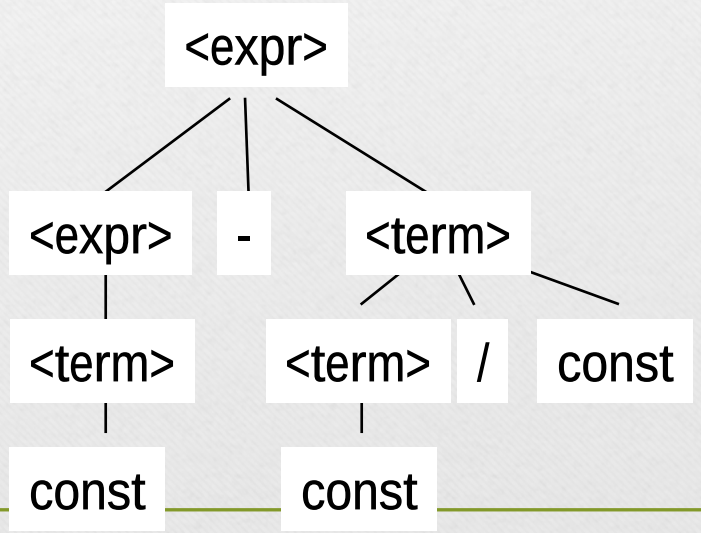
Compare to Example 3.2 which is not ambiguous but does not enforce precedence.

<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> + <expr>
| <id> * <expr>
| (<expr>)
| <id>

An Unambiguous Expression Grammar

- If we use the parse tree to indicate precedence levels of the operators, we cannot have ambiguity

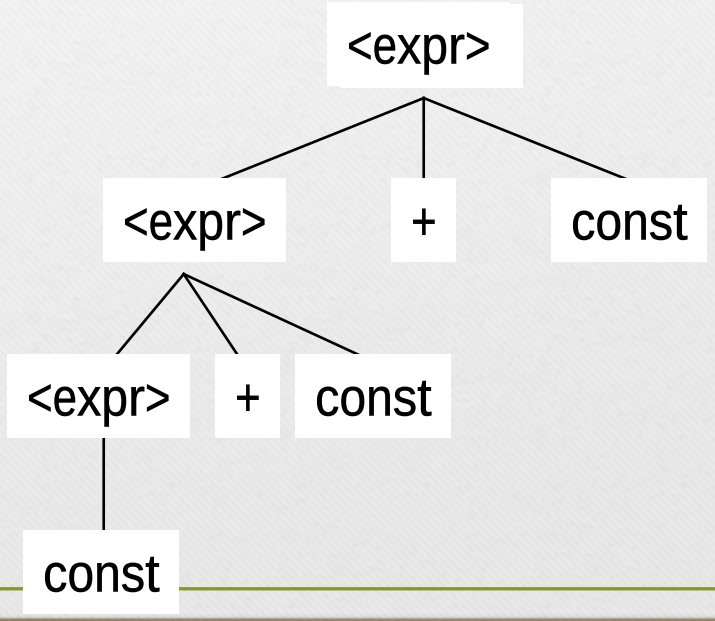
```
<expr> → <expr> - <term>  |  <term>
<term> → <term> / const| const
```



Associativity of Operators

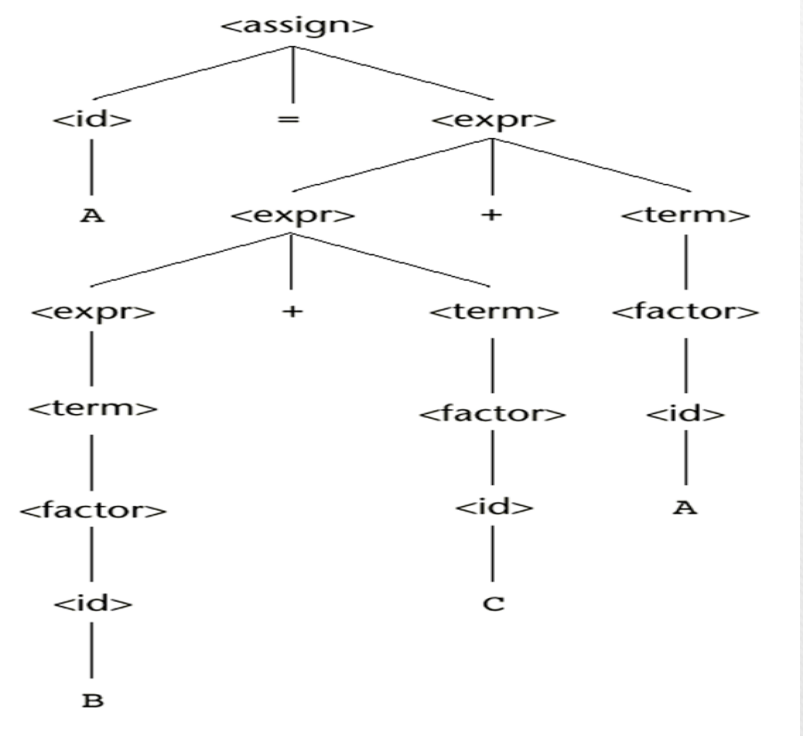
- Operator associativity can also be indicated by a grammar

```
<expr> -> <expr> + <expr> |  const  (ambiguous)
<expr> -> <expr> + const  |  const  (unambiguous)
```



Associativity of Operators

- A = B + C + A and Grammar 3.4



- For right associativity:
  - <factor> → <exp> ** <factor>
    | <exp>
    <exp> → (<expr>) | id

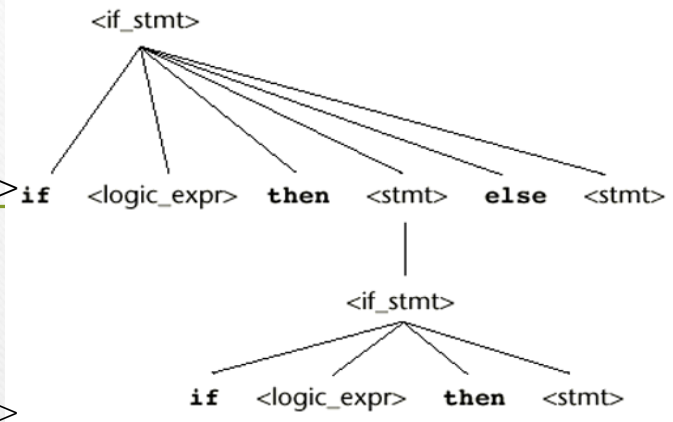
…

Figure 3.5 extra

<stmt> → <if_stmt>

<if_stmt> → if <logic_expr> then <stmt>

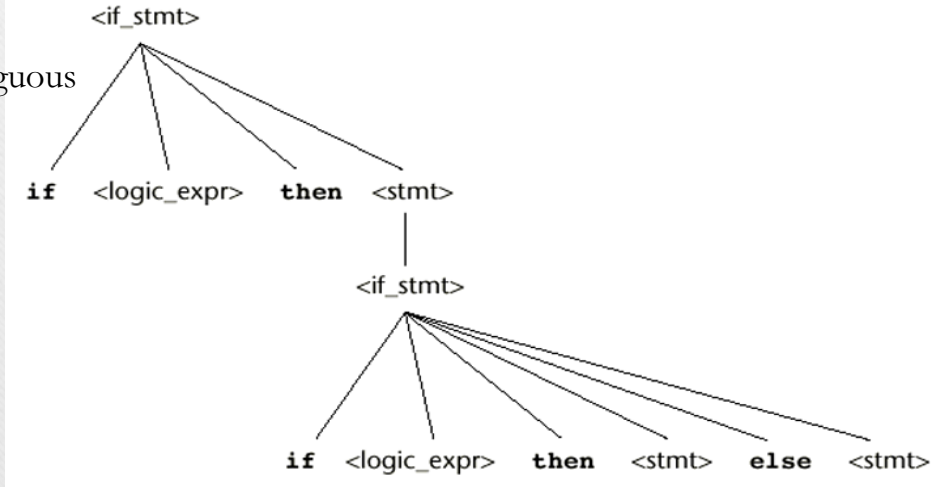
| if <logic_expr> then <stmt> else <stmt>



Consider the sentential form:

if <logic_expr> then if <logic_expr> then <stmt> else <stmt>

Yields two distinct parse trees (seel right) and is therefore ambiguous



Consider:

```
if (done == true)
    then if (denom == 0)
      then quotient = 0;
      else quotient = num / denom;
```

Problem: If the upper parse tree is used, then the outermost if will be associated with the else.

Unambiguous Grammar for the If Statement

Revised Grammar

<stmt> → <matched> | <unmatched>

<matched> → if <logic_expr> then <matched> else<matched>
| any non-if statement

<unmatched> → if <logic_expr> then <stmt>
| if <logic_expre> then <matched>
else <unmatched>

Now just one possible parse tree for the following

if <logc_expr> then if <logic_expr> then <stmt> else <stmt>

EBNF

- Extended Backus-Naur Form
  - Uses Metasymbols [], {}, | to make the notation more concise.

1. Optional RHS denoted by brackets **[]**
   - <selection> → if (<expression>) <stmt> [else <stmt>]
2. Zero or more repetitions denoted by braces **{}**
   - <ident_list> → <ident> { , <ident> }
3. Choice of a single element from a group denoted by placing options in parentheses and separated by **|**
   - <term> → <term> (* | / | %) <factor>

- Example 3.5 on page 133 (see next slide)

EXAMPLE 3.5 BNF and EBNF Versions of an Expression Grammar

BNF:

```
<expr> → <expr> + <term>
       | <expr> - <term>
       | <term>
<term> → <term> * <factor>
       | <term> / <factor>
       | <factor>
<factor> → <exp> ** <factor>
         | <exp>
<exp> → ( <expr> )
      | id
```

EBNF:

```
<expr> → <term> {(+ | -) <term>}
<term> → <factor> {(* | /) <factor>}
<factor> → <exp> { ** <exp>}
<exp> → ( <expr> )
      | id
```

BNF and EBNF

- BNF

<expr> → <expr> + <term>
| <expr> - <term>
| <term>

<term> → <term> * <factor>
| <term> / <factor>
| <factor>

- EBNF

<expr> → <term> {(+ | -) <term>}

<term> → <factor> {(* | /) <factor>}

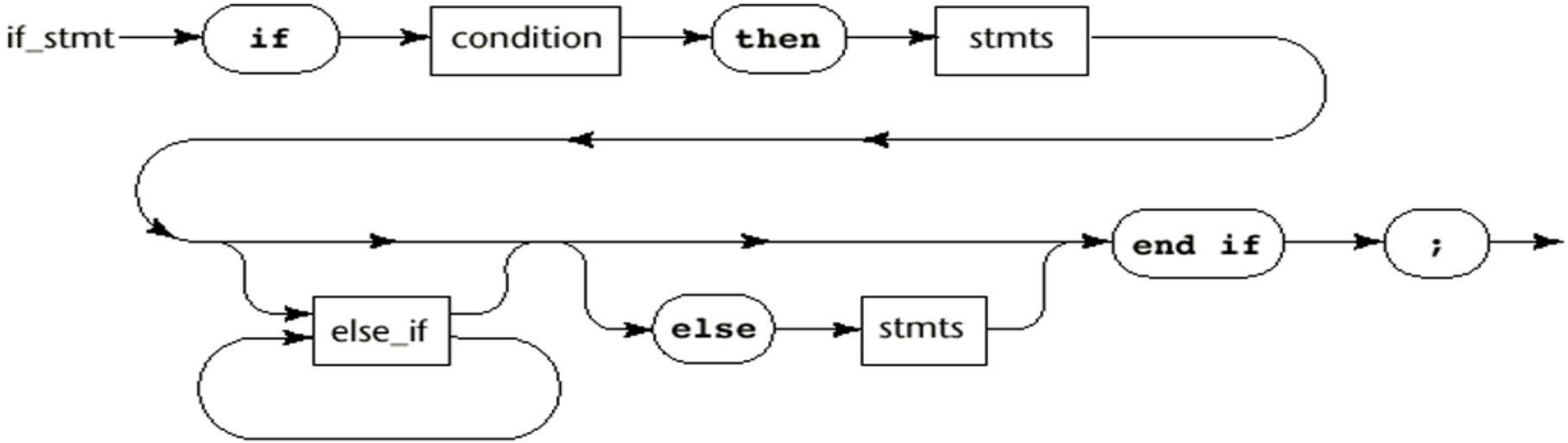
Recent Variations in EBNF

- Alternative RHSs are put on separate lines
- Use of a colon instead of `=>`
- Use of `opt` for optional parts
- Use of `oneof` for choices

Syntax Diagrams

- Syntax Diagrams (graphs) are used to represent the entire syntactic structure of a parse.
- Also used to represent the syntax of a single rule.
- Example: see next slide

The syntax diagram (graph) and EBNF descriptions of the Ada if statement



<if_stmt> ⟶ **if** <condition> **then** <stmts> {<else_if>}

[**else** <stmts>] **end if**

<else_if> ⟶ **elsif** <condition> **then** <stmts>

More Examples of Syntax Diagrams