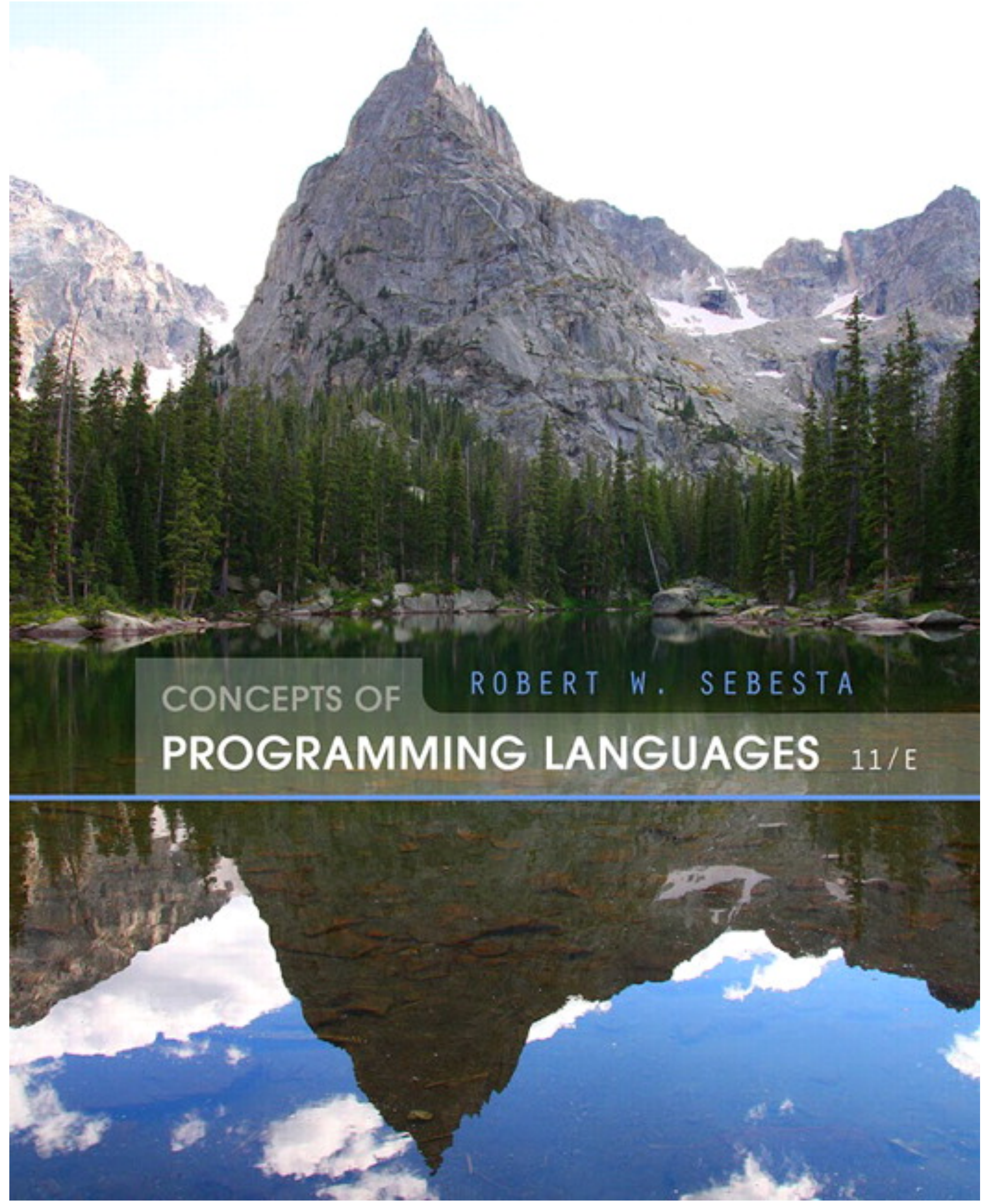
Chapter 3

Describing Syntax and Semantics

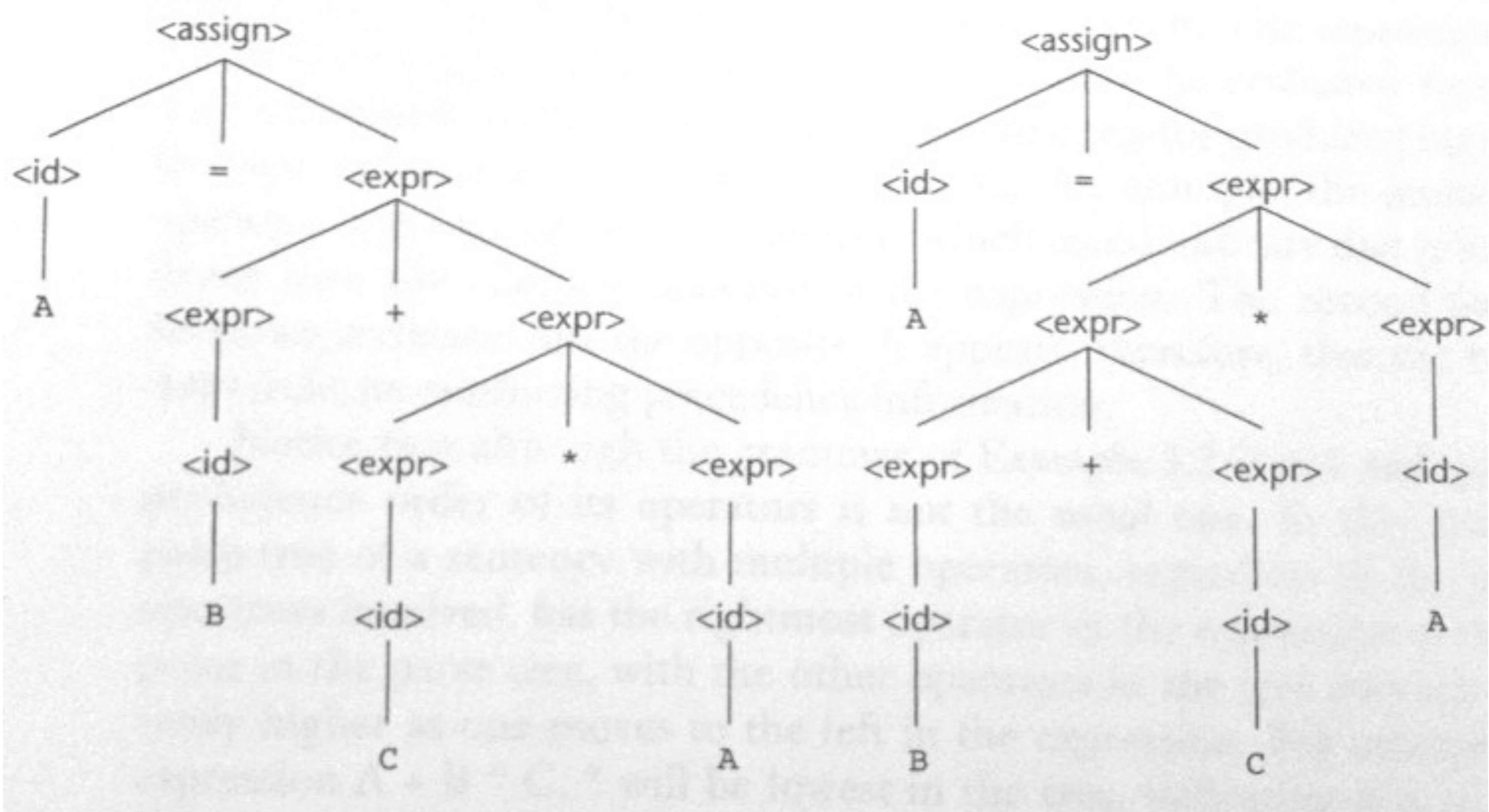


Chapter 3 Topics

- Introduction
- The General Problem of Describing Syntax
- Formal Methods of Describing Syntax
- Attribute Grammars
- Describing the Meanings of Programs: Dynamic Semantics

Ambiguous grammar

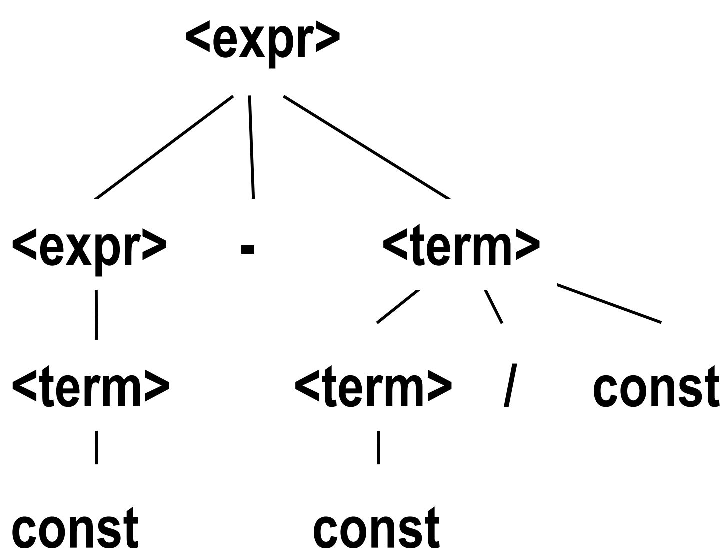
- 2 parse trees for the sentence A=B+C*A
- Operator precedence
- Conflicting precedence



An Unambiguous Expression Grammar

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

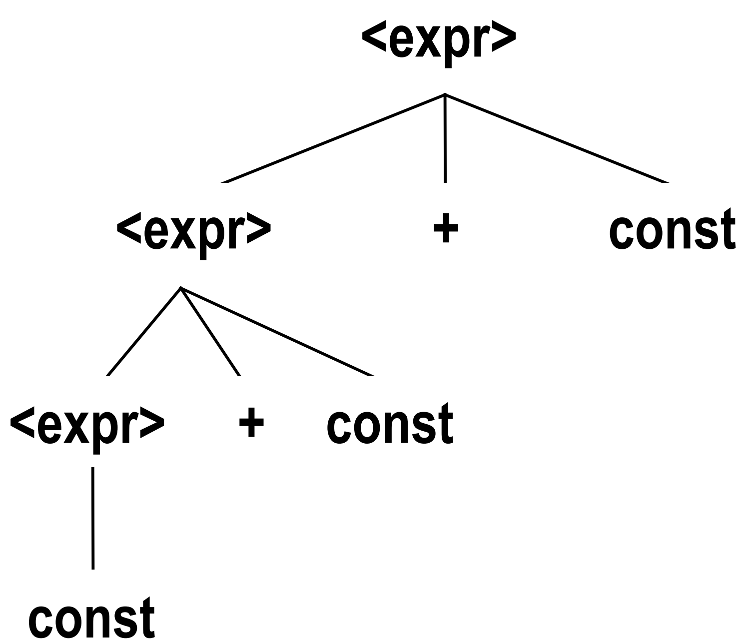
```
<expr> → <expr> - <term>  |  <term>
<term> → <term> / const| const
```



Associativity of Operators

- Operator associativity can also be indicated by a grammar

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



Extended BNF (EBNF)

- Optional parts are placed in brackets [ ]

```
<if_stmt> -> if (<expression>)
<statement> [else <statement>]
```

- Alternative parts of RHSs are placed inside parentheses and separated via vertical bars

```
<term> → <term> (+|-) const
```

- Repetitions (0 or more) are placed inside braces { }

```
<ident_list> → <identifier>
{,<identifier>}
```

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 $_{\texttt{opt}}$ for optional parts
- Use of `oneof` for choices

Example of parsing string and generating error (from chap 4)

- Parsing examples as part of compilation process (chapter 4) and generating errors
- Example recursive-descent parser using a parse tree written in C
- Follows the generative, top-down, process of the EBNF grammar, with collections of subprograms that could be recursive
- Subprogram for each non terminal rule; traces parse tree rooted at that non terminal
- Starts from root and does leftmost derivation
- We assume function lex() gets the next lexeme and puts its token code in the global variable nextToken
- This (chapter 4) material is presented as example, beyond scope of course

Example of parsing string and generating error (from chap 4)

EBNF rule: <expr> → <term> {(+ | -) <term>}

<term> -> <factor> {(* | /) <factor>}

<factor> → id | int_constant | (<expr>)

```
/* expr
   Parses strings in the language generated by the rule:
   <expr> -> <term> {(+ | -) <term>}
   */
void expr() {

printf("Enter <expr>\n");

/* Parse the first term */
  term();
/* As long as the next token is + or -, get
   the next token and parse the next term */
while (nextToken == ADD_OP || nextToken == SUB_OP) { lex();

term(); }

  printf("Exit <expr>\n");
}  /* End of function expr */
```

Example of parsing string and generating error (from chap 4)

EBNF rule: <expr> → <term> {(+ | -) <term>}

<term> -> <factor> {(* | /) <factor>)

<factor> → id | int_constant | (<expr>)

```
/* term
   Parses strings in the language generated by the rule:
   <term> -> <factor> {(* | /) <factor>)
   */
void term() {

printf("Enter <term>\n");

/* Parse the first factor */
  factor();
/* As long as the next token is * or /, get the
   next token and parse the next factor */
while (nextToken == MULT_OP || nextToken == DIV_OP) { lex();

factor(); }

  printf("Exit <term>\n");
}  /* End of function term */
```

Example of parsing string and generating error (from chap 4)

EBNF rule: <expr> → <term> {(+ | -) <term>}

<term> -> <factor> {(* | /) <factor>)

<factor> → id | int_constant | (<expr>)

```
/* factor
   Parses strings in the language generated by the rule:
   <factor> -> id | int_constant | ( <expr )
   */
void factor() {

printf("Enter <factor>\n");

/* Determine which RHS */
if (nextToken == IDENT || nextToken == INT_LIT)

/* Get the next token */
    lex();
```

Example of parsing string and generating error (from chap 4)

EBNF rule: <expr> → <term> {(+ | -) <term>}

<term> -> <factor> {(* | /) <factor>)

<factor> → id | int_constant | (<expr>)

```
/* If the RHS is ( <expr>), call lex to pass over the
   left parenthesis, call expr, and check for the right
   parenthesis */
else {
if (nextToken == LEFT_PAREN) {

lex();
expr();
if (nextToken == RIGHT_PAREN)

lex();

else

        error();
    }  /* End of if (nextToken == ... */
```

Example of parsing string and generating error (from chap 4)

EBNF rule: <expr> → <term> {(+ | -) <term>}

<term> -> <factor> {(* | /) <factor>)

<factor> → id | int_constant | (<expr>)

```
/* It was not an id, an integer literal, or a left
   parenthesis */
else

      error();
  }  /* End of else */
  printf("Exit <factor>\n");;
}  /* End of function factor */
```

Example of parsing string and generating error (from chap 4)

<ifstmt> → if (<boolexpr>) <statement> [else <statement>]

```
/* Function ifstmt
   Parses strings in the language generated by the rule:
   <ifstmt> -> if (<boolexpr>) <statement>
                 [else <statement>]
*/
void ifstmt() {
/* Be sure the first token is 'if' */
if (nextToken != IF_CODE)
error(); else {
/* Call lex to get to the next token */
   lex();
/* Check for the left parenthesis */
if (nextToken != LEFT_PAREN)
error(); else {
/* Call boolexpr to parse the Boolean expression */
    boolexpr();
/* Check for the right parenthesis */
if (nextToken != RIGHT_PAREN)
error();
```

Example of parsing string and generating error (from chap 4)

<ifstmt> → if (<boolexpr>) <statement> [else <statement>]

```
else {
/* Call statement to parse the then clause */
      statement();
/* If an else is next, parse the else clause */
if (nextToken == ELSE_CODE) {
/* Call lex to get over the else */
        lex();
        statement();
      } /* end of if (nextToken == ELSE_CODE ... */
    } /* end of else of if (nextToken != RIGHT ... */
  } /* end of else of if (nextToken != LEFT ... */
 } /* end of else of if (nextToken != IF_CODE ... */
} /* end of ifstmt */
```

Reminder: BNF and Context-Free Grammars

- Context-Free Grammars
  - Developed by Noam Chomsky in the mid-1950s for natural languages
  - Language generators, meant to describe the syntax of natural languages
  - Define a class of languages called context-free languages
- Backus-Naur Form (BNF) (1959)
  - Invented by John Backus to describe Algol 58
  - BNF is equivalent to context-free grammars

Reminder: BNF Fundamentals

- In BNF, abstractions are used to represent syntactic structures (also called nonterminal symbols, or just nonterminals)
- Terminals are lexemes or tokens
- A rule has a left-hand side (LHS), which is a nonterminal, and a right-hand side (RHS), which is a string of terminals and/or nonterminals
- Nonterminals are often enclosed in angle brackets
  - Examples of BNF rules:

```
<assign> → <var> = <expression>
<if_stmt> → if <logic_expr> then <stmt>
```

- Grammar: a finite non-empty set of rules
- A start symbol is a special element of the nonterminals of a grammar

Static semantics

- BNF form we have been discussing (Context free grammars) cannot describe all of the syntax of programming languages
- Categories of constructs that are trouble:

  - Context-free, but cumbersome (e.g., types of operands in expressions; Java floating-point value cannot be assigned to integer type, but opposite legal)

  - Non-context-free (e.g., variables must be declared before they are used)
- These type of needed specification checks are referred to as **Static Semantics**

Attribute Grammars

- Attribute grammars are used to describe more of the structure of PL than we can do with Context free grammars, e.g. to address static semantics such as type compatibility
- Attribute grammars (AGs) have additions to Context free grammars to carry some semantic info on parse tree nodes
- Primary value of AGs:
  - Static semantics specification
  - Compiler design (static semantics checking)

Attribute Grammars : Definition

- Def: An attribute grammar is a context-free grammar with the following additions:
  - For each grammar symbol x there is a set A(x) of attribute values
  - Each rule has a set of functions that define certain attributes of the nonterminals in the rule
  - Each rule has a (possibly empty) set of predicates, which state the static semantic rules, to check for attribute consistency

Attribute Grammars: Definition

- Let $X_0 \rightarrow X_1 \ldots X_n$ be a rule
- Synthesized attributes - up the parse tree from children
- Inherited attributes - down and across parse tree
- Initially, there are intrinsic attributes on the leaves (such as actual types of variables, int or real)

Attribute Grammars (continued)

- How are attribute values computed?
  - If all attributes were inherited, the tree could be decorated in top-down order.
  - If all attributes were synthesized, the tree could be decorated in bottom-up order.
  - In many cases, both kinds of attributes are used, and it is some combination of top-down and bottom-up that must be used.