Chapter 3

Describing Syntax and Semantics

Topics

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

  Dynamic Semantics

Introduction

- Syntax: the *form* or *structure* of the expressions, statements, and program units.
- Semantics: the *meaning* of the expressions, statements, and program units.
- Syntax and semantics provide a language's definition
  - Users of a language definition
    - Other language designers
    - Implementers
      - how the expressions, statements, and program units of a language are formed, and also their intended effect when executed.
    - Programmers (the users of the language)
      - Refer to a language reference manual.

Introduction (cont.)

- Semantics: the *meaning* of the expressions, statements, and program units
- E.g.) **`while`** `(Boolean_expr) statement`
- The semantics:

  If the current value of the Boolean expression is true,
  the embedded statement is executed;
  then, control implicitly returns to the Boolean expression
  to repeat the process.
  If the Boolean expression is false,
  control transfers to the statement
  following the **`while`** construct.

The General Problem of Describing Syntax: Terminology

- A *sentence* is a *string* of characters over some alphabet (Σ).
- A *language* is a set of sentences.

  e.g.) Σ = {a, b, …, *, =, ;}, w = 'could', L = {w| $w \in \Sigma^*$}= {*a, ab, aa, bb, ba, ..*}
- A *lexeme* is the lowest level *syntactic unit* of a language:

  e.g.) `*, sum, begin`
- A *token* is a *category* of lexemes (e.g., identifier)

  e.g.) `index = 2 * count + 17;`

Lexeme	Tokens	Lexeme	Tokens
`index`	identifier	`count`	identifier
`=`	equal_sign	`+`	plus_op
`2`	int_literal	`17`	int_literal
`*`	mult_op	`;`	semicolon

Formal Definition of Languages

- Recognizers (for Regular Language)
  - A computing device that
    reads input strings over the alphabet ($\Sum$) of the language (L)
    and *decides whether the input strings belong to the language.*
    i.e. For a string $w \in \Sigma^*$, $w \in L$ (or $w \notin L$) ?
  - Example: lexical analyzer of a compiler
    - Details of syntax analysis in Chapter 4.
    - Finite Automaton, Regular Grammar -- CSci 435
- Generators
  - A device that *generates* sentences of a language.
  - It can decide if the *syntax* of a *sentence* is *syntactically correct* by comparing it to the structure of the generator (i.e. grammar of a language).

Basic Concepts: Language

- *Alphabet*: a set of symbols, i.e. $\Sigma = \{a, b\}$
- *String*: a *finite sequence of symbols* from Σ,
  such as $v = aba$ and $w = abaaa$
  - So, any string $u \in \Sigma^*$
- *Operations on strings*:
  - Concatenation, Reverse, Repetition ($*$, $^+$)
- Σ^* = a set of *all strings* formed
  by concatenating *zero or more* symbols in Σ.
  e.g.) $\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, \ldots\ldots\}$
  e.g.) $\Sigma^+ = \Sigma^* - \{\varepsilon\} = \{a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, bba, bbb, \ldots\ldots\}$
- A *formal language* is *any subset* of Σ^*: $\forall L \subseteq \Sigma^*$
- A string in a language is also called a *sentence* of the language.

Basic Concepts: Grammar

- A *rule* to describe the strings in a language,

  i.e. a *syntax* of a language – not a semantics.

- A grammar G is defined as G = (V, T, S, P) where

  V: a *finite* set of *variable* or *non-terminal symbols*

  T: a *finite* set of *terminal* symbols

  S ($\in$ V): a variable called the *start* symbol

  P: a *finite* set of *rules* (a.k.a. *productions*)

- Example 1: a grammar G = (V, T, S, P) where

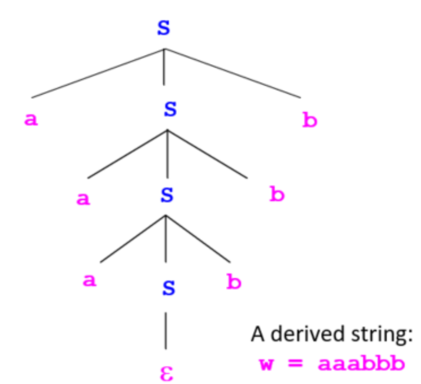
  V = { S }

  T = { *a*, *b* }

  P = { S $\rightarrow$ *aSb*, S $\rightarrow$ ε }

  Then, the language generated by G is, L(G) = $\{ a^n b^n \mid n \geq 0 \}$

  e.g.) L(G) = {ε, *ab*, *aabb*, *aaabbb*, *aaaabbbb*, ... }



Basic Concepts: Grammar (cont.)

- Beginning with the *start symbol S*, strings are derived by *repeatedly replacing Non-terminal symbols with the expression on the Right Hand Side (RHS) of any applicable rules:* LHS $\rightarrow$ RHS

  $S \Rightarrow$ sentential[1] $\Rightarrow$ … …. $\Rightarrow$ a *string* of terminals

- E.g.) Using grammar in Example 1:

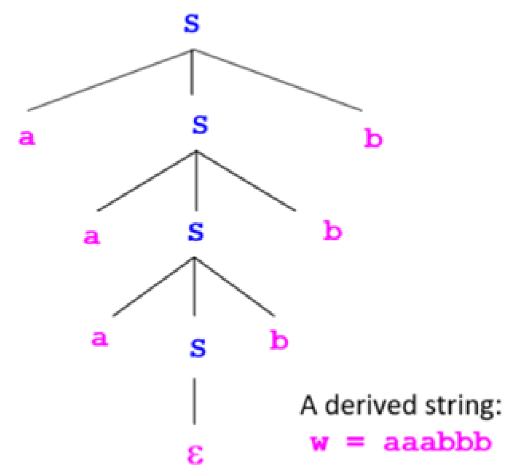
  $P = \{ S \rightarrow^1 aSb,\ S \rightarrow^2 \varepsilon \}$

  $S \Rightarrow^1 aSb$ (applying 1st rule)

  $\Rightarrow^1 aaSbb$ (applying 1st rule)

  $\Rightarrow^1 aaaSbbb$ (applying 1st rule)

  $\Rightarrow^2 aaabbb$ (applying 2nd rule)



- For a given grammar G=(V, T, S, P),

  the language generated by the grammar G,

  $L(G) = \{ w \in T^* \mid S \Rightarrow^* w \}$

  is the set of all strings derived from the start symbol.

Basic Concepts: Grammar (cont.)

- For convenience, rules with the same LHS are written on the same line: $\{S \rightarrow A,\ S \rightarrow B\} \Leftrightarrow S \rightarrow A \mid B$ (| - *OR*)
- E.g.) For a given grammar G=(V, T, S, P) with rules

$$\{S \rightarrow SS,\ S \rightarrow \varepsilon,\ S \rightarrow aSb,\ S \rightarrow bSa\}$$

$$\Leftrightarrow \qquad S \rightarrow SS \mid \varepsilon \mid aSb \mid bSa,$$

find $L(G) = \{w \in T^* \mid S \Rightarrow^* w\} =$? $= \{w \mid \#$ of a's $= \#$ of b's in $w\}$.

- Two grammars, G_1 and G_2, are *equivalent* ($G_1 \equiv G_2$) if they generate the same language: $L(G_1) = L(G_2)$.
- E.g.) G_1= (V, T, S, P) where V = { S }, T = { *a*, *b* }, P = $\{S \rightarrow aSb \mid \varepsilon\}$

G_2= (V, T, S, P) where V = { S, A }, T = { *a*, *b* },

P = $\{S \rightarrow aAb \mid \varepsilon,\ A \rightarrow aAb \mid \varepsilon\}$

G_1 and G_2 are equivalent because $L(G_1) = L(G_2) = \{a^n b^n \mid n \geq 0\}$.

Application:
Grammars for Programming Languages

- The syntax of constructs in a programming language is described with grammar.
- Assume that in a hypothetical programming language,
  - Identifiers consist of digits and the letters *a, b,* or *c*.
  - Identifiers must *begin with a letter*.
- E.g.) Rules for a grammar:

```
<id>     → <letter> <rest>
<rest>   → <letter> <rest> | <digit> <rest> | ε
<letter> → a | b | c
<digit>  → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
```

Formal Methods of Describing Syntax

BNF and Context-Free Grammar

- Context-Free Grammar (CFG)
  - Developed by Noam Chomsky in the mid-1950s.
  - Language generators, meant to describe the *syntax* of *natural/programming languages*.
  - define a class of languages called *Context-Free Languages (CFL)*.
  - Cf) Regular Grammar (RG):
    - the form of the ***tokens*** of programming language can be described.
- Backus-Naur Form (BNF, 1959)
  - A notation for describing syntax.
  - Invented by John Backus *to describe* the syntax of Algol 58.
  - CFG is denoted by BNF.

Context-Free Grammar (CFG): Formal Definition

- A grammar G = (V, T, S, P) is said to be ***context-free***

  if all rules in P have the form $LHS \to RHS$,

  where $LHS \in V$ and $RHS \in (V \cup T)^*$.

  A language generated by context-free grammar is

  Context-Free Language.

- Cf) A grammar G = (V, T, S, P) is said to be *right-linear* *(left-linear)*

  if all productions are of the form

  $$LHS \to xB \quad (LHS \to Bx) \quad \text{or} \quad LHS \to x,$$

  where $LHS, B \in V$, and $x \in T^*$.

- *i.e. at most one variable* symbol appears on the right side of any rule. If it occurs, it is the *rightmost symbol.*
- A *regular grammar* is one that is either right-linear (or left-linear).

Backus-Naur Form (BNF): Fundamentals

- BNF is a *metalanguage* for programming languages.
  - A metalanguage is a language that is used to describe another language.
- In BNF, *abstractions* are used to represent *syntactic structures.*
  - E.g.) $<assign> \rightarrow <var> = <expression>$

    Assignment statement represented by the abstraction `<assign>`.
  - They act like syntactic variables (*nonterminal symbols),* or *terminals.*
  - Left-hand side (LHS): the abstraction being defined.
  - Right-hand side (RHS): the definition of LHS.
- *Rule* or *Production* (of CFG in BNF): *LHS* $\rightarrow$ *RHS* (slide#13)
  - left-hand side (LHS): a *nonterminal/variable,*
  - right-hand side (RHS): a *string* of terminals and/or nonterminals.
- *Terminals:* lexemes or tokens.

BNF: Fundamentals (cont.)

- *Terminals:* lexemes or tokens.
- E.g.) `<assign> → <var> = <expression>` specifies that

  the abstraction `<assign>` is defined as
  an instance of the abstraction `<var>`, followed by the *lexeme* `=`,
  followed by an instance of the abstraction `<expression>`.
- *Nonterminals:*
  - Syntactic *variable*, often enclosed in < >.
  - E.g.) <ident_list> → identifier | identifier, <ident_list>

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

- (Grammar: a finite non-empty *set of rules.*
- *Start symbol:* a special element of the nonterminals
  of a grammar.)

Describing Lists

- Syntactic lists are described using *recursion*

  <ident_list> $\rightarrow$ identifier | identifier, <ident_list>

- A *derivation* is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols). – refer to slide #9

  e.g.) $S \Rightarrow$ *sentential_form*[1] $\Rightarrow$ … …. $\Rightarrow$ a *string* of terminals

  where *sentential_form* $\in (V \cup T)^*$. - refer to slide-#13

Derivations

- Every string of symbols in a derivation is a *sentential form.*
  - *sentential_form* $\in (V \cup T)^*$.
- A *sentence* is a sentential form that has only *terminal* symbols; i.e. *sentence* $\in T^*$
- *Leftmost derivation:* one in which the *leftmost nonterminal* in each sentential form is the one that is expanded.
- *Righftmost derivation:* the *rightmost nonterminal* is expanded in each sentential form.
- A derivation may be *either* leftmost *or* rightmost.
- E.g.) $V = \{ S, A, B \}$, $T = \{ a, b \}$, $P = \{S \rightarrow aAB,\ A \rightarrow bBb,\ B \rightarrow A \mid \varepsilon\}$
  - Leftmost deriv.: $S \Rightarrow aAB \Rightarrow abBbB \Rightarrow abbB \Rightarrow abb$
  - Rightmost der. : $S \Rightarrow aAB \Rightarrow aA \Rightarrow abBb \Rightarrow abb$

Example: Grammar for a small language

```
<program> → begin <stmt_list> end
<stmt_list> → <stmt> | <stmt> ; <stmt_list>
<stmt> → <var> = <expr>
<var> → A | B | C
<expr> → <var> + <var> | <var> - <var> | <var>
```

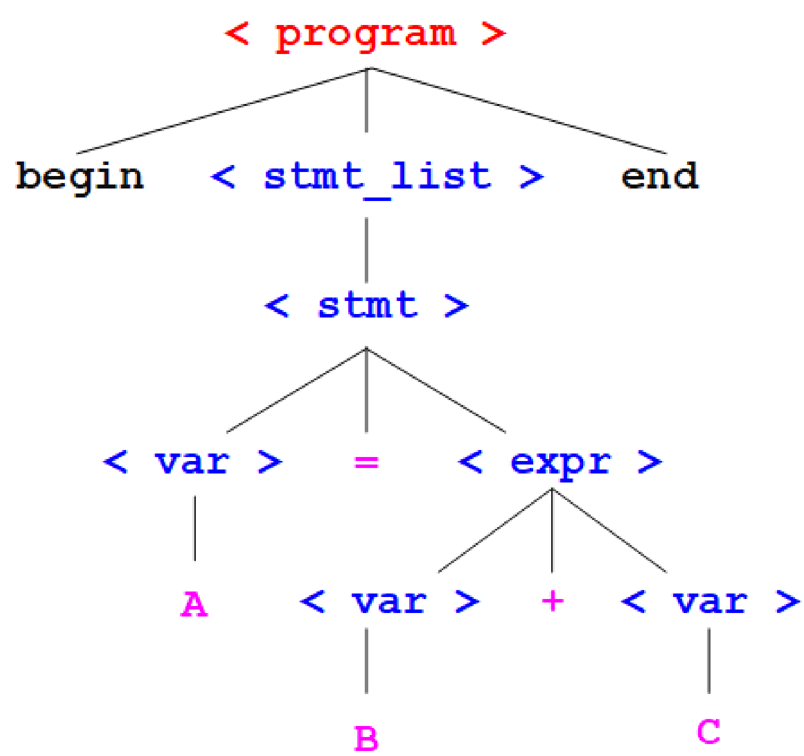
Example: (Leftmost) Derivation

```
<program> ⇒ begin <stmt_list> end
          ⇒ begin <stmt>; <stmt_list> end
          ⇒ begin <var> = <expr>; <stmt_list> end
          ⇒ begin A = <expr>; <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> = <expr> end
          ⇒ begin A = B + C; B = <expr> end
          ⇒ begin A = B + C; B = <var> end
          ⇒ begin A = B + C; B = C end
```

Parse Tree

- Parse Tree is a hierarchical representation of a derivation.

```
<program> → begin <stmt_list> end
 <stmt_list> → <stmt>
             | <stmt> ; <stmt_list>
 <stmt> → <var> = <expr>
 <var> → A | B | C
 <expr> → <var> + <var>
        | <var> - <var>
        | <var>
```



Ambiguity in Grammars

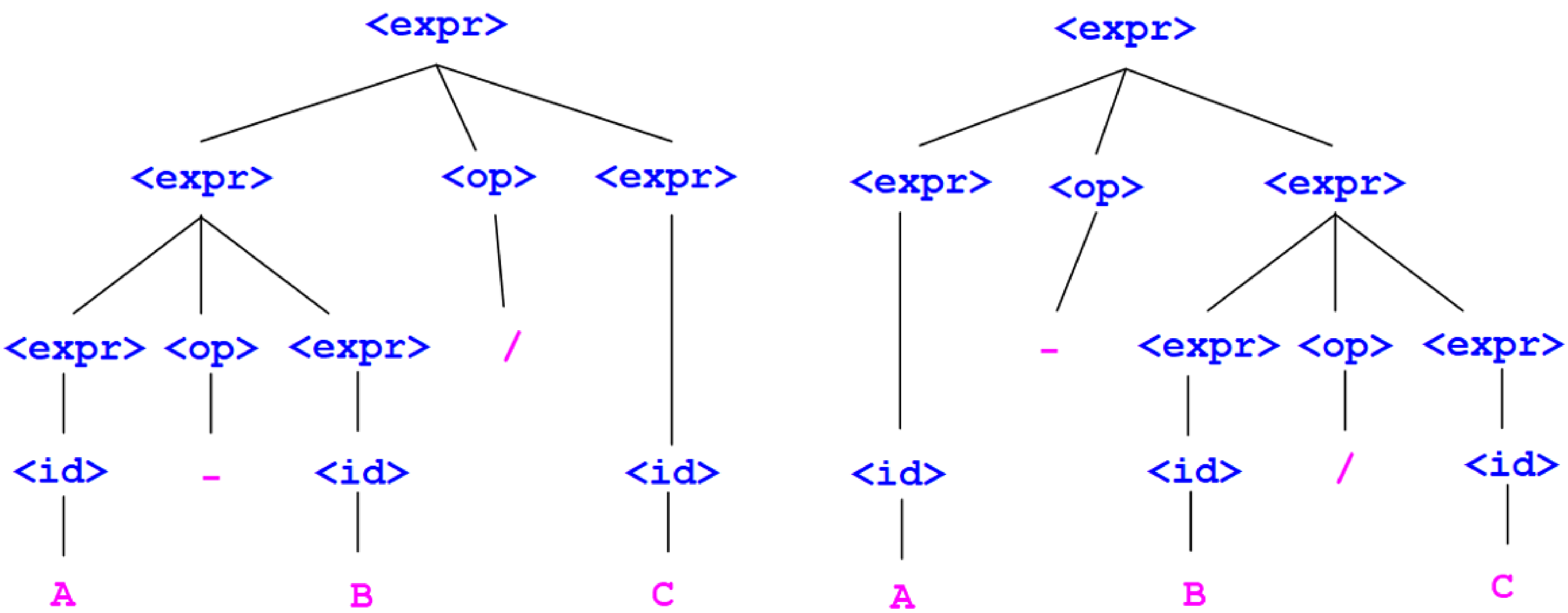
- A grammar is *ambiguous* if and only if it generates a sentential form that has two or more distinct parse trees.

An Ambiguous Grammar for Expression

<expr> → <expr> <op> <expr> | <id>

<id> → A | B | C

<op> → / | -



(A-B)/C

A-(B/C)

An Unambiguous Expression Grammar

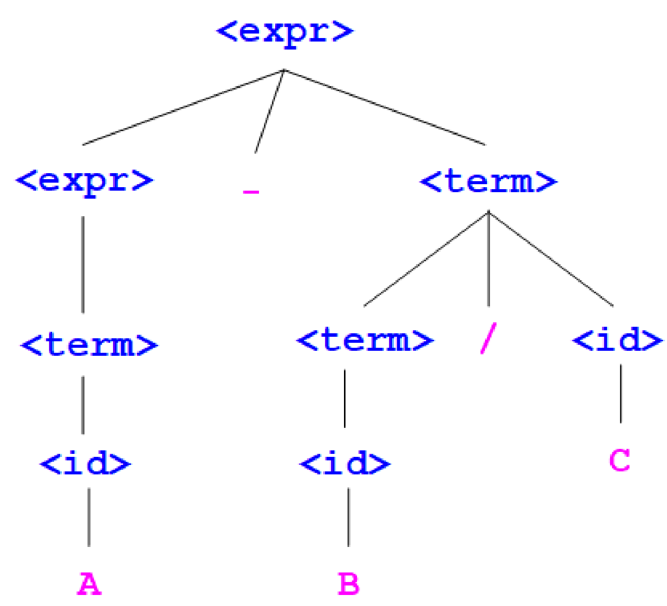
- If we use the parse tree to indicate *precedence levels* of the operators, we cannot have ambiguity.

  <expr> → <expr> - <term> | <term>
  <term> → <term> / <id> | <id>
  <id> → A | B | C

- <expr> was defined by employing a new non-terminal <term> to give the precedence of the operator.

- Note: an operator with priority in the evaluation (/) must be defined the lower level rule (in <term>).

- Derivation with *unambiguous grammar* → a *unique* parse tree!

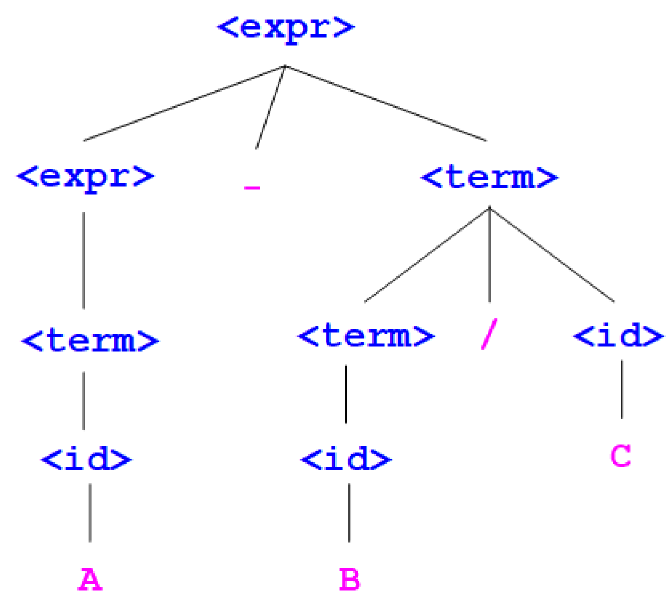


A-(B/C)

An Unambiguous Expression Grammar (cont.)

<expr> → <expr> - <term> | <term>
<term> → <term> / <id> | <id>
<id> → A | B | C

- *Unique* parse tree for the sentence.



- LeftMost derivation:
  <expr> ⇒ <expr> - <term>
  ⇒ <term> - <term>
  ⇒ <id> - <term> ⇒ A - <term>
  ⇒ A - <term> / <id>
  ⇒ A - <id> / <id>
  ⇒ A - B / <id> ⇒ A - B / C
- RightMost derivation (≠ LeftMost der.)
  <expr> ⇒ <expr> - <term>
  ⇒ <expr> - <term> / <id>
  ⇒ <expr> - <term> / C
  ⇒ <expr> - <id> / C ⇒ <expr> - B / C
  ⇒ <term> - B / C ⇒ <id> - B / C ⇒ A - B / C

- The derivation *procedure* is different, but the same sentence is derived by any derivation with Unambiguous grammar.

Associativity of Operators

- Associativity:

  A semantic rule to specify which operator should have *precedence* when an expression includes two operators that have the same precedence: e.g.) (+, *)

  - $(A + B) + C = A + (B + C)$,
  - $(A * B) * C = A * (B * C)$.

- Correct associativity may be essential for an expression that contains non-associative operators (–, ⁄).

  - $(A - B) - C \neq A - (B - C)$,
  - $(A / B) / C \neq A / (B / C)$.

Associativity of Operators

- *Left Recursive rule:*
  - The LHS of a rule also appears at the *beginning of its RHS, i.e. the left-most variable.*
  - Left recursive rule specifies *left associativity*.
  - E.g.) <term> → <term> / <id> | <id>

    <id> → A | B | C

- *Right Recursive rule:*
  - The LHS of a rule also appears at the *right end of its RHS, i.e. the right-most variable.*
  - Right recursive rule specifies *right associativity*
  - E.g.) <factor> → <exp> ** <factor> | <exp>

    <exp> → (<expr>) | id - exponentiation as a right-associative opr.

    - E.g.) $2^{2^3} = 2^8 = 256 \neq (4^3 = 64)$

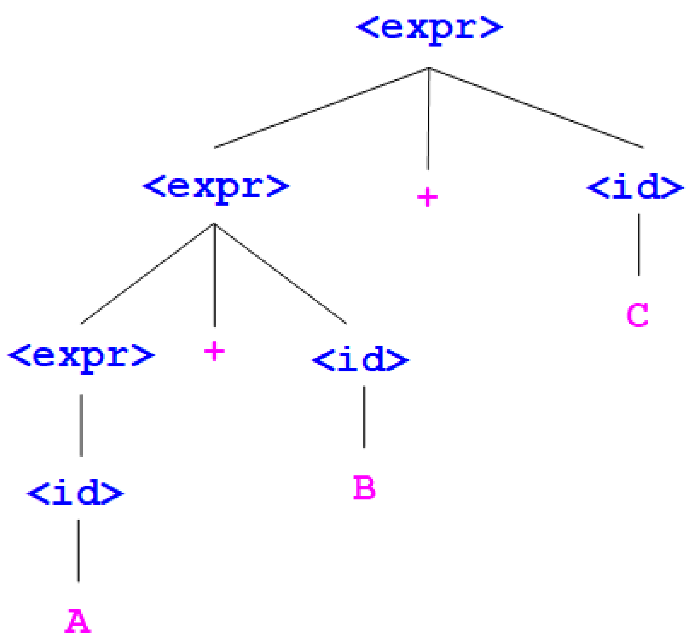
Associativity of Operators (cont.)

- Operator associativity can also be indicated by grammar.

<expr> → <expr> + <expr> | <id> (ambiguous)

Both (A+B)+C and A+(B+C) are derivable.

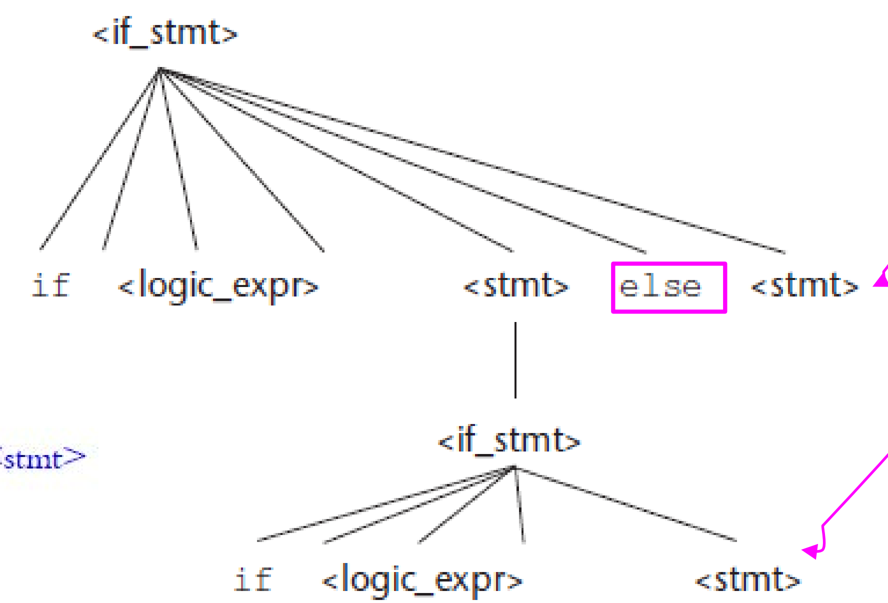
<expr> → <expr> + <id> | <id> (unambiguous, left-recursive for left-associativity)



Example: Unambiguous grammar for `if-else`

- BNF rules for a Java `if-else` statement:
  - <stmt> → <if_stmt>,
  - <if_stmt> → **if** (<logic_expr>) <stmt>
    | **if** (<logic_expr>) <stmt> **else** <stmt>
- E.g.) Sentential form:

  if (<logic_expr>) **if** (<logic_expr>) <stmt> **else** <stmt>



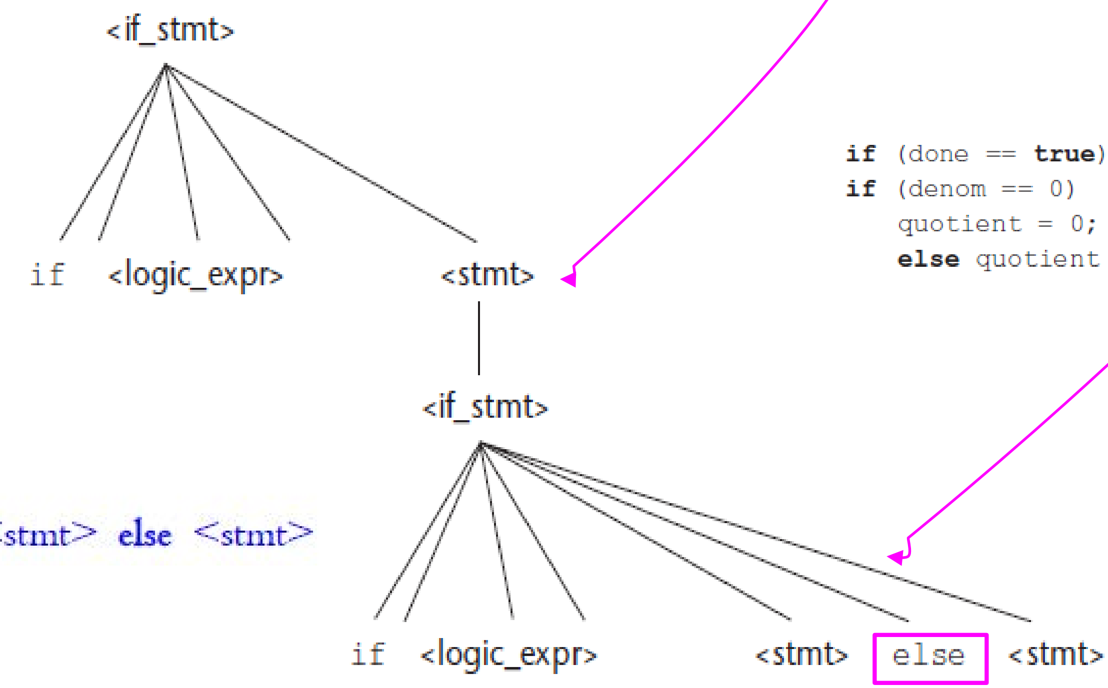
if <logic_expr> <stmt>

  if <logic_expr> <stmt>

else <stmt>

Example: Unambiguous grammar for `if-else`

- BNF rules for a Java `if-else` statement:
  - <stmt> → <if_stmt>,
  - <if_stmt> → **if** (<logic_expr>) <stmt>
    | **if** (<logic_expr>) <stmt> **else** <stmt>
- E.g.) Sentential form: **if** (<logic_expr>) **if** (<logic_expr>) <stmt> **else** <stmt>




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

if <logic_expr> <stmt>

if <logic_expr> <stmt> **else** <stmt>

- Which `then` clause does match with `else` clause? - ambiguous.
- Use different nonterminals to define the unambiguous grammar.

Example: Unambiguous grammar for `if-else`

- Rules for an *unambiguous grammar* for `if-else` statement:
  - An `else` clause, when present, is matched with the *nearest previous unmatched* `then` clause.
  - So, there can't be an `if` statement without an `else` between `then` clause and its matching `else`.
- Different categories of statement using different nonterminals:
  - `<matched>` vs. `<unmatched>`
- Unambiguous Grammar:

<stmt> → <matched> | <unmatched>

<matched> → **if** (<logic_expr>) <matched> **else** <matched>

| any non-if statement

<unmatched> → **if** (<logic_expr>) <stmt>

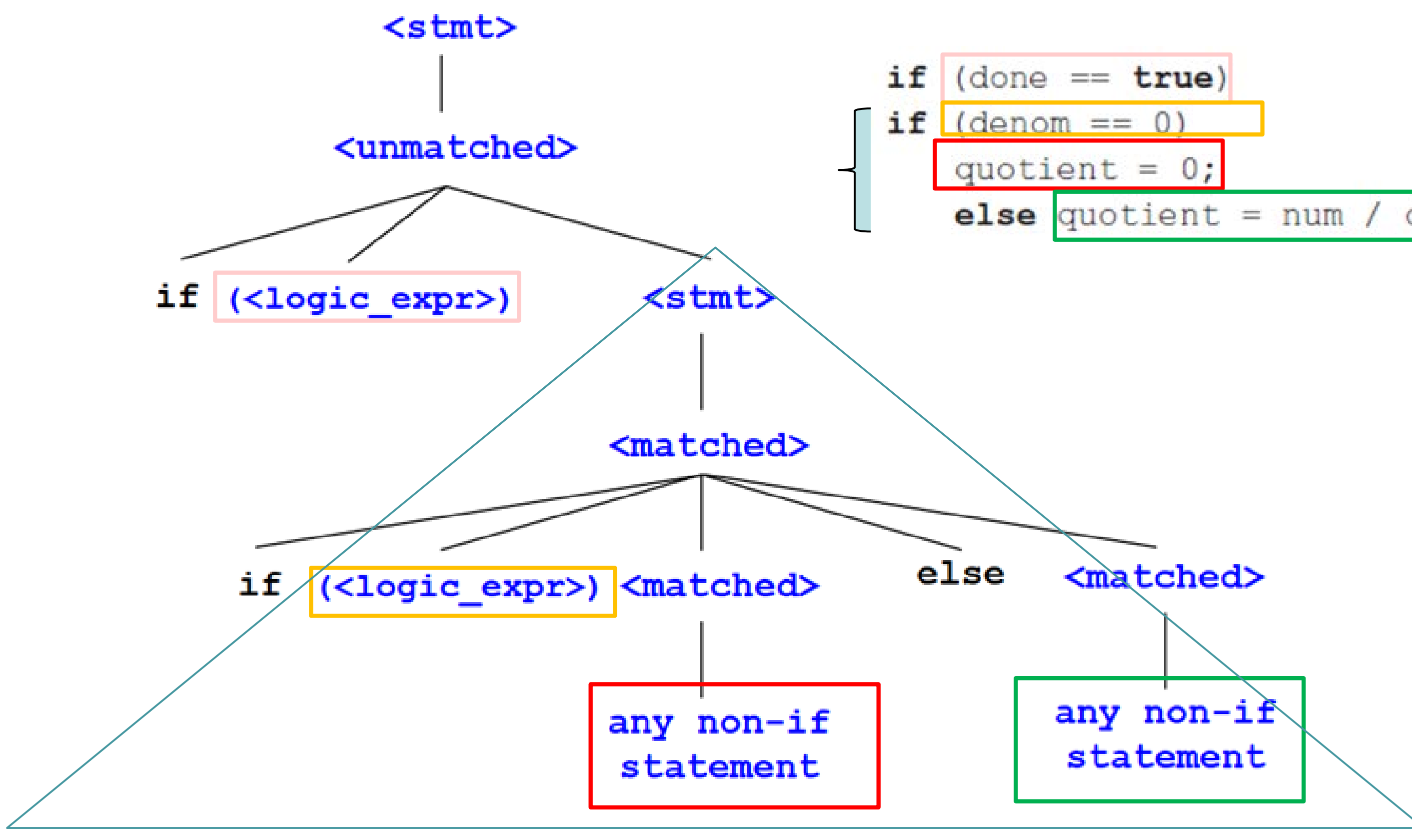
| **if** (<logic_expr>) <matched> **else** <unmatched>

- E.g.) Sentential form:

if (<logic_expr>) **if** (<logic_expr>) <stmt> **else** <stmt>

Example: Unambiguous grammar for `if-else` (cont.)

<stmt> → <matched> | <unmatched>
<matched> → **if** (<logic_expr>) <matched> **else** <matched>
| any non-if statement
<unmatched> → **if** (<logic_expr>) <stmt>
| **if** (<logic_expr>) <matched> **else** <unmatched>




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

Extended BNF

- Extend BNF to handle a few minor inconveniences.
- Optional parts are placed in brackets [ ]

  <proc_call> → <ident> [(<expr_list>)]
- 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> → letter {letter|digit}

Example 3.5: BNF and EBNF

- BNF

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

- EBNF (≡ BNF above)

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

- EBNF (≡ BNF above ? – a longer parse tree)

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

BNF and EBNF (cont.)

- Some variants in EBNF → CSci 435
  - A *numeric superscript* may indicate the # of repetition:
    e.g.) $\{\texttt{<stmt>}\}^{5}$
  - A *plus* (+) *superscript* may indicate *one or more* repetition:
    e.g.) `<stmt> {<stmt>}` is equivalent to $\{\texttt{<stmt>}\}^{+}$
  - A *star* (*) *superscript* may indicate *zero or more* repetition:
    e.g.) `{<stmt>}` is equivalent to $\{\texttt{<stmt>}\}^{*}$

Summary

- The grammar of programming language is Context-Free Grammar (CFG).
- BNF is a meta-language or a form that is used to express a CFG.
  - Well-suited for describing the syntax of programming languages.
- An attribute grammar is a descriptive formalism that can describe both the syntax and the semantics of a language.
- Three primary methods of semantics description
  - Operation, axiomatic, denotational