Describing Syntax

CS4080

Dr. Amar Raheja

Chapter 3

Syntax and Semantics

- What is the syntax and semantics of a programming language?
- **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
        - Programmers (the users of the language)

Syntax

- A well-designed programming language implies
  - Semantics follows directly from syntax
  - if (<expr>) then <statement>
- Formal system for describing syntax initially developed by Noam Chomsky in 1950s.
  - Father of Modern Linguistics
  - Chomsky Normal Form for context free grammer
- BNF :Notational system for describing context-free grammars for programming languages developed by John Backus and Peter Naur
- Three flavors: BNFs, EBNFs and syntax diagrams

General Problem of Describing Syntax

- A **sentence** is a string of characters over some alphabet
- A **language** is a set of sentences
- 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)
- Tokens have several distinct categories:
  - Reserved words (keywords): if, while, do, int etc.
  - Literals and constants: 35, 4.9, “hello” etc.
  - Special symbols(operators and separators): ; , +, ==, !, * etc.
  - Identifiers: xyz, interest_rate, hello etc.
- Identifying tokens in PLs
  - The principle of longest substring and token delimiters

Formal Definition of Languages

- Language Recognizers
  - Syntax analysis part of a compiler is a recognizer
  - Recognizes the language that the compiler translates
  - Role of recognizer-to determine if a program is in a certain language
  - Syntax analyzer determines if it is syntactically correct
  - Examples : PDAs
- Language Generators
  - Device that generates sentences of a language
  - Generate a sentence based on certain rules and submit to compiler to see if valid
  - Syntax of statement is correct by comparing it with structure of generator
  - Example: CFG
- Close connection between formal generation and recognition

BNF and Context-Free Grammars

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

BNF Fundamentals

- In BNF, abstractions are used to represent classes of syntactic structures they act like syntactic variables (also called **nonterminal symbols**, or just **terminals**)
- **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 <>
- Grammar: a finite non-empty set of rules
- A **start symbol** is a special element of the nonterminals of a grammar

BNF- Metalanguage

- A simple set of grammar rules in English
  1. sentence $\rightarrow$ noun-phrase verb-phrase '.'
  2. noun-phrase $\rightarrow$ article noun
  3. article $\rightarrow$ a | the
  4. noun $\rightarrow$ girl | dog
  5. verb-phrase $\rightarrow$ verb noun-phrase
  6. verb $\rightarrow$ sees | pets
- A legal sentence generation (called derivation) according to foregoing grammar rules

  sentence $\Rightarrow$ noun-phrase verb-phrase. (rule 1)

  sentence $\Rightarrow$ article noun verb-phrase. (rule 2)

  sentence $\Rightarrow$ the noun verb-phrase. (rule 3)

  sentence $\Rightarrow$ the girl verb-phrase. (rule 4)

  sentence $\Rightarrow$ the girl verb noun-phrase. (rule 5)

  sentence $\Rightarrow$ the girl pets noun-phrase. (rule 6)

  sentence $\Rightarrow$ the girl pets article noun. (rule 2)

  sentence $\Rightarrow$ the girl pets a noun. (rule 3)

  sentence $\Rightarrow$ the girl pets a dog. (rule 4)

Context-Free Grammars

- CFG: series of grammar rules such that
  - Left hand side which is a single structure name, followed by the metasymbol $\rightarrow$, followed by a right hand side
  - Right hand side can be mixture of lexemes , tokens or other abstractions
  - Name of abstractions called non-terminals as they can be broken into further structures (abstractions or terminals)
  - Lexemes and token symbols are called terminals, as they are never broken
  - Grammar rules are called productions
  - Symbols used | and sometimes parenthesis (<> or ()) are called metasymbols
  - Derivation: generating language sentences through a series of applications of the rules
  - Grammar for Simple Integer Arithmetic Expressions

    expr $\rightarrow$ expr + expr | expr * expr | (expr) | number

    number $\rightarrow$ number digit | digit (note recursion here)

    digit $\rightarrow$ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

    How many terminals, non-terminals and productions in the previous example?

BNF Rules

- An abstraction (or nonterminal symbol) can have more than one RHS

```
<stmt> → <single_stmt>
              | begin <stmt_list> end
```

- Syntactic lists are described using recursion

```
<ident_list> → ident
                  | ident, <ident_list>
```

- A derivation is a repeated application of rules, starting with the start symbol and ending with a sentence (all terminal symbols)

Example: Grammar of a PL

- Production Rules of a Grammar for Simple Integer Arithmetic
- How many productions, nonterminals and terminals ?

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

Example Leftmost Derivation

- A **leftmost derivation** is one in which the leftmost nonterminal in each sentential form is the one that is expanded
- A derivation may be neither leftmost nor rightmost
- Special words like, begin and end can be used.
- The start symbol is here is <program>
- Every string of symbols in the derivation is a *sentential form*
- A *sentence* is a sentential form that has only terminal symbols

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

Class Exercise

How many productions, nonterminals and terminals ?
<program> → **begin** <stmt_list> **end**
<stmt_list> → <stmt> | <stmt> ; <stmt_list>
<stmt> → <var> = <expr>
<var> → a | b | c
<expr> → <var> + <var> | <var> - <var> | <var>

Perform the following leftmost derivation:
begin a = b – c ; a = c **end**

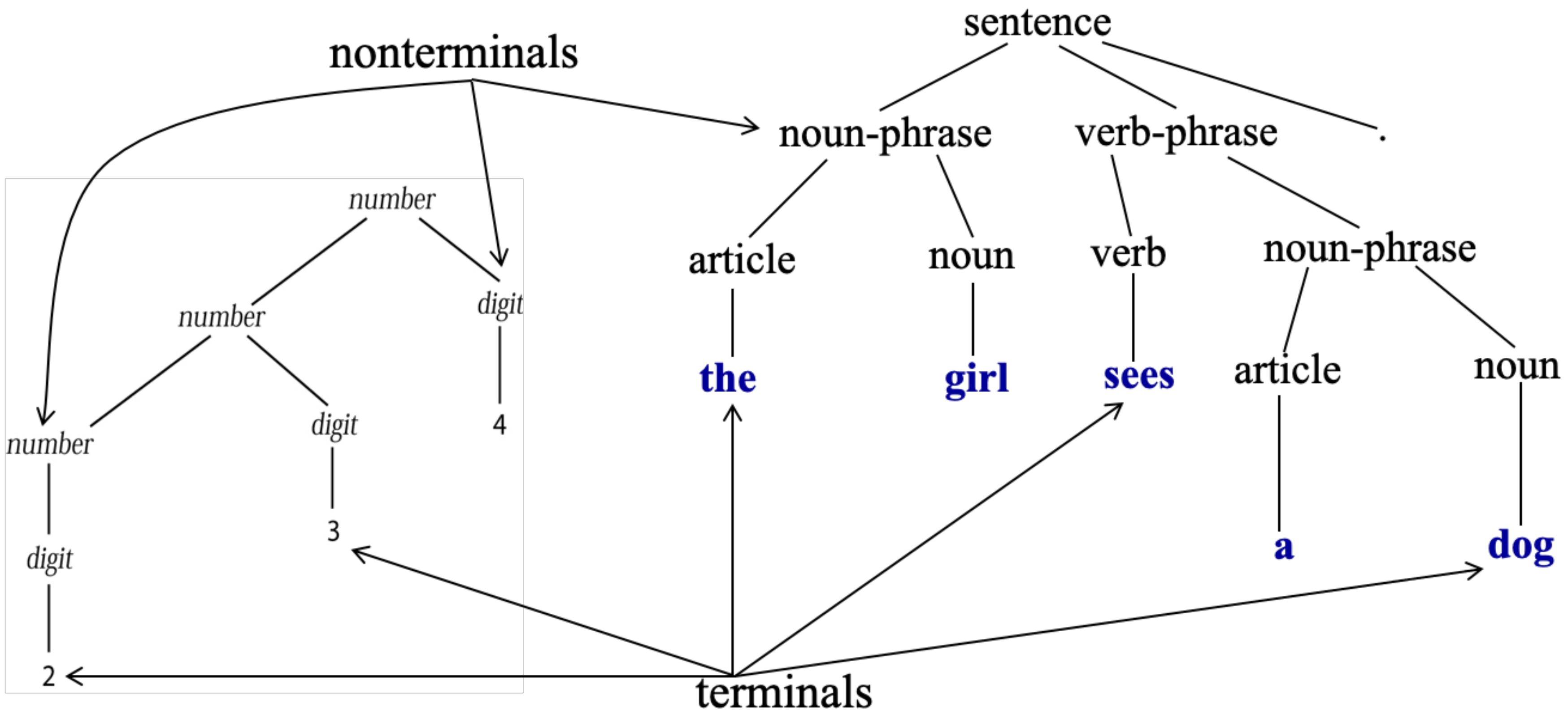
<assign> → <id> = <expr>
<id> → A | B | C
<expr> → <id> – <expr> | <id> * <expr> | (<expr>) | <id>
Perform the following leftmost derivation:
A = C * (A – B)

Why Context-Free Grammar?

- Non-terminals appear singly on the LHS of a production
- Hence each non-terminal can be replaced by any RHS choice, no matter where the non-terminal might appear
- Hence, no context under which only certain replacements can occur
  - e.g. the dog pets a girl
  - If it were context sensitive , verb "pets" used only when subject is "girl"
  - Another example of context-sensitivity is the use of a capitalized article in the beginning of the sentence
- How is it dealt with w.r.t. PLs ?
- Allow context-strings on LHS of the grammar rules or deal with it as a semantic issue, not a syntactic one

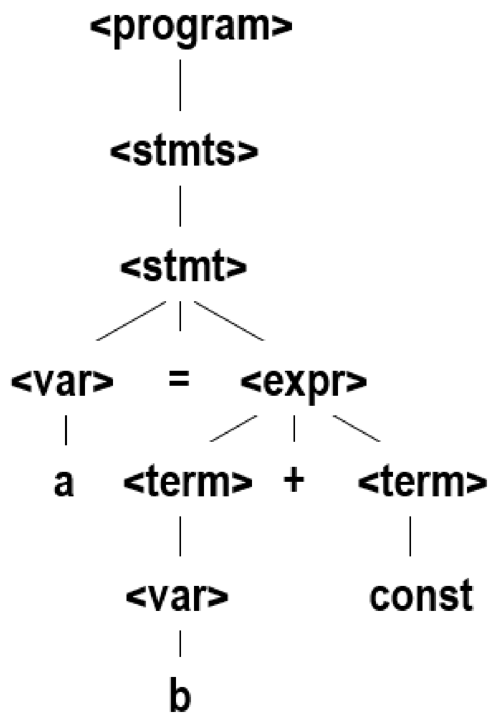
Parse Trees

- A hierarchical graphical representation of a derivation



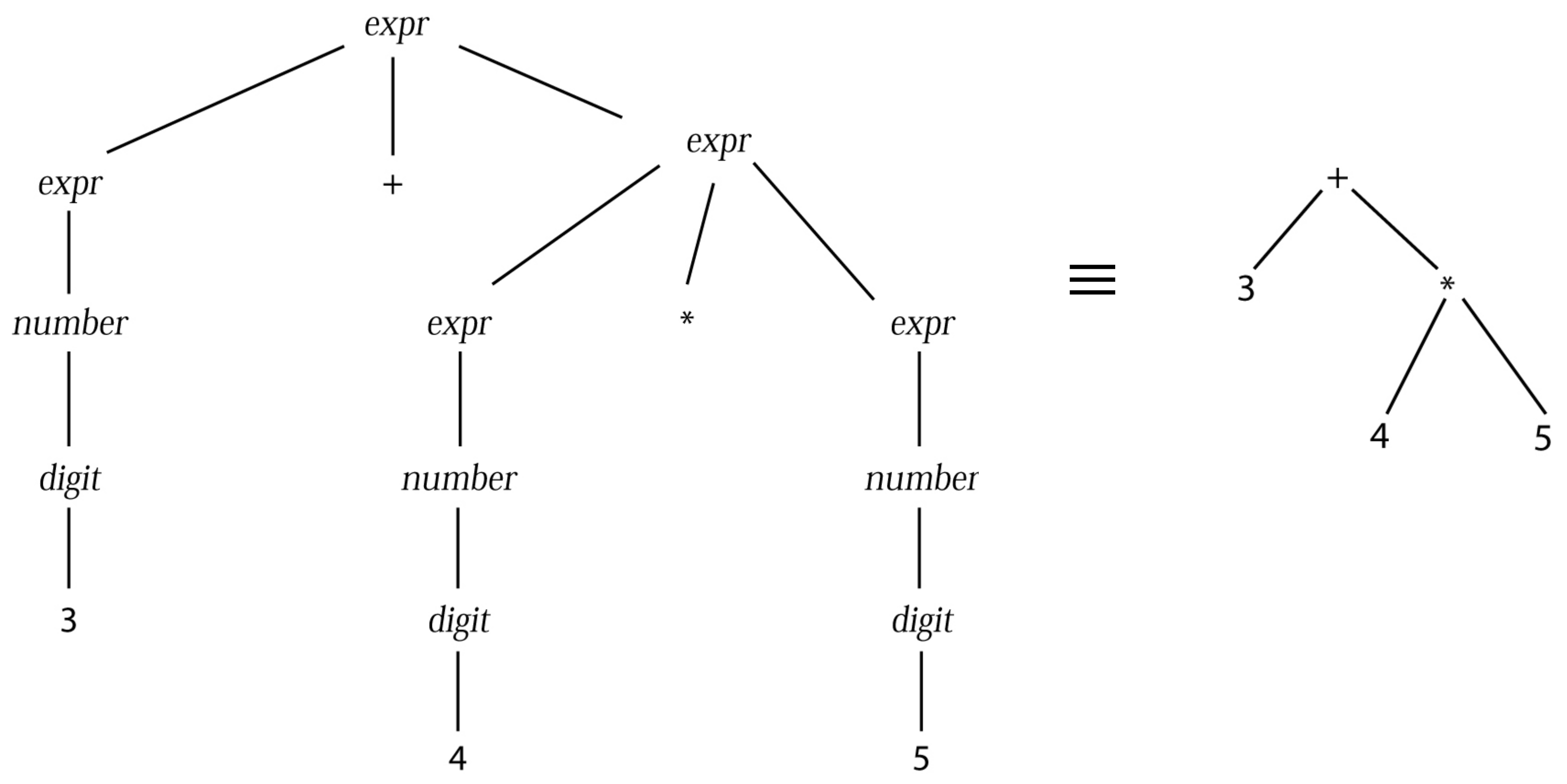
Parse Tree Represention of a Derivation

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



Abstract Syntax Tree

- Abstracting the essential structure of the tree



Ambiguity of Grammars

- A grammar that generates a sentential form for which there are 2 or more parse trees is said to be ambiguous
- expr $\rightarrow$ expr + expr | expr * expr | (expr) | number

  expr $\Rightarrow$ expr + expr

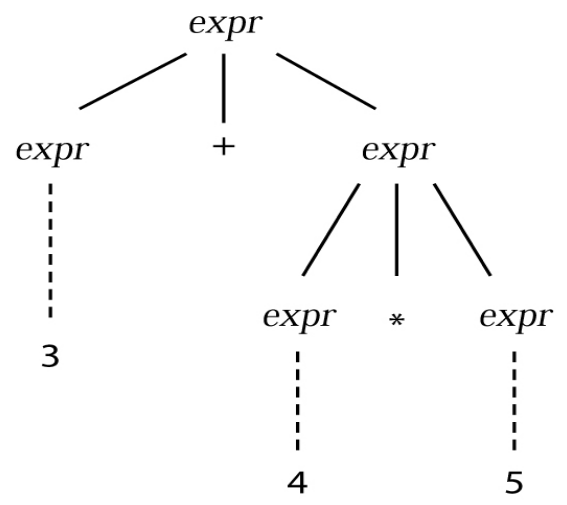
  $\Rightarrow$ expr + expr * expr (replace second expr with expr * expr)

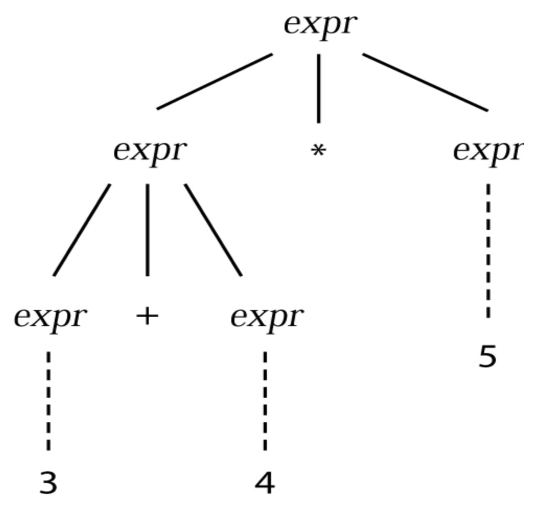
  $\Rightarrow$ number + expr * expr

  expr $\Rightarrow$ expr + expr

  $\Rightarrow$ expr + expr * expr (replace first expr with expr + expr)

  $\Rightarrow$ number + expr * expr





Leftmost Derivation

- Leftmost nonterminal is singled out for replacement at each step
- Each parse tree has a unique leftmost derivation which can be constructed by a preorder traversal of the tree
- Ambiguity can be detected by searching for more than one leftmost derivation of the same string

expr ⇒ expr + expr

⇒ number + expr

⇒ digit + expr * expr

⇒ 3 + expr

⇒ 3 + expr*expr

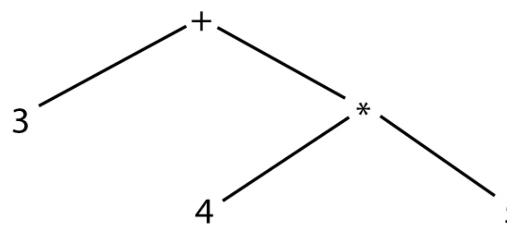
⇒ 3 + number*expr

expr ⇒ expr * expr

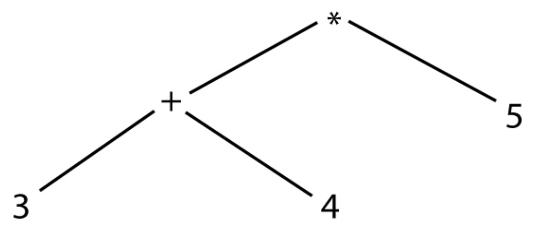
⇒ expr + expr * expr (replace first expr with expr + expr)

⇒ number + expr * expr

Tree 1



Tree 2

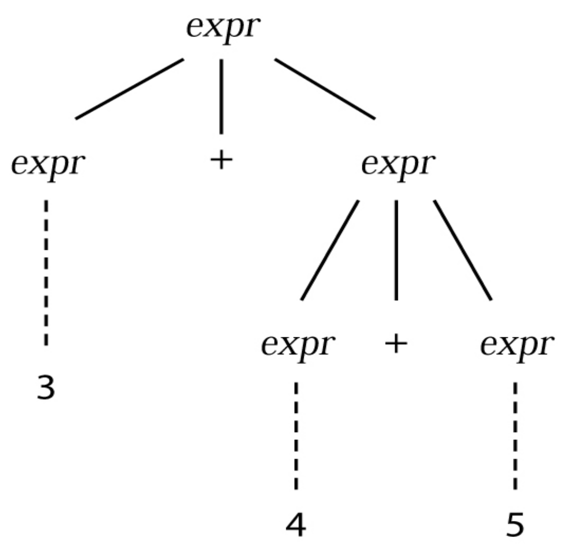


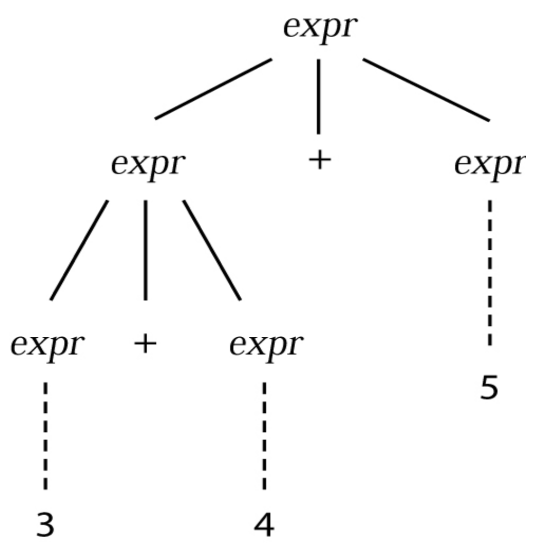
Disambiguaty Rule

- Which parse tree was the correct one from semantics point of view?
- Use the precedence rule as a disambiguating rule
- Rewrite the rule using a new grammar rule called introduces another nonterminal called term that establishes a precedence

Fixing Ambiguity using a New Nonterminal

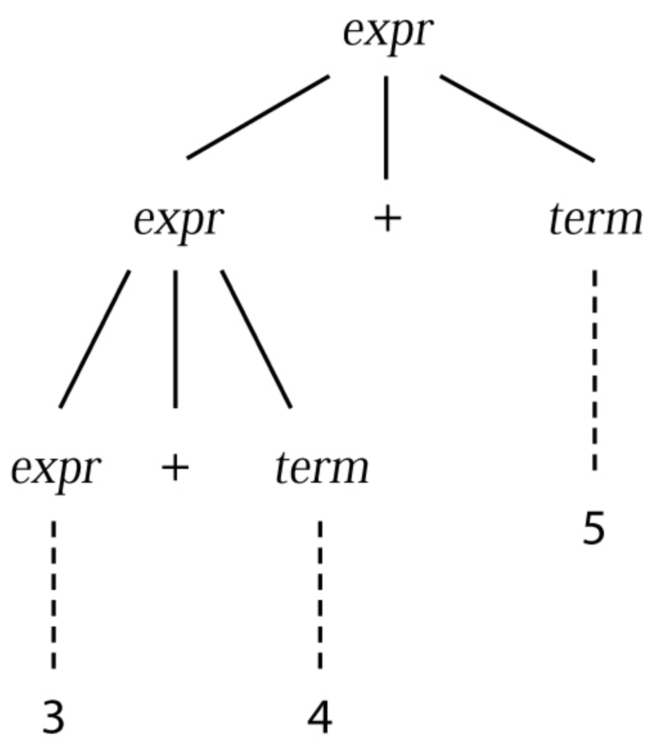
- Fixing ambiguity by precedence
  - expr $\rightarrow$ expr + expr | term
  - term $\rightarrow$ term * term | (expr) | number
- Still ambiguous 3 + (4 +5) or (3 + 4) + 5 (addition can be right or left associative

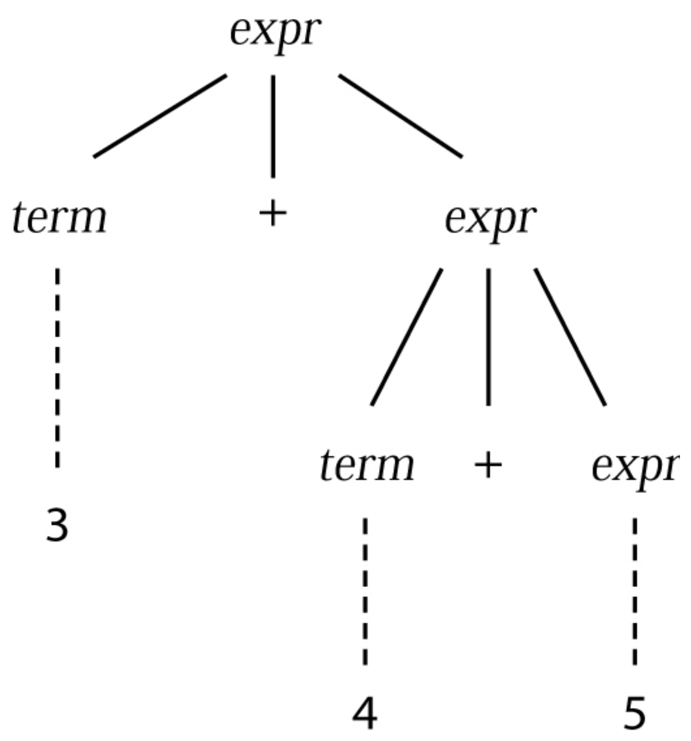




Disambiguity due to Associativity

- Ambiguity because of associativity
  - expr → expr + term (left recursive i.e. left associate) or
  - expr → term + expr (right recursive i.e right associate)





Revised Grammar

expr $\rightarrow$ expr + expr | expr * expr | (expr) | number

number $\rightarrow$ number digit | digit (note recursion here)

digit $\rightarrow$ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

- Revised grammar for Simple Integer Arithmetic Expressions (SIAE)

  expr $\rightarrow$ expr + term | term

  term $\rightarrow$ term * factor | factor

  factor $\rightarrow$ (expr) | number

  number $\rightarrow$ number digit | digit (note recursion here)

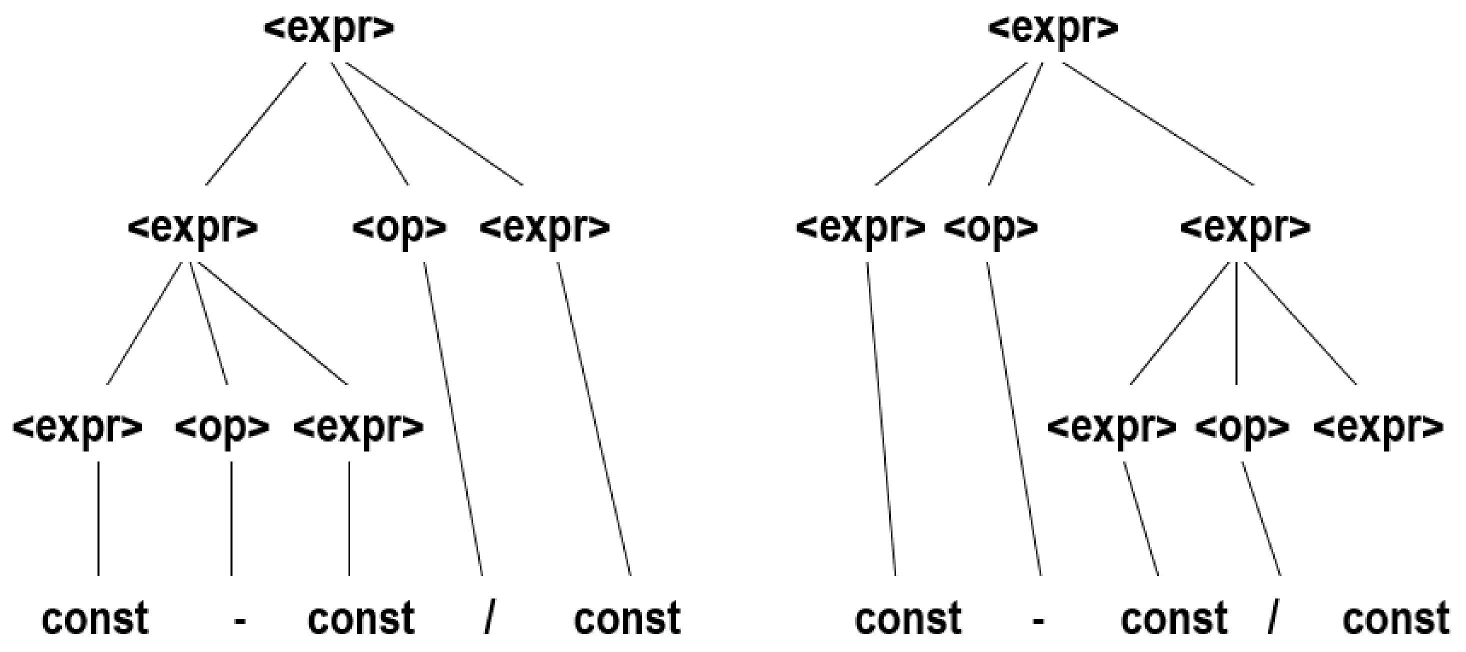
  digit $\rightarrow$ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

  Syntax trees generated by this BNF corresponds to the semantics of the arithmetic operations as they are usually defined

Is this Grammar Ambiguous?

- Is the following grammar ambiguous? Can you draw 2 parse trees?

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

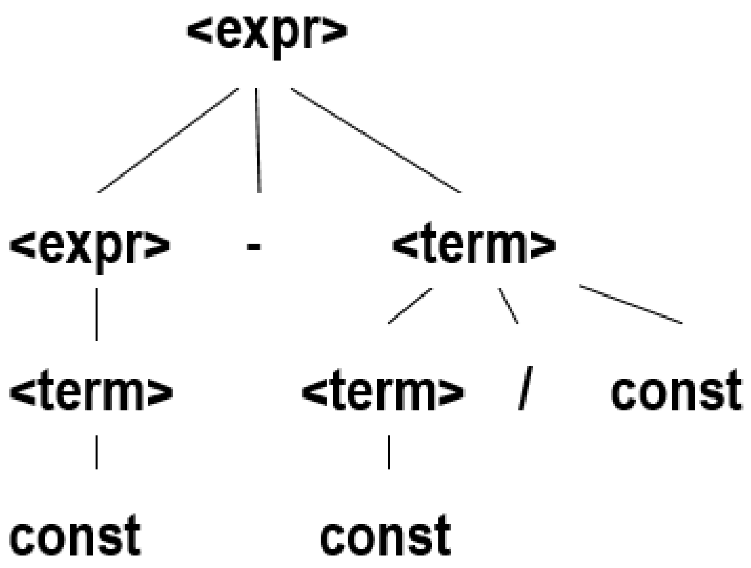


- How to remove ambiguity by adding another nonterminal ?

An Unambiguous Expression Grammar

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

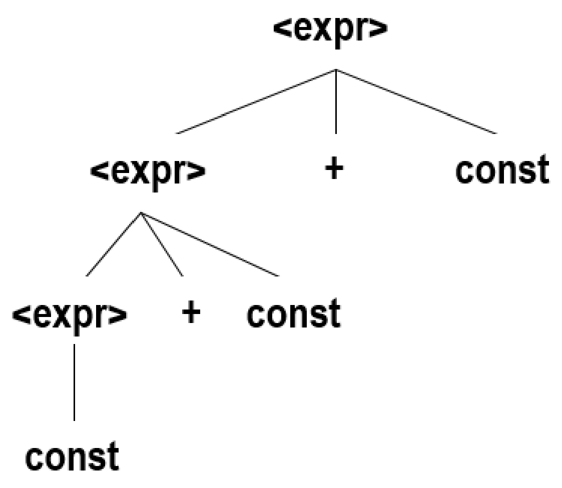
```
<expr> → <expr> - <term>  |  <term>
<term> → <term> / const| const
```



Associativity of Operators

- Operator associativity can also be indicated by a grammar

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

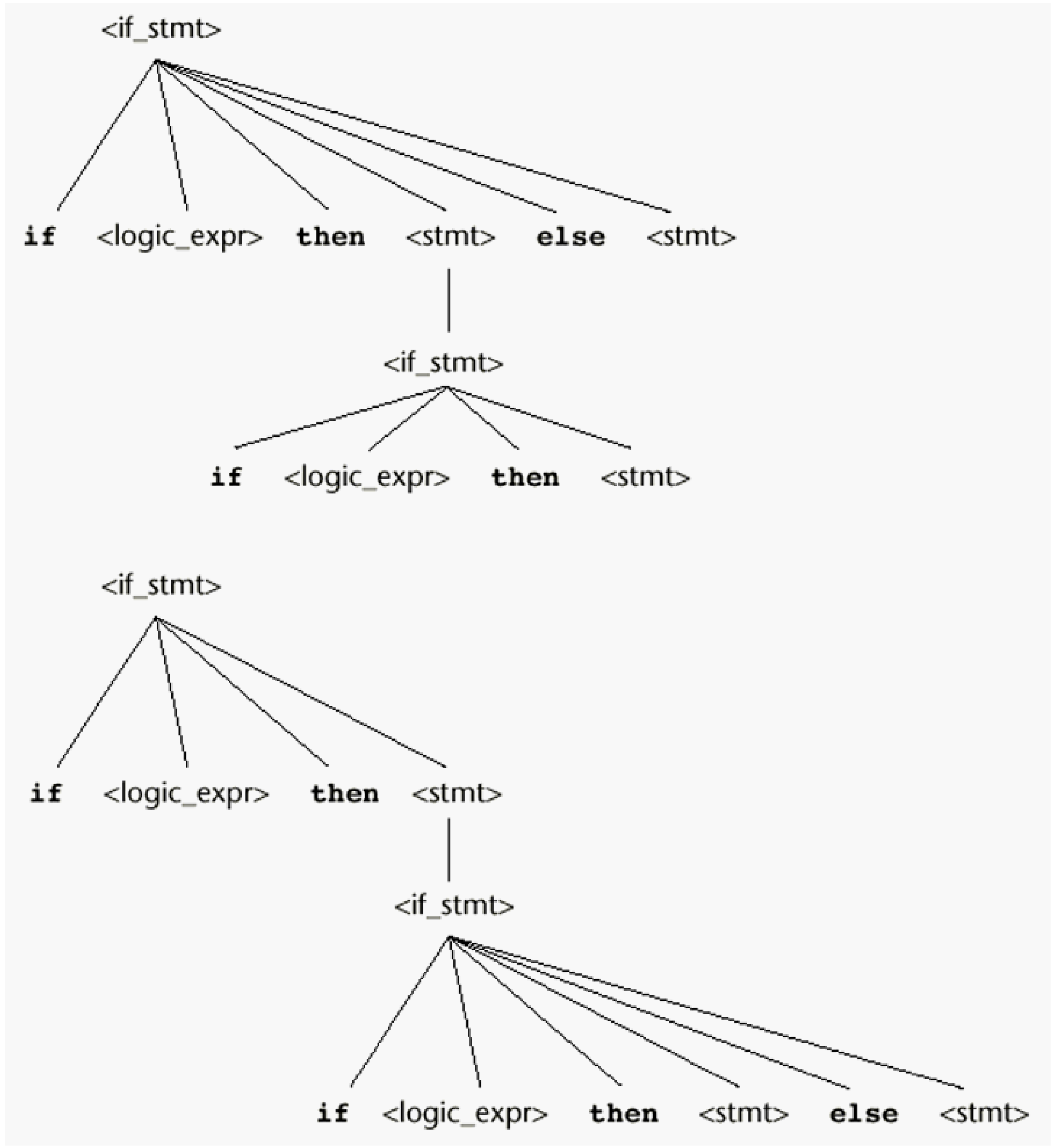


Ambiguous Grammar for Selector

- Java/C/C++ **`if-then-else`** grammar

```
<if_stmt> -> if (<logic_expr>) <stmt>
          | if (<logic_expr>) <stmt> else <stmt>
```

- Grammar above is ambiguous
- Show it using two parse trees for
    - if <logic_expr> then if <logic_expr> then <stmt> else <stmt>



Unambiguous Grammar for Selector

- Ambiguous!
  - An unambiguous grammar for if-then-else

```
<stmt> -> <matched> | <unmatched>
<matched> -> if (<logic_expr>) <stmt>
            | a non-if statement
<unmatched> -> if (<logic_expr>) <stmt>
             | if (<logic_expr>) <matched> else
                  <unmatched>
```

Extended BNF

- 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}
```

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

EBNF of SIAE

- EBNF grammar for Simple Integer Arithmetic Expressions (SIAE)

  expr $\rightarrow$ expr { + term}

  term $\rightarrow$ factor { * factor}

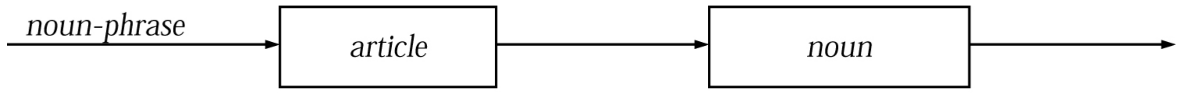
  factor $\rightarrow$ (expr) | number

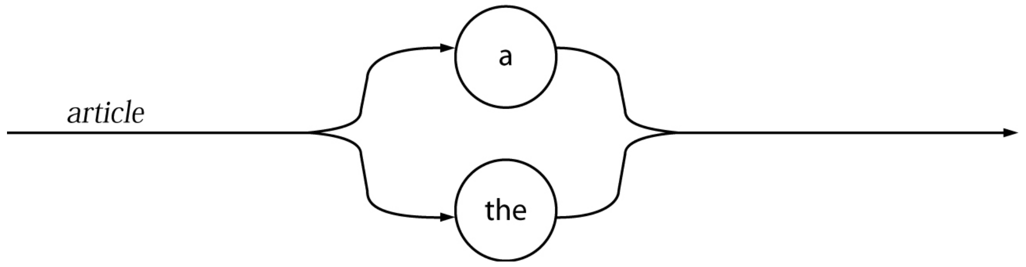
  number $\rightarrow$ digit { digit }   (note recursion here)

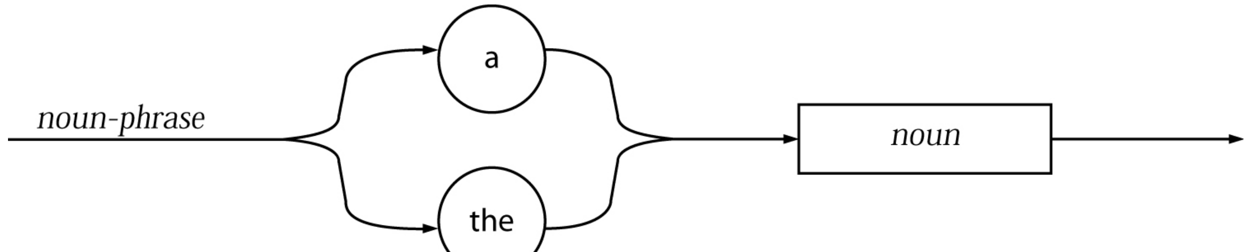
  digit $\rightarrow$ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Syntax Diagrams

- Another form of graphical representation of the grammar rule: use of circle for ? and square/rectangles for ?
- Used mainly in the past
- Very appealing visually but take up a lot of space
- Diagrams always derived from the EBNF notation







Syntax diagram for a SIAE

- Use of loops to exhibit repetitions

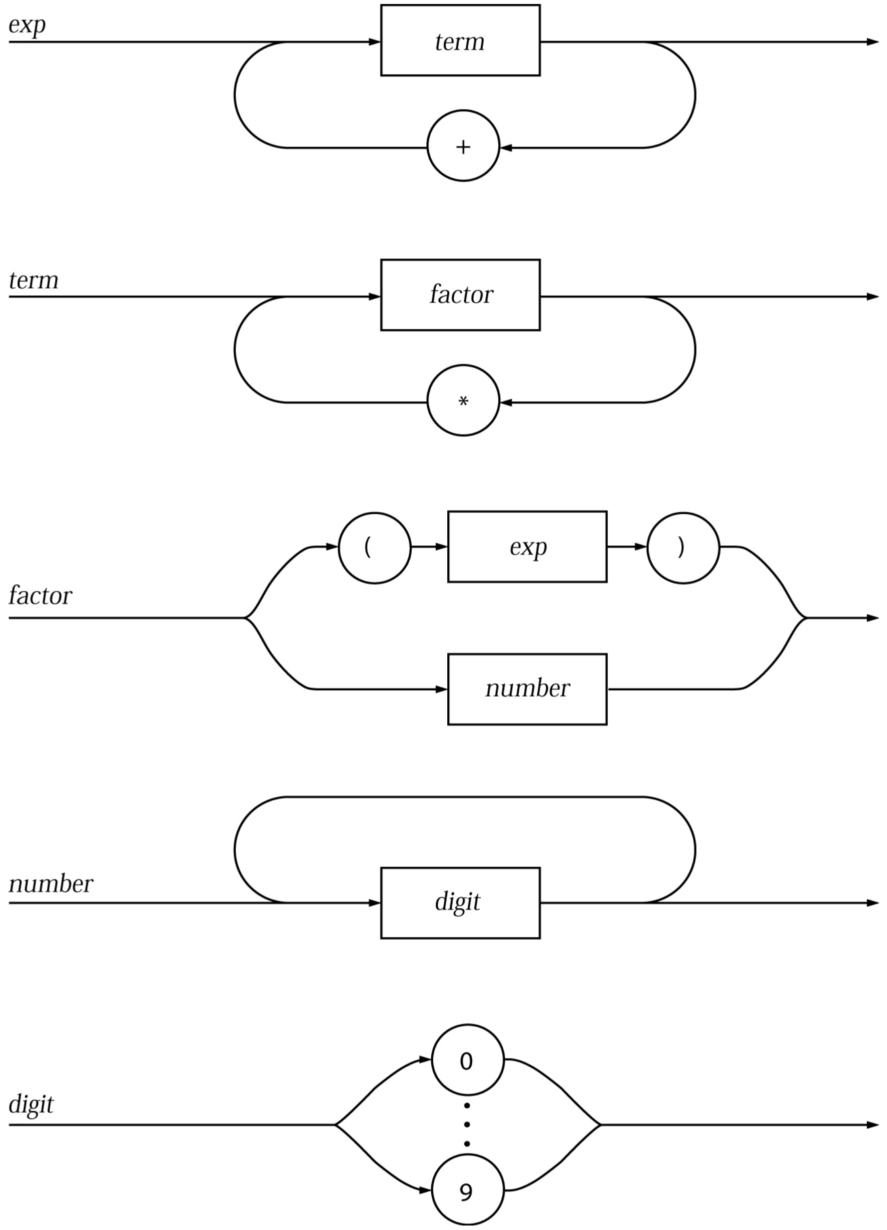
expr $\rightarrow$ expr { + term}

term $\rightarrow$ factor { * factor}

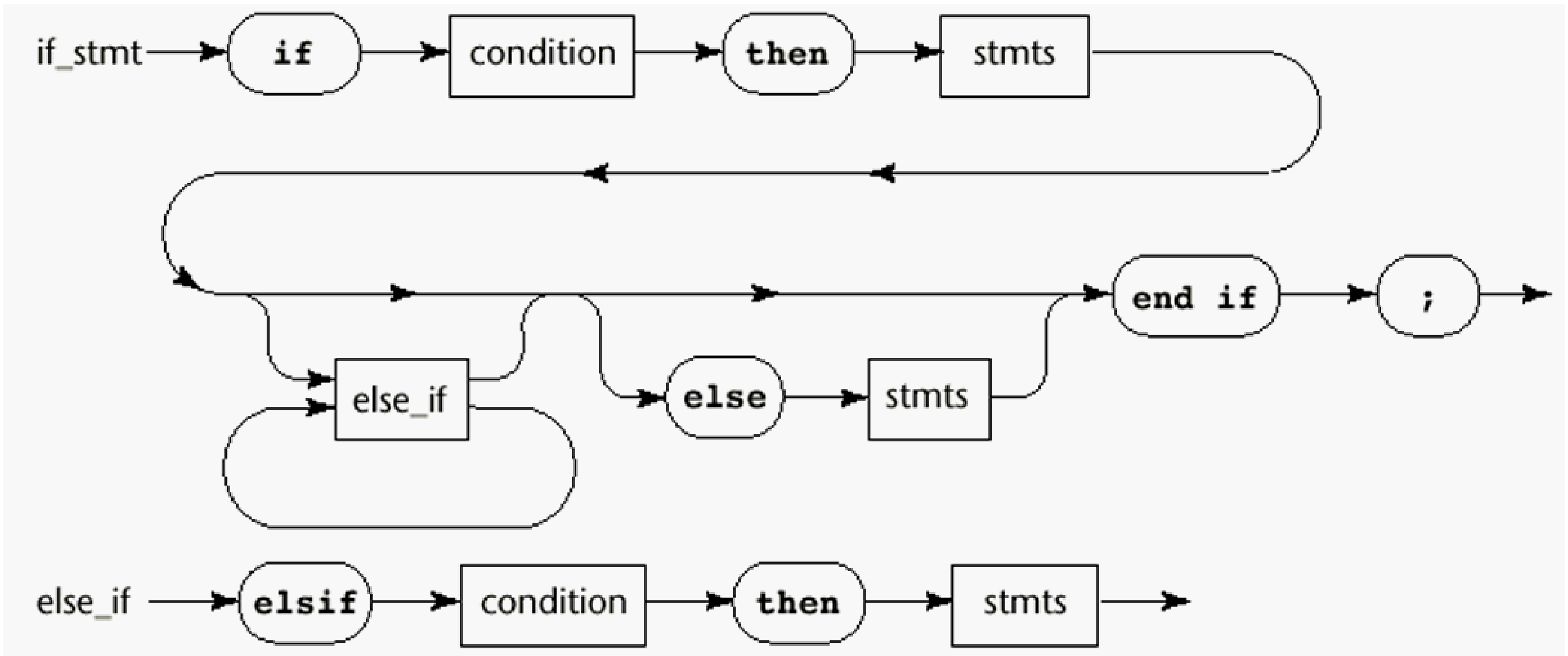
factor $\rightarrow$ (expr) | number

number $\rightarrow$ digit { digit }   (note recursion here)

digit $\rightarrow$ 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9



Syntax Graph and EBNF descriptions of if_stmt



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

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

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

Class Exercise 2

- Show a parse tree **and** left most derivation of the following statement:
  - A = (A + B) * C
  - A = B * (C * (A + B))
- Productions:

<assign> → <id> = <expr>

<id> → A | B | C

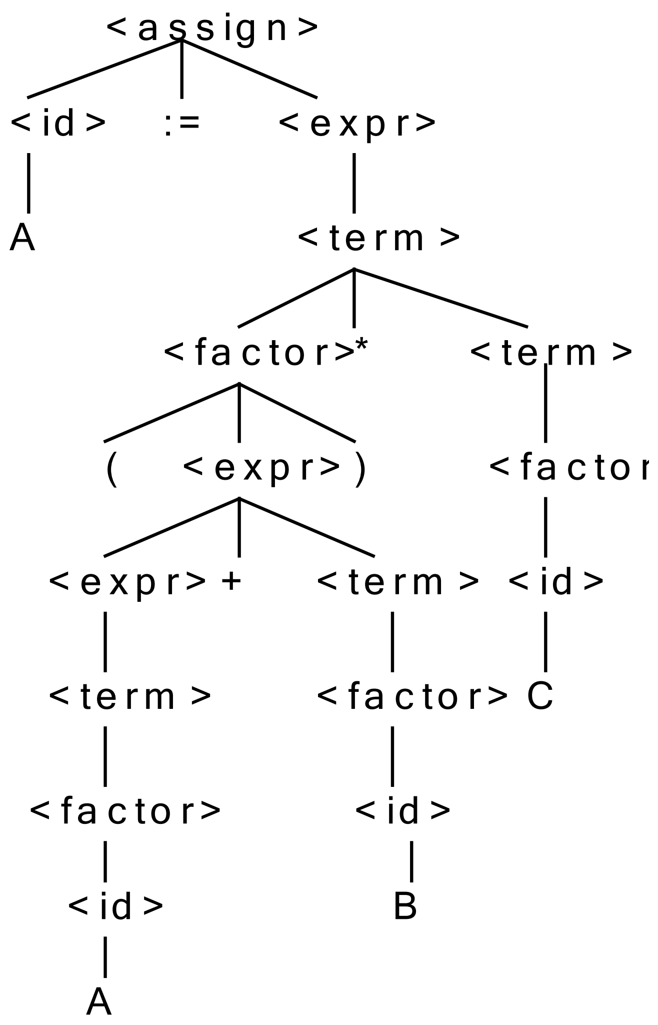
<expr> → <expr> + <term> | <term>

<term> → <term> * <factor> | <factor>

<factor> → (<expr>) | <id>

Derivation

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



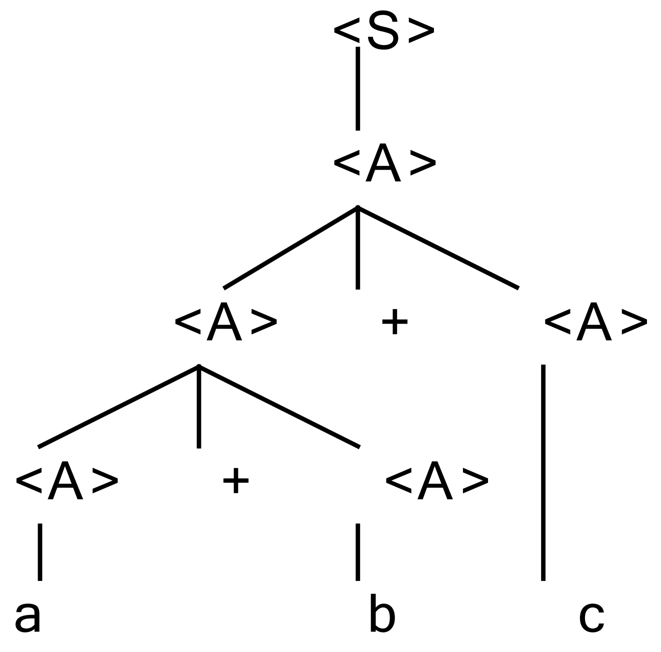
Class Exercise 3

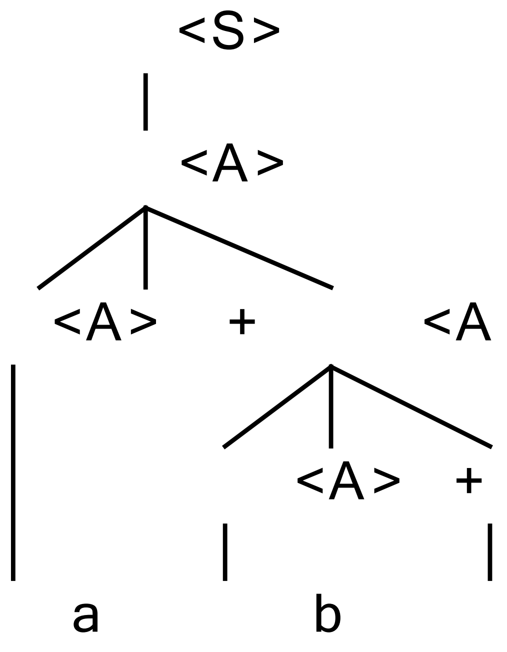
- Prove the following grammar is ambiguous by generating 2 parse trees

<S> $\rightarrow$ <A>

<A> $\rightarrow$ <A> + <A> | <id>

<id> $\rightarrow$ a | b | c





Take home question

- How would you add a production to fix the ambiguity?