BBM 301 – Programming Languages

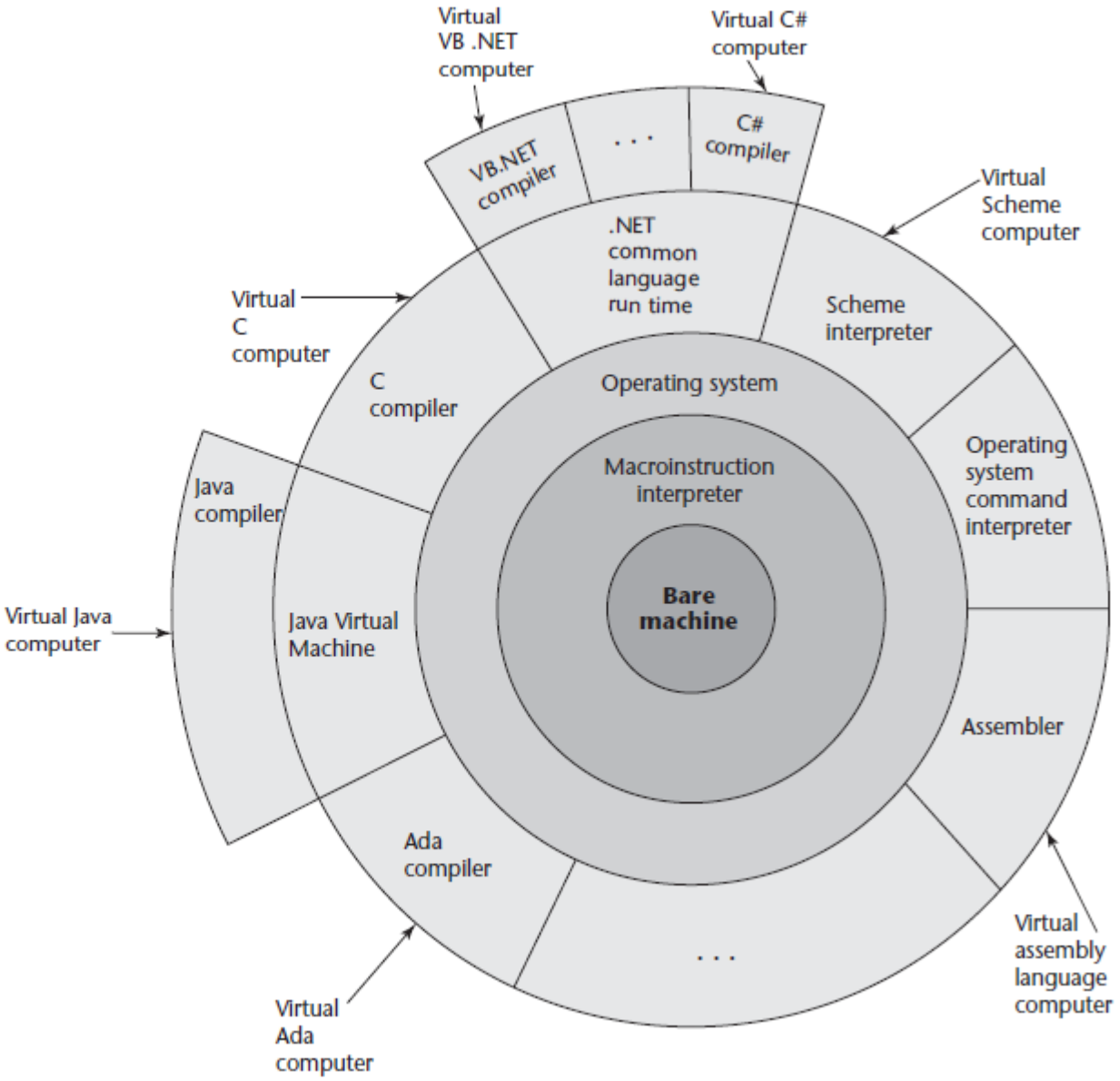
Fall 2020, Lecture 2

Today

- Implementation Methods
- Describing Syntax and Semantics (Chapter 3)

Figure 1.2

Layered interface of virtual computers, provided by a typical computer system

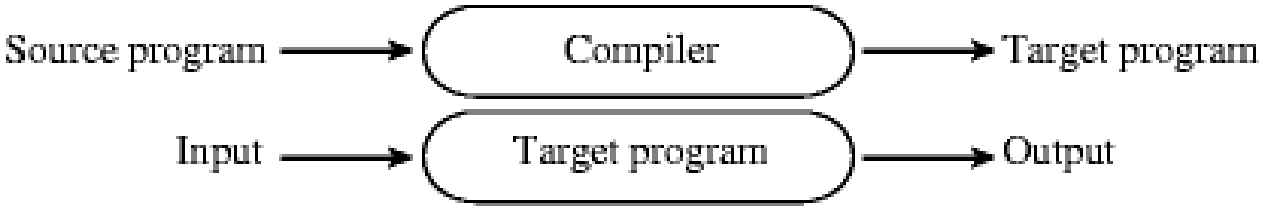


Implementation Methods

- Compilation
  - Programs are translated into machine language; includes JIT systems
  - Use: Large commercial applications
- Pure Interpretation
  - Programs are interpreted by another program known as an interpreter
  - Use: Small programs or when efficiency is not an issue
- Hybrid Implementation Systems
  - A compromise between compilers and pure interpreters
  - Use: Small and medium systems when efficiency is not the first concern

Compilation and Interpretation

- *Compiler* translates into target language (machine language), then goes away
- The target program is the locus of control at execution time



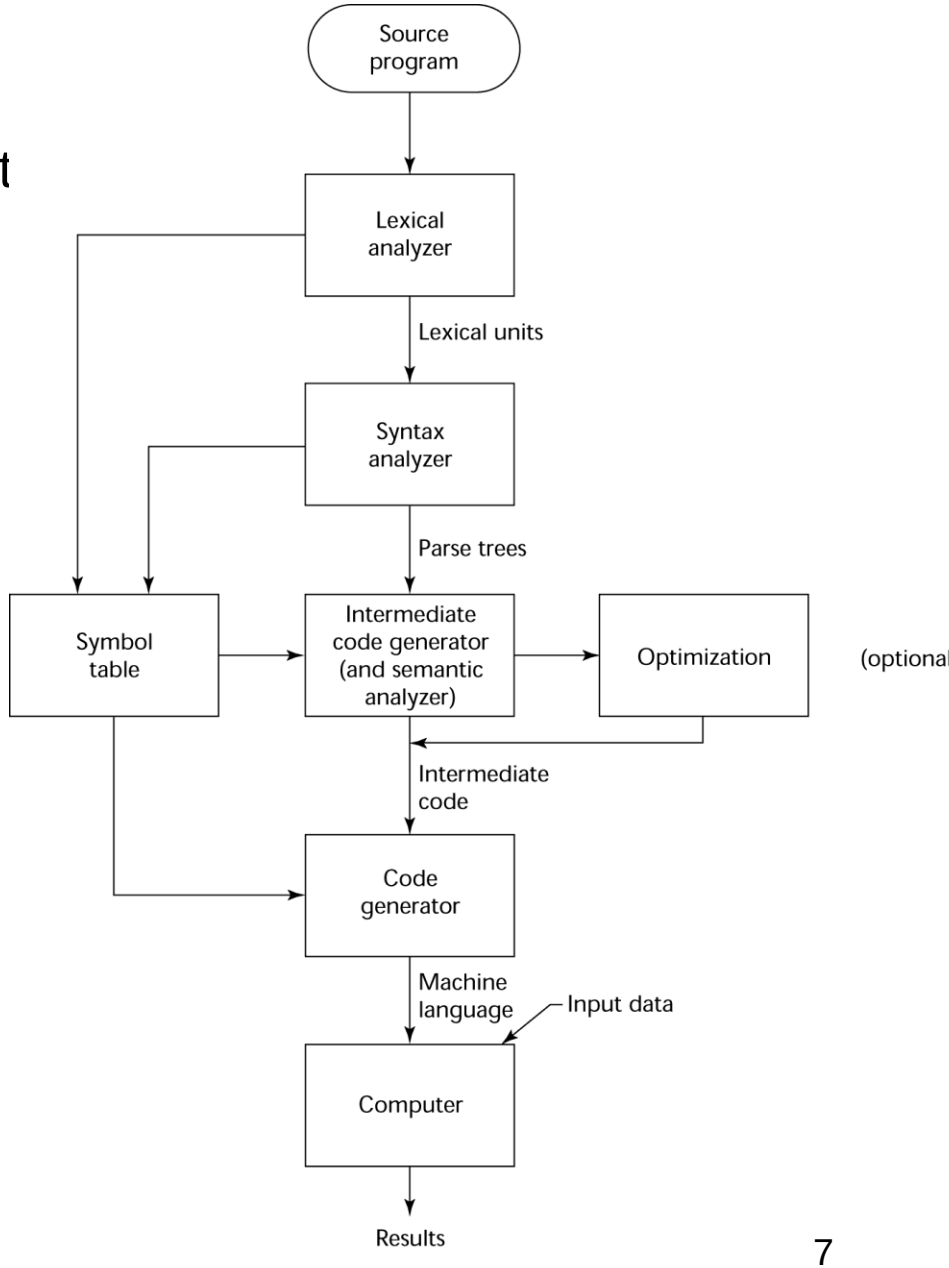
- *Interpreter* stays around at execution time
- Implements a virtual machine whose machine language is the high-level language

Compilation

- Translate high-level program (source language) into machine code (machine language)
- **Slow translation**, **fast execution**
- Compilation process has several phases:
  - lexical analysis: converts characters in the source program into lexical units
  - syntax analysis: transforms lexical units into *parse trees* which represent the syntactic structure of program
  - Semantics analysis: generate intermediate code
  - code generation: machine code is generated
- Example: C, C++, COBOL, Ada

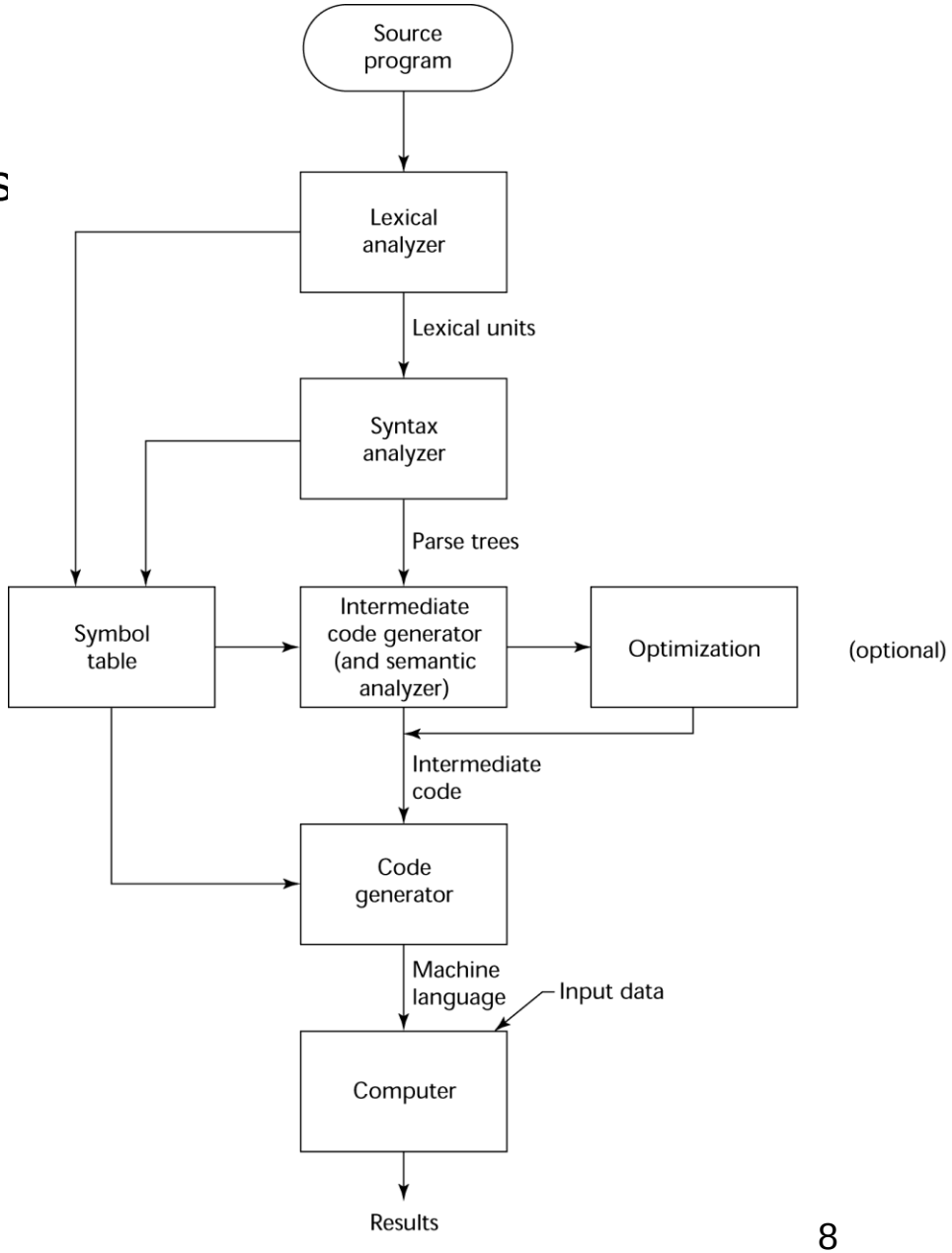
Compilation

- **Source language:** The language that the compiler translates
- **Lexical analyzer:** gathers the characters of the source program into lexical units (e.g. identifiers, special words, operators, punctuation symbols) Ignores the comments
- **Syntax analyzer:** takes the lexical units, and use them to construct parse trees, which represent the syntactic structure of the program
- **Intermediate code generator:** Produces a program at an intermediate level. Similar to assembly languages



Compilation

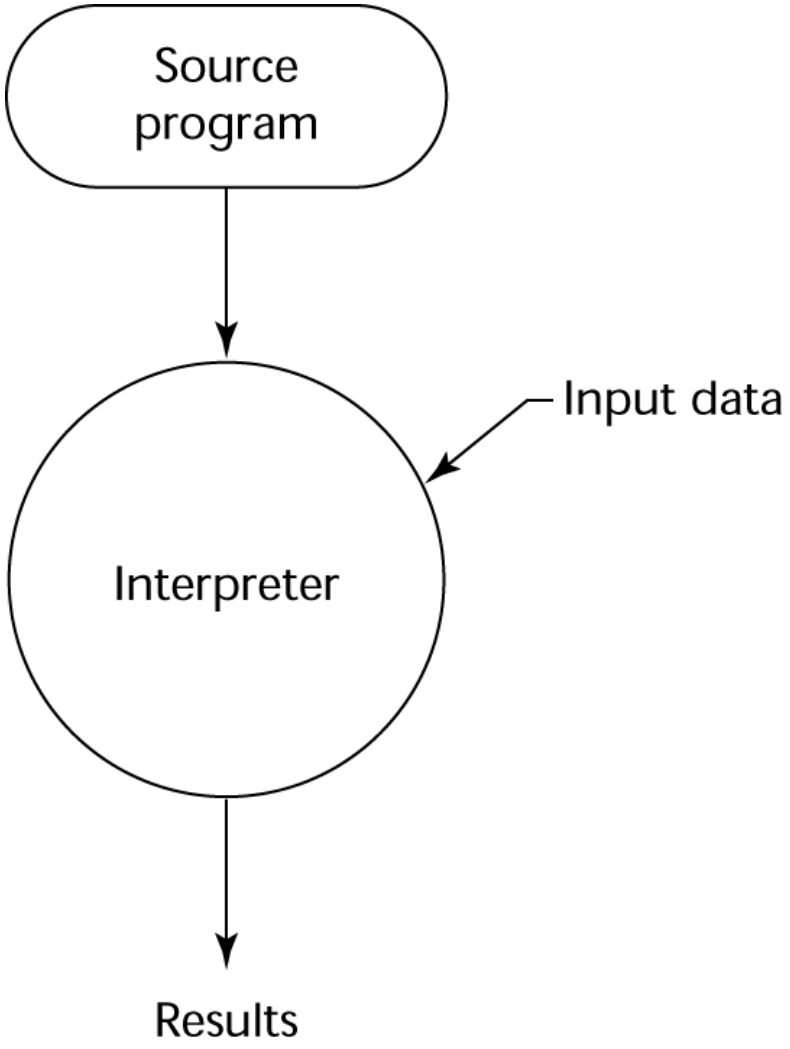
- **Semantic analyzer:** checks for errors that are difficult to check during syntax analysis, such as type errors.
- **Optimization (optional):** Used if execution speed is more important than compilation speed.
- **Code generator:** Translates the intermediate code to machine language program.
- **Symbol table:** serves as a database of type and attribute information of each user defined name in a program. Placed by lexical and syntax analyzers, and used by semantic analyzer and code generator.



Pure Interpretation

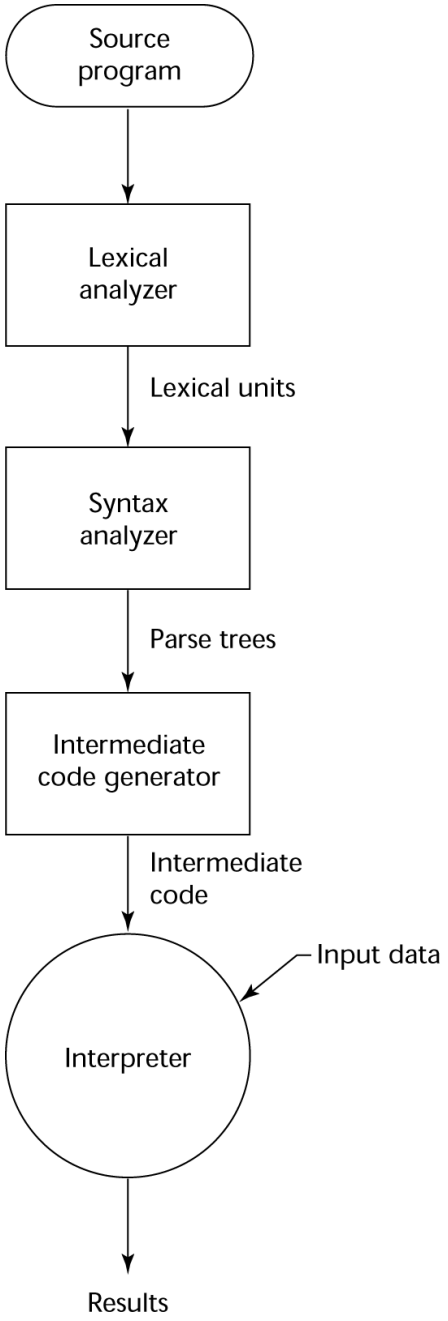
- No translation.
- Each statement is decoded and executed individually.
- The interpreter program acts as a software simulation of a machine
- **Advantages:**
  - Easier implementation of programs
  - Debugging is easy (run-time errors can easily and immediately be displayed)
- **Disadvantages:**
  - Slower execution (10 to 100 times slower than compiled programs, and statement decoding is the major bottleneck)
  - Often requires more space
- Now rare for traditional high-level languages but available in some Web scripting languages (e.g., JavaScript, PHP)

Pure Interpretation Process



Hybrid Implementation Systems

- A compromise between compilers and pure interpreters
- A high-level language program is translated to an intermediate language that allows easy interpretation
- Faster than pure interpretation
  - Example: Perl programs are partially compiled to detect errors before interpretation



Today

- Implementation Methods
- Describing Syntax and Semantics (Chapter 3)

Creating computer programs

- Each programming language provides **a set of primitive operations**
- Each programming language provides **mechanisms for combining primitives** to form more complex, but legal, expressions
- Each programming language provides **mechanisms for deducing meanings** or values associated with computations or expressions

Aspects of languages

- Primitive constructs
  - ***Programming language***: numbers, strings, simple operators
  - ***English*** : words
- **Syntax**– which strings of characters and symbols are well-formed
  - ***Programming language****:* 3.2 + 3.2 is a valid C expression
  - ***English****:* “cat dog boy” is not syntactically valid, as not in form of acceptable sentence

Aspects of languages

- Static semantics – which syntactically valid strings have a meaning
  - English – "**I are big**" has form <noun> <intransitive verb> <noun>, so syntactically valid, but is not valid English because "I" is singular, "are" is plural
  - Programming language – for example, <literal> <operator> <literal> is a valid syntactic form, but **2.3/"abc"** is a static semantic error!

Aspects of languages

- **Semantics** – what is the meaning associated with a syntactically correct string of symbols with no static semantic errors
  - English – can be ambiguous
    - "I cannot praise this student too highly"
    - "Yaşlı adamın yüzüne dalgın dalgın baktı."
  - Programming languages – always has exactly one meaning
    - But meaning (or value) may not be what programmer intended

Today

- **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)

Example: Syntax and Semantics

- `while` statement in Java
- **syntax:** `while` **(*<boolean_expr>*)**
  <*statement*>
- **semantics:** when ***boolean_expr*** is true, *statement* will be executed

- *The meaning of a statement should be clear from its syntax (Why?)*

Describing Syntax: Terminology

- ***Alphabet:*** Σ, All strings: Σ*
- A ***sentence*** is a string of characters over some alphabet
- A ***language*** is a set of sentences, L ⊆ Σ*

Describing Syntax: Terminology

- A ***language*** is a set of sentences
  - Natural languages: English, Turkish, …
  - Programming languages: C, Fortran, Java,…
  - Formal languages: a*b*, 0^n1^n
- String of the language:
  - Sentences
  - Program statements
  - Words (aaaaabb, 000111)

Lexemes

- A ***lexeme*** is the lowest level syntactic unit of a language (e.g., **`*`**, **`sum`**, **`begin`**)
- Lower level constructs are given not by the syntax but by lexical specifications.
- Examples: identifiers, constants, operators, special words.

  `total`, **`sum_of_products`**, **`1254`**, **`++`**, **`(`** **`:`**

- So, a language is considered as **a set of strings of lexemes** rather than strings of chars.

Token

- A ***token*** of a language is a category of lexemes
- For example, ***identifier*** is a token which may have lexemes, or instances, `sum` and `total`

Example in Java Language

```
x = (y+3.1) * z_5 ;
```

```
x = (y+3.1) * z_5;
```

Lexemes	*Tokens*
x	identifier
=	equal_sign
(	left_paren
)	right_paren
for	for
y	identifier
+	plus_op
3.1	float_literal
*	mult_op
z_5	identifier
;	semi_colon

Describing Syntax

- *Higher level constructs are given by syntax rules.*
- ***Syntax rules*** specify which strings from Σ* are in the language
- Examples: organization of the program, loop structures, assignment, expressions, subprogram definitions, and calls.

Elements of Syntax

- An alphabet of symbols
- Symbols are terminal and non-terminal
  - Terminals cannot be broken down
  - Non-terminals can be broken down further
- Grammar rules that express how symbols are combined to make legal sentences
- Rules are of the general form

```
non-terminal symbol ::= list of zero or more
terminals or non-terminals
```

- One uses rules to recognize (parse) or generate legal sentences

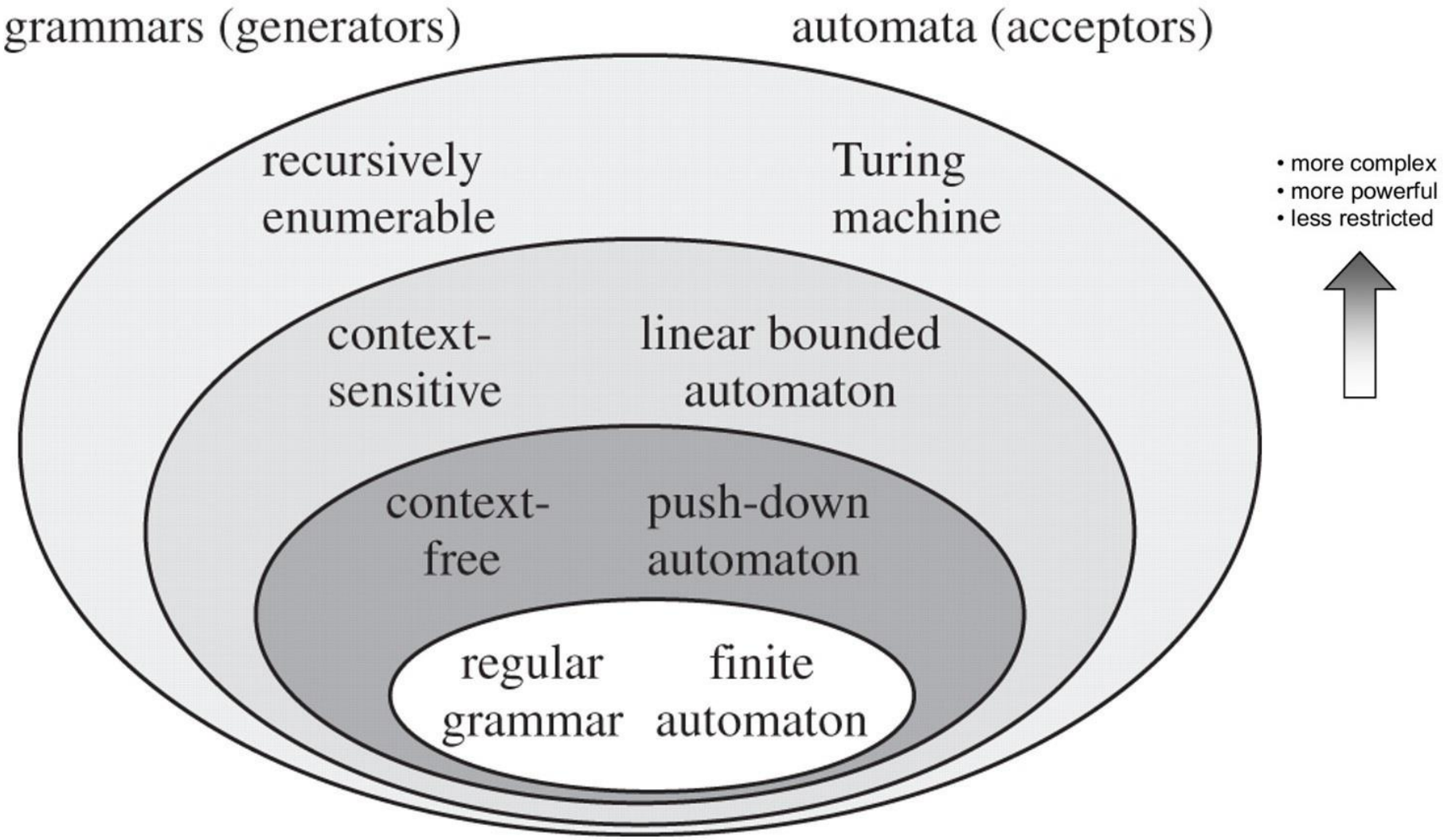
Formal Definition of Languages

- **Recognizers**
  - A recognition device reads input strings over the alphabet of the language and decides whether the input strings belong to the language
  - Example: syntax analysis part of a compiler

- **Generators**
  - A device that generates sentences of a language
  - One can determine if the syntax of a particular sentence is syntactically correct by comparing it to the structure of the generator

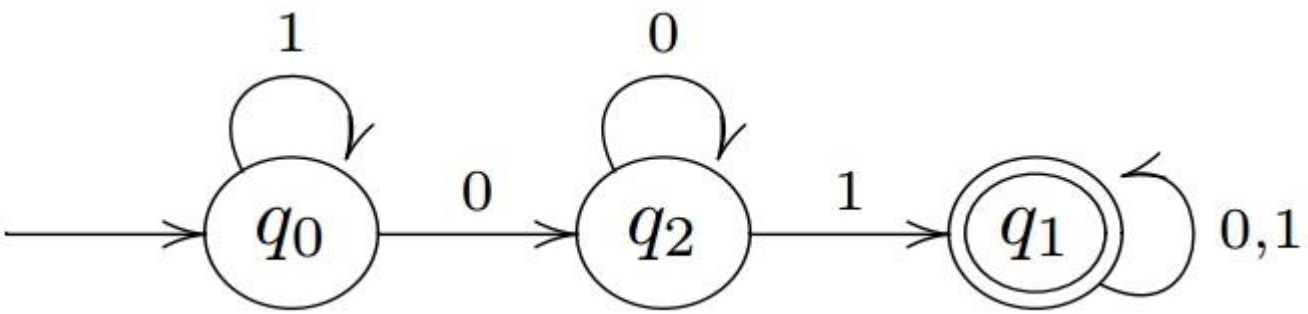
Formal Methods of Describing Syntax

- **Grammars: formal language-generation mechanisms.**
- In the mid-1950s, Chomsky, described four classes of generative devices (or grammars) that define four classes of languages.
  - Two of these grammar classes, named context-free and regular, turned out to be useful for describing the syntax of programming languages.
  - Regular grammars: The forms of the tokens of programming languages
  - Context-free grammars: The syntax of whole programming


grammars (generators)
automata (acceptors)
recursively enumerable
Turing machine
context-sensitive
linear bounded automaton
context-free
push-down automaton
regular grammar
finite automaton
• more complex
• more powerful
• less restricted

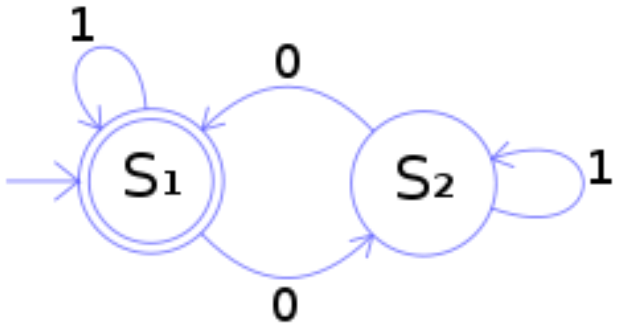
Deterministic Finite Automata

$\Sigma = \{0,1\}$

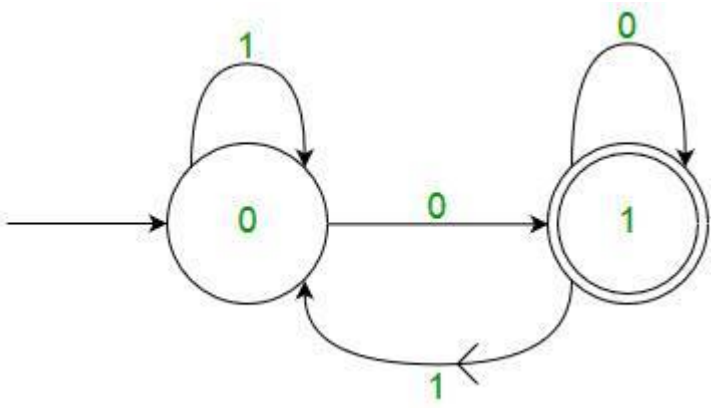


1110000100000
0000000000001
01
0000
111111

There should be at least one “01” in anywhere of the input.



the input contains an even number of 0



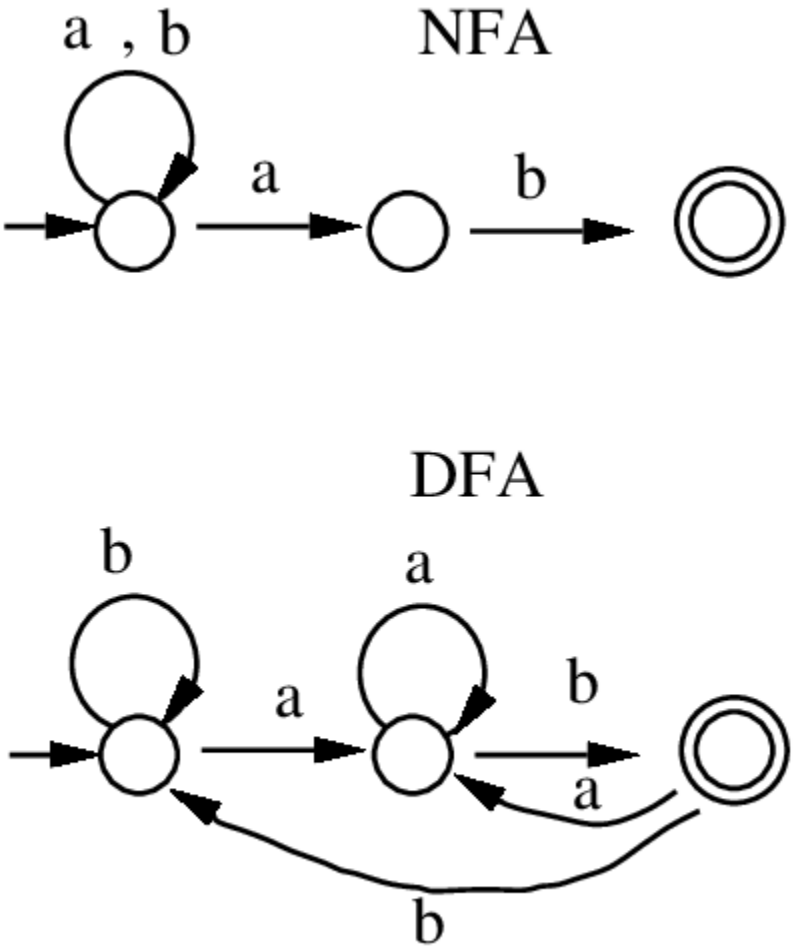
?

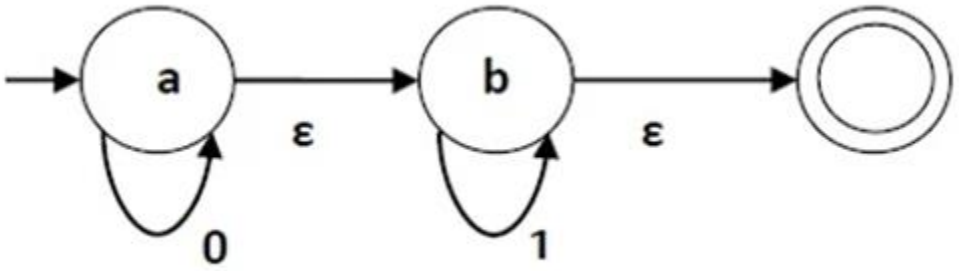
Non deterministic Finite Automata

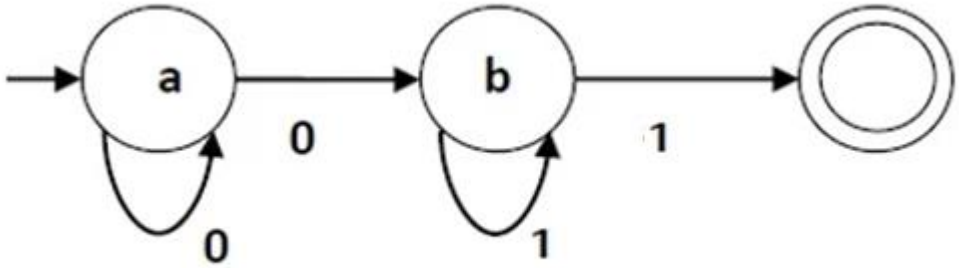
a DFA can only have one transition for each symbol going outwards from the state. But, an NFA can have multiple transitions for a symbol from the same state

an NFA is not required to have a transition for each symbol.

NFA can have a transition for an empty string.




a
b
ε
ε
0
1


a
b
0
1
0
1

Regular Languages

- Tokens can be generated using three formal rules
  - Concatenation
  - Alternation (|)
  - Kleene closure (repetition an arbitrary number of times)(*)

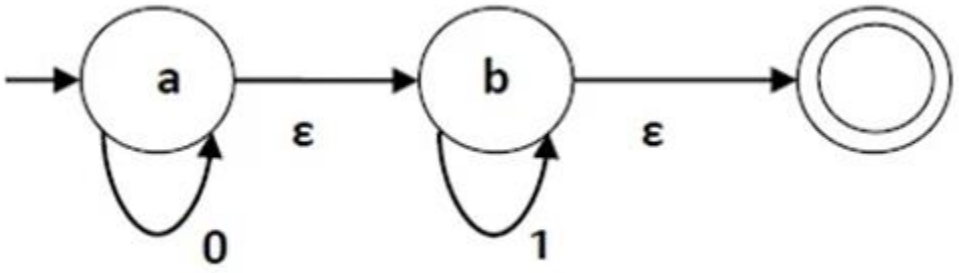
- Any sets of strings that can be defined by these three rules is called **a regular set** or a **regular language**

Regular expressions

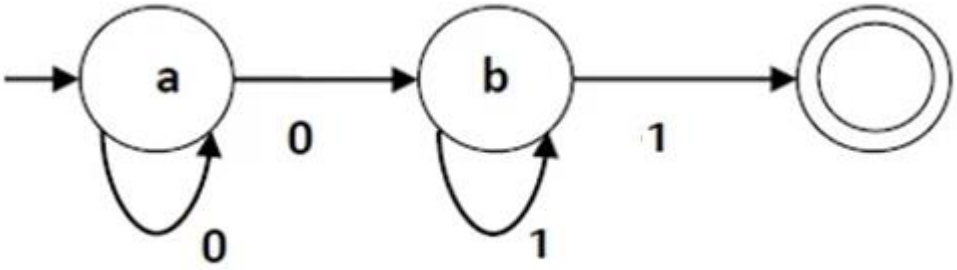
| means OR

- Means 0 OR more

0 | 1	L = {0,1}
(0 | 1) (0 | 1)	L = {00, 01, 10, 11}
0 (1 | 0)	L = {01, 00}
0*	L = {empty, 0, 00, 000, 0000….}
(0 | 10) *	L = {0, 010, 00000, 1010, 1000, 0010…}



0*1*



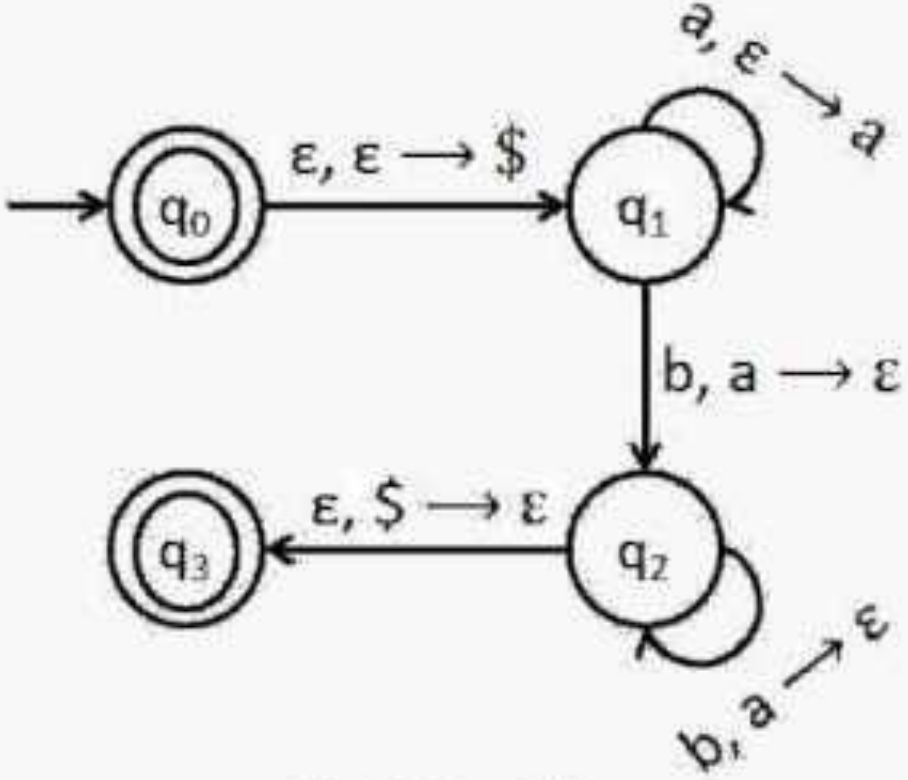
0*01*1
0+1+
+ means 1 OR more

Is the following language regular?

L = {number of 0s followed by equal number of 1s}

L = {$0^n 1^n$, n>=0}

A PDA for $\{a^n b^n : n \geq 0\}$



Context Free Grammars

L = {0nbn, n>=0 }

S -> aSb | empty

L = {0nbn, n>=1 }

S -> aSb | ab

Context sensitive grammar

$S \rightarrow aSBE$ $S \rightarrow aBE$ $eB \rightarrow eE$

$aB \rightarrow ab$ $bB \rightarrow bb$ $bE \rightarrow be$ $eE \rightarrow ee$

Context free grammar

$S \rightarrow AB$ $A \rightarrow BS \mid 0$ $B \rightarrow SA \mid 1$

Regular grammar

$S \rightarrow 0A|1B|0$ $A \rightarrow 0A|1B|0S$ $B \rightarrow 1B|1|0$

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 the class of context-free languages
  - Programming languages are contained in the class of CFL's.

Backus-Naur Form (BNF)

- A notation to describe the syntax of programming languages.
- Named after
  - John Backus – Algol 58
  - Peter Naur – Algol 60
- A ***metalanguage*** is a language used to describe another language.
- **BNF is a metalanguage used to describe PLs.**

BNF Fundamentals

- BNF uses abstractions for syntactic structures.

  <LHS> → <RHS>
- LHS: abstraction being defined
- RHS: definition
- “→” means “can have the form”
- Sometimes ::= is used for →

BNF Fundamentals

- Example, Java ***assignment*** statement can be represented by the abstraction **<assign>**
- **<assign>** → **<var> = <expression>**
- This is a ***rule*** or ***production***
- Here, **<var>** and **<expression>** must also be defined.
- example instances of this abstraction can be

```
total = sub1 + sub2
myVar = 4
```

BNF Fundamentals

- These abstractions are called **Variables** or **Nonterminals** of a Grammar.
- Lexemes and tokens are the **Terminals** of a grammar.
- Nonterminals are often enclosed in angle brackets
- Examples of BNF rules:

<ident_list> → **identifier | identifier, <ident_list>**

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

BNF Fundamentals

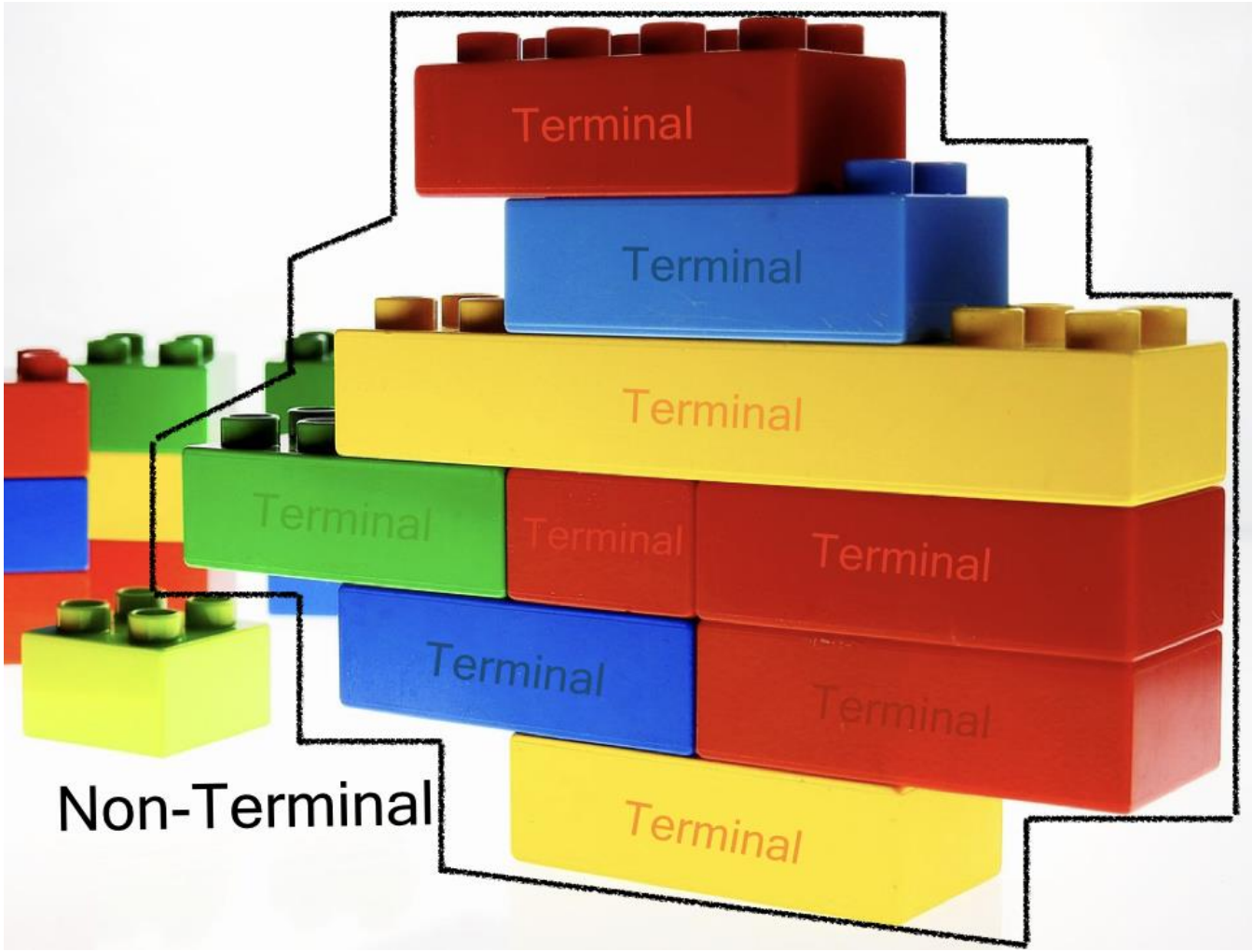
- A formal definition of ***rule***:

  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

  <LHS> → <RHS>

- **Grammar**: a finite non-empty set of rules

Additional Notes on Terminals and NonTerminals



Source: https://tomassetti.me/ebnf/

Additional Notes on Terminals and NonTerminals

- Terminals are the smallest block we consider in our grammars.
- A terminal could be either:
  - a quoted literal
  - a regular expression
  - a name referring to a terminal definition.

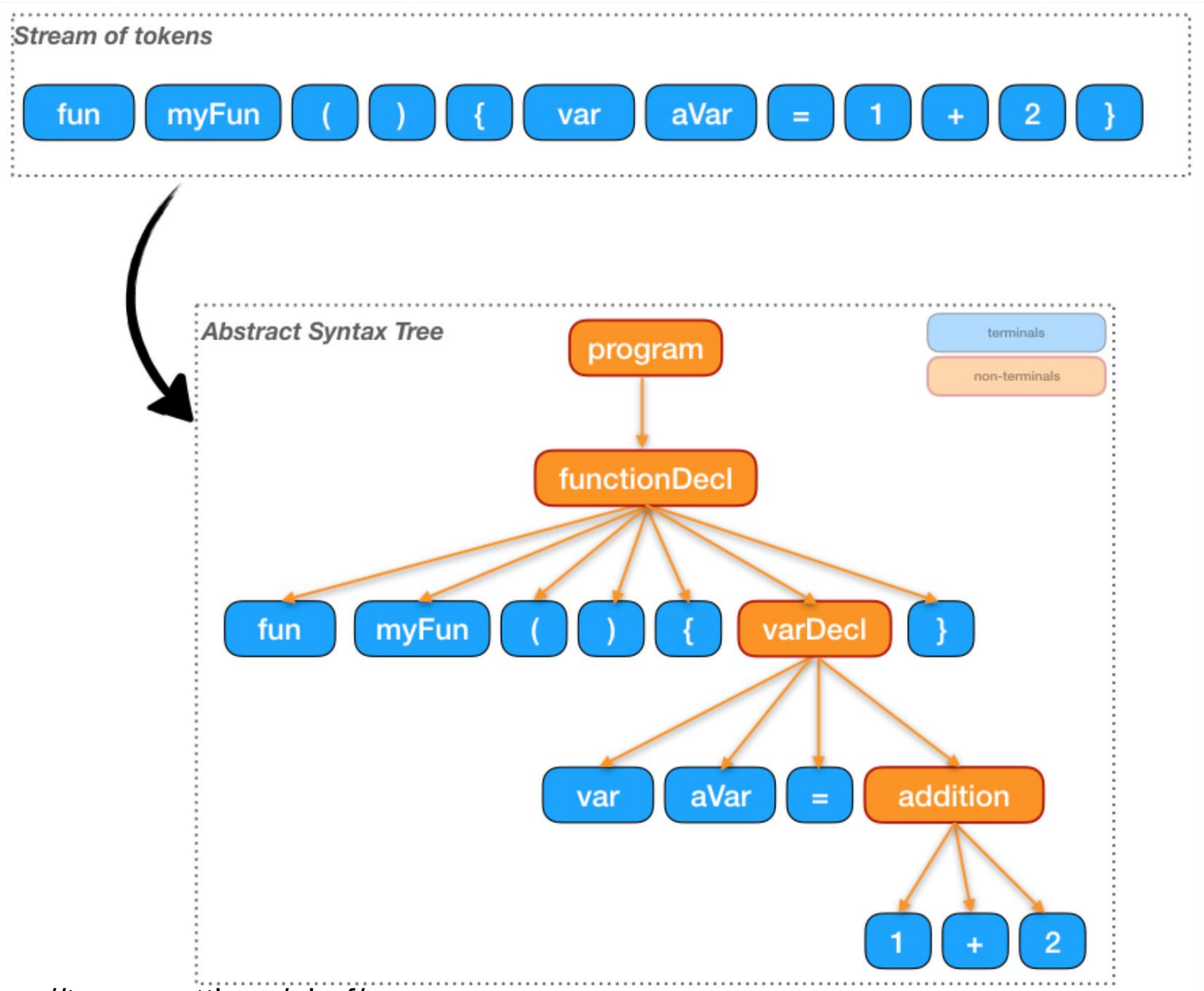
Terminals

- Let's see some typical terminals:
  - *identifiers:* these are the names used for variables, classes, functions, methods and so on.
  - *keywords:* almost every language uses keywords. They are exact strings that are used to indicate the start of a definition (think about class in Java or def in Python), a modifier (public, private, static, final, etc.) or control flow structures (while, for, until, etc.)

Terminals

- *literals:* these permit to define values in our languages. We can have string literals, numeric literal, char literals, boolean literals (but we could consider them keywords as well), array literals, map literals, and more, depending on the language
- *separators and delimiters:* like colons, semicolons, commas, parenthesis, brackets, braces
- *whitespaces:* spaces, tabs, newlines.
- *comments*

Terminals and Non-terminals



Source: https://tomassetti.me/ebnf/

Non-terminals

- Examples of non-terminals are:
  - *program/document*: represent the entire file
  - *module/classes:* group several declarations togethers
  - *functions/methods:* group statements together
  - *statements:* these are the single instructions. Some of them can contain other statements. Example : loops
  - *expressions:* are typically used within statements and can be composed in various ways

Examples

An initial example

- Consider the sentence **"Mary greets John"**

- A simple grammar

  <sentence> ::= <subject><predicate>

  <subject> ::= Mary

  <predicate> ::= <verb><object>

  <verb> ::= greets

  <object> ::= John

Alternations

- Multiple definitions can be separated by | (OR).

  <object> ::= John | Alfred

- This adds "Mary greets Alfred" to legal sentences

  <subject> ::= Mary | John | Alfred

  <object> ::= Mary | John | Alfred

- Alternation to the previous grammar

  <sentence> ::= <subject><predicate>

  <subject> ::= <noun>

  <predicate> ::= <verb><object>

  <verb> ::= greets

  <object> ::= <noun>

  <noun> ::= Mary | John | Alfred

Infinite Number of Sentences

<object> ::= John |

John again |

John again and again |

….

Instead use recursive definition

<object> ::= John |

John <repeat factor>

<repeat factor> ::= again |

again and <repeat factor>

A rule is recursive if its LHS appears in its RHS

Simple example for PLs

- How you can describe simple arithmetic?

<expr> ::= <expr> <operator> <expr> |<var>

< operator > ::= + | - | * | /

<var> ::= a | b | c | …

<var> ::= <signed number>

<signed number> ::= + <number> | - <number>

<number> ::= <number> <digit> | <digit>

….

….

Identifiers

<identifier> → <letter> |

<identifier><letter> |

<identifier><digit>

PASCAL/Ada If Statement

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

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

Or

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

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

Grammars and Derivations

- A grammar is a generative device for defining languages
- The sentences of the language are **generated** through a sequence of applications of the rules, starting from the special nonterminal called ***start symbol***.
- Such a generation is called a **derivation**.
- Start symbol represents a complete program. So it is usually named as <program>.

An Example Grammar

```
<program>    → begin <stmt_list> end
<stmt_list>  → <stmt> |
               <stmt> ; <stmt_list>
<stmt>       → <var> := <expression>
<var>        → A | B | C
<expression> → <var> |
               <var> <arith_op> <var>
<arith_op>   →        + | - | * | /
```

Derivation

- In order to check if a given string represents a valid program in the language, we try to derive it in the grammar.
- Derivation starts from the start symbol <program>.
- At each step we replace a nonterminal with its definition (RHS of the rule).

Derivations

- Every string of symbols in a derivation is a ***sentential form***
- A ***sentence*** is a sentential form that has only terminal symbols
- 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

Derive string:

begin A := B; C := A * B end

```
<program>    → begin <stmt_list> end
<stmt_list>  → <stmt> | <stmt> ; <stmt_list>
<stmt>       → <var> := <expression>
<var>        → A | B | C
<expression> → <var> | <var> <arith_op> <var>
<arith_op>   →          + | - | * | /
```

<program> ⇒ begin <stmt_list> end

Leftmost derivation

⇒ begin <stmt> ; <stmt_list> end

⇒ begin <var> := <expression>; <stmt_list> end

Rightmost derivation

⇒ begin <stmt> ; <stmt_list> end

⇒ begin <stmt>; <stmt> end

Derive string:
begin A := B; C := A * B end

```
<program>    → begin <stmt_list> end
<stmt_list>  → <stmt> | <stmt> ; <stmt_list>
<stmt>       → <var> := <expression>
<var>        → A | B | C
<expression> → <var> | <var> <arith_op> <var>
<arith_op>   →          + | - | * | /
```

<program> ⇒ begin <stmt_list> end
⇒ begin <stmt> ; <stmt_list> end
⇒ begin <var> := <expression>; <stmt_list> end
⇒ begin A := <expression>; <stmt_list> end
⇒ begin A := B; <stmt_list> end
⇒ begin A := B; <stmt> end
⇒ begin A := B; <var> := <expression> end
⇒ begin A := B; C := <expression> end
⇒ begin A := B; C := <var><arith_op><var> end
⇒ begin A := B; C := A <arith_op> <var> end
⇒ begin A := B; C := A * <var> end
⇒ begin A := B; C := A * B end

If always the leftmost nonterminal is replaced, then it is called **leftmost derivation.**

Derive string:

begin A := B; C := A * B end

```
<program>    → begin <stmt_list> end
<stmt_list>  → <stmt> | <stmt> ; <stmt_list>
<stmt>       → <var> := <expression>
<var>        → A | B | C
<expression> → <var> | <var> <arith_op> <var>
<arith_op>   →          + | - | * | /
```

<program> ⇒ begin <stmt_list> end
⇒ begin <stmt> ; <stmt_list> end
⇒ begin <stmt> ; <stmt> end
⇒ begin <stmt> ; <var> := <expression> end
⇒ begin <stmt> ; <var> := <var><arith_op><var> end
⇒ begin <stmt> ; <var> := <var><arith_op> B end
⇒ begin <stmt> ; <var> := <var> * B end
⇒ begin <stmt> ; <var> := A * B end
⇒ begin <stmt> ; C := A * B end
⇒ begin <var> := <expression>; C := A * B end
⇒ begin <var> := <var>; C := A * B end
⇒ begin <var> := B ; C := A * B end
⇒ begin A := B; C := A * B end

If always the rightmost nonterminal is replaced, then it is called **rightmost derivation**.

Derive string:

begin A := B; C := A * B end

```
<program>    → begin <stmt_list> end
<stmt_list>  → <stmt> | <stmt> ; <stmt_list>
<stmt>       → <var> := <expression>
<var>        → A | B | C
<expression> → <var> | <var> <arith_op> <var>
<arith_op>   →        + | - | * | /
```

<program> ⇒ begin <stmt_list> end
⇒ begin <stmt> ; <stmt_list> end
⇒ begin <var> := <expression>; <stmt_list> end
⇒ begin A := <expression>; <stmt_list> end
⇒ begin A := B; <stmt_list> end
⇒ begin A := B; <stmt> end
⇒ begin A := B; <var> := <expression> end
⇒ begin A := B; C := <expression> end
⇒ begin A := B; C := <var><arith_op><var> end
⇒ begin A := B; C := A <arith_op> <var> end
⇒ begin A := B; C := A * <var> end
⇒ begin A := B; C := A * B end

Leftmost derivation

<program> ⇒ begin <stmt_list> end
⇒ begin <stmt> ; <stmt_list> end
⇒ begin <stmt> ; <stmt> end
⇒ begin <stmt> ; <var> := <expression> end
⇒ begin <stmt> ; <var> := <var><arith_op><var> end
⇒ begin <stmt> ; <var> := <var><arith_op> B end
⇒ begin <stmt> ; <var> := <var> * B end
⇒ begin <stmt> ; <var> := A * B end
⇒ begin <stmt> ; C := A * B end
⇒ begin <var> := <expression>; C := A * B end
⇒ begin <var> := <var>; C := A * B end
⇒ begin <var> := B ; C := A * B end
⇒ begin A := B; C := A * B end

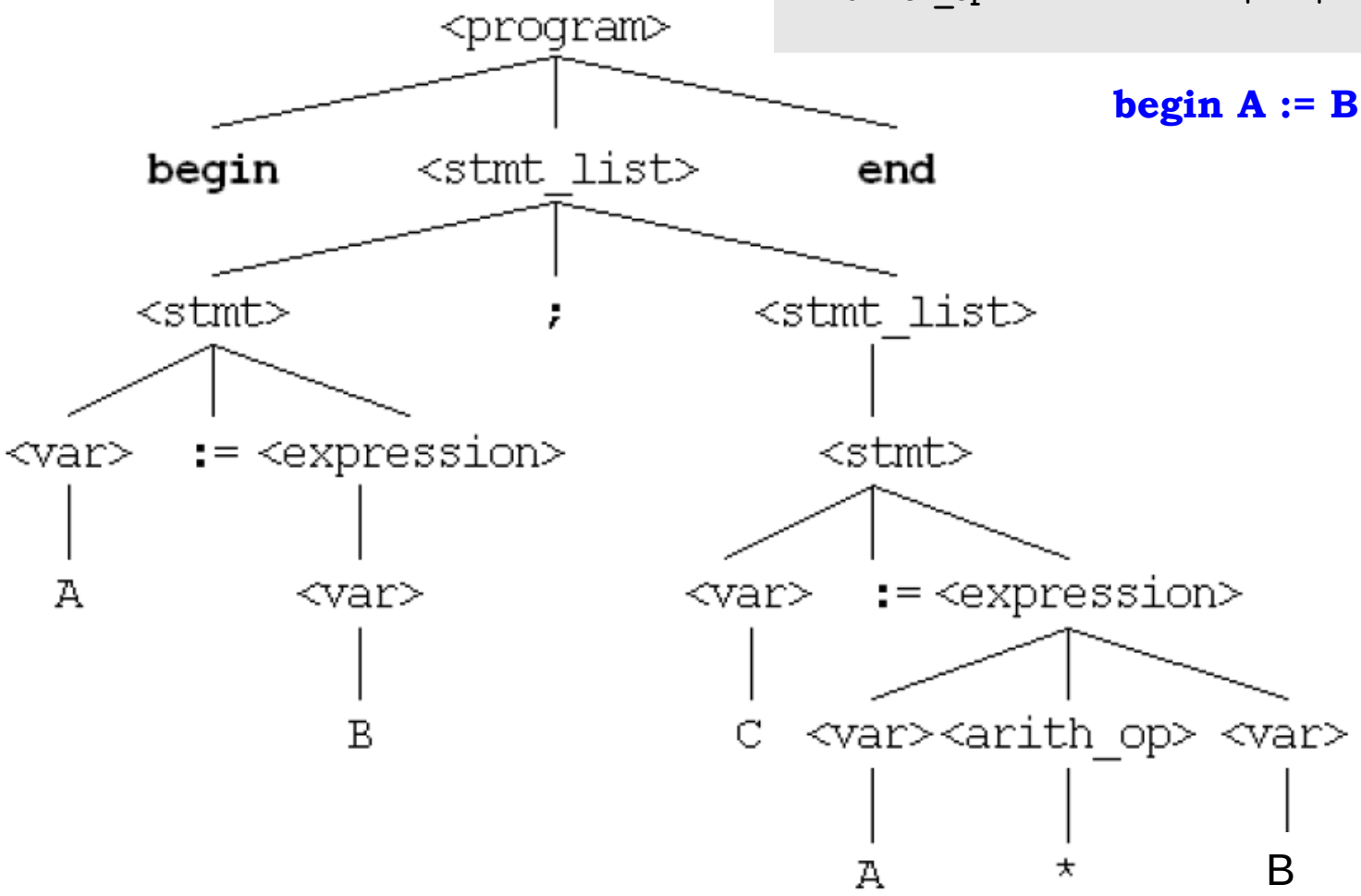
Rightmost derivation

Parse Tree

A hierarchical representation of a derivation

```
<program>    → begin <stmt_list> end
<stmt_list>  → <stmt> | <stmt> ; <stmt_list>
<stmt>       → <var> := <expression>
<var>        → A | B | C
<expression> → <var> | <var> <arith_op> <var>
<arith_op>   →        + | - | * | /
```

begin A := B; C := A * B end



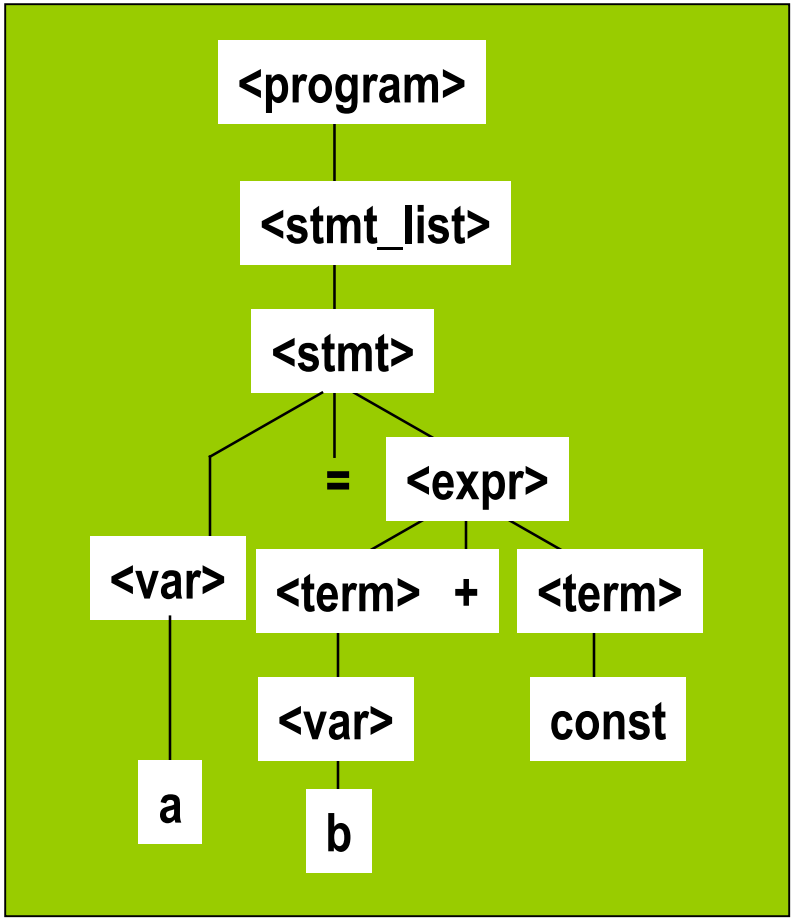
Another Example

a = b + const

⇒ <stmt_list>
⇒ <stmt>
⇒ <var> = <expr>
⇒ <var> = <term> + <term>
⇒ <var> = <term> + const
⇒ <var> = <var> + const
⇒ <var> = b + const
⇒ a = b + const

⇒ <stmt_list>
⇒ <stmt>
⇒ <var> = <expr>
⇒ a = <expr>
⇒ a = <term> + <term>
⇒ a = <var> + <term>
⇒ a = b + <term>
⇒ a = b + const

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



Ambiguity (Belirsizlik) in Grammars

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

Example

- Given the following grammar

<assign> ::= <id> = <expr>

<id> ::= A | B | C

<expr> ::= <expr> + <expr>

| <expr> * <expr>

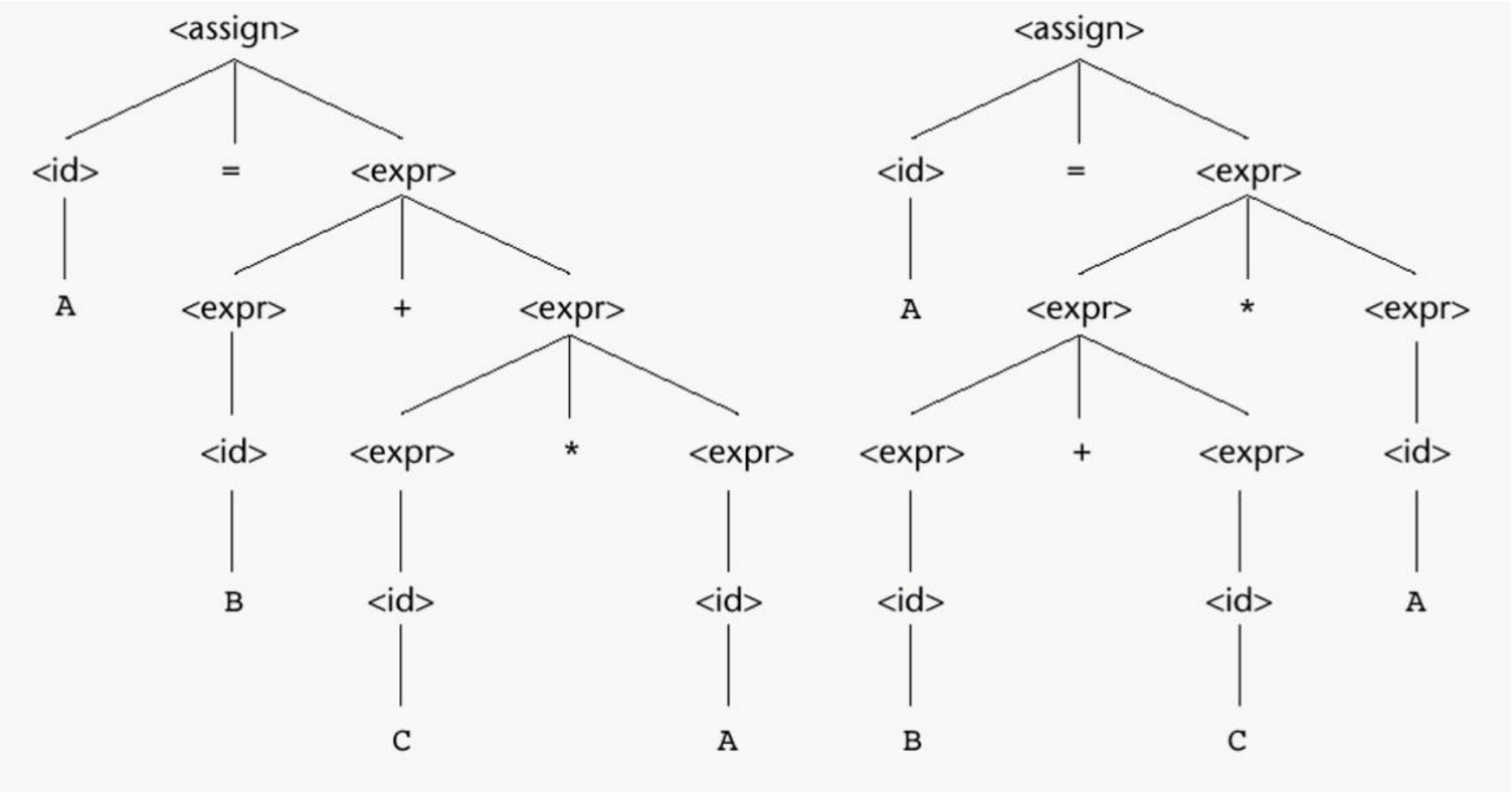
| (<expr>)

| <id>

Parse Tree(s) for A = B + C * A

Two Parse Trees for A = B + C * A

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



Two Leftmost derivations for A = B + C * A

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

<assign> => <id> = <expr>
=> A = <expr>
=> A = <expr> + <expr>
=> A = <id> + <expr>
=> A = B + <expr>
=> A = B + <expr> * <expr>
=> A = B + <id> * <expr>
=> A = B + C * <expr>
=> A = B + C * <id>
=> A = B + C * A

<assign> => <id> = <expr>
=> A = <expr>
=> A = <expr> * <expr>
=> A = <expr> + <expr> * <expr>
=> A = <id> + <expr> * <expr>
=> A = B + <expr> * <expr>
=> A = B + <id> * <expr>
=> A = B + C * <expr>
=> A = B + C * <id>
=> A = B + C * A

Two Rightmost derivations for A = B + C * A

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

<assign> => <id> = <expr>
=> <id> = <expr> + <expr>
=> <id> = <expr> + <expr> * <expr>
=> <id> = <expr> + <expr> * <id>
=> <id> = <expr> + <expr> * A
=> <id> = <expr> + <id> * A
=> <id> = <expr> + C * A
=> <id> = <id> + C * A
=> <id> = B + C * A
=> A = B + C * A

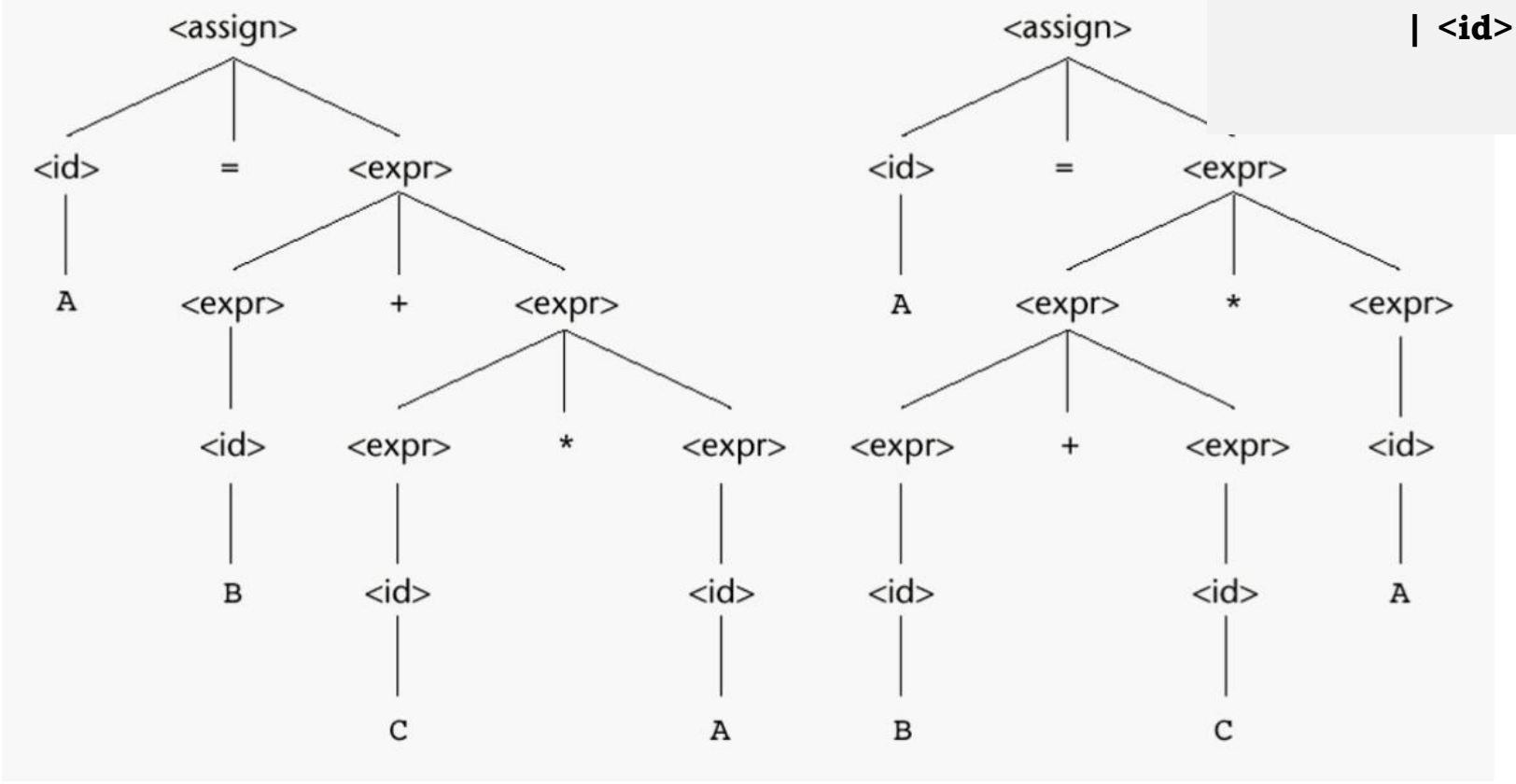
<assign> => <id> = <expr>
=> <id> = <expr> * <expr>
=> <id> = <expr> * <id>
=> <id> = <expr> * A
=> <id> = <expr> + <expr> * A
=> <id> = <expr> + <id> * A
=> <id> = <expr> + C * A
=> <id> = <id> + C * A
=> <id> = B + C * A
=> A = B + C * A

A = B + C * A

A = 3 , B = 4, C = 5

3 + 4 * 5

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



3 + [4 * 5]

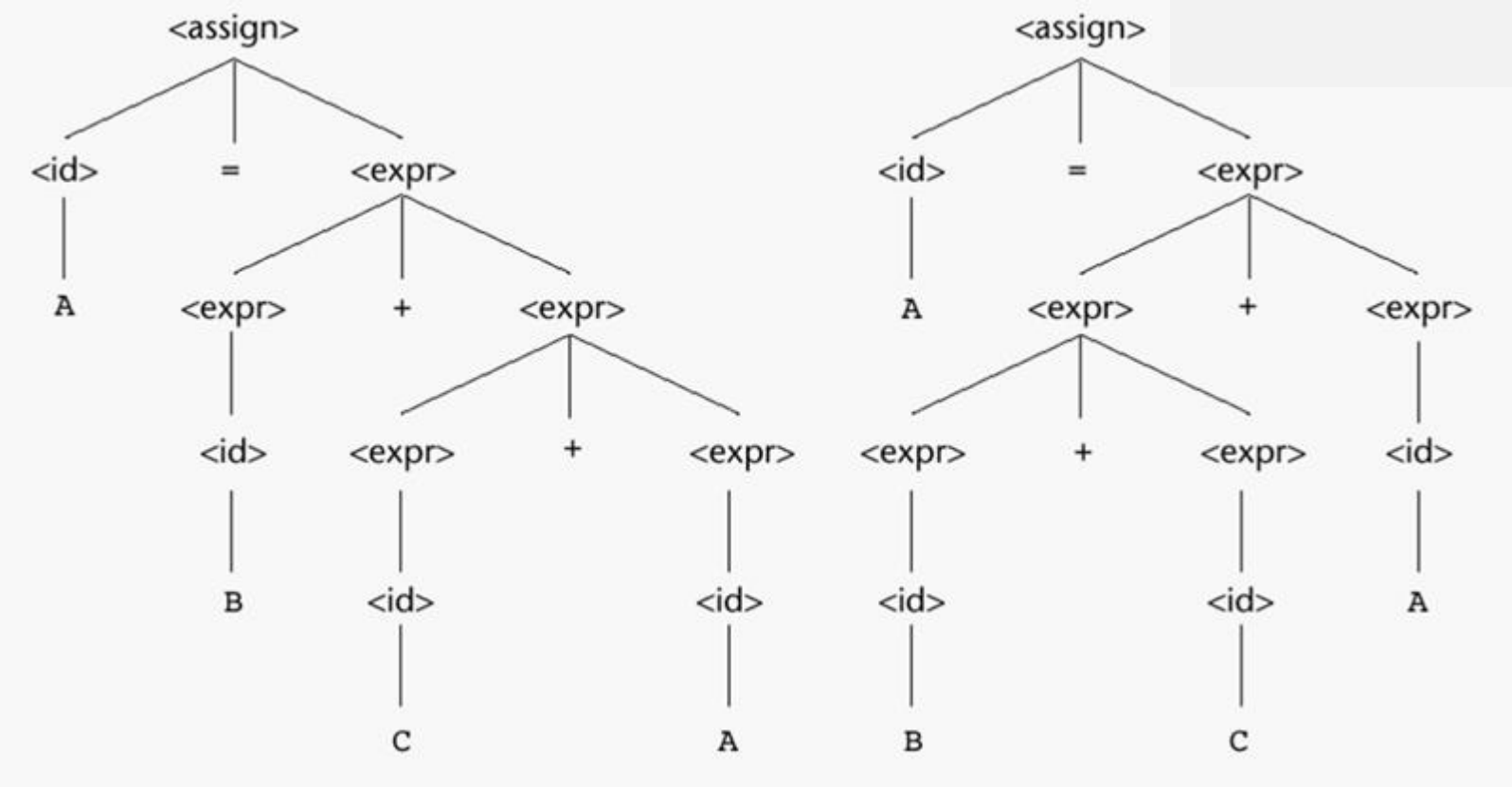
[3 + 4] * 5

A = B + C + A

A = 3 , B = 4, C = 5

3 + 4 + 5

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



3 + [4 + 5]

[3 + 4] + 5

Single leftmost derivation for A = B + C

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

<assign> => <id> = <expr>
=> A = <expr>
=> A = <expr> + <expr>
=> A = <id> + <expr>
=> A = B + <expr>
=> A = B + <id>
=> A = B + C

There is also a single rightmost derivation

And a single parse tree for A = B + C

Finding at least one string with more than a single parse tree
(or more than a single leftmost derivation
Or more than a single rightmost derivation)
is sufficient to prove ambiguity of a grammar

Handling Ambiguity

- The grammar of a PL must not be ambiguous
- There are solutions for correcting the ambiguity
  - Operator precedence
  - Associativity rules

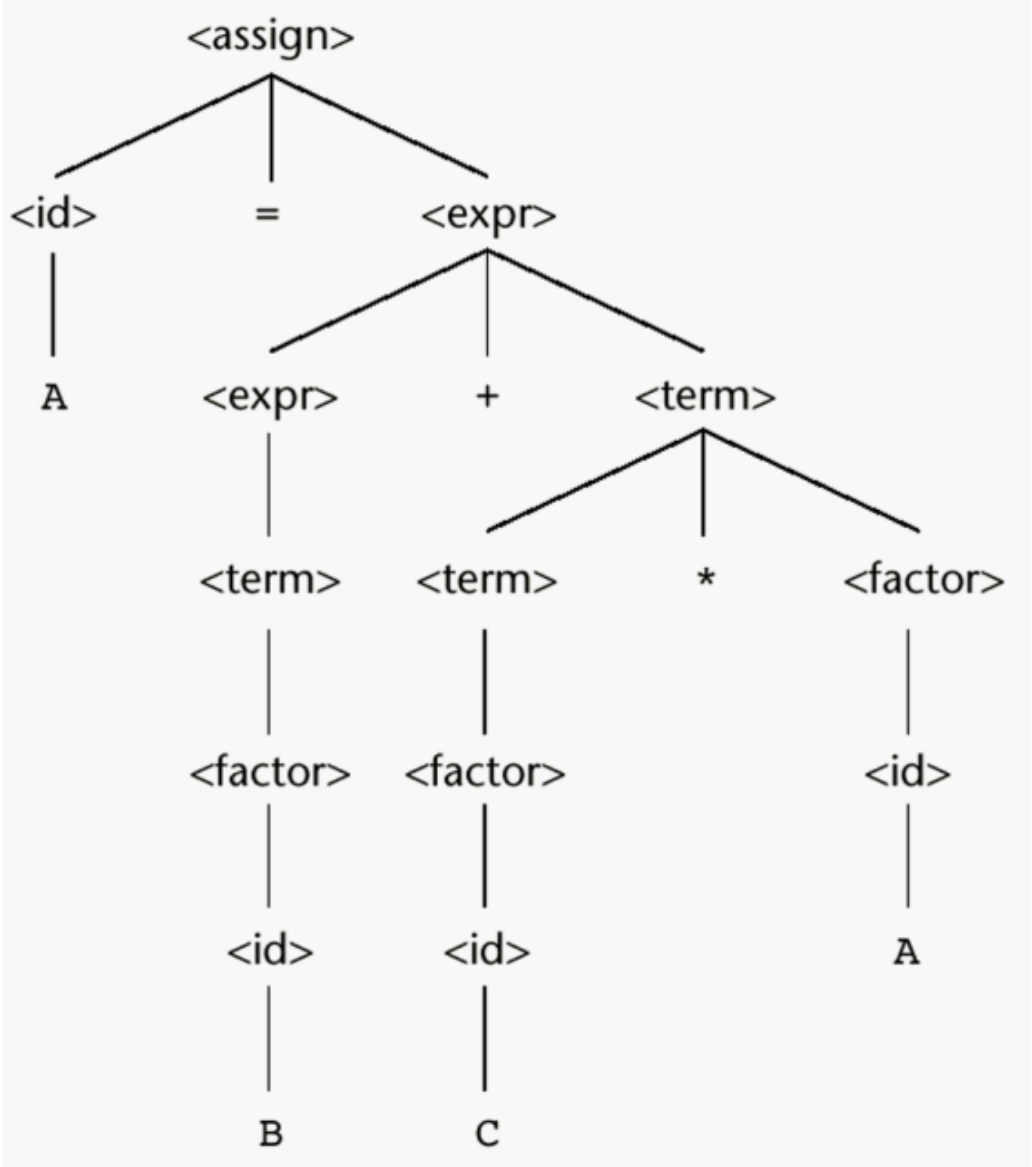
Operator Precedence

- In mathematics * operation has a higher precedence than +
- This can be implemented with extra nonterminals

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

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

Unique Parse Tree for A = B + C * A



Associativity of Operators

- What about equal precedence operators?

- In math addition and multiplication are associative
  A+B+C = (A+B)+C = A+(B+C)

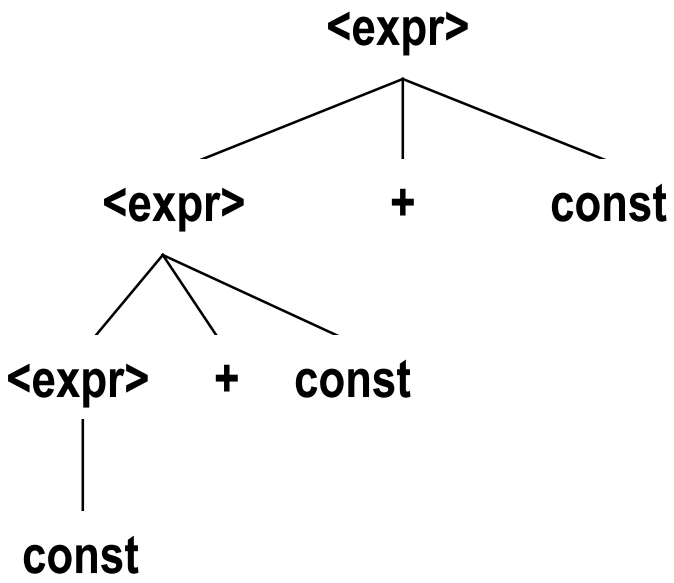
- However computer arithmetic may not be associative
- Ex: for floating point addition where floating points values store 7 digits of accuracy, adding eleven numbers together where one of the numbers is 10^7 and the others are 1 result would be 1.000001 * 10^7 only if the ten 1s are added first

- Subtraction and division are not associative
  A/B/C/D = ? ((A/B)/C)/D ≠A/(B/(C/D))

Associativity of Operators

- Operator associativity can also be indicated by a grammar

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



Associativity (birleşirlik)

- In a BNF rule, if the LHS appears at the beginning of the RHS, the rule is said to be left recursive
- Left recursion specifies left associativity

<expr> ::= <expr> + <term>

| <term>

- Similar for the right recursion
- In most of the languages exponential is defined as a right associative operation

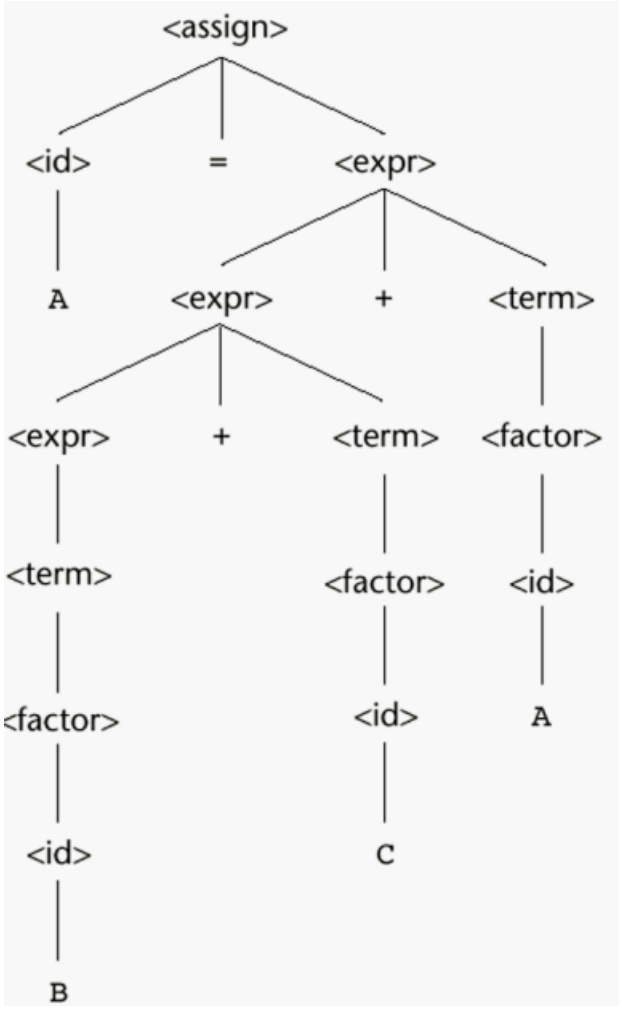
<factor> ::= <expr> ** <factor>

| <expr>

<expr> ::= (<expr>)

| <id>

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



Left associativity
Left addition is lower than the right addition

Is this ambiguous?

<stmt> ::= <if_stmt> | <other_stmt>

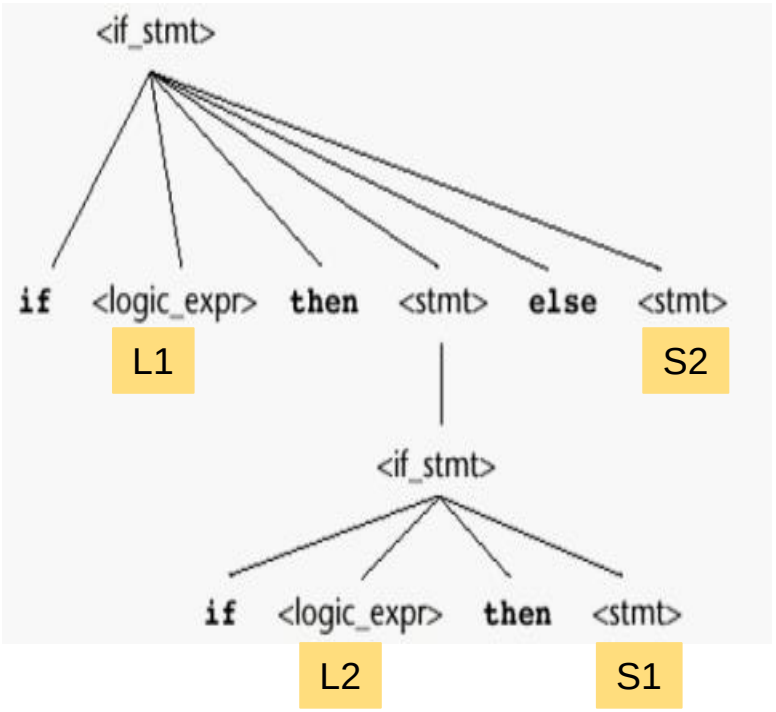
<if_stmt> ::= if <logic_expr> then <stmt>

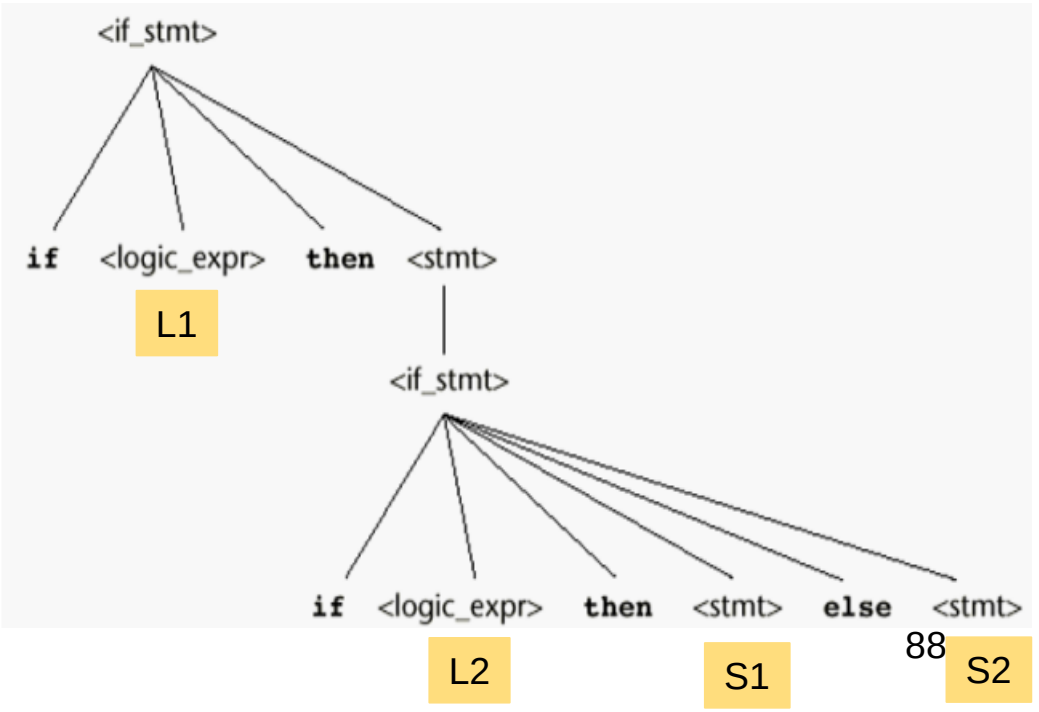
| if <logic_expr> then <stmt> else <stmt>

<other_stmt> ::= S1 | S2

<logic_expr> ::= L1 | L2

Derive for : If L1 then if L2 then S1 else S2





An Unambiguous grammar for “if then else”

- Dangling else problem: there are more if then else
- To design an unambiguous if-then-else statement we have to decide which `if` a dangling `else` belongs to
- Most PL adopt the following rule:
  - “an else is matched with the closest previous unmatched if statement”
  - (unmatched if = else-less if)

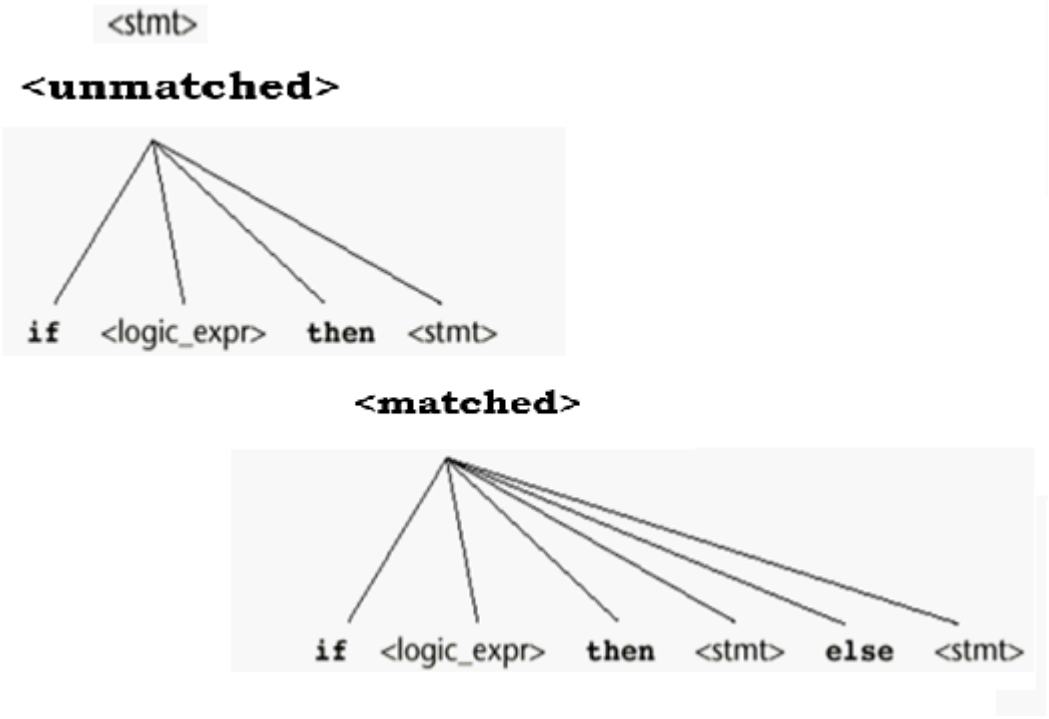
Has a unique parse tree

<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>

Draw the parse tree

<stmt> ::= <matched> | <unmatched>
<matched> ::= if <logic_expr> then <matched> else <matched>
| <other_stmt>
<unmatched> ::= if <logic_expr> then <stmt>
| if <logic_expr> then <matched> else <unmatched>

If L1 then if L2 then S1 else S2



Extended BNF

- *EBNF: Same power but more convenient*
- Optional parts are placed in brackets [ ]

  [X] : X is optional (0 or 1 occurrence)

  <writeln> ::= WRITELN [(<item_list>)]

  <selection>::= if (<expr>) <stmt> [else<stmt>]

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

  {X}: 0 or more occurrences

  <identlist> = <identifier> {,<identifier>}

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

  (X1|X2|X3) : choose X1 or X2 or X3

  <term> → <term> (+|-) const

BNF vs Extended BNF

- **BNF**

```
< expr> ::= <expr> + <term>
          | <expr>  - <term>
          | <term>
<term> ::= <term> * <factor>
         | <term> / <factor>
         | <factor>
<factor> ::= <expr> **
<factor>
         | <expr>
<expr> ::= (<expr>)
         | <id>
```

BNF vs Extended BNF

- **EBNF**

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

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