MODEL-BASED SOFTWARE DEVELOPMENT

LECTURE II.

TEXTUAL LANGUAGES

DR. FERENC SOMOGYI

TODAY'S AGENDA

I. Textual Languages

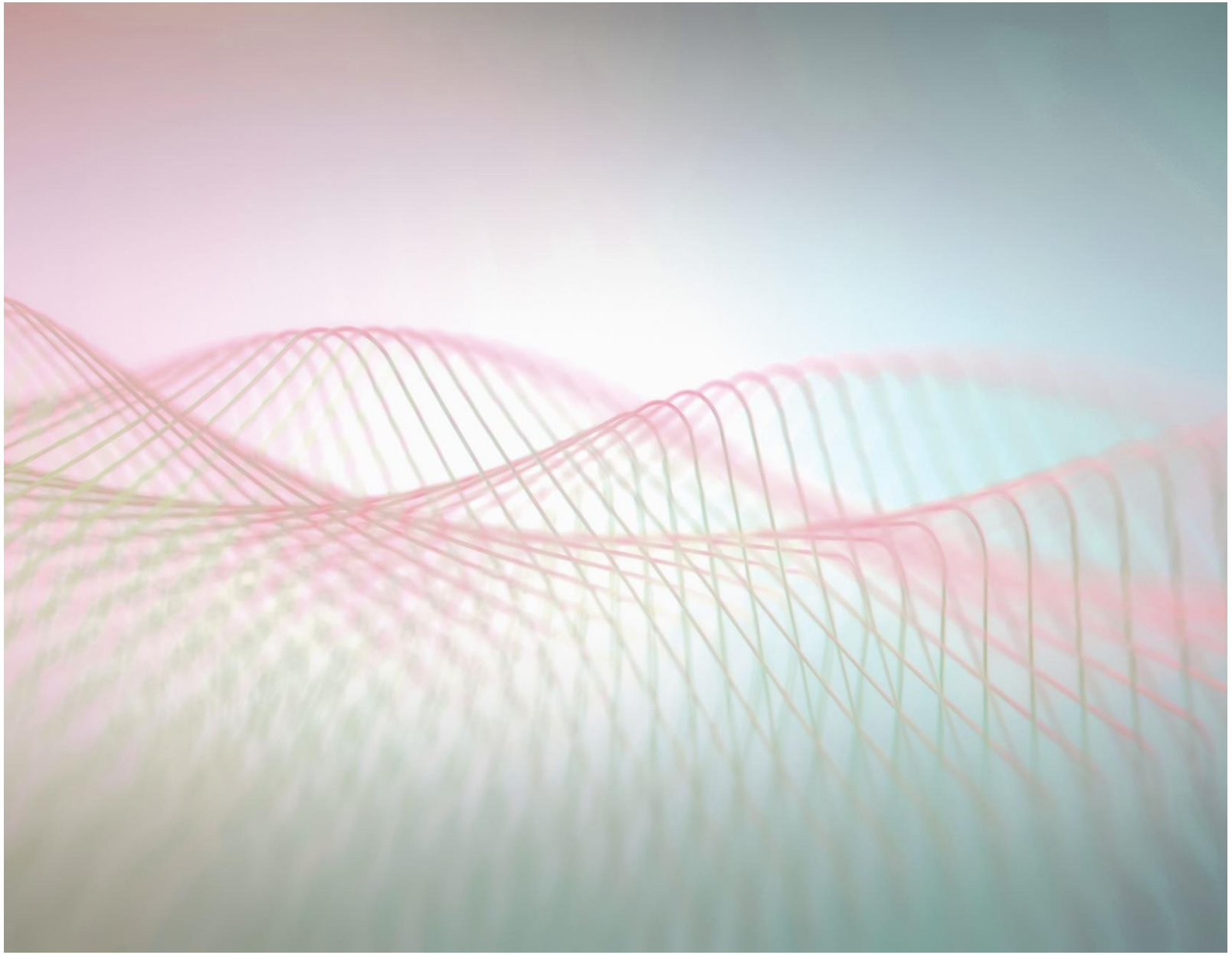
II. Introduction to Compilers

III. Lexical Analysis

IV. Regular Expressions

V. Formal Languages

VI. Syntax Analysis (Introduction)



WHAT LANGUAGES DO YOU KNOW?

Not only programming languages!

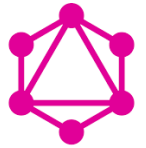
TEXTUAL LANGUAGES

- Natural Languages
  - Not relevant to this course ☺
  - English, Chinese, Hungarian
- Programming Languages
  - C, C++, C#, Java, Kotlin, Python, Rust, Logo
- Languages used in Web Development
  - HTML, CSS, JavaScript, TypeScript

TEXTUAL LANGUAGES

- Database Management Languages
  - Data Definition / Manipulation Language, etc.
  - SQL, GraphQL
- Structure Definition Languages
  - XML, XAML, JSON
- Document Description Languages
  - LaTeX, Markdown
- Hardware Description Languages
  - Verilog, VHDL



JSON

GraphQL


TEXTUAL LANGUAGES

- General-Purpose Languages
  - Usually (but not exclusively) programming languages
  - Which are considered general-purpose languages in the previous examples?
- Domain-Specific Languages
  - Describes the concepts of a particular domain
  - Which are considered domain-specific languages in the previous examples?

DOMAIN-SPECIFIC VS. GENERAL-PURPOSE LANGUAGES

Domain-Specific Languages	General-Purpose Languages
Uses the concepts of a domain (e.g. bicycles, HTML input form)	Uses general concepts (e.g. class, function, XML tag)
Made for experts	Made for programmers
More specific goals	More general goals
Less bound syntax	More bound syntax
Unique processing and environment	Development environment support

- There also exist General-Purpose Languages that are not programming languages
  - e.g. XML, JSON

DOMAIN-SPECIFIC LANGUAGES (DSL)

Internal DSL

- A General-Purpose Programming Language used in a special way
- Uses a subset of the original language features
- Processing is in the original language
- e.g. scripting languages, framework calls

External DSL

- Custom language, not based on the language of the application
- Unique syntax (or the syntax of a different language than that of the application)
- Processing is in another language
- e.g. Unix commands, SQL, HTML, CSS

TODAY'S AGENDA

I. Textual Languages

II. Introduction to Compilers

III. Lexical Analysis

IV. Regular Expressions

V. Formal Languages

VI. Syntax Analysis (Introduction)

INTERPRETER VS. COMPILER

Interpreter

- Runs in memory, virtual machine
- Executes the code statement-by-statement
- Starts fast, runs slow
- Typically runs until the first exception
- Debugging easier
- e.g. Perl, Python, DOS, UNIX shell

Compiler

- Usually generates machine code
- Processes the entire code (or most of it) at once
- Starts slow, runs fast
- Exceptions are displayed at the end
- Debugging harder (instrumentation?)
- e.g. C, C++, C#, Kotlin, Pascal

INTERPRETER VS. COMPILER

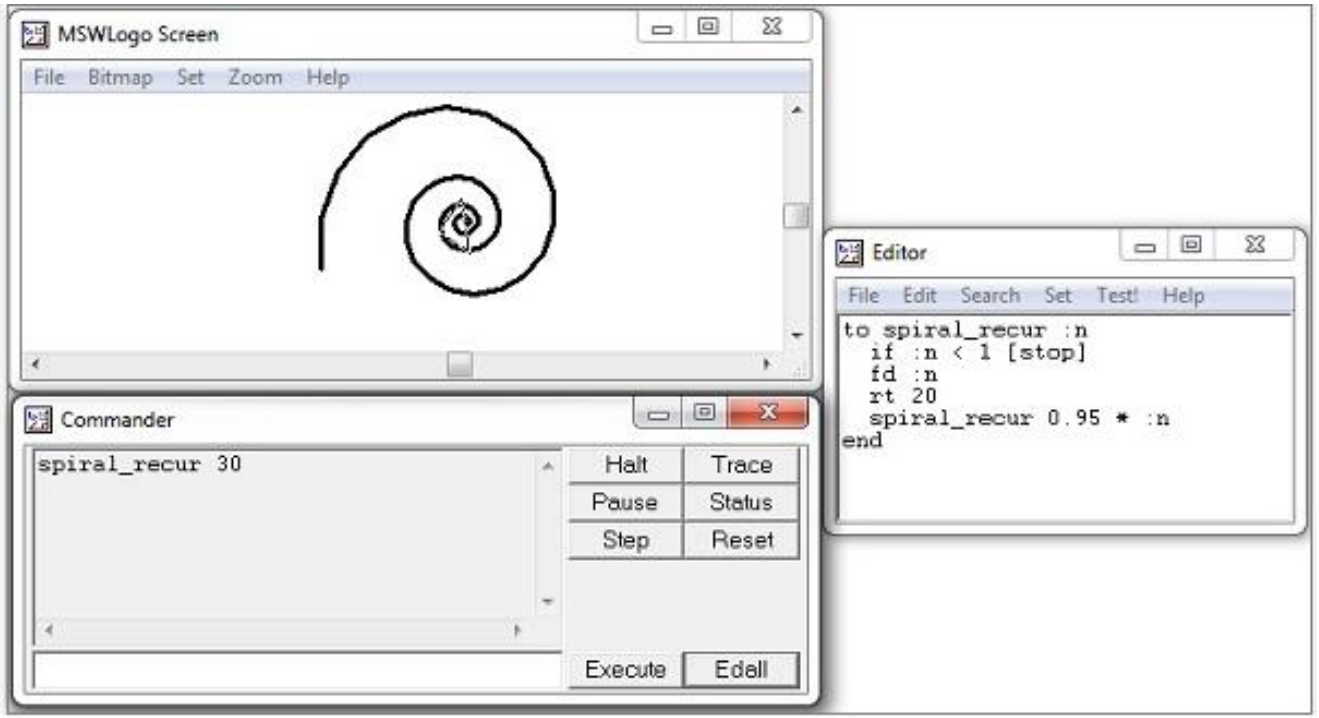
- Special cases
  - A language can be both compiled and interpreted at the same time
    - Java – Just-In-Time (JIT) compiler
  - A language can be compiled or interpreted
    - Erlang, Prolog, SQL, Logo
- Neither compiled nor interpreted languages
  - Typically, markup languages
  - XML, JSON, HTML, XAML, UML

INTERPRETER VS. COMPILER

- Just-In-Time (JIT) compiler
  - Compiles during execution instead of before execution
  - Further compiles the intermediate code that is compiled from the source code
  - Faster than interpretation but slower than standard compilation
  - Less memory required, dynamic runtime requests
  - Java JVM, .NET CLR
- Transpiler
  - Compiles to another high-level language instead of machine code
  - TypeScript → JavaScript, Python2 → Python3 compilers, etc.

INTERPRETER VS. COMPILER – EXAMPLE

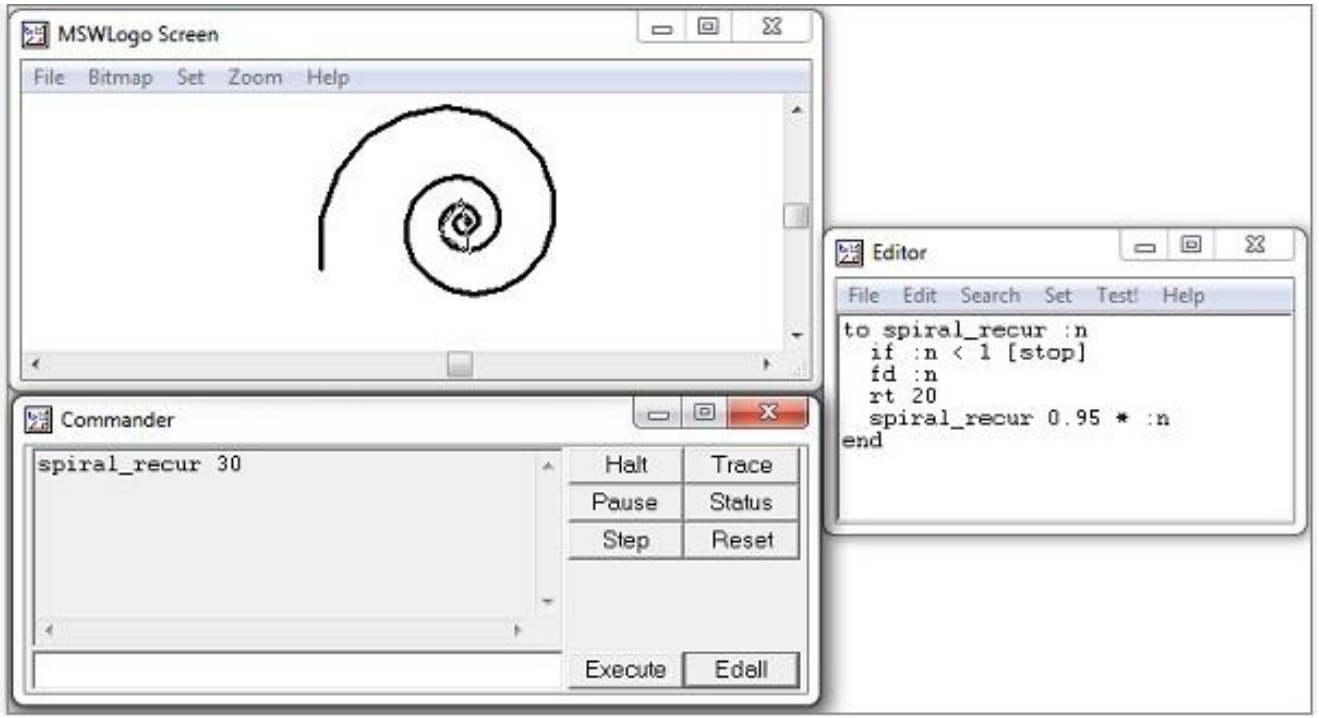
- MSWLogo
  - Programming education
  - Atomic statements
    - Move forward / back
    - Turn left / right
    - Pen up / down
- Programming concepts
  - If statements, functions, function parameters, etc.



Source: https://www.tutorialspoint.com/logo/logo_quick_guide.htm

INTERPRETER VS. COMPILER – EXAMPLE

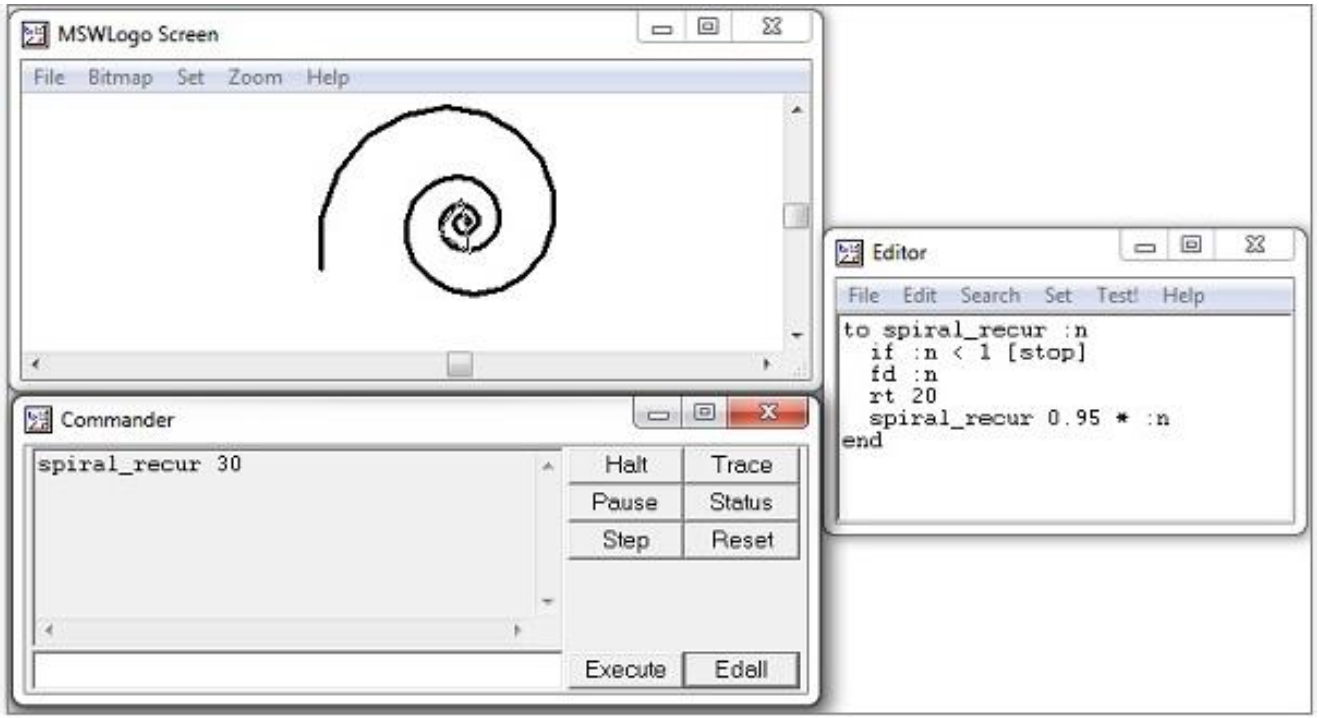
- Compiled Logo
  - Read the entire program
  - Using an arbitrary language…
    - e.g. Java
    - Implement a custom compiler (from scratch or by using a parser generator)
  - … compile to machine code
    - Or to another language (transpiler)
  - Then run the compiled code



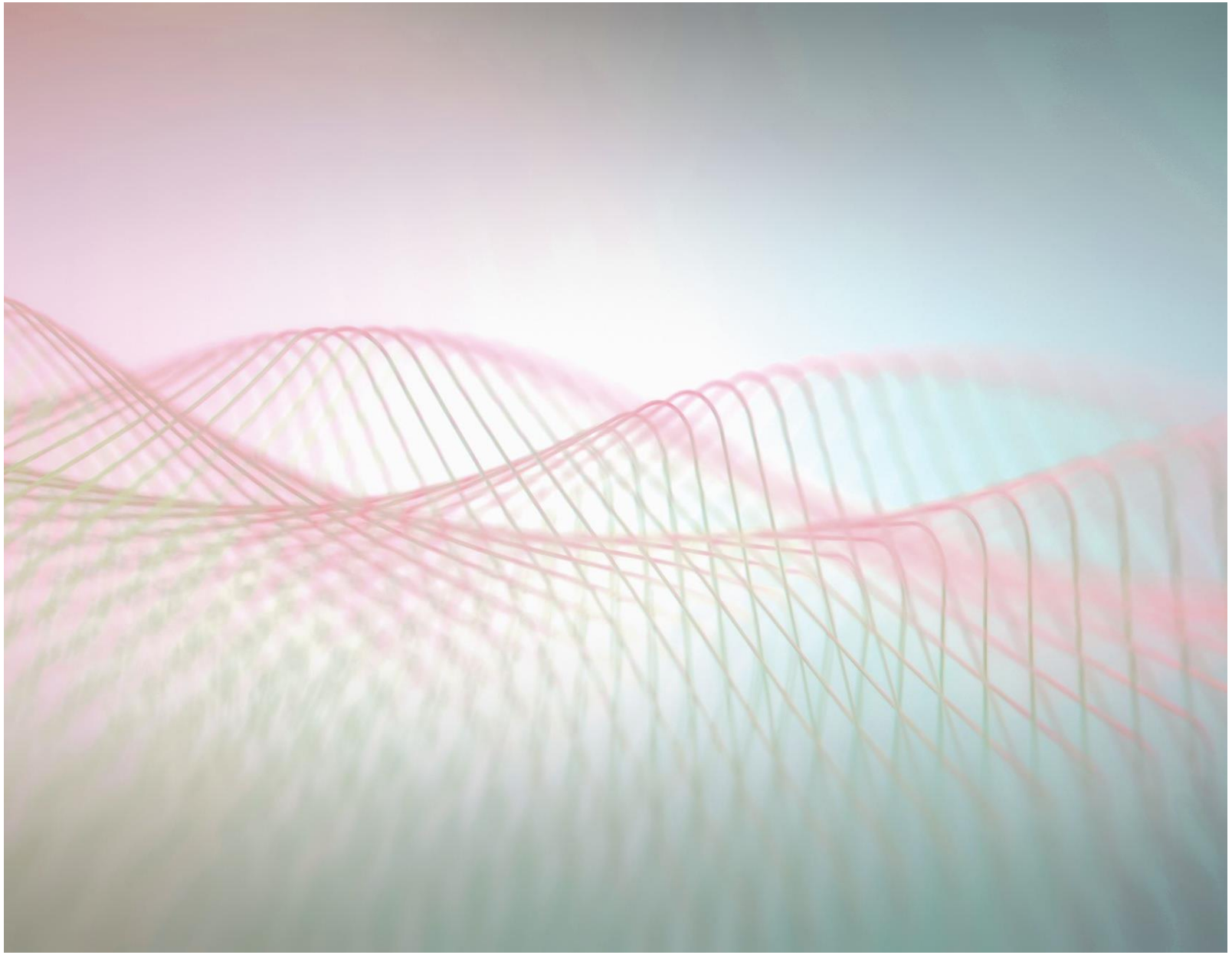
Source: https://www.tutorialspoint.com/logo/logo_quick_guide.htm

INTERPRETER VS. COMPILER – EXAMPLE

- Interpreted Logo
  - Process statement-by-statement
  - Using an arbitrary language…
    - e.g. Java
    - Implement a custom interpreter
  - … execute the statements
    - "fd :n" → Java Canvas API call

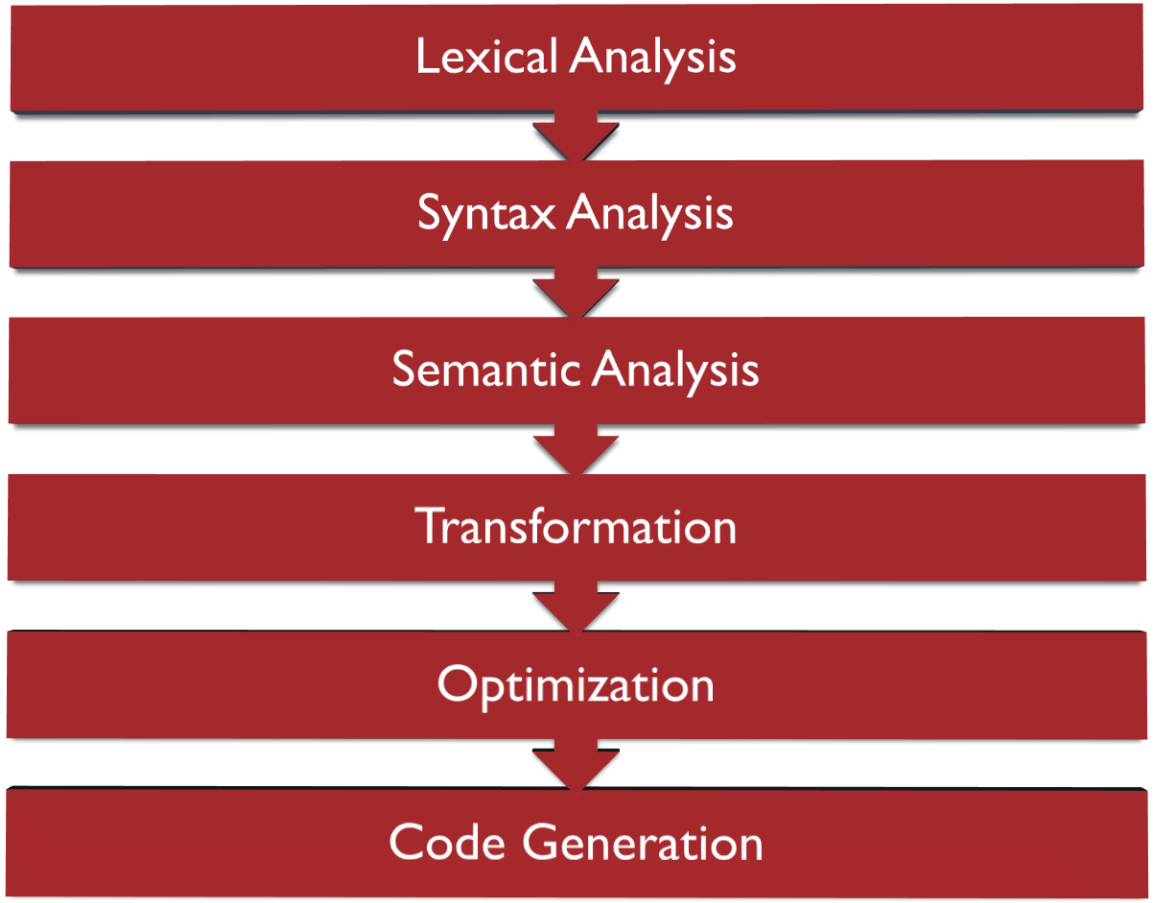


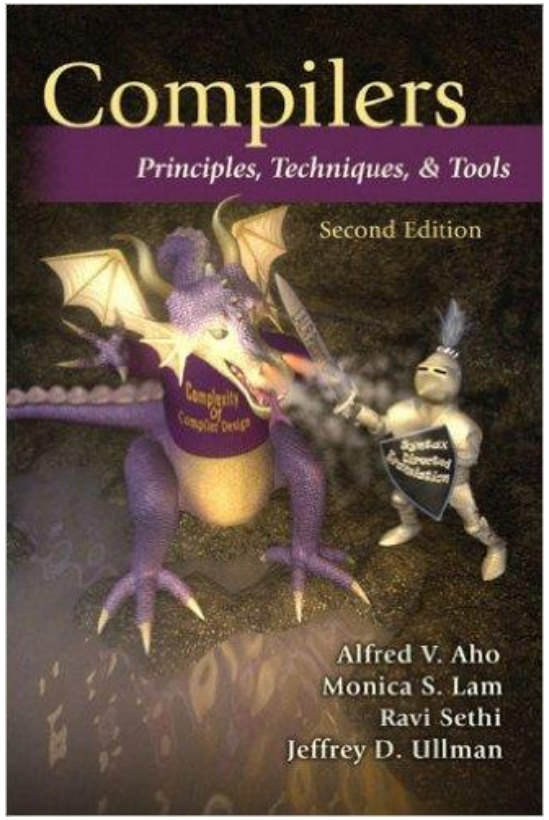
Source: https://www.tutorialspoint.com/logo/logo_quick_guide.htm



THE CLASSIC PHASES OF COMPILATION

THE CLASSIC PHASES OF COMPILATION



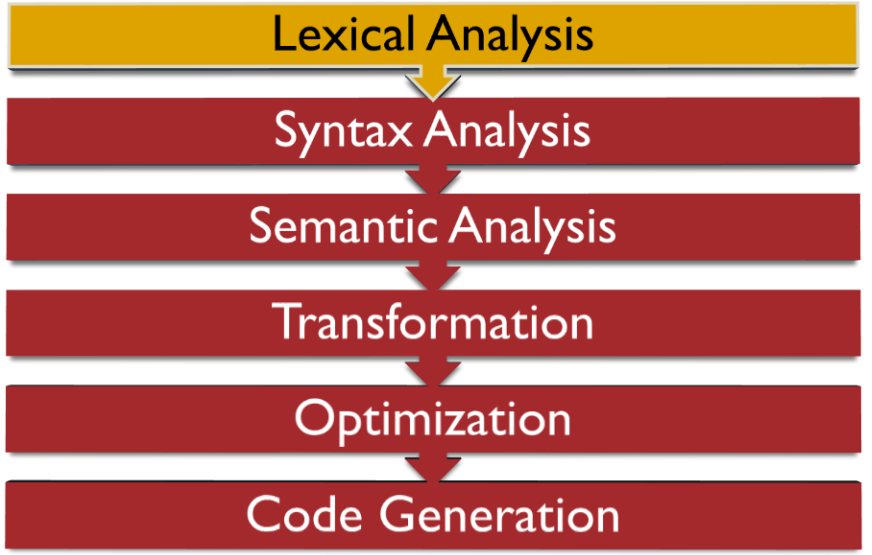


PHASE 0 – SOURCE CODE

```
while (y < z) {
    x = a + b;
    y += x;
}
```

PHASE I – LEXICAL ANALYSIS

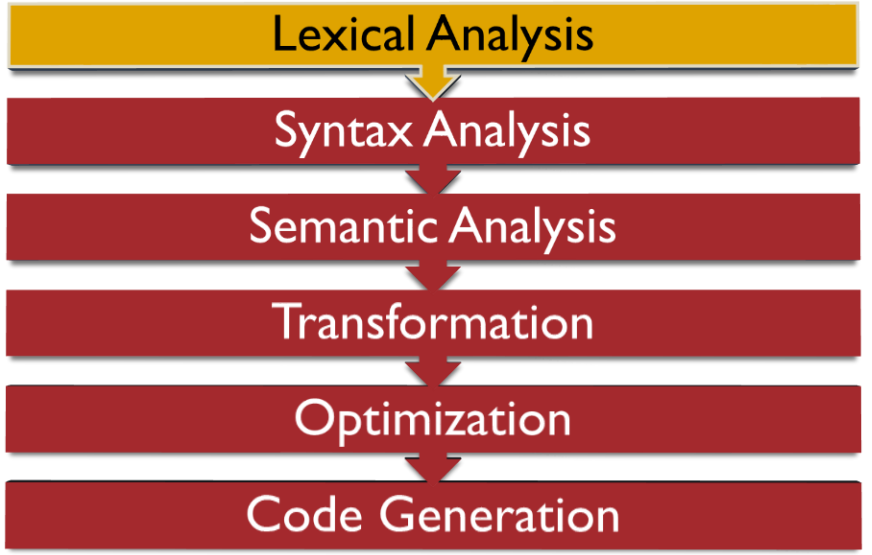
```
T_While
T_LeftParen
T_Identifier y
T_Less
T_Identifier z
T_RightParen
T_OpenBrace
T_Identifier x
T_Assign
…
```



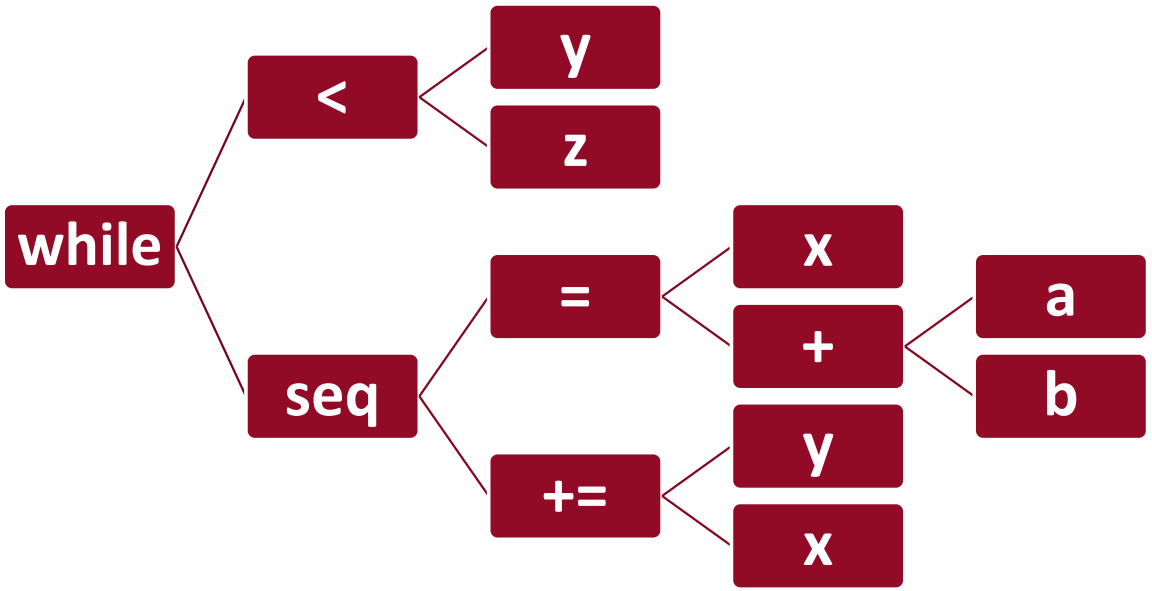

```
while (y < z) {
  x = a + b;
  y += x;
}
```

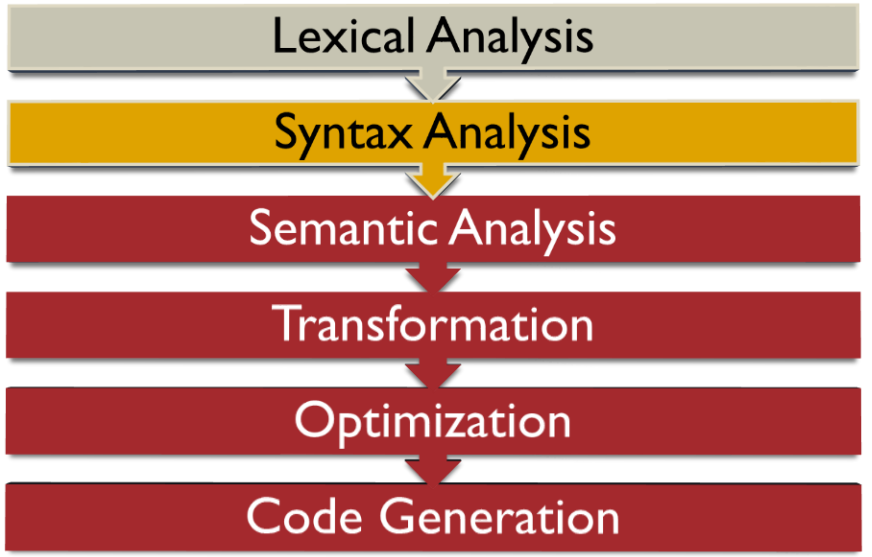
PHASE I – LEXICAL ANALYSIS

- Performed by the *lexer*
- Split the source code into atomic units
  - Atomic unit = token
  - Tokens are the basic building blocks
- Omit unnecessary characters
  - Not always performed
  - e.g. comments, whitespace



PHASE II – SYNTAX ANALYSIS

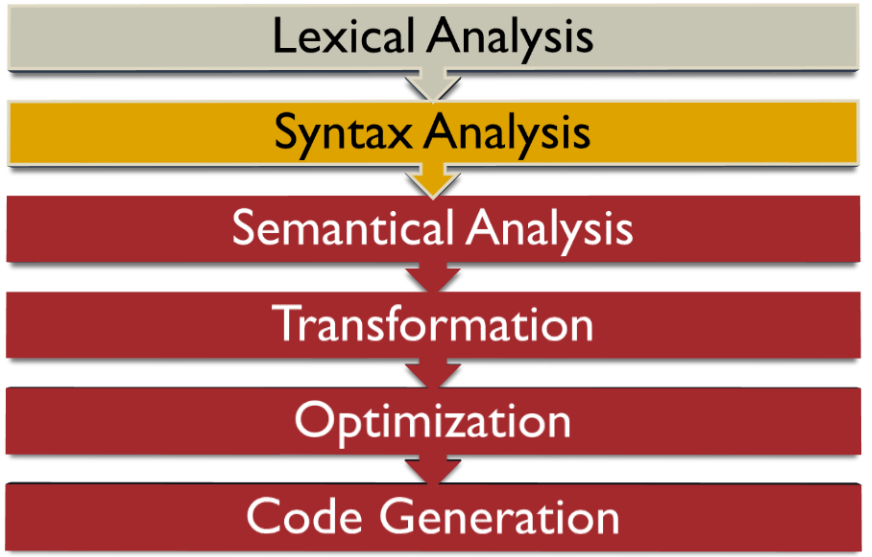




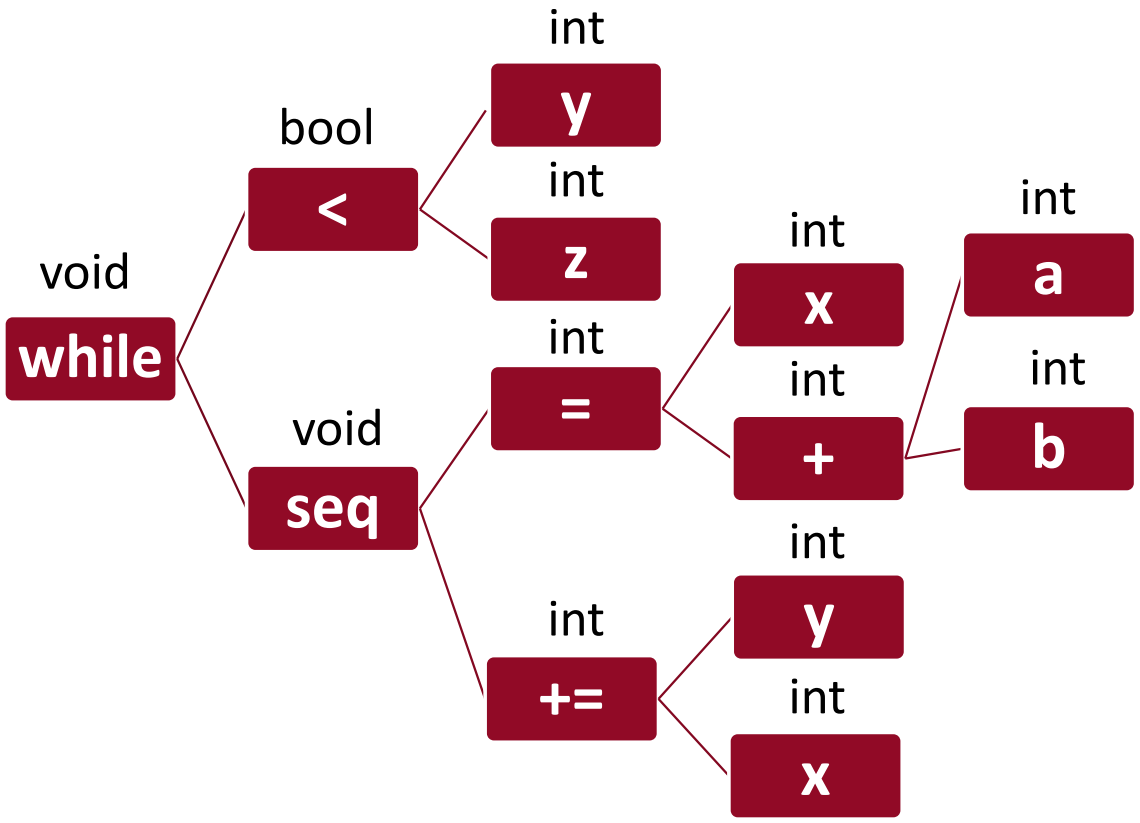

```
while (y < z) {
  x = a + b;
  y += x;
}
```

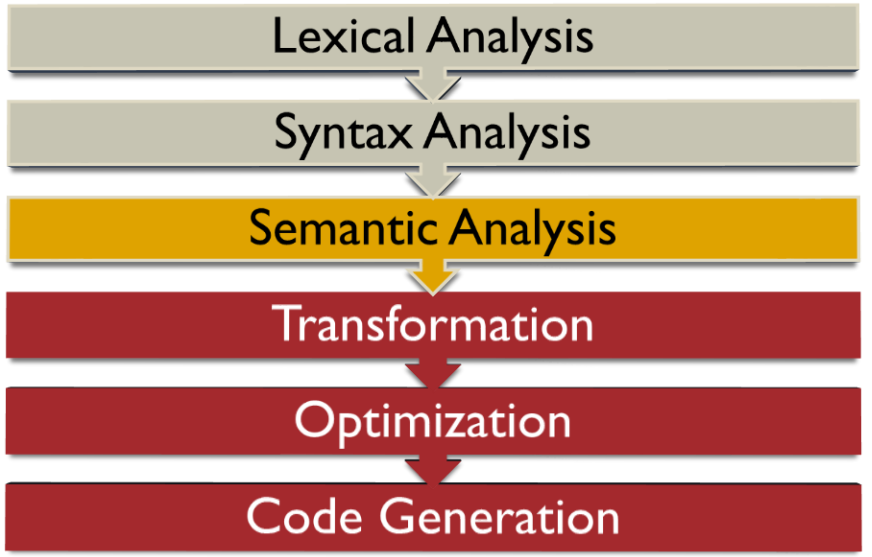
PHASE II – SYNTAX ANALYSIS

- Performed by the *parser*
  - The term parser is sometimes used in conjunction with the lexer
- Generate a structure from the source code
  - So that it can be processed by later phases
- Typically, a tree structure is generated
  - Syntax tree



PHASE III – SEMANTIC ANALYSIS

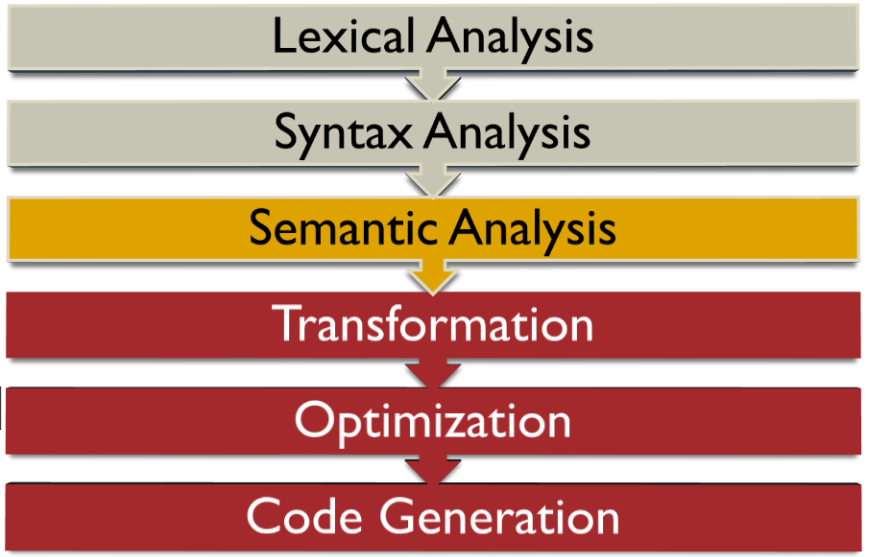




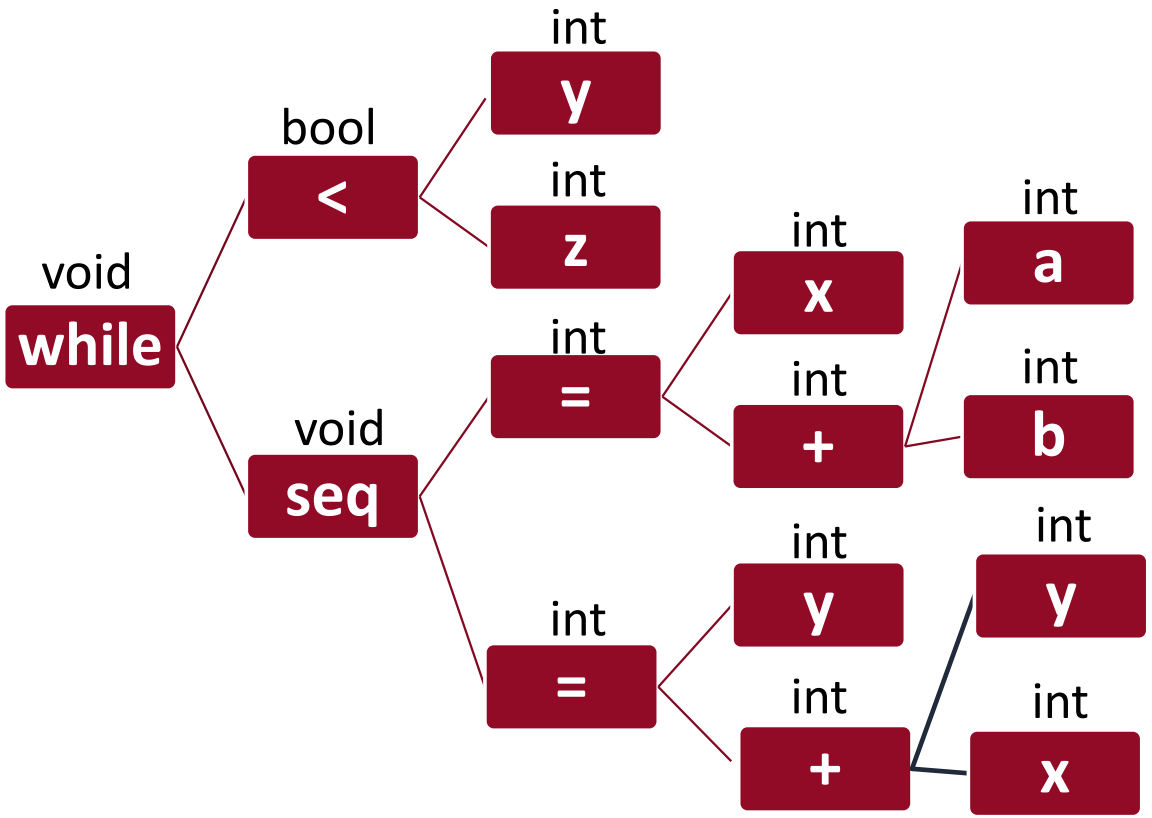

```
while (y < z) {
  x = a + b;
  y += x;
}
```

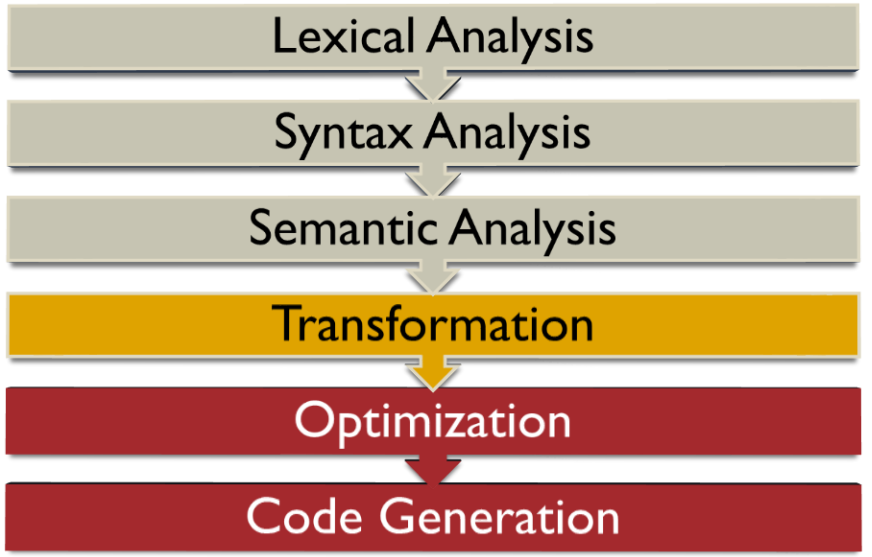
PHASE III – SEMANTIC ANALYSIS

- Associate the structure with meaning
  - Type information, variables, etc.
- Consistency checking
  - Constraints that cannot be evaluated while parsing
  - Knowing the entire structure, they can be evaluated
    - implementing interface functions, type of variables, indexing static arrays, type conforming expressions, etc.



PHASE IV – TRANSFORMATION

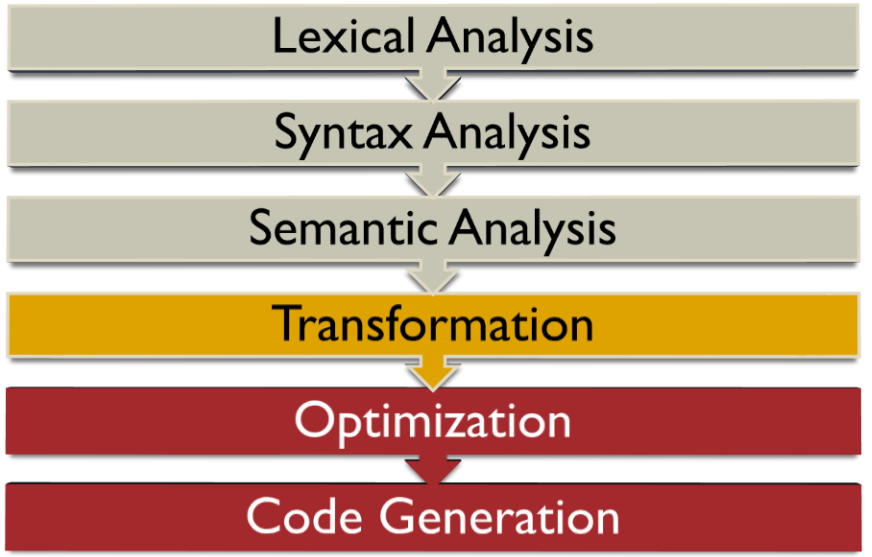




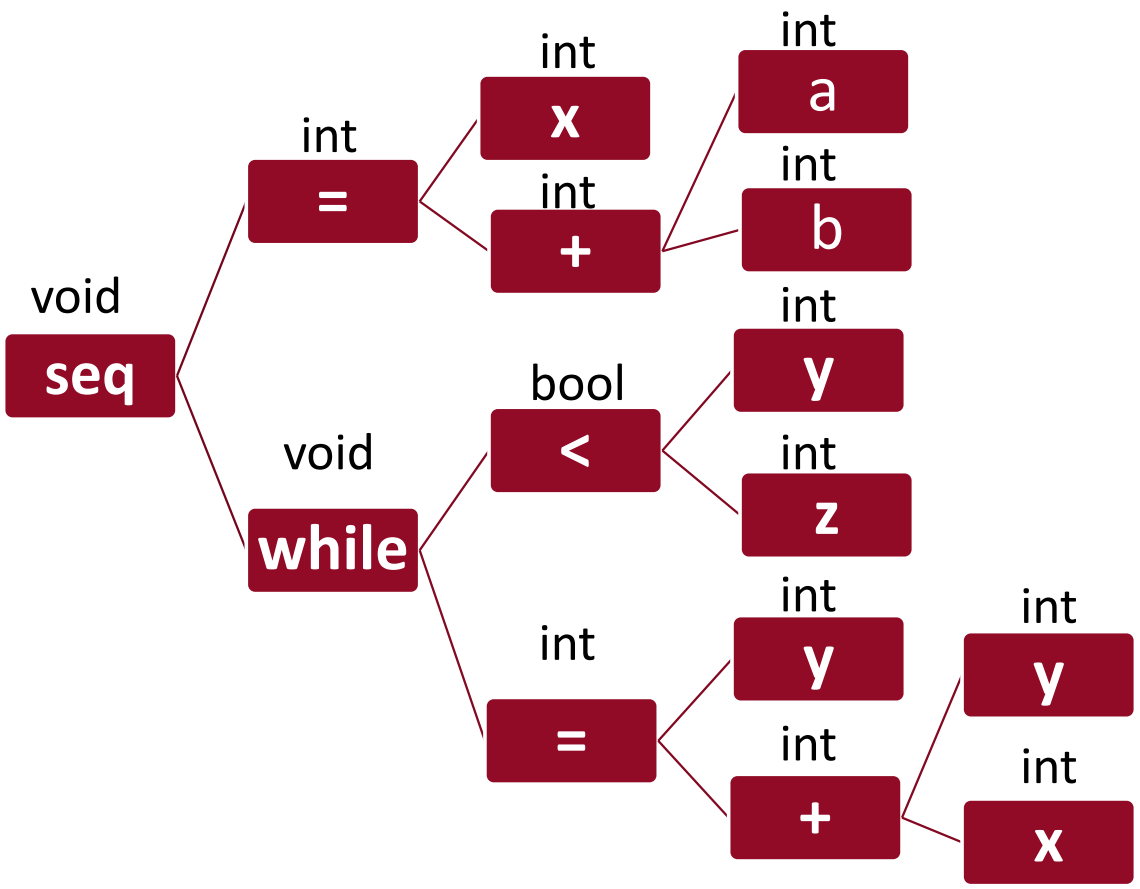

```
while (y < z) {
  x = a + b;
  y += x;
}
```

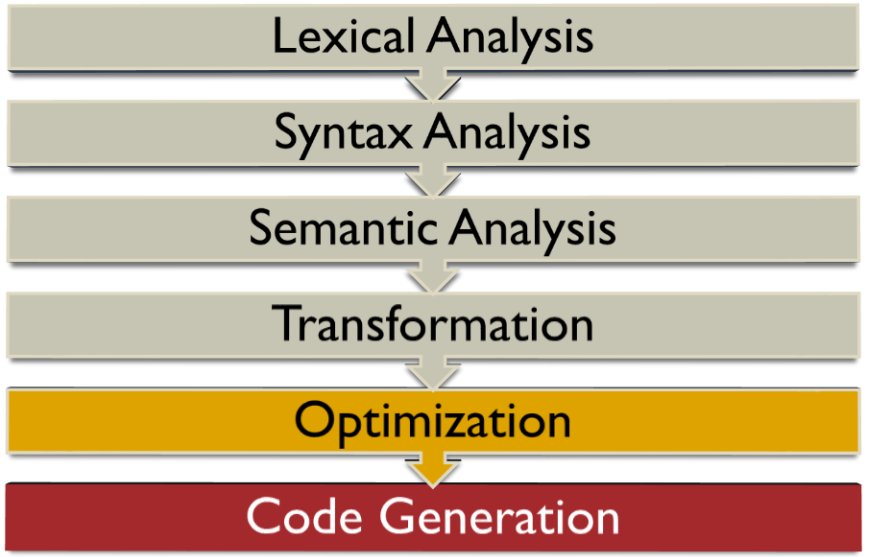
PHASE IV – TRANSFORMATION

- Optional phase
- Mapping to an intermediate language
  - Such a language is easier to use during optimization
  - Common Intermediate Language, Java Bytecode, or any other intermediate representations
- Map expressions to a uniform representation
  - "*y* += *x*" → "*y* = *y* + *x*"
  - Easier to work with and optimize



PHASE V – OPTIMIZATION

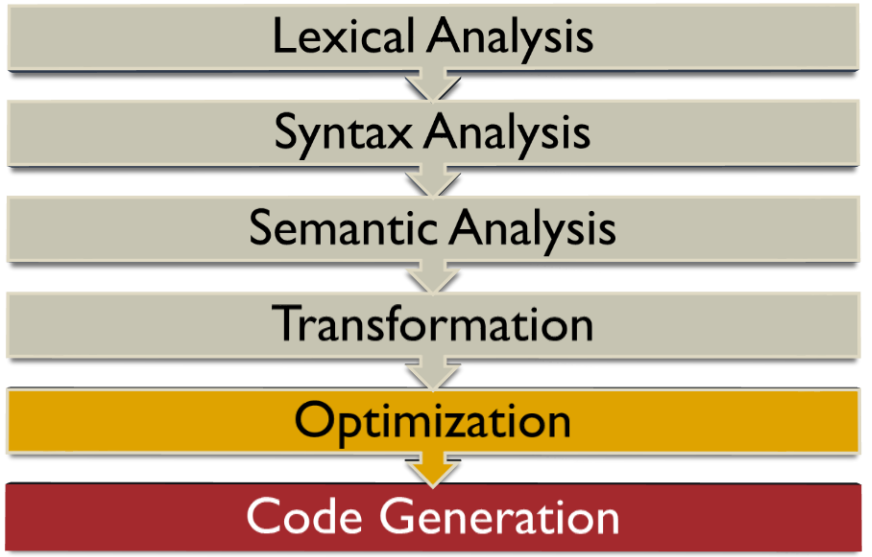





```
x = a + b;

while (y < z) {

  y += x;

}
```

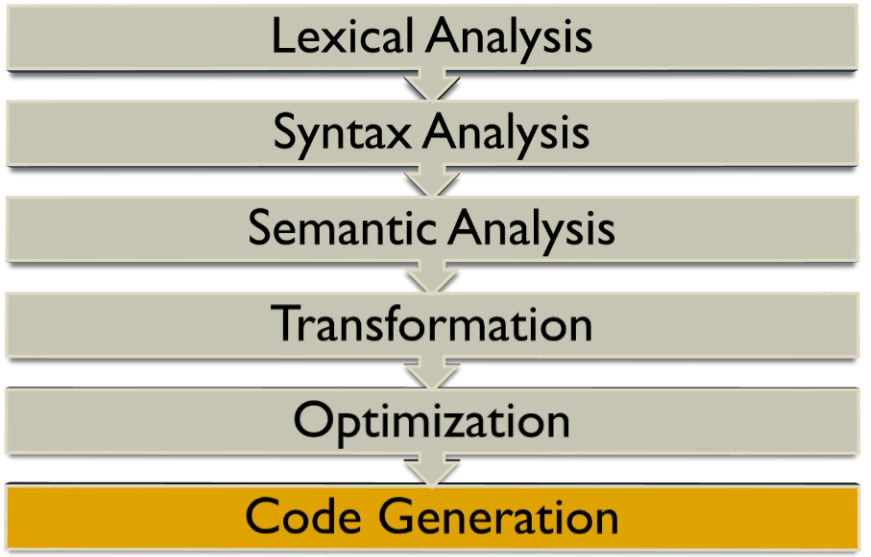
PHASE V – OPTIMIZATION

- Optional phase
  - Usually performed on the transformation's result
- Make the code faster or use less resources
  - Always ensuring that the code remains correct!
- There is no optimal code, only optimized
  - There are contradicting optimization techniques



PHASE VI – CODE GENERATION

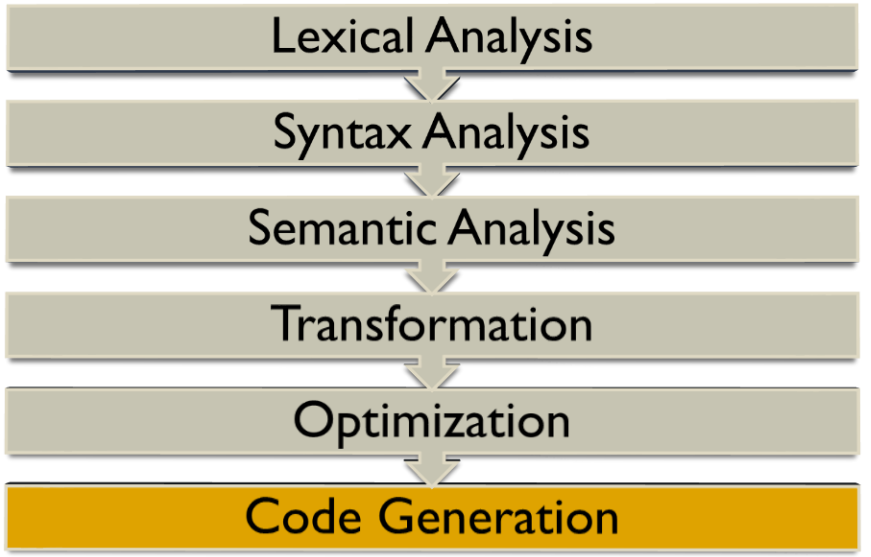
```
add   $1, $2, $3
loop: add $4, $1, $4
slt   $6, $1, $5
beq   $6, loop
-----------------------------------
x := a + b;
while y < z do
       y := y + x;
```




```
x = a + b;
while (y < z) {
  y += x;
}
```

PHASE VI – CODE GENERATION

- Generate code
  - Statements and execution order
  - Not always machine code (transpilers)
- Assembler
  - Responsible for generating machine code
- Linker
  - Responsible for linking compiled modules (if there are any)



TODAY'S AGENDA

I. Textual Languages

II. Introduction to Compilers

III. Lexical Analysis

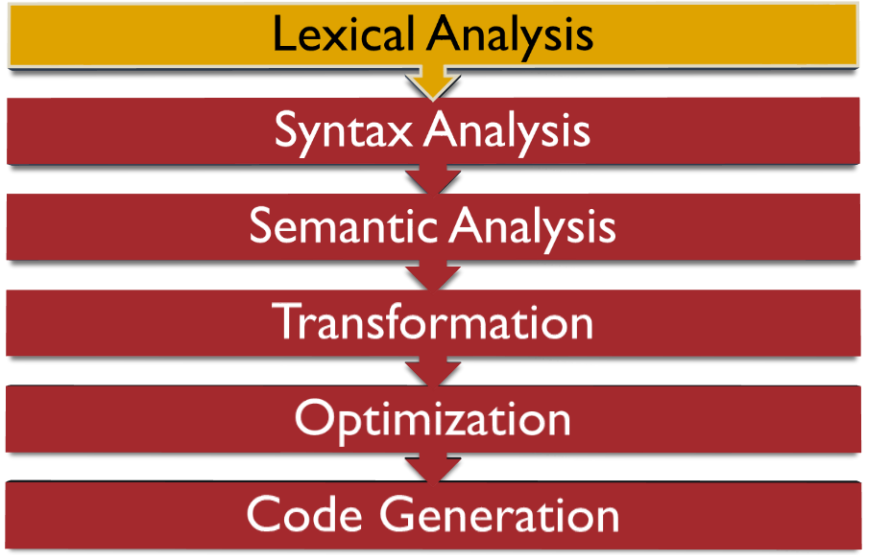
IV. Regular Expressions

V. Formal Languages

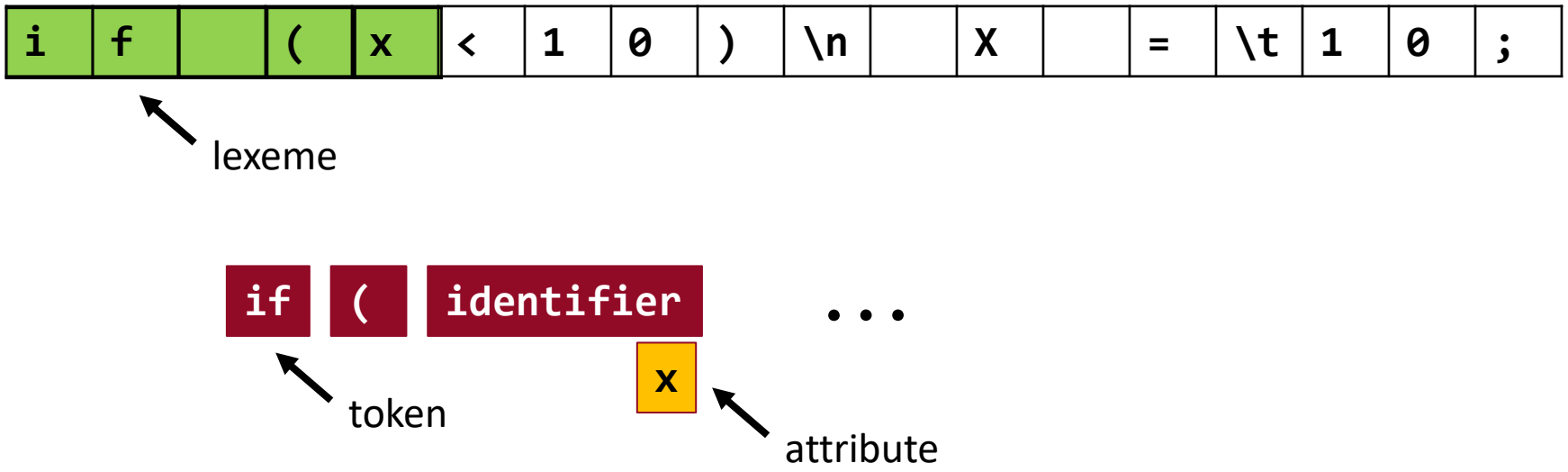
VI. Syntax Analysis (Introduction)

LEXICAL ANALYSIS

- Performed by the *lexer*
- Input = raw text (lexemes)
- Output = tokens (created from lexemes)
  - Tokenization
  - Associating tokens with lexemes
  - Tokens serve as input for the next phase (syntax analysis)

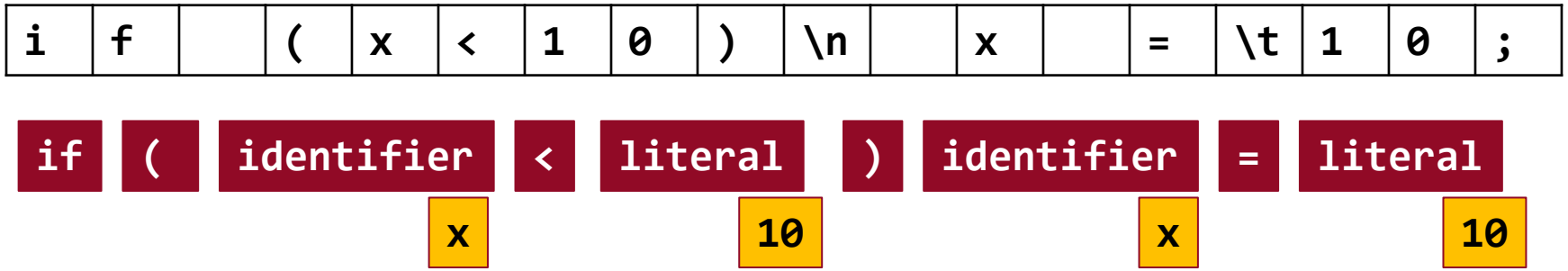


LEXICAL ANALYSIS




```
if (x<10)
 x =     10;
```

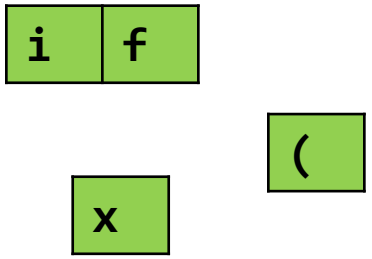
LEXICAL ANALYSIS




```
if (x<10)
 x =    10;
```

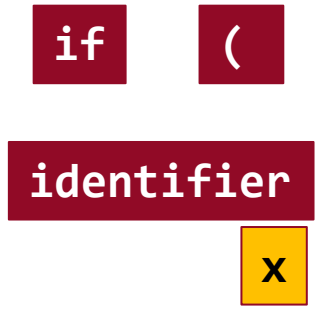
LEXEMES AND TOKENS

- Lexeme
  - Words and symbols which can have meaning in themselves
  - Not every character sequence must be a lexeme (white space, comments)
  - e.g. keywords (*if*, *while*, *for*, etc.), parentheses, semicolon, literal values
- Separating the text into lexemes
  - General algorithm:
    - Search for characters that can start a lexeme
    - Find the longest matching sequence (greedy strategy)
    - for example: *whiletrue* → *identifier* token, not *while* token



LEXEMES AND TOKENS

- Token
  - Created from lexemes, a lexeme is associated with a token
  - Has a type (e.g. T_Identifier) and can have attributes (e.g. "x")
- Tokenization
  - First, the lexer creates lexemes from the raw text
  - Then, tokens are created from the lexemes
- Nomenclature
  - In practice, lexemes are often implied and only tokens are mentioned instead

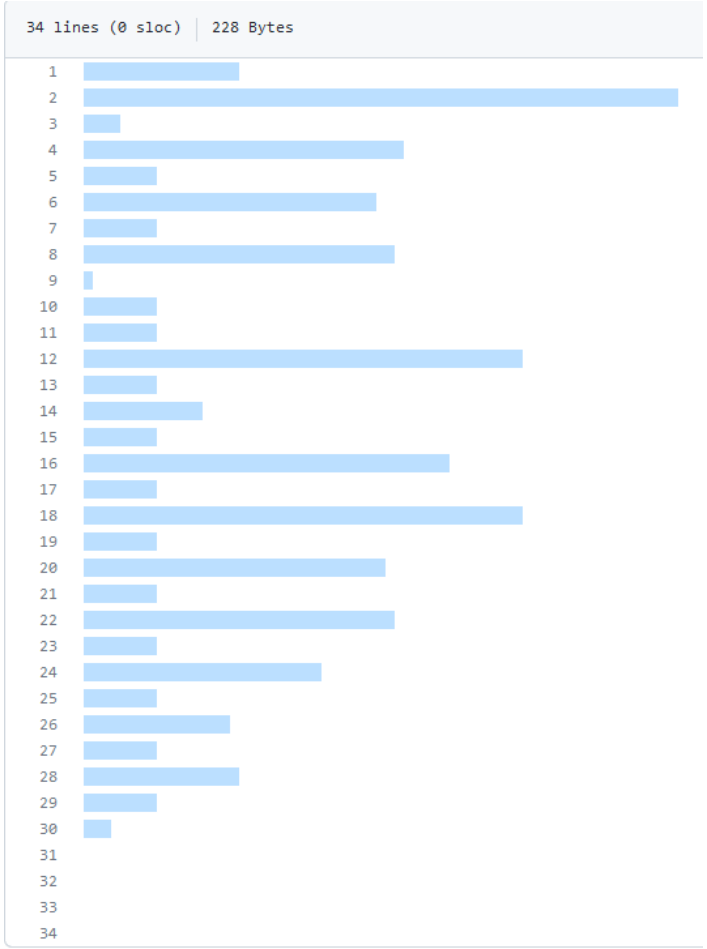


LEXEMES AND TOKENS

- What can be a token?
  - Keywords
  - Operators
  - Delimiters (e.g. semicolon)
  - Identifiers
  - Literal values (numbers, strings, etc.)
- Omitting unnecessary characters (optional)
  - No tokens will be generated from these
  - The language defines which characters (lexemes) are omittable
  - Typically, whitespace characters, comments, etc.

ESOTERIC LANGUAGES – WHEN WE HAVE VERY FEW TOKENS

- Whitespace language
  - April 1st, 2003, Turing-complete
  - Omit every non-whitespace character
- Technical details
  - Stack, heap, 22 statements
  - 3 tokens: [space], [tab], [linefeed]
  - Binary data representation
    - 001110 = [space][space][tab][tab][tab][space][linefeed]



Source: https://github.com/rdebath/whitespace/blob/master/tests/rdebath/helloworld.ws

ESOTERIC LANGUAGES – WHEN WE HAVE VERY FEW TOKENS

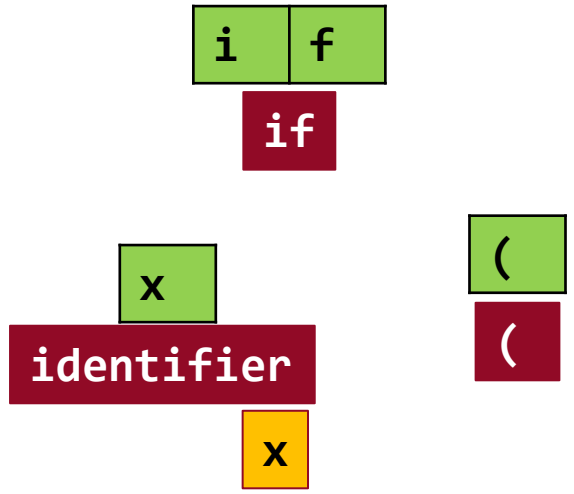
- Brainf***
  - 1993, Turing-complete
  - Omit everything except 8 special characters
- Technical details
  - Operates on an array of memory cells [ ]
  - 8 statements:
    - Move pointer: < >
    - Increase / decrease byte at pointer: + -
    - Output / input byte at pointer: . ,
    - Jump in code: [ ]

```
1.  >++++++++[<+++++++++>-]<.
2.  >++++[<+++++++>-]<+.
3.  +++++++..
4.  +++.
5.  >>++++++[<+++++++>-]<++.
6.  ------------.
7.  >++++++[<+++++++++>-]<+.
8.  <.
9.  +++.
10. ------.
11. --------.
12. >>>++++[<++++++++>-]<+.
```

Source: https://therenegadecoder.com/code/hello-world-in-brainfuck/

LEXEMES AND TOKENS

- A token can have a potentially infinite number of associated lexemes
  - *identifier* → symbol names, infinite combinations of characters
    - e.g. variable or function names
- How can a token be associated with a lexeme?
  - Regular expressions – most often used
  - Grammar rules
  - Mathematically: formal languages, finite automatons



TODAY'S AGENDA

I. Textual Languages

II. Introduction to Compilers

III. Lexical Analysis

IV. Regular Expressions

V. Formal Languages

VI. Syntax Analysis (Introduction)

REGULAR EXPRESSIONS

- Regular expression (regex)
- Often used in pattern recognition
  - Email address, password, bank account, etc. `^[\w\-\.]+@([\w-]+\.)+[\w\-]{2,4}$`
- Multiple notations
  - Mathematical notation
  - Different notations for different implementations
- Interactive regex validators
  - https://regex101.com/

REGULAR EXPRESSIONS – MATHEMATICAL OPERATIONS

- The ε symbol
  - Empty expression
- Operations with regular expressions
  - Union $R_1|R_2$
  - Concatenation R_1R_2
  - Kleene opertor R^*
  - Grouping (R)
- Operation priority from top to bottom

REGULAR EXPRESSIONS IN PRACTICE

- String start (^) and end ($) (anchor)
  - Optional flags at the end (g – global, i – case insensitive, etc.)
- Grouping – (…), logical OR – |
- Set of characters – […]
  - Abbreviations (\d – numeric, \w – alphanumeric, . – any character, etc.)
  - Escaping special characters (e.g. ,\. → dot character)
  - Negation (any character except…) – ^
- Quantifier
  - ? zero or one, * zero or more, + one or more, from N to M – {N,M}, e.g. {2,4}

REGULAR EXPRESSIONS – EXAMPLES

- Fractional number
  - **^-?([0-9]*\.)?[0-9]+$**
- URL
  - **^(https?:\/\/)?([\da-z\.-]+)\.([a-z]{2,4})\/?$**
- Source: https://www.variables.sh/complex-regular-expression-examples/
- Note: when do we need the string start and end anchors?

REGULAR EXPRESSIONS – EXAMPLES (COMMON TOKENS)

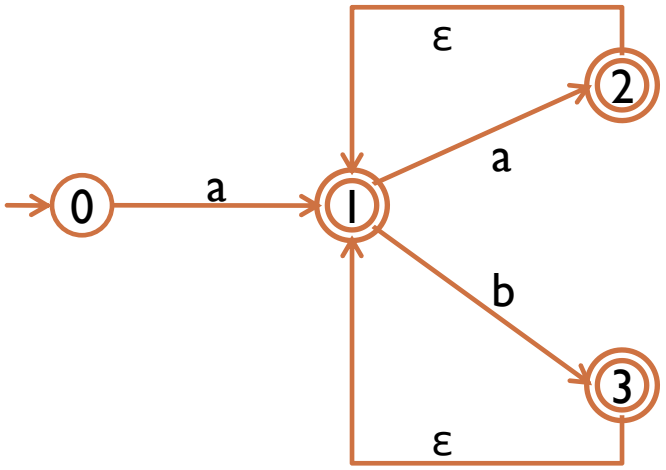
- Identifier (variable name, function name, etc.)
  - **[a-zA-Z_] [a-zA-Z_0-9]***
- Comment (single line)
  - **(\/\/)([^\r\n]*)**
- String
  - **(")([^\r\n]*)(")** or **(")([^\r\n"]*)(")**
  - What is the difference between the above alternatives?

MATHEMATICAL BACKGROUND*

- Finite automaton $M = (Q, \sum, \delta, q_0, F)$, where…
  - Q is a finite, non-empty set – the set of states
  - $\sum$ is a finite, non-empty set – alphabet (possible characters of accepted words)
  - δ is the state transition function (how to transition from one state to another)
  - q_0 is the initial state
  - $F \subseteq Q$ is the set of accepting states (if we stop here, the word is accepted)
- Source: Judit Csima, Katalin Friedl: Languages and automata - note

MATHEMATICAL BACKGROUND*

- Finite automaton from regular expressions
- a(a|b)* (mathematical notation)
- States (Q)
  - ${}_0a_1(a_2|b_3)^*$
- State transitions (δ)
  - $(0,a) \rightarrow 1, (1,a) \rightarrow 2, (1,b) \rightarrow 3, (2,\varepsilon) \rightarrow 1, (3,\varepsilon) \rightarrow 1$
- Initial states (q_0): 0
- Accepting states ($F \subseteq Q$): 1, 2, 3



TODAY'S AGENDA

I. Textual Languages

II. Introduction to Compilers

III. Lexical Analysis

IV. Regular Expressions

V. Formal Languages

VI. Syntax Analysis (Introduction)

TASK

- Describe a calculator using regular expressions! The calculator can perform the four basic arithmetic operations and grouping. For the sake of simplicity, only single digit positive natural numbers are supported.
- Solution
  - [0-9]
  - [0-9][\+\-*\/][0-9]
  - [0-9]([\+\-*\/][0-9])+
  - [0-9]([\+\-*\/]\(?[0-9]\)?)+
  - **Is this good?**

SHORTCOMINGS OF REGULAR EXPRESSIONS

- In the previous task, we could not handle recursion
  - For example, when beginning curly braces, they must also end somewhere!
- The expressive power of regular expressions is sometimes enough…
- …but in more complex cases it is usually not
  - Nested expression blocks
  - Correctly parenthesized expressions
  - etc.
- We need a stronger formalism!

FORMAL LANGUAGES (INFORMALLY)

- Alphabet – an arbitrary, non-empty set of characters
- Letter – an element of the alphabet
- Word – a finite length sequence of letters
- Language – a (finite or infinite) set of words

CONTEXT-FREE (CF) GRAMMARS

- Structure of a CF grammar:
  - Production rules
  - Nonterminal symbols (variables)
    - The intermediate nodes of the syntax tree
  - Terminal symbols (words)
    - The leaves of the syntax tree (tokens, see lexer)
  - Start symbol (initial variable)
    - Typically, the left side of the first rule (but not necessarily)

$E \rightarrow 0 \mid 1 \mid \ldots$

$E \rightarrow E\ Op\ E$

$E \rightarrow (E)$

$Op \rightarrow +$

$Op \rightarrow -$

$Op \rightarrow *$

$Op \rightarrow /$

CONTEXT-FREE (CF) GRAMMARS

- The grammar describes a formal language
- Context-free (CF)
  - On the left side of a rule there is exactly one nonterminal
  - On the right side, terminal and / or nonterminal sequence
  - CF grammars (languages) are often used in practice
- The example describes a calculator grammar
  - Positive numbers (one digit only)
  - 4 basic arithmetic operations
  - Grouping

$E \rightarrow 0 \mid 1 \mid \ldots$

$E \rightarrow E\ Op\ E$

$E \rightarrow (E)$

$Op \rightarrow +$

$Op \rightarrow -$

$Op \rightarrow *$

$Op \rightarrow /$

CF GRAMMARS – NOTATION

$E \rightarrow 0 \mid 1 \mid \ldots$

$E \rightarrow E\ Op\ E$

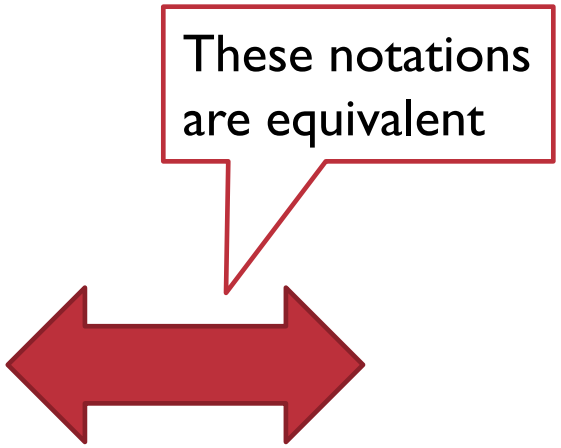
$E \rightarrow (E)$

$Op \rightarrow +$

$Op \rightarrow -$

$Op \rightarrow *$

$Op \rightarrow /$



$E \rightarrow 0 \mid 1 \mid \ldots \mid E\ Op\ E \mid (E)$

$Op \rightarrow + \mid - \mid * \mid /$

CF GRAMMARS – DERIVATION

3 * (1 + 2)

Text to be derived

Leftmost derivation

E

⇒ E Op E

⇒ 3 Op E

⇒ 3 * E

⇒ 3 * (E)

⇒ 3 * (E Op E)

⇒ 3 * (1 Op E)

⇒ 3 * (1 + E)

⇒ 3 * (1 + 2)

E → 0|1|… | E Op E | (E)

Op → + | - | * | /

The grammar (described by rules)

CF GRAMMARS – DERIVATION

3 * (1 + 2)

Text to be derived

Rightmost derivation

E

⇒ E Op E

⇒ E Op (E)

⇒ E Op (E Op E)

⇒ E Op (E Op 2)

⇒ E Op (E + 2)

⇒ E Op (1 + 2)

⇒ E * (1 + 2)

⇒ 3 * (1 + 2)

E → 0|1|… | E Op E | (E)

Op → + | - | * | /

The grammar (described by rules)

CF GRAMMARS – DERIVATION

E

⇒ E Op E

⇒ 3 Op E

⇒ 3 * E

⇒ 3 * (E)

⇒ 3 * (E Op E)

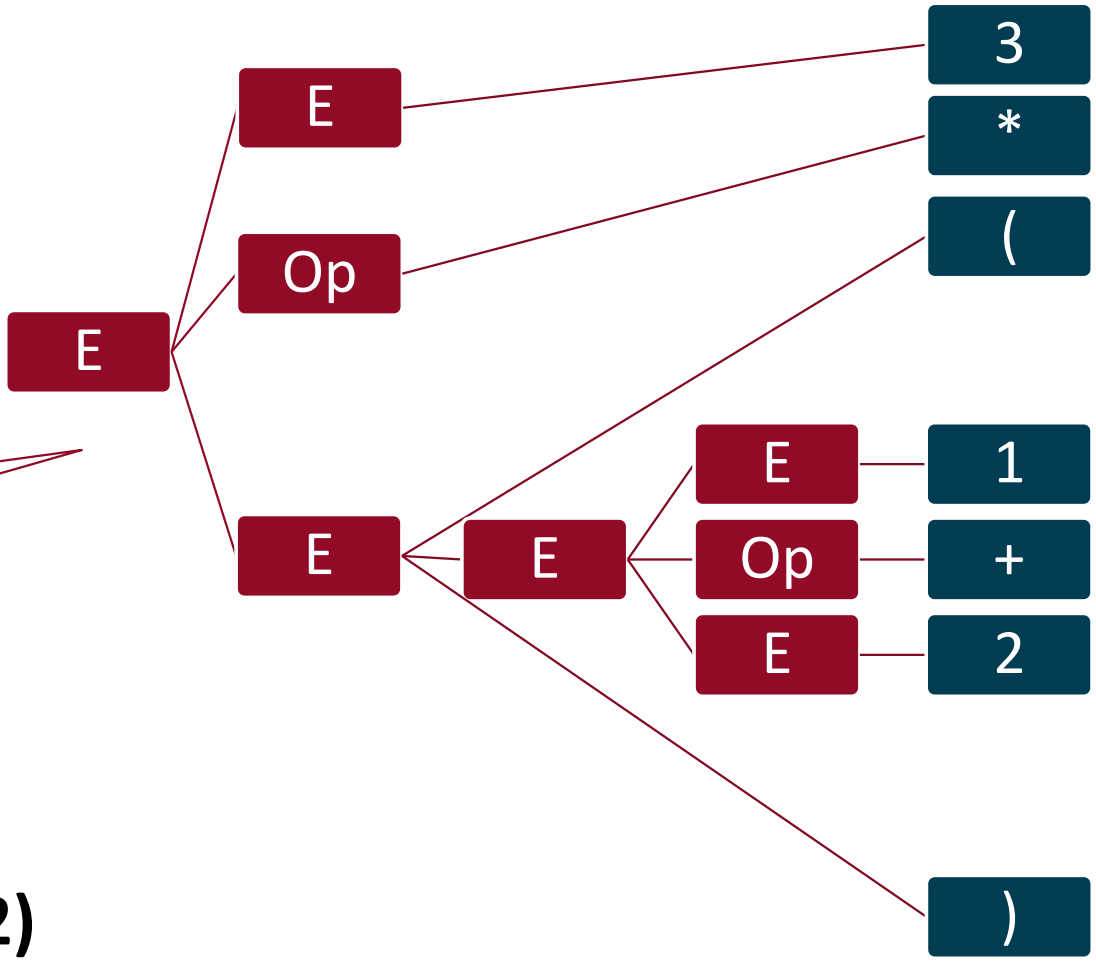
⇒ 3 * (1 Op E)

⇒ 3 * (1 + E)

⇒ 3 * (1 + 2)

Leftmost derivation

Parse tree



3 * (1 + 2)

CF GRAMMARS – DERIVATION

- Leftmost and rightmost derivation
  - The only difference is the order
  - Always "unfolding" (inserting) the left or rightmost non-terminal
  - In practice, we usually use left-hand derivation
- Parse tree
  - Data structure generated during the derivation
- CF grammar test tool
  - https://checker5965.github.io/toc.html

CF GRAMMARS – EXAMPLE (GRAMMAR)

```
public class MyClass {
  private int x;
  public string s;
}
```

Lexer gives the *id* token, we won't detail it here

S → V class id { A }

A → AA | V T idE | ε

V → public | private | …

T → int | string | …

E → ;

Class → Vis class id { Att }

Att → AttAtt | Vis Typ idEoS | ε

Vis → public | private | …

Typ → int | string | …

EoS → ;

CF GRAMMARS – EXAMPLE (DERIVATION)

```
public class MyClass {
  private int x;
  public string s;
}
```

$S \rightarrow V\ \text{class id}\ \{\ A\ \}$

$A \rightarrow AA \mid V\ T\ \text{id}\ E \mid \varepsilon$

$V \rightarrow \text{public} \mid \text{private} \mid \ldots$

$T \rightarrow \text{int} \mid \text{string} \mid \ldots$

$E \rightarrow\ ;$

S

$\Rightarrow V\ \text{class id}\ \{\ A\ \}$

$\Rightarrow \text{public class id}\ \{\ A\ \}$

$\Rightarrow \text{public class id}\ \{\ AA\ \}$

$\Rightarrow \text{public class id}\ \{V\ T\ \text{id}\ E\ A\ \}$

$\Rightarrow \text{public class id}\ \{\text{private}\ T\ \text{id}\ E\ A\ \}$

$\Rightarrow \text{public class id}\ \{\text{private int id}\ E\ A\ \}$

$\Rightarrow \text{public class id}\ \{\text{private int id;}\ A\ \}$

$\Rightarrow \text{public class id}\ \{\text{private int id;}\ V\ T\ \text{id}\ E\ \}$

$\Rightarrow \ldots$

BACHUS-NAUR FORM (BNF)

- A common, practical notation to describe CF grammars
- Nonterminal symbols: between <…>
- Terminal symbols: **simple characters**
  - Or between apostrophes: **'simple characters'**
- In production rules, instead of → we use **::=**
- Production rules are sometimes ended with '.' or ';'
  - There are a few variants of BNF in practice

BACHUS-NAUR FORM (BNF)

- Example
  - **<E> ::= <digit> | <E> <Op> <E> | (<E>)**
  - **<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9**
  - **<Op> ::= + | - | * | /**

E → 0|1|… | E Op E | (E)

Op → + | - | * | /

EXTENDED BACHUS-NAUR FORM (EBNF)

- A more convenient, more practical notation for describing CF grammars
- Nonterminal symbols: **<…>** is omitted
- Terminal symbols: always between apostrophes
  'simple characters'
- Grouping: **(…)**
- Optional element: **[…]** (same as **?** in regular expressions)
- Repeat zero or more times: *****
- Repeat one or more times: **+**

EXTENDED BACHUS-NAUR FORM (EBNF)

- Example
  - **E ::= (digit)+ | E Op E | ‘(‘ E ‘)’**
  - **digit ::= ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’**
  - **Op ::= ‘+’ | ‘-’ | ‘*’ | ‘/’**

There can be more than one digits; warning: this is not a regular notation!

E → (0|1|…)+ | E Op E | (E)

Op → + | - | * | /

TODAY'S AGENDA

I. Textual Languages

II. Introduction to Compilers

III. Lexical Analysis

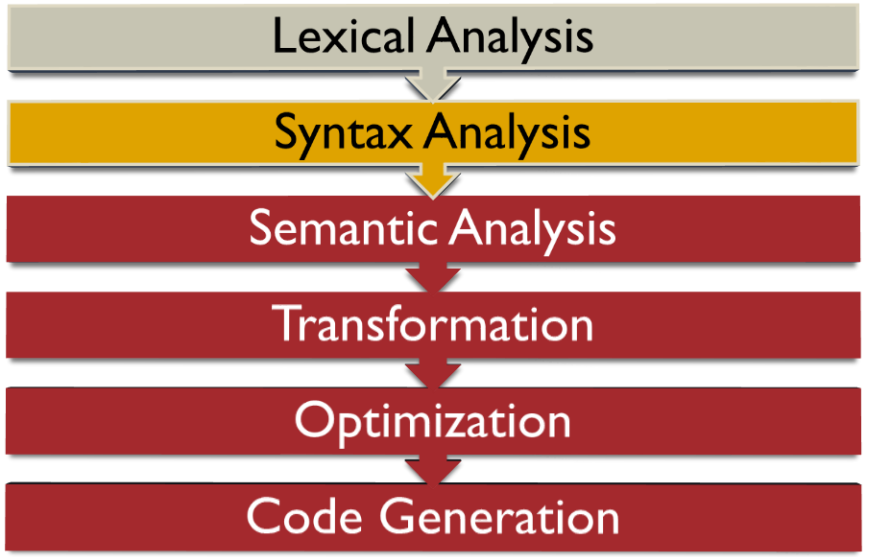
IV. Regular Expressions

V. Formal Languages

VI. Syntax Analysis (Introduction)

SYNTAX ANALYSIS

- Performed by the *parser*
- Input: tokens produced by lexical analysis
- Output: (tree) structure built from the tokens
  - Efficient processing in the later phases
  - Detecting faulty syntax – syntax errors
- Nowadays, the structure typically determines the meaning of the code
  - Arrange the code into a structure that is "readable"
  - During semantic analysis, check other semantic constraints

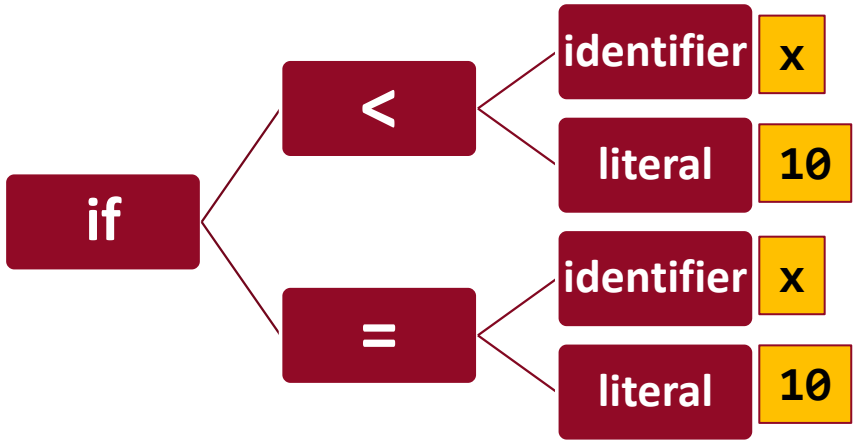


SYNTAX ANALYSIS

```
if (x<10)
 x = 10;
```

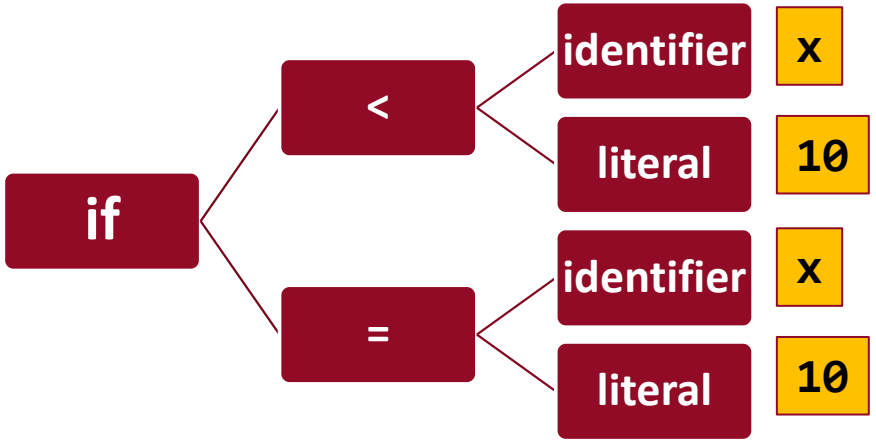
i	f		(	x	<	1	0	)	\n		x		=	\t	1	0	;

if	(	identifier	<	literal	)	identifier	=	literal
		x		10		x		10



SYNTAX ANALYSIS

- Assuming a CF grammar
  - Build the syntax tree
  - In theory, it could be another data structure…
  - …but in practice, we almost always work with syntax trees
  - Parse tree: the exact tree built during derivation
- How can the syntax tree be built?
  - Using specific algorithms
  - Details in the next lectures



CONCRETE AND ABSTRACT SYNTAX TREE

- *Concrete Syntax Tree / CST*
  - The parse tree is sometimes called a concrete syntax tree
  - It contains the exact derivation of the text, including all keywords, etc.
- *Abstract Syntax Tree / AST*
  - Contains only the essential part of the structure
  - Remove unnecessary parts from the tree
    - e.g. syntactic constraints, keywords
  - The tree hierarchy provides grouping and structure
- In practice, both CST-s and AST-s are used

THANK YOU FOR YOUR ATTENTION!