Concepts of Programming Languages

Week 3: Grammar, Lexical and syntax analysis

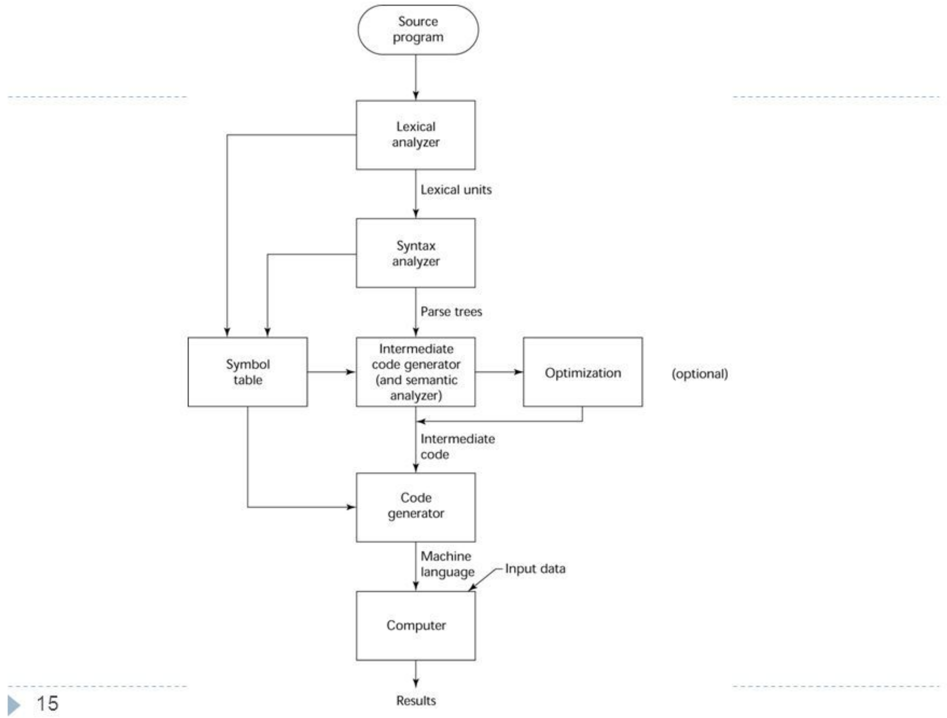
15 March 2023

Lecture plan

- Lexeme and token
- Syntax and semantics
- Regular expressions
- Grammars
- Derivations
- Compiling

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Compiling flowchart



Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Lexeme and token

- A **lexeme** is the lowest level syntactic unit of a language
- A lexical token or simply token is a string with an assigned and thus identified meaning. A token is a category of lexemes.
- Common token names are
  - identifier: names the programmer chooses;
  - keyword: names already in the programming language;
  - separator (also known as punctuators): punctuation characters and paired-delimiters;
  - operator: symbols that operate on arguments and produce results;
  - literal: numeric, logical, textual, reference literals;
  - comment: line, block (Depends on the compiler if compiler implements comments as tokens otherwise it will be stripped).

Lexeme and token

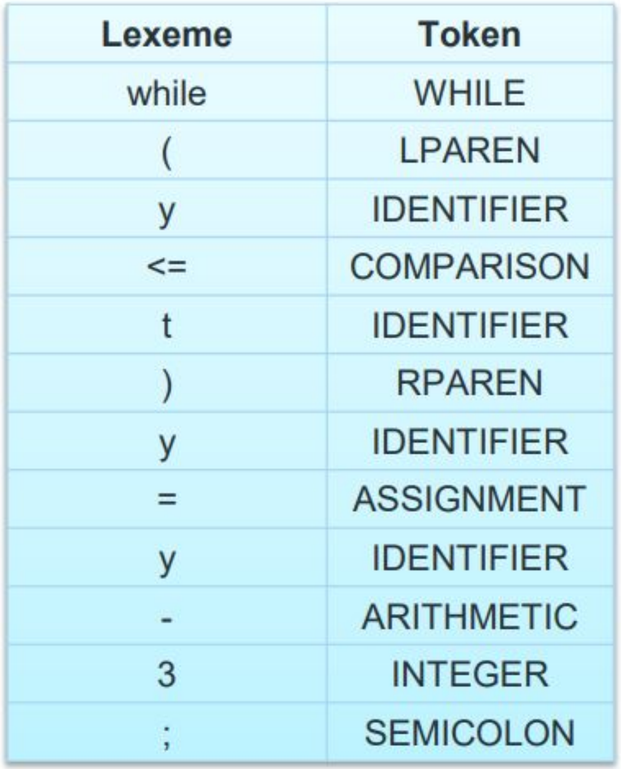
Lexemes	*Tokens*
`index`	identifier
`=`	equal_sign
`2`	int_literal
`*`	mult_op
`count`	identifier
`+`	plus_op
`17`	int_literal
`;`	semicolon

A subset of lexemes defines a language token.

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Lexeme and token

while (y >= t) y = y - 3 ;



Lexeme	Token
while	WHILE
(	LPAREN
y	IDENTIFIER
<=	COMPARISON
t	IDENTIFIER
)	RPAREN
y	IDENTIFIER
=	ASSIGNMENT
y	IDENTIFIER
-	ARITHMETIC
3	INTEGER
;	SEMICOLON

https://adys.nku.edu.tr/OpenCourse/Course/Programlama_Dilleri_Prensipleri/27702

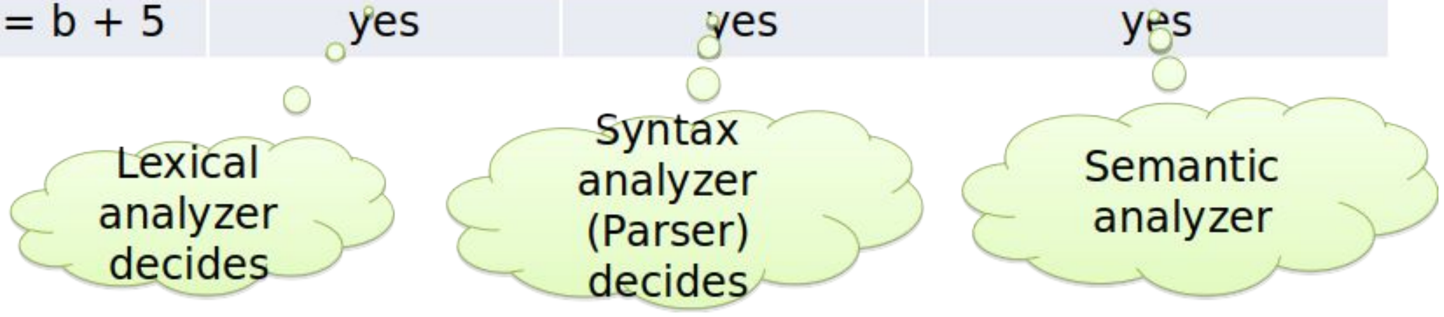
Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Lexeme, Syntax and semantics

- **Lexeme:** the lowest level syntactic unit of a language
- **Syntax**: the form or structure of the expressions, statements, and program units (the arrangement of words as elements in a sentence)
- **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)

Lexical, Syntax and Semantics analyzer

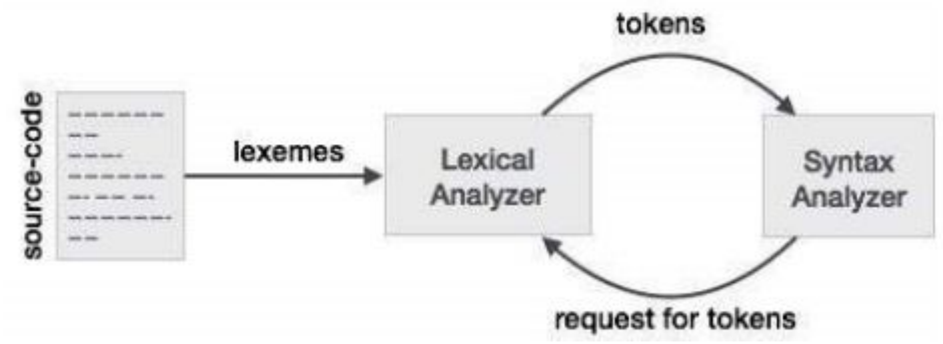
Expression	Is lexically valid?	Is syntactically valid?	Is semantically valid?
inta b + 5 =	no	no	no
int = a b + 5	yes	no	no
int 5 = a + b	yes	yes	no
int a = b + 5	yes	yes	yes



Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Lexical and Syntactic Analysis

- Two steps to discover the syntactic structure of a program
  - ***Lexical analysis (Scanner):*** to read the input characters and output a sequence of tokens
  - ***Syntactic analysis (Parser):*** to read the tokens and output a parse tree and report syntax errors if any



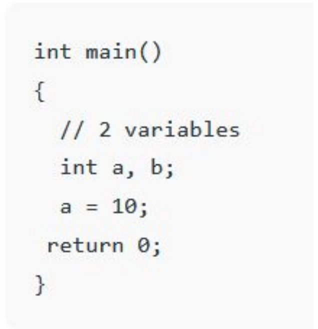
https://people.cs.vt.edu/prsardar/classes/cs3304-Spr19/lectures/CS3304-9-LanguageSyntax-2.pdf

Reasons to Separate Lexical and Syntax Analysis

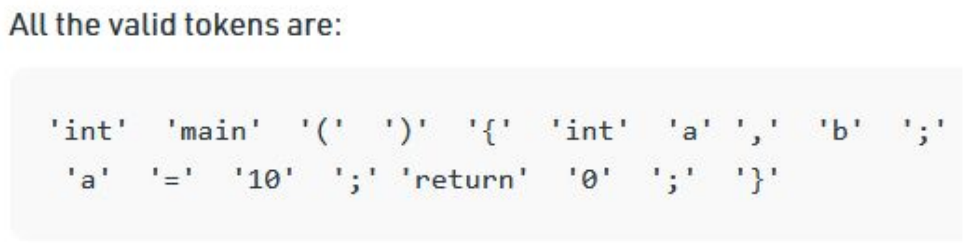
- **Simplicity** - less complex approaches can be used for lexical analysis; separating them simplifies the parser
- **Efficiency** - separation allows optimization of the lexical analyzer
- **Portability** - parts of the lexical analyzer may not be portable, but the parser is always portable

Lexical analyzer

- It is also called a **scanner**.
- Lexical analyzer is a pattern matcher for strings
    - Regular Expression
- It takes the output of the preprocessor (which performs file inclusion and macro expansion) as the input which is in a pure high-level language.
- It reads the characters from the source program and groups them into lexemes (sequence of characters that “go together”).
- Each lexeme corresponds to a token. Tokens are defined by *regular expressions* which are understood by the lexical analyzer.
- It also removes lexical errors (e.g., erroneous characters), comments, and white space.



```
int main()
{
  // 2 variables
  int a, b;
  a = 10;
 return 0;
}
```



All the valid tokens are:

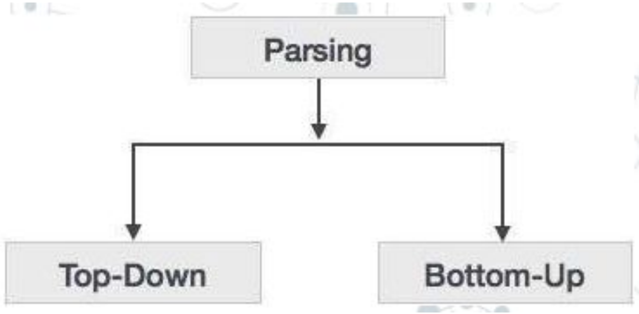
```
'int'  'main'  '('  ')'  '{'  'int'  'a' ','  'b'  ';'
 'a'  '='  '10'  ';' 'return'  '0'  ';'  '}'
```

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Syntax analyzer (parser)

- Goals of the parser, given an input program:
    - Find all syntax errors; for each, produce an appropriate diagnostic message and recover quickly
    - Produce the parse tree, or at least a trace of the parse tree, for the program

- Two categories of parsers
    - **Top down:** produce the parse tree, beginning at the root
        - Order is that of a leftmost derivation
        - Traces or builds the parse tree in preorder
    - **Bottom up:** produce the parse tree, beginning at the leaves
        - Order is that of the reverse of a rightmost derivation
- Useful parsers look only one token ahead in the input



Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Infix, prefix, postfix

Infix notation

- Summing A and B:
  - A+B
- Multiplying A and B:
  - A*B
- Operators (*,+,-,/) are written between operands (A,B)

Prefix notation

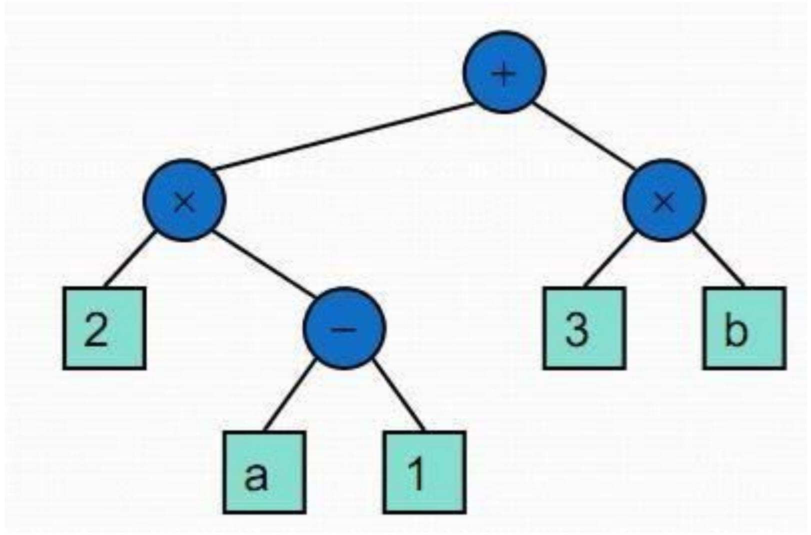
- Summing A and B:
  - + A B
- Multiplying A and B:
  - * A B
- Operator is written first

Postfix notation

- Summing A and B:
  - A B +
- Multiplying A and B:
  - A B *
- Operator is written after

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Expression tree



Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Context-Free Grammars and BNF

- Context-Free Grammars
  - Developed by ***Noam Chomsky*** in the mid-1950s (described natural languages!)
  - 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 Algol 58
  - BNF is equivalent to context-free grammars
  - BNF is a metalanguage to describe another language

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

<assign> -> <var> = <expression> : abstraction for variable assignment

x = 5;

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

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

  – Examples of BNF rules:

```
<ident_list> → identifier | identifier, <ident_list>
<if_stmt> → if <logic_expr> then <stmt>
```

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

BNF Rules and derivations

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

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

program may contain many statements

one or many statements and ; between them

statement can be a variable assignment

variable only can be a or b or c or d

expression only sum or subtraction!

term can be var or constant value

An Example Grammar

◎ **stmts**: program commands
◎ **stmt**: one command
◎ **var**: variable
◎ **expr**: expression
◎ **term**: a term
◎ **const**: constant (such as 25)

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Derivations

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

An Example Grammar

one statement
(one line code)

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

a is selected for expression

selected one of the optionals

An Example Derivation

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Derivations

<assign> → <id> = <expr>

<id> → A|B|C

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

A = B * (A + C) We can run this with such a grammar

A = B + C

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Derivations

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

A = B * (A + C)

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

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Derivation

- **Left-most Derivation**

  If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation. The sentential form derived by the left-most derivation is called the left-sentential form.

- **Right-most Derivation**

  If we scan and replace the input with production rules, from right to left, it is known as right-most derivation. The sentential form derived from the right-most derivation is called the right-sentential form.

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Parse Tree

- A parse Tree is a labeled ***tree representation of a derivation*** that filters out the order in which productions are applied to replace nonterminals.
  - The interior nodes are labeled by nonterminals
  - The leaf nodes are labeled by terminals
  - The start symbol of the derivation becomes the root of the parse tree

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Parse Tree

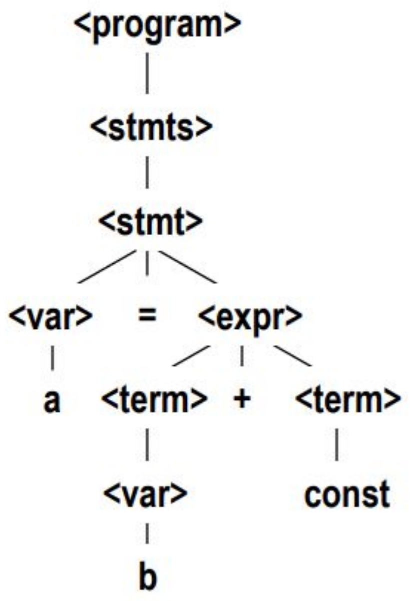
Grammar

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

Derivation

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

Parse Tree



Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Parse Tree

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

A = B * (A + C)

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

=> A = <expr>

=> A = <id> * <expr>

=> A = B * <expr>

=> A = B * (<expr>)

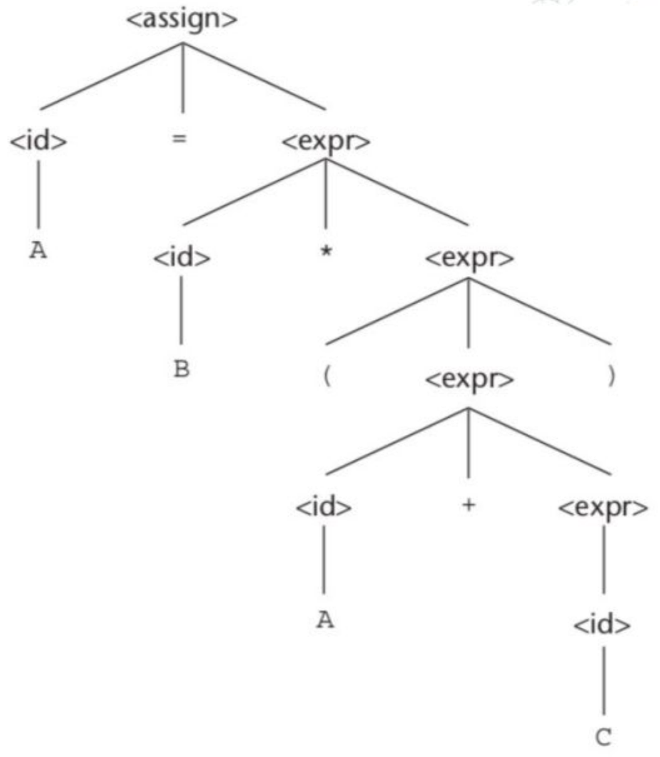
=> A = B * (<id> + <expr>)

=> A = B * (A + <expr>)

=> A = B * (A + <id>)

=> A = B * (A + C)

Parse Tree



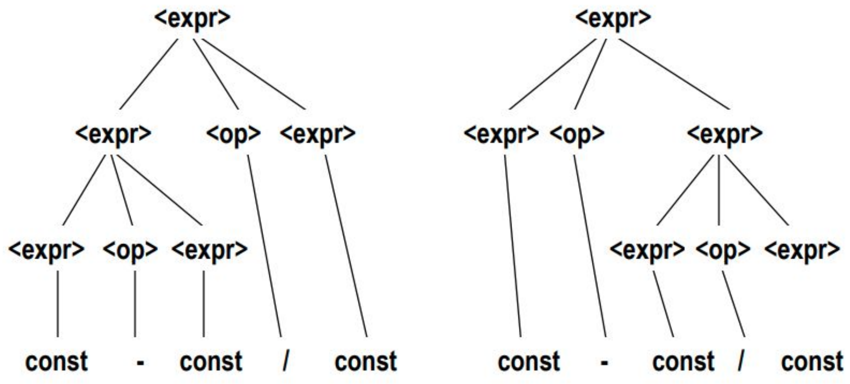
Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Ambiguity in Grammars

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

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

25 - 9 / 3



https://www.iare.ac.in/sites/default/files/PPT/PPL_ppt%27s_5units_2016-17_0.pdf

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

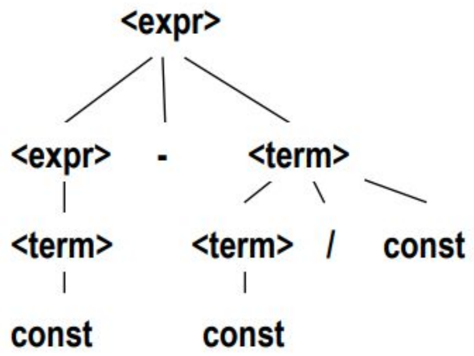
Ambiguity in Grammars

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

old (ambigious)

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

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



Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Ambiguity in Grammars

A = B + C * A

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

ambiguity

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

NO ambiguity

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

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}
```

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Extended BNF

- BNF

```
<expr> → <expr> + <term>
       | <expr> - <term>
       | <term>
<term> → <term> * <factor>
       | <term> / <factor>
       | <factor>
```

- EBNF

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

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Semantics

- Context-free grammars (CFGs - BNF) cannot describe all of the syntax of programming languages
- Categories of constructs that are trouble:
  - Context-free, but cumbersome (e.g., types of operands in expressions)
  - Non-context-free (e.g., variables must be declared before they are used)

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Semantics

- Rules that cannot or are very difficult to define with BNF are in the category of language rules called static semantic rules.
- Since the analysis of these rules is done at compile time, it is called Static semantics.
- Due to the inability to define static semantics with BNF, several more powerful mechanisms have been devised for this task.
- Attribute grammars (AGs) have additions to CFGs to carry some semantic info on parse tree nodes
- Primary value of AGs:
  - Static semantics specification
  - Compiler design (static semantics checking)

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Summary

- Lexical analyzer uses regular expressions and extracts the lexemes
- The subset of lexemes defines the token
- Syntax is the form or structure of the expressions, statements.
- Backus normal form (**BNF**) is a metasyntax notation for context-free grammars (used to describe the syntax of languages)
- Some variants such as EBNF, ABNF (augmented Backus–Naur form)

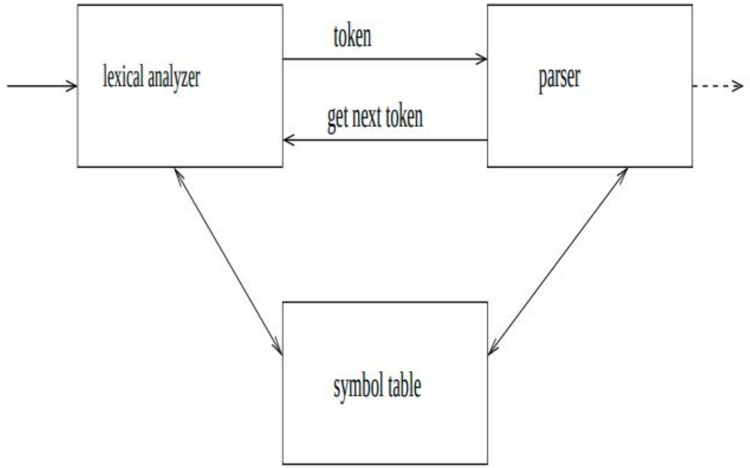
Part-2

Lexical and syntactic analysis and compiling

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Lexical analyzer (Scanner)

- In the early days of compilers, lexical analyzers would typically process an entire source program file and then generate tokens and lexemes files.
- Lexical analyzers in current compilers are *subroutines* that find the next lexeme in the source code implemented as input, determine its symbol, and return it to the syntax analyzer



Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Compiling

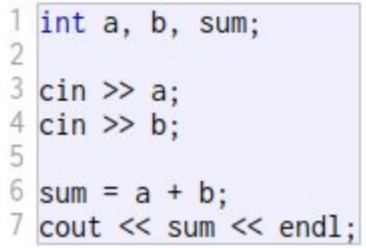
- Lexical analysis
- Syntax analysis
- Semantic analysis
- Intermediate code generator
- Code optimizer
- Code generator $\longrightarrow$ Conversion from hardware-independent intermediate code to machine code of target hardware

Compiling C++ code

What is a compiler?

Computers understand only one language and that language consists of sets of instructions made of ones and zeros. This computer language is appropriately called ***machine language***.

Programming a computer directly in machine language using only ones and zeros is very tedious and error prone. To make programming easier, high level languages have been developed. High level programs also make it easier for programmers to inspect and understand each other's programs easier.



```
int a, b, sum;

cin >> a;
cin >> b;

sum = a + b;
cout << sum << endl;
```

C++ is designed to be a compiled language, meaning that it is generally translated into machine language that can be understood directly by the system, making the generated program highly efficient.

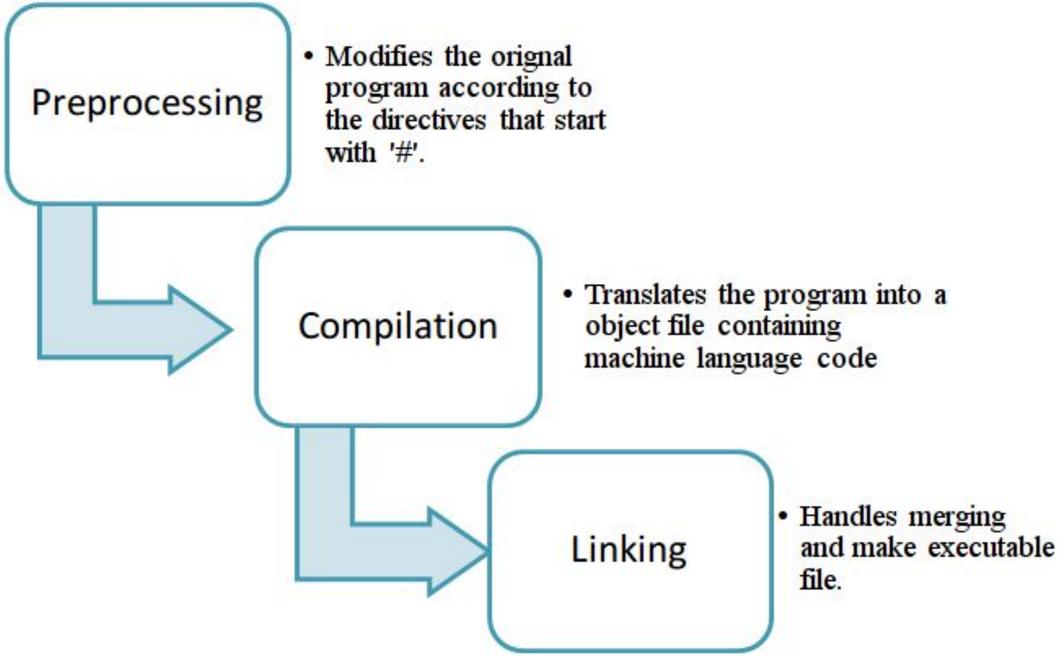
https://cplusplus.com/doc/tutorial/introduction/

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Compiling C++ code

g++ command is a GNU c++ compiler invocation command, which is used for preprocessing, compilation, assembly and linking of source code to generate an executable file.

The compiler converts an executable code into a form that the computer can run directly. This form is called machine language.

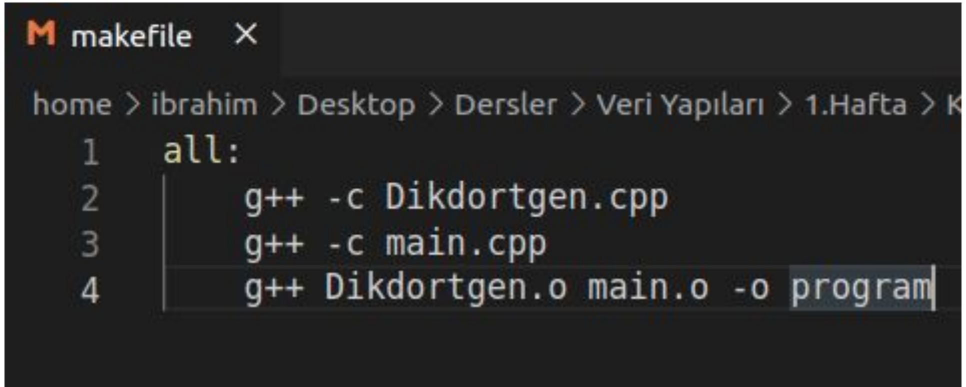


- Modifies the orignal program according to the directives that start with '#'.
- Translates the program into a object file containing machine language code
- Handles merging and make executable file.

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

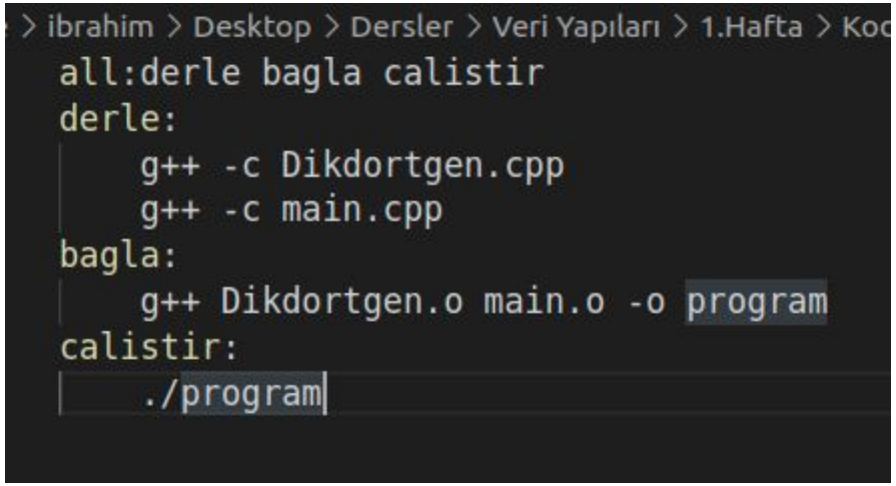
Compiling C++ code

A *makefile* is a text file that contains instructions for how to compile and link (or *build*) a set of source code files. A program (often called a *make* program) reads the makefile and invokes a compiler, linker, and possibly other programs to make an executable file. The Microsoft program is called NMAKE.


makefile

home > ibrahim > Desktop > Dersler > Veri Yapıları > 1.Hafta > K

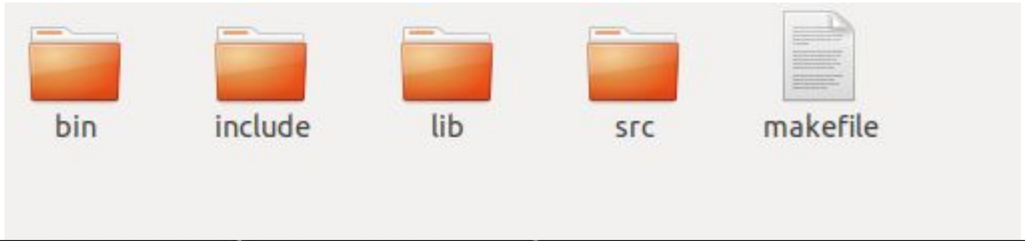
```
all:
	g++ -c Dikdortgen.cpp
	g++ -c main.cpp
	g++ Dikdortgen.o main.o -o program
```

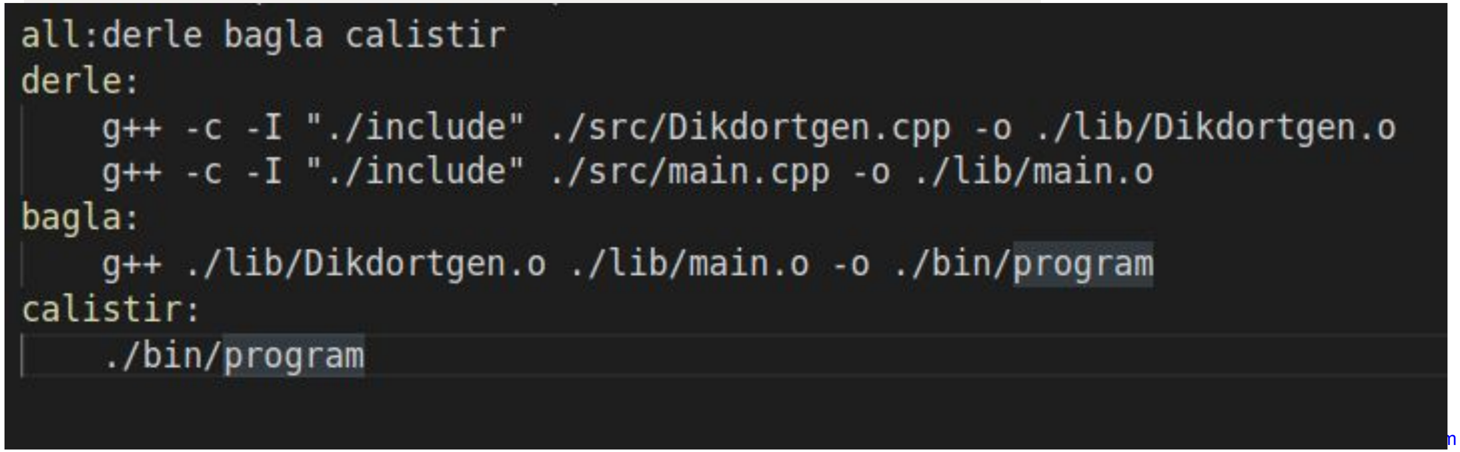

> ibrahim > Desktop > Dersler > Veri Yapıları > 1.Hafta > Koc

```
all:derle bagla calistir
derle:
	g++ -c Dikdortgen.cpp
	g++ -c main.cpp
bagla:
	g++ Dikdortgen.o main.o -o program
calistir:
	./program
```

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Compiling C++ code





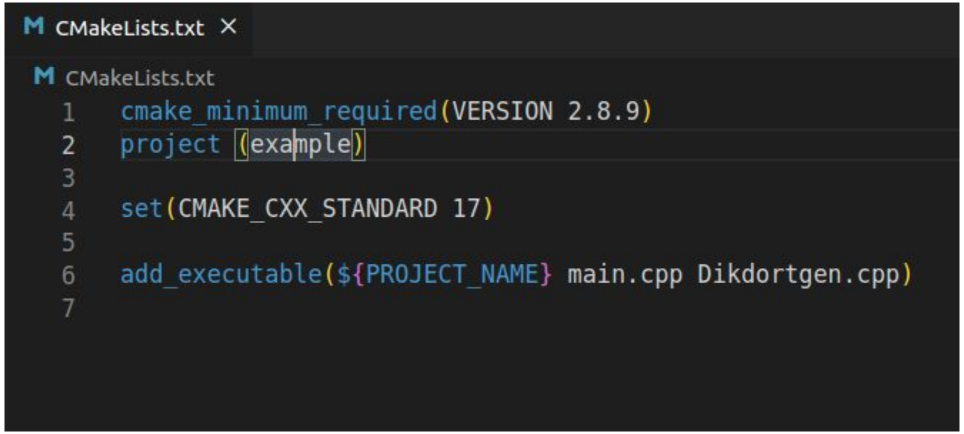
```
all:derle bagla calistir
derle:
    g++ -c -I "./include" ./src/Dikdortgen.cpp -o ./lib/Dikdortgen.o
    g++ -c -I "./include" ./src/main.cpp -o ./lib/main.o
bagla:
    g++ ./lib/Dikdortgen.o ./lib/main.o -o ./bin/program
calistir:
    ./bin/program
```

n Delibaşoğlu
Department of Software Engineering
Sakarya University

Compiling C++ code

Cmake is an open-source, cross-platform family of tools designed to build, test and package software.

CMake is used to control the software compilation process using simple platform and compiler independent configuration files, and generate native makefiles and workspaces that can be used in the compiler environment of your choice.



```
cmake_minimum_required(VERSION 2.8.9)
project (example)

set(CMAKE_CXX_STANDARD 17)

add_executable(${PROJECT_NAME} main.cpp Dikdortgen.cpp)

```

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

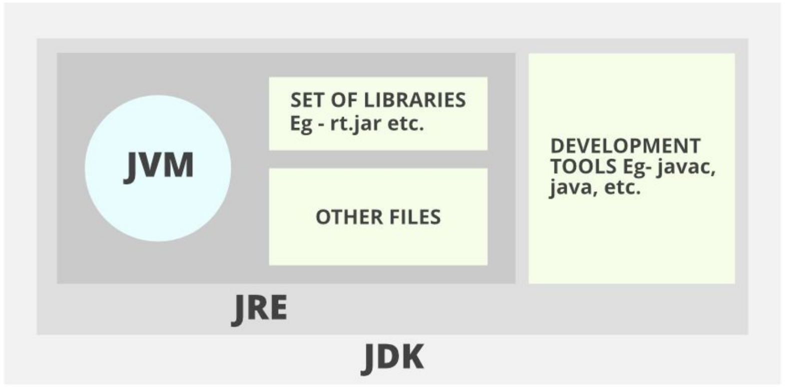
Compiling Java code

- Java is a platform-independent programming language, doesn’t work on the one-step compilation. Instead, it involves a two-step execution, first through an OS-independent compiler; and second, in a virtual machine (JVM) which is custom-built for every operating system.
    - Firstly, the source code is transformed into a machine-independent encoding, known as Bytecode.
    - The class files are passed to the JVM and then goes through three main stages before the final machine code is executed.
        - ClassLoader
        - Bytecode Verifier
        - Just-In-Time Compiler

https://www.geeksforgeeks.org/compilation-execution-java-program/

Compiling Java code

- **JDK** (Java Development Kit) is a Kit that provides the environment to **develop and execute(run)** the Java program.
- **JRE** (Java Runtime Environment) is an installation package that provides an environment to **only run(not develop)** the java program(or application)onto your machine
- **JVM** is a very important part of both JDK and JRE because it is contained or inbuilt in both. Whatever Java program you run using JRE or JDK goes into JVM and JVM is responsible for executing the java program line by line, hence it is also known as an ***interpreter*.**

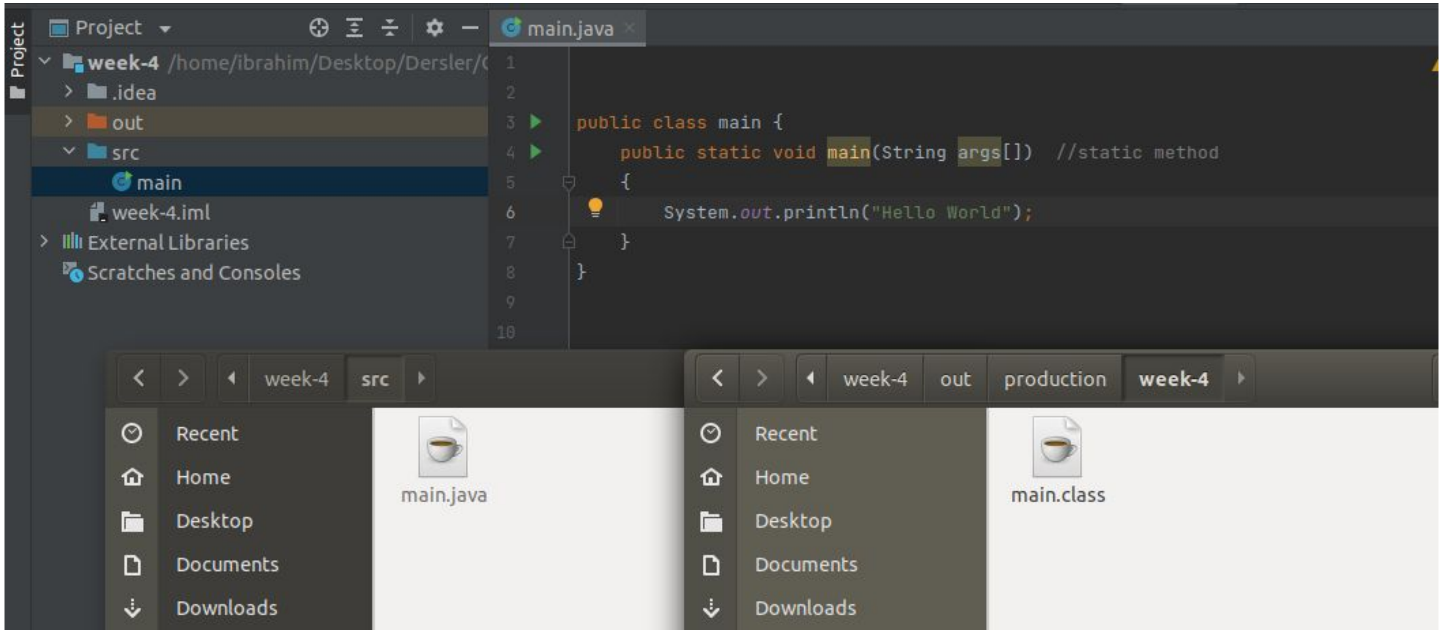

JVM
SET OF LIBRARIES Eg - rt.jar etc.
OTHER FILES
DEVELOPMENT TOOLS Eg- javac, java, etc.
JRE
JDK

https://www.geeksforgeeks.org/differences-jdk-jre-jvm/

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Compiling Java code

main.java is the source code, **Java class file (main.class)** is a file containing Java bytecode and having that can be executed by JVM.



```
public class main {
    public static void main(String args[])  //static method
    {
        System.out.println("Hello World");
    }
}
```

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University

Thanks for your attention!

Dr. İbrahim Delibaşoğlu
Department of Software Engineering
Sakarya University