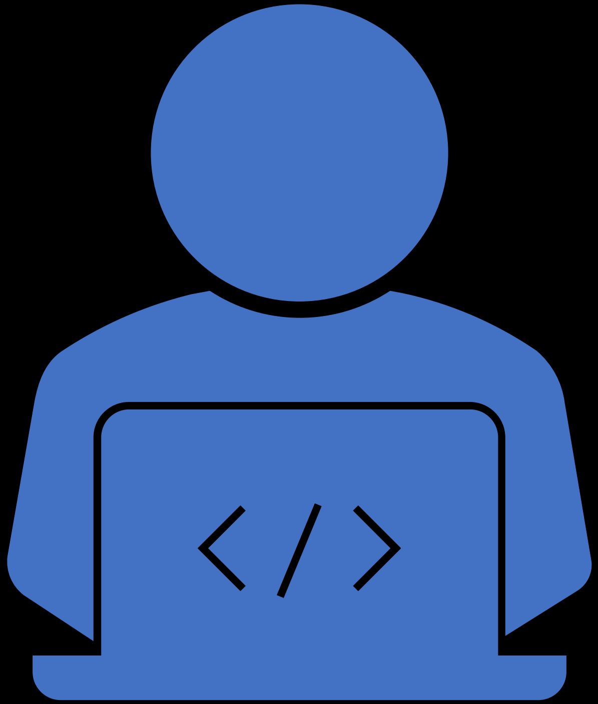


CENG204 - Programming Languages Concepts

Asst. Prof. Dr. Emre ŞATIR

Lecture 8
Lexical, Syntax, and Semantic Analysis

Lecture 8 Topics

- Introduction
- Lexical Analysis
- Syntax Analysis
- Semantic Analysis
- Code Optimization
- Code Generation

Introduction

Introduction

- **Lexical**, **syntax** and **semantic** analysis steps are also used in “Natural Language Processing (NLP)”.
- For example, consider the sentence “John ate an apple.” **The lexical analysis** provides the words (“John”, “ate”, “an”, “apple”).
- **The syntax rules** define the structure of the sentence, with the word “ate” serving as the verb.
- **Semantic analysis** helps to determine the meaning of the sentence by looking at the context of the words. In this case, the meaning is that John is eating an apple (past tense).

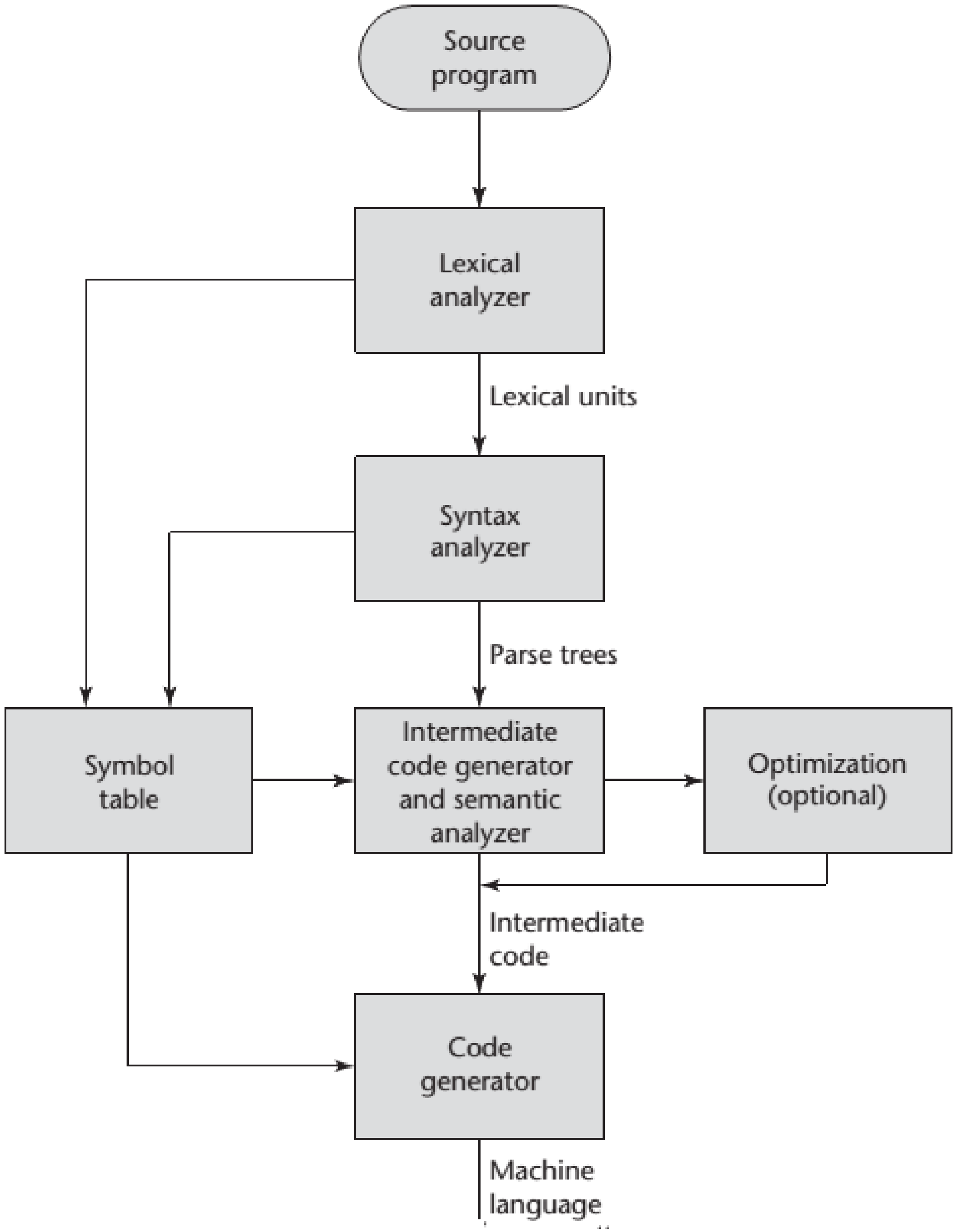
Introduction

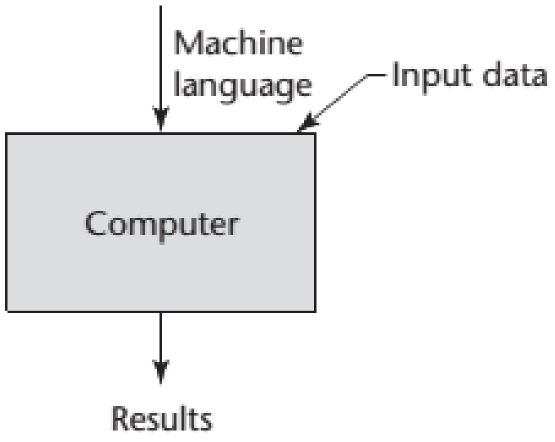
- Lexical, syntax and semantic analysis are essential components of natural language processing.
- Lexical analysis is the process of breaking down a large text into smaller parts, such as words, phrases or symbols, while syntax analysis is the process of understanding how these parts fit together to form meaningful sentences.
- Semantic analysis helps to determine the meaning of a sentence or phrase.
- By combining these three components, computers can understand natural language.

Introduction

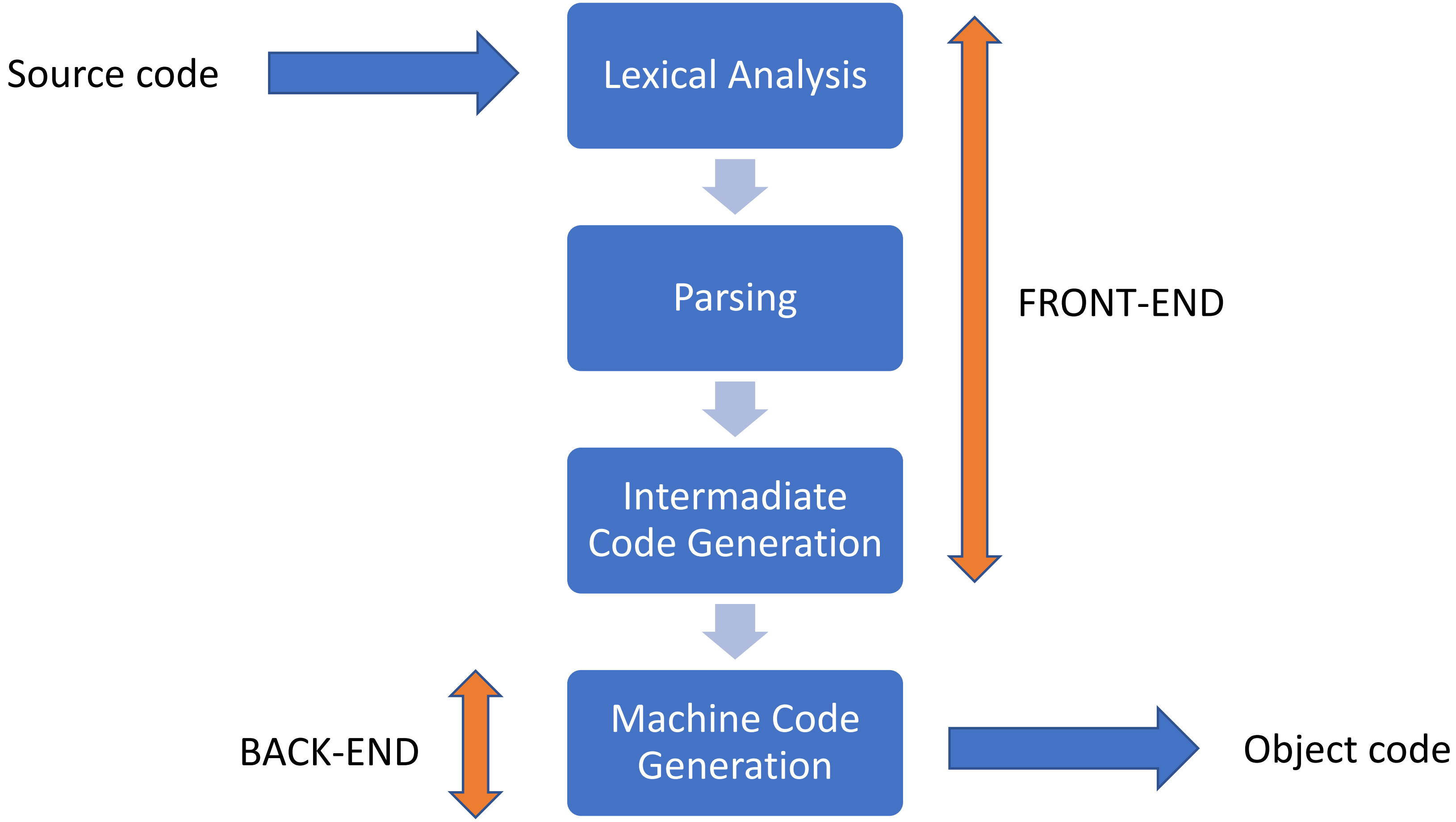
- In order for a source code written in a high-level language to be executed, it must be translated into machine language, the only language that the computer directly recognizes.
- The source code translated into machine language is called **object code**.
- Three different approaches for implementing programming languages were introduced before:
    - compilation
    - pure interpretation
    - hybrid implementation
- When a compiler converts source code to object code, it first examines whether this source code was built according to the rules. This job includes "**lexical**" and "**syntax**" analysis (All three of the implementation approaches just discussed use both lexical and syntax anayzers).

The Compilation Process





The Simplified Compilation Process



The Compilation Process

FRONT-END: In this section, the syntax and sematics of the program are analyzed according to the rules of the language, and an intermediate code is created. It is independent of the machine.

BACK-END: In this part, the machine code is created by applying the optimization (optional) from the intermediate code representation of the program. These parts are known as the **synthesis** step. It is machine dependent.

Introduction

- Nearly all compilers separate the task of analyzing syntax into two distinct parts, lexical analysis and syntax analysis for simplicity and efficiency.
- The lexical analyzer deals with small-scale language constructs, such as names and numeric literals (mathematically, a finite automaton based on a regular grammar).
- The syntax analyzer deals with the large-scale constructs, such as expressions, statements, and program units (Syntax analyzers are also called as "**parsers**") (mathematically, a push-down automaton based on a context-free grammar, or BNF).

Lexical Analysis

Lexical Analysis

- A lexical analyzer is essentially a “***pattern matcher***” for character strings. It identifies and distinguishes the lowest level units of the source program (lexemes).
- A lexical analyzer serves as the front end of a syntax analyzer.
- Technically, lexical analysis is a part of syntax analysis.
- A lexical analyzer performs syntax analysis at the lowest level of program structure.
- An input program appears to a compiler as a single string of characters.
- The lexical analyzer collects characters into logical groupings and assigns internal codes to the groupings according to their structure.

Lexical Analysis

- In chapters before, these logical groupings are named **lexemes**, and the internal codes for categories of these groupings are named **tokens**.
- Lexemes are recognized by matching the input character string against <u>character string patterns</u>.
- Although tokens are usually <u>represented as integer values</u>, for the sake of <u>readability</u> of lexical and syntax analyzers, they are often referenced through <u>named constants</u>.

Lexical Analysis

- Consider the following example of an assignment statement:

```
result = oldsum - value / 100;
```

- Following are the lexemes and tokens of this statement:

Lexemes	**Tokens**
`result`	IDENT
`=`	ASSIGN_OP
`oldsum`	IDENT
`-`	SUB_OP
`value`	IDENT
`/`	DIV_OP
`100`	INT_LIT
`;`	SEMICOLON

Lexical Analysis

- Lexical analyzer identifies the language keywords (`for`, `if`, `while`, `...`), variable names (`i`, `j`, `counter`, `...`), literals (`3`, `5`, `4.54`, …), punctuation (`'('`, `')'`,`’;'`) and operators (`+`, `*`, …) and <u>classify them as tokens</u>.
- Lexical analyzer removes <u>non-programming language symbols</u> such as <u>spaces</u> and <u>comments</u>.

Lexical Analysis

- One of the approaches to building a lexical analyzer is to design a “state transition diagram” that <u>describes the token patterns of the language</u> and write a program that implements the diagram.
- State diagrams of the form used for lexical analyzers are representations of a class of mathematical machines called “finite automata”.
- Finite automata can be designed to recognize members of a class of languages called “regular languages”.
- “Regular grammars (Regular expressions)” are generative devices for regular languages.
- The tokens of a programming language are a regular language, and a lexical analyzer is a finite automaton.

Syntax Analysis

The Parsing

- The part of the process of analyzing syntax that is referred to as "**syntax analysis**" is often called **parsing**. We will use these two interchangeably.
- Syntax analysis is performed to determine whether the words that make up the source program are in an order in accordance with the grammatical rules of the programming language.
- Parsers for programming languages construct **parse trees** for given programs. The parse tree is used as the basis for translation.

The Parsing

- Nearly all syntax analysis is based on a formal description of the syntax of the source language (BNF).
- 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 for the program.

Semantic Analysis

Semantic Analysis

- **Semantic analysis** is the creation of a code in an abstract programming language (intermediate code) using the parse tree created during syntax analysis.
  - Lexical Analysis → RGs / REs – DFA / NFA
  - Syntax Analysis → CFG / BNF - PDA
  - Semantics Analysis → Semantic features of the language
- Intermediate languages are similar to an assembly language.
- This abstract language forms an intermediate step between the compiler's source and object languages, designed to be compatible with the source language's data types and operations.

Intermediate Code

- If the compiler directly translates source code into the machine code without generating intermediate code then a full native compiler is required for each new machine.
- The intermediate code keeps the analysis portion same for all the compilers that's why it doesn't need a full compiler for every unique machine.
- Intermediate code generator receives input from its predecessor phase (semantic analyzer phase). It takes input in the form of an "annotated syntax tree".
- Using the intermediate code, the second phase of the compiler (the synthesis phase) produces the object code (changed according to the target machine).

Intermediate Code Generation Example in C Programming Language

```c
int main(){
        int dizi1[5], dizi2[5], carpim, i;
        int *p;
        int *r;
        carpim = 0;
        p = dizi1;
        r = dizi2;
        for (i=0; i<10; i++){
                carpim += *p++ * *r++;
        }
        return 0;
}
```

```
carpim=0;
p=&dizi1;
r=&dizi2;
i=0;
S1: if(i>=10) goto S2;
    T3=*p;
    T4=p+1;
    p=T4;
    T5=*r;
    T6=r+1;
    r=T6;
    T7=T3*T5;
    T8=carpim+T7;
    carpim=T8;
    T9=i+1;
    i=T9;
    goto S1;
S2:
```

Code Optimization

Code Optimization

- **Optimization** is the name given to the collection of techniques that compilers may use to decrease the size and/or increase the execution speed of the code they produce.

- Example:

```
x = 5;
y = 8;
if (x > y)
    …                    → dead code
else
    …
```

- Disposal of dead code will save space.

Code Optimization

- Example:

```
do
{
    counter = 10;
    sum = sum + counter;
}
while (sum < 100);
```

```
counter = 10;
do
{
    sum = sum + counter;
}
while (sum < 100);
```

- Since it is enough to assign the value to the “counter" variable once, taking it out of the loop will increase the running speed.

Code Optimization

- Optimization is an optional step (it can be turned off in the compiler settings).
- If little or no optimization is done, compilation can be done much faster than if a significant effort is made to produce optimized code.
- The statement order of the code resulting from the optimization may differ from the original source code, which can make debugging (locating errors) difficult.
- Remember that most optimization is done on the intermediate code.

Code Generation

Code Generation

- At this stage, the conversion from the machine independent "intermediate code" to the "machine code" of the target hardware is performed.
- This is known as **the synthesis step**.
- The best machine instructions that will perform intermediate code operations must be selected.

Code Generation

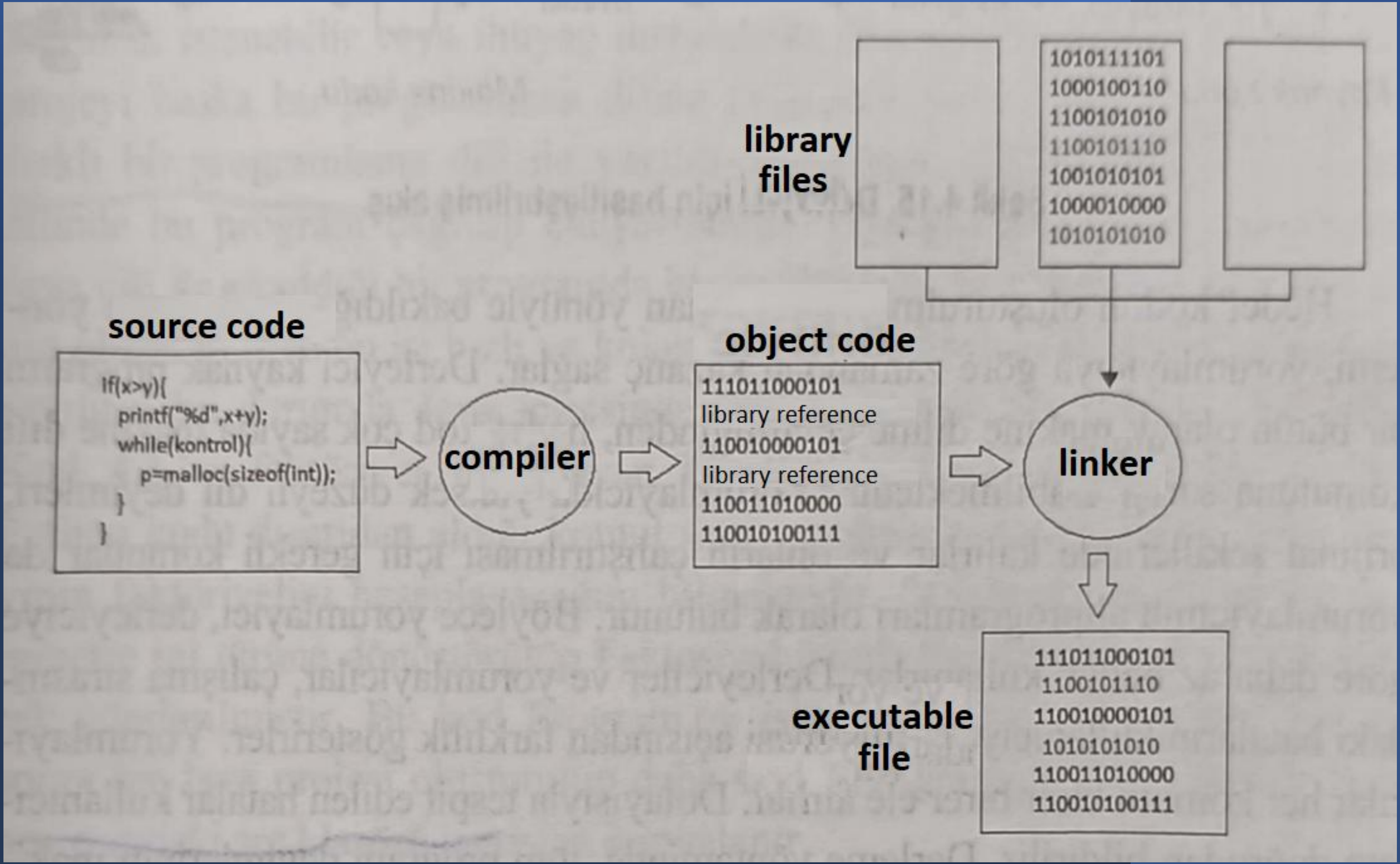
- Intermediate code generation has some advantages over directly generating machine code.
- In case no intermediate code is used, if there are x different target machines for a source language, x number of optimizations and code generators will be needed.
- The optimization part written for one machine cannot be used for another machine and will have to be rewritten, which is one of the most difficult parts to write in compiler design (We have to mentioned that many kinds of optimization are difficult to do on machine language).

Code Generation

- In general, the benefit of the “FRONT-END" and “BACK-END" studies being independent of each other is that:
  - if a new machine (platform) emerges, it enables the production of solutions using the existing “FRONT-END” (and optimization part),
  - and if a new programming language is produced (that uses the same intermediate code), using the existing “BACK-END”.

Code Generation

- The machine code produced in the code generation phase contains reference sections to library files.
- In this case, it turns out that a complete machine code has not been produced and there are missing parts.
- At this stage, the **linker** will come into play and add machine codes to the places where references are left in the library files, and the final executable version of the file will be produced.


library files
1010111101
1000100110
1100101010
1100101110
1001010101
1000010000
1010101010
source code
If(x>y){
printf("%d",x+y);
while(kontrol){
p=malloc(sizeof(int));
}
}
compiler
object code
111011000101
library reference
110010000101
library reference
110011010000
110010100111
linker
executable file
111011000101
1100101110
110010000101
1010101010
110011010000
110010100111