

# 50.051 Programming Language Concepts

## W6-S3 Introduction to Compilers

Matthieu De Mari



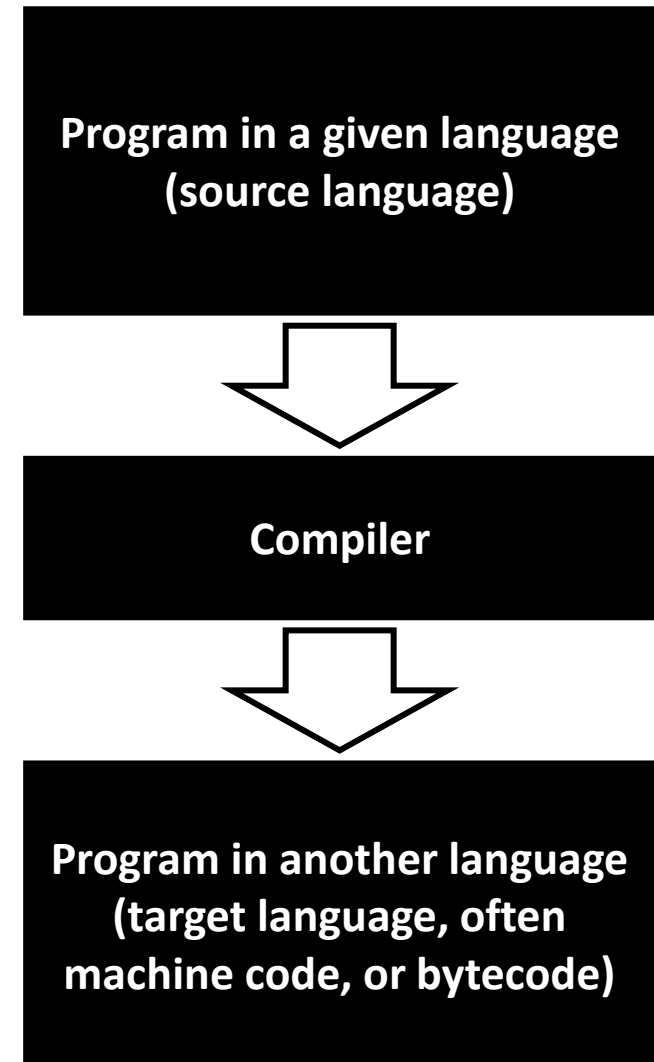
# Compiler: a definition

## Definition (**Compilers**):

**Compilers** are **computer programs** whose purpose is to

- **Translate a program** written in **one language** (called **source language**),
- Into a **program** written in **another language** (called **target language**).

Target language is often machine code or bytecode.



# Compiler: a definition

**Definition (Compilers):**  
**Compilers** are **computer programs** whose purpose is to

- **Translate a program** written in **one language** (called **source language**),
- Into a **program** written in **another language** (called **target language**).

Target language is often machine code or bytecode.

The purpose of compilers is then

- To **check if the code to be compiled is valid**,
- To **automate the translation process**, which will eventually allow for a given code from a given programming language to be translated into machine code,
- And eventually **executed by the CPU of your computer**.

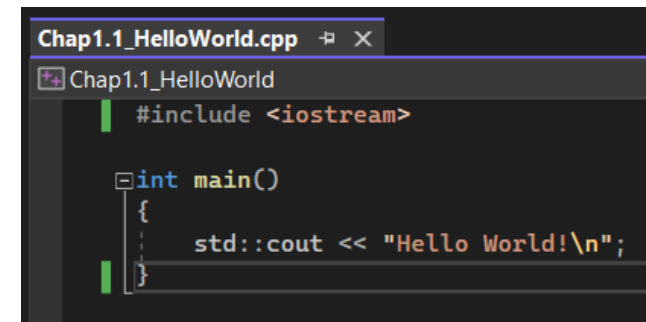
# Compiler: a definition

## Definition (**Source Program** and **Target Program**):

Typically, compilers will translate

- A **source program** often written in a **high-level language** (e.g. C/C++),
- Into a **target program** often written in a **low-level language** (e.g. machine code, ready to be executed by the CPU).

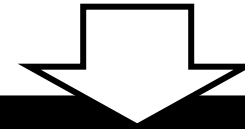
Compilers will typically prepare our source code for **execution**.



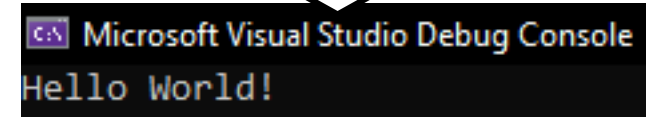
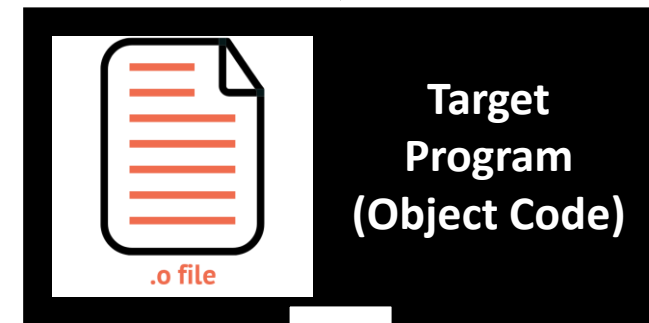
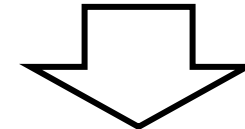
```
Chap1.1_HelloWorld.cpp
Chap1.1_HelloWorld
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
}
```

Source Program (C/C++)



Compiler



```
Microsoft Visual Studio Debug Console
Hello World!
```

Execution

# What does a compiler need?

## Definition (**Conceptual Roadmap of a Compiler**):

Translating software from one language to another language requires to:

- Understand **form/syntax** of the **source code** and the rules that govern **form/syntax** in the **source language**,
- Understand the **content/meaning** of the **source code** and **source language**,

- Understand the rules that govern **form/syntax** and **content/meaning** of the **target language**,
- Have a **scheme** for **mapping content/meaning** from the **source language** to the **target language**.

This defines a **basic conceptual roadmap of a compiler**.

# On the need of intermediate languages (IL) or intermediate representations (IR)

## **Definition (Intermediate Language (IL), Intermediate Representation (IR)):**

It might be tempting to think compilers are black-boxes capable of translating a source program directly into CPU instructions.

In practice, we prefer to **add extra steps to the translation**, typically **translating the source program** into an **Intermediate Language (IL)**, first.

This **Intermediate Representation (IR)**, often has a lower language level, than the source code, and will then be translated into our target program.

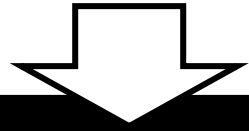
This typically allows for **optimization routines** to be run on the intermediate program **before the final translation** and **upcoming execution** (For instance, checking for errors and optimizing your computer resources).

Restricted

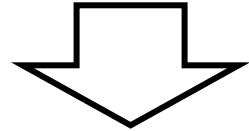
```
Chap1.1_HelloWorld.cpp
Chap1.1_HelloWorld
#include <iostream>

int main()
{
    std::cout << "Hello World!\n";
}
```

Source Program (C/C++)



Compiler



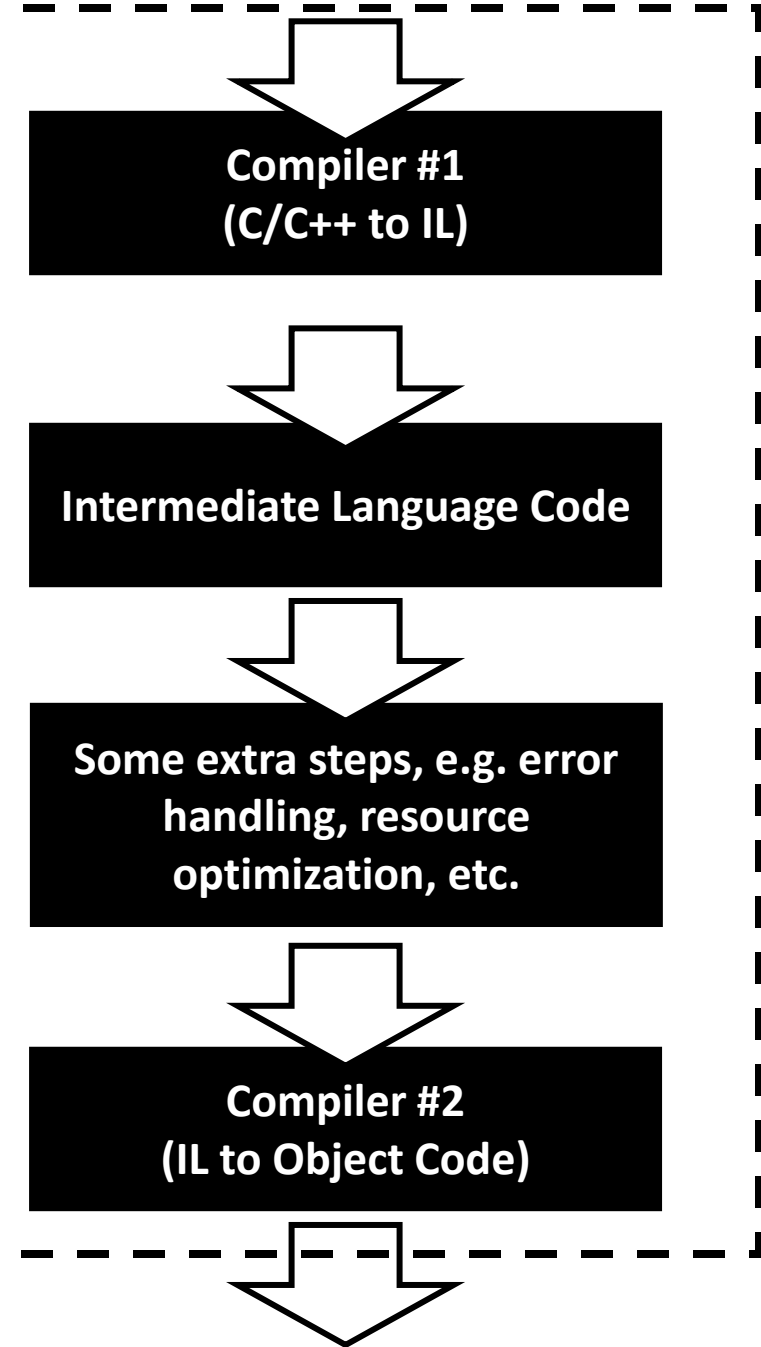
.o file

Target Program  
(Object Code)



Microsoft Visual Studio Debug Console  
Hello World!

Execution



Restricted

# Source-to-source compilers

## Definition (**Source-To-Source Compilers**):

C and C++ are among the most basic languages. These days, compilers for C and C++ are available on most computers and operating systems.

For this reason, many programming languages will therefore **compile by using C/C++ as an intermediate language**, first.

Then, all it takes is one extra compilation from C/C++ to machine code, which then leads to execution.

**Compilers that target other programming languages (typically C/C++)** are then commonly referred to as **source-to-source compilers**.



# And interpreters in all of this?

## Definition (**Interpreters**):

Many of you have probably heard about the **compilers vs. interpreters paradigm**, but what is it about?

**Interpreters** have the **same objective as compilers**, i.e. **translate a source program into target code which can be executed by the CPU**.

What changes is the translation and execution procedure:

- **Compilers** will translate the source code into target code **in its entirety first**, and THEN will execute the target code.
- **Interpreters**, on the other hand, will translate each line of the source code **one line at a time**, and execute each one of them in succession.

```
1 name = input("What is your name?\n")
2 print(f"Well, hello there {name}!")
```

Source Program (Python)



Interpreter

In [\*]:

Line 1

```
1 name = input("What is your name?\n")
```

Source Program (Python)

Translate, check for errors, etc.  
and execute

What is your name?

Results

Line 2

```
2 print(f"Well, hello there {name}!")
```

Source Program (Python)

Translate, check for errors, etc.  
and execute

Well, hello there Matt!

Results

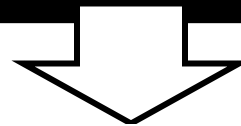
In [\*]:

...

In [\*]:

Line N

...



What is your name?  
Matt  
Well, hello there Matt!

Results  
Restricted

# Compilers vs. Interpreters (to summarize)

Compilers	Interpreters
Scans the entire program and translates it as a whole into machine code.	Translates program one statement at a time.
If there is an error in the code, compilation will crash and nothing will be executed. Error message will describe the first error encountered. All or nothing.	If there is an error in the code, execution will happen until the first error is met, at which point the execution stops, making it easier to debug. Beginner-friendly.
Compilers usually take a large amount of time to analyze the source code. However, the overall execution time is comparatively faster than interpreters.	Interpreters usually take less amount of time to analyze the source code. However, the overall execution time is comparatively slower than compilers.
Programming languages like C, C++, Java use compilers.	Programming languages like JavaScript, Python, Ruby use interpreters.

**Note:** Implementation of interpreters are out of the scope of this class.  
Discussing them for general knowledge.

# Just-in-time compilation

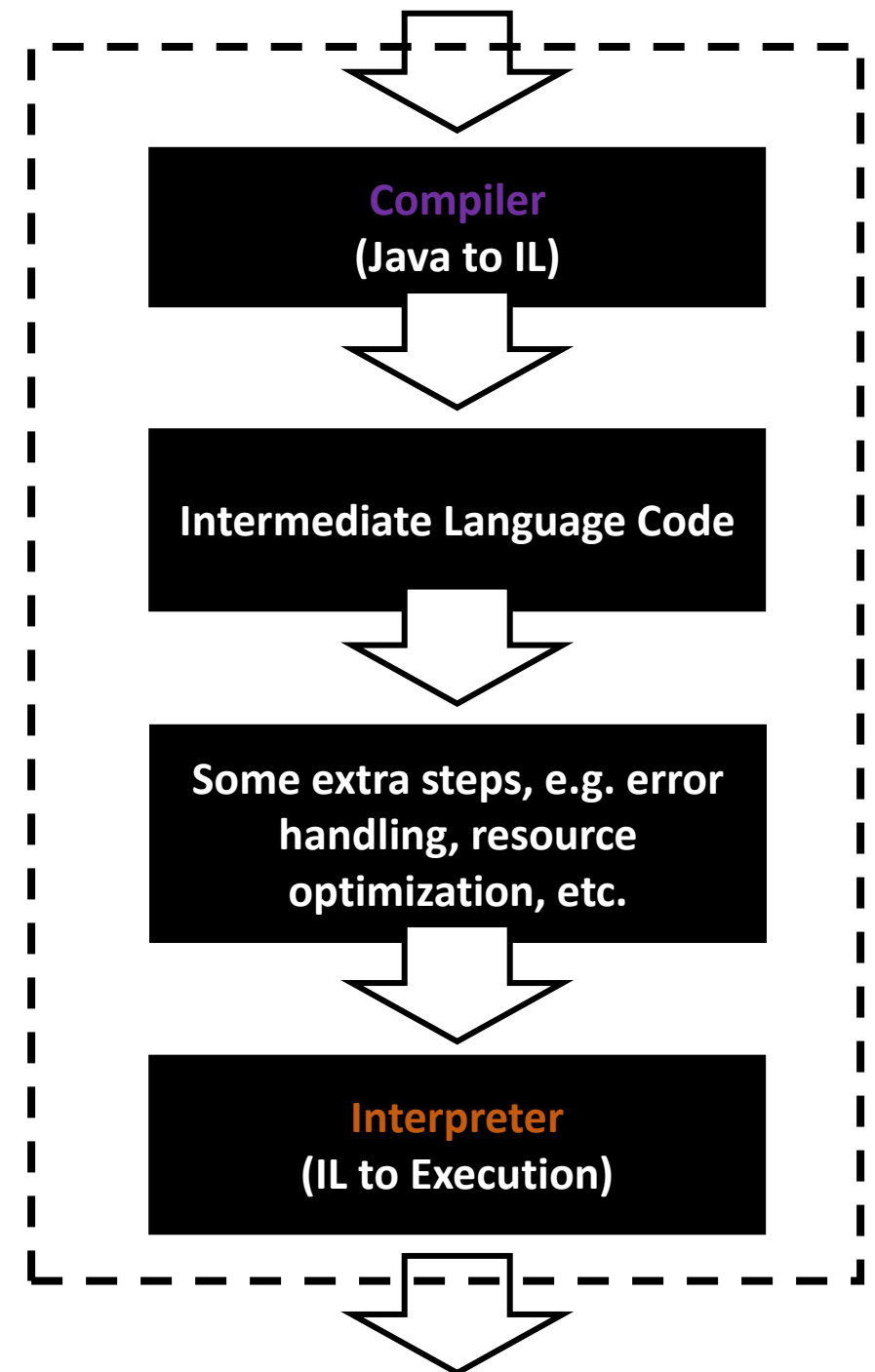
## Definition (**Just-in-time Compilers**):

A compiler can have multiple translations in succession, using several intermediate languages before reaching target code.

In certain languages (e.g., Java), some of these successive translations are done using a **combination of compilers and interpreters**.

The whole compiler is then commonly referred to as a **just-in-time compiler**.

**Note:** Implementation is out-of-scope.



# Just-in-time compilation

## Definition (**Just-in-time Compilers**):

**Just-in-time compilers (JIT)** are then a type of compiler that

- **Dynamically** compiles source code into machine code **at runtime**,
- Rather than **ahead of time**.

This allows for greater flexibility and performance optimization, as the compiler can make use of runtime information and adjust its optimization strategies accordingly.

In some languages like Java, **JIT compilation is used in combination with interpreters to achieve the benefits of both approaches.**

*(Basically, getting the best of both worlds?)*

# Why should you study compilers in a Computer Science degree?

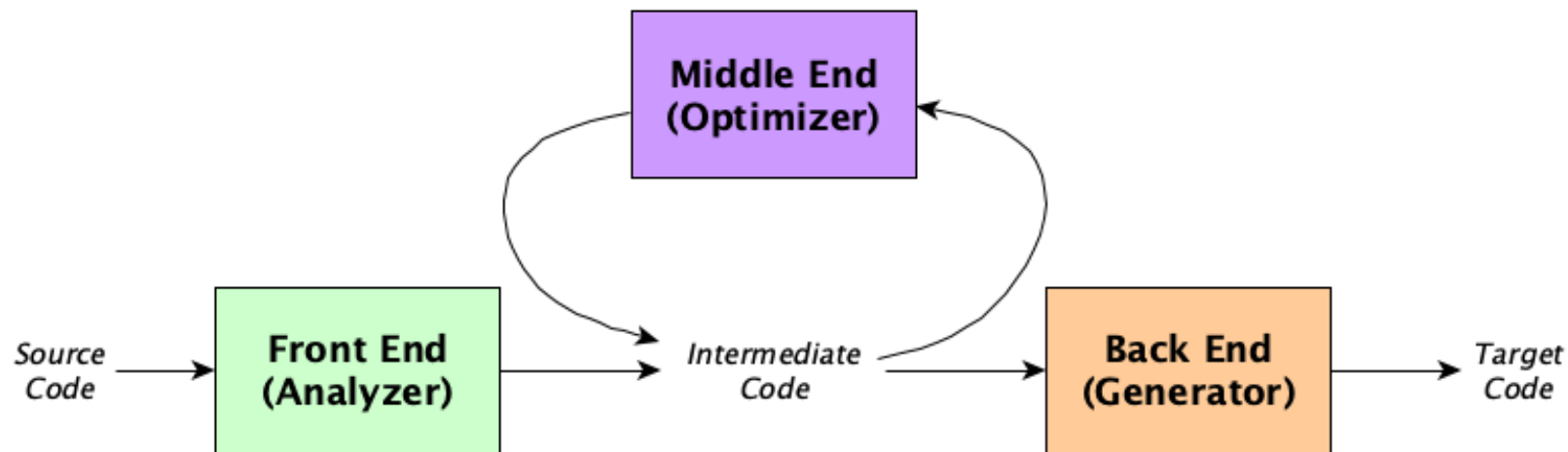
1. A great introduction to techniques for translation and improvement of programs written in any programming language.
2. A fantastic practical exercise in software engineering.
3. Provides an understanding of what happens in the background when you press the “Run” button of your programs.
4. A way to understand where typical compilation errors come from.
5. A better understanding of the missing link between any programming course and the 50.002 Computation Structures one.
6. A much-needed introduction course to compilers, if one day you plan on writing your own programming language (*don't!*).

# The architecture of a compiler

## Definition (the three parts of a typical compiler architecture):

A typical compiler architecture consists of three main components: the **front-end**, **middle-end**, and **back-end**.

Each component is responsible for a specific set of tasks and works together to generate the final executable code.

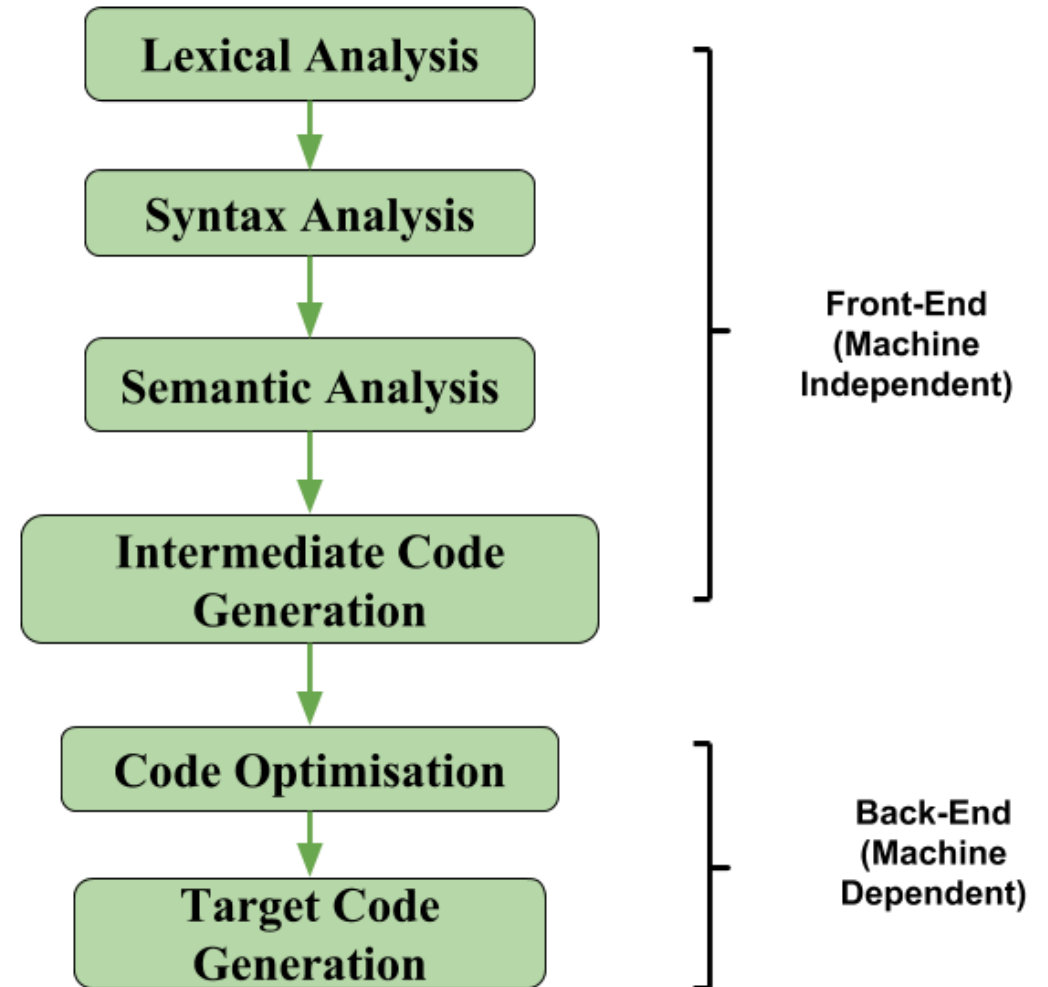


# The architecture of a compiler

## Definition (the three parts of a typical compiler architecture):

A typical compiler architecture consists of three main components: the **front-end**, (the **middle-end**,) and **back-end**.

Each component is responsible for a specific set of tasks and works together to generate the final executable code.





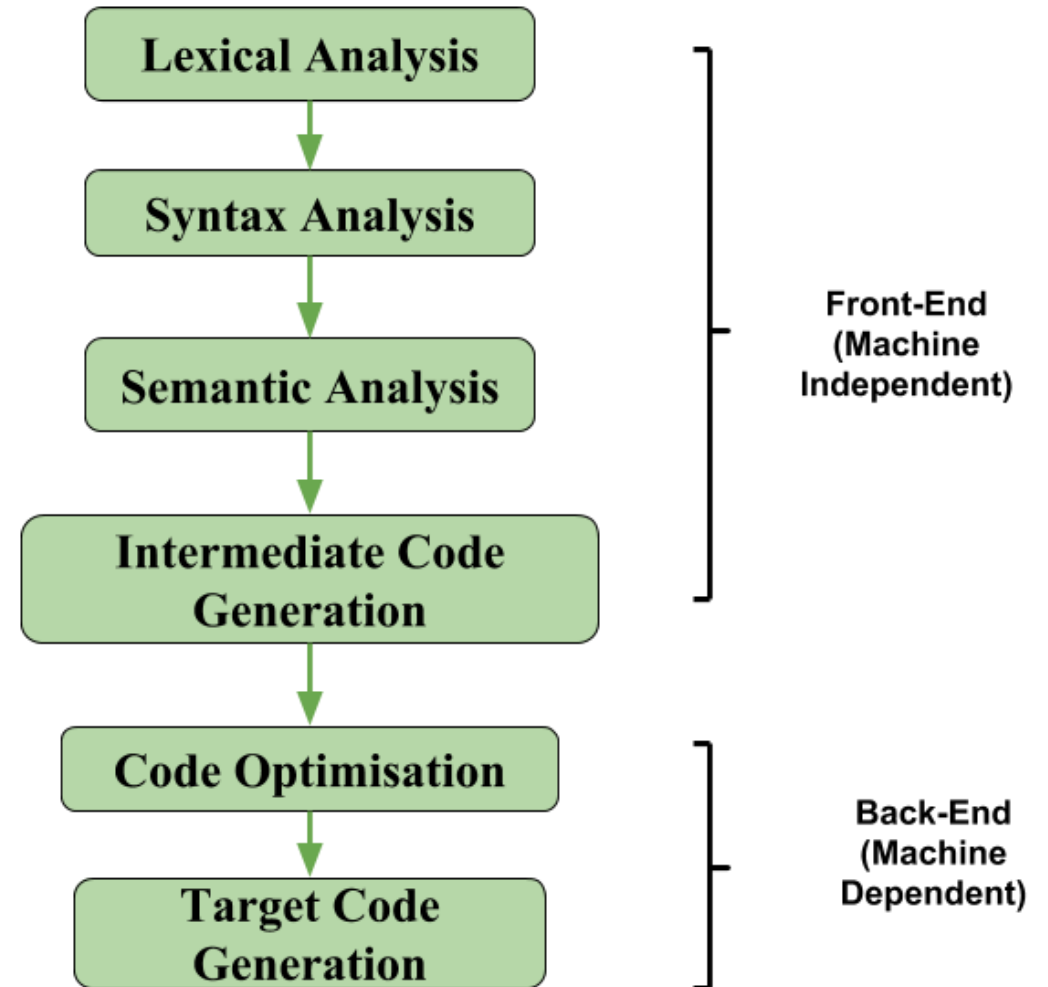
# The front-end of a compiler

**Definition (The front-end part of a compiler):**

The **front-end of a compiler** is responsible for verifying the source code, eventually converting it into a form that can be used by the rest of the compiler.

It involves tasks, such as:

- **Lexical analysis,**
- **Syntax analysis,**
- and **Semantic analysis.**



# Lexical Analysis

## Definition (**Lexical Analysis**):

During **lexical analysis**, the source code is broken down into **tokens**, which represent the individual components of the language.

It is sometimes referred to as **scanning** or **tokenization**.

```
int min(int firstNumber, int secondNumber)
{
    if (firstNumber > secondNumber) {
        return secondNumber;
    }
    else (
        return firstNumber;
    )
}
```

# Lexical Analysis

## Definition (**Tokens**):

**Tokens** are the **smallest individual units of a programming language** that the compiler can recognize and understand.

A token is a sequence of characters that has a specific meaning in the language, such as a **keyword, identifier, operator, or punctuation symbol**.

- **Keyword:** A keyword is a **reserved word** in a programming language that **has a special meaning** and cannot be used as an identifier.

Examples of keywords in the C programming language:

- int, double, long, ...
- if, else, while, ...
- return, ...
- etc.

# Lexical Analysis

## Definition (**Tokens**):

**Tokens** are the **smallest individual units of a programming language** that the compiler can recognize and understand.

A token is a sequence of characters that has a specific meaning in the language, such as a **keyword, identifier, operator, or punctuation symbol**.

- **Identifier:** An identifier is a **name given to a variable, function, or other entity** in a program. Will follow rules, such as starting with a letter or underscore and consisting of letters, digits, and underscores.

Examples of identifiers:

- x, counter, variable\_1,
- myFunction,
- etc.

# Lexical Analysis

## Definition (**Tokens**):

**Tokens** are the **smallest individual units of a programming language** that the compiler can recognize and understand.

A token is a sequence of characters that has a specific meaning in the language, such as a **keyword, identifier, operator, or punctuation symbol**.

- **Literal:** A **value** of some sort to be **assigned to a variable**.

Examples of literals in the C programming language:

- 12542
- 12654165.52
- “hello”
- Etc.

# Lexical Analysis

## Definition (**Tokens**):

**Tokens** are the **smallest individual units of a programming language** that the compiler can recognize and understand.

A token is a sequence of characters that has a specific meaning in the language, such as a **keyword, identifier, operator, or punctuation symbol**.

- **Operator:** An operator is a **symbol that performs a specific operation** on one or more values.

Examples of operators in the C programming language:

- +, -, \*, /,
- =,
- &&,
- etc.

# Lexical Analysis

## Definition (**Tokens**):

**Tokens** are the **smallest individual units of a programming language** that the compiler can recognize and understand.

A token is a sequence of characters that has a specific meaning in the language, such as a **keyword, identifier, operator, or punctuation symbol**.

- **Punctuation symbol:** A punctuation symbol is a **symbol used to separate or group different parts of a program**

Examples of punctuation symbols in the C programming language

- Braces {} and parentheses ()
- Commas, semicolons,
- Quotation marks,
- Etc.

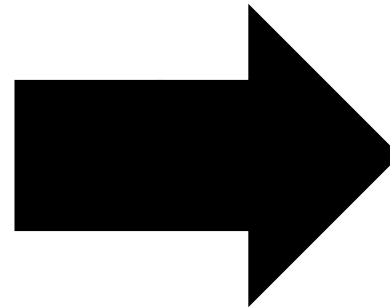
# Lexical Analysis

## Definition (**Lexical Analysis**):

During **lexical analysis**, the source code is broken down into **tokens**, which represent the individual components of the language.

It is sometimes referred to as **scanning** or **tokenization**.

```
int min(int firstNumber, int secondNumber)
{
    if (firstNumber > secondNumber) {
        return secondNumber;
    }
    else (
        return firstNumber;
    )
}
```



KEYWORD (int)  
IDENTIFIER (min)  
PUNCTUATION (open\_par)  
KEYWORD (int)  
VARIABLE (firstNumber)  
...



# Steps for Lexical Analysis

## Definition (Steps for **Lexical Analysis**):

During lexical analysis, the compiler **reads the source code character by character** and identifies each token.

- It does so, based on its **position** and **context** within the code.
- The compiler also uses **a set of rules or patterns** called **regular expressions**, to recognize different types of tokens, such as keywords, identifiers, operators, and punctuation symbols.

Once the compiler has identified the tokens, it **assigns each token a specific type/category**, based on its role and meaning in the program.

# Lexical Analysis Recap

Lexical analysis is an important part of the compilation process.

- It serves as the first step of analysing the source code, by breaking the code down into a series of tokens.
- It can catch simple errors, e.g. variable names breaking the rules of naming (special characters in name, starts with a digit, etc.).
- **After the lexical analysis phase is complete, the compiler passes the resulting series of tokens to the next stage of the compilation process, which is typically the syntax analysis or parsing phase.**

# Syntax Analysis

## Definition (**Syntax Analysis**):

**Syntax Analysis**, or **parsing**, is the second stage in the compilation process, after lexical analysis.

It involves analysing the structure of a program or source code to determine whether **it conforms to the grammar of the source programming language** (i.e. the set of rules and syntax).

# Grammar of a programming language

## Definition (**Grammar** of a programming language):

The **grammar** of a programming language refers to a set of rules that define the syntax and structure of the language.

The grammar specifies how different components of the language, such as keywords, identifiers, operators, and expressions, can be combined to form valid programs.

For instance, the expected grammar or structure for an “if” statement is:

```
if (condition) {  
    statement1;  
}  
else {  
    statement2;  
}
```

# Grammar of a programming language

## Definition (**Grammar** of a programming language):

The **grammar** of a programming language refers to a set of rules that define the syntax and structure of the language.

The grammar specifies how different components of the language, such as keywords, identifiers, operators, and expressions, can be combined to form valid programs.

In practice, we will check that the grammar follows a set of rules using **context-free grammars (CFGs)**.

It consists of a set of rules, which describe how different elements of the language can be combined to form valid statements or expressions.

```
<if_statement> ::= "if" "(" <expression> ")" <statement>  
                [ "else" <statement> ]
```

# Syntax Analysis

## Definition (**Syntax Analysis**):

**Syntax Analysis**, or **parsing**, is the second stage in the compilation process, after lexical analysis.

It involves analysing the structure of a program or source code to determine whether **it conforms to the grammar of the source programming language** (i.e. the set of rules and syntax).

In this stage, the compiler **takes the series of tokens** generated during the lexical analysis or tokenization stage.

It then attempts to **build a parse tree** (or **syntax tree**), which **represents the structure and meaning of the program**.

# Parse Tree

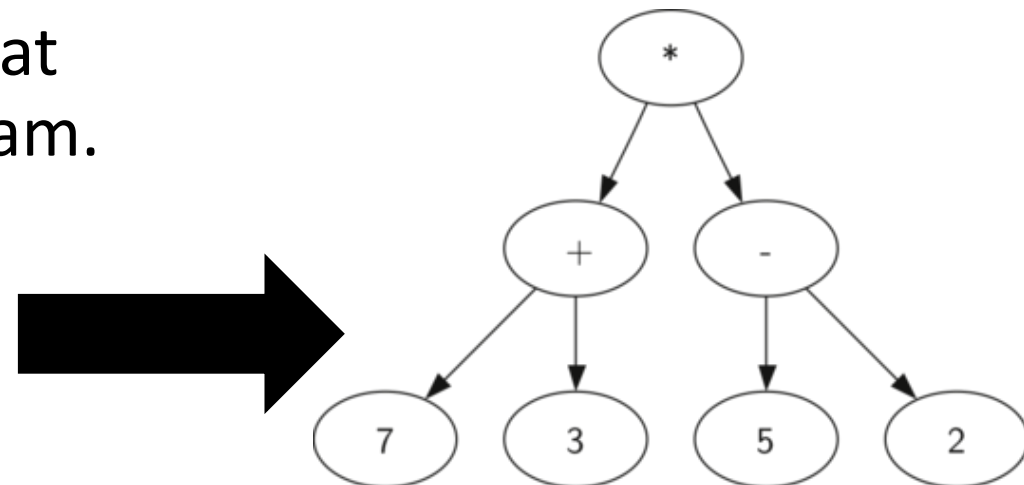
## Definition (**Parse Tree**):

The **parse tree** is a hierarchical structure, with nodes representing different parts of the program and their relationships to each other.

The root of the tree represents the entire program, and the child nodes represent the different components of the program, such as functions, loops, and conditionals.

Each node in the tree has a label or type that identifies its role and meaning in the program.

```
int x;           // Declare x as an integer variable
x = (7 + 3) * (5 - 2); // Assign the result of the expression to x
```



# Syntax Analysis and Syntax Errors

Syntax analysis can also involve the **detection and reporting of syntax errors**, which are **violations of the language rules**.

- When the compiler encounters a syntax error in the program, it stops the compilation process and generates an error message that identifies the location and nature of the error.
- Syntax errors will typically include missing or incorrect punctuation, incorrect use of keywords or identifiers, wrong uses of conditional structures and loops, wrong sequences of tokens, or mismatched parentheses or braces.
- *(More on this on our lecture on CFGs and the lab activity!)*



# Syntax Analysis Recap

Syntax analysis allows the compiler to understand the structure and meaning of the program at a deep level.

- By building a **parse tree**, the compiler can identify the different tokens of the program and their relationships to each other.
- It then breaks down all the operations in the source code, into a set of elementary operations that the CPU could run.
- **Once the syntax analysis stage is complete, the compiler passes the parse tree or syntax tree to the next stage of the compilation process, the semantic analysis.**

# Semantic Analysis

## Definition (**Semantic Analysis**):

**Semantic Analysis** is the third major phase of the front-end part of the compilation process, following the lexical and the syntax analysis.

The goal of semantic analysis is to ensure that the program is not only lexically and syntactically correct, but also **semantically meaningful and free of errors**.

For instance, this involves analysing the meaning of the program's statements and expressions and **checking them against the rules and constraints of the language's type system and semantics**.

# Semantic Analysis Steps

The semantic analysis phase typically involves three important subtasks (there are more, but these are the main ones), which are:

- **Type checking,**
- **Scope analysis,**
- **And name resolution.**

These tasks are performed in order to ensure that the program is well-formed and executable, before we attempt the translation process.

Each of these subtasks also serve to catch the final possible errors and inconsistencies that might cause the program to behave incorrectly or unpredictably at runtime. **After this phase, the code to be compiled can be deemed “legal”, and the translation process can begin.**

# Type Checking

## Definition (**Type Checking**):

**Type Checking** involves verifying that **the types of the operands and operators in expressions are compatible and consistent with the rules of the language's type system.**

For instance, in C and Python, adding two integers is a valid operation, but adding an integer and a string is not.

The type checker therefore ensures that the types of the operands in expressions match the expected types, and that the result of the expression is also of the correct type.

# Type Checking

Definition (**Type Checking**):

**Type Checking** involves verifying that the operators in expressions are compatible with the types of the language's type system.

For instance, in C and Python, adding two integers is valid, but adding an integer and a string is not.

The type checker therefore ensures that the types of expressions match the expected types, and that the result expression is also of the correct type.

```
#include <stdio.h>

int main() {
    int x = 5;
    char* str = "hello";
    int result = x + str;
    printf("The result is %d\n", result);
    return 0;
}
```

# Scope Analysis

## Definition (**Scope Analysis**):

**Scope Analysis** involves determining the visibility and accessibility of variables and other program elements, based on their declaration and context within the program.

This ensures that variables are only used in the correct context, and that they are not accidentally overwritten or used in unintended ways.

For instance: in most languages, if a variable is declared inside a function, it should only be accessible within that function and not outside of it.

# Scope Analysis

## Definition (**Scope Analysis**):

**Scope Analysis** involves determining the visibility of variables and other program elements, based on their context within the program.

This ensures that variables are only used in the context where they are not accidentally overwritten.

For instance: in most languages, if a variable is declared inside a function, it should only be accessible within that function.

```
#include <stdio.h>

void function1() {
    int x = 10;
}

int main() {
    function1();
    printf("The value of x is %d\n", x);
    return 0;
}
```

# Name Resolution

## Definition (**Name Resolution**):

**Name Resolution** involves resolving references to variables, functions, and other program elements, based on their names and context within the program.

This ensures that the correct element is referred to, and that conflicts or ambiguities between different program elements are resolved correctly.

For instance, your code could have two variables with name “x” in two different functions, but the program should understand the difference between the two.



# Semantic Analysis Recap

Overall, semantic analysis is an important part of the compilation process that ensures that programs are more than just lexically and syntactically correct.

By analysing the meaning and structure of the program's statements and expressions, semantic analysis helps to catch errors and inconsistencies that might cause the program to behave incorrectly or unpredictably at runtime.

This is the final step of the analysis part of the compiler: **a source code that passes all three analysis steps is deemed legal, well-formed and will execute!**

# Quiz

## **What is the purpose of a compiler?**

- a. To translate a program written in one language into a program written in another language
- b. To execute a program on a computer
- c. To debug a program
- d. To document a program

# Quiz

**What is the purpose of a compiler?**

- a. To translate a program written in one language into a program written in another language**
- b. To execute a program on a computer
- c. To debug a program
- d. To document a program

# Quiz

## **What are types of compilers?**

- a. Source-to-source, just-in-time (JIT)
- b. Front-end, middle-end, back-end
- c. Syntax, semantics, tokens, parse tree, intermediate code, machine code, executable code
- d. C, C++, Java, Python

# Quiz

**What are types of compilers?**

- a. Source-to-source, just-in-time (JIT)**
- b. Front-end, middle-end, back-end
- c. Syntax, semantics, tokens, parse tree, intermediate code, machine code, executable code
- d. C, C++, Java, Python

# Quiz

**Which of these operations are part of the front-end of a compiler?**

- a. Generating machine code
- b. Optimizing code
- c. Parsing source code and building an abstract syntax tree
- d. Executing the code and showing error messages to the user, if any.

# Quiz

**Which of these operations are part of the front-end of a compiler?**

- a. Generating machine code
- b. Optimizing code
- c. Parsing source code and building an abstract syntax tree**
- d. Executing the code and showing error messages to the user, if any

# Quiz

## **What are tokens in the context of a compiler?**

- a. The basic building blocks of the programming language's syntax
- b. The abstract representation of the program's semantics
- c. The set of instructions that a computer can execute directly
- d. The high-level language that a program is written in



# Quiz

**What are tokens in the context of a compiler?**

- a. The basic building blocks of the programming language's syntax**
- b. The abstract representation of the program's semantics
- c. The set of instructions that a computer can execute directly
- d. The high-level language that a program is written in

# Quiz

## **What is the parse tree?**

- a. A list of keywords and symbols used in the program
- b. A data structure that represents the hierarchical structure of the source code
- c. A set of rules that define the syntax of the programming language
- d. A set of instructions that a computer can execute directly

# Quiz

## What is the parse tree?

- a. A list of keywords and symbols used in the program
- b. A data structure that represents the hierarchical structure of the source code**
- c. A set of rules that define the syntax of the programming language
- d. A set of instructions that a computer can execute directly

# Quiz

**Which of these problems will be ignored by the lexical analysis and the syntax analysis, but caught by the semantic analysis?**

- a. A code contains a variable name, immediately followed by an integer value of some sort (e.g. “int **x 10**;”)
- b. A code contains a special symbol, which has no meaning in the language (e.g. “int x = 10 **@** 7;”)
- c. A code attempts to sum together two arrays x and y to produce a third array z, even though it is forbidden to use  $z = x + y$  on this type of variables in C
- d. A code attempts to print a variable that is not yet defined (e.g. a `printf(“%d”, x)` appears before “int x = 7;”)

# Quiz

**Which of these problems will be ignored by the lexical analysis and the syntax analysis, but caught by the semantic analysis?**

- a. A code contains a variable name, immediately followed by an integer value of some sort (e.g. "int x 10;")**
- b. A code contains a special symbol, which has no meaning in the language (e.g. "int x = 10 @ 7;")**
- c. A code attempts to sum together two arrays x and y to produce a third array z, even though it is forbidden to use  $z = x + y$  on this type of variables in C**
- d. A code attempts to print a variable that is not yet defined (e.g. a `printf("%d", x)` appears before "int x = 7;")**

# Quiz

**Which of these problems will be ignored by the lexical analysis and the syntax analysis, but caught by the semantic analysis?**

- a. A code contains a variable name, immediately followed by an integer value of some sort (e.g. "int x 10;") → Syntax Analysis**
- b. A code contains a special symbol, which has no meaning in the language (e.g. "int x = 10 @ 7;")**
- c. A code attempts to sum together two arrays x and y to produce a third array z, even though it is forbidden to use  $z = x + y$  on this type of variables in C**
- d. A code attempts to print a variable that is not yet defined (e.g. a `printf("%d", x)` appears before "int x = 7;")**

# Quiz

**Which of these problems will be ignored by the lexical analysis and the syntax analysis, but caught by the semantic analysis?**

- a. A code contains a variable name, immediately followed by an integer value of some sort (e.g. "int x 10;") → Syntax Analysis**
- b. A code contains a special symbol, which has no meaning in the language (e.g. "int x = 10 @ 7;") → Lexical Analysis**
- c. A code attempts to sum together two arrays x and y to produce a third array z, even though it is forbidden to use  $z = x + y$  on this type of variables in C**
- d. A code attempts to print a variable that is not yet defined (e.g. a `printf("%d", x)` appears before "int x = 7;")**

# Quiz

Which of these problems will be ignored by the lexical analysis and the syntax analysis, but caught by the semantic analysis?

- a. A code contains a variable name, immediately followed by an integer value of some sort (e.g. "int x 10;") → Syntax Analysis
- b. A code contains a special symbol, which has no meaning in the language (e.g. "int x = 10 @ 7;") → Lexical Analysis
- c. A code attempts to sum together two arrays x and y to produce a third array z, even though it is forbidden to use  $z = x + y$  on this type of variables in C → Semantic Analysis, but careful, this operation could be allowed in some languages!
- d. A code attempts to print a variable that is not yet defined (e.g. a `printf("%d", x)` appears before "int x = 7;") → Semantic Analysis



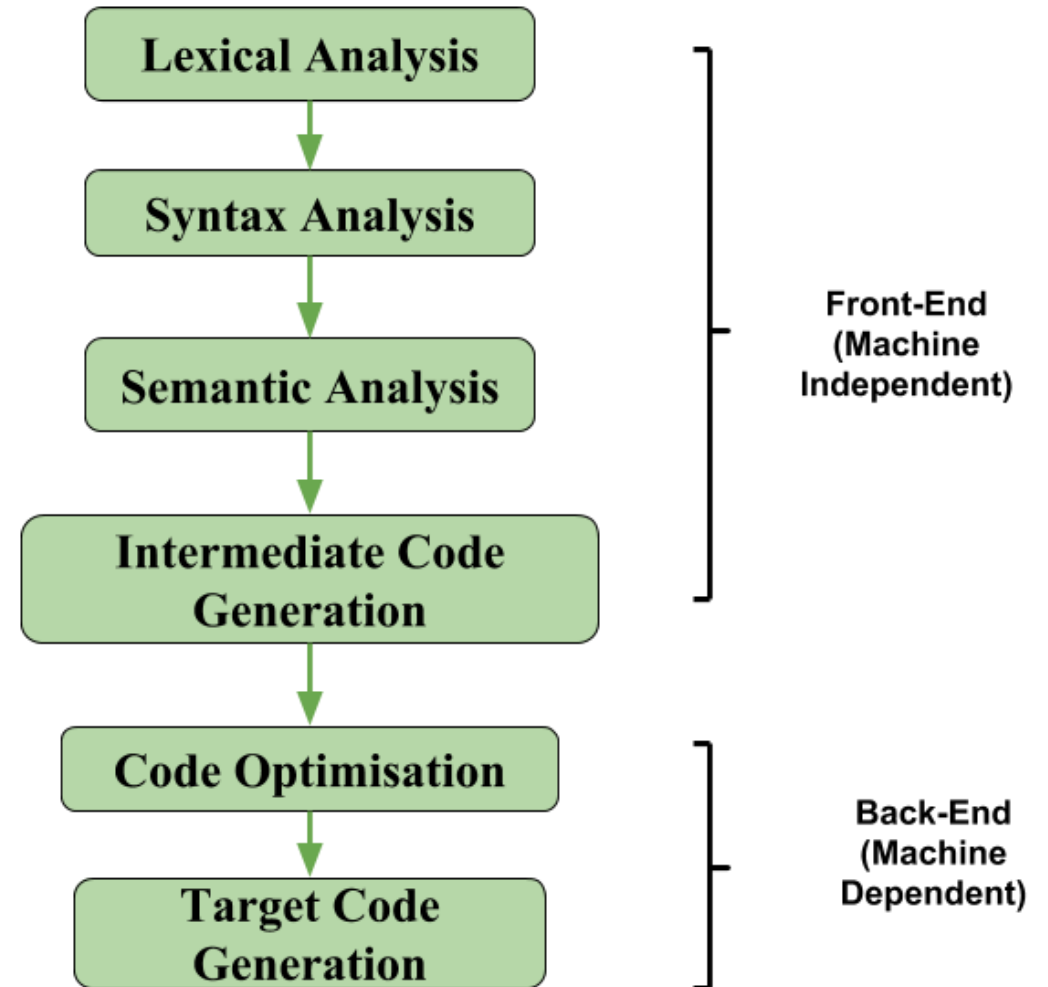
# The middle-end of a compiler

## Definition (The middle-end part of a compiler):

The **middle-end of a compiler** follows the front-end analysis and it consists of a series of operations and transformations to optimize and improve its efficiency.

It involves tasks, such as:

- **Intermediate code generation**
- **Code optimization,**
- and **Data-flow analysis.**

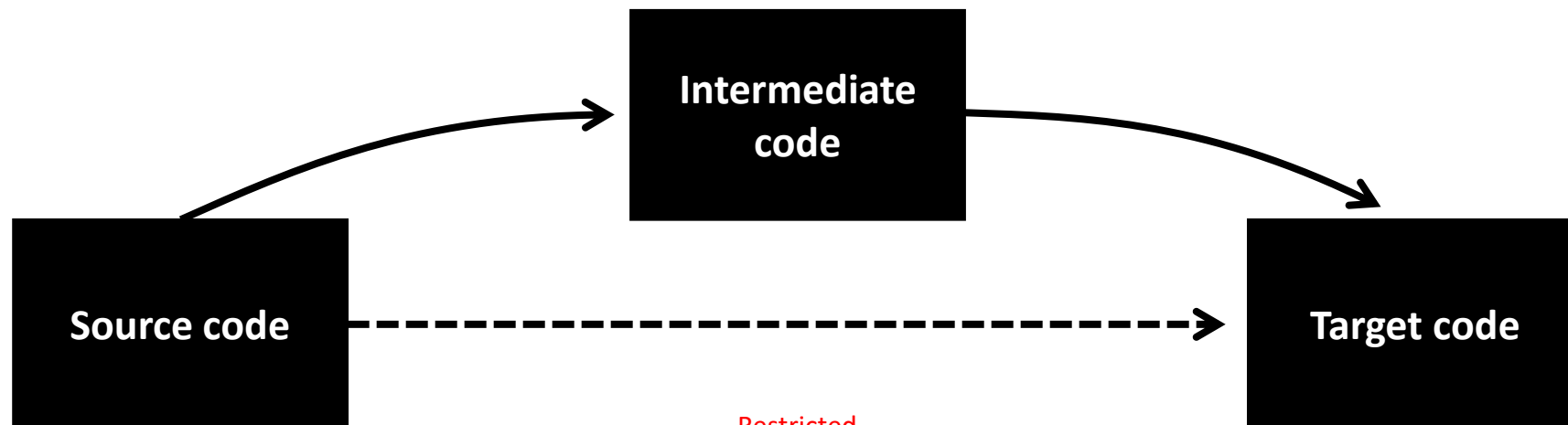


# Intermediate code generation

**Definition (Intermediate code generation):**

**Intermediate code generation** is the process of transforming the source code into a code that is more abstract and closer to machine language, but not machine language just yet.

It is a much-needed step as making a direct jump from source code to target code might prove difficult.



# Intermediate code generation

**Definition (Intermediate code generation):**

**Intermediate code generation** is the process of transforming the source code into a code that is more abstract and closer to machine language, but not machine language just yet.

It is a much-needed step as making a direct jump from source code to target code might prove difficult.

**In fact, an intermediate code representation is often easier to manipulate and allows for optimization.**

The intermediate code is not specific to any particular hardware or operating system and can be easily transformed into the final machine code during the backend phase of the compilation process.

# Intermediate code representations

## **Definition (intermediate code representations):**

The intermediate code generated during this phase is often represented in a language that is easier to manipulate than the original source code.

**Three-address code representations** are typical examples of intermediate code representation languages that we will investigate.

Other representations exist, but will not necessarily be covered:

- Virtual machine code,
- Abstract syntax trees,
- Etc.

# Three-address code representation

**Definition (Three-address code representation):**

**Three-address code** is a low-level intermediate code representation used by compilers to facilitate optimization and code generation.

It is called “**three-address**” because each instruction in the code **can have at most three operands**.

A typical three-address code instruction has the following format:

$$\textit{operand1} = \textit{operand2} \textit{operator} \textit{operand3}$$

# Three-address code representation

For instance, the C code below can be transformed...

...into its equivalent three-address code representation.

```
#include <stdio.h>

int main() {
    int x = 10;
    int y = x + 5;
    printf("The value of y is %d\n", y);
    return 0;
}
```

```
t1 = 10
t2 = t1 + 5
y = t2
printf("The value of y is %d\n", y)
```

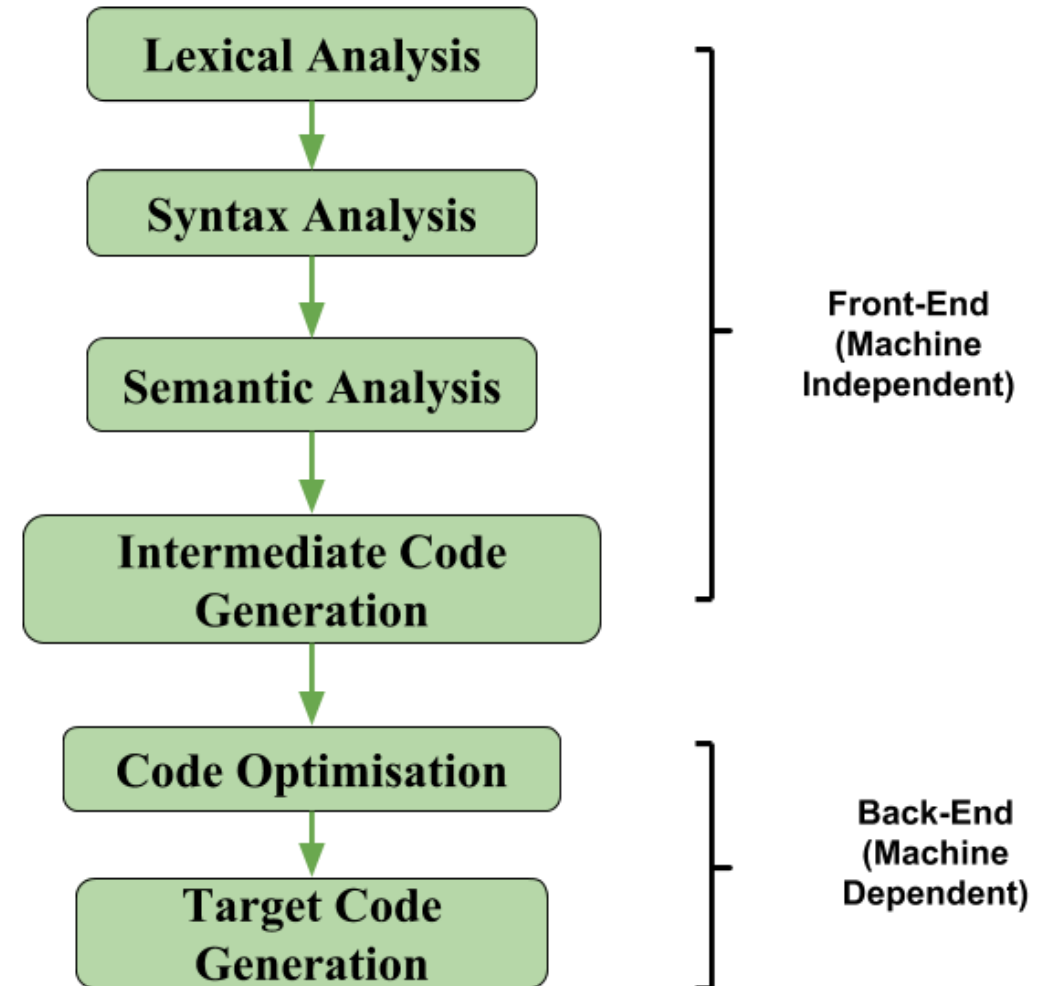
# The back-end of a compiler

**Definition (The back-end part of a compiler):**

The **back-end of a compiler** takes the optimized and transformed code generated by the middle-end and translates it into machine code that can be executed.

It involves tasks, such as:

- **Instruction selection/ordering,**
- **Register allocation,**
- **and Code generation/optimization.**



# Code generation

## Definition (**code generation**):

The **code generation** phase is responsible for generating machine code based on the intermediate code representation of the program produced by the middle-end part of the compiler.

This involves translating the intermediate representation into a low-level machine language that can be executed by the CPU (or GPU!).

In its most advanced versions, the code generator must take into account the specifics of the target architecture, such as the instruction set, memory hierarchy, and addressing modes.

And then, after the machine code is ready, we execute it, as explained in [50.002 Computation Structures!](#)



# Recall your machine code from 50.002!

The machine code, to be executed by the CPU has

- **Decomposed the source code into basic instructions understandable by the CPU (or GPU),**
- **Identified registers to use to store (and free) variables for each of these operations.**

Need a refresher? Have a look at your [50.002 Computation Structures!](#)

```
movl 10, Reg1 ;10 into Reg1
movl 5, Reg2  ;5 into Reg2
add Reg3, Reg1, Reg2 ;add Reg1 + Reg2 store in Reg3
ret  Reg3 ;return value in Reg3
```

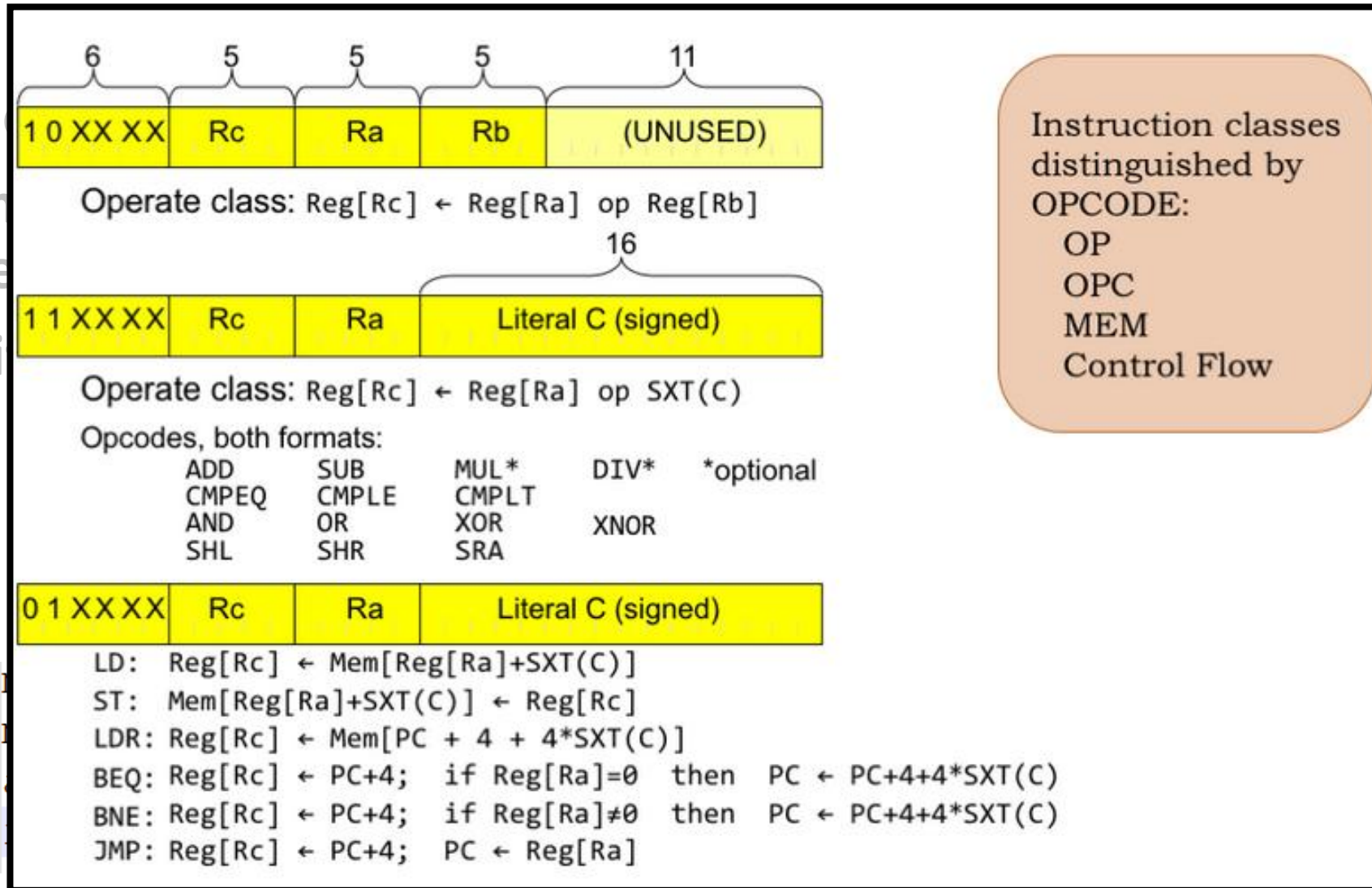
# Recall your machine code from 50.002!

The machine code is

- Decorated by the

- Identified these

Need a



undable

h of

ctures!

3

## Source code (C)

```
int main() {  
    int x = 10;  
    int y = 5;  
    int z = x + y;  
    return z;  
}
```

Restricted

## Front-end

Lexical Analysis  
Syntax Analysis  
Semantic Analysis

## After front-end, intermediate code (Three-address) and optimizations

```
t1 = 10  
t2 = 5  
t3 = t1 + t2  
z = t3  
return z
```

## Machine instructions after code generation and execution

```
movl 10, Reg1  
movl 5, Reg2  
add Reg3, Reg1, Reg2  
ret Reg3
```

Restricted

# Instruction Ordering/Scheduling

These two are equivalent but prove that there might be scenarios where reordering instructions could prove beneficial!

```
#include <stdio.h>

int main() {
    int x = 7;
    int y = 2;
    int z = 5;
    int w = 3;
    int result = (x + y) * (z - w);
    printf("The result is %d\n", result);
    return 0;
}
```

```
mov r1, [x]
mov r2, [y]
add r3, r1, r2
mov r4, [z]
mov r5, [w]
sub r6, r4, r5
mul r7, r3, r6
mov [result], r7
```

```
mov r1, [x]
mov r2, [y]
mov r4, [z]
mov r5, [w]
sub r6, r4, r5
add r3, r1, r2
mul r7, r3, r6
mov [result], r7
```

# Instruction Ordering/Scheduling

## Definition (**Instruction Ordering/Scheduling**):

The **instruction ordering** (or **scheduling**) phase is responsible for rearranging the order of instructions in the program to improve performance.

This typically involves reordering instructions to take advantage of the parallelism and pipelining features of modern processors (with many cores running in parallel, for instance).

The instruction scheduler takes into account the dependencies between instructions and the specific features of the processor.

*(Too advanced, could have been interesting, but out-of-scope).*

# Code optimization

## Definition (**Code optimization**):

**Code optimization** is a critical process that involves analysing and modifying the intermediate code to make it run faster and consume fewer resources. Some of the optimizations typically include:

1. **Dead code elimination:** identify and remove **dead code** that is not executed during program execution.  
This allows to reduce the size of the final compiled code.

# Dead code

## Definition (dead code):

**Dead code** is code that is never executed during the runtime of a program, often following from bad code logic or design.

```
#include <stdio.h>

int main() {
    int x = 10;
    if (x < 5) {
        printf("x is less than 5\n");
    }
    // This code below is dead code, as the condition above will always evaluate to false
    else if (x < 8) {
        printf("x is less than 8\n");
    }
    else {
        printf("x is greater than or equal to 8\n");
    }
    return 0;
}
```

# Dead code

**Question:** Can you see why the code shown on the right is containing some dead code?

```
#include <stdio.h>

int main() {
    int x = 10;
    if (x < 5) {
        printf("x is less than 5\n");
    }
    // This code below is dead code, as the condition above will always evaluate to false
    else if (x < 8) {
        printf("x is less than 8\n");
    }
    else {
        printf("x is greater than or equal to 8\n");
    }
    return 0;
}
```



# Dead code

**Answer:** The *else if* statement is the dead code, as the preceding if statement will always evaluate to false since x is initialized to 10.

```
#include <stdio.h>

int main() {
    int x = 10;
    if (x < 5) {
        printf("x is less than 5\n");
    }
    // This code below is dead code, as the condition above will always evaluate to false
    else if (x < 8) {
        printf("x is less than 8\n");
    }
    else {
        printf("x is greater than or equal to 8\n");
    }
    return 0;
}
```

# Code optimization

## Definition (**Code optimization**):

**Code optimization** is a critical process that involves analysing and modifying the intermediate code to make it run faster and consume fewer resources. Some of the optimizations typically include:

- 2. Constant folding:** replace multiple expressions that involve constant values with their computed values.  
This allows to reduce the number of computations performed during program execution.

# Constant folding

As an example of constant folding...

In the code below, the expression  $x = 2 + 3$  can be folded into a simpler expression  $x = 5$  during compilation, eliminating the need to perform the addition operation at runtime.

```
#include <stdio.h>

int main() {
    int x = 2 + 3; // This expression is constant folded at compile-time
    printf("The value of x is %d\n", x); // The output will be "The value of x is 5"
    return 0;
}
```

# Code optimization

## Definition (**Code optimization**):

**Code optimization** is a critical process that involves analysing and modifying the intermediate code to make it run faster and consume fewer resources. Some of the optimizations typically include:

**3. Control flow optimization:** optimize the instructions and the order of instructions being executed.

This allows, for instance, to reduce the number of necessary branch instructions, removing parts that could almost be considered **dead code** (even though they did not follow from bad logic this time).

# Control flow optimization

As an example of control flow optimization...

The program below uses an *if-else statement* to check if the value of *x* is less than the value of *y*.

```
#include <stdio.h>

int main() {
    int x = 10, y = 20;
    if (x < y) {
        printf("x is less than y\n");
    } else {
        printf("x is greater than or equal to y\n");
    }
    return 0;
}
```

# Control flow optimization

As an example of control flow optimization...

The program below uses an *if-else statement* to check if the value of *x* is less than the value of *y*.

Control flow optimization analyses the code and can determine that *x* will always be less than *y*, as they are constants.

It can simplify the code to eliminate the unnecessary *if-else statement* and simplify the code even further.

```
#include <stdio.h>

int main() {
    int x = 10, y = 20;
    printf("x is less than y\n");
    return 0;
}
```

# Code optimization

## Definition (**Code optimization**):

**Code optimization** is a critical process that involves analysing and modifying the intermediate code to make it run faster and consume fewer resources. Some of the optimizations typically include:

- **Dead code elimination,**
- **Constant folding,**
- **Control flow optimization,**
- **Many other operations: Function Inlining (substituting secondary functions into main), Loop Unrolling (replaces a loop with a fixed number of iterations with a series of unrolled iterations), etc.**

# Register Allocation

## Definition (**Register Allocation**):

During the **register allocation** phase, the compiler assigns program variables to the different types of memories, in an efficient way.

As seen in [50.002 Computation Structures](#), this is important because CPU registers are much faster than other types of memory, but are very limited in numbers (a.k.a memory hierarchy).

The register allocator must take into account the constraints imposed by the CPU architecture, such as the number and type of available registers, but also identify in the code variable that are used many times to give them priority in terms of memory space.

*(Could have been interesting, but probably out-of-scope).*



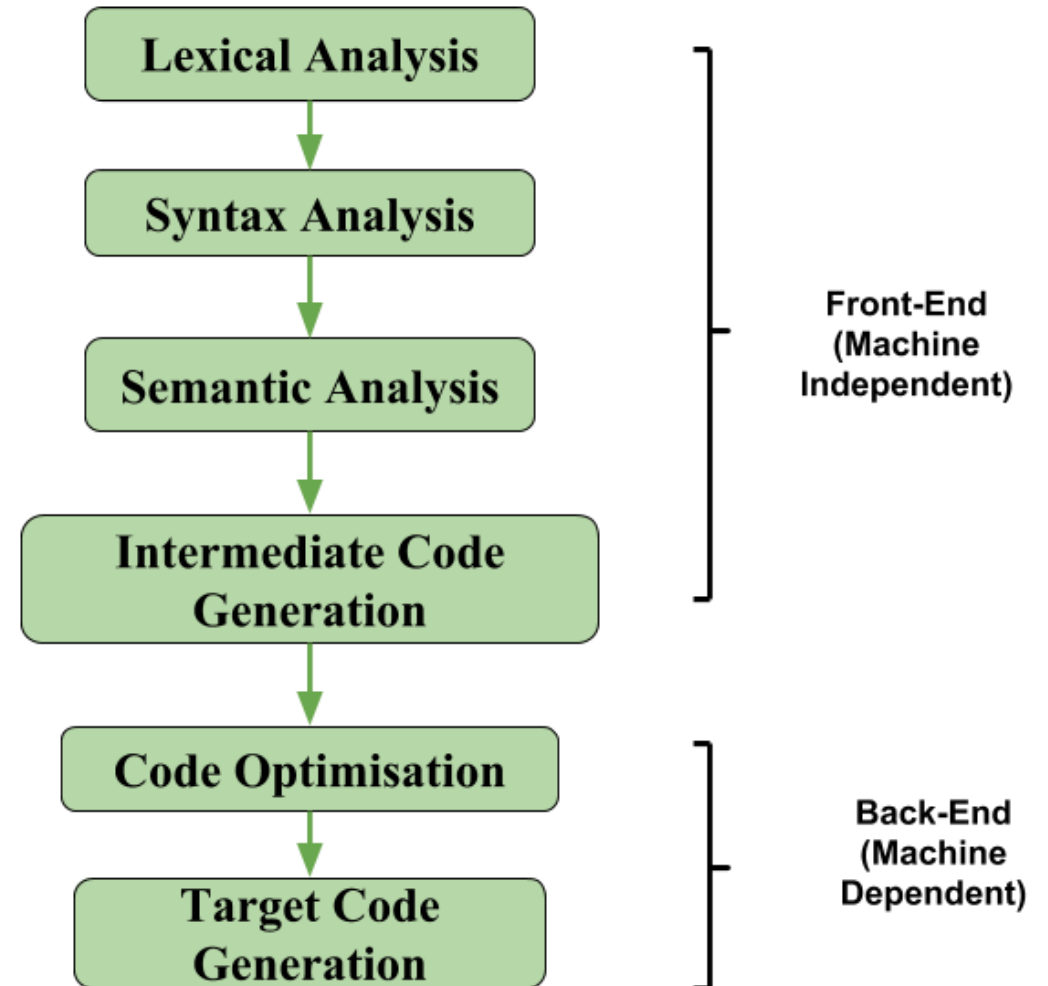
# The architecture of a compiler

## Definition (the three parts of a typical compiler architecture):

A typical compiler architecture consists of three main components: the **front-end**, **middle-end**, and **back-end**.

Each component is responsible for a specific set of tasks and works together to generate the final executable code.

**Hopefully, this gives us a roadmap!**



# Quiz

## **What is the middle-end phase of a compiler?**

- a. The phase that analyses and optimizes an abstract syntax tree or another intermediate code representation.
- b. The phase that generates machine code from the intermediate representation (IR) code.
- c. The phase that performs lexical analysis and tokenization of the source code.
- d. The phase that performs typos checking and reporting.

# Quiz

**What is the middle-end phase of a compiler?**

- a. The phase that analyses and optimizes an abstract syntax tree or another intermediate code representation.**
- b. The phase that generates machine code from the intermediate representation (IR) code.
- c. The phase that performs lexical analysis and tokenization of the source code.
- d. The phase that performs typos checking and reporting.

# Quiz

## **What is the purpose of instruction selection in a compiler?**

- a. To perform static analysis of the program to identify possible runtime errors.
- b. To generate an optimized intermediate representation (IR) that can be used for code generation.
- c. To map high-level or intermediate-level language constructs to low-level machine instructions that can be executed by the processor.
- d. To optimize the code for specific target architectures and instruction sets.

# Quiz

**What is the purpose of instruction selection in a compiler?**

- a. To perform static analysis of the program to identify possible runtime errors.
- b. To generate an optimized intermediate representation (IR) that can be used for code generation.
- c. To map high-level or intermediate-level language constructs to low-level machine instructions that can be executed by the processor.**
- d. To optimize the code for specific target architectures and instruction sets.

# Quiz

## **What is register allocation in a compiler?**

- a. The process of mapping variables to their literal values.
- b. The process of transforming high-level language constructs into low-level machine code.
- c. The process of selecting the best set of machine instructions to implement a program.
- d. The process of mapping variables to hardware registers in the target architecture (CPU or GPU).

# Quiz

## What is register allocation in a compiler?

- a. The process of mapping variables to their literal values.
- b. The process of transforming high-level language constructs into low-level machine code.
- c. The process of selecting the best set of machine instructions to implement a program.
- d. **The process of mapping variables to hardware registers in the target architecture (CPU or GPU).**

# Quiz

**The compiler is a computer program, and this program has necessarily been compiled at some point.  
Which language has then been used to write the C compiler?**

- a. HTML and PHP...!
- b. A language older than C like Fortran?
- c. C itself?!
- d. Machine code only?!?



# Quiz

**The compiler is a computer program, and this program has necessarily been compiled at some point.  
Which language has then been used to write the C compiler?**

- a. ~~HTML and PHP~~ (lol)
- b. A language older than C like Fortran? C--? B?
- c. C itself?!
- d. Machine code only?!?

# Quiz

**The compiler is a computer program, and this program has necessarily been compiled at some point.**

**Which language has then been used to write the C compiler?**

a. ~~HTML and PHP (lol)~~

b. ~~A language older than C like Fortran? C--? B?~~

(Technically no, because it would not resolve the problem of which language was used to compile this compiler in Fortran then)

c. C itself?!

d. Machine code only?!?!

# Quiz

**The compiler is a computer program, and this program has necessarily been compiled at some point.**

**Which language has then been used to write the C compiler?**

a. ~~HTML and PHP (lol)~~

b. ~~A language older than C like Fortran? C--? B?~~

(Technically no, because it would not resolve the problem of which language was used to compile this compiler in Fortran then)

c. **C itself?!**

d. Machine code only?!?

# Quiz

**The compiler is a computer program, and this program has necessarily been compiled at some point.**

**Which language has then been used to write the C compiler?**

a. ~~HTML and PHP (lol)~~

b. ~~A language older than C like Fortran? C--? B?~~

(Technically no, because it would not resolve the problem of which language was used to compile this compiler in Fortran then)

c. **C itself?!**

d. **Machine code only?!?!?**

# Bootstrapping

## Definition (**Bootstrapping**):

**The latest version of the C compiler is typically written in C itself.**

The source code for latest version of the C compiler is then compiled using an older version of the compiler that is already available.

This process is called **bootstrapping**, and it allows a C compiler to be built from scratch by reusing and improving an existing compiler.

Once a basic C compiler is built, it can be used to compile other programs written in C, including newer versions of the C compiler itself with more advanced features.

# Bootstrapping

**Definition (Bootstrapping):**

**The latest version of the C compiler is typically written in C itself.**

The source code for latest version of the C compiler is then compiled using an older version of the compiler that is already available.

This process is called **bootstrapping**, and it allows a C compiler to be built from scratch by reusing and improving an existing compiler.

Once a basic C compiler is built, it can be used to compile other programs written in C, including newer versions of the C compiler itself with more advanced features.

→ Ok, but then what was the language used for the “**first**” compiler?

# The mother of all compilers

→ Ok, but then what was the language used for the “**first**” compiler?

- The first C compiler was developed by **Dennis Ritchie** at Bell Labs in the early 1970s (this is the same person who invented the language).
- The first C compiler was written in **assembly language**.
- After that, every time a new version of the C compiler was needed, we would use **bootstrapping**.

# The mother of all compilers

This “first” compiler had some very basic features:

1. It supported the basic features of the C language, including data types, control structures, and functions.
2. It produced assembly language code as output, which could be assembled into machine code using a separate assembler.
3. It used a simple two-pass compiler architecture, where the first pass performed lexical analysis and created a symbol table, and the second pass performed syntax analysis and code generation.
4. It included a rudimentary optimizer that performed simple optimizations such as constant folding and algebraic simplification.
5. It had a limited set of error messages and debugging capabilities, which made it difficult to diagnose and fix errors in the code.



# The mother of all compilers

Despite its limitations, this “first” C compiler was a significant achievement because it allowed C to become a popular language for system programming and operating systems development.

The availability of a reliable and efficient C compiler helped to make C a popular choice for operating system development, as well as for writing other higher-level programming compilers/interpreters.

Learn more about Dennis Ritchie and his founding (and very much underrated) work on C and compilers:

<https://www.youtube.com/watch?v=g3jOJfrOknA>

# A bit of history on compilers

References and extra reading, if any (for those of you who are curious):

- A bit of history of compilers

<https://www.geeksforgeeks.org/history-of-compiler/>

# Compilers: benefits and disadvantages

## Benefits of compilers

- **Performance**: Compiled code is often **faster and more efficient** than interpreted code because it is translated into machine code beforehand. This leads to improved application performance, especially for computationally intensive tasks.
- **Optimizations**: Compilers can perform **complex optimizations** that are difficult or impossible for interpreters to achieve (to be discussed later, e.g. loop unrolling, function in-lining, and constant propagation).
- **Portability**: Compiled code can be packaged into **standalone executable files**, making it easier to distribute and run on different platforms.

## Disadvantages of compilers

- **Development time**: Compilers **require a separate compilation step before the code can be executed**. In addition, code will only compile if flawless. This can make development and testing more difficult and less beginner-friendly.
- **Memory management**: Some compiled languages may require more **complex memory management**, which can be challenging for novice programmers.
- **Platform-specific code**: Compilers can generate **machine code that is specific to the platform it is compiled on**, which can limit portability across different platforms.

# Interpreters: benefits and disadvantages

## Benefits of interpreters

- **Interactivity**: Interpreters allow for greater interactivity during the development process, because **changes to the code can be seen immediately** without the need for a separate compilation step.
- **Flexibility**: Interpreters are **more flexible** than compilers because they can **interpret code on-the-fly**, allowing for greater dynamic behavior and runtime adaptability.
- **Ease of debugging**: Because interpreters execute their code line-by-line, it is often **easier to debug** code in an interpreter than in a compiler.

## Disadvantages of interpreters

- **Performance**: Interpreted code is **often slower and less efficient** than compiled code because it is translated into machine code on-the-fly during execution.
- **Limited optimizations**: Interpreters can perform **limited optimizations** and may not be able to achieve the same level of performance optimizations as a compiler.
- **Lack of portability**: Interpreted code often **requires a specific interpreter to run**, which can limit portability across different platforms.