

50.051 Programming Language Concepts

W8-S1 Introduction to Compilers and Course Outline

Matthieu De Mari



SINGAPORE UNIVERSITY OF
TECHNOLOGY AND DESIGN

A quick word about instructors

- Matthieu (**Matt**) De Mari
- Lecturer at SUTD (Python, Deep Learning, AI, and more)
- Information Systems Technology and Design (ISTD) pillar/faculty
- PhD from CentraleSupélec (France)
- Email: matthieu_demari@sutd.edu.sg
- Office @ SUTD: 2.401.07



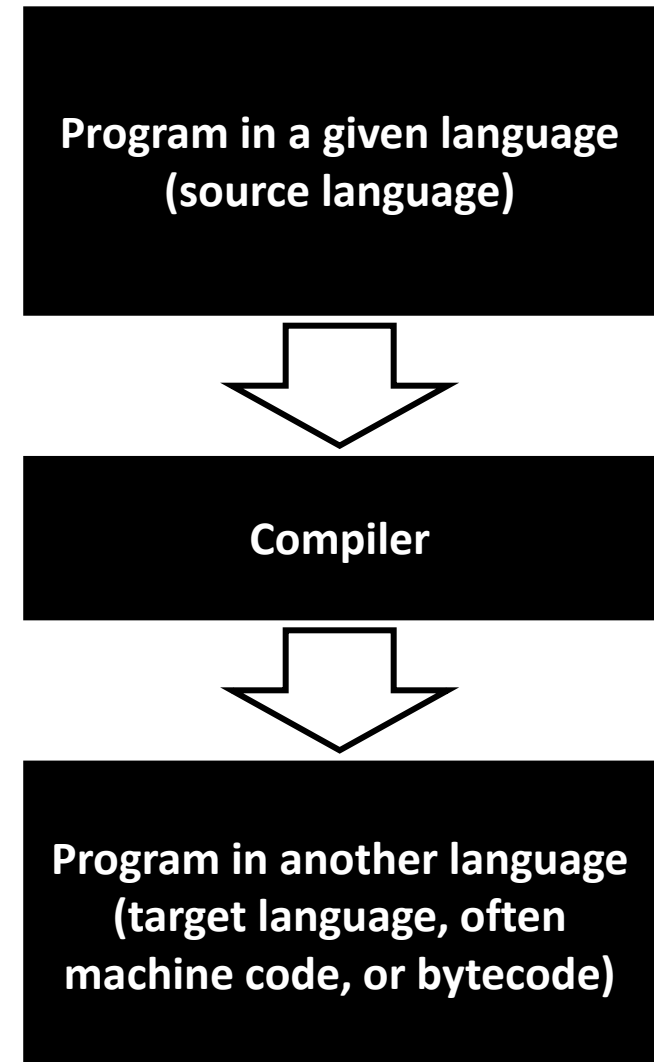
Compiler: a definition

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 **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**.

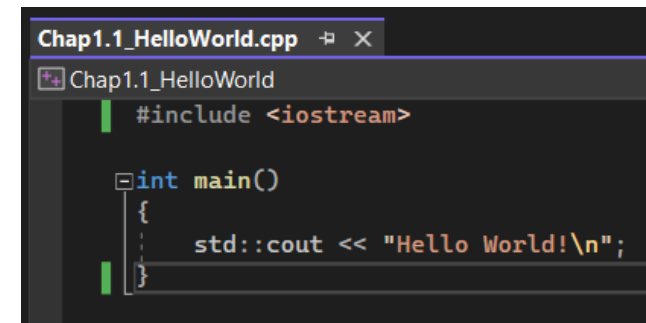
Compiler: a definition

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).

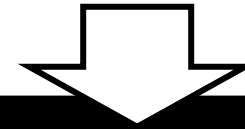
Compilers will typically prepare our source code for **execution**.



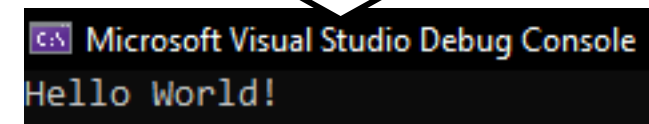
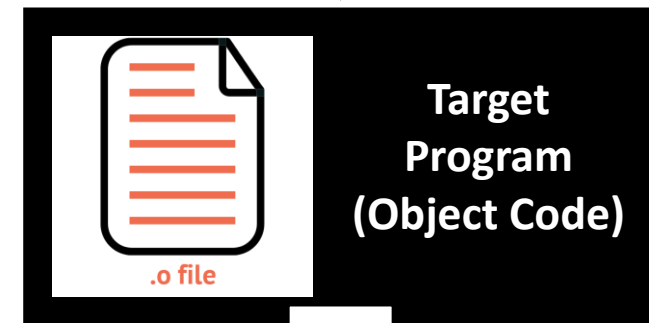
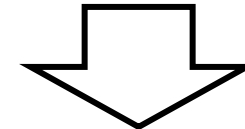
```
Chap1.1_HelloWorld.cpp
Chap1.1_HelloWorld
#include <iostream>

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.

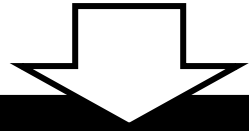
This typically allows for **optimization routines** to be run on the intermediate program **before the final translation** and **upcoming execution** (For instance, checking for errors and optimizing your computer resources).

Restricted

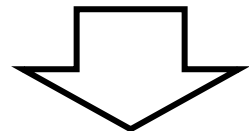
```
Chap1.1_HelloWorld.cpp
Chap1.1_HelloWorld
#include <iostream>

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

Source Program (C/C++)



Compiler



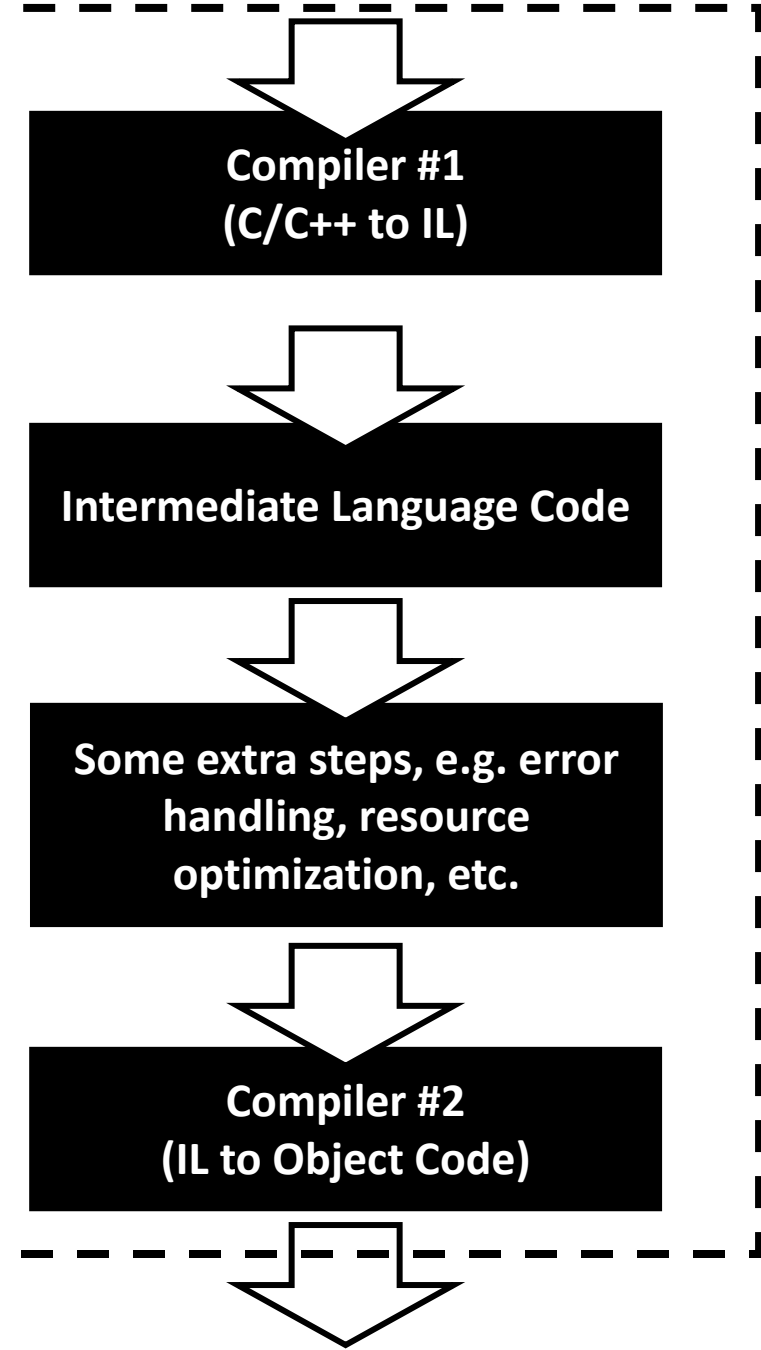
.o file

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?

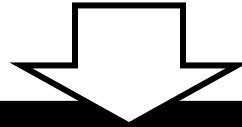
Interpreters have the **same objective as compilers**, i.e. **translate a source program into target code which can be executed by the CPU**.

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

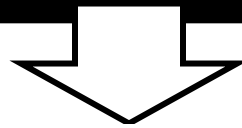
In [*]:

...

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.

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.

Just-in-time compilation

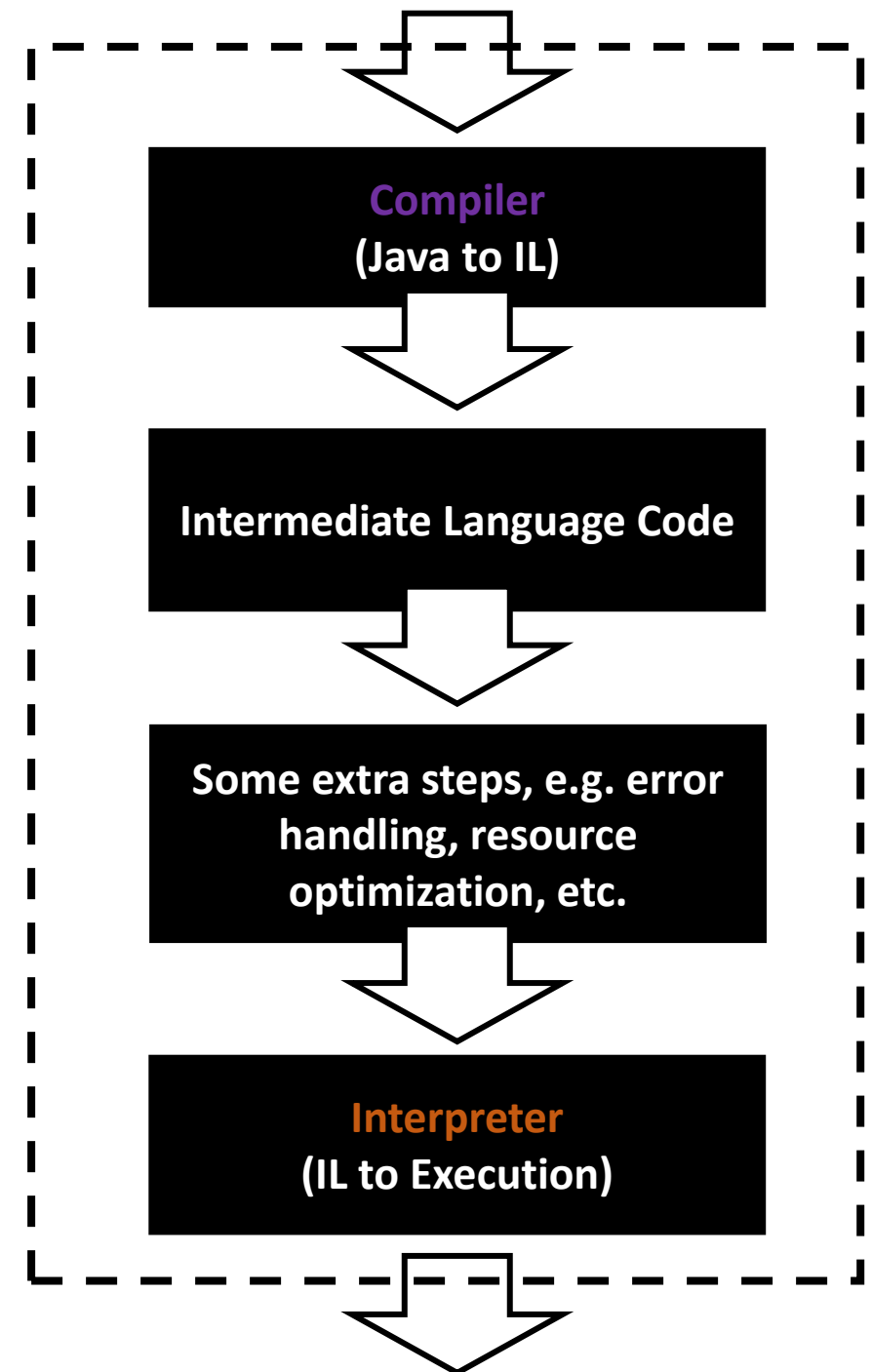
Definition (**Just-in-time Compilers**):

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 you study compilers in a Computer Science degree?

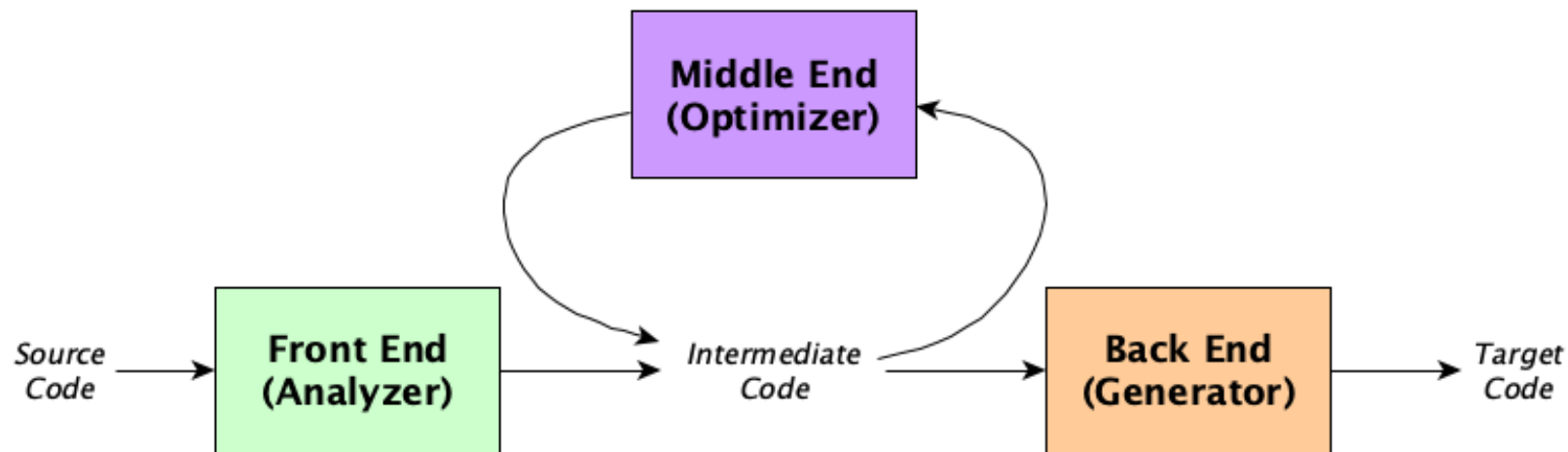
1. A great introduction to techniques for translation and improvement of programs.
2. A fantastic practical exercise in software engineering.
3. Provides an understanding on what happens in the background when pressing the “Run” button of your programs.
4. A great tool for understanding where typical compilation/interpretation errors come from.
5. A better understanding of how computer resources are being used to run programs.
6. A much needed introduction course to compilers/interpreters, if one day you plan on writing your own programming language.

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.

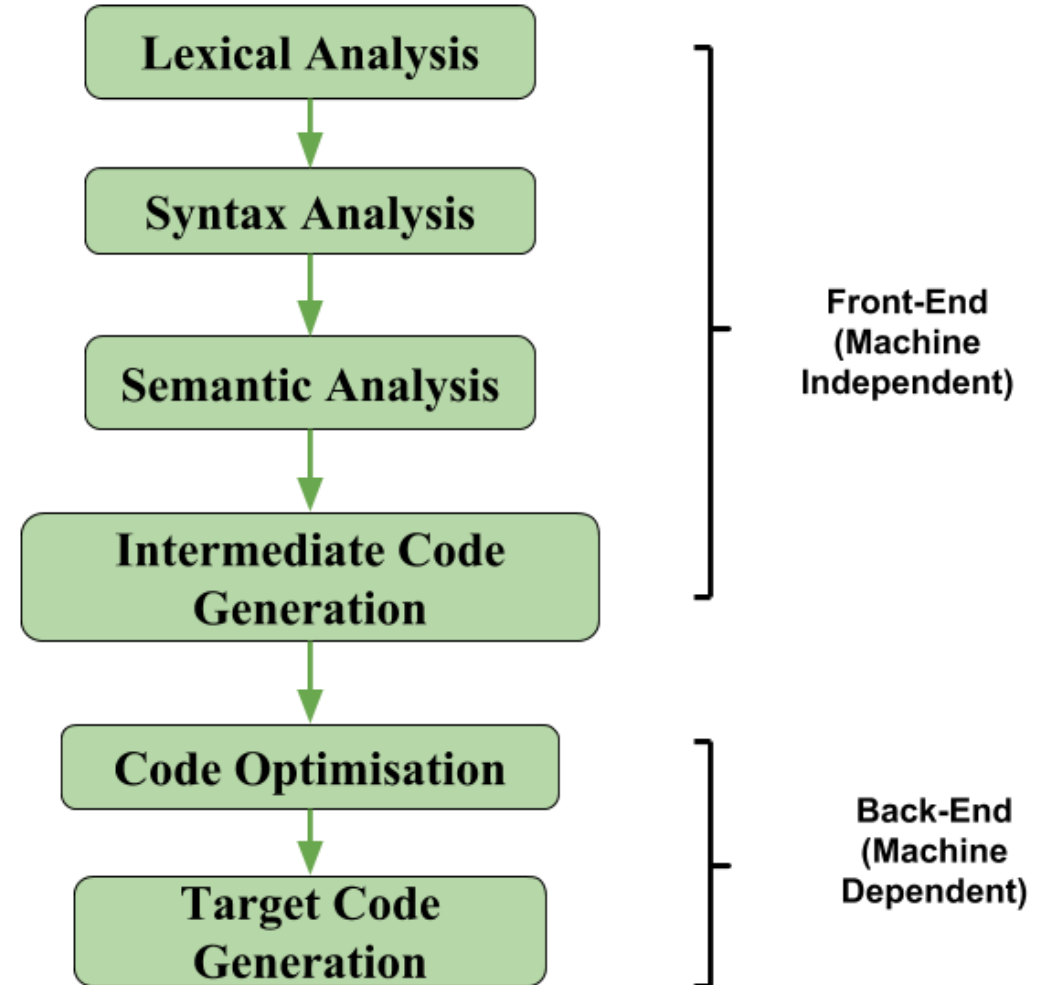


The architecture of a compiler

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

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

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



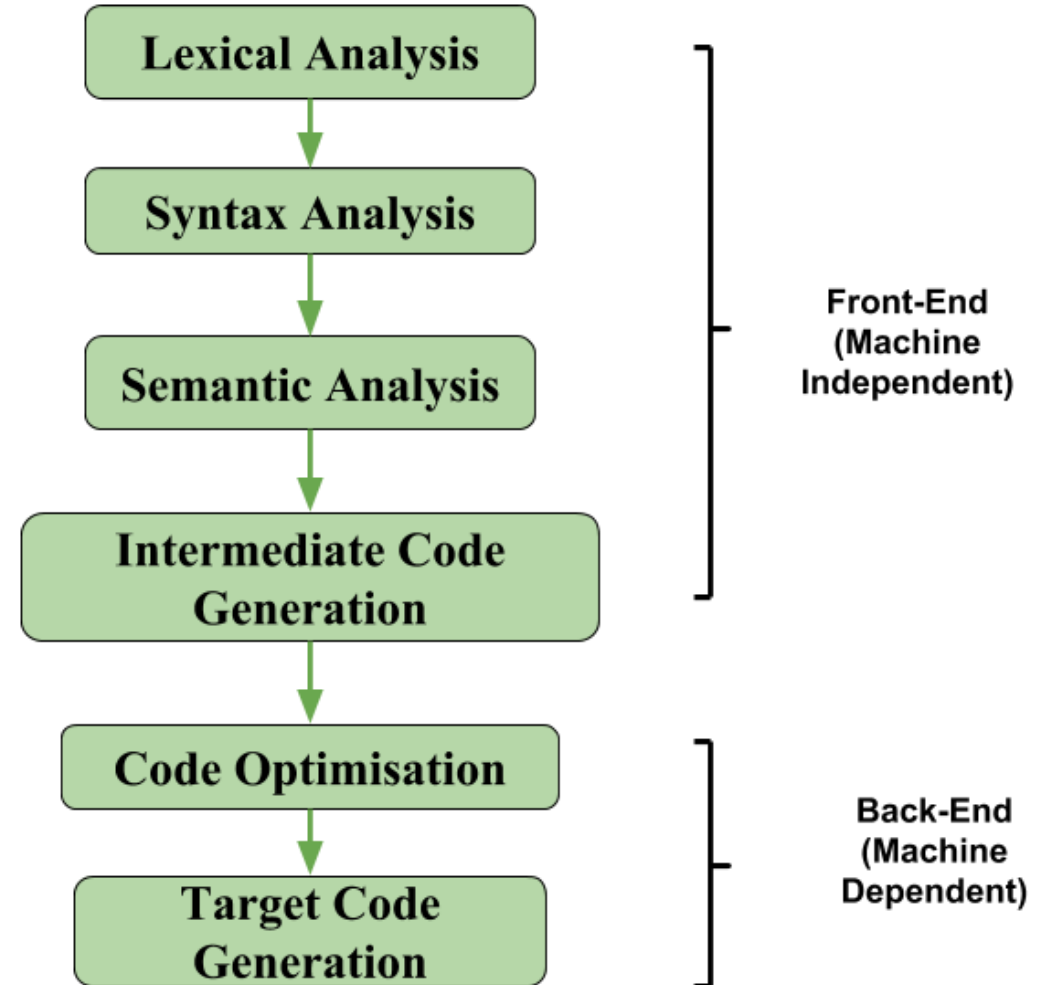
The front-end of a compiler

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

The **front-end of a compiler** is responsible for analysing the source code, and 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.**



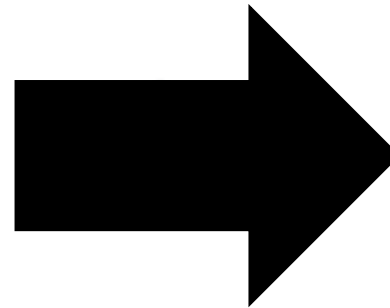
Lexical Analysis

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;
    )
}
```



KEYWORD (int)
IDENTIFIER (min)
PUNCTUATION (open_par)
KEYWORD (int)
VARIABLE (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.

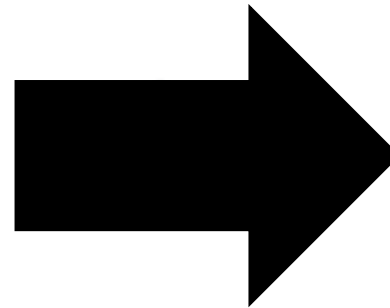
Lexical Analysis

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** (something for Week 9), **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, as it allows the compiler to understand the structure and meaning of the program at a high level.

- By breaking the code down into a series of tokens, the compiler can analyse the program more efficiently and accurately, and generate optimized machine code that can be executed by the computer.
- **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 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 the **BNF (Backus-Naur Form) notation**.

It consists of a set of production 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> ]
```


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 the **BNF (Backus-Naur Form) notation**.

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

(More on this in Week 10).

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 by the lexical analysis stage.

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

Parse Tree

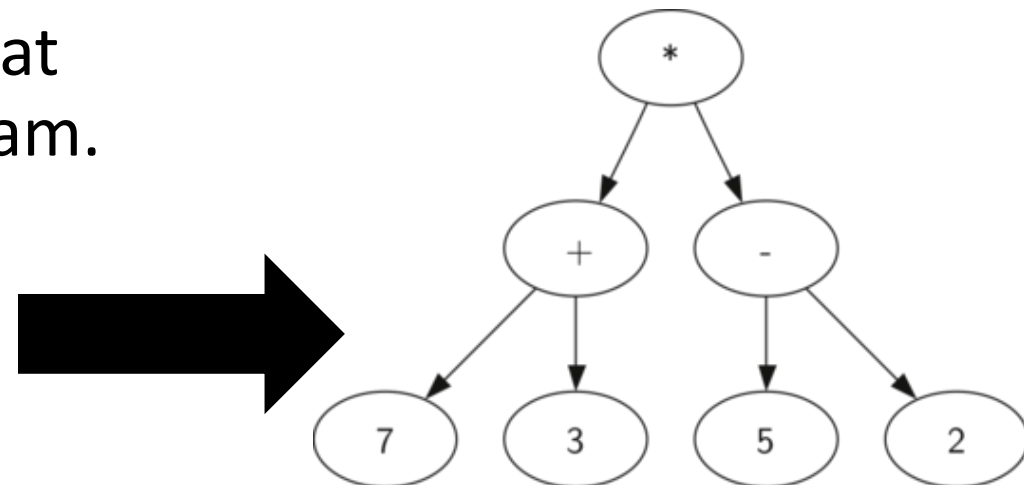
Definition (**Parse Tree**):

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
```



Parse Tree

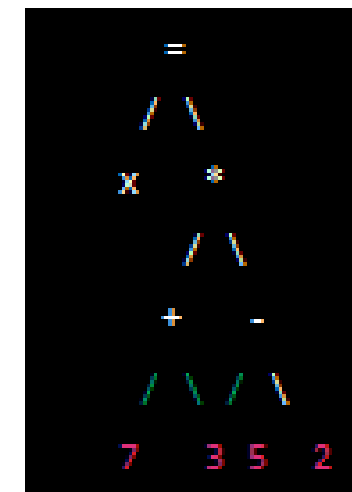
Definition (**Parse Tree**):

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, or mismatched parentheses or braces.
- *(More on this on W10 and W12-13)*

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 or syntax tree, the compiler can identify the different components/tokens of the program and their relationships to each other, and generate optimized machine code that can be executed by the computer.
- In this phase, the compiler also checks the program for semantic errors, such as type mismatches or undeclared variables.
- **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 or type checking phase.**

Semantic Analysis

Definition (**Semantic Analysis**):

Semantic Analysis is the third major phase of the compilation process, following lexical and syntax analysis.

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

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, 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.

Each of these subtasks also serve to catch errors and inconsistencies that might cause the program to behave incorrectly or unpredictably at runtime.

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

As an example for type checking,

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

Type checking will typically cover for these errors.

```
#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 some languages if a variable is declared inside a function, it should only be accessible within that function and not outside of it.

Scope Analysis

As an example for scope analysis,

- in some languages if a variable is declared inside a function,
- it should only be accessible within that function and not outside of it.

Scope analysis will typically cover for these errors.

```
#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: you could have two variables with name “x” in two different functions, but the program should understand the difference between the two.

Name Resolution

As an example for name resolution,

- you could have two variables with name “x” in two different functions,
- but the program should be able to understand the difference between the two.

Name resolution will typically cover for these scenarios.

```
#include <stdio.h>

void function1() {
    int x = 10;
    printf("The value of x in function1 is %d\n", x);
}

void function2() {
    int x = 20;
    printf("The value of x in function2 is %d\n", x);
}

int main() {
    function1();
    function2();
    return 0;
}
```

Semantic Analysis Recap

- Overall, semantic analysis is an important part of the compilation process that ensures that programs are both syntactically and semantically 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 ensures that the program is well-formed and executable.

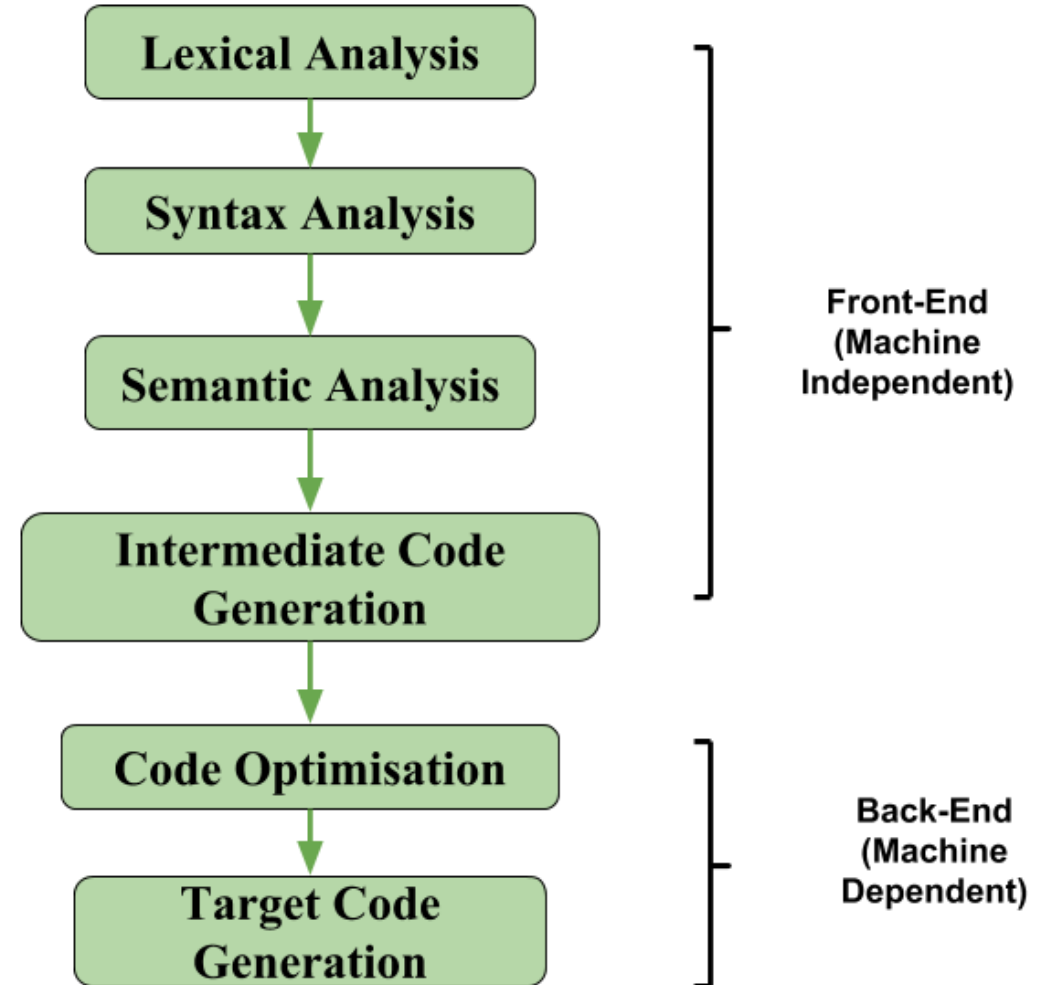
The front-end of a compiler

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

The **front-end of a compiler** is responsible for analysing the source code, and 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.**



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

A bit of history on compilers

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

- A bit of history on compilers

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