Supplementary technical guide by

Amaris Vélez

(1) Compiler theory and practical programming

Programming Language Theory (PLT) is focused on various aspects concerning a programing language such as its design, implementation, analysis, characterization, and classification [5]. PLT also intends to identify the features that are unique to a programming language. On the other hand, practical programming refers to the use of a specific programming language to solve a specific problem. Both the theoretical (compiler theory) and practical aspects behind programming will be discussed through this chapter. The specific programming language that will be used for the practical section will be C++ and Python due to their current popularity in the field of computing.

(1.1) Historical Background

Programming dates back to the 1800s when Joseph Marie Jacquard taught a loom (device used to weave cloth or tapestry) to read punch cards [6]. This loom was the first heavily multi-threaded processing unit. Furthermore, this loom inspired Charles Babbage to create the Analytical Engine in 1837, which was the first general-purpose computer. This creation presented the need for programming languages to facilitate the instruction of computers to perform specific tasks through a set of grammatical rules and an established vocabulary. As a result, programming languages were born. Ada Lovelace is credited for being the first person to write a computer program in 1843 when she described an algorithm to compute Bernoulli numbers in the Analytical Engine [7]. Then, in 1936 Alan Turing devised the Turing Machine; a mathematical model for a machine capable of manipulating symbols on a strip of tape according to a table of rules. Through this model, the solution for any problem known to be computable could be computed by a machine. Thus, the idealization of the Turing Machine enabled the creation of modern computing machines and formal programming languages.

Following Turing's model, the US Government was able to build the ENIAC in 1942, which is considered by many to be the first computer that uses electrical signals instead of physical motion (i.e. gears) to execute a computation [8]. In 1945, the physicist John von Neumann devised what is today known as the von Neumann architecture along with two important concepts, stored-program technique, and conditional control transfer, which transformed the idealization of programming languages [8] - [9]. The stored-program technique eliminated the need to hand-wire computer hardware for each program by using complex instructions instead which allowed for hardware to be reprogrammed faster [8], [10]. On the other hand, the concept of conditional control transfer introduced the idea of subroutines (small blocks of code that can be jumped to in any order) and libraries (blocks of code that can be reused) [8], [10]. Additionally, conditional control transfer stated that any computer code should be able to implement branches through logical statements such as the conditional IF-THEN constructs and loops such as the FOR construct [8].

A few years after von Neumann's work, in 1949, the language known as Short Code was devised and functioned as the first computer language for electronic devices [9] – [10]. However, this language came with the problem that forced programmer's to change their statements into 1's and 0's by hand. This problem was eventually solved by Grace Hooper in 1951 through the creation of A-0, the first compiler (a program that turns the language's statements into 0's and 1's) [11]. Then, the first widely accepted programming language, FORTRAN, was developed in 1957 by John Backus and colleagues at IBM [6] - [7]. Two years later, the second-oldest programming

language, LISP, was devised by John McCarthy [7]. Then in 1959, the first enterprise ready business-oriented programming language was invented by Grace Hooper and Bob Bemer [6] - [7]. Eventually, the first two object-oriented programming languages, Simula and Smalltalk, were developed in 1965 and 1972 respectively [7]. This inspired Bjarne Stroustrup to release in 1979 a new programming language known as C++, which is one of the most widely used programming languages according to IEEE Spectrum interactive ranking as of 2019 [7], [12]. From another point of view, Python is a bit younger than C++ since its development was initiated in 1989 by Guido van Rossum and its first version was released to the public in 1991 [7]

(1.2) Compiler theory

As previously stated in the historical background section, a compiler is a piece of software that translates a programmer's statements (written in high level language) into 1's and 0's (low level language or machine code) for the machine to understand. Compilers may also function as cross compilers or source-to-source compilers. A cross compiler is used to compile code in one machine and then executed the code in a machine different to the one in which the code was compiled [13]. On the other hand, a source-to-source compiler or transcompiler is used to translate source code from one programming language to another [13].

Many improvements in computer programming arose with the implementation of compilers since they facilitated the use of high-level programming languages. These types of languages make program development faster due to the fact that they are easier to read and understand. Additionally, compilers are capable of identifying common mistakes that may occur while programming in a high-level language and allow for the same program to be compiled in different machine languages to be able to run them in many different types of machines. From another point of view, it is important to note that a program hand-coded in machine language will always be faster than the same program written in a high-level language and then translated by a compiler into machine language. However, a good compiler must be able to produce machine code that is very close to the speed of hand-written execution code [14]. The way in which a compiler works can be divided into different phases. The ordering of these phases may vary between compilers and some phases may be split into additional phases. However, the most common division for the phases of a compiler is the following division which contains a total of seven phases [14].

Phase #1: Lexical Analysis

This phase uses a lexer to perform tokenization using regular expressions and pattern rules to define valid token. Tokenization refers to the act of breaking up a sequence of strings into pieces such as words, keywords, phrases, symbols, and other elements called tokens [13]. Tokens represent a unit of information in the source program. On the other hand, lexemes are a sequence of characters that are included in the source program according to the matching pattern of a token [14]. Thus, a lexeme represents an instance of a token. Once tokens are properly structured, they are sent to the parser to facilitate the syntax analysis phase as shown in Figure 1.

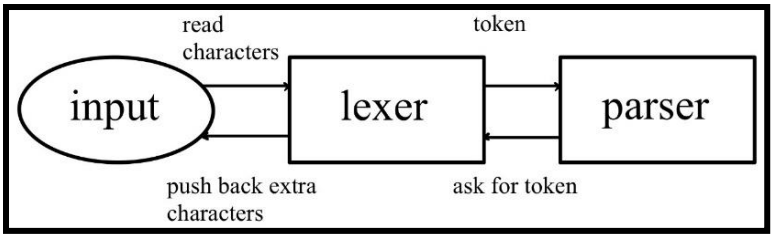

Figure 1: Relationship between lexical analyzer (lexer) and syntax analyzer (parser)

Figure 1 includes the three main steps involved in the lexical analysis phase which are the following [15]:

1. The parser asks for token.
2. After receiving token request, the lexer scan the input resulting from source program until a token is found.
3. The lexer sends the token to the parser. Note that if an invalid token is formed, the misplaced characters preventing validation will be pushed back.

Example 1.1
To obtain a better understanding of how tokenization occurs, consider the code shown in Figure

```
#include <stdio.h>
    int minimum(int x, int y) {
        // This will compare 2 numbers
        if (x < y)
            return x;
        else {
            return y;
        }
    }
```
Figure 2: Simple code in C++ for Example 1.1

By passing the code shown in Figure 2 through a lexer, the tokens shown in Table 1 are created.

Table 1: Created tokens after passing code in Figure 2 through a lexer

Lexeme	Token
int	keyword
minimum	identifier
(	operator
int	keyword
x	identifier
,	operator
int	keyword
y	identifier
)	operator
{	operator

if	keyword

Besides tokenization, the lexical analyzer also removes white space characters and comments from program code and facilitates the process of generating error messages by providing row and column number. Furthermore, it's important to note that the lexical analysis phase uses regular expressions and deterministic finite Automata (DFA) to search for patterns defined by language rules and then verify the validity of the pattern [16].

Phase #2: Syntax Analysis

This phase, also known as parsing, uses the list of tokens that was produced by the lexer and arranges them into a syntax tree or parse tree reflecting the structure of the program [14]. Parsers make use of context free grammar (CFG), which can be recognized by push down automata, to describe the syntax of programming languages [13]. As a result, parsers are able to verify if a specific program satisfies all the set of rules established by the CFG. If the program satisfies all the rules, then the parser will proceed to create the parse tree for that program which can be described as a graphical depiction of a grammar derivation.

The CFG, otherwise referred to as grammar (G), consists of one or more variables that represent classes of strings (i.e. languages) [17] - [18]. A grammar can be specified by using four principal components as follows:

$G = (V, \Sigma, P, S)$ where,

- V refers to the finite set of variables or non-terminals where each variable denotes a set of strings. Non-terminals help define the language generated by the CFG.
- Σ refers to the set of tokens known as terminal symbols which are the symbols from which strings are formed which include: lower-case letters, operator symbols, punctuation symbols, digits, and boldface strings like id or if.
- P refers to the set of production rules which specify the manner in which non-terminals and terminals can be combined to form strings. Thus, representing a recursive definition of the language.
- S refers to the start symbol that represents the language being defined.

Example 1.2

An example for the use of a CFG can be offered by showing the grammar for expressions in a typical programming language with the following specifications [18]:

- Operators include addition (+) and multiplication (*)
- Identifiers must begin with a or b, which may be followed by any string in $\{a, b, 0, 1\}*$
- Two variables are used (E and I)
- E will be used to represent expressions and functions as the start symbol
- I will be used to represent identifiers and its language is the language of the following regular expression: $(a + b)(a + b + 0 + 1) *$

Then, taking into consideration the previously mentioned specifications, the grammar can be defined as follows:

G = (V, Σ, P, S) where,

- V, the set of variables, is equal to {E, I}.
- Σ, the set of tokens, is equal to { +, ∗, (,), a, b, 0, 1}.
- S, the start symbol, is equal to E.
- Finally, P represents the set of production rules shown in Table 2

Table 2: Set of production rules for Example 1.2

Rule number	Production rule
1	E → I
2	E → E + E
3	E → E ∗ E
4	E → (E)
5	I → a
6	I → b
7	I → Ia
8	I → Ib
9	I → I0
10	I → I1

On the other hand, the grammar derivation can be described as the sequence of grammar rules that transforms the start symbol (S) into the string. By executing a derivation, the parser proves that the processed string belongs to the grammar's language. There are two types of manners in which a derivation may be performed: left-most (input is scanned and replaced from left to right), and right-most (input is scanned from right to left).

Example 1.3
To obtain a better understanding of grammar derivation and its relation to the parse tree, consider the case in which the input string is "id + id * id" and the set of production rules is as shown in Table 3 [17]:

Table 3: Set of production rules for Example 1.3

Rule number	Production rule
1	E → E + E
2	E → E ∗ E
3	E → id

In this case, the left-most derivation may be executed as shown in Table 4.

Table 4: Left-most derivation for Example 1.3

Derivation number	Production rule
1	E → E * E
2	E → E + E * E
3	E → id + E * E
4	E → id + id * E

5	E → id + id * id

Inversely, the right-most derivation may be executed as shown in Table 5.

Table 5: Right-most derivation for Example 1.3

Derivation number	Production rule
1	E → E + E
2	E → E + E * E
3	E → E + E * id
4	E → E + id * id
5	E → id + id * id

After executing the derivation, the parse tree for the given string may be devised by creating a graphical representation of the executed derivation. This graph may be created in five steps with any of the two types of derivations previously shown. If the left-most derivation is used then, the steps to create the parse tree may be described as shown in Figure 3.

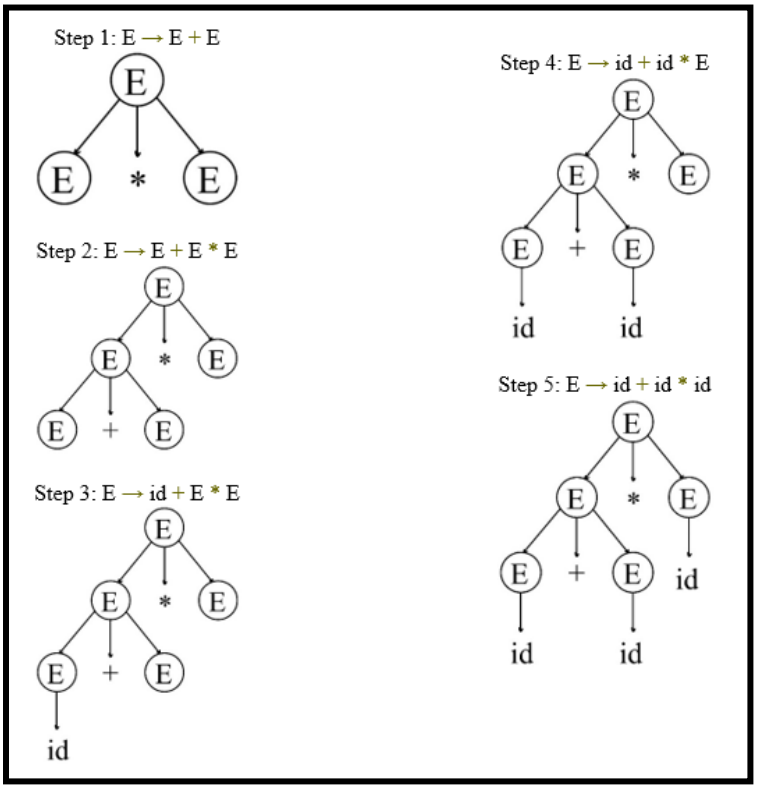

Figure 3: Creating parse tree for Example 1.3

As it may be observed through the previously executed steps towards creating the parse tree, all the leaf nodes in a parse tree are terminals (i.e. operators) and all the interior nodes are non-terminals (i.e. variables).

Phase #3: Semantic Analysis
This phase focuses on verifying the semantic consistency of the program. Depending on whether this consistency is verified at compile time or runtime, the type of semantic analysis can be classified as either static or dynamic. To ensure the sematic consistency of a program, the semantic analyzer gathers type information from the program and stores this information on either the parse tree or the symbol table. The stored type information is then used by the compiler during the intermediate code generation phase. In general, the sematic analyzer's functionality may be described through three main functions which are control flow checking, label checking, and type checking, being the latter the most important one among the three.

Type checking verifies that each operation executed in the program respects the type system of the language [19]. A type system is a collection of rules that assigns types to program constructs. Thus, these systems specify which operations are valid for each type. Some of the violations in a language's type system that may be found during this phase are the following: use of a variable that is not declared, use of a boolean value as a function pointer, etc. [14]. Furthermore, the sematic analyzer is able to identify errors by using the set of production rules provided by the context-free grammar and adding semantic rules to them as shown below:

CFG + semantic rules = Syntax Directed Definitions

Example 1.4
Consider the following line of code in C++ and identify if the compiler would find any errors in the code:

float a = "nine";

The previously shown line of code does not have any lexical or syntactical errors. However, the code has a semantical error due to the fact that a value type string is being assigned to a variable type float.

On the other hand, the semantic analysis makes use of a special form of context-free grammar to be able to append the attributes, such as type information, of the non-terminals (variables) in the parse tree as previously mentioned. This type of context-free grammar is known as attribute grammar. In attribute grammar, two types of attributes may be used: synthesized attributes and inherited attributes. Synthesized attributes are the attributes of a non-terminal whose value is synthesized from the non-terminal's child nodes in the parse tree. On the other hand, the inherited attributes are the attributes of a non-terminal whose value can be obtained from a parent node or a sibling in the parse tree.

is computed when the non-terminals on the left-hand side of the production rule. Inversely, inherited attributes are those whose value is computed when the non-terminal is on the right-hand side of the production rule. On the side note, these attributes always each of these implemented attributes have a well-defined domain of values such as integer, double, character, and expressions.

Example 1.5
To offer an example on how attribute grammar would append attributes to the non-terminals in a parse tree, consider the following production rule:

Term → Term * Factor

If the previous production rule in context-free grammar is modified to attribute grammar, the following is obtained:

Term → Term * Factor { Term.value = Term.value * Factor.value }

Phase #4: Intermediate Code Generation
This phase produces machine independent code from the semantic representation of the source program [13] – [14]. This phase is executed to reduce the number of optimizers needed for compilation. Note that if machine code is generated directly from the source code for n target machine, n optimizers and n code generators will be needed. However, if a machine independent code is generated, then only one optimizer is needed. For this reason, the code that is generated during this phase is abstract and may be created through different techniques such as postfix notation, three-address code, and syntax trees [20]. Each technique is briefly described below:

Postfix notation
Postfix notation is the notation in which operators are written before their operands. For example, consider the following expression shown in ordinary (infix) notation:

(a + b) * (c – d) / (a + d)

The postfix notation for the same expression would be the following:

ab+ cd- * ad+ /

Three-address code
Three-address code is generated by producing a sequence of three-address statements. These types of statements can contain no more than three addresses and one operator to represent an expression. The general form for a three-address statement is as follows:

a = b op c where,

- a, b, and c refer to operands such as names, constants, or temporary variables generated by the compiler.
- op refers to the operator.

Example 1.5
To provide an example of how an expression would be converted to three address code, consider the following expression:

a = b + c * 17

The previously shown expression would be written in the following form by using the three-address code method and creating three temporary variables:

- t1 = int_to_float(17)
- t2 = rate * t1
- t3 = b + t2
- a = t3

Example 1.5
Another example of three-address code translation may be provided by considering the following piece of code:

```
for(i = 1; i<=10; i++)
 {
  a[i] = b * 2;
 }
```

Figure 4: Simple code in C++ for Example 1.5

The previously shown script would be written in the following form by using the three-address code method and creating three temporary variables:

- i = 1
- L: t1 = b * 2
  - t2 = &a
  - t3 = size_of(int)
  - t4 = t3 * i
  - t5 = t2 + t4
  - t5 = t1
  - i = i + 1
- if i <= 10 go to L

Syntax tree
Finally, the last type of intermediate code representation to be discussed is the syntax tree, which is similar to the parse tree but in a condensed form. In the syntax tree, the leaf nodes are operands and the interior nodes are operators.

Example 1.6
Consider the following expression and generate its syntax tree.

a = (b + (c / d)) * (b – (d / c))

If the previously shown expression is processed during the intermediate code generation phase of the compiler, the syntax tree shown in Figure 5 would be generated.

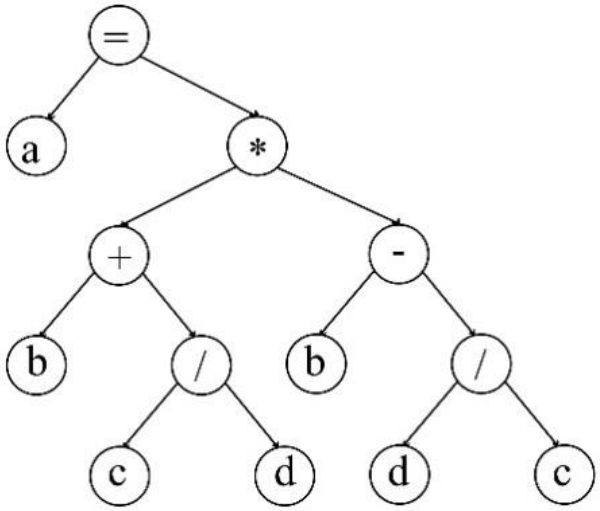

Figure 5: Syntax tree for Example 1.6

Phase #5: Code Optimization
This phase executes the translation of the symbolic variable names in the abstract code to numbers corresponding to registers in the target machine code [14]. During this phase, unnecessary lines of code may be removed and the sequence of statements may be re-arranged to decrease the program's execution time and the space it occupies.

There are two types of code optimization: machine independent optimization and machine dependent optimization. In machine independent optimization, the code generated in the previous phase (intermediate code) is optimized. This type of optimization is not concerned with CPU registers or absolute memory locations. On the other hand, in machine dependent optimization, the generated target code is optimized for the target machine architecture. This type of optimization is concerned with the efficient use of CPU registers and absolute memory locations.

Phase #6: Machine Code Generation
This phase receives input from the code optimization phase to translate the abstract code into assembly language code for a specific machine architecture [14]. The main objective of this phase is to allocate registers and memory locations and generate machine code that is relocatable. Thus, the code that is produced during this phase is capable of receiving inputs and generating outputs.

Phase #7: Assembly and Linking
This final phase transforms the assembly language code into a binary representation and determines the addresses of variables, functions, etc. [14]. Although this phase is part of the compilation process, it is not actually part of the compiler itself since it is typically done by programs supplied by the machine or operating system vendor.

It's important to note that phases 1-6 interact with symbol table manager and error handler during the compilation process. As a result, both the symbol table and error handler are updated in each phase of the compilation process. The symbol table is used to record the attributes for each identifier. On the other hand, the error handler is deals with the errors that could prevent the

compilation process from proceeding. Different types of errors for each phase of the compilation process as shown below:

- Lexical error (phase #1): Incorrectly typed identifier
- Syntactical error (phase #2): Missing semicolon or parenthesis
- Semantical error (phase #3): Incompatible value assignment
- Logical error (phase #4-5): Statement is not reachable
- Symbol table error (phase #1-6): Multiple declared identifiers

From another point of view, while phases 1-3 are collectively known as the frontend of the compiler, phases 5-7 are known as the backend of the compiler [14]. Then, phase 4 is the middle part of the compiler which often includes optimizations and transformations on the intermediate code. From another point of view, an alternative to the compiler is the interpreter. Both compilers and interpreters are capable of executing the same function of translating high-level programming language into machine code. However, the main difference between the two is that while compilers execute this translation fully before the program runs, interpreters execute the translation of individual lines of codes written in a high-level programming language while the program is running.

(1.3) Practical Programming with C++ and Python

This section will go into the topic of practical programming by discussing basic programming concepts such as variables, control flow, functions, and classes and objects, using two high-level object-oriented programming languages: C++ and Python. Both C++ and Python will be taught in parallel to offer a contrast on how syntax may vary between programming languages.

(1.3.1) Programming to solve problems

Before proceeding to the discussion of basic programming concepts such as the proper syntax for variables and functions, it is important to understand the reason people write programs and phases through which one must go through to ensure that programs are properly written. Programs are written for the sole purpose of solving problems. For this reason, programmer's must always go through the following three phases when developing a program:

1. Problem analysis: Cleary define the problem and identify the input and output data that will be needed to solve the problem. Additionally, the operations to be executed with the available data must be identified. These operations may be expressed through pseudocode or a flow diagram.
2. Write program: Taking into consideration the data to be expected as input and the data to be expected as output, identify the program variables. Then, implement the operations to be executed on the data using a specific programming language syntax.
3. Validate program: Ensure that the program correctly solves the defined problem by validating each line of code and comparing the output data with the expected result.

Example 1.7

To obtain a better understanding on how the three phases to developing a program are applied, consider the problem in which the area of a circle needs to be calculated. Each phase would be executed as follows:

(1) Problem analysis

a. Expected inputs and outputs: To calculate the area of a circle (output), the circle's radius (input) must be known.

b. Operations: Once the radius of the circle has been obtained, the its area may be calculated by calculating the square of the circle and multiplying it by the constant pi.

(2) Write program

Taking into consideration the previous analysis, Figure 6 and Figure 7 show the written program in C++ and Python, respectively.

```cpp
#include <iostream>
#include <math.h>
using namespace std;

//Declaring variables according to expected inputs and outputs
float r = 0.0;
float area = 0.0;

//Requesting circle radius
std::cout << "Enter the circle's radius in ft: ";
std::cin >> r;

//Calculating and outputting circle's area
area = pow(r, 2)*M_PI;
std::cout << endl << "Area of the circle in square ft: " << area;
```

```
Enter the circle's radius in ft:
 4.5

Area of the circle in square ft: 63.6173
```

Figure 6: Finding the area of a circle in C++

```python
from math import pi

#Requesting circle radius
r = float(input("Enter the circle's radius in ft: "))

#Calculating and outputting circle's area
area = pi*r**2
print("\nArea of the circle in square ft: ", area)
```

```
Enter the circle's radius in ft:  4.5

Area of the circle in square ft:  63.61725123519331
```

Figure 7: Finding the area of a circle in Python

(3) Validate program

If the area of the circle is manually calculated for the same radius as in previously executed pieces of code, the following result is obtained:

$$Area_{cir} = r^2 * \pi = 4.5^2 * \pi = 63.6173$$

As it may be observed, the calculated result is equal to the results obtained by executed the written scripts. Thus, the program has been validated. Note that validation is generally not this simple and sometimes a program may seem to function correctly after one execution but behave differently after another execution. For this reason, it is important to always validate program inputs and verify that each line of code has been written correctly.

(1.3.2) Program outputs – Hello World!

The Hello World! program is usually the first program that is learned when studying a programming language. This tradition of using the phrase "Hello World!" originates from the 1978 book called The C Programming Language by Brian Kernighan where one of the example programs printing the words "hello world" [21]. The simple Hello World! Program may be written in C++ and Python as shown in Figure 8 and Figure 9, respectively.

```
#include <iostream>
using namespace std;

cout << "Hello World!";
```

```
Hello World!
```

Figure 8: Hello World! program in C++

```
print("Hello World!")
```

```
Hello World!
```

Figure 9: Hello World! program in Python

While C++ uses **cout** to produce outputs, Python uses the **print( )** function to produce outputs. Additionally, C++ requires that the **<iostream>** library be included in the program to be able to execute input and output operations. On the other hand, when writing outputs it is useful to understand how characters that are preceded by \ are interpreted. These are special types of characters that may be used to format the output or to include specific special characters in the output as shown in Table 6.

Table 6: Special characters preceded by \ and their output interpretations

Special character	Output interpretation
\n	ENTER
\t	TAB
\\	\
\"	"
\'	'

The implementations of these special characters in C++ and in Python are shown in Figure 10 and Figure 11, respectively.

```cpp
#include <iostream>
using namespace std;

cout << "Implementing special characters in C++:" << endl;
cout << "Outputting tab\tOutputting enter\nOutputting backslash "
     << '\\' << "\nOutputting double quotation mark \" "
     << "Outputting single quotation mark \'";
```

```
Implementing special characters in C++:
Outputting tab  Outputting enter
Outputting backslash \
Outputting double quotation mark " Outputting single quotation mark '
```

Figure 10: Implementing special characters in program output in C++

```python
print("Implementing special characters in Python:")
print("Outputting tab\tOutputting enter\nOutputting backslash", end=' '); print('\\')
print("Outputting double quotation mark \"",
      "Outputting single quotation mark \'")
```

```
Implementing special characters in Python:
Outputting tab  Outputting enter
Outputting backslash \
Outputting double quotation mark " Outputting single quotation mark '
```

Figure 11: Implementing special characters in program output in Python

After observing Figure 10 and Figure 11, the following notes may be made concerning outputs in C++ and outputs in Python. In C++, the semicolon must always be used at the end of a **cout** statement. Also, a **cout** may be continued in the next line of code by using <<. Finally, in C++, the cursor always stays in line after executing an output statement unless the endl character is used. On the other hand, in Python to cursor always moves to the next line after executing an output statement unless end=‘ ’ is used. Also, the use of the semicolon is only necessary when the end of the statement is followed by a new statement. Furthermore, in Python, print statements may be continued in the next line of code by using commas.

(1.3.3) Program Variables

A variable can be seen as a space in memory whose content may change throughout the execution of a program. Thus, when variables are declared in a program a space in memory is being reserved for the program. In general, variables are declared for each datum in which the program must operate. Depending on the programming language, variables may be declared differently. For example, while C++ requires for the variable’s datatype to be specified in the declaration, Python does not. Instead, Python assumes the variable’s datatype according to the variable’s value. C++ has six main datatypes: bool, char, int, float, double, and void as shown in Table 7. On the other hand, Python has five main datatypes: numbers, string, list, tuple, and dictionary as shown in Table 8. Also, when executing a comparison between C++ and Python datatypes it’s important to note that the typical bit width of C++ datatypes is very much lower than that of Python datatypes. For this reason, the use of C++ tends to be preferred over Python whenever programs require speed and efficiency.

Table 7: C++ datatypes

Datatype	Description	Example declaration
bool	Used to store true or false values	bool sick = “true”;
char	Used to store a single character	char grade = ‘A’;

int	Used to store integer values	int age = 27;
float	Used to store single-precision floating point values	float height = 5.5;
double	Used to store double-precision floating point values	double NetWorth = 105347.89;
void	Used to represent the absence of type	void *vpointer;

Table 8: Python datatypes

Datatype	Description	Example declaration
numbers	Used to store numeric values such as int, long, float, and complex	grade = 98
string	Used to store a set of characters	name = “Mario”
list	Used to store items of different or the same type by separating them with commas and enclosing them with brackets [ ]	heights = [4.9, 6, 5.3, 5.8]
tuple	Used to store items of different or the same type by separating them with commas and enclosing them with parentheses ( ). Can be thought of as read-only lists.	heights = (4.9, 6, 5.3, 5.8)
dictionary	Used to store data that consists of key-value pairs. Functions as a kind of hash Tabletype.	family = {“mother” : “Maria”, “father” : “Daniel”, “son” : “Eric”, “daughter” : “Diana”}

Note that variables must always be declared with unique names. Variable names may be written with any combination of letters, digits, and underscore (_) as long as the name starts with a letter. Otherwise, the variable name is incorrect. On the side note, it is recommended to name variables according to the information they represent and write them to be as short as possible.

(1.3.4) Program inputs

Whenever a program requires user input, an input statement must be declared. The syntax for an input statement varies according to the programming language. In C++, **cin** is used to receive inputs. In Python, the **input( )** function is used to receive inputs. The behavior of the input statement when the program reaches its execution at runtime is generally as follows:

- Program is halted and waits for input data.
- After user enters the data by pressing ENTER, program reads value and stores it in a specified variable.

Figure 12 and Figure 13 offer an example of an input statement in C++ and Python, respectively. Additionally, the figures show how a program remains halted when expecting inputs.

```cpp
#include <iostream>
using namespace std;

int age = 0;

cout << "Enter your age: ";
cin >> age;
cout << endl << "Age received!";
```

```
Enter your age:
```

Figure 12: Waiting for user input in C++ program

```python
age = input("Enter your age: ")
print("\nAge received!")
```

```
Enter your age:
```

Figure 13: Waiting for user input in Python program

(1.3.4) Operators

Operators are an important element in programming languages. They are used whenever mathematical or logical operations need to be executed with program data. C++ has the following operators: arithmetic, relational, logical, bitwise, and assignment. On the other hand, Python has the following operators: arithmetic, relational, logical, bitwise, assignment, membership, and identity.

Arithmetic operators are used to execute arithmetic operations such as division, summation, multiplication, etc. Relation operators are used to evaluate a relation between two operands and output a boolean value according to whether the relation is true or false. Logical operators are used to simulate the main operators in mathematical logic which are AND, OR, and NOT. Bitwise operators are used to perform bit-by-bit operations. Assignment operators are used to assign values to variables or modify a variable's value. Membership operators, which are specific to Python, are used to verify if a variable is in a specific sequence. Finally, identity operators, which are specific to Python as well, are used to verify if two objects point to the same memory location.

Note that Table 9 shows the operators that are common to both C++ and Python. Then, Table 10 shows the operators specific to C++ and Table 11 shows the operators specific to Python.

Table 9: Operators found in both Python and C++

Operator	Description	Example
Arithmetic operators Operands for examples: a = 15, b = 10		
+	Used to add two operands.	Operation: a + b Result: 25

-	Used to subtract the right hand operand from the left hand operand.	Operation: a - b Result: 5
*	Used to multiply two operands.	Operation: a * b Result: 150
/	Used to divide the left hand operand by the right hand operand.	Operation: a / b Result: 1.5
%	Used to divide the left hand operand by the right hand operand and return the remainder.	Operation: a % b Result: 5
Relational operators **Operands for examples: a = 15, b = 10**		
==	Used to evaluate if two operands are equal. The possible results are: True (operands are equal), or false (operands are not equal).	Operation: a == b Result: false
!=	Used to evaluate if two operands are not equal. The possible results are: True (operands are not equal), or false (operands are equal).	Operation: a != b Result: true
>	Used to evaluate if the right hand operand is greater in value than the left hand operand. The possible results are: True (right hand operand is greater in value than the left hand operand), or false (right hand operand is not greater in value than the left hand operand).	Operation: a > b Result: true
<	Used to evaluate if the right hand operand is smaller in value than the left hand operand. The possible results are: True (right hand operand is smaller in value than the left hand operand), or false (right hand operand is not smaller in value than the left hand operand).	Operation: a < b Result: false
>=	Used to evaluate if the right hand operand is greater or	Operation: a >= b Result: true

	equal in value than the left hand operand. The possible results are: True (right hand operand is greater or equal in value than the left hand operand), or false (right hand operand is smaller in value than the left hand operand).	
<=	Used to evaluate if the right hand operand is smaller or equal in value than the left hand operand. The possible results are: True (right hand operand is smaller or equal in value than the left hand operand), or false (right hand operand is greater in value than the left hand operand).	Operation: a <= b Result: false
Logical operators Operands for examples: a = "true", b = "false"		
&& (C++), and (Python)	Used to implement the logical AND gate. The possible results are true (both operands are true), or false (either both or one of the operands is false).	Operation: a && b Result: false
|| (C++), or (Python)	Used to implement the logical OR gate. The possible results are true (either both or one of the operand is true), or false (both operands are false).	Operation: a || b Result: true
! (C++), not (Python)	Used to implement the logical NOT gate. The possible results are true (operand is false), or false (operand is true).	Operation: !a Result: false
Bitwise operators Operands for examples: a = 011010, b = 110111		
&	Used to implement the binary AND. Copies a bit with value 1 to the result if there is a bit with value 1 in both operands.	Operation: a & b Result: 010010
|	Used to implement the binary OR. Copies a bit with value 1 to the result if there is a bit with value 1 in either operand.	Operation: a | b Result: 111111

^	Used to implement the binary XOR. Copies a bit with value 1 to the result if there is a bit with value 1 on either operand but not on both.	Operation: a ^ b Result: 101101
~	Used to implement the binary ones complement. Result shows the effect of flipping bits.	Operation: ~ a Result: 100101
<<	Used to implement the binary left shift. The left handed operand's value is shifted to the left by the number of bits specified by the right handed operand.	Operation: a << 1 Result: 110100
>>	Used to implement the binary right shift. The left handed operand's value is shifted to the right by the number of bits specified by the right handed operand.	Operation: a >> 2 Result: 100110
Assignment operators **Operands for examples: a = 15, b = 10**		
=	Used to assign value from the right handed operand to the left handed operand.	Operation: c = a Result: c = 15
+=	Used to add the right handed operand to the left handed operand and assign the result to the left handed operand.	Operation: a += b Result: a = 25
-=	Used to subtract the right handed operand from the left handed operand and assign the result to the left handed operand.	Operation: a -= b Result: a = 5
*=	Used to multiply the right handed operand with the left handed operand and assign the result to the left handed operand.	Operation: a *= b Result: a = 150
/=	Used to divide the left handed operand by the right handed operand and assign the result to the left handed operand.	Operation: a /= b Result: a = 1.5
%=	Used to take the modulus of the left handed operand with	Operation: a %= b Result: a = 5

	the right handed operand and assign the result to the left handed operand.	

Table 10: C++ specific operators

Operator	Description	Example
Arithmetic operators Operands for examples: a = 15, b = 10		
++	Used to increment a type int operand by one.	Operation: a++ Result: 16
--	Used to decrement a type int operand by one.	Operation: a-- Result: 14
Assignment operators Operands for examples: a = 15, b = 10		
<<=	Used to implement the binary left shift on the left handed operand and store the result on the left handed operand.	Operation: a << 1 Result: a = 110100
>>=	Used to implement the binary right shift on the left handed operand and store the result on the left handed operand.	Operation: a >> 2 Result: a = 100110
&=	Used to implement the binary AND between two operands and store the result on the left handed operand.	Operation: a & b Result: a = 010010
|=	Used to implement the binary OR between two operands and store the result on the left handed operand.	Operation: a | b Result: a = 111111
^=	Used to implement the binary XOR between two operands and store the result on the left handed operand.	Operation: a ^ b Result: a = 101101

Table 11: Python specific operators

Operator	Description	Example
Arithmetic operators Operands for examples: a = 15, b = 10		
**	Used to perform exponential calculation. The left handed operand is raised to the power of the right handed operand.	Operation: a ** 2 Result: 225
//	Used to perform floor division. The result is the integral part of the quotient	Operation: a // b Result: 1

	resulting from the division of the left handed operator by the right handed operator. Note that if one of the operands is negative the result is rounded towards negative infinity.	
Relational operators Operands for examples: a = 15, b = 10		
< >	Similar to != operator. Used to evaluate if two operands are not equal. The possible results are: True (operands are not equal), or false (operands are equal).	Operation: a < > b Result: true
Assignment operators Operands for examples: a = 15, b = 10		
**=	Used to perform exponential calculation. The left handed operand is raised to the power of the right handed operand. Then, the result is assigned to the left handed operand.	Operation: a **= 2 Result: a = 225
//=	Used to perform floor division. The left handed operand is raised to the power of the right handed operand. Then, the result is assigned to the left handed operand.	Operation: a //= b Result: a = 1
Membership operators Operands for examples: a = 15, b = [a, c, d]		
in	Used to verify if a variable is in a specified sequence. The possible results are 1 (variable was found in sequence), or 0 (variable was not found in sequence).	Operation: a in b Result: 1
not in	Used to verify if a variable is not in a specified sequence. The possible results are 1 (variable was not found in sequence), or 0 (variable was found in sequence).	Operation: a not in b Result: 0
Identity operators		
is	Used to verify if variables on either side of the operator point to the same object.	a in b, true if id(a) = id(y)

is not	Used to verify if variables on either side of the operator point to different objects.	a not in b, true if id(a) ≠ id(b)

(1.3.5) Control Flow

Instructions in computer programs may be divided into three types: sequential, conditional, and iterative. Sequential instructions are simple instructions that proceed to the next instruction in line once they have finished executing. Conditional instructions allow programs to take one out of two possible execution paths depending on the result of an evaluated condition. Finally, iterative instructions allow programs to execute a set of instructions until a specific condition is fulfilled.

Among the three previously mentioned types of instructions, the last two are those which facilitate control flow in a program. Control flow refers to the act of establishing a condition-based order for the execution of instructions in a program. Furthermore, control statements refer to the statements which facilitate control flow in a program. The different control statements available in C++ and Python are shown in Figure 14. All of the control flow statements shown in Figure 14 are individually discussed below except the goto statement due to how it usually adds unnecessary complexity to programs.

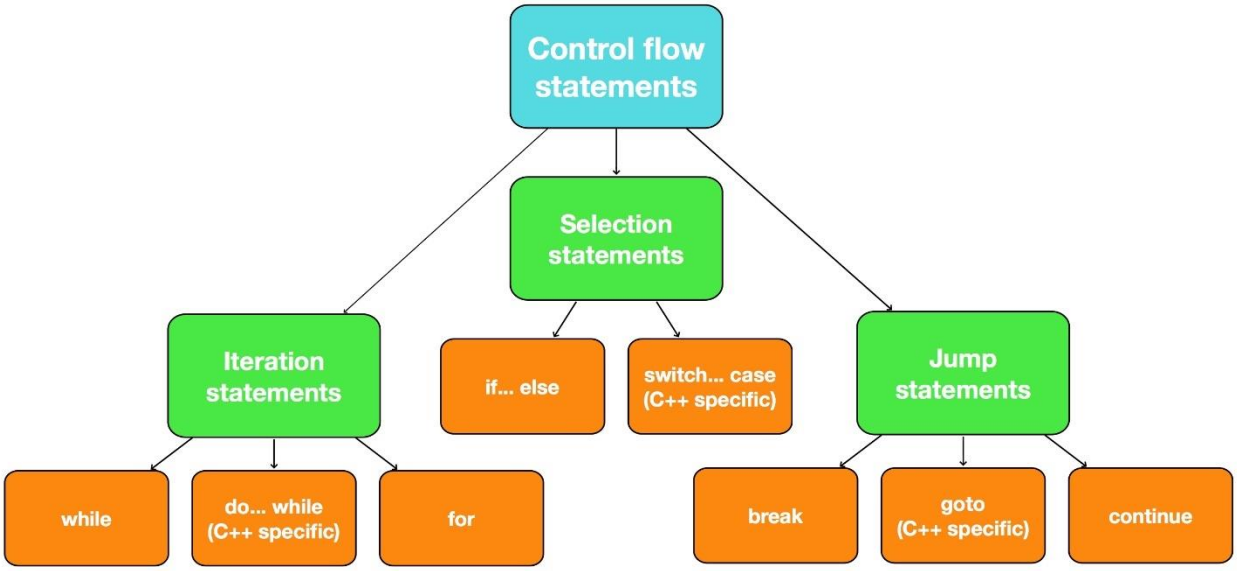

Figure 14: Control flow statements

while statement

The while statement or while loop is used to execute the code in the loop body while the result of a boolean expression remains true. The flow diagram for this statement is shown in Figure 15.

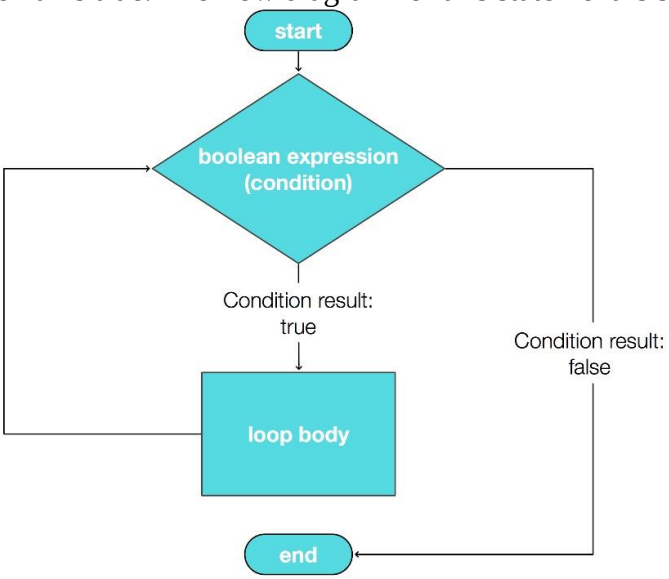

Figure 15: Flow diagram for the while statement

Both C++ and Python incorporate the while statement. However, there is a slight variation in syntax between the two languages. Figure 16 and Figure 17 show this variation by offering an example implementation of the while statement in C++ and in Python, respectively.

```cpp
#include <iostream>
using namespace std;

int i = 0;
std::cout<<"Entering loop...\n";
while(i<6)
{
    std::cout<<i<<'\n';
    i+=1;
}
std::cout<<"Left loop";
```

```
Entering loop...
0
1
2
3
4
5
Left loop
```

Figure 16: Implementation of the while statement in C++

```python
i = 0
print("Entering loop...")
while i < 6:
    print(i)
    i+=1
print("Left loop")
```

```
Entering loop...
0
1
2
3
4
5
Left loop
```

Figure 17: Implementation of the while statement in Python

do… while statement

The do… while statement or do… while loop is used to execute the code in the loop body once and then iteratively execute the code in the loop body until the result from the evaluated condition is false. The flow diagram for this statement is shown in Figure 18.

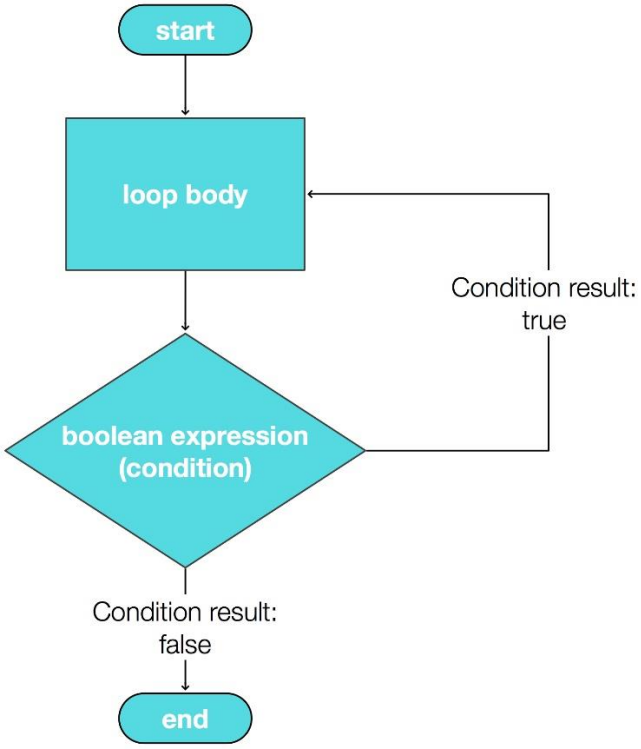

Figure 18: Flow diagram for the do… while statement

The do… while statement is a C++ specific statement. An example of its implementation is shown in Figure 19. In the example below, the do… while statement is being used for user input validation.

```
#include <iostream>
using namespace std;

int i=0;
do{
    std::cout<<"Enter a positive integer value: ";
    std::cin>>i;
}while(i<0);
```

```
Enter a positive integer value:
 -1
Enter a positive integer value:
 -3
Enter a positive integer value:
 2
```

Figure 19: Implementation of the do… while statement in C++

for statement

The for statement or for loop is uses a variable as an iterator. For each iteration, the for statement first evaluates its boolean expression. Then, if the result from the evaluated condition is true, the code in the loop body is executed and the iterator is updated. The flow diagram for this statement is shown in Figure 20.

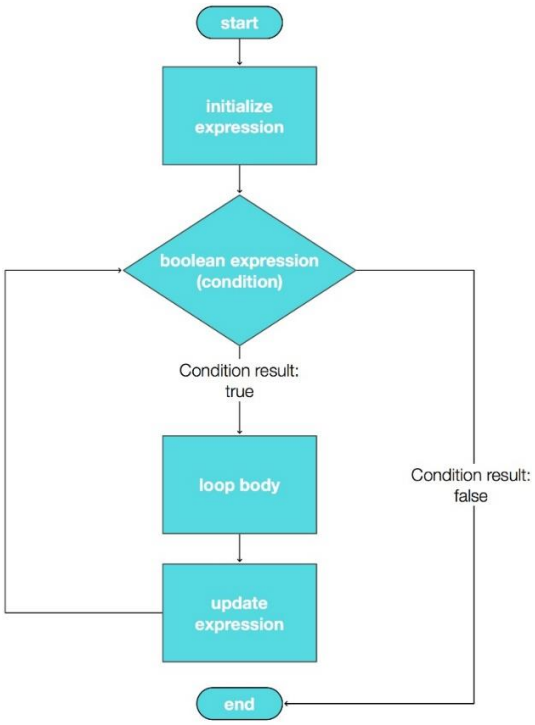

Figure 20: Flow diagram for the for statement

Both C++ and Python incorporate the for statement. However, there is a slight variation in syntax between the two languages. Figure 21 and Figure 22 show this variation by offering an example implementation of the while statement in C++ and in Python, respectively.

```
#include <iostream>
using namespace std;

for(int i = 0; i < 6; i++)
{
    cout << i << '\n';
}
```

```
0
1
2
3
4
5
```

Figure 21: Implementation of the for statement in C++

```
for i in range(0, 6, 1):
    print(i)
```

```
0
1
2
3
4
5
```

Figure 22: Implementation of the for statement in Python

Note that when implementing the for statement in Python, the range( ) function is used to establish the amount of times the function will iterate over the loop body. The initial value that was given to the iterator was 0. Additionally, the iterator was set to increment by 1 after each iteration until its value reached 6.

if… else statement

The if… else statement is used whenever there are two possible execution paths in which the program could proceed. Then, the execution path to be taken depends on the result from an evaluated condition. If the result is true then, the code in the body of the if will be executed. If the result is false then, the code in the body of the else is executed. The flow diagram for this statement is shown in Figure 23.

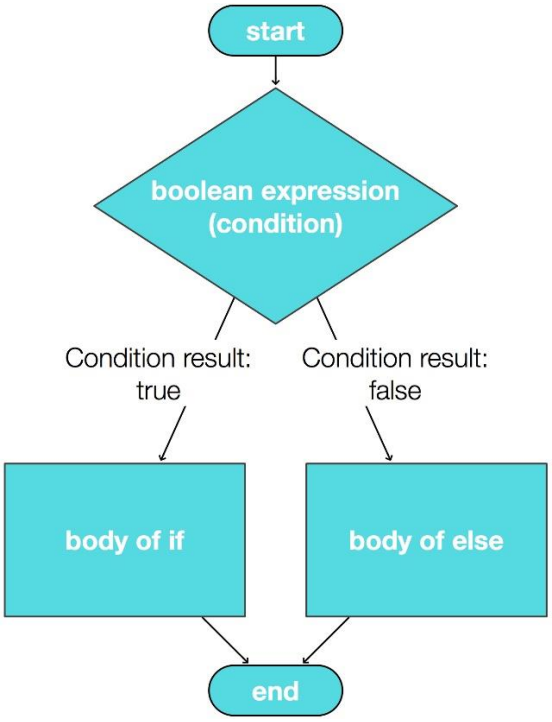

Figure 23: Flow diagram for the if… else statement

Both C++ and Python incorporate the if… else statement. However, there is a slight variation in syntax between the two languages. Figure 24 and Figure 25 show this variation by offering an example implementation of the while statement in C++ and in Python, respectively.

```cpp
#include <iostream>
using namespace std;

int grade = 80;

if(grade > 89)
{
    std::cout << "You got an A!";
}else
    if(grade > 79)
    {
        std::cout << "You got an B!";
    }else
        if(grade > 69)
        {
            std::cout << "You got an C!";
        }else
            if(grade > 59)
            {
                std::cout << "You got an D!";
            }else
                {
                    std::cout << "You got an F!";
                }
```

```
You got an B!
```

Figure 24: Implementation of the if… else statement in C++

```python
grade = 80
if grade > 89:
    print("You got an A!")
else:
    if grade > 79:
        print("You got an B!")
    else:
        if grade > 69:
            print("You got an C!")
        else:
            if grade > 59:
                print("You got an D!")
            else:
                print("You got an F!")
```

```
You got an B!
```

Figure 25: Implementation of the if… else statement in Python

switch… case statement

The switch… case statement is used to select one out of many code blocks to be executed. Each code block is represented by a case. If a case is equal to the constant expression read by the switch… case then, the code block represented by that case will be executed. The flow diagram for this statement is shown in Figure 26. Additionally, an example implementation of the switch… case statement is shown in Figure 27. Note that the switch… case statement is a C++ specific statement.

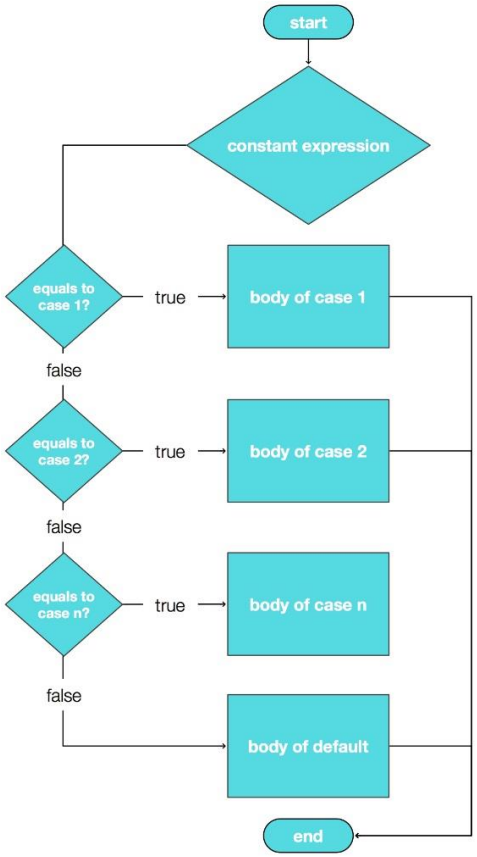

Figure 26: Flow diagram for the switch… case statement

```cpp
#include <iostream>
using namespace std;

int season = 2;
switch(season){
    case 1:
        std::cout<<"We are in Spring!";
        break;
    case 2:
        std::cout<<"We are in Summer!";
        break;
    case 3:
        std::cout<<"We are in Fall!";
        break;
    case 4:
        std::cout<<"We are in Winter!";
        break;
}
```

```
We are in Summer!
```

Figure 27: Implementation of the switch… case statement in C++

break statement

The break statement is used to leave a loop without fulfilling the loop's end condition. The flow diagram for this statement is shown in Figure 28.

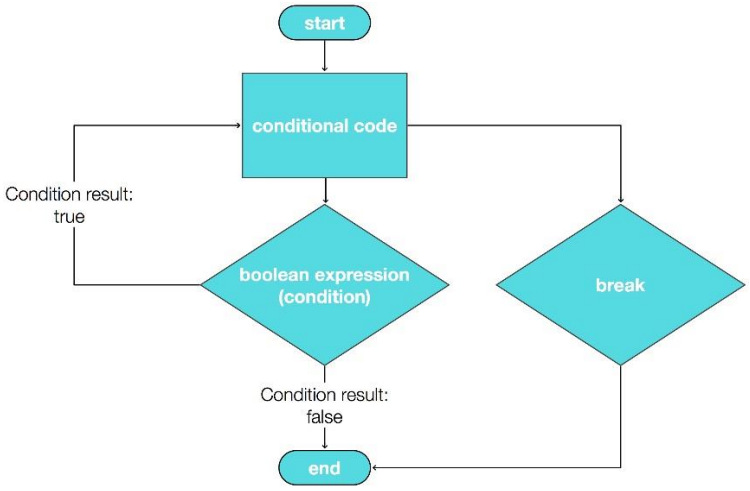

Figure 28: Flow diagram for the break statement

Both C++ and Python incorporate the break statement. Figure 29 and Figure 30 offer an example implementation of the break statement in C++ and in Python, respectively.

```cpp
#include <iostream>
using namespace std;

std::cout<<"Entering loop...\n";
for(int i = 0; i<6; i++)
{
    std::cout<<i<<'\n';
    if(i==3)
    {
        std::cout<<"Left loop using break";
        break;
    }
}
```

```
Entering loop...
0
1
2
3
Left loop using break
```

Figure 29: Implementation of the break statement in C++

```python
i = 0
print("Entering loop...")
while i < 6:
    print(i)
    if(i==3):
        print("Left loop using break")
        break;
    i+=1
```

```
Entering loop...
0
1
2
3
Left loop using break
```

Figure 30: Implementation of the break statement in Python

continue statement

The continue statement is used to skip the remaining code in an iteration and jump to the start of the next iteration. The flow diagram for this statement is shown in Figure 31.

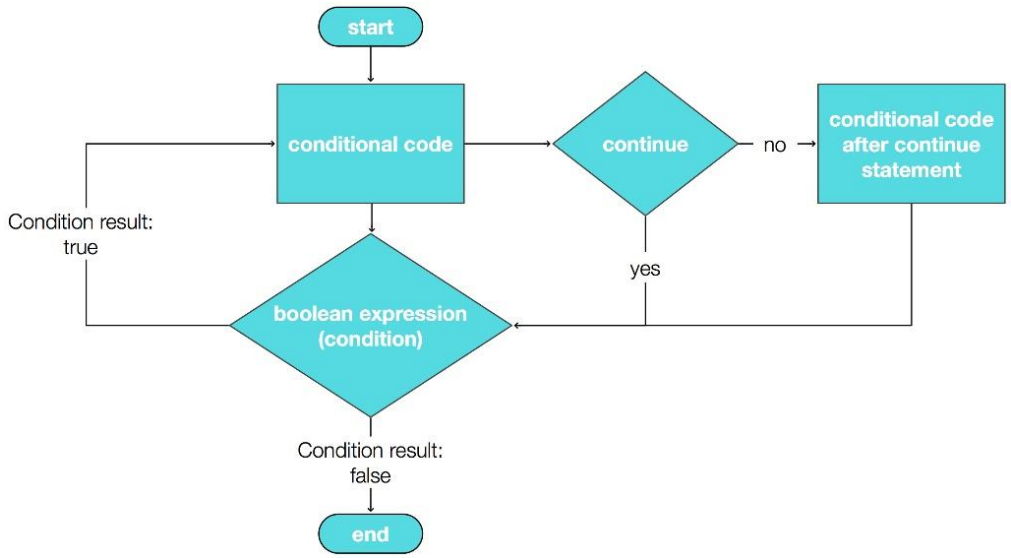

Figure 31: Flow diagram for the continue statement

Both C++ and Python incorporate the continue statement. Figure 32 and Figure 33 offer an example implementation of the continue statement in C++ and in Python, respectively.

```cpp
#include <iostream>
using namespace std;

std::cout<<"Entering loop...\n";
for(int i = 0; i<6; i++)
{
    if(i==3)
    {
        std::cout<<"Skipped 3 using continue\n";
        continue;
    }
    std::cout<<i<<'\n';
}
```

```
Entering loop...
0
1
2
Skipped 3 using continue
4
5
```

Figure 32: Implementation of the continue statement in C++

```python
i = 0
print("Entering loop...")
while i < 6:
    i+=1
    if(i==3):
        print("Skipped 3 using continue")
        continue;
    print(i)
```

```
Entering loop...
1
2
Skipped 3 using continue
4
5
6
```

Figure 33: Implementation of the continue statement in Python

(1.3.6) Arrays

The array is a particularly important data structure in programming due to the fact that the majority of the other types of data structures implement arrays in their algorithms. The array functions as a container for items of the same type, where each item or element has a specific location (index) in the array. The number of elements that fit in an array is determined by the array's size. Depending on the programming language, an array can be declared in different ways. For example, Python does not have built-in support for the array. However, Python lists can be used instead to represent arrays. Alternatively, the NumPy library may be imported to include support for arrays in Python. On the other hand, C++ does have built-in support for arrays. The syntax to declare an array in C++ is shown below.

datatype arrayname [n] = $\{element_0, element_1, \dots, element_{n-1}\}$ where, n is the size of the array

Example 1.7

To provide an example on how the previously shown syntax could be implemented, consider the following family: Father – Thomas, Mother – Monica, Son – Richard, and Daughter – Janice. Then, declare an array to store the each of the family member's name in which the indexes 0, 1, 2, and 3 contain the names of the father, mother, daughter, and son, respectively.

string family [4] = $\{"Thomas", "Monica", "Janice", "Richard"\}$

From another point of view, it's important to know the basic set of operations that are supported by the array data structure. These operations are the following: traversing, inserting, deleting, searching, and updating. Each of these operations are individually discussed below.

Traversing operation for arrays

The traversing operation is used to print array elements one-by-one. Depending on the programming language, the traversing operation may be written in different ways and produce the same result when given the same parameters. Figure 34 and Figure 35 offer an example implementation of the traversing operation in C++ and in Python, respectively.

```cpp
#include <iostream>
using namespace std;

int list[]={1,2,3,4,5};
int i=0;
for(i=0;i<5;i++)
{
    std::cout << list[i] << ' ';
}
```

```
1 2 3 4 5
```

Figure 34: Traversing operation in C++

```python
list=[1,2,3,4,5];

for i in list:
    print(i, end=' ')
```

```
1 2 3 4 5
```

Figure 35: Traversing operation in Python

Inserting operation for arrays

The inserting operation is used to insert an element into the array at a specific index. Note that to insert an element into an array, the array must be resized. Then, the array elements must be re-organized in such a way that all element stay inside the array. Figure 36 and Figure 37 offer an example implementation of the inserting operation in C++ and in Python, respectively.

```cpp
#include <iostream>
#include <vector>
using namespace std;

vector<int> list;

/*Inserting elements into dynamic array*/
    list.push_back(1);
    list.push_back(2);
    list.push_back(3);
    list.push_back(4);
    list.push_back(5);

/*Printing vector aka dynamic array before insertion*/
std::cout << "Dynamic array before insertion: ";
int i=0;
for(i=0;i<list.size();i++)
    {
        std::cout << list[i] << ' ';
    }

/*Resizing vector aka dynamic array before insertion*/
list.resize(6);

/*Inserting new element in index 3 without eliminating the original element in index 3 from the array*/
int insert_element = 5;
int insert_index = 3;

int n = list.size()-1;
while(n >= insert_index)
{
    list[n+1] = list[n];
    n-=1;

}
list[insert_index] = insert_element;

/*Printing vector after insertion*/
std::cout << endl;
std::cout << "Dynamic array after insertion: ";
int j=0;
for(j=0;j<list.size();j++)
    {
        std::cout << list[j] << ' ';
    }
```

```
Dynamic array before insertion: 1 2 3 4 5
Dynamic array after insertion: 1 2 3 5 4 5
```

Figure 36: Inserting operation in C++

```python
list=[1,2,3,4,5];

#Array before insertion
print("\nDynamic array before insertion: ", end=' ')
for i in list:
    print(i, end=' ')

#Inserting 5 in index 3
insert_element=5
insert_index=3
list.insert(insert_index, insert_element)

#Array after insertion
print("\nDynamic array after insertion: ", end=' ')
for i in list:
    print(i, end=' ')
```

```
Dynamic array before insertion:  1 2 3 4 5
Dynamic array after insertion:  1 2 3 5 4 5
```

Figure 37: Inserting operation in Python

Deleting operation for arrays

The deleting operation is used to remove an element from the array at a specific index. Note that after deleting an element from an array, all elements in the array must be re-organized. Figure 38 and Figure 39 offer an example implementation of the deleting operation in C++ and in Python, respectively.

```cpp
#include <iostream>
#include <vector>
using namespace std;

vector<int> list;

/*Inserting elements into dynamic array*/
    list.push_back(1);
    list.push_back(2);
    list.push_back(3);
    list.push_back(4);
    list.push_back(5);

/*Printing vector aka dynamic array before deletion*/
std::cout << "Dynamic array before deletion: ";
int i=0;
for(i=0;i<list.size();i++)
    {
        std::cout << list[i] << ' ';
    }

/*Deleting element in index 3 and re-organizing array*/
int delete_index=3;
int temp = 0;
int n = list.size()-1;
while(delete_index < n)
{
    list[delete_index] = list[delete_index+1];
    delete_index+=1;

}

/*Resizing vector aka dynamic array after deletion*/
list.resize(4);

/*Printing vector after insertion*/
std::cout << endl;
std::cout << "Dynamic array after deletion: ";
int j=0;
for(j=0;j<list.size();j++)
    {
        std::cout << list[j] << ' ';
    }
```

```
Dynamic array before deletion: 1 2 3 4 5
Dynamic array after deletion: 1 2 3 5
```

Figure 38: Deleting operation in C++

```python
list=[1,2,3,4,5];

#Array before deletion
print("\nDynamic array before insertion: ", end=' ')
for i in list:
    print(i, end=' ')

#Removing element in index 3
delete_index=3
list.pop(delete_index)

#Array after deletion
print("\nDynamic array after deletion: ", end=' ')
for i in list:
    print(i, end=' ')
```

```
Dynamic array before insertion:  1 2 3 4 5
Dynamic array after deletion:  1 2 3 5
```

Figure 39: Deleting operation in Python

Searching operation

The searching operation is used to search for an element in the array using a specific index or element value. Figure 40 and Figure 41 offer an example implementation of the searching operation in C++ and in Python, respectively. In this example, the searching operation is being used to find the elements in an array containing a specific value.

```cpp
#include <iostream>
using namespace std;

int list[]={1,5,3,4,5,2};
int size = 6;

//Searching for elements containing a specific value
std::cout << "Searching for elements containing the value 5... " << endl << endl;
int search_value = 5;

int j=0;
while(j<size)
{
    if(list[j] == search_value)
    {
        std::cout << "Found the value " << search_value << " at position " << j+1 << endl;
    }
    j+=1;
}
```

```
Searching for elements containing the value 5...

Found the value 5 at position 2
Found the value 5 at position 5
```

Figure 40: Searching operation in C++

```python
list=[1,5,3,4,5,2];
search_value = 5;

print("Searching for elements containing the value 5...\n")
for i in list:
    if(list[i] == search_value):
        print("Found the value ", str(search_value), " at the position ", str(i+1))
```

```
Searching for elements containing the value 5...

Found the value  5  at the position  2
Found the value  5  at the position  5
```

Figure 41: Searching operation in Python

Updating operation

The updating operation is used to update an element in the array at a specific index. Figure 42 and Figure 43 offer an example implementation of the updating operation in C++ and in Python, respectively.

```cpp
#include <iostream>
using namespace std;

int list[]={1,5,3,4,5,2};
int size = 6;

std::cout << "Array before update: ";
int i=0;
for(i=0;i<size;i++)
    {
        std::cout << list[i] << ' ';
    }

//Updating element in index 0 to contain the value 10
std::cout << endl << endl << "Updating element in index 0 to contain the value 10... " << endl;
int update_index = 0;
list[update_index] = 10;

std::cout << "Array after update: ";
int j=0;
for(j=0;j<size;j++)
    {
        std::cout << list[j] << ' ';
    }
```

```
Array before update: 1 5 3 4 5 2

Updating element in index 0 to contain the value 10...
Array after update: 10 5 3 4 5 2
```

Figure 42: Updating operation in C++

```python
list=[1,5,3,4,5,2];

#Array before deletion
print("Array before update: ", end=' ')
for i in list:
    print(i, end=' ')

#Updating element in index 0 to contain the value 10
print("\n\nUpdating element in index 0 to contain the value 10... ")
list[0] = 10

#Array after deletion
print("Array after update: ", end=' ')
for i in list:
    print(i, end=' ')
```

```
Array before update:  1 5 3 4 5 2

Updating element in index 0 to contain the value 10...
Array after update:  10 5 3 4 5 2
```

Figure 43: Updating operation in Python

Multidimensional arrays

The array data structure may also be given a multidimensional form by storing arrays inside arrays. Multidimensional arrays are useful to store data represented in matrix form.

Example 1.8

To obtain a better understanding of multidimensional arrays, consider the following matrix:

$$M = \begin{bmatrix} 1 & 4 & 5 \\ 2 & 8 & 3 \end{bmatrix}$$

The previously shown matrix is a 2x3 matrix which means that it has 2 rows and 3 columns. This matrix may be stored in an array variable called m containing one array for each row in the matrix. Figure 44 and Figure 45 show how the array variable would be declared in C++ and in Python, respectively.

```cpp
#include <iostream>
using namespace std;

int m[2][3] = {{1, 4, 5}, {2, 8, 3}};

//Printing declared multi-dimensional array
//Iteratively moving through the rows
for(int i=0; i<2; i++)
{
    //Iteratively moving through the columns and printing element
    for(int j=0; j<3; j++)
    {
        std::cout<<m[i][j]<<' ';
    }
    std::cout<<'\n';
}
```

```
1 4 5
2 8 3
```

Figure 44: Declaring multidimensional array and outputting stored matrix in C++

```python
m = [[1, 4, 5], [2, 8, 3]]

#Printing declared multi-dimensional array
#Iteratively moving through the rows
for i in range(len(m)):
    #Iteratively moving through the columns and printing element
    for j in range(len(m[i])):
        print(m[i][j], end=" ")
    print()
```

```
1 4 5
2 8 3
```

Figure 45: Declaring multidimensional array and outputting stored matrix in Python

Furthermore, it's important to note that simple array operations such as deleting, inserting, and updating are similar for the multidimensional array. The main difference is that each operation will require the inclusion of the iteration through the rows and columns of the array as shown in Figure 44 and 1.45.

(1.3.7) Pointers and References

The concept behind pointers and references is also very important for programming since they facilitate the operations that need to be executed using memory addresses instead of values. A pointer is a variable that stores the memory address of another variable. On the other hand, a reference is a variable that serves as an alias for another variable in the program. The way in which pointers and references are declared and used in a program may vary according to the programming language. While C++ uses the * and & operators to declare pointers and references, Python uses the id( ) function to access the memory address of variables and the ctypes.cast( ) function, from the ctypes library, to access the values stored in memory addresses.

In the case of programming with C++, Pointers need to be dereferenced with the * operator to access the value in the memory location it points to. References are declared by assigning them the address of another variable through the & operator. As a result, the reference may also be viewed as a constant pointer due to the fact that it stores the address of a specific variable in program. Note that whenever the reference is used, the compiler automatically applies the * operator to access the value in the memory location of the variable to which it refers.

Example 1.9

To obtain a better understanding of pointers and references in C++, consider the code shown in Figure 46 which shows three declared variables and their respective values and memory addresses.

```
#include <iostream>
using namespace std;

int a = 21;
//Declaring reference variable
int &ref = a;
//Declaring pointer variable
int *ptr = &a;

std::cout<<"Value of a: "<<a<<'\n';
std::cout<<"Memory address containing value of a: "<<&a<<"\n\n";
std::cout<<"Value of ref: "<<ref<<'\n';
std::cout<<"Memory address containing value of ref: "<<&ref<<"\n\n";
std::cout<<"Value of ptr: "<<ptr<<'\n';
std::cout<<"Value in memory address contained in ptr: "<<*ptr<<'\n';
std::cout<<"Memory address containing value of ptr: "<<&ptr<<'\n';
```

```
Value of a: 21
Memory address containing value of a: 0x7efc20bd5028

Value of ref: 21
Memory address containing value of ref: 0x7efc20bd5028

Value of ptr: 0x7efc20bd5028
Value in memory address contained in ptr: 21
Memory address containing value of ptr: 0x7efc20bd5030
```

Figure 46: Pointers and references in C++

In the case of programming with Python, pointers may be declared by assigning the memory address of one variable, using the id( ) function, to another variable. Then, as previously mentioned, value stored in the memory address contained in the pointer may be accessed by using the ctypes.cast( ) function. On the other hand, references may be declared by simply assigning one variable to another.

Example 1.10

To obtain a better understanding of pointers and references in C++, consider the code shown in Figure 47 which shows three declared variables and their respective values and memory addresses.

```python
import ctypes

a = 21
#Declaring reference variable
ref = a
#Declaring pointer variable
ptr = id(a)

print("Value of a: ", a)
print("Memory address containing value of a: ", id(a), '\n')
print("Value of ref: ", ref)
print("Memory address containing value of ref: ", id(ref), '\n')
print("Value of ptr: ", ptr)
print("Value in memory address contained in ptr: ", ctypes.cast(ptr, ctypes.py_object).value)
print("Memory address containing value of ptr: ", id(ptr))
```

```
Value of a:  21
Memory address containing value of a:  94483412676480

Value of ref:  21
Memory address containing value of ref:  94483412676480

Value of ptr:  94483412676480
Value in memory address contained in ptr:  21
Memory address containing value of ptr:  139906483358928
```

Figure 47: Pointers and references in Python

Note that for both of the previously shown examples the declaration of the reference variable did not reserve a new memory location for the program. On the other hand, the declaration of the pointer did reserve a new memory location for the program. Furthermore, if the value of the reference variable or the pointer variable using the * operator were to be changed, this change would have been reflected on the variable to which it refers. This effect is shown in Figure 48 using C++.

```cpp
#include <iostream>
using namespace std;

int a = 21;
//Declaring reference variable
int &ref = a;
//Declaring pointer variable
int *ptr = &a;

std::cout<<"Value of a: "<<a<<'\n';
std::cout<<"Value of ref: "<<ref<<"\n\n";

std::cout<<"Changing value of ref...\n";
ref = 22;
std::cout<<"Value of a: "<<a<<'\n';
std::cout<<"Value of ref: "<<ref<<"\n\n";

std::cout<<"Changing value to which ptr points...\n";
*ptr = 23;
std::cout<<"Value of a: "<<a<<'\n';
std::cout<<"Value to which ptr points "<<*ptr<<'\n';
```

```
Value of a: 21
Value of ref: 21

Changing value of ref...
Value of a: 22
Value of ref: 22

Changing value to which ptr points...
Value of a: 23
Value to which ptr points 23
```

Figure 48: Changing variable values using pointers and references

C++ has greater capabilities when dealing with pointers and references. For this reason, when dealing with problems that require the use of these types of variables, the use of C++ highly recommended.

(1.3.8) Functions

A function is a set of instructions that is used to execute a specific task in a program. Functions run only when called. As a result, they allow for code to be reused whenever a specific task needs to be executed more than once in a program or the same task is needed for a different program. Some functions require data to run properly. This data is known as the function's parameters. Additionally, while some functions return a value after executing, others do not. Libraries may be imported into a program to include sets of functions that target specific tasks as shown in Examples 1.11-12. On the other hand, programming languages include various built-in functions such as the main( ) function in C++ which allows for the execution of a program. Note that the use of the main( ) function has not been necessary for the C++ examples in this guide due to the fact that a Jupyter kernel is being used to interpret and execute the written code.

Example 1.11

In C++, the cmath library may be imported to include the sqrt( ) function in program where the square root of a number needs to be calculated as shown in Figure 49.

```
#include <iostream>
#include <cmath>
using namespace std;

int num = 36;
int num_sqrt = 0;

//Calculating square root of num
num_sqrt = sqrt(num);

//Outputting square root of num
std::cout<<num_sqrt;
```

```
6
```

Figure 49: Using the cmath library in C++ to calculate the square root of a number

Example 1.12

In Python, the numpy library may be imported to include the numpy.array( ) function in program where a list needs to converted into a numpy array as shown in Figure 50.

```
import numpy as np

#Declaring list
stock_price_list = [23.32, 23.50, 23.45, 23.53]

#Converting list into numpy array
stock_price_array = np.array(stock_price_list)

#Outputting variable types
print(type(stock_price_list))
print(type(stock_price_array))
```

```
<class 'list'>
<class 'numpy.ndarray'>
```

Figure 50: Using the numpy library in Python to convert a list into a numpy array

From another point of view, the way in which a function is defined and declared varies according to the programming language. C++ functions are defined by including the following:

- Function name: Implies the function's main task.
- Function body: Includes the necessary instructions to execute its task.
- Function parameters: Specifies names and datatypes for the arguments that the function will receive when called, if any.
- Function datatype: Specifies the datatype of value that the function will return, if any. If the function does not return a value then, the function type will be void.

An example of how a function would be defined and declared in C++ is shown in Figure 51. The function being defined in the code shown in Figure 51 executes the sum of two numbers and returns the result of the summation. Also, note that function declaration is required in C++ the function has been defined in one source file and then, called in another file.

```
//Defining sum function
int sum(int a, int b)
{
    return a + b;
}
```

```
#include <iostream>
using namespace std;

//Declaring sum function
int sum(int, int);

int x = 10;
int y = 20;
int z = 0;

//Calling sum function and storing result in z
z = sum(x,y);

//Outputting value of z
std::cout<<z;
```

```
30
```

Figure 51: Defining and declaring a function in C++ to execute the sum of two numbers and return the result

On the other hand, Python functions are declared in a way that is similar to C++. Python functions are declared by including function name, function body, and function parameters. However, the function parameters do not need to specify their datatypes and the function type does not need to be specified. An example of how a function would be declared in Python is shown in Figure 52. The function being declared in the code shown in Figure 52 executes the same task as the function declared in the code shown in Figure 51.

```
#Declaring sum function
def sum(a, b):
    return a + b

x = 10; y = 20

#Calling sum function and storing result in z
z = sum(x,y)

#Outputting value of z
print(z)
```

```
30
```

Figure 52: Declaring a function in Python to execute the sum of two numbers and return the result

Function arguments

As previously mentioned, if a function uses arguments then, these must be specified in the function's definition as the function parameters. Function parameters behave like local variables that are created upon entering the function and destroyed upon leaving the function. When calling a function in C++, there are three ways in which the arguments may be passed: by value, by pointer, or by reference. On the other side, in Python, all arguments are passed by reference.

Passing an argument by value

When passing an argument by value, the value of the argument is copied and assigned to its respective function parameter. Thus, changes made on function parameters will not be reflected on their respective arguments. An example of a function call that passes an argument by value is shown in Figure 51.

Passing an argument by pointer

When passing an argument by pointer, the memory address of the argument is copied and assigned to its respective function parameter. Thus, changes made on function parameters will be reflected on their respective arguments. An example of a function call that passes arguments by pointer is shown in Figure 53.

Passing an argument by reference

Finally, when passing an argument by reference, the reference variable of the argument is copied and assigned to its respective function parameter. Thus, changes made on function parameters will be reflected on their respective arguments. This effect may be observed in Figure 53 which offers an example of a function call that passes an argument by reference using C++. On the other hand, an example of a function that passes an argument by reference using Python may be observed in Figure 52. Note that all function arguments in Python are passed by reference.

```
//Defining function to get the average of values in an array
//and change the value of an argument
float avg(float *arr, int size, int &change)
{
    float avg = 0;
    float sum = 0;

    //Calulating the sum of the values in
    //the array passed by pointer
    for(int i=0; i<size; i++)
    {
        sum+=arr[i];
    }

    //Calculating the average of the values in
    //the array passed by pointer
    avg = sum/size;


    //Changing value of an argument passed by reference
    change = 5;

    return avg;

}
```

```
#include <iostream>
using namespace std;

//Declaring avg function
float avg(float*, int, int&);

float x[] = {21, 22, 21, 22, 22, 22};
int size = 6;
float average = 0;
int changed = 10;

//Calling avg function, passing x by pointer, and passing changed by reference
average = avg(x,size,changed);

//Outputting the calculated average and the changed argument value
std::cout<<"Calculated average: "<<average
<<"\n\nChanged argument value: "<<changed;
```

```
Calculated average: 21.6667

Changed argument value: 5
```

Figure 53: Passing arguments using pointers and references in C++

Using default values when no arguments are passed

Function parameters may also have default values such that if no values were specified using arguments when the function was called then, the default values specified in the function's definition are assigned to the function parameters. Examples of functions with default values are shown in Figure 57 and Figure 58.

```
//Defining sum function
int sum(int a=1, int b=2)
{
    return a + b;
}
```

```
#include <iostream>
using namespace std;

//Declaring sum function
int sum(int, int);

int x = 10;
int y = 20;
int z1 = 0;
int z2 = 0;

//Calling sum function and storing result in z
z1 =::sum(x);
z2 =::sum();

//Outputting value of z1 and z2
std::cout<<"Result from passing only one argument to sum(): "<<z1<<'\n';
std::cout<<"Result from passing zero arguments to sum(): "<<z2;
```

```
Result from passing only one argument to sum(): 12
Result from passing zero arguments to sum(): 3
```

Figure 57: C++ function with default values

```
#Declaring sum function
def sum(a=1, b=2):
    return a + b

x = 10; y = 20

#Calling sum function and storing result in z
z1 = sum(x)
z2 = sum()

#Outputting value of z
print("Result from passing only one argument to sum():", z1)
print("Result from passing zero arguments to sum():", z2)
```

```
Result from passing only one argument to sum(): 12
Result from passing zero arguments to sum(): 3
```

Figure 58: Python function with default values

(1.3.9) Classes and Objects

Everything in C++ and Python is associated with classes and objects. This is due to the fact that C++ and Python are both object-oriented programming (OOP) languages. The object-oriented programming approach solve problems by creating objects using classes. Objects can be viewed as a collection of data and functions that act on that data. As a result, objects can be defined through the following two characteristics: attributes and behavior.

Example 1.13

Consider the case of declaring a specific dog as an object. In this case, the object would have the following properties concerning attributes and behavior:

- Attributes: breed, age, and weight
- Behavior: bark, eat, and sleep

On the other hand, a class can be viewed as the object's blueprint. In more detail, classes are user defined datatypes that specify the data members (attributes) and member functions (behavior) for

an object. To further understand classes, consider the case presented in Example 1.13. In this case, the specific dog object may be declared by defining the dog classes which specifies the attributes and behavior to be expected from a dog.

Defining a class and declaring objects

It's important to note that no memory is allocated when defining classes since they only specify the attributes and behavior to be expected from an object. As a result, the data and functions defined in a class may only be accessed by declaring an object. The syntax for defining classes and declaring objects varies according to the programming language. Figure 59 and Figure 60 show the syntax to define a class in C++ and in Python, respectively.

```
class ClassName
{
    access specifier: //Possible accesses: private, public, or protected

    data members; //Specifies object attributes

    member functions(){} //Specifies object behavior which
                         //involves a set of functions that
                         //have access to data members
};
```

Figure 59: Syntax for defining a class in C++

```
class ClassName:
    #Statement 1
    #Statement 2
    #Statement 3
    #...
    #Statement N
```

Figure 60: Syntax for defining a class in Python

From Figure 59, one may observe that the definition of a class in C++ requires an access specifier to indicate the type of access that a data member or member function will have. On the other hand, from Figure 60, one may observe that the definition of a class in Python does not require an access specifier. This is due to the fact that all data member and member functions, specified as statements in a Python class, are public.

Example 1.14

Taking into consideration the previously shown syntax for defining classes in C++, the dog class may be defined as shown in Figure 61. Then, the dog object may be declared and its member functions may be implemented as shown in Figure 62. From Figure 61, note that a default and parametrized constructors were defined. Constructors are used to initialize the data members of an object. In other words, constructors are used to assign values to each of the object's attributes. While default constructors add default values to each of the object's attributes, parametrized constructors add user specified values. It is always good practice to define both default constructors and parametrized constructors in a class. Additionally, it is recommended to include get, set, and print functions for each object attribute when defining the member functions. This practice may be observed in Figure 61.

```cpp
#include <iostream>
using namespace std;

class dog
{
    private:

    //Data members
    std::string breed;

    public:
    //Member functions

    //Default constructor
    dog()
    {
        breed = "Dalmation";
    }

    //Parametrized constructor
    dog(std::string s_breed)
    {
        breed = s_breed;
    }

    //Functions to assign values to object attributes
    void setBreed(std::string s_breed)
    {
        breed = s_breed;
    }

    //Functions to get values from object attributes
    std::string getBreed()
    {
        return breed;
    }

    //Functions to print values of object attributes
    void printBreed()
    {
        std::cout<<breed<<'\n';
    }

    //Functions to imitate dog behavior
    void bark()
    {
        std::cout<<"Woof!\n";
    }

    void sleep()
    {
        std::cout<<"Zzzzzz\n";
    }

    void eat()
    {
        std::cout<<"Munch Munch\n";
    }
};
```

Figure 61: Defining dog class in C++

```cpp
#include <iostream>
using namespace std;

dog Spike; //Using default constructor to declare object
dog Marley("Labrador"); //Using parametrized contructor to declare object

//Printing dog breeds
std::cout<<"Spike's breed is: "; Spike.printBreed();
std::cout<<"Marley's breed is: ";Marley.printBreed();

//Calling dog functions
Spike.bark();
Marley.sleep();
Spike.eat();
```

```
Spike's breed is: Dalmation
Marley's breed is: Labrador
Woof!
Zzzzzz
Munch Munch
```

Figure 62: Declaring and implementing dog object in C++

Example 1.15
On the other hand, taking into consideration the previously shown syntax for defining classes in Python, the dog class may be defined as shown in Figure 63. Then, the dog object may be declared and its member functions may be implemented as shown in Figure 64.

```
#Defining dog class
class dog:

    #Constructor
    def __init__(self, sbreed="Dalmation"):
        #Data members
        self.breed = sbreed

    #Member functions
    #Functions to assign values to object attributes
    def setBreed(self, sbreed):
        self.breed = sbreed

    #Functions to imitate dog behavior
    def bark(self):
        print("Woof!")

    def sleep(self):
        print("Zzzzzz")

    def eat(self):
        print("Munch munch")
```

Figure 61: Defining dog class in Python

```
#Declaring and implementing dog objects
Spike = dog() #Using default constructor to declare object
Marley = dog("Labrador") #Using parametrized contructor to declare object

#Printing dog breeds
print("Spike's breed is: ", Spike.breed)
print("Marley's breed is: ", Marley.breed)

#Calling dog functions
Spike.bark()
Marley.sleep()
Spike.eat()
```

```
Spike's breed is:  Dalmation
Marley's breed is:  Labrador
Woof!
Zzzzzz
Munch munch
```

Figure 61: Defining dog class in Python

This concludes the technical guide for compiler theory and practical programming. The following section will cover the topic of algorithms and data structures.

(2) Algorithms and data structures
Every computer science and computer engineering major must have proper knowledge concerning data structures and algorithms (DSA). This knowledge is necessary to be able to understand the logic behind many of the abstractions that are present in a program's implementation. Additionally, it enables developers to identify more easily many programming errors that may emerge from lack of knowledge in DSA. For this reason, this section is divided into two main sections. The first

section introduces the concept of an algorithm and its different characterizations. The second section explains a data structure and the different forms it may take according to its real-world application.

(2.1) Algorithms

An algorithm may be defined as a sequence of precise and unambiguous instructions telling a computer what to do. It's important to note that not any sequence of instructions may be defined as an algorithm To be able to define a sequence of instructions as an algorithm, the sequence must have the following characteristics [22] – [23]:

- Clear and unambiguous: The sequence should be clear and unambiguous. The repeated execution of a sequence using the same starting parameters should always lead to the same result.
- Input clarity: If a sequence requires any inputs, these inputs must be clearly defined.
- Output clarity: The sequence must clearly define the outputs to be obtained. Also, the amount of outputs should be greater or equal to one.
- Finite-ness: The sequence should end after a finite number of steps.
- Feasible: The execution of the sequence should not depend on future technology and should by achievable with the currently available resources.
- Language independent: The instructions in the sequence should be independent of any programming language.

Computers are made of billions of switches. The simplest algorithm would instruct a computer to flip a switch (1 or 0). However, this section will focus on more complex algorithms, particularly those involved when implementing data structures. These algorithms may be divided into the following categories according to their main functionality in the data structure [22]:

- Search algorithm: Used to search for an item in the data structure
- Sort algorithm: Used to sort items in the data structure
- Insert algorithm: Used to insert an item into the data structure
- Update algorithm: Used to update an item already inside the data structure
- Delete algorithm: Used to delete an item already inside the data structure

(2.1.1) Designing an algorithm

Algorithms are designed to solve computational problems given a limited amount of resources. Thus, the problem domain must be clearly defined before writing any algorithm. Also, the amount of resources available must be taken into consideration while writing the algorithm. From another point of view, many algorithms may be devised to solve the same computational problem. Thus, when designing an algorithm for a specific problem, many solution algorithms are written and evaluated. Then, among the written algorithms, the most efficient one is selected for the final design. In general, the most efficient algorithm would be the one which consumes the least amount of memory space and executes fastest. The performance of an algorithm can be analyzed in two different phases: before its implementation (priori analysis) and after its implementation (posteriori analysis) through a programming language [22]. Both types of analyses take into consideration two important properties of the algorithm to evaluate its performance which are space complexity and time complexity [24].

The space complexity of an algorithm specifies the amount of memory space required by the algorithm during its execution. In general, algorithm require the following three types of memory space:

- Instruction space: Stores the executables version of the program.
- Environment space: Stores the environment information that is necessary when calling nested functions.
- Data space: Stores all constants and variables, including those which are temporary.

However, only the data space requirement is taken into consideration when calculating the space complexity of an algorithm. To calculate the space complexity, the value of the memory used for the different datatype variables, which can vary according to the operating system, must be known.

Example 2.1
To provide an example of how the space complexity may be calculated, consider the piece of code shown in Figure 62:

```
def add(a, b):
    return a + b
```

Figure 62: Simple code in Python for Example 2.1

The code in Figure 1 contains two variables all of which are assumed to have the same datatype (int). Since the int datatype requires 4 bytes of memory, the total memory requirement for the previous code is equal to (4*2 + 4) = 12 bytes. Note that an additional 4 bytes are summed for the return value.

The time complexity of an algorithm specifies the amount of time a program implementing the algorithm would have to run to complete its execution. The time requirements of an algorithm can be defined as a numerical function T(n), where T(n) can be measured as the number of steps with each step consuming a constant amount time. To provide an example of how T(n) could be calculated, consider the addition of two n-bit integers taking n steps to complete. In this case, the time requirement T(n) would be equal to $c * n$, where c is the time taken for the addition of two bits. As a result, it may be noted that T(n) grows linearly as the input size increases.

On the other hand, the exact time required by an algorithm can never be truly defined. For this reason, a standard notation, known as asymptotic notation, is used. By implementing an asymptotic notation, an algorithm's time requirement may be identified in terms of best case, average case, or worst case scenario. The most commonly used asymptotic notations are big-O notation, omega notation, and theta notation. Big-O notation is used to express the upper bound (worst-case) of an algorithm's run time. Omega notation is used to express the lower bound (best case) of an algorithm's run time. Finally, theta notation expresses both the upper and lower bounds (average case) of an algorithm's run time. A few of the most common asymptotic notations are shown in Table 12.

Table 12: Common asymptotic notations in big-o

Name	Big-O
Constant	O(1)
Linear	O(n)
Quadratic	O(n^2)
Cubic	O(n^3)
Polynomial	$n^{O(1)}$
Exponential	$2^{O(n)}$
Logarithmic	O(log n)
Log linear	O(n log n)

(2.1.2) Greedy algorithms

Algorithms are designed to solve a problem in an optimal manner. Through the greedy algorithm approach, the optimal solution for a problem is found at each step of the algorithm [25]. Once the optimal solution for the current step has been found, the algorithm proceeds to the next step and does not go back to evaluate alternate solutions. This greedy approach could be beneficial in minimizing the time required to generate a solution for a given problem. However, in many problems, this approach fails to generate the globally optimal solution due to how it focuses on searching for the localized optimal solution at each step of the algorithm.

Example 2.2

To obtain a better understanding on how a simple greedy algorithm would work, consider the binary trees shown in Figure 63 and 64 which aim to find the path with the largest sum.

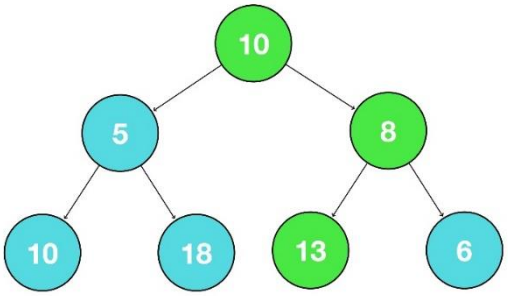

Figure 63: Finding the localized optimal solution for binary tree

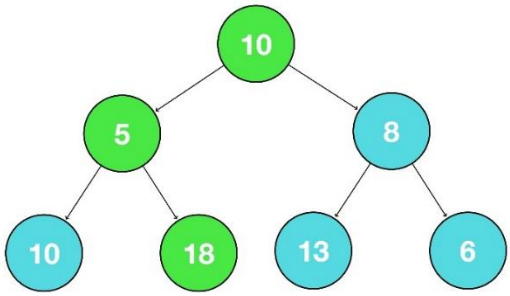

Figure 64: Finding the globally optimal solution for binary tree

The result from using the greedy approach may be observed in Figure 63 where the solution is 10 + 8 + 13 = 31. On the other hand, the result of implementing an approach that targets the globally optimal solution may be observed in Figure 64 where the solution is 10 + 5 + 18 = 33. Note that Figure 63 deviates from the globally optimal solution by searching for the localized optimal solution in each level of the binary tree. Thus, a standard greedy algorithm would proceed in the following manner at each step for a given input:

- Algorithm evaluates the following question: Does the current input provide a localized optimal solution?
    - If yes, then the input will be included into the partial solution set
    - Otherwise, the input will be discarded
- Algorithm proceeds to the next step and does not reconsider the previous input.
- Algorithm continues until the input set is completed or an optimal solution is found.

As previously mentioned, the greedy algorithm may fail to find a globally optimal solution for a problem. For this reason, the greedy approach must only be used to solve problems that have the following two properties [25]:

- Greedy choice property: The global optimum may be found by searching for the local optimum
- Optimal sub-problem: The optimal solution for the problem has optimal solutions for its sub-problems.

Some examples of problems which may implement the greedy approach are the: knapsack problem, travelling salesman problem, and job scheduling problem, among others.

(2.1.3) Divide and conquer

Through the divide and conquer approach problems are solved by dividing them into smaller indivisible (atomic) sub-problems which are solved independently [26]. Then, the solutions from each sub-problem are merged to obtain an overall solution to the initial problem. From another point of view, the divide and conquer approach may also be described as a three-step process as shown below [26]:

1. Divide: Initial problems is divided into smaller indivisible (atomic) sub-problems.
2. Solve: Atomic sub-problems are solved.
3. Merge: Solutions to atomic sub-problems are recursively combined until the solution to the initial problem is obtained.

Some examples of algorithms which apply the divide and conquer approach are merge sort, quick sort, and binary search which will be discussed in the section *searching and sorting techniques*.

(2.1.4) Dynamic programming

Dynamic programming is used whenever the solution to a problem may be optimized by dividing the problem into similar sub-problems such that their results may be re-used. As George Santayana once said: “Those who cannot remember the past are condemned to repeat it” [27] – [28]. Dynamic

programming coincides with the quote from Santayana by storing the results of the similar sub-problems so that their computation does not have to be repeated [27] – [28].

In comparison with the greedy algorithm which focuses on the local optimization for a problem, the dynamic algorithm focuses on the overall optimization for a problem. On the other hand, it's important to note that both the divide and conquer approach and the dynamic programming approach work by dividing a problem into sub-problems. However, while the divide and conquer approach solves each sub-problem independently to achieve an overall solution, the dynamic programming approach does not solve each sub-problem independently. Instead, it remembers the results from smaller sub-problems to be able to optimize the bigger sub-problem.

Example 2.3
Consider the problem in which the Fibonacci series must be calculated. The algorithm to obtain the Fibonacci series, starting with the first digit space n = 0, may be expressed as follows:

Fibonacci(n) = 1; if n = 0
Fibonacci(n) = 1; if n = 1
Fibonacci(n) = Fibonacci(n – 1) + Fibonacci(n – 2)
The first numbers of the Fibonacci series are: 1, 1, 2, 3, 5, 8, 13, … , and so on. Then, taking the previously shown algorithm into consideration, the Fibonacci series may be calculated by implementing recursion as shown in the code in Figure 65.

```
int FS_recursion (int n) {
      if (n < 2)
          return 1;
      return FS_recursion(n-1) + FS_recursion(n-2);
  }
```

Figure 65: Code in C++ using recursion to solve the Fibonacci series

Note that the code shown in Figure 65 executes repeated calls for the same input. As a result, the code may be optimized by using the dynamic programming approach as shown in Figure 66.

```
void FS ()
   {
       FSresult[0] = 1;

       FSresult[1] = 1;

       for (int i = 2; i<n; i++)
          FSresult[i] = FSresult[i-1] + FSresult[i-2];
   }
```

Figure 66: Code in C++ using dynamic programming to solve the Fibonacci series

A few more examples of problems which may implement the dynamic programming approach besides the Fibonacci series are the: tower of Hanoi problem, the shortest path by Dijkstra, knapsack problem, and job scheduling problem, among others.

(2.1.5) Searching and sorting techniques

Finally, when discussing the topic of algorithms it's important to discuss searching and sorting techniques. Searching and sorting algorithms are frequently implemented in programs due to their capabilities in increasing efficiency when dealing with a large amount of data. There are different types of searching and sorting algorithms and generally, programmers select the type of algorithm to use based on the size and structure with which they are dealing.

Searching algorithms

In terms of searching algorithms, the general problem to be solved is the following: Locate an element x in a list of elements $a_1, a_2, a_3, \dots, a_n$ or determine whether or not the x is contained in the list of elements. Some of the most commonly used searching algorithms are the following: linear search, binary search, and interpolation search.

The linear search algorithm executes the searching operation by going over each element in the list and comparing it with the element to be located [29]. In this case, the search ceases only when a match occurs or the end of the list has been reached.

The binary search is a faster search algorithm that executes the searching operation by implementing the divide and conquer approach [30]. The binary search starts its search for an element by comparing it with the middle most element in the list. If a match occurs, the search ceases. Otherwise, depending on whether the middle most element is greater or smaller than the element to be located, the search will continue in the sub-array to the left or the right of the middle most element, respectively. In this case, the search ceases only when a match occurs or when the size of the sub-array has been reduced to zero. Note that this searching technique only works when the data is in sorted form.

On the other hand, an improved variant of the binary search is the interpolation search which executes the searching operation by probing for the location of the element to be located [31]. While the binary search always goes to the middle element to search for a match, the interpolation search may go to different locations to search for a match according to the following position formula:

$$pos = \text{startIndex} + \frac{(x - arr[\text{startIndex}])(\text{endIndex} - \text{startIndex})}{arr[endIndex] - arr[\text{startIndex}]}$$

In the position formula: arr[ ] refers to the list or array of elements, x refers to the element to be located, startIndex refers to the starting index of the list, and endIndex refers to the ending index of the list. Note that the position formula will return a higher value when the element to be located is closer to arr[endIndex] and a smaller value when closer to arr[startIndex]. This searching technique only works when the data is in sorted form and equally distributed.

Sorting algorithms

In terms of sorting algorithms, the general problem to be solved is to order the elements in a list in a specific format. The most common formats in which to order elements are in numerical or lexicographical order. Sorting algorithms may be divided into different types according to the following characteristics [32]:

- In-place sorting or not-in-place sorting: While some algorithms require temporary storage to execute comparisons between elements to be sorted, other algorithms do not require temporary space an execute sorting in-place. An example of an in-place searching algorithm is the bubble sort.
- Stable or unstable sorting: While some algorithm maintain the sequence of elements with similar values (stable sorting), other algorithms do not maintain the sequence of these types of elements (unstable sorting).
- Adaptive and non-adaptive sorting: While some algorithms benefit and adapt when receiving sorted elements, other algorithms do not consider the already sorted elements in a list.

Some of the most commonly used sorting algorithms are the following: bubble sort, insertion sort, selection sort, merge sort, shell sort, and quick sort. These sorting algorithms will not be discussed in detail. However, the following brief descriptions are offered for each sorting algorithm:

Bubble sort
Comparison-based sorting algorithm in which each pair of adjacent elements is compared and if they are not in order then, they are swapped.

Insertion sort
In-place comparison-based sorting algorithm in which an always sorted sub-list is maintained. Elements are inserted into the sub-list by executing a sequential search in the list for unsorted items.

Selection sort
In-place comparison-based sorting algorithm in which the list is divided in two parts. The sorted section is in the left end of the list and the unsorted section is in the right end of the list. At the beginning of the search, the sorted section will be empty. The sorted section is then filled by searching for the smallest element in the unsorted section and swapping it with the leftmost element in the array. For each swap the boundary of the unsorted section is reduced by one.

Merge sort
Comparison-based stable sorting algorithm that uses the divide and conquer approach by dividing the array into equal halves until they can be divided no more. Then, the element are combine in the same manner as they were broken down but, in each combination the elements are placed in sorted form.

Shell sort
In-place comparison-based sorting algorithm that is based on the insertion sort and uses a calculated value known as interval to determine the spacing between the elements in the list. This algorithm uses the insertion sort when it finds elements that are widely spread and focuses on sorting elements from most widely spread to less.

Quick sort

In-place comparison-based sorting algorithm that partitions the array containing the list of elements to be ordered in two arrays based on a selected list element called pivot. While one array holds values that are smaller than the pivot, the other holds values that are larger than the pivot. These sub-arrays are then sorted recursively.

(2.2) Data structures

A data structure can be defined as a programmatic way of collecting, organizing, and managing data. Any structure capable of storing data may be referred to as a data structure. The main goal of the data structure is to increase efficiency and reduce complexity whenever operations are performed on data [33]. There are different types of data structures and these may be classified based on the following characteristics: linear, non-linear, homogenous, non-homogenous, static, or dynamic. Linear data structures, such as arrays, are characterized by using a linear sequence to arrange data items. Non-linear data structures, such as trees and graphs, are characterized by using a non-linear sequence to arrange data items. Homogenous data structures, such as arrays, are characterized by containing only elements of the same data type. Non-homogenous data structures, such as lists, are characterized by containing elements of any data type. Static data structures, such as arrays, are characterized by having a fixed size at compile time. Finally, dynamic data structures, such as linked lists created using pointers, are characterized by having a size that may increase or decrease depending on the program's needs during its execution.

Furthermore, there are two groups in which data structures may be divided which are primitive data structures and non-primitive as shown in Figure 67 [33]. Primitive data structures are those which directly operate upon machine instructions. On the other hand, non-primitive data structures are more complex data structures that are derived from primitive data structures. The following sections will focus on discussing non-primitive data structures and their basic operations such as traversing, inserting, deleting, searching, merging, and sorting.

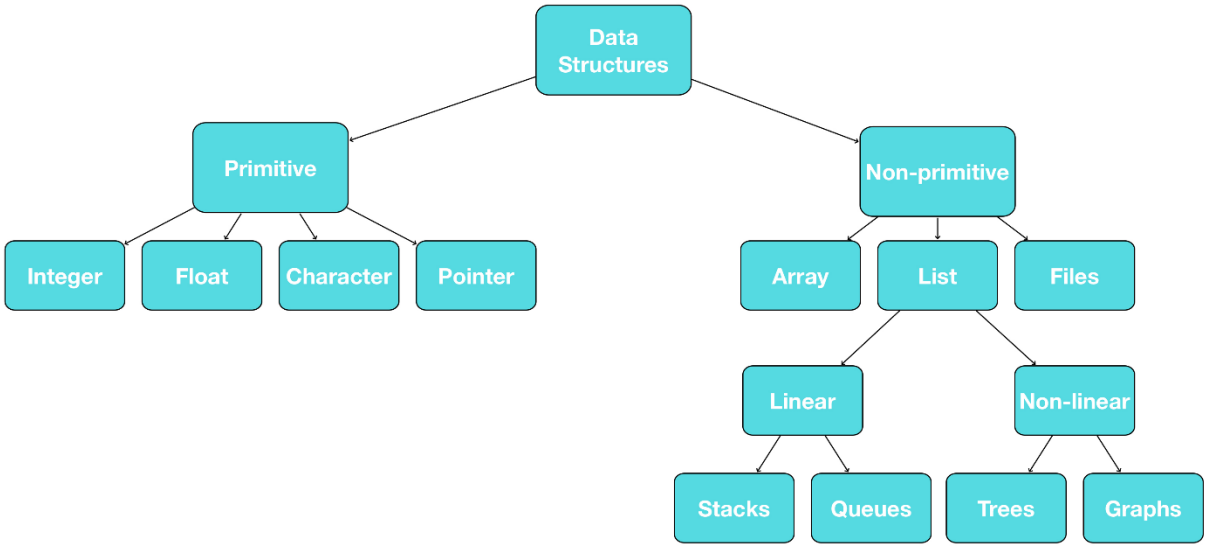

Figure 67: Types of data structures

(2.2.1) Linked Lists

A linked list may be viewed as a sequence of nodes where each node contains a data item (element) and a link to the next node [33]. The linked list is the second most-used data structure after the array. A diagram for a simple linked list is shown in Figure 68.

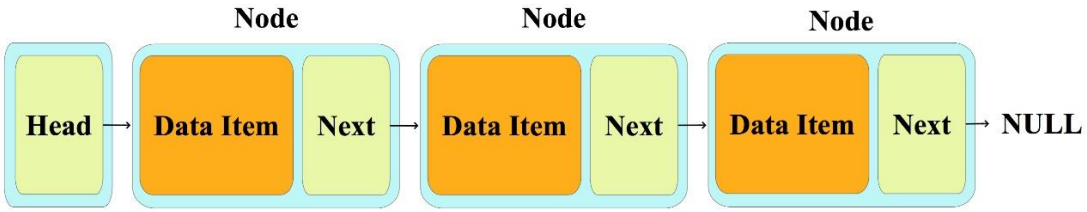

Figure 68: Simple linked list diagram

From Figure 68, note that simple linked lists always start with a link to the first node. This link is called head. On the other hand, the last node carries a link as null to mark the end of the list. The simple linked list shown in Figure 68 is characterized for using forward navigation. Also, the following operations may be executed on data by using the simple linked list data structure: insertion, deletion, display, search, and delete

Besides the simple linked list, there are two other types of linked lists that may be implemented as a data structure. These two other types of linked lists are the doubly linked list which uses forward and backward navigation, and the circular linked list which uses forward navigation and contains a link to the first node in its last. Figure 69 and Figure 70 offer a diagram for the doubly linked list and circular linked list, respectively.

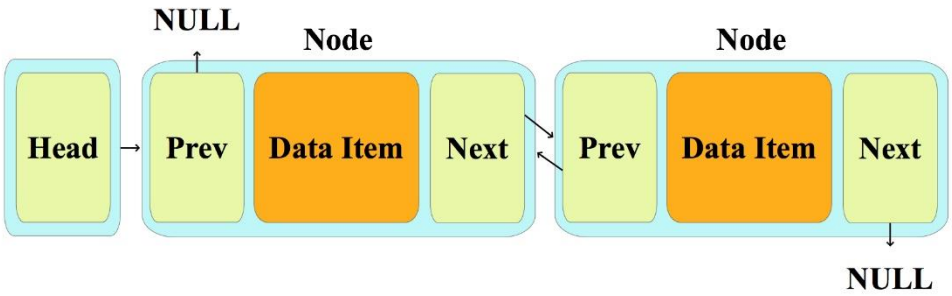

Figure 69: Doubly linked list diagram

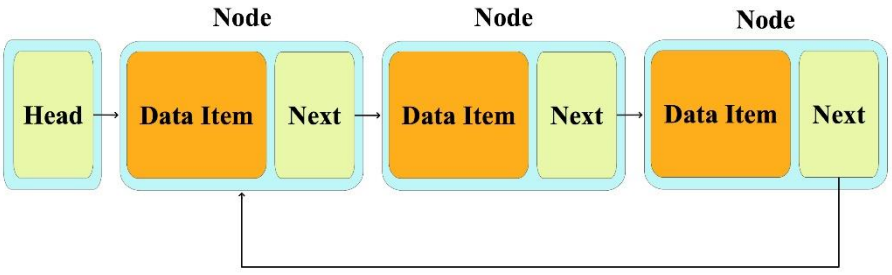

Figure 70: Circular linked list diagram

From Figure 69, note that the doubly linked list uses two links (prev and next). The prev link points to the previous node, if any. If there is no previous node then, the prev link will point to null. On

the other side, the next link points to the next node, if any. If there is no next node then, the next link will point to null. The following operations may be executed on data by using the doubly linked list data structure: insertion at beginning, insertion at end, insertion after a specific node, deletion at beginning, deletion at end, deletion of specific node, display forward, and display backward.

From Figure 70, note that the circular linked list diagram is similar to the simple linked list shown in Figure 68. The only difference is that the last node uses the next link to point to the first node instead of null. The following operations may be executed on data by using the simple linked list data structure: insertion, deletion, and display

(2.2.2) Stacks & Queues

Stacks and queues are similar data structures that only differentiate themselves by the way in which they remove elements [33]. While the stack uses the last-in-first-out (LIFO) principle, the queue uses the first-in-first-out (FIFO) principle. To better understand the concept of the stack, consider a pile of boxes where one is placed on top of another. If a box were to be removed from the pile then, the last box that was added to the pile would be the first box to go out. On the other hand, to better understand the concept of the queue, consider the waiting line to pay at a supermarket. The first person to enter the waiting line will be the first person to be attended by the cashier. Figure 71 and Figure 72 show a diagram of the stack and queue, respectively.

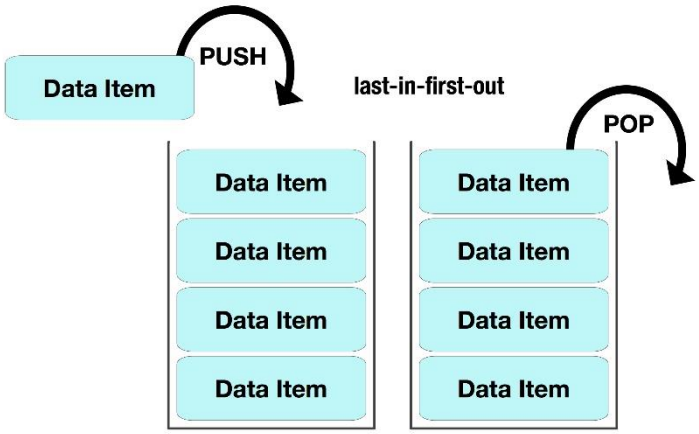

Figure 71: Stack diagram

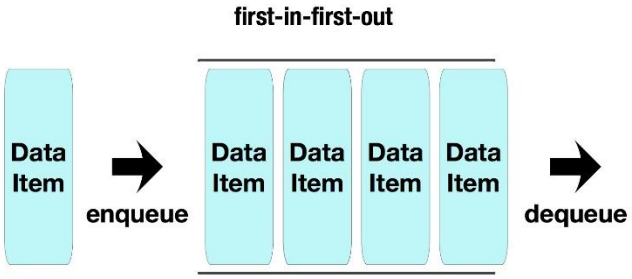

Figure 72: Queue diagram

While the stack uses the push and pop operations to add and remove data items, the queue uses the enqueue and dequeue to add and remove. On the other hand, both the stack and queue make use of the following functions: peek( ) which reads item in top of stack or at the beginning of the queue, isFull( ) which verifies if the stack or if the queue is full, and isEmpty( ) which verifies if the stack or queue is empty.

(2.2.3) Graph Data Structure

The graph data structure may be defined as a visual representation of objects in which some pairs of objects are connected by links. A graph is composed of vertices and edges. The vertices are used to represent the objects that are interconnected in the graph. The edges are used to represent the links that connect the vertices in the graph. Figure 73 shows a diagram of a graph data structure.

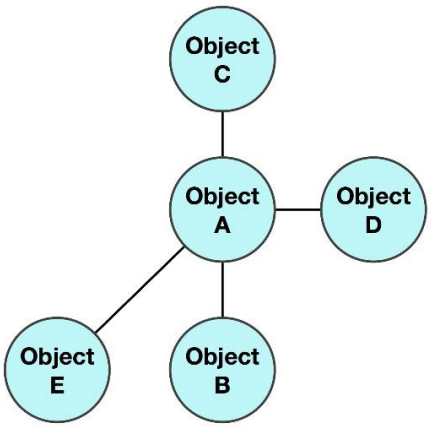

Figure 73: Graph diagram

Furthermore, a graph may be formally defined as a pair of sets (V, E), where V refers to the set of vertices in the graph and E refers to the set of edges in the graph [33]. For example, the graph shown in Figure 73 may be formally defined as follows: V = {A, B, C, D, E}, and E = {AB, AC, AD, AE}. Consequentially, graphs may be represented in a program by operating on an array of vertices and a multidimensional array of edges. The following operations may be executed on data by using the graph data structure: add vertex, add edge, display vertex.

(2.2.4) Tree Data Structure

The tree data structure may be defined as a set of nodes connected by edges [33]. Among the tree data structures, one of the most common is the binary tree which does not allow for nodes to have more than two children. The binary tree may be used for hierarchical data storage purposes. Figure 74 shows a diagram of a binary tree data structure.

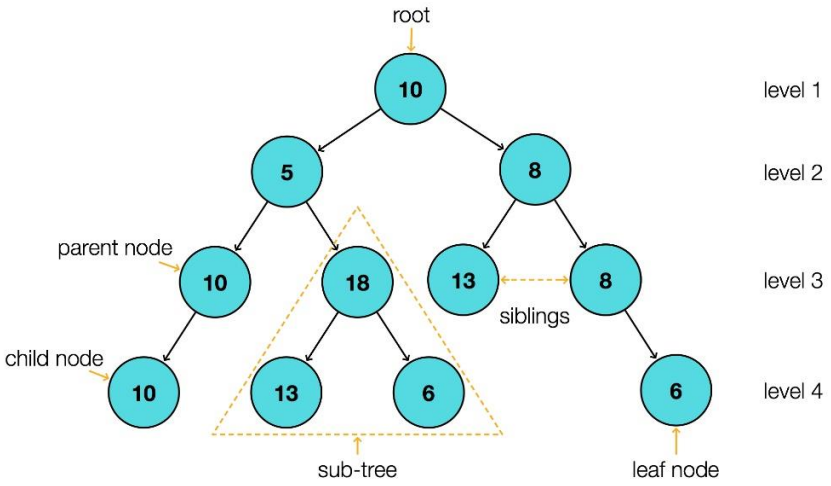

Figure 74: Binary tree diagram

The following operations may be executed on data by using the binary tree data structure: insertion, search, pre-order traversal, post-order traversal, and in-order traversal. This concludes the technical guide for algorithms and data structures. The following section is the last section of the technical guide which will cover the topic of data science for machine learning.

(3) Machine learning with Python

(3.1) Introduction

This section will focus on the topic of machine learning which is a sub-topic in artificial intelligence. The main objective in machine learning is to allow computers to learn from data by implementing learning algorithms. In the previous section, the traditional algorithm was discussed. Note that the traditional algorithm is different from the learning algorithm. The traditional algorithm aims to find a solution to a problem by taking into consideration the necessary input data and the operations to be executed on the input data in order to obtain the expected output data. In contrast, the learning algorithm aims to find the algorithm that solves problems by taking into consideration various instances of an event related to a problem. Thus, learning algorithms are intended to be multi-purpose and simple.

Furthermore, machine learning automates discovery by turning data about a specific event into knowledge. For example, the SKICAT (sky image cataloging and analysis tool) project used machine learning to identify half-billion sky objects in the Sloan Digital Sky Survey [34]. Machine learning also helps scientists solve two of their constant dilemmas which is: being overwhelmed by information and being delayed by the process of communication. For example, Manchester Institute of Biotechnology's Adam was the first machine ever to discover novel scientific knowledge independently of its human creators [35]. Adam worked at figuring out which genes encode which enzymes in yeast by using the following data in its learning algorithm: A model of yeast metabolism and general knowledge of genes and proteins. Also, Adam was capable of making hypotheses, designing experiments to test them, physically carrying them out, analyzing the results, and coming up with new hypotheses until satisfied.

From another point of view, it's important to remember that learning algorithms aim to learn an algorithm that solves any problem given the appropriate data relating to the problem. Thus, any learning algorithm could approximate any function arbitrarily closely given enough data. However, enough data could be infinite. To counteract this problem, learning algorithms have to make assumptions and these assumptions vary from one learning algorithm to another. In the best case scenario, the learning algorithm's assumptions allow it to function as a "master algorithm"; an algorithm capable of solving any problem given a finite amount of data relating to the problem [36]. According to Domingos, machine learning experts have devised five schools of thought in their journey to find the master algorithm, although no such algorithm still exists [36]. The five schools of thought are the following: symbolists, connectionists, evolutionaries, bayesians, and analogizers. Each of these schools of thought and their respective "master algorithm" will be discussed in the following section.

(3.2) The Five Tribes of Machine Learning

(3.2.1) Symbolists

This group of machine learning experts use learning algorithms that obtained intelligence by manipulating symbols. Their master algorithm is inverse deduction which means that they solve new problems by:

- Incorporating preexisting knowledge into learning
- Combining different pieces of knowledge

Before delving any further into inverse deduction, it is important to understand the concept of induction. In Principia, Newton enunciates four rules to induction, the key rule being the third one:

"The qualities of bodies, which admit neither intensification nor remission of degrees, and which are found to belong to all bodies within the reach of our experiments, are to be esteemed the universal qualities of all bodies whatsoever" [37]. This means that whatever is found to be true in something can be found to be true in everything else in the universe.

Example 3.1

To obtain a better understanding of induction and Newton's third law of induction, consider the following rules:

Rule #1: Socrates is human.
Rule #2: All humans are mortal.

Taking into consideration the previously shown rules, the following rule may be induced:

Induced rule #1: If Socrates is human, then he's mortal

On the other hand, by applying Newton's third law of induction and generalizing the induced rule to all entities (if an entity is human, then it's mortal), the following rule may be induced:

Induced rule #2: All humans are mortal

Note that by applying Newton's third law of induction one of the original rules was obtained. This is an example of inverse deduction.

When implementing inverse deduction, a truth or a concept is represented by a set of rules by turning each instance of a concept into a rule that specifies all attributes of that instance. One problem with this method is that it requires for the memorization of instances. However, the main problems with implementing inverse deduction is overfitting. This problem is generally presented in non-parametric machine learning algorithms [38]. Overfitting is an error that may occur when modeling a learning algorithm and the function is too closely fit to a limited set of data points which are the attributes of memorized instances. To avoid this problem, the number of instances that a learner can memorize is restricted in such a way that only short conjunctive concepts can be learned.

On the other hand, the algorithm of choice for symbolist to implement inverse deduction is the decision tree. The decision tree is similar to a game of n questions concerning one instance. In the tree, each node asks about the value of one attribute and, depending on the answer, a specific leaf is reached. Note that each path from root to leaf is a rule.

Example 3.2
An example implementation of the decision tree may be observed when finding a classifier that predicts the following sets of classes: Republican, Democrat, or Independent. Then, based on the votes from each class concerning laws for taxes and gun restrictions the decision tree in Figure 75 may be devised.

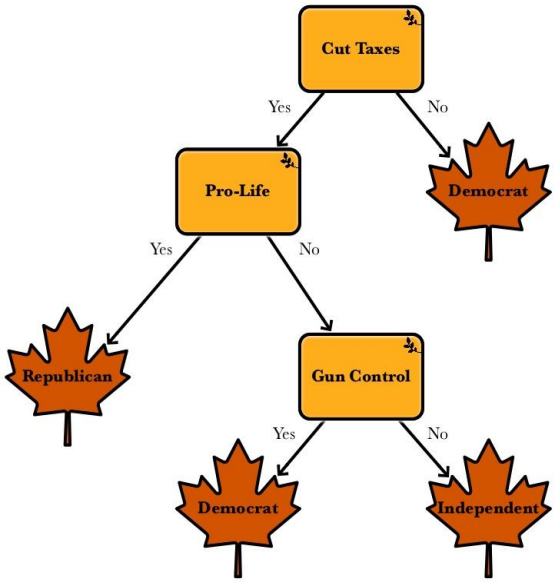

Figure 75: Decision tree to classify political party

Consequentially, to incorporate decision trees in a learning problem two classes may be devised: the decision node and the decision tree which is composed of decision nodes.

(3.2.2) Connectionists

Connectionists believe that knowledge is stored in the connections between neurons. Thus, their main goal is to simulate to most widely implemented learner in the world; the human brain. The idea of simulating the human brain to design a learning algorithm was born from Donald Hebb's rule: "Neurons that fire together wire together" [39]. Hebb was the first to propose that neurons could encode associations.

A successful simulation of the brain could be possible with a universal Turing machine and enough time and memory. Additionally, the brain must be properly understood along with Hebb's rule. To obtain a better understanding of the brain's structure, the forest analogy may be used as follows:

Forest → Brain
Tree → Neuron
Tree's trunk → Axon
Tree's root → Dendrites
Using the forest analogy, synapses may be viewed as the connections between the branches of two or more trees.

From another point of view, during Hebb's time, there was no way to measure synaptic strength or change in it and figure out molecular biology of synaptic change. However, in our current time neurons are much more properly understood. Nowadays, neurons are known to have different concentrations of ions that create voltage across membrane [40]. When a presynaptic neuron fires, neurotransmitter molecules are released into the synaptic cleft. As a result, the postsynaptic neuron's membrane opens, and lets in potassium and sodium ions and changes the voltage across the membrane. If enough presynaptic neurons fire close together then, voltage spikes and an action potential travels down the postsynaptic neuron's axon. Furthermore, when postsynaptic neurons fire, synapses grow or from anew.

Taking into consideration the knowledge that researchers have been able to obtain concerning neurons, machine learning experts started their goal of simulating the brain by modeling its most basic unit; the neuron. The first model of the neuron was proposed in 1943 by Warren McCulloch and Walter Pitts [41]. This model was similar to logic gates and was capable of executing computer operations. However, it was incapable of learning since it lacked the concept of the perceptron devised by Frank Rosenblatt [42]. The perceptron gives variable weights to the connections between neurons. A positive weight would represent an excitatory connection and a negative weight would represent an inhibitory connection. Then, the output of the neuron would be determined according to whether the weighted sum of the inputs was greater or equal to the threshold or less than the threshold. If the weighted sum of the inputs were greater or equal than the threshold then, the neuron's output would be 1. Otherwise, the neuron's output is 0. Figure 76 shows a diagram of the perceptron.

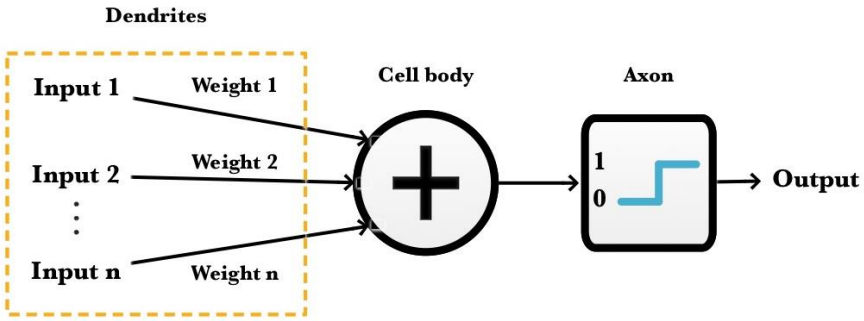

Figure 76: Diagram of the perceptron

However, in 1969, Marvin Minsky detailed the shortcomings of the perceptron [43]. Minsky found that the perceptron was incapable of learning simple things such as XOR. Additionally, the main limitation of the perceptron was that it only modeled a single neuron's learning. This problem could have been solved with layers of interconnected neurons but, there was no way to learn them and no clear way to change the weights of the neurons in the "hidden" layers and reduce error.

Given the discovered limitations of the perceptron, connectionists studied new techniques for modeling neurons and neural networks until the concept of backpropagation emerged [44]. Backpropagation is currently the connectionist's master algorithm. The key difference in backpropagation compared to the perceptron algorithm is that the neuron's output is continuous. In this case, the error (E = desired output – neuron's output) can be propagated to the hidden neurons and each neuron may decide how much more or less to fire. In general, the learning method for backpropagation may be summarized as follows:

- Adjusts strengths of connections between neurons
- Figures out which connections produce errors
- Changes erroneous connections
- Successively changes connections until desired output is reached

(3.2.3) Evolutionaries

The evolutionaries believe in designing algorithms that learn by simulating natural selection on a computer. This technique is different from backpropagation because it does not only adjust parameters. This technique simulates mating and evolution. Thus, the evolutionaries' master algorithm is genetic programming.

In the 19th century, Charles Darwin was the first to invent the algorithm for evolution. In 1953, a key sub-routine was added to the algorithm through the discovery of the double helix structure by James Watson and Francis Crick. Then, came John Holland, the pioneer of evolutionary computing [45]. John Holland created the fitness function which is the key input to a genetic algorithm. The fitness function encapsulates the human's role in the process of selective breeding. To better understand the fitness function, consider the case in which programs are being evolved to diagnose patients. One program correctly diagnoses 60% of the patients in the database. Another correctly diagnoses 55% of the patients. In this case, a possible fitness function is the fraction of correctly diagnosed cases.

In addition to the fitness function, the concept of reproduction is implemented in genetic algorithms by implementing crossovers. In nature, when a crossover occurs, genetic material is swapped between mother and father resulting in two chromosomes. To better understand the implementation of the fitness function and the executing crossovers, consider the case in which a rule is being evolved to filter spam email. In this case, the training data is composed of 10,000 different words and each candidate rule is represented by a string of 20,000 bits. The first bit corresponds to email containing word *free* (if it matches rule equal to 1, otherwise 0). The second bit corresponds to email not containing word *free* (if it matches rule equal to 1, otherwise 0). If both bits are 1, then rule has no condition on the word *free*. If both bits are 0, then no email matches the rule and all of them get through spam filter. To match a rule an email must contain the entire pattern of present and absent words. Also, the rule's fitness function is the percentage of e-mails it classifies correctly. Then, by using the fitness rule the rule is evolved as follows:

- Start with a population of random strings (generation 0)
- Pick the fittest rules and cross them over
- Repeat process until the desired rule fitness is obtained

From another point of view, in 1987, John Koza proved the method of genetic programming to be preferable over genetic algorithms [36]. If programs are seen as a tree of subroutine calls then, genetic programming allows for the evolution of complete programs by crossing over their subtrees.

(3.2.4) Bayesians

The bayesians believe in designing algorithms that learn through probabilistic inference. By implementing probabilistic inference, they may decide how to incorporate new evidence into previous view. The Bayesians' master algorithm is Bayes' theorem and its derivatives. Bayes' theorem was named after Reverend Thomas Bayes who used conditional probability to provide an algorithm that uses evidence to calculate limits on an unknown parameter [46]. Then, Pierre-Simon Laplace used conditional probability to formulate the relation of an updated posterior probability from a prior probability, given evidence [46]. Finally, Sir Harold Jeffrey put Bayes' algorithm and Laplace's formulation on an axiomatic basis. Bayes' theorem may be expressed through the following equation:

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

In the previously shown equation, A and B represent two event and P(B) is not equal to zero. Also, P(A | B) refers to the probability of event A occurring given that the event B has occurred and P(B | A) refers to the probability of event B occurring given that the event A has occurred. Note that P(A) and P(B) are the probabilities of event A and event B occurring, respectively.

Bayes' theorem focuses on updating the degree of belief in a hypothesis given evidence. If the evidence is consistent with hypothesis, the probability of the hypothesis goes up. Otherwise, the probability of the hypothesis goes down. Consequentially, learning is just another application of the Bayes' theorem where models could be seen as the hypotheses and data as the evidence. As more data is observed, some models become more likely than others. Consider the case of

diagnosing a disease. The different possible diagnoses for a patient may be seen as the different hypotheses to be evaluated. Then, the results from multiple tests may be seen as the pieces of evidence.

By applying Bayes' theorem in machine learning, the Naïve Bayes classifier was devised. This type of learner uses Bayes' theorem and all the effects are independent given the cause. However, in 1913, the Russian mathematician Andrei Markov applied probability to poetry by using Bayes' theorem and created the Markov chain [47]. By using the Markov chain, the probability of each letter depended on the letter immediately preceding it. For example, vowels and consonant tend to alternate. If you see consonant, the next letter will most likely be a vowel. Then, if $Vowel_i$ is a boolean variable, its value will be true if the i-th letter of the poem is a vowel, and false if the i-th letter is a consonant.

A few years later, the topic of Markov chains evolved into Markov networks which have been found to be useful in many areas such as computer vision [48]. Markov networks involves the use of a set of features and corresponding weights which together define a probability distribution. They can be represented by graphs with undirected arcs. To better understand how Markov networks could be used consider the Spotify music streaming service. Spotify possibly has a large set of handcrafted features to help select songs to play for you. By using a Markov network, the corresponding feature of each type of music goes up or down depending on whether you like that type of music or not. Note that the features can be learned through hill climbing and the weights can be learned through gradient descent.

(3.2.5) Analogizers

The analogizers recognize the similarities between situations and thereby inferring other similarities. There is great potential behind analogy. It has inspired many of history's greatest scientific advances such as: the theory of natural selection (inspired by Malthus's *Essay on Population*), Bohr's model of the atom (inspired by miniature solar system), and the ring shape of benzene molecules (inspired by snake eating its own tail). One could argue that human cognition in its entirety is a fabric of analogies.

The first algorithm to represent analogical reasoning was devised by Evelyn Fix and Joe Hodges. Fix and Hodges called this algorithm the nearest neighbor [36]. The nearest neighbor algorithm consists on using the models that are implicitly formed by data. This type of learning is sometimes called: "Lazy learning" [36]. To better understand how the lazy learner works, consider the problem in which we want a model to learn to define what a face should be. If the decision tree algorithm is used then, the problem would turn out to be very complex since faces could appear in different expressions, poses, and lighting conditions. On the other hand, if the nearest neighbor algorithm is used then, the following shortcut is used: If the image in the database that is most similar to the uploaded image is a face then, the uploaded image is a face as well. As it may be noted, the lazy learner forms local models instead of global ones and for this reason, it is faster than other algorithms.

A more robust version of the nearest neighbor algorithm is K-nearest-neighbor [49]. In this case, given a data set, k nearest neighbors will vote for a decision. For example, consider the case of deciding if an uploaded image is a face. If the nearest-neighbor to the new upload is a face and the next two nearest aren't then, the final decisions is that the upload is not a face. An advantage of the k-nearest-neighbor is that it will only make wrong decisions when the majority of the nearest neighbors are noisy. On the down side, the algorithm's vision will be blurrier. Note that with k-nearest neighbor, k is directly proportional to the bias and inversely proportional to the variance.

Until mid-1990s, the nearest-neighbor was the most widely used analogical learner. Then, a new similarity-based algorithm emerged; the support vector machine (SVMs). The concept of SVMs was introduced by Vladimir Vapnik, a Soviet frequentist employed at Bell labs [50]. SVMs are similar to the weighted k-nearest-neighbor. It establishes a frontier between the positive and negative classes. Then, the frontier is defined by a set of examples and their weights. A test example belongs to the positive class if, on average, it looks more like the positive examples than the negative ones. The average is weighted, and the SVM only remembers the key examples required to pin down the frontier. Note that SVMs are the analogizers' master algorithm.

References

[5] A. Nadeem, "The organized chaos of programming language design", *Medium*, 2018. [Online]. Available: https://medium.com/coinmonks/the-organized-chaos-of-programming-language-design-1e0a95067afb.

[6]C. Beyer, "A Brief Totally Accurate History Of Programming Languages", *Medium*, 2019. [Online]. Available: https://medium.com/commitlog/a-brief-totally-accurate-history-of-programming-languages-d2e2b09553f8.

[7]"Computer programming history", *Computerhope.com*, 2020. [Online]. Available: https://www.computerhope.com/history/programming.htm.

[8]A. Ferguson, "A History of Computer Programming Languages", *Cs.brown.edu*, 2004. [Online]. Available: https://cs.brown.edu/~adf/programming_languages.html.

[9]J. Veisdal, "The Unparalleled Genius of John von Neumann", *Medium*, 2019. [Online]. Available: https://medium.com/cantors-paradise/the-unparalleled-genius-of-john-von-neumann-791bb9f42a2d.

[10]M. Hoyle, "History of Computing Science: John von Neumann", *Eingang.org*, 2006. [Online]. Available: http://www.eingang.org/Lecture/neumann.html.

[11]"Software & Languages | Timeline of Computer History | Computer History Museum", *Computerhistory.org*. [Online]. Available: https://www.computerhistory.org/timeline/software-languages/.

[12]"Interactive: The Top Programming Languages", *IEEE Spectrum: Technology, Engineering, and Science News*, 2019. [Online]. Available: https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2019.

[13]I. Anjum, "Lecture 01 introduction to compiler", *Slideshare.net*, 2015. [Online]. Available: https://www.slideshare.net/IffatAnjum/lecture-01-introduction-to-compiler.

[14]T. Mogensen, *Basics of compiler design*. [Erscheinungsort nicht ermittelbar]: [Verlag nicht ermittelbar], 2010.

[15]M. Zahran, "Lecture 4: Lexical Analysis", *Cs.nyu.edu*, 2011. [Online]. Available: https://cs.nyu.edu/courses/spring11/G22.2130-001/lecture4.pdf.

[16]"Compiler Design - Regular Expressions", *Tutorialspoint.com*. [Online]. Available: https://www.tutorialspoint.com/compiler_design/compiler_design_regular_expressions.htm.

[17]"Compiler Design - Syntax Analysis", *Tutorialspoint.com*. [Online]. Available: https://www.tutorialspoint.com/compiler_design/compiler_design_syntax_analysis.htm.

[18]M. Halfeld, "Context-Free Grammars (CFG)", *Univ-orleans.fr*. [Online]. Available: https://www.univ-orleans.fr/lifo/Members/Mirian.Halfeld/Cours/TLComp/l3-CFG.pdf.

[19]J. Farrell, "Compiler Basics", *Cs.man.ac.uk*, 1995. [Online]. Available: http://www.cs.man.ac.uk/~pjj/farrell/compmain.html.

[20]P. Sharma, "Intermediate Code Generation in Compiler Design - GeeksforGeeks", *GeeksforGeeks*. [Online]. Available: https://www.geeksforgeeks.org/intermediate-code-generation-in-compiler-design/.

[21]O. Hansha, "On the Origin of “Hello, World!”", *Medium*, 2018. [Online]. Available: https://medium.com/@ozanerhansha/on-the-origin-of-hello-world-61bfe98196d5.

[22]"Data Structures - Algorithms Basics", *Tutorialspoint.com*. [Online]. Available: https://www.tutorialspoint.com/data_structures_algorithms/algorithms_basics.htm.

[23]S. Bhatt, "Characteristics of an Algorithm", *Medium*, 2019. [Online]. Available: https://medium.com/@bhattshlok12/characteristics-of-an-algorithm-49cf4d7bcd9.

[24]A. Dehal, "Introduction to Time Complexity of Algorithms.", *Medium*, 2018. [Online]. Available: https://medium.com/@bw99214/time-complexity-of-the-algorithms-5a3dae1cb873.

[25]"Greedy Algorithm", *Studytonight.com*. [Online]. Available: https://www.studytonight.com/data-structures/greedy-algorithm.

[26]T. Cormen and D. Balkcom, "Divide and conquer algorithms", *khanacademy.org*. [Online]. Available: https://www.khanacademy.org/computing/computer-science/algorithms/merge-sort/a/divide-and-conquer-algorithms.

[27]"Data Structures - Dynamic Programming", *Tutorialspoint.com*. [Online]. Available: https://www.tutorialspoint.com/data_structures_algorithms/dynamic_programming.htm.

[28]"Introduction to Dynamic Programming 1", *HackerEarth*. [Online]. Available: https://www.hackerearth.com/practice/algorithms/dynamic-programming/introduction-to-dynamic-programming-1/tutorial/.

[29]M. Zaveri, "An intro to Algorithms: Searching and Sorting algorithms", *Medium*, 2018. [Online]. Available: https://codeburst.io/algorithms-i-searching-and-sorting-algorithms-56497dbaef20.

[30]"Data Structure and Algorithms Binary Search", *Tutorialspoint.com*. [Online]. Available: https://www.tutorialspoint.com/data_structures_algorithms/binary_search_algorithm.htm.

[31]"Data Structure - Interpolation Search", *Tutorialspoint.com*. [Online]. Available: https://www.tutorialspoint.com/data_structures_algorithms/interpolation_search_algorithm.htm.

[32]"Data Structure - Sorting Techniques", *Tutorialspoint.com*. [Online]. Available: https://www.tutorialspoint.com/data_structures_algorithms/sorting_algorithms.htm.

[33]"Introduction to Data Structures and Algorithms | Studytonight", *Studytonight.com*. [Online]. Available: https://www.studytonight.com/data-structures/introduction-to-data-structures.

[34]U. Fayyad, N. Weir and S. Djorgovski, "Automated Cataloging and Analysis of Sky Survey Image Databases: the SKICAT System", *Citeseerx.ist.psu.edu*, 1993. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.502.8055.

[35]"Robot Scientist 'Eve' could boost search for new drugs", *manchester.ac.uk*, 2015. [Online]. Available: https://www.manchester.ac.uk/discover/news/robot-scientist-eve-could-boost-search-for-new-drugs/.

[36]P. Domingos, *The Master Algorithm*. Basic Books, 2015.

[37]I. Newton, "Four Rules of Scientific Reasoning from Principia Mathematica", *Apex.ua.edu*. [Online]. Available: https://apex.ua.edu/uploads/2/8/7/3/28731065/four_rules_of_reasoning_apex_website.pdf.

[38]J. Brownlee, "Parametric and Nonparametric Machine Learning Algorithms", *Machine Learning Mastery*, 2019. [Online]. Available: https://machinelearningmastery.com/parametric-and-nonparametric-machine-learning-algorithms/.

[39]J. Calbet, "The Hebb´s rule explained with an analogy.- NeuroQuotient", *Neuroquotient*, 2018. [Online]. Available: https://neuroquotient.com/en/pshychology-and-neuroscience-hebb-principle-rule/.

[40]"The Neuron", *Brainfacts.org*, 2012. [Online]. Available: https://www.brainfacts.org/brain-anatomy-and-function/anatomy/2012/the-neuron.

[41]A. Chandra, "McCulloch-Pitts Neuron — Mankind's First Mathematical Model Of A Biological Neuron", *Medium*, 2018. [Online]. Available: https://towardsdatascience.com/mcculloch-pitts-model-5fdf65ac5dd1.

[42]J. Loiseau, "Rosenblatt's perceptron, the first modern neural network", *Medium*, 2019. [Online]. Available: https://towardsdatascience.com/rosenblatts-perceptron-the-very-first-neural-network-37a3ec09038a.

[43]M. Arbib, "Review of 'Perceptrons: An Introduction to Computational Geometry' (Minsky, M., and Papert, S.; 1969)", *IEEE Transactions on Information Theory*, vol. 15, no. 6, pp. 738-739, 1969. Available: 10.1109/tit.1969.1054388.

[44]S. Kostadinov, "Understanding Backpropagation Algorithm", *Medium*, 2019. [Online]. Available: https://towardsdatascience.com/understanding-backpropagation-algorithm-7bb3aa2f95fd.

[45]M. Dennis, "John Henry Holland | American mathematician", *Encyclopedia Britannica*. [Online]. Available: https://www.britannica.com/biography/John-Henry-Holland.

[46]T. Sarkar, "Bayes' rule with a simple and practical example", *Medium*, 2020. [Online]. Available: https://towardsdatascience.com/bayes-rule-with-a-simple-and-practical-example-2bce3d0f4ad0.

[47]B. Hayes, "First Links in the Markov Chain", *American Scientist*, 2013. [Online]. Available: https://www.americanscientist.org/article/first-links-in-the-markov-chain.

[48]K. Vala, "Markov Networks: Undirected Graphical Models", *Medium*, 2019. [Online]. Available: https://towardsdatascience.com/markov-networks-undirected-graphical-models-dfb19effd8cb.

[49]L. Peterson, "K-nearest neighbor", *Scholarpedia*, vol. 4, no. 2, p. 1883, 2009. Available: 10.4249/scholarpedia.1883 [Accessed 4 July 2020].

[50]"Vladimir Vapnik", *The Franklin Institute*, 2012. [Online]. Available: https://www.fi.edu/laureates/vladimir-vapnik. [Accessed: 04- Jul- 2020].