Introduction

We can view programming as an indirect problem solving method where instead of solving the problem yourself you use an agent to solve the problem. Agents however may be different. In particular we can envision three possible types of agents; experts, assistants and zombies. The nature of the solution then depends on the type of agent at your disposal. This leads to two main programming paradigms; declarative and imperative programming paradigms. Below we present a brief description of the two paradigms and the way they have been operationalised as programming paradigms and their corresponding languages.

Declarative paradigm

For the expert, the solution constitutes specifying what you want. The expert knows how to provide the result. For example, suppose you learn that you have a sibling somewhere in South America and you want to contact your sibling. You may contact expert trackers and describe any details you have of your sibling. The expert tracker then figures out how to contact your sibling. All the tracker needs is sufficient details (specifications) of your sibling to ensure that they know when they have found the right person. From the trackers point of view this is just a search problem. All that is needed is to organize for some searching activity. The search itself may lead to success or failure.

For the assistant, you will need to have the overall plan of contacting your sibling. The plan will outline transformational activities that may progressively advance towards the final goal of contacting your sibling. Your assistant will carry out the transformational activities. Typically for each transformational activity, you specify the entry stage of that activity, and the exit stage of that activity. For example you might tell the assistant to start from Nairobi and move to Rio. The transformational activity is move. The entry point is Nairobi. The exit point is Rio. Your assistant follows the collection of transformational activities which constitute your plan.

Note that an assistant can specialise in one type of transformational activity, and then have assistants for other types of transformational activities. Alternatively you can have several assistants each specialized in a particular activity.

Programming style where an expert or an assistant is assumed is referred to as declarative programming. This is because in this style the solutions are specified in terms of declarations which describe required solution or functions which specify the required transformations, and not the commands needed for the solution. Assuming an expert leads to a programming style that aligns closely to logic and hence we have logic programming paradigm, while assuming an assistant leads to a style that aligns closely to mathematical functions and hence we have functional programming paradigm. When there are several assistants the style is closely aligned to concurrent programming.

Imperative paradigm

For the zombie, you will also need to detail each and every transformational activity and in the right sequence. For example to move from your location to the airport, you might give the following instructions or commands and in order:

1. Put your passport in your pocket.
2. Put you clothes in the travelling bag.
3. Call for taxi.
4. When the taxi comes, enter the taxi when the door is open.
5. Tell the taxi driver to take you to the international airport etc

In the above list of commands, the zombie is given step by step instruction for literally everything to be done. The actual instructions will of course depend on the language the zombie understands. The instructions must also be in the right order. This order is usually referred to as the control flow. Thus for a zombie we need to give not only commands, but also the control flow.
For example if the zombie calls for the taxi and zooms off to the airport, they cannot now put clothes in the travelling bag as they would have neither the bag nor the clothes at the airport.
Programming style where zombies are assumed is referred to as imperative programming paradigm. This is because in this style, solutions are specified by imperatives (commands or instructions) and the control flow.

Declarative Programming paradigms
The two most established programming paradigms are; functional programming paradigm and logic programming paradigm.

Functional programming Paradigm
In functional programming, programs are expressed as mathematical functions and program execution is regarded as function evaluation. Around 1936 Church Alonso provided a model of computation based on function evaluation which was named the Lambda Calculus. In this model, program flow is implicitly expressed by combining function calls.
Functions
There are two mathematical views of functions; the extensional view and the intensional view.
Extensional view
In this view each function f has a domain X and codomain Y , and a function $f : X \to Y$ is a set of pairs $f \subseteq X \times Y$ such that for each $x \in X$, there exists exactly one $y \in Y$ such that $(x, y) \in f$. Two functions $f, g : X \to Y$ are considered equal if they yield the same output on each input, i.e., $f(x) = g(x)$ for all $x \in X$. This is called the extensional view of functions, because it specifies that the only thing observable about a function is how it maps inputs to outputs.
Intensional view
In this view, a function is given by a rule for how the function is to be calculated. Often, such a rule can be given by a formula, for instance, the familiar $f(x) = x^2$ or $g(x) = \sin(e^x)$ from calculus. As before, two functions are extensionally equal if they have the same input-output behavior; but in addition they have another notion of equality: two functions are intensionally equal if they are given by (essentially) the same formula.

Lambda calculus
The lambda calculus is a theory of functions as formulas. It is a system for manipulating functions as expressions.
An expression language is needed for stating expressions. One such language of expressions is arithmetic. Arithmetic expressions are made up from variables (x, y, z . . .), numbers (1, 2, 3, . . .), and operators ("+", "−", "×" etc.). An expression such as x + y stands for the result of an addition (as opposed to an instruction to add, or the statement that something is being added).
The great advantage of this language is that expressions can be nested without any need to mention the intermediate results explicitly. So for instance, we write: $A = (x + y) \times z^2$ and not: let $w = x + y$, then let $u = z^2$, then let $A = w \times u$.
The lambda calculus extends the idea of an expression language to include functions. Where we normally write: Let f be the function $x \to x^2$. Then consider $A = f(5)$, in the lambda calculus we just write $A = (\lambda x.x^2)(5)$.

The expression $\lambda x.x^2$ stands for the function that maps x to x^2 (as opposed to the statement that x is being mapped to x^2). As in arithmetic, parentheses is used to group terms. The variable x is a local variable in the term $\lambda x.x^2$. Thus, it does not make any difference if we write $\lambda y.y^2$ instead. A local variable is also called a bound variable.

Higher order functions

One advantage of the lambda notation is that it allows higher-order functions, i.e., functions whose inputs and/or outputs are themselves functions. For example the operation $f \rightarrow f \circ f$ in mathematics, takes a function f and maps it to $f \circ f$, the composition of f with itself. In the lambda calculus, $f \circ f$ is written as $\lambda x.f\ (f\ (x))$, and the operation that maps f to $f \circ f$ is written as $\lambda f.\lambda x.f\ (f\ (x))$.

Logic programming Paradigm

Logic programming is closely aligned to mathematical logic. Over time, mathematical logic has developed into categories of logic systems based on underlying assumptions on the meaning of statements in the logic system. The logic systems include; propositional logic, first order logic, higher order logic, multi valued logic, fuzzy logic etc. Logic programming is currently based on first order logic. For this reason only the propositional logic to introduce truth value semantics and first order logic will be discussed.

Propositional logic

In propositional logic, statements are declarations about some state of the world. If a statement corresponds with the state of the world it is said to be true, otherwise it is said to be false. Thus we say the statement evaluates to true or false. The set {true, false} is said to provide some kind of meaning to statements in propositional logic (albeit a correspondence meaning). This set is said to provide the semantics for the logic statements. The value of each statement is called a truth value even though it can be true or false. A Statement is said to be atomic if it cannot be broken down into simpler statements that can be assigned truth values.

For convenience, atomic statements are usually designated by letters such as p, y, z, etc. For example the statement “it is raining” could be designated as r. If we cannot see any rain then r is false, otherwise r is true. Thus if we designate the truth value of r as V(r) we can write V(r) = {true, false} or V(r) = {T, F} where T designates true and F designates false.

Statements can be composed to form compound statements. The composition operators then provide the rules of assigning truth values to the compound statements. Examples of operators are the AND and the OR operators. For example the statement it is raining AND it is Monday is a compound statement of two atomic statements “it is raining” and “it is monday” composed with the AND operator. The truth value of this statement can be derived or inferred by applying the composition rules for assigning truth values for the AND operator. The rules are usually specified using a truth table as shown below.

r	**s**	**AND(r,s)**
F	F	F
F	T	F
T	F	F
T	T	T

AND truth table

To generate the table we consider all possible combinations of the truth values of the atomic statement. For each possible combination the AND operator assigns a correspond truth value.
Propositional logic assumes the world contains only facts which are either true or false. This leads to the following disadvantages:

- We can't directly talk about properties of individuals or relations between individuals (e.g., "Bill is tall", "Mary and John are siblings")
- Generalizations, patterns, regularities can't easily be represented (e.g., "all triangles have 3 sides").

First order logic aka predicate calculus is instead used.

First order logic

First-order logic (like natural language) assumes the world contains the following:

- Objects: people, houses, numbers, colors, baseball games, wars, etc
- Relations: red, round, prime, brother of, bigger than, part of, comes between, etc
- Functions: father of, best friend, one more than, plus, etc

We can now express:

- Objects in the world, eg Mary , chair, desk, etc
- Properties of objects eg Mary is tall i.e. tall(Mary) etc. This can be true or false.
- Relations between objects eg this is Mary's desk i.e. This(Mary, desk) etc. This relation can be true or false. The properties of objects or relations between objects are referred to as predicates. They evaluate to truth values.
- Functions that take objects and give other objects e.g add(3,2) = 5. add is a function that takes objects 3 and 2 and returns object 5.
- Just like in propositional logic, we can compose predicates with composition operators such as AND and OR. eg Mary is tall and is the owner of this desk ; tall(Mary) AND owner(Mary, desk)
- We can generalize or specialize over objects by using the existential quantifier ∃ and the universal quantifier ∀ e.g "There is only one cow present" can be expressed as $\exists y, Cows(y) \wedge Present(y)$. "All the cows are present" can be expressed as $\forall x, Cows(x) \wedge Present(x)$. A variable such as x introduced by a quantifier is said to be bound by the quantifier. It is possible for variables also to be free, that is, not bound by any quantifier.

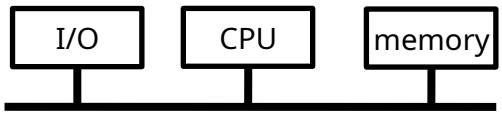
In predicate calculus, arguments to predicates and functions can only be terms—that is, combinations of variables, constants, and functions. Terms cannot contain predicates, quantifiers, or connectives i.e. operators.
Statements that are assumed to be true are called axioms. The set of logical statements that are taken to be axioms can be viewed as the logic program, and the statement or statements that are to be derived can be viewed as the input that initiates the computation. Such inputs are also provided by the programmer and are called queries or goals.

Imperative programming Paradigm

While the declarative programming paradigm is inspired by mathematics, imperative programming paradigm is inspired by the underlying machine. Perhaps the most influential machine in this regard is the Von-Neumann machine.

Von Neumann machine

The Von Neumann machine is a computing machine consisting of memory, input/output and a processing element also called the CPU. These elements are interconnected by a bus that routes information between the elements.



Von Neumann Machine

A program consists of instructions, which are composed of operators which are applied to operands. The operand may be memory locations such as M1, M2 etc; CPU registers (stores) designated R1, R2 etc, or integers such as 20, 30, etc and characters such as a, b, D, F, etc. The instructions are stored in memory. An example of an instruction is mov(M1, 20); which moves integer 20 to a memory location M1. This is actually carried out by:

1. fetching the instruction from memory to CPU
2. decoding the instruction
3. executing the instruction
4. updating memory with the result of execution

Programming at this level however is too detailed for humans to write large programs. Thus hierarchies and abstractions are employed to create higher level languages such as C/C++, Java, etc. With high level language abstractions, programs constitute of variables, operators and values or constants, which are related to the Von Neumann machine as follows:

1. memory locations are modeled as variables x,y, S, U, etc. They hold typed values.
2. operands are modeled by mathematical operators such as +, AND, *, etc.
3. moving data from one location to another is modeled by assignment operator (=)

A program now consists of a sequence of statements. A statement can be a declaration, a definition or command. The sequence defines the control flow of the program.

Declarations

Declarations serve to introduce a variable and the type of value the variable will hold. For example:

int x, y ; introduces x and y as variables that will hold integer values.

Definitions

We define the variables by putting values in them. For example:

int n = 10 ; int m = 50; defines integer variables n and m.

Commands

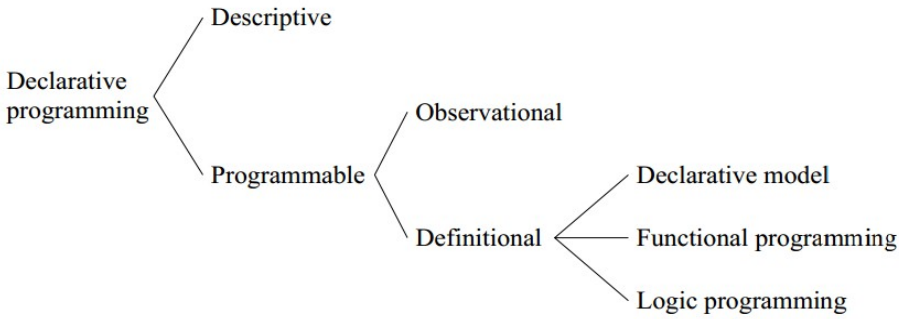
Commands execute the operations specified by the operators and then update memory with the results. The memory updating is usually designated by the assignment operator (=). For example n = m; copies the value of m in n when executed so that m will have 50 and n will have 50. The original value of n will be permanently destroyed (i.e. overwritten).

Control flow.

The assignment operator updates memory in a destructive way. The value of a variable depends on the most recent update. Therefore the control flow must be defined, to ensure that updates on variables are in the right order. For example if we have the statements in order:
x = 3 * 3;
x = 3+3;
The result for x is 6. If we reverse the order so that we have:
x = 3+3;
x = 3 * 3;
The result now is 9.

Programming languages
Corresponding to the programming paradigms, we have two classes of languages; declarative languages and imperative languages.

Declarative languages
Declarativeness of a language can be classified along several dimensions as shown in the classification diagram shown below:



The first level of classification is based on the expressiveness. There are two possibilities:

A descriptive declarativeness.
This is the least expressive. The declarative "program" just defines a data structure. Examples of such languages are a formatting language like HTML, which gives the structure of a document without telling how to do the formatting, or an information exchange language like XML, which is used to exchange information in an open format that is easily readable by all. The descriptive level is too weak to write general programs. But it consists of data structures that are easy to calculate with. The records of HTML and XML documents, can be created and transformed easily by a program.

A programmable declarativeness.
This is as expressive as a Turing machine. There are two fundamentally different ways to view programmable declarativeness:
An observational view, where declarativeness is a property of the component interface. The observational view follows the principle of abstraction: that to use a component it is enough to know its specification without knowing its implementation. The component just has to behave declaratively, i.e., as if it were independent, stateless, and deterministic, without necessarily being written in a declarative computation model.

The observational view lets us use declarative components in a declarative program even if they are written in a non-declarative model. For example, a database interface can be a valuable addition to a declarative language. Yet, the implementation of this interface is almost certainly not going to be logical or functional. It suffices that it could have been defined declaratively. Sometimes a declarative component will be written in a functional or logical style, and sometimes it will not be.
A definitional view, where declarativeness is a property of the component implementation. For example, programs written in the declarative model are guaranteed to be declarative, because of properties of the model. Two styles of definitional declarative programming have become particularly popular: the functional style exemplified by languages such as Scheme, Haskell, ML etc and the logical style exemplified by Prolog.

Imperative languages
While there are varying opinions on the difference between declarative and imperative programming, the concept of state variable and mathematical variable seems to provide more effective distinction.
The declarative concept of a variable is the *mathematical* concept of an *unknown* that is given meaning by substitution. The imperative concept of a variable, arising from low-level machine models, is instead given meaning by assignment (mutation), although, allowed to appear in expressions in a way that resembles that of a proper variable. It is actually a state variable. Imperative languages are therefore languages that embody state variables as an essential part of the language semantics. Examples of such languages include assemblers, C/C++, Java etc.
The language Rust which we shall be using can be regarded as a multiparadigm language that allows the programmer the option to mix both declarative and imperatives styles of programming.