Programming Paradigm

T2

By Archana Chaudhari

Text Book: T2. Sebesta R., "Concepts of Programming Languages", 4th Edition, Pearson Education, ISBN- 81-7808-161-X.

Introduction to Programming Paradigm

- A programming paradigm can be understood as an **abstraction of a computer system**, for example the von Neumann model used in traditional sequential computers.
- For **parallel computing**, there are many possible models typically reflecting different ways processors can be interconnected to communicate and share information.
- In **object-oriented programming**, programmers can think of a program as a collection of interacting objects, while in **functional programming** a program can be thought of as a sequence of stateless function evaluations.
- In **process-oriented programming,** programmers think about applications as sets of concurrent processes acting upon shared data structures.

Processing Paradigms

- A programming paradigm can be understood as an **abstraction** of a computer system, who is based on a certain **processing model** or paradigm.

- Nowadays, the prevalent computer processing model used is the **von Neumann model**, invented by John von Neumann in 1945, influenced by Alan Turing's "Turing machine".
  - Data and program are residing in the **memory**.
  - **Control unit** coordinates the components sequentially following the program's instructions.
  - **Arithmetic Logical Unit** performs the calculations.
  - Input/output provide interfaces to the exterior.

- The **program** and its **data** are what is **abstracted** in a programming language and translated into machine code by the compiler/interpreter.

Introduction to four main Programming paradigms

Introduction to four main Programming paradigms

- Procedural
- Object oriented
- Functional
- Logic and rule based

Procedural programming

1. Procedural Programming

- Procedural programming can also be referred to as imperative programming.
- It is a programming paradigm based upon the concept of procedure calls, in which statements are structured into procedures (also known as subroutines or functions).
- They are a list of instructions to tell the computer what to do step by step, Procedural programming languages are known as top-down languages.
- Procedures are convenient for making pieces of code written by different people or different groups, including through programming libraries.
  - specify a simple interface
  - self-contained information and algorithmic
  - reusable piece of code
- Most of the early programming languages are all procedural.
- Examples of Fortran C and Cobol etc.

Procedural programming

- The earliest imperative languages were the machine languages of the original computers. In these languages, instructions were very simple, which made hardware implementation easier, but hindered the creation of complex programs.
- FORTRAN (1954) was the first major programming language to remove through **abstraction** the obstacles presented by machine code in the creation of complex programs.
- FORTRAN was a compiled language that allowed named variables, complex expressions, subprograms, and many other features now common in imperative languages.
- In the late 1950s and 1960s, ALGOL was developed in order to allow mathematical algorithms to be more easily expressed.
- In the 1970s, Pascal was developed by Niklaus Wirth, and C was created by Dennis Ritchie.
- For the needs of the United States Department of Defense, Jean Ichbiah and a team at Honeywell began designing Ada in 1978.

Features of Procedural Code

- Procedural Programming is excellent for general-purpose programming
- The coded simplicity along with ease of implementation of compilers and interpreters
- A large variety of books and online course material available on tested algorithms, making it easier to learn.
- The source code is portable
- The code can be reused in different parts of the program, without the need to copy it
- The program flow can be tracked easily as it has a top-down approach.

Object-oriented programming

programming paradigms

Object-oriented programming

- Object-oriented programming (OOP) is a programming paradigm that uses "objects" – data structures encapsulating data fields and procedures together with their interactions – to design applications and computer programs.

- Associated programming techniques may include features such as data **abstraction**, **encapsulation**, **modularity**, **polymorphism**, and **inheritance**.

- Though it was invented with the creation of the **Simula** language in 1965, and further developed in **Smalltalk** in the 1970s, it was not commonly used in mainstream software application development until the early 1990s.

- Many modern programming languages now support OOP.

OOP concepts: class

- A class defines the abstract characteristics of a thing (object), including that thing's **characteristics** (its attributes, fields or properties) and the thing's **behaviors** (the operations it can do, or methods, operations or functionalities).
- One might say that a class is a blueprint or factory that describes the nature of something.
- Classes provide **modularity** and **structure** in an object-oriented computer program.
- A class should typically be recognizable to a non-programmer familiar with the **problem domain**, meaning that the characteristics of the class should make sense in context. Also, the code for a class should be relatively self-contained (generally using encapsulation).
- Collectively, the properties and methods defined by a class are called its **members**.

OOP concepts: object

- An object is an individual of a class created at run-time trough object **instantiation** from a class.
- The set of values of the attributes of a particular object forms its **state**. The object consists of the **state** and the **behavior** that's defined in the object's class.
- The object is instantiated by implicitly calling its constructor, which is one of its member functions responsible for the creation of instances of that class.

OOP concepts: attributes

- An **attribute**, also called data member or member variable, is the data encapsulated within a class or object.
- In the case of a regular field (also called **instance variable**), for each instance of the object there is an instance variable.
- A static field (also called **class variable**) is one variable, which is shared by all instances.
- Attributes are an object's variables that, upon being given values at instantiation (using a **constructor**) and further execution, will represent the state of the object.
- A class is in fact a data structure that may contain different fields, which is defined to contain the procedures that act upon it. As such, it represents an **abstract data type**.
- In pure object-oriented programming, the attributes of an object are local and cannot be seen from the outside. In many object-oriented programming languages, however, the attributes may be accessible, though it is generally considered bad design to make data members of a class as externally visible.

OOP concepts: method

- A **method** is a subroutine that is exclusively associated either with a class (in which case it is called a **class method** or a static method) or with an object (in which case it is an **instance method**).
- Like a subroutine in procedural programming languages, a method usually consists of a sequence of programming statements to perform an action, a set of input parameters to customize those actions, and possibly an output value (called the return value).
- Methods provide a mechanism for accessing and manipulating the encapsulated state of an object.
- Encapsulating methods inside of objects is what distinguishes object-oriented programming from procedural programming.

OOP concepts: method

- The object-oriented programming paradigm intentionally favors the use of methods for each and every means of access and change to the underlying data:
  - Constructors: Creation and initialization of the state of an object. Constructors are called automatically by the run-time system whenever an object declaration is encountered in the code.
  - Retrieval and modification of state: accessor methods are used to access the value of a particular attribute of an object. Mutator methods are used to explicitly change the value of a particular attribute of an object. Since an object's state should be as hidden as possible, accessors and mutators are made available or not depending on the information hiding involved and defined at the class level
  - Service-providing: A class exposes some "service-providing" methods to the exterior, who are allowing other objects to use the object's functionalities. A class may also define private methods who are only visible from the internal perspective of the object.
  - Destructor: When an object goes out of scope, or is explicitly destroyed, its destructor is called by the run-time system. This method explicitly frees the memory and resources used during its execution.

OOP concepts: inheritance

- Inheritance is a way to compartmentalize and **reuse** code by creating collections of attributes and behaviors (classes) which can be based on previously created classes.
- The new classes, known as **subclasses** (or derived classes), inherit attributes and behavior of the pre-existing classes, which are referred to as superclasses (or ancestor classes). The inheritance relationships of classes gives rise to a hierarchy.
- **Multiple inheritance** can be defined whereas a class can inherit from more than one superclass. This leads to a much more complicated definition and implementation, as a single class can then inherit from two classes that have members bearing the same names, but yet have different meanings.
- **Abstract inheritance** can be defined whereas abstract classes can declare member functions that have no definitions and are expected to be defined in all of its subclasses.

OOP concepts: abstraction

- **Abstraction** is simplifying complex reality by modeling classes appropriate to the problem, and working at the most appropriate level of inheritance for a given aspect of the problem.

- For example, a class Car would be made up of an Engine, Gearbox, Steering objects, and many more components. To build the Car class, one does not need to know how the different components work internally, but only how to interface with them, i.e., send messages to them, receive messages from them, and perhaps make the different objects composing the class interact with each other.

- Object-oriented programming provides **abstraction** through **composition** and **inheritance**.

OOP concepts: encapsulation and information hiding

- **Encapsulation** refers to the bundling of data members and member functions inside of a common “box”, thus creating the notion that an object contains its state as well as its functionalities
- **Information hiding** refers to the notion of choosing to either expose or hide some of the members of a class.
- These two concepts are often misidentified. Encapsulation is often understood as including the notion of information hiding.
- Encapsulation is achieved by specifying which classes may use the members of an object. The result is that each object exposes to any class a certain interface — those members accessible to that class.
- The reason for encapsulation is to prevent clients of an interface from depending on those parts of the implementation that are likely to change in the future, thereby allowing those changes to be made more easily, that is, without changes to clients.
- It also aims at preventing unauthorized objects to change the state of an object.

OOP concepts: polymorphism

- Polymorphism is the ability of objects belonging to **different typ**es to respond to method, field, or property calls of the **same name**, each one according to an appropriate **type-specific behavior**.
- The programmer (and the program) does not have to know the exact type of the object at compile time. The exact behavior is determined at run-time using a run-time system behavior known as **dynamic binding**.
- Such polymorphism allows the programmer to treat derived class members just like their parent class' members.
- The different objects involved only need to present a **compatible interface** to the clients. That is, there must be public or internal methods, fields, events, and properties with the same name and the same parameter sets in all the superclasses, subclasses and interfaces.
- In principle, the object types may be unrelated, but since they share a common interface, they are often implemented as subclasses of the same superclass.

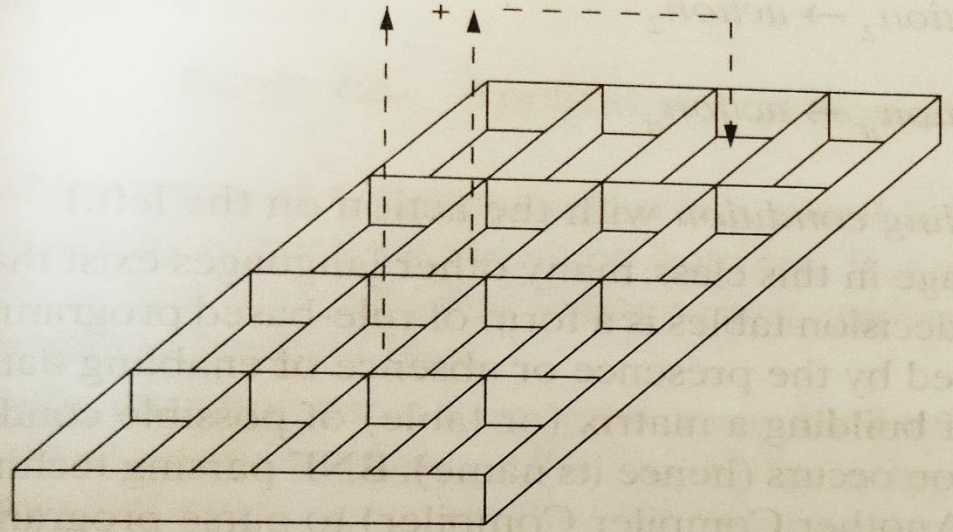
Functional Programming programming paradigm

Functional Programming Paradigm

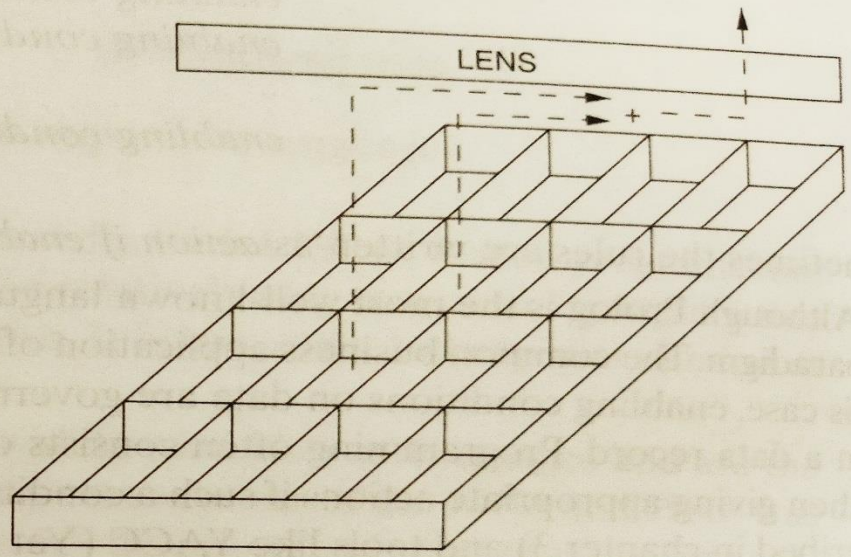
- Functional programming is a programming paradigm in which we try to bind everything in pure mathematical functions style. It **avoids state changes** and **mutable data**.

- It emphasizes the ‘what to solve’ (application of functions), in contrast to the imperative programming style, which emphasizes ‘how to solve’ (changes in state).

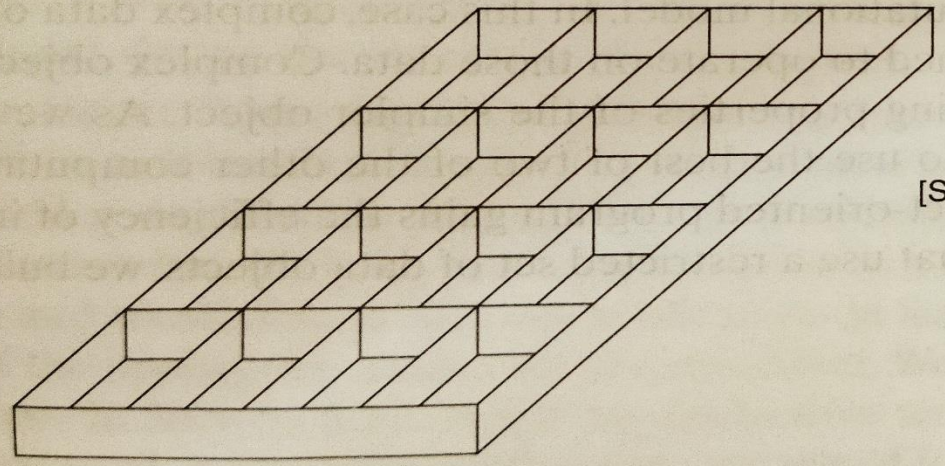
- It uses expressions instead of statements



(a) Imperative languages—
Memory is a set of boxes



(b) Applicative languages—
Change how data is accessed from memory



(c) Rule-based—
Use filters to enable state change

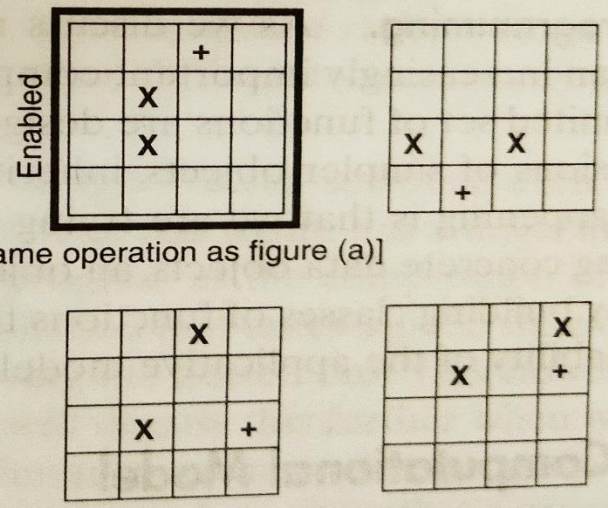


Figure 1.4 Computational models of programming languages.

Functional Programming is based on Lambda Calculus

- Functional programming has its roots in the **lambda calculus**, a formal system developed in the 1930s to investigate function definition, function application, and recursion. Many functional programming languages can be viewed as elaborations on the lambda calculus.

- LISP was the first operational functional programming language.

- Up to this day, functional programming has not been very popular except for a restricted number of application areas, such as artificial intelligence.

- John Backus presented the FP programming language in his 1977 Turing Award lecture "*Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs*".

Programming Languages that support functional programming

- Haskell,
- JavaScript,
- Scala,
- Erlang,
- Lisp,
- ML,
- Clojure,
- OCaml,
- Common Lisp,
- Racket

Concepts of functional programming:

- Pure functions
- Recursion
- Referential transparency
- Functions are First-Class and can be Higher-Order
- Eager vs. Lazy Evaluation
- Variables are Immutable

Functional Programming: Pure Function

- Functional programming consists only of PURE functions.
- These functions have two main properties.
  - First, they always produce the same output for same arguments irrespective of anything else.
  - Secondly, they have no side-effects i.e. they do modify any argument or global variables or output something.
- what do you understand by Pure functions?
  - For example, if a function relies on the global variable or class member's data, then it is not pure. And in such cases, the return value of that function is not entirely dependent on the list of arguments received as input and can also have side effects.
  - what do you understand by the term side effect?
    - A side effect is a change in the state of an application that is observable outside the called **function** other than its return value. For example: Modifying any external variable or object property such as a global variable

Functional Programming: Pure Function

- Later property is called immutability
- Programs done using functional programming are easy to debug because pure functions have no side effect or hidden I/O.
- Pure functions also make it easier to write parallel/concurrent applications.
- Example of pure function:

```
sum(x,y)        //sum is function taking x and y as arguments
  return x+y    //sum is returning sum of x and y without changing them
```

Functional Programming: Recursion

- There are no “for” or “while” loop in functional languages.
- Iteration in functional languages is usually accomplished via **recursion**.
- Recursive functions repeatedly call themselves, until it reaches the base case.
- example of the recursive function:

```
fib(n)
      if (n <= 1)
            return 1;
      else
            return fib(n - 1) + fib(n - 2);
```

Functional Programming: Higher-Order Functions

- Most functional programming languages use **higher-order functions**, which are functions that can either take other functions as arguments or return functions as results.
- Higher-order functions are closely related to **functions as first-class citizen**, in that higher-order functions and first-class functions both allow functions as arguments and results of other functions.
- The distinction between the two is subtle: "higher-order" describes a mathematical concept of functions that operate on other functions, while "first-class" is a computer science term that describes programming language entities that have no restriction on their.

Eg.
```
show_output(f)     // function show_output is declared taking argument f which are another function
        f();                // calling passed function

    print_gfg()             // declaring another function
        print("hello gfg");

    show_output(print_gfg) // passing function in another function
```

Functional Programming: Eager vs. Lazy Evaluation

- Functional languages can be categorized by whether they use strict (**eager**) or non-strict (**lazy**) evaluation, concepts that refer to how function arguments are processed when an expression is being evaluated. Under strict evaluation, the evaluation of any term containing a failing subterm will itself fail. For example, the expression

```
print length([2+1, 3*2, 1/0, 5-4])
```

- will fail under eager evaluation because of the division by zero in the third element of the list. Under lazy evaluation, the length function will return the value 4 (the length of the list), since evaluating it will not attempt to evaluate the terms making up the list.
- Eager evaluation fully evaluates function arguments before invoking the function. Lazy evaluation does not evaluate function arguments unless their values are required to evaluate the function call itself.
- The usual implementation strategy for lazy evaluation in functional languages is **graph reduction**. Lazy evaluation is used by default in several pure functional languages, including **Miranda**, **Clean** and **Haskell**.

Functional Programming: Type Inference

- Especially since the development of **Hindley–Milner type inference** in the 1970s, functional programming languages have tended to use typed lambda calculus, as opposed to the untyped lambda calculus used in Lisp and its variants (such as Scheme).
- **Type inference**, or implicit typing, refers to the ability to deduce automatically the type of the values manipulated by a program. It is a feature present in some strongly statically typed languages.
- The presence of strong compile-time type checking makes programs more **reliable**, while type inference frees the programmer from the need to **manually declare** types to the compiler.
- Type inference is often characteristic of — but not limited to — functional programming languages in general. Many imperative programming languages have adopted type inference in order to improve type safety.

Functional Programming: Variables are Immutable

- In functional programming, we can't modify a variable after it's been initialized.
- Functional programs do not have assignment statements.
- We can create new variables – but we can't modify existing variables, and this really helps to maintain state throughout the runtime of a program.
- Once we create a variable and set its value, we can have full confidence knowing that the value of that variable will never change.
- Eg. x = x + 1 // this changes the value assigned to the variable x.

  // So the expression is not referentially transparent.
- Correct is: z=x+1

Advantages:

1. Pure functions are easier to understand. Their function signature gives all the information about them i.e. their return type and their arguments.
2. It treat functions as values and pass them to functions as parameters make the code more readable and easily understandable.
3. Testing and debugging is easier.
4. It is used to implement concurrency/parallelism because pure functions don’t change variables or any other data outside of it.
5. It adopts lazy evaluation which avoids repeated evaluation because the value is evaluated and stored only when it is needed.

Disadvantage

1. Sometimes writing pure functions can reduce the readability of code.
2. Writing programs in recursive style instead of using loops can be bit intimidating.
3. Writing pure functions are easy but combining them with rest of application and I/O operations is the difficult task.
4. Immutable values and recursion can lead to decrease in performance.

Applications:

- It is used in mathematical computations.
- It is needed where concurrency or parallelism is required.

Logic and Rule based Programming Paradigm

Logic and rule based

- We now examine a radically different paradigm for programming: declarative programming
  - rather than writing control constructs (loops, selection statements, subroutines)
  - you specify knowledge and how that knowledge is to be applied through a series of rules
  - the programming language environment uses one or more built-in methods to reason over the knowledge and prove things (or answer questions)
    - in logic programming, the common approach is to apply the methods of resolution and unification
- While these languages have numerous flaws, they can build powerful problem solving systems with little programming expertise
  - they have been used extensively in AI research

Logic Programming

- Also known as declarative programming.
- A declarative program does not have code, instead it defines two pieces of knowledge
  - facts – statements that are true.
  - rules – if-then statements that are truth preserving.
- In which program statements express facts and rules about problems within a system.
- Rules are written as logical clauses with a head and a body.
- Prolog follows the Logical paradigm and is probably the most famous language in the logical programming family.

Terminology

- Logic Programming is a specific type of a more general class: production systems (also called rule-based systems)
  - a production system is a collection of facts (knowledge), rules (which are another form of knowledge) and control strategies
  - we often refer to the collection of facts (what we know) as working memory
  - the rules are simple if-then statements where the condition tests values stored in working memory and the action (then clause) manipulates working memory (adds new facts, deletes old facts, modifies facts)
  - the control strategies help select among a set of rules that match – that is, if multiple rules have matching conditions, the control strategies can help decide which rule we select this time through
  - there are other control strategies as well – whether we work from conditions to conclusions or from conclusions to conditions (forward, backward chaining respectively)

Background for Logic

- A proposition is a logical statement that is only made if it is true
  - Today is Tuesday
  - The Earth is round
- *Symbolic logic* uses propositions to express ideas, relationships between ideas and to generate new ideas based on the given propositions
- Propositions will either be true (if stated) or something to prove or disprove (determine if it is true) – we do not include statements which are false

Logic Operators

Name	Symbol	Example	Meaning
negation	$\neg$	$\neg$ a	not a
conjunction	$\cap$	a $\cap$ b	a and b
disjunction	$\cup$	a $\cup$ b	a or b
equivalence	$\equiv$	a $\equiv$ b	a is equivalent to b
implication	$\supset$ $\subset$	a $\supset$ b a $\subset$ b	a implies b b implies a
universal	$\forall$ X.P		For all X, P is true
existential	$\exists$ X.P		There exists a value of X such that P is true

- Equivalence means that both expressions have identical truth tables
- Implication is like an if-then statement
  - if a is true then b is true
  - note that this does not necessarily mean that if a is false that b must also be false
- Universal quantifier says that this is true no matter what x is
- Existential quantifier says that there is an X that fulfills the statement

$\forall$X.(woman(X) $\supset$ human(X))
– if X is a woman, then X is a human

$\exists$X.(mother(mary, X) $\cap$ male(X))
– Mary has a son (X)

Example

- Prolog follows the Logical paradigm and is probably the most famous language in the logical programming family.
- Prolog, like SQL, has two main aspects, one to express the data and another to query it. The basic constructs of logical programming, terms, and statements, are inherited from logic.

There are three basic statements:

- **Facts** are fundamental assertions about the problem domain (e.g. "Socrates is a man")
- **Rules** are inferences about facts in the domain (e.g. "All men are mortal.")
- **Queries** are questions about that domain (e.g. "Is Socrates mortal?")

Features of Logical Programming

- Logical programming can be used to express knowledge in a way that does not depend on the implementation, making programs more flexible, compressed and understandable.
- It enables knowledge to be separated from use, i.e. the machine architecture can be changed without changing programs or their underlying code.
- It can be altered and extended in natural ways to support special forms of knowledge, such as meta-level of higher-order knowledge.
- It can be used in non-computational disciplines relying on reasoning and precise means of expression.

Case Study

- A case study: Retail Sales application

Thank You