Chapter 1
Function

PART – 1
I. Choose The Best Answer

Question 1.
The small sections of code that are used to perform a particular task is called ……..
(a) Subroutines
(b) Files
(c) Pseudo code
(d) Modules
Answer:
(a) Subroutines

Question 2.
Which of the following is a unit of code that is often defined within a greater code structure?
(a) Subroutines
(b) Function
(c) Files
(d) Modules
Answer:
(b) Function

Question 3.
Which of the following is a distinct syntactic block?
(a) Subroutines
(b) Function
(c) Definition
(d) Modules
Answer:
(c) Definition

Question 4.
The variables in a function definition are called as ……..
(a) Subroutines
(b) Function
(c) Definition
(d) Parameters

Answer:
(d) Parameters

Question 5.
The values which are passed to a function definition are called ……..
(a) Arguments
(b) Subroutines
(c) Function
(d) Definition
Answer:
(a) Arguments

Question 6.
Which of the following are mandatory to write the type annotations in the function definition?
(a) Curly braces
(b) Parentheses
(c) Square brackets
(d) Indentations
Answer:
(b) Parentheses

Question 7.
Which of the following defines what an object can do?
(a) Operating System
(b) Compiler
(c) Interface
(d) Interpreter
Answer:
(c) Interface

Question 8.
Which of the following carries out the instructions defined in the interface?
(a) Operating System
(b) Compiler
(c) Implementation
(d) Interpreter
Answer:
(c) Implementation

Question 9.
The functions which will give exact result when same arguments are passed are called ……..
(a) Impure functions

(b) Partial Functions
(c) Dynamic Functions
(d) Pure functions
Answer:
(d) Pure functions

Question 10.
The functions which cause side effects to the arguments passed are called ……..
(a) Impure functions
(b) Partial Functions
(c) Dynamic Functions
(d) Pure functions
Answer:
(a) Impure functions

PART – II
II. Answer The Following Questions

Question 1.
What is a subroutine?
Answer:
Subroutines are the basic building blocks of computer programs. Subroutines are small sections of code that are used to perform a particular task that can be used repeatedly. In Programming languages these subroutines are called as Functions.

Question 2.
Define Function with respect to Programming language?
Answer:
A function is a unit of code that is often defined within a greater code structure. Specifically, a function contains a set of code that works on many kinds of inputs, like variants, expressions and produces a concrete output.

Question 3.
Write the inference you get from X: = (78)?
Answer:
Value 78 being bound to the name X.

Question 4.
Differentiate interface and implementation?
Answer:
Interface:
Interface just defines what an object can do, but won't actually do it.

Implementation:
Implementation carries out the instructions defined in the interface.

Question 5.
Which of the following is a normal function definition and which is recursive function definition?
Answer:
(I) Let Recursive sum x y:
return x + y

(II) let disp:
print 'welcome'

(III) let Recursive sum num:
if (num! = 0) then return num + sum (num – 1) else
return num

1. Recursive function
2. Normal function
3. Recursive function

PART – III
III. Answer The Following Questions

Question 1.
Mention the characteristics of Interface?
Answer:
Characteristics of interface:

1. The class template specifies the interfaces to enable an object to be created and operated properly.
2. An object's attributes and behaviour is controlled by sending functions to the object.

Question 2.
Why strlen is called pure function?
Answer:
strlen (s) is called each time and strlen needs to iterate over the whole of 's'. If the compiler is smart enough to work out that strlen is a pure function and that 's' is not updated in the lbop, then it can remove the redundant extra calls to strlen and make the loop to execute only one time. This function reads external memory but does not change it, and the value returned derives from the external memory accessed.

Question 3.
What is the side effect of impure function. Give example?
Answer:
Impure Function:

- The return value of the impure functions does not solely depend on its arguments passed. Hence, if you call the impure functions with the same set of arguments, you might get the different return values. For example, random( ), Date( ).
- They may modify the arguments which are passed to them.

Question 4.
Differentiate pure and impure function?
Answer:
Pure Function:

1. The return value of the pure functions solely depends on its arguments passed.
2. If you call the pure functions with the same set of arguments, you will always get the same return values.
3. They do not have any side effects.
4. They do not modify the arguments which are passed to them.

Impure Function:

1. The return value of the impure functions does not solely depend on its arguments passed.
2. If you call the impure functions with the same set of arguments, you might get the different return values. For example, random( ), Date( ).
3. They have side effects.
4. They may modify the arguments which are passed to them.

Question 5.
What happens if you modify a variable outside the function? Give an example?
Answer:
When a function depends on variables or functions outside of its definition block, you can never be sure that the function will behave the same every time it's called.
For example let y: = 0
(int) inc (int) x
y: = y + x;
return (y)
In the above example the value of y get changed inside the function defintion due to which the result will change each time. The side effect of the inc ( ) function is it is changing the data ' of the external visible variable 'y'.

PART – IV
IV. Answer The Following Questions

Question 1.
What are called Parameters and write a note on?
Answer:

1. Parameter without Type
2. Parameter with Type Parameters (and arguments)

Parameters are the variables in a function definition and arguments are the values which are passed to a function definition.

(I) Parameter without Type
Let us see an example of a function definition:
(requires: b> = 0)
(returns: a to the power of b)
let rec pow a b: =
if b = 0 then 1
else a * pow a (b – 1)
In the above function definition variable 'b' is the parameter and the value which is passed to the variable 'b' is the argument. The precondition (requires) and postcondition (returns) of the function is given. Note we have not mentioned any types: (data types). Some language compiler solves this type (data type) inference problem algorithmically, but some require the type to be mentioned.

In the above function definition if expression can return 1 in the then branch, by the typing rule the entire if expression has type int. Since the if expression has type 'int ', the function's return type also be 'inf. 'b 'is compared to 0 with the equality operator, so 'b 'is also a type of 'int. Since a is multiplied with another expression using the * operator, 'a' must be an int.

(II) Parameter with Type
Now let us write the same function definition with types for some reason:
(requires: b > 0)
(returns: a to the power of b)
let rec pow (a: int) (b: int): int : =
if b = 0 then 1
else a * pow b (a – 1)
When we write the type annotations for 'a ' and 'b ' the parentheses are mandatory. Generally we can leave out these annotations, because it's simpler to let the compiler infer them. There are times we may want to explicitly write down types. This is useful on times when you get a type error from the compiler that doesn't make sense. Explicitly annotating the types can help with debugging such an error message.

The syntax to define functions is close to the mathematical usage: the definition is introduced by the keyword let, followed by the name of the function and its arguments; then the formula that computes the image of the argument is written after an = sign. If you want to define a recursive function: use "let rec " instead of "let
Syntax: The syntax for function definitions:

let rec fnal a2 … an : = k
Here the fn is a variable indicating an identifier being used as a function name. The names 'al ' to 'an ' are variables indicating the identifiers used as parameters. The keyword 'rec ' is required if fn ' is to be a recursive function; otherwise it may be omitted.
For example: let us see an example to check whether the entered number is even or odd.
(requires: x> = 0)
let rec even x : = x = 0 || odd (x – 1)
return 'even'
(requires: x> = 0)
let odd x : =
x< >0 && even (x – 1)
return 'odd'
The syntax for function types:
$x \rightarrow y$
$x_1 \rightarrow x_2 \rightarrow y$
$x_1 \rightarrow \ldots \rightarrow x_n \rightarrow y$
The 'x' and 'y' are variables indicating types. The type $x \rightarrow y$ is the type of a function that gets an input of type 'x' and returns an output of type 'y'. Whereas $x_1 \rightarrow x_2 \rightarrow y$ is a type of a function that takes two inputs, the first input is of type 'x1 ' and the second input of type 'x2', and returns an output of type 'y'. Likewise x1 → … → xn → y has type 'x' as input of n arguments and 'y' type as output.

Question 2.
Identify in the following program
Answer:
let rec gcd a b : =
if b < > 0 then gcd b (a mod b) else return a
(I) Name of the function
gcd

(II) Identify the statement which tells it is a recursive function
let rec

(III) Name of the argument variable
a, b

(IV) Statement which invoke the function recursively
gcd b(a mod b) [when b < > 0]

(V) Statement which terminates the recursion
return a (when b becomes 0).

Question 3.
Explain with example Pure and impure functions?
Answer:
Pure functions:
Pure functions are functions which will give exact result when the same arguments are passed. For example the mathematical function sin (0) always results 0. This means that every time you call the function with the same arguments, you will always get the same result. A function can be a pure function provided it should not have any external variable which will alter the behaviour of that variable.
Let us see an example
let square x
return: x * x
The above function square is a pure function because it will not give different results for same input. There are various theoretical advantages of having pure functions. One advantage is that if a function is pure, then if it is called several times with the same arguments, the compiler only needs to actually call the function once. Let's see an example
let i: = 0;
if i < strlen (s) then – Do something which doesn't affect s ++ i If it is compiled, strlen (s) is called each time and strlen needs to iterate over the whole of 's'.

If the compiler is smart enough to work out that strlen is a pure function and that 's' is not updated in the loop, then it can remove the redundant extra calls to strlen and make the #loop to execute only one time. From these what we can understand, strlen is a pure function because the function takes one variable as a parameter, and accesses it to find its length.

This function reads external memory but does not change it, and the value returned derives from the external memory accessed. Impure functions: The variables used inside the function may cause side effects through the functions which are not passed with any arguments.

In such cases the function is called impure function. When a function depends on variables or functions outside of its definition block, you can never be sure that the function will behave the same every time it's called. For example the mathematical function random Q will give different outputs for the same function call, let Random number let a : = random( ) if a > 10 then
return: a
else
return: 10
Flere the function Random is impure as it is not sure what will be the result when we call the function.

Question 4.
Explain with an example interface and implementation?
Answer:
Interface Vs Implementation:
An interface is a set of action that an object can do. For example when you press a light switch, the light goes on, you may not have cared how it splashed the light. In Object Oriented Programming language, an Interface is a description of all functions that a class must have in order to be a new interface.

In our example, anything that "ACTSLIKE" a light, should have function defnitions like turn on ( ) and a turn off ( ). The purpose of interfaces is to allow the computer to enforce the properties of the class of TYPE T (whatever the interface is) must have functions called X, Y, Z, etc.

A class declaration combines the external interface (its local state) with an implementation of that interface (the code that carries out the behaviour). An object is an instance created from the class. The interface defines an object's visibility to the outside world.

The difference between interface and implementation is:

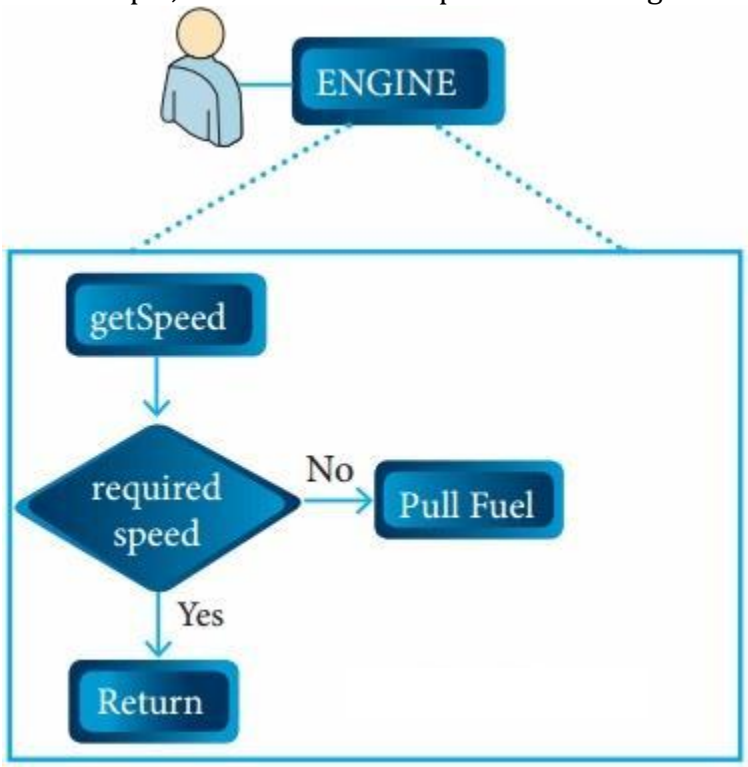
Interface:
Interface just defines what an object can do, but won't actually do it. Implementation carries out the instructions defined in the interface.

Implementation:
Implementation carries out the instructions defined in the interface.
In object oriented programs classes are the interface and how the object is processed and executed is the implementation.

Characteristics of interface

1. The class template specifies the interfaces to enable an object to be created and operated properly.
2. An object's attributes and behaviour is controlled by sending functions to the object.

For example, let's take the example of increasing a car's speed.



The person who drives the car doesn't care about the internal working. To increase the speed of the car he just presses the accelerator to get the desired behaviour. Here the accelerator is the interface between the driver (the calling / invoking object) and the engine (the called object). In this case, the function call would be Speed (70): This is the interface.

Internally, the engine of the car is doing all the things. It's where fuel, air, pressure, and electricity come together to create the power to move the vehicle. All of these actions are separated from the driver, who just wants to go faster.

Let us see a simple example, consider the following implementation of a function that finds the minimum of its three arguments:

```
let min 3 x y z : =
if x < y then
if x < z then x else z
else
if y < z then y else z
```