

AQA Computer Science A-Level
4.12.1 Functional programming
paradigm
Intermediate Notes



Specification:

4.12.1.1 Function type

Know that a function, f , has a function type $f: A \rightarrow B$ (where the type is $A \rightarrow B$, A is the argument type, and B is the result type).

Know that A is called the domain and B is called the co-domain.

Know that the domain and co-domain are always subsets of objects in some data type.

Loosely speaking, a function is a rule that, for each element in some set A of inputs, assigns an output chosen from set B , but without necessarily using every member of B . For example, $f: \{a,b,c,\dots,z\} \rightarrow \{0,1,2,\dots,25\}$ could use the rule that maps a to 0, b to 1, and so on, using all values which are members of set B .

The domain is a set from which the function's input values are chosen.

The co-domain is a set from which the function's output values are chosen. Not all of the codomain's members need to be outputs.

4.12.1.2 First-class object

Know that a function is a first-class object in functional programming languages and in imperative programming languages that support such objects.

This means that it can be an argument to another function as well as the result of a function call.

First-class objects (or values) are objects which may:

- appear in expressions
- be assigned to a variable
- be assigned as arguments
- be returned in function calls.

For example, integers, floating-point values, characters and strings are first class objects in many programming languages.



4.12.1.3 Function application

Know that function application means a function applied to its arguments.

The process of giving particular inputs to a function is called function application, for example $\text{add}(3,4)$ represents the application of the function add to integer arguments 3 and 4. The type of the function is $f: \text{integer} \times \text{integer} \rightarrow \text{integer}$ where $\text{integer} \times \text{integer}$ is the Cartesian product of the set integer with itself. Although we would say that function f takes two arguments, in fact it takes only one argument, which is a pair, for example $(3,4)$.

4.12.1.4 Partial function application

Know what is meant by partial function application for one, two and three argument functions and be able to use the notations shown opposite. The function add takes two integers as arguments and gives an integer as a result. Viewed as follows in the partial function application scheme: $\text{add}: \text{integer} \rightarrow (\text{integer} \rightarrow \text{integer})$ $\text{add } 4$ returns a function which when applied to another integer adds 4 to that integer. The brackets may be dropped so function add becomes $\text{add}: \text{integer} \rightarrow \text{integer} \rightarrow \text{integer}$ The function add is now viewed as taking one argument after another and returning a result of data type integer .

4.12.1.5 Composition of functions

Know what is meant by composition of functions. The operation functional composition combines two functions to get a new function. Given two functions $f: A \rightarrow B$ $g: B \rightarrow C$ function $g \circ f$, called the composition of g and f , is a function whose domain is A and co-domain is C . If the domain and co-domains of f and g are \mathbb{R} , and $f(x) = (x + 2)$ and $g(y) = y^3$. Then $g \circ f = (x + 2)^3$ f is applied first and then g is applied to the result returned by f .



Functions

A function is defined as a **rule** that takes **each member** of a set of inputs and **returns an output** from a set of possible outputs.

An **argument** is the name given to the pieces of data that are passed to a **function**. An **argument** could be a **number** (0, 1, 3.4, -8 e.t.c), a **character** ("a", "D", "!" e.t.c) or any **other data type**. The **function** will **specify** what **data type** is required for the **argument**.

Function Example 1:

A function is called DoubleMe. This **outputs** the double of the **input**. The inputs (set A) could be the set of **natural numbers** {0, 1, 2, 3, 4....}, and therefore the output (set B) could be the set of **even natural numbers** {0, 2, 4, 6, 8} (as doubling a value always results in an even number).

The set of even natural numbers is a **subset** of the set A. Therefore set B could also be the set of **natural numbers**, although the **odd items** would **never** be **outputted**.

The function DoubleMe could be called with a value of 6. The output would be the double of 6, which is 12. 6 is called the **argument**, and 12 is the **result**. Here is the code for such an interaction.

```
DoubleMe 6  
>> 12
```

Function Example 2:

A function is called HalfMe. This **outputs** half of the **input**. The **input** (Set A) could be the set of **positive integers** {0, 1, 2, 3, 4....} and the **output** (Set B) could be drawn from the set of **real** numbers. Most of the real numbers will never be outputted (e.g. 3.201 is not half of an integer), but the real numbers is still an appropriate set. Below are two examples of how HalfMe might be used.

```
HalfMe 200  
>> 100
```

```
HalfMe 5  
>> 2.5
```

Synoptic Link

The set of **Natural Numbers** is the set of positive whole numbers including 0. I.e. {0,1,2,3...}

Natural Numbers are covered in **Number Systems** under **Fundamentals of Data Representation**.

Synoptic Link

The set of **integers** is the set of **whole numbers** {...-1,0,1,2...}. The set of **reals** is the set of all **non-imaginary** numbers, including **negatives**, **irrationals**, and **decimals**.

Integers and reals are covered in **Number Systems** under **Fundamentals of Data Representation**.



Function Types

All functions have a **function type**. If f is the function, A is the input and B is the output, the function type can be defined as the following:

$$f: A \rightarrow B$$

A is known as the **argument type**, and B is the **result type**. This means function f **maps A to B** . In Computer Science, we describe the the **set of inputs** (A) as the **domain**, and the **set of outputs** (B) as the **co-domain**. Not all members of the co-domain have to be used as outputs. The domain and co-domain are always **subsets** of **objects** in **some data type**, as further explained below.

Function Types Example 1:

The function f returns double the input. Hence, this **function type** could be described as the following:

$$f: \{0, 1, 2, 3, 4\dots\} \rightarrow \{0, 2, 4, 6, 8 \dots\}$$

The set of **natural numbers** is the **domain** and the set of **even natural numbers** is the **co-domain**.

Domain

Co-domain

$$f: \{0, 1, 2, 3, 4\dots\} \rightarrow \{0, 2, 4, 6, 8\dots\}$$

Natural Numbers

Even Natural Numbers

In this example, the programmer only allows **positive integers** to be doubled. This function could **not** be used to double a negative number or a decimal value.

The **domain** is a **subset** of the **natural numbers**, and a **proper subset** of the **integers** and **reals**. The **co-domain** is a **proper subset** of the **naturals**, **integers** and **reals**.

However, the functionality of f could be changed if it had a different **function type**.

$$f: \text{real} \rightarrow \text{real}$$

If f was declared with this function type **any real** number could be doubled including negatives and decimal values.



Function Types Example 2:

The function g returns half of the input. Its function type could be described as below:

$$g: \{0, 1, 2, 3, 4 \dots\} \rightarrow \text{real}$$

The set of natural numbers is the domain and the set of reals is the co-domain.

$$g: \{0, 1, 2, 3, 4 \dots\} \rightarrow \text{real}$$

Domain Co-domain

Natural Numbers Reals

This function can only be used to half positive integers. Therefore not all of the reals could be used as outputs (e.g. negative numbers).

The domain is a subset of the natural numbers, and a proper subset of the integers and reals. The co-domain is a subset of the reals.

If the programmer wanted to only half integers, the domain could be the set of integers.

$$g: \{ \dots -2, -1, 0, 1, 2, 0 \dots \} \rightarrow \text{real}$$

If the programmer defined the function as

$$g: \text{real} \rightarrow \text{real}$$

then any non-imaginary number could be halved by the function.



First-class objects

A first-class object is an object which can:

- appear in expressions
- be assigned to a variable
- be assigned as arguments
- be returned in function calls

Examples of first-class objects in procedural programming include [integers](#), [floating-point values](#), [characters](#) and [strings](#).

In functional programming, [functions](#) are [first-class objects](#). Some [imperative](#) languages also support functions as first-class objects. Hence, functions can be [passed as arguments](#) or [returned as the result of another function](#).

Synoptic Link

In **imperative programming**, you must specify **how** to achieve the desired result. In a **declarative language**, you simply **describe the final result**.

Imperative and declarative languages are covered in **Classification of Programming Languages** under **Fundamentals of Computer Systems**.

Function application

Function application is just a fancy term for [applying](#) the [function rule](#) to the [arguments](#) of the function.

Function Application Example 1:

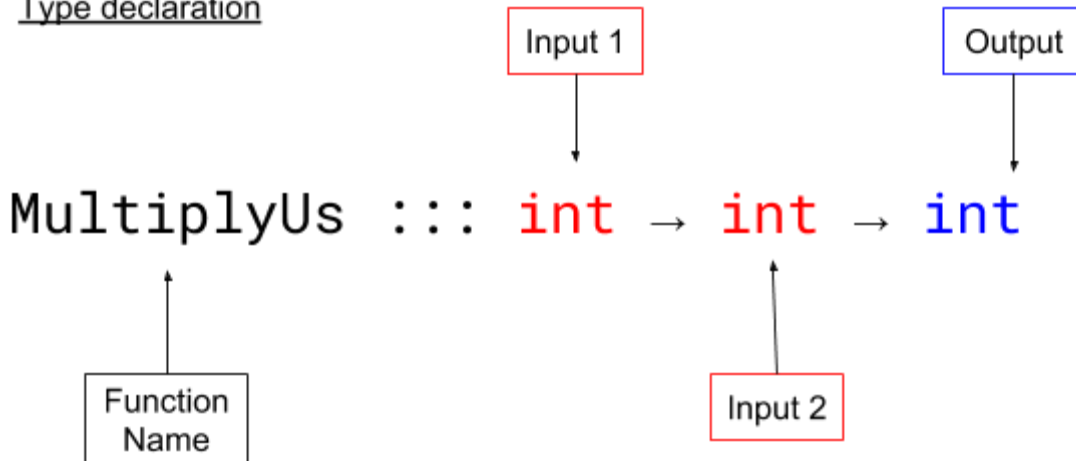
`MultiplyUs(x, y)` returns the product of `x` and `y`. First, we need a function type declaration. In this instance we will only be multiplying integers - two integers multiplied together will always produce another integer.

```
MultiplyUs :: int → int → int
```

The first two “int”s signify the inputs, and the last int is the output.



Type declaration



Next we need to specify what the function will be doing with the inputs.

MultiplyUs x y = x * y

The x and y represent the parameters - variables created when the function is declared, into which the arguments (data) is passed.

Now we are ready for [function application](#). In this instance, the arguments are 3 and 5.

MultiplyUs 3 5

>> 15

Although it may look like the multiply function is taking [two arguments](#), it is in fact [only taking one - a pair](#). Every function in a [functional programming](#) language (e.g. Haskell) only takes [one argument](#). How can this be true? If we look back at the [type declaration](#), it clearly has two inputs. However, the [type declaration](#) can also be written as this:

MultiplyUs :: int → (int → int)

Now we have [two functions](#), each with [one input](#) and [one output](#).



Function 1

Input

Output

MultiplyUs :: int → (int → int)

Function 2

Input

Output

MultiplyUs :: int → (int → int)

Function 1 is called first.

MultiplyUs :: int → (int → int)

MultiplyUs 3 → MultiplyYBy3 5

As you can see, the first function has [created a new function](#) based off the [input](#). The [output](#) of MultiplyUs 3 is MultiplyYBy3 5 (remember in this example y = 5). This new function is called.

MultiplyUs :: int → (int → int)

MultiplyYBy3 5 → 15

The [output](#) is 15. All [inputs](#) have been dealt with so MultiplyUs 3 5 returns 15.



Partial function application

[Partial function application](#) takes advantage of the [inability of a function](#) to take [more than one input](#). In partial function application, one of the [arguments](#) is fixed, leading to a more [restricted](#), [specialised](#) function.

Partial function application example:

The function `Add3Numbers` should add the three numbers given as [arguments](#) and [output](#) the total. `Add3Numbers` could be created as thus:

```
Add3Numbers :: real -> real -> real -> real
```

```
Add3Numbers x y z = x + y + z
```

In this case, the [inputs](#) and [outputs](#) are [real](#). This function can take decimals and negative numbers as arguments. `Add3Numbers` could be called as thus:

```
Add3Numbers 5 7 3.2
```

```
>> 15.2
```

However, we could use [partial function application](#) to [bind](#) one of the [arguments](#). Consider the new function `Add3ToTwoNumbers`. This new function should take a pair of [arguments](#), add them together and then add 3, before returning the [output](#). We could write a [new definition](#) for this function, or we could use [partial function application](#) to [modify](#) `Add3Numbers`.

```
Add3ToTwoNumbers :: real -> real -> real
```

```
Add3ToTwoNumbers = Add3Numbers 3
```

Calling `Add3Numbers 3` would ordinarily cause an error, as there are not enough inputs. However, by defining `Add3ToTwoNumbers` in terms of `Add3Numbers`, `Add3ToTwoNumbers` can be called with [two additional inputs](#).

```
Add3ToTwoNumbers 4 1
```

```
>> 8
```





Composition of functions

Functional composition is the process of **combining two functions** to create a **new function**. The benefit of this is that the user is able to use the functions both **separately**, and in **conjunction**. **Any two** functions can be combined as long as the **domain** of one of the functions is the **same** as the **co-domain** of the other. The symbol \circ indicates that two functions are being combined e.g. Add2 \circ DoubleMe. The function works from the **inside out**. Hence in the previous example, the input would be **doubled first**, then **2 would be added** to it.

Composition of functions Example:

Function f takes an **argument pair**. It adds them together, and then doubles the answer.

$$f ::: \text{real} \rightarrow \text{real} \rightarrow \text{real}$$

$$f(x, y) = (x + y) * 2$$

The function g squares its input.

$$g ::: \text{real} \rightarrow \text{real}$$

$$g(x) = x * x$$

How can f and g be combined?

$$f ::: \text{real} \rightarrow \text{real} \rightarrow \text{real}$$
$$g ::: \text{real} \rightarrow \text{real}$$

To be combined, the **domain** of one function must be the **same** as the **co-domain** of the other function.

The **domain** of f is a **pair of reals**; the co-domain of g is **one real**. Therefore they **cannot** be combined as $f \circ g$.

The **domain** of g is a **real**; the **co-domain** of f is also a **real**. Therefore they **can** be combined as $g \circ f$.



The **domain** of $g \circ f$ is the **domain** of f (a pair of reals), and the **co-domain** of $g \circ f$ is the **co-domain** of g (a single real).

Calling $g \circ f$ with input (3,5) would produce the following:

$$g \circ f (3, 4)$$

Step 1

Function f is applied on the inputs

$$f (3, 4) = (3 + 4) * 2 = 14$$

Step 2

The output of function f is the argument passed to g

$$g (14) = 14 * 14 = 196$$

Step 3

Answer is outputted

$$>> 196$$

