The Big Book of Science

ocket88888

Big Book of Science Series

language science press

Series information: ./LSP/lsp-seriesinfo/eotms-info.tex not found!

The Big Book of Science

ocket88888

language
science
press

Contents

1 Building Blocks: Basic Data Types

Add content in chapters/01.tex.

1.1 Functions

Functions are incredibly useful constructs throughout mathematics, the natural sciences, and in particular computer science. Functions take any number of inputs, or arguments, usually performing operations on those arguments to produce an output. The definition of a function takes the following form:

```
type FunctionName[type ArgumentName0, type ArgumentName1, ...
    type ArgumentNameN]{
  Operations
  return Possible Output
}
```

Listing 1.1: Function Example

As seen above, the first word is ”type,” which refers to the type of data that the result, or output, of the function will be. In later chapters we will define more of these datatypes, but for now we'll introduce the most simple. ”void” is the type of data that doesn't exist. That is to say, a function of type void not produce a result, and is typically used only to modify the arguments. It should be noted that an argument cannot be of this type, as it's kind of nonsensical to pass no information as an argument.
The second word is ”FunctionName,” which is, perhaps obviously, a placeholder for the name of the function. A reader with any amount of experience with trigonometry will recognize the function name Sine, for example. When using a function, only the name of and arguments to the function need be provided for its use to be complete and valid.
After the function name, a list of comma-separated arguments, with their own types, are specified. A function may take any number of arguments, but the most common is simply one. In cases of multiple arguments the order of the arguments

is important. A datum of type A may not be passed to a function that only accepts one argument of type B. Similarly, a function that takes one argument of type A, and then an argument of type B cannot be used on one argument of type B, and then an argument of type A.
The placeholders "Operations" and "Possibly Output" are contained within curly braces ({ and }). This serves to make entirely explicit the operations that the function performs, so that they may not be confused with any work done directly above or beneath them.
"return" is used to specify that what follows is the output of the function. Traditionally, this is also the endpoint of the function. That is, any operations after it are ignored. For this reason, even functions that have the "void" type may return nothing to indicate that computation is to cease at that point, but it is certainly not required.
Functions are used in a similar manner to this throughout many computer science languages. Specifically, those familiar with Java or C may find 1.1 to be reminiscent of those languages. Please note that the above is NOT representative of any programming language, it is merely a way of organizing definitions of mathematical operations. The way a function is used is called "calling" that function, and it takes the following form:

```
1 FunctionName[ArgumentName0, ArgumentName1, ... ArgumentNameN]
```

Listing 1.2: Function Call Example

The first word this time is "FunctionName," which differs from 1.1 in that it does not specify a type. Since you cannot use a function that does not exist, the type of the function's result has already been set, and does not need to be specified again.
This function call also differs from 1.1 in that it's arguments are not typed. Just as the function type need not be respecified, the type of the arguments has already been decided, and the burden of remembering which arguments are what type rests on the shoulders of the person writing the function call.

1.2 Integers

At this point it will be useful to define a data type that is not simply "void". Many readers will already be familiar with the concept of integers, but this chapter exists largely for the purpose of rigorously defining perhaps somewhat simple ideas.

Our starting point isn't to begin defining the integer, but to describe a new data type, the "whole number," which will be abbreviated in writing mathematical expressions as "wn". The definition begins with one member from which all others are defined. That is to say:

0 *is the first whole number*

Don't worry about the definition of 0's value right now, that will become apparent shortly.
Every other whole number can be obtained from a recursive definition using the so-called "Successor Function" in the following way:

```
wn SuccessorFunction[wn n]{
  return the next successive whole number
}
```

Listing 1.3: Successor Function

This can also be expressed in the following way:

$$SuccessorFunction[0] = 1$$
$$SuccessorFunction[1] = 2$$

Successor Function Example

This implies that the numbers 0-9 are just symbols attributed to the whole numbers generated by by the Successor Function. It is commonplace to use what is known as a "base-ten number system," which means that upon reaching 9, two characters arise to take its place, starting with a 1 in the furthest left, and a 0 on the right. The place a number holds is called its digit. In general, whenever a digit reaches 9, the digit directly to the left has the Successor Function called on it, which increments the digit. If there is no digit to the left, it is treated as 0, which produces the whole number 1 in that digit. Keep in mind that this is simply a way of representing an infinite number of outputs of the Successor Function, and as such should be thought of collectively as one whole number, not a collection of whole numbers.
It is easy to see at this point how simple arithmetic operations can be defined. The difference between operations and functions is largely immaterial, in fact it is useful to define operations as functions first, then show the symbol. For instance, addition can be described in the following way:

```
wn PlusOne[wn x]{
  return SuccessorFunction[x];
}
```

Listing 1.4: Addition Function

Output: $x + 1$

So as seen above, adding one to a number is the same as calling the Successor Function on it (which may have been obvious by now). Then adding a number greater than one to some whole number 'x' is as simple as calling the successor function that many times. We can now easily define a symbol '+' to represent calling the successor function x times on a whole number y if given in the form $x + y$. At this point it is useful to note that

$$SuccessorFunction[2] = 3 = SuccessorFunction[SuccessorFunction[1]]$$

which is a simple way of saying that $2 + 1 = 1 + 2$, known as the associative property of addition.

1.2.1 Test

2 Change title for chapter 2 in chapters/02.tex

Add content in chapters/02.tex

3 Change title for chapter 3 in chapters/03.tex

Add content in chapters/03.tex[1]

[1] Comrie (1981) is still useful for diversity linguistics.

Bibliography

Comrie, Bernard. 1981. *Language universals and linguistic typology*. Oxford: Basil Blackwell.

Back Title

ISBN 978-0-000000-00-2