The Big Book of Science

ocket88888

Big Book of Science Series

language
science
press

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 data-types, 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.
Lastly, the end of every line within the function definition ends with a semicolon. The semicolon serves to indicate the end of one operation, and is used rather than just a new line because sometimes a mathematician may choose to write multiple operations on one line due to space constraints or personal styling preference.
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:

```
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 a function that does not exist cannot be used, 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.

1.2.1 Whole Numbers and the Successor Function

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 that is 9 has the Successor Function called on it, the digit directly to the left has the Successor Function called on it, which increments the digit, and the digit that was 9 becomes 0. 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.

1.2.2 Addition

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.
It is here that the value of 0 becomes apparent. For example, consider the statement $1 + 0$. We know from above that this is equivalent to the statement $SuccessorFunction[0] = 1$. From here it is trivial to show that any whole number that has the number 0 added to it retains its value.

1.2.3 Negative Numbers and Subtraction

Before going any further, we will develop our understanding of whole numbers into an understanding of integers. Like with whole numbers, the easiest place to start is with a function.

```
wn PredecessorFunction[wn x]{
  Find a number y such that SuccessorFunction[y]=x;
  return y;
}
```

Listing 1.5: The Predecessor Function

The Predecessor Function, it would seem, only undoes the Successor Function, and as such is sometimes called the Inverse Successor Function. Like with addition, it may be useful to think of the operation of subtraction as an implementation of this function.

```
wn MinusOne[wn x]{
  return PredecessorFunction[x];
}
```

Listing 1.6: Minus One Function

The above function can be used to describe $x - 1$, and in general subtraction of a whole number x from a whole number y is defined as calling the Predecessor Function x times on y, given the form $y - x$.
It is now easy to define all negative numbers as simply a number for which the operations of the Successor Function and the Predecessor Function are switched. That is to say that for a number x and a number y, $y + x = y - Negativex$. There exists then a number z for every y such that $y + z = 0$, which means that when adding the negative number z to y, the Predecessor Function is called on y exactly as many times as the Successor Function must be called on 0 to obtain the value y. This is the definition of Negative y, meaning $z = Negativey$
Because whole numbers obey the original definitions of addition and subtraction, these new negative numbers cannot be said to be whole numbers. The name of these new numbers, in conjunction with the whole numbers, is ”integers” (abbreviated ”int” in mathematical expressions).
It is implied, of course, that this means that calling the Predecessor Function x times on 0 produces Negative x, that is to say that $0 - 1 = Negative1$. Furthermore, because writing out ”Negative” every time subtraction is used can get a bit cumbersome and cluttered, and because of their unique property that adding is subtracting and subtracting is adding, negative numbers are written in the notation

$$Negativex = -x$$

This is because there is an implicit 0 to the left of the $-x$, which is omitted simply because that too would get repetitive, and only the symbol '$-$' is required to indicate the negativity of an integer x. For integers, it can be seen that the statement $x - y$ is equivalent to the statement $x + Negativey = x + -y$

1.3 Sets

A set isn't so much a data-type as a way of organizing data. Sets are useful throughout all fields of natural sciences, computer science, math and logic as a way of describing large (sometimes infinite) amounts of data. A set contains only information on the unique objects it contains, and not any information about the ordering thereof. For example, suppose I have a set A that initially contains nothing, which incidentally has its own symbol: $\varnothing$.

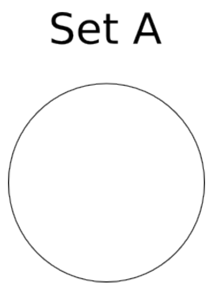

Figure 1.1: Set A

Now suppose I add the integer 1 to set A.

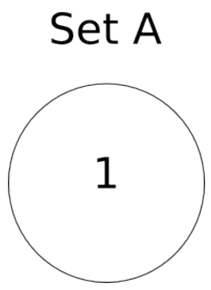

Figure 1.2: Set A

If I were to try to add 1 to set A again, this is what set A would look after I attempt that operation:

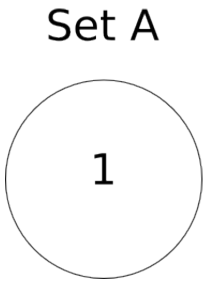

Figure 1.3: Set A

See, the set remains unchanged upon adding an element that it already contains. *A set only contains information about the **unique** items it contains.*
Now suppose that I added the integer 2 to this set.

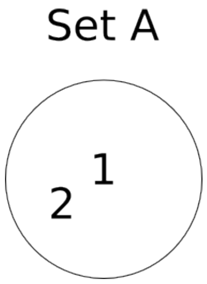

Figure 1.4: SetA

As seen above, there is no notion of the order or rank of the elements of A. Even though 1 was added to the set before 2, 1 holds no special position, and the two integers are treated equally. There is a special, named set for the set that contains all whole numbers, denoted '$\mathbb{W}$', and the set that contains all integers, commonly denoted '$\mathbb{Z}$'.

1.4 Multiplication and Division

Before we can continue to build up sets of useful constructs, we first need to define some more operations that can be preformed on integers (specifically division will be of interest later).

1.4.1 Multiplication

Of the two operations that make up the title of this section, multiplication is the easier of the two to explain. The following function describes it in loose terms.

```
int Multiply[int x, int y]{
  int result = $x_1$ + $x_2$ + $x_3$ ... + $x_y$;
  return result;
}
```

Listing 1.7: Multiplication Function

What this means, in English, is that multiplying x by y returns x added to x y times. For this reason, expressions of the form $x \times y$ are typically pronounced ”x times y”, and the '$\times$' symbol denotes multiplying the left side by the right side. This is often abbreviated, as the $\times$ symbol can be confused with a variable named x, as simply xy. This is syntactically unambiguous because all variable and function names require formatting called 'Camel Case' which means that the first letter of the name of the variable or function may be lowercase, but every time another word is added, its first letter must be capitalized. For example, 'thisIsAVariable' is a valid variable name, but 'thisisambiguous' is not. However, because it is common for function and variable names to only be one word long, the $\times$ symbol is often used to delimit them (e.g. $a \times var$).
Multiplication of a number by itself repeatedly is represented in a special way called a 'power'. A power is written with the number of times the base number (appropriately called the 'base') written to the upper-right as a superscript (called the 'exponent'). The following function describes it in a similar way that multiplication was explained in 1.7.

```
int Power[int x, int y]{
  return $x_1 \times x_2 \times x_3$ ... $\times x_y$
}
```

Listing 1.8: Power Function

The invocation of this function is written in the form 'x^y' where x is being multiplied by itself y times, also called 'raising x to the y power'.

1.4.2 Division

To put it as simply as possible, division is the operation that undoes multiplication. However, similar to subtraction, order matters. For instance, if z is the output of $x \times y$, then the output of dividing z by x is y. Similarly, the output of z divided by y is x. Because order matters, dividing y by z is *not* the same as dividing z by y. Here is a loosely-written function that helps to describe that process.

```
int DivideBy[int x, int y]{
  return the integer y is multiplied by to obtain x;
}
```

Listing 1.9: Division Function

There are, however, many cases where there is no integer that y can be multiplied by to obtain x. Consider, for example, the division of 5 by 3 (there is no integer y such that y times 3 outputs 5). In these cases, the output is left in the form of what is known as a 'fraction'. Fractions take the form $\frac{x}{y}$ or x/y when dividing x by y. The difference between them is accounted for by the versatility of the environment in which it is written, $\frac{x}{y}$ is the preferred form, but x/y may be used when such typesetting is unavailable.
In a base ten number system, it becomes bothersome to always write fractions this way, but to understand it we must first describe some properties of fractions. The value of a fraction is invariant under multiplication of both the top (called the 'numerator') and the bottom (called the 'denominator') by the same integer. Because division is merely the inverse of multiplication, the same must be true for division.

```
int x;
int y;
int c;
```
$x/y = \frac{x}{y} = \frac{x \times c}{y \times c} = \frac{x/c}{y/c}$

Listing 1.10: Fraction Properties

In a base ten system, these fractions are to be converted to fractions of ten

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

9 780000 000002