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

Chapter One outlines some basic datatypes and functional syntax used throughout the rest of the Book. Many of the function definitions herein use extremely loose syntax, which is all that's necessary for an introductory chapter. The only true reason this chapter exists is an exercise in rigor for those who enjoy such things, and a refresher course in basic concepts for those who may have forgotten. In other chapters, a level of knowledge consistent with mid-high school or higher will be assumed, and in even later chapters, an understanding of material in sections indicated at the beginning of those chapters.

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:

```
1 type FunctionName[type ArgumentName0, type ArgumentName1, ...
      type ArgumentNameN]{
2   Operations;
3   return Possible Output;
4 }
```

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:

```
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 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:

$$0 \textit{ is the first whole number}$$

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: $\varnothing$.

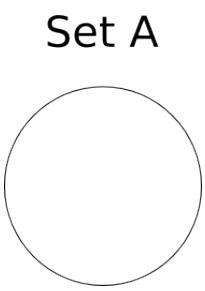

Figure 1.1: Set A

Now suppose I add the integer 1 to set A.

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

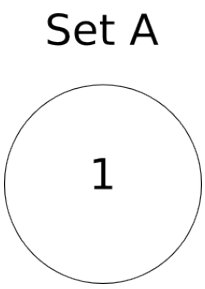

Figure 1.2: Set A

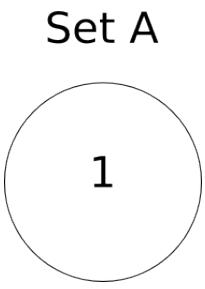

Figure 1.3: Set A

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.

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, Division and the Order of Operations

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).

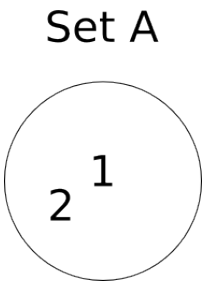

Figure 1.4: SetA

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. Some interesting properties of powers are that $X^0 = 1$ for every x of integer type (and actually all real and complex x, but those types of numbers will be discussed later), and $x^{-y} = \frac{1}{x^y}$.

```
int Power[int x, int y]{
  return x_1 × x_2 × x_3 ... ×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 Order of Operations

There are some more properties of multiplication that are necessary to understand division in base ten format, but these cannot be discussed without first knowing the Order of Operations. This is a set of rules for what order in which operations are to be preformed, otherwise an expression written by one mathematician could be evaluated differently by another mathematician. The Order of Operations is:

1. Function Calls
2. Grouping Symbols
3. Exponents
4. Multiplication and Division
5. Addition and Subtraction

The order in which these items are placed reflects their priority. For example, given the function definition

```
int PlusFive[int x]{
  return x+5;
}
```

Listing 1.9: PlusFive Example Function

and the expression

$$10 + 2 \times 4 + PlusFive[3] - 10 + (2 - 2^2),$$

it might be confusing what the value of the output is without having a canonical way of deciding which operations take precedence over others. In this example, we first look for function calls, and we find PlusFive[3]. Once that is evaluated, the expression becomes

$$10 + 2 \times 4 + 8 - 10 + (2 - 2^2)$$

From here, we must next look for grouping symbols, and the ones we find are those parenthesis. Parenthesis are grouping symbol whose only function is to indicate that the expression within them are to be indicated before the expressions outside them. In this case that expression is $2 - 2^2$. There are no functions in this expression, so we evaluate the power first, which leaves $2 - 4$. There's only one operation left, and that leaves -2. Now we replace the original grouping symbols with this output: $10 + 2 \times 4 + 8 - 10 - 2$. After grouping symbols, we do multiplication and division from left to right. There's only one multiplication, 2×4, and no division, so evaluating this and placing the output where the operation was leaves only $10 + 8 + 8 - 10 - 2$ remaining. All that's left is addition and subtraction, resulting in a final output of 14.

1.4.3 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 1.10: 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 = x/y = (x×c)/(y×c) = (x/c)/(y/c)
```

Listing 1.11: Fraction Properties

The fraction is also a type of grouping symbol, which can be useful to avoid clutter caused by parenthesis. This means that

$$\frac{x+y}{a+b}$$

is equivalent to

$$(x+y)/(a+b)$$

for all x, y, a and b. Also, because it's a grouping symbol, any valid expression can exist in either the numerator or the denominator. This means that fractions can be nested, like so:

$$\frac{\frac{1}{3}}{5}$$

or

$$\frac{a}{\frac{b}{c}}$$

In a base ten system, these fractions are to be converted to fractions of ten in whatever way is available. For example, $\frac{2}{5}$ can be multiplied by 2 on the top and bottom to obtain $\frac{4}{10}$ and the fraction $\frac{12}{30}$ can be divided by 3 on the top and bottom to obtain $\frac{4}{10}$. This fraction can then be represented in the form 0.4. As may be apparent by now, the base ten system is very closely tied with powers of ten. A number in the first digit is multiplied by 10^0 (called the "one's" place'), the second digit is multiplied by 10^1, the third by 10^2 and so on, then these products are added which gives the full value of the number. When representing fractions in base ten format, a '.' called a decimal point (or sometimes radix point to generalize between base systems) is placed behind the one's place, and to the right of it fractions of 10^{-1} are represented without the denominator. This is continuously true for all negative powers of ten, that is to say that two places to the right of the decimal point are fractions of 10^{-2}, three places to the right lie fractions of 10^{-3} and so on. Here are some examples:

$$1 + \frac{7}{10} = 1.7$$
$$3 + \frac{1}{20} = 3 + \frac{5}{100} = 3.05$$
$$1 + \frac{6}{25} = 1 + \frac{24}{100} = 1.24$$

In general, many fractions can be expressed this way because the negative powers of ten keep going to infinity, so even fractions of arbitrarily large denominators can be expressed. Decimal numbers are the preferred way of representing output values, and fractions are only used to indicate division. The set of all integers and all fractions that can be represented as a decimal number forms a special set called the 'Rational Numbers,' and it has its own special symbol: $\mathbb{Q}$.

1.4.4 Real Numbers

There is only one step left before the most useful basic data type is defined. Consider a fraction such as $\frac{1}{3}$. This number cannot be expressed as any multiple of any negative power of ten. It is recursively just a little bit bigger than $\frac{3}{10}$, $\frac{33}{100}$, $\frac{333}{1000}$ and so on. However, it's always closer to a numerator constructed of repeating digits of 3 than one that is constructed of repeating 3's that end in a 4. For this reason, the decimal system can make an approximation that takes the form of $0.3333333333333333333...$ and so on to infinity, which is abbreviated as $0.\overline{3}$. Any fraction that cannot be represented as a decimal, but can be approximated as an infinitely repeating decimal, is also considered a rational number, because it can be represented as a fraction. Actually, any number that can be represented as a fraction is a rational number. There are, however, some natural numbers that defy representation as a fraction or repeating decimal. Examples include π and e, these numbers have values that never repeat, and cannot be represented as a fraction, and only approximations to them can be made in a base ten system. This class of number is called "Irrational," and the set of all numbers that are irrational has the special symbol $\mathbb{I}$. Numbers such as pi are known due to certain relationships in nature (in pi's case the ratio of a perfect circle's diameter to its circumference), and so must be considered a part of the numbers we work with. After all, the value 4 cannot be added to the color blue. Similarly, a rational number cannot be added to one of these irrational numbers. Unless, that is, a larger set is constructed of the elements of both the rational and irrational numbers. This large set that composes most numbers dealt with in algebra and calculus is called the 'Real Numbers,' and it has the special symbol $\mathbb{R}$.

2 Formal Syntax and Operators

This chapter is not really meant to be read from top to bottom, only to be used as a reference when reading later chapters. If the way anything is worded is confusing, this chapter is meant to act as a guide. Certain similarities to computer science may become apparent in the way that mathematical expressions are written, and this is not a coincidence but it cannot be stressed enough that this is *not* a programming language (yet).

2.1 Boolean Logic

A basic datatype not covered in Chapter 1, but is also extremely useful is what's known as a 'boolean,' abbreviated 'bool'. This is much simpler to define than a set of numbers, because rather than infinite possible values, there are only two: true or false. This is useful in cases when operations are to be preformed conditionally, or procedurally. For example, the famous Dirac Delta function models the density of a point-mass and has the following form:

```
real δ[real x]{
  if[x==0]{
    return Infinity;
  }
  else{
    return 0;
  }
}
```

Listing 2.1: Dirac Delta Function

The first thing immediately apparent is the use of the new function 'if'. 'if' is a special function that allows preforming of mathematical operations within the curly braces that follow it if and only if the expression inside of its arguments evaluates to true. 'if' only takes one argument, though there is no limit to the size of the logical expression that can be passed.

2.1.1 Boolean Operators

In order to understand the types of arguments that 'if' can take, we must first define some functions that return bool's. The one used in 2.1 will be a good place to start.

```
bool Equivalent[any x, any y]{
  return true if x is equivalent to y, or false if not;
}
```

Listing 2.2: Equivalence Function

The symbol '==' is used to denote calling Equivalent[x,y] given the form 'x==y'. This is different than '=' because '=' implies assigning a value, whereas after finding the output of 'x==y' the value of x does not become y, or vice-versa. Note that the type used to pass these arguments is 'any' which predictably means that they may be of any type. This is because it is easy to tell if two things are the same type, and if they are not then they cannot be equivalent, which allows sometimes for very easy evaluation of Equivalent's output.
The next two operators are very similar, and likely very familiar.
The following function is another extremely common argument to 'if'.

```
bool GreaterThan[num x, num y]{
  if[x-y is negative]{
    return false;
  }
  return true;
}
```

Listing 2.3: Less Than Function

Calling this function is typically denoted as 'x>y', this being equivalent to 'GreaterThan[x,y]'. Predictably, this returns true if x is, in fact, greater in value than y. This function has the counterpart LessThan, defined below.

```
bool LessThan[num x, num y]{
  if[y-x is negative]{
    return false;
  }
  return true;
}
```

Listing 2.4: Less Than Function

This function is usually called by writing 'x<y', which is the same as 'LessThan[x,y]'. In contrast to 2.3, this returns true if x is *less* than y, and false otherwise.
For now, these three operations will be sufficient to describe 2.1. However, there

are two things that have been left out. First is the value Infinity. Infinity is a real number that has the special properties:

- For all 'x', as long as x is not Infinity, Infinity>x outputs true.
- For all 'x', Infinity+x outputs Infinity.
- For all 'x', Infinity-x outputs Infinity.

The other bit of information not yet explained is the 'else' in 2.1. 'else' is a special function that takes no arguments, and can *only be placed immediately following an 'if' block*. Note that an 'if' does not require an 'else', the latter is entirely optional. If included, the curly braces following an 'else' enclose a set of operations to be preformed if the argument to 'if' evaluates to false.

2.2 Boolean Operators

This is a comprehensive list of boolean operators and their definitions, as well as examples. The list begins with those above, and goes on to list every operator that can be used to compare two values. Note that some of these may deal with topics not covered yet in the Book.

- if/else if/else
  - if

    'if' is a special function that allows preforming of mathematical operations within the curly braces that follow it if and only if the expression inside of its arguments evaluates to true. 'if' only takes one argument, though there is no limit to the size of the logical expression that can be passed.

    Example:

```
1 if[Expression]{
2   Operations;
3 }
```

Listing 2.5: if Example

If 'Expression' turns out to be true, 'Operations' will be preformed, otherwise they will not.

– else if

'else if' is a special function that can only appear immediately following an 'if' block or a previous 'else if' block. Unlike 'if', the expression that else if takes as an argument is only evaluated if the previous 'if' or 'else if's expression evaluated to false. This means that if an 'else if' follows an 'else if' who's expression was not evaluated, then that following 'else if's expression is not evaluated and therefore the operations enclosed in curly braces are not preformed.

Example:

```
1 if[Expression1]{
2   Operations1;
3 }
4 else if[Expression2]{
5   Operations2;
6 }
```

Listing 2.6: else if Example

In the above example, Operations2 is/are preformed if and only if Expression1 evaluated to true, *and* Expression2 evaluated to false.

– else

'else' is a special function that takes no arguments, and can *only be placed immediately following an 'if' block*. Note that an 'if' does not require an 'else', the latter is entirely optional. If included, the curly braces following an 'else' enclose a set of operations to be preformed if the argument to 'if' evaluates to false.

Example:

```
1 if[Expression1]{
2   Operations1;
3 }
4 else if[Expression2]{
```

```
5   Operations2;
6 }
7 else{
8   Operations3;
9 }
```

Listing 2.7: else if Example

In the above example, Operations3 is/are preformed if and only if Expression1 and Expression2 both evaluated to false.

- == / < / <= / > / >=
  - ==

    '==' takes two inputs, and returns true if they are equivalent, false if they are not.

    Examples:

```
1 if[1 == 2]{
2   Operations;
3 }
```

Listing 2.8: Equivalence Example 1

In the above example, Operations is/are not preformed, because 2 is not equivalent to 1.

```
1 if[1 == 1]{
2   Operations;
3 }
```

Listing 2.9: Equivalence Example 2

In the above example, Operations is/are preformed, because 1 is equivalent to 1.

  - <

    '<' takes two arguments, and returns true if the first is less than the second, false otherwise.

    Examples:

```
if[$1<2$]{
  Operations;
}
```

Listing 2.10: Less Than Example 1

In the above example, Operations is/are preformed, because 1 is less than 2.

```
if[$1<1$]{
  Operations;
}
```

Listing 2.11: Less Than Example 2

In the above example, Operations is/are not preformed, because 1 is not less than 1.

Back Title