**How to Write Haskell Code (Syntax)**

Haskell programs are written as equations similar to algebraic equations, with some differences:

|  |  |  |
| --- | --- | --- |
| **Item** | **Algebra** | **Haskell** |
| Variables | Single letter  Latin or Greek alphabet  Other subscripts and superscripts are common  x = α’ | Name can be long  Use Latin alphabet  Must start with lowercase letter  May only have letters, numbers, underscores (\_) and primes (‘)  my\_variable1 = alpha’ |
| Multiplication | No symbol between factors  2xy + 3y(z+2) | Needs \* between factors  2\*x\*y + 3\*y\*(z+2) |
| Exponents | Denoted as a superscript | Uses symbol ^: x^2 |
| Functions | Arguments enclosed in ( )  Separated by commas  f(x,y) = g(x,y) - 1 | Arguments follow name  Separated by spaces  f x y = g x y - 1 |

There is no syntactic difference between a function, a constant and a variable. A constant is just a function of zero arguments, and in Haskell variables are actually constants, as it is not possible to change the value of a variable after its definition. However, in this document, we will use the term *variable* for a function of zero arguments and we reserve the term *literal* to refer to a value that occurs literally in the code. For example, in the equation x = 2, we will say that the variable x is defined to be equal to the literal 2.

When you introduce a new variable or function definition in Haskell, you must put the name you want to define on the left hand side of your equation, not on the right hand side. Here are some examples:

|  |  |  |
| --- | --- | --- |
| **Goal** | **Correct** | **Incorrect** |
| Define variable x | x = y + z | y + z = x |
| Define function f | f x = 2\*x | 2\*x = f x |

Definitions are usually written in separate lines. However, it is also possible to write several different definitions in a single line if you separate them with semicolons.

**Types and Literal Values**

In Haskell, as in many other programming languages, the integer number 2 and the real number 2.0 have different internal representations. In a Haskell expression, all variables and literals must have the same type. A literal number is assumed to be of type real if it includes a decimal point or if it occurs within an expression that contains other numbers of type real. Otherwise, it is assumed to be of type integer. Be aware, however, that real types can have two possible external notations: either fixed point (234.56) or floating point (2.3456e2).

There are other composite and non-numeric types in Haskell. Some of the most useful types, such as lists, tuples, Booleans and strings, are explained in other sections of this document.

**Auxiliary definitions**

When you want to define a complicated variable or function, you can introduce auxiliary definitions that will only be valid inside that function. Auxiliary definitions are introduced by the keyword where and must be aligned to each other and indented to the right with respect to the main definition. Example:

f x = double x + square x

where double x = x+x

square x = x\*x

Note that the keyword where is indented to the right, and then the auxiliary definitions are further indented to the right. Proper indentation like this is always necessary. The example above defines a function f that will double and square its argument and then add the two resulting values together. We could have written an equivalent definition of the same function like this:

f x = 2\*x + x^2 or also like this: f x = (2+x)\*x

**Important:** Always make sure that auxiliary definitions are perfectly aligned to each other and do not use TAB when you align them. Use only the SPACE bar.

Alternatively, you may put several auxiliary definitions in a single line if you separate them with semicolons and enclose everything after the where keyword in curly brackets. Example:

f x = double x + square x where { double x = 2\*x ; square x = x^2 }

**Multiple-case definitions and multiple definitions**

Like in Algebra, you can define piecewise functions that behave differently for different values of the argument. Each case is introduced by a vertical bar, and different cases must be aligned to each other. You can use the keyword otherwise to catch all cases that you did not list explicitly. For example, here is a definition of the absolute value:

abs x | x > 0 = x

| x < 0 = -x

| otherwise = 0

You can also have several definitions, either single-case or multiple-case, for the same function, as long as you put together all the definitions in consecutive lines of the same file. The definitions will be searched in the order they appear in the file, and the first definition that matches the arguments is chosen for execution. For example, given the following two definitions for the function f

f 3 = 5

f x = x+1

the value of (f 3) is 5, and the value of (f 4) is also 5. These multiple single-case definitions could have also been done with a single multiple-case definition:

f x | x == 3 = 5

| otherwise = x+1

**Operators**

An operator is just a function with a special syntax. Its name consists of special symbols instead of letters, and it is placed between its arguments instead of before the arguments. Apart from that, operators behave exactly like functions. In fact, you can place an operator before its arguments if you enclose it in parentheses:

3 + 5 is the same as: (+) 3 5

Many operators are already pre-defined, but you can create your own operators too. The most important pre-defined operators are explained in this document.

**Comments**

Comments are text included in a program but intended for humans, not for the computer. To make the computer ignore a line with comments, start it with two consecutive dashes: --

-- This is a comment line and will not be executed

Comments are sometimes used as a quick and dirty way to temporarily disable some parts of the code. This is a useful practice while you are writing your code, but you should remove commented out code from your final program. On the other hand, you should have enough comments (text, not code) in your final program to explain anything that would not be immediately obvious to a reader of your code.

Multi-line comments start when the symbol sequence {- is encountered and end when the symbol sequence -} occurs. If the comment contains the {- symbol several times inside, then the comment will not end until each opening symbol is *closed* with a corresponding -} symbol.

{-

This is a 3-line comment {- this is a nested comment -}

-}

You can also have lines with both code and comments:

x = {- surprise! -} 2\*y+1 -- This line defines x in terms of y

The compiler will read the line above as if it only had this code: x = 2\*y+1

**Boilerplate**

Most programs need some code to make the computer use some particular libraries or to prepare settings suitable for your goal. These necessary but not very interesting parts of the program are commonly referred to as *boilerplate.* Boilerplate code is usually placed at the beginning of your file, but sometimes it is placed at the end. In these lessons, sections containing boilerplate at the end of the file will be preceded by a comment such as “Do not modify the code below”. You should ignore boilerplate when you begin programming, as it will only make sense once you gain more experience.

**Booleans**

A Boolean is a special type with only two possible values: True and False. Booleans are used in multiple-case function definitions to express the different cases. Apart from using the literals True and False, you generate Booleans when you compare values by means of a relational operator. For example, the expression 2 > 3 is exactly the same value as the Boolean False.

Relational operators include less-than(<), less-than-or-equal-to(<=), greater-than (>), greater-than-or-equal-to (>=), equal-to (==) and not-equal-to (/=).

Note: Use the equality operator (==) when you need to compare two expressions. Do not confuse comparing expressions (a double equal sign) with defining a variable or function (a single equal sign).

You can bind Boolean values to variables and create expressions of type Boolean:

x = (3 == 5) -- The value of x is False.

f x = (x == x) –- The value of (f x) will always be True for any x

The built-in function not reverses the value of a Boolean:

not True = False ; not False = True

There are two operators that can act on Booleans: conjunction (&&) and disjunction (||). These are also known as AND and OR, respectively. These are known as *logical operators*.

Conjunction is defined as follows:

False && False = False ; False && True = False

True && False = False ; True && True = True

Disjunction is defined as follows:

False || False = False ; False || True = True

True || False = True ; True || True = True

An equivalent (shorter but more abstract) definition of the logical operators would be:

False && x = False ; True && x = x

x || y = not (not x && not y)

Logical operators are not used in Haskell as often as in other programming languages, because it is preferred to use multiple-case function definitions and list comprehensions (explained below), which can often produce equivalent outcomes.

**Lists**

In Haskell, you can group several items into a list and use the list as a single item. All items in a list must have the same type, so, for example, you cannot mix integers and strings in the same list. You can deconstruct a list into separate variables, but the number of variables must match the length of the list. Example:

list1 = [2,3,5,7] -- list1 is constructed

[a,b,c,d] = list1 -- list1 is deconstructed into separate variables

The previous equations will make a=2, b=3, c=5, d=7 and list1=[2,3,5,7]

The list [] denotes a list that has no elements.

The cons (:) operator lets you add an element to the front of a list:

2 : [3,4,5] -- The result is the list [2,3,4,5]

Besides being an operator, cons is also a special pattern that can be used to deconstruct lists:

x : y = [2,3,4,5] -- This results in x=2, y=[3,4,5]

Two lists can be concatenated with the (++) operator. For example, list1 and list2 below are exactly the same lists.

list1 = [2,3] ++ [4,5]

list2 = [2,3,4,5]

The pre-defined function length returns the number of elements in a list, so, for example, the expression length list2 will return 4.

There is no operator for adding elements at the end of a list, but you can still do it by using concatenation: [3,4,5] ++ [6] –- The result is the list [3,4,5,6]

When constructing and deconstructing lists, what you have on each side of the equal sign depends on your goal, as the following table illustrates:

|  |  |  |
| --- | --- | --- |
| **Goal** | **Correct** | **Incorrect** |
| Define a and b | [a,b] = p | p = [a,b] |
| Define p | p = [a,b] | [a,b] = p |

**Tuples**

Tuples are composite types similar to lists, but they have a fixed length that cannot be changed. Two-element tuples are called pairs, and three-element tuples are called triples. Unlike lists, tuples cannot be constructed with a cons operator. Elements of a tuple are enclosed in round parentheses instead of square brackets. A tuple represents an element of a Cartesian product. Tuples are treated as a single entity, so, for example, the function f(x,y) does not represent a function of two arguments but a function of a single argument which happens to be a pair.

Tuples can be deconstructed into separate variables just like lists. For example, the following lines assign the values a=17, b=31 and t=(17,31)

t = (17,31) -- construct t from 17 and 31

(a,b) = t -- deconstruct t into a and b

To define just a and b, you can also have this single line: (a,b) = (17,31)

You can mix different types in a tuple, so, for example, (2,”hello”) is a legal pair but [2,”hello”] is not a valid list.

**Ranges**

A range is a special type of list containing consecutive numbers. A range needs one or two initial values, then two dots and a final value. If only one initial value is given, the increment is assumed to be 1. Otherwise, the increment will be the difference between the first two values. Examples:

[1..5] –- Same as [1,2,3,4,5]

[2,4..10] –- Same as [2,4,6,8,10] (increment is 2)

[0.1,0.2..1] –- Same as [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9,1.0]

[1..] -- Same as the infinite list of all positive integers

[3..1] -- Same as [] (positive increment, but final < initial)

[3,2..1] -- Same as [3,2,1] (The increment is -1)

**List Comprehensions**

You can also create lists by picking those elements from another list (called the *base*) that satisfy some condition, and then applying some transformation to the chosen elements. For example, to create the list

[(2,3),(3,4),(5,6),(6,7)]

you can write the following list comprehension:

[(x,x+1) | x <- [2..6], x /= 4 ] –- The base is the list [2..6]

You can read that as: the list of pairs (x,x+1) such that x is taken from the base [2..6] and x is not 4.

**Note:** The compiler converts ranges and list comprehensions into internal low-level loops, so be aware that some list comprehensions may be slow to calculate, and if you use them with an infinite list as the base it is possible that the calculation never terminate.

**Strings**

In addition to numbers, you can manipulate other objects such as literal character strings, which you can use, for example, to print messages in your program. String literals are enclosed in double quotes, but single character literals are enclosed in single quotes. A string literal is just a list of single character literals. For example, greeting1 and greeting2 below are the same string:

greeting1 = “hello”

greeting2 = [‘h’, ’e’, ’l’, ’l’, ’o’]

Strings can be concatenated with the (++) operator. For example, text1 and text2 below are the same exact text:

text1 = “Once upon “ ++ “a time”

text2 = “Once upon a time”

**Type Annotations**

Type annotations for functions allow us to explain succinctly what type of input arguments a function has and what type of output it returns. The types are separated by arrows, where the output type is the last type listed and all the others correspond to the input types.

For example, the following type annotation denotes that function f is a function of two arguments, where the first argument is an integer and the second argument is a point. The function returns a list of points that it creates from those arguments:

f :: Integer -> Point -> [Point]

Question: How would you create a list of points when given an integer number and a single point as inputs?

Type annotations are not executable code, as they only contain information about what the data is, not about how to compute it. However, unlike comments, the Haskell compiler reads the type annotations and uses the information in them to generate better executable code. Therefore, type annotations are intended for both humans and computers.

**Eliminating parentheses**

A common pattern in functional programming is having a chain of functions that are applied successively. In mathematics, this is called function composition. When composing functions, you can reduce the number of parentheses in your expressions by replacing the parentheses around a subexpression with a dollar sign preceding it. For example, you can write:

my\_function1 (expr a b c) or: my\_function1 $ expr a b c

f (g (h x y)) or: f $ g $ h x y

f (g (h x) y) or: f $ g (h x) y

The last example shows that it is not always possible to eliminate all parentheses, but using this construction makes expressions more readable in general.

The composition operator (.) is also used to reduce the number of parentheses. It is defined as:

(f . g) x = f (g x)

You can use the composition operator to write expressions such as f $ g $ h $ x in an equivalent but more idiomatic form: f.g.h $ x

Remember that those expressions execute *backwards,* as they instruct the computer to apply the function h first, then g and finally f, which in algebra is written as: f(g(h(x)))

**Understanding and fixing compiler errors**

When you make mistakes in your code, the compiler will produce an error message. Sometimes, error messages are long, and they may be confusing to inexperienced programmers. If you follow the advice here, you will be able to understand and fix the errors faster.

1. Do not despair. Be willing to spend the time necessary to fix the error.
2. Read the error message entirely, even if you do not understand any of it. You will on time.
3. Go to the first error shown in your message. There is usually more than one error.
4. Look at the first line in your message. It shows the name of the file where the error was found followed by two numbers: the line and the column in that file.
5. Look at the file, line and column where the error occurred.
6. Check that the indentation is correct. This is the number 1 cause of errors.
7. Check for typos. This is number 2 cause of errors.
8. Check for indentation and typos again. This is number 3 cause of errors.
9. If the message mentions “parse error”, then the problem is most likely wrong indentation or typos. Check lines above the error line, as sometimes the error is before that line.
10. If the message mentions “No instance of” or “Couldn’t match type”, then the problem is that you are using the wrong value in some of the variables mentioned in the error message. Try to understand the message, as it is giving you the information you need to fix the error.
11. The probability of the compiler being wrong is extremely small, so if it says you have an error, look for it until you find it. People make mistakes often. Computers do not.
12. Add type annotations to the functions where you suspect the error is. Type annotations will help the compiler produce error messages that are easier to understand.

There is no magic in computers. Everything happens for a logical reason. Computers are not fickle or unpredictable. All errors in computers are due to some human who was not careful enough when writing the code. The only way to avoid computer errors is to be careful when you write code, and that includes being patient when reading error messages and methodical when fixing errors. There is a reason why this process is popularly known as *debugging*: it is tedious but necessary.

**Understanding runtime errors (incomplete definitions)**

Haskell programs are safe from most runtime errors, but programs can still crash if you wrote an incomplete definition somewhere in your code. An incomplete definition is a function definition that does not consider all possible values its arguments can have. A multiple-case definition without an otherwise clause or definitions that expect the values to always be within a few special cases are typical cases of incomplete functions, as are functions that do not check that divisors are not zero before performing division. Check your code for incomplete definitions if you get a runtime error. In short scripts like ours, it may be OK to let the program crash instead of handling the error more gracefully, but you will still need to understand the cause of the crash.

An incomplete definition can be explicitly inserted in your code by using the keyword undefined within a function or variable definition. This keyword is sometimes used to mark parts of your program where the implementation is not done yet, while still having a full program that can be compiled.

**Function Reference List – System Functions**

System functions are used to manipulate the data stored in the computer. Many system functions are pre-defined in a system library, but it is also possible to define new system functions that combine the existing ones into higher-level actions. Some system functions are just convenience functions to automate common tasks, but other are essential to the operation of the computer, which would not work without them. Thus, many system functions are difficult to understand and look like *magic*, but there is no magic involved. System functions are functions just like any other function, but they perform manipulations of the data in the computer memory or operate on the lower level components of a computer, which are usually hidden to the end users.

main is a special system function. It is not pre-defined, but it must be defined by the programmer. When the program is executed, the computer looks at the contents of main and starts executing the instructions defined there, so main is the entry point for any program.

**Note:** In the functions below, you can replace Point with any other type, and most functions will still work the same. Each description is preceded by two lines. The first line is a type annotation for the function described. The second line is an example of use of the function.

drop :: Integer -> [Point] -> [Point]

new\_points = drop n points

Creates the list new\_points, which contains the same elements as the list points, except that the first n points of points are deleted and do not appear in new\_points

length :: [Point] -> Integer

l = length lst

Computes the number of elements in the list lst

not :: Bool -> Bool

opposite\_condition = not original\_condition

Negates the original\_condition, so that if it was True, then opposite\_condition would be False. Similarly, if original\_condition was False, then opposite\_condition would be True.

show :: Number -> String

text = show num

Converts the number num into its textual representation. For example, if num was 8.5, then text would be the string “8.5”

shownum :: Number -> String

text = shownum num

Performs the same conversion as show, but it rounds up the result so that it shows fewer decimals. For example, shownum 3.579999996 would be converted into “3.58”. Note, however, that due to some low-level interactions, this function is not always able to round up the result properly, but in general it shows numeric results more concisely than show

take :: Integer -> [Point] -> [Point]

extracted\_points = take n points

Creates the list extracted\_points by taking the first n elements of the list points and copying them into a new list. Note that the original list points is not modified in this process.

(++) :: [Point] -> [Point] -> [Point]

combined = list1 ++ list2

Creates a new list combined which contains the concatenation of a copy of list1 with a copy of list2. Note that the original strings are not modified. This operator actually works on any list, including on strings.

(.) :: Function -> Function -> Function

h = f . g

The function composition operator takes two functions f and g and creates a new function h so that for any argument x we will have: h x = f (g x)

Note that in algebra, you would write that equation as: h(x) = f(g(x))

Actually, the definition of this operator in the system library consists of just the following line:

(f . g) x = f (g x)

**Function Reference List – Finding elements in lists**

The built-in function find can be used to find the first element in a list that has some property.

The output of this function depends on whether there was an element with the sought property or not. If there was an element p, then the output is Just p (the literal Just followed by the value p). Otherwise, the output is the literal Nothing. The literals Just and Nothing are values of a special type called Maybe. The Maybe type has only two possible values, like Booleans. They generalize Booleans, as Nothing is just like False and Just p is like True with some extra information attached.

A value of type Maybe can be deconstructed with the special case … of pattern, which is similar to multiple-case function definitions, except that different cases are not introduced with vertical bars and instead of using = to separate the case and the action, we use the symbol ->.

In the following example, we assume that the variable value1 holds a value of type Maybe, and we want to show a message that either exhibits the element or tells the user that no such object was found.

case value1 of

Just x -> “Found element “ ++ show x

Nothing -> “No element was found”

A common use of the Maybe type is as the output of the find function described below.

find :: (Point -> Bool) -> [Point] -> Maybe Point

case find predicate points of …

The first argument of this function is another function that classifies points as acceptable (True) or not acceptable (False). Functions like that, whose return type is Boolean, are called *predicates.* So, the function find takes a predicate and a list points and searches the list for the first point that satisfies predicate (i.e., the first point p such that predicate p is True). The output of this function is Just p in case such point is found. Otherwise, the output is Nothing.

The case … of pattern can be used to deconstruct the value returned by find.

Example 1:

found = case find (> 2) [1..5] of

Just x -> True

Nothing -> False

The value of variable found will be True, because there is an element in [1..5] greater than 2, such as 3. Therefore, the function find will return Just 3.

Example 2:

always7 x = case find (== 7) x of

Just v -> v

Nothing -> 7

The function always7 will always return the value 7, but for different reasons. If 7 is in the list x, then the first value v equal to 7 will obviously be 7. If 7 is not in the list x, then the Nothing clause is activated, and the function returns 7 by default.

**Warning:** If you use find in an infinite list that does not contain any appropriate element, your program will enter an infinite loop and will never stop.

We can use multiple definitions, list comprehensions, the cons deconstruction pattern, the empty list, the Maybe type and the case…of pattern to write our own definition of the function find in the following way:

find predicate list = first candidates

where candidates = [v | v <- list, predicate v]

first [] = Nothing

first (v : other) = Just v