**Coding Overview**

These lessons use a programming language called Haskell. This overview tells all you need to know about Haskell in order to follow the lessons. If you want to know more about the language, you may look at the tutorials in http://www.haskell.org/

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 + 3yz | Needs \* between factors  2\*x\*y + 3\*y\*z |
| 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 |

A Haskell program must always have a function named main, which is the starting point.

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 and constants are just the same concept, as it is not possible to change the value of a variable after its definition.

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 |
| Define a and b | [a,b] = p | p = [a,b] |
| Define p | p = [a,b] | [a,b] = p |

**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.

**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. Important pre-defined operators are:

Arithmetic operators: + - \* /

Comparison (relational) operators: > < <= >= == /=

List manipulation: : ++

Function manipulation: $ .

**Booleans**

A Boolean (denoted Bool in type annotations) 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 words True and False literally, you generate Boolean values by using the relational operators. For example, the expression 2 > 3 is exactly the same value as the Boolean False. Literal Boolean values must start with an uppercase letter. You can also assign Boolean values to variables and use them in functions:

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

**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 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]

**Tuples**

Tuples are elements of a Cartesian product. Like in algebra, tuples are enclosed in parentheses with its coordinates separated by commas. 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)

The main difference between a tuple and a list is that a tuple has no corresponding cons operator or length function, and so you must know its length when you write your program. Another difference is that you can mix different types in a tuple, so, for example, (2,”hello”) is legal. **Ranges**

A range is a special type of list containing consecutive numbers. Examples:

[1..5] –- Same as [1,2,3,4,5]

[2,4..10] –- Same as [2,4,6,8,10]

[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

**List Comprehensions**

You can also create lists by picking those elements from another list that satisfy some condition, and then applying some function 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 ]

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

**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”

**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 an 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

**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.

**Boilerplate**

All the programs you will use follow a common pattern and need some particular lines of code at the beginning of each file. These necessary but not very interesting parts of the program are commonly referred to as *boilerplate.* You should just copy the boilerplate into each new file you make and ignore it otherwise, as it is not worth wasting time on it. Our boilerplate code is as follows:

import Geometry

import Drawing

main = drawPicture myPicture

myPicture points = <actual program goes here>

The actual program will be written as auxiliary definitions inside the function myPicture

**Eliminating parentheses**

You can reduce the number of parentheses in your expressions by replacing the starting parenthesis with a dollar sign. 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 readable 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)))

**Software Tools for Coding**

**Compiler and Libraries:** The Haskell Platform package contains the Haskell compiler and common system libraries. A compiler is a program that translates the code you write into executable machine instructions. The compiler included in the Haskell Platform is called GHC. A library is a collection of pre-defined functions that makes it easier to write programs without having to reinvent the wheel each time.

**Editor:** An editor is a program that lets you write your code and save it to a file. It usually has features such as syntax highlighting, which marks keywords and other syntax markers in different colors. You can choose among many different editors which one to use, and in these lessons we chose the following:

In Windows, we use Notepad++

In MACs, we use Atom

**Terminal:** A terminal is a window where you can give instructions to the computer directly without going through menus or intermediate apps. Terminals are commonly used by programmers, system administrators and other computing professionals because they allow them to interact with the inner parts of the computer that are usually hidden to the end users. Some systems, like MACs, come with a terminal pre-installed. In other systems, such as Windows, you need to install a third party terminal program such as Git Bash.

**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**

Haskell programs are safe from most runtime errors, but programs can still crash if you wrote an incomplete function somewhere in your code. An incomplete function is a function 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 functions 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.

**Function Reference List – Drawing Functions**

Drawing functions prepare the graphical elements of a picture to be shown on the screen. Each drawing function creates a single picture, so if you want to show several objects at once you must create a *composite picture* that combines each separate picture into a single object. The (&) operator performs this composition. For example, if drawObject1 and drawObject2 are two functions that create a picture of an object1 and some other object2 respectively, then to show both objects at once in a single picture you would write: drawObject1 & drawObject2

In summary: In order to create a single picture with all your objects, you must put an ampersand (&) between each drawing command.

Even though we do not use coordinates explicitly in many of the lessons, the computer always uses coordinates internally. The coordinates are set up so that the main window shows the region between -10 and 10 in both the X and the Y axis. A point is internally stored as a pair of numbers.

coordinates :: Picture

coordinates

Draws a coordinate frame consisting of the X axis, the Y axis and guidelines spaced by 1 unit.

coordinates’ :: Number -> Picture

coordinates’ step

Draws a coordinate frame like coordinates, except that the distance between consecutive guidelines is given by the parameter step

drawArc :: (Point,Point,Point) -> Picture

drawArc (a,o,b)

Draws a circular arc that starts at point a, has its vertex at o, and ends when the circle intersects the ray (o,b)

drawCircle :: (Point,Point) -> Picture

drawCircle (c,x)

Draws a circle with center at c and passing through x

drawLabel :: Point -> String -> Picture

drawLabel p lbl

Shows the text lbl at a location close to p

drawLabels :: [Point] -> [String] -> Picture

drawLabels pts lbls

Takes each text in the list lbls and successively shows it at the locations given in the list pts. If the two lists are not the same length, the last elements in the longer list will be ignored.

drawLine :: (Point,Point) -> Picture

drawLine (p1,p2)

Draws the line passing through p1 and p2, or draws nothing if the two points are too close to each other.

drawPoint :: Point -> Picture

drawPoint p

Draws a dot at position p

drawPointLabel :: Point -> String -> Picture

drawPointLabel p l

Draws a dot at position p and shows the text l next to it.

drawPoints :: [Point] -> Picture

drawPoints pts

Draws each point in the list pts using the function drawPoint and combines all the drawings in a single picture as if the (&) operator was used between them.

drawPointsLabels :: [Point] -> [String] -> Picture

drawPointsLabels pts lbls

Draws each label in lbls at each point location in pts using drawPointLabel repeatedly. Look at drawLabels and drawPoints for further explanation.

drawSegment :: (Point,Point) -> Picture

drawSegment (p1,p2)

Draws the segment between p1 and p2. Draws nothing if the two points are the same.

drawTriangle :: (Point,Point,Point) -> Picture

drawTriangle (a,b,c)

Draws the points a,b and c and the segments (a,b),(b,c) and (c,a). It combines all drawings in a single picture. Look at drawPoints and drawSegment for more information.

message :: String -> Picture

message text

Draws text in a single line starting at the lower left corner of the main window.

messages :: [String] -> Picture

messages lines

Draws each element of the list lines in a single line, so that the last line starts at the lower left corner of the main window.

myPicture :: [Point] -> Picture

myPicture points

This function must be defined inside our program to create whatever composite picture we want to show. The argument points is an infinite list of random points that the system provides us to use in our drawings. We extract a finite amount of points from this infinite list using the system function take. We can then use geometric functions to create interesting geometric objects based on these random points, and then use the previously listed drawing functions to create pictures of those geometric objects. So write this function as if composing a symphony: you must carefully combine all the elements together to create a harmonious result.

(&) :: Picture -> Picture -> Picture

composite = pic1 & pic2

Combines two pictures pic1 and pic2 into a single picture composite. Please refer to the beginning of this section for more information on this operator.

**Function Reference List – Geometry Functions**

Geometry functions are the most interesting functions for us. Using the geometry functions, we can create rich and complex geometric objects and apply geometric transformations to them.

Some of the functions below depend on the **Betweeness relationship**, which is defined as follows:

Given 2 points A and B, we can divide the plane into 3 regions by finding perpendicular lines to segment AB passing through A and B respectively. A point X is said to be between A and B if it lies in the middle region.

The three regions will called “beyond (A,B)”, “between (A,B)” and “beyond (B,A)”.

beyond :: (Point,Point) -> Point -> Bool

yes\_no = beyond (a,b) x

Given points a, b and x, the value of Boolean yes\_no will be True whenever point x lies in the region beyond (a,b) as explained above. Otherwise, the value of yes\_no will be False.

circle\_circle :: (Point,Point) -> (Point,Point) -> [Point]

intersects = circle\_circle (o1,x1) (o2,x2)

Given a circle with center o1 and point x1 on its circumference and another circle with center o2 and point x2 on its circumference, this function creates the list intersects consisting of all the points that lie on both circumferences. Note that intersects may be an empty list if the circle do not cross, it may contain a single element if the circles are tangent and it will contain two elements otherwise.

dilate :: Point -> (Number,Number) -> Point

p’ = dilate p (x\_dilation,y\_dilation)

Performs a dilation of point p with respect to the origin. The coordinates will be dilated by x\_dilation horizontally and by y\_dilation vertically. The point p’ corresponds to the dilated point.

dist :: Point -> Point -> Number

d = dist p q

Given two points p and q, the number d will be the Euclidean distance between them.

find\_apart :: Point -> (Point,Point) -> [Point]

y = find\_apart x pts

The point y represents the first point in the list of points pts that is apart enough from the given point x to be clearly distinguishable from it. If no such point exists, this function will fail, making your program crash and produce an error message. Apart enough by default is 0.01 units.

line\_circle :: (Point,Point) -> (Point,Point) -> [Point]

intersects = line\_circle (a,b) (o,x)

Given a line passing through points a and b and a circle with center at point o and passing through point x, this function creates the list intersects consisting of all points that lie on both the line and the circle. Note that this list may contain either 0, 1 or 2 elements.

line\_line :: (Point,Point) -> (Point,Point) -> [Point]

intersects = line\_line (a,b) (c,d)

Given a line passing through points a and b and another line passing through points c and d, this function creates the list intersects consisting of either no point (if the lines are parallel) or the point at the intersection of both lines (if they cross each other.) Note that this function considers that a line is always parallel to itself and returns no point in that case.

midpoint :: Point -> Point -> Point

m = midpoint a b

The point m is the midpoint of the segment joining points a and b

mirror :: (Point,Point) -> Point -> Point

p’ = mirror (a,b) p

Given a point p and a line passing through points a and b, the point p’ is the reflection of p with respect to line (a,b)

parallel :: (Point,Point) -> Point -> (Point,Point)

(a’,b’) = parallel (a,b) c

Points a’ and b’ lie on a line passing through point c that is parallel to line (a,b)

perpendicular :: (Point,Point) -> (Point,Point)

(c,d) = perpendicular (a,b)

Given a line passing through points a and b, the points c and d lie on the perpendicular to line to line (a,b) that passes through point b

projection :: (Point,Point) -> Point -> Point

x’ = projection (a,b) x

The point x’ is at the intersection of line (a,b) with the perpendicular line passing through x

quadrilateral :: [Point] -> (Point,Point,Point,Point)

(a’,b’,c’,d’) = quadrilateral [a,b,c,d]

The points a’,b’,c’ and d’ are a reordering of points a,b,c and d in such a way that the segments between consecutive points do not cross each other. The resulting figure is a proper quadrilateral.

rotate :: Number -> Point -> Point

p’ = rotate angle center p

The point p’ is obtained by rotating point p counterclockwise angle degrees around center

translate :: (Number,Number) -> Point -> Point

p’ = translate (vx,vy) p

The point p’ is computed by moving point p horizontally by vx units and vertically by vy units.

**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 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. In our lessons, however, main must always be defined as follows: main = drawPicture myPicture

Both drawPicture and myPicture are functions too. The function drawPicture is pre-defined in our library of drawing functions. It uses myPicture to create whatever picture we want and converts that picture into screen pixels using several auxiliary system functions. The function myPicture is our *de facto* entry point. This function should create a composite picture containing all the objects we want to show on the screen. Look in the Drawing Functions section for more information on this function.

**Note:** In the functions below, you can replace [Point] with any other type of element, and most functions will still work the same.

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

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

case find predicate points of

Just p -> message $ “Found point “ ++ showpoint p

Nothing -> message “No point found”

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 example above shows how you can check for each case by enclosing the function within the special case … of construct. The syntax for checking each case is similar to multiple-case function definitions, except that instead of using = to separate the case and the action, we use the symbol -> instead.

In summary, the output of this function may be one of the two patterns Just p or Nothing. Values of this type generalize Booleans, as Nothing is just like False and Just p is like True with some extra information attached. Instead of Bool, this type is called Maybe Point.

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.

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

combined = string1 ++ string2

Creates a new list combined which contains the concatenation of a copy of string1 with a copy of string2. Note that the original strings are not modified. This operator actually works on any list, not just 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)