Notes on Haskell for CS 320

Last Update: April 2, 2019

Notice: This is a terse document that contains a lot of details. Do NOT expect to understand all of it the first time you read it! Start instead with an initial skim of the material to get a sense of what is here, and then plan/expect to come back on a regular basis during the rest of the term to fill in the gaps as you need more information (and learn more) about Haskell in particular, and programming languages in general.

Introduction

We will be using Haskell this term to introduce and demonstrate the characteristics of functional programming languages. No previous experience with Haskell is assumed, and we will only be using a small subset of the language. There are plenty of resources for learning more about Haskell on the Internet (including https://www.haskell.org, http://learnyouahaskell.com, and http://book.realworldhaskell.org). But beware: the full Haskell language has many advanced features that (a) I will not be using in class; (b) you will not need in any assignments; and (c) might be confusing, or encourage you to develop unnecessarily complicated solutions for assignments. For the purposes of this course, all the information that you need about Haskell should be covered on the slides, and other course materials, including this page. While you are welcome (and I typically would encourage you) to use other resources, I noticed that some of the students who did this in past terms ended up trying to use more advanced or complicated techniques than were necessary, which sometimes made the problems harder to solve! If you find that you're stuck on a problem involving Haskell and don't find a solution on the slides or elsewhere in the course materials, please remember that you are always welcome to ask for help!

Basic Syntax

A word about fonts: Play careful attention to the use of fonts in this document. In particular, we use `Courier` for fragments of Haskell programs, filenames, expressions and types, to distinguish these items from regular text. For example, in the sentence "the `length` function calculates the length of a list", there is an important reason for using different fonts for the two occurrences of the word "length"; the first references the syntax of a particular function in Haskell, while the second describes its semantics (i.e., its meaning: what the function does). You will also see some use of **bold**, both to mark sections and to emphasize some important technical terms.

Haskell source files: We will be creating Haskell programs using the "literate Haskell" format: programs are stored as plain text files with file names ending in "`.lhs`". Lines that are part of the program are marked with a ">" character at the start of the line; all other lines are ignored (i.e., they are treated as comments). There must, however, be at least one blank line between program lines and comment lines, as in the following example:

```
Here are some Haskell definitions (but this line is just a comment):

> welcome = "hello, world"
> number  = 42

This is another comment, separated from the code by a blank line.
```

Comments: One line comments are introduced by "`-- `" and extend to the end of the line on which they appear. Sections of code may be also be commented out by enclosing them between "`{-`" and "`-}`" markers:

```
> six = 2 * 3  -- you could add a comment here
> eight = 4 * 2  {- this is also a comment -}
```

Modules: Haskell programs that use code in multiple files treat each file as a separate `module`. A file called `X.lhs` should begin with a line of the following form:

```
> module X where
```

The first character in any Haskell module name, and hence in the corresponding source file name, should be a capital letter. If your module requires types or functions that are defined in another file/module, you can specify this by adding import declarations at the top of the file:

```
> module X where     -- the module X
> import Y           -- uses items defined in Y.lhs
> import Z           -- as well as items defined in Z.lhs
```

You cannot use constructs like this if two or more of the modules listed define a variable using the same name. The full Haskell language has ways to deal with this, but we'll be able to avoid this issue by choosing different names for the items that we define.

Indentation: Haskell programs use layout/indentation to reflect program structure. If you get the indentation wrong, then the Haskell compiler or interpreter may not be able to read the code in the way you intended. The full details are complicated, but two general rules will get you a long way: (1) make sure all the (top-level) definitions in any given Haskell file start in the same column [the examples here all start in Column 3 after a "> " in Columns 1 and 2]; (2) if a single definition or expression spans multiple lines, make sure the later lines are indented more than the first line.

```
This is valid:         -- But this would not be:

> hi                   -- > hi
>   = "hello, "        -- > = "hello, "
>     ++ "world"       -- > ++ "world"
```

I recommend that you use a fixed width font when writing or reading Haskell so that it is easy to see the indentation. Haskell assumes that tab stops appear every 8 spaces; avoid using tab characters if your editor displays tabs differently.

Identifiers: The names of variables and types in Haskell are written as sequences of alphabetic characters and numeric digits, beginning with a letter. Haskell distinguishes between upper and lower case, so "`avalue`" and "`aValue`" are treated as distinct identifiers. Haskell identifiers beginning with capitals (upper case letters) are reserved for the names of modules, types, and constructor functions (see below), while identifiers beginning with lower case letters are reserved for variables and type parameters.

Using a Haskell Interpreter

We will be using an interpreter to develop and run Haskell programs. There are two suitable interpreters installed on the departmental linux machines: one is called `hugs` and the other is called `ghci`. For our purposes, either one of these will work just as well as the other.

Getting started: To start an interpreter at the Linux command prompt, just type the corresponding interpreter name. If you'd like the interpreter to start by reading the definitions in a particular file, say `X.lhs`, then add that filename after the `hugs` or `ghci` command.

Loading definitions: If you want to load the definitions from a particular file (again let's say `X.lhs`) once the interpreter has started, then you can use the command "`:l X.lhs`" from the interpreter prompt. (The "`l`" character here is just the first letter of "load").

Changing definitions: If you change the definitions in the file "`X.lhs`" while the interpreter is running (for example, by opening "`X.lhs`" in a text editor in a different window and saving the modified file), then you can tell the interpreter to reload the file using the command "`:r`" at the interpreter prompt. (The "`r`" character her is just the first letter of "reload".)

The standard prelude: Even if you don't specify a file yourself when you start the interpreter, it will still load the definitions for a fairly large collection of builtin functions and operators that Haskell programmers refer to as **the standard prelude**. Think of this as a library of builtin features that you can extend and use in your own programs.

Leaving the interpreter: If you want to quit the interpreter, you can use the command "`:q`" at the interpreter prompt. (You've probably spotted the pattern here: the "`q`" character here is just the first letter of "quit"!)

The REPL: Under normal conditions, the interpreter works by **read**ing an expression that you type at the prompt, **eval**uating that expression, **print**ing out the result, and then **loop**ing back for the next expression. We refer to this as a read-eval-print loop or REPL. Here is a quick example to show what this looks like in practice on one of the departmental machines:

```
ada:~% ghci
GHCi, version 7.10.3: http://www.haskell.org/ghc/  :? for help
Prelude> 1+2
3
Prelude> length "hello"
5
Prelude> :q
Leaving GHCi.
ada:~%
```

Although you can probably install either of these Haskell interpreters on your own computer, it will probably be easier just to login to `linux.cs.pdx.edu` using `ssh` (for Mac, Linux, or other Unix-based systems) or `PuTTY` (for Windows). But if you really want a Haskell interpreter for your own computer, or you just want to learn more about Haskell, then I recommend visiting https://www.haskell.org. Follow the Downloads link for a copy of `ghc` (and, specifically, the `ghci` interpreter) for your computer: there are several options available, all of which should be suitable for the purposes of this class, although "The Haskell Platform" might be the easiest option for most people. If your computer is running Linux, then you might want to check your package manager to see if there is a version of `hugs` or `ghc` for your machine: quite a few distributions provide one or both of these.

Setting an editor: If you use `hugs` in a single terminal window (e.g., over `ssh` or `PuTTY`), you may find that it is convenient to be able to switch in to a text editor so that you can make changes to your code without having to leave the interpreter. The `hugs` interpreter was specifically designed to be used in this way and it will automatically reload your code when you exit the editor. However, you can only use this feature if you configure `hugs` to specify which editor you want to use. The following command allows you to set the editor (in this case, `vi`) as part of the command line when you start `hugs`:

```
hugs -E"vi +%d %s" NameofFile.lhs
```

This takes advantage of a special feature of `vi`/`vim`/`emacs` that lets you specify the line where you want the editor to start (by writing `%d` at the appropriate point where the line number should go). if your editor doesn't support this, or if you don't want that particular feature, you can just specify the name of the editor command (e.g., `hugs -Enano`).

If you prefer not to have to type a command like this every time you start the interpreter, then you might want to add something like the following to the `.cshrc` file in your home directory:

```
alias hugs 'hugs -E"vi +%d %s"'
```

and then the appropriate editor setting will be made every time you run `hugs`. Of course, you will need to adapt these instructions if you use a different editor, or if you are using a different shell. For example, if you are using `bash`, then you should add a line like the following to your `.profile` file:

```
alias hugs="hugs -E\"vi +%d %s\""
```

Please let me know if you need help setting this up. As an alternative, if you are sitting in front of one of the departmental Linux machines, then you might prefer to open two windows so that you can look at the source code for your program in one, and run `hugs` or `ghci` in the other. Just remember to save your changes after you have edited the source code, and then to use the `:r` command to reload the file in the interpreter window.

Expressions

In a functional programming language, computation is performed by evaluating expressions, so it is important to know the syntax that us used for writing expressions. Here are some examples:

Booleans: The two standard truth values are written as `True` and False. If `x` and `y` are boolean values, then `x && y` is `True` only if both `x` and `y` are `True`, and `x || y` is `True` if either `x` or `y` is `True`. Finally, `not x` is `True` if `x` is `False`, and vice versa.

Numbers: Numbers are written in the usual way and can be manipulated using the standard arithmetic operators like `+`, `-`, and `*`. The usual precedence rules apply (e.g., multiplication has higher precedence than addition) but you can always add parentheses if you need them. For example, the expression `(1 + 2) * 3` will evaluate to `9`, but `1 + 2 * 3` will evaluate to `7`.

The syntax for the integer division operators may look a little strange the first time you see it: ``7 `div` 2`` evaluates to `3` (and, ``7 `mod` 2`` evaluates to `1`, which is the remainder after dividing `7` by `2`). Note that both of these examples use backtick characters around the operator names: you probably won't need these operators during this class ... but if you do, then be careful not to confuse the backtick character with the normal quote character (which you might also use as an apostrophe).

Comparisons: Two values of the same type can be compared using the familiar infix operations ==, <, <=, >, and >=. There is also a not-equal operator, which is written as `/=`. For example, `1 < 2` and `1 /= 2` will both evaluate to `True`. All of these operations have lower precedence than the arithmetic operations, but if you ever forget the precedence rules, you can always add some parentheses ... Note that you never need to write expressions like `e == True` or `e == False`: the first of these can be simplified to `e` while the second can be simplified to `not e`.

Functions: Function calls begin with the name of the function followed by one or more arguments. For example, we write `max 3 4` to calculate the maximum of the two values `3` and `4`. Note that you do not need to put parentheses around the arguments in examples like this. More generally, function application (i.e., writing a function next to an argument) has a higher precedence than any infix operator, so an expression like `max 2 3 + 4` will be treated as if you'd written `(max 2 3) + 4`. If you want to evaluate `max 2 (3 + 4)`, then you'll have to write the parentheses in yourself.

Operators: Infix operators like `+`, `<`, or `==` are really just functions that are written using special syntax (the function goes between its arguments instead of before). If you want to use an operator symbol as a function with prefix syntax, just wrap the operator in parentheses. For example, `x + y` can also be written as `(+) x y`. Conversely, you can use an identifier as an operator by enclosing it between backticks, as in the previous example with `div` (which could also have been written `div 7 2`). As a further illustration, `max 2 3` can also be written as ``2 `max` 3``: there is no difference in semantics, so you can choose whichever notation you find easiest to understand.

Tuples: You can combine multiple component values, separated by commas, into a single tuple that is enclosed in parentheses. For example `("hello", True)` is a pair containing the string `"hello"` and the Boolean value `True`. The standard prelude includes some functions for manipulating tuples. For example, the `fst` and `snd` functions will return the first and second components of a pair, respectively, so the expression `fst ("hello", True)` will evaluate to `"hello"` while `snd (True, fst (1, 2))` will evaluate to `1`. Note that it is not necessary for all of the components of a tuple to have the same type.

Lists: You can combine any number of values *of the same type* in to a list. The empty list is written `[]`. More generally, you can write a list by enumerating the elements that it contains, as in `[True, False]`, which is a list of all the Boolean values, or `[1, 3, 5, 7, 9]`, which is a list of the odd integers less than 10. The standard prelude provides a lot of useful operators and functions for working with lists, including:

- The `++` operator allows us to append one list to another. For example, `[1,2]++[3,4]` evaluates to `[1,2,3,4]`.
- The `length` function returns the length of its input list. For example, `length [1,2,3,4]` evaluates to `4`.
- The `reverse` function returns a new list that has the elements of its input in reverse. For example, `reverse [1,2,3,4]` evaluates to `[4,3,2,1]`.
- The `null` function tests to determine whether its input is the empty list. For example, `null []` evaluates to `True`, but `null [1,2,3,4]` evaluates to `False`.
- The `head` function returns the first value in a list. For example, `head [1,2,3,4]` evaluates to `1`.
- The `tail` function returns the remaining list of elements after the head. For example, `tail [1,2,3,4]` evaluates to `[2,3,4]`.
- The `take` function returns an initial section of a list. For example, `take 3 [1,2,3,4,5]` evaluates to `[1,2,3]`.
- The `:` operator (pronounced "cons", which is short for "construct a list") builds a new list by adding one new element on to the front of another list. For example, `1 : [2,3,4]` evaluates to `[1,2,3,4]`. In fact, the expression `[1,2,3,4]` is really just a convenient shorthand for `1 : (2 : (3 : (4 : [])))`; we build up the list by adding its elements, one at a time, to the empty list. Thanks to Haskell's precedence rules, the parentheses in this last example are not actually required (the `:` operator "groups to the right"), so the same expression can also be written as `1 : 2 : 3 : 4 : []`.

It is worth repeating the earlier comment that all of the elements in a single list must have the same type. For example, `[1, True]` is not valid because it attempts to combine an integer and a Boolean in a single list. On the other hand `[(1, True), (2, False)]` is a valid list that has precisely two elements, each of which is a pair containing an integer first component and a Boolean second component.

Characters and Strings: Individual character values are written by surrounding a single character between quotes: the first character of the English alphabet is `'A'`, while the last letter, this time in lower case, is `'z'`. Special syntax is provided for characters like `'\n'` (a new line character), `'\t'` (a tab character), and `'\\'` (a backslash character).

Simple strings in Haskell are written by putting text between double quotes, as in `"hello, world"`. But all strings in Haskell are really just lists of characters. For example, the string `"hello"` is just a more convenient way to write `['h','e','l','l','o']`. One nice benefit of this is that all of the operators for working with lists can also be used with strings. For example, `fst "hello"` evaluates to `'h'`; `length "hello"` evaluates to `5`; and `"hello, " ++ "world"` evaluates to `"hello, world"`.

Types

What is a type? Types provide a way to refer to specific sets of values. A lot of the types that are used in Haskell programs are written with names that start with a capital letter, including: `Bool` (the type containing the two Boolean values `True` and `False`), `Int` (containing 32 bit integers), `Integer` (arbitrary precision integers), `Float` and `Double` (single and double precision floating point numbers), `Char` (characters), and `String` (strings, which are actually just lists of characters).

Types in Haskell: Haskell is a **strongly typed** programming language, which means that every expression has an associated type and evaluates to a value of that type (assuming that its evaluation terminates). Types play important roles in Haskell, helping to document behavior, to catch program errors, to determine how expressions are evaluated, and to enable optimizations or other implementation details. Haskell implementations use **type inference** to calculate the type of any expression automatically, immediately reporting an error if a problem is detected.

The :: or "has type" symbol: Haskell uses the double colon symbol, `::`, to indicate that an item on the left has the type given on the right. For example, `1 :: Integer` or `not True :: Bool` can be read as specifying that the number `1` has type `Integer`, and that the expression `not True` will evaluate to a Boolean result. If we want to talk about the types of expressions involving variables, then we will typically need to make some assumptions about the types of those variables. For example, the statement `not x :: Bool` (i.e. "`not x` has type `Bool`") is valid only if `x :: Bool` (i.e., "`x` has type `Bool`"). Notice that, in this example, it doesn't matter whether `x` is actually `True` or `False`: knowing the value of `x` will have an effect on the *value* of `not x`, but it has no effect on the *type* of `not x`.

List and tuple types: Haskell provides special notation ("syntactic sugar") for list and tuple types. In particular, if `r`, `s`, and `t` are arbitrary types, then:

- `[r]` is the type of lists of elements of type r. For example, `[1,2,3] :: [Integer]`, and `[True,False] :: [Bool]`, and `[[1],[2,3]] :: [[Integer]]`.
- `(r,s)` is the type of pairs whose first component is of type `r` and whose second component is of type `s`. For example, `(1, True) :: (Integer, Bool)`, and `(('a',2),(True,3)) :: ((Char, Integer), (Bool, Integer))`.
- `(r,s,t)` is the type of tuples with three components, of types `r`, `s`, and `t`, respectively. For example, `(1, True, 'a') :: (Integer, Bool, Char)`.

The last two examples here can be generalized to tuples with more than three components in the obvious way. (Although, once you start to get too many components, a tuple can become unwieldy and it might be better to use a different approach.) In fact, there is also a special case for tuples with zero components: there is a single value, written `()` and often called "unit", whose type is also written `()` and often called "the unit type". While it might seem confusing to use the same symbol for two different things, this shouldn't be a problem in practice because the context should make it clear when we are referring to a value or when we are referring to a type. For example, there is only one reasonable way to make sense of the (valid) assertion that `() :: ()`.

Function types: The type of a function that maps values of type `r` to values of type `s` is written `r -> s`. For example, the `not` operator has type `Bool -> Bool`. Functions that take more than one argument can be described using types with multiple arrow symbols. For example, an operator for comparing two `Integer` arguments (e.g., a less than or an equality test) would have type `Integer -> Integer -> Bool`. The arrow symbol is treated as an infix operator that groups to the right. This means that the preceding type with two arguments is equivalent to `Integer -> (Integer -> Bool)`: a value of this type is a function that takes one `Integer` argument and returns a function of type `(Integer -> Bool)` as its result. If we then pass a second `Integer` argument to this function, then we get the expected `Bool` result value. (Corresponding to this, function application groups to the left. When you write `f x y`, it is treated exactly as if you had written `(f x) y`: first apply `f` to argument `x`, giving another function, `f x`, that you can apply to `y` to get a result. This technique for encoding functions of multiple arguments as functions that take one argument at a time is known as **currying**.)

Polymorphic types: Identifiers in types that begin with a lower case letter are **type variables**, that can be instantiated in different ways to give different results. For example, the type of the `length` operator is written `[a] -> Int`, where the type variable `a` in the input type can be instantiated to `Char` (so that `length` can be used as a function of type `[Char] -> Int` to calculate the length of a list of characters) or to `Int` (so that `length` can be treated as a function of type `[Int] -> Int` to calculate the length of a list of integers). When a single function can be used with multiple types of values like this, we often say that it is a polymorphic function, or that it has a polymorphic type. (The word "polymorphism" means "many shapes"). As another example, the type of the ++ operator can be written as `[a] -> [a] -> [a]`, indicating that it can be used to append any pair of lists together, so long as the elements in both inputs are the same, and in which case the final result will also be a list of the same type.

Parameterized types: List types, tuples types, and function types are examples of parameterized types that have a special syntax in Haskell. More generally, a parameterized type is described by writing a capitalized type name followed by one or more parameter types. For example, the standard prelude includes the following two examples of parameterized types:

- Types of the form `Maybe t` are used to represent optional values of type `t`. There are two ways to construct values of such a type: `Nothing` represents the missing value, and `Just x` indicates that a value `x` (of type `t`) has been provided. For example, we can use `Nothing :: Maybe Integer`, and `Just True :: Maybe Bool`. Strictly speaking, `Nothing` and `Just` are actually polymorphic values of type `Maybe a` and `a -> Maybe a`, respectively.
- Types of the form `Either a b` correspond to a (disjoint) union of the types `a` and `b`. More precisely, any value of type `Either a b` must be either a value of the form `Left x` (for some value `x` of type `a`) or a value of the form `Right y` (for some value `y` of type `b`). For example, `Left True`, `Right 1`, and `Left False` are all values of type `Either Bool Integer`. In these cases, the initial `Left` or `Right` symbol serves as a "tag" that distinguishes values in the left/first argument of the `Either` type from those in the right/second argument.

We will see that it is possible to write some very general, reusable code in Haskell using parameterized types and polymorphic functions!

Definitions

What should you do if you want to work with a new type of values or a new function/operator that is not built in as part of the standard prelude or an existing library? Haskell, of course, provides a way to define new items to fill these gaps ...

Data type definitions: The following sequence of steps will guide you through the process of defining a new **data type** in Haskell. We'll use a running example — a type of binary tree structure — to illustrate this.

- The first step is to write the `data` keyword, followed by *the name of the new type* (you should use an identifier that begins with a capital letter):

  ```
  data Tree ...
  ```

- Next, add names for any *parameters*, using spaces as separators, followed by an = sign. It's common not to have any parameters at all, but if you do need them, then each one should be a type variable (i.e., an identifier beginning with a lower case letter). For our example, we'll use a single parameter `t` representing the type of values that will be stored in the tree.

  ```
  data Tree t = ...
  ```

- Now think about all of the different kinds of value that you can expect to find in the type you're defining and add a *constructor* for each one (a constructor is represented by an identifier beginning with a capital). Pick a different name for each one and use vertical bar `|` characters as necessary to separate them. For our our tree example, there are two types of value: `Leaf` nodes and `Fork` nodes:

  ```
  data Tree t = Leaf ... | Fork ...
  ```

- Finally, for each constructor, think about the *specific data fields* that you would expect to be associated with a value of that form, and add types for each of those components. For our example, there is nothing extra to add to a `Leaf` node, but each `Fork` should have a value of type `t` as well as two subtrees of type `Tree t`, one representing the left subtree and another representing the right:

  ```
  data Tree t = Leaf | Fork (Tree t) t (Tree t)
  ```

  Note that you can list the arguments of the `Fork` constructor in whatever order you like; I chose to put the `t` component in between the two subtrees here. And note also that the parentheses are not optional here: if you left them out and wrote `Fork Tree t t Tree t`, then it would look (incorrectly) as if you were trying to define a `Fork` node with five components.

Now you have a complete data type definition, and you can construct values of the new type by combining the constructors in appropriate ways. For example, the following are all valid trees according to this definition (the types are not required here; they are just included for documentation):

- `Leaf :: Tree Int`
- `Fork Leaf 'h' Leaf :: Tree Char`
- `Fork (Fork Leaf 1 Leaf) 3 Leaf :: Tree Integer`
- `Fork (Fork Leaf 1 Leaf) 3 (Fork Leaf 5 Leaf) :: Tree Integer`

More specifically, given the definition of `Tree` above, the `Leaf` and `Fork` symbols are treated as **constructor functions**:

- `Leaf :: Tree t`
- `Fork :: Tree t -> t -> Tree t -> Tree t`

For some additional examples, here are how the `Bool`, `Maybe`, and `Either` types are defined in the standard prelude:

```
data Bool = False | True

data Maybe a = Nothing | Just a

data Either a b = Left a | Right b
```

For example, every `Bool` is either `False` or `True`, and no additional information is needed in either case. On the other hand, every value of type `Either a b` is either something of the form `Left x` (for some value `x` of type `a`) or `Right y` (for some value `y` of type `b`).

And here are some additional examples of user defined types:

```
> data Month = Jan | Feb | Mar | Apr | May | Jun | Jul | Aug | Sep | Oct | Nov | Dec

> data Date  = Date Month Int Int  -- use integers for the day and year components

> data Prop  = AND Prop Prop   -- propositional logic abstract syntax trees
>            | OR Prop Prop
>            | NOT Prop
>            | VAR String
```

```
>              | TRUE
>              | FALSE
```

Notes: (1) It is possible, and common among experienced Haskell programmers, to use the same name (`Date` in the examples above) as both a type name and a constructor name. You can distinguish between the two uses by looking at the context in which any given use of the name appears. (2) The layout of these definitions is not too critical. For simple examples, you can often list all of the constructors on one line. For more complex examples, it can be clearer to spread the code over multiple lines, as in the definition of `Prop` above.

Deriving: In practice, you will often see a phrase like `deriving Show`, `deriving (Eq)`, or `deriving (Eq, Show)` at the end of a data type definition. This is just a mechanism for indicating that you would like the interpreter to generate code that will let you convert values of the new type in to printable strings that the interpreter can then use to display results (`Show` specifies this) or for testing two values of the new type to see if they are equal (`Eq` specifies this). For example, we could specify a type of `Shape` values, each of which is either a circle or a rectangle, by adding the following definitions to a file:

```
> data Shape = Circle Double | Rectangle Double Double
>              deriving (Show, Eq)
```

After the file containing this definition has been loaded, we can use values of the new type at the interpreter prompt like this:

```
Main> Circle 1 == Circle 2
False
Main> Rectangle 1 2
Rectangle 1.0 2.0
Main>
```

If the `deriving` part of the type definition had not been included, then we would see some error messages like this instead:

```
Main> Circle 1 == Circle 2
ERROR - Cannot infer instance
*** Instance   : Eq Shape
*** Expression : Circle 1 == Circle 2
Main> Rectangle 1 2
ERROR - Cannot find "show" function for:
*** Expression : Rectangle 1 2
*** Of type    : Shape
Main>
```

Type synonym definitions: If you just want to give a new name for an existing type of values, then you can use a type synonym definition of the form `type TypeName parameters = t`, as in the following examples:

```
type String = [Char] -- as defined in the standard Prelude

type IntSet = Int -> Bool

type Distance = Double

type IntOrBool = Either Int Bool

type Mapping a b = [ (a, b) ]
```

The primary reason for defining a type synonym is to give shorter and/or more intuitive names for certain types.

Function definitions: Functions are defined by writing a sequence of equations, each of the form `func args ... = result`, where `func` is the name of the function (an identifier beginning with a lower case letter), `args` is a list of (zero or more) arguments (separated by spaces), and `result` is an expression specifying the result of the function. For example:

```
> square x = x * x

> pythagorus x y = sqrt (square x + square y)

> factorial n = product [1..n]
```

It is good practice (but usually not strictly necessary) to include a type signature with each function that you define to document the type of the associated function:

```
> square   :: Double -> Double
> square x  = x * x

> pythagorus    :: Double -> Double -> Double
> pythagorus x y = sqrt (square x + square y)

> factorial     :: Integer -> Integer
> factorial n    = product [1..n]
```

Pattern matching: If an equation only applies to a certain form of function input, you can specify that by writing an appropriate pattern on the left hand side. The following example shows how we can use this to extract (and then manipulate) the components of a pair:

```
> distFromOrigin        :: (Double, Double) -> Double
> distFromOrigin (x, y) = pythagorus x y
```

If there are multiple forms of value to consider, then you can use multiple equations, each dealing with a different combination of inputs. (Note that all of the equations for a given function must have the same number of arguments.) For example:

```
> greet          :: Maybe String -> String
> greet (Just x) = "Hello, " ++ x
> greet Nothing  = "I don't know your name"
```

Because there are precisely two forms of `Maybe` value (one using `Just` and one using `Nothing`) and two corresponding equations in the definition above, we can be confident that it will work with *any* input value of type `Maybe String`. Pattern matching can also be used with multiple arguments, as in the following function for testing whether two `Maybe` values are the same (or not):

```
> sameAs                    :: Maybe a -> Maybe a -> Bool
> sameAs Nothing  Nothing  = True   -- two Nothings are the same
> sameAs (Just x) (Just y) = x==y   -- two Justs are the the same if their fields are equal
> sameAs x        y        = False  -- any other combination of values are not equal
```

This definition also illustrates some other useful details: (1) Equations are matched from top to bottom, so the third equation is only used for inputs that do not match the first two; this means that we will only use the third equation when one of the arguments is `Nothing` and the other is built using `Just`, in which case they cannot be the same. (2) The variables used in one equation are completely independent from the variables in any other equation, so the `x` and `y` in the second equation have nothing to do with the `x` and `y` in the third. (Some people might argue that we should use different variable names in each one to avoid confusion, but this is not required.)

Pattern matching on lists: As an important special case, because every list is either empty (and hence matches the pattern `[]`) or nonempty (and hence matches a pattern of the form `(x:xs)`), we can define functions that work on arbitrary list values by using function definitions like the following:

```
> f []     = ... result for empty list ...
> f (x:xs) = ... result for nonempty list with head x, tail xs ...
```

For example, the standard prelude function for adding up all of the numbers in a list could be defined as follows:

```
> sum []     = 0
> sum (x:xs) = x + sum xs
```

Notice the use of **recursion** here to compute the sum of the tail of the list, `xs`, in the second equation, adding that to the number, `x`, at the head of the list to obtain the total of all the list elements.

Nested patterns: Multiple constructor functions and variables can be combined to form more complicated patterns. For example, `(x:y:zs)`, which is equivalent to `(x:(y:zs))`, is a pattern that matches lists with at least two elements, while `(Just x : ys)`, which is equivalent to `((Just x) : ys)`, is a pattern that matches nonempty lists of `Maybe` values whose first element is constructed using `Just`. In a similar way, we can also use patterns like `[x]` (matching a list with a single element) or `('a':cs)` (matching a list of characters that starts with an `'a'`), or even just `"hello"` (matching precisely that sequence of five characters, and no other string).

Guards: Sometimes it is useful to include Boolean tests (sometimes called "guards") in addition to pattern matching, resulting in equations that look something like the following:

```
> f x y | condition1 = ... result if condition1 is True ...
>       | condition2 = ... result if condition2 is True ...
>       | ...
>       | conditionN = ... result if conditionN is True ...
```

In examples like this, each of the different conditions is tested from top to bottom, either returning the result associated with the first condition that is `True`, or continuing on to any subsequent equations if none of the conditions holds. (And if there are no subsequent equations, then the evaluation will terminate with an error message.) As a concrete example, the following code shows the standard algorithm for inserting an `Integer` at the appropriate point in an ordered list:

```
> insert                     :: Integer -> [Integer] -> [Integer]
> insert x []                 = [x]
> insert x (y:ys) | x <= y    = x : y : ys
>                 | otherwise = y : insert x ys
```

Note that the `otherwise` symbol used here is defined in the standard prelude with value `True`, which ensures that the second condition in this particular equation will always succeed.

Layout: In the definitions above, you might have noticed that I like to write my code so that all of the `::` and = symbols in a given function definition end in the same column. But this is just a matter of style: I think it makes the code easier to read, but you may not agree with that, and it is not required. That said, always be careful in the way you present your code: clearly written code will be much easier to read, understand, debug, and maintain, than code whose structure and layout is sloppy and confused.

Special Kinds of Expression

We end with a tour of some special forms of expression:

Conditionals: An expression of the form `if e then t else f` evaluates to `t` if `e` is `True`, but otherwise evaluates to `f`. For example, `if x then y else False` will evaluate to `True` only if both `x` and `y` are `True`. (This expression can also be written as `x && y`; of course, there is a corresponding "or" operation on `Bool` values too, written `x || y`.)

Case expressions: A `case` expression allows you to perform pattern matching within an expression instead of as part of a function definition (although the two forms are interchangeable, so whichever one you use is largely a matter of convenience). For example, the definition:

```
> length []     = 0
> length (x:xs) = 1 + length xs
```

can also be written as:

```
> length zs = case zs of
>               []     -> 0
>               (x:xs) -> 1 + length xs
```

The general form of a `case` expression is something like the following:

```
    case ...expr to match against patterns ... of
        pattern1 -> ... result if pattern1 matches ...
        pattern2 -> ... result if pattern2 matches ...
        ...      -> ...
        patternN -> ... result if patternN matches ...
```

Make sure that each of the patterns in a case expression is indented (by the same amount) beyond the column where the `case` keyword appears to ensure that all of the alternatives are treated as part of the same case expression.

Let expressions: An expression of the form `let x = e in e1` can be used to set the variable `x` to the value of expression `e` while evaluating expression `e1`. For example, `let x = y+1 in x*x` is equivalent to `(y+1)*(y+1)`, except that the latter requires two addition operators, while the former only requires one. Let expressions of this kind are also useful as a way to break complicated expressions in to smaller parts by giving names to the values of subexpressions.

Let expressions can also be used to perform pattern matching. For example, the expression `let (x, y) = f 1 in x` will return the first component of the pair that is produced as the result of `f 1` (assuming some suitably defined function `f`). Be careful if you use a `let` expression with a pattern that might not match. For example, an attempt to evaluate `let (x:xs) = g 2 in x * sum xs` will fail with an error if the function call `g 2` returns an empty list as its result.

Lambda expressions: A lambda expression (also called a lambda abstraction) is an expression of the form `\pats -> expr`, that represents a function. The `pats` component here is a list of one or more patterns, and `expr` describes the result produced by the function (which typically but not always involves variables introduced in the patterns). A very common case occurs when the list of patterns is just a single variable. For example `\x -> x + 1` is a function that returns the successor of its integer input; `\x -> 2 * x` is a function that returns double the input value; and `\x -> [x]` is a function that returns a singleton list containing the value of its input. Lambda expressions like these are useful in situations where you want to write a simple function value without having to give it a name and provide a top-level definition for it. We'll see some more compelling examples when we come to study higher order functions later in the term.

Missing Features or Questions?

If there are any other features of Haskell that you think you might need to know about for this course, or if there are any details in all the text above that don't quite make sense, please ask for help!