# Gentle Introduction to Haskell 98

Online Supplement

10 april. 2005

# Gentle Introduction to Haskell 98, Online Supplement
Part 1
Covers Section 2

## Introduction

This is a programming supplement to `A Gentle Introduction to Haskell', version 98, by Hudak, Peterson, and Fasel. This supplement augments the tutorial by providing executable Haskell programs which you can run and experiment with. All program fragments in the tutorial are found here, as well as other examples not included in the tutorial.

You should have a copy of both the `Gentle Introduction' and the Haskell 98 report itself to make full use of this tutorial. Although the `Gentle Introduction' is meant to stand by itself, it is often easier to learn a language through actual use and experimentation than by reading alone. Once you finish this introduction, we recommend that you proceed section by section through the `Gentle Introduction' and after having read each section go back to this online tutorial. You should wait until you have finished the tutorial before attempting to read the report. We suggest that you run this code using Hugs, a small Haskell interpreter. Everything is available for download at haskell.org.

This tutorial does not assume any familiarity with Haskell or other functional languages. Throughout the online component of this tutorial, we try to relate Haskell to other programming languages and clarify the written tutorial through additional examples and text.

## Using Hugs

If you are using this with Hugs, here are the commands you will need to use. Start up hugs and change to the directory containing these files using `:cd'. Load each part using `:l part1' (or whatever part is next). Inside each part, just type an expression to evaluate it. The expressions that are meant to be evaluated are e1, e2, and so on so if you type `e1' you will see the result of evaluating e1. You can also type more complex expressions if you want. You may also edit these .lhs files (make a fresh copy of them if you want before you start); if you change the .lhs file you have to type `:r' to reload the file into hugs. If you made any mistakes in the program you'll have to fix them to get the :r to work.

## Organization of the Online Tutorial

This online tutorial is divided into a series of file. Each file covers one or more sections in the written tutorial. Each file is a single Haskell program. Comments in the program contain the text of the online tutorial.

To create useful, executable examples of Haskell code, some language constructs need to be revealed well before they are explained in the tutorial. We attempt to point these out when they occur. Some small changes have been made to the examples in the written tutorial; these are usually cosmetic and should be ignored. Don't feel you have to understand everything on a page before you move on -- many times concepts become clearer as you move on and can relate them to other aspect of the language.

Each part of the tutorial defines a set of variables. Some of these are named e1, e2, and so on. These `e' variables are the ones which are meant for you to evaluate as you go through the tutorial. Of course you may evaluate any other expressions or variables you wish.

## The Haskell Report

While the report is not itself a tutorial on the Haskell language, it can be an invaluable reference to even a novice user. A very important feature of Haskell is the Prelude. The Prelude is a rather large chunk of Haskell code which is implicitly a part of every Haskell program. Whenever you see functions used which are not defined in the current page, these come from the Prelude. Appendix A of the report lists the entire Prelude; the index has an entry for every function in the Prelude. Looking at the definitions in the Prelude is sometimes necessary to fully understand the programs in this tutorial.

Another reason to look at the report is to understand the syntax of Haskell. Appendix B contains the complete syntax for Haskell. The tutorial treats the syntax very informally; the precise details are found only in the report.

You are now ready to start the tutorial. Start by reading the `Gentle Introduction' section 1 then proceed through the online tutorial using :l (if you are using hugs) to advance to the next part. You should read about each topic first before turning to the associated programming example in the online tutorial.

## Section: 2 Values, Types, and Other Goodies

This tutorial is written in `literate Haskell'. This style requires that all lines containing Haskell code start with `>'; all other lines are comments and are discarded by the compiler. Appendix C of the report defines the syntax of a literate program. This is the first line of the Haskell program on this page:

```
> module Part1() where
```

Comments at the end of source code lines start with `--'. We use these throughout the tutorial to place explanatory text in the program.

All Haskell programs start with a module declaration, as in the previous `module Test(Bool) where'. This can be ignored for now.

We will start by defining some identifiers (variables) using equations. You can print out the value of an identifier by typing the name of the identifier you wish to evaluate. Not all definitions are very interesting to print out - by convention, we will use variables e1, e2, ... to denote values that are interesting to print.

We will start with some constants as well as their associated type. There are two ways to associate a type with a value: a type declaration and an expression type signature. Here is an equation and a type declaration:

```
> e1 :: Int      -- This is a type declaration for the identifier e1
> e1 = 5         -- This is an equation defining e1
```

You can evaluate the expression e1 and watch the system print `5'.

Remember that Hugs evaluates expressions, not definitions. If you type `e1 = 5', a definition, you get an error. Definitions have to be placed in the module.

The type declaration for e1 is not really necessary but we will try to always provide type declarations for values to help document the program and to ensure that the system infers the same type we do for an expression.
If you change the value for e1 to `True', the program will no longer compile due to the type mismatch.

We will briefly mention expression type signatures: these are attached to expressions instead of identifiers.  Here are equivalent ways to do the previous definition:

```
> e2 = 5 :: Int
> e3 = (2 :: Int) + (3 :: Int)
```

The :: has very low precedence in expressions and should usually be placed
in parenthesis.

There are two completely separate languages in Haskell: an expression language for values and a type language for type signatures.  The type language is used only in the type declarations previously described and declarations of new types, described later.  Haskell uses a uniform syntax so that values resemble their type signature as much as possible.  However, you must always be aware of the difference between type expressions and value expressions.

Here are some of the predefined types Haskell provides:

| type | Value Syntax | Type Syntax |
|---|---|---|
| Small integers | &lt;digits&gt; | Int |

```
> e4 :: Int
> e4 = 12345
```

| | | |
|---|---|---|
| Characters | '&lt;character&gt;' | Char |

```
> e5 :: Char
> e5 = 'a'
```

| | | |
|---|---|---|
| Strings | "chars" | String |

```
> e6 :: String
> e6 = "abc"
```

| | | |
|---|---|---|
| Boolean | True, False | Bool |

```
> e7 :: Bool
> e7 = True
```

| | | |
|---|---|---|
| Floating point | <digits.digits> | Float |

```
> e8 :: Float
> e8 = 123.456
```

| | | |
|---|---|---|
| Homogeneous list | [<exp1>,<exp2>,...] | [<constituant type>] |

```
> e9 :: [Int]
> e9 = [1,2,3]
```

| | | |
|---|---|---|
| Tuple | (<exp1>,<exp2>,...) | (<exp1-type>,<exp2-type>,...) |

```
> e10 :: (Char,Int)
> e10 = ('b',4)
```

| | | |
|---|---|---|
| Functional | described later | domain type -> range type |

```
> inc :: Int -> Int    -- a function which takes an Int argument and returns Int
> inc x = x + 1        -- test this by evaluating `int 4'
```

Here's a few leftover examples from section 2:

```
> e11 = inc (inc 3)    -- you could also evaluate `inc (inc 3)' directly
```

Uncomment (by adding removing the --) this next line to see a compile time type error.

```
> -- e12 = 'a'+'b'
```

This is a rather odd error message. Essentially says that there is no way to add characters. If there were, it would be by defining an instance in the class Num (this is where + is defined) for the type Char.

Continued in part2.lhs

Gentle Introduction to Haskell 98, Online Supplement
Part 2
Covers Section 2.1

## Section: 2.1 Polymorphic Types

```
> module Part2() where
```

The following line allows us to redefine functions in the standard prelude. Ignore this for now.

```
> import Prelude hiding (length,head,tail,null)
```

Start with some sample lists to use in test cases:

```
> list1 :: [Int]
> list1 = [1,2,3]
> list2 :: [Char]          -- This is the really a String
> list2 = ['a','b','c']    -- This is the same as "abc"; evaluate list2 and see.
> list3 :: [[a]]           -- The element type of the inner list is unknown
> list3 = [[],[],[],[]]    -- so this list can't be printed
> list4 :: [Int]
> list4 = 1:2:3:4:[]       -- Exactly the same as [1,2,3,4]; print it and see.
```

This is the length function. You can test it by evaluating expressions such as `length list1'. Function application is written by simple juxtaposition: `f(x)' in other languages would be `f x' in Haskell.

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

Function application has the highest precedence, so 1 + length xs is parsed as 1 + (length xs). In general, you have to surround non-atomic arguments to a function with parens. This includes arguments which are also function applications. For example, f g x is the function f applied to arguments g and x, similar to f(g,x) in other languages. However, f (g x) is f applied to (g x), or f(g(x)), which means something quite different! Be especially careful with infix operators: f x+1 y-2 would be parsed as (f x)+(1 y)-2. This is also true on the left of the `=': the parens around (x:xs) are absolutely necessary. length x:xs would be parsed as (length x):xs.

Also be careful with prefix negation, -. The application `f -1' is f-1, not f(-1). Add parens around negative numbers to avoid this problem.

Here are some other list functions:

```
> head :: [a] -> a -- returns the first element in a list (same as car in lisp)
> head (x:xs) = x

> tail :: [a] -> [a] -- removes the first element from a list (same as cdr)
> tail (x:xs) = xs

> null :: [a] -> Bool
> null [] = True
> null (x:xs) = False

> cons :: a -> [a] -> [a]
> cons x xs = x:xs

> nil :: [a]
> nil = []
```

Length could be defined using these functions. This is not good Haskell style but does illustrate these other list functions. Haskell programmers feel that the pattern matching style, as used in the previous version of length, is more natural and readable.

```
> length' :: [a] -> Int   -- Note that ' can be part of a name
> length' x = if null x then 0 else 1 + length' (tail x)
```

A test case for length', cons, and nil

```
> e1 = length' (cons 1 (cons 2 nil))
```

We haven't said anything about errors yet. Each of the following examples illustrates a potential runtime or compile time error. The compile time error is commented out so that other examples will compile; you can uncomment e2 and see what happens.

```
> -- e2 = cons True False   -- Why is this not possible in Haskell?
> e3 = tail (tail ['a'])  -- What happens if you evaluate this?
> e4 = []       -- This is especially mysterious!
```

This last example, e4, is something hard to explain but is often encountered early by novices. We haven't explained yet how the system prints out the expressions you type in - this will wait until later. However, the problem here is that e4 has the type [a]. The printer for the list datatype is complaining that it needs to know a specific type for the list elements even though the list has no elements! This can be avoided by giving e4 a type such as [Int]. (To further confuse you, try giving e4 the type [Char] and see what happens.)

Continued in part3.lhs

Gentle Introduction to Haskell 98, Online Supplement
Part 3
Covers Section 2.1

## Section: 2.2 User-Defined Types

```
> module Part3() where
```

The type Bool is already defined in the Prelude so there is no
need to define it here.

```
> data Color = Red | Green | Blue | Indigo | Violet deriving Show
```

The `deriving Show' is necessary if you want to print a Color value.

You can now evaluate these expressions.

```
> e1 :: Color
> e1 = Red
> e2 :: [Color]
> e2 = [Red,Blue]
```

It is very important to keep the expression language and the type
language in Haskell separated. The data declaration above defines
the type constructor Color. This is a nullary constructor: it takes no
arguments. Color is found ONLY in the type language - it can not be
part of an expression. e1 = Color is meaningless. (Actually, Color could
be both a data constructor and a type constructor but we'll ignore this
possibility for now). On the other hand, Red, Blue, and so on are
(nullary) data constructors. They can appear in expressions and
in patterns (described later). The declaration e1 :: Blue would also
be meaningless. Data constructors can be defined ONLY in a data
declaration.

In the next example, Point is a type constructor and Pt is a data
constructor. Point takes one argument and Pt takes two. A data
constructor
like Pt is really just an ordinary function except that it can be used in
a pattern. Type signatures can not be supplied directly for data
constructors; their typing is completely defined by the data declaration.
However, data constructors have a signature just like any variable:
Pt :: a -> a -> Point a   -- Not valid Haskell syntax
That is, Pt is a function which takes two arguments with the same
arbitrary type and returns a value containing the two argument values.

```
> data Point a = Pt a a   deriving Show

> e3 :: Point Float
> e3 = Pt 2.0 3.0
> e4 :: Point Char
> e4 = Pt 'a' 'b'
> e5 :: Point (Point Int)
> e5 = Pt (Pt 1 2) (Pt 3 4)
> -- e6 = Pt 'a' True          -- This is a typing error
```

The individual components of a point do not have names.
Let's jump ahead a little so that we can write functions using these data types. Data constructors (Red, Blue, ..., and Pt) can be used in patterns. When more than one equation is used to define a function, pattern matching occurs top down.

A function to remove red from a list of colors.

```
> removeRed :: [Color] -> [Color]
> removeRed [] = []
> removeRed (Red:cs) = removeRed cs
> removeRed (c:cs) = c : removeRed cs   -- c cannot be Red at this point

> e7 :: [Color]
> e7 = removeRed [Blue,Red,Green,Red]
```

Pattern matching is capable of testing equality with a specific color.

All equations defining a function must share a common type. A definition such as:

foo Red = 1
foo (Pt x y) = x

would result in a type error since the argument to foo cannot be both a Color and a Point. Similarly, the right hand sides must also share a common type; a definition such as

foo Red = Blue
foo Blue = Pt Red Red

would also result in a type error.

Here are a couple of functions defined on points.

```
> dist :: Point Float -> Point Float -> Float
> dist (Pt x1 y1) (Pt x2 y2) = sqrt ((x1-x2)^2 + (y1-y2)^2)
> midpoint :: Point Float -> Point Float -> Point Float
> midpoint (Pt x1 y1) (Pt x2 y2) = Pt ((x1+x2)/2) ((y1+y2)/2)

> p1 :: Point Float
> p1 = Pt 1.0 1.0
> p2 :: Point Float
> p2 = Pt 2.0 2.0

> e8 :: Float
> e8 = dist p1 p2
> e9 :: Point Float
> e9 = midpoint p1 p2
```

The only way to take apart a point is to pattern match.
That is, the two values which constitute a point must be extracted by matching a pattern containing the Pt data constructor. Much more will be said about pattern matching later.

Haskell prints values in the same syntax used in expressions. Thus Pt 1 2 would print as Pt 1 2 (of course, Pt 1 (1+1) would also print as Pt 1 2).

Continued in part4.lhs

Gentle Introduction to Haskell 98, Online Supplement
Part 4
Covers Section 2.3

## Section: 2.3 Recursive Types

```
> module Part4() where

> data Tree a = Leaf a | Branch (Tree a) (Tree a)    deriving Show
```

The following typings are implied by this declaration. As before, this is not valid Haskell syntax.

Leaf :: a -> Tree a
Branch :: Tree a -> Tree a -> Tree a

```
> fringe :: Tree a -> [a]
> fringe (Leaf x) = [x]
> fringe (Branch left right) = fringe left ++ fringe right
```

The following trees can be used to test functions:

```
> tree1 :: Tree Int
> tree1 = Branch (Leaf 1) (Branch (Branch (Leaf 2) (Leaf 3)) (Leaf 4))
> tree2 :: Tree Int
> tree2 = Branch (Branch (Leaf 3) (Leaf 1)) (Branch (Leaf 4) (Leaf 1))
> tree3 :: Tree Int
> tree3 = Branch tree1 tree2
```

Try evaluating `fringe tree1' and others.

Here's another tree function:

```
> twist :: Tree a -> Tree a
> twist (Branch left right) = Branch right left
> twist x = x        -- This equation only applies to leaves
```

Here's a function which compares two trees to see if they have the same shape. Note the signature: the two trees need not contain the same type of values.

```
> sameShape :: Tree a -> Tree b -> Bool
> sameShape (Leaf x) (Leaf y) = True
> sameShape (Branch l1 r1) (Branch l2 r2) = sameShape l1 l2 && sameShape r1 r2
> sameShape x y = False   -- One is a branch, the other is a leaf
```

The && function is a boolean AND function.

The entire pattern on the left hand side must match in order for the right hand side to be evaluated.  The first clause requires both arguments to be a leaf' otherwise the next equation is tested. The last clause will always match: the final x and y match both leaves and branches.

This compares a tree of integers to a tree of booleans.

```
> e1 = sameShape tree1 (Branch (Leaf True) (Leaf False))
```

Continued in part5.lhs

Gentle Introduction to Haskell 98, Online Supplement
Part 5
Covers Sections 2.4, 2.5, 2.6

## Section: 2.4 Type Synonyms

```
> module Part5() where
```

Since type synonyms are part of the type language only, it's hard to write a program which shows what they do. Essentially, they are like macros for the type language. They can be used interchangeably with their definition:

```
> e1 :: String
> e1 = "abc"
> e2 :: [Char]   -- No different than String
> e2 = e1
```

In the written tutorial the declaration of `Addr' is a data type declaration, not a synonym declaration. This shows that the data type declaration as well as a signature can reference a synonym.

## Section: 2.5 Built-in Types

Tuples are an easy way of grouping a set of data values. Here are a few tuples. Note the consistancy in notation between the values and types.

```
> e3 :: (Bool,Int)
> e3 = (True,4)
> e4 :: (Char,[Int],Char)
> e4 = ('a',[1,2,3],'b')
```

Here's a function which returns the second component of a 3 tuple.

```
> second :: (a,b,c) -> b
> second (a,b,c) = b
```

Try out `second e3' and `second e4' - what happens?

Each different size of tuple is a completely distinct type. There is no general way to append two arbitrary tuples or randomly select the i'th component of an arbitrary tuple. Here's a function built using 2-tuples to represent intervals.

Use a type synonym to represent homogeneous 2 tuples

```
> type Interval a = (a,a)

> containsInterval :: Interval Int -> Interval Int -> Bool
> containsInterval(xmin,xmax)(ymin,ymax) = xmin <= ymin && xmax >=ymax

> p1 :: Interval Int
> p1 = (2,3)
> p2 :: Interval Int
> p2 = (1,4)

> e5 = containsInterval p1 p2
> e6 = containsInterval p2 p1
```

Here's a type declaration for a type isomorphic to lists:

```
> data List a = Nil | Cons a (List a) deriving Show
```

Except for the notation, this is completely equivalent to ordinary lists in Haskell.

```
> length' :: List a -> Int
> length' Nil = 0
> length' (Cons x y) = 1 + length' y

> e7 = length' (Cons 'a' (Cons 'b' (Cons 'c' Nil)))
```

It is hard to demonstrate much about the `non-specialness' of built-in types. However, here is a brief summary:

Numbers and characters, such as 1, 2.2, or 'a', are the same as nullary type constructors.

Lists have a special type constructor, [a] instead of List a, and an odd looking data constructor, []. The other data constructor, :, is not `unusual', syntactically speaking. The notation [x,y] is just syntax for x:y:[] and "abc" for 'a' : 'b' : 'c' : [].

Tuples use a special syntax. In a type expression, a 2 tuple containing types a and be would be written (a,b) instead of using a prefix type constructor such as Tuple2 a b. This same notation is used to build tuple values: (1,2) would construct a 2 tuple containing the values 1 and 2.

# Gentle Introduction to Haskell 98, Online Supplement
Part 5
Covers Sections 2.5.1, 2.5.2

```
> module Part6() where
```

## Section: 2.5.1  List Comprehensions and Arithmetic Sequences

Warning: brackets in Haskell are used in three different sorts of expressions: lists, as in [a,b,c], sequences (distinguished by the ..), as in [1..2], and list comprehensions (distinguished by the bar: |), as in [x+1 | x <- xs, x > 1].

Before list comprehensions, consider sequences:

```
> e1 :: [Int]
> e1 = [1..10]   -- Step is 1
> e2 :: [Int]
> e2 = [1,3..10] -- Step is 3 - 1
> e3 :: [Int]
> e3 = [1,-1.. -10]  -- The space before - is necessary!
> e4 :: [Char]
> e4 = ['a'..'z']   -- This works on chars too
```

We'll avoid infinite sequences like [1..] for now.  If you print one, use C-c i to interrupt the Haskell program.

List comprehensions are very similar to nested loops.  They return a list of values generated by the expression inside the loop.  The filter expressions are similar to conditionals in the loop.

This function does nothing at all!  It just scans through a list and copies it into a new one.

```
> doNothing :: [a] -> [a]
> doNothing l = [x | x <- l]
```

Adding a filter to the previous function allows only selected elements to be generated.  This is similar to what is done in quicksort.

```
> positives :: [Int] -> [Int]
> positives l = [x | x <- l, x > 0]

> e5 = positives [2,-4,5,6,-5,3]
```

Now the full quicksort function.

```
> quicksort :: [Char] -> [Char]  -- Use Char just to be different!
> quicksort [] = []
> quicksort (x:xs) = quicksort [y | y <- xs, y <= x] ++
>                     [x] ++
>                    quicksort [y | y <- xs, y > x]

> e6 = quicksort "Why use Haskell?"
```

Now for some nested loops. Each generator, <-, adds another level of nesting to the loop. The variable introduced by each generator can be used in each following generator; all variables can be used in the generated expression:

```
> e7 :: [(Int,Int)]
> e7 = [(x,y) | x <- [1..5], y <- [x..5]]
```

Now add some guards: (the /= function is `not equal')

```
> e8 :: [(Int,Int)]
> e8 = [(x,y) | x <- [1..7], x /= 5, y <- [x..8] , x*y /= 12]
```

This is the same as the loop: (going to a psuedo Algol notation)

```
for x := 1 to 7 do
 if x <> 5 then
  for y := x to 8 do
   if x*y <> 12
    generate (x,y)
```

Section: 2.5.2 Strings

```
> e9 = "hello" ++ " world"
```

Continued in part7.lhs

# Gentle Introduction to Haskell 98, Online Supplement
Part 7
Covers Sections 3, 3.1

```
> module Part7() where
> import Prelude hiding (map)
```

## Section: 3 Functions

```
> add :: Int -> Int -> Int
> add x y = x+y

> e1 :: Int
> e1 = add 1 2
```

This Int -> Int is the latter part of the signature of add:

add :: Int -> (Int -> Int)

```
> inc :: Int -> Int
> inc = add 1

> e2 :: Int
> e2 = inc 3

> map :: (a->b) -> [a] -> [b]
> map f [] = []
> map f (x:xs) = f x : (map f xs)

> e3 :: [Int]
> e3 = map (add 1) [1,2,3]
```

This next definition is the equivalent to e3

```
> e4 :: [Int]
> e4 = map inc [1,2,3]
```

Heres a more complex example.  Define flist to be a list of functions:

```
> flist :: [Int -> Int]
> flist = map add [1,2,3]
```

This returns a list of functions which add 1, 2, or 3 to their input.
Haskell should print flist as something like
[<<function>>,<<function>>,<<function>>]

Now, define a function which takes a function and returns its value when applied to the constant 1:

```
> applyTo1 :: (Int -> a) -> a
> applyTo1 f = f 1
> e5 :: [Int]
> e5 = map applyTo1 flist  -- Apply each function in flist to 1
```

If you want to look at how the type inference works, figure out how the signatures of map, applyTo1, and flist combine to yield [Int].

## Section: 3.1  Lambda Abstractions

The symbol \ is like `lambda' in lisp or scheme.

Anonymous functions are written as \ arg1 arg2 ... argn -> body
Instead of naming every function, you can code it inline with this notation:

```
> e6 = map (\f -> f 1) flist
```

Be careful with the syntax here.  `\x->\y->x+y` parses as `\ x ->\ y -> x + y.` The ->\ is all one token.  Use spaces!!

This is identical to e5 except that the applyTo1 function has no name.

Function arguments on the left of an = are the same as lambda on the right:

```
> add' = \x y -> x+y    -- identical to add
> inc' = \x -> x+1     -- identical to inc
```

As with ordinary function, the parameters to anonymous functions can be patterns:

```
> e7 :: [Int]
> e7 = map (\(x,y) -> x+y) [(1,2),(3,4),(5,6)]
```

Functions defined by more than one equation, like map, cannot be converted to anonymous lambda functions quite as easily - a case statement is also required.  This is discussed later.

Gentle Introduction to Haskell 98, Online Supplement
Part 8
Covers Sections 3.2, 3.2.1, 3.2.2

```
> module Part8() where

> import Prelude hiding ((++),(.))
```

## Section: 3.2 Infix operators

Haskell has both identifiers, like `x', and operators, like `+'.
These are just two different types of syntax for variables.
However, operators are by default used in infix notation.

Briefly, identifiers begin with a letter and may have numbers, _, and '
in them: x, xyz123, x'', xYz'_12a. The case of the first letter
distinguishes variables from data constructors (or type variables from
type constructors). An operator is a string of symbols, where
:!#$%&*+./<=>?@\^| are all symbols. If the first character is : then
the operator is a data constructor; otherwise it is an ordinary
variable operator. The - and ~ characters may start a symbol but cannot
be used after the first character. This allows a*-b to parse as
a * - b instead of a *- b.

Operators can be converted to identifiers by enclosing them in parens.
This is required in signature declarations. Operators can be defined
as well as used in the infix style:

```
> (++) :: [a] -> [a] -> [a]
> [] ++ y = y
> (x:xs) ++ y = x : (xs ++ y)
```

Table 2 (Page 54) of the report is invaluable for sorting out the
precedences of the many predefined infix operators.

```
> e1 = "Foo" ++ "Bar"
```

This is the same function without operator syntax

```
> appendList :: [a] -> [a] -> [a]
> appendList [] y = y
> appendList (x:xs) y = x : appendList xs y

> (.) :: (b -> c) -> (a -> b) -> (a -> c)
> f . g = \x -> f (g x)

> add1 :: Int -> Int
> add1 x = x+1
> e2 = (add1 . add1) 3
```

## Section: 3.2.1 Sections

Sections are a way of creating unary functions from infix binary functions. When a parenthesized expression starts or ends in an operator, it is a section. Another definition of add1:

```
> add1' :: Int -> Int
> add1' = (+ 1)

> e3 = add1' 4
```

Here are a few section examples:

```
> e4 = map (++ "abc") ["x","y","z"]

> e5 = map ("abc" ++) ["x","y","z"]
```

## Section: 3.2.2 Fixity Declarations

We'll avoid any demonstration of fixity declarations. The Prelude contains numerous examples.

Continued in part9.lhs

# Gentle Introduction to Haskell 98, Online Supplement
Part 9
Covers Sections 3.3, 3.4, 3.5

```
> module Part9() where
```

```
> import Prelude hiding (take,zip)
```

## Section: 3.3  Functions are Non-strict

Observing lazy evaluation can present difficulties.  The essential question is `does an expression get evaluated?'.  While in theory using a non-terminating computation is the way evaluation issues are examined, we need a more practical approach.  In expressions, Haskell uses `error' to denote bottom.  Evaluation of `error' will halt execution and print the attached error message.  The error function can be used to create stub functions for program components which have not been written yet or as a value to insert into data structures where a data value is required but should never be used.

```
> e1 :: Bool    -- This can be any type at all!
> e1 = error "e1"  -- evaluate this and see what happens.
```

```
> const1 :: a -> Int
> const1 x = 1
```

```
> e2 :: Int
> e2 = const1 (error "e2")  -- The bottom (the error function) is not
>                           -- needed and will thus not be evaluated.
```

## Section: 3.4  "Infinite" Data Structures

Data structures are constructed lazily.  A constructor like : will not evaluate its arguments until they are demanded.  All demands arise from the need to print the result of the computation -- components not needed to compute the printed result will not be evaluated.

```
> list1 :: [Int]
> list1 = (1:error "list1")
```

```
> e3 = head list1    -- does not evaluate the error
> e4 = tail list1    -- does evaluate error
```

Some infinite data structures.  Don't print these!  If you do, you will need to interrupt the system or kill the Haskell process.

```
> ones :: [Int]
> ones = 1 : ones
```

```
> numsFrom :: Int -> [Int]
> numsFrom n = n : numsFrom (n+1)
```

An alternate numsFrom using series notation:

```
> numsFrom' :: Int -> [Int]
> numsFrom' n = [n..]

> squares :: [Int]
> squares = map (^2) (numsFrom 0)
```

Before we start printing anything, we need a function to truncate these infinite lists down to a more manageable size. The `take' function extracts the first k elements of a list:

```
> take :: Int -> [a] -> [a]
> take 0 x      = []                 -- two base cases: k = 0
> take k []     = []                 -- or the list is empty
> take k (x:xs) = x : take (k-1) xs
```

now some printable lists:

```
> e5 :: [Int]
> e5 = take 5 ones

> e6 :: [Int]
> e6 = take 5 (numsFrom 10)

> e7 :: [Int]
> e7 = take 5 (numsFrom' 0)

> e8 :: [Int]
> e8 = take 5 squares
```

zip is a function which turns two lists into a list of 2 tuples. If the lists are of differing sizes, the result is as long as the shortest list.

```
> zip (x:xs) (y:ys) = (x,y) : zip xs ys
> zip xs ys = []   -- one of the lists is []

> e9 :: [(Int,Int)]
> e9 = zip [1,2,3] [4,5,6]

> e10 :: [(Int,Int)]
> e10 = zip [1,2,3] ones

> fib :: [Int]
> fib = 1 : 1 : [x+y | (x,y) <- zip fib (tail fib)]

> e11 = take 10 fib
```

This can be done without the list comprehension:

```
> fib' :: [Int]
> fib' = 1 : 1 : map (\(x,y) -> x+y) (zip fib (tail fib))
```

This could be written even more cleanly using a map function which maps a binary function over two lists at once. This is in the Prelude and is called zipWith.

```
> fib'' :: [Int]
> fib'' = 1 : 1 : zipWith (+) fib (tail fib)
```

For more examples using infinite structures look in the demo files that come with Yale Haskell. Both the pascal program and the primes program use infinite lists.

## Section: 3.5 The Error Function

Too late - we already used it!

Continued in part10.lhs

Gentle Introduction to Haskell 98, Online Supplement
Part 10
Covers Sections 4, 4.1, 4.2

```
> module Part10() where

> import Prelude hiding (take,(^))
```

## Section: 4  Case Expressions and Pattern Matching

Now for details of pattern matching.  We use [Int] instead of [a] since the only value of type [a] is [].

```
> contrived :: ([Int], Char, (Int, Float), String, Bool) -> Bool
> contrived ([], 'b', (1, 2.0), "hi", True) = False
> contrived x = True      -- add a second equation to avoid runtime errors

> e1 :: Bool
> e1 = contrived ([], 'b', (1, 2.0), "hi", True)
> e2 :: Bool
> e2 = contrived ([1], 'b', (1, 2.0), "hi", True)
```

Contrived just tests its input against a big constant.

Linearity in pattern matching implies that patterns can only compare values with constants.  The following is not valid Haskell:

```
member x [] = False
member x (x:ys) = True      -- Invalid since x appears twice
member x (y:ys) = member x ys
```

```
> f :: [a] -> [a]
> f s@(x:xs) = x:s
> f _ = []

> e3 = f "abc"
```

Another use of _:

```
> middle :: (a,b,c) -> b
> middle (_,x,_) = x

> e4 :: Char
> e4 = middle (True, 'a', "123")

> (^) :: Int -> Int -> Int
> x ^ 0 = 1
> x ^ (n+1) = x*(x^n)

> e5 :: Int
> e5 = 3^3
> e6 :: Int
> e6 = 4^(-2)  -- Notice the behavior of the + pattern on this one
```

## Section: 4.1  Pattern Matching Semantics

Here's an extended example to illustrate the left -> right, top -> bottom semantics of pattern matching.

```
> foo :: (Int,[Int],Int) -> Int
> foo (1,[2],3)    = 1
> foo (2,(3:_),3) = 2
> foo (1,_,3)      = 3
> foo _            = 4

> e7 = foo (1,[],3)
> e8 = foo (1,error "in e8",3)
> e9 = foo (1,1:(error "in e9"),3)
> e10 = foo (2,error "in e10",2)
> e11 = foo (3,error "in e11 (second)",error "in e11 (third)")
```

Now add some guards:

```
> sign :: Int -> Int
> sign x | x > 0  = 1
>        | x == 0 = 0
>        | x < 0  = -1

> e12 = sign 3
```

The last guard is often `True' to catch all other cases.  The identifier `otherwise' is defined as True for use in guards:

```
> max' :: Int -> Int -> Int
> max' x y | x > y      = x
>          | otherwise  = y
```

Guards can refer to any variables bound by pattern matching.  When no guard is true, pattern matching resumes at the next equation.  Guards may also refer to values bound in an associated where declaration.

```
> inOrder :: [Int] -> Bool
> inOrder (x1:x2:xs) | x1 <= x2 = True
> inOrder _                     = False

> e13 = inOrder [1,2,3]
> e14 = inOrder [2,1]
```

## Section: 4.2  An Example

```
> take :: Int -> [a] -> [a]
> take 0     _      = []
> take _     []     = []
> take (n+1) (x:xs) = x:take n xs

> take' :: Int -> [a] -> [a]
> take' _      []     = []
> take' 0      _      = []
> take' (n+1) (x:xs) = x:take' n xs

> e15, e16, e17, e18 :: [Int]
> e15 = take 0 (error "e15")
> e16 = take' 0 (error "e16")
> e17 = take (error "e17") []
> e18 = take' (error "e18") []
```

Continued in part11.lhs

Gentle Introduction to Haskell 98, Online Supplement
Part 11
Covers Sections 4.3, 4.4, 4.5, 4.6

```
> module Part11() where

> import Prelude hiding (take)
```

## Section: 4.3 Case Expressions

The function `take' using a case statement instead of multiple equations:

```
> take :: Int -> [a] -> [a]
> take m ys = case (m,ys) of
>               (0  ,_)    -> []
>               (_  ,[])   -> []
>               (n+1,x:xs) -> x : take n xs
```

The function take using if then else.  We can also eliminate the n+k pattern just for fun.  The original version of take is much easier to read!

```
> take' :: Int -> [a] -> [a]
> take' m ys = if m == 0 then [] else
>               if null ys then [] else
>                if m > 0 then head ys : take (m-1) (tail ys)
>                 else error "m < 0"
```

## Section: 4.4  Lazy Patterns

Before the client-server example, here is a contrived example of lazy patterns.  The first version will fail to pattern match whenever the the first argument is [].  The second version will always pattern match initially but x will fail if used when the list is [].

```
> nonlazy :: [Int] -> Bool -> [Int]
> nonlazy (x:xs) isNull  = if isNull then [] else [x]

> e1 = nonlazy [1,2] False
> e2 = nonlazy [] True
> e3 = nonlazy [] False
```

This version will never fail the initial pattern match

```
> lazy :: [Int] -> Bool -> [Int]
> lazy ~(x:xs) isNull  = if isNull then [] else [x]

> e4 = lazy [1,2] False
> e5 = lazy [] True
> e6 = lazy [] False
```

The server - client example is a little hard to demonstrate.  We'll avoid the initial version which loops.  Here is the version with irrefutable patterns.

```
> type Response = Int
> type Request = Int

> client :: Request -> [Response] -> [Request]
> client init ~(resp:resps) = init : client (next resp) resps

> server :: [Request] -> [Response]
> server (req : reqs) = process req : server reqs
```

Next maps the response from the previous request onto the next request

```
> next :: Response -> Request
> next resp = resp
```

Process maps a request to a response

```
> process :: Request -> Response
> process req = req+1

> requests :: [Request]
> requests = client 0 responses

> responses :: [Response]
> responses = server requests

> e7 = take 5 responses
```

The lists of requests and responses are infinite - there is no need to check for [] in this program. These lists correspond to streams in other languages.

Here is fib again:

```
> fib :: [Int]
> fib@(_:tfib) = 1 : 1 : [ a+b | (a,b) <- zip fib tfib]

> e8 = take 10 fib
```

## Section: 4.5 Lexical Scoping and Nested Forms

One thing that is important to note is that the order of the definitions in a program, let expression, or where clauses is completely arbitrary. Definitions can be arranged 'top down' or `bottom up' without changing the program.

```
> e9 = let y = 2 :: Float
>          f x = (x+y)/y
>      in f 1 + f 2

> f :: Int -> Int -> String
> f x y | y > z  = "y > x^2"
>        | y == z = "y = x^2"
>        | y < z  = "y < x^2"
>   where
>     z = x*x

> e10 = f 2 5
> e11 = f 2 4
```

## Section: 4.6  Layout

There's nothing much to demonstrate here.  We have been using layout all through the tutorial.  The main thing is to be careful line up the first character of each definition.  For example, if you change the indentation of the definition of f in e9 you will get a parse error.

Continued in part12.lhs

Gentle Introduction to Haskell 98, Online Supplement
Part 12
Covers Section 5

```
> module Part12() where

> import Prelude hiding (elem)
```

## Section: 5  Type Classes

Names in the basic class structure of Haskell cannot be hidden (they are in PreludeCore) so we have to modify the names used in the tutorial.

Here is a new Eq class:

```
> class Eq' a where
>   eq :: a -> a -> Bool
```

Now we can define elem using eq from above:

```
> elem :: (Eq' a) => a -> [a] -> Bool
> x `elem` [] = False
> x `elem` (y:ys) = x `eq` y || x `elem` ys
```

Before this is of any use, we need to admit some types to Eq'

```
> instance Eq' Int where
>   x `eq` y = abs (x-y) < 3   -- Let's make this `nearly equal' just for fun

> instance Eq' Float where
>   x `eq` y = abs (x-y) < 0.1

> list1 :: [Int]
> list1 = [1,5,9,23]

> list2 :: [Float]
> list2 = [0.2,5.6,33,12.34]

> e1 = 2 `elem` list1
> e2 = 100 `elem` list1
> e3 = 0.22 `elem` list2
```

Watch out!  Integers in Haskell are overloaded - without a type signature to designate an integer as an Int, expressions like 3 `eq` 3 will be ambiguous.

Now to add the tree type:

```
> data Tree a = Leaf a | Branch (Tree a) (Tree a)   deriving Show

> instance (Eq' a) => Eq' (Tree a) where
>    (Leaf a)       `eq` (Leaf b)       = a `eq` b
>   (Branch l1 r1) `eq` (Branch l2 r2) =  (l1 `eq` l2) && (r1 `eq` r2)
>   _              `eq` _              = False

> tree1,tree2 :: Tree Int
> tree1 = Branch (Leaf 1) (Leaf 2)
> tree2 = Branch (Leaf 2) (Leaf 1)

> e4 = tree1 `eq` tree2
```

Now make a new class with Eq' as a super class:

```
> class (Eq' a) => Ord' a where
>  lt,le :: a -> a -> Bool          -- lt and le are operators in Ord'
>  x `le` y = x `eq` y || x `lt` y  -- This is a default for le
```

The typing of lt & le is

le,lt :: (Ord' a) => a -> a -> Bool

This is identical to

le,lt :: (Eq' a,Ord' a) => a -> a -> Bool

Make Int an instance of Ord':

```
> instance Ord' Int where
>  x `lt` y = x < y+1

> i :: Int  -- Avoid ambiguity
> i = 3
> e5 :: Bool
> e5 = i `lt` i
```

Some constraints on instance declarations:
  A program can never have more than one instance declaration for
    a given combination of data type and class.
  If a type is declared to be a member of a class, it must also be
    declared in all superclasses of that class.
  An instance declaration does not need to supply a method for every
    operator in the class.  When a method is not supplied in an
    instance declaration and no default is present in the class
    declaration, a runtime error occurs if the method is invoked.
  You must supply the correct context for an instance declaration --
    this context is not inferred automatically.

This definition of Functor is in the Prelude:

```
class Functor f where
  fmap :: (a -> b) -> f a -> f b  -- a generalized map function
```

This makes Tree an instance of Functor.

```
> instance Functor Tree where
>    fmap f (Leaf x)       = Leaf   (f x)
>    fmap f (Branch t1 t2) = Branch (fmap f t1) (fmap f t2)

> e6 = fmap (+1) tree1
```

We can also make 2-tuples an instance of Functor:

```
> instance Functor ((,) a) where
>   fmap f (x,y) = (x,f y)

> e7 = fmap (+1) (1,2)
```

Note that these are a kind errors:

```
> -- instance Functor (,)

> -- instance Functor Bool
```

The error message from Hugs isn't very useful but if you use :set +k then you will see a better message.

Continued in part13.lhs

Gentle Introduction to Haskell 98, Online Supplement
Part 13
Covers Section 6.1, 6.2, 6.3

```
> module Part13() where
```

## Section 6.1

Example from the tutorial. Note that newtype supports deriving in the same manner as data. We also need to derive a Eq instance for Natural since the definition of Num has Eq as a superclass.

The Num instance of Natural allows the use of integer constants such as 1 as Naturals.

```
> newtype Natural = MakeNatural Integer deriving (Show, Eq)

> toNatural :: Integer -> Natural
> toNatural x | x < 0     = error "Can't create negative naturals!"
>             | otherwise = MakeNatural x

> fromNatural :: Natural -> Integer
> fromNatural (MakeNatural i) = i


> instance Num Natural where
>     fromInteger = toNatural
>     x + y = toNatural (fromNatural x + fromNatural y)
>     x - y = let r = fromNatural x - fromNatural y in
>               if r < 0 then error "Unnatural subtraction"
>                        else toNatural r
>     x * y = toNatural (fromNatural x * fromNatural y)

> e1 :: Natural
> e1 = toNatural 1
> e2 :: Natural
> e2 = 1
> e3 :: Natural
> e3 = 1 + 1
> e4 :: Natural
> e4 = (3 - 4) + 3
```

## Section 6.2

```
> data Point = Pt {pointx, pointy :: Float} deriving Show

> absPoint :: Point -> Float
> absPoint p = sqrt (pointx p * pointx p + pointy p * pointy p)

> e5 :: Point
> e5 = Pt {pointx = 1, pointy = 2}
> e6 :: Float
> e6 = absPoint e5
> e7 :: Float
> e7 = pointx e5
> e8 :: Point
> e8 = e5 {pointx = 4}

> data T = C1 {f :: Int, g :: Float}
>        | C2 {f :: Int, h :: Bool} deriving Show

> e9 :: T
> e9 = C1 {f = 1, g = 2}
> e10 :: T
> e10 = C2 {f = 3, h = False}
> e11 :: Int
> e11 = f e9
> e12 :: Int
> e12 = f e10
> e13 :: Float
> e13 = g e10
```

## Section 6.3

Here is a definition of head-strict lists: the head of each list is evaluated when the list is constructed.

```
> data HList a = Cons !a (HList a) | Nil deriving Show

> hd (Cons x y) = x
> tl (Cons x y) = y
```

If the "!" is removed then e17 no longer is an error.

```
> e14 :: HList Bool
> e14 = True `Cons` (error "e14" `Cons` Nil)
> e15, e16 :: Bool
> e15 = hd e14
> e16 = hd (tl e14)
> e17 :: HList Bool
> e17 = tl (tl (e14))
```

Continued in part14.lhs

Gentle Introduction to Haskell 98, Online Supplement
Part 14
Covers Section 7, 7.1, 7.2

```
> module Part14 where

> import Prelude hiding (putStr, getLine, sequence_)
> import IO (isEOFError)
```

Both putStr and getLine are actually in the prelude.

Section: 7  I/O

Section 7.1

The I/O monad divides the Haskell world into values and actions. So far, we have only used values but now we need to execute actions. Hugs looks at the type of an expression typed at the prompt. If the type is an IO type, the expression is assumed to be an action and the action is invoked. If the type is not IO t (for any t), then the expression value is printed.

```
> e1 = do c <- getChar
>         putChar c

> ready = do c <- getChar
>            return (c == 'y')
```

The print function converts a value to a string (using the Show class) and prints it with a newline at the end.

```
> e2 = do r <- ready
>         print r
```

You can't put the call to ready and print in the same expression.
This would be wrong:
e2 = print (ready)

```
> getLine :: IO String
> getLine = do c <- getChar
>              if c == '\n'
>                 then return ""
>                 else do l <- getLine
>                         return (c:l)
```

putStrLn prints a string and adds a newline at the end.

```
> e3 = do putStr "Type Something: "
>         str <- getLine
>         putStrLn ("You typed: " ++ str)
```

## Section 7.2

```
> todoList :: [IO ()]
> todoList = [putChar 'a',
>             do putChar 'b'
>                putChar 'c',
>             do c <- getChar
>                putChar c]

> sequence_        :: [IO ()] -> IO ()
> sequence_        =  foldr (>>) (return ())

> e4 = sequence_ todoList

> putStr    :: String -> IO ()
> putStr s = sequence_ (map putChar s)

> e5 = putStr "Hello World"
```

Continued in part15.lhs

Gentle Introduction to Haskell 98, Online Supplement
Part 15
Covers Section 7.3, 7.4, 7.5

```
> module Part15() where
```

```
> import IO
```

## Section 7.3

To experiment with I/O errors, we need to get a bit creative. Generating an error is generally OS specific so instead we will use errors that are directly under user control. The function

userError :: String -> IOError

generates an error value and the function

ioeGetErrorString :: IOError -> String

gets the string out of an error.

This version of getChar that raises an error when an @ or ? is entered:

```
> getCharE           :: IO Char
> getCharE           =  do c <- getChar
>                          case c of
>                            '@' -> ioError (userError "@")
>                            '?' -> ioError (userError "?")
>                            _   -> return c
```

Using this extended getChar we can build getChar' to catch only @
There is currently a bug in Hugs - if this gets fixed change the definition is isAtError.

```
> isAtError :: IOError -> Bool
> isAtError e = ioeGetErrorString e == "User error: @"   -- for bug in hugs
> -- isAtError e = ioeGetErrorString e == "@"     -- Should be this
```

```
> getChar'                :: IO Char
> getChar'                =  getCharE `catch` atHandler where
>     atHandler e = if isAtError e then return '\n' else ioError e
```

```
> getLine'  :: IO String
> getLine'  = catch getLine'' (\err -> return ("Error: " ++ show err))
>     where
>                    getLine'' = do c <- getChar'
>                                   if c == '\n' then return ""
>                                                else do l <- getLine'
>                                                        return (c:l)
```

Observe the behavior when you enter @ or ?

```
> e1 = getCharE
```

Now try lines with @ and ? in them.

```
> e2 = do l <- getLine'
>         putStrLn l
```

Section 7.4

You will have to set up some files to play with if you want to try this one.

```
> e3 = do fromHandle <- getAndOpenFile "Copy from: " ReadMode
>         toHandle   <- getAndOpenFile "Copy to: " WriteMode
>         contents   <- hGetContents fromHandle
>         hPutStr toHandle contents
>         hClose toHandle
>         putStr "Done."

> getAndOpenFile          :: String -> IOMode -> IO Handle

> getAndOpenFile prompt mode =
>    do putStr prompt
>       name <- getLine
>       catch (openFile name mode)
>             (\_ -> do putStrLn ("Cannot open "++ name ++ "\n")
>                       getAndOpenFile prompt mode)
>
```

Continued in part16.lhs

Gentle Introduction to Haskell 98, Online Supplement
Part 16
Covers Section 8, 8.1, 8.2, 8.3

```
> module Part16() where
```

Section: 8.1  Equality and Ordered Classes
Section: 8.2  Enumeration and Index Classes

No examples are provided for 5.1 or 5.2.  The standard Prelude contains many instance declarations which illustrate the Eq, Ord, and Enum classes.

Section: 8.3   The Read and Show Classes

This is the slow showTree.  The `show' function is part of the Text class and works with all the built-in types.  The context `Text a' arises from the call to show for leaf values.

```
> data Tree a = Leaf a | Branch (Tree a) (Tree a) deriving Show

> showTree :: (Show a) => Tree a -> String
> showTree (Leaf x)    = show x
> showTree (Branch l r)= "<" ++ showTree l ++ "|" ++ showTree r ++ ">"

> tree1 :: Tree Int
> tree1 = Branch (Leaf 1) (Branch (Leaf 3) (Leaf 6))

> e1 = showTree tree1
```

Now the improved showTree; shows is already defined for all built in types.

```
> showsTree  :: Show a => Tree a -> String -> String
> showsTree (Leaf x) s = shows x s
> showsTree (Branch l r) s = '<' : showsTree l ('|' : showsTree r ('>'
: s))

> e2 = showsTree tree1 ""
```

The final polished version.  ShowS is predefined in the Prelude so we don't need it here.

```
> showsTree'  :: Show a => Tree a -> ShowS
> showsTree' (Leaf x) = shows x
> showsTree' (Branch l r) = ('<' :) . showsTree' l . ('|' :) .
>                            showsTree' r . ('>' :)
> e3 = showsTree' tree1 ""
```

In the Prelude, there is a showChar function that can be used instead of ('<' :).

Now for the reading function. Again, ReadS is predefined and reads works
for all built-in types. The generators in the list comprehensions are patterns: p <- l binds pattern p to successive elements of l which match p. Elements not matching p are skipped.

```
> readsTree :: (Read a) => ReadS (Tree a)
> readsTree ('<':s)  = [(Branch l r, u) | (l, '|':t) <- readsTree s,
>                                          (r, '>':u) <- readsTree t ]
> readsTree s        = [(Leaf x,t)      | (x,t) <- reads s]

> e4 :: [(Int,String)]
> e4 = reads "5 golden rings"

> e5 :: [(Tree Int,String)]
> e5 = readsTree "<1|<2|3>>"
> e6 :: [(Tree Int,String)]
> e6 = readsTree "<1|2"
> e7 :: [(Tree Int,String)]
> e7 = readsTree "<1|<<2|3>|<4|5>>> junk at end"
```

Before we do the next readTree, let's play with the lex function.

```
> e8 :: [(String,String)]
> e8 = lex "foo bar bletch"
```

Here's a function to completely lex a string. This does not handle lexical ambiguity - lex would return more than one possible lexeme when an ambiguity is encountered and the patterns used here would not match.

```
> lexAll :: String -> [String]
> lexAll s = case lex s of
>             [("",_)] -> []    -- lex returns an empty token if none is found
>             [(token,rest)] -> token : lexAll rest

> e9 = lexAll "lexAll :: String -> [String]"
> e10 = lexAll "<1|<a|3>>"
```

Finally, the complete reader. This is not sensitive to white space as were the previous versions. When you derive the Show class for a data type the reader generated automatically is similar to this in style.

```
> readsTree' :: (Read a) => ReadS (Tree a)
> readsTree' s = [(Branch l r, x) | ("<", t) <- lex s,
>                          (l, u)   <- readsTree' t,
>                                    ("|", v) <- lex u,
>                                    (r, w)   <- readsTree' v,
>                          (">", x) <- lex w ]
>                  ++
>                  [(Leaf x, t)    | (x, t) <- reads s]
```

When testing this program, you must make sure the input conforms to Haskell lexical syntax. If you remove spaces between | and < or > they will lex as a single token as you should have noticed with e10.

```
> e11 :: [(Tree Int,String)]
> e11 = readsTree' "<1 | <2 | 3> >"
> e12 :: [(Tree Bool,String)]
> e12 = readsTree' "<True|False>"
```

Finally, here is a simple version of read for trees only:

```
> read' :: (Read a) => String -> (Tree a)
> read' s = case (readsTree' s) of
>            [(tree,"")] -> tree    -- Only one parse, no junk at end
>            []          -> error "Couldn't parse tree"
>            [_]         -> error "Crud after the tree" -- unread chars at end
>            _           -> error "Ambiguous parse of tree"

> e13 :: Tree Int
> e13 = read' "foo"
> e14 :: Tree Int
> e14 = read' "< 1 | < 2 | 3 > >"
> e15 :: Tree Int
> e15 = read' "3 xxx"
```

Continued in part17.lhs

Gentle Introduction to Haskell 98, Online Supplement
Part 17
Covers Section 8.4

## Section: 8.4 Derived Instances

We have actually been using the derived Show instances all along for printing out trees and other structures we have defined. The code in the tutorial for the Eq and Ord instance of Tree is created implicitly by the deriving clause so there is no need to write it here.

```
> data Tree a =   Leaf a
>               | Branch (Tree a) (Tree a) deriving  (Eq,Ord,Show)
```

Now we can fire up both Eq and Ord functions for trees:

```
> tree1, tree2, tree3, tree4 :: Tree Int
> tree1 = Branch (Leaf 1) (Leaf 3)
> tree2 = Branch (Leaf 1) (Leaf 5)
> tree3 = Leaf 4
> tree4 = Branch (Branch (Leaf 4) (Leaf 3)) (Leaf 5)

> e1 = tree1 == tree1
> e2 = tree1 == tree2
> e3 = tree1 < tree2

> quicksort :: Ord a => [a] -> [a]
> quicksort [] = []
> quicksort (x:xs) = quicksort [y | y <- xs, y <= x] ++
>                    [x] ++
>                    quicksort [y | y <- xs, y > x]

> e4 = quicksort [tree1,tree2,tree3,tree4]
```

Now for Enum:

```
> data Day = Sunday | Monday | Tuesday | Wednesday | Thursday |
>            Friday | Saturday     deriving (Show,Eq,Ord,Enum)

> e5 = quicksort [Monday,Saturday,Friday,Sunday]
> e6 = [Wednesday .. Friday]
> e7 = [Monday, Wednesday ..]
> e8 = [Saturday, Friday ..]
```

Continued in part18.lhs

# Gentle Introduction to Haskell 98, Online Supplement
Part 18
Covers Sections 9, 9.1, 9.2, 9.3

## Section 9.1 Monadic Classes
## Section 9.2 Built-in Monads

```
> module Part18() where

> e1 = [(x,y) | x <- [1,2,3] , y <- [1,2,3], x /= y]

> e2 = do x <- [1,2,3]
>         y <- [1,2,3]
>         True <- return (x /= y)
>         return (x,y)

> e3 = [1,2,3] >>= (\ x -> [1,2,3] >>= (\y -> return (x/=y) >>=
>      (\r -> case r of True -> return (x,y)
>                       _    -> fail "")))

> mvLift2                :: (a -> b -> c) -> [a] -> [b] -> [c]
> mvLift2 f x y          =  do x' <- x
>                              y' <- y
>                              return (f x' y')

> e4 = mvLift2 (+) [1,3] [10,20,30]
> e5 = mvLift2 (\a b -> [a,b]) "ab" "cd"
> e6 = mvLift2 (*) [1,2,4] []
```

A quick example using Maybe: first, generalize mvLift2 to all monads:

```
> lift2'              :: Monad m => (a -> b -> c) -> m a -> m b -> m c
> lift2' f x y        =  do x' <- x
>                           y' <- y
>                           return (f x' y')

> e7 = lift2' (+) (Just 1) (Just 2)
> e8 = lift2' (+) (Just 1) Nothing
```

## Section 9.3

```
> type S = Int

> data SM a = SM (S -> (a,S))  -- The monadic type

> instance Monad SM where
>   -- defines state propagation
>   SM c1 >>= fc2         =  SM (\s0 -> let (r,s1) = c1 s0
>                                               SM c2 = fc2 r in
>                                             c2 s1)
>   return k              =  SM (\s -> (k,s))

>  -- extracts the state from the monad
> readSM                  :: SM S
> readSM                  =  SM (\s -> (s,s))
>  -- extracts the state from the monad
> updateSM                :: (S -> S) -> SM ()  -- alters the state
> updateSM f              =  SM (\s -> ((), f s))

> -- run a computation in the SM monad
> runSM                   :: S -> SM a -> (a,S)
> runSM s0 (SM c)         =  c s0
```

This is fairly hard to demonstrate in a manner that makes this construction
look useful! This demonstrates the basic operation:

```
> e9 = runSM 0 (do x <- readSM  -- should be 0
>                  updateSM (+1)
>                  y <- readSM   -- now a 1
>                  return (x,y))
```

Most of the SM functions are present in the next example in slightly altered forms.

```
> type Resource     =  Integer

> data R a = R (Resource -> (Resource, Either a (R a)))

> instance Monad R where
>   R c1 >>= fc2    = R (\r -> case c1 r of
>                         (r', Left v)    -> let R c2 = fc2 v in
>                                                c2 r'
>                         (r', Right pc1) -> (r', Right (pc1 >>= fc2)))
>   return v              = R (\r -> (r, (Left v)))

> step                    :: a -> R a
> step v                  = c where
>                             c = R (\r -> if r /= 0 then (r-1, Left v)
>                                                     else (r, Right c))

> run                     :: Resource -> R a -> Maybe a
> run s (R p)             =  case (p s) of
>                              (_, Left v) -> Just v
>                              _           -> Nothing

> (|||)                   :: R a -> R a -> R a
> c1 ||| c2               =  oneStep c1 (\c1' -> c2 ||| c1')
>    where
>         oneStep          :: R a -> (R a -> R a) -> R a
>         oneStep (R c1) f =
>              R (\r -> case c1 1 of
>                         (r', Left v) -> (r+r'-1, Left v)
>                         (r', Right c1') ->  -- r' must be 0
>                          let R next = f c1' in
>                            next (r+r'-1))

> lift1                   :: (a -> b) -> (R a -> R b)
> lift1 f                 =  \ra1 -> do a1 <- ra1
>                                       step (f a1)
> lift2                   :: (a -> b -> c) -> (R a -> R b -> R c)
> lift2 f                 =  \ra1 ra2 -> do a1 <- ra1
>                                           a2 <- ra2
>                                           step (f a1 a2)

> (==*)                   :: Ord a => R a -> R a -> R Bool
> (==*)                   =  lift2 (==)
```

These null instances are needed to allow the definition of Num (R a).

```
> instance Show (R a)
> instance Eq (R a)

> instance Num a => Num (R a) where
>   (+)                   =  lift2 (+)
>   (-)                   =  lift2 (-)
>   negate                =  lift1 negate
>   (*)                   =  lift2 (*)
>   abs                   =  lift1 abs
>   fromInteger           =  return . fromInteger

> ifR                     :: R Bool -> R a -> R a -> R a
> ifR tst thn els         =  do t <- tst
>                                if t then thn else els


> inc                     :: R Integer -> R Integer
> inc x                   =  x + 1

> fact                    :: R Integer -> R Integer
> fact x                  =  ifR (x ==* 0) 1 (x * fact (x-1))

> e10 = run 0 (inc 1)  -- won't complete
> e11 = run 10 (inc 1)  -- will complete
> e12 = run 10 (fact 2)
> e13 = run 10 (fact 20)
> e14 = run 100 (fact (-1) ||| (fact 3))
```

We can dress this up a little with a nicer "run" function. This one adds a little more information:

```
> run'                   :: Show a => Integer -> R a -> IO ()
> run' maxSteps (R p)    =  case (p maxSteps) of
>                            (r, Left v)-> putStrLn ("Computed " ++
>                                                     show v ++
>                                                     " in " ++
>                                                     show
>                                                    (maxSteps-r)++
>                                                    " steps")
>                              _       -> putStrLn ("Non termination.")

> e15 = run' 100 (fact 3 ||| fact 4)
> e16 = run' 100 (fact (-1) ||| fact 4)
> e17 = run' 100 (fact 4)
```

Continued in part19.lhs

Gentle Introduction to Haskell 98, Online Supplement
Part 19
Covers Sections 10.1, 10.2, 10.3

```
> module Part19() where
```

```
> import Complex
```

Section: 10  Numbers
Section: 10.1  Numeric Class Structure
Section: 10.2  Constructed Numbers

Here's a brief summary of Haskell numeric classes.

Class Num
  Most general numeric class.  Has addition, subtraction, multiplication.
  Integers can be coerced to any instance of Num with fromInteger.
  All integer constants are in this class.
Instances: Int, Integer, Float, Double, Ratio a, Complex a

Class Real
  This class contains ordered numbers which can be converted to
  rationals.
Instances: Int, Integer, Float, Double, Ratio a

Class Integral
  This class deals with integer division.  All values in Integral can
  be mapped onto Integer.
Instances: Int, Integer

Class Fractional
  These are numbers which can be divided.  Any rational number can
  be converted to a fractional.  Floating point constants are in
  this class: 1.2 would be 12/10.
Instances: Float, Double, Ratio a

Class Floating
  This class contains all the standard floating point functions such
  as sqrt and sin.
Instances: Float, Double, Complex a

Class RealFrac
  These values can be rounded to integers and approximated by rationals.
Instances: Float, Double, Ratio a

Class RealFloat
  These are floating point numbers constructed from a fixed precision
  mantissa and exponent.
Instances: Float, Double

There are only a few sensible combinations of the constructed numerics with built-in types:
 Ratio Integer (same as Rational): arbitrary precision rationals
 Ratio Int: limited precision rationals
 Complex Float: complex numbers with standard precision components
 Complex Double: complex numbers with double precision components

The following function works for arbitrary numerics:

```
> fact :: (Num a) => a -> a
> fact 0 = 1
> fact n = n*(fact (n-1))
```

Note the behavior when applied to different types of numbers:

```
> e1 :: Int
> e1 = fact 6
> e2 :: Int
> e2 = fact 20   -- Hugs may not handle overflow gracefully!
> e3 :: Integer
> e3 = fact 20
> e4 :: Rational
> e4 = fact 6
> e5 :: Float
> e5 = fact 6
> e6 :: Complex Float
> e6 = fact 6
```

Be careful: values like `fact 1.5' will loop.

As a practical matter, Int operations are usually faster than Integer operations. Also, overloaded functions can be much slower than non-overloaded functions. Giving a function like fact a precise typing:

fact :: Int -> Int        may yield much faster code.

In general, numeric expressions work as expected. Literals are a little tricky - they are coerced to the appropriate value. A constant like 1 can be used as ANY numeric type.

```
> e7 :: Float
> e7 = sqrt 2
> e8 :: Rational
> e8 = ((4%5) * (1%2)) / (3%4)
> e9 :: Rational
> e9 = 2.2 * (3%11) - 1
> e10 :: Complex Float
> e10 = (2 * (3:+3)) / ((1.1:+2.0) - 1)
> e11 :: Complex Float
> e11 = sqrt (-1)
> e12 :: Integer
> e12 = numerator (4%2)
> e13 :: Complex Float
> e13 = conjugate (4:+5.2)
```

A function using pattern matching on complex numbers:

```
> mag :: (RealFloat a) => Complex a -> a
> mag (a:+b) = sqrt (a^2 + b^2)

> e14 :: Float
> e14 = mag (1:+1)
```

Section: 10.3  Numeric Coercions and Overloaded Literals

The Haskell type system does NOT implicitly coerce values between the different numeric types! Although overloaded constants are coerced when the overloading is resolved, no implicit coercion goes on when values of different types are mixed. For example:

```
> f :: Float
> f = 1.1
> i1 :: Int
> i1 = 1
> i2 :: Integer
> i2 = 2
```

All of these expressions would result in a type error (try them!):

```
> -- g = i1 + f
> -- h = i1 + i2
> -- i3 :: Int
> -- i3 = i2
```

Appropriate coercions must be introduced by the user to allow
the mixing of types in arithmetic expressions.

```
> e15 :: Float
> e15 = f + fromIntegral i1
> e16 :: Integer
> e16 = fromIntegral i1 + i2
> e17 :: Int
> e17 = i1 + fromInteger i2  -- fromIntegral would have worked too.
```

Continued in part20.lhs

Gentle Introduction to Haskell 98, Online Supplement
Part 20
Covers Section 10.4

```
> module Part20() where
```

```
> import Complex
```

## Section: 10.4  Default Numeric Types

Ambiguous contexts arise frequently in numeric expressions. When an expression which produces a value with a general type, such as `1' (same as `fromInteger 1'; the type is (Num a) => a), with another expression which `consumes' the type, such as `show' or `toInteger', ambiguity arises. This ambiguity can be resolved using expression type signatures, but this gets tedious fast! Assigning a type to the top level of an ambiguous expression does not help: the ambiguity does not propagate to the top level.

```
> e1 :: String  -- This type does not influence the type of the argument to show
> e1 = show 1   -- Does this mean to show an Int or a Float or ...
> e2 :: String
> e2 = show (1 :: Float)
> e3 :: String
> e3 = show (1 :: Complex Float)
```

The reason the first example works is that ambiguous numeric types are resolved using defaults. The defaults in effect here are Int and Double. Since Int `fits' in the expression for e1, Int is used. When Int is not valid (due to other context constraints), Double will be tried.

This function defaults the type of the 2's to be Int

```
> rms :: (Floating a) => a -> a -> a
> rms x y = sqrt ((x^2 + y^2) * 0.5)
```

One of the reasons for adding type signatures throughout these examples is to avoid unexpected defaulting. Many of the top level signatures are required to avoid ambiguity.

Notice that defaulting applies only to numeric classes. The

```
> --  show (read "xyz")                          -- Try this if you want!
```

example uses only class Show so no defaulting occurs.

Ambiguity also arises with polymorphic types. As discussed previously, expressions like [] have a similar problem.

```
> e4 = []   -- Won't print since [] has type [a] and `a' is not known.
```

Note the difference: even though the lists have no components, the type of component makes a difference in printing.

```
> e5 = ([] :: [Int])
> e6 = ([] :: [Char])
```

Continued in part21.lhs

Gentle Introduction to Haskell 98, Online Supplement
Part 21
Covers Section 11

Section 11

```
> module Part21() where
```

Since each of these files is a separate module we need to place
Tree in it's own module. See Tree.lhs

```
> import Tree ( Tree(Leaf, Branch), fringe)

> e1 :: [Int]
> e1 = fringe (Branch (Leaf 1) (Leaf 2))
```

You could also just `import Tree' and get everything from Tree without explicitly naming the entities you need.

This interactive Haskell environment can evaluate expressions in
any module. You can do :m Tree to get to that module for interactive evaluation.

Continued in Part22.lhs

```
> module Tree ( Tree(Leaf,Branch), fringe ) where
```

Tree(..) would work also.

```
> data Tree a = Leaf a | Branch (Tree a) (Tree a)   deriving Show

> fringe :: Tree a -> [a]
> fringe (Leaf x)             = [x]
> fringe (Branch left right)  = fringe left ++ fringe right
```

## Gentle Introduction to Haskell 98, Online Supplement
Part 22
Covers Section 11.1

### Section: 11.1  Qualified Names

```
> module Part22() where
> import Tree ( Tree(Leaf,Branch), fringe )
> import qualified Fringe ( fringe )
```

See Fringe.lhs

```
> e1 = do print (fringe (Branch (Leaf 1) (Leaf 2)))
>         print (Fringe.fringe (Branch (Leaf 1) (Leaf 2)))
```

Continued in part23.lhs

```
> module Fringe where
> import Tree ( Tree(..))

> fringe :: Tree a -> [a]   -- A different definition of fringe
> fringe (Leaf x) = [x]
> fringe (Branch x y) = fringe x
```

## Gentle Introduction to Haskell 98, Online Supplement
Part 23
Covers Sections 11.2, 11.3

### 11.2 Abstract Data Types

```
> module Part23() where

> import TreeADT
```

See TreeADT.lhs

Since the constructors for type Tree are hidden, pattern matching cannot be used.

```
> fringe :: Tree a -> [a]
> fringe x = if isLeaf x then [cell x]
>                        else fringe (left x) ++ fringe (right x)

> e1 :: [Int]
> e1 = fringe (branch (branch (leaf 3) (leaf 2)) (leaf 1))
```

Since TreeADT does not import Tree it can use the name Tree without any conflict.  Each module has its own separate namespace.

```
> module TreeADT (Tree, leaf, branch, cell, left,
>                right, isLeaf) where

> data Tree a = Leaf a | Branch (Tree a) (Tree a)    deriving Show

> leaf = Leaf
> branch = Branch
> cell (Leaf a) = a
> left (Branch l r) = l
> right (Branch l r) = r
> isLeaf (Leaf _) = True
> isLeaf _        = False
```

## 11.3 More Features

No examples (yet).

Continued in part24.lhs

Gentle Introduction to Haskell 98, Online Supplement
Part 24
Covers Sections 12, 12.1, 12.2, 12.3

## Section: 12 Typing Pitfalls

## Section: 12.1 Let-Bound Polymorphism

```
> module Part24() where

> -- f g = (g 'a',g [])    -- This won't typecheck.
```

## Section: 12.2 Overloaded Numerals

Overloaded numerics were covered previously - here is one more example.
sum is a prelude function which sums the elements of a list.

```
> average :: (Fractional a) => [a] -> a
> average xs   = sum xs / fromIntegral (length xs)

> e1 :: Float   -- Note that e1 would default to Double instead of
Integer -
>               -- this is due to the Fractional context.
> e1 = average [1,2,3]
```

## Section: 12.3 The Monomorphism Restriction

The monomorphism restriction is usually encountered when functions are defined without parameters. If you remove the signature for sum' the monomorphism restriction will apply. Hugs (at present) incorrectly defaults the type of sum' to Integer -> Integer without the type signature. If either of sumInt or sumFloat are present, these would determine the overloading. If both are present and sum' has no signature there is an error.

```
> sum' :: (Num a) => [a] -> a
> sum' = foldl (+) 0          -- foldl reduces a list with a binary function
>                             -- 0 is the initial value.

> sumInt :: Int
> sumInt = sum' [1,2,3]

> sumFloat :: Float
> sumFloat = sum' [1,2,3]
```

If you use overloaded constants you also may encounter monomorphism:

```
> x :: Num a => a
> x = 1            -- The type of x is Num a => a
> y :: Int
> y = x               -- Uses x as an Int
> z :: Integer
> z = x               -- Uses x as an Integer.  A monomorphism will occur of the
>                     -- signature for x is removed.
```

# Haskell code for the Gentle Intro 98

Introduction ........................................................................3
Using Hugs ........................................................................3
Organization of the Online Tutorial ....................................4
The Haskell Report ..............................................................4
Section 2 ........................................................................5
Section 2.1 ........................................................................8
Section 2.2 ........................................................................10
Section 2.3 ........................................................................13
Sections 2.4, 2.5, 2.6 ...........................................................15
Sections 2.5.1, 2.5.2 ............................................................17
Sections 3, 3.1 ....................................................................19
Sections 3.2, 3.2.1, 3.2.2 .....................................................21
Sections 3.3, 3.4, 3.5 ..........................................................23
Sections 4, 4.1, 4.2 .............................................................26
Sections 4.3, 4.4, 4.5, 4.6 ....................................................29
Section 5 ........................................................................32
Section 6.1, 6.2, 6.3 ............................................................35
Section 7, 7.1, 7.2 ...............................................................37
Section 7.3, 7.4, 7.5 ............................................................39
Section 8, 8.1, 8.2, 8.3 ........................................................41
Section 8.4 ........................................................................44
Sections 9, 9.1, 9.2, 9.3 .......................................................45
Sections 10.1, 10.2, 10.3 .....................................................48
Section 10.4 ........................................................................51
Section 11 ........................................................................50
Section 11.1 ........................................................................53
Sections 11.2, 11.3..............................................................53
Sections 12, 12.1, 12.2, 12.3 ...............................................55

PDF version by Dan Popa