Introduction to Programming in Haskell

Lecture Notes, CMI, 2008

Madhavan Mukund

Chennai Mathematical Institute

`https://www.cmi.ac.in/~madhavan`

Contents

Introduction to Programming, Aug-Dec 2008

Lecture 1, Monday 4 Aug 2008

Administrative matters

Resource material

Textbooks and other resource material for the course:

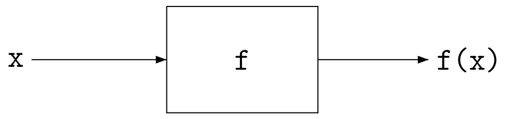
- *The Craft of Functional Programming* by Simon Thompson
- *Introduction to Functional Programming using Haskell* by Richard Bird
- *Programming in Haskell* by Graham Hutton
- *Introduction to Functional Programming* by Richard Bird and Philip Wadler
- Instructor's lecture notes, to be made available as the course progresses
- Online archive at `http://www.haskell.org`

Evaluation

- Approximately 8–10 assignments, 50%
- Midsemester examination, 20%
- Final examination, 30%

Introduction to Haskell

Programs in Haskell are functions that transform inputs to outputs. Viewed externally, a function is a black box:



The internal description of the function f has two parts:

1. The types of inputs and outputs
2. The rule for computing the output from the input

In mathematics, the type of a function is often implicit: Consider $sqr(x) = x^2$ which maps each input to its square. We could have $sqr : \mathbb{Z} \to \mathbb{Z}$ or $sqr : \mathbb{R} \to \mathbb{R}$ or $sqr : \mathbb{C} \to \mathbb{C}$, depending on the context.

Here is a corresponding definition in Haskell.

```
sqr :: Int -> Int
sqr x = x^2
```

The first line gives the type of `sqr`: it says that `sqr` reads an `Int` as input and produces an `Int` as output. In general, a function that takes inputs of type `A` and produces outputs of type `B` has the type `A -> B`. The second line gives the rule: it says that `sqr x` is `x^2`, where `^` is the symbol for exponentiation.

Basic types in Haskell

`Int` is a type that Haskell understands and roughly corresponds to the set of integers $\mathbb{Z}$ in mathematics. "Roughly", because every integer in Haskell is represented in a fixed and bounded amount of space, so there is a limit on the magnitude of the integers that Haskell can manipulate (think of what would happen if you had to do arithmetic with pencil and paper but could not write a number that was longer than one line on the page).

Here are (some of) the types that Haskell understands by default:

Int Integers. Integers are represented internally in binary. Typically, one binary digit (or bit) needs to be used to denote the sign (+/-) of the integer, and the remaining bits denote its magnitude. The exact representation is not important, but we should realize that the size of the representation is fixed (that is, how many binary digits are used to represent an `Int`), so the magnitude is bounded.

Float "Real" numbers. The word `Float` is derived from *floating point*, a reference to the fact that when writing down a real number in decimal notation, the position of the decimal point is not "fixed", but "floating". Internally, a Float is represented in *scientific notation* (for example, $1.987x10^{23}$) using two binary quantities: the mantissa and the exponent. For each of these, we reserve one bit for the sign and use the rest for the magnitude. Thus, we can represent numbers that are both very large and very small in magnitude: for instance, $1.987x10^{23}$ and $1.987x10^{-23}$.

Once again, the exact representation is not important but we should realize that `Float` is only an approximation of the set of real numbers, just as `Int` is only an approximation of the integers. In fact, the approximation in `Float` has two dimensions—there is a limit on magnitude and precision. Thus, real numbers are dense (between any two real numbers we can find a third) but floating point numbers are not.

Char Used to represent text characters. These are the symbols that we can type on the keyboard. A value of this type is written in single quotes: for instance, `'z'` is the character representing the letter z, `'&'` is the character representing ampersand etc.

A significant amount of computation involves manipulating characters (think of word processers) so this is an important type for programming.

Bool This type has two values, `True` and `False`. As we shall see, these are used frequently in programming.

Compilers vs interpreters

The languages that we use to program computers are typically "high level". These have to be converted into a "low level" set of instructions that can be directly executed by the electronic hardware inside the computer.

Normally, each program we write is translated into a corresponding low level program by a *compiler*.

Another option is to write a program that directly "understands" the high level programming language and executes it. Such a program is called an *interpreter*.

For much of the course, we will run Haskell programs through an interpreter, which is invoked by the command `hugs` or `ghci`. Within `hugs/ghci` you can type the following commands:

`:load filename` — Loads a Haskell file

`:type expression` — Print the type of a Haskell expression

`:quit` — exit from hugs

`:?` — Print "help" about more hugs commands

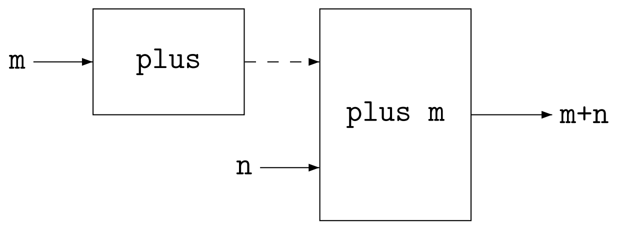
Functions with multiple inputs

One feature of function definitions that we have not emphasized is the the number of inputs. For instance, the function sqr that we saw earlier has only one input. On the other hand, we could write a function on two inputs, such as the mathematical function *plus*, below

$$\mathit{plus}(m, n) = m + n$$

Mathematically, the type of *plus* would be $\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$ (or $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$). This means that, in addition to the types of the input and the rule for computation, we also need to include information about the *arity* of the function, or how many inputs it takes.

This complication can be avoided by taking the somewhat drastic step of assuming that all functions take only one argument. How then can we define a function such as *plus* that needs to operate on two arguments? We say that *plus* first picks up the argument m and becomes a new function *plus m*, that adds the number m to its argument n. Thus, we break up a function on two arguments into a sequence of functions of one argument.



What is the type of *plus*? It takes in an integer m and yields a new function *plus m* that is like *sqr* above: it reads an integer and generates an output of the same type, so its type is $\mathbb{Z} \to \mathbb{Z}$ or, in Haskell notation, `(Int -> Int)`. Thus, in Haskell notation, `plus` reads an `Int` and generates a function of type `(Int -> Int)`, so the type of `plus` is `Int -> (Int -> Int)`. Here is a complete definition of plus in Haskell:

```
plus :: Int -> (Int -> Int)
plus m n = m + n
```

Notice that we write `plus m n` and not `plus(m,n)`—there are no parentheses around the arguments to a function. In fact, the correct bracketing for `plus m n` is `(plus m) n`. This tells us to first feed `m` to `plus` to get a function `(plus m)` to which we then feed the argument `n`.

What if we had a function of three arguments, such as `plus3(m,n,p) = m+n+p`? Once again, we assume that `plus3` consumes its arguments one at a time. Having read `m`, `plus3` becomes a function like `plus` that we defined earlier, except it adds on `m` to the sum of its two arguments. Since the type of `plus` was `Int -> (Int -> Int)`, this is the output type of `plus3`. The input to `plus3` is an `Int`, so the overall type of `plus3` is `Int -> (Int -> (Int -> Int))`.

Here is a complete definition of `plus3` in Haskell:

```
plus3 :: Int -> (Int -> (Int -> Int))
plus m n p = m + n + p
```

Once again, note the lack of brackets in `plus m n p`, which is implicitly bracketed `((plus m) n) p`.

In general, suppose we have a function `f` that reads `n` inputs `x_1`, `x_2`,..., `x_n` of types `t_1`,`t_2`,...,`t_n` and produces an output `y` of type `t`. The notation `::` introduced earlier to denote the type of a function is read as "is of type" and we can use here as well to write `x_1::t_1`, `x_2::t_2`, ..., `y::t` to denote that `x_1` is of type `t_1`, `x_2` is of type `t_2`, ..., `y` is of type `t`.

We can define the type of `f` by induction on `n`.

The base case is when `n` is 1, so `f` reads one input `x_1::t_1` and produces the output `y::t`. In this case, `f :: t_1 -> t`, as we have discussed earlier.

For the inductive step, we have a function that reads its first input `x_1::t_1` and then transforms itself into another function `g` that reads inputs `x_2::t_2`,`x_3::t_3`,...,`x_n::t_n` and produces an output `y::t`. Let the type of `g` be `T`. Then, `f : :t_1 -> T`. If we unravel the structure of `T` inductively, we find that

```
f::t_1 -> (t_2 -> (... -> (t_n -> t)...))
```

In this expression, the brackets are introduced uniformly from the right, so we can omit the brackets and unambiguously write

```
f::t_1 -> t_2 -> ... -> t_n -> t
```

More on defining functions

The simplest form of definition is the one we have seen in `sqr`, `plus` and `plus3`, where we just write a defining equation using an arithmetic expression involving the arguments to the function.

The arithmetic operators that we can use in writing such an expression are `+,-,*,/` signifying addition, subtraction, multiplication and division. As usual, we can also use `-` in front of an expression to negate its value, as in `-(x+y)`. In addition, the function `div` and `mod` signify integer division and remainder, respectively. So `div 3 2` is `1`, `div 7 3` is `2`, ... while `mod 10 6` is `4`, `mod 17 12` is `5`, ... Note that `div` and `mod` are functions, so they are written before their arguments rather than between them: it is `div 3 2` and `mod 17 12`, not `3 div 2` and `17 mod 12`.

We can also write expressions involving other types. For instance, for values of type `Bool`, the operator `&&` denotes the *and* operation, which returns `True` precisely when both its arguments are `True`. Dually, the operator `||`, pronounced `or`, returns `True` when at least one of its arguments is `True` (or, equivalently, `||` returns `False` precisely when both its arguments are `False`). The unary operator `not` inverts its argument. For instance, here is a definition of the function `xor` which returns `True` provided exactly one of its arguments is `True`.

```
xor :: Bool -> Bool -> Bool
xor b1 b2 = (b1 && (not b2)) || ((not b1) && b2)
```

We can also use operators to compare quantities. The result of such an operation is of type `Bool`. Here is a function that determines if the middle of its three arguments is larger than the other two arguments.

```
middle :: Int -> Int -> Int -> Bool
middle x y z = (x <= y) && (z <= y)
```

The comparison operators are `==` (equal to), `/=` (not equal to), `<` (less than), `<=` (less than or equal to), `>` (greater than), `>=` (greater than or equal to).

Introduction to Programming, Aug-Dec 2006

Lecture 2, Thursday 10 Aug 2006

Multiple definitions

Haskell does not limit us to a single definition for a function. We can give multiple definitions which are scanned from top to bottom. The first definition that matches is used to compute the value of the output. For instance, here is an alternative definition of `xor`.

```
xor :: Bool -> Bool -> Bool
xor True False = True
xor False True = True
xor b1 b2 = False
```

When does a function invocation match a definition? We have to check that it matches for each argument. If the definition has a variable for an argument, then any value supplied when invoking the function matches on that argument and the value supplied is uniformly substituted for the variable throughout the definition. On the other hand, if the definition has a constant value for an argument, the value supplied when invoking the function must match precisely.

For instance, in the revised definition of `xor`, if we invoke the function as `xor False True`, the first definition does not match, but the second one does. If we invoke the function as `xor True True`, the first two definitions both fail to match and we end up using the third one.

We can use multiple definitions to define a function inductively. For instance, here is a definition of the function *factorial*.

```
factorial :: Int -> Int -> Int
factorial 0 = 1
factorial n = n*(factorial (n-1))
```

If we write, for instance, `factorial 3`, then only the second definition matches, leaving us with the expression `3*(factorial 2)`, after uniformly substituting `3` for `n` and simplifying `(3-1)` to `2`. We use the second definition two more times to get `3*(2*(factorial 1))` and then `3*(2*(1*(factorial 0)))`. Now, the first definition matches, and we get `3*(2*(1*(1)))` which Haskell can evaluate using its built-in rules for `*` to return `6`.

Notice that there is no guarantee that an inductive definition in Haskell is correct, nor that it terminates on all inputs. Reflect, for instance, on what would happen if we invoked our function as `factorial (-1)`.

Observe the bracketing in the second defintion above. We write `n*(factorial (n-1))`. This says we should compute `(n-1)`, then feed this to `factorial` and multiply the result by `n`. If, instead, we write `n*(factorial n-1)`, Haskell would interpret this as `n*((factorial n)-1)`—in other words, feed `n` to factorial, subtract `1` from the result and then multiply by `n`. For arithmetic and relational expressions, the normal precedence rules of arithmetic apply, so an unbracketed expression such as `x <= 5 || y > 6` would be implicitly bracketed correctly as `(x <= 5) || (y > 6)`. However, function application binds more tightly than arithmetic operators, so `factorial n-1` is interpreted as `(factorial n)-1` rather than `factorial (n-1)`.

Function definitions with guards

Often, a function definition applies only if certain conditions are satisfied by the values of the inputs. Here is an example of how to define `factorial` to work with negative inputs. If the input is negative, we negate it and invoke `factorial` on the corresponding positive quantity.

```
factorial :: Int -> Int

factorial 0 = 1
factorial n
  | n < 0 = factorial (-n)
  | n > 0 = n * (factorial (n-1))
```

In this version of `factorial` , the second definition has two options depending on the value of `n`. If `n < 0`, the first definition applies. If `n > 0`, the second definition applies. These conditions are called *guards*, since they restrict entry to the definition that follows. Each guarded definition is signalled using `|`. Notice that lines beginning with `|` are indented. This tells Haskell that these lines are continuations of the current definition.

Observe that we can combine definitions of different types. In this example, the first definition, `factorial 0` is a simple expression while the second defintion is a conditional one.

The guards in a conditional definition are scanned from top to bottom. They may overlap, in which case the definition that is used is the one corresponding to the first guard that is satisfied. For instance, we could write:

```
factorial :: Int -> Int

factorial 0 = 1
factorial n
  | n < 0 = factorial (-n)
  | n > 1 = n * (factorial (n-1))
  | n > 0 = n * (factorial (n-1))
```

Now, `factorial 2` would match the guard `n > 1` while `factorial 1` would match the guard `n > 0`.

The guards in a conditional defintion may also not cover all cases. For instance, suppose we write:

```
factorial :: Int -> Int

factorial 0 = 1
factorial n
  | n < 0 = factorial (-n)
  | n > 1 = n * (factorial (n-1))
factorial 1 = 1
```

Now, the invocation `factorial 1` matches neither guard and falls through (fortunately) to the third definition. If we had not supplied the third definition, any invocation other than `factorial 0` would eventually have tried to evaluate `factorial 1`, for which no match would have been found, leading to the Haskell interpreter printing an error message like the following:

```
Program error: pattern match failure: factorial 1
```

Often, we do want to catch all leftover cases in the last guard. Rather than tediously specify the options that have been left out, we can use the word `otherwise`, as in the following definition of `xor`:

```
xor :: Bool -> Bool -> Bool

xor b1 b2
  | b1 && not(b2) = True
  | not(b1) && b2 = True
  | otherwise     = False
```

In this definition, note that since `b1` and `b2` are of type `Bool`, we can directly write `b1 && not(b2)` instead of the more explicit version `b1 == True && b2 == False`.

More on pattern matching

When we match a function invocation with a defintion involving variables, the variables are uniformly substituted by the values supplied. However, each input variable in the function definition would be distinct. Consider the following function, which checks if both its inputs are equal:

```
isequal :: Int -> Int -> Bool
isequal x y = (x == y)
```

It is tempting to try and rewrite this function as follows:

```
isequal :: Int -> Int -> Bool
isequal x x = True
isequal x y = False
```

The idea would be that the first definition implicitly checks whether both arguments are equal by forcing them to both match x and hence match each other. However, this is illegal in Haskell: each variable on the left hand side of a definition should be distinct.

Sometimes, an argument is not used on the right hand side of a definition. Consider the following definition that computes `x^n`.

```
power :: Float -> Int -> Float

power x 0 = 1.0
power x n | n > 0 = x * (power x (n-1)
```

Here, the value of `x^0` is `1.0` for all values of `x`. In such a situation, we can use a special variable `_` that matches any argument but cannot be used on the right hand side of a definition.

```
power :: Float -> Int -> Float

power _ 0 = 1.0
power x n | n > 0 = x * (power x (n-1)
```

Unlike normal variables, we can use more than one copy of `_` in a definition, since the corresponding value cannot be used on the righthand side in any case. As an example, here is a function that checks if at least two of its three `Bool` arguments are `True`.

```
twoofthree :: Bool -> Bool -> Bool -> Bool
twoofthree True True _ = True
twoofthree True _ True = True
twoofthree _ True True = True
twoofthree _ _ _  = False
```

How Haskell ”computes”

Computation in Haskell is like simplifying expressions in algebra. Relatively early in school, we learn that $(a + b)^2$ is $a^2 + 2ab + b^2$. This means that wherever we see $(x + y)^2$ in an expression, we can replace it by $x^2 + 2xy + y^2$.

In the same way, Haskell computes by rewriting expressions using functions and operators. We say *rewriting* rather than *simpliyfing* because it is not clear, sometimes, that the rewritten expression is “simpler” than the original one!

To begin with, Haskell has rewriting rules for operations on built-in types. For instance, the fact that $6 + 2$ is 8 is embedded in a Haskell rewriting rule that says that `6+2` can be rewritten as `8`. In the same way, `True && False` can be rewritten to `False`, etc.

In addition to the builtin rules, the function definitions that we supply are also used for rewriting. For instance, given the following definition of `factorial`

```
factorial :: Int -> Int -> Int
factorial 0 = 1
factorial n = n*(factorial (n-1))
```

here is how "factorial 3" would be evaluated. In the following, we use $\leadsto$ to denote *rewrite to*:

```
factorial 3  ⇝  3 * (factorial (3-1))
             ⇝  3 * (factorial (2))
             ⇝  3 * (2 * factorial (2-1))
             ⇝  3 * (2 * factorial (1))
             ⇝  3 * (2 * (1 * factorial (1-1)))
             ⇝  3 * (2 * (1 * factorial (0)))
             ⇝  3 * (2 * (1 * 1))
             ⇝  3 * (2 * 1)
             ⇝  3 * 2
             ⇝  6
```

When rewriting expressions, brackets may be opened up to change the order of evaluation. Sometimes, more than one rewriting path may be available. For instance, we could have completed the computation above as follows.

```
factorial 3              ⇝ 3 * (factorial (3-1))
                          ⇝ 3 * (factorial (2))
                        ⇝ 3 * (2 * factorial (2-1))
           ⇝ (3 * 2) * (factorial (2-1)) <== New expression
                         ⇝ 6 * (factorial (2-1)))
                          ⇝ 6 * (factorial (1))
                        ⇝ 6 * (1 * factorial (1-1))
                         ⇝ 6 * (1 * factorial (0))
                                ⇝ 6 * (1 * 1)
                                   ⇝ 6 * 1
                                     ⇝ 6
```

In Haskell, the "result" of a computation is an expression that cannot be further simplified. In general, it is guaranteed that any path we follow leads to the same "result", if a "result" is found. It could be that one choice of simplification could yield a result while another may not. For instance, using our definition of `power`

```
power :: Float -> Int -> Float
power _ 0 = 1.0
power x n | n > 0 = x * (power x (n-1)
```

5

we could consider the expression `power (8.0/0.0) 0`.

Using the first rule, this reduces as

```
power (8.0/0.0) 0 ⇝ 1.0
```

However, if we first try to simplify `(8.0/0.0)`, we get an expression without a value so, in a sense, we have

```
power (8.0/0.0) 0 ⇝ Error
```

Alternatively, we could even try to evaluate an expression such as

```
power (1.0 * factorial (-1)) 0
```

where the first rule for `power` yields the result `1.0` while repeatedly trying to simplify the argument `(1.0 * factorial (-1))` will go into an unending sequence of simplifications yielding no result.

Haskell uses a form of simplification that is called *lazy*—it does not simplify the argument to a function until the value of the argument is actually needed in the evaluation of the function. In particular, Haskell would evaluate both the expressions above to `1.0`. We will examine the consequences of having such a lazy evaluation strategy at a later stage in the course.

Lists

Suppose we want a function that finds the maximum of all values from a collection. We cannot use an individual variable to represent each value in the collection because when we write our function definition we have to fix the number of variables we use, which limits our function to work only with collections that have exactly that many variables.

Instead, we need a way to collectively associate a group of values with a variable. In Haskell, the most basic way of collecting a group of values is to form a list. A list is a sequence of values of a fixed type and is written within square brackets separated by commas. Thus, `[1,2,3,1]` is a list of `Int`, while `[True,False,True]` is a list of `Bool`. The underlying type of a list must be uniform: we cannot write lists such as `[1,2,True]` or `[3.0,'a']`. A list of underlying type `T` has type `[T]`. Thus, `[1,2,3,1]` is of type `[Int]`, `[True,False,True]` is of type `[Bool]`, ...

Lists can be nested: we can have lists of lists. For instance, `[[1,2],[3],[4,4]]` is a list each of whose members is a list of `Int`, so the type of this list is `[[Int]]`.

The empty list is uniformly denoted `[]` for all list types.

Internal representation of lists

Internally, Haskell builds lists incrementally, one element at a time, starting with the empty list. This incremental building can be done from left to right (each new element is tagged on at the end of the current list) or from right to left (each new element is tagged on at the beginning of the current list). For historical reasons, Haskell chooses the latter, so all lists are built up right to left, starting with the empty list.

The basic listbuilding operator, denoted `:`, takes an element and a list and returns a new list. For instance `1:[2,3,4]` returns `[1,2,3,4]`. As mentioned earlier, all lists in Haskell are built up right to left, starting with the empty list. So, internally the list `[1,2,3,4]` is actually `1:(2:(3:(4:[])))`. We always bracket the binary operator `:` from right to left, so we can unambiguously leave out the brackets and write `[1,2,3,4]` as `1:2:3:4:[]`. It is important to note that all the human readable forms of a list `[x1,x2,x3,...,xn]` are internally represented canonically as `x1:x2:x3:...:xn:[]`. Thus, there is no difference between the lists `[1,2,3]`, `1:[2,3]`, `1:2:[3]` and `1:2:3:[]`.

Defining functions on lists

Most functions on lists are defined by induction on the structure of the list. The base case specifies a value for the empty list. The inductive case specifies a way to combine the leftmost element with an inductive evaluation of the function on the rest of the list. The functions `head` and `tail` return the first element and the rest of the list for all nonempty lists. These functions are undefined for the empty list. We can use `head` and `tail` in our inductive definitions.

Here is a function that computes the length of a list of `Int`.

```
length :: [Int] -> Int

length [] = 0
length l  = 1 + (length (tail l))
```

Notice that if the second definition matches, we know that `l` is nonempty, so `tail l` retuns a valid value.

In general, the inductive step in a list based computation will use both the head and the tail of the list to build up the final value. Here is a function that computes the sum of the elements of a list of `Int`.

```
sum :: [Int] -> Int

sum [] = 0
sum l  = (head l) + (sum (tail l))
```

7

Introduction to Programming, Aug-Dec 2006

Lecture 3, Friday 11 Aug 2006

Lists . . .

We can implicitly decompose a list into its head and tail by providing a pattern with two variables to denote the two components of a list, as follows:

```
length :: [Int] -> Int

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

Here, in the second definition, the input list `l` is implicitly decomposed so that `x` gets the value `head l` while `xs` gets the value `tail l`. The bracket around `(x:xs)` is needed; otherwise, Haskell will try to compute `(length x)` before dealing with the `:`. In this example, the list is broken up into a single value `x` and a list of values `xs`. This is to be read as "the list consists of an x followed by many x's" and is a useful convention for naming lists.

Notice that in the second inductive definition of length, `x` plays no role in the right hand side of the second definition, so we could also write:

```
length :: [Int] -> Int

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

We can rewrite `sum` using list pattern matching as follows.

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

Here are some more examples of functions over lists: The function `concatenate` combines two lists into a single larger list.

```
concatenate :: [Int] -> [Int] -> [Int]
concatenate [] ys = ys
concatenate (x:xs) ys = x:(concatenate xs ys)
```

Concatenation is so useful that Haskell has a builtin binary operator `++` for this. Thus `[1,2,3] ++ [4,3]` $\rightsquigarrow$ `[1,2,3,4,3]`, etc.

We can reverse a list by first reversing the tail of the list and then appending the head of the list at the end, as follows.

```
reverse :: [Int] -> [Int]
reverse [] = []
reverse (x:xs) = (reverse xs)++[x]
```

The functions `sum`, `length` and `reverse` are actually basic list functions in Haskell, like the functions `head` and `tail`. Some other useful builtin functions are:

- `init l` returns all but the last element of `l`

  `init [1,2,3]` $\rightsquigarrow$ `[1,2]`

  `init [2]` $\rightsquigarrow$ `[]`

- last l returns the last element in l

  `last [1,2,3]` $\rightsquigarrow$ `3`

  `last [2]` $\rightsquigarrow$ `2`

An important builtin function is `concat`. This function takes a list of lists and "dissolves" one level of brackets, merging its contents into a single long list. For instance:

`concat [[1,2,3],[],[4,5],[],[6]]` $\rightsquigarrow$ `[1,2,3,4,5,6]`

We can write an inductive definition for `concat` in terms of `++`:

```
concat [[Int]] -> [Int]
concat [] = []
concat (l:ls) = l ++ (concat ls)
```

Lists are sequences of values, so the position of a value is important. In the list `[1,2,1]`, there are two copies of the value 1, at the first and third position. Haskell follows the convention that positions are numbered from 0, so the set of positions in a list of length `n` is {`0,1,...,(length n) - 1`}.

The notation `xs!!i` directly returns the element at position `i` in the list `xs`. Note that accessing the element at position `i` in a list takes time proportional to `i`.

Two more useful built-in functions in Haskell are `take` and `drop`. The expression `take n l` will return the first `n` values in `l` while `drop n l` will return the list obtained by omitting the first `n` values in `l`. For any list `l` and any integer `n` we are guaranteed that:

```
l == (take n l) ++ (drop n l)
```

In particular, this means that if `n < 0`, `take n l` is `[]` and `drop n l` is `l`, while if `n > (length l)`, `take n l` is `l` and `drop n l` is `[]`.

Polymorphism

Observe that functions such as `length` and `reverse` work in the same way for lists of any type. It would be wasteful to have to write a separate version of such functions for each different type of list. In Haskell, it is possible to say that a function works for multiple types by using type variables. For instance, we can write:

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

Here, the letter `a` in the type `[a] -> Int` is a type variable. The type `[a] -> Int` is to be read as ""or any underlying type `a`, this function is of type `[a] -> Int`". It is conventional to use letters `a`, `b`, ...to denote types. Note that the variables used in type expressions are disjoint from those used in actual function definitions so one could, in principle, use the same variable in both parts, though it is probably not a good idea from the perspective of readability.

In the same way, we can generalize the types of reverse and concat to read:

```
reverse :: [a] -> [a]

concat  :: [[a]] -> [a]
```

Here it is significant that the same letter `a` appears on both sides of the `->`. This means, for instance, that the type of the list returned by `reverse` is the same as the type of the input list. In other words, all occurrences of a type variable `a` in a type declaration must be instantiated to the same actual type. (Notice that this type of implicit pattern matching was precisely what we disallowed when writing Haskell function definitions such as `isequal x x = True`, so the way type variables are instantiated differs from the way variables in function definitions are instantiated.)

Functions that work in the same way on different types are called *polymorphic*, which means (in Greek) "taking different forms".

We must be careful to distinguish polymorphism of the type we have seen with lists from the ad hoc variety associated with overloading operators. For instance, in most programming languages, we write `+` to denote addition for both integers and floating point numbers. However, since the underlying representations used for the two kinds of numbers are completely different, we are actually using the same name (`+`, in this case) to designate functions that are computed in a different manner for different base types. This type of situation is more properly referred to as *overloading*.

In a nutshell, overloading uses the same symbol to denote similar operations on different types, but the way the operation is evaluated for each type is different. On the other hand, polymorphism refers to a single function definition with a fixed computation rule that works for multiple types in the same way.

Note that we cannot assign a simple polymorphic type for `sum`. It would be wrong to write

```
sum :: [a] -> a
```

because `sum` will work only for lists whose underlying type supports addition. We will see later that it is possible to write a conditional type expression such as `sum :: [a] -> a` provided the type `a` supports the operation `+`.

Haskell allows us to pass any type to a function, including another function. Consider the function `apply`, that takes as input a function `f` and a value `x` and returns the value `(f x)`. In other words, this functions *applies* `f` to `x`. The definition of `apply` is very straightforward:

```
apply f x = f x
```

What is the type of `apply`? The first argument is any function, so we can denote its type as `a -> b` for some arbitrary types `a` and `b`. The second argument `x` has to be fed as an input to `f`, so its type must be `a`. The output of `apply` is `f x`, which has type `b`. Thus, we have,

```
apply :: (a -> b) -> a -> b
```

Notice that we must put brackets around `(a->b)` to ensure that this is not seen as a function of three variables.

What if we change the function to apply `f` twice to `x`?

```
twice f x = f (f x)
```

In this case, we see that the output `(f x)` is fed back as an input to `f`. This means that the input and output types `a` and `b` must be the same, so `f :: a -> a` and the type of `twice` is given by

```
twice :: (a -> a) -> a -> a
```

The analysis we did by hand when trying to deduce the type of `apply` and `twice` is built in to Haskell. Thus, if we do not provide an explicit type for a function, Haskell will start with the most general assumption about the type and impose the constraints inferred from the function definitions to arrive at a final type.

As a last remark, recall that in our discussion of functions with multiple inputs we said that each input transforms the original function into a new one. Thus, when we write `plus m n = m+n`, after the input `m` is provided, we get a new function `(plus m)` which is of type `Int -> Int`. This intermediate function actually does exist in “real” life. For instance, if we write `apply (plus 7) 8`, we get the answer `15`.

The datatype `Char`

In Haskell, character constants are written within single quotes, like `'a'`, `'3'`, `'%'`, `'#'`, ... Like all other datatypes, characters are encoded in a table.

Two functions are provided that allow us to interpret characters in terms of their internal position in the table and vice versa.

```
ord :: Char -> Int
chr :: Int -> Char
```

These are inverses of each other, so for each `Char c`, `c == chr(ord c)`, and for each `Int i` that is a valid encoding of a `Char`, `i == ord (chr i)`.

The actual way in which characters are organized in a table may vary from one system to another. In practice, most current systems use the encoding called ASCII in which characters are represented by positive binary numbers from 0 to 255. However, some languages use a larger range of encodings to accommodate characters from alphabets of different languages.

We shall assume only the following facts about the encoding of characters in Haskell.

- The characters `'a'`, `'b'`, ..., `'z'` occur consecutively.
- The characters `'A'`, `'B'`, ..., `'Z'` occur consecutively.
- The characters `'0'`, `'1'`, ..., `'9'` occur consecutively.

This means that `ord 'b'` is always `(ord 'a') + 1`, and `(ord 'A' - ord 'a') == (ord 'B' - ord 'b')` etc.

Let us define a function `capitalize` that maps `'a'`, `'b'`, ..., `'z'` to `'A'`, `'B'`, ..., `'Z'` and leaves all other characters unchanged.

Here is a brute force definition of `capitalize` that makes use of the fact that we have 26 values to transform.

```
capitalize :: Char -> Char
capitalize 'a' = 'A'
capitalize 'b' = 'B'
.
.
.
capitalize 'y' = 'Y'
capitalize 'z' = 'Z'
capitalize c = c
```

The first 26 lines do the capitalization. The last line preserves all values other than `'a'`, `'b'`, ..., `'z'`.

Here is a slightly smarter version of the same function that uses the fact that the displacement between a lower case letter and its capitalized version is a constant.

5

```
capitalize :: Char -> Char
capitalize c
  | ('a' <= c && c <= 'z') = chr (ord c + (ord 'A' - ord 'a'))
  | otherwise              = c
```

Notice that we are allowed compare the order of characters—this is used to check if `c` lies between `'a'` and `'z'`. Comparison is based on the `ord` value of a character : character `x` is less than character `y` if `ord x < ord y`.

However, we cannot perform arithmetic on characters. Expressions such as `'a' + 2` are illegal. Thus, while it is true that `'c' == chr (ord 'a' + 2)`, it is nonsensical to claim that `'c' = 'a' + 2`.

Strings

Programs that manipulate text deal with sequences of characters, not single characters. A sequence of characters is normally called a *string*. In Haskell, a sequence of characters is just a list of `Char`, or a value of type `[Char]`. The word `String` is a synonym for `[Char]`. Also, instead of writing strings using somewhat tedious list notation such as `['h','e','l','l','o']` we are allowed to directly write the string in double quotes, `"hello"` in this case.

Manipulating a `String` is easy—a `String` is just a list of `Char` so all list function work on `String` just like any other list. Thus, `length` can be used to get the length of a `String`, `concat` can be used to collapse a list of `String`s into a single long `String` etc.

`length "hello"` $\rightsquigarrow$ `5`

`concat ["hello"," ","world"]` $\rightsquigarrow$ `"hello world"`

It is important to remember that single quotes denote character constants while double quotes denote strings. For instance, `'a'` is the character `a` while `"a"` is the list `['a']`.

Here is an example of a function on `String`. This function uses the function `capitalize` we wrote earlier to convert each letter in a `String` to uppercase.

```
touppercase :: String -> String

touppercase "" = ""
touppercase (c:cs) = (capitalize c):(touppercase cs)
```

Notice that in the inductive definition we use `""` to denote the empty string. This is translated to `[]`, the empty list of `Char`.

Here is another example of a function on `String`s—`exists` checks whether a given character occurs in a given string.

6

```
exists :: Char -> String -> Bool

exists c "" = False
exists c (x:xs)
  | c == x    = True
  | otherwise = exists c xs
```

Thus, as we march along the `String`, if we find a copy of the `Char` we are looking for, we report `True` and stop. Otherwise, we continue looking along the rest of the `String`. If we don't find a copy of what we are looking for, we eventually reach the empty string and return `False`.

Exercise Write a function `position :: Char -> String -> Int` such that `position c s` returns the first position in `s` where `c` occurs and returns `-1` if `c` does not occur in `s`. Note that a valid answer for this function must be either `-1` or a number in the range {`0,1,...,length s - 1`}.

Introduction to Programming, Aug-Dec 2008
Lecture 4, Wed 13 Aug 2008

Tuple types

What if we want to group together values of different types? For instance, we may want to maintain a collection of lists of [Float] in which we store, along with each list, its length. Thus, each element in our collection is a pair (fs,n) where fs::[Float] and n::Int.

In Haskell terminology, the values we store belong to the tuple type ([Float],Int). A list of tuples of this form will therefore have type [([Float],Int)]. Notice that this does not violate our requirement that a list has a uniform underlying type --- each element of the list is of type ([Float],Int).

For instance, here is a function that takes a list of strings and returns, for each string in the list, the string and its length.

```
  stringlengths :: [String] -> [(String,Int)]
  stringlengths [] = []
  stringlengths (x:xs) = (x,length x):(stringlengths xs)
```

Here is an example in which we compute the distance between points in two-dimensional space. Each point is represented as a pair of Floats.

```
  distance :: (Float,Float) -> (Float,Float) -> Float
  distance (x1,y1) (x2,y2) = sqrt ((x2-x1)*(x2-x1) +
                                   (y2-y1)*(y2-y1))
```

Notice that in the definition of the function, we can use pattern matching to directly decompose the tuple into its constituent parts.

We are not restricted to pairs when defining tuple types: we can construct n-tuples. For instance, we could easily generalize the earlier definition to three-dimensional points as follows:

```
  distance :: (Float,Float,Float) -> (Float,Float,Float) -> Float
  distance (x1,y1,z1) (x2,y2,z2) = sqrt ((x2-x1)*(x2-x1) +
                                         (y2-y1)*(y2-y1) +
                                         (z2-z1)*(z2-z1))
```

Some times, we might want to supply a new name for a compound type to ease understanding of a program. For instance, we might want to use the word Point to denote (Float,Float) so that we can define the type of distance to be

```
  distance :: Point -> Point -> Float
```

which reads better than the original definition. This is achieved using a "type" definition, as follows:

```
  type Point = (Float,Float)
```

It is important to recognize that this definition does not define a new type --- it just says that Point is a synonym for (Float,Float). Thus, if we have functions as follows:

```
  f :: Float -> Float -> Point
  g :: (Float,Float) -> (Float,Float) -> Float
```

it is legal to write an expression of the form

```
  g (f x1 y1) (f x2 y2)
```

In this expression, the two instances of f produce outputs of type Point, while g is expecting its inputs to be of type (Float,Float). However, Point and (Float,Float) are just different names for the same basic type, so the outputs of the two instance of f are compatible with the type declared for the inputs to g.

Defining local functions using where

Let us return to our function of distance.

```
  distance :: (Float,Float) -> (Float,Float) -> Float
  distance (x1,y1) (x2,y2) = sqrt ((x2-x1)*(x2-x1) +
                                   (y2-y1)*(y2-y1))
```

The expressions (x2-x1)*(x2-x1) and (y2-y1)*(y2-y1) are instances of a more general function that squares its input. So, we could write

```
  sqr :: Float -> Float
  sqr z = z*z

  distance :: (Float,Float) -> (Float,Float) -> Float
  distance (x1,y1) (x2,y2) = sqrt (sqr (x2-x1) + sqr(y2-y1))
```

It is immediately that this version of distance is more readable than the previous version, thanks to the use of an auxiliary function sqr. However, one undesirable aspect of this definition is that the auxiliary function sqr, which is required only in distance, is now globally available. One problem with this is that we cannot now define any other function called sqr because the name is in use. What we need is a way to temporarily define sqr so that it is part of the definition of distance, but is not visible outside. This can be achieved as follows:

```
  distance :: (Float,Float) -> (Float,Float) -> Float
  distance (x1,y1) (x2,y2) = sqrt (sqr (x2-x1) + sqr(y2-y1))
    where
    sqr :: Float -> Float
    sqr z = z*z
```

In this version of distance, the defintion of sqr is local to distance and is not visible outside. Observe that the word where is indented with respect to the main definition. Since the word where has a special status (Haskell knows that it is not the name of function being defined) we need not indent sqr with respect to where.

Another reason to use local declarations is to identify common subexpressions and ensure that they are computed only once. Returning to the original version of distance

```
  distance :: (Float,Float) -> (Float,Float) -> Float
  distance (x1,y1) (x2,y2) = sqrt ((x2-x1)*(x2-x1) +
                                   (y2-y1)*(y2-y1))
```

we observe two instances of the subexpressions (x2-x1) and (y2-y1) in the definition. In general, Haskell will evaluate these quantities afresh each time they are encountered. To indicate that these expressions are the same, we could write

```
  distance :: (Float,Float) -> (Float,Float) -> Float
  distance (x1,y1) (x2,y2) = sqrt ( xdiff*xdiff + ydiff*ydiff)
    where
    xdiff :: Float
    xdiff = x2-x1
    ydiff :: Float
    ydiff = y2-y1
```

In this definition, xdiff and ydiff can be thought of as constant functions of 0 arguments that always return a fixed argument.

Functions on lists

Map

Often, we need to operate on lists by transforming each element in a fixed manner. For instance, suppose we want to square each element in a list of integers. We can do this inductively, as usual:

```
  sqrall :: [Int] -> [Int]
  sqrall [] = []
  sqrall (n:ns) = (n*n):(sqrall ns)
```

The function stringlengths we wrote above is also of the same general variety. Here, each string in the input list is transformed into a pair of values in the output list.

The builtin function map allows us to apply a function f "pointwise" to each element of a list. In other words,

```
  map f [x0,x1,..,xk] = [(f x0),(f x1),...,(f xk)]
```

Here is an inductive definition of map.

```
  map f [] = []
  map f (x:xs) = (f x):(map f xs)
```

What is the type of map? The function f is in general of type a->b. The list that map operates on must be compatible with the function f, so it must be of type [a]. The list generated by map is of type [b]. Thus, we have

```
  map :: (a -> b) -> [a] -> [b]
```

Thus, map is a polymorphic function, like the functions length, sum, reverse etc that we wrote for lists earlier. Notice that there are two type variables, a and b, in the type definition for map. These can be independently instantiated to different types, and these instantiations apply uniformly to all occurrences of a and b.

An important point to notice is that we can pass a function to another function, as in the definition and use of "map", without any fuss in Haskell. There is no restriction in Haskell about what we can pass as an argument to a function: if it can be assigned a type, it can be passed.

We can now write the functions sqrall and stringlengths in terms of map:

```
  sqrall l = map sqr l
    where
    sqr :: Int -> Int
    sqr n = n*n

  stringlengths l = map strlen l
    where
    strlen :: String -> (String,Int)
    strlen s = (s,length s)
```

In sqrall, the input and output types of the function sqr passed

to map are of the same type, Int, whereas in stringlengths, the
input type of strlen is String and the output type is
(String,Int).

Filter

Another useful operation on lists is to select elements that
match a certain property. For instance, we can select the even
numbers in a list of even integers as follows.

```
  evenonly :: [Int] -> [Int]
  evenonly [] = []
  evenonly (n:ns)
    | mod n 2 == 0  = n:(evenonly ns)
    | otherwise     = evenonly ns
```

We have used the builtin function mod to check that a number is
even: mod m n returns the remainder that results when m is
divided by n. A related function is div --- div m n returns the
integer part of m divided by n.

We can think of evenonly as the result of applying the check

```
  iseven :: Int -> Bool
  iseven n = (mod n 2 == 0)
```

to each element of the input list and retaining those values that
pass this check.

This is a general principle that we can use to "filter" out
values from a list. Let l be of type [a] and let p be a
function from a to Bool. Then, we have the function

```
  filter : (a -> Bool) -> [a] -> [a]
  filter p [] = []
  filter p (x:xs)
    | (p x)     = x:(filter p xs)
    | otherwise = filter p xs
```

Notice that the output of filter is a sublist of the original
list, so the output list has the same type as the original list.

```
Introduction to Programming, Aug-Dec 2008
Lecture 5, Mon 25 Aug 2008

Notation for lists
------------------

Haskell provides some convenient notation for lists consisting of
a continuous sequence of values.  The list [m..n] is the list of
elements [m,m+1,m+2,...,n].   If n < m, this list is defined to
be empty.  Thus, for example,

   [1..7] = [1,2,3,4,5,6,7]
   [3..3] = [3]
   [4..3] = []

The starting and ending values can be variables.

An extension of this notation can be used to define arithmetic
progressions, and, as a special case, lists in descending order.
An arithmetic progression is specified by the first two values
(this defines the separation between elements in the sequence)
and an upper bound.  For instance,

  [1,3..8] = [1,3,5,7]
  [2,5..19] = [2,5,8,11,14,17]

Notice that the upper bound may not actually be part of the list
being defined.  The rule is that we keep inserting elements with
the separation defined by the first two elements so long as we do
not cross the upper bound.

The difference between the first two elements can be negative, in
which case we get a list in descending order.

  [8,7..5] = [8,7,6,5]
  [12,8..-9] = [12,8,4,0,-4,-8]

This notation works for any basic type that can be enumerated ---
that is, for each value in the type, there is a well defined
"next" value.  This is true for Char and Bool.  Bool is a
somewhat trivial case in which the two values are arranged so
that False is less than True.  Char is more interesting --- the
next character is the one with the next ord value in the table
representing characters.  As we saw, we can assume that
'a','b',..,'z' form a contiguous sequence in the table.
Likewise, the capital letters 'A','B',...,'Z' form a contiguous
sequence as do the digits '0','1',...,'9'.  Thus, we can write
lists such as

  ['a'..'f']     = ['a','b','c','d','e','f'] = "abcdef"
  ['Z','X'..'S'] = ['Z','X','V','T']         = "ZXVT"

List comprehension: combining map and filter
--------------------------------------------

In set theory, we can build new sets from old sets using notation
called "set comprehension".  For instance, given the set  of
integers {1,2,..,m}, we can define the set of squares of the even
numbers in this set as

        { n^2 | n in {1,2,..,m}, even(n)}

where even(n) is a predicate that evaluates to true precisely
when n is even.

Analogously, we can build new lists from old lists using "list
comprehension".  For instance, the list of squares of all the
even numbers from 1 to m is given by
```

```
     [ n^2 | n <- [1..m], iseven n ]
```

where iseven is the function we wrote earlier to check if n is even.

The notation "<-" is supposed to look like the "element of" notation from set theory. The way this expression is interpreted is as follows:

   For each n in the list [1..m], if "iseven n" is true, append n^2 to the output

The first part, n <- [1..m], is referred to as the generator, which supplies the initial list of values to be operated on. The function "iseven n" serves to filter the list while "n^2" on the left of the "|" is a function that is mapped onto all elements that survive the filter. We shall examine the precise relationship between list comprehension and map/filter later. For now, let us consider an illustrative example.

We first write a function to compute the list of divisors of a number n.

```
   divisors n = [ m | m <- [1..n], mod n m == 0 ]
```

This says that the divisors of n are precisely those numbers between 1 and n such that there is no remainder when n is divided by the number.

We can now, for instance, write a function that checks that n is a prime number, as follows:

```
   prime n = (divisors n == [1,n])
```

In other words, n is a prime if its list of divisors is precisely [1,n]. Notice that 1 (correctly) fails to be a prime under this criterion because divisors 1 = [1] which is not the same as [1,1].

Multiple generators

We can use more than one generator in a list comprehension. Here is an example

```
  triples n = [ (x,y,z) | x <- [1..n], y <- [1..n], z <- [1..n] ]
```

If there are multiple generators, later generators move faster than earlier ones, so the output of triples n would be

```
  [(1,1,1),(1,1,2),...,(1,1,n),(1,2,1),(1,2,2),...,
   (1,2,n),...,(n,1,1),(n,1,2),...,(n,n,n)]
```

Later generators can depend on earlier values. Thus

```
  triples n = [ (x,y,z) | x <- [1..n], y <- [x..n], z <- [y..n] ]
```

is the same as

```
  triples n = [ (x,y,z) | x <- [1..n], y <- [1..n], z <- [1..n],
                          x <= y && y <= z ]
```

A generator can use any expression that puts out a list. For instance, we can run a generator over the output of triples as follows to generate the set of all Pythagorean triples with entries less than or equal to n.

```
  pythagoras n = [(x,y,z) | (x,y,z) <- triples n,
                               x^2 + y^2 == z^2 ]
```

```
List comprehension
------------------

Translating list comprehensions
-------------------------------
```

Though list comprehension notation is extremely convenient, it is important to recognize that it is only a way of combining map and filter in a more readable format.

In general, a list comprehension is of the form [e | Q] where e is an expression and Q is a list of generators and conditions. Roughly speaking, the expressions in e correspond to functions that have to be mapped over the lists specified by the generators in Q after filtering out elements according to the conditions in Q.

We can formally translate list comprehensions by induction on Q. Here is a first attempt to translate list comprehension using map and filter:

```
  [ e | x <- xs, Q] = map f xs where f x = [ e | Q ]
  [ e | p, Q ]      = if p then [ e | Q ] else []
```

Here's how this translation would work:

```
  [ n^2 | n <- [1..7], mod n 2 == 0 ]
  ==> map f [1..7] where f n = [ n^2 | mod n 2 == 0]
  ==> map f [1..7] where f n = if (mod n 2 == 0) then [n^2] else []
  ==> [[],[4],[],[16],[],[36],[]]
```

As we can see, something has gone wrong --- there is an extra level of brackets in the output. To get rid of this we need the function concat, which dissolves one level of brackets in a list. For example:

```
  concat [[1,2,3],[4,5],[6]] = [1,2,3,4,5,6]
  concat [[],[4],[],[16],[],[36],[]] = [4,16,36]
```

Now, we can give a correct translation of list comprehension in terms of map, filter and concat.

```
  [ e | x <- xs, Q] = concat map f xs where f x = [ e | Q ]
  [ e | p, Q ]      = if p then [ e | Q ] else []
```

The previous example that we worked out now becomes:

```
  [ n^2 | n <- [1..7], mod n 2 == 0 ]
  ==> concat map f [1..7] where f n = [ n | mod n 2 == 0]
  ==> concat map f [1..7] where
        f n = if (mod n 2 == 0) then [n^2] else []
  ==> concat [[],[4],[],[16],[],[36],[]]
  ==> [4,16,36]
```

```
Operators vs binary functions
-----------------------------
```

We have seen that some builtin arithmetic operations are defined as functions --- for instance, div and mod. These are applied to their arguments like other functions we write in Haskell, by placing the arguments after the function name. For instance, we have to write "div 7 2" and "mod m 3" rather than "7 div 2" or "m mod 3" though the second form is more natural to read.

On the other hand, we have operations such as + and * which always occur between their operands (this is called infix notation).

```
For binary operations, it turns out that we freely convert
functions into operators and vice versa.  Any function can be
used as an infix operator by enclosing it in backquotes (`).
Thus, we can write

   7 `div` 2
   m `mod` 3

In the other direction, any infix operator can be used as a
function written before its arguments by enclosing it in
parenthesis.  Thus, we can write

   (+) 7 2   for  7+2
   (*) 3 5   for  3*5

Normally, when we define functions in Haskell, we use names that
start with a lower case letter and have only letters and digits
in the name.  We can also define binary functions whose names are made
up entirely of punctuation marks (with some restrictions).  Such
functions can be used as operators.  To define the type of these
functions, we need the () notation earlier.  For instance, we
could define an operator *** such that m *** n = (m+n)^3 as
follows:

(***) :: Int -> Int -> Int
(***) m n = (m+n)^3

Alternatively, we can use the operator form in the definition of
the function (but not in the definition of the type!)

(***) :: Int -> Int -> Int
m *** n = (m+n)^3


Some useful functions defined on lists
--------------------------------------

  sum xs   -- adds up the elements in xs for a list of numbers
  and xs   -- evaluates to True over xs of type [Bool] if all
              elements in xs [Bool]  are True.  In particular,
              and [] = True because for every x in [] (there
              aren't any!), x is True


The function zip
----------------

zip combines two lists into a list of pairs.  For instance,

  zip [1,2] ['a','b'] = [(1,'a'),(2,'b')]

If one list is longer than the other one, zip stops constructing
pairs when the shorter list runs out.  Thus,

  zip [1..4] ['a','b','c'] = [(1,'a'),(2,'b'),(3,'c')]

Here are a couple of examples that use zip.

Example 1 (nondecreasing)
-------------------------

Suppose we want to write a function

  nondecreasing :: [Int] -> Bool

such that nodecreasing [x_1,x_2,...,x_n] = True if
      x_1 <= x_2 && x_2 <= x_3 && ... && x_n-1 <= x_n
```

```
We can write an inductive definition of nondecreasing.

  nondecreasing [] = True
  nondecreasing [x] = True
  nondecreasing (x:y:ys) = (x <= y) && (nondecreasing (y:ys))

Another way to do this is to observe that we are checking

             x_1 <= x_2
        &&   x_2 <= x_3
        ..
        &&   x_n-1 <= x_n

If we pair up xs with (tail xs) we get the list of pairs
[(x_1,x_2), (x_2,x_3), ... , (x_n-1,x_n)]

We can thus write nondecreasing as follows:

  nondecreasing xs = and [x <= y | (x,y) <- zip xs (tail xs) ]

As we saw earlier, the function and returns True provided x <= y
for each pair (x,y) in the output of zip.

What happens in the case "nondecreasing []"?  For the empty list
[], tail is not defined and hence this expression should fail.
However, recall that Haskell uses a lazy evaluation strategy.  To
construct "zip l1 l2",  Haskell proceeds as follows:

-  Extract the head x1 of l1, if any.  If l1 has no more elements,
   quit.
-  Extract the tail x2 of l2, if any.  If l2 has no more elements,
   quit.
-  Add (x1,x2) to the output of zip and repeat this process with
   (tail l1) and (tail l2).

Thus, if l1 is [], zip quits with an empty list as output,
without evaluating l2.  This means that the invalid function
"tail []" is never evaluated in the case when we compute
"nondecreasing []".

It is matter of taste whether the definition of nondecreasing in
terms of zip is better than the original one.  For someone not
very familiar with Haskell, the first definition is more direct,
but the second would be preferred by someone who knows the
language.

Example 2 (position)
--------------------

Recall our old exercise:


    Write a function position :: Char -> String -> Int such that

      position c s

    returns the first position in s where c occurs and returns -1
    if c does not occur in s.  Note that a valid answer for this
    function must be either -1 or a number in the range
    {0,1,...,length s - 1}.


It is not difficult to write a function that
merely checks whether c occurs in s.


    exists :: Char -> String -> Bool
    exists c "" = False
    exists c (x:xs) |
      | c == x    = True
      | otherwise = exists c xs
```

The problem is to identify the position at which we have found the first occurrence of c in s, because this information is not directly available in the inductive definition of exists.

The solution is to write an auxiliary function that takes three parameters, c, s and n, where n is the current position that we are examining in s. Initially, we call this auxiliary function with n set to 0. When we inductively examine (tail s), we increment n to n+1. If we find c at (head s) we return n. If s becomes empty, we return -1, regardless of the value of n. Here is a complete definition of position in terms of this auxiliary function.

```
  position :: Char -> String -> Int
  position c s = auxpos c s 0
    where
    auxpos :: Char -> String -> Int -> Int
    auxpos c "" n = -1
    auxpos c (x:xs) n
      | c == x    = n
      | otherwise = auxpos c xs (n+1)
```

Another approach is to tag each element of xs with its position. This can be achieved using zip as follows:

```
  zip xs [0..(length xs)]
```

Notice that the last position is actually (length xs) - 1 but zip will automatically discard the spurious value.

Now, we can extract the list of all positions where x occurs in xs

```
  allpos x xs = [ i | (y,i) <- zip xs [0..(length xs)], x == y ]
```

If (allpos x xs) is nonempty, head (allpos x xs) gives us the value that we seek. What if (allpos x xs) is empty? We have to return -1 in this case. We can do this by tagging on -1 at the end of (allpos x xs). Thus, if (allpos x xs) is empty head ((allpos x xs) ++ [-1]) returns -1, otherwise it returns the first real position where x occurs in xs. Putting all this together, we get

```
  position x xs = head ([ i | (y,i) <- zip xs [0..(length xs)],
                              x == y ]
                        ++ [-1])
```

==

```
Introduction to Programming, Aug-Dec 2008
Lecture 6, Wed 27 Aug 2008

Folding functions through a list
--------------------------------

Consider the following inductive definitions over lists:

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

  and :: [Bool] -> Bool
  and [] = True
  and (x:xs) = x && (and xs)

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

All of these have the general form:

  f [] = y
  f (x:xs) = x op (f xs)

Thus, given a binary function op and an initial value y, we can
evaluate these functions from right to left as follows

     x_1   x_2   .  .  .  x_n-1  x_n  y
      |                     |     \  /
      |                     |      op
      |                     |       |
      |                     |      y_n
      |                      \    /
      |                        op
      |                        |
      |                       y_n-1
      |     .  .  .      . . .
      |
      |       y_2
       \     /
         op
          |
         y_1

where y_1 is the final value to be returned.

As can be seen from the figure, we are "folding" the binary
function op from right to left through the list.  Since op is a
binary function, we have to provide a second argument y to
combine with the last element x_n.  After this, each intermediate
value is used as the right argument to op and the next element of
the list is taken as the left argument.

This folding operation is described by the Haskell function
foldr, which takes three inputs: the function to be folded, the
initial value, and the list to be folded.

Thus, we can write

   sum xs = foldr (+) 0 xs
   and xs = foldr (&&) True xs
   concat xs = foldr (++) [] xs

More concisely, since xs is the rightmost argument on both sides,
we can write:

   sum = foldr (+) 0
   and = foldr (&&) True
   concat = foldr (++) []
```

What is the type of foldr? In the three examples above, the
function passed to foldr had the same types for both its inputs
and its output. However, there is no reason for this. Suppose
that the two inputs to the function op are of types a and b,
respectively.

Initially, we have op x_n y = y_n-1, where x_n is of type a and y
is of type b. At the next step, we have to apply op to x_n-1 and
y_n-1. Since the second argument to op must be of type b, this
constrains y_n-1 to be of type b. In other words, we have the
following general types:

```
   The function passed to foldr :: a->b->b
   The initial value :: b
   The input list :: [a]
   The output value :: b
```

So, the type of foldr is

```
   foldr :: (a->b->b) -> b -> [a] -> b
```

The function foldr itself can be defined inductively:

```
   foldr f y [] = y
   foldr f y (x:xs) = f x (foldr f y xs)
```

Notice that folding from the right is a natural consequence of
inductively decomposing lists from the left. This is the
efficient way to decompose lists because of the internal
representation of lists in Haskell.

foldl

We can define a symmetric function, foldl, that folds a function
f from left to right through a list.
Thus, given a binary function op and an initial value y, we can
evaluate op functions from left to right as follows.

```
    y   x_1   x_2   . . .  x_n
     \  /      |            |
      op       |            |
       |       |            |
      y_1     /             |
         \   /              |
          op                |
           |                |
          y_2               |
             \              |
               ...          |
                    \       |
                y_(n-1)    /
                      \   /
                       op
                        |
                       y_n
```

where y_n is the final value to be returned.

To illustrate the use of foldl, we revisit the problem of
converting a string to a number. The function we want to
construct is

```
  strtonum :: String -> Int
```

The convention we adopt is that the characters ’0’, ’1’, ..., ’9’
denote the number 0,1,...,9 and all other characters are

interpredted as 0. Thus strtonum "138" should return 138 and strtonum "1ab9" should return 1009.

We begin with a function chartonum that converts a single characte to a single digit integer. One way to write this function is to explicitly match the relevant characters ’0’, ’1’, ..., ’9’ and assign 0 to all other values of the input.

```
  chartonum :: Char -> Int
  chartonum ’0’ = 0
  chartonum ’1’ = 1
  ...
  chartonum ’9’ = 9
  chartonum  x  = 0
```

Alternatively, we could have made use of the fact that ’0’ to ’9’ are consecutive in the internal table representing characters and written

```
  chartonum c
    | (c >= ’0’ && c <= ’9’) = 0 + (ord c - ord ’0’)
    | otherwise              = 0
```

Now, to convert a String to an Int, we start from the left and inductively build up a number. At each step, we multiply the number we have so far by 10 and add the next digit.

Pictorially, we have

```
    0   d_1   d_2   . . .  d_n
    \   /      |            |
     \ /       |            |
 k_1=10*0+d_1 /             |
       \     /              |
   k_2=10*k_1+d_2           |
            \               |
              ...           |
                  \         |
                 k_n-1     /
                     \    /
                k_n=10*k_n-1+d_n
```

The operation that combines the partially constructed number with the next digit is the following:

```
  combine :: Int -> Char -> Int
  combine n c = 10*n + (chartonum c)
```

We can now write strtonum as

```
  strtonum = foldl combine 0
```

Observe that the type of foldl is the following:

```
  foldl :: (b->a->b) -> b -> [a] -> b
```

The difference is in the type of the function f passed to foldl; since we start from the left, the type of f is (b->a->b) and not (a->b->b) as in foldr, where we start from the right.

Can we define foldl inductively? A naive definition would be the following:

```
  foldl :: (b->a->b) -> b -> [a] -> b
  foldl f 0 [] = 0
  foldl f 0 l  = f (foldl f 0 (init l)) (last l)
```

However, this is not an efficient definition because computing

init l and last l takes time proportional to the length of the
list.

Exercise:

  Devise a more efficient definition for foldl. Hint: Use an
  auxiliary function that explicitly maintains the incremental
  value that is being computed.

Folding on nonempty lists

We wrote the function sum as

```
  foldr (+) 0
```

If we write a corresponding function to compute the product of
all elements of a list, it would be

```
  foldr (*) 1
```

This would work fine for nonempty lists but would give the
somewhat counterintuitive answer that the product of an empty
list is 1.

To consider another example, suppose we want to find the maximum
value in a list. Intuitively, this corresponds to folding the
builtin function max through the list. Once again, there is a
problem defining the maximum value of an empty list. Moreover,
unlike the case of product, it is not even clear how to define
this function --- the default value we supply must be smaller
than any value actually in the list, which means we have to rely
on the underlying system.

To get around these difficulties, Haskell provides the functions
foldr1 and foldl1 that work exactly like foldr and foldl,
respectively, but only for nonempty lists. If the list has only
one element, no folding is done and that element is returned. If
the list has two or more elements, the function supplied to
foldr1 (or foldl1) is be folded through the list beginning with
the last two (or first two) elementds in the list.

Here is an inductive definition of foldr1:

```
  foldr1 :: (a->a->a) -> [a] -> a
  foldr1 f [x] = x
  foldr1 f [x,y] = f x y
  foldr1 f (x:y:ys) = f x (foldr1 f (y:ys))
```

Notice that the function passed to foldr1 has type a->a->a,
unlike the type a->b->b of the function passed to foldr. This is
because both arguments to f come from the list and the answer
must again be of the same type.

Given this, we can define the functions product of a list and
maximum value of a list in terms of foldr1 (and foldl1).

```
  product = foldr1 (*) = foldl1 (*)
  maxlist = foldr1 max = foldl1 max
```

==

Accumulating intermediate values : scanl and scanr

As we have seen, the function foldl folds a function f through a
list and produces a single final value. Pictorially, we had

```
    y   x_1   x_2   . . .  x_n
```

```
     \  /      |            |
      op       |            |
       |       |            |
      y_1     /             |
         \   /              |
          op                |
           |                |
          y_2               |
             \              |
               ...          |
                    \       |
                y_(n-1)    /
                      \   /
                       op
                        |
                       y_n
```

The computation of y_n involves generating the intermediate values y, y_1, ..., y_(n-1). These correspond to "partial" answers of foldl for prefixes of the list. Sometimes, these partial answers are also interesting and they can be returned with extra effort. The Haskell function scanl achieves this. In other words,

```
  scanl f m l = [y,y_1,...,y_n]
```

where [y,y_1,...,y_n] is the list of intermediate values generated when computing

```
  foldl f m l
```

Thus, we have

```
  scanl (+) 0 [1..n] = [0,1,(1+2),....,(1+2+..+n)]
  scanl (*) 1 [1..n] = [1!,2!, 3!,...,n!]
```

Symmetrically, scanr returns the partial values generated when evaluating foldr. Recall that the picture for foldr was the following:

```
     x_1   x_2   .  .  .  x_n-1  x_n  y
      |                     |     \  /
      |                     |      op
      |                     |       |
      |                     |      y_n
      |                      \    /
      |                        op
      |                         |
      |                       y_n-1
      |     .  .  .      .  .  .
      |
      |       y_2
       \     /
         op
          |
         y_1
```

Thus,

```
  scanr f m l = [y_1,...,y_n,y]
```

where [y_1,...,y_n,y] is the list of intermediate values generated when computing

```
  foldr f m l
```

Notice that the output of scanl is the list of foldl values for longer and longer prefixes of l while the output of scanr is the list of foldr values for shorter and shorter suffixes of l.

==

Combinatorial functions on lists

We now look at some combinatorial functions on lists. All of these can be defined inductively in terms of the structure of the list. However, we will also see that we can use alternative notation to simplify these definition.

Initial segments

We begin with the function initsegs that lists out the initial segments of a list. An initial segment of a list is a prefix --- that is, a sublist that includes the first k elements of l for some k.

The idea is straightforward. The smallest initial segment of a list is the empty list. For a list of the form (x:xs), the initial segments can be obtained by inserting an x at the head of each initial segments of xs (and explicitly adding a fresh empty initial segment). Each initial segment is itself a list, so initsegs returns a list of lists.

Here is an inductive definition of initsegs:

```
  initsegs :: [a] -> [[a]]
  initsegs [] = [[]]
  initsegs (x:xs) = [[]] ++ [x:l | l <- initsegs xs]
```

An alternative, "purer", definition is

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

Notice that we can generate a much more direct definition in terms of take.

```
  initsegs l = [take n l | n <- [0..length l]]
```

One application of initsegs is to define scanl. scanl can be seen as repeated application of foldl on each initial segment of the givn list. In other words

```
  scanl f a l = map (foldl f a) (initsegs l)
```

or

```
  scanl f a l = [foldl f a ll | ll <- initsegs l]
```

All permutations of a list

Our next task is to generate a function that lists out all permutations of a list. An inductive definition would require defining "permutations (x:xs)" in terms of "permutations xs". The logical way to lift "permutations xs" to "permutations (x:xs)" is to insert x in each possible position within each permutation of xs.

We begin with a function "interleave" which achieves the task of inserting a value in every possible position of a list.

```
  interleave :: a -> [a] -> [a]
  interleave x [] = [[x]]
  interleave x (y:ys) = [x:y:ys] + map (y:) (interleave x ys)
```

Alternatively, we have

```
  interleave x l = [(take n l) ++ [x] ++ (drop n l) | n <- [0..(length l) -1]]
```

Now, we can define

```
  permutations :: [a] -> [[a]]
  permutations [] = [[]]
  permutations (x:xs) = [ zs | ys <- permutations xs ; zs <- interleave x ys ]
```

If we did not use list comprehension, we would have something
like

```
  permutations (x:xs) = concat  (map (interleave x) (permutations xs))
```

Notice the need for a concat to remove an extra level of list
brackets. Even with list comprehension, if we move the
interleave to the left hand side, we have to add a concat.

```
  permutations (x:xs) = concat [ interleave x ys | ys <- permutations xs]
```

Partitions of a list

A collection of nonempty lists l1, l2,...,lk is said to be a
partition of the list l if l == l1 ++ l2 ++ ... ++ lk. For
instance [[1],[2,3],[4,5,6]] is a partition of [1..6].

We would like to write a Haskell function "partitions" that takes
as input a list l and returns all the partitions of l.

Notice that each partition is itself written as a list of lists.
Thus, the function partitions will return a list of (list of
lists).

```
  partitions :: [a] -> [[[a]]]
```

Since we are interested only in nonempty partitions, the base
case is for the singleton list.

```
  partitions [x] = [[[x]]]
```

For the inductive case, we observe that if we have a set of
partitions for xs, the partitions of (x:xs) are either those in
which x is added to the first component of some partition of xs
or those in which [x] is added as an additional component to some
partition of xs. Thus

```
  partitions (x:xs) =    [(x:head l):(tail l)  | l <- parts xs]
                      ++ [[x]:l | l <- parts xs]
```

==

Introduction to Programming, Aug-Dec 2008
Lecture 7, Mon 01 Sep 2008

Measuring efficiency

Computation in Haskell is reduction --- that is, one sided
application of rewriting rules (function definitions)

Time taken: count the number of reduction steps

Notion of cost is model dependent.

Want the cost of a function in terms of its input size

What is the input size?

For list functions, e.g., number of elements in list

Typically, we denote by T(n) the time taken by an algorithm for
an input of size n

Notation:

```
  f(n) = O(g(n)) if there is a constant k such that
    f(n) <= k g(n) for all n > 0

  e.g.  an^2 + bn + c = O(n^2) for all choices of a,b,c
             (take k = a+b+c)
```

We will be interested in

-- "asymptotic" complexity : hence express T(n) in terms of
O(f(n)).

-- worst case complexity : what is the worst possible running
time on an input of size n? Note that the worst case
could be very, very rare!

e.g.

Cost of ++:

```
  (++) :: [a] -> [a] -> [a]

  [] ++ y = y
  (x:xs) ++ y = x:(xs++y)
```

```
  Observe that
    [1,2,3] ++ [4,5,6] ->
    1:([2,3] ++ [4,5,6]) ->
    1:(2:([3] ++ [4,5,6])  ->
    1:(2:(3:([] ++ [4,5,6])  ->
    1:(2:(3:([4,5,6])
```

We assume that all representations of a list are equivalent, so
the last line is the same a [1,2,3,4,5,6] or 1:[2,3,4,5,6] or
... or 1:2:3:4:5:6:[]

We note that if we merge lists l1 of length n1 and l2 of length
n2, then we reduce using the second definition of ++ n1 times
and reduce using the first definition 1 time. There is no
dependance on n2. In other words, the relevant input size for
++ is n1, which we call n.

Thus, for ++, T(n) = n + 1 = O(n)

Now, let us compute the cost of reverse:

```
  reverse :: [a] -> [a]

  reverse [] = []
  reverse (x:xs) = (reverse xs) ++ [x]

  We can analyze this directly, like ++, or we can write down the
  following "recurrence":

  T(0) = 1
  T(n) = T(n-1) + n

  This says that to reverse a list of length n, we have to first
  solve the same problem for its tail, a list of length n-1 (this
  yields T(n-1)) and then do O(n) work to retrieve the result for
  the original list (this is because we use ++ with input of size
  n-1, which we know is of cost n).

  We can solve this recurrence by unfolding it:

  T(n) = T(n-1) + n
       = (T(n-2) + n-1) + n
       = (T(n-3) + n-2) + n-1 + n
         ...
       = T(0) + 1 + 2 + ... + n
       = 1 + 1 + 2 + ... + n = O(n^2)



Can we do better for reverse?  We can use a function transfer
that reads two lists and moves elements from the first list to
the front of the second list, one by one.

  transfer :: [a] -> [a] -> [a]
  transfer [] l = l
  transfer (x:xs) l = transfer xs (x:l)

Clearly, the relevant input size for transfer is the length of
l1.   In this case,

  T(0 = 1
  T(n) = T(n-1) + 1 (it requires one reduction
                     to get to the problem of size n-1).

Expanding this, we get T(n) = 1+1+...+1 = O(n).

It should be clear that transfer l1 l2 = (reverse l1)++l2.
Thus, transfer l [] = (reverse l) ++ [] = reverse l, so we can
write:

  reverse :: [a] -> [a]
  reverse l = transfer l []

  For this version of reverse, T(n) = O(n).

Sorting lists
-------------

Consider the problem of sorting a list in ascending order.  A
natural inductive definition is

  isort [] = []
  isort (x:xs) = insert x (isort xs)

    where

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

This sorting algorithm is called insertion sort, which is why we
have used the name isort for the function.

Clearly, for insert, T(n) = O(n).

Then, for isort

```
  T(0) = 1
  T(n) = T(n-1) + O(n),
```

which we know (see reverse, version 1) yields O(n^2)

(Aside: Observe that we can also write isort as foldr insert []).

Can we do better?

Merge sort

Suppose we divide the list into two halves and sort them
separately. Can we combine two sorted lists efficiently into a
single sorted list? Examine the first element of each list and
pick up the smaller one, and continue to combine what remains
inductively.

```
  merge [] ys = ys
  merge xs [] = xs

  merge (x:xs) (y:ys)
    |x <= y    = x:(merge xs (y:ys))
    |otherwise = y:(merge (x:xs) ys)
```

The running time depends on the length of both input lists. If
l1 is of length n1 and l2 is of length n2, in each step we put
out exactly one element into the final list. Thus, in n1+n2
steps we achieve the merge.

Now, we split a list of size n into two lists of n/2, recursively
sort them and merge the sorted sublists as follows:

```
  mergesort [] = []
  mergesort [x] = [x]
  mergesort l = merge (mergesort (front l)) (mergesort (back l))
    where
     front l = take ((length l) `div` 2) l
     back l = drop ((length l) `div` 2) l
     merge [] ys = ys
     merge xs [] = xs
     merge (x:xs) (y:ys)
       | x < y     = x:(merge xs (y:ys))
       | otherwise = y:(merge (x:xs) ys)
```

Note that we need explicit base cases for both the empty list and
the singleton list. If we omit the case mergesort [x], we would
end up with the unending sequence of simplifications below

```
  mergesort [x] = merge (mergesort []) (mergesort [x])
                = merge [] (mergesort [x]))
                = mergesort [x]
                = ...
```

Clearly

```
   T(0) = 1
   T(n) = 2 T(n/2) +     O(n)
          --------   -------------
          recursive  initial split
            sort     and
                     final merge


Solving this, we get T(n) = O(n log n).  It is important to
recognize that the function n log n is much closer to n than to
n^2.


======================================================================
Introduction to Programming, Aug-Dec 2008
Lecture 8, Monday 08 Sep 2008

Measuring efficiency
--------------------

Quicksort
---------

In merge sort, we split the list into two parts directly.  Since
the second part could have had elements smaller than those in the
first part and vice versa, we have to spend some time merging the
two sorted lists.

What if we could locate the median (middle) value in the list?
We could then collect all the elements less than the median in
one half and those bigger than the median in the other half.
If we sort these two halves inductively, we can directly combine
them using ++ rather than a merge.

Unfortunately, finding the median value is not easier than
sorting the list.  However, we can use a variation of this idea
by picking up an arbitrary element of the list and using it to
"split" the list into a lower half and upper half.  This
algorithm is called quicksort.

  quicksort [] = []
  quicksort (x:xs) = (quicksort lower) ++ [splitter] ++ (quicksort upper)
    where
      splitter = x
      lower    = [ y | y <- xs, y <= x ]
      upper    = [ y | y <- xs, y > x ]

Notice that we use the first element of the list as the splitter,
for simplicity.  When computing lower, we include other elements that
have the same value as x, but we make sure that x itself is left
out by writing "y <- xs".  We could, symmetrically, have added
the equal values in upper instead of lower.

The problem with the worst case behaviour of quicksort is that we
cannot guarantee that lower and upper are half the size of the
original list.  In the worst case, the first element is either
the largest or the smallest element and we have to inductively
sort a list of length n-1.  The worst case recurrence becomes

   T(n) = T(n-1) + O(n)

which we know yields T(n) = O(n^2).  The O(n) factor captures the
cost of computing lower and upper and combining the answer using
++.

In practice, however, quicksort works very fast and is often the
algorithm used for implementing built-in sort functions in many
applications.  In fact, one can formallyprove that in the
"average case", quicksort is an O(n log n) algorithm, like
mergesort.   The details of this analysis are beyond the scope of
this course.


Defining complexity for arithmetic functions
--------------------------------------------

Consider the problem of deciding whether a number n is a prime.
A naive algorithm is to check whether any number between 2 and
n-1 divides n.  This requires about O(n) divisions.  One can be
slightly more sophisticated and observe that it is sufficient to
check for divisors from 2 to square root of n.  This would
suggest that checking primality can be done in time polynomial in
n.  Why, then, was it considered a sensational result when
```

Agrawal, Kayal and Saxena announced in 2002 that testing primality can be done in polynomial time?

The answer is that the input size for an arithmetic function is measured in terms of the number of digits of the input, not the number itself. Consider any basic arithmetic operation we learn in school --- for instance multiplication. When we multiply two numbers, the time it takes depends on the length of the two numbers. It does not take 100 times as much effort to multiply a 5 digit number by a 4 digit number as compared to the time it takes to multiply a 5 digit number by a 2 digit number. (Multiplying two numbers in time proportional to the value of the numbers amounts to doing repeated addition. For instance, 345*12 is 345+345+...+345 12 times while 345*1234 is 345+345+...+345 1234 times and hence takes about 100 times the effort! But, clearly this is not how we normally multiply multidigit numbers.)

In base 10, the number of digits to write down n is log_10 n. Since computers typically represent values in base 2, the input size for an arithmetic function is usually assumed to be log_2 n, which, as we have agreed, we always write as just log n.

Notice that our earlier naive procedure for primality now takes time O(2^k), where k = log n is the size of the input. Even the slightly cleverer trick of going only upto square root of n takes time O(2^{k/2}), which is still exponential in k.

The complexity of "Divide and conquer"

Recall our divide and conquer approach to multiplication. We divide each number into two k/2 digit parts, by regarding the leftmost k/2 digits and rightmost k/2 digits as separate blocks. Thus, we have

```
    a_k a_{k-1} ... a_{k/2+1}  a_{k/2} ... a_2 a_1
    <---------A_L----------->  <------A_R-------->

    b_k b_{k-1} ... b_{k/2+1}  b_{k/2} ... b_2 b_1
    <---------B_L----------->  <------B_R-------->
```

The products A_L*B_R, A_R*B_L etc are k/2 bit multiplications. It is easy to verify that A*B can be expressed as

```
   10^k(A_L*B_L) + 10^{k/2}(A_L*B_R + A_R*B_L) + A_R*B_R
```

Multiplying a number by 10^j just requires adding j 0's to the right, so the original problem reduces to performing 4 multiplications of size k/2.

We can thus write

```
  T(k) = 4 T(k/2) + O(k)
  T(1) = 1
```

If we expand this out, we get

```
  T(k) = 4 T(k/2) + O(k)
       = 4 (4 T(k/4) + O(k/2)) + O(k)
       = 4^2 T(k/2^2) + 2O(k) + O(k)
       = 4^2 (4 T(k/2^3) + O(k/4)) + 2O(k) + O(k)
       = 4^3 T(k/2^3) + 4O(k) + 2O(k) + O(k)
       = ...
       = 4^m T(1) + sum_{i = 1 to m} 2^i O(k), where  m = log k
       = k^2 + ....
```

Thus, our divide and conquer solution has the same complexity, O(k^2), as the direct algorithm we learn in school!

```
Notice that the only difference between the recurrence for this
multiplication procedure and the one for mergesort is in the
coefficient of the T(n/2) term:

  Mergesort      :  T(n) = 2T(n/2) + O(n)
  Multiplication :  T(n) = 4T(n/2) + O(n)

Clearly, the problem arises from the fact that we need to
consider 4 multiplications of size T(n/2) and not just 2.

In fact, the following can be shown:

Let T(k) = a T(k/b) + O(k).  Then,

        T(k) = O(k log n),     if (a = b)
        T(k) = O(k^(log_b a)), if (a > b)
        T(k) = O(k),           if (a < b)

We will not prove this here, but notice that this tells us that
we can modify mergesort to merge 3 sorted lists of size n/3 or 4
sorted lists of size n/4 etc and the complexity will remain the
same.

minout
------

To see an example of the last case described above

        T(k) = O(k),           if (a < b)

we look at the function "minout.  Let l be a list of distinct
natural numbers (i.e. integers from the set {0,1,2,...}).  The
function "minout l" returns the smallest natural number not
present in l (i.e., if 0 is not in l, then "minout l" returns 0,
else if 0 is in l but 1 is not in l, them "minout l" returns 1,
etc).  Note that l is NOT assumed to be sorted.  The function
should take time O(n).

Observe that if we sort l in ascending order, we can scan the
sorted list from left to right and look for the first gap.
However, sorting takes time O(n log n).

Observe that the output of minout l ranges from 0 to (length l).
Either l contains all numbers [0..(length l)-1], in which case
minout l is length l, or there is a gap in this sequence, in
which case minout l is in the range 0..(length l) - 1.   Our
strategy will be to narrow down the range for the output of
minout to half the original in each step.

The only values in l that contribute to minout are those between
0 and (length l) - 1.  Values that are less than 0 or greater
than or equal to (length l) are irrelevant.  Suppose, in one pass,
we calculate the following sublists of l:

   firsthalf  = [ x | x <- l,  x >= 0, x < (length l) `div` 2 ]
   secondhalf = [ x | x <- l,  x >= (length l) `div` 2, x < (length l)]

We have two cases to consider:

a. Not all values in the range 0 to ((length l) `div` 2) - 1
   appear in firsthalf.  To check this, we just have to verify
   that (length firsthalf) == (length l) `div` 2.  Then, we know
   that the smallest missing number lies in this range.

b. Otherwise, the missing number lies in the range
   [(length l) `div` 2 .. (length l) - 1].

In this way, we get the following recurrence for T(n):
   T(n) = T(n/2) + O(n)

We need O(n) time to construct firsthalf and secondhalf.  Having
done this, we can focus on one of the two sublists and ignore the
other.  From the result quoted above, this yields T(n) = O(n), as
required.

(Note:  When writing the definition of minout, the case when we
   focus on the second half requires us to shift the values.  For
   instance, if l = [0,1,2,3,5,6], then firsthalf = [0,1,2] and
   secondhalf is [3,5,6].  Since there is no gap in firsthalf, we
   inductively check "minout [3,5,6]".  From the definition of
   minout, the answer to this is 0, which is clearly incorrect
   for the original list!  In fact, we need to reduce
   all the values in secondhalf by 3 and check "minout [0,2,3]".
   Alternatively, we can write an auxiliary function that
   computes minout with respect to a lower bound.  In this case,
   the original function would be "auxminout 0 [0,1,2,3,5,6]"
   while the inductive call would be "auxminout 3 [3,5,6]".)


======================================================================
```

Introduction to Programming, Aug-Dec 2008
Lecture 8, Wed 10 Sep 2008

Outermost reduction and infinite data structures

Outermost reduction permits the definition of infinite data structures. For instance, the list of all integers starting at "n" is given by the function

```
   listfrom n = n: (listfrom (n+1))
```

In the definition of listfrom, the outermost expression is the one involving ":". This is thus evaluated first, resulting in the initial "n" being generated. Haskell then tries to expand "listfrom (n+1)" which, in turn, generates "n+1" and "listfrom (n+2)" and so on. Thus, the output of "listfrom m" is the infinite list [m, m+1,...} which is denoted [m..] in Haskell.

We can use infinite lists, for instance, to define in a natural way the Sieve of Eratosthenes whose output is the (infinite) list of all prime numbers.

```
   sieve (x:xs) = x : (sieve [ y <- xs | mod y x > 0])
   primes = sieve [2..]
```

The function sieve picks up the first number of its input list, constructs a new list from its input by removing all multiples of the first number and then recursively runs sieve on this list.

If we work out the reduction for this we get

```
  primes  ===> sieve [2..]
          ===> 2:(sieve [ y | y <- [3..] , mod y 2 > 0])
          ===> 2:(sieve (3:[y | y <- [4..], mod y 2 > 0])
          ===> 2:(3:(sieve [z | z <-
                         (sieve [y | y <- [4..], mod y 2 > 0]) |
                                mod z 3 > 0])
          ===> 2:(3:(5:(sieve [w | w <-
                            (sieve [z | z <-
                                (sieve [y | y <- [4..], mod y 2 > 0]) |
                                       mod z 3 > 0]) |
                                           mod w 5 > 0])
          ===> ...
```

Why is this useful? It is often conceptually easier to define a function that returns an infinite list and extract a finite prefix to get a concrete value.

For instance, from primes we can derive functions such as

```
   nthprime k = primes !! k
```

that extracts the k'th prime number.

==

Using infinite lists

We look at another example of why infinite lists are a useful conceptual tool. Suppose we have a set of cities with direct flights from some of the cities to some of the other cities. For instance, we could represent this situation as follows:

```
         A ----> B
         |<-     |
         |  \    |
         |   \   v
         |    ---C
         v       |
         D       |
         ^\      |
         | \     |
         |  \    v
         F   --->E
          ------>
```

This kind of picture is called a graph. 'A', 'B' etc are called nodes or vertices and the arrows between them are called edges. More precisely, this is a directed graph because each edge is oriented. In some applications, it helps to use undirected edges, which are not oriented.

We could represent the edges in this graph as a Haskell function, as follows:

```
   edge :: Char -> Char -> Bool
   edge 'A' 'B' = True
   edge 'A' 'D' = True
   edge 'B' 'C' = True
   edge 'C' 'A' = True
   edge 'C' 'E' = True
   edge 'D' 'E' = True
   edge 'F' 'D' = True
   edge 'F' 'E' = True
   edge  _   _  = False
```

Suppose we now wish to compute the pairs of vertices that are connected to each other. Our goal is to construct a function

```
   connected :: Char -> Char -> Bool
```

such that connected x y is True if and only if there is a path from x to y using the given set of edges.

Clearly edge x y implies connected x y. In general, we can define connected x y inductively:

   If connected x y and edge y z then connected x z

Unfortunately, we cannot translate this inductive definition of connected in the usual direct fashion into a Haskell function.

Instead, we use the following idea. We build up the set of paths of inductively. Initially we have paths of length 0 --- there is only one. Given a path of length k, we can extend it to a path of length k+1 by adding an edge. We represent a path as a list of nodes. Thus, we have

```
   type Path = [Char]

   extendpath :: Path -> [Path]
   extendpath p = [p++c | c <- ['A'..'F'], edge (last p) c]
```

Now, we can map extendpath over the list of paths of length k to get the list of paths of length k+1.

```
   extendall :: [Path] -> [Path]
   extendall [] = [[c] | c <- ['A'..'F']]
   extendall l  = concat [extend p | p <- l]
                = [ll | p <- l, ll <- extend p]
```

The base case of extendall constructs paths that consist of a single node, the start node. If we start with the singleton list of empty paths [[]] and repeatedly apply extendall, we get lists with longer and longer paths.

The next observation we have is that to check if x and y are connected, we only need to check for paths without loops from x to y --- that is, we can assume that the path from x to y does not visit an intermediate node z twice. If it did, we can excise the loop from z to z and get a shorter path that serves our purpose. If we have n nodes overall, a loop free path can have at most (n-1) edges. If there are more than (n-1) edges in the path, some node must repeat.

This suggests that to check all pairs of connected nodes, it is sufficient to apply extendall n times to the initial list containing the empty path.

Haskell has a builtin function iterate that works as follows:

```
   iterate :: (a -> a) -> a -> [a]
```

For instance

```
   iterate f x = [x, f x, f^2 x, ...]
```

We can now try

```
   iterate extendall [[]]
```

to generate the list we want. We can then extract the first n elements of this list (all paths of length upto n-1) as

```
   firstn = take n (iterate extendall [[]])
```

Now, for each path in this list, we extract the start and end points.

```
   connectedpairs  = [(head p, last p) | l <- firstn, p <- l]
   where
     firstn = take n (iterate extendall [[]])
```

Finally

```
   connected x y = (elem (x,y) connectedpairs)
```

We have used the builtin function "elem x l" that checks if x belongs to l.

Notice that we have not bothered about the fact that extendall generates paths that loop and do other unproductive things. For instance, the path ['A','B','C','A','B','C'] belongs to the sixth iteration of extendall [[]]. But, it does not matter. All that we want is a guarantee that every pair (x,y) that is connected is enumerated by the nth step.

The relation connected that we computed above is the reflexive and transitive closure of the relation edge. In general, we can use the strategy given above to compute the transitive closure of any binary relation on a set.

Search problems

An important class of problems consists of search problems, where there is no closed form for the solution and one must go through a cycle of expanding out possible solutions and then undoing partial solutions when we reach a dead end.

A classic problem of this sort is that of placing N queens on an N x N chessboard such that no two queens attack each other. Recall that two queens attack each other if they lie on the same row, column or diagonal.

From the problem description, it is immediate that in any solution to the problem, there is exactly one queen on each row (and also on each column). Thus, one strategy for solving the problem is the following:

```
  - Place the first queen on the some square of the first row
  - In each succeeding row, place a queen at the leftmost square
    that is not attacked by any of the earlier queens
```

If we follow this strategy on an 8 x 8 board after placing the first queen at the top left corner, after 7 moves, we arrive at the following configuration.

```
       -------------------------------
      | Q |   |   |   |   |   |   |   |
       -------------------------------
      |   |   | Q |   |   |   |   |   |
       -------------------------------
      |   |   |   |   | Q |   |   |   |
       -------------------------------
      |   |   |   |   |   |   | Q |   |
       -------------------------------
      |   | Q |   |   |   |   |   |   |
       -------------------------------
      |   |   |   | Q |   |   |   |   |
       -------------------------------
      |   |   |   |   |   | Q |   |   |
       -------------------------------
      |   |   |   |   |   |   |   |   |
       -------------------------------
```

Now, we find that there is no valid position on the last row for the 8th queen, so we have to abandon this solution and try another one. This can be done in a systematic way by retrying the next possibility for the 7th queen and once again trying the 8th queen. If all possibilities for the 7th queen fail, we go back and try the next possibility for the 6th queen. In this way, we work backwards and retry the previous move. This strategy is called backtracking.

We can represent an arrangement of queens as a list of integers, where the first integer is the column number of the first queen in the first row, the second integer is the column number of the second queen in the second row, etc. Thus, for instance, the position drawn above corresponds to the list [1,3,5,7,2,4,6].

Given an arrangement of k queens, we can write a function that computes all valid extensions of this arrangement to k+1 queens, analogous to the function we wrote to extend paths of length k to paths of length k+1. We have to ensure that the new element we add is not the same as any earlier entry (so that no two queens are in the same column). We also have to do some elementary arithmetic to calculate that the new position is not on any diagonal that is attacked by any of the previous positions.

As in the paths example, let us give the name "extendall" to the function that computes all valid extensions of a list of arrangements. We can now solve the n queens problem by repeatedly applying the function extendall to the empty arrangement and picking up the values generated after the nth application. The following function computes all possible arrangements of n queens on an n x n board.

```
    queens n = (iterate extend [[]])!!(n+1)
```

And, the following returns just one such arrangement --- the first one that is generated.

```
    queensone n = head ((iterate extend [])!!(n+1))
```

Notice that some of the positions after k iterations may have no valid extensions (like the arrangment of 7 queens above). This does not matter. If, at some stage, all arrangements die out as infeasible, we will get the value [] consisting of no valid arrangements (as opposed to [[]], the list consisting of the empty arrangement) which will just repeat itself indefinitely.

The best way to visualize the search for solutions to the N queens problem is to draw a tree in which the root is the initial set of arrangements and every node has as its children its valid extensions.

```
                                  []
                                   |
              -----------------------------------------
             |     |     |     |     |     |     |     |
            [1]   [2]   [3]   [4]   [5]   [6]   [7]   [8]
             |     |     |     |     |     |     |     |
             |    ...   ...   ...   ...   ...   ...   ...
             |
    -----------------------------
   |     |     |     |     |     |
 [1,3] [1,4] [1,5] [1,6] [1,7] [1,8]
   |     |     |     |     |     |
   |    ...   ...   ...   ...   ...
   |
   -----------------------
  |       |       |       |
[1,3,5] [1,3,6] [1,3,7] [1,3,8]
   |    ...   ...   ...   ...
   |
   --------
  |        |
[1,3,5,7] [1,3,5,8]
  |        |
 ...      ...
```

The arrangements at ninth level of this tree are the solutions that we are looking for. Superficially, it appears that the iterative form in which we described the search for solutions is an inefficient way to find just one solution. It seems that to find the solution, we have to compute all values in this tree one level at a time. However, the lazy rewriting strategy of Haskell will ensure that this is not the case. In fact, Haskell will expand arrangements along the leftmost path in this tree, as shown in the picture above. If this path is unsuccessful, it will go back and explore the closest successor that has not been tried before. Thus, lazy evaluation ensures that this tree is examined in a "depth first" fashion rather than a "breadth first" fashion, which would be very inefficient.

==

Introduction to Programming, Aug-Dec 2008
Lecture 10, Monday 15 Sep 2008

Conditional polymorphism

So far, we have seen two ways of assigning types to functions. One way is to assign a specific type, in terms of some of the built in types. Examples include:

```
   power :: Float -> Int -> Float
   times :: Int -> Int -> Int
```

The other option is to give a "generic" type in terms of type variables. In such a generic type, any real type can be uniformly substituted for a type variable. Examples include:

```
   reverse :: [a] -> [a]
   length  :: [a] -> Int
```

As the type of length shows, we can mix type variables and concrete types in the definition of a function.

Can we use these types to arrive at the type for quicksort? If we use concrete types, we have do define a separate version of quicksort for each type of array:

```
   iquicksort :: [Int] -> [Int]
   fquicksort :: [Float] -> [Float]
   ilistquicksort :: [[Int]] -> [[Int]]
```

This is clearly undesirable because the actual definition of quicksort in all these cases is the same.

The other option is to declare it to be of type [a] -> [a]. Is this reasonable? Can we sort a list of functions of type Int->Int->Int such as [plus, times, max]?

The answer is that we can sort a list of values provided we can compare them with each other. In other words, we need to assign the following type to quicksort:

```
   quicksort :: [a] -> [a], provided we can compare values of
                            type a
```

This information about the additional properties is represented in Haskell by identifying a subset of types (called a type class) with the required properity. In this case, the type class in question is called Ord and contains all types which support comparison. (Aside: From a formal point of view, it is not clear that the collection of all types forms a set, but we use set and subset informally to describe collections of types.)

A subset X of a set Y can also be described in terms of its characteristic function f_X where

```
   f_X(x) = True, if x in X
          = False, otherwise
```

Thus, we can think of Ord as a function that maps types to Bool and write "Ord t" to denote whether or not t belongs to Ord. The type of quicksort now becomes:

```
   quicksort :: (Ord a) => [a] -> [a]
```

This is read as "If a is in Ord, then quicksort is of type [a] -> [a]". Note the different double arrow that follows (Ord a).

An even more basic type class is Eq, the set of all types that support checking for equality. Why is this a nontrivial type class? Once again, it is not clear how to define equality for functions, just as it was not clear how to compare functions. At the very least, we would expect two functions that are equal to agree on all inputs.

```
   If f == g then for all x, (f x) == (g x)
```

Note that this is a fairly weak definition --- if this was the only property defining equality of functions in a computational setting, then all functions that sort lists would be equal, since they produce the same output on a given input. This is clearly not what we intuitively understand --- we do not expect insertion sort, mergesort and quicksort to all be equal to each other. However, this is certainly a minimum requirement for two functions to be deemed equal.

In fact, we need to first check an even more basic property --- for an input x, do both f and g terminate in a finite amount of time with a sensible output? Recall, for example, what happens when we supply a negative argument to this definition of factorial.

```
   factorial 0 = 1
   factorial n = n * factorial (n-1)
```

With this definition, factorial (-1) results in an infinite sequence of rewriting steps and never terminates. Thus, even before checking whether the values produced by f and g agree on all inputs, we would need to check

```
   If f == g then for all x, f terminates on x iff
                             g terminates on x
```

Asking whether "f terminates on x" is called the Halting Problem. Alan Turing showed that this cannot be computed --- in other words, we cannot write a function

```
      halting :: (a->b) -> a -> Bool
```

that takes as input a function f and an input to x to f and returns True if the computation of f halts on x and False if f does not halt on x. Note that halting itself is therefore expected to terminate on all inputs and report True or False in a finite amount of time. This is one of the earliest and most fundamental results in the theory of computable functions, and since we cannot check this most basic property, we surely cannot check if two functions are equal, in general, no matter what definition of equality we choose for functions.

Thus, functions do not belong to Eq. A typical example of a function that depends on Eq is the builtin function elem that checks if a value belongs to a list. Here is an inductive definition of elem.

```
   elem x [] = False
   elem x (y:ys)
      | x == y     = True
      | otherwise  = elem x ys
```

The most general type for elem is

```
   elem :: (Eq a) => a -> [a] -> Bool
```

Observe that a type can belong to Ord only if it belongs to Eq. This is because comparison involves not only the functions < and > but also <=, >= etc which imply that we can check equality. Thus, as subsets of types, Ord is a subset of Eq. Alternatively, we have that Ord a implies Eq a for any type a.

Another typical type class in Haskell is Num, the collection of all types that supports "numeric" operations such as +, - and *. For instance, we had written a function sum that adds up values in a list. For specific types of lists, we have:

```
   sum :: [Int] -> Int
```

etc.

In general, sum will work on any list whose underlying type supports addition. This means that we can assign the following generic type to sum.

```
   sum :: (Num a) => [a] -> a
```

The output of sum is the same type as the underlying type of the list.

When we invoke a function whose type depends on some property of the underlying type, Haskell will first check that this property is satisfied. Otherwise, there will be an error message. For instance, if we write

```
   elem reverse [tail, reverse]
```

Haskell will complain that it cannot infer "Eq reverse" to ensure that reverse is of a type that supports equality checking.

Defining type classes

We saw that Haskell organizes types into subsets called type classes that satisfy additional properties. How are these subsets defined? Haskell uses a very simple idea: a type belongs to a class if it has defined on it a specific collection of functions that the class requires.

For instance, a type belongs to Eq if it has defined on it the functions == and /=, both of which take two elements of the type and return Bool. Classes are defined using the following syntax:

```
  class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool
```

This definition specifies that the class Eq holds for type a provided a has defined on it the functions == and /= of the appropriate type signature. In fact, since /= is derivable from == and vice versa, the definition of Eq even fills in these connections, so that only one of /= or == actually needs to be defined.

```
  class Eq a where
    (==) :: a -> a -> Bool
    (/=) :: a -> a -> Bool

    x /= y = not (x == y)
    x == y = not (x /= y)
```

An important fact to note is that these only serve as default definitions so that if the user defines ==, /= is automatically instantiated and vice versa. However, the definition does not prevent the user from defining both == and /= and, in particular, does not attempt to check that such definitions have the property that == and /= are complementary, as suggested in the default definitions. Thus, the default definitions only provide a way of deriving one function from another, but cannot enforce any internal consistency between the actual functions defined by the users.

Likewise, for a type a to be in the class Ord, it must support the functions ==, /=, <, <=, >= and >. The first two come from Eq, so if a belongs to Ord, it must belong to Eq. Once again, some of the functions can be derived from others, so we might have a definition of Ord that looks as follows:

```
  class (Eq a) => Ord a where
    (<)  :: a -> a -> Bool
    (<=) :: a -> a -> Bool
    (>)  :: a -> a -> Bool
    (>=) :: a -> a -> Bool

    x <= y = (x < y) || (x == y)
    x > y  = not (x <= y)
    x >= y = (x > y) || (x == y)

    ... [ Similar definitions using <=, >, >= as basic operator ]
```

Notice the conditional dependence in the class definition, similar to the way we use conditional dependence in type definitions. We can read this as " if type a belongs to Eq, then it belongs to Ord provided the following functions are defined ..."

We can add a type to a class by giving an instance definition that declares the relevant function. For instance, we can make all functions belong to Eq as follows:

```
  instance Eq (a->b) where
    _ == _ = False
```

This provides a trivial definition of equality for all functions of type a->b, which says that no two functions are equal. The definition makes the function type (a->b) a member, or an instance, of the class Eq.

It is important to recognize that this definition only defines == and /= for functions of the same underlying type. For instance, with this instance definition we can compare reverse and tail, because both these functions are of type [a] -> [a], but not tail and head because these functions have different types [a] -> [a] and [a] -> a, respectively.

In an instance declaration, all type variables have to be distinct. Thus, we cannot, for instance, modify the previous instance definition as

```
  instance Eq (a->a) where
    _ == _ = False
```

to restrict our definition to only those functions whose input types are the same as their output types.

For lists and tuples, equality and ordering are defined based on the underlying type. If the underlying type is in Eq, list equality is defined by demanding that all elements of the lists be equal. The instance definition for lists is given as follows:

```
instance (Eq a) => Eq [a] where
  [] == []          = True
  (x:xs) == (y:ys) = (x == y) && (xs == ys)
```

First of all, observe the condition that Eq a must hold to have Eq [a]. Secondly, note that the definition of == for lists is given in a familiar inductive form, like many other functions we have seen for lists.

A nontrivial example of adding instances

The examples given above are all builtin to Haskell. Let us look at a nontrivial example of adding an instance. (This example is due to Vipul Naik.)

In mathematics, we can lift operations from sets to functions over those sets. For instance, given functions f and g on integers, we can define f+g as the function:

```
  f+g(x) = f(x) + g(x)
```

Similarly, we can define

```
  f*g(x) = f(x) * g(x)
```

Recall that Num is the type class consisting of types that support +,-,*. In fact, types in Num support four additional functions:

```
  abs         :: a -> a   (Absolute value)
  signum      :: a -> Int (Sign function, -1 if neg, 0 if zero,
                           +1 if pos)
  negate      :: a -> a   (Negates the value)
  fromInteger :: Integer -> a
                 (Constructs a value of type a from an Integer)
```

The last function requires a little explanation. This tells how to construct a value of the given type from an Integer. We have not seen the type Integer before --- Integer is like Int except that it has no upper and lower bound. So, Integer can represent integers that are arbitrarily large or small. Coming back to fromInteger --- the intention is that an Integer is the most basic type of Num value and we can use an Integer in any Num expression, which will be automatically converted to the right type by fromInteger. For instance, if a is Float, fromInteger might be the function that converts an Integer to a Float with the same magnitude (e.g., 4 becomes 4.0 etc).

Our goal is to say that if the type b belongs to Num, then all functions of type a->b belong to Num, using the lifting of +,-,* and other operations from the underlying type b to functions a->b as described earlier.

It turns out that Num also implies Eq and Ord, so we need the following.

```
  instance (Num b) => Eq (a->b) where
      _ == _ = False

  instance (Num b) => Ord (a->b) where
      _ < _ = False
```

There is a type Show that describes when values of a type can be printed out on the screen using a function show :: a->String. Num is a subset of Show, so we need the following definition.

```
  instance (Num b) => Show (a->b) where
      show f = "Cannot display functions"
```

Now, we can get to the juicy part, where we define how to lift +, -, *, abs, signum, negate and fromInteger from b to (a->b).

```
  instance (Num b) => Num (a->b) where
      (+) f g x     = (f x) + (g x)
      (-) f g x     = (f x) - (g x)
      (*) f g x     = (f x) * (g x)
      abs f x       = abs (f x)
```

```
       signum f x    = signum (f x)
       negate f x    = negate (f x)
       fromInteger n = always n
                         where
                         always n y = (fromInteger n)
```

These definitions are selfexplanatory except, perhaps the last
one. The last one defines how to convert an Integer to a
function of type a->b. The class definition permits us to define
any function we want here, but our choice is to map an integer n
to a constant function that returns the value n on all inputs.
This would normally be achieved by a function such as

```
   almostalways n
   where
     almostalways n y = n
```

However, note that the type of almostalways is
Integer->a->Integer, so the type of (almostalways n) is
a->Integer. Our goal is to associate with an integer n a
function of type (a->b). Since the type b is known to belong to
Num, it has its own definition of fromInteger::Integer->b. So,
we fix almostalways by invoking fromInteger for type b on n, which
gives us the definition above in which always::Integer->a->b and
hence (always n)::a->b.

[Note: Actually, the type definition always::Integer->a->b yields
an error. This is because the definition of always is local to
fromInteger and in this context, ghci/hugs lose track of the
assumption (Num b), which is required to invoke fromInteger
within always. Thus, we have to write
always::(Num b)=>Integer->a->b to get the type of always correct.]

Note the similarity of the way +, - and * are lifted. All are
instances of a general operation lift

```
   lift::(b->b->b) -> (a->b)->(a->b)->a->b
   lift oper f g x = oper (f x) (g x)
```

that lifts an operator over b to functions a->b. Using lift, we
can define:

```
instance (Num b) => Num (a->b) where
    (+) = lift (+)
    (-) = lift (-)
    (*) = lift (*)
     ... rest as before ...
```

You can check that once these instance declarations have been
added, we can write definitions such as

```
   plustwo :: Int -> Int
   plustwo m = m+2

   timestwo :: Int -> Int
   timestwo m  = m*2
```

and then evaluate

```
   (plustwo + timestwo) 7
```

to get the value

```
   (plustwo 7) + (timestwo 7) = 9 + 14 = 23
```

In fact, the definition also works for functions with two inputs
such as

```
   plus :: Int -> Int -> Int
   plus m n = m+n

   times :: Int -> Int -> Int
   times m n  = m*n

   (plus + times) 7 8 = (plus 7 8) + (times 7 8) = 15 + 56 = 71

----------------------------------------------------------------------
```

```
Introduction to Programming, Aug-Dec 2008
Lecture 11, Wednesday 17 Sep 2008


User defined datatypes
----------------------

A datatype is a collection of values with a collective name.  For
instance, the datatype Int consists of the values
{...,-2,-1,0,1,2,...} while the datatype Bool consists of the
values {False,True}.  Datatypes can have structure, and may be
polymorphic --- for example, tuples.  Datatypes can also be
recursively defined and hence of unbounded size --- for example,
lists.

In Haskell, we can extend the set of built-in types using the
the data statement.

Enumerated datatypes
--------------------

This simplest form of datatype is one consisting of a finite set
of values.  We can define such a type using the "data" statement,
as follows.

  data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat

Having introduced this new type, we can directly use it in
functions such as:

  weekday :: Day -> Bool
  weekday Sun = True
  weekday Sat = True
  weekday _   = False

We can also write a function "nextday".

  nextday :: Day -> Day
  nextday Sun = Mon
  nextday Mon = Tue
  ...
  nextday Fri = Sat
  nextday Sat = Sun

What happens if we ask Haskell to evaluate "nextday Fri"?  The
answer is computed correctly as "Sat" but we get a message

  ERROR - Cannot find "show" function for:
  *** Expression : nextday Fri
  *** Of type    : Day

Similarly, if we ask whether "Tue == Wed", the response is

  ERROR - Cannot infer instance
  *** Instance   : Eq Day
  *** Expression : Tue == Wed

The problem is that we have not associated the new datatype with
any type classes, including the most basic ones such as Eq and
Show.  One way to do this is write our own instance declarations.

  instance Eq Day where
    Sun == Sun = True
    Mon == Mon = True
    ...
    Sat == Sat = True
    _   == _   = False
  instance Show Day where
    show Sun = "Sun"
    show Mon = "Mon"
    ...
    show Sat = "Sat"
```

These are the most natural definitions for Eq and Show --- each value is distinct and equal only to itself and each value is displayed in the same way it is defined. To make things easier, we can include these "default" instance definitions for Eq and Show using the word "deriving" as follows:

```
   data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
       deriving (Eq, Show)
```

In the same way, we can derive an instance definition for Ord --- the default definition would order the values in the sequence that they are presented, namely Sun < Mon < ... < Sat.

Note that the built in datatype Bool can be thought of as defined in this way:

```
  data Bool = False | True
     deriving (Eq, Ord, Show)
```

In fact, even Char and Int (whose range is effectively finite because we use a fixed number of bits to represent an Int) can be thought of as defined in the same way.

Datatypes with parameters

We can go beyond finite enumerated types and describe datatypes with a parameter, as in the following example.

```
   data Shape = Square Float | Circle Float | Rectangle Float Float
      deriving (Eq, Ord, Show)

   area :: Shape -> Float

   area (Square x)      = x*x
   area (Circle r)      = pi*r*r
   area (Rectangle l w) = l*w
   where
       pi = 3.1415927
```

Each variant of Shape has a contructor --- Square, Circle or Float. Each constructor is attached to a group of values, which can vary from constructor to constructor. The values Sun, Mon etc in the type Day are also constructors with zero values attached.

What happens when we derive Eq for Shape? At the level of Shape, this will ensure that (Square x) will be equal to (Square y) provided x == y but (Square x) is never equal to (Circle y), etc. When we derive Ord, we have Square < Circle < Rectangle so (Square x) < (Circle y) for all x and y, (Circle z) < (Circle w) if z < w, etc.

Polymorphic datatypes

We can extend our definition of Shape to permit any numeric type as the parameter. Here is the corresponding definition. Note the conditionality on Num a. Note also that we need to include the type parameter a in the name of the type --- the datatype is "Shape a" not just "Shape".

```
   data Num a => (Shape a) = Square a | Circle a | Rectangle a a
      deriving (Eq, Ord, Show)

   size :: (Shape a) -> a

   size (Square x)      = x
   size (Circle r)      = r
   size (Rectangle l w) = l*w
```

Recursive datatypes

We can have recursive datatypes. Here is an example.

```
   data Mylist = Empty | Listof Int Mylist
```

Here the constructors are Empty and Listof. Empty has zero arguments and is hence a constant, representing the base case of the recursive type. The constructor Listof combines a value of type Int with a nested instance of Mylist. Here is a value of type Mylist corresponding to the list [1,3,2].

```
   Listof 1 (Listof 3 (Listof 2 Empty)
```

In Haskell's builtin definition of lists, Empty is written as [] and Listof is written as an infix constructor ":", so the value above becomes the more familiar

```
   1 : (3 : (2 : []))
```

from which we can eliminate the brackets using the right associativity of ":".

It is a small step to extend Mylist to be polymorphic.

```
   data Mylist a = Empty | Listof a (Mylist a)
```

Now, a term that uses the constructor Listof has a value of type "a" and a nested list of the same type. Note again that the full name of the type is "Mylist a", not just "Mylist".

If we change the definition slightly, we get a version of lists where each new element is appended to the right, rather than the left.

```
   data Mylist a = Empty | Listof (Mylist a) a
```

In this representation, a list such as [1,3,2] would be written as

```
  Listof (Listof (Listof Empty 1) 3) 2
```

For inductively defined types, we can write inductive functions to process them. Just as for builtin lists, we can use pattern matching to decompose a value into its parts. For instance, here is a definition of length corresponding to the last definition of Mylist a.

```
  length :: (Mylist a) -> Int
  length Empty = 0
  length Listof l x = 1 + length l
```

To illustrate the role played by the type variables in the definition of an inductive datatype, let us consider an example of a polymorphic type that uses multiple type variables. Suppose we want to define lists that contain elements of types a and b, such that values of types a and b alternate in the list, beginning with a value of type a. There is no restriction on the last value --- if the list has an odd number of elements, the last value is of type a, otherwise it is of type b.

Such a list will look like [x_1,y_1,x_2,y_2,....,x_m,y_m] where each x_i is of type a and each y_i is of type b. Notice that if we strip of x1, the remaining list is of the form [y_1,x_2,y_2,....,x_m,y_m]. This is again a list in which values of type a and b alternate, except that the first value is of type b. This observation leads us to the following definition.

```
   data Twolist a b = Empty | Listof a (Twolist b a)
```

Notice that within Listof, the inductive call to Twolist inverts the order of the type variables. Thus, after a value of type a, we have a list that has alternate values starting with b. The next unfolding of the inductive definition would again invert the types, so we have a list in which the first value is of type a, and so on.

==

Organizing functions in modules

For small function definitions, it is acceptable to write all definitions in a single file and include all dependent definitions. However, as programs grow in size, it is desirable to break them up into separate units for the following reasons:

1. The functions defined in one unit may be useful in many contexts. For instance, if we define quicksort and save it as a separate unit, we should be able to include it automatically in another set of functions without rewriting the definition of quicksort.

2. Keeping functions in separate units makes it easier to maintain the programs. Finished portions are guaranteed not to be touched while editing definitions still under development, thus avoiding unintended modifications to definitions that are already correct and complete.

3. By separating out functions, the interdependence of functions on each other is more clearly specified. In particular, we can identify exactly what "interface" each function provides to the rest of the world. Provided we do not change this "interface", we can reimplement the actual function without changing the correctness of the overall code. For instance, we might organize a unit containing a function "sort" to sort lists. Initially, we may have implemented "sort" using insertion sort. At a later date, if we replace the insertion sort implementation by a better algorithm, such as quicksort, the rest of the code is not affected.

The mechanism for collecting Haskell functions in a reusable unit is to declare them as a module. For simplicity, Haskell requires that each module should be in a separate file and the name of the module should be the same as that of the file containing it.

Thus, we can make a unit consisting of quicksort and mergesort as follows:

```
  module Sortfunctions where

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

  mergesort :: (Ord a) => [a] -> [a]
  mergesort [] = []
  mergesort [x] = [x]
  mergesort l = merge (mergesort (front l)) (mergesort (back l))
   where
    front l = take ((length l) `div` 2) l
    back l = drop ((length l) `div` 2) l
    merge [] ys = ys
    merge xs [] = xs
    merge (x:xs) (y:ys)
      | x < y     = x:(merge xs (y:ys))
      | otherwise = y:(merge (x:xs) ys)
```

These definitions should be stored in a file called "Sortfunctions.hs", to match the module name. Notice that other than adding an initial line

```
  module Sortfunctions where
```

we have not changed the definitions of quicksort and mergesort in any way.

We can now invoke this module in another file as follows:

```
  import Sortfunctions

  ...
```

After "import Sortfunctions", we can freely use quicksort and mergesort. The file that invokes "import Sortfunctions" need not be a module --- it can be a simple Haskell file that has some additional function definitions, which can freely use mergesort and quicksort. We have seen an example of invoking modules when we used the functions "ord" and "chr" for the Char type, which required importing the module Char.

Sometimes, we may not want to import all functions from a module. For instance, suppose we want to use only quicksort from Sortfunctions and write our own mergesort. We can then say

```
  import Sortfunctions hiding (mergesort)
```

If we did not hide mergesort, we would have to use a different name for the new implementation of mergesort because the same name cannot be given two different definitions.

The builtin functions in Haskell (e.g. take, drop, max, etc) are defined in the Standard Prelude, which is implemented as a module called Prelude.hs. This module is imported implicitly in every Haskell file. However, it is possible to explicitly import Prelude and hide some of the builtin functions in case one wants to rewrite these functions. For instance, if we wanted to write different definitions for take and drop, we could initially write

```
  import Prelude hiding (take,drop)
```

Symmetrically, it may be desirable to restrict what is visible outside a module. Suppose we use an auxiliary function in a module to define the main function. We may not want this auxiliary function to be visible outside. If we want to restrict the list of functions that a module exports, we write the list of exported functions in the module header line, as follows.

```
  module Sortfunctions(quicksort,mergesort) where
```

This line specifies that among all the possible functions that may be defined in the module Sortfunctions, only quicksort and

```
mergesort are actually visible to any file that imports this
module.

======================================================================
```

Introduction to Programming, Aug-Dec 2008
Lecture 12, Monday 29 Sep 2008

==

Abstract datatypes

Haskell provides the built in type list. It is often convenient to have additional collective types. One such type is a stack. A stack is a structure in which we can add elements one at a time and remove elements one at a time such that the element removed first is the one that was most recently added --- a last-in-first-out structure.

The insert operation is usually called push and the remove operation is usually called pop. Thus, we have:

```
  push :: a -> (Stack a) -> (Stack a)
  pop  :: Stack a -> (a,Stack a)
```

Notice that pop requires the stack to be nonempty and returns a pair of values --- the element at the top of the stack and the resulting stack with this value removed.

We also add the following function that checks if the given stack is empty.

```
  isempty :: (Stack a) -> Bool
```

We have yet to define how to represent a stack. Here is one possible definition:

```
  data Stack a = Empty | St a (Stack a)
```

We can now instantiate the functions in terms of this definition.

```
  push :: a -> (Stack a) -> (Stack a)
  push x s = St x s

  pop  :: Stack a -> (a,Stack a)
  pop (St x s) = (x,s)

  isempty :: (Stack a) -> Bool
  isempty Empty = True
  isempty _ = False
```

Note that we could directly use the builtin list type and write

```
  data Stack a = St [a]
```

We need a constructor to associate with the value [a], but otherwise all the functions we use are derived from the structure of lists.

```
  push x (St xs) =  St (x:xs)
  pop  (St (x:xs)) =  (x,St xs)
  isempty (St l) = (l == [])
```

Our aim is to provide a definition for stacks that is independent of the representation. How do we specify a stack independent of its representation? We can describe how the functions are connected to each other. For instance.

```
  pop (push x s) = (x,s)
  isempty (push x s) = False
  ...
```

We will not demonstrate a complete list of such "axioms" that describe a stack but assume that such a list exists. This is called an abstract datatype.

We can use modules to implement abstract datatypes. For instance, we could have a module of the form

```
  module Stackmodule(Stack,push,pop,isempty) where

    data Stack a = St [a]
    push x (St xs) =  St (x:xs)
    pop  (St (x:xs)) =  (x,St xs)
    isempty (St l) = (l == [])
```

This ensures that anyone who imports the module only knows the name of the datatype, Stack, and the three functions associated with this type.

Actually, we need one more function --- without knowing the internal structure of Stack, the only way to build a stack is to use push and pop starting from an empty stack. For this, we need an extra function that provides an empty stack to start with, since we cannot explicitly write out an expression for an empty stack. We thus augment the module with:

```
    emptystack :: Stack a
    emptystack = St []
```

An application of stacks

One application of stacks is in expression evaluation. Consider an arithmetic expression such as 2+3*6-4. Without parentheses, this expression can be evaluated in many ways. Normally, we associate a higher precedence to * and / than to + or - (recall the BODMAS rule from school) so we would read this expression as 2+(3*6)-4 or 2+18-4, which is 16.

If we wrote the same expression converting each infix arithmetic operator to a corresponding Haskell function, we would have the expression ((-) ((+) 2 ((*) 3 6)) 4).

The first form is called infix, to denote that each binary operators appears in between its arguments. The second form is called prefix, since each operator (or its equivalent function) appears before its arguments.

We can now define a symmetric notation called postfix, in which each operator appears after its arguments. For instance, here is the postfix form of the expression above: ((2 (3 6 *) +) 4 -)

Interestingly, postfix expressions can be evaluated unambiguously using a stack without any parenthesis. Here is how it is done:

- Scan the expression from left to rigth
- When you see a number, push it on the stack
- When you see an operator,
  - pop the stack once to get the second argument
  - pop the stack again to get the first argument
  - apply the operator to these arguments and push back the result

In the example above, we have the expression 2 3 6 * + 4 - which is evaluated step by step as follows. We use the list based implementation of the stack from last time.

Symbol	Action	Old stack	New stack
2	push 2	St []	St [2]
3	push 3	St [2]	St [3,2]

```
  6     push 6            St [3,2]         St [6,3,2]
  *     pop arg2,arg1     St [6,3,2]       St [2], arg2 = 6 arg1 = 3
        push arg1*arg2    St [2]           St [18,2]
  +     pop arg2,arg1     St [18,2]        St [], arg2 = 18, arg1 = 2
        push arg1+arg2    St []            St [20]
  4     push 4            St [20]          St [4,20]
  -     pop arg2,arg1     St [4,20]        St [], arg2 = 4, arg1 = 20
        push arg1-arg2    St []            St [16]
```

In fact, early scientific calculators manufactured by Hewlett-Packard expected their inputs to be in postfix to make evaluation easier (though it is debatable whether the average user found this convenient!) Postfix is also sometimes called Polish notation (or even "reverse Polish") because it was used by a school of logicians from Poland to write formulas in formal logic in an unambiguous way without parentheses.

Queues

A queue is a first-in-first-out structure. Like a stack, it has basic operations to add and remove elements, but the element that is removed is the one that was added earliest. Here are the operations that we would like to perform on queues:

```
  addq :: a -> (Queue a) -> (Queue a)
  removeq :: (Queue a) -> (a,Queue a)
  isemptyq :: (Queue a) -> Bool
```

Notice that the signatures are the same as those for the functions push, pop and isempty that we defined for stacks. We can again implement a queue using a list, as follows.

```
  data Queue a = Qu [a]

  addq x (Qu xs) = (Qu xs ++ [x])
  removeq (Qu (x:xs) = (x,Qu xs)
  isempty (Qu l) = (l == [])
```

Here, removeq and isemptyq have essentially the same definition as pop and isempty for stacks. Only addq is different --- the new element is appended at the end of the list rather than at the beginning.

This is an important difference --- adding an element to a queue takes time proportional to the size of the queue. Removing an element takes constant time. In contrast, in a stack, both push and pop take constant time, independent of the size of the stack.

Suppose we push and pop n elements in a stack. This will take O(n) time, regardless of the way the pushes and pops are interleaved. On the other hand, for a queue, if we perform n addq operations and n removeq operations, the time taken depends on how they are ordered. If we alternate addq with removeq, the queue never grows beyond length 1 and the overall complexity is O(n). In the worst case, however, if we first do n addq's and then n removeq's, it takes time O(n^2) to build up the queue, since each addq takes time proportional to the length of the queue.

We could, of course, reverse the representation and add elements to the front of the list and remove them from the rear. Then, addq would be a constant time operation while removeq would take time proportional to the length of the list.

Implementing queues with two lists

Can we do better? Can we find an implementation of a queue in which n addq's and n removeq's take O(n) time, regardless of the order in which these operations appear?

We imagine that we break a queue into two parts and use a separate list to represent the front and the rear. We will remove elements from the front portion, so the first element in the front should be at the head. We will add elements to the end of the rear, so, to avoid traversing the rear portion each time we add a new element, we will maintain the rear portion in reverse, with the end of the queue at the head of the list.

Here is the data declaration:

```
  data Queue a = Nuqu [a] [a]
```

Here is the definition addq:

```
  addq x (Nuqu ys zs)  = Nuqu ys (x:zs)
```

Recall that zs represents the rear of the queue, in reverse, so the last element of the queue is at the head of zs, and x is added before this element.

How about removeq? If the left list is nonempty, we just extract its head. If it is empty, we reverse the entire rear into the front and then extract its head.

```
  removeq (Nuqu (x:xs) ys)  = (x,Nuqu xs ys)
  removeq (Nuqu [] ys)  = removeq (Nuqu (reverse ys) [])
```

Why is this any better? After all, after adding n elements the queue would be Nuqu [] [xn,...,x2,x1], so the first removeq will take O(n) time. Note, however, that the O(n) time taken to extract x1 also transfers [x2,..,xn] to the front of the queue. Thus, after one removeq, we have Nuqu [x2,...,xn] []. The next (n-1) removeq operations take only O(1) time each.

In this way, overall, adding n elements and then removing them takes only O(n) operation. The O(n) cost of extracting the first element can be thought of as amortised, or spread out, over the next n-1 removeq operations.

Introduction to Programming, Aug-Dec 2008
Lecture 13, Monday 06 Oct 2008

Trees in Haskell

A tree is a structure in which each node has multiple successors, called children. There is a special node, called the root, from which the tree begins. The root is usually drawn as the topmost node in the tree. Every node other than the root has a unique parent (the node of which this is a child), and hence a unique path back to the root following parent links.

The following are not trees because some nodes have multiple parents. In particular, the structure on the left also has two "root" nodes, 4 and 8.

```
        4   8                 3
       / \ /                 /|\
      2   5                 2 7 5
     / \ / \               /  |/ \
    1   3   6             1   4   6
```

Here are some examples of trees:

```
        4                     3
       / \                   /|\
      2   5                 2 7 5
     / \   \               /   / \
    1   3   6             1   4   6
```

In the trees we have drawn, a value is stored at each node. As in lists, these values have a uniform type --- Int, in the examples above. A bottom level node with no children is called a leaf node. Non-leaf nodes are called internal nodes. Internal nodes in a tree need not have a uniform number of children. For instance, the node with value 5 in the left tree has only one child while the node with value 2 has two children. The order of the children is important. In the tree on the left, each node has upto two children and the two children are oriented as left and right. Thus, 2 is the left child of the root 4 and 3 is the right childe of 2. Notice that though 5 has only one child, 6, this is a right child, not a left child. The tree on the right has upto three children per node.

We will typically look at binary trees, in which each node has upto two children. Here is how we describe a binary data over an arbitrary type a.

```
  data BTree a = Nil | Node (BTree a) a (BTree a)
```

We could organize the constructor Node in other ways --- for instance

```
  data BTree a = Nil | Node a (BTree a) (BTree a)
```

or

```
  data BTree a = Nil | Node (BTree a) (BTree a) a
```

Our choice of putting "a" between the two instances of (Btree a) is helpful to visualize the structure of the tree.

Size vs height in trees

Lists have a linear structure, so there is only one measure of size for a list, the length of the list. Trees are two dimensional, so we consider two quantities:

```
a) size   : the number of nodes in the tree
b) height : the length of the longest path from a root to a leaf
```

In general, we cannot fix a relationship between the height of a tree and its size. For instance, a tree could be highly skewed, and have its height equal to its size, as follows.

```
                   6
                  /
                 5
                /
               4
              /
             3
            /
           2
          /
         1
```

We define a binary tree to be perfect if, at every node, the size of the left and right subtrees are equal. Here is a perfect binary tree.

```
                  4
                /   \
               2     6
              / \   / \
             1   3 5   7
```

We can assign a level to each node in a binary tree --- the root is at level 0, the children of the root are at level 1, ..., children of nodes at level i are at level i+1, ...

In a perfect binary tree, it is easy to observe that there are 2^i nodes at level i. Thus, if a perfect binary tree has height h, then it has nodes at levels 0,1,...,h-1, and thus the size is 2^0 + 2^1 + ... + 2^{h-1} = 2^h - 1. This shows that in a perfect binary tree, the size is exponential with respect to the height. Conversely, a perfect binary tree with n nodes has height log n.

A more generous notion is that of a balanced search tree --- instead of requiring each node to have equal sized left and right subtrees, we say that a tree is balanced if the size of the left and right subtrees at a node differ by at most one. Here is a balanced, but not perfect, binary tree.

```
                  4
                /   \
               2     6
              /     / \
             1     5   7
```

It is not difficult to show that the exponential gap between height and size holds for balanced trees as well.

Binary search trees

An important use of binary trees is to store values that we may want to look up later. For instance, a binary search tree could be used to store a dictionary of words. A binary search tree satisfies the following property at every node v: all values in the subtree rooted at v that are smaller than the value stored at v lie in the left subtree of v and all values in the subtree rooted at v that are larger than the value stored at v lie in the right subtree of v. To emphasize that the values in the tree can be ordered, we write a elaborate slightly on the Haskell definition of binary trees to describe search trees.

```
  data (Ord a) => STree a = Nil | Node (STree a) a (STree a)
```

Observe that the structure of an STree is identical to that of a normal Tree, but there is a type class dependence, similar to the ones we have seen for polymorphic functions such as mergesort and quicksort.

Here are two examples of search trees over the values [1,2,3,4,5,6].

```
        4                     3
       / \                   / \
      2   5                 2   5
     / \   \               /   / \
    1   3   6             1   4   6
```

Both trees look reasonably well "balanced". This is not always the case. For instance, here is a highly unbalanced search tree over the same set of values.

```
                   6
                  /
                 5
                /
               4
              /
             3
            /
           2
          /
         1
```

To find a value in a binary search tree, we start at the root. At each node, if we have not already found the value we are looking for, we can use the search tree property to decide whether to search in the right subtree or the left subtree. We keep walking down the tree in this fashion till we find the value we seek or we reach a leaf node from where we cannot descend further. Thus, each lookup in a binary search tree traverses, in the worst case, a single path from the root to a leaf node.

How much time does it take to look up a value in a search tree with n nodes? Let us say that a tree is balanced if at each node the size of the left subtree differs from the size of the right subtree by at most 1. Initially, we search for the value in the entire tree, with n nodes. If we do not find the value at the root, we search either the left or the right subtree. Since the tree is balanced, the number of nodes in each of these subtrees is at most n/2. In this way, we successively search trees of size n, n/2, n/4, ... till we reach a leaf node, a subtree of size 1. The length of this sequence is clearly bounded by log n. Another way of stating this is that the height of a balanced search tree with n nodes is log n.

Here is a Haskell definition of the search procedure we just described:

```
  findtree :: (Stree a) -> a -> Bool
  findtree Nil x = False
  findtree (Node tleft y tright) x
    | x == y    = True
    | x < y     = findtree tleft x
    | otherwise = findtree tright x
```

Observe that a search tree does not contain duplicate values.

Exercise: What does inorder traversal of a search tree yield?

Search trees are not static objects. In general, we have to insert new values into search trees and remove stale values from search trees. There are two aspects to keep in mind while defining these operations:

a) We have to ensure that the updated tree remains a search tree

b) We have to preserve the balanced structure of the tree

For the moment we concentrate on the first aspect. We will tackle the problem of maintaining the balanced structure later.

Where should we insert a value into a search tree? From the definition of a search tree, there is only one possibility. Search for the value in the tree. If it already exists, there is nothing to be done. Otherwise, we reach a leaf node. This is the same path that we would have to follow to find the new value after it has been inserted. So, insert the new value as a left or right child of the leaf node where the unsuccessful search terminates.

```
inserttree :: (Stree a) -> a -> (Stree a)
inserttree Nil x = Node Nil x Nil
inserttree (Node tleft y tright) x
  | x == y    = Node tleft y tright
  | x < y     = Node (inserttree tleft x) y tright
  | otherwise = Node tleft y (inserttree tright x)
```

Clearly, the maximum number of steps required to insert a value into a search tree is equal to the length of the longest path in the tree. Thus, if the search tree is balanced and has n nodes, inserttree takes time log n, but will not in general preserve the balanced structure of the tree.

How do we delete a value from a tree?

First, we have to agree on what happens if the value to be deleted does not occur in the tree. One approach is to declare this an error. It is easier, how ever, to interpret "delete x from t" as "delete x from t if the value exists in t", so if we try to delete x and it is not found in t, the tree t is unchanged.

Suppose we want to delete a value x from a tree whose root is y. If x < y, we inductively delete x from the left subtree of y. Similarly, if x > y, we inductively delete x from the right subtree of y. So, the interesting case is when x==y.

```
         y == x
        /   \
       w     z
      / \   / \
    t1 t2 t3 t4
```

If we remove y, we have a "hole" at the root of this tree. It is tempting to move either w (or z) into this place and recursively delete w from the left subtree (or z from the right subtree). However, this would not preserve the structure of the tree --- for instance, if we move w up to the root, values in the tree t2, which are bigger than w, will end up to the left of w.

The correct solution is to move the largest value from the left subtree of y or the smallest value from the right subtree of y in place of y. The smallest value in a search tree can be found easily, by following the leftmost path in the tree. Removing this value from a tree is also a relatively easy operation. Here is a function that removes the minmum value from a nonempty tree, returning both the value and the modified tree, after deletion.

```
  deletemin :: (STree a) -> (a,STree a)
  deletemin (Node Nil y t2) = (y,t2)
  deletemin (Node t1 y t2) = (z, Node t1 y tz)
    where (z,tz) = deletemin t1
```

We can now rewrite deletetree as follows:

```
  deletetree :: (Stree a) -> a -> (Stree a)
  deletetree Nil x = Nil
  deletetree (Node tleft y tright) x
    | x < y   = Node (deletetree tleft x) y tright
    | x > y   = Node tleft y (deletetree tright x)

  -- In all cases below, we must have x == y

  deletetree (Node tleft y Nil) x   = tleft
  deletetree (Node tleft y tright) x = Node tleft z tz
    where (z,tz) = deletemin tright
```

Exercise: Define the function deletemax and change the definition of deletetree in the "interesting case" to move the largest value from the left subtree in place of the node being deleted.

```
Introduction to Programming, Aug-Dec 2006
Lecture 15, Tuesday 17 Oct 2006

Balanced binary trees
---------------------

Having described how to preserve the inductive structure of a
search tree after insert and delete operations, it remains to be
seen how we can maintain the balanced nature of the tree.

A balanced tree is one in which, at each node, the sizes of the
left and right subtrees differ by at most one.  This definition
of balance is difficult to maintain without incurring a heavy
rebalancing cost.  A less stringent definition is to ask that the
heights of the left and right subtrees differ by at most one.
Such trees are called height-balanced trees or AVL trees, after
Adelson-Velskii and Landis, the first people to propose the use
of such balanced trees.

It is easy to see that a size-balanced tree is always
height-balanced.  The converse is not true in general.  For
instance, the following tree is height-balanced but not
size-balanced.  The left subtree of the root has size 3 while the
right subtree has size 1.

         4
        / \
       2   5
      / \
     1   3

To use height-balanced trees in place of size-balanced trees, we
have to ensure that the height of a height-balanced tree is
logarithmic in the number of nodes in a tree.  To establish this,
we turn the problem around and consider the following: for a
fixed height h, what is the size S(h) of the smallest
height-balanced tree with height h?

Starting with h=1, we can compute S(h) and draw the corresponding
tree T(h), ignoring the actual values:

  h     S(h)                T(h)

  1      1                    *

  2      2                    *
                             /
                            *

  3      4                    *
                             / \
                            *   *
                           /
                          *

  ...   ...                  ...

  k     S(k-1)+S(k-2)+1       *
                            /   \
                        T(k-1)  T(k-2)

Thus, the "worst" tree of height k is inductively constructing by
fusing together the worst trees of height k-1 and k-2.  The
recurrence for S(h), given by

  S(1) = 1
  S(2) = 2
  S(k) = S(k-1) + S(k-2) + 1, k > 2
```

is very similar to that for the Fibonacci numbers

```
  F(1) = 1
  F(2) = 1
  F(k) = F(k-1) + F(k-2), k > 2
```

In the case of the Fibonacci numbers, we can show that F(k) is exponentially large with respect to k. Analogously, we can show that S(k) grows exponentially with respect to k. This means that even the "most skewed" height-balanced tree of height k has a size that is exponential in k. In other words, the "most skewed" height-balanced tree with n nodes has height logarithmic in n.

Let us assume that we have written a function rebalance that reestablished the height-balanced property after an insert or delete.

Here is how we would modify our functions inserttree and deletetree.

```
  inserttree :: (Stree a) -> a -> (Stree a)
  inserttree Nil x = Node Nil x Nil
  inserttree (Node tleft y tright) x
    | x == y    = Node tleft y tright
    | x < y     = rebalance (Node (inserttree tleft x) y tright)
    | otherwise = rebalance (Node tleft y (inserttree tright x))


  deletetree :: (Stree a) -> a -> (Stree a)
  deletetree Nil x = Nil
  deletetree (Node tleft y tright) x
    | x < y   = rebalance (Node (deletetree tleft x) y tright)
    | x > y   = rebalance (tleft y (deletetree tright x))

  -- In all cases below, we must have x == y

  deletetree (Node Nil y tright) x   = tright
  deletetree (Node tleft y tright) x = rebalance (Node tz z tright)
    where (z,tz) = deletemax tleft
```

Let us define the slope of a node to be the difference in heights between the left and right subtrees of the node. In a height-balanced tree, the slope of each node is in the range {-1,0,+1}. Inserting or deleting a node can at most change the slope by 1. In the modified functions inserttree and deletetree, we apply rebalancing bottom up, starting from the lowest node where we potentially disrupt the balance. Thus, when we need to rebalance a node, we can assume that its slope is either 2 or -2 and its subtrees are inductively height-balanced.

Suppose a node has slope 2 (the case -2 will be symmetric) with both its subtrees height-balanced. Then, the height of the left subtree is two more than that of the right subtree. We shall consider two cases.

a) The slope at the root of the left subtree is 0 or 1.

   This situation corresponds to the following picture, where t[h] denotes that t has height h

```
                        x
                       / \
                      y   t3[h]
                    /   \
               t1[h+1]  t2[h or h+1]
```

   We can rotate this tree to the right as follows

```
                         y
                       /   \
                t1[h+1]     x
                          /   \
               t2[h or h+1]  t3[h]
```

It is easy to observe that the slope at y is 0 or -1 and the slope at x is either 1 or 0, so the rotated tree is height-balanced (recall that t1, t2 and t3 were inductively assumed to be height-balanced).

We have to verify that the resulting tree remains a search tree. To this, it is sufficient to verify that the inorder traversals of the two trees match. It is easy to verify that the inorder traversal for both trees is given by

```
  (inorder t1) ++ [y] ++ (inorder t2) ++ [x] ++ (inorder t3)
```

b) The slope at the root of the left subtree is -1.

This situation corresponds to the following picture, where t[h] denotes that t has height h

```
                         x
                        / \
                       y   t3[h]
                     /   \
                t1[h]  t2[h+1]
```

where t2 looks like

```
                         z
                       /   \
          t21[h or h-1]     t22[h or h-1]
```

with at least one of t21 or t22 being of height h.

We can now rotate the subtree rooted at y to the left, as follows.

```
                         x
                       /   \
                     z      t3[h]
                   /   \
                 y      t22[h or h-1]
                / \
           t1[h]   t21[h or h-1]
```

First, we verify that in both cases, the inorder traversal of the left subtree of x is given by

```
 (inorder t1) ++ [y] ++ (inorder t21) ++ [z] ++ (inorder t22)
```

so this left rotation preserves the search tree property.

Notice that at this stage, the slope at z may be 2 (because it is possible that t1[h], t21[h] and t22[h-1]).

However, we now apply a right rotation at x to get

```
                 _____________z____________
                /                          \
              y                             x
             / \                           / \
        t1[h]   t21[h or h-1]  t22[h or h-1]   t3[h]
```

At this point, clearly the slope of x is -1 or 0, the slope of y is 1 or 0 and the slope of z is 0. Further, the inorder

```
   traversal of this tree yields

     (inorder t1)++[y]++(inorder t21)++[z]++
                          (inorder t22)++[x]++(inorder t3)

   which is the same as that of the original tree, before the two
   rotations.
```

We can now write the function rebalance:

```
   rebalance :: (Stree a) -> (Stree a)

   rebalance (Node t1 y t2)
       | abs (sy) < 2          = Node t1 y t2
       | sy == 2 && st1 /= -1 = rotateright (Node t1 y t2)
       | sy == 2 && st1 == -1 = rotateright (Node (rotateleft t1) y t2)
       | sy == -2 && st2 /= 1 = rotateleft (Node t1 y t2)
       | sy == -2 && st2 == 1 = rotateleft (Node t1 y (rotateright t2))
      where
        sy  = slope (Node t1 y t2)
        st1 = slope t1
        st2 = slope t2

        rotateright (Node (Node t1 y t2) x t3) = Node t1 y (Node t2 x t3)
        rotateleft  (Node t1 x (Node t2 y t3)) = Node (Node t1 y t2) x t3
```

How do we compute the slope? A naive implementation would be as follows

```
  slope :: (Stree a) -> Int
  slope Nil = 0
  slope (Node t1 x t2) = (height t1) - (height t2)

  height :: (Stree a) -> Int
  height Nil = 0
  height (Node t1 x t2) = 1 + (max (height t1) (height t2))
```

Clearly, computing height requires examining each node in the tree, so computing the slope of a tree is proportional to n, rather than log n.

The solution is to augment the information stored at each node to inductively maintain the height of a node.

```
  data (Ord a) => STree a = Nil | Node Int (STree a) a (STree a)
```

where a node of the form (Node m t1 x t2) records that the value at this node is x and the subtree rooted at this node has height m.

With this additional information, we have

```
  height :: (Stree a) -> Int
  height Nil = 0
  height (Node m t1 x t2) = m
```

Thus, computing the slope at any node takes constant time.

It remains to modify all the functions we have written so far---insert, delete, rebalance---to correctly compute the height value at a node after each update. This is left as an exercise.

Introduction to Programming, Aug-Dec 2008
Lecture 14, Monday 13 Oct 2008

Tree Traversals

One way to describe a tree is to list out all the values stored in the nodes. There are three systematic ways to do this, assuming we read a tree from left to right:

```
a) Preorder:   At each node, first list out the value at the node,
               then inductively list out the left and right
               subtrees in the same manner.


b) Postorder:  At each node, inductively list out the left and
               right subtrees in the same manner, then list out
               the value at the node.

a) Inorder:    At each node, first list out the left subtree,
               then list out the value at the node, and finally
               list out the right subtree.
```

Here are some examples:

```
        4           preorder  : [4,2,1,3,5,6]
       / \
      2   5         postorder : [1,3,2,6,5,4]
     / \   \
    1   3   6       inorder   : [1,2,3,4,5,6]


        3           preorder  : [3,2,1,5,4,6]
       / \
      2   5         postorder : [1,2,4,6,5,3]
     /   / \
    1   4   6       inorder   : [1,2,3,4,5,6]
```

These traversals can be defined as inductive functions in the obvious way:

```
  preorder :: (BTree a) -> [a]
  preorder Nil = []
  preorder (Node t1 a t2) = [a] ++ (preorder t1) ++ (preorder t2)

  inorder :: (BTree a) -> [a]
  inorder Nil = []
  inorder (Node t1 a t2) = (inorder t1) ++ [a] ++ (inorder t2)

  postorder :: (BTree a) -> [a]
  postorder Nil = []
  postorder (Node t1 a t2) = (postorder t1) ++ (postorder t2) ++ [a]
```

Reconstructing binary trees from tree traversals

In general, a single tree traversal does not uniquely define the structure of the tree. For example, as we have seen, for both the following trees, an inorder traversal yields [1,2,3,4,5,6].

```
        4                      3
       / \                    / \
      2   5                  2   5
     / \   \                /   / \
    1   3   6              1   4   6
```

The same ambiguity is present for preorder and postorder traversals. The preorder traversal for the first tree above is [4,2,1,3,5,6]. Here is a different tree with the same preorder traversal.

```
        4
       / \
      2   1
         / \
        3   6
         \
          5
```

Similarly, we can easily construct another tree whose postorder traversal [1,3,2,6,5,4] matches that of the first tree above.

Can we unambiguosly reconstruct a tree with preorder traversal [4,2,1,3,5,6] if we fix the inorder traversal to be [1,2,3,4,5,6]? Here is how we would do it, by example, on the tree above.

```
  Inorder  : [1,2,3,4,5,6]
  Preorder : [4,2,1,3,5,6]
```

From the preorder traversal, we know that 4 is at the root. The rest of the preorder traversal breaks up as two segments, corresponding to the preorder traversals of the left and the right subtrees. From the position of 4 in the inorder traversal, we know that [1,2,3] is the inorder traversal of the left subtree and [5,6] is the inorder traversal of the right subtree. Since the left subtree has three nodes, we can split the tail of the preorder traversal after three values. Thus, we have identified the root node and the subset of nodes in the left and right subtrees and recursively broken up the reconstruction problem as follows:

```
                          4
                        /   \
          Left subtree       Right subtree
      Inorder : [1,2,3]      Inorder : [5,6]
      Preorder: [2,1,3]      Preorder: [5,6]
```

This suggests the following Haskell program:

```
  reconstruct :: [a] -> [a] -> (Btree a)
  -- First argument is inorder traversal, second is preorder traversal

  reconstruct [] [] = Nil
  reconstruct [x] [x] = Node Nil x Nil

  reconstruct (x:xs) (y:ys) = Node (reconstruct leftin leftpre) y
                                   (reconstruct rightin rightpre)
    where
    leftsize = length (takeWhile (/= y) (x:xs))
    leftin   = take leftsize (x:xs)
    rightin  = drop (leftsize+1) (x:xs)
    leftpre  = take leftsize ys
    rightpre = drop leftsize ys
```

In the definition above, "takeWhile p l" is the builtin function that returns the longest prefix of l all of whose elements satisfy the condition p. Observe that our reconstruction procedure implicitly assumes that all values in the tree have distinct values.

Exercise: Write a Haskell function to reconstruct a binary tree from its inorder and postorder traversals.

Is is possible to reconstruct a binary tree uniquely from its preorder and postorder traversals? The following example shows that this cannot be done in general:

```
       1   and   1    both have preorder  : [1,2]
      /           \             postorder : [2,1]
     2             2
```

However, if we impose additional structure on binary trees---for instance, no node can have a right child without having a left child---preorder and postorder traversals together uniquely fix the shape of a tree. Here is how we could do it, by example

```
  Preorder  : [4,2,1,3,5,6]
  Postorder : [1,3,2,6,5,4]
```

4 is clearly the root. From the preorder traversal we know that 2 is the root of the left subtree and from the postorder traversal we know that 5 is the root of the right subtree. This information is sufficient to recursively breakup the problem as follows:

```
                        4
                      /   \
       Preorder : [2,1,3]  Preorder : [5,6]
       Postorder: [1,3,2]  Postorder: [6,5]
```

Exercise: Write a Haskell function to reconstruct a binary tree from its preorder and postorder traversals with the restriction that no node can have a right child without having a left child.

Input/Output

So far, we have invoked Haskell functions interactively, via the Haskell interpreter. In this mode of operation, there is no need for a Haskell program to interact with the "outside world". However, invoking functions interactively has its limitations. It is tedious to type in large inputs manually and read off voluminous output from the screen. Instead, it would be much more convenient to be able to read and write data to files.

There is also a natural need for programs to function offline, without direct interaction from the user. Such programs also need mechanisms to take inputs from the environment and write their output back.

Why is Input/Output an issue in Haskell?

In Haskell, computation is rewriting, using function definitions to simplify expressions. This rewriting is done lazily --- that is, the arguments to a function are evaluated only when needed. A highly desirable goal is confluence --- the order of evaluation of independent subexpressions should not affect the outcome of the computation.

An obvious approach would be to make input and output operations functions. For instance, suppose we have a function "read" that reads an integer from the keyboard. Consider the following expression that reads two integers and computes their difference:

```
  difference = read - read
```

An immediate problem with this expression is confluence: the order of evaluation of the two (independent) occurrences of read changes the value computed.

There is a more subtle problem, arising out of lazy evaluation. Consider a list of the form [7, factorial 8, 3+5]. We can extract its length through the expression:

```
  length [factorial 8, 7, 3+5]
```

The function length only needs to check the number of elements in the list, and not the actual values. Under Haskell's lazy evaluation mechanism, this expression evaluates to 3 without actually computing "factorial 8" or "3+5". On the other hand, computing

```
  head [factorial 8, 7, 3+5]
```

would result in evaluating "factorial 8", but not "3+5".

Consider now, corresponding expressions

```
  length [read, read, read]
```

and

```
  head [read, read, read]
```

Using lazy evaluation, no values are actually read when evaluating the first expression! On the other hand, evaluating the second expression would read one value.

This means that an expression that includes functions that perform input or output is not guaranteed to actually execute the operation.

From these observations, we see that input/output actions need to be done in a specific order. Further, there should be no uncertainty as to whether such an action has been performed.

Actions

To fix this problem, Haskell introduces a new quantity called an action. We can think of the world of a Haskell program as divided into two parts. There is the ideal world of "Values" that contains Ints, Floats, Chars and functions involving such quantities. This is the world that we have been dealing with so far.

Side by side with this world is the "real" world with a keyboard, screen, data files ... Actions are used to transfer information from the real world to the ideal world and back.

For this course, the only actions we deal with are those involving input and output. Recall that a function can be viewed as a black box with an input and an output. The inputs and outputs to a function are not to be confused with the input and output that we are trying to formalize. To avoid confusion, we will refer to the input of a function as its argument and the output of a function as its result. Here is our abstract view of a function.

```
                  -----------------
       Argument  |                 |  Result
      ---------->|                 |--------->
                 |                 |
                  -----------------
```

Both the argument and the result of a function lie in the abstract world of Values. Remember that the argument or result could itself be another function over Values.

Actions, on the other hand, simultaneously interact both with the world of Values and the real world. Here is an abstract picture of an action

```
                  -----------------
       Argument  |                 |  Result
      ---------->|                 |--------->
                 |       /\        |
  Value world    |      /||\       |
.................|.......||........|........................
   Real world    |       ||        |
                  -------||---------
                        \||/
                         \/
```

The vertical arrow that penetrates inside the box denoting the action represents the fact that an action transfers data between the Value world and the Real world.

One might argue that we should be more careful and describe whether the data flows upwards or downwards in an action, or both ways. However, as we have observed above, the data that flows across this boundary is inherently sequential. An action that reads two data items and then writes one is different from an action that writes one data item between two reads. Hence, we cannot separate out the upward and downward data streams into two "channels" because, if we did so, we might lose information about the order in which reads and writes occur. Instead, we should think of the action as providing a single doorway between the Value world and the Real world through which data items pass one at a time, either upwards or downwards.

For instance, an action that reads a character does not need an argument. It interacts with the real world to fetch a character and returns the character that is read as its result. In Haskell, the action that does all this is called getChar and has the follwing type:

```
   getChar :: IO Char
```

The word IO indicates that this action performs input/output. There is no argument, only a result type.

How about the symmetric action that takes a Char value as argument and prints it out. This action has an argument, but no result. Haskell has a type containing no values, denote "()". Using this, we can describe the type of putChar as:

```
   putChar :: Char -> IO ()
```

Here, the argument is a Char and the () indicates that the action returns nothing.

Notice that we did not need to use () to describe the lack of an argument to getChar because it is legal for a function/action to have only a result (in the world of functions, such a quantity would be a constant that always returns a fixed value). However, we cannot write an expression that leaves out the result type, so we need to use the empty type () in this context.

Notice also that a function that reads an argument and does not generate a result is completely useless --- what we do with such an entity? On the other hand, actions can be "one-sided" with respect to the world of Values because they perform something nontrivial with respect to the Real world.

The occurrence of an action changes the state of the Real world. For instance, getChar consumes one character from the keyboard, leaving the input data pointing to the next character that has been typed. Similary, putChar produces a character on the screen. In the literature, this behaviour of updating the Real world while reading and generating values is referred to as a "side effect" of the function.

Composing actions

In any nontrivial Haskell program, we compose simple functions to create more complex ones. When we compose functions, we set up a pipe feeding the result of one function as the argument to another one. It is natural to want to the same with actions.

For instance, suppose we want to combine getChar and putChar to generate a complex action that reads a character from the keyboard and prints it out to the screen.

In function notation, it would suffice to write

```
   putChar(getChar)
```

However, this notation hides the fact that we are composing actions, not functions. For instance, we have to evaluate getChar before putChar, which is not the order that normal Haskell evaluation would choose.

To get around these difficulties, Haskell provides an explicit operator, >>=, to compose actions. The complex action we are trying to define is written

```
  getChar >>= putChar
```

This is to be interpreted as "first do getChar, then feed the result as the argument to putChar". As with functions, we can give this complex action a name. For instance, we can write

```
  echo = (getChar >>= putChar)
```

What is the type of echo? The first part of echo is getChar, which does not require an argument. The last part of echo is putChar, which does not produce a result. The argument that putChar requires is supplied internally by getChar and is not a visible part of echo. Thus, echo has type

```
  echo :: IO ()
```

Seemingly, echo does nothing at all, since it does not require an argument and does not produce a result! What saves the day is the tag IO, which says that though echo has no visible effect in the Value world, it does perform some interaction with the Real world.

What is the type of >>=? It takes two actions and connects the output of one to the input of the other. A generic action is of type (a -> IO b) where a and b stand for normal types in the Value world. We might guess that the type of >>= is

```
  (>>=) :: (a -> IO b) -> (b -> IO c) -> (a -> IO c)
```

However, >>= is restricted to combining actions in which the first action has no argument, so the actual type of >>= is

```
  (>>=) :: IO a -> (a -> IO b) -> IO b
```

Now, suppose we want to repeat echo --- that is, compose echo with itself. We could try

```
  echoTwice = (echo >>= echo)
```

but we have a problem. We observed that echo does not require an argument and does not produce a result, so it is not accurate to talk of the result of the first echo action being fed as the argument to the second echo. What we need is an alternative composition operator that discards the result of the first action. This operator is called >> and is of type

```
  (>>) :: IO a -> IO b -> IO b
```

Thus

```
  echoTwice = (echo >> echo)
```

What if we want to read a character and print it twice? We want a complex action of the form

```
  getChar -----> putChar    -----> putChar
            |              |
             -------------
```

We can achieve this by writing, first

```
  put2char :: Char -> IO ()
  put2char c = (putChar c) >> (putChar c)
```

and then writing

```
  modifiedecho = (getChar >>= put2Char)
```

do: an easier way to generate sequences of actions

We have seen that composing actions generates, in general, a sequence of actions. In this sequence, the results of some actions may be passed as arguments to one or more later actions. Haskell provides a simple notation to describe such sequences, using the special word "do". For instance,

```
  do
    putChar c
    putChar c
```

is a complex action that generates two putChars in a row. Thus, we can rewrite the function put2char as

```
  put2char :: Char -> IO ()
  put2char =
    do
      putChar c
      putChar c
```

Observe that the lines below do are indented in a systematic way.

How do we capture the result of an earlier action to reuse as an argument later? The operator <- binds a variable to a value. Thus, we can write

```
   echo :: IO()
   echo = do
            c <- getChar
            putChar c
```

In the first line, c is bound to result of getChar. This value is then used in the second line as an argument to putChar.

We can now write getput2char directly using do notation as follows.

```
getput2char :: IO ()
getput2char =
  do
    c <- getChar
    putChar c
    putChar c
```

```
Introduction to Programming, Aug-Dec 2008
Lecture 15, Wednesday 15 Oct 2008


return: promoting Values to Actions
-----------------------------------

Suppose we want to write a function that reads a character and
checks if it is '\n', the character corresponding to a newline.
The function we want has the following type

  isnewline :: IO Bool

because it takes no argument, does some IO (reads a character)
and generates a result of type Bool (was the character equal to
'\n'?).

Here is an attempt at writing isnewline.

  isnewline = do
                c <- getChar
                if (c == '\n') then ? else ??

At ? and ?? we have to generate a result of type Bool.
Unfortunately, it is not enough at this point to just generate
True or False, as follows.

  isnewline = do
                c <- getChar
                if (c == '\n') then True else False

This is because we want the final item in the do to be an action
of type IO Bool, not a value of type Bool.  The function return
allows us to promote a simple value to an action.  This leads us
to the following version of isnewline.

  isnewline = do
                c <- getChar
                return (c == '\n')


Composing actions recursively
-----------------------------

Suppose we want to read a line of characters, terminated by '\n',
and return the line as a String.  The function we want is

  getLine = IO String

which clearly requires as sequence of getChar actions.  Here is
an inductive definition of getLine:

  getLine =  read a character c
             if c is '\n'
               the line is empty, return ""
             else
               read the rest of the line as s
               return (c:s)

The actual definition is pretty much the same:

  getLine =  do
               c <- getChar
               if (c == '\n')
                  return ""
               else
                  cs <- getLine
                  return (c:cs)
```

```
Notice the recursive call to getLine within the do.

Exception handling
------------------

One of the complications associated with input/output in any
language is that interaction with the real world often produces
unexpected problems.  For instance, the programe may be asked to
read from a nonexistent file, or the disk might become full when
writing.  A program that interacts over the network might find
that the network connection has temporarily failed.

Such problems are called exceptions and should be clearly
differentiated from computational errors such as dividing by zero
or trying to extract the head of an empty list.  There is no
sensible way to continue execution when such a computational
error arises.  However, an exception such as "file not found" can
be dealt with by asking the user to supply an alternate file
name.

Like many other languages, Haskell provides a mechanism for
exception handling.  The idea is to bundle an external "exception
handler" along with the main function that may "raise" (or
"throw") an exception.  If no exception arises, the main function
executes normally and produces a result.  If an exception arises,
the exception is passed to the handler function, which takes
suitable action.  The main point to be kept in mind is that the
exception handler is expected, in the best case, to restore the
computation to its normal course.  Thus, the type of the result
produced by the exception handler should be identical to that of
the original function so that, when composed with other
functions, the function+exception handler combination looks the
same as the function alone.

Schematically, we have the following picture in mind:

                  -------------------
       Argument  |    --------       |  Result
      -------------->|function|----------------->
                 |    --------    |  |
                 |       |        |  |
                 |   exception    |  |
                 |       |        |  |
                 |       v        |  |
                 |   ---------    |  |
                 |  |exception|   |  |
                 |  | handler |---   |
                 |   ---------       |
                  -------------------


The argument is passed to the function.  If the function
terminates normally, it produces a result.  If it terminates
abnormally, information about the exception is passed to the
handler.  If the handler can recover from the exception, it
produces a result whose type is compatible with the original
function.

Haskell provides a function "catch" to combine a function with an
exception handler into a single entity.  Before going into the
details of catch, let us look at a concrete example.

In the function getLine that we wrote above, we assume that each
line of text is terminated by a '\n'.  It is possible, however,
that the last line of text terminates with an end of file marker,
without an explicit '\n'.  Thus, while reading till the end of
the current line, getLine could encounter an exceptional
situation where end of file is reached.
```

How should we recover from this? If we assume that the program can recognize an end of file situation, a good strategy is to treat this as a pseudo "end of line" and return the characters read at the end as the last line of text.

Since the actual reading of input is done by getChar, the end of file error will arise there. In getLine, we check whether the character returned by getChar is '\n' is newline. What we can do is to bundle getChar with an excepttion handler that generates a '\n' when an end of file error occurs. Since, getLine uses getChar as a blackbox to read the next character, it has no way of knowing whether the '\n' that getChar returns was genuinely supplied in the input or was inserted by the exception handler in response to an end of file error.

Thus, our aim is to provide an exception handler for getChar. This handler takes an exception as its argument and generates a Char as its result. Haskell has a type IOError for exceptions that arise from input/output operations. Since getChar is of type IO Char, the exception handler we seek can be written as

```
   eofhandler :: IOError -> IO Char
```

so that its result type is compatible with that of getChar.

What eofhandler has to do is to check if the error it receives is indeed an end of file error --- any error that getChar encounters would be passed onto eofhandler, but only an end of file error can be dealt with in the way we have described.

The function

```
  isEOFError :: IOError -> Bool
```

can be used to test whether the argument passed to eofhandler is an end of file error.

Note that we do not actually check the value of IOError explicitly but rely on an abstract predicate to let us know what type of error we have got. An alternative would be to make IOError a datatype that enumerated all possible IO errors. The main reason for not doing this is that some implementations may support different kinds of such errors, and datatypes in Haskell are not extensible. Although the predicate approach is more cumbersome to use, it is easy to add support for a new IO error by simply including an additional predicate.

Here then is a definition for eofhandler.

```
   eofhandler :: IOError -> IO Char
   eofhandler e
     | isEOFError e = return '\n'
```

Note the "return" to make sure that the result type is IO Char and not just Char. Note also that eofhandler will itself generate an error if any other IOError is passed to it, since it can only respond to one case, where the argument is an EOFError. We will return to this point later.

First, we return to catch. The function catch simply combines a function and its handler into a single unit. For instance, if we write

```
   getCharEOF = catch getChar eofhandler
```

we get a function

```
   getCharEOF :: IO Char
```

that can be used in place of getChar and that can convert end of
file exceptions into '\n'. Thus, we can seamlessly replace
getChar by getCharEOF in getLine as follows.

```
  getLine =  do
               c <- getCharEOF
               if (c == '\n')
                  return ""
               else
                  cs <- getLine
                  return (c:cs)
```

The type of catch is

```
  catch :: IO a -> (IOError -> IO a) -> IO a
```

In other words, catch combines a function and an exception
handler to produce a new function with the same type as the
original one.

Let us get back to the question of what happens when getCharEOF
receives an IOError other than an EOFError. As things stand,
eofhandler will crash, saying "pattern match error". Instead, we
can explicitly pass the error back up one level to the function
that called getCharEOF using ioError, as follows.

```
   eofhandler :: IOError -> IO Char
   eofhandler e
      | isEOFError e = return '\n'
      | otherwise    = ioError e
```

The error passed on by ioError can be caught by getLine. For
example, we could write

```
   getLinehandler :: IOError -> IO String
   getLinehandler e = return ("Error: "++(show e))
```

and

```
   getLinenew = catch getLine getLinehandler
```

In this way, each exception or error can be caught and passed on
to a handler associated with the function where it occurs. The
handler can either explicitly deal with the error, pass it on or
ignore it. If the handler ignores the error, the program
terminates. If the handler passes it on, the error propogates
one level up. Ultimately, the error will reach the top level.
We can assume that there is an implicit error hander at the top
level that prints out information about the error and terminates
the program. This is the error message we see when a program
fails.

Reading and writing files

In principle, reading from and writing to a file is no different
from keyboard input and screen output. We just need to supply an
extra parameter, namely the file name. In Haskell, a file name
(or a fully qualified path) is just a String, but for the sake of
abstractness, Haskell uses the type definition

```
  type FilePath = String
```

We could then conceive of a function getCharFile that reads a
character from a file with type

```
  getCharFile :: FilePath -> IO Char
```

In other words, getCharFile takes the file name as its argument and returns the next character from that file.

In practice, however, invoking a file explicitly when we read or write is very inefficient. If we supply the file name with getChar and putChar, each time we read or write a character, we have to open and close the file. Operating systems incur some overhead with opening and closing files so it is better not to repeatedly do this.

Instead, we set up a connection to a file through a device called a "handle". When we want to use a file, we first open it and associate with it a handle. We then perform our input/output with respect to the handle. Finally, when we are done, we close the file and discard the handle.

To begin with, we need to import the IO module to use the file functions.

```
  import IO
```

Opening a file requires the file name and the mode of opening the file. The different modes in which a file can be opened are given by an enumerated type, whose values are self-explanatory.

```
  data IOMode  =  ReadMode | WriteMode | AppendMode | ReadWriteMode
```

Here, then, is the function to open a file.

```
  openFile :: FilePath -> IOMode -> IO Handle
```

When we open a file, we get back a Handle. We can then use the functions hGetChar and hPutChar to read and write characters via the given Handle (the h at the beginning of the function name indicates that the function works with respect to handles). Here are the types of hGetChar and hPutChar.

```
  hGetChar :: Handle -> IO Char
  hPutChar :: Handle -> Char -> IO()
```

There are also functions

```
  hGetLine :: Handle -> IO String, and
  hPutStr  :: Handle -> String -> IO()
```

that read a line from a file and write a string to a file, respectively. Note that hPutStr does not automatically insert '\n' to signify the end of a line --- you have to put this in explicitly wherever you want a line to end.

Finally,

```
  hClose   :: Handle -> IO ()
```

closes a file.

There is also a function

```
  hGetContents :: Handle -> IO String
```

that reads in the entire text from a file as a String. This is done lazily, so a function such as

```
  copyfile :: Handle -> Handle
  copyfile fromhandle tohandle =
    do
       s <- hGetContents fromhandle
       hPutStr tohandle s
```

will not necessarily read in the entire file associate with fromhandle. Typically, if its argument s is beyond a certain limit, hPutStr will write out its argument in blocks, generating a fixed amount of text at time. What happens is that hGetContents reads the contents of fromhandle in blocks corresponding to the way hPutStr writes out its values.

Here is now a more elaborate program that reads two file names from the keyboard and copies the contents of the first file into the second one. putStr is a builtin function that prints a String to the screen.

```
   main = do fromhandle <- getAndOpenFile "Copy from: " ReadMode
             tohandle   <- getAndOpenFile "Copy to: " WriteMode
             copyfile fromhandle tohandle
             hClose fromhandle
             hClose tohandle
             putStr "Done."

   getAndOpenFile :: String -> IOMode -> IO Handle

   getAndOpenFile prompt mode =
      do putStr prompt
         name <- getLine
         catch (openFile name mode) openhandler


   openhandler :: (String -> IOMode -> IO Handle) ->
                  (IOError -> String -> IOMode -> IO Handle) ->
                  (String -> IOMode -> IO Handle)
   openhandler e =
     do
        putStr ("Cannot open "++ name ++ "\n")
        getAndOpenFile prompt mode
```

getAndOpenFile reads a file name and opens it with the appropriate mode. If opening the file generates an error, openhandler prints an error message and repeats the process of asking for a filename to open.

==

```
Introduction to Programming, Aug-Dec 2008
Lecture 16, Monday 20 Oct 2008
```

lazy IO

In the last lecture, we discussed an example of "lazy IO" in Haskell.

```
  copyfile :: Handle -> Handle
  copyfile fromhandle tohandle =
    do
       s <- hGetContents fromhandle
       hPutStr tohandle s
```

We said that hGetContents will not necessarily read in the entire file associate with fromhandle. Typically, if its argument s is beyond a certain limit, hPutStr will write out its argument in blocks, generating a fixed amount of text at time. What happens is that hGetContents reads the contents of fromhandle in blocks corresponding to the way hPutStr writes out its values.

Lazy IO has to be handled carefully. Suppose we expand copy file to do the opening and closing of handles as well as the actual reading of the file. Then, the function we have written above expands as:

```
  newcopyfile :: FilePath -> FilePath
  newcopyfile fromfile tofile =
    do
       fromhandle <- openFile fromfile ReadMode
       tohandle <- openFile tofile WriteMode
       s <- hGetContents fromhandle
       hPutStr tohandle s
       hClose fromfile
       hClose tofile
```

This works in the same way as the previous copyfile. Logically speaking, since hGetContents is supposed to read the entire file into s, we can close fromhandle before we write s to tohandle. This gives us the following variant of copyfile.

```
  badcopyfile :: FilePath -> FilePath
  badcopyfile fromfile tofile =
    do
       fromhandle <- openFile fromfile
       tohandle <- openFile tofile
       s <- hGetContents fromhandle
       hClose fromfile
       hPutStr tohandle s
       hClose tofile
```

How does this version behave? Well, hGetContents is lazy and there is no demand made on s, so nothing is read before fromhandle is closed. So, this version does nothing.

The moral of the story is that lazy IO should be handled with care. The results you get may vary unexpectedly with slight perturbations of your code, as we saw here.

Actions are like values

Actions can be thought of as special types of functions. Thus, just as we can use functions in place of simple types --- for instance, we can construct lists of functions and pass a function as an argument or obtain a function as a resutl --- we can use actions like simple types.

```
Here is a list of actions of type [IO ()]

  [ putChar 'c', putChar 'z', echo ]

We can write, for instance, a function that takes a list of
actions and executes them as a sequence:

  dolist :: [IO ()] -> IO ()
  dolist [] = return ()
  dolist (c:cs) = do
                   c
                   dolist cs

Haskell has a builtin function sequence of the following type:

  sequence :: [IO a] -> IO [a]

In other words, sequence combines the results of a list of
actions into a single list.  Here is how sequence is defined:

  sequence [] = return []
  sequence (c:cs) = do
                    r <- c
                    rs <- sequence cs
                    return (r:rs)

Notice the similarity in structure between sequence and getLine.


  getLine =  do
              c <- getChar
             ------------------
            | if (c == '\n')   |
            |    return ""     |
            | else             |
             ------------------
                 cs <- getLine
                 return (c:cs)

This is not surprising, since getLine combines the result of a
sequence of getChar's into a list of Char, or String.  The only
difference is that the list of actions in getLine is terminated
by reading '\n', so there is a condition to be checked before
making a recursive call to itself.

User defined "control" structures
---------------------------------

getLine is an example of a "loop" in which we call an action
recursively.  We can easily control this behaviour a bit more.
Suppose we want to write a version of getLine that reads n lines,
for a input integer n, and returns a list of strings, one per
line read.

   getNlines :: Int -> IO [String]
   getNlines 0 = return []
   getNlines n = do
                   thisline <- getLine
                   morelines <- getNlines (n-1)
                   return (thisline:morelines)

In general, if we want to repeat an action n times, we can write

   doNtimes 1 act = act
   doNtimes n act = do
                     act
                     doNtimes (n-1) act

Using let
```

```
---------

So far, for local values in a function, we have used where.  For
example:

  mergesort l = merge (mergesort left) (mergesort right)
    where
      n = (length l) `div` 2
      left = take n l
      right = drop n l

Dually, we can put the local definitions before the function
using let, as follows:

  let
    n = (length l) `div` 2
    left = take n l
    right = drop n l
    in
       mergesort l = merge (mergesort left) (mergesort right)

Inside a do block, we can use a variation of let, without "in",
to reuse the return value of a function.

  do
    line <- getLine
    let revline = reverse line
    putStr revline

In other words, <- allows us to "remember" the return value of an
action and "let" allows us to "remember" the return value of a
function.

Using the Haskell compiler
--------------------------

One of the standard Haskell compilers is the Glasgow Haskell
Compiler which can be invoked using the command ghc.

When you use an interpreter, you interact directly and can choose
the function you want to evaluate.  A compiled program runs
autonomously, so there has to be an unambiguous way of specifying
where the computation should start.  Like many other languages,
ghc expects computation to start with a function called main of
type IO(), located in a module Main.

One useful way to organize Haskell code is to put the actual code
in a separate module and use main in module Main to just call the
relevant function and print out its result using the builtin
function.

   print :: Show a => a -> IO ()

For instance, suppose all our code is in a module called MyModule
and the function to be invoked in MyModule is mymainfunction.
Then, the module Main would look like the following:

  module Main where
  import MyModule
  main = print (mymainfunction)

How do we actually compile the file?  The command is

  ghc --make Main.hs -o outputfilename

In this command, ghc is the name of the compiler while Main.hs is
the module to compile.  The flag "--make" tells ghc to look up
and compile all modules referred to and required by Main.hs.  The
flag "-o" is used to specify the name of the final executable
```

```
command.  If this is left out, the default is to produce an
executable called a.out.
```

```
Introduction to Programming, Aug-Dec 2008
Lecture 17, Wednesday 22 Oct 2008

The abstract datatype Set
-------------------------

A set is a collection of elements without repetitions and without
any prespecified order.  We associate two collections of
operations with sets.

Dictionary operations:

  empty   :: Set a
  isempty :: Set a -> Bool
  member  :: Set a -> a -> Bool
  insert  :: Set a -> a -> Set a
  delete  :: Set a -> a -> Set a

Set operations:

  union     :: Set a -> Set a -> Set a
  intersect :: Set a -> Set a -> Set a
  setdiff   :: Set a -> Set a -> Set a

Possible implementations:

1. The first possibility is to represent a set as a lists,
   possibly with elements repeated.  We use the definition:

   data Set a = Setof [a]

   With this definition, the dictionary operations can be defined
   as follows.

   empty              = Setof []
   isempty Setof l    = l == []
   member (Setof l) x = elem x l
   insert (Setof l) x = Setof (x:l)
   delete (Setof l) x = Setof (filter (/= x) l)

     or, alternatively,

   delete (Setof l) x = Setof [y | y <- l, y /= x]

   If the size of the set is n, the size of the representation N,
   may be much larger than n, because of repetitions.

   Complexity of dictionary operations:

   empty   : O(1)
   isempty : O(1)
   member  : O(N)
   insert  : O(1)
   delete  : O(N)

   Other set operations:

   union     (Setof xs) (Setof ys) = (Setof xs++ys)
   intersect (Setof xs) (Setof ys) = [y | y <- ys, elem y xs]
   setdiff   (Setof xs) (Setof ys) = [x | x <- xs, not (elem x ys)]

   Let M and N be the size of the representations Setof xs and
   Setof ys, respectively.  Then, union takes time proportional
   to M+N while intersect and setdiff take time proportional to
   MN.

2. To reduce the size of the list representation to be equal to
   the size of the set being represented, we can inductively
   maintain the set as a list without repetitions.  The data
   definition remains the same:

   data Set a = Setof [a]

   Under the assumption that the list has no repetitions, the
   dictionary operations can be defined as follows.

   empty              = Setof []
   isempty Setof l    = l == []
   member (Setof l) x = elem x l
   insert (Setof l) x = Setof (x:(filter (/= x) l))
   delete (Setof l) x = Setof (filter (/= x) l)

   The only change is in the definition of insert, where we have
   ensured that all existing copies of x are removed before we
   put in the new value.

   Now, if the size of the set is n, the complexity of the
   dictionary operations is given by:

   empty   : O(1)
   isempty : O(1)
   member  : O(n)
   insert  : O(n)
   delete  : O(n)

   For the other set operations, intersect and setdiff can be
   retained as in the earlier representation.  When computing the
   union, we have to removed duplicates.  One way is to
   systematically insert all the values from the second set into
   the first set, making use of the fact that insert handles the
   problem of filtering out duplicates.

   union (Setof xs) (Setof [])     = Setof xs
   union (Setof xs) (Setof (y:ys) = union (insert (Setof xs) y)
                                          (Setof ys)

   With this definition, the union of two sets of size m and n
   takes time mn, as do intersect and setdiff.

3. If the values being stored in the set can be compared, we can
   maintain the set as a sorted list without repetitions.

   data (Ord a) => Set a = Setof [a]

   empty              = Setof []
   isempty Setof l    = l == []
   member (Setof l) x = elem x l
   insert (Setof l) x = Setof (insert x l)
   delete (Setof l) x = Setof (filter (/= x) l)

   Here, the insert function on sets calls the insert function on
   sorted lists that we have seen in the definition of insertion
   sort.  We do not gain anything in terms of complexity for
   dictionary operations.  We still have:

   empty   : O(1)
   isempty : O(1)
   member  : O(n)
   insert  : O(n)
   delete  : O(n)

   However, we can now use the fact that we can merge two sorted
   lists in linear time to write more efficient implementations
   of the other set operations.

   union (Setof xs) (Setof ys) = Setof (unionmerge xs ys)
     where
       unionmerge [] ys = ys
```

```
      unionmerge xs [] = xs
      unionmerge (x:xs) (y:ys)
        | x < y     = x:(unionmerge xs (y:ys))
        | y < x     = y:(unionmerge (x:xs) ys)
        | otherwise = x:(unionmerge xs ys)

  intersect (Setof xs) (Setof ys) = Setof (intersectmerge xs ys)
    where
      intersectmerge [] ys = []
      intersectmerge xs [] = []
      intersectmerge (x:xs) (y:ys)
        | x < y     = (intersectmerge xs (y:ys))
        | y < x     = (intersectmerge (x:xs) ys)
        | otherwise = x:(intersectmerge xs ys)


  setdiff (Setof xs) (Setof ys) = Setof (setdiffmerge xs ys)
    where
      setdiffmerge [] ys = []
      setdiffmerge xs [] = xs
      intersectmerge (x:xs) (y:ys)
        | x < y     = x:(intersectmerge xs (y:ys))
        | y < x     = (intersectmerge (x:xs) ys)
        | otherwise = (intersectmerge xs ys)
```

Merging two lists of length m and n takes time m+n, so each of union, intersect and setdiff now takes time m+n.

4. If the elements of the set can be compared with each other, we can move from a sorted list representation to a balanced search tree representation.

```
data (Ord a) => Set a = Setof STree a
```

Recall that to maintain balanced search trees, we store the height of a node along with the value so we have

```
data (Ord a) => STree a = Nil | Node Int (STree a) a (STree a)
```

In this reprentation, it is clear that the dictionary operations member, insert and deleet take time O(log n) for a set of size n, thus beating the other representations by a long way.

How about union, intersection and setdiff?

The naive implementation of union would be to insert each element of the second set in the first, or vice versa, yielding an algorithm with complexity min(n log m, m log n). To implement this, we can first flatten out one of the sets into a list using inorder, preorder or postorder traversal and then systematically run through the elements of this list.

Can we do better? We know that we can perform unionmerge, intersectmerge and setdiffmerge on two sorted lists in linear time. We can obtain a sorted list from a (balanced) search tree using inorder. We can also write a function mkbtree to construct a balanced search tree from a sorted list. If inorder and mkbtree work in linear time, we can implement union, intersect and setdiff in linear time in the balanced search tree representation.

Let us start with analyzing inorder:

```
inorder :: (STree a) -> [a]
inorder Nil = []
inorder (Node m t1 x t2) = (inorder t1)++[x]++(inorder t2)
```

If t is balanced, the left and right subtrees t1 and t2 are half the size of the main tree. Let T(n) denote the time required to generate the inorder traversal of a balanced tree with n nodes. The time taken to combine the two recursive inorder traversals is proportional to the length of (inorder t1) because ++ takes time proportional to the length of its left argument. Thus, we have

```
  T(n) = 2 T(n/2) + O(n)
```

We have seen this recurrence before (e.g., mergesort) and the solution is T(n) = O(n log n), which is larger than the O(n) solution that we seek.

The problem arises because Haskell lists are built up right to left, so it is inefficient to write a function that build a list left to right. An example of this is the naive reverse function that we have seen earlier, which takes quadratic time.

```
  reverse [] = []
  reverse (x:xs) = (reverse xs) ++ [x]
```

As with reverse, the key to making inorder more efficient is to use an auxiliary parameter. Let us define a new function

```
  inorderaux :: (STree a) -> [a] -> [a]
```

such that

```
  inorderaux t l
```

yields

```
  (inorder t)++l
```

We can then recover inorder t as inorderaux t [].

Here is a definition of inorderaux.

```
  inorderaux Nil l = l
  inorderaux (Node t1 x t2) l =
                     inorderaux t1 (x:(inorderaux t2 l))
```

In other words, we compute the inorder traversal of Node t1 x t2 from right to left, first placing inorder t2 to the left of l, then adding x on the left of the resulting list and finally placing inorder t1 at the leftmost position.

The complexity of inorderaux is given by:

```
   T(n) = 2 T(n/2) + O(1)
```

The crucial feature is that we need only constant time to combine the two recursive computations of size n/2. For this recurrence, the solution is T(n) = O(n), which is what we seek.

The next function that we have to achieve in linear time is to mkbtree, that constructs a balanced search tree from a sorted list. We need not insist that the input list be sorted. We shall write mkbtree in such a way that the output of mkbtree is a balanced tree and

```
    inorder (mkbtree l) == l
```

If l is sorted, this ensures that mkbtree generates a search tree.

The naive way to make a balanced tree from a list is to use the centre element of the list as the root and recursively construct balanced left and right subtrees from the first and second halves of the list:

```
    mkbtree :: [a] -> (STree a)

    mkbtree [] = Nil
    mkbtree [x] = Node Nil x Nil
    mkbtree l =  Node (mkbtree left) root (mkbtree right)
      where
        m = (length l) `div` 2
        root == l!!m
        left = take m l
        right = drop (m+1) l
```

The complexity of mkbtree is given by the following recurrence:

```
  T(n) = 2 T(n/2) + O(n)
```

The O(n) factor comes because it takes linear time to compute the midpoint of the input list and break it up into two halves.

```
Introduction to Programming, Aug-Dec 2008
Lecture 18, Wednesday 29 Oct 2008

The abstract datatype Set
-------------------------
```

The naive way to make a balanced tree from a list is to use the centre element of the list as the root and recursively construct balanced left and right subtrees from the first and second halves of the list:

```
    mkbtree :: [a] -> (STree a)

    mkbtree [] = Nil
    mkbtree [x] = Node Nil x Nil
    mkbtree l =  Node (mkbtree left) root (mkbtree right)
      where
        m = (length l) `div` 2
        root == l!!m
        left = take m l
        right = drop (m+1) l
```

The complexity of mkbtree is given by the following recurrence:

```
  T(n) = 2 T(n/2) + O(n)
```

The O(n) factor comes because it takes linear time to compute the midpoint of the input list and break it up into two halves.

To do better, we need a more sophisticated version of the trick we used to improve the efficiency of inorder.   This time, rather than constructing a list as output, our input is a list. For optimum efficiency, we need to process the input from left to right. To achieve this, we write a function

```
  mkbtreeaux :: [a] -> Int -> (STree a, Int)
```

whose behaviour is as follows:

```
  mkbtreeaux l n = (t,lrest)
```

where t is a balanced tree made up from the first n elements of l and lrest is the used part of l (i.e., drop n l)

Here is how we define mkbtreeaux:

```
  mkbtreeaux [] n = (Nil, [])
  mkbtreeaux l  0 = (Nil, l)
  mkbtreeaux l  n = (Node t1 root t2, l2)
    where
    m = n `div` 2
    (t1,(root:rest)) = mkbtreeaux l m
    (t2,l2) = mkbtreeaux rest (n - (m+1))
```

As before, we observe that to construct a balanced tree from the first n elements of l, we need to select the midpoint of the n elements as the root and inductively make balanced left and right subtrees from the left and right halves. However, instead of explicitly finding the midpoint and breaking up the list into two parts, we build up the tree from left to right using the same function.

Pictorially, we have

```
                         l
|-----------------------------------------------------------|
                n                              l2
<------------------------------------>|--------------------|
        m                               rest
<----------------> root |------------------------------------|
                         n-(m+1)
 becomes t1             <------------->
                                              l2
                         becomes t2    |---------------------|
```

```
For this function, the time complexity is given by

   T(n) = 2 T(n/2) + O(1)

which yields T(n) = O(n), as required.

Of course, we define mkbtree using mkbtreeaux as

   mkbtree l = fst (mkbtreeaux l (length l))

Thus, we can now implement union, intersect and setdiff for
the balanced search tree representation of sets in linear time
as a sequence of three operations:

1. inorder => generates  sorted lists from the sets in linear time
2. appropriate merge => combines sorted lists in linear time
3. mkbtree => reconstructs balanced search tree in linear time
```

```
Priority queues
---------------

A priority queue is like a queue, except that elements that enter
the queue have priorities, and hence do not exit the queue in the
order that they entered.  (Think of VIP's waiting for darshan at
Tirupati.)

Each item in a priority queue is thus a pair (p,v) where v is the
actual value and p is the priority.  For simplicity, we denote
priorities using integers.  Let us decide that priority p is
higher than p' if p is bigger than p'.

  [Observe that everything we say henceforth can be done
   equivalently by reversing this condition and saying that p is
   higher priority than p' if p is smaller than p'.]

We then need to implement the following operations in a priority
queue:

  insert :: (PriorityQueue a) -> a -> (PriorityQueue a)
  delmax :: (PriorityQueue a) -> (a,(PriorityQueue a))

The first operation inserts an element into the queue while the
second removes an element with the highest priority value.

As with sets, we can quickly run through various implementations
of priority queues and analyze the complexity of implementing the
basic operations.

1. Unsorted lists:

   If we maintain a priority queue as an unsorted list of pairs
   (p,v), insert takes time O(1) while delmax takes time O(n) for
   a queue with n elements.

2. Sorted lists:

   If we sort the list in 1 in descending order of priority
   values, insert takes O(n) time while delmax takes time O(1)
   because the maximum priority value is always at the head of
   the list.
```

3. Balanced search trees:

   Here, we take time O(log n) to insert a value. The maximum value in a search tree is found by following the rightmost path to the leaf. Since all paths are of length O(log n), finding the largest value takes time O(log n). We can then delete it in time O(log n).

   One difficulty with the balanced search tree approach is that, so far, we have always assumed that we maintain search trees with at most one copy of any value. In a priority queue, many items may share the same priority value, so we have to modify our definitions of search trees accordingly.

Note: From now on, we shall ignore the fact that elements in a priority queue are pairs (p,v) of priorities and values and think of them as single entities. Effectively, we are only going to store and manipulate the priorities.

Heaps (or Heap Trees)

Constructing a balanced binary search tree from a priority queue is essentially equivalent to sorting the elements in the queue, because we can use our efficient inorder traversal to generate a sorted list from a binary search tree.

Intuitively, it should not be necessary to order *all* elements in the queue to extract the one with minimum priority value. A weaker ordering should suffice. This is achieved by heap trees.

Heap trees are binary trees, just like search trees, except that the inductive property relating values at various nodes is different. Here is the data definition (in which the components of HNode are arranged differently from those in a binary search tree to emphasize that this tree has a different organization).

```
  (Eq a) => data HTree a = HNil | HNode a (HTree a) (HTree a)
```

The "heap property" is the following:

  The value at a node is larger than the values at its two children.

A heap tree is one in which every node satisfies the heap property. A simple inductive argument establishes that the largest value in a heap tree is found at the root. However, we cannot say anything about the relative order of the values in the two subtrees. All of the following are valid heap trees.

```
        6             6               6
       / \           / \             / \
      5   2         4   5           3   5
     / \   \       / \   \         / \   \
    3   4   1     2   3   1       1   2   4
```

Thus, in a heap tree, finding the largest element is straightforward --- it is always at the root!

  [The heap condition we have defined corresponds to what are called max-heap trees. Dually, we can insist that every node be smaller than its children and obtain a min-heap tree.]

How do we build a heap tree from a list of values? We can begin by using mkbtree, defined earlier, to construct a (size) balanced binary tree from the list in linear time.

Thus, it is sufficient to describe how to convert a (balanced) binary tree into a (balanced) heap tree.

Assume that we have a node x whose left and right subtrees, L rooted at y and R rooted at z are already heap trees.

```
                  --x--
                 /     \
                y       z
                |       |
               / \     / \
              / L \   / R \
              -----   -----
```

What do we need to do to ensure that the combined tree has the heap property? Clearly, if x >= max(y,z), there is no problem. If this does not hold, we exchange x with max(x,y). This brings x either to the root of L or the root of R. The heap property now holds at the top of this tree and also continues to hold in the subtree that was not affected by the exchange. In the subtree where x has moved, we have to inductively repeat the same step. In this process, a "light" value x will come down as far as it needs to before settling in a stable position. This is often called sifting. (A sieve that is used to clean flour lets smaller particles of flour through and retains stones and other large particles above.)

Here is the function that we just described.

```
   sift :: (HTree a) -> (HTree a)
   sift HNil = HNil
   sift (HNode x HNil HNil) = (HNode x HNil HNil)
   sift (HNode x (HNode y t1 t2) HNil)
        | x >= y    = HNode x (HNode y t1 t2) HNil
        | otherwise = HNode y (sift (HNode x t1 t2)) HNil
   sift (HNode x HNil (HNode z t3 t4))
        | x >= z    = HNode x HNil (HNode z t3 t4)
        | otherwise = HNode z HNil (sift (HNode x t3 t4))
   sift (HNode x (HNode y t1 t2) (HNode z t3 t4))
        | x >= max(y,z) = HNode x (HNode y t1 t2) (HNode z t3 t4)
        | y >= max(x,z) = HNode y (sift (HNode x t1 t2)) (HNode z t3 t4)
        | z >= max(x,y) = HNode z (HNode y t1 t2) (sift (HNode x t3 t4))
```

How long does it take to sift a tree? Observer that the value at the root descends one level after each sift. Thus, sift can call itself at most log n times if the original tree is balanced. Moreover, sift does not alter the structure of the tree, only the positions of the values, so the balance is retained.

We can now systematically heapify a balanced tree bottom up.

```
   heapify :: (BTree a) -> (HTree a)

   heapify Nil = HNil

   heapify (Node t1 x t2) = sift (HNode x (heapify t1) (heapify t2))
```

How much time does heapify take? In order to heapify a tree, we have to first heapify its left and right subtrees and then sift the value at the top to its correct position. Assuming the original tree is balanced, we have the recurrence

```
  T(N) = 2 T(N/2) + O(log N)
         --------   --------
        2*heapify     sift
```

If we work this out, which we won't [see Bird's book "Introduction to Programming in Haskell" for the calculation], it turns out that T(N) = O(N).

```
Thus, we have the following O(N) construction of a heap tree from a
list of values:

         mkbtree                     heapify
  list ----------->  balanced tree -----------> heap tree
          O(N)                        O(N)
```

```
Introduction to Programming, Aug-Dec 2008
Lecture 19, Monday 03 Nov 2008
```

Listing out a heap tree in sorted order

For balanced search trees, inorder traversal produces a sorted list of values (in linear time). We can also list out the values of a heap tree in sorted order. We know that the largest value is at the root, so we put out the root value first. Now, both the left and right subtrees are heap trees. If we list them out using the same process, they will independently yield sorted lists. We can then merge these lists to get a single sorted list.

```
   horder :: (HTree a) -> [a]
   horder HNil = []
   horder (HTree x h1 h2) = x:(merge (horder h1) (horder h2))
      where
        merge :: (Ord a) => [a] -> [a] -> [a]
        merge l1 [] = l1
        merge [] l2 = l2
        merge (x:xs) (y:ys)
          | x <= y    = x:(merge xs (y:ys))
          | otherwise = y:(merge (x:xs) ys)
```

Note that the output of horder is in descending order. Applying reverse to the output will produce a list in ascending order in linear time.

What is the complexity of horder? Assuming the heap tree is size balanced, we have to inductively horder both subtrees of size N/2 and merge them, so we have:

```
  T(N) = 2T(N/2) + O(N)
```

or

```
  T(N) = O(N log N)
```

Can we do better? Observe that we can now sort an arbitrary list by constructing a heap tree and listing it out using horder. Sorting takes at least O(N log N) time. We can construct a heap in time O(N). Thus, if we could improve the complexity of horder below O(N log N), we would have a sorting algorithm that is below O(N log N)!

To complete this discussion, we formally define heapsort:

```
  heapsort l = horder (heapify (mkbtree l))
```

or, using the builtin operator "." for function composition:

```
  heapsort = horder . heapify . mkbtree
```

Leftist heaps

We have seen how to construct a size balanced heap (we shall henceforth write just "heap" for "heap tree") from a list in linear time. Recall that to implement a priority queue using heap, we need to implement the following operations:

```
  insert :: (PriorityQueue a) -> a -> (PriorityQueue a)
  delmax :: (PriorityQueue a) -> (a,(PriorityQueue a))
```

Thus, the one-shot heap construction procedure we have described is not enough. We need an efficient way to update heaps incrementally.

We shall describe a technique to combine two heaps of size M and N in time O(log(M + N)). This will solve both the problems above.

1. To insert a value x in heap h, we construct a trivial heap of one element containing x and use the union algorithm to combine this with h.

2. To delete the maximum value in a heap, we remove the root and then combine the left and right subtrees using the union algorithm.

Since union takes time O(log(M+N)), we can implement both the required operations in logarithmic time.

To define our union operation, we need "leftist heaps". A leftist heap is one in which, at every node, the left subtree is has at least as many nodes as the right subtree. Recall that the definition of a heap does not require any specific order between the subtrees. Thus, if we exchange the left and right subtrees of a heap, we still have a heap. We can use this fact to write a procedure to convert an arbitrary heap into a leftist one, bottom up. We first write a function that realigns the left and right subtrees, according to size:

```
realign :: (HTree a) -> (HTree a)

realign HNil = HNil

realign (HNode x h1 h2)
  | (size h1) < (size h2) = HNode x h2 h1
  | otherwise             = HNode x h1 h2
  where
    size :: (HTree a) -> Int
    size HNil = 0
    size (HTree x h1 h2) = 1 + (size h1) + (size h2)
```

Thus, realign just reorders the left and right subtrees if the leftist property is violated at a node. As usual, we can convert size into a constant time function by storing the size of a heap as one of the values under HNode. In other words, we redefine HTree to be

```
(Eq a) => data HTree a = HNil | HNode Int a (HTree a) (HTree a)
```

However, for simplicity, we shall stick to the original definition in the rest of this exposition.

We can now make an entire heap leftist using realign.

```
mkleftist :: (HTree a) -> (HTree a)

mkleftist HNil = HNil
mkleftist (HNode x h1 h2) = realign (HNode x lh1 lh2)
  where
    lh1 = mkleftist h1
    lh2 = mkleftist h2
```

Let us call the rightmost path in a heap the "right spine". The main property of a leftist heap of size n is that the length of the right spine is less than log n. This can be proved easily, by induction on the size of the heap. Let lrs(h) denote the length of the right spine of heap h.

  n = 0 : The heap is empty and the result is trivial

```
  n > 0 : Consider a heap h with root x and left and right
    subtrees h1 and h2 of size p and q, respectively.  Then

     lrs(h) =  1 + lrs(h2)      -- By definition of right spine
            <  1 + (log q)      -- By induction hypothesis on h2
            =  log 2 + log q    -- Arithmetic ...
            =  log 2q           -- Arithmetic ...
            <= log (p + q)      -- h is leftist, so p >= q
            <  log (1 + p + q) -- Arithmetic ...
            =  log (size h)
```

Union of leftist heaps

Let us look at the problem of combining two leftist heaps:

```
Suppose that h =    x     and h' =    y
                   / \               / \
                 h1   h2           h3   h4
```

Clearly, the bigger of x and y should become the root of the combined heap. Suppose x is bigger. We then merge h' with the right subheap h2, using mkleftist to preserve the leftist nature of the heap. Symmetrically, if y is bigger, we inductively merge h with h4. Here is the definition of union.

```
  union :: (HTree a) -> (HTree a) -> (HTree a)

  union h HNil = h
  union HNil h = h
  union (HTree x h1 h2) (HTree y h3 h4)
    | x < y     = realign (HTree y h3 (union (HTree x h1 h2) h4))
    | otherwise = realign (HTree x h1 (union h2 (HTree y h3 h4)))
```

Each step of union makes one move down the right spine of either h or h'. Since we have already seen that lrs(h) < log (size h) for leftist heaps, it follows that each evaluation of union takes at most log(size h) + log(size h') steps. For any integers m and k, log m^k = k log m, so log m^k = O(log m). From this, it follows that

```
  O(log m + log n) = O(log mn) = O(log max(m,n)) = O(log (m+n))
```

Hence, O(log(size h)) + O(log(size h')) = O(log(size h + size h')), which is the bound we want for union.

==

Memoization

Most of the functions we have written have natural inductive definitions. However, naive evaluating an inductive definition is often computationally wasteful because we repeatedly compute the same value.

A typical example of this is the function to compute the nth Fibonacci number.

```
  fib :: Int -> Int

  fib 0 = 1
  fib 1 = 1
  fib n = (fib (n-1)) + (fib (n-2))
```

Let us look at the computations involved in, say, evaluating fib 4.

```
                  fib 4
                /       \
           fib 3          fib 2
          /       \       /     \
      fib 2     fib 1  fib 1   fib 0
     /   \        |      |       |
  fib 1  fib 0    1      1       1
    |      |
    1      1
```

Observe that fib 2 is evaluated twice, independently. Now, suppose we compute fib 6:

```
                  fib 6
                /       \
           fib 5          fib 4
          /      \        /     \
    fib 4      ...       ...   ....
```

Here, fib 6 produces two independent computations of the entire tree we saw above for fib 4, each of which duplicates the computation for fib 2.

In this way, as we compute fib n for larger and larger, more and more evaluations are duplicated. The result is that computing fib n becomes exponential in n. However, we know that we can naively enumerate the nth Fibonacci number in n steps by counting upwards from the first two as follows : 1,1,2,3,5,8,13,21,....

One way to get around the wasteful recomputation we saw above is to remember the values we have computed and not recompute any value that is already known. We maintain a table, or a "memo", where we note down all known values. Before computing a fresh value, we look up this memo. If the value already exists in the memo, we use it. Otherwise, we use the inductive definition to calculate the value and write it into the memo for future reference. This process is called "memoization".

How do we maintain the memo? One simple way is to just maintain an association list of (value,function) pairs. Here is a how a simple memoized version of fib would look:

```
  fib :: Int -> Int
  fib n = memofib n []
```

The function memofib takes an argument for which the function is to be computed, together with the current memo, and returns the function value with an updated memo.

```
  memofib :: Int -> [(Int,Int)] -> (Int,[(Int,Int)])
  memofib n l
    | elem n [x | (x,y) <- l]
        -- value found in memo, return it with the memo unchanged
        = (head [y | (x,y) <- l, x == n],l)

    | n = 0
        -- base case, add it to memo
        = (1,(0,1):l)

    | n = 1
        -- base case, add it to memo
        = (1,(1,1):l)

    | otherwise
        -- compute the new value and update the memo
        = (valc,memoc)

        where
```

```
     -- need to sequentially compute the inductive values so
     -- updates to the memo are preserved

       (vala,memoa) = memofib (n-1) l
       (valb,memob) = memofib (n-2) memoa
       valc = vala + valb
       memoc = (n,valc):memob
```

Note that each value fib n is computed exactly once. Computing a value involves looking up two values in the memo and updating one value into the memo. To count the total number of memo lookups, we note that each value fib n is looked up twice in the memo after it is computed, when computing fib (n+1) and fib (n+2). Thus, the overall complexity is 2n*(memo lookup cost) + n*(memo update cost. In the naive list based memo, the look up cost for the memo is O(n) while the update cost is O(1). Thus, the complexity of this implementation is O(n^2).

We can improve the complexity by making the memo more efficient. It is clear that to compute fib n, we only need to look at values between fib 0 and fib n. We can thus store the memo in an array indexed by (0,n). This would reduce the lookup cost to log n and bring down the overall complexity to O(n log n).

Dynamic programming

Recall that we said we could compute the nth fibonacci number in linear time by enumerating the list of values from fib 0 as 1,1,2,3,5,...

This amounts to filling out the memo table systematically from the bottom up, using the inductive definition of the function to update the memo table. This bottom up approach to computing a memoized function is called "dynamic programming".

In other words, if we have understood the structure of the memo sufficiently, we can "directly" fill it up, without making inductive calls to make each update. In an ideal world, both the top down memoized approach, where updates to the memo table happen whenever the inductive call hits an uncomputed value, and the bottom up dynamic programming approach, which updates the memo directly using the inductive definition, should be equally efficient.

However, in standard implementations, making recursive function calls incurs some overheads where we have to remember the state of values in the calling function etc, so the top down approach is NOT as efficient as the bottom up approach.

However, it is to be emphasized that almost *any* function can be memoized in reasonably uniform way (the list based memo given above should work for any function whose argument satisfies Eq a), but implementing a memoized function using dynamic programming requires understanding the dependency of the memo values on each other and filling them up appropriately.

Modularizing the memo

Ideally, we should make our memoized function independent of the choice of memo. For this, we first define an abstract datatype Table that stores values of functions from a to b and supports the following operations ::

```
  emptytable :: returns an empty memo table
  memofind   :: tells whether a value exists in the memo table
  memolookup :: looks up a value in the memo table
  memoupdate :: adds a new entry to the memo table
```

```
Here is an implementation of this datatype using lists, as we
have done above.  Note the requirement (Eq a) for the domain type
of the function.

  module Memo(Table,emptytable,memofind,memolookup,memoupdate) where

  data (Eq a) => Table a b = T [(a,b)]
    deriving (Eq,Show)

-- emptytable n v creates a table to hold n values with initial
-- "undefined" value v.  This is required, for instance, if we
-- use an array to represent the table.
  emptytable :: (Eq a) => Int -> b -> (Table a b)
  emptytable n v = T []

  memofind ::  (Eq a) => (Table a b) -> a-> Bool
  memofind (T []) _ = False
  memofind (T ((y,m):l)) x
    | x == y    = True
    | otherwise = memofind (T l) x

  memolookup :: (Eq a) => (Table a b) -> a -> b
  memolookup (T ((y,m):l)) x
    | x == y    = m
    | otherwise = memolookup (T l) x

  memoupdate :: (Eq a) => (Table a b) -> (a,b) -> (Table a b)
  memoupdate (T l) (x,n) =  T ((x,n):l)

Now, we can write our memoized version of the fib function a
little more "abstractly", using the Memo module.

  import Memo

  fib :: Int -> Int
  fib n = memofib n emptytable

  memofib :: Int -> [Table Int Int] -> (Int,Table Int Int)
  memofib n t
   | memofind t n = (memolookup t n, t)
   | n = 0        = (1, memoupdate t (0,1))
   | n = 1        = (1, memoupdate t (1,1))
   | otherwise    = (valc,memoc)
        where
          (vala,memoa) = memofib (n-1) l
          (valb,memob) = memofib (n-2) memoa
          valc = vala + valb
          memoc = memoupdate memob (n,valc)

The advantage is that if we decide to modify the memo table
implementation to be more efficient, we need not change anything
in the definition of fib/memofib.

======================================================================
Introduction to Programming, Aug-Dec 2008
Lecture 20, Wednesday 05 Nov 2008

Memoization and Dynamic Programming
-----------------------------------

Last time, we saw how we could make the computation of an
inductive definition with overlapping subcomputations more
efficient by "memoizing" intermediate results.

Today we look at some more examples and also study a related idea
from the theory of algorithms called dynamic programming.

Pinball game
------------

A board has obstacles arranged in a triangle, as follows.


                           ()
                          /  \
                        ()    ()
                       /  \  /  \
                     ()    ()    ()
                    /  \  /  \  /  \
                  ()    ()    ()    ()
                 /  \  /  \  /  \  /  \
               ()    ()    ()    ()    ()


Each obstacle has a number of points associated with it.  For
instance, we could have the following assignment of points to the
obstacles in the board above.


                           15
                          /  \
                        28    33
                       /  \  /  \
                     18    22    16
                    /  \  /  \  /  \
                  35    15    11    17
                 /  \  /  \  /  \  /  \
               29    13    14    26    12


When we drop a ball on the topmost obstacle, it bounces off and
goes down, either left or right.  Depending on which way it goes,
it bounces off an obstacle at the next level, and again gets
deflected left or right.  This continues till it reaches the
bottom row of obstacles.  When the ball hits the final obstacle
on its run, we add up the points of all the obstacles that it has
collided against to obtain our score for the game.

With sufficient skill, it is possible to control the ball as it
moves down the board and select the path along with it travels.
The aim is to choose a path that maximizes the score.

For the moment, let us concentrate on calculating the maximum
score that one can obtain on a given board (note that there may
be more than one way to achieve this maximum score).  Later, we
will come back to the question of determining a path that
actually yields this maximum score.

First, we need a way to identify each obstacle unambiguously.  We
can assign a pair of coordinates (i,j) to each obstacle where i
in {1,2,...,n} is the row in which the obstacle lies and j in
{1,2,...,i} is the position of the obstacle within that row, from
left to right.  Let points(i,j) be the points assigned to the
```

```
obstacle at location (i,j).  If we rearrange the triangle a bit,
we have the following arrangement.

           |   1     2     3     4     5   ---> j
        ---+----------------------------
        1  |  15
           |
        2  |  28    33
           |
        3  |  18    22    16
           |
        4  |  35    15    11    17
           |
        5  |  29    13    14    26    12
        |  |
        v  |
        i  |

We define score(i,j) to be the maximum score that we can achieve
if the ball starts its path at the obstacle labelled (i,j).  The
quantity we want to calculate is given by score(1,1),
corresponding to the obstacle at the top of the triangle.

There is a natural inductive definition of score(i,j).  After
bouncing of (i,j), the ball will next go to either (i+1,j), if it
goes left, or (i+1,j+1), if it goes right.  If we already know
the scores at these two positions, we can computute score(i,j) as
follows:

  score(i,j) = points(i,j) + max [ score(i+1,j), score(i+1,j+1) ]

The base case is when i = n, corresponding to the last row.  For
all j in {1,2,...,n} we have

  score(n,j) = points(n,j)

Computing score(i,j) naively would result in recomputing
intermediate values.  For instance, both score(3,2) and
score(3,3) would require the value of score(4,3).  Through
memoiziation, we can avoid wasteful recomputation, just as we saw
in the fibonacci example.

Dynamic programming
-------------------

Another way to approach the problem of overlapping
subcomputations is to examine the order in which values are
required.  In each inductive definition, we have a base case for
which the value is immediately available.  The inductive
definition specifies a relationship between a value and its
"neighbours".  Once we have calculated the base cases, we can
compute the positions that lie in the neighbourhood of these base
cases.  In this way, we can systematically "grow" the set of
known values, ensuring at each stage that we already have at hand
the values that we need to compute the current value.

For example, in the fibonacci case, the base cases are "fib 0"
and "fib 1".  Having calculated these, we can immediately
calculate "fib 2", since the values it depends on are both known.
With "fib 2" in hand, we can compute "fib 3".  In this way, we
proceed to "fib 4", "fib 5", ...  We can regard this as filling
up the memo table systematically for n = 0,1,2,...  Notice that
this corresponds to the natural way in which we enumerate the
fibonnacci sequence as 1 1 2 3 5 ...

Consider our second problem, the pinball game.  The base cases in
this computation are the values score(n,j), for j in {1,2,...,n}.
With this in hand, we can immediately compute the values
score(n-1,k), k in {1,2,...,n-1} since both neighbours of the
```

obstacle at (n-1,k) lie in the range of values that are already known. Having calculated row n-1, we can proceed back to row n-2, n-3, ... till we reach the top row.

This technique is called dynamic programming. In dynamic programming, we systematically compute the entries of the memo table "bottom up" by analyzing the order in which values would be computed had we used the normal "top down" inductive definition. In a sense, both approaches are equivalent, because we actually compute each value only once. However, in practice, dynamic programming is more efficient because we avoid the overhead associated with keeping many function calls pending while we move down to the base case.

Expression evaluation

Consider infix arithmetic expressions over integers using the operators + and * without parentheses and without any assumptions about the order in which to evaluate subexpressions. Thus, an expression such as 6*3+2*5 may be evaluated as (6*3)+(2*5) = 28 or 6*((3+2)*5) = 150 or ((6*3)+2)*5 = 100, depending on the order of evaluation. Our aim is to find a way of bracketing the expression so as to achieve the maximum overall value. As with the pinball game, for the moment we disregard the problem of finding the actual bracketing and concentrate on calculating the best possible value we can achieve with this expression.

For simplicity, let us assume that the numbers in the expression are all single digits. If we number the positions in the input expression from 1, every odd position is an integer and every even position is an arithmetic operator.

When we bracket an expression, we explicitly fix an order in which subexpressions are evaluated. Starting with the initial expression, we combine values in pairs, using one of the operators, and keep reducing it till we end up with a single number. Let us focus on what happens in the last step.

Consider the expression we saw earlier, 6*3+2*5. The last step could involve any of the three operators. We have no reason to believe, a priori, that one of these three is better than the others, so we have to consider all of them:

```
Case 1:  The first occurrence of * is the last operator applied.
         Then, prior to this, we have optimally evaluated
         subexpressions 6 and 3+2*5.

Case 2:  The + is the last operator applied.
         Then, prior to this, we have optimally evaluated
         subexpressions 6*3 and 2*5.

Case 3:  The second occurrence of * is the last operator applied.
         Then, prior to this, we have optimally evaluated
         subexpressions 6*3+2 and 5.
```

In each case, the best value we can achieve by applying the last operator is obtained by maximizing the values we obtain from the corresponding subexpressions. (This is a consequence of the choice of operators + and *, which grow monotonically in both their arguments. What would happen if we allowed the operator - in our expression?)

The subexpression we have to evaluate is a part of the original expression. The subexpressions generated by subcomputations could overlap. For instance, in Case 2, we have to compute the values of 6*3 and 2*5. The case 6*3 will also occur one step later in Case 3, while 2*5 will occur one step later in Case 1. Hence, it makes sense to memoize these values.

How do we identify subexpressions unambiguously, to ensure that we can recognize when we have (or have not) yet computed the value for a given subexpression? Notice that each subexpression is actually a substring of the original expression. Thus, we can identify it by noting its starting and ending points.

Thus, in general, we want to compute a function maxval(i,j) where i < j are indices in the range 1,2,...,length(e) for a input expression e. Moreover, assuming that all the integers in the expression are single digit numbers, we have already observed that a subexpression will start and end at an odd index. We can write down an inductive defintion for maxval(i,j).

```
  maxval(i,j) =    max        op(k)(maxval(i,k-1),maxval(k+1,j))
               k in [i..j]
                 k even
```

Here, op(k) refers to the function corresponding to the operator at position k --- op(k) is multiplication if the symbol is * and addition if the symbol is +. All the arithmetic operators are at even positions, so we restrict ourselves to even numbers k in the range [i..j]. We pick all such k, inductively compute the subexpressions resulting from splitting the expression at k, and take the maximum.

The base case is when the expression has length 1, so that we have a single value. For odd positions i in the range [1..length(e)]

```
  maxval(i,i) = value(i)
```

where value(i) is the integer value corresponding to the digit at position i.

Using memoization, we can calculate the values we need without recomputing any intermediate values. The final answer we want is maxval(1,length(e)).

How would we systematically calculate thse values using dynamic programming. If we place the values maxval(i,j) in a two dimensional array, we have the following picture:

```
           |  1     3     5     7     9   ---> j
        ---+---------------------------
        1  |  o     x     x     x     x
           |
        3  |  .     o     x     x     x
           |
        5  |  .     .     o     x     x
           |
        7  |  .     .     .     o     x
           |
        9  |  .     .     .     .     o
        |  |
        v  |
        i  |
```

Notice that we only consider odd values for i and j because each subexpression starts and ends at an odd position. Positions marked . correspond to pairs (i,j) where j < i and hence do not denote valid subexpressions. Positions marked "o" correspond to values of the form maxval(i,i) which are the base case. The remaining positions, marked "x" have to be computed using the inductive definition.

According to the inductive definition, to compute maxval(i,j) we need to know maxval(i,k), where i < k <= j and maxval(k,j) where

```
i <= k < j.

Suppose we are trying to compute maxval(1,7).  This requires us
to know maxval(1,5), maxval(1,3) and maxval(1,1) along the row i
= 1 and maxval(3,7), maxval(5,7) and maxval(7,7) along the column
j = 7.

Pictorially, we have the following situation, where y marks the
value we are trying to compute and z marks the values that are
required to compute it.

          |  1     3     5     7     9   ---> j
       ---+----------------------------
        1 |  z     z     z     y     x
          |
        3 |  .     o     x     z     x
          |
        5 |  .     .     o     z     x
          |
        7 |  .     .     .     z     x
          |
        9 |  .     .     .     .     x
        | |
        v |
        i |


Thus, in the array, to compute a value, we need to know all
values to its left and all values directly below it.  Notice that
each value to the left pairs up with a value below it, according
to the inductive definition --- for instance, maxval(1,1) goes
with maxval(3,7) corresponding to splitting the expression at
position 2, maxval(1,3) goes with maxval (5,7) corresponding to
splitting the expression at position 4 and maxvao(5,7) goes with
maxval(7,7) corresponding to splitting the expression at position
6.

As we observed earlier, the base cases are the values on the
diagonal.  Having computed the values on the diagonal, we can
next tackle the values one off the diagonal, of the form
maxval(i,i+2), because, for these positions, we know all the
values in the same row and column.  We can then proceed to the
elements two positions off the diagonal of the from maxval(i,i+4)
and so on.

In the picture below, we describe the order in which values are
calculated.  The diagonal elements can be calculated (in any
order) in the first pass, so they are labelled 1.  The off
diagonal elements can then be calculated (in any order) in the
second pass, so they are labelled 2.  Thus, in 5 passes, we can
compute the value maxval(1,9) that is four steps off the
diagonal.


          | 1     3     5     7     9   ---> j
       ---+----------------------------
        1 | 1     2     3     4     5
          |
        3 | .     1     2     3     4
          |
        5 | .     .     1     2     3
          |
        7 | .     .     .     1     2
          |
        9 | .     .     .     .     1
        | |
        v |
        i |
======================================================================
```

Introduction to Programming, Aug-Nov 2008
Lecture 21, Monday 17 Nov 2008

Arrays

The last abstract datatype we shall consider is the array. We can think of an array as a fixed sequence of "cells", each position labelled by an index, where we can access each cell directly by specifying the index value.

```
   index->   1   2   3   4   5   6   7   8
            -------------------------------
   value-> | a | c | f | i | j | k | a | d |
            -------------------------------
```

The values stored in an array are all of a uniform type. An array is like a list with two important differences:

1. Normally, arrays have a fixed number of elements.

2. The time taken to access the i^th element of an array is independent of i. (Recall that this takes time O(i) in a list).

Arrays are built-in to Haskell, so we shall first look at how to use arrays in Haskell, and then consider how this datatype may be implemented in an efficient manner.

To use arrays in Haskell, you have to first import the module Array as follows:

```
  import Array
```

This makes available the abstract datatype

```
  Array a b
```

Notice that Array has two types associated with it. The first type variable, a, refers to the type of the index variable, while the second type variable, b, refers to the type of the data stored in the array.

The index variable must belong to the type class Ix, which is a subclass of Ord a. Intuitively, Ix consists of types which have a total order but which are also discrete, so that one can enumerate all values that list between a lower bound and upper bound. Thus, Bool, Char and Int would belong to Ix but not Float. Also, if t1, t2, ..., tk belong to Ix, the tuple (t1,t2,...,tn) belongs to Ix.

Recall that tuples are ordered lexicographically, or in dictionary order. Thus, if we have index values from (2,3) to (3,5), they will be listed in order as

```
  [(2,3),(2,4),(2,5),(3,3,(3,4),(3,5)]
```

Formally, the class Ix is defined as follows:

```
class  (Ord a) => Ix a  where
  range       :: (a,a) -> [a]
  index       :: (a,a) a -> Int
  inRange     :: (a,a) -> a -> Bool
```

The function range takes a lower and upper bound of values and returns a list of all values between these bounds. The function index returns the position of a specific value in the list of values between a lower and upper bound. Finally, inRange specifies whether a given index lies within a given bound.

```
For instance:

  range ((2,3),(3,5)) = [(2,3),(2,4),(2,5),(3,3,(3,4),(3,5)]
  index ((2,3),(3,5)) (2,5) = 2
  inRange ((2,3),(3,5)) (3,3) = True
  inRange ((2,3),(3,5)) (3,2) = False

In what follows, we will not bother too much about the details of
the class Ix.  We will be content to use types such as Int and
(Int,Int) as index types of arrays.

It is best to regard an array as a collection of (index,value)
pairs.  Such a list of (index,value) pairs is often called an
"association list".

Creating arrays
---------------

Haskell supports three functions to construct arrays:


1. array :: (Ix a) => (a,a) -> [(a,b)] -> Array a b

   As the type suggests, the function takes a pair of index
   values, the lower and upper bounds of the indices, and an
   association list, in any order, to generate an Array.

   Here are some examples:

     squares = array (1,4) [(2,4),(3,9),(1,1),(4,16)]

     squares = array (1,4) [(i,i*i) | i <- [1..4]]

     somesquares = array (1,4) [(2,4),(1,1),(4,16)]

   The first example explicitly lists out the (index,value) pairs
   in the array, in a random order.  The second example provides
   the same list using list comprehension.  The third example
   shows that not all (index,value) pairs need to be defined.  If
   an index is repeated in the list, the value at that index is
   undefined.  If an index value is out of range, the entire
   array is undefined.

   Here is an example of an array whose indices are pairs of
   integers such that the entry at (i,j) is i+j.

     sample = array ((2,3),(3,5)) [((i,j),i+j) | i <- [2..3],
                                                 j <- [3..5] ]


2. listarray :: (Ix a) => (a,a) -> [b] -> Array a b

   In this form, all array values are entered in the correct
   order of index.  For example:

     squares = listarray (1,4) [1,4,9,16]
     sample = listarray ((2,3),(3,5)) [5,6,7,6,7,8]

   Using the range function on Ix defined above, we can define
   listarray in terms of array as follows:

     listarray bounds values = array bounds (zip (range bounds) values)

3. The final form is allows us to accumulate values in an array.
   The general idea is to extend the "array" function with the
   ability to provide multiple values for each index, with an
   explicit function for combining these multiple values.
```

The function is called accumArray. It takes four arguments:

```
accumArray accumulate-function initial-value index-bounds index-value-list
```

As an illustrative example, suppose we have a list of observations and want to add up all the observations with the same index value to build a "histogram". Here is how we could do it

```
 histogram = accumArray (+) 0 (lower,upper) observations
```

where each value in the histogram for indices in the range (lower,upper) is initialized to 0, observations is list of (index,value) pairs and each such pair is combined with the existing value at that index using the function +.

Extracting and updating values

1. The value at index i in array arr is denoted arr!i

2. We can update an array by providing fresh (index,value) pairs with the operator //

   For instance, if we write

   ```
     squares = (listarray (1,4) [1,4,9,16])//[(1,3),(3,7)]
   ```

   the resulting array corresponds to the association list

   ```
     [(1,3),(2,4),(3,7),(4,16)]
   ```

   As with the "array" function for creating arrays, it is an error to give duplicate index values in the list when updating an array --- if this happens, the value at that index becomes undefined.

Recovering information from an array

Given an unknown array, we can recover information about its indices and values:

1. bounds :: (Ix a) => (Array a b) -> (a,a)

   bounds returns the lower and upper bounds of the indices of the array.

   Examples (from the arrays described above):

   ```
     bounds square = (1,4)
     bounds sample = ((2,3),(3,5))
   ```

2. indices :: (Ix a) => (Array a b) -> [a]

   indices returns the list of indices of the array. Since range applied to a pair of indices returns the list of values between the two indices, we have

   ```
     indices = range . bounds
   ```

   Examples (from the arrays described above):

   ```
     indices square = [1,2,3,4]
     bounds sample = [(2,3),(2,4),(2,5),(3,3),(3,4),(3,5)]
   ```

3. elems :: (Ix a) => (Array a b) -> [b]

   elems returns the list of values in the array, in order of

```
    the indices.

      elems arr = [ arr!i | i <- indices arr ]

    Examples (from the arrays described above):

      elems square = [1,4,9,16]
      elems sample = [5,6,7,6,7,8]

 4. assocs :: (Ix a) => (Array a b) -> [(a,b)]

    assocs returns the contents of the array as a list of
    (index,value) pairs.

    Examples (from the arrays described above):

      assocs square = [(1,1),(2,4),(3,9),(4,16)]
      elems sample = [((2,3),5),((2,4),6),((2,5),7),
                       ((3,3),6),((3,4),7),((3,5),8)]


An illustration: Matrix multiplication
--------------------------------------

To illustrate the ideas above, let us consider a function to
multiply two matrices.  We want to define a function

  matmult :: (Num b) => (Array ((Int,Int),(Int,Int)) b) ->
                        (Array ((Int,Int),(Int,Int)) b) ->
                        (Array ((Int,Int),(Int,Int)) b)

  Suppose we have a matrix A with rA rows, cA columns and matrix
  B with rB rows and cB colums.  We can multiply A by B to get a
  matrix C with rA rows and cB columns provide cA = rB.  We have
  the following formula to compute the (i,j)th entry in C:

    C(i,j) =   sum    A(i,k)*B(k,j)
             1<=k<=cA

  We can then write matmult as follows:

  matmult arra arrb =
    | (uca-lca) == (urb-lrb) =
        array cbounds [((i,j), val i j | (i,j) <- range cbounds ]
    where
      ((lra,lca),(ura,uca)) = bounds arra
      ((lrb,lcb),(urb,ucb)) = bounds arrb
      lrc = 1
      urc = (ura-lra)+1
      lcc = 1
      ucc = (ucb-lcb)+1
      cbounds = ((lrc,lcc),(urc,ucc))

      val :: Int -> Int -> b
      val i j =
        sum  [arra!(lra+i-1,lca+k-1)*arrb!(lrb+k-1,lcb+j-1) |
                    k <- [1..(uca-lca)+1] ]

   We could also use accumArray to eliminate the function val as
   follows:

  matmult arra arrb =
    | (uca-lca) == (urb-lrb) =
        accumArray (+) 0 cbounds
          [((i,j), arra!(lra+i-1,lca+k-1)*arrb!(lrb+k-1,lcb+j-1) |
                      (i,j) <- range cbounds, k <- [1..(uca-lca)+1] ]
    where
      ((lra,lca),(ura,uca)) = bounds arra
      ((lrb,lcb),(urb,ucb)) = bounds arrb
```

```
lrc = 1
urc = (ura-lra)+1
lcc = 1
ucc = (ucb-lcb)+1
cbounds = ((lrc,lcc),(urc,ucc))
```

```
Introduction to Programming, Aug-Nov 2008
Lecture 22, Monday 19 Nov 2008

Memoization using arrays
------------------------

Recall that we proposed to modularize the memo table by defining
a datatype Table a b to store the values of functions of type
a->b supporting the following functions.

  emptytable :: returns an empty memo table
  memofind   :: tells whether a value exists in the memo table
  memolookup :: looks up a value in the memo table
  memoupdate :: adds a new entry to the memo table

Here is an implementation of a memo using Arrays.

The function "emptytable" initializes all entries in the array to
"initval".  To check whether a value exists in the table,
"memofind" compares the existing value to "initval".  For this,
we have to store "initval" permanently in the table, so the
definition of Table a b includes both "Array a b" for the memo
and a separate value of type b, the initial value.  Futher, since
"memofind" checks equality for values im the array, there is a
dependence Eq b.  "memolookup" just returns the value at position
i in the array using the operator "!" while "memoupdate" updates
the value at position i using the "//" operator.

module Arraymemo(Table,emptytable,memofind,memolookup,memoupdate)  where

import Array

data (Ix a,Eq b) => Table a b = T (Array a b) b

emptytable :: (Ix a,Eq b) => (a,a) -> b -> Table a b
emptytable bounds initval =
     T (array bounds [ (i,initval) | i <- range bounds]) initval

memofind ::  (Ix a,Eq b) => (Table a b) -> a-> Bool
memofind (T table initval) index
   | table!index == initval = False
   | otherwise           = True

memolookup :: (Ix a,Eq b) => (Table a b) -> a -> b
memolookup (T table initval) index = table!index

memoupdate :: (Ix a,Eq b) => (Table a b) -> (a,b) -> (Table a b)
memoupdate (T table initval) (index,value)
    = T (table//[(index,value)]) initval


Now, let us see how to memoize the recursive program to compute
the number of paths in a rectangular grid.  Recall that we have a
rectangular array of intersections with m rows and n columns,
with each intersection connected by a one way road going right
and another one way road going down, so from intersection (i,j)
one can go right to intersection (i,j+1) or down to intersection
(i+1,j).  Some intersections are blocked and we want to compute
the number of paths from (1,1) to (m,n).   We assume the input is
0-1 array grid with indices (1,1) to (m,n) such that

   grid(i,j) = 1 if the intersection is open
               0 if the intersection is blocked

Let gridpaths(i,j) denote the number of paths from (i,j) to
(m,n).  In general, we have the recursive definition

   gridpaths(i,j) =   gridpaths(i,j+1) + gridpaths(i+1,j)
```

The base case is

```
   gridpaths(m,n) = 1
```

If grid(i,j) = 0, we have gridpaths(i,j) = 0 since no paths can
pass through intersection (i,j)

For the bottom row, we have

```
   gridpaths(m,j) = gridpaths(m,j+1)
```

and for the rightmost column we have

```
   gridpaths(i,n) = gridpaths(i+1,n)
```

Here is a recursive definition of gridpaths in Haskell. The
first argument is the input grid and the second argument is the
location of an intersection, so

```
   hgridpaths grid (i,j)
```

computes the function gridpaths(i,j) described above.

```
  hgridpaths :: (Array (Int,Int) Int) -> (Int,Int) -> Int
  hgridpaths grid (i,j) =
      | grid!(i,j) == 0   = 0
      | i == m && j == n  = 1
      | i == m            = hgridpaths grid (i,j+1)
      | j == n            = hgridpaths grid (i+1,j)
      | otherwise         = (hgridpaths grid (i,j+1)) +
                            (hgridpaths grid (i+1,j))

    where
      ((lr,lc),(ur,uc)) = bounds grid
      m = ur - lr + 1
      n = uc - lc + 1
```

To memoize this, we write the following. Recall that the
memoized function takes the current memo as an extra argument and
returns a pair of (value, newmemo).

```
  module Gridpaths where

  import Arraymemo
  hgridpaths :: (Array (Int,Int) Int) -> (Int,Int) -> Int
  hgridpaths grid (i,j) = fst (memogridpaths grid (i,j) emptymemo)
    where
      ((lr,lc),(ur,uc)) = bounds grid
      m = ur - lr + 1
      n = uc - lc + 1

      emptymemo :: Table (Int,Int) Int
      emptymemo  = emptytable ((1,m),(1,n)) (-1)

      memogridpaths :: (Array (Int,Int) Int) -> (Int,Int) -> (Table (Int,Int) Int) -> (Int,
Table (Int,Int) Int)
      memogridpaths grid (i,j) memo
        | memofind memo (i,j) = (memolookup memo (i,j),memo)
        | grid!(i,j)  == 0    = (0, memoupdate memo ((i,j),0))
        | i == m && j == n    = (1, memoupdate memo ((i,j),1))
        | i == m              = (lastrowvalue, lastrowmemo)
        | j == n              = (lastcolvalue, lastcolmemo)
        | otherwise           = (valc,memoc)
         where
          (lastrowvalue,memo1) = memogridpaths grid (i,j+1) memo
          lastrowmemo  = memoupdate memo1 ((i,j),lastrowvalue)
          (lastcolvalue,memo2) = memogridpaths grid (i+1,j) memo
          lastcolmemo  = memoupdate memo1 ((i,j),lastcolvalue)
          (vala,memoa) = memogridpaths grid (i+1,j) memo
          (valb,memob) = memogridpaths grid (i,j+1) memoa
          valc         = vala + valb
          memoc        = memoupdate memob ((i,j),valc)
```

```
Longest common subsequence
--------------------------
```

The unix utility "diff" compares two text files and reports the
difference between them in an optimal way. To do this, it tries
to optimally match the two files line by line.

For instance, if file1 has 4 lines x1,x2,x3,x4 and file2 has 5
lines y1,y2,y3,y4,y5 such that x1 = y2, x3 = y4 and x4 = y5, diff
will match up x1,x3,x4 with y2,y4,y5 and report that file2 has an
extra line y1 before this sequence, and between x1=y2 and x3=y4,
file1 has an extra line x2 while file2 has an extra line y3.

In this scenario, the matching sequences x1,x3,x4 and y2,y4,y5
are subsequences of the two original sequences of lines. What
diff does is to compute the longest common subsequence of lines
between two files.

Abstractly, we can think of the two sequences as consisting of a
set of letters, such as

```
  V = T G C A T G G A T C
  W = G T T T G C A C
```

Now we can drop some positions and obtain a subsequence. For
instance "T G A C" is a subsequence of both words. "G T G A C"
is a longer common subsequence as is "T G C A C". These are
(probably!) the longest common subsequences in this example,
which shows that the longest common subsequence is not unique.

Our aim is to find the longest common subsequence (lcs), but for the
moment we concentrate on finding the *length* of the longest
common subsequence (llcs) instead.

Consider the example above. When we match the first G in W to a
G in V, we can match it to either the G at position 2 in V or at
position 6 or 7 in V. Suppose we have a solution in
which we match the first G in to the G at position 7 in V. Then,
all further matches from W will be to the right of position 7 in
V. If we take this matching and shift the match for the first G back to
the G at position 2, we still have a matching. And, by moving
this match back, perhaps we might enable longer matches between W
and V. Thus, in general it is best to match a letter as far to
the left as possible.

Let LLCS(i,j) denote the length of the longest common subsequence
starting at position i in V and at position j in W.

We have two cases to consider.

1. Suppose V[i] == W[j].

   If there is any matches that uses V[i] or W[j], we can shift
   it so that V[i] matches W[j] and get a new matching that is at
   least as good. So, we may as well match V[i] to W[j] and
   continue from (i+1,j+1). This suggests that

   ```
   If V[i] == W[j], LLCS(i,j) = 1 + LLCS(i+1,j+1)
   ```

2. On the other hand, suppose V[i] /= W[j]

```
   We clearly cannot use both V[i] and W[j] in the final matching
   --- if we use V[i], it matches with W[k] for some k > j and
   nothing beyond V[i] can then match W[j].  A symmetric argument
   shows that if we use W[j] in the final matching, V[i] can play
   no role.

   However, a priori, we have no information about which of V[i]
   or W[j] to drop from the solution, so we try both and take the
   maximum.

   If V[i] /= W[j], LLCS(i,j) = max(LLCS(i+1,j),   <-- drop V[i]
                                    LLCS(i,j+1))   <-- drop W[j]

3. Base cases

   If V is of length m and W is of length n, we have

   LLCS(m,n) = 1, if V[m] == W[n]
               0, otherwise

   If we are at the last letter of V, we have

   LLCS(m,j) = 1,           if V[m] == W[j]
               LLCS(m,j+1), otherwise

   Symmetrically, at the last letter of W, we have

   LLCS(i,n) = 1,           if V[i] == W[n]
               LLCS(i+1,n), otherwise

We can derive a dynamic programming solution to this problem by
filling up the values LCSS[i,j] in an M x N array, starting with
the last row and column as base cases.
```