Haskell
A Tutorial Introduction

Jerud J. Mead

Computer Science Department
Bucknell University
Lewisburg, PA 17837

Contents

1 What is Functional Programming?

Programming is the process of describing the transformation of input values to corresponding output values. Assuming that a program is deterministic (the same input will always produce the same output), this means that a program acts like a *function* which maps the input values to output values. And when I use the term *function*, I use it in the mathematical sense – i.e., as discussed in a course on discrete mathematics! It is this fundamental characteristic of programming which is at the heart of the functional paradigm. Though this functional characteristic is present in familiar languages such as Pascal and Java , it is in modern functional programming languages that the functional paradigm is most thoroughly exploited.

What differentiates mathematical functions from those with which we are familiar in programming? In mathematics when we write

$$f(x) = x^2 - 3x + 2$$

the independent variable x acts as a place holder for a value which is supplied. In the Java function

```
int poly (int x) {
    // inefficient function definition for x^2 -3x +2

    int y;

    y = x*x;

    return y - 3*x +2
}
```

the variable y represents not just a value, but also a location in memory where that value is stored, i.e., Java functions have local state, meaning memory locations to which new values can be assigned. This idea of state does not exist for mathematical functions. Thus, in *pure* functional programming there are no variables as such, only constants, parameters, and values. The following Java function behaves just as the one above but is written in the functional style.

```
int poly (int x) {
    // inefficient function definition for x^2 -3x +2

    return x*x - 3*x +2
}
```

Some functional programming languages, such as the lisp-family of languages (scheme, common lisp, etc.) and ML provide a functional environment, but allow for state in some situations: these are referred to as *impure* functional languages. Languages such as Miranda and Haskell on the other hand, are called *pure* functional languages because they make no provision for state. See Chapter 14 in *Programming Language Design Concepts* by David Watt for a more extensive discussion of the characteristics of functional programming languages.

Historically, one problem of functional programming languages has been that the execution time for a functional program is considerably longer than an equivalent program written in an imperative language such as Pascal or Java. On the other hand, one of the main advantages of the functional paradigm is that it allows for very (mathematically) precise function definitions. The result is that functional programs are considerably shorter and easier to prove correct than their imperative counterparts. For these reasons, functional programming languages have been used as specification languages. A functional program serves as an executable "wiring diagram", and an imperative program is written to follow the functional specification.

1.1 What is a Functional Program?

A functional programming language is one which implements the functional paradigm. A typical program consists of a sequence of function definitions and an expression that references these definitions. Execution of the program

simply involves the evaluation of the expression. The expression, of course, coincides with the main program or function in an imperative program. Since most functional programming languages are interpreted, the user typically specifies a file containing function definitions and then interactively enters an expression to be evaluated — the interpreter responds with the result of the evaluation. The file of function definitions is what we will think of as constituting the functional program.

There are four important features of modern functional programming languages: the use of lazy evaluation, true polymorphic functions, higher-order functions, and strong typing.

1. Lazy evaluation refers to the fact that the interpreter doesn't evaluate a function parameter until it is actually used. This has various advantages, including the ability to represent potentially infinite data structures such as the set of natural numbers.

2. A polymorphic function is one whose definition is valid over a range of types. Thus, a function for adding something to a list can be written independent of the type of the values on the list.

3. A higher-order function is one which takes other functions as parameters. These functions give the programmer a powerful mechanism for elegant problem solutions. Higher-order functions are also often reusable in other programs.

4. Strong typing in functional languages means that the type of every function can be determined before expression evaluation. Since the programmer is encouraged (not required) to specify types for all functions, the interpreter can compare its deduced type to the programmer's specified type. If they are not the same an error is given. Experience seems to indicate that when the programmer and interpreter agree on the types of functions, the program is correct.

Other features which are peculiar to modern functional programming languages are algebraic types (rather than pointers), function definitions using pattern matching, and the use of lists as the primary data structure in problem solving.

1.2 A Functional Programming Environment

A functional programming environment is like a very fancy calculator, where the *user* gets to define the functions which can be used. The functions are defined by writing descriptions into a text file. The environment loads the specified file of functions definitions, verifying correct syntax and translating the functions into a convenient internal form. The environment then allows the user to specify expressions which should be evaluated. An expression, of course, must be written in terms of functions which the interpreter knows about. Of course it is unreasonable to expect the user to define *all* functions which will be used. Instead, each functional programming environment will somehow provide a collection of pre-defined functions which are either automatically loaded when the environment is run or already built into the structure of the environment. Many interpreters have both characteristics. The most basic functions, such as the arithmetic functions, are built in. Other commonly used functions are provided by the environment as a standard startup file. This startup file has become known as a *prelude*.

1.3 Is Functional Programming *Real* Programming?

When dealing with different programming paradigms it is natural to ask the question "Is one paradigm fundamentally more powerful than another?" If we compare the functional paradigm with the more familiar imperative paradigm, the answer is NO. If you can write a program in Java or Pascal, then you can write an equivalent program in a functional language; and the reverse is true as well.

One of the most difficult aspect of programming in any language is input and output. In functional programming, as you will see, non–interactive I/O is actually easier than with imperative languages. Interactive I/O, on the other hand, is very difficult for pure functional programming languages (those without state), much more so than imperative languages. But there are well defined techniques that make interactive I/O feasible for functional programming. So it is important for you to remember as you go through this tutorial, functional programming may seem a bit strange at first, but any program you have written in Pascal or Java can also be written using the functional paradigm.

1.4 What is *Hugs*?

Above I referred to Haskell as a functional programming language. This is not quite right. It is really the definition of a family of functional languages — when that definition is actually implemented you get a useful language. *Hugs* is such an implementation – we say that *Hugs* implements the Haskell standard – it was developed by Mark P. Jones as a testbed for his research activities into various aspects of functional programming languages. While the Haskell standard has been around for several years, it continues to be the primary vehicle for research into new programming language structures, especially type systems. If you want to learn more about the Haskell research activities visit their web at http://www.haskell.org.

First released in 1991, *Hugs* continues to be developed, with the most recent version coming out in 2006. Versions of *Hugs* now exist for almost all popular operating systems. All versions are free and can be downloaded from the following address http://www.haskell.org/hugs. The *Hugs* interpreter will be fully described in the next few sections.

Hugs has features that are standard in modern functional languages, including lazy evaluation, polymorphic functions, pattern matching, algebraic types, higher-order functions, and strong typing.

2 Using the *Hugs* Interpreter

To start the *Hugs* interpreter simply type 'hugs' at a unix prompt. Do so! You should see the following displayed on the screen.

```
[myaccount@server path]$ hugs
__   __ __  __  ____   ___      ----------------------------------------
||   || ||  || ||  || ||__      Hugs 98: Based on the Haskell 98 standard
||___|| ||__|| ||__||  __||     Copyright (c) 1994-2005
||---||         ___||           World Wide Web: http://haskell.org/hugs
||   ||                         Bugs: http://hackage.haskell.org/trac/hugs
||   || Version: September 2006 ----------------------------------------

Haskell 98 mode: Restart with command line option -98 to enable extensions

Type :? for help
Hugs>
```

Though it is not mentioned in the startup information, a file called `Prelude.hs` is automatically loaded. This file is the basic library of pre-defined functions for *Hugs* and provides many function definitions intended to make programming easier. You can find the file at

```
/usr/lib/hugs/packages/base/Prelude.hs
```

If you look in this file you will see examples of *Hugs* function and type definitions. [Be warned that the file contains features which we will not be discussing in class.]

Continuing with the interpreter... The '`Hugs>`' prompt is not very creative, but it has the advantage that it doesn't take much space from the line. (If you check out the ':set' command you will see that you can set your own prompt.) We have answered the first of the three essential questions for the use of any system:

- How do I get in?
- How do I get help?
- How do I get out?

The initial output from *Hugs* tells us how to get help (that's the first two taken care of). If you follow the directions you should get the following list on your screen of all the system level commands understood by *Hugs*. (Notice that you must type ':?' at the prompt, not just'?'.)

```
Hugs> :?
LIST OF COMMANDS:  Any command may be abbreviated to :c where
c is the first character in the full name.

:load <filenames>   load modules from specified files
:load               clear all files except prelude
:also <filenames>   read additional modules
:reload             repeat last load command
:project <filename> use project file
:edit <filename>    edit file
:edit               edit last module
:module <module>    set module for evaluating expressions
<expr>              evaluate expression
:type <expr>        print type of expression
```

```
:?                  display this list of commands
:set <options>      set command line options
:set                help on command line options
:names [pat]        list names currently in scope
:info <names>       describe named objects
:browse <modules>   browse names defined in <modules>
:find <name>        edit module containing definition of name
:cd dir             change directory
:gc                 force garbage collection
:version            print Hugs version
:quit               exit Hugs interpreter
Hugs>
```

OK! Now we can see the answer to the other question. To get out of *Hugs* you just type ':quit' (actually, ':q' works as well). Do so. You should see:

```
Hugs> :quit
[Leaving Hugs]
```

The other commands listed by the help command will be introduced as we go along. Your use will most likely be limited to the commands `load`, `reload`, `type`, and `quit`.

Before proceeding, we will take a quick look at the '`set <options>`' command, since there is an option we want to set. Restart *Hugs* and at the prompt type '`:set`' and observe the output.

```
Hugs> :set
TOGGLES: groups begin with +/- to turn options on/off resp.
s    Print no. reductions/cells after eval
t    Print type after evaluation
f    Terminate evaluation on first error
g    Print no. cells recovered after gc
l    Literate modules as default
.    Print dots to show progress
q    Print nothing to show progress
w    Always show which modules are loaded
k    Show kind errors in full
u    Use "show" to display results
I    Display results of IO programs
T    Apply 'defaulting' when printing types
R    Enable root optimisation

OTHER OPTIONS: (leading + or - makes no difference)
hnum Set heap size (cannot be changed within Hugs)
pstr Set prompt string to str
rstr Set repeat last expression string to str
Pstr Set search path for modules to str
Sstr Set list of source file suffixes to str
Estr Use editor setting given by str
cnum Set constraint cutoff limit
Fstr Set preprocessor filter to str

Current settings: +quR -stgl.QwkIT -h1000000 -p"%s> " -r$$i -c40
Search path     : -Pi.:{Home}/lib/hugs/packages/*:/usr/local/lib/hugs/packages/*:{Hugs}/packages/*
Editor setting  : -Eemacs
Preprocessor    : -F
```

```
Compatibility   : Haskell 98 (+98)
Hugs>
```

Most of these features will be of no interest to us during the term, but we do want to set the very first of the options. So at the prompt enter

```
Hugs> :set +s
```

which indicates to always print execution statistics when an expression is evaluated.

2.1 Setting Things Up

The common way to work with *Hugs* is to have your editor window and hugs session both open and then just switch back and forth. If you edit and save a file, you then need to either `:load` (`:l`) the file again to see the changes or `:reload` (`:r`) to bring in changes from all previously loaded files in *Hugs*.

So that you do not have set the flags every time you run *Hugs*, you need to add an environment variable to your Bash environment. Add the following line to your `.bashrc`:

```
export HUGSFLAGS='+sl +u'
```

Remember that you will need to "source" your `.bashrc` file in order to see the changes.

```
source ~/.bashrc
```

2.2 Simple calculations

Now that the important questions have been answered we can move on to the computational facilities of the *Hugs* interpreter. Using *Hugs* is rather like using a high-level programmable calculator; once the interpreter is loaded, the system prints a prompt '`Hugs>`' and waits for you to enter an expression (or a *Hugs* command) and press the enter (Return) key. Once the input is complete, *Hugs* evaluates the expression and prints its value on the screen; it then returns to the original prompt and waits for the next expression. Try some examples. At the prompt enter the expression '(2+3)*8'. (Notice that the expression is not preceded by a colon.) When you press return the expression is evaluated by *Hugs* and the result '40' printed on the screen.

```
Hugs> (2+3)*8
40
(12 reductions, 10 cells)
```

The extra bit of information printed on the last line is a measure of *Hugs*'s system utilization in evaluating the expression: '5 reductions' is a measure of CPU usage (the number of computational steps, in a sense) and '9 cells' is a measure of memory usage. These measures can provide a convenient mechanism for comparing the efficiency of different algorithmic solutions to a problem.

[You may find that the number of reductions and the number of cells are different in these examples – that's because they were produced with an earlier version of *Hugs*.]

2.3 Arithmetic Expressions

Arithmetic expressions are handled in *Hugs* much as you would expect, but not exactly. The operations `+, -, *, /` are defined on both integer and floating point values (but not mixed!) and are used in the infix style. There are two other standard integer operations which are used not in infix style, but in *prefix* style – this means that the operator is written first followed by its operands. The two operators are `'div'` and `'mod'`. Enter the following expression:

```
3 + (div 10 3)
```

and you will see the following output.

```
Hugs> 3 + (div 10 3)
6
(13 reductions, 9 cells)
```

Of course you may be the sort of person who just has to use infix notation (or just has to use prefix notation – take your choice!). *Hugs* has accommodated both styles. To turn a binary prefix operator into an infix operator just put back-quote characters around it

```
Hugs> 10 `div` 3
3
(10 reductions, 9 cells)
```

To turn a binary infix operator into a prefix operator surround it with a pair of parentheses.

```
Hugs> (+) 3 5
8
(10 reductions, 9 cells)
```

A point about the *Hugs* functional notation. In most languages you have encountered, function application requires that function arguments appear in a comma-separated list enclosed by parentheses. This is not the *Hugs* style. Rather, a function's arguments appear as a space-separated list and not enclosed in any sort of bounding notation. This notation turns out to be very convenient and we will revisit this discussion in future sections.

For now, it is up to you to experiment with the arithmetic expressions to find out what is legal and what is not. Be sure to try some mixed mode arithmetic to see what happens.

To make this experimentation and future work more convenient, the interpreter provides a special "command" which will be replaced by the most recent expression typed into *Hugs*. Here is a sequence of expressions to enter at the *Hugs* prompt to see how this command works.

```
2 + 3
$$         -- this is the ``repeat last expression'' command
$$ + $$
2 * $$ + 4
```

As an exercise for yourself, write down what expression replaces the `$$` in each instance. The result computed for each expression should help answer any questions.

2.4 Boolean and Conditional Expressions

In addition to integer and floating point numbers *Hugs* also provides the Boolean type along with all the familiar operators which generate Boolean results. Try some of the following expressions in the interpreter to see what it does.

```
5 == (3 + 2)
(div 5 3) /= 2     -- this is the not-equal operator
(<) 3 5
(3 < 5) == (4 < 3)
(3 < 5) && (4 < 3)
(3 < 5) || (4 < 3)
```

Clearly, all kinds of combinations are possible. You might want to see what happens when you leave out some of the parentheses, just to get an idea of how *Hugs* handles precedence.

Another very convenient type of expression borrowed from *C* is the conditional expression. This expression looks a lot like the standard `if...then...else` statement (Pascal notation) but cast in the context of values rather than statements. Here is an example.

```
Hugs> if (4 < 3) then 4 else 5
5
(16 reductions, 12 cells)
Hugs> if (3 < 4) then 4 else 5
4
(14 reductions, 12 cells)
```

2.5 Defining functions

Of course we aren't using *Hugs* so that we can do slick computations like the ones above. We want to use *Hugs* as a programming language. The thing you must remember is that *Hugs* only understands functions and the application of a function to its arguments. If you think back to programs which you have written, few will seem to fit into that mold. However, we will see that a functional language like *Hugs* has all the flexibility and computational power of Pascal and Java.

First, remember that a *Hugs* program is a list of function definitions and an expression to be evaluated. The way we do this in *Hugs* is to create a file containing the function definitions, have the *Hugs* system load that file, and then enter interactively the expression(s) that we want to evaluate. Here is a first program example.

The first step is to learn to define your own functions. We will start out simply. Open your editor and enter the following lines.

```
This file contains the definition for the factorial function.

>fact :: Int -> Int   -- this specifies the type of the function fact
>
>fact 0 = 1
>fact n = n * (fact (n-1))
```

There are three things to note here. First, the type specification for '`fact`' matches the common mathematical notation for functions. In this case the indication is that '`fact`' takes one integer argument and produces an integer result. This definition of factorial looks just like the definition you see in a math text – `fact` is a function with domain `int` and range `int`.

Second, notice the natural use of recursion, which is the hallmark of any functional programming language.

Finally notice carefully the six lines of the program – they illustrate a style of programming called *literate programming*. In this style the roles of comment and program line are reversed – comments are free-form and require no special notation, while program lines must begin with the greater than sign; a sequence of program lines must also be preceded and followed by blank lines! This style makes creating a handin file much easier. You can take your normal program file (with program lines starting with '`>`') and freely copy in program output or commentary.

> *Note: In order for literate programming set when you load a file, you must give the file a name which ends* `.lhs`*: if you use the* `.hs` *ending the non-literate programming style will be assumed.*

Now save the content of the editor in a file (say '`labA.lhs`'). This file contains what we will get used to calling a *Hugs* program. Now you must make your *Hugs* interpreter session aware of the program file you have created. This is where the 'load' command comes in. If you have reproduced the above lines accurately then the following should appear when you execute the indicated *Hugs* command. [Note: The two characters '–' mark the remainder of a line as a comment.]

```
Hugs> :load labA -- :load can be abbreviated to :l
Reading file "labA.lhs":

Hugs session for:
/usr/local/share/hugs/lib/Prelude.hs
labA.hs
Main>
```

Notice that the prompt has changed to reflect that a new context has been loaded into *Hugs* on top of the standard prelude.

(In this example, if you have not reproduced the lines accurately you may see some error messages. Return to the editor and double check what you have entered. After correcting any mistakes resave the file and repeat the load procedure in *Hugs*.) So now what is in the file `labA.lhs` should be integrated into *Hugs*. Let's see. Determine the factorial for a few values using 'fact'. If you try some appropriate (not very large) values you should get an idea as to the integer range supported by *Hugs*. Here are results that I got.

```
Main> fact 3
6
(58 reductions, 66 cells)
Main> fact 7
5040
(110 reductions, 133 cells)
Main> fact 10
3628800
(149 reductions, 184 cells)
Main>
```

Problem 1

1. Let's try to integrate 'fact' into more complex computations. There are 10 ways to select 9 students from a class of 10 (i.e., the are 10 ways to leave a student out!). But how many ways are there to select 5 students from a class of 10? The answer is given by the general equation

$$\binom{n}{k} = \frac{n!}{k! * (n-k)!}$$

   Write an expression for $\binom{10}{5}$, 10 choose 5, based on this equation and evaluate it in *Hugs*. Remember! The values we are working with are integers, so be sure to use `div` rather than '`/`', as you would do in Java.

2. Before we leave this example, re-edit your test file and answer the following questions (give complete answers).

   (a) Comment out the line defining 'fact 0', and re-save the file. Now re-load the file into the *Hugs* system and evaluate the expression 'fact 4'. What happens? Why does it happen? Correct the problem.

   (b) Are the parentheses around `fact (n - 1)` necessary? How about around `n - 1`? Run appropriate tests to answer both these questions. Explain your answers.

3. Now lets try to make the choose expression a function! The `choose` function takes two integer arguments `n` and `k` and returns the result of $\binom{n}{k}$. When `n` is 0, the function should return the value 0. Mimicking what you did for factorial function, *add* a definition to your test file for this function, load the file into *Hugs*, and evaluate the function at a couple of argument values.

4. What happens if you try to get the factorial of 50? What happens if you try to find $\binom{50}{5}$?

5. There is another way to calculate the choose function that is more recursive.

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

Implement a function *choose*2 which implements the choose expression recursively.

6. What happens if you try to run `choose2 50 5`? Does it work? Why does it take a while to provide a result?
7. One last thing for choose: comment out the type signature for the `fact` and `choose` functions. Do they work now? Why?
8. Now lets try a new function. The Fibonacci function takes an integer argument `n` and returns the sum of the $(n-1)$st and $(n-2)$nd Fibonacci numbers. For the arguments `0` and `1`, the function returns the value `1`. Implement this function as `fib` in your current lab Haskell file. Verify that it is correct by testing it on a few values and copying the results under the function implementation.

2.6 Conditional expressions

The two functions you defined above are examples of functions defined by cases. There are three alternative mechanisms for implementing the functions. We will work with the factorial function in the following development.

2.6.1 `if...then...else` expressions

You are very familiar with the if statement from procedural languages such as Pascal and Java. In a functional language this structure is used as part of an expression. Instead of giving alternative actions to take, alternative values are specified. Thus, to define the function 'fact' using the `if...then...else` expression we merely write

```
fact n = if n==0 then 1 else n*(fact (n-1))
```

Notice that the equality operator is '==', just as in Java. This expression can be spread over more than 1 line as follows:

```
>fact n = if n==0
>            then 1
>            else n*(fact (n-1))
```

There are stringent rules governing the use of indentation, but we'll save those for a later discussion.

Problem 2

Write a new definition for the `fib`, call it `fibsel`, making use of the `selection` expression. Add it to your file following the definition of `fib`. [Nesting the selection expressions should work the way you expect.]

2.6.2 `case` expressions

Hugs also has a case expression which is similar to the Java `select` statement. Our factorial function could be written using the case expression as follows:

```
>fact n = case n of
>              0 -> 1
>              n -> n*(fact (n-1))
```

The components of the case expression are evaluated from top to bottom until one is found which matches the argument value. So the value 1 is returned if the argument is 0, and 'n*(fact (n-1))' is returned in every other case.

Problem 3

Write a new definition for the `fib`, call it `fibcase`, making use of the `case` expression. Add it to your file following the definition of `fibsel`.

2.6.3 guarded expressions

The third conditional expression is called a guarded expression. The 'fact' function is defined as follows (it is really just a reformulation of the `if...then...else` syntax).

```
>fact n
>     | n == 0    = 1
>     | otherwise = n*(fact (n-1))
```

In this method each guard (the part between the '|' and the '=') is a Boolean expression. As we will see in later examples, the guards do not have to be disjoint but they must be all inclusive - thus the use of the 'otherwise' guard. Try the 'fib' function using this method.

Problem 4

1. Write a function

   ```
   grade ::  Int -> Char
   ```

   which converts a grade in the range from 0 to 100 to a letter according to the following:

Scores	Grades
90 – 100	'A'
80 – 89	'B'
70 – 79	'C'
60 – 69	'D'
0 – 59	'F'

   Any other argument value should return 'E', for error. Use the appropriate expression type for implementing this (there's really just one).

2. Write a function `abs2` which determines the absolute value of an integer. Test it out. Notice that when you use a negative number as an argument to the function you must enclose it in parentheses. So use the following form.

   ```
   abs2 (-5)
   ```

   If you were to enter

   ```
   abs2 -5
   ```

   the interpreter would read the minus sign as a separate argument – i.e., it would think `abs2` was taking two arguments, the minus operation and the integer 5.

3 *Hugs* Types

In the last section, you saw some basic features of the *Hugs* interpreter and a very small bit of the *Hugs* language. In this section, you will learn about various types in *Hugs* including simple types, function types, and the important list and tuple types. You will also learn about more language features of *Hugs* and how to use them.

3.1 Simple Types

There are four simple types: Boolean (Bool), character (Char), integer (Int), and real (Float) [the *Hugs* name for the type is in parentheses]. Each of these types has the usual array of associated functions, which are summarized in Figure 1. Notice that in *Hugs* ,all function names must begin with a lower-case letter —- all type names (and type constructor names) must begin with an upper-case letter (for example, the Boolean values are True and False).

A bit of experimentation on your part (by trying out various expressions in the interpreter) should give you the basic idea of how these familiar functions work. One thing to mention about Figure 1, the last entry labeled 'others', shows the usual comparison functions are defined. The 'a' in the type definition part is in place of each of the types mentioned above. This saves having to write a separate definition for each individual type.

3.2 Function Types

In a language such as Pascal or Java the definition of function includes the name of the function, a list of parameters and their types, and the type of the value to be returned by the function. In *Hugs*, we have the same requirements, but the syntax is quite different and derived from (hopefully) familiar mathematical notation. In mathematics, we are used to writing

$$f : A \longrightarrow B$$

to specify a function named 'f' which associates with each value in the set 'A' a unique value from the set 'B' – 'f' takes a value from 'A' and returns a value from 'B'. This notation is used in *Hugs*, and we have seen it above. In previous examples, however, we have dealt with functions with only one parameter. How do we extend the notation to functions on more than one parameter? The function 'gcd' (greatest common divisor), for example, takes two integer parameters and returns an integer value. To specify this in *Hugs*, we write

```
gcd :: Int -> Int -> Int
```

The first `Int` is the type of the first parameter, the second `Int` is the type of the second parameter, and the third `Int` is the type of the value returned. This scheme generalizes in the obvious way to functions of more than two parameters. We will see examples of this notation throughout the rest of this tutorial. Two things to remember: the very last type name is always the type of the return value, and the number of parameters is one less than the number of types in the definition (or just the number of `->`'s).

Problem 5

> Add to your test file the definition of a function 'hyp' that takes two integer values (representing the lengths of the sides of a right triangle) and returns a real value representing the length of the hypotenuse for the triangle (assume the integer values are non-negative). This function requires the use of `fromIntegral` (see Figure 1[1]) to convert from the integer parameters to `Float` values. Modify the definition so that for negative values in either of the parameters the value '0' is returned?
>
> Be sure to include the type specification for the new function.

∎

[1] The type of `fromIntegral` is actually `(Integral a, Num b) => a -> b`. This notation will be explained later, but for now the simple type given in Figure 1 will suffice.

Bool	&&	::	Bool -> Bool -> Bool -- infix and
	||	::	Bool -> Bool -> Bool -- infix or
	not	::	Bool -> Bool
Char	ord	::	Char -> Int
	chr	::	Int -> Char
	isAscii, isControl,		
	isPrint, isSpace	::	Char -> Bool
	toUpper, toLower	::	Char -> Char
Int	div, mod, quot, rem	::	Int -> Int -> Int
	+, -, *	::	Int -> Int -> Int
	^	::	Int -> Int -> Int-- exponentiation
	even, odd	::	Int -> Int
	gcd, lcm	::	Int -> Int -> Int
	even, odd	::	Int -> Int
	fromIntegral	::	Int -> Float
Float	+, -, *, /	::	Float -> Float -> Float
	^	::	Float -> Int -> Float -- exponentiation
	truncate	::	Float -> Int
	sin, cos, tan, log,		
	exp, sqrt, log, log10	::	Float -> Float
	pi	::	Float -- pi = 3.1415926535
others	==, /=, <, <=, >, >=	::	a -> a -> Bool

Figure 1: Basic Functions

3.3 Constructing Types

Hugs provides three mechanisms for combining previously defined types. In this section we will discuss two of these mechanisms: lists and tuples. In class we will discuss these types in a more mathematical way, but for now we just want to be able to use them and build some intuition as to when to use them.

3.3.1 Tuples

Tuples in *Hugs* most closely correspond to records in Pascal or classes without methods in Java . They are representations of the Cartesian products of two or more types–this is a term which you should be familiar from a discrete mathematics course. Recall that a record is defined by specifying a fixed number of named components, where each component has a (possibly different) type. Tuples have these same characteristics, except that the components are not identified by names, but by their position in the tuple. A tuple type combining a character value, an integer value, and a real value could be defined in *Hugs* as

```
(Char, Int, Float)
```

One convenient use of tuples is for returning more than one value from a function.

Example 1 – `fancyDiv`

The following *Hugs* function takes two integer parameters and returns the quotient and remainder when you divide the first by the second parameter.

```
fancyDiv :: Int -> Int -> (Int, Int)

fancyDiv n m = (n `quot` m, n `rem` m)
```

Add this function to your test file, load it into *Hugs*, and try it out on a few values.

Important: Remember that in *Hugs*, as in most modern functional programming languages, function application is handled a bit differently from more familiar languages. In Pascal or Java we could expect to call the function by

```
fancyDiv (12, 5)
```

since it takes two parameters. In *Hugs*, however, this call would indicate that `fancyDiv` takes one parameter, and that one parameter is a tuple. In *Hugs*, of course,

```
fancyDiv 12 5
```

is the correct syntax.

Example 2 – `fancyDiv (I)`

We can investigate two more aspects of *Hugs* by extending the `fancyDiv` example a bit. If you evaluate the expression

```
fancyDiv 12 0
```

you will get a program error. It would be nice to have a way to handle such errors with a bit more grace — for example, if we could put in our own error message. This can be done by utilizing a special *Hugs* function called 'error':

```
fancyDiv :: Int -> Int -> (Int, Int)

fancyDiv n m = if (m==0)
                  then error "Tried to divide by zero"
                  else (n `quot` m, n `rem` m)
```

Give it a try.

Example 3 – `fancyDiv (II)`

Another modification we can make to the example is a bit contrived. It is often the case, when doing a computation, that the final result is quite complex, and that breaking it up into parts is a bit more understandable. In this example we could do the following:

```
fancyDiv :: Int -> Int -> (Int, Int)

fancyDiv n m = if (m==0)
                  then error "Tried to divide by zero\n"
                  else (quotient, remainder)
               where
                    quotient  = n `quot` m
                    remainder = n `rem` m
```

This is an example of creating definitions local to a function definition. There are two points to make about this example. First, a note on indentation. Local definitions are introduced by the `where` keyword. When `where` appears, the first character of the next definition (in this case `quotient`) sets a column. Every other local definition in the `where`-clause must begin in this same column. This indentation rule is quite natural and saves having to use lots of parentheses to mark scope. If you follow the form that I use in examples, you will have no problems. Second, the identifiers 'quotient' and 'remainder' are *not* integer variables. The value of 'quotient' is the actual expression on the right-hand side. The identifiers are used only in a substitutional way. Thus, the last example is equivalent to the previous. The `where`-clause just helps the programmer decompose the complexity of a return expression. This technique will be used commonly.

Problem 6

1. Try the techniques illustrated with `fancyDiv` on the following problem. Implement a function `roots` which will take three integer parameters, representing the coefficients of a quadratic equation, and return the two roots of the equation, if there are any. The type of this function will be as follows.

   ```
   roots ::  Int -> Int -> Int -> (Float, Float)
   ```

   Remember the quadratic equation:

   $$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

   The function should behave as follows.

   ```
   Main> roots 1 2 1
   (-1.0,-1.0)
   (112 reductions, 171 cells)
   Main> roots 2 4 1
   (-0.292893,-1.70711)
   (112 reductions, 180 cells)
   Main> roots 1 2 4

   Program error: No real roots
   (60 reductions, 275 cells)
   ```

   You might want to do this in stages as we did the 'fancyDiv' function. In the end you should make sure that the function prints an error message if there are no real roots. Also, if there is just one root, return an ordered pair with the same value in each position. To determine the number of roots you must compute the value of the discriminant (the part under the square root!). Use a 'where' clause to get `Float` versions of each parameter, and then to compute the discriminant and the roots. Using the `where` clause in this way keeps the form of your return value simpler and easier to understand.

   Be sure to add the function definition to your test file.

2. When there are no real roots for a quadratic equation, there are always two imaginary roots from the set of complex numbers (using $i = \sqrt{-1}$). Modify the definition of 'roots' to make a new function 'allRoots' which has the same parameters but returns two roots, where each root has an imaginary and real part.

   There are two important ideas to consider here. First, you should represent a complex number as a pair of `Floats`; this means that the function should return a pair of pairs – i.e., it should have type as follows.

   ```
   allRoots ::  Int -> Int -> Int -> ((Float,Float),(Float,Float))
   ```

   Here's a sample execution of `allRoots`.

```
Main> allRoots 1 2 3
((-1.0,1.41421),(-1.0,-1.41421))
(148 reductions, 386 cells)
Main> allRoots 1 2 1
((-1.0,0.0),(-1.0,-0.0))
(132 reductions, 258 cells)
```

[The result for the first example above would be more traditionally written as `-1.0 + 2.41421i` and `-1.0 - 2.41421i`.]

Second, look at the different possibilities for the answers and then use the where clause to compute partial results which are then used in combination to determine the values returned – for example, compute the real part of a result and the imaginary part separately.

∎

There are a couple of important functions associated with 2-tuples (pairs). The functions `fst` and `snd` are predefined in *Hugs* as follows:

```
fst :: (a, b) -> a
fst (x,y) = x

snd :: (a, b) -> b
snd (x,y) = y
```

You can define similar functions for 3-tuples (triples)[2]:

```
fst3 :: (a, b, c) -> a
fst3 (x,y,z) = x

snd3 :: (a, b, c) -> b
snd3 (x,y,z) = y

thd3 :: (a, b, c) -> c
thd3 (x,y,z) = z
```

These functions give us a way to pick apart a pair or triple to get at the components. Note that in the function type specification above we have used identifiers `a`, `b`, and `c` in place of known types. In this situation `a`, `b`, and `c` are known as type variables, and they indicate that any types could be substituted. The function definitions are independent of the component types involved. These are examples of polymorphic function definitions.

3.3.2 Lists

We are gradually building up more machinery for solving problems. But so far we have only worked with numbers (except when using the `error` function). The real key to the power of a functional language is the list type constructor. Lists have two characteristics which differentiate them from tuples. First, unlike a tuple, every element of a list must have the same type; this is more restrictive. Second, the length of a list is variable, where a tuple has a fixed number of components. In fact in *Hugs*, a list can be (potentially) infinite in length.

The type of a list is designated by using square brackets. The function `sum`, which is defined in the standard prelude, takes a list of numbers as its parameter and returns a result which is the sum of the values in the list. The type of `sum` is written as:

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

[2]The triple functions are *not* predefined in *Hugs*.

Examples of concrete list types include: `[Int]` for lists of integers, `[Char]` for lists of characters (more on this type in a minute), and `[Float]` for lists of real numbers. Notice that in the `sum` function type, there is a polymorphic list type `[a]` which matches a list of any element type. But we can also have lists of constructed types; so that `[(Int, Char)]` specifies the type of lists, where each element on the list is a pair consisting of an integer and a character. Combinations of these type constructors can clearly get out of hand, but problems usually give a pretty clear indication of the appropriate combinations.

Actual lists can be specified in a several ways. One, called list comprehension, will be saved for Section 5.4. Specific lists of values are specified by enclosing a (comma-separated) sequence of values within closed brackets: `[4, 3, -1, 5]` is a list of integers, with 4 being at the *head* of the list and 5 at the end. The empty list is indicated by `[]`.

Lists of characters are a special case. The character list

```
['l', 'i', 's', 't']
```

can also be (and usually is) written `"list"`. Notice that this is just a `String` value! That's because `String` and `[Char]` are aliases for the same type. It is important to realize that the type of `'L'` is Char, while the type of `"L"` is `[Char]` (list of `Char`).

In order to specify sequences of values, two other methods are used. The notation `[5..22]` specifies the list of numbers from 5 to 22 inclusive. An arithmetic sequences of numbers is specified in a similar manner; `[3,7..24]` specifies the list containing 3, 7, 11, 15, 19, 23. In a similar manner `['c'..'t']` is the list of letters from `'c'` to `'t'`, while `['c','f'..'t']` is the list starting with `'c'` and including every third letter following – i.e., the list `"cfilor"`. Try out a few of these list expressions. This technique can be useful for finding where characters sit in the ASCII sequence. [Try `['Z'..'a']`, for example.]

An interesting characteristic of *Hugs* is that it is possible to specify infinite lists. The two methods for specifying sequences of numbers can be slightly modified to specify infinite sequences and infinite arithmetic sequences. `[5..]` is the (unbounded) sequence of numbers starting with 5 and `[3,7..]` is the (unbounded) arithmetic sequence starting with 3 and then every fourth number following. Try entering some infinite list descriptions at the *Hugs* prompt and see what happens – be ready to press `cntl-c`! What do you expect will happen if you try the same technique with characters? Test out your answer.

3.4 List representation

Lists in *Hugs* are stored internally just like linked lists in Pascal or Java, but the programming environment and language hide the linked-list behind the function *cons*, denoted '`:`'. So lists are seen as built from the empty list by successively adding elements using the cons function. The *Hugs* expression

```
[1, 2, 3, 4]
```

is an abbreviation for the expression

```
1 : (2 : (3 : (4 : [])))
```

It is important to understand this structure. Try some of the following expressions at the *Hugs* prompt.

```
1 : []
1 : (2 : 3)  --  why does this one give an error?
1 : 2        --  same thing?
1 : [2..5]
[2..5] : 1   --  why doesn't this work?
```

So every non-empty list has two parts: the element at the head of the list (called the head) and the rest of the list, referred to as the tail. *Hugs* supplies two functions, not surprisingly called `head` and `tail`, which take a list as the only argument and return either the first element of the list (`head`) or the list without the head element (`tail`). It

is important to note that these two functions are defined only if the list is *non-empty*! Try applying `head` and `tail` to a few lists (including the empty and 1-element lists) to see how they work. In fact try some of these expressions in the *Hugs* system.

```
head [3,4,8,6]
head [3,8..100]
head (tail [5,2..(-10)])  --  why are the parentheses needed for -10?
head []
tail []
head [5]
tail [5]
tail (head [1..100]) -- what do you think this will do?  Answer first!
```

Example 4 – head reversal

The following function takes a list and reverses the first two elements. It illustrates how messy functional programming can be. Add this to your file, load it into the system and try it out. What happens if it is applied to a list with one element?

```
revHead ls = (head (tail ls)) : ((head ls) : (tail (tail ls)))
```

It is not necessary that the definition of `revHead` be so difficult to understand. The following definition takes advantage of the `where` clause, which we have seen earlier. Remember, the `where`-clause gives the programmer the ability to define new constants or functions which are useful in the definition of a function.

```
revHead1 l = e2 : (e1 : t2)
             where
                  e1 = head l
                  t1 = tail l
                  e2 = head t1
                  t2 = tail t1
```

Again, enter this definition and try it out. There are a couple of things to notice in this example. First, if the length of the list is zero or one, then there is no need to go through all this concatenation – the answer is just `l`! But the problem is not just one of efficiency. The solution above will, in fact, give an error if the length of the list is less than 2 – can you see why? (Check with your instructor to see if your answer is correct.) We can modify the solution above to yield a new, more robust solution which will give the correct answer regardless of the argument.

```
revHead l = if length l < 2
               then l
               else e2 : (e1 : t2)
             where
               e1 = head l
               t1 = tail l
               e2 = head t1
               t2 = tail t1
```

In this solution, we introduce the function `length`, another basic list function, which returns an integer representing the length of the specified list.

Here, it is important to point out that the things that look like assignment statements in the `where` clauses above are not assignment statements at all – no values are being stored. Also, there is no notion of sequence as far as run-time is concerned. All that these lines are doing is defining substitutions, and the substitutions only occur if they are needed in the calling expression. In fact, you can replace the whole definition for `revHead` by the `where`-less definition which follows.

```
revHead l = if length l < 2
               then l
               else (head (tail l)) : ((head l) : (tail (tail l)))
```

Certainly, the `where`-clause makes things much easier to understand!

Problem 7

Here is an interesting problem dealing with the run-time complexity of the `head` and `tail` functions. How many reductions are required to find the head of any list? Considering the form of a list we might expect that the number of reductions is independent of the length of the list. Do you find this? (Evaluate

```
head [1..n]
```

for `n = 10, 100, 1000`, and record the number of reductions for each case.) Now, if determining the head of a list takes a constant number of reductions, finding the tail of a list should take no more work. Try the same experiment with tail, once again recording the number of reductions. What happened?

Since that didn't work out quite the way we expected, let's try something a bit more subtle. Determine the head of the tail of the sequence of lists as above (recording the number of reductions). Now what can you conclude about the number of reductions for `tail`? Why is there a discrepancy in the results of the two tests? As a hint, try evaluating the lists used as parameters on their own; i.e., evaluate expressions of the form

```
[1..n]
```

and record the number of reductions.

3.5 Basic List Functions

In the previous section, we used three list functions – `':'`, `'head'`, `'tail'`. However, there are many other important basic functions provided by *Hugs* for manipulating lists. The most familiar of these is probably the concatenate function, denoted `'++'`. This function has the following type

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

Notice that both argument lists must have the same element type. This is because when we combine the lists the resulting list must have elements all of the same type. Figure 2 contains a summary of basic list functions. Try each of them out to see what they do.

Problem 8

1. Write a function that takes a list as a parameter and returns its argument minus its first and last elements. The function should display an error message if the list is empty, but should work otherwise.

```
Main> firstLast [1..10]
[2,3,4,5,6,7,8,9]
(506 reductions, 795 cells)
Main> firstLast [2..3]
[]
(104 reductions, 150 cells)
```

2. In a similar vein, write a function 'strip n l' that returns the argument list with its first 'n' and last 'n' elements removed. For example,

   ```
   strip 3 [1..10]
   ```

   will yield the result

   ```
   [4,5,6,7]
   ```

   Hint: use drop and take, after doing some arithmetic with the length of the list (remember the function `length`).

3. [A bit harder!] Write a function that takes two integer lists, which are in ascending order and returns a single sorted list. This is just an implementation of the merge sort algorithm. Call the function 'mrg'. DO NOT use the sort functions supplied with *Hugs*.

   When you define 'mrg' remember that there are three cases: one of the parameters can be the empty list (that's two cases) or both can be non-empty. If one of the lists is empty then the other is returned as the value. If neither is empty, then use the minimum of the two heads of the lists as the headof a list of the result of merging the remaining elements of the two lists (not just the tails, mind you).

`(:)`	`:: a -> [a] -> [a]`	adds an element to the head of a list
`(++)`	`:: [a] -> [a] -> [a]`	concatenate two lists
`head`	`:: [a] -> a`	returns the head of the list
`tail`	`:: [a] -> [a]`	returns the list without the head element
`last`	`:: [a] -> a`	returns the last element in the list
`init`	`:: [a] -> [a]`	returns the list without the last element
`take`	`:: Int -> [a] -> [a]`	returns the list of the first n elements of the list
`drop`	`:: Int -> [a] -> [a]`	returns the list without the first n elements
`reverse`	`:: [a] -> [a]`	returns the list in reverse order
`elem`	`:: a -> [a] -> Bool`	tests an element for membership in the list
`notElem`	`:: a -> [a] -> Bool`	tests for non-membership in the list
`(!!)`	`:: [a] -> Int -> a`	(alist `!!` n) returns the `n`th element of the list `alist` where n < (length alist)

Figure 2: List Functions

3.6 Polymorphism

We have all had the experience in programming of having to write two routines which are almost identical except for the types of the parameters. For example, suppose you are writing a program in which there are two different lists, one holding integers and another holding characters. You need to be able to attach new values to the head of each list. In Java , you would use generics to provide this functionality. For example:

```
<T> void AddInt(List<T> l, T elem) {
     l.add(elem);
}
```

This works because the author is explicitly saying that the function is generic on the type of the element to store in the list.

The property of a function definition working for a range of parameter types is referred to as *polymorphism* and it is one of the principle features of modern functional programming languages including *Hugs*. In fact you have already seen *Hugs*'s polymorphism at work. In the table in Figure 2 each function's type is given using type variables rather than explicit type names. For example we have

```
head :: [a] -> a
```

The `'a'` is a type variable (they must begin with lower case letters) and indicates that the function `head` is defined no matter what type is substituted for `'a'`. The same trick idea for tuples. Watch in future sections for polymorphic function definitions – they are the rule rather than the exception in *Hugs* programming.

4 Basic List Processing

It was pointed out in the last section that lists are the central data type for functional programming. For this reason a great deal of work has gone into the development of basic list-manipulation functions which have wide application. In this section we will look at several new list-manipulation functions, a new way to define functions, and some standard list description and processing techniques.

4.1 Function Definition via Pattern Matching

The most straight forward way to define functions in *Hugs* is to write a single expression in terms of the formal parameters of the function, as in

```
fact n = if n == 0 then 1 else n * fact(n-1)
```

A straight translation to *C* gives

```
int fact(int n) {
    if (n==0) return 1;
    else      return n*fact(n-1);
}
```

There's nothing surprising here. The basic action of the function is to determine whether the parameter is zero and return an appropriate value. The only problem is that these definitions do not very well match the definition we would see in a math text. Mathematically we would write

```
fact (0) = 1
fact (n) = n*fact(n-1)   for n > 0
```

In this form the functions's formal parameter is used to help break up the cases. In a sense the definition says that if the actual parameter turns out to be zero then use the first definition, and if not use the second definition. This definition by cases is more natural and is easier to understand.

In *Hugs* we can write the definition of `fact` almost exactly as it appears in the mathematical definition:

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

The *Hugs* definition only differs in not requiring the parentheses around the formal parameter. In evaluating an application of `fact` to an actual parameter (e.g., `fact 25`)*Hugs* always tries to apply the definitions in order, finding the first definition whose actual parameter matches the *pattern* specified by the formal parameter. A similar example is the following *Hugs* definition of the Fibonacci function.

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

Again, notice how closely this definition structure matches the mathematical notation usually used for defining the function.

The idea illustrated here is called *pattern matching* and is a way of defining functions by cases, depending on the structure of the data. Pattern matching is a powerful mechanism for simplifying the definition of functions. This notion of a *pattern* extends beyond simple values, as illustrated above. Here is a very typical example.

Example 5 – `head` **revisited**

Let's consider the definition of the list function '`head`'. This function, as we have seen, takes a list as its only parameter. An actual parameter can either be the empty list or a list that has a head element attached to the rest of the list, via the `:` operator. In other words, there are two cases in the definition. We use this structure to write the following definition for the function `head`.

```
head :: [a] -> a

head []       = error "Can't take head of an empty list\n"
head (h:rest) = h
```

Notice how the two parts of the definition reflect the cases we identified and how the formal parameter in each line reflects the possible structure of the anticipated actual parameter.

Now consider evaluating the expression `head [1,2,3]`. The first thing we must remember is that the expression `[1,2,3]` is a *Hugs* abbreviation; the list actually has the form

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

– remember this from our earlier discussion. Now, in evaluating `head [1,2,3]` *Hugs* will first try the pattern of the first definition line – and clearly the parameter we have is not the empty list. Second, *Hugs* will try to match the parameter to the formal parameter `h:rest`. When we look at the real form of our actual parameter we can see that the `1` matches the `h`, the colon operators match up and `(2:(3:[]))` matches `rest`.

The matches we have identified become the bindings which are used in the computation of the function value and so `head [1,2,3]` evaluates to `1`.

Write the corresponding definition for `tail` using the pattern matching technique.

As another example of this pattern matching technique, recall the definition of the function `fst` for 2-tuples

```
fst :: (a, b) -> a
fst (x,y) = x
```

According to this definition `fst` has just one parameter and that parameter is a pair. In defining the action of `fst` we have used the structure of the data (its a pair) in the formal parameter. In this definition, while there is just one formal parameter, it is as though there are actually two formal paramters, `x` and `y`. When `fst` is called the binding between these new formal parameters and the actual parameter is what you would expect – `x` gets bound to the first component of the pair and `y` gets bound to the second component. Thus, the binding of actual to formal parameters is based on the structure we know the actual parameters must have.

Problem 9

For each argument/pattern pair indicate the result of the attempted pattern match: specify the bindings which will result if the match succeeds.

Pattern	Argument	Succeeds?	Bindings
`1`	`1`		
`2`	`1`		
`x`	`1`		
`x:y`	`[1,2]`	yes	`x = 1, y = [2]`
`x:y`	`[[1,2]]`		
`x:y`	`"Bucknell"`		
`x:y`	`["Bucknell"]`		
`x:y:z`	`[1,2,3,4,5]`		
`x`	`[]`		
`[1,x]`	`[2,2]`		
`[]`	`[2,2]`		
`(x,y)`	`[1,2,3,4]`		
`((x:y),(z:w))`	`([1],"Bucknell")`		

Example 6 – `revHead` **revisited**

We can use the pattern matching technique to write a clearer definition for the `revHead` function defined earlier.

```
revHead :: [a] -> [a]

revHead []          = []
revHead [x]         = [x]
revHead (x:y:rest) = y:x:rest
```

According to this definition an actual parameter will match the first, second, or third pattern only if the actual is the empty list, a singleton list, or a list with at least two elements, respectively. Notice that no actual parameter can match more than one pattern. This need not be the case; when there is an overlap, the first pattern matched is the definition which is used.

Example 7 – **Inserting in a List**

Most interesting functions written in Haskell make use of pattern matching somehow. Another good example is the function which inserts a value in a sorted list. The type definition of the function can be written as follows.

```
insert' :: Int -> [Int] -> [Int]
```

In thinking about how to write this function remember that it is critical to think recursively. This means we need a base case and then a general (recursive) case. The simplest case is where we insert a value into an empty list. The empty list is certainly sorted and putting a new value in the list will maintain the order.

```
insert' n [] = [n]
```

Notice how we use pattern matching in the second argument to separate out the empty-list case.

Now for the more difficult general case. First, it is usual to begin a recursive case definition with the following left-hand side.

```
insert' n (x:rest) = ?????
```

`(x:rest)` just means that the actual argument will be broken down into its head value `x` and the rest of the list called `rest`. What do we do with `n`? If it is less than or equal to `x` we can just put `n` at the beginning of the list, but otherwise we must insert it further down the list – i.e., into `rest`. That's where the recursion comes in. The definition just described is written as follows.

```
insert' n (x:rest) =
        if (n < x) then n:x:rest
                   else x:(insert' n rest)
```

If the value ultimately belongs at the end of the list then the second case will continue to be used until `rest` is empty – then the first case kicks in again.

You might be wondering about the legality of the name `insert'`. Well, the name `insert'` is legal – *Hugs* allows the apostrophe to be used in identifiers. This makes it easier to use alternate forms of familiar names. In this case there is already a function `insert` included in the standard *Hugs* prelude module.

4.2 Order Matters

There is one final important point to make aobout using pattern matching in function definitions. When *Hugs* sees a function application, for example:

```
insert' 23 ls
```

It looks at the definition of `insert'` beginning with the first pattern case and continuing *in order* through all the cases until a match is found. If no match is found an error is given. But imagine we have the following definition for `insert'`.

```
insert' n [] = [n]
insert' n list = if (n < (head list))
                 then n:list
                 else (head list):(insert' n (tail list))
```

This definition works perfectly well, but if we reverse the order of the definition, giving the base case second, then the definition own't work. why? Since the two parameters are given by simple variable names, they will match any argument (remember the rule for pattern matching), so even in the case when the list is empty, the first definition case will be applied. This will fail, of course, because `head` applied to an empty list produces an error.

Try this definition in both orientations to see that the order above works, but the reverse order does not.

Problem 10

1. Assume that we use a pair to represent a rational number, i.e., `(1,4)` represents 1/4. Write a function `addRat` for adding rational numbers, where the type is

   ```
   addRat ::  (Int, Int) -> (Int, Int) -> (Int, Int)
   ```

   Use pattern matching to get the components of each argument. Be sure to check for zero divisors. This can be done in the pattern matching style as well.

   A bit of a hint – use the following code to detect one of the divide-by-zero error cases.

   ```
   addRat (a, 0) (x, y) = error "Can't divide by zero!"
   ```

   Actually, there is an even better alternative. When using pattern matching it is sometimes the case that you want a pattern matched, but don't care about the bindings which would result. The following definition illustrates the use of the underscore as a wildcard to indicate "match but don't bind".

```
addRat (_, 0) (_, _) = error "Can't divide by zero!"
```

In this case, the 0 must be present in the first actual parameter but otherwise we don't care about the values.

Don't forget to reduce your rational number to its canonical form after adding (you might want a separate function for that)!

2. Go back and look at the function you defined to merge two sorted lists (Problem 3.5.3. This can also be conveniently rewritten using pattern matching – do it.

∎

4.3 Functional Loops

Recursive function definition is the basic mechanism for repetition in *Hugs*. You have experienced recursion in previous courses and in the previous problem set. Remember that recursion works based on two fundamental notions: a base case and a recursive pattern, where the next value is determined based on previously calculated values. *Hugs*'s pattern matching mechanism plays right into the recursive definition style.

One of the interesting things about functional programming languages is that, since there are no statements as such, the usual `while` or `for` loops don't exist. Instead, recursion is used whenever repetition is called for.

Example 8 – finding the maximum

Suppose we want to find the largest value in a list of integers.

```
myMax :: [Int] -> Int
```

We can scan through the list keeping track of the largest value we have found so far. At the end of the list we should have the maximum value. This algorithm is great if we are working in Java, but it is not so straight forward to translate it into a *Hugs* function. We need to think recursively, not sequentially.

If we want a recursive solution we have to ask two questions: what is the base case and what is the recursive case (i.e., at some point in the middle of the process)?

Base:
For a singleton list `[x]`, `x` is the maximum.

Recursion
For a longer list, if the head is greater than the maximum of the rest of the list (i.e, the tail) then the head is the maximum; otherwise the maximum of the tail is the maximum of the whole list.

Making use of pattern matching we can write the definition of this function as follows.

```
myMax :: [Int] -> Int

myMax []       = error "The empty list has no maximum!\n"
myMax [a]      = a
myMax (a : ls) = if (a > (myMax ls)) then a else (myMax ls)
```

This is a natural definition. But we can simplify things a bit by making use of a `where` clause.

```
myMax :: [Int] -> Int

myMax []       = error "The empty list has no maximum!\n"
myMax [a]      = a
myMax (a : ls) = if (a > m) then a else m
                 where
                      m = myMax ls
```

[A reminder. This last implementation makes use of a `where` clause. The definition within the where clause is for substitution purposes – it is not meant as an assignment statement.]

∎

In this example you might notice one important thing. In working sequentially we take each element as it comes and then do something with the next. Notice that we are repeating the same activity on each repetition. With recursion what we have to do is call the same function on each repetition. Let's look in detail at how `myMax` would be applied to a small list – in particular we will unpeal the recursive calls according to the pattern matching.

```
myMax [5, 3, 9, 1]  -->
      myMax (5 : [3, 9, 1]) :: need myMax [3, 9, 1] first
myMax [3, 9, 1]  -->
      myMax (3 : [9, 1])    :: need myMax [9, 1] first
myMax [9, 1]  -->
      myMax (9 : [1])       :: need myMax [1] first
myMax [1] = 1
      myMax (9 : [1])       = if (9 > 1) then 9 else 1 --> 9
      myMax (3 : [9, 1])    = if (3 > 9) then 3 else 9 --> 9
      myMax (5 : [3, 9, 1]) = if (5 > 9) then 5 else 9 --> 9
myMax [5, 3, 9, 1] = 9
```

What really happened here was we worked our way to the end of the list, applied to base case to stop, and then worked our way out of the recursion. This is standard. If you had any trouble with the problems in the last section you might return them now and see this discussion has helped.

Problem 11

Try this technique on the following two functions. For each function, write out the answers to the recursion questions: "what is the base case?", " what is the recursive case?" as illustrated in Example 4.3. Work out a small example first (as above) to set your recursive intuition [recursion intuition!] – but you don't have to hand in the example.

1. Write a function `sorted`, which takes a list of integers as parameter and returns a Boolean value which indicates whether the list is in ascending order.
2. Write a function `mySum`, which takes a list of integers and returns the sum of the values in the list. Assume that the sum of an empty list is zero.
   [You may not use the library function called `sum` in your solution.]

∎

4.4 List Mapping

The functions developed in the previous section are characteristic of many list-processing functions, where the partial result is folded into the next partial result. In the next section we will look at special functions in the standard prelude which facilitate this type of list processing. In this section we are interested in a simpler type of processing – in which each value in a list is treated in the same way. This process of modifying each value in a list in the same way is called *mapping*.

Example 9 – convert to upper case

As our first example of mapping, suppose that we must insure that all characters in a string are upper case. We could write a recursive function (in the style of the last section) which does the conversion as follows.

```
cap :: [Char] -> [Char]

cap [] = []
cap (a : ls) = (toUpper a) : (cap ls)

Prelude> cap "abcdEF"
ABCDEF
(63 reductions, 82 cells)
```

[The `toUpper` function is defined in the `Char` module. You will need to import this module to call the function.] This is fine and, in fact, easy to handle. But suppose that you also need another function which converts to lower case. It would certainly be nice to take advantage of the similarities of the two functions.

In the *Hugs* standard prelude is a function `map` which makes this mapping easier. What `map` does is take a list as one parameter, a function as the other parameter and then applies the function to each element of the list. We would apply `map` (rather than cap) to the problem above as follows

```
Prelude> map toUpper "abcdEF"
ABCDEF
(59 reductions, 83 cells)
```

(It's interesting that the number of reductions is less with `map` than with our definition of `cap`!)

This function `map` is not like functions we have seen before. It takes another function as an argument. What is its type?

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

While this looks a bit messy, if we read from left to right we can clearly see what is happening – especially since we know how parentheses are usually used. According to this definition (which is polymorphic, notice) there are two parameters. The second parameter is a list of elements of type `'a'`, the function's result is a list of elements of type `'b'`, and the first parameter is a function from type `'a'` to type `'b'`.

See if you can come up with the appropriate application of `map` to solve the following problems. You do not need to define functions for these problems, just demonstrate how to use `map` to get the necessary behavior. **Problem 12**

1. Convert a list of integers into a list of absolute values of the original list.
2. Convert a list of Strings into a list of integers where each integer in the result is the length of the corresponding string in the argument.

One final point in this section is to look at the definition of our new function. It seems that a function with such flexibility might be a bit complicated. But it actually looks very much like the `cap` function above.

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

map f []       = []
map f (a : ls) = (f a) : (map f ls)
```

A couple of hopefully obvious points. First, the parameter `'f'` is of type `'a -> b'`, i.e., it is a function which takes one parameter of type `'a'` and returns a value of type `'b'`. The second point follows from the first — that is that `'(f a)'`, then, must be of type `'b'`. So `'(f a)'` is an element of type `'b'` which is being attached to a new list `'(map f ls)'` of elements of type `'b'`.

4.5 List Folding

In this section we will investigate more closely those list-processing functions, where the partial result from part of the list is folded into the next partial result. Good illustrations of this can be seen in the functions `sum` and `prod`, which add or multiply together all the integer values in a list. Again, this is a situation where the processing is the same in each case; the only difference is the function (multiplication or addition) used to do the processing.

If we look at the way we sum a list of integers we will get a better idea about this folding operation. If we want to sum the values in the list `[1,2,3,4]` we would get the following:

```
4 + (3 + (2 + (1 + 0)))
```

The 0 is there to get things started (its the additive identity). This may look backwards, but since we begin inside the innermost parentheses, it is really correct. What's really happening here. We add 1 to 0 and save the result for the next step. We take the result of the previous step (1) and add that to the second value in the list. We take the result of this step (3) and add that to the next value (getting 6) and add that result to the final value in the list getting 10. Each subsequent result is folded back in as an argument to our basic function, in this case addition.

Let's say we want to define a function which will carry out this operation if we pass it the addition function as a parameter — we'll call this function `fold`. We would apply it as follows:

```
fold f i ls
```

with `'f'` being a binary function, `'i'` the initial value for second parameter to `'f'`, and `'ls'` the list over which `'f'` is to be folded. At this point we won't be too explicit about the type of `fold`, but if we assume we are to do things like add or multiply integer values in a list, then we have the following.

```
fold :: (Int -> Int -> Int) -> Int -> [Int] -> Int
```

Wow! Three parameters: a binary function, an initial value, and a list. The function value returned (in this case) will be an integer.

One qualification we must make before we try this new function. There is no function in the standard prelude called `fold`; there are two functions called `foldr` and `foldl`, for folding from the right end or the left end. There are some other differences between these two: `foldr` can work on infinite lists if the folding function does not need to look at all elements of the list; `foldl` will not work on infinite lists, but is tail recursive, and can be run very quickly. Because multiplication and addition are commutative it doesn't matter which we use, but in the following example we will see a situation where it matters.

Example 10 – **sum, product**

Let's see how we apply what we have seen to determine the sum and product of a list. The first thing to note, since the fold functions take a binary function as parameter it is important to distinguish between infix and prefix functions. The fold functions require that the function arguments be prefix forms. This means that we cannot use `'+'` and `'*'`, but rather must use `'(+)'` and `'(*)'`. With that said, we can write expressions for the sum and product of values in a list.

```
? foldr (+) 0 [1,2,3,4]
10
(10 reductions, 25 cells)
? foldr (*) 1 [1,2,3,4]
24
(11 reductions, 25 cells)
```

∎

Example 11 – **insertion sort**

Now to see more of the power and flexibility of folding. Suppose that we consider the insertion sort algorithm. In this sorting technique we begin with an empty list and then repeatedly take values from the list to be sorted and insert them in the new (originally empty) list. Each time we do an insertion the resulting list is used in the next step for another insertion. This is clearly a folding operation – we just have to determine the various parameters to the fold opertion.

function: The function is clearly the insertion function. It takes two parameters (so it is binary), a list item and a list and produces a list as result.

```
insert ::  a -> [a] -> [a]
```

initial value: The initial value with which `insert` should start (for its second argument, remember) is the empty list.

list: The final parameter is, of course, the list to be sorted.

The following results.

```
? foldr insert [] [2,4,3,6,1,5]
[1, 2, 3, 4, 5, 6]
(44 reductions, 119 cells)
```

Problem 13

Suppose we have a character string which we know to consist completely of digits. You are to define a function which will take this string (a list of characters, remember), remove leading zeros, and then return the `Int` value which the string represents.

```
convert2Int :: [Char] -> Int
```

You will probably want to write a couple of functions. First, define a helper function to do the zero stripping. Then define a function which does the conversion – HINT: `x*10 + y`!

It should be pretty clear that this problem requires the application of a folding function. But does it matter which way you fold? Does it matter at which end of the list you start? Of course, the other tricky question is, if you apply a folding function, what binary operation do you apply through the folding function?

4.6 Higher-order Functions

This section is just a wrapup for the previous two. In these two sections we have looked at two functions, `map` and `fold`, which have the interesting property of taking as parameters other functions. Functions which take other functions as parameters are referred to as *higher-order functions*. We will see as we go along that looking for already defined or new higher-order functions can make the programming process much simpler and consequently easier to understand. Watch for more on this topic in future sections.

5 More List Processing

5.1 A New Kind of Function Definition

It was pointed out early in this tutorial that the syntactic form that function application takes in *Hugs* is not what we are used to either from mathematics or other programming environments. The function `drop` defined in Figure 2, for example, takes two parameters; the correct syntax for its application is illustrated in the following expression:

```
drop 2 [1,2,3,4]
```

In a more conventional language we would expect to see

```
drop (2, [1,2,3,4])
```

Why this form rather than the more usual one? Let's see if we can get an answer to this question.

Startup the *Hugs* environment and execute the following expression at the prompt.

```
drop 2 [1,2,3,4]
```

Not surprisingly the answer `[3,4]` is returned.

But this new notation does beg the question of what happens if we leave off the last parameter. *Hugs* has a nice feature which, when given an expression, will report the type of the expression's result. The command is the `type` command and is issued as either

`:type`, or `:t` for short.

Let's try this out on our problem. First let's see what *Hugs* reports for the type of `drop` alone.

```
Prelude> :t drop
drop 2 :: Int -> [a] -> [a]
Prelude>
```

Fine, that's what we believe we have, a function which takes two arguments, an integer and a list, and returns another list (like the original). But now try the following.

```
Prelude> :t drop 2
drop 2 :: [a] -> [a]
Prelude>
```

Apparently *Hugs* thinks that `'drop 2'` is a function itself, taking one list parameter and returning another list (of the same kind). Let's try it out on its own. Enter the following command and you should get the accompanying result.

```
Prelude> (drop 2) [1,2,3,4]
[3,4]
Prelude>
```

Investigate this type command a bit further. Determine the type of each of the following expressions.

```
drop
length
map
```

Here's a simple application of what we have just learned. Suppose we want to write a function which will take a list of strings and return the same list except with the first two letters dropped from each string – remember that a string is just the type `'[Char]'`. We can define this function using `map` and a function which drops the first two elements from a list – like `'drop 2'`. We define our new function as follows.

```
listDrop2 :: [[a]] -> [[a]]
listDrop2 ls = map (drop 2) ls
```

Try it out on a list of string constants and see what happens.

Actually, since the one argument of `listDrop2` appears as the final "argument" on the right hand side, we can leave it off entirely, as follows.

```
listDrop2' :: [[a]] -> [[a]]
listDrop2' = map (drop 2)
```

Trythis definition in the same way you tested `listDrop2`. [OK, you don't have to turn them in, but do it anyway to reinforce the ideas – it will just take a minute.]

5.2 Currying and Operator Sections

There is a principle here. Let's consider a more general case: let a function `F` have the following type definition:

```
F :: T1 -> T2 -> T3 -> T4 -> T
```

Then, if `a1, a2, a3, and a4` are expressions of type `T1, T2, T3, and T4`, respectively, the following expressions have the following types:

```
F a1          :: T2 -> T3 -> T4 -> T
F a1 a2       :: T3 -> T4 -> T
F a1 a2 a3    :: T4 -> T
F a1 a2 a3 a4 :: T
```

This general method of representing function applications is called Currying. [This method is named for the late Haskell Curry, a long-time member of the Penn State mathematics department. His early work in the theory of computation provides the mathematical basis for all functional programming languages.]

One snag in this method is providing a convenient syntax for Currying functions which are used as infix operators, such as `*` or `+`. The method used by most functional languages is called an *operator section*. The following annotated examples should make the syntax clear. [The parentheses in these examples are necessary!]

```
(* 2)    -- "doubling" function
(+ 4)    -- "add 4" function
(1 /)    -- "reciprocal" function
(/ 4)    -- "one fourth" function
(> 5)    -- "bigger than 5?" function
(== ' ') -- "equals blank" function
```

Use *Hugs* to determine the type of each of these functions. Try the "reciprocal" function along with `map` to determine the inverse of each value in a list of integers (no 0's please). Don't forget to include the parentheses around the section – they are an integral part of the section syntax. These operator sections will be important in later lessons.

5.2.1 Important Aside

When you check out the types of the operator sections above you will notice a strange bit of syntax...

```
Prelude> :t (2 *)
(2 *) :: Num a => a -> a
Prelude> :t (> 5)
flip (>) 5 :: (Num a, Ord a) => a -> Bool
```

Here's a quick explanation of the notation. *Hugs* has a class structure which is used to define the various basic types. In fact, an instance of a *Hugs* class is a type, not an object as you'd expect in Java. In the notation above,

```
'Num a => a -> a'
```

is read "assuming `'a'` (a type variable, remember) is an instance of the `Num` class, `(2 *)` has type `'a -> a'`." In other words, if you apply `(2 *)` to an instance of `Num` (such as an integer) then `(2 *)` will return a result of the same type; if you apply it to something which is not a `Num` then you will get an error.

What is the `Num` class? The `Num` class is primarily defined as a set of operations – any instance of `Num` must supply implementations for the operations. In particular, all the numeric types are instances of `Num`. The class `Ord` mentioned in the second example, defines ordering and its definition looks like this.

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y      = not (x==y)

class (Eq a) => Ord a where
    compare                :: a -> a -> Ordering
    (<), (<=), (>=), (>)   :: a -> a -> Bool
    max, min               :: a -> a -> a
```

Notice that each of these class definitions takes a type variable as an argument. The type variable becomes the type in the type specifications for the operations. This is a similar mechanism to the the 'template' in C^{++}.

Okay, you got more than just `Ord`, but to define `Ord` we also need a definition of the `Eq` class – the simplest of all *Hugs* class definitions. Notice that in each definition, a sequence of functions is defined, with each function associated with a type. In the case of `Eq`, the function `'=='` is not defined, but `'/='` is defined – in terms of `'=='`. For a type to be an instance of `Eq`, a definition for `'=='` will have to be supplied.

Now looking at the second type above (for `(> 5)`), we see that in this case the type variable `'a'` is restricted to being an instance of **both** `Ord` and `Num`. From this we learn that a type can be an instance of more than one class – a kind of multiple inheritance.

To apply these ideas: Suppose you want to write a function which checks to see if a list is in order. We can define the function in a straightforward recursive way.

```
isSorted :: [a] -> Bool
isSorted []       = True
isSorted [x]      = True
isSorted (x:y:ls) = if (x < y)
                    then isSorted (y:ls)
                    else False
```

Now, what happens when we try to enter this into *Hugs*? Put the definition in your file and load the file. You should see something like this.

```
Prelude> :l test
Reading file "test.lhs":
Type checking
ERROR "test.lhs" (line 2): Declared type too general
*** Expression    : isSorted
```

```
*** Declared type : [a] -> Bool
*** Inferred type : Ord a => [a] -> Bool
```

We could avoid this problem by making our original definition specify, say, integer lists rather than generic lists. The tip off to *Hugs* that something is wrong is the use of the operator `'<'` – this is defined for instances of `Ord`! So we can solve our problem by indicating that type restriction. "Assuming `'a'` is an instance of `Ord`, ..."

```
isSorted :: Ord a => [a] -> Bool
isSorted []       = True
isSorted [x]      = True
isSorted (x:y:ls) = if (x < y)
                    then isSorted (y:ls)
                    else False
```

5.3 Simplifying Function Definitions

Currying is an important feature of *Hugs* and has interesting consequences to function definitions. As a first example, let's return to the sorting function defined with `foldr` at the end of the previous section. In that example we saw the following expression evaluation:

```
? foldr insert [] [2,4,3,6,1,5]
[1, 2, 3, 4, 5, 6]
(44 reductions, 119 cells)
```

If we use this expression as a basis to define a sorting function, how should we proceed? First, the type of a sorting function should clearly be

```
mySort :: [a] -> [a]
```

Let's apply the Currying principle to the expression above.

```
foldr insert [] [2,4,3,6,1,5] :: [Int]
foldr insert []               :: [Int] -> [int]
```

Now this looks promising. It would seem that the second expression can serve as the definition of our sorting function.

```
mySort :: [a] -> [a]

mySort = foldr insert []
```

Notice that we haven't specified any parameters in this function definition. But that's OK since the type of `foldr insert []` implies there must be one parameter.

Using this technique (and the operator sections of the previous section) we can define many new functions:

```
mySort = foldr insert []   -- of type [a] -> [a]
double = (* 2)             -- of type Int -> Int
square = (^ 2)             -- of type Int -> Int
emptyL = (== [])           -- of type [a] -> Bool
```

Finally, we can apply some of these new functions via the `map` function as follows:

```
doubleL = map (* 2)        -- of type [Int] -> [Int]
squareL = map (^ 2)        -- of type [Int] -> [Int]
```

Problem 14

Recall the formula for computing the standard deviation of a list of values:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2 - \left(\frac{\sum_{i=1}^{N} x_i}{N}\right)^2}$$

Define a function `stdDev` with the following type.

```
stdDev ::  [Float] -> Float
```

You may want to use earlier functions you have written or that are provided by the system – `map` and/or `foldl` or `foldr`.

YOU MAY NOT USE THE FUNCTION `sum`!

5.4 Simple Comprehensions

One of the most fascinating of *Hugs* features, once again a feature shared with most modern functional programming languages, is called *list comprehension*. This technique is a translation of set comprehension which you are familiar with from many math courses. Set comprehension defines the elements of a set by giving a list of conditions satisfies by each element of the set. For example $\{x|x < 5, x >= 0\}$ specifies a set containing the numbers between 0 and 5, including 0.

A list comprehension, then, is an expression which defines the elements in a list by giving defining expressions which are satisfied by each element of the list. Here is an annotated list of *Hugs* examples. Notice the implied syntactic structure.

```
[ x | x <- [1..10]]   --  This is a long-winded description of [1..10].
                      --  This illustrates the important 'x <- ls'.
                      --  notation - it says "x comes from the list ls"
                      --  Read this expression as "the list of elements
                      --  x such that x comes from the list [1..10]".

[ (x, x*x)
     | x <- [0..] ]   --  This defines the set of ordered pairs where
                      --  the second component is the square of the first.
                      --  In other words, this is the definition of the
                      --  square function on the non-negative integers.
                      --  Notice, this is an infinite set!

[ (x, y)
     | x <- [2,3,4],
       y <- [5,6]  ]  --  This is [2,3,4]x[5,6] - i.e., the Cartesian
                      --  product of [2,3,4] and [5,6]:
                      --  [(2,5),(2,6),(3,5),(3,6),(4,5),(4,6)]

[ (i,j)
     | i <- [1..10],
       even i,
       j <- [i..10],
       odd j        ] --  This evaluates to the list
```

```
                    --  [(2,3), (2,5), (2,7), (4,5), (4,7), (6,7),
(6,9), (8,9)]
```

Example 12 – **quicksort**

List comprehension gives the programmer a powerful tool for expression computation. See if you can remember how to implement (say in *C*) the quicksort algorithm. [Remember that quicksort is the one where you select a pivot element, and then break the list into two parts: those values less than the pivot and those values greater or equal to the pivot. And you sort each of those list (recursively).] I'll give you 5 minutes to implement and test one.

...Time passes...

How'd you do? Are you finished yet? Well, if *C* had list comprehension you might have been able to do it more quickly. Let's review the algorithm. We choose a pivot element (let's use the head of the list) and then rearrange the list so the values less than the pivot come first, the values greater than or equal to the pivot come last and the pivot goes in the middle. Then we sort separately the list of elements less than the pivot (we can describe that with a list) and the list of elements greater than or equal to the pivot (again we can use list comprehension), and then we concatenate the two lists. But that's trivial using list comprehension.

```
quicksort []     = []
quicksort (y:ls) = (quicksort [ x | x <- ls, x < y]) ++
                   [y] ++
                   (quicksort [ x | x <- ls, x >= y])
```

Wow! Four lines is all it takes (without the type definition). But more important than the length is the clarity. This function specification makes quicksort understandable. Notice the use of simple pattern matching!

Example 13 – **concatenate**

If you want to concatenate together all the lists in a 'list of lists' then the function type can be described as follows:

```
concatenate :: [[a]] -> [a]
```

We could certainly define this function by using pattern matching and recursion, but is there another more concise way? Well, what we want to do is to take each element from the original list and then from each of those elements extract the elements. This can be done with the following definition.

```
concatenate xss = [ x | xs <- xss, x <- xs]
```

Again, a very concise and clear definition. If you write down the obvious recursive definition (referred to initially) you will see that this conciseness is not at the expense of clarity.

Problem 15

1. Assuming that a list is ordered, write a function `insert` which takes a list and a new element and returns the list with the element inserted in the correct position in the list. Use list comprehension for your solution. [Hint: Look at the quicksort function for inspiration.]

2. Remember the function `map`? You are to pretent that there is no `map` function provided in Hugs but that you need one for a series of functions you want to write. Write your own version of `map` – but you must implement it using list comprehension – call your function `myMap`, so it doesn't class with the existing function name.

3. Write a function `factors` which takes an integer parameter and returns the list containing the parameters prime divisors. Do this in two steps.

   - Write a list comprehension for the list of prime numbers.
     [Hint: You can use the Sieve of Eratosthenese from class if you wantt!]
   - Use the previous result to implement the function `factors`

   .

6 User Defined Types

One thing you should have learned in your programming courses is that in designing a program it is always a good idea to start by figuring out what data is to be manipulated. This strategy works whether you are using a traditional language like Pascal, a so called object-oriented language like Java, or a functional language like *Hugs*. *Hugs* provides three basic mechanisms for defining data types:

- built-in type constructors for tuples and lists, which we have seen in previous lessons,
- type aliases, and
- algebraic types for defining new type structures.

In this section we will focus on type aliases and algebraic types.

6.1 Type Aliases

A *type alias* is a name which can be used in place of a specified *type expression*. Here is an example. Earlier we discussed an example in which pairs of integers were to represent rational numbers. In *Hugs* we can define a type alias `Rat` and utilize it in the following way.

```
type Rat = (Int, Int)

addRat :: Rat -> Rat -> Rat

addRat (n1,d1) (n2,d2) = ...
```

In this case '`(Int, Int)`' is a type expression for which we are defining the alias `Rat`. Notice the use of the keyword `type`, which must appear as the initial token in an alias definition. The one restriction which is placed on type aliases is that the name must begin with an upper case letter. One type alias which is part of the standard prelude, which is always loaded when *Hugs* runs, is the string type:

```
type String = [Char]
```

While this aliasing mechanism is very useful for improving program clarity, it doesn't allow us to expand our collection of types beyond what is possible with the built-in constructors.

6.2 Algebraic Type

A type alias is a name for a combination of known types with known type constructors – a type constructor being the notation used to specify an ordered pair or triple:

```
type Something = (Int, Char, Float)
```

A type alias does not introduce any new data values.

An algebraic type, on the other hand, is one in which the basic values of a new type are defined by the user. Notice that for the `Rat` and `String` types we just used types which are already defined in *Hugs*. The simplest and probably most familiar of this category of types is the enumerated type, in which the specific values of the type are specified. The type `Bool` is an enumerated type defined defined in *Hugs* as

```
data Bool = True | False
```

Easy enough. Notice that the type name and the constituent values all start with an upper case letter and are separated by the '`|`' symbol which can be read *or*. Thus, a value of type `Bool` can have the value `True` *or* the value `False`. Also notice that in defining a new type, rather than a type alias, you must use the key word `data`. Here's another example with colors.

```
data Color = Red    |
             Orange |
             Yellow |
             Green  |
             Blue   |
             Indigo |
             Violet
```

This is common layout for such definitions. One more point of notation: the new identifiers defined as values of the algebraic type are called *type constructors*.

But the algebraic type facility can do considerably more than just define enumerated types. Using algebraic types we can define, for example, an alternative to the `Rat` type defined above. Here it is.

```
data Rat = Quot Int Int

addRat :: Rat -> Rat -> Rat

addRat (Quot n1 d1) (Quot n2 d2) = Quot n d
       where
            n = n1*d2 + n2*d1
            d = d1*d2
```

The new constructor `Quot` takes two integer arguments and binds them together into a new kind of data value. As the example shows we can use the form of the data value for pattern matching purposes. Now in fact, this new type, `Rat`, is really just a syntactic reshuffling of the ordered pair – i.e.,

`Quot x y` $\equiv (x, y)$

The advantage of the new form is that when doing pattern matching, for example, the nature of the data is explicitly represented by the name of the constructor. If we simply use an ordered pair, the connection between the data representation and its use is lost.

Another example. Suppose that we are to implement a program which will manipulate coordinate data for 1-, 2-, and 3-dimensional systems. We can use algebraic types to define a type `Coord` as follows.

```
data Coord = Point1D Float |
             Point2D Float Float |
             Point3D Float Float Float
```

In this definition we are saying that there are three possible data-value forms – each one is defined with its own type constructor. Using this type definition we can write a general function for finding the distance between two points. Notice how pattern matching makes the function definition easy.

```
distance :: Coord -> Coord -> Float

distance (Point1D x1) (Point1D x2)
    = abs (x1 - x2)
distance (Point2D x1 y1) (Point2D x2 y2)
    = sqrt ( (x1 - x2)^2 + (y1 - y2)^2 )
distance (Point3D x1 y1 z1) (Point3D x2 y2 z2)
    = sqrt ( (x1 - x2)^2 + (y1 - y2)^2  + (z1 - z2)^2 )
distance p1 p2
    = error "The two points have incompatible dimensions."
```

The examples we have seen of algebraic types are just more readable versions of tuples. In this last example of coordinates for three different dimensions we stepped a bit beyond tuples by using the algebraic type facility for

making a union of types. But the most important application of algebraic types is for recursive types. We will see these in a later lesson when we investigate implementations for lists and trees.

6.3 Type Properties

When we define an algebraic type remember that we are defining values for a new type. The new values are embodied in the constructors of the new type. There are two questions that arise because of these constructors.

6.3.1 Making a new type an instance of a class

First, can we use new types in ways we are used to using already existing types. For example, can we compare values of these new types? In other words, is it possible for an algebraic type to be an instance of an existing class – e.g. `Eq` or `Ord`? The answer is yes and the instance can be established in two ways.

Consider the type `Rat` defined above. We would like to be able to determine whether two values are equal (the `Eq` class) or to compare the order of two values (the `Ord` class). Since the `Ord` class is defined using the `Eq` class, we need to make `Rat` an instance of each of them. We do so by giving function definitions for the operations of the class for the new type values – we are really defining overloaded operations.

```
data Rat = Quot Int Int

instance Eq Rat where
  (Quot a b) == (Quot r s) = (a*s == b*r)

instance Ord Rat where
  (Quot a b) > (Quot r s) = (a*s > b*r)
  (Quot a b) < (Quot r s) = (a*s < b*r)
  a >= b = (a > b || a == b)
  a <= b = (a < b || a == b)
```

With these definitions you can compare values of type `Rat`. This general strategy works for any user defined type.

Enter these definitions into a test Haskell file and try out the following expressions:

```
(Quot 1 3) < (Quot 2 5)
(Quot 3 6) == (Quot 1 2)
(Quot 4 7) <= (Quot 3 5)
Quot 4 7
```

Now when you try that last one you will get an error – something about "Cannot fine "show" function..." This error occurs because the *Hugs* system doesn't know how to display these new values for `Rat`.

6.3.2 Displaying values of a new type

When you define a new data type it is very likely that you will want to display values of the type. How you do this depends on exactly what you want to appear on the screen. For example, if you define the following type

```
data Cplx = Complex Float Float
```

then when the system displays a value of type `Cplx` you could expect to see either

```
Complex 3.0 4.0)
```

or, perhaps even better,

```
3.0 + 4.0i)
```

which gives a representation closer to what we expect to see in a mathematical setting.

To accomplish the first version is quite easy, and is the strategy which you will often use. The second requires code similar to that given in the previous section, because it involves defining an instance of a class known as `Show`. To define an instance of `Show` it only requires that you define an overloaded version of the operation

```
show ::  a -> String
```

Here is an instance for `Rat`.

```
data Rat = Quot Int Int

instance Show Rat where
  show (Quot a b) = (show a) ++ "/" ++ (show b)
```

Enter this last one into your test file and try the expression which caused an error in the last section.

Now there is an alternative which is often satisfactory. The idea is to let *Hugs* figure out for itself how to display the values in an orderely way. We tell *Hugs* to figure out how to display values by appending a statement to the end of the type definition. Here it is for `Cmplx`

```
Data Cplx = Complex Float Float
            deriving (Show)
```

You can get the implication – *Hugs* will "derive" an instance of `Show` for us. In this case *Hugs* will use the simplest mechanism for displaying values – that is by just displaying them as they appear in definitions. Try a similar thing for your `Rat` definition and try it out on the value you just tried above. Be sure to "comment out" the instantiation of `Rat` which you just did. You should see something along these lines.

```
Main> Quot 3 5
Quot 3 5
(47 reductions, 118 cells)
```

Problem 16

1. Return to problem 3.3.1 in Section 3. Use the algebraic type for complex (imaginary) numbers discussed above and redo your solution to part 2 of the problem making use of the new type – call it `Cplx`. Just so you get off on the right track, notice that you should define the type of the function `roots` as follows.

   ```
   roots ::  Int -> Int -> Int -> (Cplx, Cplx)
   ```

   Be sure that your definition of `Cplx` is made an instance of `Show` by redefining the operation `show` as illustrated above. `show` should display a `Cplx` value in the form `"a + bi"`, where "a" and "b" are the two `Float` components.

7 A Program Example

In this section we will investigate writing a program which makes extensive use of the ideas from the previous lesson on types. The program simulates a very simple calculator. The calculator will have a memory of configurable size and a set of 6 operations – add, subtract, multiply, divide, mod, and power (i.e., raise a value to a specified power). The program will take a sequence of calculator commands and display the final content of memory. Each calculator command has an opcode, an integer value indicating the memory location involved, and an integer value. For example, the command `'Div 3 15'` would take the value at location 3 in memory and divide it by 15 and put the result at location 3 in memory.

7.1 Data Types for the Calculator

In this program there are three data types of interest.

- Memory is a an array of integer values.
- A program is a sequence of calculator commands.
- A calculator command has an opcode, a memory address, and a value operand.

The first two of these types are easy to define since both can be represented as lists: memory will be of type `[Int]` and a program will be of type `[Command]`, if we first define `Command` to be the type of the commands for the calculator. For the first two type we have simple applications of the type alias:

```
type Memory  = [Int]

type Program = [Command]
```

The last alias presupposes a definition for `Command`. Create a new file to contain the calculator program and enter these two type aliases.

Since the program command combines several values into one command, we will make use of an algebraic type for its definition. Here is the type definition we need for our calculator commands.

```
data Command = Add Int Int |
               Sub Int Int |
               Mul Int Int |
               Div Int Int |
               Exp Int Int |
               Mod Int Int
               deriving (Show)
```

Notice that there are six type constructors and with each constructor we associate two other entries, both of type `Int`. The following are all values of type `Command`.

```
Add 2 6
Exp 0 3
Sub 6 10
```

Before proceeding, enter this new type definition into your new program file.

7.2 Program Structure

With the basic types for the program identified, we should be able to sketch out a basic structure for our program – remember, though, that this just means figuring out what functions to define. To begin with, the user of the program

will expect to supply an integer value (the number of memory locations) and a list of commands (the program), and to see displayed the contents of memory when all commands have been applied. But this can be easily described as the following function.

```
calculate :: Int -> Program -> Memory
```

[We allow a bit of a trick here. Since the *Hugs* system displays the resulting value returned by an expression, we can just have the final memory value returned and have the system automatically display that.]

Now for the interesting bit. What is it that the function `calculate` is supposed to do when called? It would seem that it should construct an initial memory (let's say filled with 0's), and go through the list applying the commands and using the resulting memory for applying the next command. This should sound very much like a folding operation — it is!

If we work in the usual style, when we make a list of commands, it will be the one at the head of the list which is to be done first. This means that we want to fold from the left rather than the right.

```
calculate :: Int -> Program -> Memory

calculate n prog = foldl ? ?? prog
```

This is the required form for our function. We know about `foldl` and `prog`, so we must determine what should replace `'?'` and `'??'`.

Let's start with `'??'`. Since it is the memory which is going to be produced by each command execution, and since this second operand to `foldl` is the initial value for the operation, `'??'` must be an expression for an initial memory. This shouldn't be too difficult. We need a list of `'n'` 0's. A trivial sort of list comprehension will define this easily.

```
[ 0 | x <- [1..n]]
```

The dummy identifier `'x'` is used merely to help count out the number of positions required in the list. We can include this in our definition as follows.

```
calculate :: Int -> Program -> Memory

calculate n prog = foldl ? mem prog
                   where mem = [ 0 | x <- [1..n]]
```

Obviously, the `'?'` in the above definition must be replaced by a function which will perform a single calculator command. If we call this function `operate`, the we know that `operate` will take a command (from the program) and the current memory as operands and then return a new value for memory. But because we will use `operate` as an argument to `foldl`, there are restrictions on the order of the operands. Enter `':t foldl'` at the *Hugs* prompt. You should get

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

We know that in this case `'a' = Memory` and `'b' = Command`. Therefore, the type of `operate` must be

```
operate :: Memory -> Command -> Memory
```

We will turn immediately to the definition of operate, but first, here is the definition of our main function as we know it to this point.

```
calculate :: Int -> Program -> Memory

calculate n prog = foldl operate mem prog
                   where mem = [ 0 | x <- [1..n]]
```

Enter the definition for `calculate` and for the type of `operate` at this point. In the next section we will define the final two functions required for our program.

7.3 Pattern Matching Over Algebraic Types

There are two remaining problems to deal with. First, we need to be able to use the memory list as an array (i.e., get a value from a particular offset and then store a value back) and we need to be able to decode a command to apply the correct arithmetic operation.

In an imperative program we would probably have used a record type (or `struct`) to represent a command, with one entry of the record being an operation code. Decoding a command would just use a case statement over the operation code. In *Hugs* that opcode is represented by the type constructor. One nice feature of *Hugs* is that the pattern matching, which we have found so useful in defining list functions, can also be used in the context of algebraic types. This means that we can use pattern matching to define a function which will return the correct function to apply.

Before we see the function's definition, it is critical that we remember that there are actually three things which must be returned: the arithmetic operation and the two integer operands. From our earlier discussion of tuples, it would appear that we need to return a triple from this function. Here, then, is the definition of the function which will convert from the command structure.

```
convert :: Command -> (Int -> Int -> Int, Int, Int)

convert (Add a b) = ( (+), a, b)
convert (Sub a b) = ( (-), a, b)
convert (Mul a b) = ( (*), a, b)
convert (Div a b) = ( quot, a, b)
convert (Exp a b) = ( (^), a, b)
convert (Mod a b) = ( rem, a, b)
```

First, notice how the arithmetic operation is designated – we just give its prefix name. Second, consider how the pattern matching must work. When `convert` is called with an actual parameter, the form (or pattern) of that parameter is examined. *Hugs* knows that the argument must start with a constructor so it checks for that. Having matched the constructor it assigns to the formal parameters `'a'` and `'b'` the corresponding integer values in the actual parameter. Any algebraic type can be dealt with in this way; we will see other examples in later sections.

At this point write the definition of `convert` into your growing program file.

Finally we can complete the definition of the function `operate`. Remember that it takes a command and a position in memory as its arguments and returns the modified memory.

```
operate :: Memory -> Command -> Memory
```

Since `operate` returns a new value for the memory, with a new value at the designated position, apparently that return value has the same initial and final parts – its only the one position which changes. So the definition we want is something like this.

```
operate :: Memory -> Command -> Memory

operate mem comm = h ++ [v] ++ t
                   where
                        h = take n mem     -- n comes from 'comm'
                        t = drop (n+1) mem
                        v = ???
```

But how do we get the values out of the parameter `comm`? We do it through another use of pattern matching. We use a definition line as follows.

```
( op, n, a) = convert comm
```

When `convert comm` is evaluated, values for `op, 'n',` and `'a'` are determined from the pattern of the result. So our complete definition for `operate` looks like this.

```
operate :: Memory -> Command -> Memory

operate mem comm = h ++ [v] ++ t
                   where
                        h = take n mem     -- n comes from 'comm'
                        t = drop (n+1) mem
                        v = op (mem !! n) a
                        (op,n,a) = convert comm
```

Oh, what about that `'mem !! n'`? Well, that is just the *Hugs* mechanism for array access – it returns the value in `mem` at offset `'n'`.

Now you can enter this definition into your file, load the file into *Hugs*, fix any typos, and then try the following problems.

Problem 17

1. Try the following expressions:
   (a) `calculate 2 [Add 0 5, Add 1 10, Mul 1 4]`
   (b) `calculate 4 [Add 0 5, Add 3 10, Exp 1 4, Div 0 2]`
   (c) `calculate 5 [Add 0 5, Add 7 10]`
   You might want to look in the standard prelude at the definition of `'!!'` to see why you get the error message you do – at first glance it looks inappropriate.
   (d) `calculate 2 [Add 0 5, Div 0 0]`
2. Make a copy of your program file. Modify the code in the copy so that the commands in the program are executed from right-to-left, rather than from left-to-right. You will want to look carefully at the type of `foldr`.
3. To your original program add code which will trap errors: divide or mod by zero, raising to a negative power, using an illegal memory address. There may be more.

7.4 Functional Purity

Earlier there was a lot made about the difference between pure and impure functional languages. When you look at the definition for `operate` in your file you may think that passing in that whole memory array is a bit of a waste. Why not just pass in the one position and fit it back in when the function returns? The reason is that in *Hugs* there is no such thing as variable storage or program state. To do what was just suggested would imply that during the time `operate` is executing, the mem array would still exist; when `operate` returns the memory array is still there and we can place the new value in it. But there is no way in *Hugs* to store the memory array while `operate` executes. Therefore, we must pass the whole thing to `operate` and have `operate` pass the whole thing back, modified appropriately.

8 Hugs IO

8.1 File Input

The input we have used so far has all been supplied on the command line in the form of parameters. *Hugs*, however, provides a sophisticated mechanism which implements IO, including interactive and file IO. In a functional language this is not so straightforward, since everything is written in terms of functions without state. To setup the *Hugs* approach to IO, remember how IO is done in Java. The stream extraction operator (`'>>'`) is actually defined as a function with the following interface structure:

```
istream & operator >> (istream & in, int v)
```

This is for the input of `int` values, but other types can be input via overloading. Now, the interesting thing about this operation is that it returns an `istream` value – the object resulting from the io operation (what results after removing the input value from the stream). In *Hugs* IO functions have the same characteristic – they return a special IO value which represents the altered state of the IO environment.

8.1.1 The Type `IO`

There is a type named `IO` in *Hugs* whose values are IO states. A few examples will illustrate how this type is used. Use the `':t'` command to get *Hugs* to tell you the type of the function `putStrLn`. You should get the following response.

```
Prelude> :t putStrLn
putStrLn :: String -> IO ()
```

So this function is one of these IO functions and takes a string value as a parameter. What is does, is to display the string on a line followed by a new line character, which the function supplies. Try it out.

```
Prelude> putStrLn "This is a line"
This is a line

(11 reductions, 30 cells)
Prelude>
```

If we embed a new line character of our own, the resulting string gets displayed as two lines.

```
Prelude> putStrLn "This is a line\nand so is this"
This is a line
and so is this

(11 reductions, 45 cells)
Prelude>
```

Notice the odd looking return type for `putStrLn` – `'IO ()'`. The empty parens form the notation for the one element type — rather like the type `void`. The type `IO` apparently takes an argument. Here's another example which will help clarify this feature.

```
getALine :: String -> IO ()

getALine str = do putStr str
                  line <- getLine
                  putStrLn line
```

When executed, this function displays the string `str` as a prompt, inputs the line which is terminated by a return, and then displays the line. Enter this function into a test file and try it out — here is what you might see (given the same parameter and input).

```
Main> getALine "Enter a line >> "
Enter a line >> this one?
this one?

(106 reductions, 218 cells)
```

What do we have in this function. First, and most obviously, there is a new "control" structure (operation) – `'do'`. This operations takes a sequence of function applications and carries them out one at a time, feeding the return value of one as input to the next. And of course, the return value is always of type `IO`. Check out the types of each of the functions involved.

```
Main> :t putStr
putStr :: String -> IO ()
Main> :t getLine
getLine :: IO String
Main> :t putStrLn
putStrLn :: String -> IO ()
```

The first and last of these functions aren't new. Each takes a parameter and displays it (notice what happens if you put `putStrLn` in place of `putStr`!). The second function, `getLine`, does not have the same structure — the argument with `IO` this time is `String`. This argument indicates that a value of type `String` is also returned by the function (as part of an ordered pair, if we could look behind the scenes). This extra value helps explain the syntax of the line containing the call to `getLine`.

```
line <- getLine
```

indicates that the `String` value returned by `getLine` is associated (in a substitutional way) with the identifier `line`. The value associate with `line` as well as the `IO` return value are both fed into the next (third) component of `do`.

Problem 18

Write an interactive function defined as

```
iChoose ::  IO ()
```

which interactively enters two integer values (each value comes in as a string), converts the values to integers, computes `choose` on the parameters and displays the result.

Hints:

- Use the function you wrote a couple of weeks ago to convert the input string to an integer value. See problem 4.5 – it was extra credit.
- Enter the two values separately – i.e., two prompts and two separate inputs.
- Remember that each element of a `do` operation must return the `IO` type, so the use of auxilliary functions should be considered.
- Do the function incrementally – do the inputs and check that they are correct. Then convert the input strings to `Int` and pass to `show` and that whole lot to `putStrLn`.
  Continue incrementally.

8.1.2 File IO

What we are really interested in is the ability to input from a file. This turns out to be realy easy. There is a function `readFile` which takes a path name as parameter and returns `'IO String'` – the `String` part is the list of characters in the file! Here is a function to enter into your test file and try out.

```
getFile :: String -> IO ()

getFile str = do line <- readFile str
                 putStr line
```

Notice that our function `getFile` takes a string parameter and returns the `IO` type. The action of the function is to open and read the file with pathname in `str` and associate the character list from the file with the identifier `line` (first line of the `do`) and then to print this character list to the screen. Give it a try giving the name of a file in the current directory.

Problem 19

You are to provide a *Hugs* implementation of the Unix command `wc` which displays the number of characters, words, and lines in a (single) Unix text file. Try the following in your *Hugs* session.

```
Prelude> :! wc test
      93     371    2315 test
```

[This illustrates the use of '`:!`' as a way to ask Unix to execute a command without leaving *Hugs*.]

Apparently the file 'test' has 2315 characters, 371 words, and 93 lines. How can we produce this same result without using the Unix `wc` command? The answer, of course, is to make use of the `readFile` function. If this function returns a list containing the characters in the file, then we should be able to use the `length` function to determine the length of that list. But this doesn't provide the formatting the `wc` command does or the values for number of lines and words.

You should implement the following function:

```
wc :: String -> IO ()

wc name = ???
```

- There is a function `words` defined in `Prelude.hs`. Check out its type. The function takes a character string and converts it to a list of strings where each string contains a single word (a list of strings).
- There is a function `lines` – it converts a character string into a list of strings, where each string is a line of the original string (i.e., terminated with the new line character).
- You have to write an auxiliary function which computes the three values needed and then puts them into the correct format and returns the resulting string. You might want to write more than one auxiliary function. These auxiliary functions will not be `IO` functions!

9 Recursive Data Structures in *Hugs*

In the previous section we introduced the notion of algebraic types as a substitute for the imperative record structure. In fact we found that algebraic types combine both enumerated as well as record types. One of the standard applications of records in imperative programming is to build list, tree, and graph nodes. These structures are dynamic and inherently recursive. How are they dealt with in *Hugs*? In this section we will look at two applications: a list type and a tree type.

9.1 A List Type for *Hugs*

You may have the impression that the list type in *Hugs* is built into the structure of the language. In fact, it is just a polymorphic algebraic type. The following could be substituted for the definition of list.

```
data MyList a = Empty |
                Node a (MyList a)
                deriving Show
```

In this new type `'a'` is a type variable, of course, and `Empty` and `'Node x y'` stand in place of the more familiar

and `'x:y'`, respectively. Notice the explicit recursion in the definition – this works just like the standard linked list implementation in Java, except our *Hugs* version is polymorphic.

```
MyList = struct{
                int     value;
                MyList *link;
         }
```

In a Java program an empty list will be represented by the `NULL` pointer. The elements of a longer list are accessed by following the `link` component. In *Hugs* the same structure exists, but there are no pointers, just constructors.

The values of the type `MyList` have a very similar structure to those of the standard *Hugs* list, but constructors are used in place of `[ ]` and `':'`. For example, the corresponding value for `[1,2]` would be

```
Node 1 (Node 2 Empty)
```

The type `MyList` above can also be used in function definitions by pattern matching. For this new list definition we can write two new functions.

```
myHead :: MyList a -> a

myHead (Node n rest) = n

myTail :: MyList a -> MyList a

myTail (Node n rest) = rest
```

These function definitions derive directly from the structure of the type definition for `MyList`. Each value of this type is in one of two forms – either `Empty` or `Node n r`, where `n` is of type `Int` and `rest` is of type `MyList`. Instead of using pointers we make use of natural recursion.

Problem 20

1. Write the `MyList` equivalent for the following lists:

(a) `'[ ]'`
(b) `[5]`
(c) `[5, 3, 1]`
(d) "Help!"

2. Edit a new file and call it 'mylist'. Into this file copy the type definition given above for `MyList` as well as the functions `myHead` and `myTail`. Load this into *Hugs* and try out the functions on a few lists. If you get what seems to be a funny result, try the same thing with the corresponding function from `Prelude.hs` – i.e., `'head [1,2]'` rather than `'myHead (Node 1 (Node 2 Empty))'`.

3. Implement a function myFilter which takes a Boolean function and a `MyList` and returns the `MyList` which contains those elements of the original list which satisfy the function (parameter).

```
myFilter ::  (a -> Bool) -> (MyList a) -> (MyList a)
```

4. (Extra Credit!) Add to the 'mylist' file a function with the following type

```
myDrop ::  Int -> (MyList a)-> (MyList a)
```

which drops the first `n` elements from the list (be sure to check the number of elements to be dropped against the length of the list). Notice that `drop n l` should return `l` if `n` is zero, but should return the list resulting from dropping `n-1` elements from the tail of `l`!

9.2 Trees

The tree structure is commonly used in programming. Having just seen how *Hugs* can be used to define a list structure, it should come as no surprise that we can use the same mechanism to define a type for trees. The recursive nature of trees suggests that there are three forms, the null tree, a value node, and a tree node, which has a sequence of subtrees (we will stick to binary trees). We can capture this structure in the following definition:

```
data Tree a = Null    |
              Value a |
              Branch (Tree a) (Tree a)
              deriving Show
```

This type restricts the value nodes to the leaves of the tree. If every node requires a value then the following could be used.

```
data Tree a = Null      |
              Branch a (Tree a) (Tree a)
              deriving Show
```

If we want to write a function `treeInsert` for integer trees we could do the following:

```
treeInsert :: Tree Int -> Int -> Tree Int

treeInsert Null v = Branch v Null Null
treeInsert t@(Branch n ltree rtree) v
            | v < n     = Branch n (treeInsert ltree v) rtree
            | v > n     = Branch n ltree (treeInsert rtree v)
            | otherwise = t
```

In this definition the insert function ignores values which already appear in the tree. To make this definition more general we must require that the type of the node data be an instance of the `Ord` class – so we would rewrite the type definition as follows (the function definition remains unchanged).

```
treeInsert :: (Ord a) => Tree a -> a -> Tree a
```

This definition will work, then, for any type of data which is an instance of the `Ord` class.

Example 14 – [building a sorted binary tree]

How can we build a sorted binary tree if given a list of values? As expected, we go through the list one element at a time and call `treeInsert` for each element, passing also the current value of the tree. There is a very small problem in how to get started. We want

```
treeBuild :: (Ord a) => [a] -> Tree a
```

but know that `treeInsert` takes a two parameters. We solve the problem by providing an auxiliary function to do the building and use `treeBuild` to extract the final value as the return value.

```
treeBuild' :: (Ord a) => Tree a -> [a] -> Tree a

treeBuild' tree []     = tree
treeBuild' tree (v:vs) = treeBuild' (treeInsert tree v) vs

treeBuild :: [Int] -> Tree Int

treeBuild = treeBuild' Null
```

Here is another trick we can use because of the use of Currying. We have defined a function without specifying a parameter. But a bit of thought indicates the parameter is superfluous. If we look at the right side of the function definition we see '`treeBuild' Null`', which, according to our discussion of Currying, has type '`[a] -> Tree a`', which is exactly the type we need for `treeBuild`. We are defining `treeBuild` to behave exactly as '`treeBuild' Null`' behaves. Since we have already specified this behavior, we need define no more.

Problem 21

1. Write a function which takes a tree as a parameter and returns a list of values from the nodes. The values should appear in "in-order" order – assume the tree has its values in the nodes.
2. Now write a function which sorts a list by first creating a sorted binary tree and then extracting the values from the tree. It should be obvious that this calls for the use of the function `treeBuild` and the function from the previous problem.
3. (Extra Credit!) Write a function `treeMap` which behaves like the list version – i.e., it takes a function and applies the function to each node of the tree, thus producing a new tree.

   ```
   treeMap :: (a -> b) -> (Tree a) -> (Tree b)
   ```

10 Processing Character Data - A Tokenizer

One of the important applications of lists in *Hugs* is to sequences of characters from input (for example a text file). In processing a language, for example, the first step is often to reduce the list of input characters to a list of tokens, as we have discussed in class. In this example we will see how pattern matching can be used to write a function which converts the head of a character list to a token.

10.1 Generating Tokens

Let's consider the functional programming language defined in the class project (click here to see the grammar). What we want to do at this point is to understand what tokens are inherent in the language and to write first a data type for the tokens and then write a function which converts an input character list (string) into a result list of tokens.

As we discussed in class, tokens are of two types: constant tokens (key words and punctuation) and variable tokens which have an associated value, for example `FunctionName` above which has a token category and a value which is the character string representing the actual name. We can use an algebraic type to define the token classes, using constant values for the key word and punctuation tokens and constructors for the variable tokens. Try this on your own first before looking at the following answer. First write down informally what the syntactic elements are and then categorize them. Here's the *Hugs* type definition for tokens for this language.

```
data Token = Id String    |
             Num Int      |
             Error String |  -- the parameter is the erroneous string
             If           |
             Then         |
             Else         |
             EndIF        |
             Let          |
             In           |
             Assign       |  -- for =
             LP           |  -- for left paren
             RP           |  -- for right paren
             SColon       |  -- semicolon
             Period       |  -- for .
             Plus         |  -- for +
             Times        |  -- for *
             Minus        |  -- for -
             Divide       |  -- for /
             Lt           |  -- for <
             Le           |  -- for <=
             Gt           |  -- for >
             Ge           |  -- for >=
             Eq           |  -- for ==
             Ne           |  -- for /=
             EOP             -- stands for end of program
                             -- (this is very handy)
             deriving (Eq, Show)
```

Notice the last line of the definition. This line indicates that the data type `Token` will be automatically made an instance of the classes `Eq` and `Show`. This means that values of type `Token` can be compared for equality and can be converted to a `String` value by calling `show`. This automatic instantiation is a byproduct of the algebraic type structure.

We ultimately want to define a function

```
convert2Tokens :: String -> [Token]
```

but we will start with a more basic function which takes the token substring at the head of the list and returns the appropriate value of type `Token`. Part of this function definition is easy since we can use pattern matching to deal with all the punctuation tokens. The function `getToken` should have the following type and (partial) definition.

```
getToken :: String -> Token

getToken []                 = EOP -- when the list is empty we're at
                                  -- the end of the program
getToken ('=':rest)         = Assign
--
-- you will fill this in
--
getToken input              = Error -- if non of the previous cases
                                    -- apply then return the error
                                    -- token regardless of the pattern
```

At this point anything which is not punctuation will be converted to the error token. This, of course, is not acceptable in the long run. The following list indicates what problems await.

1. What if one input token is the initial part of another input token? An example would be the two comparison operators `<` and `<=`. Another example could be (in Modula-2) `IF` and `IFELSE` (two key words) or `FOR` and `FORTH` (a key word and an identifier).
2. What if there are blanks at the beginning of the input?
3. What about recognizing variable length tokens such as identifiers and numeric constants?

The second two problems will be discussed in the next section. They are problematic because initial blanks, identifiers, and numeric constants involve an unpredictable number of characters – in this case pattern matching does not work well. The first problem will be dealt with below. But first, try the following problems.

Problem 22

1. Enter the type and function definitions just described into a new file called 'lexical'.
2. Complete the definition of `getToken` according to the pattern established above. Test it out with the following type of expression:

```
getToken "= anything here"
```

∎

Now we will address the problem of distinguishing tokens where one is the initial substring of the other. Suppose that we extend our language so that the usual comparison operators (`< <= == /= >= >`) are included. It should be easy to extend the definition of the type `Token` to include token versions of each of these operators (`LE` could be used for `<=`, for example). But converting to tokens presents a slight problem. Remember that the way pattern matching works is the first line of the function definition is tried to see if the corresponding actual parameter matches the pattern. If not, the second pattern is tried and so on until a matching pattern is encountered (that's why the last pattern above is just a variable name, since that will always be matched). This means that the order in which the patterns appear makes a difference. What would happen if the following lines were inserted into the definition of `getToken`?

```
getToken ('<':rest)     = Lt
getToken ('<':'=':rest) = Le
```

The answer is, the second pattern would never be matched, because if the first character of the input is `<` then even if the second character is `=` the first pattern will match. So the various pattern's to be used must be carefully ordered.

One final comment on the current example. We want to use the function `getToken` repeatedly to convert the input character list to an output token list. But in order to be able to get the second token (and subsequent tokens) we must know where we left off after the first token. Fortunately this is easy. We just have `getToken` return a pair, the token and the rest of the input list.

Problem 23

1. Extend the code in the 'lexical' file to include the comparison operators which you are familiar with from *C*, i.e., `<, <=, ==, /=, >=, >`. Chooses appropriate token names as indicated above. Modify `getTokens` so that it will recognize these tokens. [Do this in a copy of your 'lexical' file since we won't be using these operators in future examples.]
2. Change the type definition of `getToken` so that it returns a pair as indicated (token in the first component). Then modify the function definition to reflect this.
3. Even though we haven't finished `getToken` you should still be able to write the definition of `convert2Tokens`. Give it a try. Recursion is the key, of course, but be careful not to generate a list which is backwards.

10.2 String Processing

Remember from an earlier lesson we defined the following input function.

```
getFile :: String -> IO ()

getFile str = do line <- readFile str
                 putStr line
```

When the function `getFile` is evaluated the content of the file named by `str` is returned to `line` as a `String` value and then the value of `line` is displayed. We will want to make use of this function in order to print a list of tokens – so that we can test the tokenizer before incorporating it into a larger function – i.e., a parser. [Don't worry, that's not part of the tutorial – but it can be done in *Hugs* in a very eligant way.]

What we will want to do here is to replace the identifier `line` in the call to `putStr` with a function call as follows.

```
getFile :: String -> IO ()

getFile str = do line <- readFile str
                 putStr (f line)
```

The idea is that the function `f` will process the characters in the string `line` and eventually return a different string value. Suppose, for example we want to print the "words" in a file.

We could use the following function definition for `f`.

```
f :: String -> String
f str = unlines (words str)
```

The function `words` (from `Prelude.hs`) takes a string and returns a list of strings, where the strings in the list are the words from the original string. Here's an example.

```
Main> words "this is a .. <>--- test! for 'words'."
["this","is","a","..","<>---","test!","for","'words'."]
```

Notice that a "word" is apparently any sequence of non-whitespace characters. If we use `unlines` in conjunction with `words` then we get the following.

```
Main> unlines (words "this is a ..for 'words'.")
"this\nis\na\n..for\n'words'.\n"
```

What we have here is a string with embedded new line characters. If this value is given to `putStr` then the separate words should be displayed on separate lines. Give that a try in your test file. Before testing it create a small data file containing just a few words on a few lines – give that file name as the parameter to `getFile`. If you use a file which looks like this (the first and last lines are not part of the file – just to mark off the file contents)

```
============================
This is a ....
test       >==<

of getFile
============================
```

then the output from a call to getFile should look like this

```
Main> getFile "data"
This
is
a
....
test
>==<
of
getFile
```

What you will need to do later to test your tokenizer is make a similar modification to `getFile` so that when your tokenizer process a file of data, the tokens get written out one per line. The function `show` will probably come in handy at that time.

10.3 More String Processing

In Example 10.1 we introduced the syntax for a simple functional programming language and investigated the problem of converting a character representation of a token to an internal value of type `Token`. At that time, however, we only considered converting very simple tokens. There were three remaining problems listed which we will now consider. You should see this discussion in more general terms. Though we will look at processing lists of characters, the techniques discussed will often be applicable to more general list processing.

We found in Example 10.1 that pattern matching is great for processing a fixed number of characters at the beginning of a string. But it is common to want to access, for example, all leading characters which meet some specific criterion: all leading characters which are alphabetic (meaning upper or lower case letters). This general functionality is provided by two polymorphic list functions. The function `takeWhile` generalizes `head` in that it returns the initial substring of characters which satisfy a particular condition (there is another function `takeUntil`, with hopefully obvious behavior). `dropWhile`, on the other hand, generalizes `tail` by returning the original string after removing the leading substring of characters satisfying a particular condition.

To make use of these two new functions it is obviously essential to have a good understanding of how to specify a "condition". In the setting of our current example (processing tokens) a condition will be a boolean function on characters. Some examples are

```
isPrint    :: Char -> Bool  --  is a printable character
```

```
isSpace    :: Char -> Bool  --  is a character which prints as a space
                                   or moves the cursor
isUpper    :: Char -> Bool  --  is an upper case letter
isLower    :: Char -> Bool  --  is a lower case letter
isAlpha    :: Char -> Bool  --  is a letter
isDigit    :: Char -> Bool  --  is one of 0,1,2,3,4,5,6,7,8, or 9
isAlphanum :: Char -> Bool  --  is a letter or a digit
(== ' ')   :: Char -> Bool  --  is a blank
(/= ' ')   :: Char -> Bool  --  is not a blank
```

(see the *Hugs* user's manual).

10.3.1 removing leading blanks

Now if we revisit the problem of processing a token at the beginning of a string, we can try the following enhancement of `getToken`.

```
getTOKEN :: String -> (Token, String)

getTOKEN st = getToken st'
              where
                   st' = dropwhile isSpace st
```

What this does is to first drop all leading space characters (blanks, tabs, new lines, etc.) and then to apply our `getToken` function to a string which we know starts with an actual character. [Question: if we are worried that some wierd stuff may slip into the file (non-printable characters, for example) how might you modify this function to accomodate?] Now we can work on `getToken` with the assumption that a non-space character is at the head of its argument.

10.3.2 recognizing variable length tokens

We now want to deal with recognizing variable length tokens. In light of the previous sections this seems straigt forward enough. To handle an identifier just use '`takeWhile isAlpha`' to access a name string and '`takeWhile isDigit`' to access a numeric string. In processing a language, however, we must be careful not to recognize things which should be illegal. For example, identifiers must consist of only letters, no digits are allowed. So what should be done with a string like '`factorial123`'. Should that be illegal or should it be seen as indicating `factorial` applied to 123? For our language the answer is 'illegal': identifiers cannot contain digits. So when we find the head of the input string containing a letter or a digit, we must first remove the leading sequence of letters and digits and then determine whether this string is legal, i.e., consists of all digits or all letters. Notice that this means an identifier or number can only be immediately followed by an operator, punctuation, or a space character.

The following function is defined assuming that the character at the head of the parameter is a letter and that every character in the string is a letter or digit. We want to extract the initial sequence of letters, determine the appropriate token, and then return this token. Of course, if the string contains a digit we want to return the token `Error`.

```
getIdToken :: String -> Token

getIdToken st = if r == []
                   then Id id
                   else Error
                where
                     id = takeWhile isAlpha st
                     r  = dropWhile isAlpha st
```

The function `getNumToken`, to return a number token, is very similar. You give that a try.

There is one part of the problem which is still not solved. Our function `getIdToken` returns an identifier token for every string starting with a letter. But this means that any keyword will be seen as an identifier, rather than as a keyword. We can easily fix this by defining a new function to be called in the `then`-clause. This function will take the string and determine the appropriate token.

```
convertId2Token :: String -> Token

convertId2Token st
       | st == "IF"   = If
       | st == "THEN" = Then
       | st == "ELSE" = Else
       | otherwise    = Id st
```

Example 15 – the completed token extractor

But how do we combine these functions `getIdToken` and `getNumToken` into our function `getToken` from Example 10.1? Remember that our earlier definition of `getToken` returned the error token if an operator or punctuation symbol was not seen as first character. We need to replace that line with a new definition which takes care of the cases we have just discussed. Also, our processing will be determined by whether the first character is a letter or a digit (the only remaining legal alternatives). We would like to be able to do something like this

```
getToken (id ++ rest) = (token, rest)
                        where
                             ...
```

where we use pattern matching to pull off the initial string of letters and digits, as described above. But this is not allowed; the `++` operator is not legal in a pattern. Also, we need to have access to the first character of the parameter so we can extract a number or identifier token as appropriate.

Hugs has a convenient mechanism to help us write an elegant definition for this component of `getToken`.

```
getToken st@(ch:_) = (token, rest)
                     where
                          rest = dropWhile isAlphanum st
                          st'  = takeWhile isAlphanum st

                          token | isAlpha ch = getIdToken st'
                                | isDigit ch = getNumToken st'
                                | otherwise  = Error
```

The new bit is '`getToken st@(ch:_)`'. The parameter has two new twists. First ignore the `st@` part. The rest looks like pattern matching (the `ch` represents the first character of the list), but the underscore we haven't seen before. The underscore in pattern matching means match the pattern (in this case the rest of the list) but don't associate a name with it. Of course this is only reasonable if you don't want to refer to that part of the parameter. Now, in this case we do want to be able to refer, not the the part following the leading character, but to the whole parameter. The `st@` notation before a pattern associates a name, in this case `st`, with the whole parameter. So the we have two parameter names, `st` refers to the whole parameter, while `ch` refers only to the first character of the parameter.

We can now write down the completed `getTOKEN` and `getToken` functions.

```
getTOKEN :: String -> (Token, String)

getTOKEN st = getToken st'
```

```
              where
                   st' = dropWhile isSpace st


getToken :: String -> (Token, String)
         --
         -- we can assume that st has no leading spaces
         --
getToken []         = (EOP, [])
getToken ('=':rest) = (Assign, rest)
getToken ('(':rest) = (LP, rest)
getToken (')':rest) = (RP, rest)
getToken ('.':rest) = (Period, rest)
getToken (';':rest) = (SColon, rest)
getToken ('+':rest) = (Plus, rest)
getToken ('*':rest) = (Times, rest)
getToken ('-':rest) = (Minus, rest)
getToken ('/':rest) = (Divide, rest)
getToken st@(ch:\_)  = (token, rest)
                       where
                           rest = dropWhile isAlphanum st
                           st'  = takeWhile isAlphanum st

                           token | isAlpha ch = getIdToken st'
                                 | isDigit ch = getNumToken st'
                                 | otherwise  = Error
```

∎

Problem 24

1. Complete the above example by implementing `getNumToken` and entering all components of the `getTOKEN` function into your 'lexical' file. Test out your implementation.
2. What if an illegal character appears in your file, e.g., a `:` or `|`? This can be dealt with in the final component of the definition of `getToken`. Clearly the error token must be returned if a bad character is seen, but we don't want to have that character appear at the head of the returned string. Modify the definition of `getToken` to accomodate this situation. [Hint: Use the condition `"not isAlphanum"` to check for illegal characters. You will have to figure out from the manual how to write this condition.]
3. Go back to Problem 23.3

∎