The Little Haskeller

Cordelia Hall and John Hughes

November 9, 1993

1 Introduction

This document is a basic tutorial on Haskell programming. Work through it, doing all the exercises, and you should develop a basic knowledge of the Haskell constructs, datatypes, and programming methods.

Begin by starting up the Haskell interpreter.

```
Welcome to interactive Haskell B. version 0.999.5 SPARC 1993 Oct 28!
Loading prelude... 1127 values, 72 types found.
Type "help;" to get help.
>
```

You should try out the examples in this document using the interpreter as you read it. In places you will be asked questions that you can only answer by experimenting with the interpreter. Throughout the text you'll be guided by extracts from an Haskell session, set in the typewriter font used above.

2 Types, Values, Classes, and Definitions

Haskell is a strongly typed language, like Pascal. That means that the Haskell programmer is concerned about two different kinds of object...values, such as 1, 2, and 3, and types, such as Int, Bool, and Char. Values and types are related in that every value has a type. The type of a value defines the operations that can be applied to it: for example, values of type Int may be added together, while values of type Char may not. The Haskell system includes a type checker that ensures that these constraints are respected.

Some operations, such as +, are *overloaded* in the sense that they may be applied to many different types of value. Haskell groups types into *classes* supporting a particular set of operations. For example, + can be applied to arguments of any type in class Num. A type may belong to many different classes.

Values may be given names in definitions, signalled by the keyword `let`. Type in the definition

```
> let {x = -10};
x :: (Num a) => a
```

This defines the name `x`, and gives it the value `-10`. It also gives it the type of `-10`, which is `(Num a) => a`. Whenever a name is defined the Haskell interpreter echoes its name and type, as it has done here. `x` is overloaded: it can any type `a` in the class `Num`.

We can check that `x` has the value we expect by evaluating it.

```
> x;
-10
```

Since `x` is overloaded, we can evaluate it at many different types.

```
> x :: Int;
-10
> x :: Float;
-10.0000000
> x :: Double;
-10.000000000000000
> x :: (Complex Float);
(-10.0000000) :+ 0.00000000
```

Several definitions may be given at once:

```
> let {y = 5; z = 10};
y :: (Num a) => a
z :: (Num a) => a
```

The system echoes each name defined, along with its type.

There is no need to squeeze everything onto one line like this — a more usual layout would be

```
> let {y = 5;
       z = 10};
y :: (Num a) => a
z :: (Num a) => a
```

Giving many declarations together will be important later when we write mutually recursive functions.

We'll return to definitions that name types rather than values later in the tutorial.

2.1 The Haskell Layout Rule

If you type definitions in the form above, then layout is completely insignificant in Haskell, as it is in most programming languages. But Haskell also allows you to use layout instead of bracketing if you wish. Whenever it's acceptable to write a sequence of things enclosed in curly brackets and separated by semicolons, you can leave out the brackets and semicolons *provided the entries in the sequence appear on separate lines and start in the same column.* For example, the definition above can instead by given as

```
> let y = 5
#     z = 10
# ;
y :: (Num a) => a
z :: (Num a) => a
```

(Ignore the # here — it's just a prompt for a continuation line). Notice that the last semicolon has to appear on a line by itself now, so that it isn't confused with a semicolon separating declarations. Try varying the column that `z` appears in.

The layout rule isn't helpful for one-line definitions, but we will usually use it for larger examples.

3 Basic Types

We'll begin this tutorial by taking you through some of the built-in types and operators of the language.

3.1 Numbers

Numbers are built into all programming languages. Haskell has a wide variety, but we will concentrate on Int and Float. Numbers are written in the usual way, with the standard arithmetic operators. Evaluate

```
> 2+3;
5
> -10;
-10
> 2*(-10);
-20
```

These operations work on any numeric type — by default `Int`. Division, though, works only on 'fractional' types — those of class `Fractional` — by default `Double`.

```
> 10/3;
3.3333333333333335
> 10/3 :: Float;
3.33333325
```

See what happens if you try to force the result of a division to be `Int`:

```
> 10/3 :: Int;
???
```

There are also integral division and remainder operators, which work only on types in class `Integral`.

```
> 10 `div` 3;
3
> 10 `rem` 3;
1
```

Notice that when words are used as operator names they are enclosed in back-quotes.

There is another arithmetic operator. Evaluate

```
> 2^3;
???
> 3^2;
???
```

and figure out what it does.

3.2 Strings

Haskell strings are written in double quotes – for example

```
> "hello world!";
hello world!
```

Special characters such as newline can be included via an escape convention like that of C:

```
> "hello\nworld!";
hello
world!
```

We will have more to say about strings later. For the moment we just introduce one operator – string concatenation.

```
> "hello " ++ "world!";
hello world!
```

3.3 Booleans

Like other languages, Haskell has a type of booleans or truth values, written `Bool`. There are two truth values

```
> True;
True
> False;
False
```

Try out the primitive operators

```
> True && False;
False
> True || False;
True
> not True;
False
```

We can produce booleans by comparing numeric values.

```
> 1==2;
False
> 2<3;
True
> 3*3<=4;
False
> 2*2/=2+2;
False
> 2>3;
False
> 3*3>=4;
True
```

Notice that the equality test is written with a *double* equals sign.

These comparison operators can also be applied to other kinds of values, for example

```
> True < False;
False
> "hello" < "world";
True
```

They define a self-consistent ordering for every type in class `Ord`. Figure out, by trying several examples, what the ordering is for strings.

The most important use of booleans is in conditional expressions. These correspond to if-then-else in a language like Pascal. As an example, let's compute the absolute value of `x`, which we defined earlier.

```
> if x<0 then -x else x;
10
```

What is the value of

```
> if x*x < 60 then x*x else x*x - 60;
```

3.4 Enumeration Types

Haskell has user-defined types like Pascal enumeration types. For example, let's define a type of colours:

```
> data Colour = Red | Green | Blue;
data Colour = Red | Green | Blue
```

After this definition, we can use the names `Red`, `Green` and `Blue` in expressions.

```
> Red;
Red
```

For example, let's explore the ordering of colours.

```
> Red < Green;
True
> Green < Blue;
True
> Blue < Red;
False
```

Notice that both the new type, `Colour`, and the new values we've defined have names beginning with a capital letter. Haskell insists on this.

Define a type called `Days`, whose values are the days of the week. Check that `Monday` is less than `Tuesday`, and so on.

4 Functions

Most of programming in Haskell consists of defining functions – that's why it's called a functional programming language.

4.1 Simple Function Definitions

For example, a function to increment its argument could be defined by

```
> let {increment x = x+1};
increment :: (Num a) => a -> a
```

x is the function's argument. Note that, in Haskell, no brackets are required around the argument. No brackets are required in function calls either:

```
> increment 12;
13
```

Try the following example:

```
> increment 3*2;
```

Does this expression apply the increment function to (3*2), or apply it to 3 and multiply the result by 2? In general, which binds more tightly... function call or built-in operators?

Define and test a function which doubles its argument.

Here is a function with two arguments:

```
> let {sumsq x y = x*x + y*y};
sumsq :: (Num a) => a -> a -> a
```

Note that no brackets are necessary in this case either...we just write the function name followed by all its arguments. The syntax is the same in function calls.

```
> sumsq 3 4;
25
```

Evaluate the two expressions

```
> sumsq 3 2*2;
> sumsq 3 (2*2);
```

When do you need to use brackets in Haskell? Add brackets to the first expression to make its meaning clear.

4.2 Function Types

Here is a function to test whether a `Colour` is `Red`.

```
> let {isRed c = c == Red};
isRed :: Colour -> Bool
```

and another to test whether two colours are both non-red:

```
> let {neitherRed c1 c2 = c1/=Red && c2/=Red};
neitherRed :: Colour -> Colour -> Bool
```

Look at the types that the Haskell system assigned to these functions. All functions have types, like these, containing arrows. The type of a function consists of the type of each argument, followed by an arrow, with the type of the result at the end. Suppose we have a function with the rather strange type

```
strange_function::Int->Bool->Bool->Char
```

How many arguments has this function? What are their types? What is the type of its result? When you think you know the answers, look at the footnote to see if you are correct. [1]

Overloaded functions can have many types, but *all of the same form.* For example, the `sumsq` function above can have any type of the form `a -> a -> a`, where `a` is a type in class `Num`. Since `Int` is in class `Num`, it can have the type `Int -> Int -> Int`. We can check this by forcing `sumsq` to have this type:

```
> sumsq :: (Int -> Int -> Int);
<<(Int -> (Int -> Int))>>
```

Investigate what other types `sumsq` has. Does it have the type `Int -> Float -> Float`?

If we try to apply a function to arguments of the wrong type, the Haskell type checker will signal an error. Try it out:

[1] The answers are: the function has three arguments because there are three arrows in its type. The first argument is an integer, the second and third are booleans, and the result is a character.

```
> isRed True;
[65] Cannot unify types:
    Colour
and Bool
 in  isRed True
```

You will probably find the error messages less helpful than you are used to – let's study this one to see what it means.

The first line,

```
[65] Cannot unify types:
```

tells you that the type-expected two types to be the same, but they were not. The two following lines give the two types which the type-checker expected to match. In this case, the type-checker complains that the types `Colour` and `Bool` are not the same. The remaining part of the message contains the part of the program being checked when the error was found. Looking at it, we see the reason why the type-checker attempted to match the types `Colour` and `Bool`: the former is the type that `isRed` expects its second argument to have, while `Bool` is the type of the argument actually supplied.

Here's another example:

```
> sumsq 1 True;
[56] Not an instance Num Bool in  sumsq (fromInteger 1I) True
```

Remember that `sumsq` is an overloaded function which works for any numeric type. `True` is a boolean, so if only `Bool` were in class `Num` everything would be OK! The type-checker complains that it isn't — that's what "Not an instance `Num Bool`" means. Since so many operations are overloaded you will often see this kind of message.

Try some more examples, such as

```
> 1+true;
> increment "hello";
> 1&true;
```

You will discover that, while the type-checker can always tell you the types that fail to match, its idea of where the error happened is often less than helpful.

Unlike compilers for languages like Pascal, the Haskell system never produces a message of the form "`Bool` found where `Int` expected". There's a good reason for this – it genuinely cannot tell which of the two types given is wrong. This is because, while a Pascal programmer defines the types of her functions, a Haskell programmer does not. As a result, the system does not know what types the programmer intended each argument to have, and can only check the different uses of a function for consistency.

4.3 Using Conditionals

More interesting functions usually involve conditionals. What does the following function do?

```
> let {abs x = if x<0 then -x else x};
abs :: (Ord a, Num a) => a -> a
```

Define functions max and min, of two arguments, which return the maximum and minimum respectively of the two parameters they are passed. For example:

```
> let {max x y = ...};
max :: (Ord a) => a -> a -> a
> let {min x y = ...};
min :: (Ord a) => a -> a -> a
> max 1 2;
2
> min 1 2;
1
```

Define another function sign, of one argument, that returns 0 given 0, -1 given a negative argument, and +1 given a positive argument.

4.4 Guards

Because if-then-else is so common in functional programs, Haskell offers a special syntax to make it convenient. Functions can be defined by several equations, with a "guard" deciding when each equation applies. For example, the abs function could have been defined by

```
> let abs x | x<0 = -x
#     abs x | x>=0 = x
# ;
abs :: (Ord a, Num a) => a -> a
```

Look at the expressions following the — signs – these are the guards.

Try retyping the definition as

```
> let abs x | x<0 = -x
#     abs x       = x
# ;
abs :: (Ord a, Num a) => a -> a
```

Under what conditions is the second equation used?

Redefine the functions max and min, using guards instead of if-then-else this time.

4.5 Pattern Matching

Here is a definition for the function sign, using guards.

```
> let sign x | x<0 = -1
#     sign x | x==0 = 0
#     sign x | x>0 = 1
# ;
sign :: (Ord a, Num a, Num b) => a -> b
```

Look at the middle equation. It defines the `sign` function for just one argument value – zero. This kind of equation is so common that there is a special way of writing it:

```
> let sign x | x<0 = -1
#     sign 0       = 0
#     sign x | x>0 = 1
# ;
sign :: (Ord a, Num a, Num b) => a -> b
```

We can always write an equation defining a function for a particular argument value by writing that value as a constant on the left hand side of the equation. We call this mechanism "pattern matching". We think of the 0 on the left hand side as a "pattern" which the actual argument must match for the equation to apply.

Pattern matching is not confined to numbers. We could use pattern matching to define a function `tomorrow` on your type `days`:

```
> let tomorrow Monday = Tuesday
#     tomorrow Tuesday = Wednesday
#     tomorrow Wednesday = Thursday
#     tomorrow Thursday = Friday
#     tomorrow Friday = Saturday
#     tomorrow Saturday = Sunday
#     tomorrow Sunday = Monday
# ;
tomorrow :: Days -> Days
```

Define a function which converts a day to an appropriate three-letter string.

4.6 Recursion

Haskell has no while loops or for loops, or indeed any other kind of loops. Any iterative calculation must be expressed using recursion. We'll take as examples some simple recursive functions on integers.

The general method of defining a recursive function is

- write an equation that computes the function for sufficiently small arguments – the "base case".
- write an equation that reduces the problem for any other arguments to a combination of the problem for smaller arguments – the "recursive case".

When such a function is called, it calls itself recursively with smaller and smaller arguments until the argument is so small that it can be solved by the base case equation. The results are then combined to compute the result of the first call.

4.6.1 A Simple Form of Recursion

In the case of positive integers, we often choose just the argument zero as the base case...this is the smallest positive integer. In the recursive case, we derive the smaller argument for the recursive call by subtracting one from the given argument.

Let's apply these ideas to addition. Suppose that for some reason we don't want to use the built-in addition or subtraction operators, except to add and subtract one. We can in fact define addition in general in terms of these operations. We'll define a recursive function `add` with two arguments, and we must first choose one of them to be the "recursive argument" – the one which will become smaller and smaller in recursive calls. Let's choose the first. So we need to write an equation defining addition when the first argument is zero. That's easy:

```
> let add 0 n = n
```

because obviously, `0+n` is `n`. Now consider the recursive case, in which the first argument `m` is greater than zero. We know that we can use a recursive call

```
add (m-1) n
```

with a smaller first argument, and we know that it should compute `(m-1) + n`. All we need to do is compute `m + n` from that, and we can do so by adding one. So the equation for the recursive case is

```
#     add m n = 1 + add (m-1) n
# ;
```

Putting these two equations together to get a complete definition, we get

```
> let add 0 n = n
#     add m n = 1 + add (m-1) n
# ;
add :: (Num a, Num b) => a -> b -> b
```

Try out the function `add` to make sure that it works.

Now suppose that for some reason, we don't want to use the built-in multiplication operator. Define a recursive function `mul` which returns the product of two positive integers, computed by repeated additions. First consider the base case (multiplication by zero), and then consider the recursive case...you will need to figure out how to compute `m*n` from `(m-1)*n`.

Define a recursive function `power` which computes its first argument to the power of its second argument by repeated multiplication.

Define a function `fac` which computes factorial numbers...`fac n` should compute the product of all the numbers from `1` to `n`.

4.6.2 General Recursion

Recursive functions can have a more general form than this. Let's consider the problem of computing Fibonacci numbers. The first two Fibonacci numbers have the value 1, and the later Fibonacci numbers are obtained by adding together the two previous ones. Here's a function to compute the `n`th Fibonacci number:

```
> let fib n | n <= 2 = 1
#     fib n           = fib (n-1) + fib (n-2)
# ;
fib :: (Ord a, Num a, Num b) => a -> b
```

This is a recursive function like the ones we've been looking at, and indeed the first equation is the base case and second is the recursive case as usual. But in this example the base case doesn't solve the problem for zero – it solves it for one and two. Meanwhile the recursive case doesn't just subtract one from the argument, it makes two recursive calls with two different smaller arguments. But whatever positive integer we start with, subtracting one and two repeatedly will eventually lead to the base case. Recursion works as long as, by choosing smaller and smaller arguments, we eventually end up with an argument which can be processed by the base case equation. So this definition will work just fine.

Let's return to the problem of computing `m` to the power `n`. Powers can be computed more efficiently by repeated squaring than by repeated multiplication. Define

```
> let {sq m = m*m};
sq :: (Num a) => a -> a
```

Now a fast way to compute `power m n` is to compute `sq (power m (n `div` 2))` – provided `n` is even. Make a recursive definition of `power` that uses squaring to compute even powers, and multiplication to compute odd ones. This example is a little more complicated because there are two recursive cases (even and odd numbers) to be reduced to smaller problems, and one base case.

You'll need a function to test whether a number is even: here it is.

```
> let {even n = n `rem` 2 == 0};
even :: (Integral a) => a -> Bool
```

5 Data Structures

5.1 Tuples

5.1.1 Tuple Values and Types

Values may be grouped together into data-structures. Here are some examples:

```
> (1,2);
(1, 2)
> (True, "hello");
(True, "hello")
> (1,2,3);
(1, 2, 3)
```

Any number of values may be grouped together, and they may be of any types. A group of values such as `(1,2)` is called a *tuple*, and is itself a value in its own right. A tuple may be treated in every respect as a single value: it may be passed as a single argument to a function, or returned as a result, or even built into another tuple.

```
> ((1,2),(True,"hello"));
((1, 2), (True, "hello"))
```

Since tuples are themselves values, they have their own types. You can discover the type of a value by binding it to a name...remember that the Haskell interpreter then tells you the type of the name. Let's find the type of `(True,"hello")`.

```
> let {p = (True,"hello")};
p :: (Bool, String)
This means that {\tt p} is a pair, containing a boolean and a string.
Investigate the types of the other tuples we used as examples in the same
way.

\subsubsection{Functions on Tuples}
We usually define functions that operate on tuples using pattern matching.
Here is a function that expects a pair of numeric values, and adds them
together.
\begin{verbatim}
> let {addpair (x,y) = x+y};
addpair :: (Num a) => (a, a) -> a
```

Here `(x,y)` is a pattern...for an argument to match it, it must be a tuple of two values. Matching a pair against this pattern not only checks that it is a pair, it names the first component `x` and the second component `y`. We can then use these names in the right hand side of the equation.

Let's define a pair `p` and apply `addpair` to it.

```
> let {p = (1,2)};
p :: (Num b, Num a) => (a, b)
> addpair p;
3
```

When we call `addpair`, the value of `p`, which is the pair `(1,2)`, is matched against the pattern `(x,y)`. The match succeeds, and the first component (`1`) is named `x`, and the second component named `y`. Now we evaluate the right hand side of the equation defining `addpair`, which is `x+y`. The result is `3` – the sum of the two components.

Note that `addpair` is a function of one argument, which just happens to be a pair. Try applying it to two arguments:

```
> addpair 1 2;
```

Now define a function `addpairs` with two arguments, each a pair of integers, which returns a pair containing the first component of the first pair plus the first component of the second, and the second component of the first pair plus the second component of the second.

5.1.2 Polymorphic Types

Two very useful functions on pairs are

```
> let fst (x,y) = x
#     snd (x,y) = y
# ;
fst :: (a, b) -> a
snd :: (a, b) -> b
What do they do?

Let's try a few examples.
\begin{verbatim}
> fst (1,2);
1
> fst (true,"hello");
true
> fst ((1,2),(3,4));
(1,2)
```

Just like overloaded functions, this function can be applied to arguments of many different types. This is what the 'type variables' `a` and `b` mean. But unlike an overloaded function, the types `a` and `b` don't have to belong to any particular class. These functions can be applied to pairs of any type at all! A function with many types is called *polymorphic* – such functions are common in Haskell.

Define a function to swap the two elements of a pair, and make sure you understand its type.

Can you predict the type of the following function

```
> let {pick (x,y,z) = if x then y else z};
```

It's a polymorphic function, and you need to know that the **then** and **else** branches of a conditional are required to have the same type. Once you think you know the answer, type it in and see if you were right.

5.2 User-defined Types

We are not obliged to use tuples as our only type of data-structure: Haskell allows us to define our own. Let's define a type `Vector`, containing an x-coordinate and a y-coordinate, to represent locations on a screen.

```
> data Vector = Coord Int Int;
data Vector = Coord Int Int
```

This declaration defines a new type `Vector`, whose values are written in the form

```
> let {v = Coord 1 2};
v :: Vector
```

Try defining

```
> let {w = Coord 1 True};
```

The types following `Coord` in the type declaration tell us the number and types of the components of `Vector` values.

Here are two functions to extract the x and y coordinates from a vector.

```
> let xcoord (Coord x y) = x
#     ycoord (Coord x y) = y
# ;
xcoord :: Vector -> Int
ycoord :: Vector -> Int
```

Try them out by applying them to `v`.

Here's a function that returns a vector double the length of the vector it's passed.

```
> let {doublevec (Coord x y) = Coord (2*x) (2*y)};
doublevec :: Vector -> Vector
```

Now define a function `addvec` with two vector arguments that adds them together and returns a vector representing their sum. (Its x coordinate should be the sum of the x coordinates of its arguments, and its y coordinate should be the sum of the y coordinates of its arguments). Define another vector `w` and evaluate

```
> xcoord (addvec v w);
> ycoord (addvec v w);
```

Check that the results are as you expect.

5.2.1 Types with Many Variants

`vector` is a type all of whose elements have the same form. Here is a type with elements of three different forms:

```
> data Money = SEK Int | Pounds Int | Dollars Int;
data Money = SEK Int | Pounds Int | Dollars Int
```

and here's a function to convert money into crowns.

```
> let cvt_to_SEK (Pounds n) = n*12
#     cvt_to_SEK (Dollars n) = n*8
#     cvt_to_SEK (SEK n) = n
# ;
cvt_to_SEK :: Money -> Int
```

Let's try it out:

```
> cvt_to_SEK (Dollars 200);
1600
> cvt_to_SEK (Pounds 200);
2400
```

Try printing out a `money` value.

```
> Pounds 10;
Pounds 10
```

We might want to display money amounts in a more familiar way. Let's define a function to convert `Money` into a more readable string. We'll need the standard function `show`, which converts anything (of a type in class `Text`!) into a string. Test it:

```
> show 2;
2
```

Oh: how can we tell the difference between the way Haskell prints a number and the way it prints a string? Here's one way:

```
> (2, show 2);
(2, "2")
```

Haskell only prints strings without quotes if the entire value being printed is a string.

Now let's define the function to display money nicely:

```
> let showMoney (SEK n) = "SEK "++show n
#     showMoney (Pounds n) = "UK# "++show n
#     showMoney (Dollars n) = "$"++show n
# ;
showMoney :: Money -> [Char]
```

Now try `show_money` out:

```
> showMoney (SEK 20);
SEK 20
> showMoney (Pounds 3);
UK# 3
> showMoney (Dollars 5);
$5
```

Let's define a type to represent "cash units" or pieces of money...either coins or notes. Every coin or note has a value, and in addition coins are made out of a metal – either copper, silver, or gold (in an ideal world!). First we need a type of metals.

```
> data Metal = Copper | Silver | Gold;
data Metal = Copper | Silver | Gold
```

Now here's the type of pieces of cash.

```
> data Cash = Coin Metal Money | Note Money;
data Cash = Coin Metal Money | Note Money
```

Define a function `showCash` to display these values in a readable form.

Now try defining your own types. Define a type to represent the courses available to you, and then define a type `Student` with two kinds of value:

- a full-time student, taking two courses
- a part-time student, taking one course

Define a function to convert a `Student` to a string.

5.3 Parameterised Types

Let's suppose we are implementing a system that manipulates a mixture of confidential data, and publicly available data, and that we want to keep track of which data is private and which public. We could define the following type to keep track of the confidentiality of integers:

```
> data ConfidentialInt = PublicInt Int | PrivateInt Int;
data ConfidentialInt = PublicInt Int | PrivateInt Int
```

Now we could define operations that make sure that public data is not derived from private data, such as

```
> let add (PublicInt m) (PublicInt n) = PublicInt (m+n)
#     add (PublicInt m) (PrivateInt n) = PrivateInt (m+n)
#     add (PrivateInt m) (PublicInt n) = PrivateInt (m+n)
#     add (PrivateInt m) (PrivateInt n) = PrivateInt (m+n)
# ;
add :: ConfidentialInt -> ConfidentialInt -> ConfidentialInt
```

and we could define appropriate functions for displaying confidential numbers.

```
> let showConfidentialInt (PublicInt n) = show n
#     showConfidentialInt (PrivateInt n) = "PRIVATE!"
# ;
showConfidentialInt :: ConfidentialInt -> String
```

But now suppose we want to define a comparison operation on confidential numbers that respects confidentiality. We will need to define a type of `Confidential_Bool` of confidential booleans so that we can mark its result as private or public. And in general, we might want to keep track of the confidentiality of any kind of data. We do not want to have to define a new confidential type for every type of data we ever manipulate.

There is a way to avoid this. We can define a single *parameterised* type `Confidential`. A parameterised type is rather like a function from types to types. We will apply the parameterised type `Confidential` to other types to create new types for confidential data. For example, the type `Confidential Int` will correspond closely to the type `Confidential_Int` we defined earlier. But rather than write out another definition for `Confidential_Bool`, we will simply construct a type of confidential booleans as `Confidential Bool`. Similarly we will be able to write down types such as `Confidential (Int,Int)`, the type of confidential pairs of integers.

The definition of a parameterised type looks very like the definition of a function. In this example, we define `confidential` by

```
> data Confidential a = Public a | Private a;
data Confidential a = Public a | Private a
```

This defines all the types `Confidential Int`, `Confidential Bool`, etc. It also defines `Public` and `Private` as *polymorphic* notation for writing down elements. So we can now write `Public 1` for a public integer, or `Private "John Hughes"` for a private string.

```
> let n = Public 1
#     s = Private "John Hughes"
# ;
n :: (Num a) => Confidential a
s :: Confidential String
```

The same notation is used for confidential values, no matter what type their component is.

Define your own parameterised type `Maybe` to represent the result of a computation that may or may not succeed. Values of type `Maybe` should take one of two forms:

- `Success v` – representing successfully computing `v`
- `Failure` – representing failure to compute anything.

5.3.1 Functions on Parameterised Types

We can use the same notation, via pattern matching, to define functions on parameterised types. Here's the new definition of confidential addition:

```
> let add (Public m) (Public n) = Public (m+n)
#     add (Public m) (Private n) = Private (m+n)
#     add (Private m) (Public n) = Private (m+n)
#     add (Private m) (Private n) = Private (m+n)
# ;
add :: (Num a) => (Confidential a) -> (Confidential a) -> Confidential a
```

Define a function

```
divide :: (Fractional a) => a -> a -> Maybe a
```

which divides its first argument by its second, but returns `Failure` if the second argument is zero.

Define a function

```
add :: (Num a) => (Maybe a) -> (Maybe a) -> Maybe a
```

which succeeds if both its arguments succeeded, and adds their values together, and otherwise fails.

6 Lists

Unlike tuples, lists are data structures which can be of any length. Some, in fact, are infinitely long. However, all of the elements of a list must have the same type, unlike those of tuples, which may have different types. For example:

```
> [1,2,3,4,5];
[1, 2, 3, 4, 5]
> [1,'h'];
[56] Not an instance Num Char in  (:) (fromInteger 1I) ((:) 'h' ([]))

> (1,"hey");
(1, "hey")
```

In this example, the typechecker expected to see an expression of a numeric type because the other list element was numeric, and so it determined that it was typechecking a list of numeric elements. Instead, it saw an element of type Char, and produced an error message.

6.1 List primitives

The empty list has the type of any list, and is written as `[]`. Primitives for taking apart data structures include `head`, which selects the first element of a list:

```
> head [1,2];
1
>
```

and `tail`, which returns the list without its first element.

```
> tail [1,2,3];
[2, 3]
>
```

Lists can be constructed using the `cons` primitive, written ":".

```
> 1 : [2];
[1, 2]
> 1 : 2;
[56] Not an instance Num [a] in  fromInteger 2I
>
```

Notice that the second argument must be a list of elements with the same type as that of the first argument to `cons`. In the example above the second argument was a numeric constant, which can have any numeric type, but unfortunately lists are not numeric.

6.2 Lists and pattern matching

Lists can appear as patterns in the definitions of functions. For example, it is often useful to test a value which may be the empty list. There are at least two ways to do this. One is:

```
> let empty xs | xs == [] = True
#      empty xs = False
# ;
empty :: (Eq a) => [a] -> Bool
> empty [1,2];
False
```

Here, the argument is tested explicitly. A more readable (and more efficient) way to define `empty` is:

```
> let empty [] = True
#      empty xs = False
# ;
empty :: [a] -> Bool
> empty [];
True
> empty [1,2];
False
```

(This definition also has the advantage that it doesn't require the list elements to be in classs `Eq` — that is, to have an equality defined).

Lists can also be taken apart using pattern matching. For example, it's possible to write `head` and `tail` as follows:

```
> let head (x:xs) = x
#     tail (x:xs) = xs
# ;
head :: [a] -> a
tail :: [a] -> [a]
> head [1,2];
1
> tail [1,2];
[2]
>
```

7 Lists and recursion

Recursion is a natural way in which to explore and build lists. You should become as familiar as possible with recursion over lists, because it is extremely common. In fact, once you are comfortable with it, you can write powerful functions which handle the recursion for you, so that the chance of introducing errors is reduced.

7.1 Examining lists using recursion

Suppose you want to define a function that tests a list to determine whether it contains zeros. Here is the definition of the function `hasZero`:

```
> let hasZero [] = False
#     hasZero (0:xs) = True
#     hasZero (x:xs) = hasZero xs
# ;
hasZero :: (Num a) => [a] -> Bool
> hasZero [1,2,3];
False
> hasZero [1,0,3];
True
> hasZero [];
False
```

Notice that there are only three forms in which the argument is presented to the function. There are several tricks to writing good recursive definitions. One of the most useful is learning the forms in which the argument should be presented. This comes with practice. Notice that there was no form which explicitly tested the second element of the list. Why is this? [2]

Here's a similar recursive definition over lists.

```
> let isin a [] = False
#     isin a (x:xs) = x==a || isin a xs
# ;
isin :: (Eq a) => a -> [a] -> Bool
> isin 2 [];
False
> isin 2 [1,2,3];
```

[2] The list is either empty or nonempty. If the list is empty, it has no second element. If it is nonempty, then either its first element is zero or it isn't. If the first element is zero, there is no further need to test the second. Otherwise, the second one will be tested when the function is called again on the tail of the argument

```
True
> isin 2 [1,3];
False
```

Write a function that does the same thing as `hasZero`, but uses `isin` instead.

7.2 Building lists with recursion

Lists are often built from other lists using recursion. Generally, some kind of test is performed on the argument list to determine where and how the new list gets built. A simple example is the function `removeFirst`, which searches down a list until it finds an element that is equal to its first argument, and then removes it. The result is the original list, with this first instance removed.

Here's an initial attempt to write this function:

```
> let removeFirst a [] = []
#     removeFirst a (x:xs) | x==a = xs
#     removeFirst a (x:xs) = removeFirst a xs
# ;
removeFirst :: (Eq a) => a -> [a] -> [a]
> removeFirst 1 [2,3,4];
[]
```

This isn't a good definition, because it doesn't reconstruct the front of the list, which contains all the elements tested so far. To reconstruct this part of the list, we'll modify the definition slightly:

```
> let removeFirst a [] = []
#     removeFirst a (x:xs) | x==a = xs
#     removeFirst a (x:xs) = x : removeFirst a xs
# ;
removeFirst :: (Eq a) => a -> [a] -> [a]
> removeFirst 1 [2,3,4];
[2, 3, 4]
> removeFirst 1 [2,3,1,4];
[2, 3, 4]
> removeFirst 1 [2,3,1,4,1,5];
[2, 3, 4, 1, 5]
```

Write `removeAll`, which removes all instances of elements equal to its first argument from its second argument. [3]

The general form of these functions is as follows:

- First specify the smallest possible list, then
- Specify the way in which components will be added to this smallest element to build up the final list value

This is similar to the way in which recursion over numbers was done earlier.

- First specify the smallest possible integer, then
- Specify the way in which components will be combined with this smallest element to build up the final integer value

[3] Change the second line of the definition so that it returns `removeAll a xs` instead of `xs`.

For example, look at the following definition of `increment`:

```
> let increment 0 = 1
#     increment n = 1 + increment (n-1)
# ;
increment :: (Num a, Num b) => a -> b
> increment 2;
3
```

Write a function with a structure similar to this one, except that it counts the elements of a list. [4]

Another list building function which has a structure related to the previous definitions is `heads`:

```
> let heads [] = []
#     heads ((x1:xs1):xs) = x1:heads xs
# ;
heads :: [[a]] -> [a]
> heads [[1,11,111],[2,22,222]];
[1, 2]
```

The function `heads` collects the first element in each of the list elements of its argument. Write a function similar to `heads` which does the same for the second elements.

The following function inserts a value to the right of particular elements of a list.

```
> let toRight old new [] = []
#     toRight old new (x:xs) | x==old = old : new : toRight old new xs
#     toRight old new (x:xs) = x : toRight old new xs
# ;
toRight :: (Eq a) => a -> a -> [a] -> [a]
> toRight 1 0 [0,0,0];
[0, 0, 0]
> toRight 0 1 [0,0,0];
[0, 1, 0, 1, 0, 1]
```

Write a function that inserts to the left. [5]

The function `subst` substitutes a value for another in a list. Here is its definition:

```
> let subst old new [] = []
#     subst old new (x:xs) | x==old = new : subst old new xs
#     subst old new (x:xs) = x : subst old new xs
# ;
subst :: (Eq a) => a -> a -> [a] -> [a]
> subst 2 3 [2,2,4];
[3, 3, 4]
> subst 2 3 [1,1,2];
[1, 1, 3]
> subst 2 3 [1,1,3];
[1, 1, 3]
```

The function `append` can be defined as follows:

[4] `let len [] = 0; len (x:xs) = 1+len xs;`

[5] Exchange `new` and `old` in the right hand side of the second line (but not in the call to `toLeft`)

```
> let append [] ys = ys
#     append (x:xs) ys = x : append xs ys
# ;
append :: [a] -> [a] -> [a]
> append [1,2,3] [4,5,6];
[1, 2, 3, 4, 5, 6]
> append [] [1,2,3];
[1, 2, 3]
```

Auxiliary functions are often useful when creating definitions. For example, the function `rev`, which reverses the order in which elements appear in a list, uses `append` to rebuild the list.

```
> let rev [] = []
#     rev (x:xs) = append (rev xs) [x]
# ;
rev :: [a] -> [a]
> rev [1,2,3];
[3, 2, 1]
> rev [1];
[1]
> rev [];
[55] Ambiguously overloaded tyvar(s) in class(es) Text in  show (rev ([]))
```

(The last example isn't an error. It's just that Haskell doesn't know what type the list elements are, and so refuses to print the list).

This can be done more efficiently by giving `rev` a second argument which contains the list as it is being built. This is sometimes called an accumulating parameter.

```
> let rev2 [] acc = acc
#     rev2 (x:xs) acc = rev2 xs (x:acc)
# ;
rev2 :: [a] -> [a] -> [a]
> rev2 [1,2,3] [];
[3, 2, 1]
```

A comparison of the two function definitions shows that the first performs an `append` which must reconstruct the tail of the list for each element of that list. Each one of these append operations must use several cons operations. The second uses one cons for each element of the list. On the other hand, the accumulating parameter must be passed an initial value, which makes the call to a function written in this way a bit messier.

8 Strings

A string is represented by a list of characters in Haskell. The functions `head` and `tail` behave just as they would on lists. For example:

```
> head "11 2 3";
'1'
> tail "11 2 3";
1 2 3
```

Notice that single character values, of type `Char`, are enclosed in *single* quotes, while string values, of type `List Char`, are enclosed in double quotes.

The function `append` can also be written as `++`, and is used to concatenate strings.

```
> "a b c "++"d e f";
a b c d e f
>
```

There a number of useful functions that test whether a character falls in various classes, such as digits, letters, and white space characters. For example,

```
> isDigit '9';
True
> isAlpha 'a';
True
> isAlpha 'A';
True
> isAlpha '9';
False
```

Other functions in this group are `isUpper`, `isLower`, and `isSpace` – try them and work out what they do.

As a more difficult exercise, define a function `words` that splits a string up into a list of words. Consider a word to be any sequence of non-space characters separated by spaces. For example,

```
> words "here is an example!";
["here", "is", "an", "example!"]
> words "spaces at the end are ignored ";
["spaces", "at", "the", "end", "are", "ignored"]
```

Hint Use two mutually recursive auxiliary functions, one which skips spaces and then calls the other, and one which actually collects up a word, using an accumulating parameter to hold the part of the word collected so far.

9 Recursive User-Defined Types

In Haskell, as in Pascal, we can define recursive types. For example, we might define a type of linked lists, where each non-nil node holds an integer.

```
> data Li = Nil | Node Int Li;
data Li = Nil | Node Int Li
```

In fact, the standard list data-type behaves just as though it had been declared like this:

```
> data [a] = [] | a : [a];
```

except, of course, that "`[]`" and `[a]` are not valid names in Haskell.

Notice that it is also possible to create a list data type in which every other element is a character. Here is an example:

```
> data Lic = Nil | NodeInt Int Lic | NodeChar Char Lic;
data Lic = Nil | NodeInt Int Lic | NodeChar Char Lic
> NodeInt 2 (NodeChar 'a' Nil);
NodeInt 2 (NodeChar 'a' Nil)
```

Just as in Pascal, user-defined recursive types let us define all kinds of tree types. Here is an example: a tree containing an integer at every node.

```
> data TreeInts = Nil | Node Int TreeInts TreeInts;
data TreeInts = Nil | Node Int TreeInts TreeInts
> Node 1 (Node 2 Nil Nil) (Node 3 (Node 4 Nil Nil) Nil);
Node 1 (Node 2 Nil Nil) (Node 3 (Node 4 Nil Nil) Nil)
```

Define a function `substTree`, like `subst` on lists, that replaces all occurrences of one number in a tree by another. So for example, if `t` is the tree above, then `substTree 3 5 t` should be the tree

```
Node 1 (Node 2 Nil Nil) (Node 5 (Node 4 Nil Nil) Nil)
```

Now define your own parameterised tree type, whose nodes may contain values of any type. For example, `Tree Int` should correspond to the type `TreeInts` above. Define `substTree` for this new type.

10 Modules, Definitions, and Programs

10.1 Modules

So far we have concentrated on typing definitions directly into `hbi`. However, this is not a practical way to build a real program because there is no way to save the definitions already entered when you quit from `hbi`. So next time you use Haskell you have to enter your definitions all over again.

If definitions are to be kept from one session to the next they must be stored in a *module*. A module is just a file containing one or more Haskell declarations. Such files must be given names ending in "`.hs`".

Here is an example of a module containing a definition of the factorial function:

```
module Fac where
fac 0 = 1
fac n = n*fac(n-1)
```

The first line begins the module and names it — the module should appear in a file called Fac.hs so that file name and module name agree. It's followed by the definitions, all of which are exported by default. Notice that the keyword `let` does not appear – function definitions in a module consist of everything *following* the keyword `let` in `hbi`.

We could load this module into `hbi` like this:

```
> load "Fac.hs";
Loading "Fac.hs"
fac :: (Num a) => a -> a
```

The interpreter echoes the names defined in the usual way.

Modules can of course contain many definitions. For example, here is a module that defines a type and two functions.

```
module Tree where -- this module defines binary trees and their operations

data Tree a = Empty | Branch (Tree a) a (Tree a)

lookup v Empty = False
lookup v (Branch l x r) | v<x = lookup v l
lookup v (Branch l x r) | v==x = True
lookup v (Branch l x r) | v>x = lookup v r
```

```
insert v Empty = Branch Empty v Empty
insert v (Branch l x r) =
        if v<x then Branch (insert v l) x r
else if v==x then Branch l x r
else Branch l x (insert v r)
```

Here's what happens when we load this module:

```
> load "Tree.hs";
Loading "Tree.hs"
lookup :: (Ord a) => a -> (Tree a) -> Bool
insert :: (Ord a) => a -> (Tree a) -> Tree a
data Tree a = Empty | Branch (Tree a) a (Tree a)
```

However, if you try to use Tree now you'll discover there's a problem

```
> insert 3 (insert 4 Empty);
[56] Not an instance Text (Tree a) in  show (insert (fromInteger 3I)
(insert (fromInteger 4I) Empty))
```

The problem is that the type `Tree a` is not in class `Text`, and so cannot be printed. Yet when we typed the same definition directly into the interpreter, `Tree a` *was* in class `Text`!

In general, types only become members of classes when the programmer explicitly declares it — see the Haskell report for details. However, the interpreter automatically makes interactively declared types members of all the standard classes (where that makes sense). When a type is declared in a module we can instruct Haskell to make a member of some of the standard classes automatically, by adding a 'deriving clause' to the type definition. By changing the type definition in the module above to

```
data Tree a = Empty | Branch (Tree a) a (Tree a)
    deriving (Text, Eq, Ord)
```

we can instruct the compiler to insert the new type into the classes `Text`, `Eq` and `Ord`, which will enable us to print and compare trees[6].

The Monomorphism Restriction Haskell allows overloaded *functions* to be exported from modules, but doesn't allow overloaded *values* to be exported, for efficiency reasons. For example, if were were to amend the module Fac.hs as follows:

```
module Fac where

fac 0 = 1
fac n = n*fac(n-1)

f6 = fac 6
```

then the interpreter would refuse to load it. The problem is that `f6` is an overloaded value, and would actually have to be recomputed each time it was used. The simplest solution is to add a *type declaration* to the module to force `f6` to have some particular type. For example,

[6] A `deriving` clause generates standard definitions for printing and comparison operations. It's also possible to define your own print functions, for example, by inserting the new type into class `Text` explicitly. See the report if you want to do this!

```
module Fac where

fac 0 = 1
fac n = n*fac(n-1)

f6 :: Int
f6 = fac 6
```

The problem is solved.

10.1.1 Compiling Modules

Modules can be compiled using the `hbc` command. Create the module `tree.hs` shown above. You can compile it with the command

```
% hbc -c Tree.hs
```

Note that this is a UNIX command! If you type it to the Haskell interpreter, it will make no sense of it.

The compiler produces two files:

- `Tree.hi` – containing the types of exported names
- `Tree.o` – containing the compiled code of the module

You can now load the compiled module into the interpreter like this:

```
> load "Tree.o";
Loading "Tree.o"
insert :: (Ord a) => a -> (Tree a) -> Tree a
lookup :: (Ord a) => a -> (Tree a) -> Bool
data Tree a = Empty | Branch (Tree a) a (Tree a)
```

Compiled code runs much faster than interpreted code, but because the compiler takes a little while to run it's generally better to compile modules only when you think they work.

If you just give the command

```
> load "Tree";
Loading "Tree.o"
insert :: (Ord a) => a -> (Tree a) -> Tree a
lookup :: (Ord a) => a -> (Tree a) -> Bool
data Tree a = Empty | Branch (Tree a) a (Tree a)
```

then `hbi` loads the compiled version of the module if it's up-to-date. If the source has been changed since compilation, it loads the source and interprets it. This is quite useful if you can't remember whether or not you've compiled the latest version of your module.

10.1.2 Using Modules in Other Modules

To use the functions in a module interactively, you just have to load it into `hbi`. To use them within another module, you have to explicitly include them. For example, here is a module which uses `tree.hs`.

```
module Foo where
import Tree   -- this line makes the functions
              -- in Tree.hs available
initialTree :: Tree Int
initialTree = insert 3 (insert 4 Empty)
```

(Notice we need a type declaration here because of the monomorphism restriction). Before this module can be compiled or loaded into `hbi`, the file `Tree.hi` must be created by compiling `Tree.hs`.

Putting an `import` command in a module just lets the definitions in it refer to the names in the included module. You still have to load *both* modules into `hbi` before you can run your program.

10.2 Local Declarations

As you begin to write larger functions, you will find you want to declare *local* variables inside them. You can do so like this:

```
> let fourthpower x =
#        let xsq = x*x in
#           xsq * xsq
# ;
fourthpower :: (Num a) => a -> a
```

Here `xsq` is a local variable of the function `fourthpower`. It is defined within the expression following the keyword `in`.

10.3 Stand-alone Programs

Until now we have always run Haskell programs from within `hbi`. It's also possible to create programs that run independently, and can be invoked from the shell just like programs written in any other language. Such programs just consist of an expression in a file. When they are run, the expression is evaluated and the result printed.

As an example, we'll write a program to print the sixth factorial number. We'll use the module `fac.hs` above. Here's a program which uses that module, and computes the sixth factorial number.

```
import Fac
main = print (fac 6)
```

Let's put this into the file `Main.hs`.

Assuming we've previously compiled the module `Fac.hs`, we can compile the program.

```
% hbc -c Main.hs
```

We still can't run it, however: we have to first combine the code for the program with the code for the module it uses to create an executable file. We can do so with a variation of the `hbc` command.

```
% hbc Main.o Fac.o -o fac6
```

Notice that we supply the *code* files as arguments, not the source files. The filename following "`-o`" is where the executable code is put. So we can now run our program:

```
% fac6
720
```

Stand-alone programs can be much more complicated than this: they can interact with the user, write to multiple files, and issue UNIX commands...see the Haskell manual for details.