A Taste of Functional Programming – 1

Madhavan Mukund

Functional programming has its roots in Alonzo Church's lambda calculus. A functional program is a collection of functions that work together to transform data. Though Lisp brought functional programming to public attention in the 1950s, it was John Backus's 1977 Turing Award lecture criticizing the limitations of imperative programming languages that led to a resurgence of interest in this field. The 1970s and 1980s saw a number of advances, leading to the development of the language Haskell, which carefully combined ideas from many earlier languages. In this article, we explore some basic notions of functional programming via Haskell.

Madhavan Mukund has been on the Computer Science faculty at Chennai Mathematical Institute since 1992, where he is currently Professor. His research interests are primarily in formal methods for specifying and verifying computing systems. He is the Secretary of the Indian Association for Research in Computing Science (IARCS) and the National Coordinator of the Indian Computing Olympiad.

1. Imperative Programming

Programming languages such as C, C++ and Java, as well as their predecessors FORTRAN and Pascal, all share a broadly similar operational style. In each of these languages, a program is a set of instructions to manipulate values that are stored in named variables. The process of computation consists of transforming the initial *state* of these variables to a desired final state. This style of programming is sometimes called *imperative*, because a program is a sequence of commands describing how to manipulate the state.

Imperative programming languages have a close connection with the way modern computers are designed. This is usually called the von Neumann architecture, after the famous mathematician John von Neumann who proposed this design in the 1940s [1].

In the von Neumann model, programs and data reside in

[1] Others also had similar ideas around the same time, but the name has stuck.

Keywords
Programming languages, functional programming, declarative programming, Haskell, lazy reduction, infinite data structures.

Backus proposed a language that emphasized the role of functions and mechanisms for combining them to form larger programs. Though he made a distinction between his proposal and the style known as *functional programming*, there were enough similarities in the two approaches that his address inspired a much greater interest in functional programming.

a common *memory* from which an individual item can be identified and manipulated using its *address*. All the manipulation happens in a separate *central processing unit*, or CPU. Thus, executing a program consists of the CPU continuously fetching data from the memory and updating it in small units.

In 1977, John Backus, the inventor of FORTRAN, received the prestigious Turing Award for his contributions to Computing Science. Somewhat unexpectedly, in his Turing Award address, entitled "Can Programming be Liberated from the von Neumann Style?", he strongly criticized the way that the von Neumann architecture had influenced the evolution of programming languages.

He coined the phrase *von Neumann bottleneck* to describe the constraint placed on computation by having all data flow through a narrow channel between the CPU and the memory. His thesis was that programming languages that were based on this architecture were obliged to describe computation as a sequence of updates to individual memory locations. This style of programming obfuscated the structure of the overall problem being addressed and made it much harder to break up large systems into small functional blocks that could be combined in a reliable manner.

Backus proposed a language that emphasized the role of functions and mechanisms for combining them to form larger programs. Though he made a distinction between his proposal and the style known as *functional programming*, there were enough similarities in the two approaches that his address inspired a much greater interest in functional programming.

2. Functional Programming

Functional programming has its roots in an abstract model of computation called the lambda calculus, pro-

posed by Alonzo Church in the 1930s. The lambda calculus provided a notation for describing functions that could be *effectively computed*, even though the electronic computer was yet to be invented at the time! In parallel with Church's approach, Alan Turing proposed a more operational model of an abstract computational device, now called a Turing machine, in which symbols are written on a tape and manipulated by a head that moves up and down the tape, reading and modifying symbols. The Turing machine is a direct intellectual ancestor of the von Neumann architecture.

In its purest form, data in a functional program flows from the input to the output and gets transformed along the way by each box that it passes through.

A functional program can be viewed as a black box that transforms inputs to outputs. These boxes can be chained together and combined in other ways to create larger boxes. In its purest form, data in a functional program flows from the input to the output and gets transformed along the way by each box that it passes through. There is no explicit memory, or state, and hence the way in which computation is described is fundamentally different from imperative programming.

One of the first functional languages to achieve a degree of popularity was LISP, invented by John McCarthy in the 1950s. LISP became popular for writing programs that performed *symbolic* computations–for example, reading and understanding plain text or symbolic differentiation and integration of algebraic expressions.

Values in LISP are treated as abstract symbols–there is no real notion of *data type*. In the 1970s, Robin Milner invented a typed functional programming language called ML as part of a theorem proving system called LCF. Typed languages are typically easier to program in than untyped ones and ML led to a resurgence of interest in functional programming. Backus's 1977 lecture inspired a lot of research into functional programming that led to a number of so-called *lazy* functional languages such as Miranda.

In general, a function that takes inputs of type A and produces outputs of type B has the type A -> B, where -> is intended to resemble the arrow →.

In the 1980s, the functional programming community made a conscious decision to combine forces and come up with a single language that unified the features from the many functional languages that had been developed. This led to the language Haskell being defined in the mid 1980s. Haskell has since become a standard language for teaching and research in functional programming. The language was standardized in 1998 as Haskell 98. Since then, a number of extensions have been proposed and may result in a new standard being defined soon.

3. Haskell

Programs in Haskell are functions that transform inputs to outputs. The description of a function f has two parts:

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

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

Here is a corresponding definition in Haskell.

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

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

The basic types in Haskell correspond to those in many other programming languages. For instance, `Int` denotes the set of integers, whose range is bounded, as

[2] It turns out that types can be automatically calculated for functions defined in Haskell, so we need not explicitly specify the type. We will discuss this in the second part of this article.

usual, by the number of bytes allocated to store a value of this type. (Haskell also has the type `Integer` for integers of arbitrary size.) The type `Bool` contains the two boolean values, written `True` and `False`. The type `Float` contains non-integral numbers–the name comes from *floating-point*–while `Char` denotes the type of characters. This is not an exhaustive list. In this quick introduction to Haskell, we will generally stick to the types `Int` and `Bool`.

In addition to the types of the input and output and the rule for computation, we also need to include information about the *arity* of the function, or how many inputs it takes.

3.1 *Functions with Multiple Inputs*

One attribute that we have not been included in our function definition is the number of inputs that it reads. The function `sqr` that we saw earlier has only one input. On the other hand, we could write a function with two inputs, such as the mathematical function plus: $plus(m, n) = m + n$.

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

This complication can be avoided by the somewhat drastic assumption that all functions take only one argument. How then can we define a function such as *plus* that needs to operate on two arguments? We say that *plus* first picks up the argument m and becomes a new function *plus* m, that adds the number m to its argument n. Thus, we break up a function of two arguments into a sequence of functions of one argument [3].

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

[3] This process is called *currying*, named after the logician Haskell B Curry, after whom Haskell is also named. Ironically, currying was not invented by Haskell Curry. This idea was first proposed Gottlob Frege and later refined by Moses Schönfinkel.

In general, suppose we have a function f that reads n inputs x1, x2, ..., xn of types t1, t2, ..., tn and produces an output y of type t. We can verify that the type of this function would work out to f::t1 -> (t2 -> (... -> (tn -> t)...)).

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

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

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

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

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

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

In general, suppose we have a function `f` that reads `n` inputs `x1`, `x2`, ..., `xn` of types `t1`, `t2`, ..., `tn` and produces an output `y` of type `t`. We can verify that the type of this function would work out to `f::t1 -> (t2 -> (... -> (tn -> t)...))`. In this expression, the brackets are introduced uniformly from the right, so we can omit the brackets and unambiguously write `f::t1 -> t2 -> ... -> tn -> t`.

3.2 *More on Defining Functions*

The simplest form of definition is the one we have seen in `sqr`, `plus` and `plus3`, where we just write a defining

equation using an arithmetic expression involving the arguments to the function.

We can also write expressions involving other types. For instance, for values of type `Bool`, the operator `&&` denotes *and*, the operator `||` denotes *or* and the unary operator `not` inverts its argument. Using these, we can define, for instance, the function `xor` that checks that precisely one of its arguments is `True`.

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

3.3 *Pattern Matching*

Haskell does not limit us to a single definition for a function. It offers pattern matching as a convenient mechanism to break up a single definition with many cases into independent multiple definitions that are scanned from top to bottom. The first definition that matches is used to compute the output. For instance, here is an alternative definition of `xor`.

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

Haskell does not limit us to a single definition for a function. It offers pattern matching as a convenient mechanism to break up a single definition with many cases into independent multiple definitions that are scanned from top to bottom.

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

For instance, in the revised definition of `xor`, if we invoke the function as `xor False True`, the first definition does not match, but the second one does. If we

An inductive definition of a function has the following structure:
f(0) is defined explicitly
For $n > 0$, *f*(*n*) is defined in terms of *f*(*n*–1), *f*(*n*–2), ... *f*(0).

invoke the function as `xor True True`, the first two definitions both fail to match and we end up using the third one with `b1` and `b2` set to `True`.

3.4 *Inductive Definitions*

An inductive definition of a function has the following structure:

- $f(0)$ is defined explicitly
- For $n > 0$, $f(n)$ is defined in terms of $f(n-1)$, $f(n-2)$, $\ldots f(0)$.

For instance, we can define the familiar factorial function in this way.

- $factorial(0) = 1$
- For $n > 0$, $factorial(n) = n \cdot factorial(n-1)$

We can use multiple definitions with pattern matching to define a function inductively in Haskell. For instance, here is a definition of the function `factorial`.

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

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

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

Observe the bracketing in the second definition above. We write `n*(factorial (n-1))`. This says we should compute `(n-1)`, then feed this to `factorial` and multiply the result by `n`. If, instead, we write `n*(factorial n-1)`, Haskell would interpret this as `n*((factorial n)-1)` because function application binds more tightly than arithmetic operators.

Notice that there is no guarantee that an inductive definition in Haskell is correct, nor that it terminates on all inputs.

3.5 *Conditional Definitions*

Often, a function definition applies only if certain conditions are satisfied by the values of the inputs. For example, to define `factorial` to work with negative inputs, we could negate negative inputs and invoke `factorial` on the corresponding positive quantity.

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

In this version of `factorial`, the second definition has two options depending on the value of `n`. If `n < 0`, the first definition applies. If `n > 0`, the second definition applies. These conditions are called *guards*, since they restrict entry to the definition that follows. Each guarded definition is signalled using `|`.

Notice that lines beginning with `|` are indented. This tells Haskell that these lines are continuations of the current definition. Haskell uses the layout of the program via indentation to recover the structure of the program, making the use of braces, semicolons and other punctuation optional.

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

The guards in a conditional definition are scanned from

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

top to bottom. They may overlap, in which case the definition that is used is the one corresponding to the first guard that is satisfied. For instance, we could write:

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

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

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

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

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

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

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

4. How Haskell 'Computes'

Haskell computes by rewriting expressions using functions and operators. We say *rewriting* rather than *simplifying* because it is not clear, sometimes, that the rewritten expression is "simpler" than the original one!

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

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

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

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

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

here is how `factorial 3` would be evaluated. We use $\leadsto$ to denote *rewrites to*:

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

In Haskell, the *result* of a computation is an expression that cannot be further simplified. In general, it is guaranteed that any path we follow leads to the same result, if a result is found.

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

```
  factorial 3  ⇝  3 * (factorial (3-1))
               ⇝  3 * (factorial (2))
               ⇝  3 * (2 * factorial (2-1))
⇝ (3 * 2) * (factorial (2-1)) New expression!
               ⇝ 6 * (factorial (2-1)))
               ⇝ ...
               ⇝ 6
```

In Haskell, the *result* of a computation is an expression that cannot be further simplified. In general, it is guaranteed that any path we follow leads to the same result, if a result is found. It could happen that one choice of simplification yields a result while another does not. For instance, consider the following definition of the function `power` that computes x^n for non-negative integer exponents.

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

If we consider the expression `power (8.0/0.0) 0`, we could use the first rule and observe that the value `x` is not used in the final answer, resulting in the following reduction:

`power (8.0/0.0) 0` ⇝ `1.0`

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

`power (8.0/0.0) 0` ⇝ `Error`

Haskell uses a form of simplification that is called *lazy*– it does not simplify the argument to a function until the

value of the argument is actually needed in the evaluation of the function. In particular, Haskell would evaluate the expression above to `1.0`. We will examine the consequences of having such a lazy evaluation strategy in the second part of this article.

Haskell uses a form of simplification that is called *lazy* – it does not simplify the argument to a function until the value of the argument is actually needed in the evaluation of the function.

5. Lists

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

Instead, we need a way to collectively associate a group of values with a variable. In Haskell, the most basic way of collecting a group of values is to form a list. A list is a sequence of values of a fixed type and is written within square brackets separated by commas. Thus, `[1,2,3,1]` is a list of `Int`, while `[True,False,True]` is a list of `Bool`. Lists can be nested: we can have lists of lists. For instance, `[[1,2],[3],[4,4]]` is a list, each of whose members is a list of `Int`. The underlying type of a list must be uniform: we cannot write lists such as `[1,2,True]` or `[[7],8]`.

A list of underlying type `T` has type `[T]`. Thus, `[1,2,3,1]` is of type `[Int]`, `[True,False,True]` is of type `[Bool]`, and `[[1,2],[3],[4,4]]` is of type `[[Int]]`. The empty list is uniformly denoted `[]` for all list types.

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

Internally, Haskell builds lists incrementally, one element at a time, starting with the empty list. This incremental building can be done from left to right (each new element is tagged on at the end of the current list) or from right to left (each new element is tagged on at the beginning of the current list).

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

5.1 *Defining Functions on Lists*

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

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

```
length :: [Int] -> Int
length [] = 0
length l  = 1 + (length (tail l))
```

In general, the inductive step in a list based computation will use both the head and the tail of the list to build up the final value.

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

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

```
sum :: [Int] -> Int
sum [] = 0
sum l  = (head l) + (sum (tail l))
```

5.2 *Using Pattern Matching in List Functions*

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

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

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

By using pattern matching to directly decompose a list into its head and tail, we can avoid having to invoke the built-in functions `head` and `tail` explicitly in most function definitions.

Here is a function `append` that combines two lists into a single larger list. For example `append [3,2] [4,6,7]` should evaluate to `[3,2,4,6,7]`.

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

This is such a useful function that Haskell has a built-in binary operator `++` for this. Thus `[1,2,3] ++ [4,3]` $\rightsquigarrow$ `[1,2,3,4,3]`.

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

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

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

The functions `sum`, `length` and `reverse` are actually basic list functions in Haskell, like the functions `head` and `tail`. Two other useful built-in functions in Haskell are `take` and `drop`. The expression `take n l` returns the first `n` values in `l` while `drop n l` returns the list obtained by omitting the first `n` values in `l`. For any list `l` and any integer `n` we are guaranteed that `l == (take n l) ++ (drop n l)`. In particular, this means that if `n < 0`, `take n l` is `[]` and `drop n l` is `l`, while if `n > (length l)`, `take n l` is `l` and `drop n l` is `[]`.

6. Map, Filter and List Comprehension

Suppose we want to square each element in a list of integers. Assuming we have already defined `sqr`, we could write the following.

```
squareall :: [Int] -> [Int]
squareall [] = []
squareall (x:xs) = (sqr x):(squareall xs)
```

On the other hand, suppose we have a list each of whose elements is itself of type `[Int]`, and we want to compute the length of each of these lists. We could write the following.

```
listlengths :: [[Int]] -> [Int]
listlengths [] = []
listlengths (x:xs) = (length x):(listlengths xs)
```

Notice that these two functions share a similar structure. We have an input list ℓ and a function f that is applied to each element in ℓ. The built-in function `map` allows us to apply a function f 'point wise' to each element of a list. In other words,

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

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

We can now write the functions `squareall` and `listlengths` in terms of `map`:

```
squareall l = map sqr l
listlengths l = map length l
```

Actually, the process of reduction in functional programming can take place anywhere within an expression, so we can always replace 'equals by equals'. This allows us to further simplify these definitions as:

```
squareall = map sqr
listlengths = map length
```

Then, in any context of the form `squareall l`, reduction allows us to replace `squareall` by `map sqr` to get `map sqr l`.

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

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

We have used the built-in function `mod` to check that a number is even: `mod m n` returns the remainder that results when `m` is divided by `n`. A related function is `div` – `div m n` returns the integer part of (`m` divided by `n`). We can think of `evenonly` as the result of applying the test

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

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

Actually, the process of reduction in functional programming can take place anywhere within an expression, so we can always replace 'equals by equals'.

Let l be list and let p be a function from the underlying type of the list to Bool. Then, filter p l retains those elements in l for which p evaluates to True.

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

This is a general principle that we can use to *filter* out values from a list. Let `l` be list and let `p` be a function from the underlying type of the list to `Bool`. Then, `filter p l` retains those elements in `l` for which `p` evaluates to `True`. Thus, we can write `evenonly` as `filter iseven`.

6.1 *List Comprehension: Combining Map and Filter*

In set theory, we can build new sets from old sets using notation called *set comprehension*. For instance, given the set of integers $\{1, 2, \ldots, m\}$, we can define the set of squares of the even numbers in this set as $\{n^2 \mid n \in \{1, 2, \ldots, m\}, even(n)\}$, where $even(n)$ is a predicate that evaluates to true precisely when n is even.

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

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

where `iseven` is the function we wrote earlier to check if `n` is even and `[1..m]` is Haskell's shortcut for denoting the list `[1,2,...,m]`. The notation `<-` is supposed to look like the *element of* notation $\in$ from set theory. This expression is interpreted as follows:

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

The first part, `n <- [1..m]`, is referred to as the generator, which supplies the initial list of values to be operated on. The function `iseven n` serves to filter the list while `n^2` on the left of the `|` is a function that is mapped onto all elements that survive the filter. In fact, it is possible to give a precise translation of list comprehension in terms of `map`, `filter` and a function called

`concat`, which takes a list of lists and "dissolves" one level of brackets, merging its contents into a single long list.

Rather than go into the precise translation of list comprehension in terms of `map`, `filter` and `concat`, we look at some illustrative examples of how to use this notation.

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

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

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

We can now write a function that checks whether `n` is prime, as follows:

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

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

7. Declarative Programming

One advantage of functional programming is that it allows us to describe algorithms in a form that directly reflects the structure of the algorithm. This style is sometimes referred to as *declarative programming*–the program declares what it computes in a natural way. We illustrate the declarative aspect of Haskell by writing functions for three common algorithms for sorting a list.

One advantage of functional programming is that it allows us to describe algorithms in a form that directly reflects the structure of the algorithm. This style is sometimes referred to as *declarative programming* – the program declares what it computes in a natural way.

7.1 *Insertion Sort*

Suppose we had a deck of cards to be arranged in sequence. We begin with the top card and create a new

deck with just one card, that is trivially sorted. We then take each card from the unsorted deck, one at a time, and *insert* it into the correct slot of the partially sorted deck. We can formalize this for lists using the following inductive definition.

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

This sorting algorithm is called insertion sort, which is why we have used the name `isort` for the function. We have used the Haskell construct `where` that allows us to add a new *local* definition attached to the main function.

7.2 *Merge Sort*

A more efficient way to sort lists is to use a technique called *divide and conquer*. Suppose we divide the list into two halves and sort them separately. Can we combine two sorted lists efficiently into a single sorted list? We examine the first element of each list and pick up the smaller one, and continue to combine what remains inductively. We call this process *merging* and define it as follows.

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

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

```
mergesort [] = []
mergesort [x] = [x]
  mergesort l = merge (mergesort (front l))
                      (mergesort (back l))
```

```
  where
   front l = take (div (length l) 2) l
   back l = drop (div (length l) 2) l
```

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

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

7.3 *Quicksort*

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

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

Unfortunately, finding the median value is not easier than sorting the list. However, we can use a variation of this idea by picking up an arbitrary element of the list and using it to *split* the list into a lower half and upper half. This algorithm is called *quicksort*, and is due to C A R Hoare (1961).

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

This definition of quicksort, in particular, shows how effective functional programming is for expressing algorithms declaritively.

This definition of quicksort, in particular, shows how effective functional programming is for expressing algorithms declaritively. In imperative programming languages, the clarity of the underlying algorithm is obfuscated by a scan that swaps elements within the array to partition the given list into the form `lower ++ [splitter] ++ upper`. Programming this partitioning process correctly takes a bit of practice and typically results in novices being afraid of quicksort, despite the fact that it is one of the most efficient ways in practice to sort a list.

8. Experimenting with Haskell

A number of compilers and interpreters are available in the public domain for Haskell. The easiest to install and use is the interpreter `Hugs`, which runs on both Windows and Unix based systems. There is an active website, `http://www.haskell.org`, that has pointers to software, tutorials, books and reference material on Haskell.

9. What next?

In this article, we have described some of the basic features of Haskell. Building on this foundation, we will look, in the next part, at some of the ideas that make functional programming a very powerful and attractive paradigm. These include:

- *Polymorphism* – the ability to define functions that behave in the same way on many different input types.

- *Infinite data types and computations* – these arise as a consequence of lazy evaluation and can be fruitfully used to simplify some types of computations.

- *User defined data types* – how we can define data types such as binary trees and manipulate them directly.

Address for Correspondence
Madhavan Mukund
Chennai Mathematical Institute
H1, SIPCOT IT Park
Padur PO
Siruseri 603 103, India
Email:madhavan@cmi.ac.in
http://www.cmi.ac.in/~madhavan