Beginning with the Haskell Programming Language

About the Tutorial

Navigation

Navigating through the tutorial is easy:

- Use the Next and Previous buttons to move forward and backward.
- Use the Menu button to return to the tutorial menu.
- If you'd like to tell us what you think, use the Feedback button.
- If you need help with the tutorial, use the Help button.

Is this tutorial right for you?

This tutorial targets programmers of *imperative* languages wanting to learn about *functional programming* in the language Haskell. Many users of imperative languages have given little thought to what it means to program in an imperative (also called "procedural") language, and some will be unaware that other *paradigms* of programming exist at all. If you have programmed in languages such as C, Pascal, Fortran, C++, Java, Cobol, Ada, Perl, TCL, REXX, JavaScript, Visual Basic, or many others, you have been using an imperative paradigm. This tutorial provides a gentle introduction to the paradigm of functional programming, with specific illustrations in the Haskell 98 language.

Programmers with a background in functional programming will probably find this tutorial a bit slow; however, programmers who have not used Haskell 98 in particular can still get a quick sense of the language by browsing the tutorial.

About Haskell

Haskell is just one of a number of functional programming languages. Others include Lisp, Scheme, Erlang, Clean, Mercury, ML, OCaml, and others. The common adjunct languages SQL and XSL are also functional. Like functional languages, *logical* or *constraint-based* languages like Prolog are *declarative*. In contrast, both *procedural* and *object-oriented languages* are (broadly speaking) *imperative*. Some languages, such as Python, Scheme, Perl, and Ruby, cross these paradigm boundaries; but, for the most part, programming languages have a particular primary focus.

Among functional languages, Haskell is in many ways the most idealized language. Haskell is a *pure* functional language, which means it eschews all side effects (more later). Haskell has a *non-strict* or *lazy* evaluation model, and is *strictly typed* (but with types that allow ad hoc polymorphism). Other functional languages differ in each of these features--for reasons important to their design philosophies--but this collection of features brings one, arguably, farthest into the functional way of thinking about programs.

On a minor note, Haskell is syntactically easier to get a handle on than are the Lisp-derived languages (especially for programmers who have used lightly punctuated languages like Python, TCL and REXX). Most operators are infixed rather than prefixed. Indentation and module organization *looks* pretty familiar. And perhaps most strikingly, the extreme depth of nested parentheses (as seen in Lisp) is avoided.

Obtaining Haskell

Haskell has several implementations for multiple platforms. These include both an interpreted version called *Hugs*, and several Haskell compilers. The best starting place for all of these is Haskell.org. Links lead to various Haskell implementation. Depending on your operating system, and its packaging system, Haskell may have already been installed, or there may be a standard way to install a ready-to-run version. I recommend those taking this tutorial obtain Hugs for purposes of initial experimentation, and for working along with this tutorial, if you wish to do so.

Additional reading

Two recent books on Haskell bear particular recommendation for readers wishing to learn more.

- *Haskell: The Craft of Functional Programming* (Second Edition). Simon Thompson. Addison-Wesley. 1999.
- *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Paul Hudak. Cambridge University Press. 2000.

What's not covered

Unfortunately, in an introductory tutorial, many of Haskell's most powerful--but complex--features cannot be covered. In particular, the whole area of *type classes* and *algebraic types* (including *abstract data types*) is a bit too much for a first introduction. For readers whose interest is piqued, it is enough to mention that Haskell allows you to create your own data types, and to inherit properties of those data types in *type instances*. The Haskell type system contains the fundamental features of object-oriented programming (inheritance, polymorphism, encapsulation); but in a way that is almost impossible to grasp within a C++/Java/Smalltalk/Eiffel style of thinking.

The other significant element omitted is a discussion of monads, and therefore of I/O. It seems strange to write a tutorial that does not even start with a "Hello World!" program, but thinking in a functional style requires a number of shifts. While that "Hello World!" is quite simple, it also involves the mini "pseudo-imperative" world of monads. It would be easy for a beginner to be lulled in by the pseudo-imperative style of I/O, and miss what is really going on. Swimming is best learned by getting in the water.

Contact

David Mertz is a writer, a programmer, and a teacher, who always endeavors to improve his communication to readers (and tutorial takers). He welcomes any comments; please direct them to mertz@gnosis.cx.

Taking the Vows

Giving things up

The most difficult part of starting to program with Haskell is giving up many of the most familiar techniques and ways of thinking within imperative programming. A first impression is often that it must simply be impossible to write a computer program if you cannot do X, Y, or Z, especially since X, Y, and Z are some of the most common

cannot do A, T, or Z, especially since A, T, and Z are some of the most common patterns in "normal" imperative programming. In this section, let's review a few of the most "shocking" features of Haskell (and of functional programming in general, for the most part).

No mutable variables

One of the most common programming habits in imperative programming is to assign a variable one value, then assign it a different value; perhaps along the way we test whether the variable has obtained certain key values. Constructs like the C examples below are ubiquitous (other imperative languages are similar):

```
if (myVar==37) {...}
myVar += 2
for (myVar=0; myVar<37; myVar++) {...}
```

In Haskell, by contrast, variables of this sort do not exist at all. A name can be bound to a value, but once assigned, the name simply stands for that value throughout the program. Nothing is allowed to change.

In Haskell, "variables" are much like the variables in mathematical equations. They may need to satisfy certain rules, but they are not "counters" or "containers" in the style of imperative programming. Just to get headed in the right way of thinking, consider some linear equations like the ones below as an inspiration:

```
10x + 5y - 7z  + 1 = 0
17x + 5y - 10z + 3 = 0
 5x - 4y + 3z  - 6 = 0
```

In this type of description, we have "unknowns," but the unknowns do not change their value while we are figuring them out.

Isolate side effects

In Haskell, function computation cannot have side effects within the program. Most of the side effects in imperative programs are probably the sort of variable reassignment mentioned in the last panel (whether global variables, or local, or dictionaries, lists, or other storage structures), but every I/O event is also a sort of side-effect. I/O *changes the world* rather than being part of a computation per se. Naturally, there are many times when what you want to do is change the world in some manner (if not, you cannot even know a program has run). Haskell circumscribes all such side effects within a very narrow "box" called **`Monadic IO`**. Nothing in a monad can get out, and nothing outside a monad can get in.

Often, structured imperative programming approaches functional programming's goals of circumscribing I/O. Good design might require that input and output only happens in a limited set of appropriately named functions. Less structured programming tends to read and write to STDIO, files, graphic devices, etc., all over the place and in a way that is difficult to predict. Functional programming takes the circumscription to a much higher level.

No loops

Another interesting feature of Haskell is its lack of any *loop* construct. There is no **`for`** and no **`while`**. There is no **`GOTO`** or **`branch`** or **`jmp`** or **`break`**. One would almost think it impossible to control what a program does without such basic (imperative) constructs;

impossible to control what a program does without such basic (imperative) constructs, but getting rid of these things is actually quite liberating.

The lack of loops is really the same as the matter of no side effects. Since one pass through a loop cannot have variables with different values than another pass, there is nothing to distinguish them; and the need to branch is usually in order to *do* a different program activity. Since functional programming doesn't have *activities*, but only, *definitions*, why bother branching.

However, I should try to stay honest about things. It actually proves possible to simulate almost all of the usual loop constructs, often using the same keywords as in other languages, and in a style that looks surprisingly similar to imperative constructs. Simon Thompson provides many examples of this in his book (see Additional reading).

No program order

Another thing Haskell lacks--or does not need--is program order. The set of definitions that make up a program can occur in any order whatsoever. This might seem strange when definitions look a great deal like assignments in imperative languages. For example, this might seem like a problem:

```
-- Program excerpt
j = 1+i
i = 5

-- Hugs session after loading above program
--   Main> i
--   5 :: Integer
--   Main> j
--   6 :: Integer
```

The thing to understand in a program like the one above is that **`i`** and **`j`** are not *assigned* values, but are rather *defined* in the manners given. In fact, even in the above, **`i`** and **`j`** are functions, and the examples above are of function definitions. In many imperative programming languages, you are also not allowed to define functions multiple times (at least in the same scope).

A New Expressiveness

What's in a Haskell program?

```
myNum :: Int                  -- int myNum() {
myNum = 12+13                 --   return 12+13; }

square :: Int -> Int          -- int square(int n) {
square n = n*n                --   return n*n;  }

double :: Int -> Int          -- int double(int n) {
double n = 2*n                --   return 2*n;  }

dubSqr :: Int -> Int          -- int dubSqr(int n) {
dubSqr n = square (double n)  --   return square(double(n)); }
```

A Haskell program consists, basically, of a set of function definitions. Functions are bound to names in a manner that looks very similar to variable assignment in other languages. However, it really is not the same thing; a Haskell bound name is much more similar to a binding in a mathematical proof, where we might say "Let tau refer to the

similar to a binding in a mathematical proof, where we might say "Let tad refer to the equation . . ." A name binding just provides a shorthand for later use of an equation, but the name can only be bound a single time within a program--trying to change it generates a program error.

Defining functions

```
example :: Int
example = double (myNum - square (2+2))

dubSqr2 :: Int -> Int
dubSqr2 = square . double           -- Function composition
```

There are (optionally) two parts to a function definition. The first part (conceptually, not necessarily within a listing) is the type signature of a function. In a function, the type signature defines all the types of the input, and the type of the output. Some analogous C definitions are given in the end-of-line comments in the example.

The second part of a function definition is the actual computation of the function. In this second part, often (but not always) some ad hoc variables are provided to the left of the equal sign that are involved in the computation to the right. Unlike variables in C, however--and much like variables in mathematics--the Haskell variables refer to the exact same "unknown quantity" on both sides of the equal sign (not to a "container" where a value is held).

It is also often possible to bypass explicit naming of variables entirely in function definitions. In **`dubSqr2`**, it is enough to say that we should **`square`** whatever thing is **`double`**'d. For this, there is no need to mention a variable name since the thing **`dubSqr2`**'d is just whatever expression follows the bound name in later expressions. Of course, **`double`** must itself take the same type of input **`dubSqr2`** expects, and in turn output the type of output **`square`** needs as input.

More simple function definitions

```
-- Average of three Integers as floating point
averageThree :: Int -> Int -> Int -> Float
averageThree l m n = fromInt(l+m+n) / 3
                  -- float averageThree(int l, int m, int n) {
                  --   return ((float)(l+m+n))/3; }

difSquare x y = (x-y)^2      -- C lacks polymorphic type inference
```

Much as in C, Haskell is rigidly typed. The **`averageThree`** is a good example of a function that requires type coercion in order to return the right value type. However, the **`difSquare`** function shows something distinct to Haskell. **`difSquare`** has no type signature, so Haskell will *infer* the appropriate type signature from the operations involved in the function definition. At first appearance this might seem to be the same thing that dynamically or loosely typed languages do; but what Haskell does is quite different. **`difSquare`** is rigidly typed at compile time--there is no runtime dynamism to this, but the type of **`difSquare`** has a *Type Class* that includes both integers and floats (and also rationals, complex numbers, etc.). We can find the inferred type within Hugs:

```
Main> :type difSquare
difSquare :: Num a => a -> a -> a
```

That is, both of the input arguments, as well as the output, are inferred to have the Type Class **`Num`**. Had we explicitly declared a type like **`Int`**, the function would operate over a

Class num. Had we explicitly declared a type like Int, the function would operate over a narrower range of values (which is good or bad, depending on our needs).

Recursion

```
-- Factorial by primitive recursion on decreasing num
fac1 :: Int -> Int
fac1 n = if n==1 then 1 else (n * fac1 (n-1))

-- Factorial by primitive recursion on list tail
fac2 :: Int -> Int
fac2 n = prodList [1 .. n]
prodList lst =
    if (length lst)==1 then head lst
                       else head lst*(prodList (tail lst))
```

Absent loop structures, flow in Haskell programs is usually expressed as recursion. Thinking about all flow in terms of recursion takes some work, but it turns out to be just as expressive and powerful as the **`while`** and **`for`** constructs in other languages.

The trick to recursion is that we would like it to terminate eventually (at least we usually do). One way to guarantee termination of recursion is to use *primitive recursion*. This amounts to taking a "thing" to recurse on, and making sure that the next call is closer to a terminal condition than the call that got us here. In practice, we can assure this either by decrementing an integer for each call (and terminating at zero, or some other goal), or by taking only the **`tail`** of a list for each successive call (and terminating at an empty list). Both versions of factorial listed in the example assume they will be passed an integer greater than zero (and will fail otherwise; exercise: how?).

Non-primitive recursion also exists, but it is more difficult to know for sure that a recursion will terminate. As well, mutual recursion between functions is allowed (and frequently encountered), but primitive recursion is still the safest and most common form.

Pattern matching

```
prodLst2 [] = 0      -- Return 0 as product of empty list
prodLst2 [x] = x     -- Return elem as prod of one-elem list
prodLst2 (x:xs) = x * prodLst2 xs

third (a,b,c,d) = c     -- The third item of a four item tuple
three = third (1,2,3,4) -- 'three' is 3

-- Is a sequence a sub-sequence of another sequence?
isSubseq [] _       = True
isSubseq _ []       = False
isSubseq lst (x:xs) = (lst==start) || isSubseq lst xs
  where start = take (length lst) (x:xs)
```

In functional programming, we are "more concerned with how something is defined than with the specifics of how it is calculated" (take this motto with a grain of salt, however, efficiency still matters in some cases). The idea is that it is a compiler or interpreter's job to figure out how to reach a solution, not the programmer's.

One useful way of specifying how a function is defined is to describe what results it will return given different types of inputs. A powerful way of describing "different types of inputs" in Haskell is using *pattern matching*. We can provide multiple definitions of a function, each having a particular pattern for input arguments. The first listed definition that succeeds in matching a given function call is the one used for that call. In this

that succeeds in matching a given function call is the one used for that call. In this manner, you can pull out the head and tail of a list, match specific input values, identify empty lists as arguments (for recursion usually), and analyze other patterns. You cannot, however, perform value comparisons with pattern matching (e.g., "**`n <= 3`**" must be detected differently). An underscore is used in a position where something should match, but where the matched value is not used in the definition.

Guards

```
prodLst3 lst        -- Guard version of list product
  | length lst==0   = 0
  | length lst==1   = head lst
  | otherwise       = head lst * prodLst3 (tail lst)

-- A sublist is a string that occurs in order, but not
-- necessarily contiguously in another list
isSublist [] _               = True
isSublist _ []               = False
isSublist (e:es) (x:xs)
  | e==x && isSublist es xs  = True
  | otherwise                = isSublist (e:es) xs
```

Somewhat analogous to pattern matching, and also similar to **`if .. then .. else`**, constructs (which we saw examples of earlier) are *guards* in function definitions. A guard is simply a condition that might obtain, and a definition of a function that pertains in that case. Anything that could be stated with pattern matching can also be rephrased into a guard, but guards allow additional tests to be used as well. Whichever guard matches first (in the order listed) becomes the definition of the function for the particular application (other guards might match also, but they are not used for a call if listed later).

In terms of efficiency, pattern matching is usually best, when possible. It is often possible to combine guards with pattern matching, as in the **`isSublist`** example.

List comprehensions

```
-- Odd little list of even i's, multiple-of-three j's,
-- and their product; but limited to i,j elements
-- whose sum is divisible by seven.
myLst :: [(Int,Int,Int)]
myLst = [(i,j,i*j) | i <- [2,4..100],
                     j <- [3,6..100],
                     0==((i+j) `rem` 7)]

-- Quick sort algorithm with list comp and pattern matching
-- '++' is the list concatenation operator; we recurse on both
-- the list of "small" elements and the list of "big" elements
qsort [] = []
qsort (x:xs) = qsort [y | y<-xs, y<=x] ++ [x] ++ qsort [y | y<-xs, y>x
```

One of the most powerful constructs in Haskell is *list comprehensions* (for mathematicians: this term comes from the "Axiom of Comprehension" of Zermelo-Frankel set theory). Like other functional languages, Haskell builds a lot of power on top of manipulation of lists. In Haskell, however, it is possible to generate a list in a compact form that simply states where the list elements come from and what criteria elements meet. Lists described with list comprehensions must be generated from other starting lists; but fortunately, Haskell also provides a quick "enumeration" syntax to specify starting lists.

Lazy evaluation I

```
f x y = x+y             -- Non-lazy function definition
comp1 = f (4*5) (17-12) -- Must compute arg vals in full

g x y = x+37            -- Lazy function definition
comp2 = g (4*5) (17-12) -- '17-12' is never computed

-- Lazy guards and patterns
-- Find the product of head of three lists
prodHeads :: [Int] -> [Int] -> [Int] -> Int
prodHeads [] _ _ = 0    -- empty list gives zero product
prodHeads _ [] _ = 0
prodHeads _ _ [] = 0
prodHeads (x:xs) (y:ys) (z:zs) = x*y*z
-- Nothing computed because empty list matched
comp3 = prodHeads [1..100] [] [n | n <- [1..1000], (n `rem` 37)==0]
-- Only first elem of first and third list computed by lazy evaluation
comp4 = prodHeads [1..100] [55] [n | n <- [1..1000], (n `rem` 37)==0]
```

In imperative languages--and also in some functional languages--expression evaluation is strict and immediate. If you write **`x = y+z;`** in C, for example, you are telling the computer to perform a computation and put a value into the memory called 'x' *right now*! (whenever the code is encountered). In Haskell, by contrast, evaluation is *lazy*--expressions are only evaluated when, and as much, as they need to be (in fairness, C does include shortcutting of Boolean expressions which is a minor kind of laziness). The definitions of **`f`** and **`g`** in the example show a simple form of the difference.

While a function like **`g`** is somewhat silly, since **`y`** is just not used, functions with pattern matching or guards will very often use particular arguments only in certain circumstances. If some arguments have certain properties, those or other arguments might not be necessary for a given computation. In such cases, the needless computations are not performed. Furthermore, when lists are expressed in computational ways (list comprehensions and enumeration ellipsis form), only as many list elements as are actually utilized are actually calculated.

Lazy evaluation II

```
-- Define a list of ALL the prime numbers
primes :: [Int]
primes = sieve [2 .. ]   -- Sieve of Eratosthenes
sieve (x:xs) = x : sieve [y | y <- xs, (y `rem` x)/=0]

-- Given an ordered list and a thing, is the thing in the list?
memberOrd :: Ord a => [a] -> a -> Bool
memberOrd (x:xs) n
  | x<n        = memberOrd xs n
  | x==n       = True
  | otherwise  = False

isPrime n = memberOrd primes n
-- 'isPrime 37' is True
-- 'isPrime 427' is False
```

A truly remarkable thing about Haskell--and about lazy evaluation--is that it is possible to work with *infinite* lists. Not just large ones, but actual infinities! The trick, of course, is that those parts of the list which are unnecessary for a particular calculation are not calculated explicitly (just the rule for their expansion is kept by the runtime

calculated explicitly (just the rule for their expansion is kept by the runtime environment).

A famous and ancient algorithm for finding prime numbers is the Sieve of Eratosthenes. The idea here is to keep an initial element of the list of integers, but strike off all of its multiples as possible primes. The example does this, but is performed only as far as needed for a specific calculation. The list **`primes`**, however, really is exactly the list of **all** the prime numbers!

First class functions (passing functions)

```
-- Quick sort algorithm with arbitrary comparison function
qsortF :: (a -> a -> Bool) -> [a] -> [a]
qsortF f [] = []
qsortF f (x:xs) = qsortF f [y | y<-xs, f y x] ++
                  [x] ++
                  qsortF f [y | y<-xs, not (f y x)]

-- Comparison func that alphabetizes from last letter back
tailComp :: String -> String -> Bool
tailComp s t = reverse s < reverse t

-- List of sample words
myWords = ["foo", "bar", "baz", "fubar", "bat"]

-- tOrd is ["foo","bar","fubar","bat","baz"]
tOrd = qsortF tailComp myWords
-- hOrd is ["bar","bat","baz","foo","fubar"]
hOrd = qsortF (<) myWords
```

A powerful feature of Haskell (as with all functional programming) is that functions are *first class*. What the first class status of functions means is that functions are themselves simply values. Just as you might pass an integer as an argument to a function, in Haskell you can pass another function to a function. To a limited extent, you can do the same with function pointers in a language like C, but Haskell is far more versatile.

The power of Haskell's first class functions lies largely in Haskell's type checking system. In C, one might write a "quicksort" function that accepted a function pointer as an argument, much as in the Haskell example. However, in C you would have no easy way to make sure that the function (pointed to) had the correct *type signature*. That is, whatever function serves as an argument to **`qsortF`** must take two arguments of the same type ("a" stands for a generic type) and produce a **`Bool`** result. Naturally, the list passed as the second argument to **`qsortF`** must also be of the same type "a." Notice also that the type signature given in the sample code is only needed for documentation purposes. If the signature is left out, Haskell infers all these type constraints automatically. **`tailComp`** meets the right type signature, with the type **`String`** being a specialization of the generic type allowed in **`qsortF`** arguments (a different comparison function might operate over a different type or type class).

First class functions (function factories)

```
-- Make an "adder" from an Int
mkAdder n = addN where addN m = n+m
add7 = mkAdder 7    -- e.g. 'add7 3' is 10

-- Make a function from a mapping; first item in pair maps
-- to second item in pair, all other integers map to zero
```

```
mkFunc :: [(Int,Int)] -> (Int -> Int)
mkFunc []         = (\n -> 0)
mkFunc ((i,j):ps) = (\n -> if n==i then j else (mkFunc ps) n)

f = mkFunc [(1,4),(2,3),(5,7)]
-- Hugs session:
--   Main> f 1
--   4 :: Int
--   Main> f 3
--   0 :: Int
--   Main> f 5
--   7 :: Int
--   Main> f 37
--   0 :: Int
```

Passing functions to other functions is only half the power of first class functions. Functions may also act as *factories*, and produce new functions as their results. The ability to create functions with arbitrary capabilities within the program machinery can be quite powerful. For example, one might computationally produce a new comparison function that, in turn, was passed to the **`qsortF`** function in the previous panel.

Often, a means of creating a function is with *lambda notation*. Many languages with functional features use the word "lambda" as the name of the operator, but Haskell uses the backslash character (because it looks somewhat similar to the Greek letter, lambda). A lambda notation looks much like a type signature. The arrow indicates that a lambda notation describes a function from one type of thing (the thing following the backslash) to another type of thing (whatever follows the arrow).

The example factory **`mkFunc`** packs a fair amount into a short description. The main thing to notice is that the lambda indicates a function from **`n`** to the result. By the type signature, everything is an **`Int`**, although type inference would allow a broader type. The form of the function definition is primitive recursive. An empty list produces a result of zero. A non-empty list produces either the result given by its **`head`** pair, or the result that would be produced if only its **`tail`** is considered (and the tail eventually shrinks to empty by recursion).

Modules and Program Structure

Basic syntax

So far in this tutorial, we have seen quite a bit of Haskell code in an informal way. In this final section, we make explicit some of what we've been doing. In fact, Haskell's syntax is extremely intuitive and straightforward. The simplest rule is usually to "write what you mean."

Haskell and literate Haskell

The examples in this tutorial have used the standard Haskell format. In the standard format, comments are indicated with a double dash to their left. All comments in the examples are end-of-line comments, which means that everything following a double dash on a line is a comment. You may also create multi-line comments by enclosing blocks in the pair "{-" and "-}". Standard Haskell files should be named with the **`.hs`** extension.

Literate scripting is an alternative format for Haskell source files. In files named with

Literate scripting is an alternative format for Haskell source files. In files named with the **`.lhs`** extension, all program lines begin with the greater than character. Everything that is not a program line is a comment. This style places an emphasis on program description over program implementation. It looks something like:

```
Factorial by primitive recursion on decreasing num

>    fac1 :: Int -> Int
>    fac1 n = if n==1 then 1 else (n * fac1 (n-1))

Make an "adder" from an Int

>    mkAdder n = addN where addN m = n+m
>    add7 = mkAdder 7
```

The offside rule

Sometimes in Haskell programs, function definitions will span multiple lines and consist of multiple elements. The rule for blocks of elements at the same conceptual level is that they should be indented the same amount. Elements that *belong to* a higher level element should be indented more. As soon as an *outdent* occurs, further lines are promoted back up a conceptual level. In practice, it is obvious, and Haskell will almost always complain on errors.

```
-- Is a function monotonic over Ints up to n?
isMonotonic f n
  = mapping == qsort mapping  -- Is range list the same sorted?
    where                     -- "where" clause is indented below "="
    mapping = map f range     -- "where" definition remain at least as
    range   = [0..n]          --   indented (more would be OK)

-- Iterate a function application n times
iter n f x
  | n == 0     = x            -- Guards are indented below func name
  | otherwise  = f (iter (n-1) f x)
```

I find that two spaces is a nice looking indentation for a subelement, but you have a lot of freedom in formatting for readability (just don't outdent within the same level).

Operator and function precedence

Operators in Haskell fall into multiple levels of precedence. Most of these are the same as you would expect from other programming languages. Multiplication takes precedence over addition, and so on (so "**`2*3+4`**" is 10, not 14). Haskell's standard documentation can provide the details.

There is, however, one "gotcha" in Haskell precedence where it is easy to make a mistake. Functions take precedence over operators. The result of this is that the expression "**`f g 5`**" means "apply **`g`** (and 5) as arguments to f" **not** "apply the result of **`(g 5)`** to **`f`**." Most of the time, this sort of error will produce a compiler error message, since, for example, **`f`** will require an **`Int`** as an argument rather than another function. However, sometimes the situation can be worse than this, and you can write something valid but wrong:

```
double n = n*2
res1 = double 5^2         -- 'res1' is 100, i.e. (5*2)^2
res2 = double (5^2)       -- 'res2' is 50, i.e. (5^2)*2
res3 = double double 5    -- Causes a compile-time error
```

```
res4 = double (double 5)  -- 'res4 is 20, i.e. (5*2)*2
```

As with other languages, parentheses are extremely useful in disambiguating expressions where you have some doubt about precedence (or just want to document the intention explicitly). Notice, by the way, that parentheses are *not* used around function arguments in Haskell; but there is no harm in pretending they are, which just creates an extra expression grouping (as in **`res2`** above).

Scope of names

Readers might think there is a conflict between two points in this tutorial. On the one hand, we have said that names are defined as expressions only once in a program; on the other hand, many of the examples use the same variable names repeatedly. Both points are true, but need to be refined.

Every name is defined exactly once *within a given scope*. Every function definition defines its own scope, and some constructs within definitions define their own narrower scopes. Fortunately, the "offside rule" that defines subelements also precisely defines variable scoping. A variable (a name, really) can only occur once *with a given indentation block*. Let's see an example, much like previous ones:

```
x x y                   -- 'x' as arg is in different scope than func
  | y==1      = y*x*z   -- 'y' from arg scope, but 'x' from 'where' sc
  | otherwise = x*x     -- 'x' comes from 'where' scope
  where
  x = 12                -- define 'x' within the guards
  z = 5                 -- define 'z' within the guards
n1 = x 1 2              -- 'n1' is 144 ('x' is the function name)
n2 = x 33 1             -- 'n2' is 60  ('x' is the function name)
```

Needless to say, the example is unnecessarily confusing. It is worth understanding, however, especially since arguments only have a scope within a particular function definition (and the same names can be used in other function definitions).

Breaking down the problem

One thing you will have noticed is that function definitions in Haskell tend to be extremely short compared to those in other languages. This is partly due to the concise syntax of Haskell, but a greater reason is because of the emphasis in functional programming of breaking down problems into their component parts (rather than just sort of "doing what needs to be done" at each point in an imperative program). This encourages reusability of parts, and allows much better verification that each part really does what it is supposed to do.

The small parts of function definitions may be broken out in several ways. One way is to define a multitude of useful support functions within a source file, and use them as needed. The examples in this tutorial have mostly done this. However, there are also two (equivalent) ways of defining support functions within the narrow scope of a single function definition: the **`let`** clause and the **`where`** clause. A simple example follows.

```
f n = n+n*n
f2 n
  = let sq = n*n
    in n+sq
f3 n
  = n+sq
  where sq = n*n
```

```
  where sq = n*n
```

The three definitions are equivalent, but **f2** and **f3** chose to define a (trivial) support function **sq** within the definition scope.

Importing/exporting

Haskell also supports a module system that allows for larger scale modularity of functions (and also for types, which we have not covered in this introductory tutorial). The two basic elements of module control are specification of imports and specification of exports. The former is done with the **import** declaration; the latter with the **module** declaration. Some examples include:

```
-- declare the current module, and export only the objects listed
module MyNumeric ( isPrime, factorial, primes, sumSquares ) where

import MyStrings    -- import everything MyStrings has to offer
                    -- import only listed functions from MyLists
import MyLists ( quicksort, findMax, satisfice )
                    -- import everything in MyTrees EXCEPT normalize
import MyTrees hiding ( normalize )
                    -- import MyTuples as qualified names, e.g.
                    --   three = MyTuples.third (1,2,3,4,5,6)
import qualified MyTuples
```

You can see that Haskell provides considerable, and fine-grained control of where function definitions are visible to other functions. This module system helps build