CSci 450: Org. of Programming Languages
Basic Haskell Functional Programming

H. Conrad Cunningham

26 October 2017

Contents

Acknowledgements: I adapted and revised much of this work in Summer and Fall 2016 from my previous materials.

- Defining Our First Haskell Functions from chapter 3 of my *Notes on Functional Programming with Haskell*
- Surveying the Basic Types from chapter 5 of my *Notes on Functional Programming with Haskell*
- Using Top-Down Stepwise Refinement (square root module) from my earlier implementations of this algorithm in Scala, Elixir, and Lua and from section 1.1.7 of Abelson and Sussman's *Structure and Interpretation of Computer Programs*.
- Using Data Abstraction (rational arithmetic module) mostly from chapter 5 of my *Notes on Functional Programming with Haskell*, from my Lua-based implementations, and from section 2.1 of Abelson and Sussman's *Structure and Interpretation of Computer Programs*.
- Modular Design and Programming from my Data Abstraction and Modular Design notes

In 2017, I continue to develop this chapter.

I maintain these notes as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to HTML, PDF, and other forms as needed.

Advisory: The HTML version of this document may require use of a browser that supports the display of MathML. A good choice as of October 2017 is a recent version of Firefox from Mozilla.

TODO:

- Discuss Haskell modules along with Factorial, give links to code
- Discuss testing along with Factorial, give links to code
- Perhaps move introductory discussion of module back to the Factorial subsection
- Better integrate the general discussion of data abstraction and modular design
- Add more examples
- Edit and add more exercises

2 Basic Haskell Functional Programming

2.1 Chapter Introduction

This chapter introduces the basic features of the Haskell language needed for programming using first-order functions, primitive data types, and tuples. The goal is for the students to be able to use these features to develop small Haskell programs.

Upon successful completion of this chapter, students should be able to:

1. describe the basic syntax and semantics of first-order Haskell functions
2. describe the basic Haskell data types
3. execute Haskell programs from the REPL
4. develop first-order Haskell functional programs using recursion and the primitive data types and tuples
5. apply top-down refinement and data abstraction to develop Haskell modules
6. relate the basic Haskell features to similar features in other languages

2.2 Defining Our First Haskell Functions

Let's look at our first function definition in the Haskell language, a program to implement the factorial function for natural numbers.

The Haskell source file `Fact.hs` holds the Haskell function definitions for this section. The source file `TestFact.hs` gives a simple test script.

2.2.1 Factorial function specification

We can give two mathematical definitions of factorial, *fact* and *fact'*, that are equivalent for all natural number arguments. We can define *fact* using the product operator as follows:

$$fact(n) = \prod_{i=1}^{i=n} i$$

For example,

$$fact(4) = 1 \times 2 \times 3 \times 4.$$

By definition

$$fact(0) = 1$$

which is the *identity* element of the multiplication operation.

We can also define the factorial function *fact'* with a *recursive* definition (or *recurrence relation*) as follows:

$$fact'(n) = 1, \text{ if } n = 0$$
$$fact'(n) = n \times fact'(n-1), \text{ if } n \geq 1$$

Since the domain of *fact'* is the set of natural numbers, a set over which induction is defined, we can easily see that this recursive definition is well defined.

- For $n = 0$, the base case, the value is simply 1.
- For $n \geq 1$, the value of $fact'(n)$ is recursively defined in terms of $fact'(n-1)$. The argument of the recursive application decreases toward the base case.

In a previous chapter, we proved that $fact(n) = fact'(n)$ by mathematical induction.

The Haskell functions defined in the following subsections must compute $fact(n)$ when applied to argument value n.

2.2.2 Factorial function using if-then-else: `fact1`

One way to translate the recursive definition *fact'* into Haskell is the following:

```
fact1 :: Int -> Int
fact1 n = if n == 0 then
              1
          else
              n * fact1 (n-1)
```

- The first line above is the *type signature* for function `fact1`. In general, type signatures have the syntax *object* `::` *type*.

  Haskell type names begin with an uppercase letter.

  The above defines object `fact1` as a function (denoted by the `->` symbol) that takes one argument of type integer (denoted by the first `Int`) and returns a value of type integer (denoted by the last `Int`).

  Haskell does not have a built-in natural number type. Thus we choose type `Int` for the argument and result of `fact1`.

  The `Int` data type is a bounded integer type, usually the integer data type supported directly by the host processor (e.g., 32- or 64-bits on most current processors), but it is guaranteed to have the range of at least a 30-bit, two's complement integer (-2^{29} to 2^{29}).

- The declaration for the function `fact1` begins on the second line. Note that it is an equation of the form

fname parms `=` *body*

where *fname* is the function's name, *parms* are the function's parameters, and *body* is an expression defining the function's result.

Function and variable names begin with lowercase letters optionally followed by a sequence of characters each of which is a letter, a digit, an apostrophe (`'`) (sometimes pronounced "prime"), or an underscore (`_`).

A function may have zero or more parameters. The parameters are listed after the function name without being enclosed in parentheses and without commas separating them.

The parameter names may appear in the *body* of the function. In the evaluation of a function *application* the actual argument values are substituted for parameters in the *body*.

- Above we define the *body* function `fact1` to be an `if-then-else` *expression*. This kind of expression has the form

  `if` condition `then` expression1 `else` expression2

  where

  > *condition* is a Boolean expression, that is, an expression of Haskell type `Bool`, which has either `True` or `False` as its value
  >
  > *expression1* is the expression that is returned when the condition is `True`
  >
  > *expression2* is the expression (with the same type as *expression1*) that is returned when the condition is `False`

  Evaluation of the `if-then-else` expression in `fact1` yields the value `1` if argument `n` has the value `0` (i.e., `n == 0`) and yields the value `n * fact1 (n-1)` otherwise.

- The `else` clause includes a recursive application of `fact1`. The whole expression `(n-1)` is the argument for the recursive application, so we enclose it in parenthesis.

  The value of the argument for the recursive application is less than the value of the original argument. For each recursive application of `fact` to a natural number, the argument's value thus moves closer to the termination value `0`.

- Unlike most conventional languages, the *indentation is significant* in Haskell. The indentation indicates the nesting of expressions.

  For example, in `fact1` the `n * fact1 (n-1)` expression is nested inside the `else` clause of the `if-then-else` expression.

- This Haskell function does not match the mathematical definition given above. What is the difference?

Notice the domains of the functions. The evaluation of `fact1` will go into an "infinite loop" and eventually abort when it is applied to a negative value.

In Haskell there is *only* one way to form more complex expressions from simpler ones: *apply* a function.

Neither parentheses nor special operator symbols are used to denote function application; it is denoted by simply listing the argument expressions following the function name. For example, a function `f` applied to argument expressions `x` and `y` is written in the following *prefix* form:

```
f x y
```

However, the usual prefix form for a function application is not a convenient or natural way to write many common expressions. Haskell provides a helpful bit of syntactic sugar, the *infix* expression. Thus instead of having to write the addition of `x` and `y` as

```
add x y
```

we can write it as

```
x + y
```

as we have since elementary school. Here the symbol `+` represents the addition function.

Function application (i.e., juxtaposition of function names and argument expressions) has higher precedence than other operators. Thus the expression `f x + y` is the same as `(f x) + y`.

2.2.3 Factorial function using guards: `fact2`

An alternative way to differentiate the two cases in the recursive definition is to use a different equation for each case. If the Boolean *guard* (e.g., `n == 0`) for an equation evaluates to true, then that equation is used in the evaluation of the function. A guard is written following the `|` symbol as follows:

```
fact2 :: Int -> Int
fact2 n
  | n == 0    = 1
  | otherwise = n * fact2 (n-1)
```

Function `fact2` is equivalent to the `fact1`. Haskell evaluates the guards in a top-to-bottom order. The `otherwise` guard always succeeds; thus it's use above is similar to the trailing `else` clause on the `if-then-else` expression used in `fact1`.

2.2.4 Factorial function using pattern matching: `fact3` and `fact4`

Another equivalent way to differentiate the two cases in the recursive definition is to use *pattern matching* as follows:

```
fact3 :: Int -> Int
fact3 0 = 1
fact3 n = n * fact3 (n-1)
```

The parameter pattern `0` in the first *leg* of the definition only matches arguments with value 0. Since Haskell checks patterns and guards in a top-to-bottom order, the `n` pattern matches all nonzero values. Thus `fact1`, `fact2`, and `fact3` are equivalent.

To stop evaluation from going into an "infinite loop" for negative arguments, we can remove the negative integers from the function's domain. One way to do this is by using guards to narrow the domain to the natural numbers as in the definition of `fact4` below:

```
fact4 :: Int -> Int
fact4 n
    | n == 0 =  1
    | n >= 1 =  n * fact4 (n-1)
```

Function `fact4` is undefined for negative arguments. If `fact4` is applied to a negative argument, the evaluation of the program encounters an error quickly and returns without going into an infinite loop. It prints an error and halts further evaluation.

We can define our own error message for the negative case using an `error` call as in `fact4'` below.

```
fact4' :: Int -> Int
fact4' n
  | n == 0    =  1
  | n >= 1    =  n * fact4' (n-1)
  | otherwise = error "fact4' called with negative argument"
```

In addition to displaying the custom error message, this also displays a stack trace of the active function calls.

2.2.5 Factorial function using a built-in library function: `fact5`

The four definitions we have looked at so far use recursive patterns similar to the recurrence relation *fact'*. Another alternative is to use the library function `product` and the list-generating expression `[1..n]` to define a solution that is like the function *fact*:

```
fact5 :: Int -> Int
```

```
fact5 n = product [1..n]
```

The list expression `[1..n]` generates a *list* of consecutive integers beginning with `1` and ending with `n`. We study lists in a later chapter.

The library function `product` computes the product of the elements of a finite list.

If we apply `fact5` to a negative argument, the expression `[1..n]` generates an empty list. Applying `product` to this empty list yields 0, which is the identity element for multiplication. Defining `fact5` to return 0 is consistent with the function *fact* upon which it is based.

Which of the above definitions for the factorial function is better?

Most people in the functional programming community would consider `fact4` (or `fact4'`) and `fact5` as being better than the others. The choice between them depends upon whether we want to trap the application to negative numbers as an error or to return the value 1.

2.3 Using the Glasgow Haskell Compiler (GHC)

See the Glasgow Haskell Compiler Users Guide for information on the Glasgow Haskell Compiler (GHC) and its use.

GHCi is an environment for using GHC interactively. That is, it is a REPL (Read-Evaluate-Print-Loop) command line interface using Haskell. The "Using GHCi" chapter of the User Guide describes its usage.

Below, we show a GHCi session where we load source code file (module) `Fact.hs` and apply the factorial functions to various inputs. The instructor ran this in a Terminal session on an iMac running macOS 10.12.6 (Sierra) with `ghc` 8.2.1 installed.

1. Start the REPL.

   ```
   bash-3.2$ ghci
   GHCi, version 8.2.1: http://www.haskell.org/ghc/  :? for help
   ```

2. Load module `Fact` that holds the factorial function definitions. This assumes the `Fact.hs` file is in the current directory. The `load` command can be abbreviated as just `:l`.

   ```
   Prelude> :load Fact
   [1 of 1] Compiling Fact             ( Fact.hs, interpreted )
   Ok, 1 module loaded.
   ```

3. Inquire about the type of `fact1`.

   ```
   *Fact> :type fact1
   fact1 :: Int -> Int
   ```

4. Apply function `fact1` to 7, 0, 20, and 21. Note that the factorial of 21 exceeds the `Int` range.

```
*Fact> fact1 7
5040
*Fact> fact1 0
1
*Fact> fact1 20
2432902008176640000
*Fact> fact1 21
-4249290049419214848
```

5. Apply functions `fact2`, `fact3`, `fact4` and `fact5` to 7.

```
*Fact> fact2 7
5040
*Fact> fact3 7
5040
*Fact> fact4 7
5040
*Fact> fact5 7
5040
```

6. Apply functions `fact1`, `fact2`, and `fact3` to -1. All go into an infinite recursion, eventually terminating with an error when the runtime stack overflows its allocated space.

```
*Fact> fact1 (-1)
*** Exception: stack overflow
*Fact> fact2 (-1)
*** Exception: stack overflow
*Fact> fact3 (-1)
*** Exception: stack overflow
```

7. Apply functions `fact4` and `fact4'` to -1. They quickly return with an error.

```
*Fact> fact4 (-1)
*** Exception: Fact.hs:(33,1)-(35,29): Non-exhaustive patterns
    in function fact4
*Fact> fact4' (-1)
*** Exception: fact4' called with negative argument
CallStack (from HasCallStack):
  error, called at Fact.hs:50:17 in main:Fact
```

8. Apply function `fact5` to -1. It returns a 1 because it is defined for negative integers.

```
*Fact> fact5 (-1)
1
```

9. Set the `+s` option to get information about the time and space required and the `+t` option to get the type of the returned value.

```
*Fact> :set +s
*Fact> fact1 20
2432902008176640000
(0.00 secs, 80,672 bytes)
*Fact> :set +t
*Fact> fact1 20
2432902008176640000
it :: Int
(0.00 secs, 80,720 bytes)
*Fact> :unset +s +t
*Fact> fact1 20
2432902008176640000
```

10. Exit GHCi.

```
:quit
Leaving GHCi.
```

Suppose we had set the environment variable `EDITOR` to our favorite text editor in the Terminal window. For example, on a Mac OS system, your instructor might give the following command in shell (or in a startup script such as `.bash_profile`):

```
export EDITOR=Aquamacs
```

Then the `:edit` command within GHCi allows us to edit the source code. We can give a filename or default to the last file loaded.

```
:edit
```

Or we could also use a `:set` command to set the editor within GHCi.

```
:set editor Aquamacs
...
:edit
```

See the Glasgow Haskell Compiler Users Guide for more information about use of GHC and GHCi.

2.4 Surveying the Basic Types

The type system is an important part of Haskell; the compiler or interpreter uses the type information to detect errors in expressions and function definitions. To each expression Haskell assigns a *type* that describes the kind of value represented by the expression.

Haskell has both built-in types (defined in the language or its standard libraries) and facilities for defining new types. In the following we discuss the primary built-in types. As we have seen, a Haskell type name begins with a capital letter.

In these notes, we sometimes refer to the types `Int`, `Float`, `Double`, `Bool`, and `Char` as being *primitive* because they likely have direct support in the host processor's hardware.

2.4.1 Integers: `Int` and `Integer`

The `Int` data type is usually an integer data type supported directly by the host processor (e.g., 32- or 64-bits on most current processors), but it is guaranteed to have the range of at least a 30-bit, two's complement integer.

The type `Integer` is an unbounded precision integer type. Unlike `Int`, host processors usually do not support this type directly. The Haskell library or runtime system typically supports this type in software.

Haskell supports the usual integer literals (i.e., constants) and operations such as `+`, `-`, `*`, `<`, etc.

For integer division, Haskell provides functions `div` and `rem` such that `div m n` returns the integral quotient from dividing `m` by `n` and `rem m n` returns the remainder.

2.4.2 Floating point numbers: `Float` and `Double`

The `Float` and `Double` data types are usually the single and double precision floating point numbers supported directly by the host processor.

Haskell floating point literals must include a decimal point; they may be signed or in scientific notation: `3.14159`, `2.0`, `-2.0`, `1.0e4`, `5.0e-2`, `-5.0e-2`.

Haskell supports the usual operations on floating point numbers. Division is denoted by `/` as usual.

2.4.3 Booleans: `Bool`

The `Bool` data type is usually supported directly by the host processor as one or more contiguous bits.

The Boolean literals are `True` and `False` (note capitals). Haskell supports Boolean operations such as `&&` (and), `||` (or), and `not`.

Functions can match against patterns using the Boolean constants. For example, we could define a function `myAnd` as follows:

```
myAnd :: Bool -> Bool -> Bool
myAnd True  b = b
myAnd False _ = False
```

Above the pattern `_` is a wildcard that matches any value but does not bind a value that can be used on the right-hand-side of the definition.

2.4.4 Characters: `Char`

The `Char` data type is usually supported directly by the host processor by one or more contiguous bytes.

Haskell uses Unicode for its character data type. Haskell supports character literals enclosed in single quotes—including both the graphic characters (e.g., `'a'`, `'0'`, and `'Z'`) and special codes entered following the escape character backslash `\` (e.g., `'\n'` for newline, `'\t'` for horizontal tab, and `'\\'` for backslash itself).

In addition, a backslash character `\` followed by a number generates the corresponding Unicode character code. If the first character following the backslash is `o`, then the number is in octal representation; if followed by `x`, then in hexadecimal notation; and otherwise in decimal notation.

For example, the exclamation point character can be represented in any of the following ways: `'!'`, `'\33'`, `'\o41'`, `'\x21'`

2.4.5 Functions: `t1 -> t2`

If `t1` and `t2` are types then `t1 -> t2` is the type of a function that takes an argument of type `t1` and returns a result of type `t2`.

Function and variable names begin with lowercase letters optionally followed by a sequences of characters each of which is a letter, a digit, an apostrophe (`'`) (sometimes pronounced "prime"), or an underscore (`_`).

Haskell functions are *first-class* objects. They can be arguments or results of other functions or be components of data structures. Multi-argument functions are *curried*—that is, treated as if they take their arguments one at a time.

For example, consider the integer addition operation `(+)`. (Surrounding the binary operator symbol with parentheses refers to the corresponding function.) In mathematics, we normally consider addition as an operation that takes a *pair* of integers and yields an integer result, which would have the type signature

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

In Haskell, we give the addition operation the type

```
(+) :: Int -> (Int -> Int)
```

or just

```
(+) :: Int -> Int -> Int
```

since Haskell binds `->` from the right.

Thus `(+)` is a one argument function that takes some `Int` argument and returns a function of type `Int -> Int`. Hence, the expression `((+) 5)` denotes a function that takes one argument and returns that argument plus `5`.

We sometimes speak of this `(+)` operation as being *partially applied* (i.e., to one argument instead of two).

This process of replacing a structured argument by a sequence of simpler ones is called *currying*, named after American logician Haskell B. Curry who first described it.

The Haskell library, called the *standard prelude* (or just Prelude), contains a wide range of predefined functions including the usual arithmetic, relational, and Boolean operations. Some of these operations are predefined as *infix* operations.

2.4.6 Tuples: `(t1,t2,...,tn)`

If `t1`, `t2`, $\cdots$, `tn` are types, where `n` is finite and `n >= 2`, then is a type consisting of *n-tuples* where the various components have the type given for that position.

Each element in a tuple may have different types. The number of elements in a tuple is fixed.

Examples of tuple values with their types include the following:

```
('a',1) :: (Char,Int)
(0.0,0.0,0.0) :: (Double,Double,Double)
(('a',False),(3,4)) :: ((Char, Bool), (Int, Int))
```

We can also define a *type synonym* using the `type` declaration and the use the synonym in further declarations as follows:

```
type Complex = (Float,Float)
makeComplex :: Float -> Float -> Complex
makeComplex r i = (r,i)`
```

A type synonym does not define a new type, but it introduces an alias for an existing type. We can use `Complex` in declarations, but it has the same effect as using `(Float,Float)` expect that `Complex` provides better documentation of the intent.

2.4.7 Lists: `[t]`

The primary built-in data structure in Haskell is the *list*, a sequence of values. All the elements in a list must have the same type. Thus we declare lists with

notation such as `[t]` to denote a list of zero or more elements of type `t`.

A list literal is a comma-separated sequence of values enclosed between `[` and `]`. For example, `[]` is an empty list and `[1,2,3]` is a list of the first three positive integers in increasing order.

We will look at programming with lists in a later chapter.

2.4.8 Strings: `String`

In Haskell, a *string* is just a list of characters. Thus Haskell defines the data type `String` as a *type synonym* :

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

We examine lists and strings in a later chapter, but, because we use strings in a few examples in this subsection, let's consider them briefly.

A `String` literal is a sequence of zero or more characters enclosed in double quotes, for example, `"Haskell programming"`.

Strings can contain any graphic character or any special character given as escape code sequence (using backslash). The special escape code `\&` is used to separate any character sequences that are otherwise ambiguous.

For example, the string literal `"Hotty\nToddy!\n"` is a string that has two newline characters embedded.

Also the string literal `"\12\&3"` represents the two-element list `['\12','3']`.

Because strings are represented as lists, all of the Prelude functions for manipulating lists also apply to strings. We look at manipulating lists and strings in later chapters of these notes.

2.5 Using Top-Down Stepwise Refinement

As we saw in a previous chapter, there are two processes of abstraction in program design and implementation.

Procedural abstraction: the separation of the logical properties of an *action* from the details of how the action is implemented.

Data abstraction: the separation of the logical properties of *data* from the details of how the data are represented.

This section focuses on procedural abstraction. Later sections focus on data abstraction.

A useful and intuitive design process for a small program is to begin with a high-level solution and incrementally fill in the details. We call this process top-down stepwise refinement. Here we introduce it with an example.

2.5.1 Developing a square root package

Consider the problem of computing the nonnegative square root of a nonnegative number x. Mathematically, we want to find the number y such that

$$y \geq 0 \text{ and } y^2 = x.$$

A common algorithm in mathematics for computing the above y is to use Newton's method of successive approximations, which has the following steps for square root:

1. Guess at the value of y.
2. If the current approximation (guess) is sufficiently close (i.e. good enough), return it and stop; otherwise, continue.
3. Compute an improved guess by averaging the value of the guess y and x/y, then go back to step 2.

To encode this algorithm in Haskell, we work top down to decompose the problem into smaller parts until each part can be solved easily. We begin this *top-down stepwise refinement* by defining a function with the type signature:

```
sqrtIter :: Double -> Double -> Double
```

We choose type `Double` (double precision floating point) to approximate the real numbers. Thus we can encode step 2 of the above algorithm in Haskell as follows:

```
sqrtIter guess x
    | goodEnough guess x = guess
    | otherwise          = sqrtIter (improve guess x) x
```

We define `sqrtIter` to take two arguments–the current approximation `guess` and number `x` for which we need the square root. We have two cases:

- When the current approximation `guess` is sufficiently close to `x`, we return `guess`.

  We abstract this decision into a separate function `goodEnough` with type signature:

  ```
  goodEnough :: Double -> Double -> Bool
  ```

- When the approximation is not yet close enough, we reduce the problem to another application of `sqrtIter` itself to an improved approximation.

  We abstract the improvement process into a separate function `improve` with type signature:

  ```
  improve :: Double -> Double -> Double
  ```

  To ensure termination of `sqrtIter`, the argument `(improve guess x)` on the recursive call must get closer to a value that satisfies its base case.

The function `improve` takes the current `guess` and `x` and carries out step 3 of the algorithm, thus averaging `guess` and `x/guess`, as follows:

```
improve :: Double -> Double -> Double
improve guess x = average guess (x/guess)
```

Here we abstract `average` into a separate function as follows:

```
average :: Double -> Double -> Double
average x y = (x + y) / 2
```

The new guess is closer to the square root than the previous guess. Thus the algorithm will terminate assuming a good choice for function `goodEnough`, which guards the base case of the `sqrtIter` recursion.

How should we define `goodEnough`? Given that we are working with the limited precision of computer floating point arithmetic, it is not easy to choose an appropriate test for all situations. Here we simplify this and use a tolerance of 0.001.

We thus postulate the following definition for `goodEnough`:

```
goodEnough :: Double -> Double -> Bool
goodEnough guess x = abs (square guess - x) < 0.001
```

In the above, `abs` is the built-in absolute value function defined in the standard Prelude library. We define square as the following simple function (but could replace it by just `guess * guess`):

```
square :: Double -> Double
square x = x * x
```

What is a good initial guess? It is sufficient to just use 1. So we can define an overall square root function `sqrt'` as follows:

```
sqrt' :: Double -> Double
sqrt' x | x >= 0 = sqrtIter 1 x
```

(A square root function `sqrt` is defined in the Prelude library, so a different name is needed to avoid the name clash.)

2.5.2 Making the package a Haskell module

We can make this package into a Haskell module by putting the definitions in a file (e.g., named `Sqrt.hs`) and adding a module header at the beginning as follows:

```
module Sqrt
    (sqrt')
where
  -- give the definitions above for functions sqrt',
```

```
--   sqrtIter, improve, average, and goodEnough,
```

The header gives the module the name `Sqrt` and defines the names in parenthesis as being *exported* to other modules that *import* this module. The other symbols (e.g., `sqrtIter`, `goodEnough`) are local to (i.e., hidden inside) the module.

In the above Haskell code, the symbol "`--`" denotes the beginning of an end-of-line comment. All text after that symbol is text ignored by the Haskell compiler.

The Haskell module for the Square root case study is in file `Sqrt.hs`. Limited testing code is in module `TestSqrt.hs`.

2.5.3 Top-down stepwise refinement

The program design strategy known as *top-down stepwise refinement* is a relatively intuitive design process that has long been applied in the design of structured programs in imperative procedural languages. It is also useful in the functional setting.

In Haskell, we can apply top-down stepwise refinement as follows.

1. Start with a high-level solution to the problem consisting of one or more functions. For each function, identify its type signature and functional requirements (i.e., its inputs, outputs, and termination condition).

   Some parts of each function are abstracted as "pseudocode" expressions or as-yet-undefined function calls.

2. Choose one of the incomplete parts. Consider its type signature and functional requirements. Refine the incomplete part by breaking it into subparts or, if simple, defining it directly in terms of Haskell expressions (including calls to the Prelude or other available library functions).

   When refining an incomplete part, consider the various options according to the relevant design criteria (e.g., time, space, generality, understandability, elegance, etc.)

   The refinement of the function may require a refinement of the data being passed. If so, back up in the refinement process and readdress previous design decisions as needed.

   If it not possible to design an appropriate refinement, back up in the refinement process and readdress previous design decisions.

3. Continue step 2 until all parts are fully defined in terms of Haskell code and data and the resulting set of functions meets all required criteria.

For as long as possible, we should stay with terminology and notation that is close to the problem being solved. We can do this by choosing appropriate

function names and signatures and data types. (In later chapters, we examine Haskell's rich set of builtin and user-defined types.)

For stepwise refinement to work well, we must be willing to back up to earlier design decisions when appropriate. We should keep good documentation of the intermediate design steps.

The stepwise refinement method can work well for small programs, but it may not scale well to large, long-lived, general purpose programs. In particular, stepwise refinement may lead to a module structure in which modules are tightly coupled and not robust with respect to changes in requirements. A combination of techniques may be needed to develop larger software systems.

2.6 Using Data Abstraction

A design technique that can help make a program robust with respect to change in the data is to use data abstraction. As in the previous subsection, let's begin with an example.

2.6.1 Rational number arithmetic

For this example, let's implement a group of Haskell functions to perform rational number arithmetic, assuming that the Haskell library does not contain such a data type.

In mathematics we usually write rational numbers in the form $\frac{\texttt{x}}{\texttt{y}}$ where `x` and `y` are integers and $\texttt{y} \neq \texttt{0}$.

For now, let's assume we have a special type `Rat` to represent rational numbers and a constructor function

```
makeRat :: Int -> Int -> Rat
```

to create a rational number instance from its numerator `x` and denominator `y`. That is, `makeRat x y` constructs rational number $\frac{\texttt{x}}{\texttt{y}}$.

Further, let us assume we have selector functions `numer` and `denom` with signatures

```
numer, denom :: Rat -> Int
```

that each take a `Rat` argument and return the numerator and denominator, respectively. That is, they satisfy the equalities:

```
numer (makeRat x y) == x
denom (makeRat x y) == y
```

We consider how to implement rational numbers in Haskell later, but for now let's look at rational arithmetic using the constructor and selector functions above.

Given the knowledge of rational arithmetic from mathematics, we can define the operations for unary negation, addition, subtraction, multiplication, division, and equality.

```
negRat :: Rat -> Rat
negRat x = makeRat (- numer x) (denom x)

addRat, subRat, mulRat, divRat :: Rat -> Rat -> Rat
addRat x y = makeRat (numer x * denom y + numer y * denom x)
                     (denom x * denom y)   -- x + y
subRat x y = makeRat (numer x * denom y - numer y * denom x)
                     (denom x * denom y)   -- x - y
mulRat x y = makeRat (numer x * numer y)
                     (denom x * denom y)   -- x * y
divRat x y = makeRat (numer x * denom y)
                     (denom x * numer y)   -- x / y

eqRat :: Rat -> Rat -> Bool
eqRat x y = (numer x) * (denom y) == (numer y) * (denom x)
```

Above we give the type signatures for all four functions in the same type declaration by listing them separated by commas.

These functions all use the type `Rat`, constructor function `makeRat`, and selector functions `numer` and `denom` assumed above. They do not depend upon any specific representation for rational numbers.

The above six functions work on rational numbers as a *data abstraction* defined by the type `Rat`, constructor function `makeRat`, and selector functions `numer` and `denom`.

The goal of a data abstraction is to separate the logical properties of *data* from the details of how the data are represented.

2.6.2 Rational number data representation

Now, how can we represent rational numbers?

For this package, we define a type synonym `Rat` to denote this type:

```
type Rat = (Int, Int)
```

For example, `(1,7)`, `(-1,-7)`, `(3,21)`, and `(168,1176)` all represent $\frac{1}{7}$.

As with any value that can be expressed in many different ways, it is useful to define a single *canonical* (or *normal*) form for representing values in the rational number type `Rat`.

It is convenient for us to choose a rational number representation `(x,y)` that satisfies the following property, which we call an *invariant*:

`y > 0`, `x` and `y` are relatively prime, and zero is denoted uniquely by `(0,1)`.

By *relatively prime*, we mean that the two integers have no common divisors except 1.

By *invariant*, we mean that the logical assertion always holds for every rational number created by `makeRat` and manipulated only by the operations in the `RationalCore` and `Rational` modules.

This representation has the advantage that the magnitudes of the numerator `x` and denominator `y` are kept small, thus reducing problems with overflow arising during arithmetic operations.

We thus provide a function for constructing rational numbers in this canonical form. We define constructor `makeRat` as follows.

```
makeRat :: Int -> Int -> Rat
makeRat x 0 = error ( "Cannot construct a rational number "
                      ++ showRat (x,0) ++ "\n" )
makeRat 0 _ = (0,1)
makeRat x y = (x' `div` d, y' `div` d)
    where x' = (signum' y) * x
          y' = abs' y
          d  = gcd' x' y'
```

Above we use features of Haskell we have not used in the previous examples:

- Instead of leaving the `(x,0)` case undefined, we define an explicit `error` call that returns a custom error message as a `String`.
- To concatenate two strings, we use the infix `++` (read "append") operator. (We discuss `++` in the chapter on lists.)
- Putting backticks (`` ` ``) around an alphanumeric function name enables us to use that function as an infix operator. The function `div` denotes integer division. Above the `` `div` `` operator denotes the integer division function used in an infix manner.
- The `where` clause introduces `x'`, `y'`, and `d` as a local definitions within the body of `makeRat`. It can be called from within `makeRat` but not from outside the function. In contrast, `sqrtIter` in the Square Root example is at the same level as `sqrt'`, so it can be called by other functions (in the same Haskell module at least).

  The `where` feature allows us to introduce new definitions in a top-down manner–first using a symbol and then defining it.
- Instead of defining the types of the local definitions `x'`, `y'`, and `d`, we use *type inference*.

  These parameterless functions could be declared

```
x', y', d :: Int
```

but it was not necessary because Haskell can infer the types from the types involved in their defining expressions.

Type inference can be used more broadly in Haskell, but explicit type declarations should be used for any function called from outside.

The function `signum'` (similar to the more general function `signum` in the Prelude) takes an integer and returns the integer `-1`, `0`, or `1` when the number is negative, zero, or positive, respectively.

```
signum' :: Int -> Int
signum' n | n == 0 =  0
          | n > 0  =  1
          | n < 0  = -1
```

The function `abs'` (similar to the more general function `abs` in the Prelude) takes an integer and returns its absolute value.

```
abs' :: Int -> Int
abs' n | n >= 0 =  n
       | n <  0 = -n
```

The function `gcd'` (similar to the more general function `gcd` in the Prelude) takes two integers and returns their greatest common divisor.

```
gcd' :: Int -> Int -> Int
gcd' x y = gcd'' (abs' x) (abs' y)
         where gcd'' x 0 = x
               gcd'' x y = gcd'' y (x `rem` y)
```

Prelude operation `rem` returns the remainder from dividing its first operand by its second.

Given `makeRat` defined as above, we can define `numer` and `denom` as follows:

```
numer, denom :: Rat -> Int
numer (x,_) = x
denom (_,y) = y
```

Finally, to allow rational numbers to be displayed in the normal fractional representation, we include function `showRat` in the package. We use function `show`, found in the Prelude, here to convert an integer to the usual string format and use the list operator `++` to concatenate the two strings into one.

```
showRat :: Rat -> String
showRat x = show (numer x) ++ "/" ++ show (denom x)
```

Unlike `Rat`, `makeRat`, `numer`, and `denom`, function `showRat` (as implemented) does not use knowledge of the data representation, but it is used by `makeRat`. We could optimize it slightly by allowing it to access the structure of the tuple directly.

2.6.3 Modularization

There are three groups of functions in this package:

1. the six public rational arithmetic functions `negRat`, `addRat`, `subRat`, `mulRat`, `divRat`, and `eqRat`
2. the public type `Rat`, public constructor function `makeRat`, public selector functions `numer` and `denom`, and string conversion function `showRat`
3. the private utility functions called only by the second group, but just reimplementations of Prelude functions anyway

As we have seen, `Rat`, `makeRat`, `numer`, `denom`, and `showRat` are the *interface* to the *data abstraction* that hides the information about the representation of the data. We can *encapsulate* this group of functions in a Haskell module as follows. This source code must also be in a file named `RationalCore.hs`.

```
module RationalCore
    (Rat, makeRat, numer, denom, showRat)
where
    -- Rat, makeRat, numer, denom, showRat definitions
```

We can encapsulate the utility functions in a separate module, which would enable them to be used by several other modules.

However, given that the only use of the utility functions is within the data representation module, we choose not to separate them at this time. We leave them in the data abstraction module. Of course, we could also eliminate them and use the corresponding Prelude functions directly.

Similarly, `negRat`, `addRat`, `subRat`, `mulRat`, `divRat`, and `eqRat` use the core data abstraction and, in turn, extend the interface to include rational number arithmetic operations. We can encapsulate these in another Haskell module that imports the module giving the data representation. This module must be in a file named `Rational.hs`.

```
module Rational
  ( Rat, makeRat, numer, denom, showRat, -- from RatioalCore
    negRat, addRat, subRat, mulRat, divRat, eqRat )
where
  import RationalCore
  -- negRat, addRat, subRat, mulRat, divRat, eqRat definitions
```

Other modules that use the rational number package can import module `Rational`.

This modular approach to program design and implementation offers the potential of scalability and robustness with respect to change.

The key to this *information-hiding* approach to design is to identify the aspects

of a software system that are most likely to change from one version to another and make each a design *secret* of some module.

The secret of the `RationalCore` module is the rational number data representation used. The secret of the `Rational` module itself is the methods used for rational number arithmetic.

2.6.4 Alternative rational number data representation

In the rational number data representation above, constructor `makeRat` creates pairs in which the two integers are relatively prime and the sign is on the numerator. Selector functions `numer` and `denom` just return these stored values.

An alternative representation is to reverse this approach, as shown in the following module (in file `RationalDeferGCD.hs`.)

```
module RationalDeferGCD
  (Rat, makeRat, numer, denom, showRat)
where

type Rat = (Int,Int)

makeRat :: Int -> Int -> Rat
makeRat x 0 = error ( "Cannot construct a rational number "
                      ++ showRat (x,0) ++ "\n" )
makeRat 0 y = (0,1)
makeRat x y = (x,y)

numer :: Rat -> Int
numer (x,y) = x' `div` d
    where x' = (signum' y) * x
          y' = abs' y
          d  = gcd' x' y'

denom :: Rat -> Int
denom (x,y) = y' `div` d
    where x' = (signum' y) * x
          y' = abs' y
          d  = gcd' x' y'

showRat :: Rat -> String
showRat x = show (numer x) ++ "/" ++ show (denom x)
```

This approach defers the calculation of the greatest common divisor until a selector is called.

The invariant for this rational number representation requires that, for `(x,y)`,

`y` $\neq$ `0` and zero is represented uniquely by `(0,1)`.

Furthermore, function `numer` and `denom` satisfy the equalities

```
numer (makeRat x y) == x'
denom (makeRat x y) == y'
```

where `y' > 0`, `x'` and `y'` are relatively prime, and $\frac{x}{y} = \frac{x'}{y'}$.

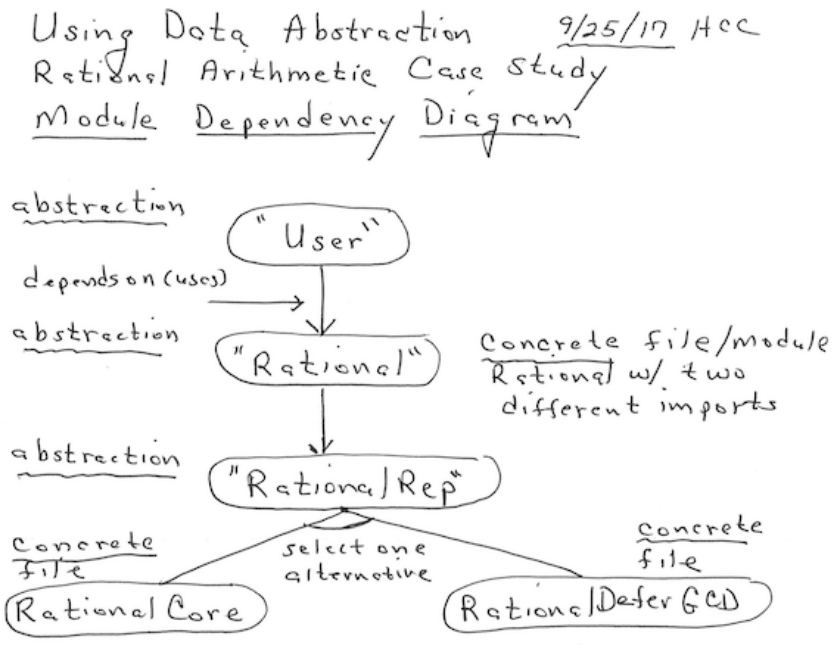
Question:

> What are the advantages and disadvantages of the two data representations?

Like module `RationalCore`, the design secret for module `RationalDeferGCD` is the rational number data representation.

Regardless of which approach is used, the definitions of the arithmetic and comparison functions do not change. Thus the `Rational` module can import data representation module `RationalCore` or `RationalDeferGCD`.

The figure below shows the dependencies among the modules we have examined in the rational arithmetic case study.



Module Dependency Diagram for Rational Arithmetic Case Study

We can consider the `RationalCore` and `RationalDeferGCD` modules as two concrete instances (Haskell `module`s) of a more abstract module we call "RationalRep" in the diagram.

The abstract module "Rational" relies on the abstract module "RationalRep" for an implemenation of rational numbers. In the Haskell code above, there are really two versions of the Haskell module `Rational` that differ only in whether they import `RationalCore` or `RationalDeferGCD`.

We could also replace alias `Rat` by a user-defined type to get another alternative definition of "RationalRep", as long as the interface functions do not have to work with types other than `Int`.

2.7 Modular Design and Programming

Now let's step back from the rational arithmetic case study and consider the general issues of data abstraction and modular design and programming.

Software engineering pioneer David Parnas defines a *module* as "a work assignment given to a programmer or group of programmers" [Parnas 1978]. This is a *software engineering* view of module.

In a programming language like Haskell, a `module` is also a program unit defined with a construct or convention. This is a *programming language* view of a module.

Ideally, a language's module features should support the software engineering module methods.

2.7.1 Information-hiding modules

According to Parnas, the goals of *modular design* are to [Parnas 1972]:

1. enable programmers to understand the system by focusing on one module at a time (i.e., *comprehensibility*).
2. shorten development time by minimizing required communication among groups (i.e., *independent development*).
3. make the software system flexible by limiting the 'number of modules affected by significant changes (i.e., *changeability*).

Parnas advocates the use of a principle called *information hiding* to guide decomposition of a system into appropriate modules (i.e. work assignments). He points out that the connections among the modules should have as few information requirements as possible [Parnas 1972].

In the Parnas approach, an information-hiding module:

- forms a *cohesive* unit of functionality *separate* from other modules

- *hides* a design decision (its *secret*) from other modules
- *encapsulates* an aspect of system likely to change (its secret)

Aspects likely to change independently of each other become secrets of separate modules. Aspects unlikely to change can become interactions (connections) among modules.

This approach supports the goal of changeability (goal 2). When care is taken to design the modules as clean abstractions with well-defined and documented interfaces, the approach also supports the goals of independent development (goal 1) and comprehensibility (goal 3).

Information hiding has been absorbed into the dogma of contemporary object-oriented programming. However, information hiding is often oversimplified as merely hiding the data and their representations [Weiss 2001].

The secret of a well-designed module may be much more than that. It may include such knowledge as a specific functional requirement stated in the requirements document, the processing algorithm used, the nature of external devices accessed, or even the presence or absence of other modules or programs in the system [Parnas 1972, 1979, 1985]. These are important aspects that may change as the system evolves.

2.7.2 Interfaces

It is important for information-hiding modules to have well-defined and stable interfaces.

According to Britton et al, an *interface* is a "set of assumptions ... each programmer needs to make about the other program ... to demonstrate the correctness of his own program" [Britton 1981].

An interface includes the type signature of each function (i.e., name, arguments, and return value) and the constraints on the environment and argument values (e.g., the invariants).

An *abstract interface* is an interface that does not change when one module implementation is substituted for another [Britton 1981; Parnas 1978]. It concentrates on module's essential aspects and obscures incidental aspects that vary among implementations.

Information-hiding modules and abstract interfaces enable us to design and build software systems with multiple versions. The information-hiding approach seeks to identify aspects of a software design that might change from one version to another and to hide them within independent modules behind well-defined abstract interfaces.

We can reuse the software design across several similar systems. We can reuse an existing module implementation when appropriate. When we need a new

implementation, we can create one by following the specification of the module's abstract interface.

2.7.3 Haskell information-hiding modules

As we have seen, in Haskell the `module` construct can be used to encapsulate an information-hiding module. The features exported form part of the interface to the module. One module can `import` another module, specifying its dependence on the interface of the other module.

We define each Haskell module in a separate file. The Haskell compiler can compile a module independently of others except that the modules it depends on must be previously compiled. The Haskell build and package management tools `cabal-install` and `stack` support Haskell modules as their primary units.

The interface of a Haskell module consists of the names and type signatures of its exported types and functions plus the constraints on the functions and the expected properties of the "objects" manipulated.

In the Rational Arithmetic case study, we defined two information-hiding modules:

1. "RationalRep", whose secret is how to represent the rational number data and whose interface consists of the data type `Rat`, operations (functions) `makeRat`, `numer`, `denom`, and `showRat`, and the constraints on these types and functions
2. "Rational", whose secret is how to implement the rational number arithmetic and whose interface consists of operations (functions) `negRat`, `addRat`, `subRat`, `mulRat`, `divRat`, and `eqRat`, the other module's interface, and the constraints on these types and functions

We developed two distinct Haskell modules, `RationalCore` and `RationalDeferGCD`, to implement the "RationalRep" information-hiding module. We developed one distinct Haskell module, `Rational`, to implement the "Rational" information-hiding module. Haskell module `Rational` can be paired (i.e., by chaning the `import` statement) either of the other two variants of "RationalRep".

Unfortunately, Haskell 2010 has a relatively weak module system that does not support multiple implementations as well as we might like. There is no way to declare that multiple Haskell modules have the same interface other than copying the common code into each module and documenting the interface carefully. We must also have multiple versions of `Rational` that differ only in which other module is imported.

Together the Glasgow Haskell Compiler (GHC) release 8.2 (July 2017) and the Cabal-Install package manager release 2.0 (August 2017) support a new extension, the Backpack mixin package system. This new system remedies the above shortcoming. In this new approach, we would define the abstract

module "RationalRep" as a signature file and require that `RationalCore` and `RationalDeferGCD` conform to it.

Further discussion of this new module system is beyond the scope of this chapter.

2.7.4 Invariants

As we saw in the Rational Arithmetic case study, a module that provides a data abstraction must ensure that the objects it creates and manipulates maintain their integrity–always have a valid structure and state. An invariant for the data abstraction can help us design and implement such objects.

Invariant: A logical assertion that must always true for every "object" created by the public constructors and manipulated only by the public operations of the data abstraction.

Often, we separate an invariant into two parts.

Interface invariant: An invariant stated in terms of the public features and abstract properties of the "object".

Implementation (representation) invariant: A detailed invariant giving the required relationships among the internal features of the implementation of an "object"

An interface invariant is a key aspect of the abstract interface of a module. It is useful to the users of the module, as well to the developers.

In the Rational Arithmetic case study, the interface invariant for the "Rational-Rep" abstract module is the following.

> For all integers `x` and nonzero integers `y`,
>
> ```
> numer (makeRat x y) == x'
> denom (makeRat x y) == y'
> ```
>
> where `y' > 0`, `x'` and `y'` are relatively prime, $\frac{x}{y} = \frac{x'}{y'}$ and if `x'` $= 0$, then `y'` $= 1$.

An implementation invariant guides the developers in the design and implementation of the internal details of a module. It relates the internal details to the interface invariant.

We can state an implementation invariant for the `RationalCore` module as follows.

> For all integers `x` and nonzero integers `y`,
>
> ```
> makeRat x y == (x',y')
> ```
>
> where `y' > 0`, `x'` and `y'` are relatively prime, $\frac{x}{y} = \frac{x'}{y'}$ and if `x'` $= 0$, then `y'` $= 1$.

The implementation invariant implies the interface invariant. Although `makeRat` does quite a bit of work, `numer` and `denom` are simple.

We can state an implementation invariant for the `RationalDeferGCD` module as follows.

> For all integers `x` and nonzero integers `y`,
>
> ```
> makeRat x y == (x,y)
> ```

In this module implementation, `makeRat` is trivial, thus `numer` and `denom` must do most of the work to establish the interface invariant.

We will return to the invariant concepts in later chapters.

2.7.5 Design criteria for interfaces

What makes a good interface for an information-hiding module?

In designing an interface for a module, we should also consider the following criteria. Of course, some of these criteria conflict with one another; a designer must carefully balance the criteria to achieve a good interface design.

Note: These are general principles; they are not limited to Haskell or functional programming. In object-oriented languages, these criteria apply to class interfaces.

- **Cohesion:** All operations must logically fit together to support a single, coherent purpose. The module should describe a single abstraction.
- **Simplicity:** Avoid needless features. The smaller the interface the easier it is to use the module.
- **No redundancy:** Avoid offering the same service in more than one way; eliminate redundant features.
- **Atomicity:** Do not combine several operations if they are needed individually. Keep independent features separate. All operations should be *primitive*, that is, not be decomposable into other operations also in the public interface.
- **Completeness:** All primitive operations that make sense for the abstraction should be supported by the module.
- **Consistency:** Provide a set of operations that are internally consistent in
  - naming convention (e.g., in use of prefixes like "set" or "get", in capitalization, in use of verbs/nouns/adjectives),
  - use of arguments and return values (e.g., order and type of arguments),
  - behavior (i.e., make operations work similarly).

Avoid surprises and misunderstandings. Consistent interfaces make it easier to understand the rest of a system if part of it is already known.

The operations should be consistent with good practices for the specific language being used.

- **Reusability:** Do not customize modules to specific clients, but make them general enough to be reusable in other contexts.
- **Robustness with respect to modifications:** Design the interface of an module so that it remains stable even if the implementation of the module changes. (That is, it should be an abstract interface for an information-hiding module as we discussed above.)
- **Convenience:** Where appropriate, provide additional operations (e.g., beyond the complete primitive set) for the convenience of users of the module. Add convenience operations only for frequently used combinations after careful study.

We must trade off conflicts among the criteria. For example, we must balance:

- completeness versus simplicity
- reusability versus simplicity
- convenience versus consistency, simplicity, no redundancy, and atomicity

We must also balance these design criteria against efficiency and functionality.

2.8 Exercises

TODO: Add more exercises for the techniques and features introduced in this section. Make sure what is here still make sense.

For each of the following exercises, develop and test a Haskell function or set of functions.

1. Develop a Haskell function `sumsqbig` that takes three numbers as arguments and returns the sum of the squares of the two larger numbers. That is `sumsqbig 2.0 1.0 3.0` yields `13`.
2. Develop a Haskell function `xor` that takes two Booleans and returns the "exclusive-or" of the two values. An exclusive-or operation returns `True` when exactly one of its arguments is `True` and returns `False` otherwise.
3. Develop a Haskell Boolean function `div23n5` such that `div23n5 n` returns `True` if and only if `n` is divisible by 2 or divisible by 3 but not divisible by 5. That is, `div23n5 6` yields `True` and `div23n5 30` yields `False`.
4. Develop a Haskell function `notDiv` such that `notDiv n d` returns `True` if and only if integer `n` is not divisible by `d`. That is, `notDiv 10 5` yields `False` and `notDiv 11 5` yields `True`.

5. Develop a Haskell function `mult` that takes two *natural numbers* (i.e., nonnegative integers) and returns their product. The function must not use the multiplication (`*`) or division (`div`) operators. Hint: Multiplication can be done by repeated addition.

6. Develop a Haskell function `addTax` that takes two `Double` values such that `addTax c p` returns `c` with a sales tax of `p` *percent* added. For example, `addTax 2.0 9.0` returns `2.18`.

   Also develop a function `subTax` that is the inverse of `addTax`. That is, `subTax (addTax c p) p` yields `c`.

7. The time of day can be represented by a tuple `(hours,minutes,m)` where `m` indicates either "AM" or "PM". Develop a Boolean Haskell function `comesBefore` that takes two time-of-day tuples and determines whether the first is an earlier time than the second.

8. Develop a Haskell function

   ```
   minf :: (Int -> Int) -> Int
   ```

   such that `minf g` returns the smallest integer `m` such that `0 <= m <= 10000000` and `g m == 0` (if such an integer exists).

9. Develop a Haskell module (or modules) for line segments on the two-dimensional coordinate plane using the *rectangular coordinate* system.

   We can represent a line segment with two points–the starting point and the ending point. Develop the following Haskell functions:

   - constructor `newSeg` that takes two points and returns a new line segment
   - selectors `startPt` and `endPt` that each take a segment and return its starting and ending points, respectively

   We normally represent the plane with a *rectangular coordinate* system. That is, we use two axes–an `x` *axis* and a `y` *axis*–intersecting at a right angle. We call the intersection point the *origin* and label it with 0 on both axes. We normally draw the `x` axis horizontally and label it with increasing numbers to the right and decreasing numbers to the left. We also draw the `y` axis vertically with increasing numbers upward and decreasing numbers downward. Any point in the plane is uniquely identified by its `x`-coordinate and `y`-coordinate.

   Define a data representation for points in the rectangular coordinate system and develop the following Haskell functions:

   - constructor `newPtFromRect` that takes the `x` and `y` coordinates of a point and returns a new point
   - selectors `getx` and `gety` that takes a point and returns the `x` and `y` coordinates, respectively

- display function `showPt` that takes a point and returns an appropriate `String` representation for the point

Now, using the various constructors and selectors, also develop the Haskell functions for line segments:

- `midPt` that takes a line segment and returns the point at the middle of the segment
- display function `showSeg` that takes a line segment and returns an appropriate `String` representation

Note that `newSeg`, `startPt`, `endPt`, `midPt`, and `showSeg` can be implemented independently from how the points are represented.

10. Develop a Haskell module (or modules) for line segments that represents points using the *polar coordinate* system instead of the rectangular coordinate system used in the previous exercise.

    A polar coordinate system represents a point in the plane by its *radial coordinate* `r` (i.e., the distance from the *pole*) and its *angular coordinate* `t` (i.e., the angle from the *polar axis* in the reference direction). We sometimes call `r` the *magnitude* and `t` the *angle*.

    By convention, we align the rectangular and polar coordinate systems by making the origin the pole, the positive portion of the `x` axis the polar axis, and let the first quadrant (where both `x` and `y` are positive) be the smallest positive angles in the reference direction. That is, with a traditional drawing of the coordinate systems, we measure and the radial coordinate `r` as the distance from the origin measure the angular coordinate `t` counterclockwise from the positive `x` axis.

    Using knowledge of trigonometry, we can convert among rectangular coordinates `(x,y)` and polar coordinates `(r,t)` using the equations:

    ```
    x = r * cos(t)
    y = r * sin(t)
    r = sqrt(x^2 + y^2)
    t = arctan2(y,x)
    ```

    Define a data representation for points in the polar coordinate system and develop the following Haskell functions:

    - constructor `newPtFromPolar` that takes the magnitude `r` and angle `t` as the polar coordinates of a point and returns a new point
    - selectors `getMag` and `getAng` that each take a point and return the magnitude `r` and angle `t` coordinates, respectively
    - selectors `getx` and `gety` that return the `x` and `y` components of the points (represented here in polar coordinates)

- display functions `showPtAsRect` and `showPtAsPolar` to convert the points to strings using rectangular and polar coordinates, respectively,

Functions `newSeg`, `startPt`, `endPt`, `midPt`, and `showSeg` should work as in the previous exercise.

11. Modify the solutions to the previous two line-segment module exercises to enable the line segment functions to be in one module that works properly if composed with either of the two data representation modules. (The solutions may have already done this.)

12. Modify the solution to the previous line-segment exercise to use the Backpack module system.

13. Modify the modules in the previous exercise to enable the line segment module to work with both data representations in the same program.

14. Modify the solution to the Rational Arithmetic case study to use the Backpack module system.

2.9 References

[**Abelson-Sussman 1996**] Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs* (SICP), Second Edition, MIT Press, 1996.

[**Bird-Wadler 1998**] Richard Bird and Philip Wadler. *Introduction to Functional Programming*, Second Edition, Addison Wesley, 1998. [First Edition, 1988]

[**Britton 1981**] K. H. Britton, R. A. Parker, and D. L. Parnas. "A Procedure for Designing Abstract Interfaces for Device Interface Modules," In *Proceedings of the 5th International Conference on Software Engineering*, pp. 195-204, March 1981.

[**Chiusano-Bjarnason 2015**]] Paul Chiusano and Runar Bjarnason, *Functional Programming in Scala*, Manning, 2015.

[**Cunningham 2014**] H. Conrad Cunningham. *Notes on Functional Programming with Haskell*, 1994-2014.

[**Dale-Walker 1996**] Nell Dale and Henry M. Walker. *Abstract Data Types: Specifications, Implementations, and Applications*, D. C. Heath, 1996. (Especially chapter 1 on "Abstract Specification Techniques")

[**Horstmann 1995**] Cay S. Horstmann. *Mastering Object-Oriented Design in C++*, Wiley, 1995. (Especially chapters 3-6 on "Implementing Classes", "Interfaces", "Object-Oriented Design", and "Invariants")

[**Meyer 1997**] Bertrand Meyer. *Object-Oriented Program Construction*, Second Edition, Prentice Hall, 1997. (Especially chapter 6 on "Abstract Data Types" and chapter 11 on "Design by Contract")

[**Mossenbock 1995**] Hanspeter Mossenbock. *Object-Oriented Programming in*

Oberon-2, Springer-Verlag, 1995. (Especially chapter 3 on "Data Abstraction")

[**Parnas 1972**] D. L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules," *Communications of the ACM*, Vol. 15, No. 12, pp.1053-1058, 1972.

[**Parnas 1976**] D. L. Parnas. "On the Design and Development of Program Families," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 1, pp. 1-9, March 1976.

[**Parnas 1978**] D. L. Parnas. "Some Software Engineering Principles," *Infotech State of the Art Report on Structured Analysis and Design*, Infotech International, 10 pages, 1978. Reprinted in *Software Fundamentals: Collected Papers by David L. Parnas*, D. M. Hoffman and D. M. Weiss, editors, Addison-Wesley, 2001.

[**Parnas 1979**] D. L. Parnas. "Designing Software for Ease of Extension and Contraction," *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 1, pp. 128-138, March 1979.

[**Parnas 1985**] D. L. Parnas, P. C. Clements, and D. M. Weiss. "The Modular Structure of Complex Systems," *IEEE Transactions on Software Engineering*, Vol. SE-11, No. 3, pp. 259-266, March 1985.

[**Thompson 2011**] Simon Thompon. *Haskell: The Craft of Programming, Third Edition*, Pearson, 2011.

[**Weiss 2001**] D. M. Weiss. "Introduction: On the Criteria to Be Used in Decomposing Systems into Modules," In *Software Fundamentals: Collected Papers by David L. Parnas*, D. M. Hoffman and D. M. Weiss, editors, Addison-Wesley, 2001.

2.10 Terms and Concepts

Factorials, function definition and application, recursion, function domains, `error`, `if`, guards, basic types (`Int`, `Integer`, `Bool`, `Char`, functions, tuples, lists, `String`), `type` aliases, library (Prelude) functions, REPL, `ghci` commands and use, procedural abstraction, top-down stepwise refinement, abstract code, termination condition for recursion, Newton's method, Haskell `module`, module exports and imports, rational number arithmetic, data abstraction, properties of data, data representation, invariant, interface invariant, implementation or representation invariant, canonical or normal forms, information hiding, module secret, encapsulation, interface, abstract interface, design criteria for interfaces, software reuse, Haskell `where` local definition, type inference, use of Haskell modules to implement information-hiding modules rational number arithmetic, data abstraction, abstract properties of data, data representation, invariant, interface invariant, implementation or representation invariant, canonical or normal forms, information hiding, module secret, encapsulation, interface, abstract interface, design criteria for interfaces, software reuse, Haskell `where` local definition, type inference, use of Haskell modules to implement information-hiding modules