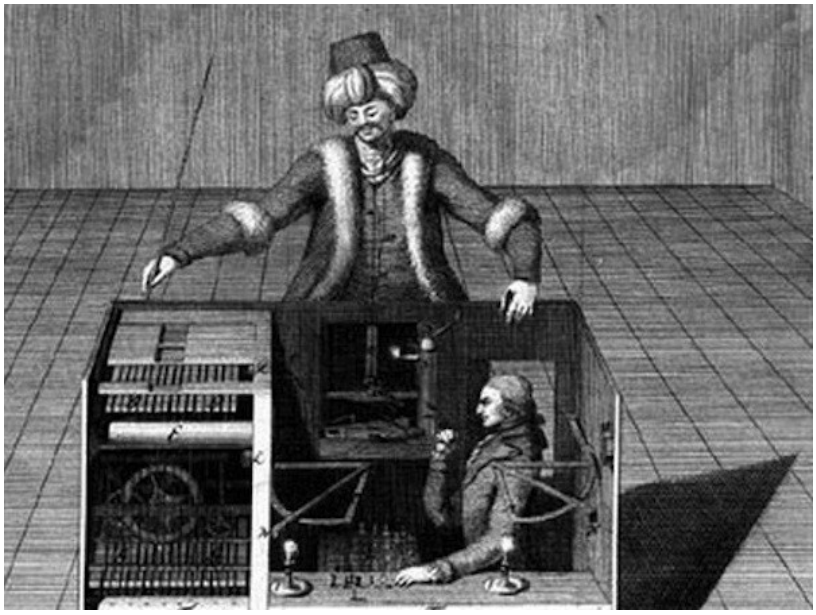
Purely Functional Algorithm Specification

ESSLLI 2012 Lecture Notes

– Version April 26, 2012 –

Jan van Eijck



Preface

This course develops a perspective on algorithm specification in terms of purely functional programming.

An algorithm is an effective method expressed as a list of instructions describing a computation for calculating a result. Algorithms have to be written in human readable form, either using pseudocode (natural language looking like executable code), a high level specification language like Dijkstra's Guarded Command Language, or an executable formal specification formalism such as Z.

The course will adopt a purely functional view on the key ingredients of imperative programming: while loops, repeat loops, and for loops, and demonstrate how this can be used for specifying (executable) algorithms, and for automated testing of Hoare correctness statements about these algorithms.

Inspiration for this was the talk by Leslie Lamport at CWI, Amsterdam, on the executable algorithm specification language *PlusCal* [26], plus Edsger W. Dijkstra, "EWD472: Guarded commands, nondeterminacy and formal derivation of programs" [13]. Instead of formal program derivation, we demonstrate test automation of Hoare style assertions.

This course is based on course notes for a Master Course in Software Specification and Testing that I have developed over the past five years. Further material will be made available on the website on *Purely Functional Algorithm Specification*, at address `http://www.homepages.cwi/~jve/pfas/`.

The course has made several trial runs in Amsterdam, in a Master Course on software specification and testing at the University of Amsterdam. See `http://homepages.cwi.nl/~jve/courses/testing2011/`.

Prerequisites This is course has no formal prerequisites at all. The introduction will be at a serious speed, while remaining accessible to students with a wide variety of backgrounds. Anyone with a willingness to learn formal methods should be able to follow. Students with some previous experience with (functional) programming will be at an advantage, though.

Acknowledgements Thanks to Pieter Brandwijk, Pedro Correia, Paul Klint, Bert Lisser, Kyndylan Nienhuis, Atze van der Ploeg, Floor Sietsma, Tijs van der Storm, Christina Unger, Jurgen Vinju, . . . , and the students of my courses on Software Specification and Testing at the University of Amsterdam.

Contents

Chapter 1

Why Learn Haskell?

Abstract

This is an introductory chapter, to be skipped or skimmed over by those who are already acquainted with Haskell. From Chapter 2 on, I will assume that you know Haskell, or are willing to figure out tricks of the language for yourself.

Key words: Functional programming, functional algorithm design.

1.1 A Short History of Haskell

In the 1980s, efforts of researchers working on functional programming were scattered across many languages (Lisp, Scheme, SASL, KRC, Hope, Id, Miranda, ML,...).

In 1987 a dozen functional programmers decided to meet in order to reduce unnecessary diversity in functional programming languages by designing a common language [22]. The new language should be:

- based on ideas that enjoyed wide consensus;
- suitable for further language research as well as applications, including building large systems;
- freely available (in particular: anyone should be permitted to implement the language and distribute it to whomever they please).

The new language was called *Haskell*, after the logician and mathematician Haskell Brooks Curry (1900–1982). Curry is known for his work on the lambda calculus and on combinatory logic. The lambda calculus is the foundation of Haskell.

Haskell B. Curry

In 1990, the first *Haskell* specification was published [].

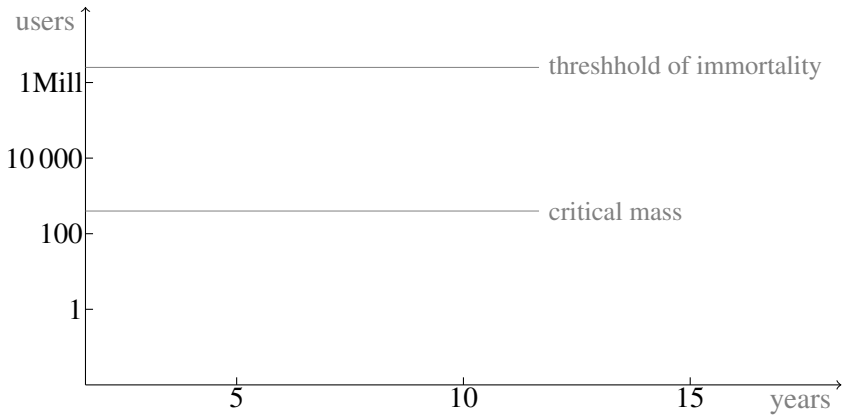
Right now, Haskell has a flourishing (and very friendly) user community, and many enthousiastic supporters. If asked why learning Haskell is a good idea, they have things like this to say:

> Haskell is a wide-spectrum language, suitable for a variety of applications. It is particularly suitable for programs which need to be highly modifiable and maintainable.
>
> Much of a software product's life is spent in specification, design and maintenance, and not in programming. Functional languages are superb for writing specifications which can actually be executed (and hence tested and debugged). Such a specification then is the first prototype of the final program.
>
> Functional programs are also relatively easy to maintain, because the code is shorter, clearer, and the rigorous control of side effects eliminates a huge class of unforeseen interactions.
>
> From: `http://www.haskell.org/haskellwiki/Introduction`

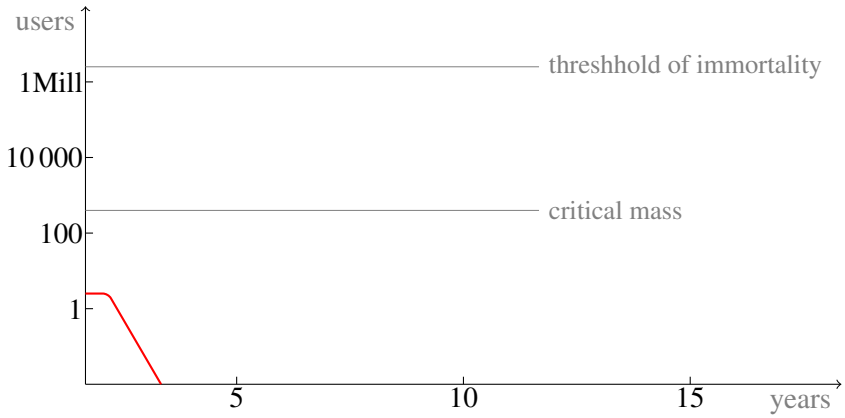
Simon Peyton Jones, one of the moving forces behind Haskell, once expressed the following amusing view on the life cycle of programming languages [].

Simon Peyton Jones

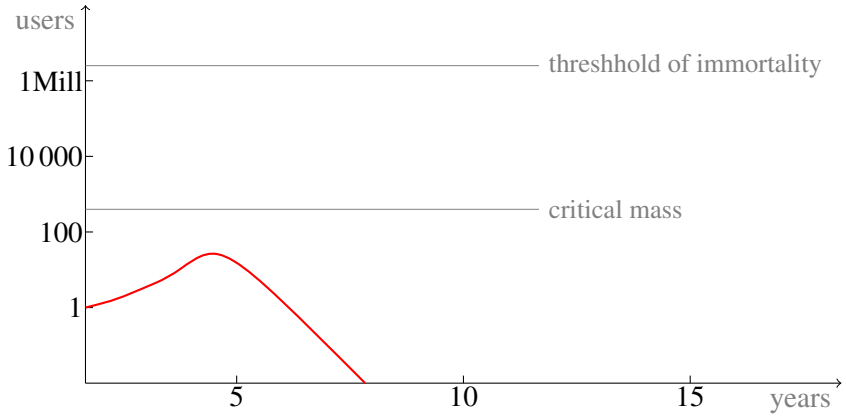
The Life Cycle of Programming Languages



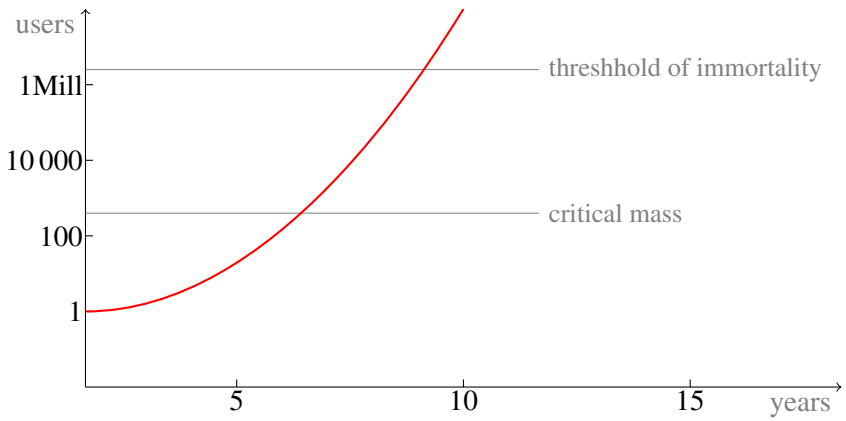
Most Research Languages (the quick death)



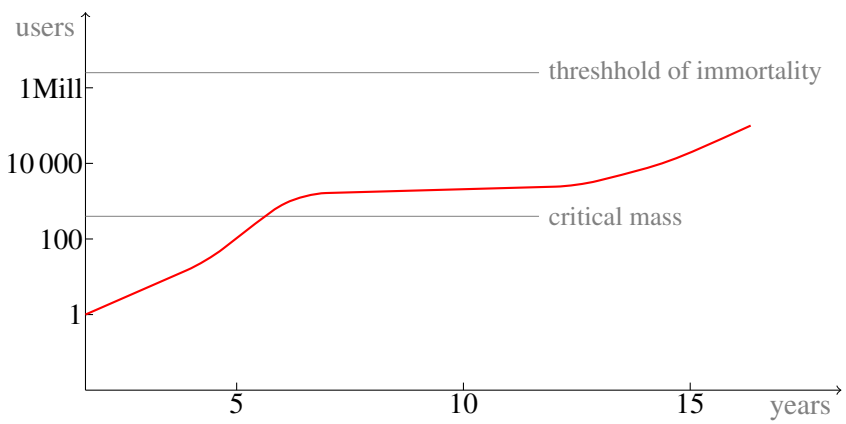
Successful Research Language (the slow death)



C++, Java, Perl (the absence of death)



The Life Cycle of Haskell



So Haskell may yet become immortal, but it also “ . . . may just be a passing fancy, that in time will go.”

1.2 Literate Programming

We will use literate Haskell in what follows. The Haskell code that we mention in this chapter is collected into a so-called Haskell module. See [25] for the benefits of literate programming.

```
module WLH

where

import List
```

You can find the module `WLH.hs` on the course website, at address `http://www.homepages.cwi/~jve/pfas/`.

1.3 How Haskell is Different

Here is a quote from an interview with John Goerzen, one of the authors of *Real World Haskell* [28].

> Question: One of Haskell's benefits is that you can use it as a purely functional language – but that's really different for people who've come up in the imperative or object-oriented worlds. What does it take to learn how to think in a pure fashion?
>
> Goerzen: That's probably the single biggest mind-shift that you deal with coming from a different language; that and laziness. Both of those are both pretty big.
>
> As for how to learn about it, it's a lot of relearning how to do some very basic things and then building upon that. For instance, in imperative languages, you have for loops and you have while loops. There's a lot of having a variable there and then incrementing it as you iterate over something. In Haskell, you tend to take a recursive approach rather than that. It can be a little bit scary at first because you might be thinking if you're using a language such as C, incrementing a variable is a pretty cheap operation.
>
> `http://broadcast.oreilly.com/2009/01/why-you-should-learn-haskell.html`

Haskell allows for abstract, high order programming. (Ideally, more thinking and less writing and debugging.)

Haskell is based on the lambda calculus, therefore the step from formal specification to implementation is very small.

Haskell offers you a new perspective on programming, it is powerful, and it is fun.

The type system behind Haskell is a great tool for writing specifications that catch many coding errors.

Your Haskell understanding will influence the way you look at programming: you will start to appreciate abstraction.

Haskell comes with great tools for automated test generation: a tool we will employ at some point is *QuickCheck* [10], which has served as inspiration for the development of similar tools for many other programming languages.

Haskell is functional A Haskell program consists entirely of functions. Running a Haskell program consists in evaluating expressions (basically functions applied to arguments).

The main program itself is a function with the program's input as argument and the program's output as result.

Typically the main function is defined in terms of other functions, which in turn are defined in terms of still more functions, until at the bottom level the functions are language primitives.

This means that Haskell, like all functional languages, is extensible. If you need a certain programming construct that the language does not provide as a primitive, it is up to you to define it. We will use this feature a lot in this course.

Functions are first-class citizens, which means that they can be used as arguments to other (higher order) functions. This is extremely powerful and extremely useful.

A shift in thinking If you are an imperative thinker, you think mainly of:

- variables as pointers to storage locations whose value can be updated all the time
- sequences of commands telling the computer what to do (how to change certain memory locations) step by step.

Here are some examples:

- initialize a variable `examplelist` of type integer list,
  then add 1, then add 2, then add 3.
- in order to compute the factorial of n, initialize an integer variable `f` as 1, then for all `i` from 1 to n, set `f` to `f`$\times$`i`

If you are a functional thinker, you view bound variables as place-holders for function arguments, and you view unbound variables as identifiers for immutable persistent values, or as names for functions.

Instead of telling the computer what actions to perform in what order, you prefer telling the computer what things are.

Running through the same examples again:

- `examplelist` is a list of integers containing the elements
  1, 2, and 3
- the factorial of n is the product of all integers from 1 to n.

Here is the Haskell code for the factorial function:

```
factorial :: Integer -> Integer
factorial n = product [1..n]
```

All the same, this course will stress that functional programming is an *inclusive* paradigm, well capable of expressing the *while* loops, *repeat* loops and *for* loops that play such an important role in pseudo-code presentations of algorithms.

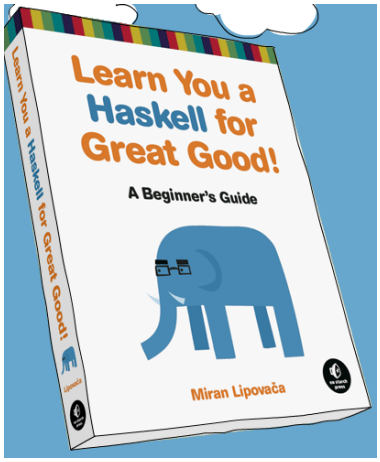
1.4 Where Should I Begin?

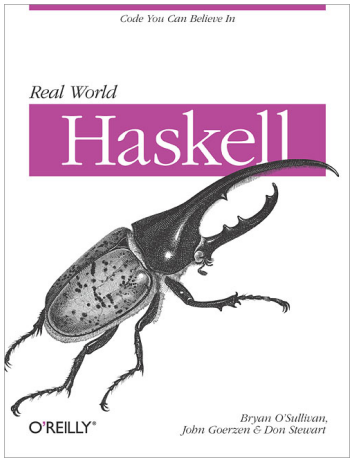
Resources For everything Haskell-related, start at `http://haskell.orghaskell.org`.

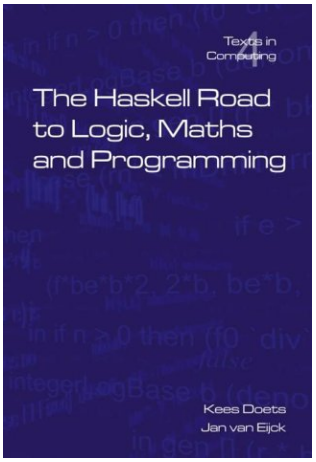
There are lots of free tutorials you may wish to consult:

- Chapter 1 of "The Haskell Road" [14], freely available from `http://homepages.cwi.nl/~jve/HR/`
- *Real World Haskell* [28], `http://book.realworldhaskell.org/read/`
- *Learn you a Haskell for great good* `http://learnyouahaskell.comlearnyouahaskell.com`
- *A gentle introduction to Haskell* `http://haskell.org/tutorial`.
- The Haskell Wikibook, `secure.wikimedia.org/wikibooks/en/wiki/Haskell`
- Hal Daume's *Yet Another Haskell Tutorial* [11] is also available as a Wikibook, from `secure.wikimedia.org/wikibooks/en/wiki/Haskell/YAHT`.

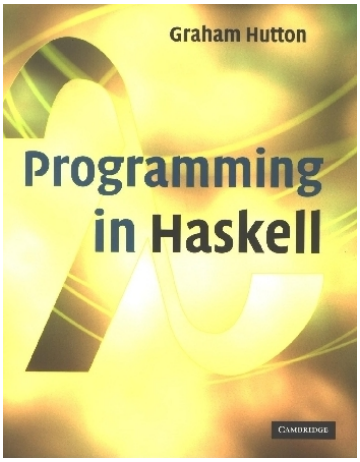
Some recommended books:

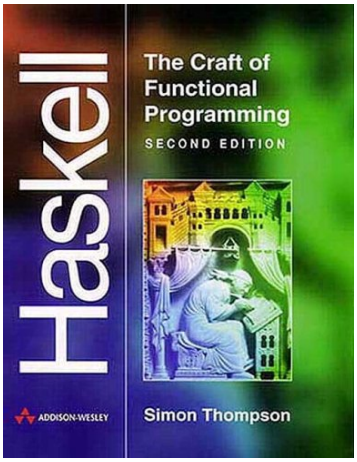


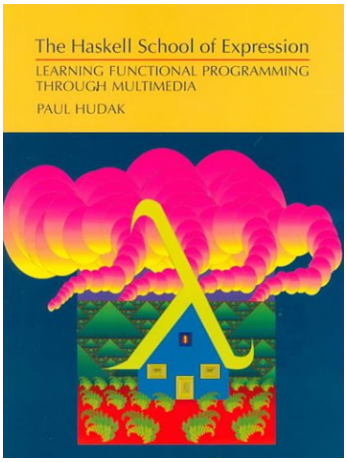




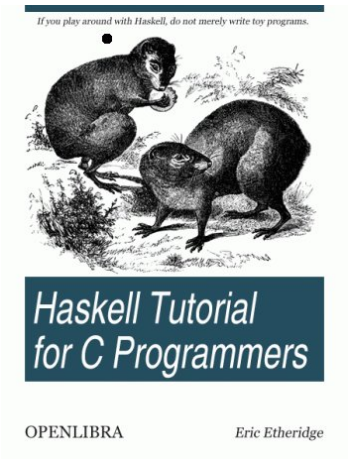
And some more:

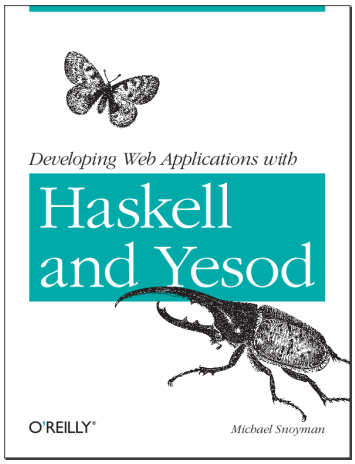


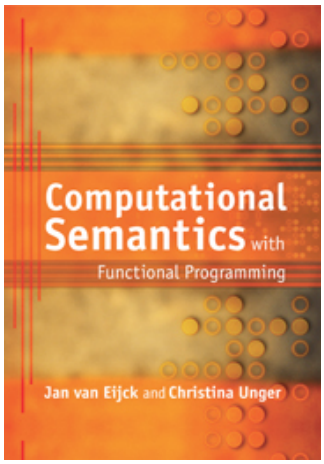




If you are coming to Haskell from elsewhere, or are into specific applications, one of the following might be for you:







1.5 Really Getting Started

Get the Haskell Platform:

- `http://hackage.haskell.org/platform/`

This includes the Glasgow Haskell Compiler (GHC) together with standard libraries and the interactive environment GHCi. Follow the instructions to install the platform.

Haskell as a Calculator Start the interpreter:

```
jve@vuur:~/courses/12/esslli12$ ghci
GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
Prelude>
```

The prompt `Prelude>` you are seeing indicates that the so-called Haskell Prelude, consisting of a list of useful predefined functions, is loaded.

GHCi can be used to interactively evaluate expressions.

```
Prelude> 2+3

Prelude> 2+3*4

Prelude> 2^10

Prelude> (42-10)/2

Prelude> (+) 2 3
```

Your first Haskell program

1. Write the following code to a text file and save it as `first.hs`:

   ```
   double :: Int -> Int
   double n = 2 * n
   ```

2. Inside GHCi, you can load the program with `:l first.hs`
   (or by running `ghci first.hs`).
   With `:r` you can reload it if you change something.

3. Now you can evaluate expressions like `double 5`,
   `double (2+3)`, and `double (double 5)`.

4. With `:t` you can ask GHCi about the type of an expression.

5. Leave the interactive environment with `:q`.

Some simple samples of lazy lists "Sentences can go on and on and on (and on)*"

Here is a so-called lazy list implementation:

```
sentence = "Sentences can go " ++ onAndOn

onAndOn = "on and " ++ onAndOn
```

This uses the operation ++ for list concatenation plus the double quote notation for character strings.

If you grab the module `WLH.hs` from `http://www.homepages.cwi/~jve/pfas/` and load it, you will see the following:

```
jve@vuur:~/courses/12/esslli12$ ghci WLH.hs
GHCi, version 7.0.3: http://www.haskell.org/ghc/  :? for help
Loading package ghc-prim ... linking ... done.
Loading package integer-gmp ... linking ... done.
Loading package base ... linking ... done.
[1 of 1] Compiling WLH              ( WLH.hs, interpreted )
Ok, modules loaded: WLH.
*WLH>
```

If you now type `sentence` and hit the return (enter) key, you will see that sentences can indeed go on and on and . . . . You can quit the infinite loop by hitting `^C` (control-C). Next, check an initial segment:

```
*WLH> take 65 sentence
"Sentences can go on and on and on and on and on and on and on and on and"
*WLH>
```

Next, consider the following:

```
sentences = "Sentences can go on" :
              map (++ " and on") sentences
```

New ingredients here are `:` for putting a item in front of a list, and the function `map` that will be explained below.

The function `sentences` generates an infinite list of sentences. Here is the start of the list:

```
*WLH> take 10 sentences
["Sentences can go on","Sentences can go on and on","Sentences can go
 on and on and on","Sentences can go on and on and on and on","Sentences
 can go on and on and on and on and on","Sentences can go on and on and
 on and on and on and on","Sentences can go on and on and on and on and
 on and on and on","Sentences can go on and on and on and on and on and
 on and on and on","Sentences can go on and on and on and on and on and
 on and on and on and on","Sentences can go on and on and on and on and
 on and on and on and on and on and on and on"]
*WLH>
```

Lambda Abstraction in Haskell In Haskell, `\ x` expresses lambda abstraction over variable `x`.

```
sqr :: Int -> Int
sqr = \ x -> x * x
```

The standard mathematical notation for this is $\lambda x \mapsto x * x$. Haskell notation aims at remaining close to mathematical notation.

- The intention is that variabele `x` stands proxy for a number of type `Int`.
- The result, the squared number, also has type `Int`.
- The function `sqr` is a function that, when combined with an argument of type `Int`, yields a value of type `Int`.
- This is precisely what the type-indication `Int -> Int` expresses.

String Functions in Haskell

```
Prelude> (\ x -> x ++ " emeritus") "professor"
"professor emeritus"
```

This combines **lambda abstraction** and **concatenation**.

The types:

```
Prelude> :t (\ x -> x ++ " emeritus")
\x -> x ++ " emeritus" :: [Char] -> [Char]
Prelude> :t "professor"
"professor" :: String
Prelude> :t (\ x -> x ++ " emeritus") "professor"
(\x -> x ++ " emeritus") "professor" :: [Char]
```

Concatenation The type of the concatenation function:

```
Prelude> :t (++)
(++) :: forall a. [a] -> [a] -> [a]
```

The type indicates that `(++)` not only concatenates strings. It works for lists in general.

More String Functions in Haskell

```
Prelude> (\ x -> "nice " ++ x) "guy"
"nice guy"
Prelude> (\ f -> \ x -> "very " ++ (f x))
              (\ x -> "nice " ++ x) "guy"
"very nice guy"
```

The types:

```
Prelude> :t "guy"
"guy" :: [Char]
Prelude> :t (\ x -> "nice " ++ x)
(\ x -> "nice " ++ x) :: [Char] -> [Char]
Prelude> :t (\ f -> \ x -> "very " ++ (f x))
(\ f -> \ x -> "very " ++ (f x))
  :: forall t. (t -> [Char]) -> t -> [Char]
```

Characters and Strings

- The Haskell type of characters is `Char`. Strings of characters have type `[Char]`.
- Similarly, lists of integers have type `[Int]`.
- The empty string (or the empty list) is `[]`.
- The type `[Char]` is abbreviated as `String`.
- Examples of characters are `'a'`, `'b'` (note the single quotes).
- Examples of strings are `"Turing"` and `"Chomsky"` (note the double quotes).
- In fact, `"Chomsky"` can be seen as an abbreviation of the following character list:

  `['C','h','o','m','s','k','y'].`

Properties of Strings

- If strings have type `[Char]` (or `String`), properties of strings have type `[Char] -> Bool`.
- Here is a simple property:

```
aword :: [Char] -> Bool
aword [] = False
aword (x:xs) = (x == 'a') || (aword xs)
```

- This definition uses *pattern* matching: `(x:xs)` is the prototypical non-empty list.
- The `head` of `(x:xs)` is `x`, the `tail` is `xs`.
- The head and tail are glued together by means of the operation `:`, of type `a -> [a] -> [a]`.
- The operation combines an object of type `a` with a list of objects of the same type to a new list of objects, again of the same type.

List Patterns

- It is common Haskell practice to refer to non-empty lists as `x:xs`, `y:ys`, and so on, as a useful reminder of the facts that `x` is an element of a list of `x`'s and that `xs` is a list.
- Note that the function `aword` is called again from the body of its own definition. We will encounter such **recursive** function definitions again and again.
- What the definition of `aword` says is that the empty string is not an `aword`, and a non-empty string is an `aword` if either the head of the string is the character `a`, or the tail of the sring is an `aword`.
- The list pattern `[]` matches only the empty list,
- the list pattern `[x]` matches any singleton list,
- the list pattern `(x:xs)` matches any non-empty list.

List Reversal The reversal of the string "CHOMSKY" is the string "YKSMOHC". The reversal of the string "GNIRUT" is the string "TURING".

```
reversal :: [a] -> [a]
reversal []     = []
reversal (x:t) = reversal t ++ [x]
```

Reversal works for any list, not just for strings. This is indicated by the type specification `[a] -> [a]`.

Haskell Basic Types

- `Int` and `Integer`, to represent integers. Elements of `Integer` are unbounded (can be of any size).
- `Float` and `Double` represent floating point numbers. The elements of `Double` have higher precision.
- `Bool` is the type of Booleans.

- `Char` is the type of characters.

Note that the name of a type always starts with a capital letter.

To denote arbitrary types, Haskell allows the use of *type variables*. For these, `a`, `b`, ..., are used.

Haskell Derived Types Derived types can be constructed in the following way:

- By list-formation: if `a` is a type, `[a]` is the type of lists over `a`. Examples: `[Int]` is the type of lists of integers; `[Char]` is the type of lists of characters, or strings.
- By pair- or tuple-formation: if `a` and `b` are types, then `(a,b)` is the type of pairs with an object of type `a` as their first component, and an object of type `b` as their second component. If `a`, `b` and `c` are types, then `(a,b,c)` is the type of triples with an object of type `a` as their first component, an object of type `b` as their second component, and an object of type `c` as their third component ...
- By function definition: `a -> b` is the type of a function that takes arguments of type `a` and returns values of type `b`.
- By defining your own datatype from scratch, with a `data` type declaration. More about this in due course.

Mapping If you use the Hugs command `:t` to find the types of the predefined function `map`, you get the following:

```
Prelude> :t map
map :: forall a b. (a -> b) -> [a] -> [b]
```

The function `map` takes a function and a list and returns a list containing the results of applying the function to the individual list members.

This is an example of higher order functional programming, of a function taking another function as an argument.

If `f` is a function of type `a -> b` and `xs` is a list of type `[a]`, then `map f xs` will return a list of type `[b]`. E.g., `map (^2) [1..9]` will produce the list of squares

```
[1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Sections But let us first explain the notation `(^2)`.

- In general, if `op` is an infix operator, `(op x)` is the operation resulting from applying `op` to its righthand side argument.
- `(x op)` is the operation resulting from applying `op` to its lefthand side argument.
- `(op)` is the prefix version of the operator.

- Thus `(2^)` is the operation that computes powers of 2, and `map (2^) [1..10]` will yield

  `[2, 4, 8, 16, 32, 64, 128, 256, 512, 1024]`

- Similarly, `(>3)` denotes the property of being greater than 3, and `(3>)` the property of being smaller than 3.

- `(++ " and on")` denotes the operation of appending `" and on"` to a string.

Map again If p is a property (an operation of type `a -> Bool`) and `l` is a list of type `[a]`, then `map p l` will produce a list of type `Bool` (a list of truth values), like this:

```
Prelude> map (>3) [1..6]
[False, False, False, True, True, True]
Prelude>
```

Here is a definition of `map`, including a type declaration.

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = (f x) : map f xs
```

The code is in `light grey` . This indicates that this definition will not be added to the chapter module. Adding it to the chapter module would result in a compiler error, for `map` is already defined.

Filter A function for filtering out the elements from a list that satisfy a given property:

```
Prelude> filter (>3) [1..10]
[4,5,6,7,8,9,10]
```

The type declaration and the function definition:

```
filter :: (a -> Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs) | p x       = x : filter p xs
                | otherwise =     filter p xs
```

List comprehension List comprehension is defining lists by the following method:

```
[ x | x <- xs, property x ]
```

This defines the sublist of `xs` of all items satisfying `property`. It is equivalent to:

```
filter property xs
```

Here are some examples:

```
someEvens    = [ x | x <- [1..1000], even x ]

evensUntil n = [ x | x <- [1..n], even x ]

allEvens     = [ x | x <- [1..], even x ]
```

Equivalently:

```
someEvens    = filter even [1..1000]

evensUntil n = filter even [1..n]

allEvens     = filter even [1..]
```

Nub The function `nub` removes duplicates, as follows:

```
nub :: Eq a => [a] -> [a]
nub [] = []
nub (x:xs) = x : nub (filter (/= x) xs)
```

Note the indication `Eq a =>` in the type declaration. This is to indicate that the type `a` has to satisfy some special properties, to ensurethat `(/=)`, the operation for non-equality, is defined on it.

Function Composition The composition of two functions f and g, pronounced 'f after g' is the function that results from first applying g and next f.

Standard notation for this: $f \cdot g$. This is pronounced as "f after g".

Haskell implementation:

```
(.) :: (a -> b) -> (c -> a) -> (c -> b)
f . g = \ x -> f (g x)
```

Note the types!

Elem, all, and

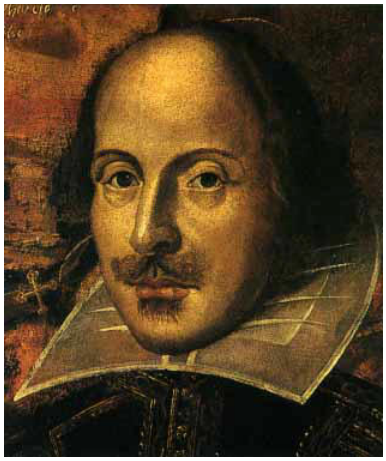
```
elem :: Eq a => a -> [a] -> Bool
elem x []     = False
elem x (y:ys) = x == y || elem x ys
```

```
all :: Eq a => (a -> Bool) -> [a] -> Bool
all p = and . map p
```

Note the use of `.` for function composition.

```
and :: [Bool] -> Bool
and [] = True
and (x:xs) = x && and xs
```

Shakespeare's Sonnet 73



```
sonnet73 =
 "That time of year thou mayst in me behold\n"
 ++ "When yellow leaves, or none, or few, do hang\n"
 ++ "Upon those boughs which shake against the cold,\n"
 ++ "Bare ruin'd choirs, where late the sweet birds sang.\n"
 ++ "In me thou seest the twilight of such day\n"
 ++ "As after sunset fadeth in the west,\n"
 ++ "Which by and by black night doth take away,\n"
 ++ "Death's second self, that seals up all in rest.\n"
 ++ "In me thou see'st the glowing of such fire\n"
 ++ "That on the ashes of his youth doth lie,\n"
 ++ "As the death-bed whereon it must expire\n"
 ++ "Consumed with that which it was nourish'd by.\n"
 ++ "This thou perceivest, which makes thy love more strong,\n"
 ++ "To love that well which thou must leave ere long."
```

Counting

```
count :: Eq a => a -> [a] -> Int
count x []                 = 0
count x (y:ys) | x == y    = succ (count x ys)
               | otherwise = count x ys
```

```
average :: [Int] -> Rational
average [] = error "empty list"
average xs = toRational (sum xs) / toRational (length xs)
```

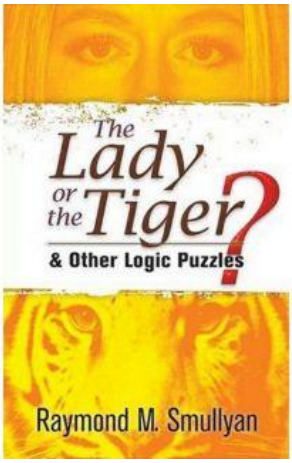
Some commands to try out

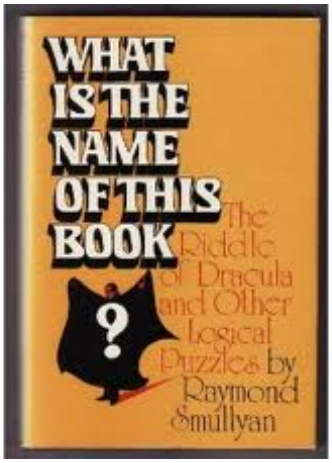
- `putStrLn sonnet73`
- `map toLower sonnet73`
- `map toUpper sonnet73`
- `filter (`elem` "aeiou") sonnet73`
- `count 't' sonnet73`
- `count 't' (map toLower sonnet73)`
- `count "thou" (words sonnet73)`
- `count "thou" (words (map toLower sonnet73))`

Next, attempt the programming exercises from Chapter 1 and 2 of "The Haskell Road" [14].

1.6 Solving Logic Puzzles with Haskell

We can use Haskell to solve logical puzzles such as the famous Lady or Tiger puzzles by Raymond Smullyan [34].





Here is the first puzzle. There are two rooms, and a prisoner has to choose between them. Each room contains either a lady or a tiger. In the first test the prisoner has to choose between a door with the sign "In this room there is a lady, and in the other room there is a tiger", and a second door with the sign "In one of these rooms there is a lady and in the other room there is a tiger." A final given is that one of the two signs tells the truth and the other does not. Here is a Haskell implementation that states the puzzle:

```
data Creature = Lady | Tiger deriving (Eq,Show)

sign1, sign2 :: (Creature,Creature) -> Bool
sign1 (x,y) = x == Lady && y == Tiger
sign2 (x,y) = x /= y
```

And here is the Haskell solution:

```
solution1 :: [(Creature,Creature)]
solution1 = [ (x,y) | x <- [Lady,Tiger],
                      y <- [Lady,Tiger],
                      (sign1 (x,y) && not (sign2 (x,y)))
                      || (not (sign1 (x,y)) && sign2 (x,y))]
```

Running this reveals that the first room has a tiger in it, and the second room a lady:

```
*WLH> solution1
[(Tiger,Lady)]
```

Exercise 1.1 *The second puzzle of the book runs as follows. Again there are two signs. The sign on the first door says: "At least one of these rooms contains a lady." The sign on the second door says: "A tiger is in the other room." This time either the statements are both true or both false. Give a Haskell implementation of* `solution2` *that solves the puzzle. You will also have to write functions for the new signs, of course.*

On the island of knights and knaves made famous in another logic puzzle book by Raymond Smullyan [35], there are two kinds of people. Knights always tell the truth, and knaves always lie. Of course, if you ask inhabitants of the island whether they are knights, they will always say "yes."

Suppose John and Bill are residents of the island. They are standing next to each other, with John left and Bill right. John says: "We are both knaves." Who is what? Here is a Haskell solution:

```
data Islander = Knight | Knave deriving (Eq,Show)

john :: (Islander,Islander) -> Bool
john (x,y) = (x,y) == (Knave,Knave)

solution3 :: [(Islander,Islander)]
solution3 = [(x,y) | x <- [Knight,Knave],
                     y <- [Knight,Knave],
                     john (x,y) == (x == Knight) ]
```

This reveals that John is a knave and Bill a knight:

```
*WLH> solution3
[(Knave,Knight)]
```

Exercise 1.2 *In this puzzle, again John is on the left, Bill on the right. John says: "We are both of the same kind." Bill says: "We are both of different kinds." Who is what? Implement a Haskell solution.*

A Puzzling Program Use a minute or so to analyze the following program.

```
main = putStrLn (s ++ show s)
  where s = "main = putStrLn (s ++ show s) \n  where s = "
```

This has the following ingredients that may still be unfamiliar to you:

- `show` for displaying an item as a string (if the item to be displayed is already a string, then this string is quoted);
- `\n` for the newline character.

Now that this was explained to you, reflect again, and tackle the exercises below.

Exercise 1.3 *Predict what will happen when the function* `main` *is executed. Next write down your prediction, and check it by executing the function.*

Exercise 1.4 *(Only for those who know some logic.) What does this have to do with logic? Hint: think of Kurt Gödel's famous proof of the incompleteness of the first order theory of arithmetic.*

1.7 Summary

If this was your first acquaintance with Haskell, make sure to actually *play around* and *do* some exercises. You will find that you will be up and running in no time.

If you already have some Haskell experience, this first chapter should have been plain sailing for you.

In the rest of this course we will focus on a particular aspect of functional programming: the use of executable specifications for programs.

Chapter 2

Functional Imperative Style

Abstract This chapter explains how imperative algorithms can be written directly in functional style. We show how to define while loops as functional programs, and how this trick can be used to write what we call functional imperative code.

Key words: Functional programming versus imperative programming, loops versus recursion, while loops, repeat loops, for loops.

2.1 Introduction

We will use literate Haskell again.

```
module FIS

where

import List
```

2.2 Loops Versus Recursion

It is common wisdom that functional programmers use recursion where-ever imperative programmers use loops. Consider the following imperative program:

```
x := 0;
n := 0;
while n < y do
   {
     x := x + 2*n + 1;
     n := n + 1;
   }
return x;
```

To show that this computes the value of y^2 in x, the key is to show that the loop invariant $x = n^2$ holds for the while loop.

A functional programmer would perhaps write this program as follows:

```
f :: Int -> Int
f y = f' y 0 0

f' :: Int -> Int -> Int -> Int
f' y x n = if n < y then
             let
               x' = x + 2*n + 1
               n' = n + 1
             in f' y x' n'
             else x
```

This is quite close to the imperative style. The call `f' y 0 0` initializes the parameters that correspond to the local variables x and n in the imperative version. The parameter y corresponds to the global variable y in the imperative version.

Instead of proving a loop invariant, one can prove that f' indeed returns the square of y, for non-negative y. Recursive procedures suggest inductive proofs. In this case we can use induction on y, as follows.

Base case If $y = 0$, then $f'\ 0\ 0\ 0$ returns 0, by the definition of f'. This is correct, for $0^2 = 0$.

Induction step Assume for $y = m$ the function call $f'\ m\ x\ m$ returns x with $x = m^2$. We have to show that for $y = m + 1$, the function call $f'(m + 1)\ x\ m$ returns $(m + 1)^2$.

Consider $f'(m+1)\ x\ m$. The test $m < m+1$ succeeds, so f' is called recursively, with arguments $f'(m + 1)(x + 2m + 1)(m + 1)$. The induction hypothesis gives $x = m^2$, so we get

$$f'(m + 1)(x + 2m + 1)(m + 1) = f'(m + 1)(m^2 + 2m + 1)(m + 1).$$

Basic algebra gives:

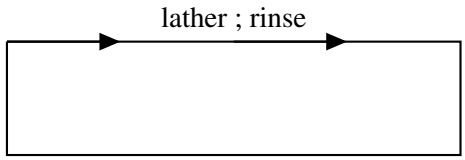
$$f'(m + 1)(m^2 + 2m + 1)(m + 1) = f'(m + 1)(m + 1)^2(m + 1),$$

and the definition of the function yields that $(m+1)^2$ is returned. This proves the induction step.

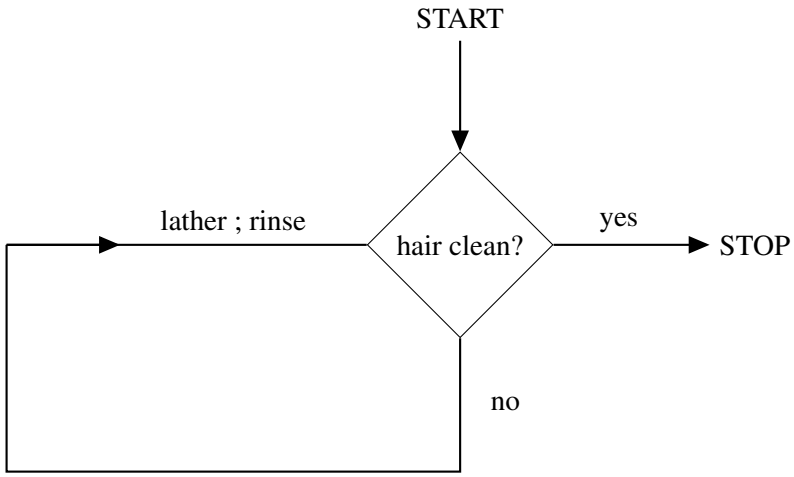
Ahem, this is not really any easier than proving a loop invariant. In the imperative version we have to deal with three variables x, y, n, and in the recursive functional version we reason about three function arguments.

2.3 While Loops in Functional Imperative Style

If taken literally, the compound action 'lather, rinse, repeat' would look like this:



Repeated actions usually have a *stop condition*: repeat the lather rinse sequence until your hair is clean. This gives a more sensible interpretation of the repetition instruction:



Written as an algorithm:

Hair wash algorithm

- while hair not clean do:
  1. lather;
  2. rinse.

Now let's add a while construct to Haskell. The two ingredients are a *test for loop termination* and a *step function* that determines the parameters for the next step in the loop. The termination test takes a number of parameters and returns a boolean, the step function takes the same parameters and computes new values for those parameters.

Suppose for simplicity that there is just one parameter. Here is another example loop:

- while even x do
  $x := x \div 2$.

Here $\div$ is the 'div' operator for integer division. The result of $x \div y$ is the integer you get if you divide x by y and throw away the remainder. Thus, $9 \div 2 = 4$.

A functional version, with the loop replaced by a recursive call:

```
g x = if even x then
      let
        x' = x `div` 2
      in g x'
      else x
```

g has a single parameter, and one can think of its definition as a combination of a test p and a step h, as follows:

```
p x = even x
h x = x `div` 2

g1 x = if p x then g1 (f x)
       else x
```

Let's make this explicit. Here is a definition of the general form of a while loop with a single parameter:

```
while1 :: (a -> Bool) -> (a -> a) -> a -> a
while1 p f x
  | p x       = while1 p f (f x)
  | otherwise = x
```

Another way to express this is in terms of the built-in Haskell function `until`:

```
neg :: (a -> Bool) -> (a -> Bool)
neg p = \x -> not (p x)

while1 = until . neg
```

This allows us to write the function g as follows:

```
g2 = while1 p h
```

It looks like the parameters have disappeared, but we can write out the test and step functions explicitly:

```
g3 = while1 (\x -> even x) (\x -> x `div` 2)
```

This is the functional version of the loop. This is how close functional programming really is to imperative programming.

As another example, consider the following loop for computing the least fixpoint of a function (in case you don't know what a least fixpoint computation is, you should consult Section 4.3 in Chapter 4 below):

Least fixpoint algorithm

- while $x \neq f(x)$ do
  $x := f(x)$.

In pure functional style, this looks as follows:

```
lfp :: Eq a => (a -> a) -> a -> a
lfp f x | x == f x  = x
        | otherwise = lfp f (f x)
```

In functional imperative style, we get:

```
lfp' :: Eq a => (a -> a) -> a -> a
lfp' f = while1 (\x -> x /= f x) (\x -> f x)
```

A denser version of *while1* combines the two functions:

```
while1' :: (a -> (Bool, a)) -> a -> a
while1' f x
  | fst (f x) = while1' f (snd (f x))
  | otherwise = x
```

The least fixpoint function using *while1'*:

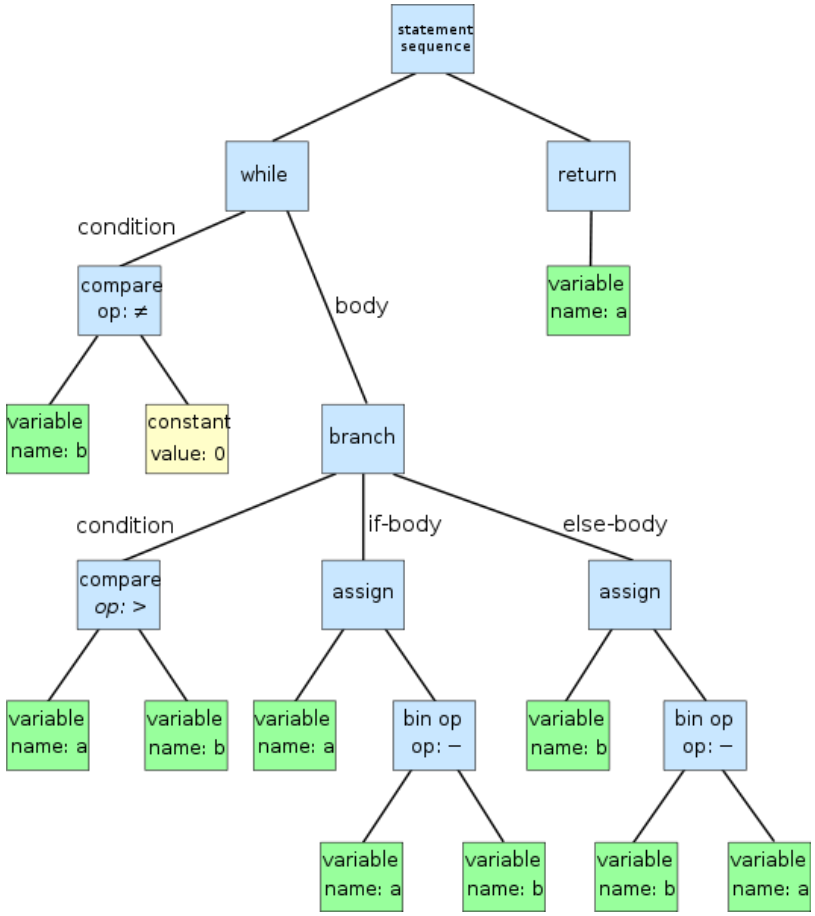
```
lfp'' :: Eq a => (a -> a) -> a -> a
lfp'' f = while1' (\x -> (x /= f x, f x))
```

That is not really more transparent. Let's not fall into the functional programmer's trap of trying to outsmart the human readers of our code.

Here is a version of `while` for the case where there a two parameters:

```
while2 :: (a -> b -> Bool)
       -> (a -> b -> (a,b))
       -> a -> b -> b
while2 p f x y
  | p x y     = let (x',y') = f x y in while2 p f x' y'
  | otherwise = y
```

As an example of the use of this, consider Euclid's GCD algorithm.



First in imperative pseudo-code.

Euclid's GCD algorithm

1. while $x \neq y$ do
   if $x > y$ then $x := x - y$ else $y := y - x$;
2. return y.

Next in functional imperative style:

```
euclidGCD :: Integer -> Integer -> Integer
euclidGCD = while2
            (\ x y -> x /= y)
            (\ x y -> if x > y
                      then (x-y,y)
                      else (x,y-x))
```

We can now write the original squaring function in functional imperative style:

```
sqr :: Int -> Int
sqr y = let
          x = 0
          n = 0
        in sqr' y x n

sqr' y = while2
          (\ _ n -> n < y)
          (\ x n -> (x + 2*n + 1, n+1))
```

Note that since the y parameter does not change it does not have to be inside the *while* part of the definition. Thus, functional imperative style allows for a clear distinction between the global and local parameters of a loop instruction.

Versions of *while* for the case where there are three parameters or more are straightforward:

```
while3 :: (a -> b -> c -> Bool)
       -> (a -> b -> c -> (a,b,c))
       -> a -> b -> c -> c
while3 p f x y z
  | p x y z   = let
                  (x',y',z') = f x y z
                in while3 p f x' y' z'
  | otherwise = z
```

2.4 Repeat Loops in Functional Imperative Style

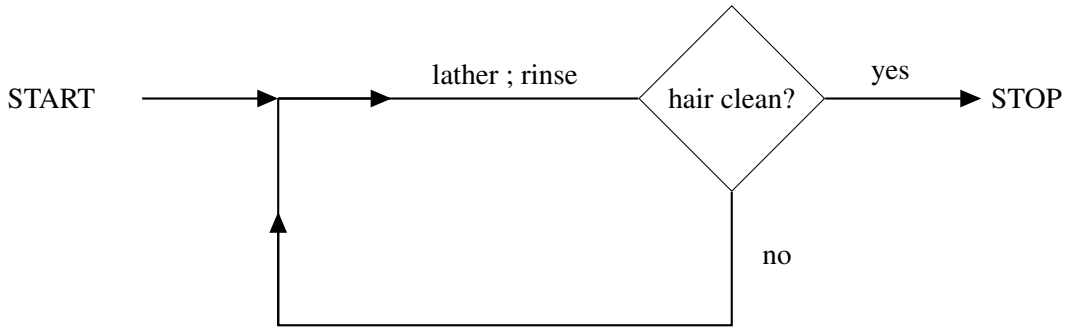
Consider the following alternative hair wash algorithm:

Another hair wash algorithm

- repeat
  1. lather;
  2. rinse;

  until hair clean.

This can be pictured as follows:



The difference with the previous algorithm is that this one starts with a 'lather; rinse' sequence, no matter the initial state of your hair.

Repeat loops can be defined in terms of while loops, as follows:

$$\text{repeat } P \text{ until } C$$

is equivalent to:

$$P;\ \text{while } \neg C \text{ do } P.$$

This gives us a recipe for repeat loops in functional imperative style, using repeat wrappers such as the following:

```
repeat1 :: (a -> a) -> (a -> Bool) -> a -> a
repeat1 f p = while1 (\ x -> not (p x)) f . f
```

```
repeat2 :: (a -> b -> (a,b))
        -> (a -> b -> Bool) -> a -> b -> b
repeat2 f p x y = let
     (x1,y1) = f x y
     negp    = (\ x y -> not (p x y))
  in while2 negp f x1 y1
```

```
repeat3 :: (a -> b -> c -> (a,b,c))
        -> (a -> b -> c -> Bool) -> a -> b -> c -> c
repeat3 f p x y z = let
     (x1,y1,z1) = f x y z
     negp       = (\ x y z -> not (p x y z))
  in while3 negp f x1 y1 z1
```

2.5 For Loops in Functional Imperative Style

A natural way to express an algorithm for computing the factorial function, in imperative style, is in terms of a "for" loop:

Factorial Algorithm

1. $t := 1$;
2. for i in $1 \ldots n$ do $t := i * t$;
3. return t.

For a faithful rendering of this in Haskell, we define a function for the "for" loop:

```
for :: [a] -> (a -> b -> b) -> b -> b
for [] f y = y
for (x:xs) f y = for xs f (f x y)
```

Note that this is a variant of `foldl`, witness the following alternative definition:

```
for xs f y = foldl (flip f) y xs
```

This gives the following Haskell version of the algorithm:

```
fact :: Integer -> Integer
fact n = for [1..n] (\ i t -> i*t) 1
```

If we wish, we can spell out the initialisation, as follows:

```
fact :: Integer -> Integer
fact n = let
           t = 1
         in fact' n t

fact' :: Integer -> Integer -> Integer
fact' n = for [1..n] (\ i t -> i*t)
```

Let's contrast this with a version of the algorithm that uses a "while" loop:

Another Factorial Algorithm

1. $t := 1$;
2. while $n \neq 0$ do
   (a) $t := n * t$;
   (b) $n := n - 1$;
3. return t.

In functional imperative style, this becomes:

```
factorial :: Integer -> Integer
factorial n = let
                t = 1
              in factorial' n t

factorial' = while2 (\ n _ -> n /= 0)
                    (\ n t ->  let
                      t' = n*t
                      n' = n-1
                    in (n',t'))
```

It is clear that the version in terms of the "for" loop is simpler.

In fact, wherever imperative programmers use "for" loops, functional programmers tend to use `fold` constructions.

The standard way to define the factorial function in functional programming is:

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

The function `product` is predefined. If we look up the definition of `sum` and `product` in the Haskell prelude, we find:

```
sum, product      :: (Num a) => [a] -> a
sum               =  foldl (+) 0
product           =  foldl (*) 1
```

Here is a version of "for" where the step function has an additional argument:

```
for2 :: [a] -> (a -> b -> c -> (b,c))
           -> b -> c -> c
for2 [] f _ z = z
for2 (x:xs) f y z = let
    (y',z') = f x y z
  in
    for2 xs f y' z'
```

With two additional arguments:

```
for3 :: [a] -> (a -> b -> c -> d -> (b,c,d))
           -> b -> c -> d -> d
for3 [] f _ _ u = u
for3 (x:xs) f y z u = let
    (y',z',u') = f x y z u
  in
    for3 xs f y' z' u'
```

And so on.

We can also count down instead of up:

```
fordown :: [a] -> (a -> b -> b) -> b -> b
fordown = for . reverse
```

```
fordown2 :: [a] -> (a -> b -> c -> (b,c))
                -> b -> c -> c
fordown2 = for2 . reverse
```

```
fordown3 :: [a] -> (a -> b -> c -> d -> (b,c,d))
                      -> b -> c -> d -> d
fordown3 = for3 . reverse
```

2.6 Summary, and Further Reading

This chapter has introduced you to programming in functional imperative style.

Iteration versus recursion is the topic of chapter 2 of the classic [1]. This book is freely available on internet, from address `http://infolab.stanford.edu/~ullman/focs.html`.

Chapter 3

Algorithm Specification for Assertive Coding

Abstract We show how to specify preconditions, postconditions, assertions and invariants, and how to wrap these around functional code or functional imperative code. We illustrate the use of this for writing programs for automated testing of code that is wrapped in appropriate assertions. We call this assertive coding. An assertive version of a function f is a function f' that behaves exactly like f as long as f complies with its specification, and aborts with error otherwise. This is a much stronger sense of self-testing than what is called self-testing code (code with built-in tests) in test driven development. The chapter gives examples of how to use (inefficient) specification code to test (efficient) implementation code, and how to turn assertive code into production code by replacing the self-testing versions of the assertion wrappers by self-documenting versions that skip the assertion tests.

Key words: Functional algorithm specification, algorithm verification, specification based testing, test automation, assertive coding, self-testing code.

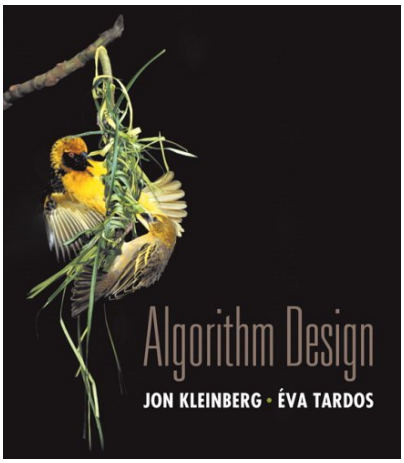
3.1 Introduction

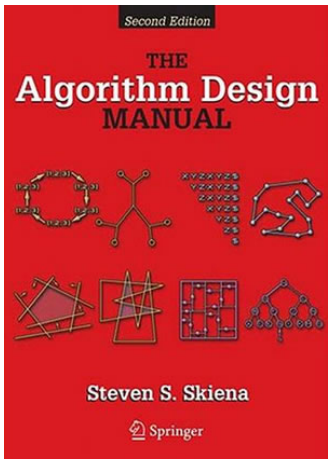
Declaration of a literate Haskell module. This module imports the standard `List` module plus the module with the code of the previous chapter.

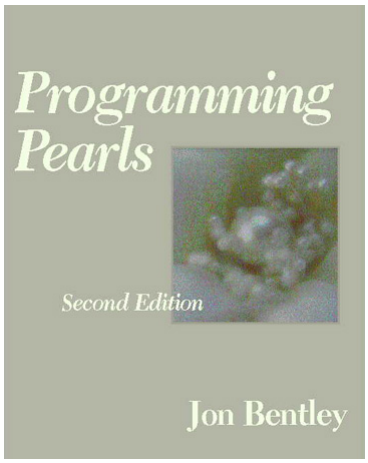
```
module AS

where

import List
import FIS
```

3.2 Algorithm Design and Specification

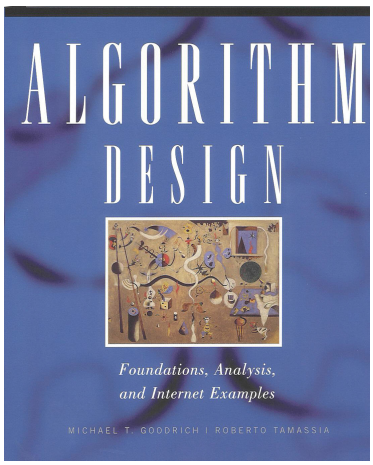
Some excellent books:

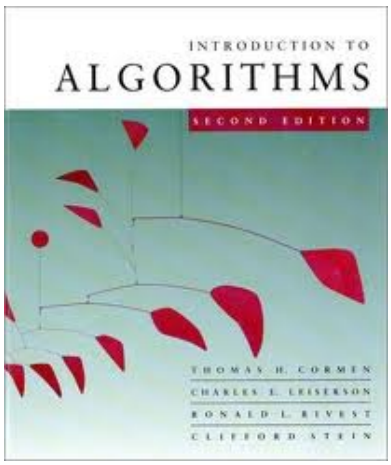


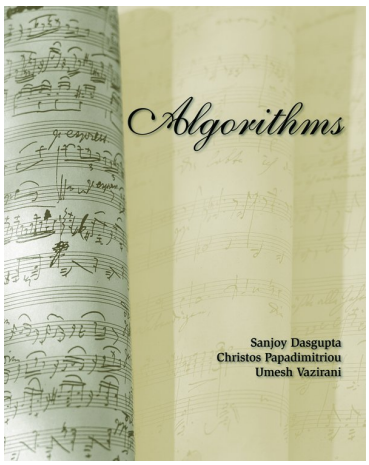




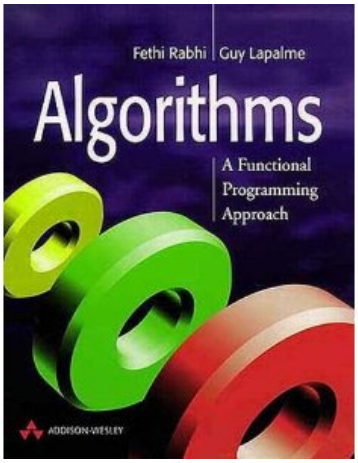
And some more:

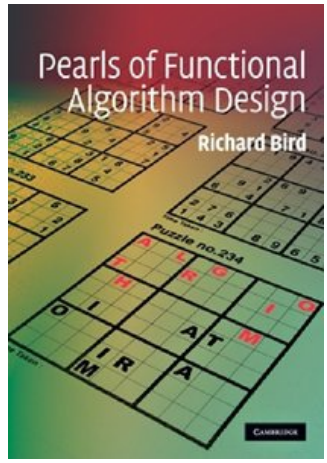


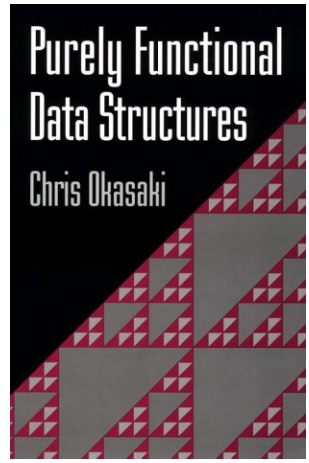




This course will teach you a purely functional way to look at algorithms as they are designed, presented and analyzed in these books. This complements the approach of [31] and [5], which propose to give 'functional' solutions for 'classical' algorithmic problems. Instead, this course will show that classical algorithmic problems *plus their classical solutions* can be presented in a purely functional way.







3.3 Preconditions, Postconditions, Assertions and Invariants

A (Hoare) assertion about an imperative program [21] has the form

$$\{\text{Pre}\}\ \text{Program}\ \{\text{Post}\}$$

where *Pre* and *Post* are conditions on states.

This Hoare statement is true in state s if truth of *Pre* in s guarantees truth of *Post* in any state s' that is a result state of performing *Program* in state s.

One way to write assertions for functional code is as wrappers around functions. This results in a much stronger sense of self-testing than what is called self-testing code (code with built-in tests) in test driven development [3].

The precondition of a function is a condition on its input parameter(s), the postcondition is a condition on its value.

Here is a precondion wrapper for functions with one argument. The wrapper takes a precondition property and a function and produces a new function that behaves as the old one, provided the precondition is satisfied,

```
pre1 :: (a -> Bool) -> (a -> b) -> a -> b
pre1 p f x = if p x then f x
             else error "pre1"
```

A postcondition wrapper for functions with one argument.

```
post1 :: (b -> Bool) -> (a -> b) -> a -> b
post1 p f x = if p (f x) then f x
              else error "post1"
```

This can be used to specify the expected behaviour of a function. The g function should always output an odd integer:

```
godd = post1 odd g
```

Note that `godd` has the same type as `g`, and that the two functions compute the same value whenever `godd` is defined (i.e., whenever the output value of `g` satisfies the postcondition).

More generally, an assertion is a condition that may relate input parameters to the computed value. Here is an assertion wrapper for functions with one argument. The wrapper wraps a binary relation expressing a condition on input and output around a function and produces a new function that behaves as the old one, provided that the relation holds.

```
assert1 :: (a -> b -> Bool) -> (a -> b) -> a -> b
assert1 p f x = if p x (f x) then f x
                else error "assert1"
```

Example use:

```
gA = assert1 (\ i o -> signum i == signum o) g
```

Note that `gA` has the same type as `g`. Indeed, as long as the assertion holds, `gA` and `g` compute the same value.

An *invariant* of a program P in a state s is a condition C with the property that if C holds in s then C will also hold in any state that results from execution of P in s. Thus, invariants are Hoare assertions of the form:

$$\{C\}\ \text{Program}\ \{C\}$$

If you wrap an invariant around a step function in a loop, the invariant documents the expected behaviour of the loop.

First an invariant wrapper for the case of a function with a single parameter: a function `f :: a -> a` fails an invariant `p :: a -> Bool` if the input of the function satisfies p but the output does not:

```
invar1 :: (a -> Bool) -> (a -> a) -> a -> a
invar1 p f x =
  let
    x' = f x
  in
  if p x && not (p x') then error "invar1"
  else x'
```

Example of its use:

```
gsign = invar1 (>0) g
```

Another example:

```
gsign' = invar1 (<0) g
```

We can also use the invariant inside a *while* loop:

```
g3' =  while1 (\x -> even x)
              (invar1 (>0) (\x -> x `div` 2))
```

3.4 An Assertive List Merge Algorithm

Consider the problem of merging two sorted lists into a result list that is also sorted, and that contains the two original lists as sublists.

For writing specifications an operator for Boolean implication is good to have.

```
infix 1 ==>

(==>) :: Bool -> Bool -> Bool
p ==> q = (not p) || q
```

The specification for merge uses the following property:

```
sortedProp :: Ord a => [a] -> [a] -> [a] -> Bool
sortedProp xs ys zs =
  (sorted xs && sorted ys) ==> sorted zs

sorted :: Ord a => [a] -> Bool
sorted [] = True
sorted [_] = True
sorted (x:y:zs) = x <= y && sorted (y:zs)
```

Each list should occur as a sublist in the merge:

```
sublistProp :: Eq a => [a] -> [a] -> [a] -> Bool
sublistProp xs ys zs = sublist xs zs && sublist ys zs

sublist :: Eq a => [a] -> [a] -> Bool
sublist [] _ = True
sublist (x:xs) ys = elem x ys && sublist xs (ys \\ [x])
```

Here is an assertion wrapper for functions with two parameters:

```
assert2 ::  (a -> b -> c -> Bool)
            -> (a -> b -> c) -> a -> b -> c
assert2 p f x y =
  if p x y (f x y) then f x y
  else error "assert2"
```

A merge function:

```
merge :: Ord a => [a] -> [a] -> [a]
merge xs [] = xs
merge [] ys = ys
merge (x:xs) (y:ys) = if x <= y
                        then x : merge xs (y:ys)
                        else y : merge (x:xs) ys
```

And an assertive version of the `merge` function:

```
mergeA :: Ord a => [a] -> [a] -> [a]
mergeA = assert2 sortedProp
           $ assert2 sublistProp merge
```

We have wrapped an assertion around a wrap of an assertion around a function. This cause no problems, for the wrap of an assertion around a function has the same type as the original function.

Note that `sortedProp` is an implication. If we apply test-merge to a list that is not sorted, the property still holds:

```
*AS> mergeA [2,1] [3..10]
[2,1,3,4,5,6,7,8,9,10]
```

Exercise 3.1 `merge` *can be used as follows, to create a function for list sorting:*

```
mergeSrt ::  Ord a => [a] -> [a]
mergeSrt [] = []
mergeSrt (x:xs) = merge [x] (mergeSrt xs)
```

Find a suitable assertion, and write an assertive version of this.

3.5 Assertive Versions of the GCD Algorithm

A precondition of the GCD algorithm is that its arguments are positive integers. This can be expressed as follows:

```
euclid = assert2 (\ m n _ -> m > 0 && n > 0)
                 euclidGCD
```

This function has the same type as `euclidGCD`. Both `euclid` and `euclidGCD` are partial functions; in fact they are the same partial function. The difference is that `euclid` aborts where `euclidGCD` diverges.

If we want to check that `euclid` behaves correctly, we can test by using very specific input. E.g., if p and q are different prime numbers, we know that the GCD of p and q equals 1. Here is a test of Euclid's algorithm based on this knowledge.

```
testEuclid1 :: Int -> Bool
testEuclid1 k = let
    primes = take k (filter prime [2..])
  in
    and [ euclid p q == 1 |
          p <- primes, q <- primes, p /= q ]
```

This uses the following implementation for the property of being a prime number:

```
prime :: Integer -> Bool
prime n =
  n > 1 && all (\ x -> rem n x /= 0) xs
  where xs = takeWhile (\ y -> y^2 <= n) [2..]
```

We can also test using assertions. An example assertion for Euclid's algorithm is that the result value is the greatest common divisor of the two inputs. To express that, we implement a bit of logic.

The Haskell function `all` has type

`(a -> Bool) -> [a] -> Bool`.

Sometimes it is more convenient to have a universal quantifier of type

`[a] -> (a -> Bool) -> Bool`.

Here it is:

```
forall = flip all
```

The definition of GCD is given in terms of the *divides* relation. An integer n divides another integer m if there is an integer k with $nk = m$, in other words, if the process of dividing m by n leaves a remainder 0.

```
divides :: Integer -> Integer -> Bool
divides n m = rem m n == 0
```

An integer n is the GCD of k and m if n divides both k and m, and every divisor of k and m also divides n.

```
isGCD :: Integer -> Integer -> Integer -> Bool
isGCD k m n = divides n k && divides n m &&
              forall [1..min k m]
              (\ x -> (divides x k && divides x m)
                        ==> divides x n)
```

Here is an assertive version of Euclid's GCD function:

```
euclid' :: Integer -> Integer -> Integer
euclid' = assert2 isGCD euclid
```

In case you don't see what's happening: the new function `euclid'` computes the same value as `euclid`, but the assertion guarantees that if it would ever compute the *wrong* value, an exception will be raised.

Exercise 3.2 *Suppose we want to use this for performing automated tests for large ranges of values, as follows:*

```
testEuclid :: Integer -> Bool
testEuclid k = let
    triv = (\_ -> True)
  in
    and [ triv $ (assert2 isGCD euclid) n m |
                      n <- [1..k], m <- [1..k] ]
```

Use your Haskell system to check if this is a reasonable test.

In any case, the following works:

```
testEuclid :: Integer -> Bool
testEuclid k =
  and [ (assert2 isGCD euclid) n m > 0 |
                       n <- [1..k], m <- [1..k] ]
```

The subtlety here is that the assertion allows us to define what is right and what is wrong. Imperative programs are dumb. Functional programs are dumb, too. But assertions allow us to relate what a program *does* to what the program is *supposed to do*.

The following test succeeds in under 25 seconds on my 3 Ghz dualcore machine:

```
*AS> testEuclid 300
True
```

Another thing we can do is find a suitable invariant. An invariant for Euclid's step function is that the set of common divisors does not change. The following function gives the common divisors of two integers:

```
divisors :: Integer -> Integer -> [Integer]
divisors m n = let
    k = min m n
  in [ d | d <- [2..k], divides d m, divides d n ]
```

We can use this in an assertion about the step function, as follows:

```
sameDivisors x y (x',y') = divisors x y == divisors x' y'
```

Wrapping the assertion around the step function gives:

```
euclidGCD' :: Integer -> Integer -> Integer
euclidGCD' = while2
             (\ x y -> x /= y)
             (assert2 sameDivisors
             (\ x y -> if x > y
                       then (x-y,y)
                       else (x,y-x)))
```

Note that the assertion `sameDivisors` is in fact an invariant, stated as a relation between input and output of the step function.

For invariants that do not relate input and output, we can use the following invariant wrapper for step functions with two arguments:

```
invar2 :: (a -> b -> Bool) ->
          (a -> b -> (a,b)) ->
           a -> b -> (a,b)
invar2 p f x y =
  let
    (x',y') = f x y
  in
    if p x y && not (p x' y') then error "invar2"
    else (x',y')
```

As an example of how this is used, consider the following invariant. If d divides both x and y before the step, then d should divide x and y after the step.

Let's add a parameter for such a divisor d, and state the invariant:

```
euclidGCD'' :: Integer -> Integer -> Integer -> Integer
euclidGCD'' = \ d -> while2
             (\ x y -> x /= y)
             (invar2 (\ x y -> divides d x && divides d y)
              (\ x y -> if x > y
                        then (x-y,y)
                        else (x,y-x)))
```

Here is how this can be used in a test:

```
testEuclid2 :: Integer -> Bool
testEuclid2 k =
  and [ euclidGCD'' d n m >= 0 |
          n <- [1..k], m <- [1..k], d <- [2..min n m] ]
```

3.6 The Extended GCD Algorithm

The extended GCD algoritm extends the Euclidean algorithm, as follows. Instead of finding the GCD of two (positive) integers M and N it finds two integers x and y satisfying the so-called Bézout identity (or: Bézout equality):

$$xM + yN = \gcd(M, N).$$

For example, for arguments $M = 12$ and $N = 26$, the extended GCD algorithm gives the pair $x = -2$ and $y = 1$. And indeed, $-2 * 12 + 26 = 2$, which is the GCD of 12 and 26.

Here is an imperative (iterative) version of the algorithm:

Extended GCD algorithm

1. Let positive integers a and b be given.
2. $x := 0$;
3. lastx $:= 1$;
4. $y := 1$;
5. lasty $:= 0$;
6. while $b \neq 0$ do
   (a) $(q, r) :=$ quotRem(a, b);
   (b) $(a, b) := (b, r)$;
   (c) $(x, \text{lastx}) := (\text{lastx} - q * x, x)$;
   (d) $(y, \text{lasty}) := (\text{lasty} - q * y, y)$.
7. Return $(\text{lastx}, \text{lasty})$.

Functional imperative version, in Haskell:

```
ext_gcd :: Integer -> Integer -> (Integer,Integer)
ext_gcd a b = let
   x = 0
   y = 1
   lastx = 1
   lasty = 0
  in ext_gcd' a b x y (lastx,lasty)

ext_gcd' = while5 (\ _ b _ _ _ ->  b /= 0)
                  (\ a b x y (lastx,lasty) -> let
                    (q,r)   = quotRem a b
                    (x',lastx') = (lastx-q*x,x)
                    (y',lasty') = (lasty-q*y,y)
                  in (b,r,x',y',(lastx',lasty')))
```

This uses a `while5` loop:

```
while5 :: (a -> b -> c -> d -> e -> Bool)
       -> (a -> b -> c -> d -> e -> (a,b,c,d,e))
       -> a -> b -> c -> d -> e -> e
while5 p f x y z v w
  | p x y z v w = let
                    (x',y',z',v',w') = f x y z v w
                  in while5 p f x' y' z' v' w'
  | otherwise = w
```

Bézout's identity is turned into an assertion, as follows:

```
bezout :: Integer -> Integer -> (Integer,Integer) -> Bool
bezout m n (x,y) = x*m + y*n == euclid m n
```

Use of this to produce assertive code for the extended algorithm:

```
ext_gcdA = assert2 bezout ext_gcd
```

A functional (recursive) version of the extended Euclidean algorithm:

```
fct_gcd :: Integer -> Integer -> (Integer,Integer)
fct_gcd a b =
  if b == 0
  then (1,0)
  else
     let
       (q,r) = quotRem a b
       (s,t) = fct_gcd b r
     in (t, s - q*t)
```

Again, an assertive version of this:

```
fct_gcdA = assert2 bezout fct_gcd
```

3.7 More Assertion and Invariant Wrappers

An assertion wrapper for functions with three arguments:

```
assert3 ::  (a -> b -> c -> d -> Bool)
             -> (a -> b -> c -> d) -> a -> b -> c -> d
assert3 p f x y z =
  if p x y z (f x y z) then f x y z
  else error "assert3"
```

An invariant wrapper for step functions with three arguments:

```
invar3 :: (a -> b -> c -> Bool) ->
          (a -> b -> c -> (a,b,c)) ->
           a -> b -> c -> (a,b,c)
invar3 p f x y z =
  let
    (x',y',z') = f x y z
  in
   if p x y z && not (p x' y' z') then error "invar3"
   else (x',y',z')
```

An assertion wrapper for functions with four arguments:

```
assert4 ::  (a -> b -> c -> d -> e -> Bool)
             -> (a -> b -> c -> d -> e)
             -> a -> b -> c -> d -> e
assert4 p f x y z u =
  if p x y z u (f x y z u) then f x y z u
  else error "assert4"
```

An invariant wrapper for step functions with four arguments:

```
invar4 :: (a -> b -> c -> d -> Bool) ->
          (a -> b -> c -> d -> (a,b,c,d)) ->
           a -> b -> c -> d -> (a,b,c,d)
invar4 p f x y z u =
  let
    (x',y',z',u') = f x y z u
  in
   if p x y z u && not (p x' y' z' u')
   then error "invar4"
   else (x',y',z',u')
```

An assertion wrapper for functions with five arguments:

```
assert5 ::  (a -> b -> c -> d -> e -> f -> Bool)
             -> (a -> b -> c -> d -> e -> f)
             -> a -> b -> c -> d -> e -> f
assert5 p f x y z u v =
  if p x y z u v (f x y z u v) then f x y z u v
  else error "assert5"
```

An invariant wrapper for step functions with five arguments:

```
invar5 :: (a -> b -> c -> d -> e -> Bool) ->
          (a -> b -> c -> d -> e -> (a,b,c,d,e)) ->
           a -> b -> c -> d -> e -> (a,b,c,d,e)
invar5 p f x y z u v =
  let
    (x',y',z',u',v') = f x y z u v
  in
   if p x y z u v && not (p x' y' z' u' v')
   then error "invar5"
   else (x',y',z',u',v')
```

3.8 Assertive Code is Efficient Self-Documenting Code

More often than not, an assertive version of a function is much less efficient than the regular version: the assertions are inefficient specification algorithms to test the behaviour of efficient functions.

But this does not matter. To turn assertive code into self-documenting production code, all you have to do is load a module with alternative definitions of the assertion and invariant wrappers.

Take the definition of `assert1`. This is replaced by:

```
assert1 :: (a -> b -> Bool) -> (a -> b) -> a -> b
assert1 _ = id
```

And so on for the other wrappers. See module `AssertDoc` in Appendix C.

The assertions are still in the code, but instead of being executed they now serve as documentation. The assertive version of a function executes exactly as the version without the assertion. Assertive code comes with absolutely no efficiency penalty.

3.9 Summary

State-of-the-art functional programming languages are well suited not only for implementing algorithms but also for specifying algorithms and wrap the specifications around the implementations, to produce assertive code. In fact you can do this with any language, but the cost is higher (sometimes much higher) for less abstract languages.

We have seen that, contrary to popular opinion, imperative algorithms can be implemented directly in a functional language. The while loops that were employed above are explicit about the parameters that are used for checking loop termination and for defining the loop step. This often makes the implemented algorithms more perspicuous.

Suppose a program (implemented function) fails its implemented assertion. What should we conclude? This is a pertinent question, for the assertion itself is a piece of code too, in the same programming language as the function that we want to test. So what are we testing, the correctness of the code or the correctness of the implememented specification for the code?

In fact, we are testing both at the same time. Therefore, the failure of a test can mean either of two things, and we should be careful to find out wat our situation is:

1. There is something wrong with the program.
2. There is something wrong with the specification of the assertion for the program.

It is up to us to find out which case we are in.

In both cases it is important to find out where the problem resides. In the first case, we have to fix a code defect, and we are in a good position to do so because we have the specification as a yardstick. In the second case, we are not ready to fix code defects. First and foremost, we have to fix a defect in our understanding of what our program is supposed to do. Without that growth in understanding, it will be very hard indeed to detect and fix possible defects in the code itself.

Chapter 4

Graph Algorithms

Abstract This chapter discusses a number of well-known graph algorithms, develops testable specifications for them, and uses the specifications to write assertive (self-testing) versions of the algorithms.

Key words: Graph algorithms, functional programming, algorithm specification, algorithm verification, specification based testing, test automation, self-testing code.

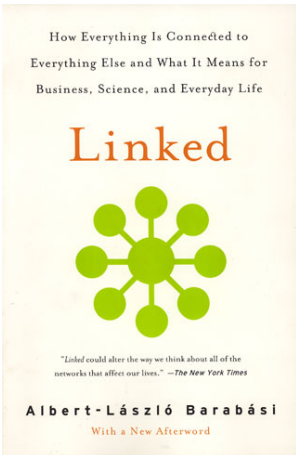
4.1 Introduction

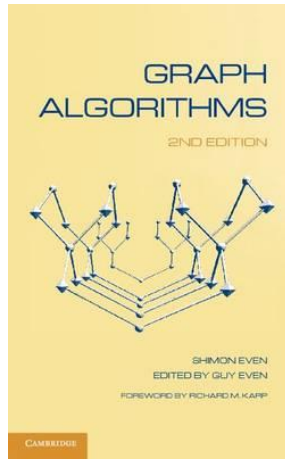
This chapter discusses a number of well-known graph algorithms, develops testable specifications for them, and uses the specifications to write assertive versions of the algorithms. The implementations use literate Haskell.

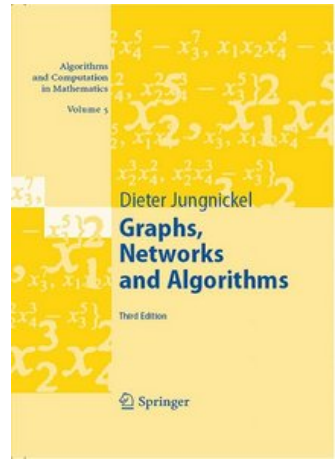
```
module GA

where

import List
import While
import Assert
```

The modules `While` and `Assert` give the code for while loops and for assertion and invariant wrappers that was developed and discussed in the chapter on Algorithm Specification.

Background on the key importance of graph theory for the analysis of what goes on in social and other networks can be found in [2]. An enlightening introduction to graph algorithms is [17].

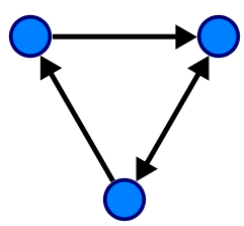






4.2 A Graph Reachability Algorithm

A directed graph G without parallel edges is a pair (V, E) with $E \subseteq V^2$. If $(v_1, v_2) \in E$, we write this as $v_1 \to v_2$, and we say that there is an edge from v_1 to v_2.



A vertex y is reachable from a vertex x in $G = (V, E)$ if there is a path of $\to$ edges from x to y, or, equivalently, if $(x, y) \in E^*$, where E^* is the reflexive transitive closure of E.

In an algorithm for this, we can assume G is given by its edge set E. Here is an algorithm that computes the set of reachable vertices from a given vertex.

Graph reachability algorithm

- Let edge set E and vertex x be given;
- $C := \{x\}$, $M := \{x\}$;
- while $C \neq \emptyset$ do:
  - select $y \in C$;
  - $N := \{z \mid (y, z) \in E, z \notin M\}$;
  - $C := (C - \{y\}) \cup N$;
  - $M := M \cup N$;
- return M.

The algorithm works by maintaining a set C of current nodes. Initially, only x is current. Each node can be either marked (in M) or unmarked. The marked nodes are the nodes that have been current (have been in C) at some stage in the past, or are current now. Each step in the algorithm removes an item y from C, and adds all its unmarked successors to C, and to M. The algorithm halts when C becomes empty, at which stage M gives the nodes that are reachable from x.

Note that the while loop in the algorithm has two parameters, C and M; the variable N is local to the loop. Here is a Haskell version of the algorithm that implements these set parameters as lists:

```
reachable :: Eq a => [(a,a)] -> a -> [a]
reachable g x = reachable' g [x] [x]

reachable' :: Eq a => [(a,a)] -> [a] -> [a] -> [a]
reachable' g = while2
   (\ current _ -> not (null current))
   (\ current marked -> let
     (y,rest) = (head current, tail current)
     newnodes = [ z  | (u,z) <- g, u == y,
                       notElem z marked   ]
     current' = rest ++ newnodes
     marked'  = marked ++ newnodes
  in
     (current', marked'))
```

Exercise 4.1 *How can this algorithm be tested? Can you find a reasonable assertion or a reasonable step invariant?*

Exercise 4.2 *Another way to implement a graph $G = (V, E)$ is as a list of vertices (a list of type $[a]$) together with an edge function (edge matrix), i.e. a function of type $a \to a \to$ Bool. Implement the* reachable *and* reachable' *functions using this alternative representation. The type declarations are:*

```
reachable1 :: Eq a => ([a], a -> a -> Bool) -> a -> [a]

reachable1' :: Eq a => ([a],a->a->Bool) -> [a] -> [a] -> [a]
```

A directed graph is *(strongly) connected* if for every pair of vertices x, y, there is a path from x to y. Here is an implementation in terms of reachability.

```
isConnected :: Eq a => [(a,a)] -> Bool
isConnected g = let
    xs = map fst g `union` map snd g
  in
    forall xs (\x -> forall xs (\ y ->
       elem y (reachable g x)))
```

Exercise 4.3 *Write a function* `cyclic :: Eq a => [(a,a)] -> Bool` *that checks whether a list of edges has cycles. A cycle is a path $x \to \cdots \to x$, for some node x.*

4.3 Operations on Relations

The algorithm from the previous section should fit the following specification:

$$\text{reachable } E\ x = \{y \mid xE^*y\}$$

where E^* is the reflexive transitive closure of E.

To implement this specification, we need some operations on relations.

Binary relations as lists of ordered pairs:

```
type Rel a = [(a,a)]
```

The $\subseteq$ relation on sets (represented by lists):

```
containedIn :: Eq a => [a] -> [a] -> Bool
containedIn xs ys = forall xs (\ x -> elem x ys)
```

Set equality on sets is defined as $A = B \iff A \subseteq B \wedge B \subseteq A$. Implementation for sets represented as lists:

```
equalS :: Eq a => [a] -> [a] -> Bool
equalS xs ys = containedIn xs ys && containedIn ys xs
```

The relational composition of two relations R and S on a set A:

$$R \circ S = \{(x, z) \mid \exists y \in A\ (xRy \wedge ySz)\}$$

For the implementation, it is useful to declare a new infix operator for relational composition.

```
infixr 5 @@

(@@) :: Eq a => Rel a -> Rel a -> Rel a
r @@ s =
  nub [ (x,z) | (x,y) <- r, (w,z) <- s, y == w ]
```

Computing reflexive transitive closure of a binary relation: the reflexive transive closure of R is the relation that is the least fixpoint of the operation $\lambda S \mapsto S \cup (R \circ S)$, applied to the identity relation I.

Here is the least fixpoint operation on functions:

```
lfp :: Eq a => (a -> a) -> a -> a
lfp f x | x == f x  = x
        | otherwise = lfp f (f x)
```

To apply the least fixpoint of $\lambda S \mapsto S \cup (R \circ S)$, to I, you start with I.

In the first step you apply $\lambda S \mapsto S \cup (R \circ S)$ to I. This gives $I \cup (R \circ I)$, that is: $I \cup R$.

In the second step you apply $\lambda S \mapsto S \cup (R \circ S)$ to $I \cup R$. This gives

$$I \cup R \cup (R \circ (I \cup R)),$$

that is: $I \cup R \cup R^2$.

And so on, until the process reaches a fixpoint, that is, until there is some n for which

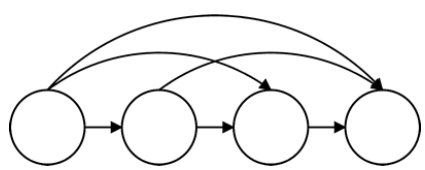
$$I \cup R \cup R^2 \cup \cdots \cup R^n = I \cup R \cup R^2 \cup \cdots \cup R^n \cup R^{n+1}.$$

At this point you have computed the smallest relation S that is reflexive and transitive and contains R, i.e., you have reached the reflexive transitive closure of R.

Here is the implementation (if you understand the code you can forget about the elaborate explanation above):

```
rtc :: Eq a => Rel a -> Rel a
rtc r = let
          xs = map fst r `union` map snd r
          i  = [(x,x) | x <- xs ]
        in lfp (\ s -> (s `union` (r@@s))) i
```

Computing transitive closure: same as computing reflexive transitive closure, but starting out from the relation R. Notation for the transitive closure of R is R^+.



```
tc :: Eq a => Rel a -> Rel a
tc r = lfp (\ s -> (s `union` (r @@ s))) r
```

Exercise 4.4 *Write a version of* `isConnected` *in terms of reflexive transitive closure of the edge list of a graph.*

If R is a binary relation on A and $x \in A$, then xR is the set $\{y \mid xRy\}$. Using this notation we can write the specification for `reachable` as:

$$\text{reachable } E\ x = xE^*.$$

Implementation of xR:

```
image :: Eq a => a -> Rel a -> [a]
image x r = [ y | (z,y) <- r, x == z ]
```

Use this for an assertive version of the graph reachability algorithm:

```
reachableA :: Eq a =>  [(a,a)] -> a -> [a]
reachableA =
  assert2 (\ g x ys -> equalS ys (image x (rtc g)))
           reachable
```

Note that this uses an inefficient algorithm for computing E^* to specify and test an efficient algorithm for E^*.

Extending the notation xR, let ${}_CR = \bigcup_{x \in C} xR$. In terms of this, a loop invariant for

```
reachable' E C M
```

can be expressed as:

$$xE^* = {}_CE^+ \cup M.$$

Exercise 4.5 *Check that this invariant holds for the step function of* `reachable'`*. Deduce that the return value of* `reachable'` *satifies* $xE^* = M$.

Exercise 4.6 *Write an assertive version of* `reachable'` *that uses this invariant.*

4.4 Minimum Spanning Trees of Graphs

A *weighted undirected graph* is a symmetric graph with weights assigned to the edges, in such manner that edges (x, y) and (y, x) have the same weight. Think of the weight as an indication of distance.

Here is a datatype for weighted graphs:

```
type Vertex = Int
type Edge = (Vertex,Vertex,Float)
type Graph = ([Vertex],[Edge])
```

If (x, y, w) is an edge, then the edge is from vertex x to vertex y, and its weight is w.

The following function makes a list of edges into a proper symmetric graph, while also removing self loops and edges with non-positive weights.

```
mkproper :: [Edge] -> [Edge]
mkproper xs = let
  ys = List.filter
    (\ (x,y,w) -> x /= y && w > 0) xs
  zs = nubBy (\ (x,y,_) (x',y',_) ->
    (x,y) == (x',y') || (x,y) == (y',x')) ys
  in foldr
      (\ (x,y,w) us -> ((x,y,w):(y,x,w):us))
      [] zs
```

This gives, e.g.:

```
*GA> mkproper [(1,2,3),(2,3,4),(3,2,-5)]
[(1,2,3),(2,1,3),(2,3,4),(3,2,4)]
```

Getting the vertices from an edge list:

```
nodes :: (Eq a,Ord a) => [(a,a,b)] -> [a]
nodes = sort.nub.nds where
  nds = foldr (\ (x,y,_) -> ([x,y]++)) []
```

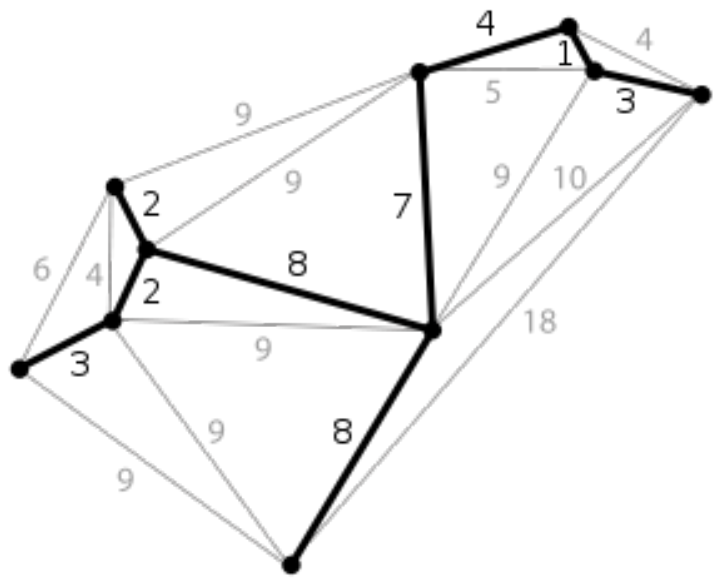
Making a graph from a list of edges:

```
mkGraph ::  [Edge] -> Graph
mkGraph es = let
   new = mkproper es
   vs  = nodes new
 in (vs,new)
```

Let G be a weighted, undirected (i.e., symmetric) and connected graph. Assume there are no self-loops. (Or, if there are self-loops, make sure their weight is set to 0.)

A *spanning tree* for weighted graph G is a tree with a node set that coincides with the vertex set of the graph, and an edge set that is a subset of the set of edges of the graph.

A *minimum spanning tree* for weighted graph G is a spanning tree of G whose edges sum to minimum weight. Caution: minimum spanning trees are not unique.



Applications: finding the least amount of wire necessary to connect a group of workstations (or homes, or cities, or . . .).

Prim's minimum spanning tree algorithm finds a minimum spanning tree for an arbitrary weighted symmetric and connected graph. See [30], [33, 4.7].

Prim's minimum spanning tree algorithm

1. Select an arbitrary graph node r to start the tree from.
2. While there are still nodes not in the tree
   (a) Select an *edge of minimum weight* between a tree and non-tree node.
   (b) Add the selected edge and vertex to the tree.

A more formal version of this uses variables V_{in} for the current vertex set of the tree, V_{out} for the current vertex set outside the tree, and E_{tree} for the current edge set of the tree.

Prim's minimal spanning tree algorithm (ii)

1. Let $G = (V, E)$ be given.
2. Select an arbitrary graph vertex $r \in V$ to start the tree from. $V_{\text{in}} := \{r\}$, $V_{\text{out}} := V - \{r\}$, $E_{\text{tree}} := \emptyset$.
3. While $V_{\text{out}} \neq \emptyset$ (there are still nodes not in the tree):
   (a) Select a member (x, y, w) from E with $x \in V_{\text{in}}$, $y \in V_{\text{out}}$, such that there is no $(x', y', w') \in E$ with $x' \in V_{\text{in}}$, $y' \in V_{\text{out}}$ and $w' < w$ (the edge (x, y, w) is an edge of minimum weight between a tree and non-tree node).
   (b) $V_{\text{in}} := V_{\text{in}} \cup \{y\}$, $V_{\text{out}} := V_{\text{out}} - \{y\}$, $E_{\text{tree}} := E_{\text{tree}} \cup \{(x, y, w)\}$.
4. Return E_{tree}.

Note that the assumption that the graph is connected ensures that the "select a member (x, y, w) from E ..." action succeeds.

It is not at first sight obvious that Prim's algorithm always results in a minimum spanning tree, but this fact can be checked by means of Hoare assertions, which can be tested.

Element in a non-empty list with minimal f-value:

```
minim :: (Ord b) => (a -> b) -> [a] -> a
minim f = head .
          (sortBy (\ x y -> compare (f x) (f y)))
```

Prim's Algorithm, initialisation:

```
prim :: [Edge] -> [Edge]
prim es = let
   (v:vs) = nodes es
   vout   = vs
   vin    = [v]
   tree   = []
 in prim' es vout vin tree
```

Prim's Algorithm, while loop:

```
prim' :: [Edge] -> [Vertex] -> [Vertex] -> [Edge] -> [Edge]
prim' es = while3
   (\ vout _ _ -> not (null vout))
   (\ vout vin tree -> let
        links = [(x,y,w) | (x,y,w) <- es,
                           elem x vin, elem y vout ]
        e@(x,y,w) = minim (\ (_, _, w) -> w) links
      in
        (vout\\[y],y:vin,e:tree))
```

How can we see that this is correct? Here is one way. The property to prove is:

If `es` is the edge list of a symmetric connected weighted graph G, then `prim es` gives the edges of a minimum spanning tree on G.

Let's look at subgraphs of G. If $G = (V, E)$, and $A \subseteq V$, then let G^A be the subgraph with node set A and edge set

$$\{(x, y, w) \mid (x, y, w) \in E, x \in A, y \in A\}.$$

We prove the following. At every stage of the algorithm,

`prim' es vout vin tree`

`tree` is a minimum spanning tree on the subgraph of G given by $G^{\texttt{vin}}$.

Induction on the size of `vin`. `prim'` is initialized with `vin = [v]` and `tree = []`. This is correct, for $(\{v\}, \emptyset)$ is a minimum spanning tree for $G^{\{v\}}$.

Induction step. Assume at stage `prim' es vout vin tree` the edges in `tree` form a minimum spanning tree for $G^{\texttt{vin}}$. Let $e = (x, y, w) \in \texttt{es}$ be a link with $x \in \texttt{vin}$, $y \in \texttt{vout}$, with minimum weight. Then the next stage of the algorithm is

`prim' es (vout\\[y]) (y:vin) (e:tree).`

We have to show that the edges in `e:tree` are a minimum spanning tree for $G^{(\texttt{y:vin})}$. Suppose they are not. Then, since `tree` is a minimum spanning tree for $G^{\texttt{vin}}$, there is some vertex x' inside `vin` and an edge $(x', y, w') \in E$ such that $w' < w$. Contradiction with the way in which the link (x, y, w) was selected.

Now that we have seen the proof, we can also turn it into an assertion or a loop invariant. Here is the specification for a minimal spanning subtree of a graph.

```
mst :: [Edge] -> [Edge] -> Bool
mst _ [] = True
mst g [e] = elem e g
mst g (e@(x,y,w):es) = let
    vs = nodes es
  in
     elem e g
     && mst g es
     && elem x vs
     && notElem y vs
     && forall g  (\ (x',y',w') ->
            (elem x' vs && notElem y' vs) ==> w <= w')
```

Use this for a self-testing version of `prim`:

```
primA :: [Edge] -> [Edge]
primA = assert1 mst prim
```

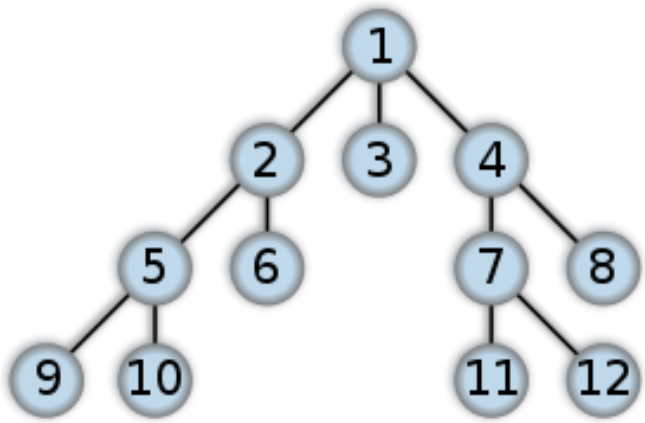
We can also use it to formulate a loop invariant:

```
prim'' :: [Edge] -> [Vertex] -> [Vertex] -> [Edge] -> [Edge]
prim'' es = while3
   (\ vout _ _ -> not (null vout))
   (invar3 (\ _ _ tree -> mst es tree)
   (\ vout vin tree -> let
        links = [(x,y,w) | (x,y,w) <- es,
                           elem x vin, elem y vout ]
        e@(x,y,w) = minim (\ (_, _, w) -> w) links
      in
        (vout\\[y],y:vin,e:tree)))
```

Exercise 4.7 *Let G be a symmetric, undirected weighted graph. Suppose all edges have different positive weights. Show that the minimum spanning tree of G is unique.*

4.5 Shortest Path Algorithms

For finding the shortest path between two vertices in an unweighted graph we can use breadth first search.



In this case the distance $d(x, y)$ between two vertices x and y in the graph is given as the length (number of edges) in the shortest path from x to y.

Breadth first search algorithm

1. Let G be given by its edge set E. Let s be some vertex of G.
2. $T := \{s\}$, $d(s) :=\uparrow$.
3. while $Q \neq []$ do:
   (a) remove the first vertex u from Q;
   (b) for every edge $u \to v \in E$ with $v \notin T$ do:
      i. $T := T \cup \{v\}$,
      ii. $d(v) := d(u) + 1$,
      iii. $Q := Q ++ [v]$.
4. Return the function d.

Update of a function for a single argument-value pair:

```
update :: Eq a => (a -> b) -> (a,b) -> a -> b
update f (y,z) x = if x == y then z else f x
```

Update of a function for a list of argument-value pairs:

```
updates :: Eq a => (a -> b) -> [(a,b)] -> a -> b
updates = foldl update
```

Initialisation of the breadth first search (the initial distance function gives everywhere ↑):

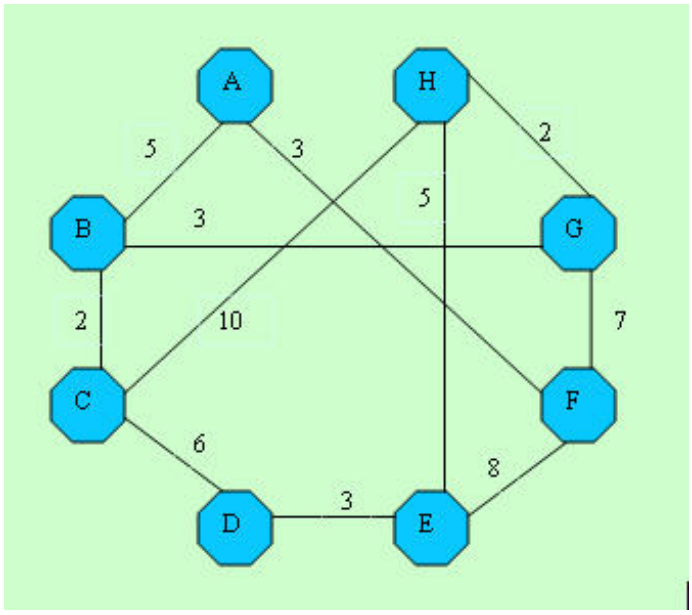
```
bfs :: Eq a => [(a,a)] -> a -> a -> Int
bfs g s = bfs' g [s] [s] (\_ -> undefined)
```

While loop of the breadth first search:

```
bfs' :: Eq a => [(a,a)] -> [a] -> [a]
             -> (a -> Int) -> a -> Int
bfs' g = while3 (\ q _ _ -> not (null q))
                (\ (u:q) t d -> let
                    new = [ y | (x,y) <- g,
                                x == u, notElem y t ]
                    q' = q ++ new
                    t' = t `union` new
                    pairs = [(v, d u + 1) | v <- new ]
                    d' = updates d pairs
                  in (q',t',d'))
```

Exercise 4.8 *Find a reasonable assertion for* `bfs`*, and use this to write an assertive version* `bfsA`*.*

In case the edges have positive weights, a modification of this algorithm works for finding shortest paths. This is called Dijkstra's algorithm [12].



First, here is an example weighted graph (same as the picture, but with the nodes A, B, ... renamed as 0, 1, ...

```
exampleG = mkproper
           [(0,1,5),(0,5,3),(1,2,2),(1,6,3),(2,3,6),(2,7,10),
            (3,4,3),(4,5,8),(5,6,7),(6,7,2)]
```

Dijkstra's algorithm

1. Let G be given by its edge set E. Let s be some vertex of G.
2. $T := \{s\}$, $P := \emptyset$, $d(s) := 0$.
3. while $T \neq \emptyset$ do:
   (a) choose a $v \in T$ for which $d(v)$ is minimal;
   (b) $T := T - \{v\}$;
   (c) $P := P \cup \{v\}$;
   (d) for every edge $(v, u, w) \in E$ do:
   if $u \in T$ then $d(u) := \min(d(u), d(v) + w)$
   else if $u \notin P$ then do:
      i. $d(u) := d(v) + w$;
      ii. $T := T \cup \{u\}$
4. Return the function d.

Note throughout the execution of the algorithm, the function d is defined for all members of T and P. In other words:

$$\{v \in V \mid d(v) \neq \uparrow\} = T \cup P$$

is an invariant of the algorithm.

Implementation of Dijkstra's algorithm: initialisation.

```
dijkstra :: [Edge] -> Vertex -> Vertex -> Float
dijkstra es s = let
    t = [s]
    p = []
    d = (\ x -> if x == s then 0 else undefined)
  in
    dijkstra' es t p d
```

Implementation of Dijkstra's algorithm: while loop.

```
dijkstra' :: [Edge] -> [Vertex] -> [Vertex]
          -> (Vertex -> Float) -> Vertex -> Float
dijkstra' es = while3
  (\ t _ _ -> not (null t))
  (\ t p d -> let
                v = minim (\ x -> d x) t
                pairs = [(u,w)| (x,u,w) <- es, x == v]
                old   = [(u, min (d u) (d v + w)) |
                         (u,w) <- pairs, elem u t ]
                new   = [(u, d v + w) | (u,w) <- pairs,
                          notElem u t, notElem u p ]
                d'    = updates d (old ++ new)
                t'    = (t \\ [v]) `union` (map fst new)
              in (t',v:p,d'))
```

In Haskell, we cannot test for definedness without the risk of generating errors, because undefined is implemented as error.

Instead we can assert that a distance function is defined on a given list of inputs by checking that it computes a value ≥ 0, as follows:

```
definedOn :: (Num b,Ord b) => [a] -> (a -> b) -> Bool
definedOn xs f = and (map (triv.f) xs)
  where triv = (\ x -> x >= 0)
```

An invariant of `dijkstra'` is that its d function is defined on its parameters T and P:

```
dijkstraA :: [Edge] -> Vertex -> Vertex -> Float
dijkstraA es s = let
    t = [s]
    p = []
    d = (\ x -> if x == s then 0 else undefined)
  in
    dijkstraA' es t p d
```

```
dijkstraA' :: [Edge] -> [Vertex] -> [Vertex]
          -> (Vertex -> Float) -> Vertex -> Float
dijkstraA' es = while3
  (\ t _ _ -> not (null t))
  (invar3 (\ t p d -> definedOn (t `union` p) d)
  (\ t p d -> let
                v = minim (\ x -> d x) t
                pairs = [(u,w)| (x,u,w) <- es, x == v]
                old   = [(u, min (d u) (d v + w)) |
                         (u,w) <- pairs, elem u t ]
                new   = [(u, d v + w) | (u,w) <- pairs,
                           notElem u t, notElem u p ]
                d'    = updates d (old ++ new)
                t'    = (t \\ [v]) `union` (map fst new)
              in (t',v:p,d')))
```

Another algorithm for shortest path is the Floyd-Warshall algorithm, which computes a distance table for the whole graph. Assume the graph is given as a node set N plus a distance function f that gives the distance between x and y if there is a direct link in the graph from x to y, and 0 otherwise.

The Floyd-Warshall algorithm uses a different representation from Dijkstra's algorithm, so the following conversion function is useful:

```
edges2fct :: [(Int, Int, Float)] -> Int -> Int -> Float
edges2fct es i j = let
   links = filter (\ (x,y,w) -> x == i && y == j) es
   ws = map (\ (_,_,w) -> w) links
  in
   if null ws then 0 else head ws
```

And conversely:

```
fct2edges :: Int -> (Int->Int->Float) -> [(Int, Int, Float)]
fct2edges n f =
  [(x,y,f x y) | x <- [0..n-1], y <- [0..n-1],
                 f x y > 0 ]
```

Let $\{0, \ldots, N-1\}$ be the nodes in the graph. The algorithm computes shortest distances in subgraphs $\{0, \ldots, k-1\}$ for $k \in \{0, \ldots, N-1\}$. The nodes in $\{0, \ldots, k-1\}$ are the nodes that are available for stopovers.

The output of the algorithm is a function $d : N^3 \to \mathbb{R}$, where $d(k, i, j)$ gives the shortest distance in G between i and j when all stopover nodes are $< k$.

Initially, no nodes are available for stopovers, so we have $d(0, i, j) = f(i, j)$.

Here is the algorithm in pseudo-code:

Floyd-Warshall algorithm for shortest path

1. Let $N = \{0, \ldots, N-1\}$ be the nodes of a graph G. Let f be the distance function for G.
2. For each $(i, j) \in N^2$, let $d(0, i, j) :=$ if $f(i, j) \neq 0$ then $f(i, j)$ else ∞.
3. For each $k \in N$, all $(i, j) \in N^2$,
   let $d(k+1, i, j) := \min(d(k, i, j), d(k, i, k) + d(k, k, j))$.

This construction, by the way, is the cornerstone of Kleene's famous theorem stating that languages generated by automata are regular [24]. In fact, Floyd's and Warshaw's descriptions of the algorithm were provided later (in [18] and [39], respectively).

Here is the Haskell version. First a definition of ∞:

```
infty :: Float
infty = 1/0
```

Initialisation of the distance table:

```
fw :: Int -> (Int -> Int -> Float) -> Int -> Int -> Float
fw n f = let
    init = \ i j ->
             if f i j > 0 then f i j else infty
 in
    fw' n init
```

For the computation of the distance table we will use a variation on `update`:

```
update2 :: (Eq a,Eq b) => (a -> b -> c) -> (a,b,c) -> a -> b -> c
update2 f (u,v,w) x y = if u == x && v == y then w else f x y
```

Computation of the distance table, by means of successive updates of the initial distance table:

```
fw' ::  Int -> (Int -> Int -> Float)
            -> Int -> Int -> Float
fw' n = let
    nodes = [0..n-1]
    pairs = [(i,j) | i <- nodes, j <- nodes]
  in
    for pairs (\ (i,j) ->
       for nodes (\ k d -> let
          k' = k-1
        in
          update2 d (i,j,min (d i j) (d i k' + d k' j))))
```

We can use Dijkstra's algorithm to write an assertive version of the Floyd-Warshall algorithm:

```
isShortest :: Int -> (Int -> Int -> Float)
           -> Int -> Int -> Float -> Bool
isShortest n f i j w = let
    g = fct2edges n f
  in
    w == dijkstra g i j
```

Assertive version of Floyd-Warshall:

```
fwA = assert4 isShortest fw
```

Try this out on an example graph function:

```
exampleF :: Int -> Int -> Float
exampleF = edges2fct exampleG
```

```
*GA> fwA 8 exampleF 0 4
11.0
```

If one also wants to find the actual shortest path, a simple modification of the Floyd-Warshall algorithm suffices. An additional function next : $N^2 \to N$ is constructed, with next(i, j) equal to -1, to indicate that no intermediate point has been found on a shortest path yet.

Floyd-Warshall algorithm with path reconstruction

1. Let $N = \{0, \ldots, N-1\}$ be the nodes of a graph G. Let f be the distance function for G.
2. For all $(i, j) \in N^2$, let $d(0, i, j) :=$ if $f(i, j) \neq 0$ then $f(i, j)$ else ∞.
3. For all $(i, j) \in N^2$, let next$(i, j) := -1$.
4. For all $k \in N$, all $(i, j) \in N^2$,
   let $d(k+1, i, j) := \min(d(k, i, j), d(k, i, k) + d(k, k, j))$;
   if $d(k, i, k) + d(k, k, j)) < d(k, i, j)$ then next$(i, j) := k$.

The path can now be reconstructed from the two functions d and next, as follows:

GetPath(i,j)

1. if $d(i, j) = \infty$ then return "no path".
2. $k :=$ next(i, j).
3. if $k = -1$ then return []
   else return GetPath(i, k) $++[k]++$GetPath(k, j).

Implementation of the modified Floyd-Warshall algorithm: the modified algorithm returns a function that computes a pair (w, k) consisting of a distance w and an intermediate node k. Initialisation of the distance-next table:

```
mfw :: Int -> (Int -> Int -> Float) ->
       Int -> Int -> (Float, Int)
mfw n f = let
  init = \ i j ->
            (if f i j > 0 then f i j else infty, -1)
 in
    mfw' n init
```

Computation of the distance-next table:

```
mfw' ::  Int -> (Int -> Int -> (Float, Int))
              -> Int -> Int -> (Float, Int)
mfw' n = let
    nodes = [0..n-1]
    pairs = [(i,j) | i <- nodes, j <- nodes]
  in
    for pairs (\ (i,j) ->
       for nodes (\ k d -> let
           k' = k-1
           f  = \ x y -> fst (d x y)
         in
           if f i k' + f k' j < f i j then
             update2 d (i,j,(f i k' + f k' j, k'))
           else d))
```

Reconstructing the path:

```
getPath :: (Int -> Int -> (Float, Int))
        -> Int -> Int -> [Int]
getPath d i j = let
    dist = fst (d i j)
    k    = snd (d i j)
  in
    if dist == infty then error "no path"
    else if k == -1 then []
    else getPath d i k ++ [k] ++ getPath d k j
```

This gives:

```
*GA> getPath (mfw 8 exampleF) 0 1
[]
*GA> getPath (mfw 8 exampleF) 0 2
[1]
*GA> getPath (mfw 8 exampleF) 0 3
[1,2]
*GA> getPath (mfw 8 exampleF) 0 4
[5]
*GA> getPath (mfw 8 exampleF) 0 5
[]
*GA> getPath (mfw 8 exampleF) 0 6
[1]
```

```
*GA> getPath (mfw 8 exampleF) 0 7
[1,6]
```

Warshall's original version (see [39]) computed transitive closure for directed graphs without weights. First in pseudo-code:

Warshall's algorithm for transitive closure

1. Let $N = \{0, \ldots, N-1\}$ be the nodes of a graph G.
   Let $r : N^2 \to \{T, F\}$ give its edge relation.
2. For all $(i, j) \in N^2$, let $t(0, i, j) := r(i, j)$, $t(k, i, j) :=\uparrow$ if $k > 0$.
3. For all $k \in N$, all $(i, j) \in N^2$,
   let $t(k+1, i, j) := t(k, i, j) \vee (t(k, i, k) \wedge t(k, k, j))$.

Initialisation of the t function is not necessary, for it is given by the graph function itself.

```
warshall :: Int -> (Int->Int->Bool) -> Int -> Int -> Bool
warshall n = let
    nodes = [0..n-1]
    pairs = [(i,j) | i <- nodes, j <- nodes]
  in
    for pairs (\ (i,j) ->
       for nodes (\ k t -> let
         k' = k-1
       in
         update2 t (i,j, t i j || (t i k' && t k' j))))
```

We can use our earlier definition of transitive closure to turn this into assertive code:

```
isTC :: Int -> (Int->Int->Bool) -> (Int->Int->Bool) -> Bool
isTC n r t = let
    r' = [(i,j)| i <- [0..n-1], j <- [0..n-1], r i j ]
    t' = [(i,j)| i <- [0..n-1], j <- [0..n-1], t i j ]
  in
    equalS t' (tc r')
```

```
warshallA :: Int -> (Int->Int->Bool) -> Int -> Int -> Bool
warshallA = assert2 isTC warshall
```

It is not always possible to use one algorithm to check another in this simple way. The Bellman-Ford algorithm for shortest path [4] serves the same purpose as Dijkstra's algorithm (computing shortest paths from a given vertex in a weighted directed graph), but for a wider class of graphs: unlike in the case of Dijkstra's algorithm, weights are allowed to be negative.

To ensure that the notion of shortest path still makes sense in this context, it is assumed that the graph contains no *negative cycles*. A negative cycle in a weighted directed graph is a path

$$v_1 \xrightarrow{w_1} v_2 \xrightarrow{w_2} v_3 \rightarrow \cdots \rightarrow v_1$$

with the property that the weights on the path from v_1 to v_1 sum to a negative number. In the presence of negative cycles, the notion of shortest path loses its sense, for one could always make a path shorter by taking one more turn through a negative loop.

Here is the Belmann-Ford algorithm for solving this problem:

Belmann-Ford algorithm for shortest path in the presence of negative weights

1. Let G be given as (V, E), with $E \subseteq V^2$. Let s be the source.
2. For each $v \in V$ do:
   (a) if $v = s$ then $d(v) := 0$ else $d(v) := \infty$;
   (b) if $(v, s) \in E$ then $p(v) := s$ else $p(v) :=\uparrow$.
3. For i from 1 to $|V| - 1$ do:
   for each $u \xrightarrow{w} v$ in E do: if $d(u) + w < d(v)$ then:
   (a) $d(v) := d(u) + w$;
   (b) $p(v) := u$.
4. For each $u \xrightarrow{w} v$ in E:
   if $d(u) + w < d(v)$ then error "Graph contains a negative-weight cycle".

Implementation of the initialisation of the two functions:

```
bfInit :: [Edge] -> Vertex -> (Vertex -> Float,Vertex -> Vertex)
bfInit es s = let
    pairs = [ (u,w) | (x,u,w) <- es, x == s ]
    d  = updates (\ _ -> infty) pairs
    p  = \ x -> if elem x (map fst pairs) then s else undefined
  in
    (d,p)
```

Implementation of the main "for" loop:

```
bfLoop ::  [Edge] -> (Vertex -> Float,Vertex -> Vertex)
              -> (Vertex -> Float,Vertex -> Vertex)
bfLoop es = let
    ns = nodes es
    is = [1..length(ns) - 1]
  in
    for is (\ _ (d,p) -> let
          us    = filter (\ (u,v,w) -> d(u) + w < d(v)) es
          pairs = map (\ (u,v,w) -> (v,d(u) + w)) us
          vs    = map (\ (u,v,_) -> (v,u)) us
       in
          (updates d pairs, updates p vs))
```

The whole program:

```
bf :: [Edge] -> Vertex -> (Vertex -> Float,Vertex -> Vertex)
bf es s = bfLoop es (bfInit es s)
```

We can use the predecessor function to reconstruct the shortest path, by tracing back from the destination to the source, using the predecessor function:

```
traceBack :: (Vertex -> Vertex)
          -> Vertex ->  Vertex -> [Vertex]
traceBack p s t =
   if s == t then [s]
   else t : traceBack p s (p t)
```

This gives, e.g.:

```
*GA> traceBack (snd (bf exampleG 0 )) 0 7
[7,6,1,0]
```

Implementation of the check for negative cycles:

```
bfCheck :: [Edge] -> (Vertex -> Float) -> Bool
bfCheck es d =
  forall es (\ (u,v,w) -> d u + w >= d v)
```

Exercise 4.9 *Can you give a proof that the check for negative cycles at the end of the Belmann-Ford algorithm is actually correct?*

We can use the check for negative cycles — once we have convinced ourselves that it is correct — as an assertion about the computed distance function:

```
bfLoopA ::  [Edge] -> (Vertex -> Float,Vertex -> Vertex)
              -> (Vertex -> Float,Vertex -> Vertex)
bfLoopA = assert2 (\ es _ (d,_) -> bfCheck es d) bfLoop
```

The whole program again, with the assertion added:

```
bfA :: [Edge] -> Vertex -> (Vertex -> Float,Vertex -> Vertex)
bfA es s = bfLoopA es (bfInit es s)
```

Exercise 4.10 *Find other suitable assertions to wrap around* `bfLoop`.

Exercise 4.11 *Look up Yen's improvement of the Bellman-Ford algorithm, in [40] or on Wikipedia. Implement it.*

4.6 Summary

If we have several graph algorithms that perform similar computations (connectedness, transitive closure, shortest path), then we can use one algorithm as a specification for the other, and vice versa.

Assertive code provides documentation and tests at the same time. It describes what algorithms are supposed to compute, but it has to do so in terms of other algorithms. If an assertive function raises an error both the implementation of the function and the implementation of the asserted specification can be at fault.

Chapter 5

Algorithms for Matching and Assignment

Abstract This chapter deals with algorithms for matching and assignment. The matching problem is the problem of connecting nodes in a bipartite graph, while making sure that certain requirements are met. An assignment problem is a problem of assigning tasks to agents. Any agent can be assigned to perform any task, incurring some cost that may vary depending on the agent-task assignment. The requirement is to perform all tasks by assigning distribing the tasks over the agents in such a way that the total cost of the assignment is minimized.

Key words: Stable marriage problem, college admission problem, assignment problem, Hungarian algorithm.

5.1 Introduction

Declaration of a Haskell module.

```
module AMA

where

import List
import While
import Assert
```

5.2 The Stable Marriage Algorithm

Suppose equal sets of men and women are given, each man has listed the women in order of preference, and vice versa for the women. A stable marriage match between men and women is a one-to-one mapping

between the men and women with the property that if a man prefers another woman over his own wife then that woman does *not* prefer him to her own husband, and if a woman prefers another man over her own husband, then that man does *not* prefer her to his own wife.

The computer scientists Gale and Shapley proved that stable matchings always exist, and gave an algorithm for finding such matchings, the so-called Gale-Shapley algorithm [19]. This has many important applications, also outside of the area of marriage counseling. See `http://en.wikipedia.org/wiki/Stable_marriage_problem` for more information.

An imperative style pseudo-code version of the Gale-Shapley algorithm is given here:

Gale-Shapley algorithm for stable marriage

1. Initialize all m in M and w in W to free;
2. while there is a free man m who still has a woman w to propose to
   (a) w is the highest ranked woman to whom m has not yet proposed
   (b) if w is free, (w, m) become engaged
   else (some pair (w, m') already exists)
   if w prefers m to m'
      i. (w, m) become engaged
      ii. m' becomes free
   else (w, m') remain engaged

Here is a slightly more formal version using named variables:

Gale-Shapley algorithm for stable marriage (ii)

1. Let equal-sized sets M and W of men and women be given.
   Let pr be a preference function.
2. $F := M$, $E := \emptyset$.
3. While $F \neq \emptyset$ do
   (a) take $m \in F$;
   (b) $w :=$ the highest ranked woman to whom m has not yet proposed;
   (c) delete w from the preference list of m;
   (d) if $\neg\exists m' : (w, m') \in E$ then
      i. $E := E \cup \{(w, m)\}$;
      ii. $F := F - \{m\}$;
   else $(\exists m' : (w, m') \in E)$ if $\text{pr}_w mm'$ then do
      i. $E := (E - \{(w, m'\}) \cup \{(w, m)\}$;
      ii. $F := (F - \{m\}) \cup \{m'\}$;

If we want to write this in functional imperative style, we have to check the number of parameters for the while loop. These are:

- the preference list of the men
- the list of current engagements
- the list of current free men.

It may look like we also need the preference list of the women as a parameter, but this list is not used for stating the exit condition and it is not changed by the step function, and we can keep it outside the loop. So it is possible to phrase the algorithm in terms of *while3*.

First some type declarations, for readability of the code:

```
type Man   = Int
type Woman = Int
type Mpref = [(Man,[Woman])]
type Wpref = [(Woman,[Man])]
type Engaged = [(Woman,Man)]
```

Here is an auxiliary function for converting a preference list to a function, which allows us to express *w prefers m to m'* in a simple way.

```
type PrefFct = Int -> Int -> Int -> Bool

plist2pfct :: [(Int,[Int])] -> PrefFct
plist2pfct table x y y' =
  let
    Just prefs = lookup x table
  in elem y (takeWhile (/= y') prefs)
```

Initialisation of the loop: the list of all men is extracted from the table of men's preferences, all men are free, no-one is engaged to start with. Note that `map fst mpref` gives us the list of all men.

```
stableMatch :: Wpref -> Mpref -> Engaged
stableMatch wpref mpref =
  let
     men     = map fst mpref
     free    = men
     engaged = []
  in
  stable wpref mpref free engaged
```

The test function for the while loop just checks whether the list of free men is exhausted.

The step function for the while loop has an argument of type `Mpref` for the current list of men's preferences, `Engaged` for the list of currently engaged (w, m) pairs, and an argument of type `[Man]` for the list of currently free (not engaged) men. The list of men's preferences changes in the loop step, for each woman that a man proposes to is crossed from his preference list.

```
stable ::  Wpref -> Mpref -> [Man] -> Engaged -> Engaged
stable wpref = let
   wpr = plist2pfct wpref
 in while3 (\ _ free _ -> not (null free))
           (\ mpr (m:free) engaged  ->
            let
               Just (w:ws) = lookup m mpr
               match = lookup w engaged
               mpr' = (m,ws) : (delete (m,w:ws) mpr)
               (engaged',free') = case match of
                   Nothing -> ((w,m):engaged,free)
                   Just m' ->
                     if wpr w m m' then (
                         (w,m) : (delete (w,m') engaged),
                          m':free)
                     else (engaged, m:free)
            in (mpr',free',engaged'))
```

The algorithm assumes that there are equal numbers of men and women to start with, and that both men and women have listed all members of the opposite sex in order of preference.

Here are some example tables of preferences:

```
mt :: Mpref
mt = [(1,[2,1,3]), (2, [3,2,1]), (3,[1,3,2])]

wt :: Wpref
wt = [(1,[1,2,3]),(2,[3,2,1]), (3,[1,3,2])]
```

In `mt`, the entry `(1,[2,1,3])` indicates that man 1 prefers woman 2 over woman 1 and woman 1 over woman 3. We assume that preferences are transitive, so man 1 also prefers woman 2 over woman 3.

A simple example, to demonstrate how the function `stableMatch` is used:

```
makeMatch = stableMatch mt wt
```

This gives:

```
*AS> makeMatch
[(2,2),(3,3),(1,1)]
```

This gives (w, m) pairs. So woman 1 is married to man 1, and so on. Note that the first woman ends up with the man of her preference, but the other two women do not. But this match is still stable, for although the second woman is willing to swap her husband for the third man, she is at the bottom of *his* list. And so on.

To show that this algorithm is correct, we have to show two things: termination and stability of the constructed match.

Exercise 5.1 *Show that if there are equal numbers of men and women, the algorithm always terminates. Hint: analyze what happens to the preference lists of the men. Observe that no man proposes to the same woman twice.*

To show stability, we have to show that each step through the step function preserves the invariant "the set of currently engaged men and women is stable."

What does it mean for E to be stable on W and M? Let us use $\text{pr}_w mm'$ for w prefers m over m'.

- $\forall(w, m) \in E \forall(w', m') \in E$: if $\text{pr}_m w'w$ then $\text{pr}_{w'} m'm$;
- $\forall(w, m) \in E \forall(w', m') \in E$: if $\text{pr}_w mm$ then $\text{pr}_{m'} w'w$.

What is the requirement on `free`?

- free equals the set of all men minus the men that are engaged.

We see that these requirements hold for the first call to `stable`, for in that call `engaged` is set to [] and `free` is set to the list of all men. The empty list of engaged pairs is stable by definition.

Next, inspect what happens in the step function for `stable`. The precondition for the step to be performed is that there is at least one free man m left. m proposes to the woman w who is on the top of his list. If w is free, w accepts the proposal, and they become engaged. Is the new list of engaged pairs stable? We only have to check for the new pair (w, m).

- Suppose that there is a free w' with $\text{pr}_m w'w$. This cannot be, for w is at the top of m's list.
- Suppose there is m' with $\text{pr}_w m'm$. Then if m' is engaged, this must mean that not $\text{pr}_{m'} ww'$, where w' is the fiancee of m'. For otherwise m' would have proposed to w instead of to w'.
- The new list of free men equals the old list, minus m. This is correct, for m just got engaged.

Now the other case: w is already engaged. There are two subcases. In case w prefers her own current fiancee, nothing happens. The resulting list of engaged pairs is still stable. The list of free men remains the same, for m proposed and got rejected.

In case w prefers m to her own fiancee m', she swaps: (w, m) replaces (w, m') in the list of engaged pairs. Again, it is easy to see that the resulting list of engaged pairs is stable. m gets replaced by m' on the list of free men.

So the two stability requirements are satisfied.

The requirement that in any call to `stable` its last argument contains the list of currently free men is also satisfied, for m gets replaced by m' on the list of free men.

Exercise 5.2 *Someone proposed to test the algorithm by interchanging the preference tables of the men and women, and checking whether the result would be the same match (but with men and women reversed). Good idea or not? Why?*

Exercise 5.3 *Is the Gale-Shapley algorithm more favourable for the men, or for the women? Or doesn't it matter? Motivate your answer.*

5.3 An Assertive Version of the Stable Marriage Algorithm

An alternative to reasoning about the correctness of an algorithm is specification-based testing, which we will explore now.

One of the properties that has to be maintained through the loop step function for stable marriage is: "Each man is either free or engaged, but never both." This is implemented as follows:

```
freeProp :: Mpref -> [Man] -> Engaged -> Bool
freeProp mpref free engaged = let
    men  = map fst mpref
    emen = map snd engaged
  in forall men (\x -> elem x free == notElem x emen)
```

The other invariant is the stability property. Here is the definition of stability for a relation E consisting of engaged (w, m) pairs:

$$\forall (w, m) \in E \forall (w', m') \in E \qquad ((\mathrm{pr}_w m' m \rightarrow \mathrm{pr}_{m'} w' w) \wedge (\mathrm{pr}_m w' w \rightarrow \mathrm{pr}_{w'} m' m)).$$

What this says (once more) is: if w prefers another guy m' to her own fiancee m then m' does prefer his own fiancee w' to w, and if m prefers another woman w' to his own fiancee w then w' does prefer her own fiancee m' to m.

Once it is written like this it is straightforward to implement it, for we have all the ingredients:

```
isStable :: Wpref -> Mpref -> Engaged -> Bool
isStable wpref mpref engaged = let
    wf = plist2pfct wpref
    mf = plist2pfct mpref
  in
    forall engaged (\ (w,m) -> forall engaged
          (\ (w',m') -> (wf w m' m ==> mf m' w' w)
                         &&
                        (mf m w' w ==> wf w' m' m)))
```

This property can be used as a test on the output of `stableMatch`, as follows:

```
stableMatch' :: Wpref -> Mpref -> Engaged
stableMatch' = assert2 isStable stableMatch
```

Another possibility is to use the assertion as an invariant. This version has to check stability for a new pair:

```
stablePair :: Wpref -> Mpref
           -> (Woman,Man) -> Engaged -> Bool
stablePair wpref mpref (w,m) engaged = let
    wf = plist2pfct wpref
    mf = plist2pfct mpref
  in
    forall engaged
          (\ (w',m') -> (wf w m' m ==> mf m' w' w)
                         &&
                        (mf m w' w ==> wf w' m' m))
```

We need to distinguish now between the full preference list of the men and the current preference list. We use the current preference list to pick the current most preferred woman of the first free man. This excludes the women he has been rejected by, or who have swapped him for a better match in the preceding steps of the algorithm. Here is a check that in case the current most preferred woman of the first free man is still free, the result of their engagement does not spoil stability:

```
stillStable :: Wpref -> Mpref
            -> Mpref -> [Man] -> Engaged -> Bool
stillStable wpr mpr currentmpr [] eng = True
stillStable wpr mpr currentmpr (m:free) eng = let
   Just (w:ws) = lookup m currentmpr
   match = lookup w eng
 in
   match == Nothing ==> stablePair wpr mpr (w,m) eng
```

Finally, we write out a version of the code including the two invariants:

```
stableMatchA :: Wpref -> Mpref -> Engaged
stableMatchA wpref mpref =
  let
     men     = map fst mpref
     free    = men
     engaged = []
  in
  stableA wpref mpref mpref free engaged
```

We have one more parameter, for the preference function for the men. This is needed to implement the stability check correctly. The code inside the `while3` loop does not change.

```
stableA ::  Wpref -> Mpref ->
            Mpref -> [Man] -> Engaged -> Engaged
stableA wpref mpref = let
    wf     = plist2pfct wpref
 in while3 (\ _ free _ -> not (null free))
  (invar3 freeProp
  (invar3 (stillStable wpref mpref)
  (\ mpr (m:free) engaged  ->
      let
        Just (w:ws) = lookup m mpr
        match = lookup w engaged
        mpr' = (m,ws) : (delete (m,w:ws) mpr)
        (engaged',free') = case match of
          Nothing -> ((w,m):engaged,free)
          Just m' ->
            if wf w m m' then (
                  (w,m) : (delete (w,m') engaged),
                   m':free)
            else (engaged, m:free)
      in (mpr',free',engaged'))))
```

Note that we have stacked the two invariant tests, by wrapping one invariant around the wrap of the other invariant around the step function.

An example preference list for the men:

```
mt2 = [(1, [1, 5, 3, 9, 10, 4, 6, 2, 8, 7]),
       (2, [3, 8, 1, 4, 5, 6, 2, 10, 9, 7]),
       (3, [8, 5, 1, 4, 2, 6, 9, 7, 3, 10]),
       (4, [9, 6, 4, 7, 8, 5, 10, 2, 3, 1]),
       (5, [10, 4, 2, 3, 6, 5, 1, 9, 8, 7]),
       (6, [2, 1, 4, 7, 5, 9, 3, 10, 8, 6]),
       (7, [7, 5, 9, 2, 3, 1, 4, 8, 10, 6]),
       (8, [1, 5, 8, 6, 9, 3, 10, 2, 7, 4]),
       (9, [8, 3, 4, 7, 2, 1, 6, 9, 10, 5]),
       (10, [1, 6, 10, 7, 5, 2, 4, 3, 9, 8])]
```

And for the women:

```
wt2 =[(1, [2, 6, 10, 7, 9, 1, 4, 5, 3, 8]),
      (2, [2, 1, 3, 6, 7, 4, 9, 5, 10, 8]),
      (3, [6, 2, 5, 7, 8, 3, 9, 1, 4, 10]),
      (4, [6, 10, 3, 1, 9, 8, 7, 4, 2, 5]),
      (5, [10, 8, 6, 4, 1, 7, 3, 5, 9, 2]),
      (6, [2, 1, 5, 9, 10, 4, 6, 7, 3, 8]),
      (7, [10, 7, 8, 6, 2, 1, 3, 5, 4, 9]),
      (8, [7, 10, 2, 1, 9, 4, 8, 5, 3, 6]),
      (9, [9, 3, 8, 7, 6, 2, 1, 5, 10, 4]),
      (10, [5, 8, 7, 1, 2, 10, 3, 9, 6, 4])]
```

And a test run with this:

```
makeMatch2 = stableMatchA mt2 wt2
```

This gives:

```
*AS> makeMatch2
[(4,6),(5,10),(1,9),(9,8),(7,7),(8,5),(10,1),(6,2),(3,4),(2,3)]
```

Exercise 5.4 *The task in the* stable roommate problem *is like that in the stable marriage problem: find a stable matching of* $2n$ *students. Unlike the stable marriage case, the set of students is not broken up into male and female. Any person can prefer anyone in the same set.*

Look up the algorithm given for this in [23] or on wikipedia (`http://en.wikipedia.org/wiki/Stable_roommates_problem`*), implement it, and next develop an assertive version for it.*

5.4 The Weighted Matching Problem

The weighted matching problem seeks to find a matching in a weighted bipartite graph that has maximum weight. Maximum weighted matchings do not have to be stable, but in some applications a maximum weighted matching is better than a stable one.

5.5 College or Hospital Admission

In the college/hospital admission problem, several students or doctors can propose to the same college or hospital.

5.6 The Task Assignment Problem

5.7 Summary

...

Chapter 6

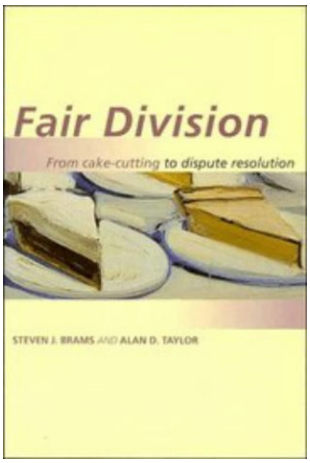
Fair Division Algorithms

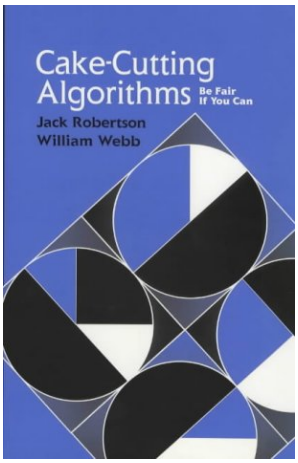
Abstract This chapter discusses fair division algorithms, develops testable specifications for them, and uses the specifications to write self-testing versions of the algorithms.

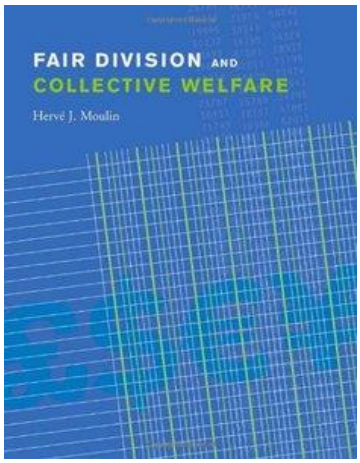
Key words: Functional algorithm specification, algorithm verification, specification based testing, test automation, self-testing code, fair division algorithms, cake cutting algorithms, estate division, chore division.

6.1 Introduction

This chapter discusses fair division algorithms, develops testable specifications for them, and uses the specifications to write self-testing versions of the algorithms. For background on fair division, see [7, 8, 6, 27, 32].







The implementations use literate Haskell.

```
module FDA

where

import List
import While
import Assert
```

The modules `While` and `Assert` give the code for while loops and for assertion and invariant wrappers that was developed and discussed in the chapter on Algorithm Specification (Chapter 3).

6.2 A Tale From India

Two farmers, Ram and Shyam were eating chapatis. Ram had 3 pieces of the flat, round bread and Shyam had 5. A traveller who looked hungry and tired rode up to the two men. Ram and Shyam decided to share their chapatis with him. The 3 men stacked the 8 chapatis (like pancakes) and cut the stack into 3 equal parts. They shared the pieces equally and ate until nothing was left. The traveller, who was a nobleman, was so grateful that he gave the two farmers 8 gold coins for his share of the food.

After the traveller left, Ram and Shyam wondered how they should share the 8 gold coins. Ram said that there were 8 coins and only 2 people, so each person should get an equal share of 4 coins. "But thats not fair," said Shyam, "since I had 5 chapatis to begin with." Ram could see his point, but he didnt really want to give 5 of the coins to Shyam. So he suggested they go see Maulvi, who was very wise. Shyam agreed.

Ram and Shyam told the whole story to Maulvi. After thinking for a long time, he said that the fairest way to share the coins was to give Shyam 7 coins and Ram only 1 coin. Both men were surprised. But when they asked Maulvi to explain his reasoning, they were satisfied that it was a fair division of the 8 coins.

T.V. Padma, *Mathematwist: Number Tales from Around the World* [29]

Exercise 6.1 *Use this example to analyze the notion of fairness. Give three definitions of fairness, to match the three divisions mentioned in the tale. Argue that according to your first definition, the division 'each an equal share of 4 coins' is fair, that according to the second definition, '5 coins for Shyam and 3 for Ram' is fair, and that according to the third definition, the solution they finally arrived at is fair. Draw a moral from this.*

One could easily conclude from this tale (and from Exercise 6.1) that fairness is too subjective to allow for formal analysis. We will demonstrate in this chapter that that would be a mistake.

6.3 Cake Cutting Algorithms

Problems of fair division can often be represented as cake cutting problems. We will assume the participants in a cake cutting process are unashamedly selfish. Each wants the largest piece he can get away with. Here is a relevant quote from [32]:

> We encourage the reader to envision yourself in a life-or-death struggle with siblings to see that you get a "fair share" of a literal leftover piece of your favorite cake. Don't give up a crumb!

We will assume that the cake is non-heterogeneous: some parts may be valued differently by the different participants. Without this assumption cake-cutting is easy, of course. Just give each of N contestants $\frac{1}{N}$ of the cake, with $\frac{1}{N}$ calculated according to the valuation they all agree on.

In case a pie is to be shared between two people, we can use the oldest social procedure of the world. *Cut and choose* (also known as 'I cut, you choose') is a procedure for two-person fair division of some desirable or undesirable heterogeneous good.

Indeed, let X be a set representing the good to be divided. A valuation function V for X is a function from $\mathcal{P}(X)$ to $[0, 1]$ with the properties that $V(\emptyset) = 0$, $V(X) = 1$, and $A \subseteq B \subseteq X$ implies $V(A) \leq V(B)$. If two valuation functions V_m and V_y (for my valuation and your valuation of X) are different, this means that you and I value some items in X differently.

Cake cutting for two: "I Cut, You Choose"

1. Let X be given, together with two valuation functions $f_1, f_2 : \mathcal{P}(X) \to [0..1]$.
2. Person 1 uses f_1 to divide X into X_1 and X_2 with $f_1(X_1) = f_1(X_2)$.
3. If $f_2(X_1) \geq f_2(X_2)$, person 2 gets X_1, otherwise person 2 gets X_2.

If the two participants have different value judgements on parts of the goods, it is possible that both participants feel they received more than 50 percent of the goods. It follows, as was already observed by Hugo Steinhaus in 1948, that there exists a division that gives both parties more than their due part; "this fact disproves the common opinion that differences in estimation make fair division difficult"[36].

It matters whether the valuations are known to the other party. Such knowledge can be used to advantage by the one who cuts. First consider the case that your valuation is unknown to me, and vice versa. Then if I cut, the best I can do is follow the algorithm above: pick sets $A, B \subseteq X$ with $A \cap B = \emptyset$, $A \cup B = X$, and $V_m(A) = V_m(B)$. If you choose, you will use V_y to pick the maximum of $\{V_y(A), V_y(B)\}$. It follows immediately that cutting guarantees a fair share, but no more than that, while choosing holds a promise for a better deal. So if you ever get the choice between cutting and choosing in a situation where both parties only know their own valuation, then it is to your advantage to leave the cutting to the other person.

Below, to keep matters simple, we will assume that valuations are private (not known to the other participants). Here is a specification for the "Moving Knife" cake cutting algorithm for N participants, with $N \geq 2$ (described in [15]).

'Moving knife' algorithm for cake cutting

A knife is slowly moved at constant speed parallel to itself over the top of the cake. At each instant the knife is poised so that it could cut a unique slice of the cake. As time goes by the potential slice increases monotonely from nothing until it becomes the entire cake. The first person to indicate satisfaction with the slice then determined by the position of the knife receives that slice and is eliminated from further distribution of the cake. (If two or more participants simultaneously indicate satisfaction with the slice, it is given to ny one of them.) The process is repeated with the other $N-1$ participants and with what remains of the cake [15].

To get a bit closer to implementation, assume that X is the line segment from 0 to 1. A valuation function for this is a function f that assigns values to sub-intervals $[a_i \ldots b_i]$ with $0 \leq a_i < b_i \leq 1$, with the following properties:

1. If I, J are subintervals of $[0..1]$ with $I \subseteq J$, then $f(I) \leq f(J)$,
2. $f([0..1]) = 1$, $f(\emptyset) = 0$,
3. if $0 \leq a < b \leq 1$, then $f[a..b] > 0$.

If a participant i holds valuation function f_i, then $\text{Eval}(i, a, b) := f_i[a..b]$ expresses how i values the interval $[a..b]$.

Another useful function is $\text{Cut}(i, a, v)$, which gives the smallest $b > a$ for which $f_i[a..b] \geq v$.

Here is an algorithm due to the Polish mathematicians Banach and Knaster (described in [36]) using these concepts.

Banach-Knaster algorithm for cake cutting

1. Let N be the set of people sharing interval $[a..1]$.
2. If $N = 1$, the remaining person gets $[a..1]$.
3. Otherwise, let $i \in N$ be such that $b = \text{Cut}(i, a, \frac{1-a}{|N|})$ is minimal.
4. Give $[a..b]$ to i, and continue the cake cutting with $N - \{i\}$ and interval $[b..1]$.

Note the definition of $b = \text{Cut}(i, a, \frac{1-a}{|N|})$. This ensures that, in the valuation of i, the portion $[a..b]$ of the cake is worth $\frac{1}{|N|}$ of the current piece under discussion, which is $[a..1]$.

This is often called Banach and Knaster's *trimming algorithm*, because one might think of the determination of

"let $i \in N$ be such that $b = \text{Cut}(i, a, \frac{1-a}{|N|})$ is minimal"

as a process of trimming a given piece of cake down until no participant considers it too big anymore. Another way to think about this is as a process of determining where to cut, taking the valuations of all participants into account. (Looked at this way, the algorithm proposes a single cut, not a succession of cuts that get closer and closer to the final decision.)

Useful type abbreviations for an implementation:

```
type Agent    = Int
type Value    = Rational
type Boundary = Rational
type Valuation = Agent -> Boundary -> Value
```

The evaluation function can be defined from a valuation table:

```
eval :: Valuation -> Agent -> Boundary -> Boundary -> Value
eval f i a b = (f i) b - (f i) a
```

To illustrate this, it is useful to define an example valuation table. To start with, we define some goodies that are available (at some position) on the cake:

```
cherry, plum, chocolate, turkishdelight, lemon :: Boundary
(cherry, plum, chocolate, turkishdelight, lemon)
   = (1/6, 2/6, 3/6, 4/6, 5/6)
goodies = [cherry, plum, chocolate, turkishdelight, lemon]
```

Here is an example valuation table. `table i b` gives the value that agent i assigns to the interval $[0..b]$. Agents 1 and 2 have uniform valuation: all parts of the cake are alike to them.Agent 3 and 4 values the first half more than the second half. With agent 5 and 6 it is the reverse. Agents 7 and 8 value the first one third of the cake less than the rest.

```
table :: Valuation
table 1 b = b
table 2 b = b
table 3 b = if b < 1/2
            then b * (3/2)
            else (3/4) + (1/2)*(b - 1/2)
table 4 b = if b < 1/2
            then b * (3/2)
            else (3/4) + (1/2)*(b - 1/2)
table 5 b = if b < 1/2
            then b * (1/2)
            else (1/4) + (3/2)*(b - 1/2)
table 6 b = if b < 1/2
            then b * (1/2)
            else (1/4) + (3/2)*(b - 1/2)
table 7 b = if b < 1/3
            then b * (1/2)
            else (1/6) + (5/4)*(b - 1/3)
table 8 b = if b < 1/3
            then b * (1/2)
            else (1/6) + (5/4)*(b - 1/3)
```

It is left to the reader to check that all valuation functions defined so far in this table are proper valuation functions. Here is a partial check:

```
*FDA> take 10 [ eval table 1 0 (1/fromIntegral(n)) | n <- [1..] ]
[1 % 1,1 % 2,1 % 3,1 % 4,1 % 5,1 % 6,1 % 7,1 % 8,1 % 9,1 % 10]
*FDA> take 10 [ eval table 3 0 (1/fromIntegral(n)) | n <- [1..] ]
```

```
[1 % 1,3 % 4,1 % 2,3 % 8,3 % 10,1 % 4,3 % 14,3 % 16,1 % 6,3 % 20]
*FDA> take 10 [ eval table 5 0 (1/fromIntegral(n)) | n <- [1..] ]
[1 % 1,1 % 4,1 % 6,1 % 8,1 % 10,1 % 12,1 % 14,1 % 16,1 % 18,1 % 20]
*FDA> take 10 [ eval table 7 0 (1/fromIntegral(n)) | n <- [1..] ]
[1 % 1,3 % 8,1 % 6,1 % 8,1 % 10,1 % 12,1 % 14,1 % 16,1 % 18,1 % 20]
```

Next, some weird valuations:

```
table 9 b = let
    n = fromIntegral (length [ x | x <- goodies, x < b  ])
  in  n/5
table 10 b = if chocolate <= b then 1 else 0
```

Agent 9 cares only about the goodies, while agent 10 is obsessed with chocolate: she cares about nothing else. Note that the valuations for these agents are not proper: they do not satisfy the condition that each non-empty segment of the cake should have a positive value. The following check illustrates this:

```
*FDA> take 10 [ eval table 9 0 (1/fromIntegral(n)) | n <- [1..] ]
[1 % 1,2 % 5,1 % 5,1 % 5,1 % 5,0 % 1,0 % 1,0 % 1,0 % 1,0 % 1]
*FDA> take 10 [ eval table 10 0 (1/fromIntegral(n)) | n <- [1..] ]
[1 % 1,1 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 % 1,0 % 1]
```

We will use these weird valuations below to illustrate that the fairness of the Banach-Knaster algorithm depends on properties of the valuation functions.

The cut function can be defined from a valuation table provided we allow for some margin of error. The reason for this is that for cutting we need the inverse of the valuation, to determine the cut that yields a particular value.

```
epsilon = 1/10000

cut :: Valuation -> Agent -> Boundary -> Value -> Boundary
cut f i a v = approxCut f i v epsilon a a 1
```

The right cut location is found by binary search:

```
approxCut :: Valuation -> Agent -> Value -> Boundary ->
             Boundary ->  Boundary -> Boundary -> Boundary
approxCut f i v error a left right = let
   m = left + (right - left)/2
   guess = eval f i a m
 in
   if guess == v || right - left < error then m
   else if guess > v then approxCut f i v error a left m
   else approxCut f i v error a m right
```

Some example cuts:

```
*FDA> cut table 3 (1/2) (1/8)
3 % 4
*FDA> cut table 1 (1/2) (1/8)
5 % 8
*FDA> cut table 3 (1/2) (1/8)
3 % 4
*FDA> cut table 3 0 (1/2)
10923 % 32768
*FDA> cut table 7 0 (1/2)
19661 % 32768
*FDA> cut table 2 (1/2) (1/8)
5 % 8
*FDA> cut table 2 (1/2) (1/5)
22937 % 32768
*FDA> cut table 3 (1/2) (1/5)
29491 % 32768
*FDA> cut table 5 (1/2) (1/5)
20753 % 32768
*FDA> cut table 3 0 (1/2)
10923 % 32768
```

In the implementation, the function for ”I cut, you choose” has the following type:

```
cutAndChoose :: Valuation -> Agent -> Agent -> Boundary
             -> [(Agent,Boundary,Boundary)]
```

The implementation assumes that the first argument represents the cutter and the second argument the chooser:

```
cutAndChoose f i j a = let
    b = cut f i a ((1-a)/2)
  in
    if eval f j a b >= eval f j b 1
      then [(j,a,b),(i,b,1)]
      else [(i,a,b),(j,b,1)]
```

Note that the idea is to fairly divide the part $[a..1]$ of the interval $[0..1]$. Half of this part should have value $\frac{1-a}{2}$.

Now we can illustrate our earlier point that if valuations are private, it is better to choose that to cut. This is what happens if agent 1, who has uniform valuation, is the cutter:

```
*FDA> cutAndChoose table 1 3 0
[(3,0 % 1,1 % 2),(1,1 % 2,1 % 1)]
```

Agent 3 gets the part that is much more valuable to her:

```
*FDA> eval table 3 0 (1/2)
3 % 4
*FDA> eval table 3 (1/2) 1
1 % 4
```

If agent 3 does the cutting, this is what happens:

```
*FDA> cutAndChoose table 3 1 0
[(3,0 % 1,10923 % 32768),(1,10923 % 32768,1 % 1)]
```

Agent 3 gets a smaller part now. This is much better for agent 1, for:

```
*FDA> eval table 1 0 (10923/32768)
10923 % 32768
*FDA> eval table 1 (10923/32768) 1
21845 % 32768
```

To implement the Banach-Knaster algorithm, we need an auxiliary function for picking an element in a non-empty list with minimal f-value:

```
minim :: (Ord b) => (a -> b) -> [a] -> a
minim f = head .
          (sortBy (\ x y -> compare (f x) (f y)))
```

In the implementation of Banach-Knaster we use `minim` to pick the agent who cuts the smallest piece. The function `fromIntegral` converts `n` to a floating point number, which is needed for the division `(1/n)`.

```
bk :: Valuation -> [Agent] -> Boundary
   -> [(Agent,Boundary,Boundary)]
bk f [i] a = [(i,a,1)]
bk f js a = let
   n = fromIntegral (length js)
   g = \ j -> cut f j a ((1-a)/n)
   i = minim g js
   b = g i
 in
   (i,a,b) : bk f (js \\ [i]) b
```

An auxiliary function for evaluating all the pieces:

```
evl :: Valuation -> [(Agent,Boundary,Boundary)] -> [(Agent,Value)]
evl table = let
    f = \ (i,a,b) -> (i, eval table i a b)
  in map f
```

We call a cake division among N agents *fair* if each agent receives a slice that she values as at least $\frac{1}{N}$ of the cake.

```
fair :: Valuation -> Boundary
     -> [(Agent,Boundary,Boundary)] -> [Bool]
fair table a xs = let
    pairs = evl table xs
    n     = fromIntegral (length xs)
    f     =  \ (i,v) -> v + epsilon >= (a-1)/n
  in
    map f pairs
```

```
*FDA> bk table [1,3] 0
[(3,0 % 1,10923 % 32768),(1,10923 % 32768,1 % 1)]
*FDA> fair table 0 $ bk table [1,3] 0
[True,True]
```

```
*FDA> bk table [1,3,5] 0
[(3,0 % 1,7281 % 32768),(1,7281 % 32768,40049 % 65536),
                        (5,40049 % 65536,1 % 1)]
*FDA> fair table 0 $ bk table [1,3,5] 0
[True,True,True]
```

We can use this to write an assertive version of the Banach-Knaster cake cutting algorithm:

```
bkA :: Valuation -> [Agent] -> Boundary
   -> [(Agent,Boundary,Boundary)]
bkA = assert3 (\ f _ a outcome -> and (fair f a outcome)) bk
```

Here is an iterative formulation of the Banach-Knaster algorithm:

Banach-Knaster algorithm for cake cutting (iterative version)

1. Let N be the set of people sharing interval $[a..1]$.
2. While $|N| > 1$ do
   (a) let $i \in N$ be such that $b = \text{Cut}(i, a, \frac{1-a}{|N|})$ is minimal;
   (b) give $[a..b]$ to i;
   (c) $N := N - \{i\}$;
   (d) $a := b$.
3. The remaining person gets $[a..1]$.

An implementation of the iterative version of the Banach-Knaster algorithm:

```
bki ::  Valuation -> [Agent] -> [(Agent,Boundary,Boundary)]
bki f js = bki' f js 0 []

bki' ::  Valuation -> [Agent] -> Boundary
   -> [(Agent,Boundary,Boundary)] -> [(Agent,Boundary,Boundary)]
bki' f = while3
        (\ js _ _  -> not (null js))
        (\ js a xs -> let
            n = fromIntegral (length js)
            g = \ j -> cut f j a ((1-a)/n)
            i = minim g js
            b = if n > 1 then g i else 1
          in
            (js\\[i], b, (i,a,b):xs))
```

And here is another version, using a "for" loop:

```
bki ::  Valuation -> [Agent] -> [(Agent,Boundary,Boundary)]
bki f js = let
    k = length js in
  bki' f k js 0 []

bki' ::  Valuation -> Int -> [Agent] -> Boundary
   -> [(Agent,Boundary,Boundary)] -> [(Agent,Boundary,Boundary)]
bki' f k = for3 [1..k]
        (\ m js a xs -> let
            n = fromIntegral ((k+1) - m)
            g = \ j -> cut f j a ((1-a)/n)
            i = minim g js
            b = if n > 1 then g i else 1
          in
            (js\\[i], b, (i,a,b):xs))
```

Exercise 6.2 *Give your own version of* `bki'`*, using a* `fordown3` *loop rather than a* `for3` *loop.*

Will a cake partition found by the Banach-Knaster algorithm always be fair? That depends on the properties of the valuations. For valuations that satisfy all requirements stated above, the answer is 'yes'. This can easily be proved by induction, as follows.

If there is only one agent, Banach-Knaster gives the cake fragment $[a..1]$ to that agent. This is fair by definition.

Suppose the Banach-Knaster division is fair for any cake fragment $[b..1]$ and n agents. We have to show that the Banach Knaster division is also fair for any cake fragment and $n+1$ agents.

Banach-Knaster, for $n+1$ agent set A and cake fragment $[a..1]$, instructs us to compute $[a..b]$, where $b = \min_{i \in A} \text{Cut}(i, a, \frac{1-a}{N+1})$, and give this to an $i \in A$ with $\text{Cut}(i, a, \frac{1-a}{n+1}) = b$.

Then $f_i[a..b] = \frac{1}{n+1}[a..1]$, so i gets a fair share of $[a..1]$. Now take some arbitrary $j \in A - \{i\}$. Then $\text{Cut}(j, a, \frac{1-a}{n+1}) \geq b$. Will we be able to give j a fair share of $[b..1]$? If the valuation function for j is proper, then the answer is yes, for then it follows from the fact that $\text{Cut}(j, a, \frac{1-a}{n+1}) \geq b$ that $f_j[a..b] \leq \frac{1}{n+1}[a..1]$, and therefore,

$$f_j[b..1] = f_j[a..1] - f_j[a..b] \geq \frac{n}{n+1}[a..1].$$

By the induction hypothesis, j gets a fair share of $[b..1]$.

6.4 Some Background on Valuation and Measure

A valuation function with the properties mentioned on page 6-4 is often called a *measure function*.

Measure functions arise in probability theory as well, and it turns out that the valuation functions we need for expressing the individual appreciation of parts of a cake are related to so-called mass density functions in the same way as in probability theory.

A probability density function (p.d.f.) expresses the relative likelihood that a random variable takes a particular value. A probability density function is non-negative everywhere, and its integral is equal to one.

In our context, a mass density function is a function that is positive on any x in the interval $[0..1`]$, and has an integral equal to one.

The uniform valuation (the valuation of agents 1 and 2 in `table`) arises from the mass density function $\lambda x \mapsto 1$. It is the function F that has $\lambda x \mapsto 1$ as its derivative, i.e., the function $F = \lambda x \mapsto x$. For if $F(x) = x$ then $F'(x) = 1$.

And so on.

6.5 Concern with the Number of Cuts

The mathematical cake-cutting literature is concerned with the question of finding cake-cutting algorithms with a minimum number of cuts. In the case of the Banach-Knaster algorithm, for n participants, it is argued that n cuts are needed for determining the first slice, $n-1$ cuts for the next slice, and so on. All in all this gives

$$n + (n-1) + \cdots + 1 = \frac{n(n+1)}{2}$$

cuts, which gives $O(n^2)$ cuts. Can we do better?

Before going on, let us note that the Banach-Knaster algorithm is efficient, and divides the cake into continuous slices. This is because the 'extra cuts' are just needed to determine the place of the actual cut; they do not destroy the cake.

This being said, we will follow the mathematician's trail anyway. The following *Divide and Conquer* algorithm (first proposed in [16]) uses fewer cuts than the Banach-Knaster algorithm:

Divide and Conquer Algorithm for Cake Cutting

1. If $n = 1$, all of the cake goes to the single participant.
2. Otherwise, if $n = 2k$, let all but one participant i indicate where they would cut the cake $[a..b]$ in halves.
   (a) Let m be the middle cut.
   (b) i chooses between $[a..m]$ and $[m..b]$.
   (c) If i chooses $[a..m]$, let i plus all participants that have their cut in $[a..m]$ divide $[a..m]$, and let the remaining participants divide $[m..b]$. Otherwise, proceed the other way around.
3. Otherwise, if $n = 2k + 1$, let all but one participant i indicate where they would cut the cake $[a..b]$ in the ratio $k : k + 1$.
   (a) Let m be the k-th cut from left to right.
   (b) i chooses either $[a..m]$ as at least $\frac{k}{2k+1}$ of the cake, or $[m..b]$ as at least $\frac{k+1}{2k+1}$ of the cake.
   (c) In the first case, i plus the $k - 1$ participants whose cuts are in $[a..m]$ divide $[a..m]$ and the others divide $[m..b]$. In the other case, i plus the k participants that have their cut in $[m..b]$ divide $[m..b]$ and the others divide $[a..m]$.

Exercise 6.3 *Implement this algorithm.*

Exercise 6.4 *Next, express the assertion that the algorithm is fair, and implement an assertive version of the algorithm.*

6.6 Cake Cutting with Unequal Shares

Suppose Alice and Bob have to share a cake, and it is agreed that Alice is entitled tod $\frac{5}{13}$ of the cake and Bob to $\frac{8}{13}$. Then an obvious thing to do would be to generalize *Cut and Choose* as follows.

Generalized Cut and Choose, for Cake Division into $\frac{k}{k+m}$, $\frac{m}{k+m}$

1. Let the participants be 1 and 2. Assume 1 is entitled to $\frac{k}{k+m}$ and 2 is entitled to $\frac{m}{k+m}$.
2. 1 cuts the cake into $k+m$ shares X with $f_1(X) = \frac{1}{k+m}$.
3. 2 selects m of the shares; the other shares go to 1.

```
ramsey :: Int -> Int -> [Int]
ramsey x y = if x == y then [x,1]
             else if x > y then x: ramsey (x-y) y
             else y: ramsey x (y-x)
```

6.7 Envy-Free Cake Cutting

We have seem that the cake divisions produced by Banach-Knaster are fair. Still, the results of the division may cause hard feelings among the participants. Suppose Alice gets her fair share which she considers worth $\frac{1}{3}$ of the cake. Next, Bob and Carol divide the rest. Both get a fair share: at least half of the rest, in their own evaluation. But Alice sees to her dismay that the piece that Bob considered a fair share is worth *much more* than the piece she got herself. (It follows that Alice also believes that the piece that Carol received is worth *much less* than her own piece, but such findings are generally easier to live with.)

Can we cut cakes in such a way that these feelings of envy are avoided? In the case of cake cutting for two, we can, for we know that "cut and choose" has as result that the cutter has one half of the cake (in his own estimation), while the chooser estimates to have at least one half. Neither of them wants to swap with the other.

In the case of three or more, matters are different. The example above shows that the Banach-Knaster algorithm does not guarantee envy-freeness, even in the simple case of sharing a cake with three participants.

Stromquist [37] describes the following ingenious algorithm for envy-free cake cutting for three.

Envy-free Cake Cutting for Three: Moving Knives

"A referee moves a sword from left to right over the cake, hypothetically dividing it in a small left piece and a large right piece. Each player holds a knife over what he considers to be the midopoint of the right piece. As the referee moves his word, the players continually adjust their knives, aways keeping them parallel tot the sword [..]. When any player shouts "cut" the cake is cut by the sword and by whichever of the players' knives happens to be in the middle of the three.
The player who shouted "cut" receives the left piece. He must be satisfied, because he knew what all three pieces would be when he said the word. Then the player whose knife ended nearest to the sword, if he didn't shout "cut," takes the center piece; and the player whose knife was farthest from the sword, if he didn't shout "cut," takes the right piece. The player whose knife was used to cut the cake, if he hasn't already taken the left piece, will be satisfied with whatever piece is left over. If ties must be broken — either because two or three players shout simultaneously or because two or three knives coincide — they may be broken arbitrarily." [37]

If we reflect on this, we see how we can turn this into a determinate algorithm.

Let's put ourselves in the place of Alice. The sword moves to a position ready to cut off one third of the cake, according to her valuation. Will she shout "Stop"? That depends. If it so happens that her knife is in the middle (between the two knives of Bob and Carol) over the larger part of the cake, she will. For if she shouts stop, the sword will cut off one third in her valuation, and the remaining two thirds will be cut in halves, again in her valuation. Fair and envy-free for Alice.

Matters are different, however, if her knive is to the left of the two other knives, or to the right of the two other knives. Shouting "stop" now will surely create envy, for if she does that, Alice will end up with a fair piece for her, but either Bob's or Carol's knife will not chop the remaining part of the cake in two (according to Alice's valuation), but, horror of horrors, into uneven parts one of which is more desirable than her own piece. Envy raises its ugly head.

How could this be prevented? Let δ be the value (in Alice's coinage) of the slice between the cutting knife and her own knife. Her own knife cuts the remaining part into two parts that are each worth exactly one third of the whole cake, according to Alice. One half of that slice will not be grudged by the others.

Making a quick calculation, Alice decides not to go for one third of the cake in her first go, but for a portion worth $\frac{1}{3} + \frac{\delta}{2}$ of the whole cake. OK, suppose the sword is exactly at this position? Will she shout "Stop"? In her knife is the middle knife now, or if the distance between the cutting knife and her own knife is less than $\frac{\delta}{2}$ now, she will. Otherwise, she will make another quick calculation, and claim one half of the slice that is now under dispute. And so on.

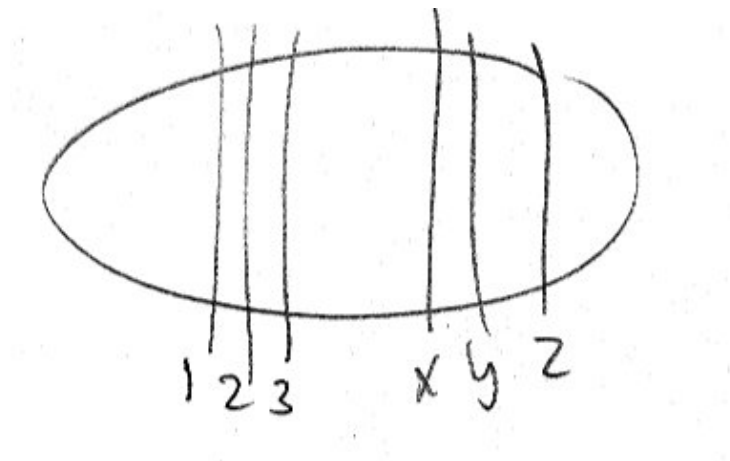
Here is how an extra slice can be claimed:

```
claim :: Valuation -> Agent -> (Agent,Agent)
      -> Boundary -> Boundary -> Value
claim f x1 (x2,x3) a b = let
    a1 = cut f x1 a ((b-a)/3)
    b1 = cut f x1 a1 ((b-a1)/2)
    b2 = cut f x2 a1 ((b-a1)/2)
    b3 = cut f x3 a1 ((b-a1)/2)
    g  = \ (_,x) (_,y) -> compare x y
    [(i1,c1),(i2,c2),(i3,c3)] =
      sortBy g [(x1,b1),(x2,b2),(x3,b3)]
  in
    if x1 == i2 then 0
    else if x1 == i1 then (eval table x1 c1 c2)/2
    else (eval table x1 c2 c3)/2
```

Use this in an envy-free version of cake cutting for three:

```
envyFree :: Valuation -> (Agent,Agent,Agent)
             -> [(Agent,Boundary,Boundary)]
envyFree f (x1,x2,x3) = let
   g = \ j -> let
           [k,l] = [x1,x2,x3] \\ [j]
       in claim f j (k,l) 0 1
   i = minim g [x1,x2,x3]
   [k,l] =  [x1,x2,x3] \\ [i]
   v = g i
   a = cut f i 0 ((1/3)+v)
   b = cut f i a ((1-a)/2)
  in
    if eval f k a b <= eval f k b 1 then
       [(i,0,a),(k,a,b),(l,b,1)]
    else
       [(i,0,a),(l,a,b),(k,b,1)]
```

Let's start again. Let each player indicate what slice of the cake, counting from the left, would be worth one third of the whole cake. Call these cuts $1, 2, 3$. Next, let them do the same, but now starting from the right side. Call these cuts x, y, z. So the piece of cake to the right of x is worth one third of the cake to the player who made the cut.



Now there are six cases to consider.

1. $(1, 2, 3) = (x, y, z)$. That is, cut x was made by 1, y by 2, z by 3.

The following algorithm for envy free division for three is attributed (in [9]) to Conway and Selfridge.

Envy Free Division for 3 Players (Conway, Selfridge)

1. Call the measures of the three players μ_1, μ_2, μ_3.
2. Player 1 cuts cake into 3 equal pieces (according to μ_1).
3. If the two largest pieces according to μ_2 have unequal sizes, player 2 cuts the largest of them down to size by slicing off a piece L. This gives pieces X, Y, Z, and maybe a leftover piece L, trimmed from X.
4. Players 3, 2, 1, in that order, choose a piece. If 2 trimmed X, he has to choose X if player 3 does not choose it.
5. If there was no trimming we are done. Otherwise, let x be the player that received X, and let y be the other from players $2, 3$. Let y cut L into three equal pieces L_1, L_2, L_3 (according to μ_y).
6. The three pieces L_1, L_2, L_3 are divided by letting x choose first and y next, while 1 has to take the remaining piece.

```
findMaxValues :: [(a,a,Value)] -> [(a,a)]
findMaxValues pieces = let
    values = map (\(_,_,v) -> v) pieces
    best   = filter
               (\(_,_,v) -> v + epsilon >= maximum values) pieces
  in
    map (\ (a,b,_) -> (a,b)) best
```

```
pick :: Valuation -> Agent
     -> [(Boundary,Boundary)] -> [(Boundary,Boundary)]
pick f i pieces = let
    xs = map (\ (a,b) -> (a,b,eval f i a b)) pieces
  in
    findMaxValues xs
```

```
trim :: Valuation -> Agent -> Value -> Boundary -> Boundary
     -> [(Boundary,Boundary)] -> [(Boundary,Boundary)]
trim f i v b1 b2 = let
    c1 = cut f i b1 v
  in
    [(b1,c1),(c1,b2)]
```

```
cs :: Valuation -> (Agent,Agent,Agent)
                -> [(Agent,Boundary,Boundary)]
cs f (x1,x2,x3) = let
   a = cut f x1 0 (1/3)
   b = cut f x1 a (1/3)
   pieces = [(0,a),(a,b),(b,1)]
   xs = pick f x2 pieces
   (a1,a2) = head xs
   ys = pick f x3 (pieces \\ [(a1,a2)])
   (b1,b2) = head ys
   (c1,c2) = head (pieces \\ [(a1,a2),(b1,b2)])
  in
   if length xs >= 2 then [(x2,a1,a2),(x3,b1,b2),(x1,c1,c2)]
   else let
     [(piece1,piece2,_)] =
        sortBy (\ x y -> compare (eval f x2 y) (eval f x2 x)) pieces

   in error "envy"
```

```
try f (x1,x2,x3) = let
   a = cut f x1 0 (1/3)
   b = cut f x1 a (1/3)
  in
   pick f x2 [(0,a),(a,b),(b,1)]
```

6.8 Dividing an Estate, or Dividing a Burden

Suppose we want to divide m desirable objects (an 'estate') or undesirable objects (a burden) among n participants who are each entitled to an equal share of the estate (or obliged to an equal share in the burden). Each participant has her own valuation of the items. Can we make a division that is fair and envy-free? Each participant should receive at least $\frac{1}{n}$ of the estate, in her own estimation. Or in the case of a burden: each participant has to take care of at most $\frac{1}{n}$, in her own estimation. And (envy-freeness) no participant should be willing to trade her share with that of any other participant. Here is a simple example.

Alice, Bob and Carol have to divide an estate consisting of a cabrio car, a station car, a sailing boat, a grand piano and a collector's wrist watch. Given that these items are so diverse, they each have different valuations for the objects. Is there a procedure for fair division that will not cause hard feelings?

It turns out that there is. The big equalizer is money. This section will explain and implement the

procedure proposed by Haake, Raith and Su in [20], where money is used to establish fairness and envy-freeness through internal market pricing.

Rather than indicating where to make a cut in a cake, players can make bids on the items in the estate. We have to assume that the utility of money for the players is linear. That is, twice as much money is twice as useful to them. When really big money is involved, this assumption is unrealistic, but in the present setting we will ignore this.

6.9 Splitting the Rent

> My friend's dilemma was a practical question that mathematics could answer, both elegantly and constructively. He and his housemates were moving to a house with rooms of various sizes and features, and were having trouble deciding who should get which room and for what part of the total rent. He asked, "Do you think there's always a way to partition the rent so that each person prefers a different room?" [38]

6.10 Summary

...

Appendix A

While

```
module While

where
```

```
while1 :: (a -> Bool) -> (a -> a) -> a -> a
while1 p f x
  | p x       = while1 p f (f x)
  | otherwise = x
```

Another way to express this is in terms of the built-in Haskell function `until`:

```
neg :: (a -> Bool) -> (a -> Bool)
neg p = \x -> not (p x)

while1 = until . neg
```

```
while2 :: (a -> b -> Bool)
       -> (a -> b -> (a,b))
       -> a -> b -> b
while2 p f x y
  | p x y     = let (x',y') = f x y in while2 p f x' y'
  | otherwise = y
```

```
while3 :: (a -> b -> c -> Bool)
       -> (a -> b -> c -> (a,b,c))
       -> a -> b -> c -> c
while3 p f x y z
  | p x y z   = let
                  (x',y',z') = f x y z
                in while3 p f x' y' z'
  | otherwise = z
```

```
while4 :: (a -> b -> c -> d -> Bool)
       -> (a -> b -> c -> d -> (a,b,c,d))
       -> a -> b -> c -> d -> d
while4 p f x y z v
  | p x y z v = let
                 (x',y',z',v') = f x y z v
                in while4 p f x' y' z' v'
  | otherwise = v
```

```
while5 :: (a -> b -> c -> d -> e -> Bool)
       -> (a -> b -> c -> d -> e -> (a,b,c,d,e))
       -> a -> b -> c -> d -> e -> e
while5 p f x y z v w
  | p x y z v w = let
                    (x',y',z',v',w') = f x y z v w
                  in while5 p f x' y' z' v' w'
  | otherwise = w
```

Similarly, we can define repeat wrappers:

```
repeat1 :: (a -> a) -> (a -> Bool) -> a -> a
repeat1 f p = while1 (\ x -> not (p x)) f . f
```

Another way to express this is in terms of the built-in function `until`, as follows:

```
repeat1 :: (a -> a) -> (a -> Bool) -> a -> a
repeat1 f p = until p f . f
```

```
repeat2 :: (a -> b -> (a,b))
        -> (a -> b -> Bool) -> a -> b -> b
repeat2 f p x y = let
     (x1,y1)  = f x y
     negp    = (\ x y -> not (p x y))
  in while2 negp f x1 y1
```

```
repeat3 :: (a -> b -> c -> (a,b,c))
        -> (a -> b -> c -> Bool) -> a -> b -> c -> c
repeat3 f p x y z = let
     (x1,y1,z1)  = f x y z
     negp    = (\ x y z -> not (p x y z))
  in while3 negp f x1 y1 z1
```

```
repeat4 :: (a -> b -> c -> d -> (a,b,c,d))
        -> (a -> b -> c -> d -> Bool)
        -> a -> b -> c -> d -> d
repeat4 f p x y z u = let
     (x1,y1,z1,u1)  = f x y z u
     negp    = (\ x y z u -> not (p x y z u))
  in while4 negp f x1 y1 z1 u1
```

```
repeat5 :: (a -> b -> c -> d -> e -> (a,b,c,d,e))
        -> (a -> b -> c -> d -> e -> Bool)
        -> a -> b -> c -> d -> e -> e
repeat5 f p x y z u v = let
     (x1,y1,z1,u1,v1)  = f x y z u v
     negp    = (\ x y z u v -> not (p x y z u v))
  in while5 negp f x1 y1 z1 u1 v1
```

For loops:

```
for :: [a] -> (a -> b -> b) -> b -> b
for [] f y = y
for (x:xs) f y = for xs f (f x y)
```

Note that this is a variant of `foldl`, witness the following alternative definition:

```
for xs f y = foldl (flip f) y xs
```

Here is a version of "for" where the step function has an additional argument:

```
for2 :: [a] -> (a -> b -> c -> (b,c))
           -> b -> c -> c
for2 [] f _ z = z
for2 (x:xs) f y z = let
    (y',z') = f x y z
  in
    for2 xs f y' z'
```

With two additional arguments:

```
for3 :: [a] -> (a -> b -> c -> d -> (b,c,d))
           -> b -> c -> d -> d
for3 [] f _ _ u = u
for3 (x:xs) f y z u = let
    (y',z',u') = f x y z u
  in
    for3 xs f y' z' u'
```

And so on.

We can also count down instead of up:

```
fordown :: [a] -> (a -> b -> b) -> b -> b
fordown = for . reverse
```

```
fordown2 :: [a] -> (a -> b -> c -> (b,c))
                -> b -> c -> c
fordown2 = for2 . reverse
```

```
fordown3 :: [a] -> (a -> b -> c -> d -> (b,c,d))
                      -> b -> c -> d -> d
fordown3 = for3 . reverse
```

Appendix B

Assert

```
module Assert

where
```

```
infix 1 ==>

(==>) :: Bool -> Bool -> Bool
p ==> q = (not p) || q
```

```
forall = flip all
```

```
assert1 :: (a -> b -> Bool) -> (a -> b) -> a -> b
assert1 p f x = if p x (f x) then f x
                else error "assert1"
```

```
assert2 ::  (a -> b -> c -> Bool)
             -> (a -> b -> c) -> a -> b -> c
assert2 p f x y =
  if p x y (f x y) then f x y
  else error "assert2"
```

```
assert3 ::  (a -> b -> c -> d -> Bool)
             -> (a -> b -> c -> d) -> a -> b -> c -> d
assert3 p f x y z =
  if p x y z (f x y z) then f x y z
  else error "assert3"
```

```
assert4 ::  (a -> b -> c -> d -> e -> Bool)
             -> (a -> b -> c -> d -> e)
             -> a -> b -> c -> d -> e
assert4 p f x y z u =
  if p x y z u (f x y z u) then f x y z u
  else error "assert4"
```

```
assert5 ::  (a -> b -> c -> d -> e -> f -> Bool)
             -> (a -> b -> c -> d -> e -> f)
             -> a -> b -> c -> d -> e -> f
assert5 p f x y z u v =
  if p x y z u v (f x y z u v) then f x y z u v
  else error "assert5"
```

```
invar1 :: (a -> Bool) -> (a -> a) -> a -> a
invar1 p f x =
  let
    x' = f x
  in
  if p x && not (p x') then error "invar1"
  else x'
```

```
invar2 :: (a -> b -> Bool) ->
          (a -> b -> (a,b)) ->
           a -> b -> (a,b)
invar2 p f x y =
  let
    (x',y') = f x y
  in
    if p x y && not (p x' y') then error "invar2"
    else (x',y')
```

```
invar3 :: (a -> b -> c -> Bool) ->
          (a -> b -> c -> (a,b,c)) ->
           a -> b -> c -> (a,b,c)
invar3 p f x y z =
  let
    (x',y',z') = f x y z
  in
   if p x y z && not (p x' y' z') then error "invar3"
   else (x',y',z')
```

```
invar4 :: (a -> b -> c -> d -> Bool) ->
          (a -> b -> c -> d -> (a,b,c,d)) ->
           a -> b -> c -> d -> (a,b,c,d)
invar4 p f x y z u =
  let
    (x',y',z',u') = f x y z u
  in
   if p x y z u && not (p x' y' z' u')
   then error "invar4"
   else (x',y',z',u')
```

```
invar5 :: (a -> b -> c -> d -> e -> Bool) ->
          (a -> b -> c -> d -> e -> (a,b,c,d,e)) ->
           a -> b -> c -> d -> e -> (a,b,c,d,e)
invar5 p f x y z u v =
  let
    (x',y',z',u',v') = f x y z u v
  in
   if p x y z u v && not (p x' y' z' u' v')
   then error "invar5"
   else (x',y',z',u',v')
```

Assertion Wrappers for Debugging The following assertion wrappers are less general than the ones used above, but more useful for debugging:

```
assrt1 :: (Show a, Show b) => String
                           -> (a -> b -> Bool)
                           -> (a -> b) -> a -> b
assrt1 info p f x =
  if p x (f x) then f x
  else error ("assrt1:" ++ info ++ show(x,f x))
```

```
assrt2 :: (Show a, Show b,Show c)
   => String
   -> (a -> b -> c -> Bool)
   -> (a -> b -> c) -> a -> b -> c
assrt2 info p f x y =
  if p x y (f x y) then f x y
  else error ("assrt2:" ++ info ++ show(x,y,f x y))
```

```
assrt3 :: (Show a, Show b,Show c,Show d)
  => String
  -> (a -> b -> c -> d -> Bool)
  -> (a -> b -> c -> d) -> a -> b -> c -> d
assrt3 info p f x y z =
  if p x y z (f x y z) then f x y z
  else error ("assrt3" ++ info ++ show(x,y,z,f x y z))
```

```
assrt4 ::  (Show a, Show b,Show c,Show d,Show e)
  => String
  -> (a -> b -> c -> d -> e -> Bool)
  -> (a -> b -> c -> d -> e)
  -> a -> b -> c -> d -> e
assrt4 info p f x y z u =
  if p x y z u (f x y z u) then f x y z u
  else error ("assrt4" ++ info ++ show(x,y,z,u,f x y z u))
```

```
assrt5 ::  (Show a, Show b,Show c,Show d,Show e,Show f)
  => String
  -> (a -> b -> c -> d -> e -> f -> Bool)
  -> (a -> b -> c -> d -> e -> f)
  -> a -> b -> c -> d -> e -> f
assrt5 info p f x y z u v =
  if p x y z u v (f x y z u v) then f x y z u v
  else error ("assert5"++info++show(x,y,z,u,v,f x y z u v))
```

And similarly for the invariants:

```
invr1 ::  Show a => String -> (a -> Bool)
   -> (a -> a) ->  a -> a
invr1 info p f x =
   if p x  && not (p (f x)) then
      error ("invr1:" ++ info ++ show(x,f x))
   else f x
```

```
invr2 ::  (Show a, Show b)
   => String
   -> (a -> b -> Bool)
   -> (a -> b -> (a,b))
   ->  a -> b -> (a,b)
invr2 info p f x y =
  let
    (x',y') = f x y
  in
   if p x y  && not (p x' y') then
      error ("invr2:" ++ info ++ show(x,y,f x y))
   else (x',y')
```

```
invr3 ::  (Show a, Show b,Show c)
   => String
   -> (a -> b -> c -> Bool)
   -> (a -> b -> c -> (a,b,c))
   ->  a -> b -> c -> (a,b,c)
invr3 info p f x y z =
  let
    (x',y',z') = f x y z
  in
   if p x y z && not (p x' y' z') then
      error ("invr3:" ++ info ++ show(x,y,z,f x y z))
   else (x',y',z')
```

```
invr4 ::  (Show a, Show b,Show c,Show d)
   => String
   -> (a -> b -> c -> d -> Bool)
   -> (a -> b -> c -> d -> (a,b,c,d))
   ->  a -> b -> c -> d -> (a,b,c,d)
invr4 info p f x y z u =
  let
    (x',y',z',u') = f x y z u
  in
   if p x y z u && not (p x' y' z' u') then
      error ("invr4:" ++ info ++ show(x,y,z,u,f x y z u))
   else (x',y',z',u')
```

```
invr5 ::  (Show a, Show b,Show c,Show d,Show e)
   => String
   -> (a -> b -> c -> d -> e -> Bool)
   -> (a -> b -> c -> d -> e -> (a,b,c,d,e))
   ->  a -> b -> c -> d -> e -> (a,b,c,d,e)
invr5 info p f x y z u v =
  let
    (x',y',z',u',v') = f x y z u v
  in
   if p x y z u v && not (p x' y' z' u' v') then
      error ("invr5:" ++ info
              ++ show(x,y,z,u,v,f x y z u v))
   else (x',y',z',u',v')
```

Appendix C

AssertDoc

```
module AssertDoc

where
```

```
infix 1 ==>

(==>) :: Bool -> Bool -> Bool
p ==> q = (not p) || q
```

```
forall = flip all
```

Fake versions of assert and invariant statements, to turn off assertion and invariant testing.

```
assert1 :: (a -> b -> Bool) -> (a -> b) -> a -> b
assert1 _ = id
```

```
assert2 ::  (a -> b -> c -> Bool)
             -> (a -> b -> c) -> a -> b -> c
assert2 _ = id
```

```
assert3 ::  (a -> b -> c -> d -> Bool)
             -> (a -> b -> c -> d) -> a -> b -> c -> d
assert3 _ = id
```

```
assert4 ::  (a -> b -> c -> d -> e -> Bool)
             -> (a -> b -> c -> d -> e)
             -> a -> b -> c -> d -> e
assert4 _ = id
```

```
assert5 ::  (a -> b -> c -> d -> e -> f -> Bool)
             -> (a -> b -> c -> d -> e -> f)
             -> a -> b -> c -> d -> e -> f
assert5 _ = id
```

```
invar1 :: (a -> Bool) -> (a -> a) -> a -> a
invar1 _ = id
```

```
invar2 :: (a -> b -> Bool) ->
          (a -> b -> (a,b)) ->
           a -> b -> (a,b)
invar2 _ = id
```

```
invar3 :: (a -> b -> c -> Bool) ->
          (a -> b -> c -> (a,b,c)) ->
           a -> b -> c -> (a,b,c)
invar3 _ = id
```

```
invar4 :: (a -> b -> c -> d -> Bool) ->
          (a -> b -> c -> d -> (a,b,c,d)) ->
           a -> b -> c -> d -> (a,b,c,d)
invar4 _ = id
```

```
invar5 :: (a -> b -> c -> d -> e -> Bool) ->
          (a -> b -> c -> d -> e -> (a,b,c,d,e)) ->
           a -> b -> c -> d -> e -> (a,b,c,d,e)
invar5 _ = id
```

Appendix D

Answers to Selected Exercises

Chapter 1

```
module WLHA where

import WLH
```

Answer to Exercise 1.1

```
sign3,sign4 :: (Creature,Creature) -> Bool
sign3 (x,y) = x == Lady || y == Lady
sign4 (x,y) = x == Tiger

solution2 :: [(Creature,Creature)]
solution2 = [ (x,y) | x <- [Lady,Tiger],
                      y <- [Lady,Tiger],
                      (sign3 (x,y) && sign4 (x,y))
                       || (not (sign3 (x,y)) && not (sign4 (x,y)))]
```

This gives:

```
*WLHA> solution2
[(Tiger,Lady)]
```

Answer to Exercise 1.2

```
john2, bill2 :: (Islander,Islander) -> Bool
john2 (x,y) = x == y
bill2 (x,y) = x /= y

solution4 :: [(Islander,Islander)]
solution4 = [(x,y) | x <- [Knight,Knave],
                     y <- [Knight,Knave],
                     (x == Knight) == john2 (x,y),
                     (y == Knight) == bill2 (x,y) ]
```

This gives:

```
*WLHA> solution4
[(Knave,Knight)]
```

Answer to Exercise 1.3:

The function displays its own definition on the screen.

Answer to Exercise 1.4:

Gödel's incompleteness proof constructs a sentence that says of itself that it cannot be proved. The big insight is that the language of first order arithmetic is expressive enough to *talk about itself*. The function `main` is *talking about itself* in exactly the same way.

Chapter 3

```
module ASA where

import AS
```

Answer to Exercise 3.1:

A suitable assertion is: the output list of the function is sorted. We already have a function `sorted`, so writing an assertive version of `mergeSrt` is easy peasy:

```
mergeSrtA ::  Ord a => [a] -> [a]
mergeSrtA = assert1 (\_ ys -> sorted ys) mergeSrt
```

Answer to Exercise 3.2:

This does not work, for the Haskell compiler is too clever: it knows that it does not have to evaluate f on its arguments in order to find out that `triv` $\cdot$ f will yield True.

Chapter 4

```
module GAA where

import List
import While
import Assert
import GA
```

Answer to Exercise 4.1:

A reasonable assertion is:

$$y \in \text{reachable } E\ x \text{ iff } xE^*y.$$

A test using this assertion is given in Section 4.3.

Answer to Exercise 4.2: the changes are minimal; see below.

```
reachable1 :: Eq a => ([a], a -> a -> Bool) -> a -> [a]
reachable1 g x = reachable1' g [x] [x]

reachable1' :: Eq a => ([a],a->a->Bool) -> [a] -> [a] -> [a]
reachable1' (nodes,r) = let
   pairs = [ (x,y) | x <- nodes, y <- nodes ]
   in while2
   (\ current _ -> not (null current))
   (\ current marked -> let
     (y,rest) = (head current, tail current)
     newnodes = [ z  | (u,z) <- pairs, r u z, u == y,
                       notElem z marked   ]
     current' = rest ++ newnodes
     marked'  = marked ++ newnodes
  in
     (current', marked'))
```

Answer to Exercise 4.3:

```
cyclic :: Eq a => [(a,a)] -> Bool
cyclic g =
   any (\ (x,y) -> elem x (reachable g y)) g
```

Note that the simpler test `elem x (reachable g x)` cannot be used, for any x is reachable from itself.

Answer to Exercise 4.4: a graph is connected if its reflexive transitive closure equals the total relation on the node set.

```
isConn :: Eq a => [(a,a)] -> Bool
isConn r = let
    nodes = nub (map fst r ++ map snd r)
    total = [ (x,y) | x <- nodes, y <- nodes ]
  in
    equalS total (rtc r)
```

Answer to Exercise 4.5:

...

Answer to Exercise 4.6:

...

Answer to Exercise 4.7:

...

Answer to Exercise 4.9:

...

Answer to Exercise 4.10:

...

Answer to Exercise 4.11:

...

Chapter 5

Answer to Exercise 5.1:

...

Answer to Exercise 5.2:

...

Answer to Exercise 5.3:

...

Answer to Exercise 5.4:

...

Chapter 6

```
module FDAA where

import List
import While
import Assert
import FDA
```

Answer to Exercise 6.1:

Here are three things one of the participants could say to the other:

1. "If the traveller who brought us wealth hadn't arrived, we would have shared the chapatis equally. So it is only fair if we now share the eight coins equally as well."
2. "If the traveller who brought us wealth hadn't arrived, I would have bought one chapati from you at the going rate for chapatis. Now that the traveller was so generous, the going rate suddenly went up to one gold coin for a chapati. So my chapatis turned out to be worth three gold coins and yours five gold coins. It is only fair if I get three coins and you get five."
3. "Listen, the traveller paid for what he has eaten. The traveller has eaten one third of eight chapatis. You had only three chapatis to start with, and therefore he has eaten one third chapati from you and seven/thirds chapatis from me. So it is only fair if you get one coin and I get seven."

A moral of this could be that there is no obviously correct notion of fairness, in this case, and in many cases.

Answer to Exercise 6.2:

```
bki' ::  Valuation -> Int -> [Agent] -> Boundary
   -> [(Agent,Boundary,Boundary)] -> [(Agent,Boundary,Boundary)]
bki' f k = fordown3 [1..k]
        (\ m js a xs -> let
            n = fromIntegral m
            g = \ j -> cut f j a ((1-a)/n)
            i = minim g js
            b = if n > 1 then g i else 1
          in
            (js\\[i], b, (i,a,b):xs))
```

Answer to Exercise 6.3:

...

Answer to Exercise 6.4:

. . .

Bibliography

[1] Alfred V. Aho and Jeffrey D. Ullman. *Foundations of Computer Science — C edition*. W. H. Freeman, 1994.

[2] Albert-László Barabási. *Linked*. Penguin, 2002.

[3] Kent Beck. *Test Driven Development By Example*. Addison-Wesley Longman, Boston, MA, 2002.

[4] Richard Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[5] Richard Bird. *Pearls of Functional Algorithm Design*. Cambridge University Press, 2010.

[6] S.J. Brams and A.D. Taylor. *Fair Division: From Cake-Cutting to Dispute-Resolution*. Cambridge University Press, 1996.

[7] Steven Brams. Fair division. In Barry R. Weingast and Donald Wittman, editors, *Oxford Handbook of Political Economy*. Oxford University Press, 2005.

[8] Steven Brams. *Mathematics and Democracy:Designing Better Voting and Fair Division Procedures*. Princeton University Press, 2008.

[9] Steven J. Brams and Alan D. Taylor. An envy-free cake division protocol. *The American Mathematical Monthly*, 102(1):9–18, January 1995.

[10] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proc. Of International Conference on Functional Programming (ICFP)*, ACM SIGPLAN, 2000.

[11] Hal Daume. Yet another Haskell tutorial. `www.cs.utah.edu/~hal/docs/daume02yaht.pdf`.

[12] E.W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[13] E.W. Dijkstra. Guarded commands, nondeterminacy and the formal derivation of programs. *Communications of the ACM*, 18:453–457, 1975.

[14] K. Doets and J. van Eijck. *The Haskell Road to Logic, Maths and Programming*, volume 4 of *Texts in Computing*. College Publications, London, 2004.

[15] L.E. Dubisn and E.H. Spanier. How to cut a cake fairly. *American Mathematical Monthly*, 68(1):1–17, 1961.

[16] S. Even and A. Paz. A note on cake cutting. *Discrete Applied Mathematics*, 7:285–296, 1984.

[17] Shimon Even. *Graph Algorithms, 2nd Edition*. Cambridge University Press, 2012.

[18] Robert W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5(6):345, June 1962.

[19] D. Gale and L. Shapley. College admissions and the stability of marriage. *American Mathematical Monthly*, 69:9–15, 1962.

[20] Claus-Jochen Haake, Matthias G. Raith, and Francis Edward Su. Bidding for envy-freeness: a procedural approach to n-player fair-division problems. *Social Choice and Welfare*, 19(4):723–749, 2002.

[21] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):567–580, 583, 1969.

[22] Paul Hudak, John Hughes, Simon Peyton Jones, and Philip Wadler. A history of Haskell: Being lazy with class. In *Third ACM SIGPLAN History of Programming Languages Conference (HOPL-III)*, 2007.

[23] Robert W. Irving. An efficient algorithm for the "stable roommates" problem. *Journal of Algorithms*, 6(4):577–595, 1985.

[24] S.C. Kleene. Representation of events in nerve nets and finite automata. In C. E. Shannon and J. McCarthy, editors, *Automata Studies*, pages 3–42. Princeton University Press, 1956.

[25] D.E. Knuth. *Literate Programming*. CSLI Lecture Notes, no. 27. CSLI, Stanford, 1992.

[26] Leslie Lamport. The PlusCal algorithm language. In Martin Leucker and Carroll Morgan, editors, *Theoretical Aspects of Computing – ICTAC 2009*, number 5684 in Lecture Notes in Computer Science, pages 36–60. Springer, 2009.

[27] Hervé Moulin. *Fair Division and Collective Welfare*. MIT Press, 2004.

[28] Bryan O'Sullivan, John Goerzen, and Don Stewart. *Real World Haskell*. O'Reilly, 2009.

[29] T.V. Padma. *Mathematwist: Number Tales from Around the World*. Tulika Publishers, Chennai, India, 2007.

[30] R.C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[31] F. Rabhi and G. Lapalme. *Algorithms: a Functional Programming Approach*. Addison-Wesley, 1999.

[32] Jack Robertson and William Webb. *Cake-Cutting Algorithms – Be Fair if You Can*. A.K. Peters, 1998.

[33] Steven S. Skiena. *The Algorithm Design Manual*. Springer Verlag, New York, 1998. Second Printing.

[34] Raymond M. Smullyan. *The Lady or the Tiger?: and Other Logic Puzzles*. Dover, 2009. First edition: 1982.

[35] Raymond M. Smullyan. *What is the name of this book?* Dover, first edition 1990 edition, 2011.

[36] H. Steinhaus. The problem of fair division. *Econometrica*, 16:101–104, 1948.

[37] Walter Stromquist. How to cut a cake fairly. *American Mathematical Monthly*, 87(8):640–644, 1980.

[38] Francis Edward Su. Rental harmony: Sperner's lemma in fair division. *American Mathematical Monthly*, 106(10):930–942, 1999.

[39] Stephen Warshall. A theorem on Boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[40] Jin Y. Yen. An algorithm for finding shortest routes from all source nodes to a given destination in general networks. *Quarterly of Applied Mathematics*, 27:526–530, 1970.