CPSC 312
Functional and Logic Programming

October 29, 2015

Project 1 First Submissions

Hand-in will remain open until Monday midnight.

Anything submitted after Friday midnight will be considered later submission. After Monday, I will upload the solutions, so no more submissions will be accepted.

How to submit: Solutions to Question 3 and 4, in particular, work with codes in starter file. Put all your solutions into one .pl file for submission, but use comments to make it clear, if a part of your solution should be included in one of the starter files to work and which one.

Include any code that your program imports and uses as well. If you've changed any of the starter files, state that in your comments as well.

Project 1 First Submissions: Collaboration

Questions 3 and 4 are significantly more difficult than 1 and 2. If you're working on either one of them, make sure you get help from your teammates.

More on Haskell Text Editors

Sublime Text is actually a proprietary software...you can find alternatives here:
https://wiki.haskell.org/Editors

Lots of resources to be found at haskell.org:
https://www.haskell.org/documentation

Documentation for Built-in Functions (Prelude):
https://downloads.haskell.org/~ghc/latest/docs/html/libraries/base-4.8.1.0/Prelude.html

Review

Characteristics of FP: statelessness, no side effects and referential transparency.

A bit of history of Functional Programming

Lambda-calculus: lambda (λ) is an anonymous function with one argument. Lambda calculus is a formalism that can express all computations (Alonzo Church's theory).

Haskell programs much more concise: Quick Sort in Java vs Haskell

Functions as first-class values

```
function startAt(x)
   function incrementBy(y)
       return x + y
   return incrementBy

variable closure₁ = startAt(1)
variable closure₂ = startAt(5)
```

In functional programming, functions are treated as data and can be passed around as such. Closure (above), is an example of imperative languages borrowing from FP.

Interacting with GHCi

```
$ ghci
GHCi, version 7.10.2: 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> :load test
[1 of 1] Compiling Main
Ok, modules loaded: Main.
*Main> square 5
25
*Main> square 1024
1048576
*Main> :quit
Leaving GHCi
```

Anatomy of a Haskell function

$\texttt{square(x) = x}^2$

in math is the equivalent of this in Haskell:

```
square :: Int -> Int
square n = n ^ 2
```

Anatomy of a Haskell function

This is the type declaration for the function

```
square :: Int -> Int
square n = n ^ 2
```

Anatomy of a Haskell function

type of the formal parameter

name of function

type of the result to be returned

```
square :: Int -> Int
square n = n ^ 2
```

Anatomy of a Haskell function

```haskell
square :: Int -> Int
square n = n ^ 2
```

This is the actual function definition

Anatomy of a Haskell function

```
square :: Int -> Int
square n = n ^ 2
```

formal parameter

function name

result (function body defined in terms of formal parameters)

Anatomy of a Haskell function

What's a function definition with no parameters?

A constant:

```
mypi :: Float
mypi = 3.14159
```

Anatomy of a Haskell function

What if the function has more than one parameter?

```
multiply :: Int -> Int -> Int
multiply x y = x * y
```

Or

```
multiply :: Int -> Int -> Int -> Int

multiply x y z = x * y * z
```

Anatomy of a Haskell function

What if the function has more than one parameter?

```
multiply :: Int -> Int -> Int
multiply x y = x * y
```

notice that the type declaration uses the same sign (->) between the two arguments as between arguments and results.

(Instead of say: `Int, Int -> Int`)

Anatomy of a Haskell function

notice that the type declaration uses the same sign (->) between the arguments of a function as between arguments and the result. why? the reason is advanced information, only for the enthusiasts (i.e. not in the exam)

Anatomy of a Haskell function

notice that the type declaration uses the same sign (->) between the arguments of a function as between arguments and the result. why? the reason is advanced information, only for the enthusiasts (i.e. not in the exam)

recall that lambda is a function with a single argument, so a Haskell function with two args (or more) is, internally, a composition of two functions (or more) each with one argument.

Anatomy of a Haskell function

a Haskell function with two args (or more) is, internally, a composition of two functions with one argument:

```
multiply x y = x * y = (*) x y = (* x) y
```

function application is “left-associative” so when the type declaration says Int->Int->Int, the precedence for application of arguments is from left to right. With parantheses, this precedence can be emphasized:

```
multiply :: Int -> (Int -> Int)
multiply x y = (multiply x) y.
```

Basic data types

The Boolean data type in Haskell is called `Bool`
The Boolean values are `True` and `False`

The logical operators are

`&&`	and
`||`	or
`not`	not

Basic data types

The integer data type in Haskell is called `Int`
The range of values for the `Int` type is *at least* $[-2^{29} .. 2^{29}-1]$

The arithmetic operators include

`+`	addition
`-`	subtraction
`*`	multiplication
`/`	division
`^`	power
`div`	whole number division (prefix)
`mod`	remainder from whole number division (prefix)

Basic data types

The integer data type in Haskell is called `Int`
The range of values for the `Int` type is *at least* $[-2^{29} .. 2^{29}-1]$

The relational operators include

`>`	greater than
`>=`	greater than or equal to
`==`	equal to
`/=`	not equal to
`<=`	less than or equal to
`<`	less than

Basic data types

The integer data type in Haskell is called `Int`
The range of values for the `Int` type is *at least* $[-2^{29} .. 2^{29}-1]$

Arbitrarily large integers need the `Integer` data type

Basic data types

There's also

`Char`	character
`Float`	floating point
`Double`	floating point with more precision
`[Char]`	string

Read the book for more (Ch. 3)

Basic data types

There's no automatic conversion from `Int` to `Float` as there is in Java, for example. (That's a result of strong typing.) Use `fromIntegral` to convert from `Int` to `Float`.

```
Main> (floor 5.6) + 6.7
ERROR - Unresolved overloading
*** Type       : (Fractional a, Integral a) => a
*** Expression : floor 5.6 + 6.7

Main> fromIntegral(floor 5.6) + 6.7
11.7
```

Identifiers

Function names and variable (parameter) names begin with lower case letters. Type names (like `Int`) begin with upper case letters.

Identifiers

Function names and variable (parameter) names begin with lower case letters. Type names (like `Int`) begin with upper case letters.

The same identifier can be used to name a function and a variable. For example, this

```
foobar :: Int -> Int
foobar foobar = foobar * foobar

Main> foobar 3
9
```

works just fine. Never ever ever do this.

Haskell comments

```
-- precedes a one-line comment

{-   this is a block of
  comments     -}
```

Haskell comments

You could also program in the "literate style" where comments are the norm and have no special indicators. Instead, executable lines of code are preceded by >

```
Here's a comment in literate style
and below is the executable code

> square :: Int -> Int
> square n = n ^ 2
```

Names of literate Haskell scripts end with the `.lhs` suffix, not the `.hs` suffix.

Conditional Expression

If then else:

```
expensive p = if p>9000 then True else False
```

The else part is mandatory in Haskell.

Pattern matching (similar to Prolog):

```
qsort [] = []

qsort (p:xs) = (qsort lesser) ++ [p] ++ (qsort
greater) ...
```

Guards

A boolean expression used in a conditional expression in this format is called a guard. Here's an example:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
    | x >= y && x >= z     = x
    | y >= z               = y
    | otherwise            = z
```

Guards

A boolean expression used in a conditional expression in Haskell is called a guard. Here's an example:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
    | x >= y && x >= z      = x
    | y >= z                = y
    | otherwise             = z
```

If x >= y and x >= z

Guards

A boolean expression used in a conditional expression in Haskell is called a guard. Here's an example:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
    | x >= y && x >= z     = x
    | y >= z               = y
    | otherwise            = z
```

If x >= y and x >= z then substitute the expression x for the expression max3 x y z.

Guards

A boolean expression used in a conditional expression in Haskell is called a guard. Here's an example:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
    | x >= y && x >= z     = x
    | y >= z               = y
    | otherwise            = z
```

Else if y >= z

Guards

A boolean expression used in a conditional expression in Haskell is called a guard. Here's an example:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
    | x >= y && x >= z      = x
    | y >= z                = y
    | otherwise             = z
```

Else if y >= z then substitute the expression y for the expression max3 x y z.

Guards

A boolean expression used in a conditional expression in Haskell is called a guard. Here's an example:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
   | x >= y && x >= z     = x
   | y >= z               = y
   | otherwise            = z
```

Else

Guards

A boolean expression used in a conditional expression in Haskell is called a guard. Here's an example:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
    | x >= y && x >= z     = x
    | y >= z               = y
    | otherwise            = z
```

Else substitute the expression z for the expression max3 x y z.

Guards

A boolean expression used in a conditional expression in Haskell is called a guard. Here's an example:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
    | x >= y && x >= z      = x
    | y >= z                = y
    | otherwise             = z
```

The `otherwise` is not required but...

Guards

A boolean expression used in a conditional expression in Haskell is called a guard. Here's an example:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
    | x >= y && x >= z      = x
    | y >= z                = y
    | otherwise             = z
```

The `otherwise` is not required but...

- good programming style demands that you make explicit
  - what you intend when no boolean expression is True

Guards

A boolean expression used in a conditional expression in Haskell is called a guard. Here's an example:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
   | x >= y && x >= z     = x
   | y >= z               = y
   | otherwise            = z
```

The `otherwise` is not required but...

- good programming style demands that you make explicit
  - what you intend when no boolean expression is True
- Haskell blows up at run time when no boolean expression
  - is True...oops!

Guards

A boolean expression used in a conditional expression in Haskell is called a guard. Here's an example:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
   | x >= y && x >= z    = x
   | y >= z              = y
```

*Main> max3 1 2 3
*** Exception: tests.hs:(6,1)-(8,28): Non-exhaustive patterns in function max3

Guards

A boolean expression used in a conditional expression in Haskell is called a guard. Here's an example:

```
max3 :: Int -> Int -> Int -> Int
max3 x y z
   | x >= y && x >= z     = x
   | y >= z               = y
   | otherwise            = z
```

Why "guard"? The guard directs flow of control. Think of it as protecting the expression which follows from being evaluated unless specific conditions are met (i.e., the boolean expression between the | and the =).

Layout

A Haskell program, or script, is a series of definitions. When does one definition end and the next one begin?

It's based on indentation. A definition ends by the first piece of text which lies at the same indentation as or to the left of the start of the definition.

```
max2 :: Int -> Int -> Int
max2 x y
    | x >= y    = x
    | otherwise = y

max3 :: Int -> Int -> Int -> Int
max3 x y z
    | x >= y && x >= z     = x
    | y >= z               = y
    | otherwise            = z
```

Modules

A module is a collection of Haskell definitions in one file. Defining your program file as a module is a way of packaging the collections together for the purpose of reusing. Example:

```
module Square where

square :: Int -> Int

square n = n ^ 2
```

You can now import other modules (your own or library modules) into this module or export your own modules out of it. Any function that is to be imported into another module, has to be declared for export in its own module. (more on this later).

Questions?

Questions?

This covers Chapters 1 to 3 of your textbook. Make sure you have a read. It’s a very quick and easy read compared to the Prolog book (with lots of pictures!).

Functional design (Sections 4.1 and 4.2)

Our book points out a couple of the tenets of designing functional programs:

1. Can I break the problem down into simpler parts?
2. Can I reuse a function I've already defined?

Functional design

Our book points out a couple of the tenets of designing functional programs:

1. Can I break the problem down into simpler parts?
2. Can I reuse a function I've already defined?

The first point could be restated as:

"What if I had any functions I wanted: which ones could I use in writing the solution?"

We write the solution assuming the functions we want already exist, and we worry later about how they're actually defined.

Functional design

For instance, consider the problem of finding the roots of a quadratic, given by the legendary quadratic formula:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Functional design

For instance, consider the problem of finding the roots of a quadratic, given by the legendary quadratic formula:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

What functions would I use in writing this function if they already existed?

Functional design

For instance, consider the problem of finding the roots of a quadratic, given by the legendary quadratic formula:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

What functions would I use in writing this function if they already existed?
There are two roots, so I might define
my quadratic function in terms of largerRoot and smallerRoot functions that don't exist yet...I'll just assume that they'll be defined later.

Functional design

```
quadratic :: Float -> Float -> Float -> (Float,Float)
quadratic a b c = (largerRoot a b c, smallerRoot a b c)
```

Side note:

`(Float, Float)` -> this is a floating-point type tuple in Haskell.

```
fst:: (a,b) -> a --retieve the first item
snd:: (a,b) -> b --retrieve the second item
```

are built-in functions that work with tuples

Functional design

For instance, consider the problem of finding the roots of a quadratic, given by the legendary quadratic formula:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

I can define the largerRoot and smallerRoot functions in terms of their numerators, which I'll call largerNumerator and smallerNumerator, and their denominator which is the same in both cases.

Functional design

```
quadratic :: Float -> Float -> Float -> (Float,Float)
quadratic a b c = (largerRoot a b c, smallerRoot a b c)

largerRoot :: Float -> Float -> Float -> Float
largerRoot a b c = (largerNumerator a b c) /
denominator a

smallerRoot :: Float -> Float -> Float -> Float
smallerRoot a b c = (smallerNumerator a b c) /
denominator a
```

Functional design

For instance, consider the problem of finding the roots of a quadratic, given by the legendary quadratic formula:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Both the numerators have some serious computation happening under the square root sign. The result of that computation is called the discriminant, so a function that computes the square root of the discriminant might be called sqrtOfDiscriminant. So the largerNumerator function could be defined like this:

Functional design

```
quadratic :: Float -> Float -> Float -> (Float,Float)
quadratic a b c = (largerRoot a b c, smallerRoot a b c)

largerRoot :: Float -> Float -> Float -> Float
largerRoot a b c = (largerNumerator a b c) /
denominator a

smallerRoot :: Float -> Float -> Float -> Float
smallerRoot a b c = (smallerNumerator a b c) /
denominator a

largerNumerator :: Float -> Float -> Float -> Float
largerNumerator a b c = -b + sqrtOfDiscriminant a b c
```

Functional design

```
quadratic :: Float -> Float -> Float -> (Float,Float)
quadratic a b c = (largerRoot a b c, smallerRoot a b c)

largerRoot :: Float -> Float -> Float -> Float
largerRoot a b c = (largerNumerator a b c) /
denominator a

smallerRoot :: Float -> Float -> Float -> Float
smallerRoot a b c = (smallerNumerator a b c) /
denominator a

largerNumerator :: Float -> Float -> Float -> Float
largerNumerator a b c = -b + sqrtOfDiscriminant a b c

smallerNumerator :: Float -> Float -> Float -> Float
smallerNumerator a b c = -b - sqrtOfDiscriminant a b c
```

Functional design

For instance, consider the problem of finding the roots of a quadratic, given by the legendary quadratic formula:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The sqrtOfDiscriminant function can be written with no newly-defined functions...it's all just arithmetic.

Functional design

```
quadratic :: Float -> Float -> Float -> (Float,Float)
quadratic a b c = (largerRoot a b c, smallerRoot a b c)

largerRoot :: Float -> Float -> Float -> Float
largerRoot a b c = (largerNumerator a b c) / denominator a

smallerRoot :: Float -> Float -> Float -> Float
smallerRoot a b c = (smallerNumerator a b c) / denominator a

largerNumerator :: Float -> Float -> Float -> Float
largerNumerator a b c = -b + sqrtOfDiscriminant a b c

smallerNumerator :: Float -> Float -> Float -> Float
smallerNumerator a b c = -b - sqrtOfDiscriminant a b c

sqrtOfDiscriminant :: Float -> Float -> Float -> Float
sqrtOfDiscriminant a b c = sqrt ((b ^ 2) - (4 * a * c))
```

Functional design

For instance, consider the problem of finding the roots of a quadratic, given by the legendary quadratic formula:

$$\frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

And the denominator function is just arithmetic too.

Functional design

```
quadratic :: Float -> Float -> Float -> (Float,Float)
quadratic a b c = (largerRoot a b c, smallerRoot a b c)

largerRoot :: Float -> Float -> Float -> Float
largerRoot a b c = (largerNumerator a b c) / denominator a

smallerRoot :: Float -> Float -> Float -> Float
smallerRoot a b c = (smallerNumerator a b c) / denominator a

largerNumerator :: Float -> Float -> Float -> Float
largerNumerator a b c = -b + sqrtOfDiscriminant a b c

smallerNumerator :: Float -> Float -> Float -> Float
smallerNumerator a b c = -b - sqrtOfDiscriminant a b c

sqrtOfDiscriminant :: Float -> Float -> Float -> Float
sqrtOfDiscriminant a b c = sqrt ((b ^ 2) - (4 * a * c))

denominator :: Float -> Float
denominator a = 2 * a
```

Functional design

That's a lot of functions to just compute the two roots of a quadratic. But they are produced out of the "what functions could I use if they already existed?" approach to problem solving. If you've heard of abstraction, this may seem familiar.

Functional design

That's a lot of functions to just compute the two roots of a quadratic. But they just sort of fall out of the "what functions could I use if they already existed?" approach to problem solving. If you've heard of abstraction, this may seem familiar.

abstraction: treating something complex as if it were simpler, giving that simple thing a name, and throwing away (or postponing) the details.

Functional design

"The most important concept in all of computer science is abstraction. Computer science deals with information and complexity. We make complexity manageable by judiciously reducing it when and where possible.

"I regret that I cannot recall who remarked that computation is the art of carefully throwing away information: given an overwhelming collection of data, you reduce it to a usable result by discarding most of its content."

Guy Steele

Is this extreme? Maybe...

```
quadratic :: Float -> Float -> Float -> (Float,Float)
quadratic a b c = (largerRoot a b c, smallerRoot a b c)

largerRoot :: Float -> Float -> Float -> Float
largerRoot a b c = (largerNumerator a b c) / denominator a

smallerRoot :: Float -> Float -> Float -> Float
smallerRoot a b c = (smallerNumerator a b c) / denominator a

largerNumerator :: Float -> Float -> Float -> Float
largerNumerator a b c = -b + sqrtOfDiscriminant a b c

smallerNumerator :: Float -> Float -> Float -> Float
smallerNumerator a b c = -b - sqrtOfDiscriminant a b c

sqrtOfDiscriminant :: Float -> Float -> Float -> Float
sqrtOfDiscriminant a b c = sqrt ((b ^ 2) - (4 * a * c))

denominator :: Float -> Float
denominator a = 2 * a
```

...but here's the other extreme

```
badquad :: Float -> Float -> Float -> (Float,Float)
badquad a b c =
  ( (-b + sqrt (b ^ 2 - 4 * a * c)) / (2 * a),
    (-b - sqrt (b ^ 2 - 4 * a * c)) / (2 * a) )
```

This version is succinct but bugs might be harder to track down. If we continue with this monolithic style as programs get bigger, this could get downright ugly.

Is this better?

```
betterquad :: Float -> Float -> Float -> (Float,Float)
betterquad a b c =
           ( (-b + sqrtOfDiscriminant a b c) / denominator a,
             (-b - sqrtOfDiscriminant a b c) / denominator a  )


sqrtOfDiscriminant :: Float -> Float -> Float -> Float
sqrtOfDiscriminant a b c = sqrt ((b ^ 2) - (4 * a * c))


denominator :: Float -> Float
denominator a = 2 * a
```

Is this better?

```
betterquad1 :: Float -> Float -> Float -> (Float,Float)
betterquad1 a b c =
   betterquad2 a b c (sqrtOfDiscriminant a b c) (denominator a)


betterquad2 :: Float -> Float -> Float -> Float -> Float ->
               (Float,Float)
betterquad2 a b c sod denom = ( (-b + sod) / denom,
                                (-b - sod) / denom )


sqrtOfDiscriminant :: Float -> Float -> Float -> Float
sqrtOfDiscriminant a b c = sqrt ((b ^ 2) - (4 * a * c))


denominator :: Float -> Float
denominator a = 2 * a
```

Is this better?

```
betterquad3 :: Float -> Float -> Float -> (Float,Float)
betterquad3 a b c =
   let sod = sqrtOfDiscriminant a b c; denom = 2 * a in
                              ( (-b + sod) / denom,
                                (-b - sod) / denom )


sqrtOfDiscriminant :: Float -> Float -> Float -> Float
sqrtOfDiscriminant a b c = sqrt ((b ^ 2) - (4 * a * c))
```

Is this better?

```
betterquad3 :: Float -> Float -> Float -> (Float,Float)
betterquad3 a b c =
   let sod = sqrtOfDiscriminant a b c; denom = 2 * a in
                              ( (-b + sod) / denom,
                                (-b - sod) / denom )
```

```
sqrtOfDiscriminant :: Float -> Float -> Float -> Float
sqrtOfDiscriminant a b c = sqrt ((b ^ 2) - (4 * a * c))
```

If you continue with functional programming, you'll develop your own style. But don't let it be this:

```
badquad :: Float -> Float -> Float -> (Float,Float)
badquad a b c =
  ( (-b + sqrt (b ^ 2 - 4 * a * c)) / (2 * a),
    (-b - sqrt (b ^ 2 - 4 * a * c)) / (2 * a) )
```

Substitution model of evaluation

The behaviour of pure functional programs can be examined easily through the substitution model of evaluation.

The substitution model of evaluation simply says that when the interpreter is given a function name with arguments to evaluate:

```
> sqrtOfDiscriminant 1 (-5) 6
```

Substitution model of evaluation

The behaviour of pure functional programs can be examined easily through the substitution model of evaluation.

The substitution model of evaluation simply says that when the interpreter is given a function name with arguments to evaluate:

```
> sqrtOfDiscriminant 1 (-5) 6
```

it does so by retrieving the function body

```
sqrtOfDiscriminant a b c = sqrt ((b ^ 2) - (4 * a * c))
```

Substitution model of evaluation

The behaviour of pure functional programs can be examined easily through the substitution model of evaluation.

The substitution model of evaluation simply says that when the interpreter is given a function name with arguments to evaluate:

```
> sqrtOfDiscriminant 1 (-5) 6
```

making the appropriate substitutions

```
sqrtOfDiscriminant 1 (-5) 6 = sqrt (((-5) ^ 2) - (4 * 1 * 6))
```

Substitution model of evaluation

The behaviour of pure functional programs can be examined easily through the substitution model of evaluation.

The substitution model of evaluation simply says that when the interpreter is given a function name with arguments to evaluate:

```
> sqrt (((-5) ^ 2) - (4 * 1 * 6))
```

making the appropriate substitutions

```
sqrtOfDiscriminant 1 (-5) 6 = sqrt (((-5) ^ 2) - (4 * 1 * 6))
```

and then replacing the original function invocation to be evaluated with this new one

Substitution model of evaluation

The behaviour of pure functional programs can be examined easily through the substitution model of evaluation.

The substitution model of evaluation simply says that when the interpreter is given a function name with arguments to evaluate:

```
> sqrt (((-5) ^ 2) - (4 * 1 * 6))
```

making the appropriate substitutions

```
sqrtOfDiscriminant 1 (-5) 6 = sqrt (((-5) ^ 2) - (4 * 1 * 6))
```

It's just referential transparency in action.

Questions?

We’ve covered most of chapter 1 to 3 + 4.1 and 4.2 from your textbook.

Next week, we’ll talk about lists and recursion. (Ch. 4,5,6)