CSE399: Advanced Programming

Handout 1

What is Advanced Programming?

Good programmers...

- write code that gets the job done

Excellent programmers...

- write code that other people can read, understand, maintain, and modify
- rewrite code to make it clear and elegant
- design with abstractions
- design for reuse
- etc.

This course is about developing these skills.

(Rough) Plan For The Semester

- Functional programming
  - Black-belt Haskell
  - Many small-scale case studies (on efficient functional data structures, interesting combinator libraries, etc., etc.)
  - Concurrent / parallel programming (with both locks and transactional memory)
  - Maybe some OCaml and/or F# (especially for the module system)
  - Transplanting ideas from Haskell/OCaml to Java/C#
- Programming in general
  - Modular decomposition
  - Testing
  - Strategies for performance tuning
  - etc.
- Major projects (in a functional language of your choice)

This plan is intentionally fluid and open-ended. Let me know if there is something you particularly want to see covered.

What This Course is Not

CSE350, Software Engineering, takes a different perpsective on “advanced programming.”

- Software Engineering focuses on programming in the large — the special problems that arise in software projects that are too large for one person
  Specific topics include:
  - Software lifecycle models
  - Project management
  - Design modeling notations (e.g., UML)
  - Formal specification
  - etc.
- Naturally there is some overlap, but the present course is focused on the skills that make excellent individual programmers.

The courses are designed so that you can productively take both, if you want.

Logistics

Basics

Web page:	`http://www.cis.upenn.edu/` `~bcpierce/courses/advprog`
Mailing list:	TBA
Textbooks:	selections from The Haskell School of Expression, by Paul Hudak (hereafter SOE) others to be decided later

Personnel

Instructor:	Benjamin Pierce Levine 562 bcpierce at cis.upenn.edu Office hours: TBA
Teaching Assistant:	Joey Schorr jschorr at seas.upenn.edu Office hours: TBA

Administrative Assistant: Kamila Dyjas Mauro, Levine 311

Grading

Course grades will be based on:

- Class participation [20%]
- Weekly programming assignments [40%]
- Final project [40%]

Assignments will be accepted late, but will lose 5% of their value for each day (or partial day) after the due date.

There will be reading assignments following most lectures. “Class participation” means (among other things) staying on top of these.

First Homework Assignment

- The first homework assignment (on basic Haskell programming) is due at 2:00 on Wednesday afternoon.
- You will need:
    - a machine where Haskell is installed, such as the Linux lab machines in Moore 100 (you can also install Haskell on your own machine if you like — see the Resources page of the course web site)
    - Chapters 1 and 2 of SOE as background reading

On to Haskell

Programming Languages Don’t Matter

- Languages come and go.
- You have probably used several by now (Java, C, assembly language, HTML, PHP, JavaScript, etc.); in the course of your career, you will use many more.
- The hardest aspects of software construction — understanding requirements, specification, design, etc. — are largely independent of the language(s) being used.

Programming Languages Matter

- However, a programming language is a power tool.
- Many details must be mastered in order to use it effectively.
- Moreover, each language embodies some style of programming — some particular view of how software should be constructed. To use it well, you have to be able to get into this mindset.

  Trying to use a language "against the grain" tends to lead to abominations.

Functional Programming

Haskell is a functional programming language — i.e., one in which the functional programming style is the dominant idiom. Other well-known functional languages include Lisp, Scheme, OCaml, and Standard ML. The functional style can be described as a combination of...

- persistent data structures (which, once built, are never changed)
- recursion as a primary control structure
- heavy use of higher-order functions (functions that take functions as arguments and/or return functions as results)

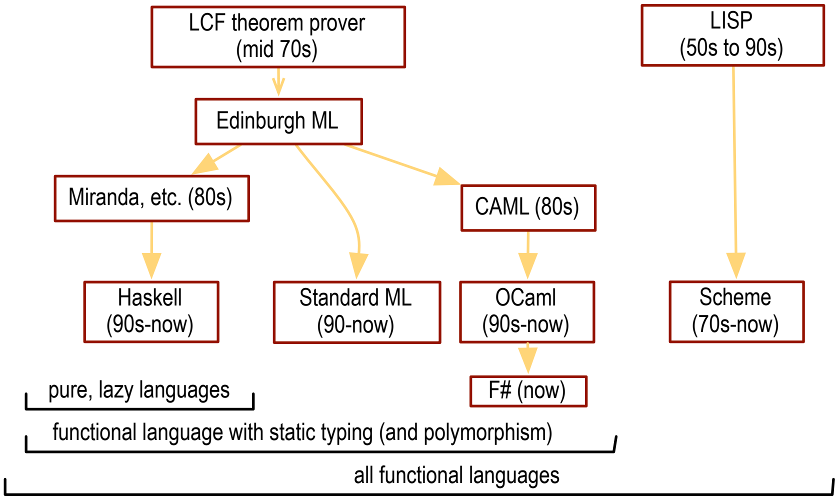
Imperative languages, by contrast, emphasize

- mutable data structures
- looping rather than recursion
- first-order programming (though many object-oriented “design patterns” involve higher-order idioms—e.g., Subscribe/Notify, Visitor, etc.)

Applications of Functional Languages

- compilers, theorem provers, and other symbol processing applications
- web programming (e.g., Yahoo shopping engine)
- systems programming (e.g., Unison file synchronizer, MLDonkey p2p client)
- high-assurance crypto software (e.g., Galois)
- military and manufacturing planning, financial decision making (e.g., Aetion)
- etc.

(Simpilified!) Functional Language Geneology



Haskell vs. OCaml

- Both Haskell and OCaml were strong contenders for this course
- Both have
  - rich type systems based on parametric polymorphism
  - type inference
  - powerful algebraic pattern matching
  - excellent support for declarative and higher-order programming
- OCaml has...
  - probably the world's best module language
  - support for object-oriented programming
- Haskell has...
  - type classes
  - monads for controlling computational effects
  - lazy evaluation
- In the end, Haskell won (by a hair) for the first part of the course, as being a little more mind-expanding

Haskell Implementations

- There are several implementations of Haskell
  - The Glasgow Haskell Compiler (GHC)
  - The Hugs interpreter
  - etc.
- For this course, we will use GHC.
- GHC is installed on the Linux systems in Moore 100.
- If you want to use your own computer for assignments, you'll need to install GHC, plus a couple of additional libraries. See the homework assignment for more information.

Programming in Haskell

Computation by Calculation

- Computation by calculation is a simple, familiar concept: replacement of equals by equals according to some set of given laws:

```
   3 * (4 + 5)
⇒ { add }
   3 * 9
⇒ { multiply }
   27
```

- This sort of thing can be found in pretty much every language. What is unusual about Haskell is that computation by calculation is all you need to know, to understand everything about how programs behave.

Abstraction

- We are also interested in abstraction: the process of recognizing a repeating pattern and capturing it succinctly in one place instead of many.
- For example, the repeating pattern in

  ```
  3*(4+5)     9*(1+3)     333*(123+567)
  ```

  can be captured as a function

  ```
  easy x y z = x * (y + z)
  ```

  and specific instances can be written as:

  ```
  easy 3 4 5    easy 9 1 3    easy 333 123 567
  ```

Abstraction

“Computation by calculation” is extended to deal with function abstractions by unfolding (and sometimes folding) definitions:

```
   easy 3 4 5
⇒ { unfold }
   3 * (4 + 5)
⇒ { add }
   3 * 9
⇒ { multiply }
   27
```

Symbolic Calculation

In addition to computing with concrete quantities, we can use computation by calculation to reason symbolically:

```
  easy a b c
⇒ { unfold }
  a * (b + c)
⇒ { commutativity of + }
  a + (c + b)
⇒ { fold }
  easy a c b
```

Expressions, Values, and Types

- The phrases on which we calculate are called expressions.
- When no more unfolding of user-defined functions or application of primitives like `+` is possible, the resulting expression is called a value.
- A type is a collection of expressions with common attributes. Every expression (and thus every value) belongs to a type.
- We write `exp :: T` to say that expression `exp` has type `T`.

Basic Types

Haskell offers the usual assortment of primitive types and associated operators:

- Integers:
  ```
  3 + 4 * 5 :: Integer
  ```
- Floats:
  ```
  3 + 4.5 * 5.5 :: Float
  ```
- Characters:
  ```
  'a' :: Char
  ```

Note that the `+` and `*` operators are overloaded. We will have (much) more to say about this later.

Functions

The type of a function taking arguments of types `A` and `B` and returning a result of type `C` is written `A->B->C`.

```
(+) :: Integer -> Integer -> Integer
easy :: Integer -> Integer -> Integer -> Integer
```

Note that `(+)` is syntax for treating a symbolic (infix) name as a regular one. Conversely, an alphabetic name enclosed in backquotes can be used in infix position. E.g., if we define

```
    plus x y = x + y
```

then `` 6 `plus` 7 `` is a valid expression.

Tuples

A tuple is a sequence of values enclosed in parens and separated by commas.

```
('b',4) :: (Char,Integer)
```

The types of tuples are written according to the same convention.

Tuples are destructed by pattern matching:

```
easytoo :: (Integer,Integer,Integer) -> Integer
easytoo (x,y,z) = x+y*z
```

Lists

One very handy structure for storing collections of data values is a list. Lists are provided as a built-in type in Haskell and a number of other popular languages (e.g., OCaml, Scheme, and Prolog—but not, unfortunately, Java or C#).

We can build a list in Haskell by writing out its elements, enclosed in square brackets and separated by commas.

```
[1, 3, 2, 5] :: [Integer]
```

The empty list, written `[]`, is also pronounced “nil.”

The Types of Lists

We can build lists whose elements are drawn from any of the basic types (`Integer`, `Char`, etc.).

```
[1, 2, 3] :: [Integer]
['a', 'b', 'c'] :: [Char]
```

In fact, `String` is just a synonym for `[Char]`.

We can also build lists of lists:

```
[[1, 2], [2, 3, 4], [5]] :: [[Integer]]
```

Indeed, for every type `T`, we can build lists of type `[T]`.

Lists are Homogeneous

Haskell does not allow different types of elements to be mixed within the same list:

```
[1, 2, 'c']

No instance for (Num Char)
  arising from the literal ‘1’ at <interactive>:1
In the list element: 1
In the definition of ‘it’: it = [1, 2, 'c']
```

(Promise: This sort of error message will make sense in a few weeks!)

Constructing Lists

Haskell provides a number of built-in operations that return lists. The most basic one creates a new list by adding an element to the front of an existing list. It is written `:` and pronounced “cons” (because it constructs lists).

```
1 : [2, 3] :: [Ingeger]

add123 :: [Integer] -> [Integer]
add123 l = 1 : 2 : 3 : l
```

```
  add123 [5, 6, 7]
⇒ [1, 2, 3, 5, 6, 7]

add123 []
⇒ [1, 2, 3]
```

Constructing Lists

In fact, the square bracket syntax for lists is just syntactic sugar for expressions using cons and nil:

$$\texttt{[}\ x_1,\ x_2,\ \ldots,\ x_n\ \texttt{]}$$

is simply a shorthand for

$$x_1\ \texttt{:}\ x_2\ \texttt{:}\ \ldots\ \texttt{:}\ x_n\ \texttt{:}\ \texttt{[]}$$

Some Recursive Functions that Generate Lists

```
copies :: Integer -> Integer -> [Integer]
copies k n =    -- A list of n copies of k
  if n == 0 then []
  else k : copies k (n-1)

fromTo :: Integer -> Integer -> [Integer]
fromTo m n =    -- A list of the numbers from m to n
  if n < m then []
  else m : fromTo (m+1) n
```

```
  copies 7 12
⇒ [7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7, 7]

fromTo 9 18
⇒ [9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
```

Destructing Lists

Like tuples, lists can be destructed by pattern matching in function headers.

```
listSum :: [Integer] -> Integer
listSum [] = 0
listSum (x:xs) = x + listSum xs
```

We have split the definition of `listSum` into two separate clauses, one for empty lists and the other for lists with at least one element.

The Interactive Top Level

The GHC system can be used in two modes:

- The batch compiler, invoked by the command `ghc`, is the standard way of compiling and executing larger programs. We'll come back to it later.
- The interactive top level, invoked by the command `ghci`, is a convenient way of experimenting with small programs.

The mode of interacting with the top level is typing in a series of expressions; `ghci` evaluates them as they are typed and displays the results.

When run with a filename as a command-line argument, `ghci` loads this file and makes its definitions available for interactive testing.

GHCi Example

Suppose file `foo.hs` contains the following:

```
module Foo where

foo x y = x + y*3

bar x = foo x 7
```

Then...

GHCi Example

```
/mnt/castor/seas_home/b/bcpierce/tmp> ghci foo.hs

GHCi, version 6.8.2: http://www.haskell.org/ghc/  :? for help
Loading package base ... linking ... done.
[1 of 1] Compiling Foo              ( foo.hs, interpreted )
Ok, modules loaded: Foo.
*Foo> bar 6
27
*Foo> foo 4 6
22
*Foo> :quit
Leaving GHCi.
```

Local Definitions

Names need not be declared at the top level, if their use is confined to a local scope:

```
foo1 z =
  let triple x = x*3
  in triple z
```

Or:

```
foo2 z = triple z
         where triple x = x*3
```

Type Synonyms

It is often convenient (and a useful aid to reading) to give names to compound types:

```
    type Coordinate = (Float,Float)
```

Using type names introduced in this way does not change the meaning of programs: writing `Coordinate` is exactly the same as writing `(Float,Float)`.

Data Types

Haskell also provides another — much more powerful — facility for defining new types, called “data types.”

As a motivating example, suppose we are building a (very simple) graphics program that displays circles and squares. We can represent each of these with three real numbers...

Data Types

A circle is represented by the co-ordinates of its center and its radius. A square is represented by the co-ordinates of its bottom left corner and its width. So we can represent both shapes as elements of the type:

```
type CircleType = (Float, Float, Float)
type SquareType = (Float, Float, Float)
```

However, because their types are identical, there is nothing to prevent us from mixing circles and squares. For example, if we write

```
areaOfSquare (_,_,d) = d * d
```

we might accidentally apply the `areaOfSquare` function to a circle and get a nonsensical result.

Data Types

We can improve matters by defining `Circle` and `Square` as data types:

```
data CircleDataType = Circle (Float,Float,Float)
data SquareDataType = Square (Float,Float,Float)
```

These declarations do two things:

- They create new types called `CircleDataType` and `SquareDataType` that are different from any other type in the system.
- They create constructors called `Circle` and `Square` that can be used to create a circle or square from three floats.

For example:

```
mySquare :: SquareDataType
mySquare = Square (1.1, 2.2, 3.3)
```

Deconstructing Data Types

We take instances of data types apart with (surprise, surprise...) pattern matching.

```
areaOfSquare1 :: SquareDataType -> Float
areaOfSquare1 (Square (_,_,d)) = d*d
```

I.e., the `Square` constructor can be used both as a function and as a pattern.

Note that, since `SquareDataType` and `CircleDataType` are different types, there is no danger of applying `areaOfSquare` to a circle.

Variant Types

Going back to the idea of a graphics program, we obviously want to have several shapes on the screen at once. For this we'd probably need a list of circles and squares, but such a list would be heterogenous. How do we make such a list?

The solution is to declare a data type whose instances can be either circles or squares.

```
data CircleOrSquare =
    Circle (Float,Float,Float)
  | Square (Float,Float,Float)

mySquare :: CircleOrSquare
mySquare = Square (1.1, 2.2, 3.3)

myCircle :: CircleOrSquare
myCircle = Circle (0, 0, 10)
```

Variant Types

A data type that can have more than one form is often called a variant type.

We can also write functions that do the right thing on all forms of a variant type. Again we use pattern matching:

```haskell
area :: CircleOrSquare -> Float
area (Circle (_,_,r)) = pi * r * r
area (Square (_,_,d)) = d * d
```

Another Datatype of Shapes

Chapter 3 of SOE develops a more sophisticated data type of geometric shapes.

```
data Shape =
    Rectangle Float Float
  | Ellipse Float Float
  | RtTriangle Float Float
  | Polygon [(Float,Float)]
```

(Quick check: why is the pair of floats enclosed in parens in one place and not the other?)

Another Datatype of Shapes

We can make this a bit more readable with a few auxiliary type synonyms.

```
data Shape =
    Rectangle Side Side
  | Ellipse Radius Radius
  | RtTriangle Side Side
  | Polygon [Vertex]

type Side = Float
type Radius = Float
type Vertex = (Float,Float)
```

Calculating Areas

```
area :: Shape -> Float

area (Rectangle s1 s2)  = s1*s2
```

Calculating Areas

```
area (RtTriangle s1 s2) = s1*s2/2

area (Ellipse r1 r2)  = pi*r1*r2
```

Calculating Areas

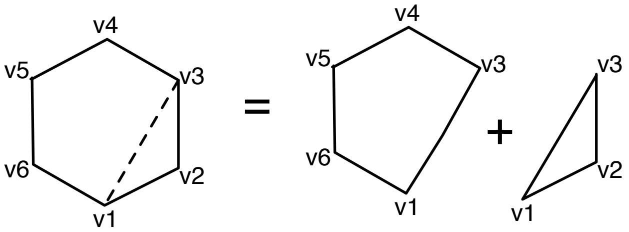
What about polygons?

Calculating Areas

What about polygons?

The area of a convex polygon can be calculated as follows:

- Compute the area of the triangle formed by the first three vertices.
- Delete the second vertex to form a new polygon.
- Sum the area of this new polygon and the area of the triangle from the first step.



Calculating Areas

In Haskell:

```
area (Polygon (v1:v2:v3:vs))
  = triArea v1 v2 v3 + area (Polygon (v1:v3:vs))
area (Polygon _)
  = 0
```

(`triArea`, which calculates the area of a triangle, is defined in the book.)

Polygon Areas Again

Digression...

SOE goes on to reformulate the polygon area code as follows:

```
area (Polygon (v1:vs))  = polyArea vs
  where polyArea               :: [Vertex] -> Float
        polyArea (v2:v3:vs') = triArea v1 v2 v3
                                  + polyArea (v3:vs')
        polyArea _           = 0
```

Is this an improvement?

Polygon Areas Again

Optimizations:

- Avoids reconstructing a `Polygon` on each recursive call.
- Performs only one cons instead of two on each recursive call (by holding on to `v1` in a local variable)

Polygon Areas Again

Optimizations:

- Avoids reconstructing a `Polygon` on each recursive call.
- Performs only one cons instead of two on each recursive call (by holding on to `v1` in a local variable)

Pessimizations:

- Somewhat harder to read, write, and understand

Polygon Areas Again

Optimizations:

- Avoids reconstructing a `Polygon` on each recursive call.
- Performs only one cons instead of two on each recursive call (by holding on to `v1` in a local variable)

Pessimizations:

- Somewhat harder to read, write, and understand

I.e., the new version trades a small decrease in readability for a small increase in efficiency.

Polygon Areas Again

Optimizations:

- Avoids reconstructing a `Polygon` on each recursive call.
- Performs only one cons instead of two on each recursive call (by holding on to `v1` in a local variable)

Pessimizations:

- Somewhat harder to read, write, and understand

I.e., the new version trades a small decrease in readability for a small increase in efficiency.

This is a bad trade!!

Premature Optimization

- In most programs, the vast majority of the time is spent in a tiny portion of the code.
- Programmers (even good ones) are notoriously bad at predicting which portion.
- Machines are real fast and getting faster.
- Programmers (even excellent ones) are not getting faster.

Premature Optimization

- In most programs, the vast majority of the time is spent in a tiny portion of the code.
- Programmers (even good ones) are notoriously bad at predicting which portion.
- Machines are real fast and getting faster.
- Programmers (even excellent ones) are not getting faster.

Moral:

- Write code that is manifestly correct
- Measure performance
- Tune bottlenecks as needed.

(flame off)

Input and Output Actions

I/O in Haskell

Q: Of course, most programs don’t just calculate values: they have effects on the world — displaying text or graphics, reading or writing the file system and network...

How does this square with Haskell’s value-oriented, calculational style of computation?

I/O in Haskell

A: Haskell provides a special kind of value, called an action, that describes an effect on the world.

Pure actions, which just do something and have no interesting "result," are values of type `IO ()`.

For example, the `putStr` function takes a string and yields an action describing the act of displaying this string on `stdout`.

```
    putString :: String -> IO ()
```

I/O in Haskell

To actually perform an action, we make it the value of the special name `main`.

```
    main :: IO ()
    main = putStr "Hello world\n"
```

I/O in Haskell

```
/mnt/castor/seas_home/b/bcpierce/tmp> ghci hello.hs

GHCi, version 6.8.2: http://www.haskell.org/ghc/  :? for help
Loading package base ... linking ... done.
[1 of 1] Compiling Main             ( hello.hs, interpreted )
Ok, modules loaded: Main.
*Main> main
Hello world
```

The Batch Compiler

The command

```
ghc -o file file.hs
```

will compile the file `file.hs` into a binary executable (named `file`) that, when executed, will perform the action bound to the top-level name `main`.

```
/mnt/castor/seas_home/b/bcpierce/tmp> ghc -o hello hello.hs
/mnt/castor/seas_home/b/bcpierce/tmp> ./hello
Hello world
```

Actions

Actions are descriptions of effects on the world. Simply writing an action does not, by itself, cause anything to happen.

```
hellos :: [IO ()]
hellos = [putStr "Hello somebody\n",
          putStr "Hello world\n",
          putStr "Hello universe\n"]
main = head (tail hellos)
```

I/O in Haskell

```
/mnt/castor/seas_home/b/bcpierce/tmp> ghc -o hellos hellos.hs
/mnt/castor/seas_home/b/bcpierce/tmp> ./hellos
Hello world
```

Combining Actions

The infix operator `>>` takes two actions `a` and `b` and yields an action that describes the effect of executing `a` and `b` in sequence.

```haskell
hello1 :: IO ()
hello1 = putStr "hello " >> putStr "world\n"
```

"Do" Notation

To avoid writing `>>` all the time, Haskell provides special syntax for sequencing actions:

```
hello2 = do putStr "hello "
            putStr "world\n"
```

In general, if `act1`, `act2`, ..., `actn` are actions, then

```
do act1
   act2
   ...
   actn
```

is an action that represents performing them in sequence.

Note the use of the "layout convention" here: the first action begins right after the `do` and the others are laid out vertically beneath it.

Input Actions

Some actions have an effect on the world and yield a result.

For example,

```
    getLine :: IO String
```

is an action that, when executed, consumes the next line from the standard input and returns it.

“Do” Notation for Input Actions

The `do` syntax provides a way to bind the result of an action to a variable so that it can be referred to later.

```
main =
  do putStr "Please type a line...\n"
     s <- getLine
     putStr "You typed '"
     putStr s
     putStr "'\n"
```