

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.

What This Course is Not

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

- CSE350 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.
- The present course is mostly about the skills that make excellent **individual** programmers.

Naturally, there is some overlap. However, the courses are designed so that you can productively take both, if you want. (Plans are to offer CSE350 in Spring '06.)

Logistics

Web page: `http://www.cis.upenn.edu/
~bcpierce/courses/advprog`

Newsgroup: `upenn.cis.cse399`

Textbook: The Haskell School of Expression,
by Paul Hudak
(hereafter SOE)

Note: The newsgroup is intended for announcements and for discussions among class participants; we will not be monitoring it continuously.

Instructor: Benjamin Pierce
Levine 304
`bcpierce` at `cis.upenn.edu`
Office hours: TBA

Guest instructors: Fernando Pereira (Mon, Wed this week)
Brian Aydemir (Fri this week)

Teaching Assistant: Dimitrios Vytiniotis
Levine 565
`dimitriv` at `cis.upenn.edu`
Office hours: TBA

Administrative Assistant: Jennifer Finley, Levine 308

Both Benjamin Pierce and Dimitris Vytiniotis are out of town this week (presenting papers at a programming languages conference).

We've arranged an exciting lineup of guest lecturers, who should be able to answer most of your questions about the material.

We can also both be reached by email.

Course grades will be based on:

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

Assignments will be accepted late, but will lose 10% 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 this Sunday.
- 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
- Read the Haskell Style Guide on the course web page. The rules listed on that page will be strictly enforced when homework sets and projects are graded. (Exception: no testing code is needed for the first assignment.)
- Please read Chapter 1 of SOE before Wednesday's lecture.

Rough Course Outline

- Haskell (first 200 or so pages of SOE), interleaved with material on...
 - Efficient functional data structures
 - XML transformation
 - Modular design
 - Unit testing
 - Performance tuning
 - Debugging
- Larger case study:
 - Web server design
- Concurrency (time allowing)
- Transplanting ideas from Haskell to Java/C#

N.b.: This is the first time this course is being offered, so there are sure to be **many** revisions to the plans!

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)

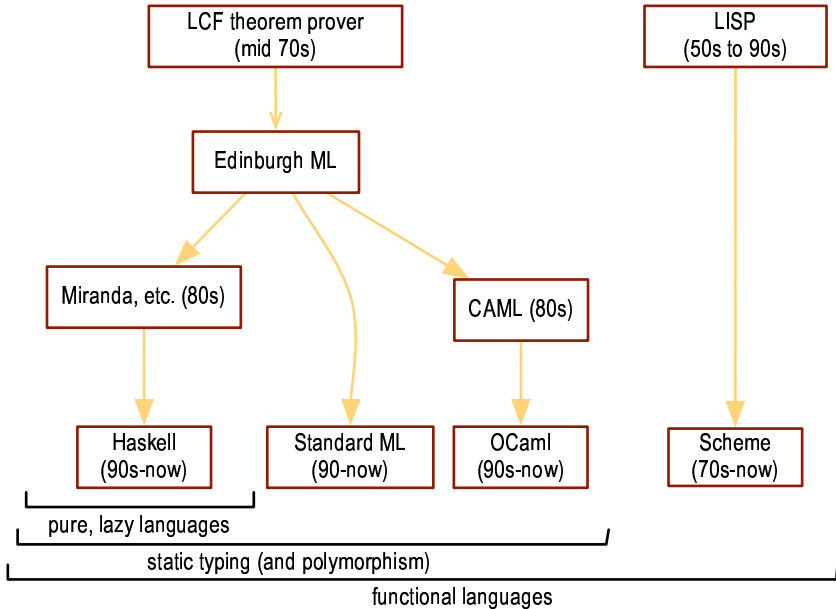
Imperative languages, by contrast, emphasize

- **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.

Haskell Genealogy



- 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
 - fully lazy evaluation
- In the end, Haskell won (by a hair) as being a little more mind-expanding

Haskell Implementations

- There are two main implementations of Haskell:
 - The **Glasgow Haskell Compiler** (GHC)
 - The **Hugs** interpreter
- For most of the course, we will use Hugs. Some assignments (in particular those involving graphics) **require** Hugs.
- Hugs is installed on the Linux systems in Moore 100.
- If you want to use your own computer for assignments, you'll need to install Hugs, plus a couple of additional libraries. See the **Resources** page under the main course web page for instructions.

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.

- 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

$\text{easy } x \ y \ z = x * (y + z)$

and specific instances can be written as:

$\text{easy } 3 \ 4 \ 5$ $\text{easy } 9 \ 1 \ 3$ $\text{easy } 333 \ 123 \ 567$

The process of computation by calculation is extended to deal with function abstractions by **folding** and **unfolding** definitions:

easy 3 4 5

⇒ { unfold }

3 * (4 + 5)

⇒ { add }

3 * 9

⇒ { multiply }

27

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

easy a b c

\Rightarrow { unfold }

a * (b + c)

\Rightarrow { commutativity of + }

a + (c + b)

\Rightarrow { fold }

easy a c b

Expressions, Values, and Types

- The phrases on which we calculate are called **expressions**.
- When no more unfolding (of either primitives or user-defined functions) 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 $\text{exp} :: T$ to say that expression exp has type T .

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
```

- Strings are just lists of characters

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

The type of a function taking arguments of types A and B and returning a result of type C is written $A \rightarrow B \rightarrow C$.

```
(+) :: 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.

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 destructured by **pattern matching**:

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

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.”

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]
```

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!)

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 **con**structs lists).

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

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

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

```
add123 []  
⇒ [1, 2, 3]
```

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

$$[x_1, x_2, \dots, x_n]$$

is simply a shorthand for

$$x_1 : x_2 : \dots : x_n : []$$

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, 7]
```

```
fromTo 9 18
```

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

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 Hugs interpreter can be used in two modes:

- The **batch interpreter**, invoked by the command `runhugs`, is the standard way of executing larger programs. We'll come back to it next time.
- The **interactive top level**, invoked by the command `hugs`, is a convenient way of experimenting with small programs.

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

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

Suppose file `foo.hs` contains the following:

```
module Foo where

foo x y = x + y*3

bar x = foo x 7
```

Then...

Hugs Example

```
~/advprog/lectures> hugs foo.hs
```

```

--      --  --  --  -----  ---
||    || ||  ||  ||  ||  ||__  Hugs 98: Based on the Haskell 98 standard
||___||  ||__||  ||__||  __||  Copyright (c) 1994-2003
||---||           ___||  World Wide Web: http://haskell.org/hugs
||    ||
||    || Version: November 2003  -----
```

Haskell 98 mode: Restart with command line option `-98` to enable extensions

```
Type :? for help
```

```
Foo> bar 6
```

```
27
```

```
Foo> foo 4 6
```

```
22
```

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
```

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)`.

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...

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.

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.

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)
```

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:

```
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)
```

```
area :: Shape -> Float
```

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



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

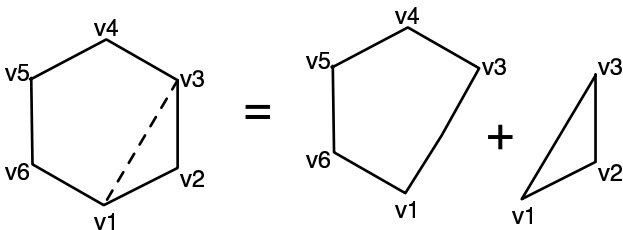
```
area (Ellipse r1 r2) = pi*r1*r2
```

What about polygons?

What about polygons?

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

- 1 Compute the area of the triangle formed by the first three vertices.
- 2 Deleted the second vertex to form a new polygon.
- 3 Sum the area of this new polygon and the area of the triangle from (1).



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.)

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?

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)

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

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.

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 not a good trade!!

Premature Optimization

- ① In most programs, the **vast majority of the time** is spent in a **tiny portion of the code**.
- ② Programmers (even good ones) are notoriously bad at predicting which portion.
- ③ Machines are **real fast** and getting faster. Programmers (even good ones) are not getting faster.

Premature Optimization

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

Moral:

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

(flame off)