Advanced Programming
Andrew Black and Tim Sheard

Lecture 2
Intro to Haskell
Course URLs

- [http://web.cecs.pdx.edu/~cs410aph](http://web.cecs.pdx.edu/~cs410aph)
  - Course web page
  - Links to lecture notes and assignments

- [http://samlapdx.edu/portal](http://samlapdx.edu/portal)
  - Homework submissions
  - Class mailing list.
Course Evaluation

• Daily Assignments
• Class participation, including code walks
• Projects
Haskell

• Haskell is a pure functional language.
  – Yet it supports imperative programming with the use of monads
• There are several implementations
  – HUGS - http://haskell.org/hugs/
• Haskell has a sophisticated type system that catches many errors, and supports advanced abstraction mechanisms
Characteristics of Haskell

• Applicative style
  – input output description of problem.

• First class functions
  – pass as parameters
  – return as value of a function
  – store in data-structures

• Less Importantly:
  – Automatic memory management (G.C. no new or malloc)
  – Use of a strong type system which uses type inference, i.e. no declarations but still strongly typed.
GHC or Hugs?

• Should we use GHC or Hugs?
  – Hugs is an interpreter
    • Great for quick edit/compile/test turnaround
  – GHC is a compiler
    • Fast run-time binaries
    • Extensive tool support (profiling)
    • Many extensions

• For most uses they are compatible, and many programs will run on either system
• So – I recommend you install and use both!
In addition to the command-line compiler, GHC also comes with an interactive environment, GHCI, it works much like Hugs (after which it was modeled). Try it out!
The Hugs Interpreter

Hugs Demo

_________________________________________

Hugs 98: Based Haskell 98 standard
Copyright (c) 1994-2003
World Wide Web: http://haskell.org/hugs
Report bugs to: hugs-bugs@haskell.org
Version: Nov 2003

Hugs mode: Restart with command line option +98 for Haskell 98 mode

Hugs session for:
C:\Program Files\Hugs98\libraries\Hugs\Prelude.hs
C:\Program Files\Hugs98\libraries\Prelude.hs
D:\work\sheard\Courses\CSE502\Fall2004\lect01.html
Type :? for help
Main>
O.O. vs. Functional

- Objects behave
- Functions transform
Basic Approach

• The basic approach to programming in Haskell is two fold
  – Define some data structures
  – Define functions over those data structures

• In contrast with Smalltalk, where the model is simulation of the real world, the model is transformative. I.e. we think of transforming data from one form to another.
IntLists

data IntList
    = Cons Int IntList |
    | Nil

len Nil = 0
len (Cons x xs) = 1 + len xs

addAll Nil = 0
addAll (Cons x xs) = x + addAll xs
Data Declarations

- Data declarations describe tree-like structures allocated in the heap.

```
Cons 3 ─── Cons 2 ─── Cons 9 ─── Nil

2 fields, one an Int the other another list

0 fields
Note since Nil is a constant, there need be only 1 such cell in the system.
```
Constructing Data

• Tree-like structures are created using constructor functions.
  – These have the same name as the tags
  – These names always start with an uppercase letter

\[
data \text{ IntList} = \text{Cons} \; \text{Int} \; \text{IntList} \; | \; \text{Nil}
\]

\[
(\text{Cons} \; 3 \; (\text{Cons} \; 2 \; (\text{Cons} \; 9 \; \text{Nil})))
\]
Exercise #1

• Define a Haskell Data declaration to capture the following tree-like structure.

Data Tree1 = ...
Exercise #2

• Construct the tree using the constructors of your data declaration.
Exercise #3

1. Define a Haskell Data declaration to capture the following tree-like structure.
2. Construct the tree
Functions

• Functions are defined by cases over the data using **patterns**.
• Patterns appear on the left-hand-side of a case.
• Patterns match tree-like structures and bind variables to values.

\[
\text{len } \text{Nil} = 1 \\
\text{len } (\text{Cons } x \text{ xs}) = 1 + \text{len } \text{xs}
\]
Pattern Construction

• Patterns are composed of
  – Constructors
  – Variables
  – Constants
  – Tuples

  – For example:
    Cons x xs
    Cons 5 Nil
    ( y , Cons x Nil )
    constructors and variables
    constructors and constants
    tuples, constructors, and variables
Pattern Matching

• Every pattern matches and binds variables.
• A pattern can match many structures.
• E.g. \((\text{Cons } x \ x \ x s)\) and \(\text{Nil}\)

\[
\begin{array}{c}
\text{Cons} \\
\begin{array}{c}
5 \\
\end{array} \\
\rightarrow \\
\text{Nil} \\
\end{array}
\]

\[
\begin{array}{c}
\text{Nil} \\
\end{array}
\]

\[
\begin{array}{c}
\text{x} \\
\text{binds to } 5 \\
\end{array}
\]

\[
\begin{array}{c}
\text{x} \text{S} \\
\text{binds to } \text{Nil} \\
\end{array}
\]
Nested patterns and constants

• \((\text{Cons } x \ (\text{Cons } 5 \ x))\) matches

\[
\begin{align*}
\text{Cons} & \quad \text{Cons} & \quad \text{Nil} \\
22 & \quad 5 & \quad \text{(Cons 22 \ (Cons 5 Nil))}
\end{align*}
\]

\[
\begin{align*}
\text{Cons} & \quad \text{Cons} & \quad \text{Cons} & \quad \text{Nil} \\
3 & \quad 5 & \quad 9 & \quad \text{(Cons 3 \ (Cons 5 \ (Cons 9 Nil)))}
\end{align*}
\]

• But NOT

\[
\begin{align*}
\text{Cons} & \quad \text{Nil} \\
3 & \quad \text{(Cons 3 Nil)}
\end{align*}
\]

\[
\begin{align*}
\text{Cons} & \quad \text{Cons} & \quad \text{Cons} & \quad \text{Nil} \\
5 & \quad 4 & \quad 22 & \quad \text{(Cons 5 \ (Cons 4 \ (Cons 22 Nil)))}
\end{align*}
\]
Selection or Deconstruction

• Pattern matching is used to select or take apart tree-like heap structures.
• We can define selection functions using patterns

```
listHead (Cons x xs) = x
listTail (Cons x xs) = xs
first (x,y) = x
second (x,y) = y
```
A second look

len Nil = 0
len (Cons x xs) = 1 + len xs

addAll Nil = 0
addAll (Cons x xs) = x + addAll xs

empty Nil = True
empty (Cons x xs) = False

Recursion is the only primitive control structure
Pattern Matching vs Selection

addAll Nil = 0
addAll (Cons x xs) = x + addAll xs

add2 x =
    if empty x
    then 0
    else (listHead x) +
        (add2 (listTail x))
Built in Polymorphic Lists

• Haskell has a complete set of functions that work on polymorphic lists.
  
  \[ 1, 4, 6 \] : \textbf{[Int]}  
  \[ \text{True, False} \] : \textbf{[Bool]}  
  \[ \text{Cons 2 Nil} \] : \textbf{[IntList]}  

• Such lists use special syntactic sugar.
• It is as if they were defined using
  
  \[
  \text{data} \ [a] \ = \ (\ :) \ a \ [a] \ | \ []
  \]
Special Syntax, but just the same

- \(1 : (2 : (5 : []))\) or \([1,2,5]\)

- \([\text{True}, \text{False}]\) or \((\text{True} : \text{False} : [\text{[]}])\)

Right associativity of \((:)\) means we don’t need parentheses.
Functions on polymorphic lists

sum :: [Int] -> Int
sum [] = 0
sum (x : xs) = x + sum xs

length :: [a] -> Int
length [] = 0
length (x : xs) = 1 + length xs

null :: [a] -> Bool
null [] = True
null (x : xs) = False

Polymorphic type means length works on all kinds of lists.
Strings and characters

• Strings are nothing more than lists of characters
  - type String = [ Char ]

• Strings have special syntax
  - “abc” is the same as [ ‘a’, ‘b’, ‘c’ ]

• We can access library functions over characters by using the Char library

    import Char

    Char.ord :: Char -> Int
    Char.chr :: Int -> Char
test.hs

import Char

test = Char.ord (head "abc")
test2 = Char.chr test
Tuples

• Tuples are heterogenous collections of fixed width

• (4, True) : (Int, Bool)
• (3, 4, 5) : (Int, Int, Int)
• (True, "zx", False, 5) : (Bool, String, Bool, Int)
Pattern matching

• As we saw in first and second, we can pattern match over tuples.
  
  \[\text{first } (x,y) = x\]
  \[\text{second } (x,y) = y\]

\[\text{oneOfThree } (x,y,z) = x\]
\[\text{twoOfThree } (x,y,z) = z\]
\[\text{down2 } (((x,y),z),w) = x\]
\[\text{firstOfHead } ((x,y):xs) = x\]
Mundane issues

• Lexical issues
• The offsides rule (indenting matters!)
• Setting the search path
• Getting information from Hugs
• Using Hugs
  – load files
  – Change directories
  – Get type information
Syntactic Elements 1

• Literals
  – Char ‘a’
  – Integer 345
  – Float 23.56
  – String “abc”

• Constructors
  – Constructors start with an Uppercase letter
  – Primitive operator constructors [ ] and :
Syntactic Elements 2

• Operator chars: operators are combinations of
  - : # $ % & * + - = . / \ < > ? ! @ ^ |
  - Reserved operators: (Special sequences of op-chars)
    • :: = .. -- @ \ | <- - > ~ =>

• Operators are used in an “infix” manner
  - 5 + 3
  - (f x) ==> (\ x -> 3 + 4)
  - Any identifier can be made infix by using backquotes.
    • E.g. 10 `in` w and 3 `choose` 5
  - ( op ) makes an infix operator a prefix function
    • (+) 4 6
Syntactic Elements 3

• Identifiers start with a lower case letter followed by digits or other letters or primes
  – Valid Examples: a a3 a’b aF
  – Invalid Examples: F1 Good

• Reserved words
  – case class data else
  – if in infix infixl
  – infixr instance let of
  – primitive then type where
Syntax 3

• The offside rule. Indentation matters
  – Continuation across lines with deeper indentation continues the same declaration.
  – Less indentation means the start of the next declaration.
  – Consequence
    
    ALL DECLARATIONS IN THE SAME GROUP MUST BE EQUALLY INDENTED.
OK

\[
\begin{align*}
\text{length} \; [\;] & = 0 \\
\text{length} \; (x : \; xs) & = 1 + \text{length} \; xs
\end{align*}
\]

\[
\begin{align*}
\text{null} \; [\;] & = \text{True} \\
\text{null} \; (x : \; xs) & = \text{False}
\end{align*}
\]
length [] = 0
length (x : xs) = 1 + length xs

null [] = True
null (x : xs) = False
Indentation matters at top level and

- **Case**

  \[
  f \ x = \text{case } x \text{ of}
  \]
  \[
  \quad [] \rightarrow 0
  \]
  \[
  \quad (x:xs) \rightarrow 1 + f \ x
  \]

- **Let**

  \[
  \text{let } f \ (x,y) = x
  \]
  \[
  \quad a = f \ (2,3)
  \]
  \[
  \quad \text{in } a
  \]

- **Where**

  \[
  f \ x = g \ \text{init } x
  \]
  \[
  \text{where } g \ \text{init } [] = \text{init}
  \]
  \[
  \quad g \ \text{init } (x:xs) = g \ (\text{init } + x) \ \text{x} \ \text{s}
  \]
  \[
  \quad \text{init } = 5
  \]
• Hugs has two modes
  – Command mode
    : command
  – Evaluation mode
    expression
Expression Evaluation

? cd /ogi/staff/sheard/cse502
? 5+2
7
? 5 * 2 + 3
13
? sqrt 4.0
2.0
? sum [2,3,4]
9
? length [2,3,4,5]
4
? sort [3,4,1,2,77,6]
[1, 2, 3, 4, 6, 77]
?
LIST OF COMMANDS: Any command may be abbreviated to :c where
C is the first character in the full name.

:load <filenames>    load modules from specified files
:load                clear all files except prelude
:also <filenames>    read additional modules
:reload              repeat last load command
:edit <filename>     edit file
:edit                edit last module
:module <module>     set module for evaluating expressions
<expr>               evaluate expression
:type <expr>         print type of expression
:?                   display this list of commands
:set <options>       set command line options
:set                 help on command line options
:names [pat]         list names currently in scope
:info <names>        describe named objects
:browse <modules>    browse names exported by <modules>
:find <name>         edit module containing definition of name
:!command             shell escape
:cd dir              change directory
:gc                   force garbage collection
:version              print Hugs version
:quit                 exit Hugs interpreter

Main>
Command: typing expressions

Main> :t sort
sort :: Ord a => [a] -> [a]

Main> :t mymap
mymap :: (a -> b) -> [a] -> [b]

Main> :t mymap (\x -> x+1)
mymap (\x -> x + 1) :: Num a => [a] -> [a]

Main>
Functions

• Functions are defined in Files and loaded into to Hugs.
  Example:  :l lect01.hs

• Functions on numbers
  – Type of numbers:  Int and Float
  – Conversion functions:  fromInteger  round

• Functions on Booleans
  – Relational operators
    <  >  <=  >=  ==  /=
  – Combinators
    &&  ||  not

• Examples
  ? 5 > 7
  False
  ? 1==4
  False
Function Definitions

• Functions defined in files by writing equations visit a sample file.hs

• Example:
  
  plusone x = x + 1

• Sample Output when loading:

  Hugs session for:
  E:\programs\Hugs98Dec01\lib\Prelude.hs
  E:\work\sheard\Courses\CSE502\web\lecturenotes\lect 01.hs
  Type :? for help
  Main>
  ? plusone
  plusone

  ? plusone 4
  5
Functions with Multiple Arguments

• Example Definitions
  difference x y = x - y
  choose x y z = if x then y else z

• Example Session:
  Hugs session for:
  E:.. . .\lect01.hs
  ? :type difference
difference :: Int -> Int -> Int
  ? :type choose
  choose :: Bool -> a -> a -> a
  ?

• Arrow is right Associative
  a -> b -> c = a -> (b -> c)
Setting the search path

• When hugs looks for libraries and modules, it looks in the directories denoted by the search path.

• `:set`  
  – will display the path (and lots of other things)

• `:set -P;c:\xxx;d:\yyy`  
  – will add c:\xxx and d:\yyy to the existing path (indicated by the leading semicolon)
Types of Errors

• Syntax errors (use of symbols in unexpected places)
  \[ f \ x = (x*0) = x \]
  
  ? :r
  Reading script file "lect01.hs":
  Parsing
  ERROR "lect01.hs" (line 36): Syntax error in input
  (unexpected `=')
  
  ?

• Undefined variables
  \[ \text{lastone2} \ = \ \text{head} \ . \ \text{rev} \]
  
  ? :r
  Reading script file "lect01.hs":
  Dependency analysis
  ERROR "lect01.hs" (line 38): Undefined variable "rev"
  ?
Types of Errors (cont.)

- Type errors

```haskell
? :type hd
hd :: [a] -> a

? :type plusone
plusone :: Int -> Int

? hd 3

ERROR: Type error in application
*** expression : hd 3
*** term : 3
*** type : Int
*** does not match : [a]

? plusone [1,2]
ERROR: Type error in application
*** expression : plusone [1,2]
*** term : [1,2]
*** type : [Int]
*** does not match : Int
```
Assignment #2

• You are to write a number of functions in Haskell
  - `explode:: String -> [Char]`
  - `digitValue:: [Char] -> [Int]`
  - `reverse:: [Int] -> [Int]`
  - `pairedWithPowersOf10:: [Int] -> [(Int,Int)]`
  - `pairWiseProduct ::[(Int,Int)] -> [Int]`
  - `sum :: [Int] -> Int`
  - `toInt :: String -> Int`
• Write 5 functions (sum & reverse are primitives, and don’t need to be written).
• Write each one using pattern matching and recursion.
• Create a number of tests for each function
  - `test1 = explode “tim” == [‘t’, ‘i’, ‘m’]`
• Turn in an elegant program that communicates your solution well.