CS 457/557 Functional Programming

Lecture 1

Course Overview and Introduction

Course Information

- CS457/557 - Functional Programming
  - Tuesday & Thursday 2:00-3:30pm
  - NH 341
  - Guest Instructor: Mark Jones
  - Regular Instructor (starting with 4th lecture): Andrew Tolmach
  - Phone: 725-5492
  - Email: apt@cs.pdx.edu
  - Office hours: TuTh 4-5 or by appt.
  - Web page: http://www.cs.pdx.edu/~apt/cs457
- Assignments:
  - Weekly programming assignments, due Tuesdays (40%)
- Exams:
  - Midterm exam (30%); Final exam (30%)

Texts

- Text Book (for basic Haskell techniques)
  - Paul Hudak, "The Haskell School of Expression," Cambridge University Press, 2000.
- Auxiliary text:
  - Simon Thompson, "Haskell: The Craft of Functional Programming", 2nd ed., Addison-Wesley, 1999.
- Handouts of other papers for more advanced topics
- Copies of lecture slides are available from web page
  - Thanks to Tim Sheard for many of the slides.
- Web page will be also be used to distribute other course material electronically

What does “functional” mean?

- Programs consist of functions with no side-effects
  - “Applicative” style
  - Input/output description of problem
  - Build programs by function composition
  - No accidental coupling between components
  - Flexible evaluation order
- Functions are “first class” values
  - Pass as parameters
  - Return as value of a function
  - Store in data-structures
  - Supports higher-level, “declarative” programming style

Functional Languages

- Applicative style
  - Encouraged or required, depending on language.
- First-class functions
- Emphasis on types
  - Built-in support for lists and other recursive data types
  - Type inference = strong static type checking but no declarations needed
  - Type system separates pure computations from actions (computations with side effects)
- Automatic memory management
  - Garbage collection; no **`new`** or **`malloc`**
- Emphasis on (informal) program proof
  - Easy laws for program transformation

Why/how study Functional Programming?

- Learn a new way of thinking about problem solving.
- Learn a new way to specify and implement programs.
- Learn by doing. (Homework is essential!)
- Important examples of functional languages
  - Lisp, Scheme
    - » "strict," impure, dynamically typed
  - Standard ML, CAML
    - » "strict," impure, statically typed
  - Haskell, Miranda
    - » "lazy", pure, statically typed

Haskell

- Developed by committee in late 1980's
  - Combined and standardized several earlier languages.
  - Now dominant "lazy" pure FP language.
  - Current stable version is "Haskell 98"
  - Many experimental extensions available.
- We will use an interpreter called Hugs.
  - Available for most platforms
  - Installed on PSU Solaris network (package hugs)
  - Easy to download to your PC (get Hugs98, November2002 version)
- There are also other interpreters, compilers.
  - May want to explore.
- The Haskell homepage has lots of useful information:
  - http://www.haskell.org

Simple expressions in Hugs

```
Prelude> 5+2
7
Prelude> 5 * 2 + 3
13
Prelude> sqrt 4.0
2.0
Prelude> sum [2,3,4]
9
Prelude> length [2,3,4,5]
4
Prelude> sort [3,4,1,2,77,6]
[1, 2, 3, 4, 6, 77]
Prelude>
```

Syntactic Elements

- Identifiers start with a lower case letter followed by letters, digits, primes, or underscores
  - Valid Examples: **`a  a3  ab'  aF  a_b7`**
  - Invalid Examples: **`F1  Good`**
  - Excludes these reserved words:
    - » **`case class data default deriving do else if`**
    - » **`import in infix infixl infixr instance let module`**
    - » **`newtype of then type where as qualified hiding`**
- Types and constructors start with upper case letter
  - Examples: **`Int  Bool  True  False  Just`**
  - Some special cases: **`[]  :  (,)`**

Syntactic Elements (cont.)

- Operators
  - Formed by combinations of
    - `! # $ % & * + . / < = > ? @ \ ^ | - ~ :`
  - Excluding certain reserved sequences:
    - `.. :: = \ | <- -> @ ~ =>`
  - Used in an "infix" manner:
    - E.g. `5 + 3`
  - Can be made "prefix" by enclosing in parentheses
    - E.g. `(+) 5 3`
  - Any identifier can be made infix by using backquotes.
    - E.g. `` 10 `in` w `` or `` 3 `choose` 5 ``
- Literals
  - Integers, e.g. `123` `39949993` `0xff7f` `0o722`
  - Floating point, e.g. `3.14` `7.0` `0.45` `8.5e7`
  - Characters,e.g. `'a'` `'Z'` `'\n'` Strings, e.g. `"abc"` `"def\n"`

Functions

- Functions are defined by equations in files
- Example file lect01.hs:

```
plusone :: Int -> Int
plusone x = x + 1
```

- Example dialog in hugs:

```
Prelude> :l lect01.hs
Reading file "lect01.hs":
Hugs session for:
C:\hugs\lib\Prelude.hs
lect01.hs
Main> plusone 41
42
```

Functions with Multiple Arguments

- Example Definitions

```
difference :: Int -> Int -> Int
difference x y = if x <= y then y-x else x-y
```

- Example Session:

```
Main> difference 3 6
3
Main> :type difference
difference :: Int -> Int -> Int
Main> difference
ERROR - Cannot find "show" function for:
*** Expression : difference
*** Of type    : Int -> Int -> Int
```

- Arrow is right associative

```
a -> b -> c =  a -> (b -> c)
```

Constructing Lists

- The Empty List `[]`
- The "Cons" (:) Constructor

```
Prelude> 3 : [3,4,5]
[3, 3, 4, 5]
```

- The Dot Dot notation

```
Prelude> [1 .. 4]
[1, 2, 3, 4]
```

- The Comprehension notation

```
Prelude> [x + 1 | x <- [2..4]]
[3, 4, 5]
Prelude> [ (x,y) | x <- [1..2], y <- [3,5,7]]
[(1,3), (1,5), (1,7), (2,3), (2,5), (2,7)]
Prelude> [ x * 2 | x <- [1..10], even x]
[4, 8, 12, 16, 20]
```

Taking Lists Apart

```
Prelude> head [1,2,3]
1

Prelude> tail [1,2,3]
[2, 3]

Prelude> null [2]
False

Prelude> take 2 [1,2,3]
[1,2]

Prelude> drop 2 [1,2,3]
[3]
```

Exercise

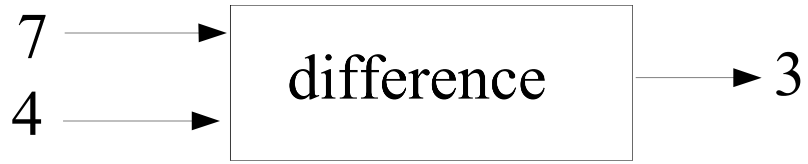
- Define prefix and lastone in terms of head, tail and reverse. First make a file “lect02.hs”
- Sample Hugs run

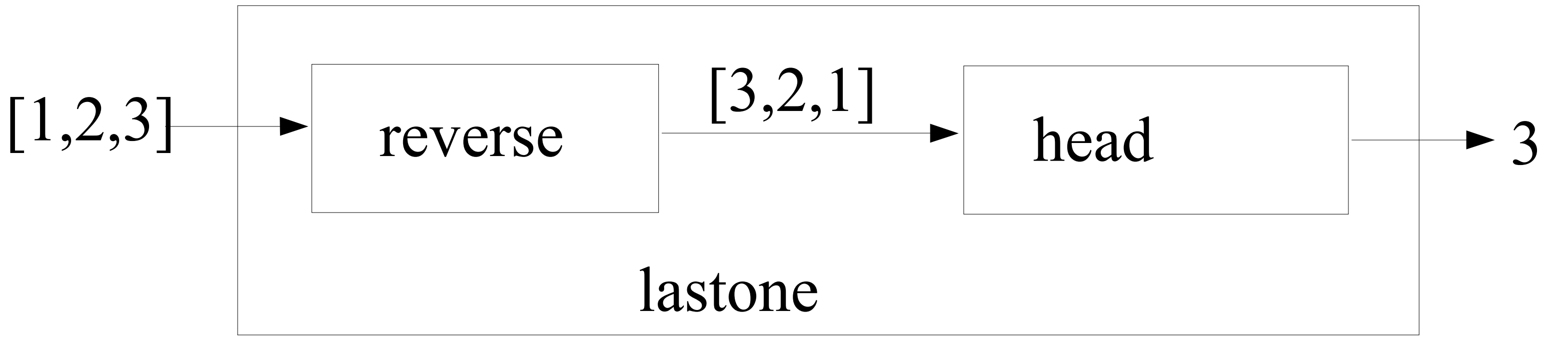
```
Prelude> :l lect02.hs
Reading file "lect02.hs":
Hugs session for:

C:\hugs\lib\Prelude.hs
lect02.hs
Main> lastone [1,2,3,4]
4
Main> prefix [1,2,3,4]
[1, 2, 3]
Main>
```

Thinking about Functions

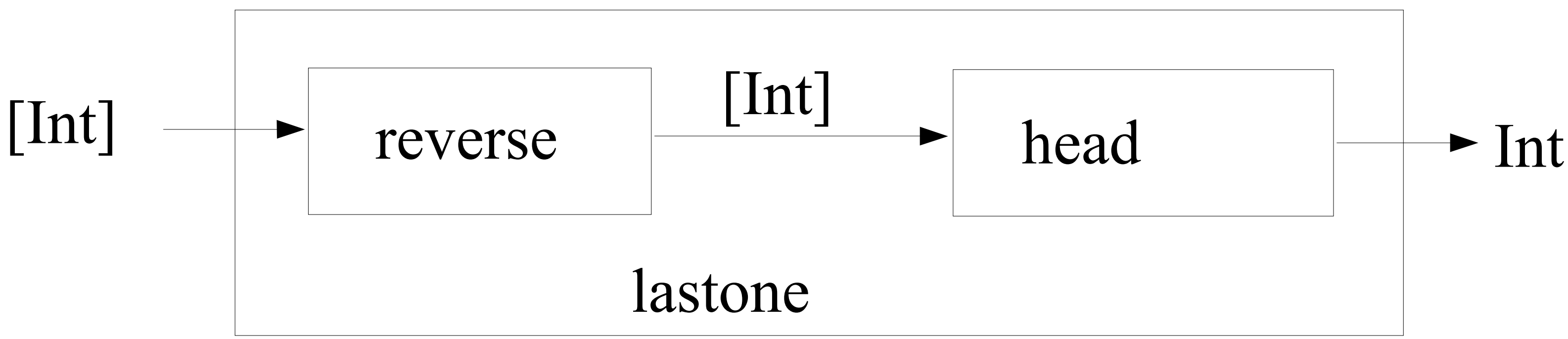
- Can picture function as a box with some inputs and an output:





Thinking about Types

- A type is a collection of values. Functions can only be applied to arguments of appropriate types.



Computation by Calculation

- In a pure functional language, we can always perform computation by replacing defined symbols by their definitions:

```
(7-3)*2 ==>
4*2 ==>
8
```

- Given

```
a = 10
b = 7
difference x y = if x <= y then y-x else x-y
```

- Can calculate

```
difference a b ==>
if a <= b then b-a else a-b ==>
if 10 <= 7 then 7-10 else 10-7 ==>
if False then 7-10 else 10-7 ==> 10-7 ==> 3
```