

A Brief Intro to Haskell

CS395T - Domain Specific Languages
Greg Lavender
Department of Computer Sciences
The University of Texas at Austin



Haskell

- Named after Haskell Curry
 - inventor of combinatory logic
 - equivalence to λ -calculus shown by J. B. Rosser
- Key features
 - Higher-order polymorphic functional language
 - Referentially transparent
 - Non-strict (lazy) evaluation of expressions
 - Monads & List comprehensions
- See www.haskell.org for info
 - Hugs98 - interpreter
 - GHC-6.x - compiler/interpreter



Simple Expressions

- Integers, floats, string, characters and infix operators
 - Strings are list of characters

```
1+2+3 => 6
3.14 * 2 => 6.28
4/2 => 2.0
4 `div` 2 => 2
1 < 2 => True
```

```
"Hello " ++ "World" => "Hello World"
ord('a') => 97
chr(97) => 'a'
'a' : 'b' : ['c'] => "abc"
"abc" > "def" => False
```

3/1/05

3



Operator Precedence & Associativity

- Builtin bindings

```
infixr 9  .
infixl 9  !!
infixr 8  ^, ^^, **
infixl 7  *, /, `quot`, `rem`, `div`, `mod`, :%, %
infixl 6  +, -
infixr 5  :
infixr 5  ++
infix  4  ==, /=, <, <=, >=, >, `elem`, `notElem`
infixr 3  &&
infixr 2  ||
infixl 1  >>, >>=
infixr 1  <<<
infixr 0  $, $!, `seq`
```

3/1/05

4



Functions

- Functions are statically typed and may be polymorphically typed

```
square :: Float->Float
square x = x * x
```

```
fact :: Integer->Integer
fact n = if (n==0) then 1 else n * fact(n-1)
```

```
succ :: Int->Int
succ 0 = 1
succ n = n + 1
```

```
id :: a -> a
id x = x
```

3/1/05

5



Functions

- Definition by patterns is common

```
fact :: Integer -> Integer
fact 0 = 1
fact n = n * fact(n-1)
```

```
fact n | n == 0 = 1
      | n > 0 = n * fact (n-1)
```

```
-- inductively recursive version
fact 0 = 1
fact (n+1) = (n+1) * fact n
```

```
-- iterative version
fact 0 = 1
fact n = foldl (*) 1 [1..n]
```

```
-- tail recursive
fact n = tfact n 1
      where tfact 0 m = m
            tfact n m = tfact (n-1) (n*m)
```

3/1/05

6



List Functions

- Lists are homogenous collections of some type 'a' often denoted as [a]
 - Haskell provides a large set of list functions

```
head      :: [a] -> a
head (x:_) = x

last      :: [a] -> a
last [x]  = x
last (_:xs) = last xs

tail      :: [a] -> [a]
tail (_:xs) = xs

init      :: [a] -> [a]
init [x]   = []
init (x:xs) = x : init xs

null      :: [a] -> Bool
null []    = True
null (_:_) = False
```

3/1/05

7



List Functions

```
(++)      :: [a] -> [a] -> [a]
[] ++ ys  = ys
(x:xs) ++ ys = x : (xs ++ ys)

map       :: (a -> b) -> [a] -> [b]
map f xs  = [ f x | x <- xs ]

filter    :: (a -> Bool) -> [a] -> [a]
filter p xs = [ x | x <- xs, p x ]

concat    :: [[a]] -> [a]
concat    = foldr (++) []

length    :: [a] -> Int
length    = foldl' (\n _ -> n + 1) 0

(!!)      :: [a] -> Int -> a
(x:_) !! 0 = x
(_:xs) !! n | n > 0 = xs !! (n-1)
(_:_) !! _ = error "Prelude.!!!: negative index"
[] !! _ = error "Prelude.!!!: index too large"
```

3/1/05

8



More List Functions

```
iterate      :: (a -> a) -> a -> [a]
iterate f x  = x : iterate f (f x)

repeat      :: a -> [a]
repeat x     = xs where xs = x:xs

replicate   :: Int -> a -> [a]
replicate n x = take n (repeat x)

cycle       :: [a] -> [a]
cycle []     = error "Prelude.cycle: empty list"
cycle xs    = xs' where xs'=xs++xs'

take        :: Int -> [a] -> [a]
take n _ | n <= 0 = []
take _ []      = []
take n (x:xs)  = x : take (n-1) xs

drop        :: Int -> [a] -> [a]
drop n xs | n <= 0 = xs
drop _ []        = []
drop n (_:xs)    = drop (n-1) xs
```

3/1/05

9



Folding over a List

```
-- non-strict folding, not that foldl is tail recursive
foldl      :: (a -> b -> a) -> a -> [b] -> a
foldl f z []      = z
foldl f z (x:xs) = foldl f (f z x) xs

foldl1     :: (a -> a -> a) -> [a] -> a
foldl1 f (x:xs)  = foldl f x xs

foldr      :: (a -> b -> b) -> b -> [a] -> b
foldr f z []      = z
foldr f z (x:xs) = f x (foldr f z xs)

foldr1     :: (a -> a -> a) -> [a] -> a
foldr1 f [x]      = x
foldr1 f (x:xs)  = f x (foldr1 f xs)

-- strict folding
foldl'     :: (a -> b -> a) -> a -> [b] -> a
foldl' f a []      = a
foldl' f a (x:xs) = (foldl' f $! f a x) xs
```

3/1/05

10



Higher Order Polymorphic Functions

- Map and fold are commonly used in Haskell to apply a function over a list

```
map :: (a -> b) -> [a] -> [b]
map f [] = []
map f (x:xs) = f x : map f xs

-- alternatively using a list comprehension
map f xs = [ f x | x <- xs ]

len :: [a] -> Int
len xs = foldl (\n _ ->n+1) 0 xs

sum,prod :: Num a => [a] -> Int
sum xs = foldl (+) 0 xs
prod xs = foldl (*) 1 xs
```

3/1/05

11



Functions on Pairs

```
fst :: (a,b) -> a
fst (x,_) = x

snd :: (a,b) -> b
snd (_,y) = y

swap :: (a,b) -> (b,a)
swap (a,b) = (b,a)
```

3/1/05

12

Avoiding Redundant List Traversal

- Computing the mean of a list of ints
 - sum divided by the length

```
len xs = foldl (\_ n ->n+1) 0 x
sum xs = foldl (+) 0 xs
mean xs = (sum xs) / fromIntegral (len xs)
```

- How do we avoid two list traversals?
 - compute the pair (sum,len) as we traverse

```
suml :: [Int] -> (Int,Int)
suml xs = foldl (\(s,l) x -> (s+x,l+1)) (0,0) xs

mean xs = (fst p) / fromIntegral (snd p)
          where p = suml xs
```

3/1/05

13

Recursive Functions

- Normal order Y combinator

```
fix f = f (fix f)
-- alternatively: fix f = let x = f x in x

-- factorial
fact :: Integer->Integer
fact = fix (\f n -> if n == 0 then 1 else n * f (n-1))

-- length of a list
len :: [a] -> Int
len = fix (\f xs -> case xs of
                    [] -> 0
                    (x:xs) -> 1 + (f xs))

-- map
map :: (a->b) -> [a] -> [b]
map = fix (\f g xs -> case xs of
                    [] -> []
                    (x:xs) -> g x : f g xs)4
```

3/1/05



Lazy Evaluation

- Infinite lists

```
zeros = 0:zeros => [0,0,0,...]
ones = 1:ones => [1,1,1,...]

primes :: [Integer]
primes = sieve [2 .. ] -- Sieve of Eratosthenes

-- lazy list using a list comprehension
sieve (x:xs) = x : sieve [y | y <- xs, (y `rem` x) /= 0]

nthprime n = take n primes
```

3/1/05

15



Fibonacci Numbers

```
-- Two fibonacci functions: one slow, one fast
-- standard fibonacci function
fibonacci :: Int -> Integer
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci n = fibonacci(n-2) + fibonacci(n-1)

-- infinite fibonacci sequence
fibs :: [Integer]
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

fibn :: Int -> [Integer]
fibn n = take n fibs

-- fibonacci function that selects nth element of sequence
fib :: Int -> Integer
fib n = fibs !! N

-- using a lazy list comprehension
fibs = 1 : 1 : [a+b | (a,b) <- zip fibs (tail fibs)]
```

3/1/05

16



Lambda Functions

- Lambda forms are used to define curried functions and inline functions

```
-- curried successor function
succ1 = \n -> n + 1
succ2 = (\a -> \b -> a + b) 1

-- generate an infinite list of squares
map (\x -> x * x) [1..] => [1,4,9,14,25,...]

-- even map can be written as:
map f = foldr (\x ys -> (f x):ys) []
```

3/1/05

17



Lists and List Comprehensions

- Lists are a special builtin polymorphic data type that conceptually is defined as:

```
data [a] = [] | a : [a] deriving (Eq, Ord)
```

- Note that the "cons" operator ":" is right associative

```
1:2:3:[] => 1:2:[3] => 1:[2,3] => [1,2,3]
```

- a list comprehension for computing a cross product

```
cross :: [Int] -> [Int] -> [(Int,Int)]
cross xs ys = [(x,y) | x<-xs, y<-ys]
```

3/1/05

18



Data Types

- **A Simple Stack**

```
data Stack a = Empty | Stk [a]
              deriving (Eq,Read,Show)

push :: a -> Stack a -> Stack a
push x Empty = Stk [x]
push x (Stk xs) = Stk (x:xs)

pop :: Stack a -> (a, Stack a)
pop (Stk []) = error "Empty stack"
pop (Stk (x:xs)) = (x, Stk xs)

peek :: Stack a -> a
peek (Stk (x:xs)) = x

empty :: Stack a -> Bool
empty Empty = True
empty (Stk x) = False
```

3/1/05

19



Recursive Data Type

- **Natural numbers object using Peano Arithmetic**

```
data Nat = Zero | Succ Nat
          deriving (Eq,Ord,Read,Show)

isZero :: Nat -> Bool
isZero Zero = True
isZero (Succ n) = False

add :: Nat -> Nat -> Nat
add m Zero = m
add m (Succ n) = Succ (add m n)

mult :: Nat -> Nat -> Nat
mult m Zero = Zero
mult Zero (Succ n) = Zero
mult m (Succ n) = add m (mult m n)
```

3/1/05

20