Lecture 8: Regular Expressions in Haskell

CSCI 101
Spring, 2019

Kim Bruce

New Homework

- Haskell programming

Recursive Datatype Examples

- data Tree a = Niltree |
  Maketree (a, Tree a, Tree a)

Binary Search Using Trees

```
insert new Niltree = Maketree(new,Niltree,Niltree)
insert new (Maketree (root,l,r)) =
  if new < root
    then Maketree (root,(insert new l),r)
    else Maketree (root,l,(insert new r))

buildtree [] = Niltree
buildtree (fst : rest) =
                insert fst (buildtree rest)
```

Binary Search Tree

```
find elt Niltree = False
find elt (Maketree (root,left,right)) =
  if elt == root
     then True
     else if elt < root then find elt left
     else find elt right      -- elt ≥ root

bsearch elt list = find elt (buildtree list)
```

Haskell is Lazy!

Lazy vs. Eager Evaluation

- Eager: Evaluate operand, substitute operand value in for formal parameter, and evaluate.
- Lazy: Substitute operand for formal parameter and evaluate body, evaluating operand only when needed.
  - Each actual parameter evaluated either not at all or only once! (Essentially cache answer once computed)
  - Like left-most outermost, but more efficient

Lazy evaluation

- Compute f(1/0,17) where f(x,y) = y
- Computing head(qsort[5000,4999..1]) is faster than qsort[5000,4999..1}
- Compare time of computations of:
  - fib 32
  - dble (fib 32) where dble x = x + x

Lazy Lists

```
fib 0 = 1
fib 1 = 1
fib n = fib (n-1) + fib (n-2)
```

complexity O(fib n) ~ O(2ⁿ)

```
fibList = f 1 1
  where f a b = a : f b (a+b)
fastFib n = fibList!!n
```

complexity O(n)

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

primes = sieve [ 2.. ]
       where
         sieve (p:x) = p :
              sieve [ n | n <- x, n `mod` p > 0]
```

Call-by-need

- Efficient implementation of call-by-name (Algol 60)
  - *Most languages use call be value!*
- If purely functional language then may evaluate expression at most once, because can never change.

Input/Output in Haskell

Instance of Monads in Haskell, which we will not discuss.

Printing is easy

- main = putStrLn "hello world!"
  - program that prints to screen — without quotes
- putStrLn :: String -> IO()
  - returns I/O action with no value
- Main program will always be an IO action

Why an IO "action"

- IO is a "side-effect" and Haskell doesn't allow side effects.
  - Side effect is a change that causes expressions to return different values.
  - If x = 10, then x + x causes no side effects — every time you evaluate it, you get the same answer.
  - ++x + (++x) has side effects — every time it is evaluated x increases by 2.
- Input and output have side effects
  - change screen, or use up input

I/O in Haskell

- The I/O language is external to Haskell, but can call pure Haskell programs
- getLine :: IO String
  - IO action that gets a string
  - Need a way to access string value
- "do" construct allows us to glue together IO actions.

More I/O

- Recall:
  - putStrLn :: String -> IO()
  - getLine :: IO String
  - Can't write
    - echo = putStrLn (getLine)
    - types can't compose.

Do to Rescue!

- do clause can extract value from an IO action to be used in a later Haskell function or IO action

```
main = do
  putStrLn "Type your name"
  name <- getLine
  putStrLine ("Hi, "++ name)
```

- name is String that can be used in next line
  - but do must always result in IO action

More IO

```
ask :: String -> String -> IO()
ask prompt ansPrefix =
        do putStr (prompt++" ")
           response <- getLine
           putStrLn (ansPrefix ++ " " ++ response)

getInteger :: IO Integer
getInteger = do putStr "Enter an integer: "
                line <- getLine
                return (read line)
        -- converts string to Integer then to IO Integer
```

Using IO in Haskell

```
ifIO :: IO Bool -> IO a -> IO a -> IO a
ifIO b tv fv = do { bv <- b;
                    if bv then tv else fv}
```

- Can build language at IO monad level:

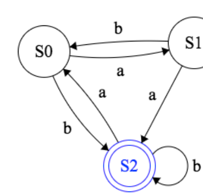
```
whileIO :: IO Bool -> IO( ) -> IO( )
whileIO b m = ifIO b
                 (do {m; whileIO b m})
                 (return( ))
```

Sets in Haskell

- Differ from lists as order not important
- Intended to be imported qualified:
  - import Data.Set(Set)
    import qualified Data.Set as Set
  - Allows you to just use Set rather than Data.Set
  - avoids clashes with Prelude functions
  - operations: Set.singleton, Set.empty, Set.union, Set.member

DFSM in Haskell

- Write function `accepts` that takes a configuration (state,input) and computes the state after all input read.



Simulating DFSM

- Code in Haskell:
  - Simple emulation for fixed machine
    - See function decide in SimpleExampleFSM.hs
  - General emulation for arbitrary NDFSM
    - See function gAccept in same file
    - Uses record, which automatically generates functions to extract each field. E.g., startingState(fsm), transitionFunction(fsm), ...

General simulation

- Provide transition function as set of triples
- Build machine from start state, triples, and set of final states
  - myFSM = FSM start triples final
- To apply write
  - gAccept myFSM input
- Returns True iff it accepts it.