June 4, 2009 St. Louis Lambda Lounge Haskell Presentation

Alex Stangl

Overview and study Vending Machine code

Haskell overview

- Purely functional
- Strongly typed with type inference, polymorphism
- Pattern Matching
- Lazy (non-strict), by default

Haskell overview cont'd

- Stable Haskell 98 standard, GHC (de facto standard), new standard coming
- Concise, Powerful, Open
- Research lab for new ideas

Purely Functional

- Like a Mathematical function, result based only on arguments, with no side effects
- Referential Transparency – replacing pure function call with its result value has no effect on program semantics; no difference between reference to thing and thing itself
- Well suited for Memoization (Dynamic Programming / caching)
- Code easier to reason about
- Intelligent compiler can optimize

Comment block.
{- Can also {- nest -}
comments. -}

```
-- Vending Machine Simulator
-- written 02/18/2009 by Alex Stangl
-- for 5/2009 STL Lambda Lounge shootout

module VendMachine where

import Data.Char(chr, ord, toUpper)
import Data.List((\\), sortBy, stripPrefix)
import Data.Ord(comparing)
```

Simple module declaration syntax.

```
-- Ven   g Machine Simulator
-- wri  en 02/18/2009 by Alex Stangl
-- for 5/2009 STL Lambda Lounge shootout

module VendMachine where

import Data.Char(chr, ord, toUpper)
import Data.List((\\), sortBy, stripPrefix)
import Data.Ord(comparing)
```

Imports from libraries into local namespace

```
-- Vendin
-- written 02     2009 by Alex Stangl
-- for 5/2009      Lambda Lounge shootout

module VendMa hine where

import Data.Char(chr, ord, toUpper)
import Data.List((\\), sortBy, stripPrefix)
import Data.Ord(comparing)
```

Operators named with symbols must be enclosed in parens when not used infix.

```
-- Vending Machine Simulator
-- written 02/18/2009 by Alex Stangl
-- for 5/2009 STL Lambda Lounge shootout

module VendMachine where

import Data.Char(chr, ord, toUpper)
import Data.List((\\), sortBy, stripPrefix)
import Data.Ord(comparing)
```

Algebraic datatype
often parameterized,
but not here

Type
constructor

```
-- currency representation, either coins or bills
data Currency = Nickel | Dime | Quarter | Dollar
  deriving (Eq, Show)
```

Data
constructors.

```haskell
-- currency representation, either coins or bills
data Currency = Nickel | Dime | Quarter | Dollar
  deriving (Eq, Show)
```

Automatically generate functions for equality and show.

This is a type signature.
Read “::” as “... has type ...”

```
-- return amount of currency, in cents
amount :: Currency -> Int
amount Nickel   = 5
amount Dime     = 10
amount Quarter = 25
amount Dollar   = 100
```

amount “has type”
function taking 1
Currency argument,
returning an Int

```haskell
-- return amount of currency, in cents
amount :: Currency -> Int
amount Nickel  = 5
amount Dime    = 10
amount Quarter = 25
amount Dollar  = 100
```

Function comprised of 4 equations covering the possibilities. Simple pattern matching on data constructors here.

```
-- return display/input name of currency
name :: Currency -> String
name x = map toUpper $ show x
```

Type signature of simple function taking 1 Currency argument and returning a String.

```
-- return display/input name of currency
name :: Currency -> String
name x = map toUpper $ show x
```

Pattern variable x bound to Currency argument.

Low precedence right-associative function application.

First use show to convert Currency argument to its String representation...

```haskell
-- return display/input name of currency
name :: Currency -> String
name x = map toUpper $ show x
```

First use show to convert Currency argument to its String representation...

```
-- return display/input name of currency
name :: Currency -> String
name x = map toUpper $ show x
```

...then apply map toUpper to the String. String is a List of Char, and map applies toUpper to each Char, producing an uppercased String.

Type signature for function taking 1 Int argument and returning a String, converting 0-based Int to a slot name (ala Excel column names.)

```
--- return slotname for slot: 0.. -> A..
slotname :: Int -> String
slotname slot = slotnamR (slot+1)
slotnamR slot
  | slot <= 26 = alpha slot
  | rem == 0   = slotnamR (quo-1) ++ "Z"
  | otherwise  = slotnamR quo ++ alpha slot
    where (quo, rem) = slot `quotRem` 26
          alpha n = [chr (64+n)]
```

slotname delegates to helper function slotnamR, passing it a 1-based slot number.

```
--- return slotname for slot: 0.. -> A..
slotname :: Int -> String
slotname slot = slotnamR (slot+1)
slotnamR slot
  | slot <= 26 = alpha slot
  | rem == 0   = slotnamR (quo-1) ++ "Z"
  | otherwise  = slotnamR quo ++ alpha slot
    where (quo, rem) = slot `quotRem` 26
          alpha n = [chr (64+n)]
```

recursive helper slotnamR uses guards and a where clause that applies across all the guards.

```
--- return slotn     for slot: 0.. -> A..
slotname :: Int    String
slotname slot = slotnamR (slot+1)
slotnamR slot
  | slot <= 26 = alpha slot
  | rem == 0   = slotnamR (quo-1) ++ "Z"
  | otherwise  = slotnamR quo ++ alpha slot
    where (quo, rem) = slot `quotRem` 26
          alpha n = [chr (64+n)]
```

Return both quotient and remainder of divide by 26

```
{- return slot number, given its name (ala Excel
   column names: A..Z == 0..25,
   AA..AZ == 26..51, BA..BZ == 52..77, etc.) -}
slotnumber :: String -> Int
slotnumber s = sltnR 0 0 s
sltnR _ t []     = t
sltnR 0 0 (x:xs) = s
sltnR l t (x:xs) = s                             5) xs
```

Inverse of previous function – given a slot name string, return its corresponding 0-based number.

Delegates to helper sltnR which takes 3 arguments: character index, result accumulator, and remaining characters to process.

```
{- return st                                a Excel
   column names: A..Z == 0..2
   AA..AZ == 26..51, BA..BZ =      ..77, etc.) -}
slotnumber :: String -> Int
slotnumber s = sltnR 0 0 s
sltnR _ t []      = t
sltnR 0 0 (x:xs) = sltnR 1 (ord(x)-65) xs
sltnR l t (x:xs) = sltnR (l+1) (26*(t+1)+ord(x)-65) xs
```

First equation handles end-of-string [] in which case we return accumulator t. Pattern matching becomes more obvious here.

```
{- return slot number, g
   column names: A..Z == 0
   AA..AZ == 26..51, BA..      = 52..77, etc.) -}
slotnumber :: String ->      t
slotnumber s = sltnR 0 0 s
sltnR _ t []      = t
sltnR 0 0 (x:xs) = sltnR 1 (ord(x)-65) xs
sltnR l t (x:xs) = sltnR (l+1) (26*(t+1)+ord(x)-65) xs
```

Second equation handles first slotname character as a special case, recurses to general case.

```
{- return slot number, giver          ame (ala Excel
   column names: A..Z == 0..2
   AA..AZ == 26..51, BA..BZ =        ..77, etc.) -}
slotnumber :: String -> Int
slotnumber s = sltnR 0 0 s
sltnR _ t []     = t
sltnR 0 0 (x:xs) = sltnR 1 (ord(x)-65) xs
sltnR l t (x:xs) = sltnR (l+1) (26*(t+1)+ord(x)-65) xs
```

(:) list constructor used as a deconstructing pattern here, will not match empty list.

ord is inverse of chr, returning Unicode code for specified character

```haskell
{- return slot number, given its name (ala Excel
   column names: A..Z == 0..25,
   AA..AZ == 26..51, BA..BZ == 52..77, etc.) -}
slotnumber :: String -> Int
slotnumber s = sltnR 0 0 s
sltnR _ t []     = t
sltnR 0 0 (x:xs) = sltnR 1 (ord(x)-65) xs
sltnR l t (x:xs) = sltnR (l+1) (26*(t+1)+ord(x)-65) xs
```

General recurrence case: multiply accumulated result by 26, add in current character's offset from 'A', tail recurse.

```
{- find change of specified total from l, if possible,
   using greedy, relatively efficient algorithm. Use
   either flip or negation to reverse sort order. -}
getChange l total = findTotal (sortBy (flip $
                      comparing amount) l) [] total
findTot   [] _ total = Nothing
findTo     :xs) acc total
                            Total (dropWhile (==x) xs)
                              tota
                             (x:acc
                             l xs (    )
                             tal
                              (d
                             al
```

No explicit type signature here, but l is list of Currency, total is desired total. Haskell type inferrence figures out type signature. getChange :: [Currency] -> Int -> Maybe [Currency]

getChange delegates to helper findTotal, this time a top-level function rather than defined in where clause. TIMTOWDI

What's going on here? Inside the parens we call sortBy (which takes a comparison function) on l, our list of Currency.

```
                            total from l, if possible,
                              ficient algorithm. Use
                           to reverse sort order. -}
getChange l total = findTotal (sortBy (flip $
                      comparing amount) l) [] total
findTotal [] _ total = Nothi
findTotal (x:xs) acc to
  |                               (dropWhile (==x) xs)
                                  l
  |                               c)
  |                               (x:acc)
                                  amount x) of
                                  While (==x) xs)
```

comparing is a function (or “combinator”) from library that takes another function (amount) and returns a function. The returned function here would compare 2 Currency values (via amount) and return an Ordering for sortBy

```
{- find change of specified total from l, if possible,
   using greedy, relatively efficient algorithm. Use
   either flip or negation to reverse sort order. -}
getChange l total = findTotal (sortBy (flip $
                      comparing amount) l) [] total
findTotal [] _ total = Nothing
findTotal (x:xs) acc
  | amount x >  to                            x) xs)

  | amount x == to
  | otherwise = ca

         Nothing -> findTotal (dropWhile (==x) xs)
                      acc total
         Just a -> Just a
```

flip applied to the function returned by comparing flips the order of its 2 arguments. Net effect: sort the list of Currency in descending order findTotal takes 3 arguments: list of remaining Currency to consider, result accumulator, and (remaining) desired total.

```
{- find change                                          ble,
   using greedy, relatively efficien          rithm. Use
   either flip or negation to revers  sort order. -}
getChange l total = findTotal (sortBy (flip $
                      comparing amount) l) [] total
findTotal [] _ total = Nothing
findTotal (x:xs) acc total
  | amount x >  total = findT   al (dropWhile (==x) xs)
                          acc   tal
  | amount x == total = Just (    c)
  | otherwise = case findTotal      :acc)
         Nothing -> findTo
                      acc
         Just a -> Just a
```

First equation handles case where we've run out of Currency to consider w/o finding a solution, so returns Nothing

Second equation handles remaining cases with if/else if/else and case clause. Note deconstructing pattern used to bind x to head Currency and xs to tail.

```
{- find change c                                    ,
   using greedy,
   either flip or n                                 }
getChange l total = f         al (sortBy (flip $
                      comparing amount) l) [] total
findTotal [] _ total = Nothing
findTotal (x:xs) acc total
  | amount x >  total = findTotal (dropWhile (==x) xs)
                          acc total
  | amount x == total =    ust (x:acc)
  | otherwise = case      Total xs (x:acc)
                          (total- amount x) of
        N                                       ) xs)

        J
```

If the amount of this currency exceeds our remaining total, then recurse after discarding all instances of this Currency

```
{- find change of specified total from l, if possible,
   using greedy, relatively efficient algorithm. Use
   either flip or negation to reverse sort order. -}
getChange l total = findTotal (sortBy (flip $
                      comparing amount) l) [] total
findTotal [] _ total = Nothing
findTotal (x:xs) acc total
  | amount x >  total = findTotal (dropWhile (==x) xs)
                          acc total
  | amount x == total = Just (x:acc)
  | otherwise = case findTotal xs (x:acc)
                          (tota
         Nothing -> findTotal
                      acc tota
         Just a -> Just a
```

If amount of x exactly matches remaining total, we're done – return result using Maybe's Just data constructor.

Try recursing with x added to accumulator. If Nothing returned, then no solution possible with x, so recurse after dropping all of that Currency.

```
{- find ch                                            e,
   using g
   either
getChange l tota              al (sortBy (flip $
                          paring amount) l) [] total
findTotal [] _ t          Nothing
findTotal (x:xs)          total
  | amount x >      al = findTotal (dropWhile (==x) xs)
                           acc total
  | amount x == total = Just (x:acc)
  | otherwise = case findTotal xs (x:acc)
                           (total- amount x) of
       Nothing -> findTotal (dropWhile (==x) xs)
                       acc total
       Just a -> Just a
```

If solution found, return it.

```haskell
{- find change of specified total from l, if possible,
   using greedy, relatively efficient algorithm. Use
   either flip or negation to reverse sort order. -}
getChange l total = findTotal (sortBy (flip $
                      comparing amount) l) [] total
findTotal [] _ total = Nothing
findTotal (x:xs) acc total
  | amount x >  total = findTotal (dropWhile (==x) xs)
                          acc total
  | amount x == total = Just (x:acc)
  | otherwise = case findTotal xs (x:acc)
                          (total- amount x) of
         Nothing -> findTotal (dropWhile (==x) xs)
                      acc total
         Just a -> Just a
```

```
-- break string up into list of commands delimited by
-- space and/or comma
cmds :: String -> [String]
cmds s = let isSpace = (`elem` [' ', ','])
         in case dropWhile isSpace s of
              "" -> []
              s' -> c : cmds s''
                    where (c, s'') = break isSpace s'
```

Use of a “section”, a special case of partial application, using a binary operator. Here we create function that takes 1 Char argument and returns True if it is space or comma.

```
-- break string up into list of c      ds delimited by
-- space and/or comma
cmds :: String -> [String]
cmds s = let isSpace = (`elem` [' ', ','])
         in case dropWhile isSpace s of
                  " -> []
                    > c : cmds s''
                      where (c, s'') = break isSpace s'
```

let creates local definitions, similar to where clause. let can be used anywhere you write an expression

Drop characters off head of list while predicate isSpace returns True (drop leading commas and spaces).

```
-- break string up into       of commands delimited by
-- space and/or comma
cmds :: String -> [String]
cmds s = let isSpace = (`elem` [' ', ','])
         in case dropWhile isSpace s of
              "" -> []
              s' -> c : cmds s''
                    where (c, s'') = break isSpace s'
```

Apply case expression to remaining list. If empty, return empty list.

```haskell
-- break string up into list of commands delimited by
-- space and/or comma
cmds :: String -> [String]
cmds s = let isSpace = (`elem` [' ', ','])
         in case dropWhile isSpace s of
              "" -> []
              s' -> c : cmds s''
                    where (c, s'') = break isSpace s'
```

break takes predicate function and splits list at point when predicate returns True. So c is all characters up to space or comma, s'' is remainder of string.

```
-- break string up into list of commands delimited by
-- space and/or comma
cmds :: String -> [String]
cmds s = let isSpace = (`elem` [' ', ','])
         in case dropWhile isSpace s of
              "" -> []
              s' -> c : cmds s''
                    where (c, s'') = break isSpace s'
```

... so result is c prepended to result of recursively calling cmds on remainder of string

Algebraic data type representing state of vending machine. deposits is Currency deposits not yet spent on purchases. coinbox contains all other Currency in the machine.

```haskell
{- tuple representing machine's current state:
   inventory of coins and bills, user's unspent total,
   count of vending items remaining in each slot -}
data MachineState =
  MachineState{coinbox :: [Currency],
                           deposits :: [Currency],
                           itemCounts :: [(Int, Int)]}
```

Originally just quantity, now itemCounts contains (quantity, price) for each slot. I should have renamed field and used type synonyms to make this more clear.

```
{- process machine transitions, taking initial state,
   list of commands, output list, and returning tuple
   of new state and output -}
machine :: MachineState -> [String] -> [String] ->
(MachineState, [String])
machine t [] os = (t, os)
machine t@(MachineState coinbox deposits itemCounts)
(c:cs) os =
  case stripPrefix               " c of
    Just a -> l                        a in
              i                        ounts then

              e
                                       CT CODE"])
    Nothing -> case c of
     "NICKEL" -> machine t {deposits =
                   Nickel : deposits} cs os
     "DIME" -> machine t {deposits =
                   Dime : deposits} cs os
```

First equation handles end of command list, returning tuple of new state and accumulated output.

```
{- process machine tr
   list of commands,
   of new state and
machine :: MachineSt
(MachineState, [String])
machine t [] os = (t, os)
machine t@(MachineState coinbox deposits itemCounts)
(c:cs) os =
  case stripPrefix "GET-" c of
    Just a -> let slotnum = slotnumber a in
              if st    um < length itemCounts then
                 vend s      m t cs os
              else
                 [                           )
    Nothing -> case
     "NICKEL" -> ma

     "DIME" -> mach
                 DI
```

Deconstructing machine state using its data constructor. Also using t@ “as pattern” to efficiently refer to tuple without having to reconstruct it.

stripPrefix attempts to drop prefix (“GET-”) from string, returning Just remainder of string, or Nothing if string doesn't start with “GET-”

```
{- process machine transitions, taking initial state,
   list of                                       ng tuple
   of new
machine ::                                     ] ->
(MachineSt
machine t []
machine t@(Mach          e coinbox deposits itemCounts)
(c:cs) os =
  case stripP    efix "GET-" c of
    Just a -> let slotnum = slotnumber a in
              if slotnum < length itemCounts then
                 vend slotnum t cs os
              else machine t cs (os ++
                 ["REPORT INVALID PRODUCT CODE"])
    Nothing -> case c of
     "NICKEL" -> machine t {deposits =
                   Nickel : de
     "DIME" -> machine
                   Dime : d
```

if pattern matches Just a, then a contains remainder of String. Compute slot number based on a. If slot number within range, delegate to vend, else report error.

If result of stripPrefix “GET-” pattern matches Nothing, then try parsing other non GET- commands...

```
Nothing -> case c of
 "NICKEL" -> machine t {deposits =
                 Nickel : deposits} cs os
 "DIME" -> machine t {deposits =
                 Dime : deposits} cs os
 "QUARTER" -> machine t {deposits =
                 Quarter : deposits} cs os
 "DOLLAR" -> machine t {deposits =
                 Dollar : deposits} cs os
 "COIN-RETURN" -> mach  ne t {deposits =
                 []}    (os ++ map name deposits)
 "SERVICE" -> ser       t cs os
 _ -> machi
                                       D " ++ c])
```

t bound to MachineState via “as pattern”. Field update syntax used here to update single field in t in recursive call, prepending Currency to deposits.

```
Nothing -> case c of
  "NICKEL" -> machine
                Nicke
  "DIME" -> machine t {depo
                Dime : de       s} cs os
  "QUARTER" -> machine t {d      its =
                Quarter :    posits} cs os
  "DOLLAR" -> machine t {de  osits =
                Dollar : deposits} cs os
  "COIN-RETURN" -> machine t {deposits =
                []} cs (os ++ map name deposits)
  "SERVICE" -> service t cs os
  _ -> machine t cs (os ++
              ["REPORT DON'T UNDERSTAND " ++ c])
```

COIN-RETURN uses field update syntax to empty deposits, and put deposits back in to output stream via map name

Service commands delegate to service. Report error for any other input.

```
-- vend item
vend :: Int -> MachineState -> [String] -> [String] ->
(MachineState, [String])
vend slot t@(MachineState coinbox deposits itemCounts)
cs os =
  let unspent =   m $ map amount deposits
      newinv =         t==0 then Left ("REPORT " ++
                                          + " EMPTY")
                                          Counts) ++
                                          ce)] ++
                                          temCounts))
      (cou
  in case n
    Left a -> machine t cs (os ++ [a])
```

vend takes 4 arguments – slot #, machine state, input stream, output stream, returns tuple of new machine state, output stream.

Use helper amount with map and sum to compute unspent amount.

```
-- vend item
vend :: Int -> Machine                               ring] ->
(MachineState, [String])
vend slot t@(MachineState c      nbox deposits itemCounts)
cs os =
  let unspent = sum $ map amount deposits
      newinv = if count==0 then Left ("REPORT " ++
                                (slotname slot) ++ " EMPTY")
               else Right ((take slot itemCounts) ++
                            [(count - 1, price)] ++
                            (drop (slot+1) itemCounts))
      (count, price) = itemCounts !! slot
  in case newinv of
    Left a -> machine t cs (os ++ [a])
```

```
-- vend item
vend :: Int -> Machine                        ring] ->
(MachineState, [String])
vend slot t@(MachineState c   nbox deposits itemCounts)
cs os =
  let unspent = sum $ map amount deposits
      newinv = if count==0 then Left ("REPORT " ++
                         (slotname slot) ++ " EMPTY")
               else Right ((take slot itemCounts) ++
                           [(count - 1, price)] ++
                           (drop (slot+1) itemCounts))
      (count, price) = itemCounts !! slot
  in case newinv of
    Left a -> machine t cs        [a])
```

Use helper amount with map and sum to compute unspent amount.

Use (!!) index operator and deconstructing tuple pattern to retrieve count and price for selected slot.

```
-- vend item
vend :: Int -> Machine                        ring] ->
(MachineState, [String])
vend slot t@(MachineStat   inbox deposits itemCounts)
cs os =
  let unspent = sum $ map amount deposits
      newinv = if count==0 then Left ("REPORT " ++
                         (slotname slot) ++ " EMPTY")
               else Right ((take slot itemCounts) ++
                           [(count - 1, price)] ++
                           (drop (slot+1) itemCounts))
      (count, price) = itemCounts !! slot
  in case newinv of
    Left a -> machine t cs        [a])
```

Compute new inventory as Either String [(Int, Int)] to return an error if slot is empty or else return new inventory.

Use (!!) index operator and deconstructing tuple pattern to retrieve count and price for selected slot.

```
-- vend item
vend :: Int -> MachineState -> [String] -> [String] ->
(MachineState, [String])
vend slot t@(MachineState coinbox deposits itemCounts)
cs os =
  let unspent = sum $ map amount deposits
                                 eft ("REPORT " ++
                                ne slot) ++ " EMPTY")
                                 slot itemCounts) ++
                           [(count - 1, price)] ++
                           (drop (slot+1) itemCounts))
      (count,  rice) = itemCounts !! slot
  in case newinv of
    Left a -> machine t cs (os ++ [a])
```

If newinv pattern matches Left a, then return the error message bound to a from vend.

```
Right a -> if unspent < price then
              machine t cs (os ++
                ["REPORT INSUFFICIENT DEPOSIT"])
           else if unspent == price then
              machine t {coinbox = coinbox ++
                deposits, deposits=[],
                itemCounts=a} cs (os ++
                [slotname slot])
           else
                  change = getChange
                             (coinbox++deposits)
                             (unspent-price)
              in case change of
                Nothing -> machine t cs (os++
                  ["REPORT USE EXACT CHANGE"])
                Just c -> machine t {coinbox =
                  ((coinbox++deposits) \\ c),
                  deposits=[], itemCounts=a} cs
                  (os++(slotname slot):
                       (map name c))
```

If unspent deposits less than price, report error.

```
Right a -> if unspent < price then
              machine t cs (os ++
                ["REPORT INSUFFICIENT DEPOSIT"])
           else if unspent == price then
              machine t {coinbox = coinbox ++
                deposits, deposits=[],
                itemCounts=a} cs (os ++
                [slotname slot])
           lse
              let change = getChange
                             (coinbox++deposits)
                             (unspent-price)
                             of
                            achine t cs (os++
                            SE EXACT CHANGE"])
                            chine t {coinbox =
                            +deposits) \\ c),
                  deposits=[], itemCounts=a} cs
                  (os++(slotname slot):
                       (map name c))
```

If unspent deposits exactly equal price, move deposits to coinbox, adopt new inventory, and add item to output stream.

Otherwise, try to make change. If unsuccessful, report message to use exact change. If successfully made change, then move deposits to coinbox except for change, adopt new inventory, and add item and change to output stream.

(\\) is list difference operator.

```
                           DEPOSIT"])
                          hen
                          inbox ++
                          ,
     Iteme              es (os ++
     [slotnam          t])
else
   let change = getChange
                  (coinbox++deposits)
                  (unspent-price)
   in case change of
     Nothing -> machine t cs (os++
       ["REPORT USE EXACT CHANGE"])
     Just c -> machine t {coinbox =
       ((coinbox++deposits) \\ c),
       deposits=[], itemCounts=a} cs
       (os++(slotname slot):
            (map name c))
```

```
{- loop, parsing list of commands from stdin,
   sending it to machine, displaying output,
   and then tail recursing -}
vendmachine :: MachineState -> IO ()
vendmachine i =
  do input <- getLine
     let (newstate, st                          t) []
     mapM (\x -> putSt
     vendmachine newst
```

vendMachine takes a current MachineState and generates an IO action that must run within the IO monad.

IO action composed
of sequence of
other IO actions

```
{- loop, parsing st of commands from stdin,
   sending it to achine, displaying output,
   and then tail recursing -}
vendmachine :: MachineState -> IO ()
vendmachine i =
  do input <- getLine
     let (newstate, strs) = machine i (cmds input) []
     mapM (\x -> putStrLn ("-> " ++ x)) strs
     vendmachine newstate
```

First get line from
stdin, bind it to input

```
{- loop, parsing list of c             n,
   sending it to machine, d    aying output,
   and then tail recursin    }
vendmachine :: MachineSt    te -> IO ()
vendmachine i =
  do input <- getLine
     let (newstate, strs) = machine i (cmds input) []
     mapM (\x -> putStrLn ("-> " ++ x)) strs
     vendmachine newstate
```

Next, call pure functional code to parse cmds from input and process them, returning new machine state and output stream.

```
{- loop, parsing list of comm
   sending it to machine, dis
   and then tail recursing -}
vendmachine :: MachineState ->     ()
vendmachine i =
  do input <- getLine
     let (newstate, strs) = machine i (cmds input) []
     mapM (\x -> putStrLn ("-> " ++ x)) strs
     vendmachine newstate
```

```
{- loop, parsing list of
   sending it to machine,
   and then tail recursin
vendmachine :: MachineSta
vendmachine i =
  do input <- getLine
     let (newstate, strs) = machine i (cmds input) []
     mapM (\x -> putStrLn ("-> " ++ x)) strs
     vendmachine newstate
```

Next, use mapM to apply anonymous function returning IO action over a list into an IO action performing all the output.

```
{- loop, parsing list of commands from stdin,
   sending it to machine, displaying output,
   and then tail recursing -}
vendmachine :: MachineState -> IO ()
vendmachine i =
  do input <- getLine
     let (newstate, strs) = machine i (cmds input) []
     mapM (\x -> putStrLn ("-> " ++ x)) strs
     vendmachine newstate
```

Finally, invoke the same computation again using the new current machine state.

Run vending machine, starting with initial inventory and pricing, and empty coinbox, no deposits.

```
{- run machine, starting off initially empty
   coinbox, empty deposits, three each of
   65 cents, $1.00 and $1.50 items -}
runMachine = vendmachine (MachineState [] []
                 [(3, 65), (3, 100), (3, 150)])
```

What's left? service. If you understand the rest, you should be able to figure out service.

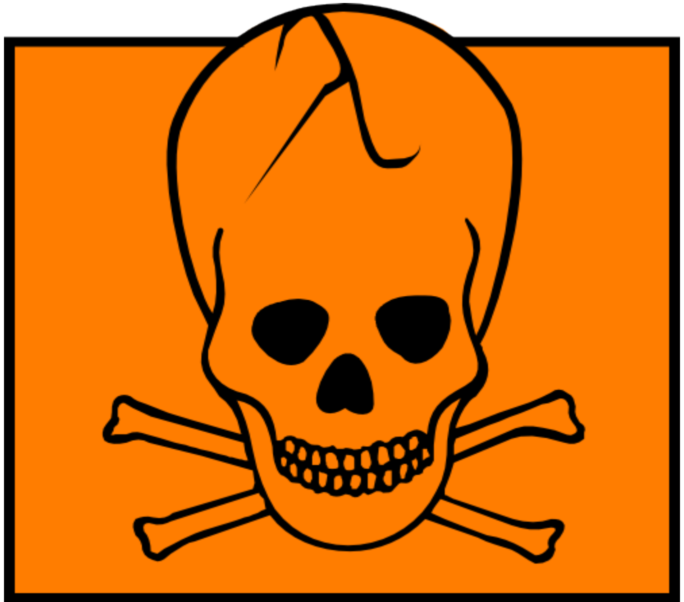
```
-- factorial of n (2 different implementations)
fact 0 = 1
fact n = n * fact (n - 1)
fact' n = foldr (*) 1 [1..n]
-- Fibonacci sequence: 0, 1, 1, 2, 3, ..
fib = 0 : 1 : zipWith (+) fib (tail fib)

-- powers of 2
pow2 = map (2^) [0..]

-- prime numbers
primes = 2:[x | x <- [3,5..],
            all (/= 0) $ map (x `mod`) [2..x-1]]

-- Simple stable sort
sort :: (Ord a) => [a] -> [a]
sort [] = []
sort (x:xs) = sort [l | l <- xs, l < x] ++ [x] ++
              sort [r | r <- xs, r >= x]
```

Goodies, Success Stories

- quickcheck – unit tests from assertions
- hackage – huge DB of contributed code
- cabal – nifty build system (or use ghc –make for simple projs)
- haddock – produce HTML docs ala Javadoc
- darcs – distributed source control
- FFI – interface with C, etc. code
- STM – Software Transactional Memory
- Parsec – easy parser generator
- Parallel, Concurrent, Template Haskell
- Scrap Your Boilerplate (SYB)
- HappS(tack) – applications server
- xmonad – tiny tiling X window manager
- pugs – First Perl 6 implementation
- Monadius, Frag – videogames
- Galois, Inc.
- Credit Suisse
- Functional Reactive Programming

Gotchas

- Silent Int overflow
- Error messages seem scary, at first
- Scary Category Theory, Abstract Algebra underpinnings
- Easy to write hard-to-decipher code
- Can get burned with “space leaks” -- sometimes laziness bites you and you have to force strict evaluation
- Learning curve may be daunting, especially if you dive head-first into category theory and reading whitepapers about folding, functors, morphisms, arrows, monads, etc.



Additional Resources – we only scratched the surface

- Real World Haskell, physical book or free online
- GHC library HTML docs
- http://www.haskell.org
- Haskell 98 Report (the standard)
- Typeclassopedia (in The Monad Reader, issue 13)
- Many other books, online tutorials, wikis, Haskell IRC
- Project Euler – gain fluency and strain your brain