Programming in Haskell
Aug-Nov 2015

LECTURE 15

OCTOBER 6, 2015

S P Suresh
Chennai Mathematical Institute

Modules

- A module consists of functions that are related to each other
- The name of the file must match the name of the module
- The module can be used in any other file in the same directory

Queue module

- The `Queue` module, defined in `Queue.hs`
- ```
  module Queue(NuQu(..), empty, isEmpty, enqueue,
  dequeue, show) where
  data Queue a = NuQu [a] [a]
  empty = ...
  isEmpty = ...
  enqueue = ...
  ```
- We can use the `Queue` module in another file in the same directory by adding the line `import Queue`
  ```

Queue module …

- Hiding mechanism: internal representation can be hidden from the outside world
- Only those functions listed inside `( )` in the module statement is visible on `import Queue`
- Auxiliary functions can be hidden from users of the module

Queue module ...

- Our Queue is not abstract enough, since the internal representation (as two lists) is visible outside
- Fix this by hiding the constructor
- ```
  module Queue(makeQueue, empty, isEmpty, enqueue,
  dequeue, show) where

  data Queue a = NuQu [a] [a]

  makeQueue :: [a] -> Queue a
  makeQueue l = Queue l []
  ```
  ```

Queue module ...

- ```
  empty :: Queue a
  empty = NuQu [] []
  ```

- ```
  isempty :: Queue a -> Bool
  isempty (NuQu [] []) = True
  isempty (NuQu _ _) = False
  ```

- ```
  enqueue x (NuQu ys zs) = NuQu ys (x:zs)
  ```

- ```
  dequeue (NuQu (x:xs) ys) = (x, NuQu xs ys)
  dequeue (NuQu [] ys) = (z, NuQu zs [])
            where z:zs = reverse ys
  ```

Queue module ...

- One can add instance declarations inside a module

- ```
  instance (Show a) => Show (Queue a) where
  show (NuQu xs ys) =
       show "{[" ++ printElems (xs ++ reverse ys)
                 ++"]}"
  ```

- ```
  printElems :: (Show a) => [a] -> String
  printElems [] = ""
  printElems [x] = show x
  printElems (x:xs) = show x ++ "->"
                           ++ printElems xs
  ```

- ```
  show "{[" ++
           intercalate "->" (map show (xs ++ reverse ys)
           ++ "]}"
  ```
  ```

Using the Queue module

- One uses the `Queue` module by adding `import Queue` at the start of a file (before defining any functions)
- The constructor `NuQu` and the function `printElems` are not available outside of `Queue.hs`
- One creates new queues by invoking the `makeQueue` function

```
newq = makeQueue [1..100]
```

A Stack module

* module Stack(Stack(..), empty, push, pop, isempty, show) where

```
Stack a = Stack [a]

empty = Stack []

push x (Stack xs) = Stack (x:xs)

pop (Stack (x:xs)) = (x, Stack xs)

isempty (Stack []) = True
isempty (Stack _) = False
```

A Stack module

- ```
  instance (Show a) => Show (Stack a) where
      show (Stack l) =
                        intercalate "->" (map show l)
  ```
  ```

Postfix expressions

- A **postfix expression** is an arithmetic expression where the operator appears after the operands
- No parentheses required in a postfix expression
- 3 5 8 * + = (3 (5 8 *) +) = 43
- 2 3 + 7 2 + - = ((2 3 +) (7 2 +) -) = -4

Postfix expressions

- Every bracket-free expression can be converted uniquely to a bracketed one
- Scan from the left
  - If it is a number, it is a standalone expression
  - If it is an operator, bracket it with the previous two expressions
- 3 5 8 * + = (3 (5 8 *) +) = 43
- 2 3 + 7 2 + - = ((2 3 +) (7 2 +) -) = -4

Evaluating postfix expressions

- Follow the bracketing algorithm and use a stack
- Scan from the left
  - If it is a number, **push** it onto the stack
  - If it is an operator
    - remove the top two elements of the stack
    - apply the operator on them
    - push the result onto the stack

A calculator program

- A postfix expression is a list of integers and operators
- We represent it as a list of **tokens**
-
```
import Stack

data Token = Val Int | Op Char
type Expr = [Token]
```

Evaluation: one step

- `evalStep :: Stack Int -> Token -> Stack Int`

- ```
  evalStep st (Val i) = push i st
  evalStep st (Op c)
     | c == '+' = push (v2+v1) st2
     | c == '-' = push (v2-v1) st2
     | c == '*' = push (v2*v1) st2
       where (v1,st1) = pop st
             (v2,st2) = pop st1
  ```
  ```

Evaluating an expression

- ```
  evalExp :: Expr -> Int
  evalExp = fst . pop . evalExp' empty
  ```

- ```
  evalExp' :: Stack Int -> Expr -> Stack Int
  evalExp' st [] = st
  evalExp' st (t:ts) =
      evalExp' (evalStep st t) ts
  ```

- Alternatively

  ```
  evalExp = fst . pop . (foldl' evalStep empty)
  ```

Maybe

- Functions in modules ought to be as general as possible
- No assumptions about usage
- ```
  max :: [Int] = Int
  max [x] = x
  max (x:xs)
     | x > y     = x
     | otherwise = y
     where y = max xs
  ```
- What is `max []` ?
  ```

Maybe

- Option 1:

- ```
  max :: [Int] = Int
  max [] = -1
  max [x] = x
  max (x:xs)
     | x > y     = x
     | otherwise = y
     where y = max xs
  ```

- `-1` is a default, works if the input list contains only nonnegative integers
  ```

Maybe

- Option 2:

- ```
  max :: [Int] = Int
  max [] = error "Empty list"
  max [x] = x
  max (x:xs)
     | x > y     = x
     | otherwise = y
     where y = max xs
  ```

- `error :: [Char] -> a` is a function that prints the error message supplied and causes an exception
  - Aborts execution
  ```

Maybe

- Neither option is robust when we define `max` inside a module
- No idea what the context the function would be used in
- Use the built in type constructor `Maybe`
- `Maybe a = Just a | Nothing`

Maybe

*
```
max :: [Int] = Maybe Int    -- inside a module
max [] = Nothing
max [x] = Just x
max (x:xs)
   | x > y     = Just x
   | otherwise = Just y
   where Just y = max xs
```

*
```
printmax :: [Int] -> String   -- outside the module
printmax l = case (max l) of
             Nothing -> "Empty list"
             Just x -> "Maximum = " ++ show x
```

Maybe

* Consider a table datatype that stores key-value pairs

* ```
  type Key = Int
  type Value = String
  type Table = [(Key, Value)]
  ```

* `myLookup :: Key -> Table -> Maybe Value`

* looks up the `value` corresponding to `key` in `table`, if `key` occurs in `table`
  ```

Maybe

- ```
  myLookup :: Key -> Table -> Maybe Value
  myLookup k []          = Nothing
  myLookup k ((k1,v1):kvs)
   | k == k1          = Just v1
   | otherwise            = myLookup k kvs
  ```

- Built-in function

  `lookup :: Eq a => a -> [(a,b)] -> Maybe b`

- More robust than returning error or some default value on absence of key
  ```

Summary

* Hiding implementation details using modules
* Examples - `Stack` and `Queue` modules
* Using `Stack`s to evaluate postfix expressions
* Use of `Maybe` for more robust implementations