Introduction To Haskell Programming

Prof. S. P. Suresh

Chennai Mathematical Institute

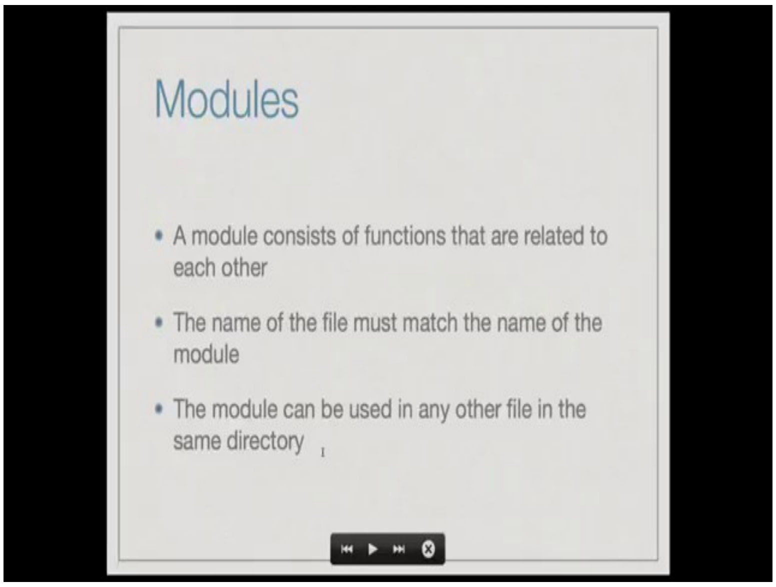
Module # 05

Lecture – 03

Modules

In this lecture, we shall introduce another abstraction device namely Modules.

(Refer Slide Time: 00:08)

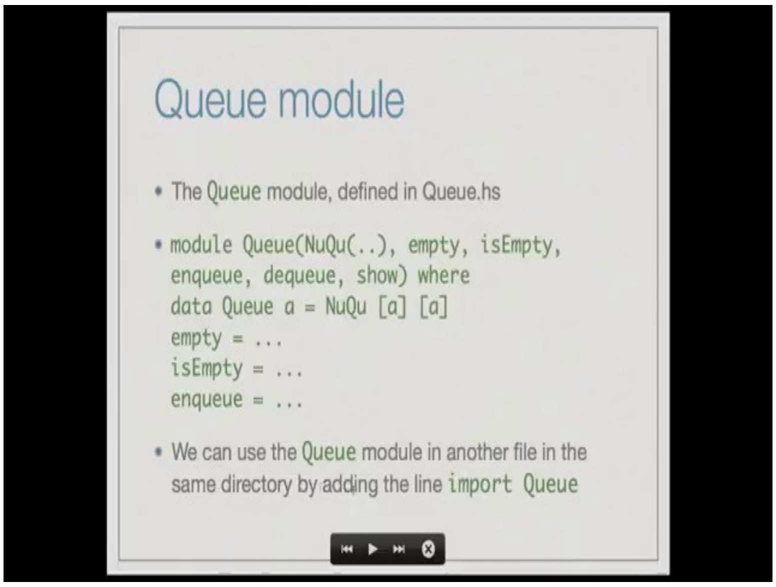


Modules

- 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

A module consists of functions that are related to each other and defined in a single file. 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 by using the command import as we shall see later.

(Refer Slide Time: 00:29)



Queue module

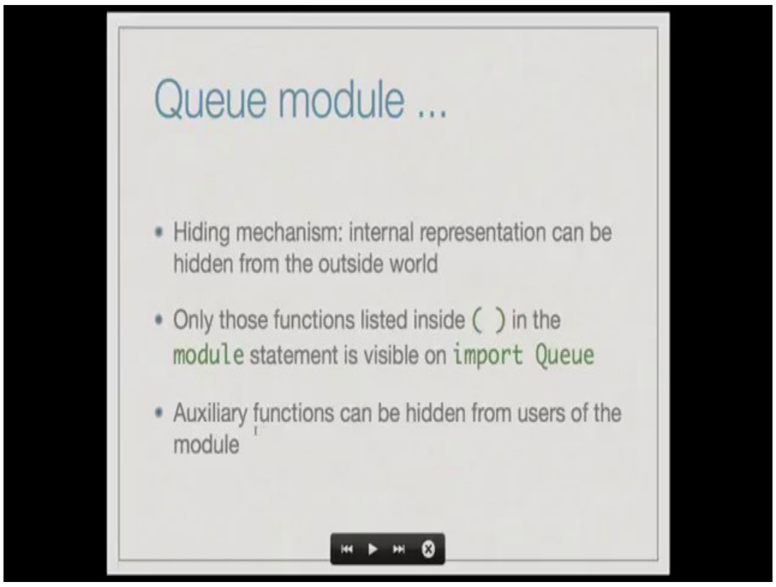
- The Queue module, defined in Queue.hs

-
  ```
  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`

Here is an example of a Queue module, the Queue module is defined in the file Queue.hs. The first line is as follows, module Queue(NuQu(..), empty, isEmpty, enqueue, dequeue, show) where data Queue a = NuQu [a] [a].... module Queue within parenthesis the names of many functions, 'where' and then you give the definition of functions and data types as usual., module Queue, within paranthesis, I have given the name of the constructor NuQu, notice the special syntax you have to say NuQu followed by parenthesis and within parenthesis you put two dots.

And then, these are some other functions that I want to be visible outside this file, empty, isEmpty, enqueue, dequeue, show etc. The definitions of empty, isEmpty, enqueue, etc were already been presented in the last lecture. We can use the Queue module in another file in the same directory by adding the line import Queue.

(Refer Slide Time: 01:34)

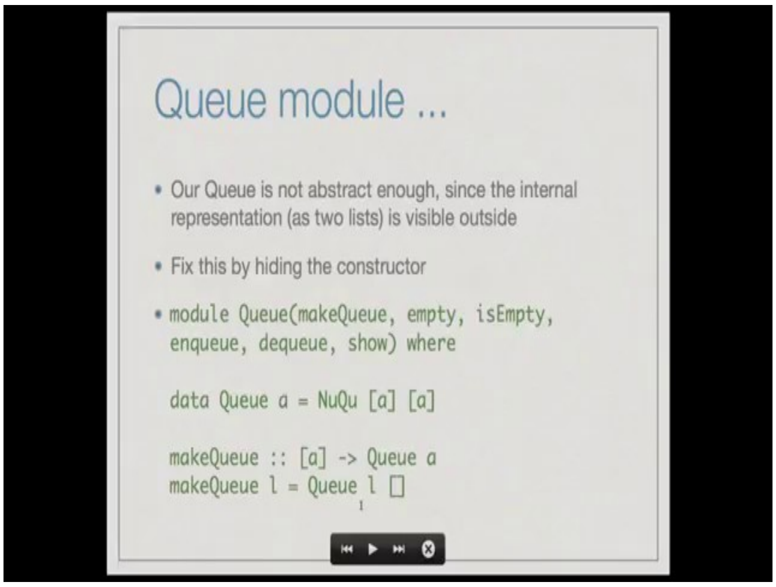


Queue module ...

- 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

A module acts as a hiding mechanism, the internal representation of the data types can be hidden from the outside world. Only those functions listed inside the parenthesis in the module statement are visible to another file, when it does not import Queue. This means that auxiliary functions can be hidden from users of the module.

(Refer Slide Time: 02:03)



Queue module ...

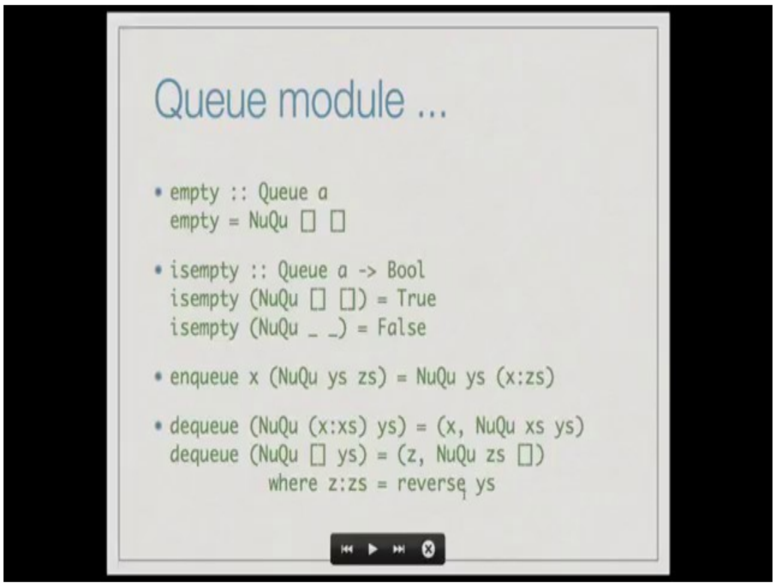
- Our Queue is not abstract enough, since the internal representation (as two lists) is visible outside
- Fix this by hiding the constructor

-
  ```
  module Queue(makeQueue, empty, isEmpty,
  enqueue, dequeue, show) where

  data Queue a = NuQu [a] [a]

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

Let us look our Queue module in more detail, our Queue is not abstract enough. Since, the internal representation is visible outside, recall that the queue is represented in terms of two lists, one half of the list and the reverse of the rest of the list. We can fix this by hiding the constructor, in this new declaration the NuQu function is not exported. Instead we have a makeQueue function, which takes any list of elements of type a, [a] and produces a Queue a.

The new declaration is as follows, module Queue(makeQueue, empty, isEmpty, enqueue dequeue, show) where data Queue a = NuQu [a] [a] and we have a new function makeQueue, which takes a lists of objects of type a and returns the Queue a, makeQueue of l is nothing but, Queue l empty list.

(Refer Slide Time: 03:10)



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
```

The other functions are as before, empty just returns a Queue, which is empty NuQu [] [], isempty will check whether a queue is empty, isempty (NuQu [] []) = True,

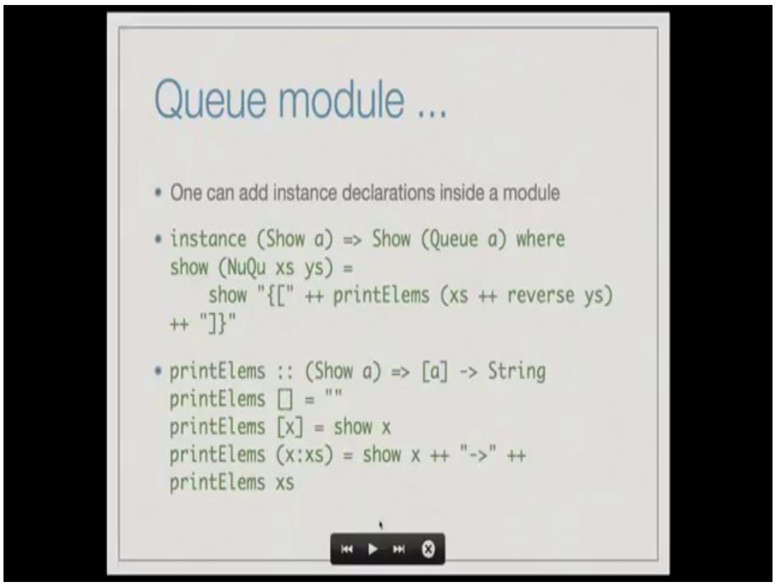
isempty NuQu of anything else is false : isempty (NuQu _ _) = False.

enqueue x (NuQu ys and zs) = NuQu ys (x:zs) .

Recall that we add x to the head of the second list, dequeue (NuQu (x:xs) ys) is nothing but, (x, NuQu xs ys), recall that in a dequeue we remove the element from the head of the first list.

And if the first list is empty, then we reverse the second list into the first list and extract the first element. So, dequeue (NuQu [] ys) = (z, NuQu zs []) where z:zs = reverse ys.

(Refer Slide Time: 04:21)



Queue module ...

- One can add instance declarations inside a module

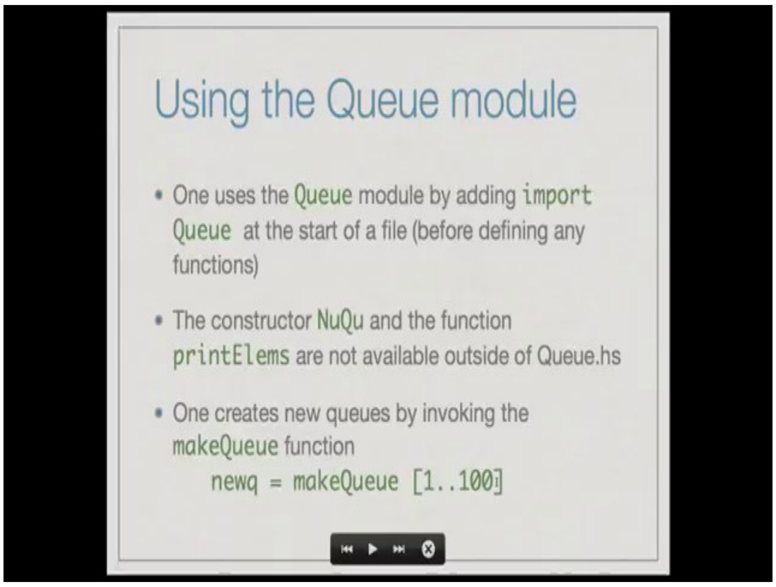
-
  ```
  instance (Show a) => Show (Queue a) where
  show (NuQu xs ys) =
      show "{[" ++ printElems (xs ++ reverse ys)
  ++ "]}"
  ```

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

One can also add instance declarations inside a module, here we see that as before we are declaring Queue a to be an instance of the type class Show provided a itself is a member of the type class show. instance(Show a) implies Show(Queue a) where the show function on NuQu xs and ys is just show of some fancy characters "{[" ++ print all the elements of the queue, printElems (xs ++ reverse ys) ++ "]}".

So, in our representation you have a brace followed by a square bracket followed by all the elements listed in order, followed by closing the brackets and the brace. PrintElems is a function from list a to string, [a] -> String, provided a is of type Show, a belongs to the type class Show. PrintElems of empty list [] is the empty string "", printElems of the singleton x is nothing but show x, printElems of x:xs is nothing but, show x ++ "->" ++ printElems xs.

in this case we have an arrow (Refer Slide Time: 05:44)



Using the Queue module

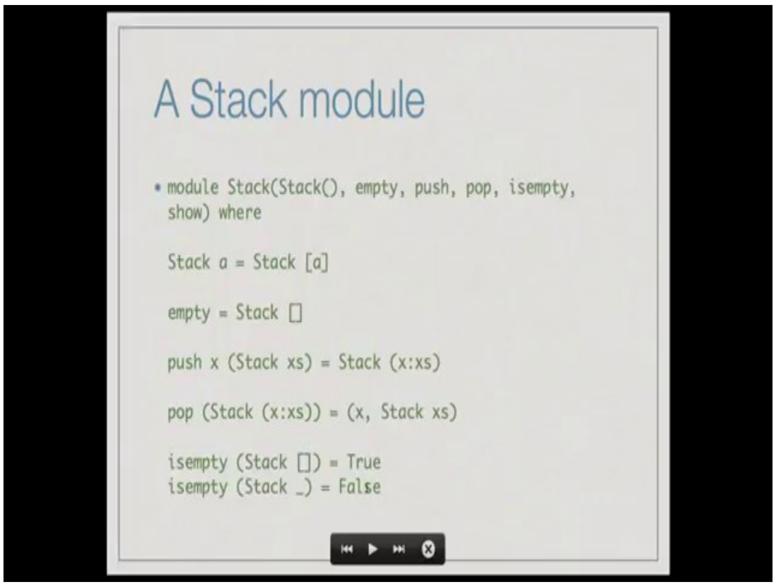
- One uses the Queue module by adding `import Queue` at the start of a file (before defining any functions)
- The constructor NuQu and the function `printElems` are not available outside of Queue.hs
- One creates new queues by invoking the makeQueue function

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

So, one uses the Queue module by adding import queue at the start of a file, before defining any functions in that file. When we do this, the constructor NuQu and the function printElems are not available outside of Queue.hs The constructor NuQu we are not making available externally, because we want to hide the internal representation and printElems is just some helper function that we need in order to define show.

So, therefore, there is no need for it to be visible outside the queue module. One creates new queues by invoking the makeQueue function, for instance you might say newq = makeQueue [1..100].

(Refer Slide Time: 06:32)



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
```

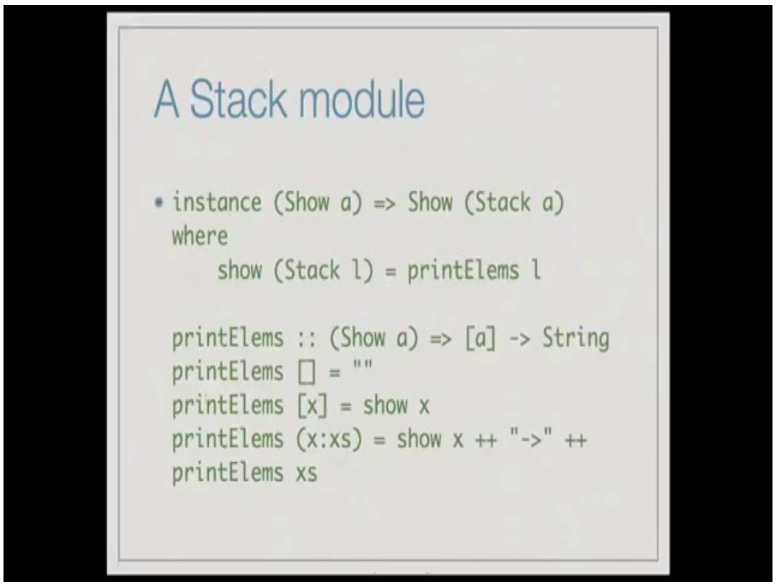
Here is another example a stack module, module stack, stack there are to be two dots here, stack… Here is, this is the data constructor and we have empty, push, pop, isempty, show, etc. We are exporting all these functions and the definitions are as before.

Stack a = Stack [a], Stack on the right hand side is the value constructor and then you have a list a. The empty , push and pop functions are as defined below.

empty = Stack [], push x (Stack xs) = Stack (x:xs), pop (Stack (x : xs)) =(x, Stack xs)

isempty checks whether a stack is empty, if on stack empty list, isempty(Stack []) = True, it will return true, for a stack anything else , isempty (Stack _) = False , it will return false.

(Refer Slide Time: 07:26)



A Stack module

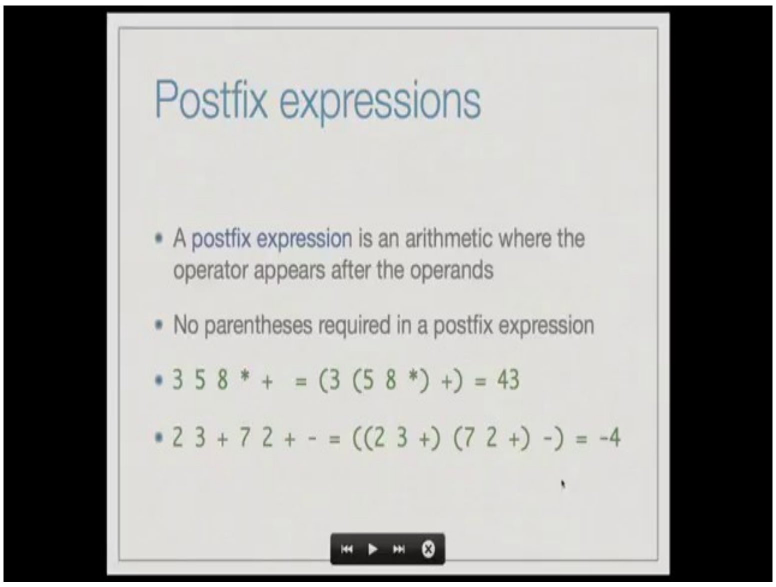
- instance (Show a) => Show (Stack a)

```
where
    show (Stack l) = printElems l

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

Similar, to queue we can also add instance declaration inside the stack module.

instance (Show a) => Show (Stack a), where show(Stack l) = printElems l, just like in the queue example. Here again we are printing the elements with arrows in between, this is exactly the same as the printElems function that we had earlier.

(Refer Slide Time: 07:52)



Postfix expressions

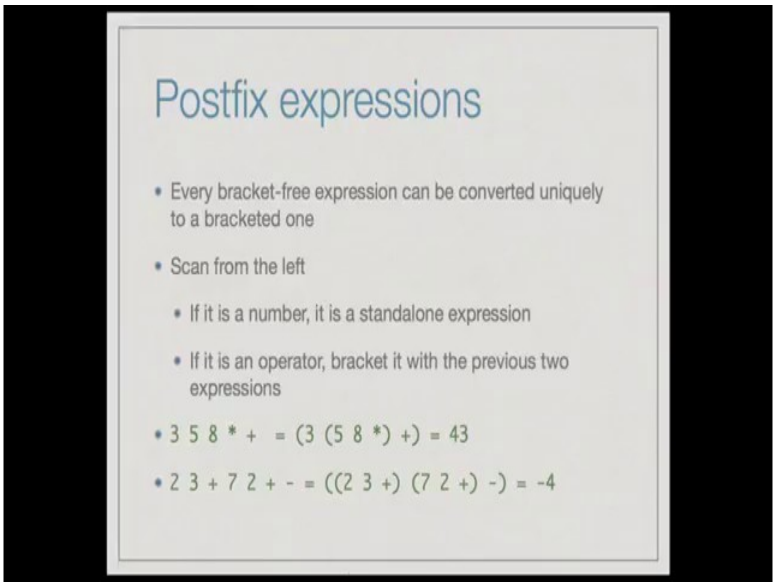
- A postfix expression is an arithmetic where the operator appears after the operands
- No parentheses required in a postfix expression
- 3 5 8 * + = (3 (5 8 *) +) = 43
- 2 3 + 7 2 + - = ((2 3 +) (7 2 +) -) = -4

Let us say an extended example of using stacks, in this example we will try to compute the value of postfix expressions. A postfix expression is an arithmetic expression, where the operator appears after the operands, importantly no parenthesis are required in a postfix expression. Here, is an example 3 5 8 * + is a postfix expression and the way to interpret it as that this star occurs after two numbers.

So, we have to take this as a sub expression this is the sub expression 5 8 *, which stands for 5*8. Then, we have a + following a string of numbers and operators, this is to be interpreted as the plus operator applied on the previous two expressions, the expression previous to + is 5 8 * and the expression previous that is 3. So, this whole thing stands for 3+ (5 * 8) =43, here is another example 2 3 + 7 2 +, so this + occurs after two numbers.

So, this is to be taken as one expression, 2 3 + which stands for 2+3. 7 2 + this stands for 7+2, which is 9 and then there is a minus this stands for the there are two expressions preceeding the minus. So, it is the first expression minus the second expression. So, 2 3 + is an expression 7 2 + is an expression and this expression followed by that expression minus is another expression and it stands for (2+3) - (7+2) namely -4.

(Refer Slide Time: 09:55)



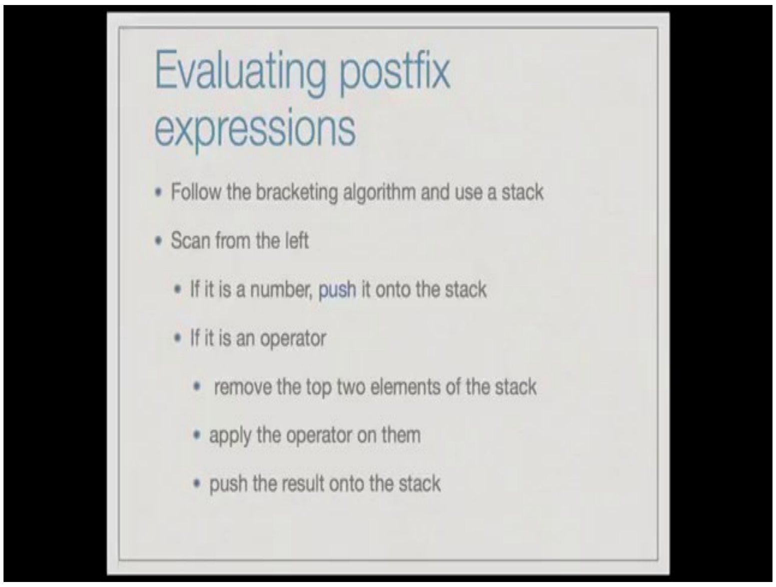
Postfix expressions

- 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

Every bracket free expression can be converted uniquely to a bracketed one that is the specialty of postfix expressions, the way to do it is to scan from left, if it is a number then it is a standalone expression, if it is an operator you bracket it with the previous two expressions. So, 3 is a standalone expression, 5 is a standalone expression, 8 is a standalone expression you encounter a * and this creates a new expression, which is got by applying by bracketing this with the previous two expressions namely 5 and 8.

So, (5 8 *) becomes a new expression. When you reach + you have two expressions before it, namely 3 and (5 8 *), so you bracket this plus along with this two. Similarly, in 2 3 + 7 2 + -, 2 is a standalone expression, 3 is a standalone expression, when you encounter the plus it creates a new expression, which is the previous two expressions bracketed along with plus. So, you get (2 3 +) you move on, 7 is a standalone expression, 2 is a standalone expression. When you encounter the plus (7 2 +) becomes an expression, when you encounter the minus here it is to be grouped with the previous two expressions, which is (2 3 +) and (7 2 +). So, ultimately you get a value of -4.

(Refer Slide Time: 11:18)



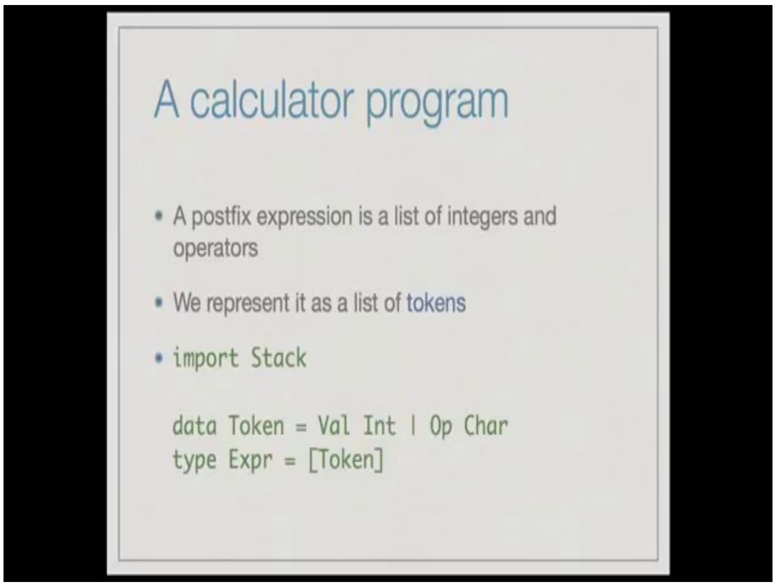
Evaluating postfix expressions

- 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

We shall now consider, how we can automatically evaluate postfix expressions. You can just follow in the bracketing algorithm and use a stack, here is how it proceeds. Scan from the left. if it is a number it is a standalone expression, so push it on to the stack, if it is an operator, we have to apply the operator to the previous two expressions and the strategy we follow is that we push all expressions on to the stack.

So, we remove the top two elements of the stack, which correspond to the two expressions that this operator has to be bracketed with, you apply the operator on the two expressions and push the result onto the stack. This is the overall strategy. Now, we will have to use a stack module and implement this algorithm.

(Refer Slide Time: 12:15)



A calculator program

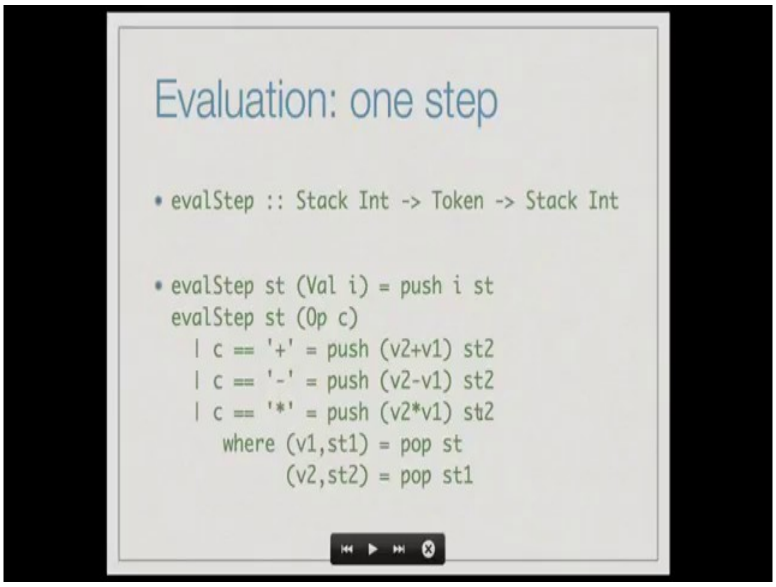
- A postfix expression is a list of integers and operators
- We represent it as a list of tokens
- `import Stack`

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

Here is how we would program this calculator, to calculate the value of postfix expressions. A postfix expression is given as a list of integers and operators. So, in this a list of a mixed type, we need to create a new data type, we will call these tokens. A token is either an integer or an operator, here is the definition. data Token= Val Int | Op Char, it is either an integer given by the data constructor Val or it is an operator.

And we denote the fact that is an operator by using the data constructor Op, if it is an integer it has a parameter, which is of type Int. If it is an operator it has a parameter, which is of type Char. And, importantly we need to import stack before we program this calculator and we can define a type synonym for list of tokens, type Expr = [Token]. So, an expression for us is just a list of tokens.

(Refer Slide Time: 13:27)



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
```

Recall that our strategy for evaluating an expression is to read the expressions from left to right and when we encounter a number to push it on to a stack and when we encounter an operator to apply the operator on the top two elements of the stack. This leads us to the function evalStep, which is one step in this computation. evalStep is a function that takes a stack of integers and a token as input and produces a stack of integers as output. evalStep st (Val i) =push i st.

Recall that Val is a data constructor that indicates that the token that we have encountered is an integer and the integer is given by i, which we push on to the stack. evalStep st (Op c) recall that Op is a data constructor that indicates that the token that we have encountered is a operator. And, the operator happens to be c and for simplicity we will assume that c is either

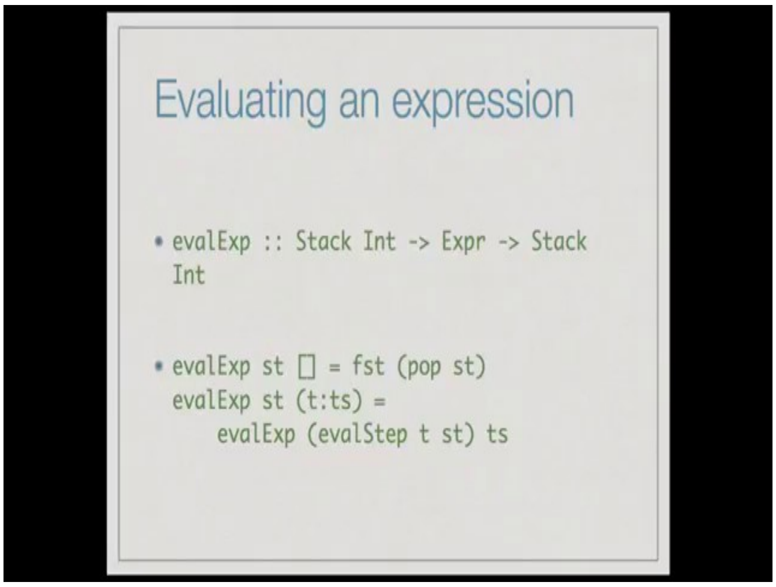
+ or - or *. the definition is as follows evalstep st (Op c) is push (v2+v1) on to st2, if c == ‘+’.

Here v1 is the value on top of the stack given by pop st and v2 is the value that is the second from the top of the stack, which is given by pop st1. recall that pop is a function that returns a pair, the top element to the stack and the stack that you obtain from removing the top element from the stack. So, st1 is, what you get if you remove the top element from st and st2 is, what you get if you remove the top element from st1. So, v1 and v2 are the top two elements in the stack and st2 is the stack that you obtain after you are remove the top two elements.

So, you remove the top two elements from the stack, add them and push them on to the stack, so push(v2 + v1) on to s2. If c is minus then you would push (v2 - v1) on to the stack, recall that if you encounter something of the form 3 5 -, what you intend is 3 - 5 and when you encounter 3, 3 would have been pushed on to the stack and when you encounter 5, 5 would have been pushed on top of 3.

So, the second element from the top is v2 and you have to subtract the top element from the second element of the stack. So, you have to push (v2 - v1) onto st 2, if c equals * then you do push of (v2*v1) onto st2.

(Refer Slide Time: 16:23)



Evaluating an expression

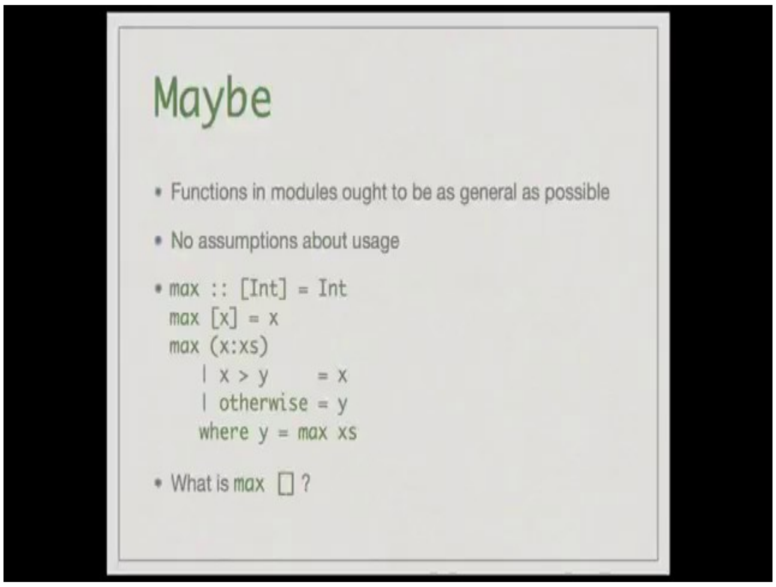
- evalExp :: Stack Int -> Expr -> Stack Int

-
  ```
  evalExp st [] = fst (pop st)
  evalExp st (t:ts) =
      evalExp (evalStep t st) ts
  ```

Once you have defined evalStep, we can actually evaluate an expression. evalExp :: Stack Int -> Expr -> Stack Int, evalExp is a function from Stack of Int to Expr to Stack of int. So, it is a function that takes two arguments as input, a stack of integers and an expression and it produces a stack of integers, typically the stack of integers that an expression takes is the empty stack. So, evalExp st [] this denotes that there is nothing to be done with the expression is pop st, but recall that pop st returns a the top of the stack and the remainder of the stack.

And first of pop st will be the top element of the stack. evalExp st (t:ts), t is a token followed by another bunch of tokens, evalExp (evalStep t st) ts. So, this evalStep of t st modifies this state of the stack st by taking into consideration one token t, and what we do is we evaluate the rest of the expression namely ts on the modified stack. So, this is how we define the calculator module based on the stack module.

(Refer Slide Time: 18:07)



Maybe

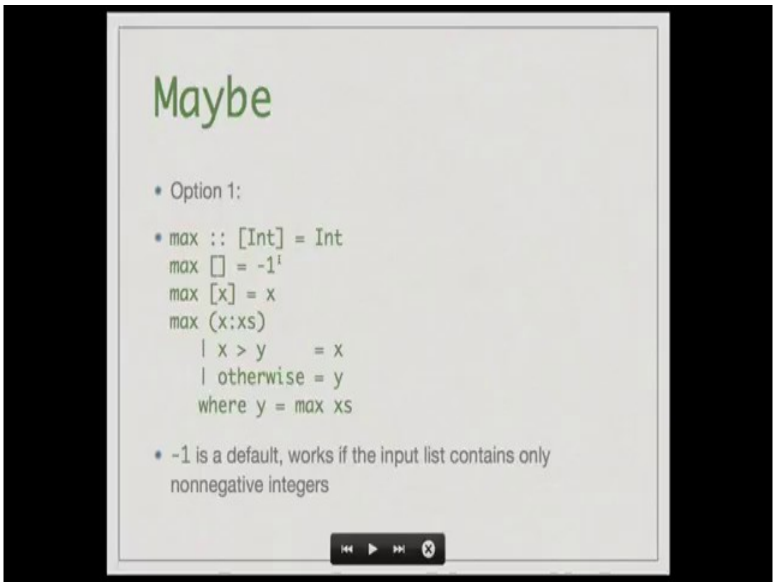
- Functions in modules ought to be as general as possible
- No assumptions about usage

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

After having seen a non-trivial example, we shall consider some more features on modules. One thing to remember in modules is that the functions that we provide in modules ought to be as general as possible. We should not make any assumptions about how the functions would be used by other modules that import this module. Here is an example, consider the function max which tries to find the maximum of a list of integers. max :: [Int] = Int. max is a function from list of Int to Int.

So, max of the singleton list [x] is just x. max of (x:xs) is x if x > y, otherwise is equal to y, where y is nothing but, max of xs. So, this max should not be confused with the built in function max, which takes two integers as arguments and produces the larger of the two integers. we use the name max here for simplicity. But, in this definition what is the case that we have to use for max of empty list, max [ ] ?

(Refer Slide Time: 19:28)



Maybe

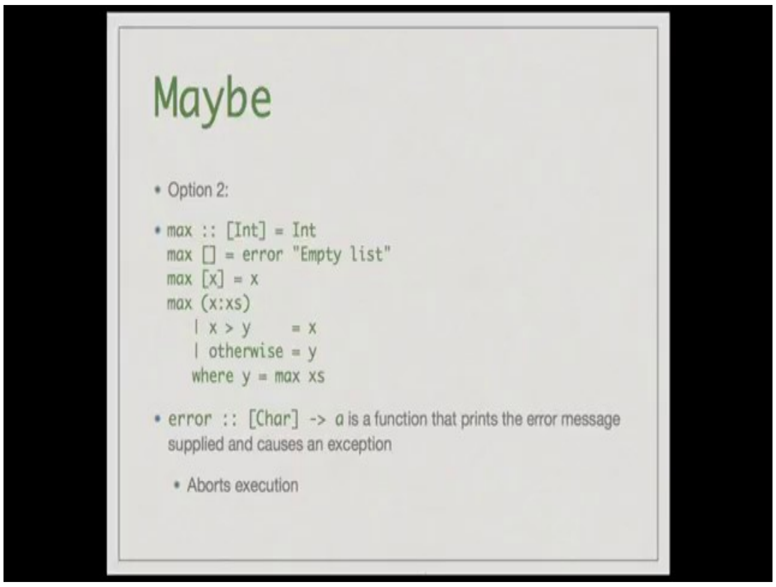
- Option 1:

```
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

One option is to define max of empty list to be some default value like -1, -1 is a default value and it works if the input list contains only non-negative integers. Then, this -1 serves to indicate that the list that was input was empty. But, what if the input list could also contain negative numbers, then this option does not work you might have to provide a different default.

(Refer Slide Time: 20:03)



Maybe

- Option 2:

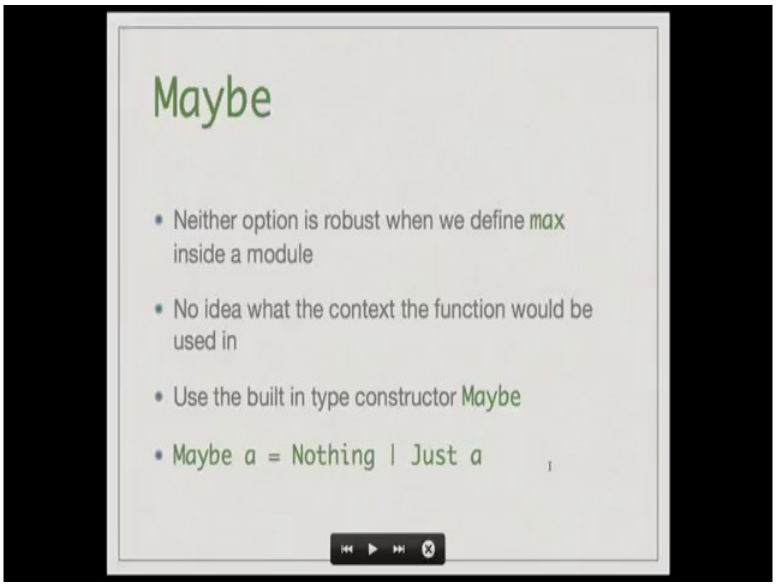
```
max :: [Int] = Int
max [] = error "Empty list"
max [x] = x
max (x:xs)
    | x > y     = x
    | otherwise = y
    where y = max xs
```

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

Another option which is slightly more robust is to define max of empty list to be error “Empty list”. error is a function that is of this type String -> a, any type a, whatsoever. It is a function that takes an error message as a parameter and it just causes an exception. it prints the error message out on screen and aborts the execution. So, this is slightly more robust, than what we had earlier, because if we default to -1, then if your input list does contain negative numbers, then you do not know whether a -1 was returned.

And because the list was empty or because, -1 was actually the maximum of the list. But even this is not very desirable. Because, the error function actually aborts the execution.

(Refer Slide Time: 21:11)

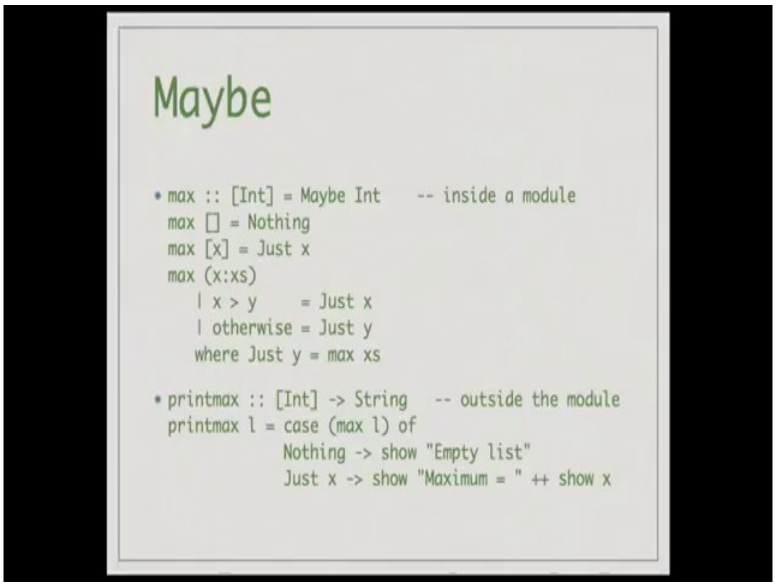


Maybe

- 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 = Nothing | Just a

We want something that works on most inputs and it also does not abort execution. The point is that we have no idea what the context is in which the function would be used. In these cases one can actually use the built in type constructor Maybe, Maybe is defined as follows. Maybe a = Nothing | Just a. So, no matter what type a is, for instance if a is Int we can build a type Maybe Int, which is given by nothing or just Int.

(Refer Slide Time: 22:03)



Maybe

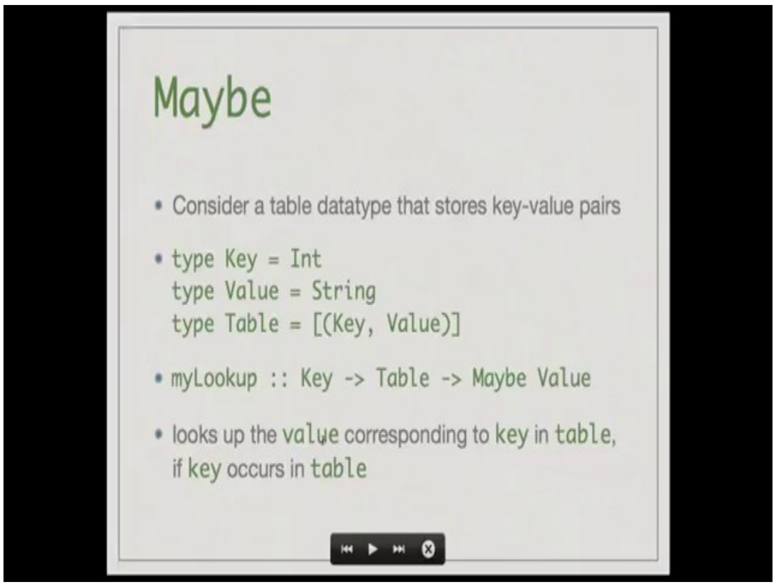
-
  ```
  max :: [Int] = Maybe Int    -- inside a module
  max [] = Nothing
  max [x] = Just x
  max (x:xs)
      | x > y     = Just x
      | otherwise = Just y
      where Just y = max xs
  ```

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

Here is how we use it. Here we define max to be a function from list of Int to Maybe Int, this is inside the module. max [] = Nothing; max [x] = Just x; max (x:xs) is if max of xs is Just y, and x> y, then max (x:xs) is Just x, otherwise it is Just y. This is how we would define this max function to return a Maybe Int rather than just an Int and outside the module we might have a routine like printmax, which takes a list of integers as input and produces a String as output. printmax of l is in case (max l) of Nothing then show "Empty list".

In case it is Just x then you show "Maximum = " ++ show x. this is a scenario, where we leave it to the user of the module to use the function in the appropriate manner. So, therefore, we should neither return a default value assuming the type of usage of the function nor should we abort. Here we see that even in the case when the list is empty the print max routine would like to print something meaningful rather than just abort.

(Refer Slide Time: 23:36)



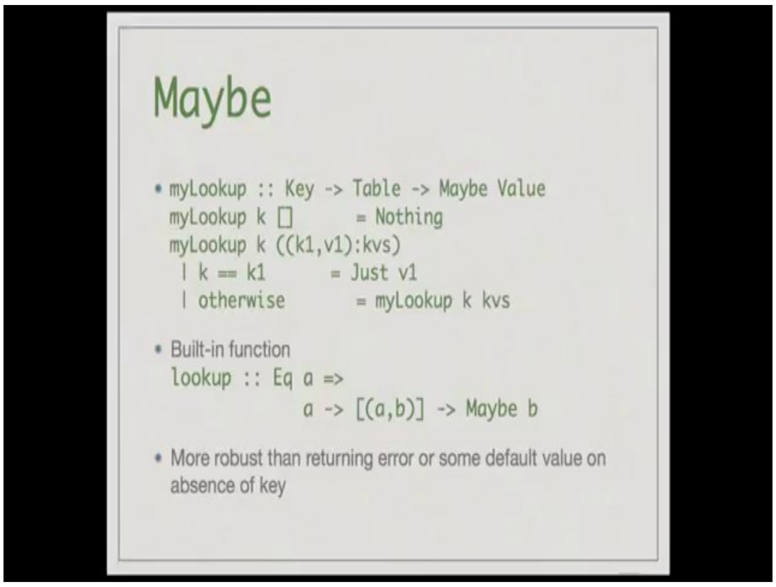
Maybe

- 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`

Here is another example of the use of Maybe. consider a table data type thar stores (Key, Value) pairs. type Key is just integer, let us say and type Value is just String, let us say. So, type Table is just list of (Key, Value) pairs. We want to define a function myLookup, which is a function of type Key -> Table -> Maybe value. What it does is to look up the value corresponding to the key in table, if key occurs in table. if key occurs in table it will print the corresponding value otherwise it will return nothing.

(Refer Slide Time: 24:16)

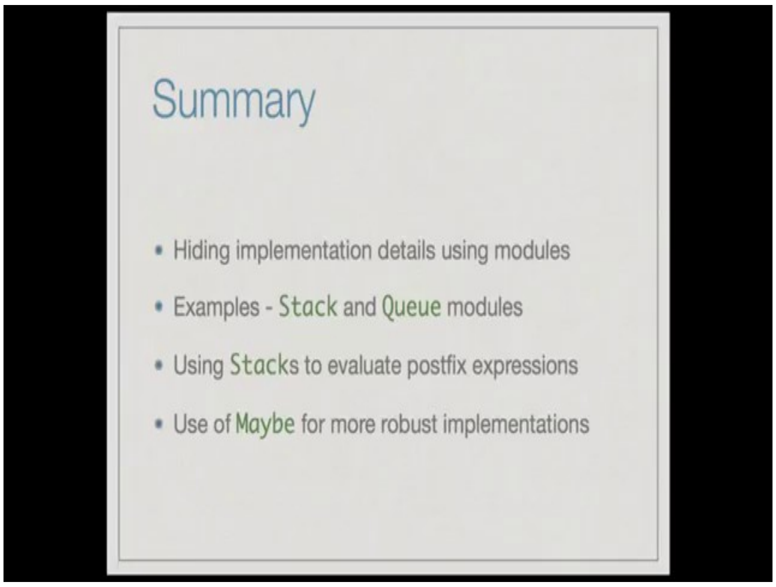


Maybe

-
  ```
  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

Here is the definition. myLookup of k empty list is Nothing. myLookup of k and ((k1,v1):kvs) is if k equals k1 you return just v1. Otherwise you do a myLookup of k in kvs in the remainder of the table. Actually, this behavior is given by the built in function lookup whose type is Eq a implies a ->[(a,b)] -> Maybe b. this is more robust than returning error or some default value on the absence of the key.

(Refer Slide Time: 25:02)



Summary

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

To summarize, we have introduced modules and shown how we can hide implementation details using modules. We have seen the examples of Stack and Queue modules , we have seen how to use the Stack module to do some non trivial computation namely to evaluate postfix expressions. We have also shown the use of Maybe for more robust implementations especially, when we define functions inside the modules.