Functional Programming in Haskell

Prof. Madhavan Mukund and S. P. Suresh

Chennai Mathematical Institute

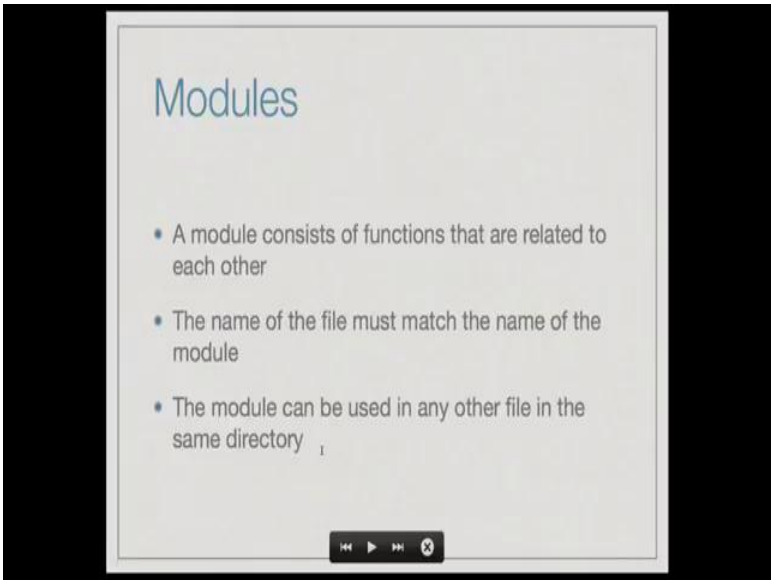
Module # 05

Lecture – 03

Modules

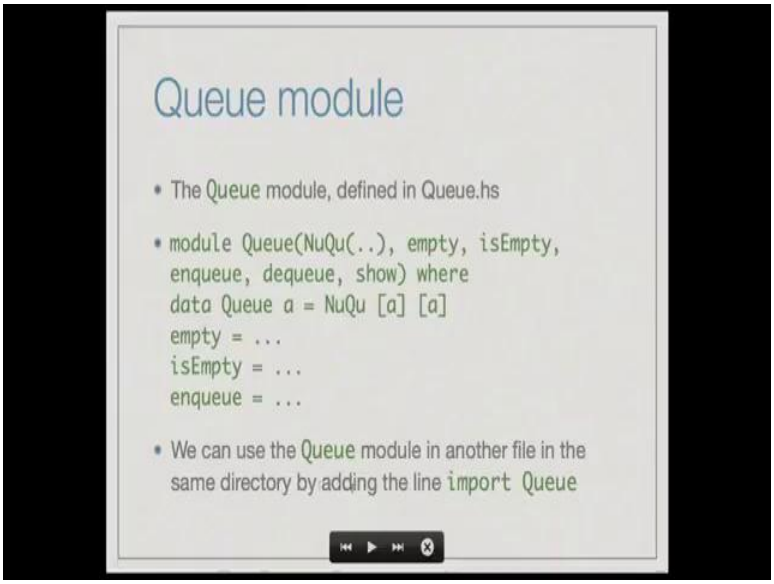
In this lecture, we should introduce another abstraction device namely Modules.

(Refer Slide Time: 00:08)



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 usual see later.

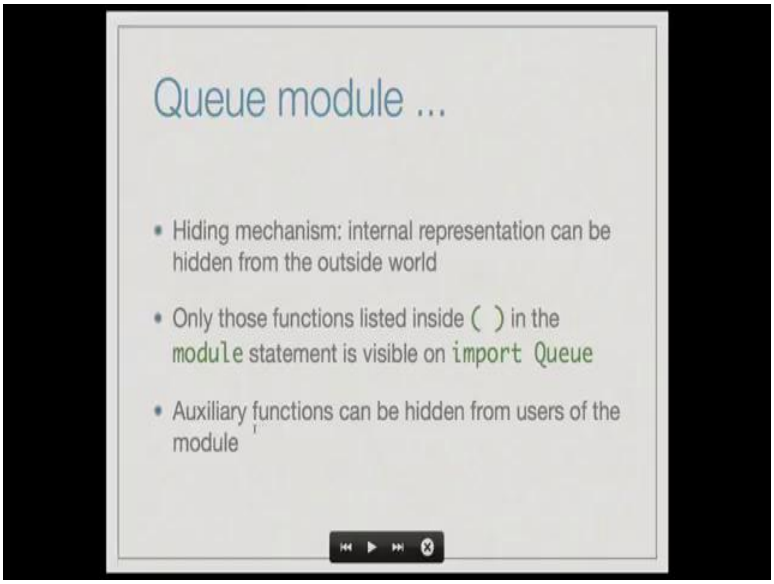
(Refer Slide Time: 00:29)



Here is an example of a queue module, the queue module is defined in the file queue dot h s. The first line is as follows, module q within parenthesis the names of many functions, where and then you give the definition of functions and data types as usual. Module q within parenthesis, I have given the name of the constructor NuQu, notice the special syntax you to say NuQu followed by parenthesis and within parenthesis you put two dots.

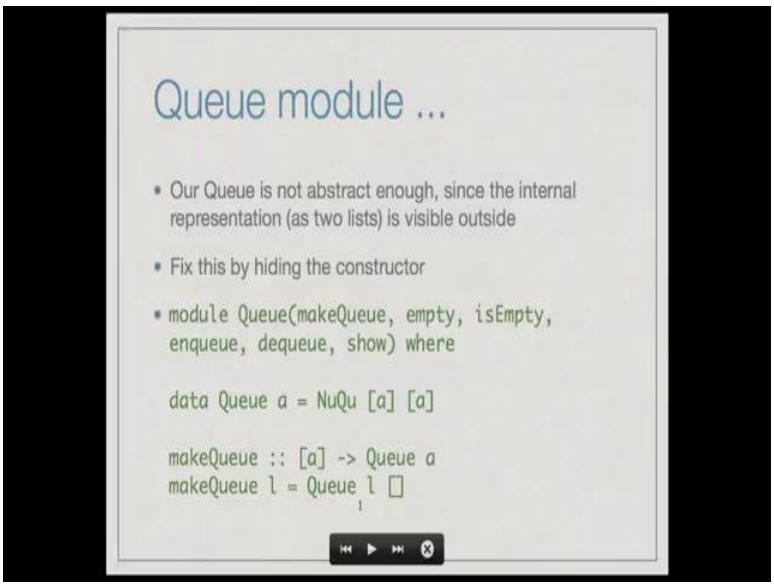
And then, these are some other functions that I want to be visible outside this file empty, isEmpty, enqueue, dequeue, show etcetera. The definitions of empty, isEmpty, enqueue, etcetera well already been presented in the last lecture. We can use the queue module in another file in the same directory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

(Refer Slide Time: 01:34)



A module access 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)

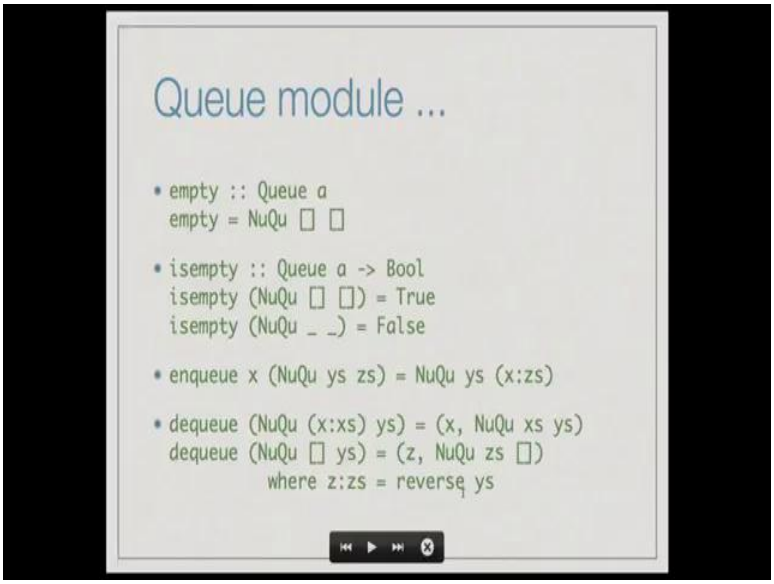


Let us look our queue module in more detail, our queue is not abstract enough. Since, the internal representation is visible outside, recall that the queue is represented in terms of two lists, one half of the list and the reverse of the rest of the list. We can fix this by hiding the constructor, in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

a queue a.

The new declaration is as follows, module queue, makeQueue, empty, isEmpty, enqueue dequeue, show, etcetera, where data queue a is as before NuQu list a list a and we have a new function makeQueue, which takes a lists of objects of type a and return the queue a, makeQueue of l is nothing but, queue l empty list.

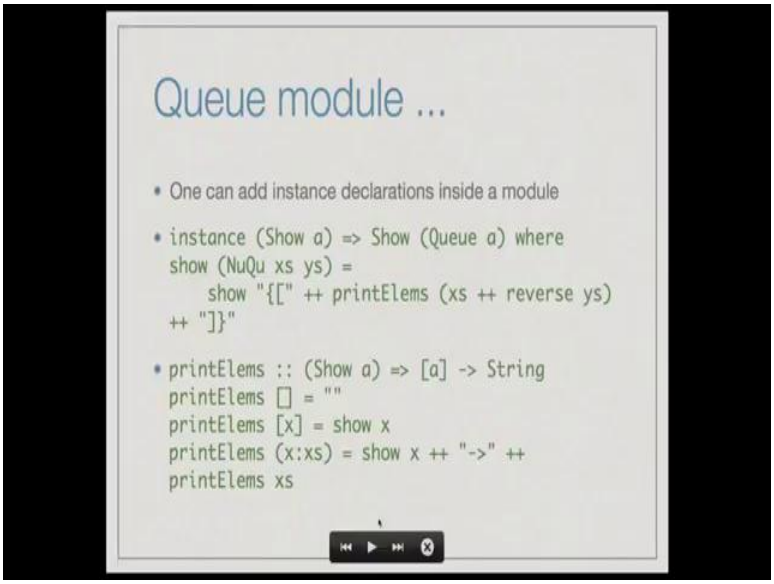
(Refer Slide Time: 03:10)



The other functions are as before, empty just returns a queue, which is empty NuQu empty list empty list, isempty will check whether a queue is empty, isempty NuQu of empty list empty list is true, isempty NuQu of anything else is false. Enqueue x of NuQu y s and z s is NuQu y is x colon z s. Recall that we add x to the head of the second list, dequeue of a NuQu x colon x s y s is nothing but, x comma NuQu x s and y s, 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 of NuQu empty list y s equals z comma NuQu of z s and empty list, where z colon z s is reverse of y s.

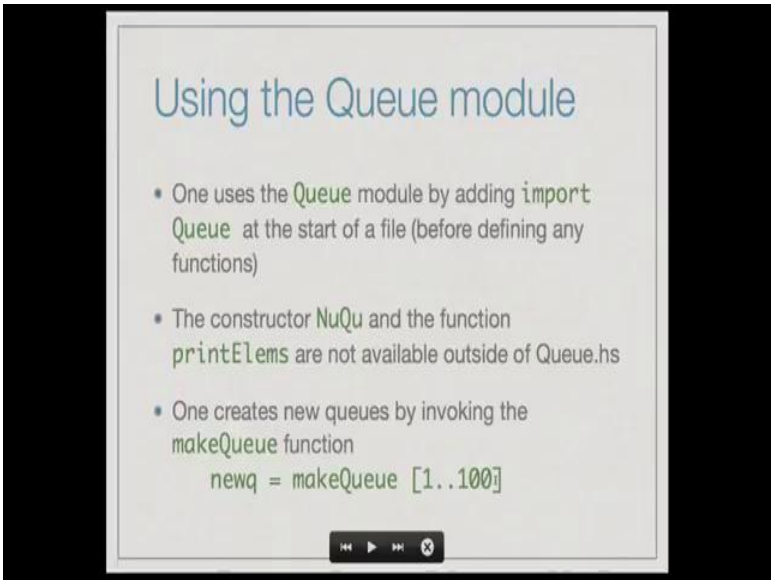
(Refer Slide Time: 04:21)



One can also add instance declaration 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 x s and y s is just show of some fancy characters plus print all the elements of the queue, printElems of x s plus, plus reverse of y s plus plus close the braces.

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 on the brace. PrintElems is a function from list a to 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 single term x is nothing but, show x printElems of x colon and x s is nothing but, show x plus in this case we have an arrow plus plus printElems of x s.

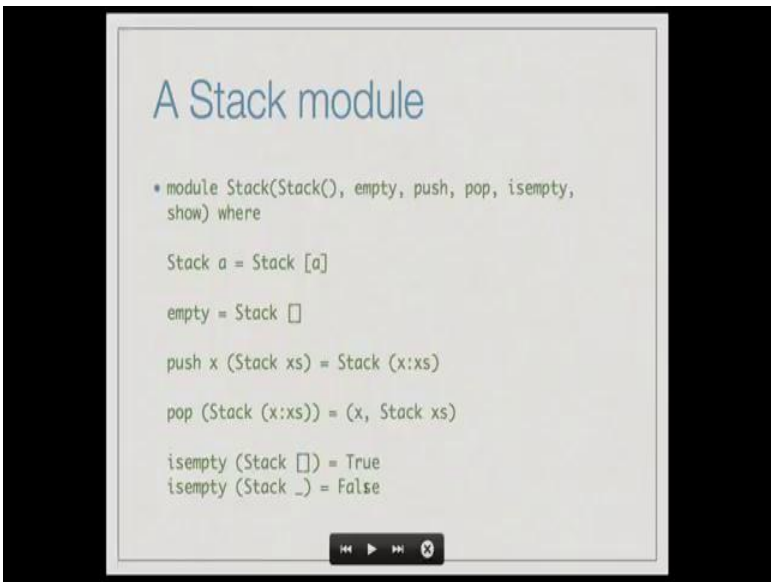
(Refer Slide Time: 05:44)



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 dot h s. 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.

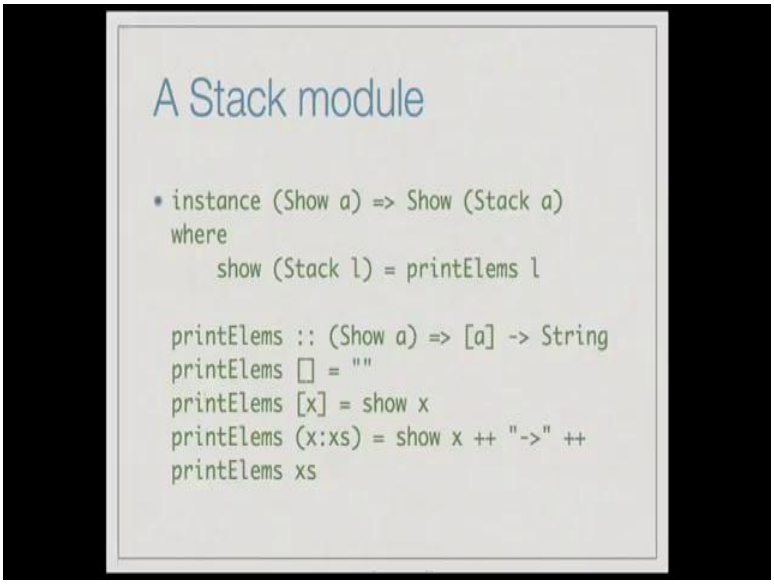
So, therefore, there is no need for it to be visible outside the queue module. One creates NuQu by invoking the makeQueue function, for instance you might say newq equals makeQueue of the list 1 to 100.

(Refer Slide Time: 06:32)



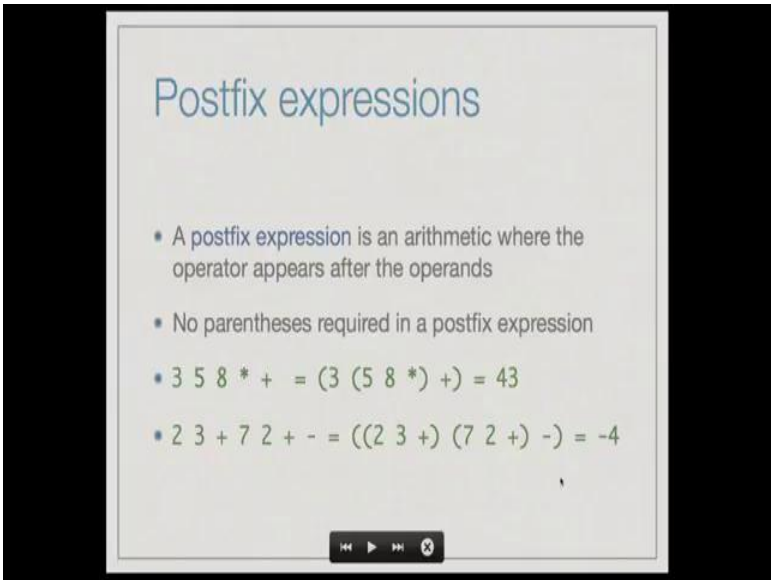
Here another example a stack module, module stack, stack there are to be two dots here, stack… Here is, this is the data constructor and we have empty, push, pop, isempty, show, etcetera. We are exporting all these functions and the definitions are asked before. Stack a is nothing but, stack this is the value constructor and then you have a list a, empty equals stack of empty list, push of x and stack x s is nothing but, stack of x colon x s, pop of stack of x colon x s equals x comma stack of x s, isempty checks whether a stack is empty, if on stack empty list it will return true, for a stack anything else it will return false.

(Refer Slide Time: 07:26)



Similar, to queue we can also add instance declaration inside the stack module instance show a implies show stack a, where show stack l equals printElems of 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)

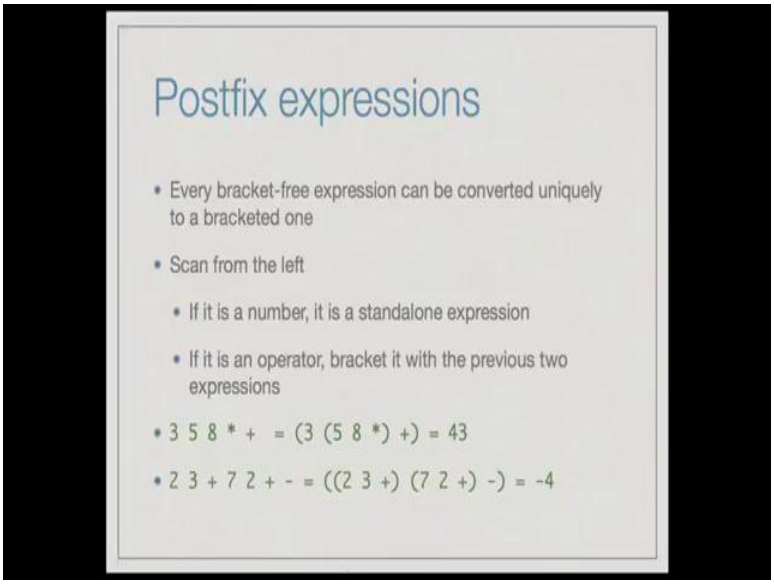


Let us say an extended example of using stacks, in this example we will try to compute the value of postfix x postfix expressions. A postfix expression is an arithmetic expression, where the operator appears after the operands importantly no parenthesis are required in a postfix expression. Here, is an example 3 5 8 star plus 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 star with stands for 5 into 8. Then, we have a plus 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 plus is 5 8 star and the expression previous that is 3. So, this whole thing stands for 3 plus 5 into 8 43, here is another example 2 3 plus 7 2 plus, so this plus occurs after two numbers.

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

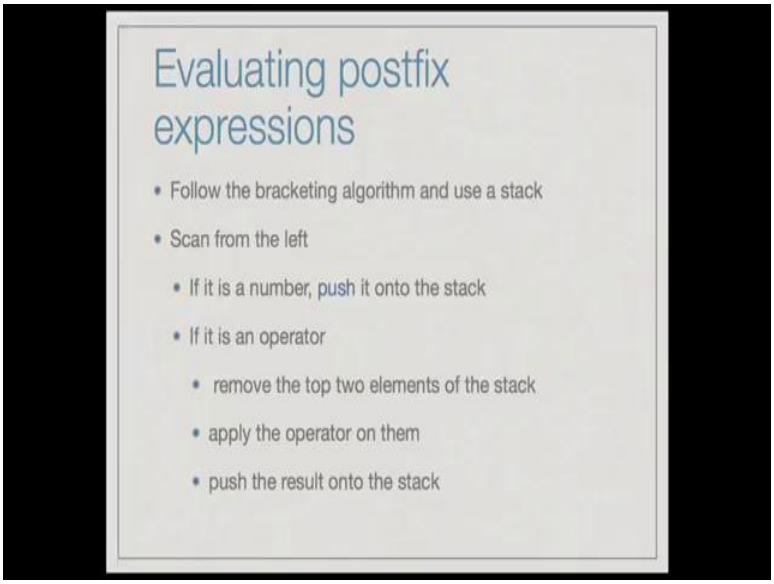
(Refer Slide Time: 09:55)



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 it is a standalone expression if it an operator you bracket it with the previous two expressions. So, 3 is standalone expression, 5 is standalone expression, 8 is a standalone expression you encounter a star and this creates a new expression, which is gotten by applying by bracketing this with the previous two expressions namely 5 and 8.

So, 5 8 star becomes a new expression, when you reach plus you will two expressions before it, namely 3 and 5 8 star, so you bracket this plus along with this two. Similarly, if 2 3 plus 7 2 plus minus 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 plus you move on 7 is a standalone expression, 2 is a standalone expression. When encounter the plus 7 2 plus becomes an expression, when encounter the minus here it test to be group two the previous two expressions, which is 2 3 plus and 7 2 plus. So, ultimately you get a value of minus 4.

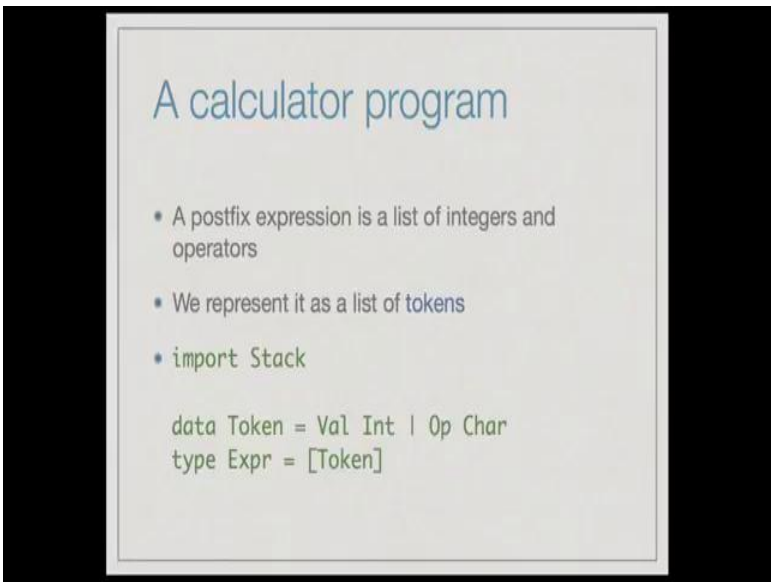
(Refer Slide Time: 11:18)



We shall no consider, how we can automatically evaluate postfix expressions you can just follow in the bracketing algorithm and use a stack, here is also 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, it 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 a bracket with you apply the operator on the two expressions and push the result onto the stack this is the overall strategy. Now, we have to use a stack module and implement this algorithm.

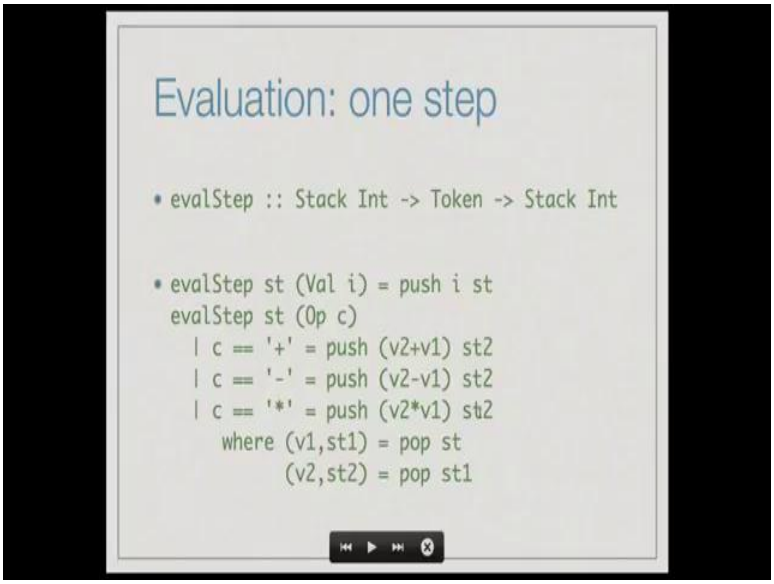
(Refer Slide Time: 12:15)



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 mixer type, we need to create a new data type, we will call these tokens a token is either are integer or an operator, here is the definition. Data token equals 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 a type char. And, importantly we need to imports stack before we program is calculated and we can define a type synonym for list of tokens type expression equals list of tokens. So, an expression for us is just a list of tokens.

(Refer Slide Time: 13:27)



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

Recall that val is a data constructor that indicates that the token that we have encounter is a number an integer and the integers given by i, which we push on to the stack. Evalstep st Op c recall that Op is a data constructor that indicates with the token, that we have encounter is a operator. And, the operator happens to be c and for simplicity we will

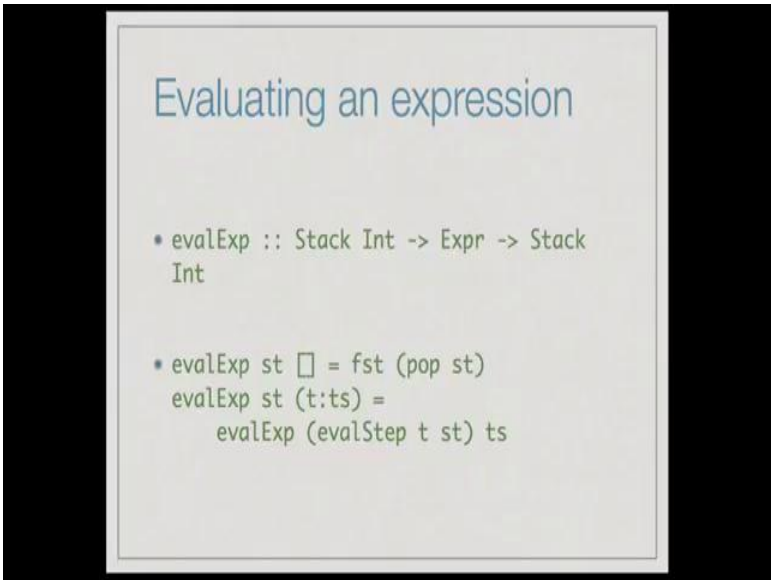
assume that c is either plus or minus or star the definition is as follows evalstep st of op c is push v 2 plus v 1 on to st 2, if c equals plus.

Here v 1, v 1 is the value on top of the stack given by st and v 2 is the value that the second from the top of the stack, which is given by pop st 1 recall that pop is a function that returns a pair that top element to the stack and the stack that you obtain from remove in the top element from the stack. So, st 1 is, what you get if you remove the top elements from st and st 2 is, what you get if remove the pop element from st 1. So, v 1 and v 2 are the top two elements the stack and st 2 is the stack that you obtain 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 v 2 plus v 1 on to s 2 st 2. I, c is minus then you would push v 2 minus v 1 on to the stack, recall that if you encounter something of the form 3 5 minus, what you intend is 3 minus 5 and when you encounter 3, 3 would push down to the stack and when encounter of 5 5 would I mean pushed on top of 3.

So, the second element from the top is v 2 and you have to subtract the top element from the second element of the stack. So, you have to push v 2 minus v 1 onto st 2, if c equals star then you do push of v 2 times v 1 onto st 2.

(Refer Slide Time: 16:23)

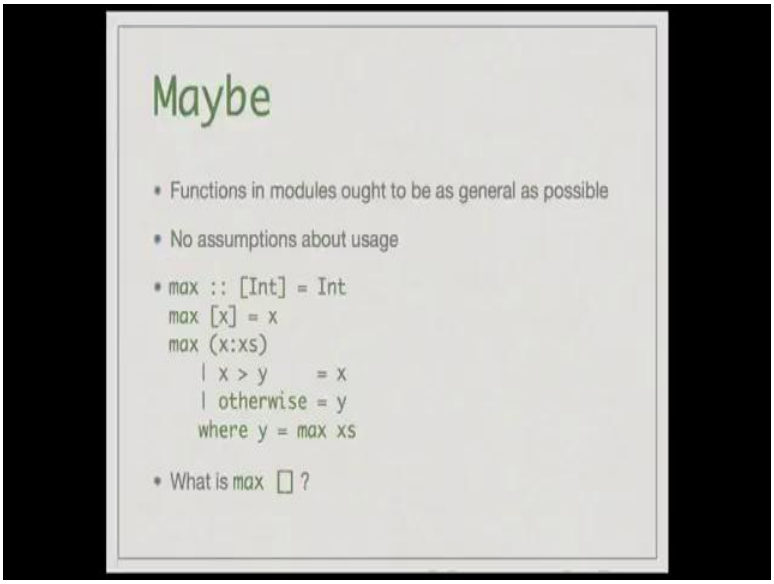


Once you have defined evalstep, we can actually evaluate the expression eval expression is a function from stack of Int to expression to stack of int. So, it is a function let takes two arguments as input a stack of integers and then expression and it produce as a stack

of integers, typically the stack of integers that an expression it takes is the empty stack. So, eval expression of st empty list 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 top st will be the top element of the stack eval expression st and t followed by t st is a token followed by another bunch of tokens is eval expression of evalstep of t st and then ts. So, this evalstep of t st modifies this state of the stack st by taken into consideration one token t, and what we do is we evaluate the rest of expression namely ts on the modified stack. So, this is, so we define the calculator module based on the stack module.

(Refer Slide Time: 18:07)

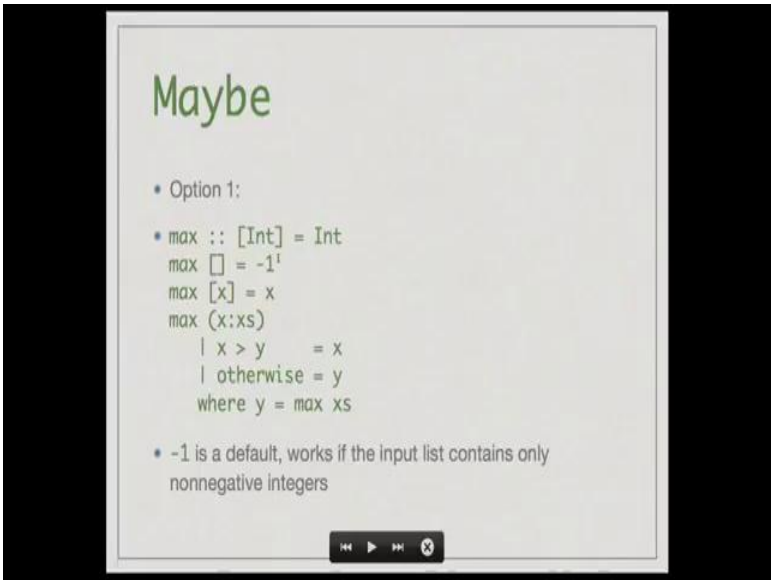


After having seen a non-trivial example, we shall consider some more features on modules. One thing to remember in a module is that the functions that we provide in modules ought to be as general as possible. We should not make any assumptions about how the functions would be used by other modules that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

So, max of the single term list x is just x max of x colon x s is x if x is greater than y, otherwise is equal to y, where y is nothing but, max of x s. So, this max should not we confuse with the built in function max, which takes two integers of the arguments and produces the larger of the two integers we use the name max of simplicity. But, in this

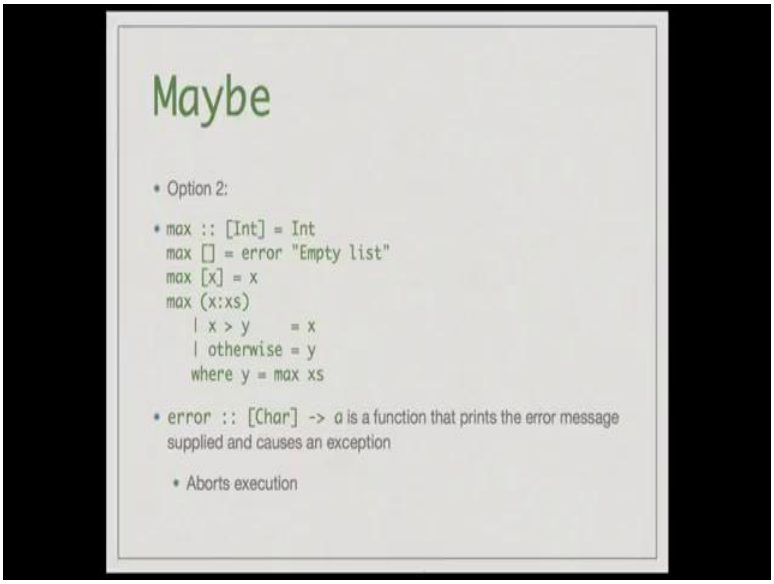
definition, what do we do if what, what is the case that we have to use for max of empty list.

(Refer Slide Time: 19:28)



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

(Refer Slide Time: 20:03)

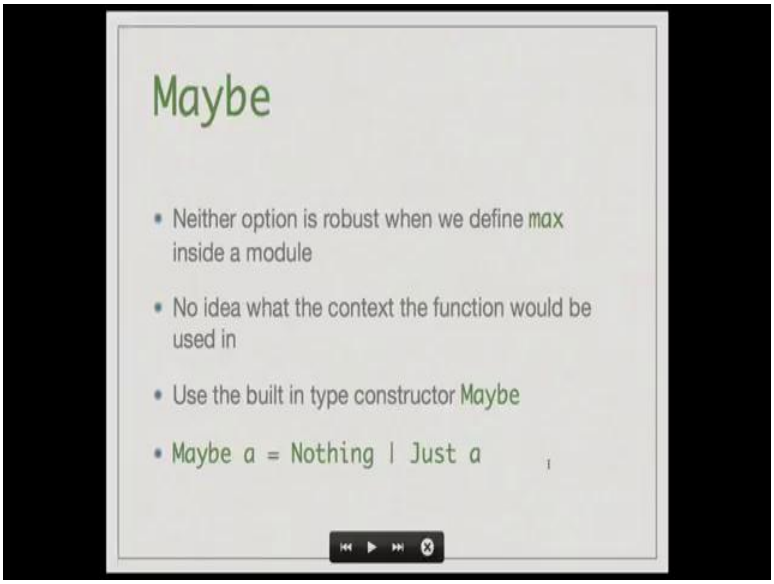


Another option which a slightly more robust is to define max of empty list to be error

empty list error is a function that is of this type string to a any type a, what is are there. It is a function that takes on error messages 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, then what we had earlier, because if we default come minus 1, then if your input list does contain a negative numbers, then you do not know whether a minus 1 was returned.

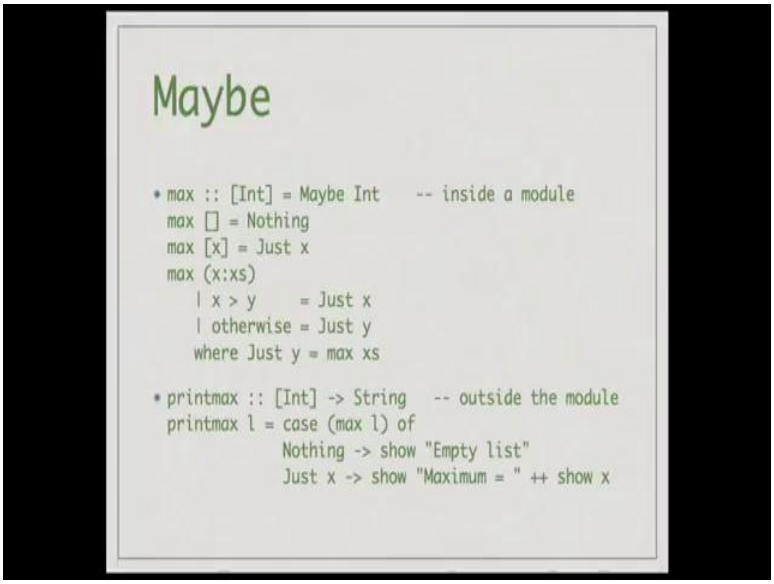
And because the list was empty or because, minus 1 was actually the maximum of the list would even this is not very desirable. Because, the error function actually aborts the execution.

(Refer Slide Time: 21:11)



We want something that works on most inputs and it also does not abort execution. In the point as, that we no idea what the context if, in which the function of the used in these cases one can actually use the built in type the built in type constructor maybe, maybe is defined as follows maybe a equals nothing or just a. So, no matter what type a is you can if for instance a is Int we can built a type maybe Int, which is given by nothing or just int.

(Refer Slide Time: 22:03)



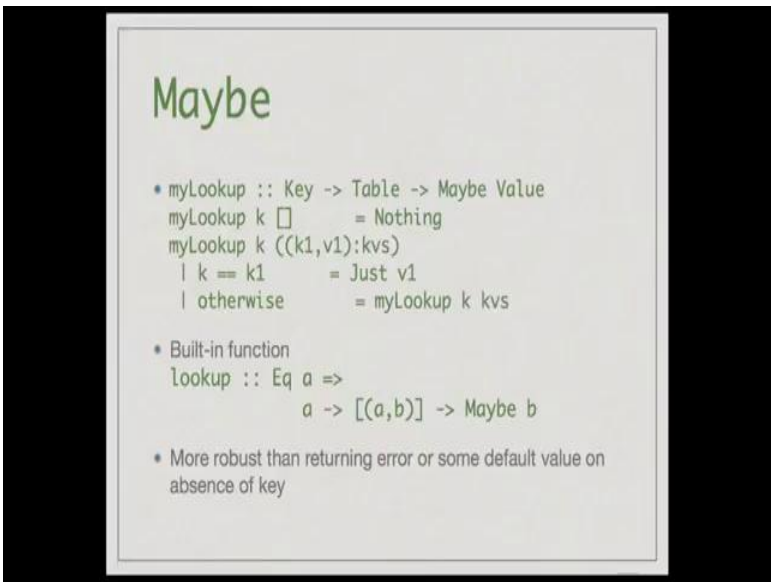
Here is, so we use it here we define max to be a function from list of Int to maybe Int this is inside the module. Max empty list equals nothing max is a singleton x equals just x max of x colon x s is if max of x s is just y and x is greater than y, then max of x colon x s is just x otherwise it is just y. This is on define this max function to return a maybe Int rather than just an Int and outside the module we might have a routine like print max, which takes a list of integers as input and produces string of output print max of l is in case max l is nothing then show the empty list.

In case it is just x then you show maximum equal to plus, plus show x this is a scenario, where we leave it 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 be 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)

Here is another example of the use of maybe consider a table data type the 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 arrow table arrow maybe value. What it does is to look up, the value corresponding to the key in table, if key occurs in table it will print the corresponding value otherwise it will return nothing.

(Refer Slide Time: 24:16)



Here is the definition myLookup of k empty list is nothing myLookup of k and k 1 v 1 followed by kvs is if k equals k 1 you return just v one. 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 look up whose type is Eq a implies a arrow list of pairs a comma b arrow maybe b this is more robust then returning error or some default value on the absence of the key.

(Refer Slide Time: 25:02)

To summaries, we introduce modules and show how we can hiding implementation details using modules. You seen the examples of stack and queue modules we have seen how to use the stack module to do some non trivial competition namely the evaluate postfix expressions. We have also shown the use of maybe for more robust implementations especially, when we define functions inside the modules.