Introduction to Programming: Lecture 10

K Narayan Kumar

Chennai Mathematical Institute

http://www.cmi.ac.in/~kumar

10 Sep 2013

Evaluating postfix expressions

- Follow the bracketing algorithm, and evaluate each expression as it is created.

Evaluating postfix expressions

- Follow the bracketing algorithm, and evaluate each expression as it is created.

  `3 5 8 * +`

Evaluating postfix expressions

- Follow the bracketing algorithm, and evaluate each expression as it is created.

  `3 5 8 * +`

  `3`

Evaluating postfix expressions

- Follow the bracketing algorithm, and evaluate each expression as it is created.

  `3 5 8 * +`

  `3 5`

Evaluating postfix expressions

- Follow the bracketing algorithm, and evaluate each expression as it is created.

  3 5 8 * +

  3 5 8

Evaluating postfix expressions

- Follow the bracketing algorithm, and evaluate each expression as it is created.

  `3 5 8 * +`

  `3 (5 8 *)`

Evaluating postfix expressions

- Follow the bracketing algorithm, and evaluate each expression as it is created.

  `3 5 8 * +`

  `3 40`

Evaluating postfix expressions

- Follow the bracketing algorithm, and evaluate each expression as it is created.

  `3 5 8 * +`

  `(3 40 +)`

Evaluating postfix expressions

- Follow the bracketing algorithm, and evaluate each expression as it is created.

  3 5 8 * +

  43

Evaluating postfix expressions

- Follow the bracketing algorithm, and evaluate each expression as it is created.

  `3 5 8 * +`

  `43`

- Another example:

Evaluating postfix expressions

- Follow the bracketing algorithm, and evaluate each expression as it is created.

  `3 5 8 * +`

  `43`

- Another example:

  `2 3 + 7 2 + -`

Evaluating postfix expressions

- Follow the bracketing algorithm, and evaluate each expression as it is created.

  `3 5 8 * +`

  43

- Another example:

  `2 3 + 7 2 + -`

  2

Evaluating postfix expressions

- Follow the bracketing algorithm, and evaluate each expression as it is created.

  `3 5 8 * +`

  43

- Another example:

  `2 3 + 7 2 + -`

  2 3

Evaluating postfix expressions

- Follow the bracketing algorithm, and evaluate each expression as it is created.

  `3 5 8 * +`

  `43`

- Another example:

  `2 3 + 7 2 + -`

  `(2 3 +)`

Evaluating postfix expressions

- Follow the bracketing algorithm, and evaluate each expression as it is created.

  `3 5 8 * +`

  `43`

- Another example:

  `2 3 + 7 2 + -`

  `5`

Evaluating postfix expressions

- Follow the bracketing algorithm, and evaluate each expression as it is created.

  3 5 8 * +

  43

- Another example:

  2 3 + 7 2 + -

  5 7

Evaluating postfix expressions

- Follow the bracketing algorithm, and evaluate each expression as it is created.

  `3 5 8 * +`

  `43`

- Another example:

  `2 3 + 7 2 + -`

  `5 7 2`

Evaluating postfix expressions

- Follow the bracketing algorithm, and evaluate each expression as it is created.

  `3 5 8 * +`

  `43`

- Another example:

  `2 3 + 7 2 + -`

  `5 (7 2 +)`

Evaluating postfix expressions

- Follow the bracketing algorithm, and evaluate each expression as it is created.

  `3 5 8 * +`

  `43`

- Another example:

  `2 3 + 7 2 + -`

  `5 9`

Evaluating postfix expressions

- Follow the bracketing algorithm, and evaluate each expression as it is created.

  `3 5 8 * +`

  `43`

- Another example:

  `2 3 + 7 2 + -`

  `(5 9 -)`

Evaluating postfix expressions

- Follow the bracketing algorithm, and evaluate each expression as it is created.

  `3 5 8 * +`

  `43`

- Another example:

  `2 3 + 7 2 + -`

  `-4`

Evaluating postfix expressions

- Follow the bracketing algorithm, and evaluate each expression as it is created.

  `3 5 8 * +`

  `43`

- Another example:

  `2 3 + 7 2 + -`

  `-4`

Evaluating postfix expressions

- Follow the bracketing algorithm, and evaluate each expression as it is created.

  `3 5 8 * +`

  `43`

- Another example:

  `2 3 + 7 2 + -`

  `-4`

- Keep a stack of numbers.
  - If you see a number, push it on to the stack.
  - If you seen an operator, remove the top two elements from the stack, evaluate and push the result on the stack.

Programming the calculator in Haskell

- The structure of the program:

Programming the calculator in Haskell

- The structure of the program:
  - A module to manage stacks.

Programming the calculator in Haskell

- The structure of the program:
  - A module to manage stacks.
  - A module that handles expressions and their evalutation.

The Stack module

- Methods `empty`, `push`, `pop` and `isempty`.

The Stack module

- Methods `empty`, `push`, `pop` and `isempty`.
- As general a type as possible for `Stack`.

The Stack module

- Methods `empty`, `push`, `pop` and `isempty`.
- As general a type as possible for `Stack`.

```
data Stack a = Empty | Stack a (Stack a)
empty :: Stack a
empty = Empty

push :: a -> Stack a -> Stack a
push x st = Stack x st

pop :: Stack a -> (a, Stack a)
pop (Stack x st) = (x, st)

isempty :: Stack a -> Bool
isempty Empty = True
isempty _     = False
```

The Stack module

- Methods `empty`, `push`, `pop` and `isempty`.
- As general a type as possible for `Stack`.

```
data Stack a = Empty | Stack a (Stack a)
empty :: Stack a
empty = Empty

push :: a -> Stack a -> Stack a
push x st = Stack x st

pop :: Stack a -> (a, Stack a)
pop (Stack x st) = (x, st)

isempty :: Stack a -> Bool
isempty Empty = True
isempty _     = False
```

- It looks very much like a `list`!

The Stack module via lists

```haskell
data Stack a = Stack [a]

empty :: Stack a
empty = Stack []

push :: a -> Stack a -> Stack a
push x (Stack ls) = Stack (x:ls)

pop :: Stack a -> (a, Stack a)
pop (Stack (x:ls)) = (x, Stack ls)

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

The `Stack` Module

- The module exports the following

The `Stack` Module

- The module exports the following
- The data type `Stack` without any of its constructors.

The `Stack` Module

- The module exports the following
- The data type `Stack` without any of its constructors.
- The methods `empty`, `push`, `pop` and `isempty`

The `Stack` Module

- The module exports the following
- The data type `Stack` without any of its constructors.
- The methods `empty`, `push`, `pop` and `isempty`

```
module Stack(Stack(),empty,push,pop,isempty) where
data Stack a =  ...

empty :: Stack a
...

push :: a -> Stack a -> Stack a
...

pop :: Stack a -> (a, Stack a)
...

isempty :: Stack a -> Bool
...
```

The calculator module

- The postfix expression is a sequence of integers and operators.

The calculator module

- The postfix expression is a sequence of integers and operators.
- How do we represent it in Haskell?

The calculator module

- The postfix expression is a sequence of integers and operators.
- How do we represent it in Haskell?
- We use the word `Token` to denote an element of the expression

The calculator module

- The postfix expression is a sequence of integers and operators.
- How do we represent it in Haskell?
- We use the word `Token` to denote an element of the expression

```
data Token = Val Int | Op Char
```

The calculator module

- The postfix expression is a sequence of integers and operators.
- How do we represent it in Haskell?
- We use the word `Token` to denote an element of the expression

```
data Token = Val Int | Op Char

type Expr = [Token]
```

Evaluating an expression

- An one step evaluation function:

Evaluating an expression

- An one step evaluation function:

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

Evaluating an expression

- An one step evaluation function:

```
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

- An one step evaluation function:

```
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
```

- How to iterate this and evaluate the entire expression?

Evaluating an expression

Evaluating an expression

- The top of stack has the answer at the end.

Evaluating an expression

- The top of stack has the answer at the end.

  ```
  evalExp st [] = fst (pop st)
  ```

Evaluating an expression

- The top of stack has the answer at the end.

  `evalExp st [] = fst (pop st)`

- Otherwise, evaluate recursively using `evalStep`

Evaluating an expression

- The top of stack has the answer at the end.

  `evalExp st [] = fst (pop st)`

- Otherwise, evaluate recursively using `evalStep`

  `evalExp st (e:es)  = evalExp (evalStep st e) es`

Evaluating an expression

- The top of stack has the answer at the end.

```
evalExp st [] = fst (pop st)
```

- Otherwise, evaluate recursively using `evalStep`

```
evalExp st (e:es)  = evalExp (evalStep st e) es

evaluate exp = evalExp empty exp
```

Evaluating an expression

- The top of stack has the answer at the end.

```
evalExp st [] = fst (pop st)
```

- Otherwise, evaluate recursively using `evalStep`

```
evalExp st (e:es)  = evalExp (evalStep st e) es

evaluate exp = evalExp empty exp
```

- Now, we can use any implementation of the `Stack` and it works identically.

Queues

- Queues follow the First-in first-out rule.

Queues

- Queues follow the First-in first-out rule.
- Required operations

Queues

- Queues follow the First-in first-out rule.
- Required operations

```
emptyq :: Queue a
addq :: a -> (Queue a) -> (Queue a)
removeq :: (Queue a) -> (a,Queue a)
isemptyq :: (Queue a) -> Bool
```

Queues

- Queues follow the First-in first-out rule.
- Required operations

```
emptyq :: Queue a
addq :: a -> (Queue a) -> (Queue a)
removeq :: (Queue a) -> (a,Queue a)
isemptyq :: (Queue a) -> Bool
```

- Again, represent a queue using a list

```
data Queue a = Qu [a]

emptyq = Qu []
addq x (Qu xs) = (Qu (xs ++ [x]))
removeq (Qu (x:xs)) = (x,Qu xs)
isempty (Qu l) = (l == [])
```

Queues . . .

- Inserting takes $O(n)$ time!

Queues . . .

- Inserting takes $O(n)$ time!
- If we reverse the representation?

```
addq x (Qu xs) = (Qu (x:xs))
removeq (Qu xs) = ((last x),Qu (init xs))
```

Queues ...

- Inserting takes $O(n)$ time!
- If we reverse the representation?

```
addq x (Qu xs) = (Qu (x:xs))
removeq (Qu xs) = ((last x),Qu (init xs))
```

Now, removing an element takes $O(n)$ time.

Adding and removing n elements could take $O(n^2)$ time

Queues with two lists

Queues with two lists

- Use two lists

Queues with two lists

- Use two lists
- Split queue and store the rear in reversed order in a reversed

  Represent $[q_1, q_2, \ldots, q_n]$
  as $[q_1, q_2, \ldots, q_i]$, $[q_n, q_{n-1}, \ldots, q_{i+1}]$

Queues with two lists

- Use two lists
- Split queue and store the rear in reversed order in a reversed

  Represent $[q_1, q_2, \ldots, q_n]$
  as $[q_1, q_2, \ldots, q_i]$, $[q_n, q_{n-1}, \ldots, q_{i+1}]$
- `addq` adds an element at beginning of second list in time $O(1)$

Queues with two lists

- Use two lists
- Split queue and store the rear in reversed order in a reversed

  Represent $[q_1, q_2, \ldots, q_n]$
  as $[q_1, q_2, \ldots, q_i]$, $[q_n, q_{n-1}, \ldots, q_{i+1}]$
- `addq` adds an element at beginning of second list in time $O(1)$
- `removeq` removes the element at the beginning of first list in time $O(1)$, if this list is nonempty!

Queues with two lists

- Use two lists
- Split queue and store the rear in reversed order in a reversed

  Represent $[q_1, q_2, \ldots, q_n]$
  as $[q_1, q_2, \ldots, q_i]$, $[q_n, q_{n-1}, \ldots, q_{i+1}]$

- `addq` adds an element at beginning of second list in time $O(1)$
- `removeq` removes the element at the beginning of first list in time $O(1)$, if this list is nonempty!
- What happens if the first list is empty?

Queues with two lists

- Use two lists
- Split queue and store the rear in reversed order in a reversed

  Represent $[q_1, q_2, \ldots, q_n]$
  as $[q_1, q_2, \ldots, q_i]$, $[q_n, q_{n-1}, \ldots, q_{i+1}]$
- `addq` adds an element at beginning of second list in time $O(1)$
- `removeq` removes the element at the beginning of first list in time $O(1)$, if this list is nonempty!
- What happens if the first list is empty?
- If the first list is empty, reverse the second list on to the first list and then remove the first element.

Queues . . .

```
data Queue a = Nuqu [a] [a]
```

Queues . . .

```
data Queue a = Nuqu [a] [a]
```

- Definition of queue functions
- `addq` adds to second list

```
addq x (Nuqu ys zs)  = Nuqu ys (x:zs)
```

Queues ...

```
data Queue a = Nuqu [a] [a]
```

- Definition of queue functions
- `addq` adds to second list

```
addq x (Nuqu ys zs)  = Nuqu ys (x:zs)
```

- `removeq` takes from first list, reversing elements from second list into first list if necessary

```
removeq (Nuqu (x:xs) ys)  = (x,Nuqu xs ys)
removeq (Nuqu [] ys)  = removeq (Nuqu (reverse ys) [])
```

Queues . . .

```
data Queue a = Nuqu [a] [a]
```

- Definition of queue functions
- `addq` adds to second list

```
addq x (Nuqu ys zs)  = Nuqu ys (x:zs)
```

- `removeq` takes from first list, reversing elements from second list into first list if necessary

```
removeq (Nuqu (x:xs) ys)  = (x,Nuqu xs ys)
removeq (Nuqu [] ys)  = removeq (Nuqu (reverse ys) [])
```

- If we add n elements, we get a queue `Nuqu [] [qn,...,q1]`

Queues ...

```
data Queue a = Nuqu [a] [a]
```

- Definition of queue functions
- `addq` adds to second list

```
addq x (Nuqu ys zs)  = Nuqu ys (x:zs)
```

- `removeq` takes from first list, reversing elements from second list into first list if necessary

```
removeq (Nuqu (x:xs) ys)  = (x,Nuqu xs ys)
removeq (Nuqu [] ys)  = removeq (Nuqu (reverse ys) [])
```

- If we add n elements, we get a queue `Nuqu [] [qn,...,q1]`
  - Next `removeq` takes $O(n)$ time to reverse the second list

Queues . . .

```
data Queue a = Nuqu [a] [a]
```

- Definition of queue functions
- `addq` adds to second list

```
addq x (Nuqu ys zs)  = Nuqu ys (x:zs)
```

- `removeq` takes from first list, reversing elements from second list into first list if necessary

```
removeq (Nuqu (x:xs) ys)  = (x,Nuqu xs ys)
removeq (Nuqu [] ys)  = removeq (Nuqu (reverse ys) [])
```

- If we add n elements, we get a queue `Nuqu [] [qn,...,q1]`
  - Next `removeq` takes $O(n)$ time to reverse the second list
  - After one `removeq`, we have `Nuqu [q2,...,qn] []`
  - Next $n-1$ `removeq` operations take time $O(1)$!

Amortised analysis

- How many times is an element touched.

Amortised analysis

- How many times is an element touched.
  - Once when it is inserted (into the second list)

Amortised analysis

- How many times is an element touched.
  - Once when it is inserted (into the second list)
  - Twice when it moves from the second list to the first list.

Amortised analysis

- How many times is an element touched.
  - Once when it is inserted (into the second list)
  - Twice when it moves from the second list to the first list.
  - Once when it is removed (from the first list)

Amortised analysis

- How many times is an element touched.
  - Once when it is inserted (into the second list)
  - Twice when it moves from the second list to the first list.
  - Once when it is removed (from the first list)
- Each element can be touched only four times.

Amortised analysis

- How many times is an element touched.
  - Once when it is inserted (into the second list)
  - Twice when it moves from the second list to the first list.
  - Once when it is removed (from the first list)
- Each element can be touched only four times.
- In any sequence of N instructions at most N elements are involved.

Amortised analysis

- How many times is an element touched.
  - Once when it is inserted (into the second list)
  - Twice when it moves from the second list to the first list.
  - Once when it is removed (from the first list)
- Each element can be touched only four times.
- In any sequence of N instructions at most N elements are involved.
- Any sequence of N instructions can take only $O(N)$ steps!

The `Set` datastructure

- Maintain a collection of distinct elements and support the following operations

The `Set` datastructure

- Maintain a collection of distinct elements and support the following operations
  - `insert` : inserts a given value into the set.

The `Set` datastructure

- Maintain a collection of distinct elements and support the following operations
  - `insert` : inserts a given value into the set.
  - `delete` : deletes a given value from the set.

The `Set` datastructure

- Maintain a collection of distinct elements and support the following operations
  - `insert` : inserts a given value into the set.
  - `delete` : deletes a given value from the set.
  - `search` : checks whether a given value is an element of the set.

The Set datastructure

- Maintain a collection of distinct elements and support the following operations
  - `insert` : inserts a given value into the set.
  - `delete` : deletes a given value from the set.
  - `search` : checks whether a given value is an element of the set.

```
data Eq a => Set a = Set [a]
```

The `Set` datastructure

- Maintain a collection of distinct elements and support the following operations
  - `insert` : inserts a given value into the set.
  - `delete` : deletes a given value from the set.
  - `search` : checks whether a given value is an element of the set.

```
data Eq a => Set a = Set [a]
search x (Set y)= elem x y
```

The Set datastructure

- Maintain a collection of distinct elements and support the following operations
  - insert : inserts a given value into the set.
  - delete : deletes a given value from the set.
  - search : checks whether a given value is an element of the set.

```
data Eq a => Set a = Set [a]
search x (Set y)= elem x y
insert x (Set s)
  | elem x  (Set s)  = Set s
  | otherwise = Set (x:s)
```

The `Set` datastructure

- Maintain a collection of distinct elements and support the following operations
  - `insert` : inserts a given value into the set.
  - `delete` : deletes a given value from the set.
  - `search` : checks whether a given value is an element of the set.

```
data Eq a => Set a = Set [a]
search x (Set y)= elem x y
insert x (Set s)
  | elem x  (Set s)  = Set s
  | otherwise = Set (x:s)
delete x (Set s)  = Set (filter (/= x) s)
```

Complexity of the operations

Complexity of the operations

- `search` takes linear time.

Complexity of the operations

- `search` takes linear time.
- `insert` takes linear time.

Complexity of the operations

- `search` takes linear time.
- `insert` takes linear time.
- `delete` takes linear time.

Complexity of the operations

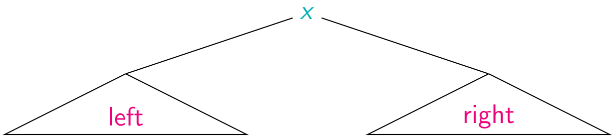
- `search` takes linear time.
- `insert` takes linear time.
- `delete` takes linear time.
- A sequence of N operations can take $O(N^2)$ time.

Complexity of the operations

- `search` takes linear time.
- `insert` takes linear time.
- `delete` takes linear time.
- A sequence of N operations can take $O(N^2)$ time.
- We can do better if the elements of the type `a` can be ordered.

A datatype for binary trees

- Trees are recursive datatypes
- A tree is either
  - Empty
  - Or is a node containing a value and two trees



The binary tree datatype

The binary tree datatype

```
data Btree a = Nil | Node (Btree a) a (Btree a)
```

The binary tree datatype

```
data Btree a = Nil | Node (Btree a) a (Btree a)
```

- `Nil` and `Node` are the constructors.

The binary tree datatype

```
data Btree a = Nil | Node (Btree a) a (Btree a)
```

- `Nil` and `Node` are the constructors.
- `Nil` represents the empty tree.

The binary tree datatype

```
data Btree a = Nil | Node (Btree a) a (Btree a)
```

- `Nil` and `Node` are the constructors.
- `Nil` represents the empty tree.
- A nonempty tree (identified by the constructor `Node`) has three parts
  - A left (sub-)tree
  - A value
  - A right (sub-)tree

Examples of trees

```
Node (Node Nil 2 Nil)  3 (Node Nil 5 Nil)
```

Examples of trees

```
Node (Node Nil 2 Nil) 3 (Node Nil 5 Nil)

        3
       / \
      2   5
```

Examples of trees

```
Node (Node Nil 2 Nil)  3 (Node Nil 5 Nil)

        3
       / \
      2   5

Node (Node Nil 4 Nil) 6
     (Node (Node Nil 2 Nil)  3 (Node Nil 5 Nil))
```

Examples of trees

```
Node (Node Nil 2 Nil)  3 (Node Nil 5 Nil)

        3
       / \
      2   5

Node (Node Nil 4 Nil) 6
     (Node (Node Nil 2 Nil)  3 (Node Nil 5 Nil))

        6
       / \
      4   3
         / \
        2   5
```

Binary Trees ...

- What about

```
      4
     / \
    2   5
   / \
  1   3
```

Binary Trees ...

- What about

```
       4
      / \
     2   5
    / \
   1   3
```

`Node (Node (Node Nil 1 Nil) 2 (Node Nil 3 Nil))`
`4 (Node Nil 5 Nil)`

Functions on Binary trees

- `size` – Number of nodes in the tree

Functions on Binary trees

- `size` – Number of nodes in the tree

```
size :: Btree a -> Int
size Nil = 0
size (Node tl x tr) = 1 + (size tl) + (size tr)
```

Functions on Binary trees

- `size` – Number of nodes in the tree

```
size :: Btree a -> Int
size Nil = 0
size (Node tl x tr) = 1 + (size tl) + (size tr)
```

- `height` – Longest path from the root to a leaf

Functions on Binary trees

- size – Number of nodes in the tree

```
size :: Btree a -> Int
size Nil = 0
size (Node tl x tr) = 1 + (size tl) + (size tr)
```

- height – Longest path from the root to a leaf

```
height :: Btree a -> Int
height Nil = 0
height (Node t1 x tr) =
      1 + (max  (height tl) (height tr))
```

Levels

- List nodes level by level and from left to right within each level.

```
    4
   / \
  2   5
 / \
1   3
```

Levels

- List nodes level by level and from left to right within each level.

```
    4
   / \
  2   5
 / \
1   3
```

```
[4,2,5,1,3]
```