Introduction to Programming: Lecture 09

K Narayan Kumar

Chennai Mathematical Institute

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

05 Sep 2013

User defined polymorphic datatypes

User defined polymorphic datatypes

```
data Box a = MyBox a
```

User defined polymorphic datatypes

```
data Box a = MyBox a
```

- Examples

User defined polymorphic datatypes

```
data Box a = MyBox a
```

- Examples

```
MyBox "Monday"
```

User defined polymorphic datatypes

```
data Box a = MyBox a
```

- Examples

```
MyBox "Monday"
MyBox 2.5
```

User defined polymorphic datatypes

```
data Box a = MyBox a
```

- Examples

```
MyBox "Monday"
MyBox 2.5
MyBox [2,3,4]
```

User defined polymorphic datatypes

```
data Box a = MyBox a
```

- Examples

```
MyBox "Monday"
MyBox 2.5
MyBox [2,3,4]
```

- Oops! No `show` function or equality.

Box: 2nd attempt

```
data Box a = MyBox a
  deriving (Eq, Show)
```

`Box`: 2nd attempt

```
data Box a = MyBox a
  deriving (Eq, Show)
```

- `Mybox 7` – works as expected.

`Box`: 2nd attempt

```
data Box a = MyBox a
  deriving (Eq, Show)
```

- `Mybox 7` – works as expected.
- `Mybox (++)` — Runtime error!!

`Box`: 2nd attempt

```
data Box a = MyBox a
  deriving (Eq, Show)
```

- `Mybox 7` – works as expected.
- `Mybox (++)` — Runtime error!!
- Bad definition. Too generous with type `a`.

Box: 3rd attempt

```
data (Eq a, Show a) => Box a = MyBox a
  deriving (Eq, Show)
```

`Box`: 3rd attempt

```
data (Eq a, Show a) => Box a = MyBox a
  deriving (Eq, Show)
```

- `Mybox (++)` is caught at compile time!

User defined polymorphic datatypes

- A polymorphic version of `Shape`

User defined polymorphic datatypes

- A polymorphic version of `Shape`

```
data Num a => (Shape a) =
    Square a | Circle a | Rectangle a a
  deriving (Eq, Ord, Show)

size :: Num a => Shape a -> a

size (Square x)      = x
size (Circle r)      = r
size (Rectangle l w) = l*w
```

User defined polymorphic datatypes

- A polymorphic version of `Shape`

```
data Num a => (Shape a) =
    Square a | Circle a | Rectangle a a
  deriving (Eq, Ord, Show)

size :: Num a => Shape a -> a

size (Square x)      = x
size (Circle r)      = r
size (Rectangle l w) = l*w
```

- `Square 3.0 ::  (frac t) => Shape t`

User defined polymorphic datatypes

- A polymorphic version of `Shape`

```
data Num a => (Shape a) =
    Square a | Circle a | Rectangle a a
  deriving (Eq, Ord, Show)

size :: Num a => Shape a -> a

size (Square x)      = x
size (Circle r)      = r
size (Rectangle l w) = l*w
```

- `Square 3.0 :: (frac t) => Shape t`
- `Square 3 :: (Num t) => Shape t`

Recursive datatypes

Recursive datatypes

```
data Mylist = Empty | Listof Int Mylist
```

Recursive datatypes

```
data Mylist = Empty | Listof Int Mylist
```

- Representation of `[1,3,2]` is

```
list1 = Listof 1 (Listof 3 (Listof 2 Empty))
```

Recursive datatypes

```
data Mylist = Empty | Listof Int Mylist
```

- Representation of `[1,3,2]` is

  ```
  list1 = Listof 1 (Listof 3 (Listof 2 Empty))
  ```

- `myHead (Listof x rest) =  x`

Recursive datatypes

```
data Mylist = Empty | Listof Int Mylist
```

- Representation of `[1,3,2]` is

  ```
  list1 = Listof 1 (Listof 3 (Listof 2 Empty))
  ```

- `myHead (Listof x rest) =  x`
- `myTail (Listof x rest) = rest`

Recursive datatypes

```
data Mylist = Empty | Listof Int Mylist
```

- Representation of `[1,3,2]` is

```
list1 = Listof 1 (Listof 3 (Listof 2 Empty))
```

- `myHead (Listof x rest) =  x`
- `myTail (Listof x rest) = rest`

```
myHead list1 = 1
```

Recursive datatypes

```
data Mylist = Empty | Listof Int Mylist
```

- Representation of `[1,3,2]` is

```
list1 = Listof 1 (Listof 3 (Listof 2 Empty))
```

- `myHead (Listof x rest) =  x`
- `myTail (Listof x rest) = rest`

```
myHead list1 = 1
myTail list1 = Listof 3 (Listof 2 Empty)
```

Recursive polymorphic types

- Making `Mylist` polymorphic is easy

```
data Mylist a = Empty | Listof a (Mylist a)
```

Recursive polymorphic types

- Making `Mylist` polymorphic is easy

```
data Mylist a = Empty | Listof a (Mylist a)
```

- Builtin lists: `Empty` is `[]`, `Listof` is `:`

Recursive polymorphic types

- Making `Mylist` polymorphic is easy

```
data Mylist a = Empty | Listof a (Mylist a)
```

- Builtin lists: `Empty` is `[]`, `Listof` is `:`
- Notice that the new type is `Mylist a`, not just `Mylist`

Recursive Datatypes: Example

- Drawing Software: xfig, ...

Recursive Datatypes: Example

- Drawing Software: xfig, ...
- A figure is a collection of objects that can be manipulated.

Recursive Datatypes: Example

- Drawing Software: xfig, ...
- A figure is a collection of objects that can be manipulated.
- An object is
  - a shape like a line, a square, a rectangle ...

Recursive Datatypes: Example

- Drawing Software: xfig, ...
- A figure is a collection of objects that can be manipulated.
- An object is
  - a shape like a line, a square, a rectangle ...
  - or a compound object consisting of small objects

Recursive Datatypes: Example

- Drawing Software: xfig, ...
- A figure is a collection of objects that can be manipulated.
- An object is
  - a shape like a line, a square, a rectangle ...
  - or a compound object consisting of small objects
- How to represent a figure using an Haskell type?

Figures ...

- A `point` is a pair of integer coordinates.

Figures ...

- A `point` is a pair of integer coordinates.

```
type Point = (Int,Int)
```

Figures ...

- A `point` is a pair of integer coordinates.

```
type Point = (Int,Int)
```

- A line is given by its endpoints.

Figures ...

- A `point` is a pair of integer coordinates.

```
type Point = (Int,Int)
```

- A line is given by its endpoints.
- A rectangle by its top-left and bottom-right coordinates.

Figures ...

- A `point` is a pair of integer coordinates.

```
type Point = (Int,Int)
```

- A line is given by its endpoints.
- A rectangle by its top-left and bottom-right coordinates.

```
data Figure  = Figure [Object]
data Object = Line Point Point | Rect  Point Point
            | CompObject [Object]
```

Figures ...

- A `point` is a pair of integer coordinates.

```
type Point = (Int,Int)
```

- A line is given by its endpoints.
- A rectangle by its top-left and bottom-right coordinates.

```
data Figure  = Figure [Object]
data Object = Line Point Point | Rect  Point Point
            | CompObject [Object]

f1 = Figure [Line (3,2) (3,6),Rect (0,0) (3,4),
      CompObject [Line (0,0) (2,2),CompObject []]]
```

Manipulating a figure

```
f1 = Figure [Line (3,2) (3,6),Rect (0,0) (3,4),
      CompObject [Line (0,0) (2,2),CompObject []]]
```

Manipulating a figure

```
f1 = Figure [Line (3,2) (3,6),Rect (0,0) (3,4),
       CompObject [Line (0,0) (2,2),CompObject []]]
```

- Count the number of objects in a given figure

Manipulating a figure

```
f1 = Figure [Line (3,2) (3,6),Rect (0,0) (3,4),
      CompObject [Line (0,0) (2,2),CompObject []]]
```

- Count the number of objects in a given figure

```
fcount f1 = 3
```

Manipulating a figure

```
f1 = Figure [Line (3,2) (3,6),Rect (0,0) (3,4),
      CompObject [Line (0,0) (2,2),CompObject []]]
```

- Count the number of objects in a given figure

```
fcount f1 = 3
```

```
fcount (Figure l) = length l
```

Manipulating a figure

```
f1 = Figure [Line (3,2) (3,6),Rect (0,0) (3,4),
      CompObject [Line (0,0) (2,2),CompObject []]]
```

- Count the number of objects in a given figure

```
fcount f1 = 3
```

```
fcount (Figure l) = length l
```

- Count the number of simple objects in a given figure.

```
scount f1 = 3
```

Counting simple objects

```
ocount :: Object -> Int
ocount (Line _ _) = 1
ocount (Rect _ _) = 1
ocount (CompObject l) = sum  (map ocount l)

 scount (Figure l) = ocount (CompObject l)
```

Organizing functions as Modules

- Organize functions into modules.

Organizing functions as Modules

- Organize functions into modules.
- Each module contains functions that are related to each other.

Sorting Module

- The name of the file must match the name of the `module`

Sorting Module

- The name of the file must match the name of the `module`

```
module SortingFns where

insert :: Ord a => a -> [a] -> [a]
insert x [] = ...
          ...

isort :: Ord a => a -> [a] -> [a]
```

Sorting Module

- The name of the file must match the name of the `module`

```
module SortingFns where

insert :: Ord a => a -> [a] -> [a]
insert x [] = ...
          ...

isort :: Ord a => a -> [a] -> [a]
```

- To invoke functions from a module it must be imported

```
import SortingFns

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

Modules: Hiding some functions

- Helper functions such as `insert, merge`

Modules: Hiding some functions

- Helper functions such as `insert, merge`
  ... should not be revealed to the user

Modules: Hiding some functions

- Helper functions such as `insert, merge`
  ... should not be revealed to the user
- Clearly separates functionality from details of implementation.

Modules: Hiding some functions

- Helper functions such as `insert, merge`
  ... should not be revealed to the user
- Clearly separates functionality from details of implementation.
- Allows the modification of the implementation without affecting other code.

Modules: Hiding some functions

- Helper functions such as `insert, merge`
  ... should not be revealed to the user
- Clearly separates functionality from details of implementation.
- Allows the modification of the implementation without affecting other code.

The set of functions revealed constitutes the interface of that module.

Modules: Hiding some functions

```
module SortingFns (isort,mergesort)  where

insert :: Ord a => a -> [a] -> [a]
insert x [] = ...
...

isort :: Ord a => a -> [a] -> [a]
```

Modules: Hiding some functions

```
module SortingFns (isort,mergesort)  where

insert :: Ord a => a -> [a] -> [a]
insert x [] = ...
...

isort :: Ord a => a -> [a] -> [a]
```

- Used as before

Modules: Hiding some functions

```
module SortingFns (isort,mergesort)  where

insert :: Ord a => a -> [a] -> [a]
insert x [] = ...
...

isort :: Ord a => a -> [a] -> [a]
```

- Used as before

```
import SortingFns

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

A Calculator

- The traditional notation for expressions is called infix notation.

```
3+(5*8)
```

```
(2+3)-(7+2)
```

A Calculator

- The traditional notation for expressions is called infix notation.

  ```
  3+(5*8)
  ```

  ```
  (2+3)-(7+2)
  ```

  Operators appear between their arguments.

A Calculator

- The traditional notation for expressions is called infix notation.

```
3+(5*8)
```

```
(2+3)-(7+2)
```

  Operators appear between their arguments.

- Haskell allows `prefix` notation.

```
(+) 3 ((*) 5 8)
```

```
(-) ((+) 2 3) ((+) 7 2)
```

A Calculator

- The traditional notation for expressions is called infix notation.

  ```
  3+(5*8)
  ```

  ```
  (2+3)-(7+2)
  ```

  Operators appear between their arguments.

- Haskell allows `prefix` notation.

  ```
  (+) 3 ((*) 5 8)
  ```

  ```
  (-) ((+) 2 3) ((+) 7 2)
  ```

  Operators appear before the arguments.

A Calculator

- The traditional notation for expressions is called infix notation.

```
3+(5*8)
(2+3)-(7+2)
```

  Operators appear between their arguments.

- Haskell allows `prefix` notation.

```
(+) 3 ((*) 5 8)
(-) ((+) 2 3) ((+) 7 2)
```

  Operators appear before the arguments.

- Postfix notation places the operator after its arguments.

```
3 (5 8 *) +
(2 3 +) (7 2 +) -
```

A Calculator

- The traditional notation for expressions is called infix notation.

```
3+(5*8)
(2+3)-(7+2)
```

  Operators appear between their arguments.

- Haskell allows `prefix` notation.

```
(+) 3 ((*) 5 8)
(-) ((+) 2 3) ((+) 7 2)
```

  Operators appear before the arguments.

- Postfix notation places the operator after its arguments.

```
3 (5 8 *) +
(2 3 +) (7 2 +) -
```

- Do we need brackets in the postfix notation?

Postfix without brackets

```
3 5 8 * +
```

Postfix without brackets

```
3 5 8 * +
```

- Every bracket-free expression can be converted uniquely into a bracketed one.

Postfix without brackets

```
3 5 8 * +
```

- Every bracket-free expression can be converted uniquely into a bracketed one.
- Scan from the left
  - If it is a number it is a standalone expression

Postfix without brackets

```
3 5 8 * +
```

- Every bracket-free expression can be converted uniquely into 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 to get a new expression.

Postfix without brackets

```
3 5 8 * +
```

- Every bracket-free expression can be converted uniquely into 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 to get a new expression.

Postfix without brackets

```
3 5 8 * +
```

- Every bracket-free expression can be converted uniquely into 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 to get a new expression.

```
3
```

Postfix without brackets

```
3 5 8 * +
```

- Every bracket-free expression can be converted uniquely into 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 to get a new expression.

```
3 5
```

Postfix without brackets

```
3 5 8 * +
```

- Every bracket-free expression can be converted uniquely into 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 to get a new expression.

```
3 5 8
```

Postfix without brackets

```
3 5 8 * +
```

- Every bracket-free expression can be converted uniquely into 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 to get a new expression.

```
3 (5 8 *)
```

Postfix without brackets

```
3 5 8 * +
```

- Every bracket-free expression can be converted uniquely into 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 to get a new expression.

```
(3 (5 8 *) +)
```

Postfix without brackets

```
3 5 8 * +
```

- Every bracket-free expression can be converted uniquely into 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 to get a new expression.

```
(3 (5 8 *) +)
```

- Here's another example

```
2 3 + 7 2 + -
```

Postfix without brackets

```
3 5 8 * +
```

- Every bracket-free expression can be converted uniquely into 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 to get a new expression.

```
(3 (5 8 *) +)
```

- Here's another example

```
2 3 + 7 2 + -
```

```
2
```

Postfix without brackets

```
3 5 8 * +
```

- Every bracket-free expression can be converted uniquely into 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 to get a new expression.

```
(3 (5 8 *) +)
```

- Here's another example

```
2 3 + 7 2 + -
```

```
2  3
```

Postfix without brackets

```
3 5 8 * +
```

- Every bracket-free expression can be converted uniquely into 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 to get a new expression.

```
(3 (5 8 *) +)
```

- Here's another example

```
2 3 + 7 2 + -
```

```
(2 3 +)
```

Postfix without brackets

```
3 5 8 * +
```

- Every bracket-free expression can be converted uniquely into 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 to get a new expression.

```
(3 (5 8 *) +)
```

- Here's another example

```
2 3 + 7 2 + -
```

```
(2 3 +) 7
```

Postfix without brackets

```
3 5 8 * +
```

- Every bracket-free expression can be converted uniquely into 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 to get a new expression.

```
(3 (5 8 *) +)
```

- Here's another example

```
2 3 + 7 2 + -
```

```
(2 3 +) 7 2
```

Postfix without brackets

```
3 5 8 * +
```

- Every bracket-free expression can be converted uniquely into 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 to get a new expression.

```
(3 (5 8 *) +)
```

- Here's another example

```
2 3 + 7 2 + -
```

```
(2 3 +) (7 2 +)
```

Postfix without brackets

```
3 5 8 * +
```

- Every bracket-free expression can be converted uniquely into 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 to get a new expression.

```
(3 (5 8 *) +)
```

- Here's another example

```
2 3 + 7 2 + -
```

```
((2 3 +) (7 2 +) -)
```

From infix to postfix

- Easy translation by induction on the structure of the expression.

From infix to postfix

- Easy translation by induction on the structure of the expression.
- A value is translated as it is.

From infix to postfix

- Easy translation by induction on the structure of the expression.
- A value is translated as it is.
- Any other expression is of the form `(E1 op E2)`

From infix to postfix

- Easy translation by induction on the structure of the expression.
- A value is translated as it is.
- Any other expression is of the form `(E1 op E2)`

  Translate `E1` Translate `E2` `op`

From infix to postfix

- Easy translation by induction on the structure of the expression.
- A value is translated as it is.
- Any other expression is of the form `(E1 op E2)`

  Translate `E1` Translate `E2` `op`
- Consider `(((3+4)-2)*(5+6))`

From infix to postfix

- Easy translation by induction on the structure of the expression.
- A value is translated as it is.
- Any other expression is of the form `(E1 op E2)`

  Translate `E1` Translate `E2` `op`
- Consider `(((3+4)-2)*(5+6))`

`*`

From infix to postfix

- Easy translation by induction on the structure of the expression.
- A value is translated as it is.
- Any other expression is of the form `(E1 op E2)`

  Translate `E1` Translate `E2` `op`
- Consider `(((3+4)-2)*(5+6))`

  `-` `*`

From infix to postfix

- Easy translation by induction on the structure of the expression.
- A value is translated as it is.
- Any other expression is of the form `(E1 op E2)`

  Translate `E1` Translate `E2` `op`
- Consider `(((3+4)-2)*(5+6))`

  `+` `-` `*`

From infix to postfix

- Easy translation by induction on the structure of the expression.
- A value is translated as it is.
- Any other expression is of the form `(E1 op E2)`

  Translate `E1` Translate `E2` `op`
- Consider `(((3+4)-2)*(5+6))`

  `3 + - *`

From infix to postfix

- Easy translation by induction on the structure of the expression.
- A value is translated as it is.
- Any other expression is of the form `(E1 op E2)`

  Translate `E1` Translate `E2` `op`
- Consider `(((3+4)-2)*(5+6))`

  `3 4 + - * `

From infix to postfix

- Easy translation by induction on the structure of the expression.
- A value is translated as it is.
- Any other expression is of the form `(E1 op E2)`

  Translate `E1` Translate `E2` `op`
- Consider `(((3+4)-2)*(5+6))`

  `3 4 + 2 -  *`

From infix to postfix

- Easy translation by induction on the structure of the expression.
- A value is translated as it is.
- Any other expression is of the form `(E1 op E2)`

  Translate `E1` Translate `E2` `op`
- Consider `(((3+4)-2)*(5+6))`

  `3 4 + 2 -  + *`

From infix to postfix

- Easy translation by induction on the structure of the expression.
- A value is translated as it is.
- Any other expression is of the form `(E1 op E2)`

  Translate `E1` Translate `E2` `op`
- Consider `(((3+4)-2)*(5+6))`

  `3 4 + 2 - 5 + *`

From infix to postfix

- Easy translation by induction on the structure of the expression.
- A value is translated as it is.
- Any other expression is of the form `(E1 op E2)`

  Translate `E1` Translate `E2` `op`
- Consider `(((3+4)-2)*(5+6))`

  `3 4 + 2 - 5 6 + *`

Evaluating postfix expressions

- The structure of postfix expressions allows it to be evaluated easily.

Evaluating postfix expressions

- The structure of postfix expressions allows it to be evaluated easily.
- Follow the bracketing algorithm, and evaluate each expression as it is created.

Evaluating postfix expressions

- The structure of postfix expressions allows it to be evaluated easily.
- Follow the bracketing algorithm, and evaluate each expression as it is created.

```
3 5 8 * +
```

Evaluating postfix expressions

- The structure of postfix expressions allows it to be evaluated easily.
- Follow the bracketing algorithm, and evaluate each expression as it is created.

```
3 5 8 * +
```

```
3
```

Evaluating postfix expressions

- The structure of postfix expressions allows it to be evaluated easily.
- Follow the bracketing algorithm, and evaluate each expression as it is created.

```
3 5 8 * +
```

```
3 5
```

Evaluating postfix expressions

- The structure of postfix expressions allows it to be evaluated easily.
- Follow the bracketing algorithm, and evaluate each expression as it is created.

```
3 5 8 * +
```

```
3 (5 8 *)
```

Evaluating postfix expressions

- The structure of postfix expressions allows it to be evaluated easily.
- Follow the bracketing algorithm, and evaluate each expression as it is created.

```
3 5 8 * +
```

```
3 40
```

Evaluating postfix expressions

- The structure of postfix expressions allows it to be evaluated easily.
- Follow the bracketing algorithm, and evaluate each expression as it is created.

```
3 5 8 * +
```

```
(3 40 +)
```

Evaluating postfix expressions

- The structure of postfix expressions allows it to be evaluated easily.
- Follow the bracketing algorithm, and evaluate each expression as it is created.

```
3 5 8 * +
```

```
43
```

Evaluating postfix expressions

- The structure of postfix expressions allows it to be evaluated easily.
- Follow the bracketing algorithm, and evaluate each expression as it is created.

```
3 5 8 * +
```

```
43
```

- Another example:

Evaluating postfix expressions

- The structure of postfix expressions allows it to be evaluated easily.
- 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

- The structure of postfix expressions allows it to be evaluated easily.
- 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

- The structure of postfix expressions allows it to be evaluated easily.
- 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

- The structure of postfix expressions allows it to be evaluated easily.
- 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

- The structure of postfix expressions allows it to be evaluated easily.
- 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

- The structure of postfix expressions allows it to be evaluated easily.
- 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

- The structure of postfix expressions allows it to be evaluated easily.
- 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

- The structure of postfix expressions allows it to be evaluated easily.
- 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

- The structure of postfix expressions allows it to be evaluated easily.
- 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

- The structure of postfix expressions allows it to be evaluated easily.
- 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

- The structure of postfix expressions allows it to be evaluated easily.
- 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

- The structure of postfix expressions allows it to be evaluated easily.
- 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

- The structure of postfix expressions allows it to be evaluated easily.
- 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)

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)

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

```
data Stack a = Stack [a]

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 via lists

```
data Stack a = Stack [a]

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
```

- Must add function `empty :: Stack a`.

The `Stack` module via lists

```
data Stack a = Stack [a]

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
```

- Must add function `empty :: Stack a`.

```
empty = Stack []
```

The `Stack` module via lists

```
data Stack a = Stack [a]

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
```

- Must add function `empty :: Stack a`.

  ```
  empty = Stack []
  ```

- Must add this function to the earlier definition to hide the implementation details.