Introduction to Programming: Lecture 8

K Narayan Kumar

Chennai Mathematical Institute

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

03 September 2013

Testing if a number is prime

- Check if a given number n is a prime.

Testing if a number is prime

- Check if a given number n is a prime.
- Naive algorithm:

```
prime n = and (map f [1..(div n 2)])
  where
    f i = ((n mod i) /= 0)
```

Testing if a number is prime

- Check if a given number n is a prime.
- Naive algorithm:

```
prime n = and (map f [1..(div n 2)])
  where
    f i = ((n mod i) /= 0)
```

- This function takes $O(n)$ to decide if n is a prime.

  Is this an efficient algorithm?

Testing if a number is prime

- Check if a given number n is a prime.
- Naive algorithm:

```
prime n = and (map f [1..(div n 2)])
  where
    f i = ((n mod i) /= 0)
```

- This function takes $O(n)$ to decide if n is a prime.

  Is this an efficient algorithm?
- Is it efficient to take 2^{64} steps to decide if a 64 bit number is prime?

Testing if a number is prime

- Check if a given number n is a prime.
- Naive algorithm:

```
prime n = and (map f [1..(div n 2)])
  where
    f i = ((n mod i) /= 0)
```

- This function takes $O(n)$ to decide if n is a prime.

  Is this an efficient algorithm?
- Is it efficient to take 2^{64} steps to decide if a 64 bit number is prime?
- The size of the input is the number of bits required to write it down.

  The above algorithm is takes 2^n steps to decide if a number of size n is prime.

Lazy Evaluation

- Any Haskell expression is of the form `f e` where
  - `f` is the outermost function
  - `e` is the expression to which it is applied.

Lazy Evaluation

- Any Haskell expression is of the form `f e` where
  - `f` is the outermost function
  - `e` is the expression to which it is applied.
- Consider `sum (2:map (+1) [1,2,3])`

Lazy Evaluation

- Any Haskell expression is of the form `f e` where
  - `f` is the outermost function
  - `e` is the expression to which it is applied.
- Consider `sum (2:map (+1) [1,2,3])`

```
f = sum
e = 2:map (+1) [1,2,3]
```

Lazy Evaluation

- Any Haskell expression is of the form `f e` where
  - `f` is the outermost function
  - `e` is the expression to which it is applied.
- Consider `sum (2:map (+1) [1,2,3])`

  ```
  f = sum
  ```

  ```
  e = 2:map (+1) [1,2,3]
  ```
- When `f` is a simple function name and not an expression, Haskell reduces `f e` using the definition of `f`

Lazy Evaluation

- Any Haskell expression is of the form `f e` where
  - `f` is the outermost function
  - `e` is the expression to which it is applied.
- Consider `sum (2:map (+1) [1,2,3])`

  `f = sum`

  `e = 2:map (+1) [1,2,3]`
- When `f` is a simple function name and not an expression, Haskell reduces `f e` using the definition of `f`

  `sum (2:map (+1) [1,2,3])`

  $\leadsto$ `2 + sum (map (+1) [1,2,3])`

Lazy Evaluation

- Any Haskell expression is of the form `f e` where
  - `f` is the outermost function
  - `e` is the expression to which it is applied.
- Consider `sum (2:map (+1) [1,2,3])`

  `f = sum`

  `e = 2:map (+1) [1,2,3]`
- When `f` is a simple function name and not an expression, Haskell reduces `f e` using the definition of `f`

  `sum (2:map (+1) [1,2,3])`

  $\leadsto$ `2 + sum (map (+1) [1,2,3])`
- The argument is not evaluated if the function definition does not force it to be evaluted.

Lazy Evaluation

- Any Haskell expression is of the form `f e` where
  - `f` is the outermost function
  - `e` is the expression to which it is applied.
- Consider `sum (2:map (+1) [1,2,3])`

  ```
  f = sum
  ```

  ```
  e = 2:map (+1) [1,2,3]
  ```
- When `f` is a simple function name and not an expression, Haskell reduces `f e` using the definition of `f`

  ```
  sum (2:map (+1) [1,2,3])
  ```

  $\leadsto$ `2 + sum (map (+1) [1,2,3])`
- The argument is not evaluated if the function definition does not force it to be evaluted.

  ```
  sum (2:  map (+1) [1,2,3])
  ```

  $\leadsto$ `2 + sum (map (+1) [1,2,3])`

Lazy Evaluation

- Any Haskell expression is of the form `f e` where
  - `f` is the outermost function
  - `e` is the expression to which it is applied.
- Consider `sum (2:map (+1) [1,2,3])`

  `f = sum`

  `e = 2:map (+1) [1,2,3]`
- When `f` is a simple function name and not an expression, Haskell reduces `f e` using the definition of `f`

  `sum (2:map (+1) [1,2,3])`

  $\leadsto$ `2 + sum (map (+1) [1,2,3])`
- The argument is not evaluated if the function definition does not force it to be evaluted.

  `sum (2: map (+1) [1,2,3])`

  $\leadsto$ `2 + sum (map (+1) [1,2,3])`
- Reduce the argument only if necessary.

Lazy Evaluation ...

- Sometimes the argument has to be evaluated to evaluate the function definition

Lazy Evaluation ...

- Sometimes the argument has to be evaluated to evaluate the function definition

```
sum (map (+1) [1,2,3])
```

$\leadsto$ `sum (2:  map (+1) [2,3])`

$\leadsto$ `2 + sum (map (+1) [2,3])`

Lazy Evaluation ...

- Sometimes the argument has to be evaluated to evaluate the function definition

```
sum (map (+1) [1,2,3])
```

$\leadsto$ `sum (2:  map (+1) [2,3])`

$\leadsto$ `2 + sum (map (+1) [2,3])`

Evaluate only as much of the argument as is necessary.

Lazy Evaluation ...

- Sometimes the argument has to be evaluated to evaluate the function definition

  ```
  sum (map (+1) [1,2,3])
  ```

  $\leadsto$ `sum (2:  map (+1) [2,3])`

  $\leadsto$ `2 + sum (map (+1) [2,3])`

  Evaluate only as much of the argument as is necessary.

- What if `f` is an expression?

  For eg. `elem (3+7) (map (+1) [8,9])`

Lazy Evaluation ...

- Sometimes the argument has to be evaluated to evaluate the function definition

  ```
  sum (map (+1) [1,2,3])
  ```

  $\leadsto$ `sum (2:  map (+1) [2,3])`

  $\leadsto$ `2 + sum (map (+1) [2,3])`

  Evaluate only as much of the argument as is necessary.

- What if `f` is an expression?

  For eg. `elem (3+7) (map (+1) [8,9])`

  ```
  f = elem (3+7)
  e = map (+1) [8,9]
  ```

Lazy Evaluation ...

- Sometimes the argument has to be evaluated to evaluate the function definition

  ```
  sum (map (+1) [1,2,3])
  ```

  $\leadsto$ `sum (2:  map (+1) [2,3])`

  $\leadsto$ `2 + sum (map (+1) [2,3])`

  Evaluate only as much of the argument as is necessary.

- What if `f` is an expression?

  For eg. `elem (3+7) (map (+1) [8,9])`

  ```
  f = elem (3+7)
  e = map (+1) [8,9]
  ```

- In what order are they reduced?

Lazy Evaluation ...

- Sometimes the argument has to be evaluated to evaluate the function definition

  ```
  sum (map (+1) [1,2,3])
  ```

  $\leadsto$ `sum (2:  map (+1) [2,3])`

  $\leadsto$ `2 + sum (map (+1) [2,3])`

  Evaluate only as much of the argument as is necessary.

- What if `f` is an expression?

  For eg. `elem (3+7) (map (+1) [8,9])`

  ```
  f = elem (3+7)
  e = map (+1) [8,9]
  ```

- In what order are they reduced?
- Haskell reduces the function first.

Lazy Evaluation ...

- Sometimes the argument has to be evaluated to evaluate the function definition

  ```
  sum (map (+1) [1,2,3])
  ```

  $\leadsto$ `sum (2:  map (+1) [2,3])`

  $\leadsto$ `2 + sum (map (+1) [2,3])`

  Evaluate only as much of the argument as is necessary.

- What if `f` is an expression?

  For eg. `elem (3+7) (map (+1) [8,9])`

  ```
  f = elem (3+7)
  e = map (+1) [8,9]
  ```

- In what order are they reduced?
- Haskell reduces the function first.

  The argument `e` is reduced only if nothing else is possible

  ... and the same rule is applied recursively in reducing subexpressions.

Lazy Evaluation: examples

- The following program does not generate an error

```
tail (3/0):[3,5,9]
```

Lazy Evaluation: examples

- The following program does not generate an error

```
tail (3/0):[3,5,9]
⇝ [3,5,9]
```

- Allows us to deal with infinite objects:

```
take 3 [1,2..]  = [1,2,3]
```

Lazy Evaluation: examples

- The following program does not generate an error

```
tail (3/0):[3,5,9]
⇝ [3,5,9]
```

- Allows us to deal with infinite objects:

```
take 3 [1,2..]  = [1,2,3]

take 3 [1,2..]
```

Lazy Evaluation: examples

- The following program does not generate an error

```
tail (3/0):[3,5,9]
⇝ [3,5,9]
```

- Allows us to deal with infinite objects:

```
take 3 [1,2..]  = [1,2,3]
```

```
take 3 [1,2..]
⇝ 1:(take 2 [2,3..])
```

Lazy Evaluation: examples

- The following program does not generate an error

```
tail (3/0):[3,5,9]
⇝ [3,5,9]
```

- Allows us to deal with infinite objects:

```
take 3 [1,2..]  = [1,2,3]
```

```
take 3 [1,2..]
⇝ 1:(take 2 [2,3..])
⇝ 1:(2:take 1 [3,4..])
```

Lazy Evaluation: examples

- The following program does not generate an error

```
tail (3/0):[3,5,9]
⇝ [3,5,9]
```

- Allows us to deal with infinite objects:

```
take 3 [1,2..]  = [1,2,3]
```

```
take 3 [1,2..]
⇝ 1:(take 2 [2,3..])
⇝ 1:(2:take 1 [3,4..])
⇝ 1:(2:(3:take 0 [4..]))
```

Lazy Evaluation: examples

- The following program does not generate an error

```
tail (3/0):[3,5,9]
⇝ [3,5,9]
```

- Allows us to deal with infinite objects:

```
take 3 [1,2..]  = [1,2,3]
```

```
take 3 [1,2..]
⇝ 1:(take 2 [2,3..])
⇝ 1:(2:take 1 [3,4..])
⇝ 1:(2:(3:take 0 [4..]))
⇝ 1:(2:(3:[]))
```

Type Classes: A brief introduction

- Consider a simple sorting algorithm such as `isort`

```
insert x [] = [x]
insert x (y:ys)
  | (x < y) = x:(y:ys)
  | otherwise = y : insert x ys

isort  = foldr insert []
```

- These functions are not polymorphic, in the sense used so far, as they use the function `<` which is not necessarily defined on all types.

Type Classes: A brief introduction

- Consider a simple sorting algorithm such as `isort`

```
insert x [] = [x]
insert x (y:ys)
  | (x < y) = x:(y:ys)
  | otherwise = y : insert x ys

isort  = foldr insert []
```

- These functions are not polymorphic, in the sense used so far, as they use the function `<` which is not necessarily defined on all types.
- Do we have to define a sorting function for each type?
  - `Int`
  - `Char`
  - `(Int,Int)`
  - ...

Type Classes: A brief introduction

- Consider a simple sorting algorithm such as `isort`

```
insert x [] = [x]
insert x (y:ys)
  | (x < y) = x:(y:ys)
  | otherwise = y : insert x ys

isort  = foldr insert []
```

- These functions are not polymorphic, in the sense used so far, as they use the function `<` which is not necessarily defined on all types.
- Do we have to define a sorting function for each type?
  - `Int`
  - `Char`
  - `(Int,Int)`
  - ...
- ... particularly when the same definition works for all these types!

Type Classes ...

- Ideally, the type of `isort` should be
  "`[a] -> [a]` provided the type `a` has `<` defined on it"

Type Classes ...

- Ideally, the type of `isort` should be
  "`[a] -> [a]` provided the type `a` has `<` defined on it"
- Haskell's Type Classes permit us to do precisely this.

  A type class is a collection of types.

Type Classes ...

- Ideally, the type of `isort` should be
  "`[a] -> [a]` provided the type `a` has `<` defined on it"
- Haskell's Type Classes permit us to do precisely this.

  A type class is a collection of types.
- We may define the type of `isort` to be

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

  "`Ord a =>`" should be read as "If the type `a` belongs to the type-class `Ord` then "

Type Classes ...

- Ideally, the type of `isort` should be
  "`[a] -> [a]` provided the type `a` has `<` defined on it"
- Haskell's Type Classes permit us to do precisely this.

  A type class is a collection of types.
- We may define the type of `isort` to be

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

  "`Ord a =>`" should be read as "If the type `a` belongs to the type-class `Ord` then "
- A type `a` belongs to the type-class `Ord` if it has functions `<,>,<=,>=,==,/=` (all of type `a -> a -> Bool`) defined on it.

A type for sort functions

- Thus we may write

```
insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys)
  | (x < y) = x:y:ys
  | otherwise = y : insert x ys

isort :: Ord a => [a] -> [a]
isort l = foldr insert [] l
```

A type for sort functions

- Thus we may write

```
insert :: Ord a => a -> [a] -> [a]
insert x [] = [x]
insert x (y:ys)
  | (x < y) = x:y:ys
  | otherwise = y : insert x ys

isort :: Ord a => [a] -> [a]
isort l = foldr insert [] l
```

- If you omit the type definition, ghc will infer the type correctly.

Type Classes ...

- In Haskell type-classes are identified by a signature: a bunch of functions that must be defined for membership in the class.

Type Classes ...

- In Haskell type-classes are identified by a signature: a bunch of functions that must be defined for membership in the class.
- `Ord` is defined by the signature `<,>,<=,>=,==,/=`

Type Classes ...

- In Haskell type-classes are identified by a signature: a bunch of functions that must be defined for membership in the class.
- `Ord` is defined by the signature `<,>,<=,>=,==,/=`
- `Eq` is defined by the signature `=,/=`

Type Classes ...

- In Haskell type-classes are identified by a signature: a bunch of functions that must be defined for membership in the class.
- `Ord` is defined by the signature `<,>,<=,>=,==,/=`
- `Eq` is defined by the signature `=,/=`
- `Show` is defined by the signature `show`.

Type Classes ...

- In Haskell type-classes are identified by a signature: a bunch of functions that must be defined for membership in the class.
- `Ord` is defined by the signature `<,>,<=,>=,==,/=`
- `Eq` is defined by the signature `=,/=`
- `Show` is defined by the signature `show`.
- All basic types (`Int, Float, Char, Bool ...`) are members of all these type-classes.

Type Classes ...

- In Haskell type-classes are identified by a signature: a bunch of functions that must be defined for membership in the class.
- `Ord` is defined by the signature `<,>,<=,>=,==,/=`
- `Eq` is defined by the signature `=,/=`
- `Show` is defined by the signature `show`.
- All basic types (`Int, Float, Char, Bool ...`) are members of all these type-classes.
- All lists and tuples of types that belong to any of these type classes are also members of these classes.

Type Classes ...

- In Haskell type-classes are identified by a signature: a bunch of functions that must be defined for membership in the class.
- `Ord` is defined by the signature `<,>,<=,>=,==,/=`
- `Eq` is defined by the signature `=,/=`
- `Show` is defined by the signature `show`.
- All basic types (`Int, Float, Char, Bool ...`) are members of all these type-classes.
- All lists and tuples of types that belong to any of these type classes are also members of these classes.
- Higher-order types do NOT belong to `Ord, Eq, Show`.

Type Classes ...

- In Haskell type-classes are identified by a signature: a bunch of functions that must be defined for membership in the class.
- `Ord` is defined by the signature `<,>,<=,>=,==,/=`
- `Eq` is defined by the signature `=,/=`
- `Show` is defined by the signature `show`.
- All basic types (`Int, Float, Char, Bool ...`) are members of all these type-classes.
- All lists and tuples of types that belong to any of these type classes are also members of these classes.
- Higher-order types do NOT belong to `Ord, Eq, Show`.
- Other type-classes we have encountered include `Num, Frac, Integral, ...`

Type Classes ...

- In Haskell type-classes are identified by a signature: a bunch of functions that must be defined for membership in the class.
- `Ord` is defined by the signature `<,>,<=,>=,==,/=`
- `Eq` is defined by the signature `=,/=`
- `Show` is defined by the signature `show`.
- All basic types (`Int, Float, Char, Bool ...`) are members of all these type-classes.
- All lists and tuples of types that belong to any of these type classes are also members of these classes.
- Higher-order types do NOT belong to `Ord, Eq, Show`.
- Other type-classes we have encountered include `Num, Frac, Integral, ...`
- We can create our own type-classes and also add types to type-classes. (As we shall see later.)

User defined datatypes

- The `data` keyword is used to define new types.

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

User defined datatypes

- The `data` keyword is used to define new types.

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

- Can directly use new type in functions

```
weekend :: Day -> Bool
weekend Sun = True
weekend Sat = True
weekend _   = False
```

User defined datatypes

- The `data` keyword is used to define new types.

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

- Can directly use new type in functions

```
weekend :: Day -> Bool
weekend Sun = True
weekend Sat = True
weekend _   = False
```

- What about

```
weekend2 :: Day -> Bool
weekend2 d
  | (d == Sat || d == Sun) = True
  | otherwise              = False
```

User defined datatypes

- The `data` keyword is used to define new types.

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
```

- Can directly use new type in functions

```
weekend :: Day -> Bool
weekend Sun = True
weekend Sat = True
weekend _   = False
```

- What about

```
weekend2 :: Day -> Bool
weekend2 d
  | (d == Sat || d == Sun) = True
  | otherwise              = False
```

- `ERROR - Instance of Eq Day required for definition of weekend2`

User defined datatypes . . .

- How about

```
nextday :: Day -> Day
nextday Sun = Mon
nextday Mon = Tue
...
nextday Fri = Sat
nextday Sat = Sun
```

User defined datatypes . . .

- How about

```
nextday :: Day -> Day
nextday Sun = Mon
nextday Mon = Tue
...
nextday Fri = Sat
nextday Sat = Sun
```

- What happens if we invoke `nextday Fri` in ghci?

User defined datatypes . . .

- How about

```
nextday :: Day -> Day
nextday Sun = Mon
nextday Mon = Tue
...
nextday Fri = Sat
nextday Sat = Sun
```

- What happens if we invoke `nextday Fri` in ghci?
- To display a value, its type should be in the class `Show` with

Adding user-defined classes to type-classes

- "deriving" the appropriate type-classes.

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
  deriving (Eq,Show)
```

Adding user-defined classes to type-classes

- "deriving" the appropriate type-classes.

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
  deriving (Eq,Show)
```

- Default behaviour is that
  - No two values are equal to each other
    `Mon == Mon`, `Tue /= Fri`

Adding user-defined classes to type-classes

- "deriving" the appropriate type-classes.

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
  deriving (Eq,Show)
```

- Default behaviour is that
  - No two values are equal to each other
    `Mon == Mon`, `Tue /= Fri`
  - Each value is displayed as defined
    `show Wed == "Wed"`

Adding user-defined classes to type-classes

- "deriving" the appropriate type-classes.

```
data Day = Sun | Mon | Tue | Wed | Thu | Fri | Sat
  deriving (Eq,Show)
```

- Default behaviour is that
  - No two values are equal to each other
    `Mon == Mon`, `Tue /= Fri`
  - Each value is displayed as defined
    `show Wed == "Wed"`
- Can also derive `Ord`
  - `Sun` $<$ `Mon` $<$ $\cdots$ $<$ `Sat`

Datatypes with parameters

- Attaching parameters to new types.

Datatypes with parameters

- Attaching parameters to new types.

```
data Shape =
  Square Float | Circle Float | Rectangle Float Float
  deriving (Eq, Ord, Show)
```

Datatypes with parameters

- Attaching parameters to new types.

```
data Shape =
  Square Float | Circle Float | Rectangle Float Float
  deriving (Eq, Ord, Show)
Square 3.0, Circle 2.1, Rectangle 2.5 3.1
```

Datatypes with parameters

- Attaching parameters to new types.

```
data Shape =
  Square Float | Circle Float | Rectangle Float Float
  deriving (Eq, Ord, Show)
```

```
Square 3.0, Circle 2.1, Rectangle 2.5 3.1
```

- ```
  area :: Shape -> Float
  area (Square x)      = x*x
  area (Circle r)      = pi*r*r
  area (Rectangle l w) = l*w
  where
      pi = 3.1415927
  ```
  ```

Datatypes with parameters

- Attaching parameters to new types.

```
data Shape =
  Square Float | Circle Float | Rectangle Float Float
  deriving (Eq, Ord, Show)
Square 3.0, Circle 2.1, Rectangle 2.5 3.1
```

- ```
  area :: Shape -> Float
  area (Square x)      = x*x
  area (Circle r)      = pi*r*r
  area (Rectangle l w) = l*w
  where
      pi = 3.1415927
  ```

- `Square`, `Circle`, ... are called constructors, as are `Sun`, `Mon`, ...
  ```

Datatypes with parameters

- Attaching parameters to new types.

```
data Shape =
  Square Float | Circle Float | Rectangle Float Float
  deriving (Eq, Ord, Show)
Square 3.0, Circle 2.1, Rectangle 2.5 3.1
```

- ```
  area :: Shape -> Float
  area (Square x)      = x*x
  area (Circle r)      = pi*r*r
  area (Rectangle l w) = l*w
  where
      pi = 3.1415927
  ```

- `Square`, `Circle`, ... are called constructors, as are `Sun`, `Mon`, ...
- Here `deriving Eq` literally derives `Eq` from underlying `==` for `Float`
  ```

Constructors ...

- Constructors such as `Sun` or `Circle` are functions.

Constructors ...

- Constructors such as `Sun` or `Circle` are functions.

```
Sun :: Day
Circle :: Float -> Shape
```

Constructors ...

- Constructors such as `Sun` or `Circle` are functions.

```
Sun :: Day
Circle :: Float -> Shape
```

- They can be used exactly as other functions are used.

Constructors ...

- Constructors such as `Sun` or `Circle` are functions.

```
Sun :: Day
Circle :: Float -> Shape
```

- They can be used exactly as other functions are used.

```
map Circle :: [Float] -> [Shape]
map Circle [3.0,2.8] = [Circle 3.0, Circle 2.8]
```

Example: Films

- Store information about films

Example: Films

- Store information about films
  - the name of the film
  - the director
  - the cast

Example: Films

- Store information about films
  - the name of the film
  - the director
  - the cast

```
data Film = FilmC String String [String]
```

Example: Films

- Store information about films
  - the name of the film
  - the director
  - the cast

```
data Film = FilmC String String [String]

film1 = FilmC "Aakrosh" "Govind Nihalni"
  ["Naseeruddin","Om Puri", "Smita"]
film2 = FilmC "Ishqiya" " Abhishek Chaubey"
  ["Naseeruddin", "Vidya Balan", "Arshad Warsi"]
film3 = FilmC "Omkara" "V Bharadwaj"
  ["A Devgan","Saif","Kareena"]
```

Example: Films

- Store information about films
  - the name of the film
  - the director
  - the cast

```
data Film = FilmC String String [String]

film1 = FilmC "Aakrosh" "Govind Nihalni"
  ["Naseeruddin","Om Puri", "Smita"]
film2 = FilmC "Ishqiya" " Abhishek Chaubey"
  ["Naseeruddin", "Vidya Balan", "Arshad Warsi"]
film3 = FilmC "Omkara" "V Bharadwaj"
  ["A Devgan","Saif","Kareena"]
```

- Extract the name of director.

Example: Films

- Store information about films
  - the name of the film
  - the director
  - the cast

```
data Film = FilmC String String [String]

film1 = FilmC "Aakrosh" "Govind Nihalni"
  ["Naseeruddin","Om Puri", "Smita"]
film2 = FilmC "Ishqiya" " Abhishek Chaubey"
  ["Naseeruddin", "Vidya Balan", "Arshad Warsi"]
film3 = FilmC "Omkara" "V Bharadwaj"
  ["A Devgan","Saif","Kareena"]
```

- Extract the name of director.

```
director :: Film -> String
director (FilmC x y z) = y
```

Example: Films

It is customary to use the name of the type as the constructor if there only one constructor.

```
data Film = Film String String [String]
```

Example: Films

It is customary to use the name of the type as the constructor if there only one constructor.

```
data Film = Film String String [String]

film1 = Film "Aakrosh" "Govind Nihalni"
  ["Naseeruddin","Om Puri", "Smita"]
film2 = Film "Ishqiya" " Abhishek Chaubey"
  ["Naseeruddin", "Vidya Balan", "Arshad Warsi"]
film3 = Film "Omkara" "V Bharadwaj"
  ["A Devgan","Saif","Kareena"]
```

Example: Films

It is customary to use the name of the type as the constructor if there only one constructor.

```
data Film = Film String String [String]
film1 = Film "Aakrosh" "Govind Nihalni"
  ["Naseeruddin","Om Puri", "Smita"]
film2 = Film "Ishqiya" " Abhishek Chaubey"
  ["Naseeruddin", "Vidya Balan", "Arshad Warsi"]
film3 = Film "Omkara" "V Bharadwaj"
  ["A Devgan","Saif","Kareena"]
```

- Extract the name of director.

Example: Films

It is customary to use the name of the type as the constructor if there only one constructor.

```
data Film = Film String String [String]

film1 = Film "Aakrosh" "Govind Nihalni"
  ["Naseeruddin","Om Puri", "Smita"]
film2 = Film "Ishqiya" " Abhishek Chaubey"
  ["Naseeruddin", "Vidya Balan", "Arshad Warsi"]
film3 = Film "Omkara" "V Bharadwaj"
  ["A Devgan","Saif","Kareena"]
```

- Extract the name of director.

```
director :: Film -> String
director (Film x y z) = y
```

More on Films

- We might want to list not just the cast, but the music directors, editors, light boys,...

More on Films

- We might want to list not just the cast, but the music directors, editors, light boys,...

```
type Name = String
type Task = String
data Film = Film Name [Credits]
data Credits = Credits Task [Name]
```

More on Films

- We might want to list not just the cast, but the music directors, editors, light boys,...

```
type Name = String
type Task = String
data Film = Film Name [Credits]
data Credits = Credits Task [Name]

film1 = Film "Pulp Fiction"
  [ Credits "Direction" ["Q Tarantino"],
    Credits "Cast"
      ["Uma Thurman","J Travolta","S Jackson"],
    Credits "Stunts" ["Cameron","Jackson"] ]
```

More on films ...

```
type Name = String
type Task = String
data Film = Film Name [Credits]
data Credits = Credits Task [Name]
```

More on films ...

```
type Name = String
type Task = String
data Film = Film Name [Credits]
data Credits = Credits Task [Name]
```

- Extract all the tasks for which credits are available?

More on films ...

```
type Name = String
type Task = String
data Film = Film Name [Credits]
data Credits = Credits Task [Name]
```

- Extract all the tasks for which credits are available?

```
listTasks :: Film -> [Task]
listTasks  = (map nameTask) . extractCredits
  where
  extractCredits :: Film -> [Credits]
  extractCredits (Film n l) = l
  nameTask :: Credits -> Task
  nameTask (Credits x _) = x
```

More on films ...

```
type Name = String
type Task = String
data Film = Film Name [Credits]
data Credits = Credits Task [Name]
```

- Extract all the tasks for which credits are available?

```
listTasks :: Film -> [Task]
listTasks  = (map nameTask) . extractCredits
  where
  extractCredits :: Film -> [Credits]
  extractCredits (Film n l) = l
  nameTask :: Credits -> Task
  nameTask (Credits x _) = x
```

- `listTasks film1 = ["Direction","Cast","Stunts"]`