# Introduction to Programming: Lecture 19

K Narayan Kumar

Chennai Mathematical Institute

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

22 October 2013

# Actions: More details

- Actions are terms of type `IO a` for some `a`.

# Actions: More details

- Actions are terms of type `IO a` for some `a`.
- Two basic functions used to construct and combine actions are:

```
return ::  a -> IO a
>>= ::  IO a -> (a -> IO b) -> IO b
```

# Actions: More details

- Actions are terms of type `IO a` for some `a`.
- Two basic functions used to construct and combine actions are:

```
return ::  a -> IO a
>>= ::  IO a -> (a -> IO b) -> IO b
```

- Executing the action `(ac1 >>= ac2)` executes `ac1` unboxes the resulting value, applies `ac2` to get an action and executes that action.

# Actions: More details

- Actions are terms of type `IO a` for some `a`.
- Two basic functions used to construct and combine actions are:

```
return ::  a -> IO a
>>= ::  IO a -> (a -> IO b) -> IO b
```

- Executing the action `(ac1 >>= ac2)` executes `ac1` unboxes the resulting value, applies `ac2` to get an action and executes that action.

Actually, `IO` is an example of a Monad and these functions are available in any Monad.

# Actions: More details

- Actions are terms of type `IO a` for some `a`.
- Two basic functions used to construct and combine actions are:

  ```
  return ::  a -> IO a
  >>= ::  IO a -> (a -> IO b) -> IO b
  ```

- Executing the action `(ac1 >>= ac2)` executes `ac1` unboxes the resulting value, applies `ac2` to get an action and executes that action.

  Actually, `IO` is an example of a Monad and these functions are available in any Monad.

- There are other functions such as `readLn, putStrLn, ...` that are specfic to certain `IO` types.

# Composing Actions

- Read a line and print it.

# Composing Actions

- Read a line and print it.

  ```
  getLine >>= putStrLn
  ```

# Composing Actions

- Read a line and print it.

  `getLine >>= putStrLn`
- What if we want to print the length of the string?

# Composing Actions

- Read a line and print it.

  `getLine >>= putStrLn`

- What if we want to print the length of the string?

  `getLine :: IO String`

# Composing Actions

- Read a line and print it.

  `getLine >>= putStrLn`

- What if we want to print the length of the string?

  `getLine ::  IO String`

- That should be coupled with a term of type `String -> IO ()` which prints the length of a given string.

# Composing Actions

- Read a line and print it.

  `getLine >>= putStrLn`
- What if we want to print the length of the string?

  `getLine ::  IO String`
- That should be coupled with a term of type `String -> IO ()` which prints the length of a given string.

  `getLine >>= \x -> (putStrLn (show (length x)))`

# Composing Actions

- Read a line and print it.

  `getLine >>= putStrLn`

- What if we want to print the length of the string?

  `getLine ::  IO String`

- That should be coupled with a term of type `String -> IO ()` which prints the length of a given string.

  `getLine >>= \x -> (putStrLn (show (length x)))`

- What if we wanted to print the length two times?

# Composing Actions

- Read a line and print it.

  ```
  getLine >>= putStrLn
  ```

- What if we want to print the length of the string?

  ```
  getLine ::  IO String
  ```

- That should be coupled with a term of type `String -> IO ()` which prints the length of a given string.

  ```
  getLine >>= \x -> (putStrLn (show (length x)))
  ```

- What if we wanted to print the length two times?

  ```
  getLine >>= \x ->
           (putStrLn (show (length x)) >>=
                   putStrLn (show (length x)))
  ```

# Composing Actions

- Read a line and print it.

  ```
  getLine >>= putStrLn
  ```

- What if we want to print the length of the string?

  ```
  getLine ::  IO String
  ```

- That should be coupled with a term of type `String -> IO ()` which prints the length of a given string.

  ```
  getLine >>= \x -> (putStrLn (show (length x)))
  ```

- What if we wanted to print the length two times?

  ```
  getLine >>= \x ->
        (putStrLn (show (length x)) >>=
             putStrLn (show (length x)))
  ```

- Alas, that does not type check!

# Composing Actions

- Read a line and print it.

  ```
  getLine >>= putStrLn
  ```

- What if we want to print the length of the string?

  ```
  getLine ::  IO String
  ```

- That should be coupled with a term of type `String -> IO ()` which prints the length of a given string.

  ```
  getLine >>= \x -> (putStrLn (show (length x)))
  ```

- What if we wanted to print the length two times?

  ```
  getLine >>= \x ->
          (putStrLn (show (length x)) >>=
                 putStrLn (show (length x)))
  ```

- Alas, that does not type check!

  ```
  getLine >>= \x ->
          (putStrLn (show (length x)) >>=
             \y -> putStrLn (show (length x)))
  ```

# The operator `>>`

```
p  >> q =  p >>= \n -> q
```

# The operator `>>`

```
p  >> q =  p >>= \n -> q
```

- Thus we may rewrite the previous program as

# The operator >>

```
p  >> q =  p >>= \n -> q
```

- Thus we may rewrite the previous program as

```
getLine >>= \x ->
       (putStrLn (show (length x)) >>
           putStrLn (show (length x)))
```

# Translating `do` ... using `>>=`

- A single actions needs no `do`

```
do              --->        exp
 exp
```

# Translating `do` . . . using `>>=`

- A single actions needs no `do`

```
do             --->        exp
 exp
```

- No `<-` in the first action,

# Translating `do` . . . using `>>=`

- A single actions needs no `do`

```
do             --->         exp
 exp
```

- No `<-` in the first action, any value it returns can be thrown away!

```
do             --->      exp >> do
 exp                             S
 S
```

# Translating `do` ... using `>>=`

- A single actions needs no `do`

```
do               --->         exp
 exp
```

- No `<-` in the first action, any value it returns can be thrown away!

```
do               --->      exp >> do
 exp                                S
 S
```

- `name <- exp` is the first action.

# Translating `do` ... using `>>=`

- A single actions needs no `do`

```
do                --->          exp
 exp
```

- No `<-` in the first action, any value it returns can be thrown away!

```
do                --->       exp >> do
 exp                                 S
 S
```

- `name <- exp` is the first action. Bind the value returned by the first action to the name `name` and ...

```
do                ---->      exp >>=
 name <- exp                  \name -> do
 S                                      S
```

# Example

```
main = do
         putStrLn ("Please enter your name:")
         name <- getLine
         putStrLn ("Hello " ++ name)
```

# Example

```
main = do
         putStrLn ("Please enter your name:")
         name <- getLine
         putStrLn ("Hello " ++ name)
```

$\rightsquigarrow$

```
main =   putStrLn ("Please enter your name:") >>
            do
             name <- getLine
             putStrLn ("Hello " ++ name)
```

# Example

```
main = do
         putStrLn ("Please enter your name:")
         name <- getLine
         putStrLn ("Hello " ++ name)
```

$\leadsto$

```
main =   putStrLn ("Please enter your name:") >>
            do
             name <- getLine
             putStrLn ("Hello " ++ name)
```

$\leadsto$

```
main =   putStrLn ("Please enter your name:") >>
             getLine >>=
                  \name ->
                        do
                         putStrLn ("Hello " ++ name)
```

$\leadsto$

# Example ...

```
main =   putStrLn ("Please enter your name:") >>
             getLine >>=
                  \name ->
                      do
                       putStrLn ("Hello " ++ name)
```

# Example ...

```
main =   putStrLn ("Please enter your name:") >>
              getLine >>=
                   \name ->
                       do
                        putStrLn ("Hello " ++ name)
```

⇝

```
main =   putStrLn ("Please enter your name:") >>
              getLine >>=
                   \name ->
                        putStrLn ("Hello " ++ name)
```

# Reading and Writing into Files

- To supply input from a file to an executable we can use input redirection

  ```
  $./myprogram < inputfile
  ```

# Reading and Writing into Files

- To supply input from a file to an executable we can use input redirection

  ```
  $./myprogram < inputfile
  ```

- The output of a program can be saved in a file we can use output redirection

  ```
  $./myprogram > outputfile
  ```

# Reading and Writing into Files

- To supply input from a file to an executable we can use input redirection

  ```
  $./myprogram < inputfile
  ```

- The output of a program can be saved in a file we can use output redirection

  ```
  $./myprogram > outputfile
  ```

- We can redirect input and output together

  ```
  $./myprogram < inputfile > outputfile
  ```

# Reading and Writing into Files

- To supply input from a file to an executable we can use input redirection

  ```
  $./myprogram < inputfile
  ```

- The output of a program can be saved in a file we can use output redirection

  ```
  $./myprogram > outputfile
  ```

- We can redirect input and output together

  ```
  $./myprogram < inputfile > outputfile
  ```

- File input output can also be done directly from within the program.

# File I/O ...

- To read or write from file, it has to be opened in the approriate mode.

  ```
  myHandle <- openFile "inputfile" ReadMode
  ```

# File I/O ...

- To read or write from file, it has to be opened in the approriate mode.

  ```
  myHandle <- openFile "inputfile" ReadMode
  ```

- The type of `openFile` is

  ```
  FilePath -> IOMode -> IO Handle
  ```

# File I/O ...

- To read or write from file, it has to be opened in the approriate mode.

  ```
  myHandle <- openFile "inputfile" ReadMode
  ```

- The type of `openFile` is

  ```
  FilePath -> IOMode -> IO Handle
  ```

  It takes the name of the file, the mode it which it is to be opened, and returns a handle.

# File I/O ...

- To read or write from file, it has to be opened in the approriate mode.

  `myHandle <- openFile "inputfile" ReadMode`

- The type of `openFile` is

  `FilePath -> IOMode -> IO Handle`

  It takes the name of the file, the mode it which it is to be opened, and returns a handle.

- The mode can be one of `ReadMode`, `WriteMode`, `AppendMode` and `ReadWriteMode`.

# File I/O ...

- To read or write from file, it has to be opened in the approriate mode.
  `myHandle <- openFile "inputfile" ReadMode`
- The type of `openFile` is
  `FilePath -> IOMode -> IO Handle`

  It takes the name of the file, the mode it which it is to be opened, and returns a handle.
- The mode can be one of `ReadMode`, `WriteMode`, `AppendMode` and `ReadWriteMode`.
  - `WriteMode` begins by creating the file from scratch, deleting anything that is already there.
  - `AppendMode` appends to the end of an existing file and creating the file if it does not exist.
- The handle is used in operations pertaining to the opened file.

# File I/O ...

- To read or write from file, it has to be opened in the approriate mode.
  `myHandle <- openFile "inputfile" ReadMode`
- The type of `openFile` is
  `FilePath -> IOMode -> IO Handle`

  It takes the name of the file, the mode it which it is to be opened, and returns a handle.
- The mode can be one of `ReadMode`, `WriteMode`, `AppendMode` and `ReadWriteMode`.
  - `WriteMode` begins by creating the file from scratch, deleting anything that is already there.
  - `AppendMode` appends to the end of an existing file and creating the file if it does not exist.
- The handle is used in operations pertaining to the opened file.
- The function `hClose :: Handle -> IO ()` closes the file.

# File I/O ...

- To read or write from file, it has to be opened in the approrate mode.
  `myHandle <- openFile "inputfile" ReadMode`
- The type of `openFile` is
  `FilePath -> IOMode -> IO Handle`

  It takes the name of the file, the mode it which it is to be opened, and returns a handle.
- The mode can be one of `ReadMode, WriteMode, AppendMode` and `ReadWriteMode`.
  - `WriteMode` begins by creating the file from scratch, deleting anything that is already there.
  - `AppendMode` appends to the end of an existing file and creating the file if it does not exist.
- The handle is used in operations pertaining to the opened file.
- The function `hClose :: Handle -> IO ()` closes the file.
- The module `System.IO` has to be imported.

# File I/O

- Once a file is open we can perform I/O operations using a variety of functions.

# File I/O

- Once a file is open we can perform I/O operations using a variety of functions.
- To read a line we can use

  ```
  hGetLine ::  Handle -> IO String
  ```

# File I/O

- Once a file is open we can perform I/O operations using a variety of functions.
- To read a line we can use

  `hGetLine :: Handle -> IO String`

  - Starts from the first position.
  - Each successive call continues reading from the position where the where the previous call ended.

# File I/O

- Once a file is open we can perform I/O operations using a variety of functions.
- To read a line we can use

  ```
  hGetLine :: Handle -> IO String
  ```

  - Starts from the first position.
  - Each successive call continues reading from the position where the where the previous call ended.

  and to write to a file we can use

  ```
  hPutStrLn :: Handle -> String -> IO ()
  ```

# File I/O

- Once a file is open we can perform I/O operations using a variety of functions.
- To read a line we can use

  `hGetLine :: Handle -> IO String`

  - Starts from the first position.
  - Each successive call continues reading from the position where the where the previous call ended.

  and to write to a file we can use

  `hPutStrLn :: Handle -> String -> IO ()`

- To read other types combine `hGetLine` with the function

  `read :: Read a => String -> a` that works like `readLn`

# File I/O

- Once a file is open we can perform I/O operations using a variety of functions.
- To read a line we can use

  ```
  hGetLine ::  Handle -> IO String
  ```

  - Starts from the first position.
  - Each successive call continues reading from the position where the where the previous call ended.

  and to write to a file we can use

  ```
  hPutStrLn ::  Handle -> String -> IO ()
  ```

- To read other types combine `hGetLine` with the function

  `read ::  Read a => String -> a` that works like `readLn`
- Here is function to read in an `Int` from a line

  ```
  hReadInt  h = (hGetLine h) >>=
                     \x -> return ((read :: Int) x)
  ```

# File I/O – examples

- Copy two lines from the file `FileA` to the file `FileB`

```
copyOneLine :: Handle  -> Handle -> IO ()
copyOneLine hi ho = do
                    inp  <- hGetLine hi
                    hPutStrLn ho inp


main = do
       hinp <- openFile "FileA" ReadMode
       hout <- openFile "FileB" WriteMode
       copyOneLine hinp hout
       copyOneLine hinp hout
       hClose hinp
       hClose hout
```

# Copying Files

- To copy the entire file, we must know when to stop.

# Copying Files

- To copy the entire file, we must know when to stop.

  The function `hIsEOF :: Handle -> IO Bool` ”returns” `True` if the current position has reached the end of the file.

# Copying Files

- To copy the entire file, we must know when to stop.

  The function `hIsEOF ::  Handle -> IO Bool` ”returns” `True` if the current position has reached the end of the file.
- We can copy a files using

```
copyAll :: Handle -> Handle  -> IO ()
copyAll hi ho =
       do
        over <- hIsEOF hi
        if over
         then return ()
         else
           do
            copyOneLine hi ho
            copyAll hi ho
```

# The `stdin`,`stdout` and `stderr`

- The Handle `stdin` is available to read from the standard input which is the keyboard unless the input has been redirected.

# The `stdin`,`stdout` and `stderr`

- The Handle `stdin` is available to read from the standard input which is the keyboard unless the input has been redirected.
- The Handle `stdout` is available to write to the standard output which is the screen unless the output has been redirected.

# The `stdin`,`stdout` and `stderr`

- The Handle `stdin` is available to read from the standard input which is the keyboard unless the input has been redirected.

- The Handle `stdout` is available to write to the standard output which is the screen unless the output has been redirected.

- The Handle `stderr` is available to write to the standard error, where error messages are printed, again set to the screen unless explicitely redirected.

# The `stdin`,`stdout` and `stderr`

- The Handle `stdin` is available to read from the standard input which is the keyboard unless the input has been redirected.
- The Handle `stdout` is available to write to the standard output which is the screen unless the output has been redirected.
- The Handle `stderr` is available to write to the standard error, where error messages are printed, again set to the screen unless explicitely redirected.

  `copyAll stdin stdout` will act like echo

# The `stdin`,`stdout` and `stderr`

- The Handle `stdin` is available to read from the standard input which is the keyboard unless the input has been redirected.
- The Handle `stdout` is available to write to the standard output which is the screen unless the output has been redirected.
- The Handle `stderr` is available to write to the standard error, where error messages are printed, again set to the screen unless explicitely redirected.

  `copyAll stdin stdout` will act like echo
- To generate `EOF` from the keyboard use `^D`

# Moving around a file

- A file is a sequence of characters. The positions are numbered starting at 0.

# Moving around a file

- A file is a sequence of characters. The positions are numbered starting at `0`.
- The `Handle` maintains our current position within the file.

# Moving around a file

- A file is a sequence of characters. The positions are numbered starting at `0`.
- The `Handle` maintains our current position within the file.
- Each time we read or write a character the position increases by 1.

# Moving around a file

- A file is a sequence of characters. The positions are numbered starting at `0`.
- The `Handle` maintains our current position within the file.
- Each time we read or write a character the position increases by 1.
- The function `hTell` ”returns” the current position.

# Moving around a file

- A file is a sequence of characters. The positions are numbered starting at `0`.
- The `Handle` maintains our current position within the file.
- Each time we read or write a character the position increases by 1.
- The function `hTell` "returns" the current position.
- The function `hSeek` can be used to move to a different position within the file.
  - Relative to the current position (`RelativeSeek`)
  - Relative to beginning of the file (`AbsoluteSeek`)
  - Relative to the end of the file (`SeekFromEnd`)

# Seek and Tell

- Write at the word "The" at the beginning of the file.

# Seek and Tell

- Write at the word "The" at the beginning of the file.

  ```
  hSeek h AbsoluteSeek 0
  hPutChar h 'T'
  hPutChar h 'h'
  hPutChar h 'e'
  ```

  Assumes that `h` is a handle to a file opened in write or read-write mode.

# Seek and Tell

- Write at the word ”The” at the beginning of the file.

  ```
  hSeek h AbsoluteSeek 0
  hPutChar h 'T'
  hPutChar h 'h'
  hPutChar h 'e'
  ```

  Assumes that `h` is a handle to a file opened in write or read-write mode.
- Copy the current character to the next position.

# Seek and Tell

- Write at the word "The" at the beginning of the file.

  ```
  hSeek h AbsoluteSeek 0
  hPutChar h 'T'
  hPutChar h 'h'
  hPutChar h 'e'
  ```

  Assumes that `h` is a handle to a file opened in write or read-write mode.
- Copy the current character to the next position.

  ```
  do
   c <- hGetChar h
   hPutChar h c
  ```

  Assumes that `h` is a handle opened in read-write mode.

# Seek and Tell

- Write at the word "The" at the beginning of the file.

  ```
  hSeek h AbsoluteSeek 0
  hPutChar h 'T'
  hPutChar h 'h'
  hPutChar h 'e'
  ```

  Assumes that `h` is a handle to a file opened in write or read-write mode.
- Copy the current character to the next position.

  ```
  do
   c <- hGetChar h
   hPutChar h c
  ```

  Assumes that `h` is a handle opened in read-write mode.
- Copy the current character to the end of the file.

# Seek and Tell

- Write at the word "The" at the beginning of the file.

  ```
  hSeek h AbsoluteSeek 0
  hPutChar h 'T'
  hPutChar h 'h'
  hPutChar h 'e'
  ```

  Assumes that `h` is a handle to a file opened in write or read-write mode.
- Copy the current character to the next position.

  ```
  do
   c <- hGetChar h
   hPutChar h c
  ```

  Assumes that `h` is a handle opened in read-write mode.
- Copy the current character to the end of the file.

  ```
  do
   c <- hGetChar h
   hSeek h SeekFromEnd 0
   hPutChar h c
  ```

# Files and File I/O

- Files exist across different runs of a program.

  Any data that needs to be stored across runs has to be in files.

# Files and File I/O

- Files exist across different runs of a program.

  Any data that needs to be stored across runs has to be in files.
- Files allow different programs to exchange information.

  One program may write its output to a file from which another reads.

# Files and File I/O

- Files exist across different runs of a program.

  Any data that needs to be stored across runs has to be in files.
- Files allow different programs to exchange information.

  One program may write its output to a file from which another reads.
- Files reside on the disk and can typically store much more data than in memory.

  Large databases are manipulated by programs by loading just parts into the memory.

# Files and File I/O

- Files exist across different runs of a program.

  Any data that needs to be stored across runs has to be in files.
- Files allow different programs to exchange information.

  One program may write its output to a file from which another reads.
- Files reside on the disk and can typically store much more data than in memory.

  Large databases are manipulated by programs by loading just parts into the memory.
- Files can also be used to exchange data between programs running in different machines.

# Files: Examples

- A simple Music Library.
    - Song
    - Artist
    - Album
    - the number of times listed to.

# Files: Examples

- A simple Music Library.
  - Song
  - Artist
  - Album
  - the number of times listed to.
- Each item occupies four lines.

# Files: Examples

- A simple Music Library.
    - Song
    - Artist
    - Album
    - the number of times listed to.
- Each item occupies four lines.
- Entries are separated by a blank line.

# Files: Examples

- A simple Music Library.
  - Song
  - Artist
  - Album
  - the number of times listed to.
- Each item occupies four lines.
- Entries are separated by a blank line.
- Find the song that I have listened to most?

# Files: Examples

- A simple Music Library.
    - Song
    - Artist
    - Album
    - the number of times listed to.
- Each item occupies four lines.
- Entries are separated by a blank line.
- Find the song that I have listened to most?
- List the songs in the order of the number of times listened to?

# Music Library ...

```
Body and Soul
Django Reinhardt
Djangology Vol 5
4
Entertainer
Scott Joplin
A-Z Encyclopedia of Jazz
10
Romanian Folk Dances for the Piano
Bela Bartok
Concertos for the piano and orchestra Vol 2/3
4
Raag Ek Nishad Ka Behagda
Mallikarjun Mansur

12
...
```

# Load and process

- Each entry is a four tuple

```
type Entry = (String,String,String,Int)
     deriving (Eq,Ord,Show)
main = do
hd <- openFile "MusicData" ReadMode
el <- fillEList hd []
---         Process the list el using pure functions
---         Write down el at the end.
```

# Load and process

- Each entry is a four tuple

```
type Entry = (String,String,String,Int)
     deriving (Eq,Ord,Show)
main = do
hd <- openFile "MusicData" ReadMode
el <- fillEList hd []
---          Process the list el using pure functions
---          Write down el at the end.
```

- Construct a list of entries from the file.

```
fillEList ::  Handle -> [Entry] -> IO [Entry]
```

- Find the song most listened to.

```
most ::  [Entry] -> Entry
```

- Increment its count by 1 (listen to it).

```
incCount ::  Entry -> Entry
```

...

# fillElist...

```
fillEList h l =
        do
         over <- hIsEOF h
         if (over)
         then
          return l
         else
          do
           song <- hGetLine h
           artist <- hGetLine h
           album <- hGetLine h
           times <-
            (hGetLine h >>= (read::Int))
           hGetLine h
           nl <- fillEList h l
           return  ((song,artist,album,times):nl)
```

- Populates a list with the contents of the file.

# Records in Haskell : An aside

- Tuples allow grouping if values.

```
type Entry = (String,String,String,Int)
```

# Records in Haskell : An aside

- Tuples allow grouping if values.

```
type Entry = (String,String,String,Int)
```

- We may also use user defined datatypes.

```
data Entry = Entry String String String Int
```

# Records in Haskell : An aside

- Tuples allow grouping if values.

```
type Entry = (String,String,String,Int)
```

- We may also use user defined datatypes.

```
data Entry = Entry String String String Int
```

- In both cases we will need extractor functions to access the components:

```
song :: Entry -> String
artist :: Entry -> String
album :: Entry -> String
times :: Entry -> Int
```

# Records in Haskell : An aside

- Tuples allow grouping if values.

```
type Entry = (String,String,String,Int)
```

- We may also use user defined datatypes.

```
data Entry = Entry String String String Int
```

- In both cases we will need extractor functions to access the components:

```
song :: Entry -> String
artist :: Entry -> String
album :: Entry -> String
times :: Entry -> Int
```

- Haskell allows you define extractor functions simultaneously with the definition of the data type.

# Record Syntax

- We can define `Entry` as follows:

```
data Entry = Entry {
 song :: String,
 artist :: String,
 album :: String,
 times :: Int
} deriving (Ord, Eq, Show)
```

# Record Syntax

- We can define `Entry` as follows:

```
data Entry = Entry {
 song :: String,
 artist :: String,
 album :: String,
 times :: Int
} deriving (Ord, Eq, Show)
```

- This has the effect of

# Record Syntax

- We can define `Entry` as follows:

```
data Entry = Entry {
 song :: String,
 artist :: String,
 album :: String,
 times :: Int
} deriving (Ord, Eq, Show)
```

- This has the effect of
  - Defining a datatype
    `data Entry = Entry String String String Int`

# Record Syntax

- We can define `Entry` as follows:

```
data Entry = Entry {
 song :: String,
 artist :: String,
 album :: String,
 times :: Int
} deriving (Ord, Eq, Show)
```

- This has the effect of
  - Defining a datatype
    `data Entry = Entry String String String Int`
  - providing accessor functions `song, artist, album` and`times`

# Record Syntax

- We can define `Entry` as follows:

```
data Entry = Entry {
 song :: String,
 artist :: String,
 album :: String,
 times :: Int
} deriving (Ord, Eq, Show)
```

- This has the effect of
  - Defining a datatype
    `data Entry = Entry String String String Int`
  - providing accessor functions `song, artist, album` and`times`

- Values can be created as

```
entry = Entry { song = "Tunji", artist = "John Coltrane
                album = "In a Soulful Mood",
                times = 3 }
```

# Record ...

```
data Entry = Entry {
 song :: String,
 artist :: String,
 album :: String,
 times :: Int

entry = Entry { song = "Tunji", artist = "John Coltrane
                album = "In a Soulful Mood",
                times = 3 }
```

# Record ...

```
data Entry = Entry {
 song :: String,
 artist :: String,
 album :: String,
 times :: Int

entry = Entry { song = "Tunji", artist = "John Coltrane
                album = "In a Soulful Mood",
                times = 3 }
```

- As expected we have

  ```
  song entry = "Tunji"
  ```

  ```
  times entry = 3
  ```

  ...

# Record ...

- The order of entries is unimportant if we use the record notation.

# Record ...

- The order of entries is unimportant if we use the record notation.

  ```
  entry = Entry { song = "Tunji",
                  album = "In a Soulful Mood",
                  artist = "John Coltrane",
                  times = 3 }
  ```

  has the same effect.

# Record ...

- The order of entries is unimportant if we use the record notation.

  ```
  entry = Entry { song = "Tunji",
                  album = "In a Soulful Mood",
                  artist = "John Coltrane",
                  times = 3 }
  ```

  has the same effect.

- We are also free to the datatype version.

  ```
  entry = Entry "Tunji" "In a Soulful Mood"
                "John Coltrane" 3
  ```

# Back to the Music Album

- Our current scheme of

# Back to the Music Album

- Our current scheme of
  - Load the data from the file to a list

# Back to the Music Album

- Our current scheme of
  - Load the data from the file to a list
  - work with the list

# Back to the Music Album

- Our current scheme of
    - Load the data from the file to a list
    - work with the list
    - Write the list to the file

# Back to the Music Album

- Our current scheme of
  - Load the data from the file to a list
  - work with the list
  - Write the list to the file

  has some flaws.

# Back to the Music Album

- Our current scheme of
  - Load the data from the file to a list
  - work with the list
  - Write the list to the file

  has some flaws.
- What if we change only one or two entries?

# Back to the Music Album

- Our current scheme of
  - Load the data from the file to a list
  - work with the list
  - Write the list to the file

  has some flaws.

- What if we change only one or two entries?

  We still have to write the entire file.

# Back to the Music Album

- Our current scheme of
  - Load the data from the file to a list
  - work with the list
  - Write the list to the file

  has some flaws.
- What if we change only one or two entries?

  We still have to write the entire file.
- Can we selectively write only the changed entries?

# Selective Writing

# Selective Writing

- Seek to the correct position in the file and write.

# Selective Writing

- Seek to the correct position in the file and write.
- How do we know the correct position in the file?

# Selective Writing

- Seek to the correct position in the file and write.
- How do we know the correct position in the file?
- Modify `fillELlist` to also record for each entry the position where it begins in the file.

```
type Entry = (String,String,String,Int,Int)
...

fillEllist
...
     do
      pos = hTell h
      song = hGetLine h
      ...
```

# Selective Writing

- Seek to the correct position in the file and write.
- How do we know the correct position in the file?
- Modify `fillELlist` to also record for each entry the position where it begins in the file.

```
type Entry = (String,String,String,Int,Int)
...

fillEllist
...
     do
      pos = hTell h
      song = hGetLine h
      ...
```

- To write an entry, seek to its position and then write.

# A Flaw in our idea

- What happens when the count goes from `9` to `10`.

# A Flaw in our idea

- What happens when the count goes from `9` to `10`.
- We overwrite the first character in the next entry!

# A Flaw in our idea

- What happens when the count goes from 9 to 10.
- We overwrite the first character in the next entry!
- Either write everything from there on.

# A Flaw in our idea

- What happens when the count goes from 9 to 10.
- We overwrite the first character in the next entry!
- Either write everything from there on.
- or assuming fixed length changable fields.

# A Flaw in our idea

- What happens when the count goes from `9` to `10`.
- We overwrite the first character in the next entry!
- Either write everything from there on.
- or assuming fixed length changable fields.

  Allocate 4 characters for `times` always.