# PROGRAMMING IN HASKELL



Monads

# Introduction

In the course we have made two uses of the do-notation so far:



- Parsers
- IO



Parsers and IO are two instanc stract structure called a monad.

# Parsing using the do-notation

A sequence of parsers can be combined as a single composite parser using the keyword do.



For example:





```
p :: Parser (Char,Char)
p = do x ← item
       item
       y ← item
       return (x,y)
```

# IO using the do-notation

A sequence of actions can be combined as a single composite action using the keyword do.



For example:





```
a :: IO (Char,Char)
a  = do x ← getChar
        getChar
        y ← getChar
        return (x,y)
```

# Monads

- Monads are a structure composed of two basic operations (bind and return), which capture a common pattern that occurs in many types.



- In Haskell Monads are defined using type classes:

```
class Monad m where
  (>>=)  :: m a -> (a -> m b) -> m b
  return :: a -> m a
```

# Do-notation

- The do-notation is just simple syntactic sugar on top of the monad operations:



```
do  pattern
    morelines
```

Is converted to code using bind:

```
exp >>= (\pattern -> do morelines)
```

# Do-notation

- The do-notation is just simple syntactic sugar on top of the monad operations:



```
do   exp
     morelines
```

Is converted to code using bind:

```
exp >>= (\_ -> do morelines)
```

# Do-notation

- The do-notation is just simple syntactic sugar on top of the monad operations:



```
do   return e
```



Is simplified to:

```
return exp
```

# Parsing using the do-notation

Lets rewrite a program using the do-notation!



```
do x ← item
   item
   y ← item
   return (x,y)
```

# Parsing using the do-notation

Lets rewrite a program using the do-notation!

```
item >>= (\x ->
  item >>= (
    item >>=
      return (x,y))))
```



What are the steps?

# Parsing using the do-notation

Lets rewrite a program using the do-notation!

```
item >>= \x
   do  item
       y ← item
       return (x,y)
```

# Parsing using the do-notation

Lets rewrite a program using the do-notation!

```
item >>= \x
   item >>= \
     do y ← item
        return (x,y)
```

# Parsing using the do-notation

Lets rewrite a program using the do-notation!

```
item >>= \x ->
   item >>= \y ->
      do return (x,y)
```

# Parsing using the do-notation

Lets rewrite a program using the do-notation!

```
item >>= \x ->
   item >>= \y ->
      return (x,y)
```

# Creating Monads

- Monads are created using an instance of the Monad type class.



- The Parser m Monad





- The IO Monad is built-in

# Parser Monad

Here is the Parser Monad:

```
data Parser a =  P (String -> [(a,String)])
```



instance Monad



```
   — return :: a -> Parser a
   return v =  P (\inp -> [(v,inp)])
   — (>>=) :: Parser a -> (a -> Parser b) -> Parser b
   p >>= f  =  P (\inp -> case parse p inp of
                            []       -> []
                            [(v,out)] -> parse (f v) out)
```

# Parser Monad

- Simply providing this instance allows us to use the do-notation!



```
newtype Parser a =  P (String -                  )])

instance Monad Parser where
   return v =  P (\inp -> [(v,inp)])
   p >>= f  =  P (\inp -> case parse p inp of
                            []       -> []
                            [(v,out)] -> parse (f v) out)
```

# IO

IO is special in Haskell, since it is a type built-in the language. The compiler provides an instance of the Monad class with suitable return and >>= functions:



return :: a -> IO



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

# Creating a Simple Monad

Once we understand monads it is easy to create our own monads. An example is a variation of the Maybe type



```
data Option a =
```



```
instance Monad Option where
   ???
```



Lets create the Option Monad!

# Creating a Simple Monad

With option we can create a safer version of arithmetic expressions:



sdiv :: Option Int n Int



sadd :: Option Int -> Option In n Int



That track whether division by 0 errors occur.

# Monad Laws

It is not enough to implement bind and return. A proper monad is also required to satisfy some laws:



return a >>= k

m >>= return =



m >>= (\x -> k x >>= h) =



>>= h

# Monads, Functional Programming and Interpreters

- Monads were introduced to Functional Programming by Philip Wadler
- See the paper b which motivates monads through interpreters (much like the interpreters in the class)



The essence of Functional Programming, Philip Wadler, 1992