# CIS 1904: Haskell

Libraries

# Logistics

- HW11 will be released tonight
  - Last homework!
  - NOT based on the content of today’s class
  - Manually graded

# Parsec

- Haskell library for parsing
- Commonly used for domain-specific languages (DSLs)
  - Another package, Happy, is often used for compilers
- Focuses on modular design using *combinators*

# Parsec

```
data Parser a = ??
```

# Parsec

```
data Parser a = String -> a
```

# Parsec

```
data Parser a = String -> a
```

What if the input string cannot be parsed as an `a`?

# Parsec

```
data Parser a = String -> Maybe a
```

# Parsec

```haskell
data Parser a = String -> Maybe a
```

How do we access the rest of the string after parsing the first character?

# Parsec

```
data Parser a = String -> Maybe (a, String)
```

# Parsec

```
data Parser a = String -> Maybe (a, String)
```

Note: the actual Parsec definitions are much more complicated.

Parsec uses *monad transformers*, which let us model multiple effects at once by “nesting” several monads.

The above definition already includes the Maybe and (implicitly) State monads! We could also write the same thing as `StateT String Maybe a`.

State monad transformer

state type

inner monad

result type

# Parsec

```
data Parser a = … g -> Maybe (a, String)
```

Note: the actua… m…

Parsec uses *monad tran*… ultiple effects at once by “nesting” several monads…

The above definiti… ly) State monads! We could also … g as State…

State mo… sformer

result type

state type

inner monad

We will use simplified definitions here.

# Parsec: Combinators

```
data Parser a = String -> Maybe (a, String)
```

`satisfy` parses a character iff that character satisfies the input predicate.

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy f (x : xs) | f x = Just (x, xs)
satisfy _ _ = Nothing
```

# Parsec

```
data Parser a = String -> Maybe (a, String)
```

`char` parses the input character.

```
char :: Char -> Parser Char
char = satisfy . (==)
```

`digit` parses any digit.

```
digit :: Parser Char
digit = satisfy isDigit
```

# Parsec

```
data Parser a = String -> Maybe (a, String)
```

```
many :: Parser a -> Parser [a]
```

- applies the input parser zero or more times to parse a list of `a`s.

```
many1 :: Parser a -> Parser [a]
```

- applies the input parser one or more times to parse a list of `a`s.

Example:

```
chars :: Char -> Parser String
chars = many1 . char
```

# Parsec

```
data Parser a = String -> Maybe (a, String)
```

Remember, parsers are monads!

Example:

```
abcs :: Parser String
abcs = do
    xsA <- chars ‘a’
    xsB <- chars ‘b’
    xsC <- chars `c’
    return (xsA ++ xsB ++ xsC)
```

# Parsec

```
data Parser a = String -> Maybe (a, String)
```

`between` applies the third input parser, but only if it can apply the others around it.

```
between :: Parser o -> Parser c -> Parser a -> Parser a
```

Examples:

```
bracketedX :: Parser Char
bracketedX = between (char ‘[‘) (char ‘]’) (char ‘x’)

digitInAbcs :: Parser Char
digitInAbcs = between abcs abcs digit
```

# Parsec

```
data Parser a = String -> Maybe (a, String)
```

```
(<|>) :: Alternative f => f a -> f a -> f a
```

More specifically, we can use it to try one parser and, if that fails, try a second:

```
(<|>) :: Parser a -> Parser a -> Parser a
```

Examples

```
xOry :: Parser Char
xOry = char ‘x’ <|> char ‘y’
```

# stm

- Short for “Software Transational Memory”
- Designed to be modular and maintain abstractions
- Meant to address some of the traditional shortcomings of locks
  - Easy to forget, leading to race conditions
  - Easy to accidentally cause deadlock (or livelock)
  - Hard to scale
- Idea: model blocks of actions as atomic *transactions*. *After* block execution:
  - If no other thread accessed the same data while the block ran, make changes available to other threads
  - Otherwise, discard changes and restart the block
- Monadic structure and types enforce these ideas

# Concurrency with IORef

```
forkIO :: IO () -> IO ThreadId
```

As an I/O action, forks a new thread, starts the input action in that thread, and returns the ID of the new thread.

Note: we are not talking about stm specifically yet.

# Concurrency with IORef

```haskell
f :: IO ()
f = do
    print 1
    print 2
```

```haskell
g :: IO ()
g = do
    print 3
    print 4
```

```haskell
example :: IO ()
example = do
    forkIO f
    g
```

How many possible outputs does `example` have?

# Concurrency with IORef

```haskell
f :: IO ()
f = do
    print 1
    print 2
```

```haskell
g :: IO ()
g = do
    print 3
    print 4
```

```haskell
example :: IO ()
example = do
    forkIO f
    g
```

How many possible outputs does `example` have? 6

1234, 1324, 1342, 3124, 3142, 3412

# Concurrency with IORef

```
type Account = IORef Int
```

`IORef` is a way of modelling mutable state.
It is usually not the preferred option, for reasons we will see.

```
newIORef :: a -> IO (IORef a)

readIORef :: IORef a -> IO a

writeIORef :: IORef a -> a -> IO ()
```

# Concurrency with IORef

```
deposit :: Account -> Int -> IO ()
deposit acc amount = do
    balance <- readIORef acc
    writeIORef acc (balance + amount)


withdraw :: Account -> Int -> IO ()
withdraw acc amount = do
    balance <- readIORef acc
    writeIORef acc (balance - amount)
```

# Concurrency with IORef

```
transfer :: Account -> Account -> Int -> IO ()
transfer from to amount = do
    withdraw from amount
    deposit to amount


view :: Account -> IO ()
view acc = do
    balance <- readIORef acc
    print balance
```

# Concurrency with IORef

```
bankStuff :: IO ()
bankStuff = do
   acc1 <- newIORef 100                      create account and get its ID
   acc2 <- newIORef 100                      create account and get its ID
   forkIO (view acc1 >> view acc2)           in a new thread, print both balances
   replicateM_ 10 (transfer acc1 acc2)       transfer 10 from acc1 to acc2 10 times
```

Printing can happen mid-transfer!

If you run `bankStuff` enough times, you will get outputs like:

```
10
180
```

# stm

```haskell
type Account = TVar Int
```

`TVar` is a better way of modelling mutable state, for the reasons we mentioned at the beginning.

```haskell
newTVar :: a -> STM (TVar a)

readTVar :: TVar a -> STM a

writeTVar :: TVar a -> a -> STM ()
```

# stm

```
deposit :: Account -> Int -> TVar ()
deposit acc amount = do
    balance <- readTVar acc
    writeTVar acc (balance + amount)


withdraw :: Account -> Int -> TVar ()
withdraw acc amount = do
    balance <- readTVar acc
    writeTVar acc (balance - amount)
```

# stm

```
transfer :: Account -> Account -> Int -> STM ()
transfer from to amount = do
    withdraw from amount
    deposit to amount


view :: Account -> IO ()
view acc = do
    balance <- readTVarIO acc
    print balance
```

# stm

```
bankStuff :: IO ()
bankStuff = do
   acc1 <- newTVar 100                      create account and get its ID
   acc2 <- newTVar 100                      create account and get its ID
   forkIO (view acc1 >> view acc2)          in a new thread, print both balances
   replicateM_ 10 (atomically
     (transfer acc1 acc2))                  transfer 10 from acc1 to acc2 10 times
```

Printing cannot happen mid-transfer, so the values will always add up to 200.

# stm

```
bankStuff :: IO ()
bankStuff = do
    acc1 <- newTVar 100                      create account and get its ID
    acc2 <- newTVar 100                      create account and get its ID
    forkIO (view acc1 >> view acc2)          in a new thread, print both balances
    atomically (replicateM_ 10
        (transfer acc1 acc2))                transfer 10 from acc1 to acc2 10 times
```

Printing cannot happen mid-10-transfers, so the values will always either be both 100, or acc1 will have 0 and acc2 will have 200..

# stm

- `atomically` executes an STM action and *then* makes its result available in the IO monad
    - Enforces the atomic nature of it: if the STM action fails, we have not actually done any IO yet
- More details in the book Real World Haskell (chapter 28)
    - Available for free online
- Other key functions: `retry`, `orElse`

# Exercises

- Exercises.hs has parsing exercise
- Concurrent.hs has the previous example to read, no exercises
- No markdown files this week