Announcements

- Today: the Parsing Domain-specific Language
  - We' ll start describing the parsing operators in Parsing.lhs
  - …and how they are used to construct parsers, lexers, etc
  - Those who know Lex/Flex and Yacc/Bison will appreciate the ease with which parsers are constructed this way

Review: Handling an error

Haskell has a tool for representing computations that might fail:

```
data Maybe a = Just a | Nothing
```

- If (interp e) doesn't have any errors, then return (Just v) where v is the value of e
- Otherwise, signify error within (interp e) by returning Nothing

This is what we want

```
Arith> interp ex1
Just 3
Arith> interp ex2
Nothing
```

Review: do Notation

```
interp3 (Aexp Plus e1 e2) =
          do v1 <- interp3 e1
             v2 <- interp3 e2
             Just (v1+v2)
```

- evaluate (interp3 e1) first; if it is:
  - (Just v1), then strip off the Just,
  - Nothing, then the whole do expression is Nothing
- evaluate (interp3 e2) next; if it is:
  - (Just v2), then strip off the Just,
  - Nothing, then the whole do expression is Nothing
- If you've got both v1 and v2, Just (v1+v2)

Review: Interpreter, v3.0

```
interp3 :: Exp -> Maybe Int

interp3 (Const v)            = Just v
interp3 (Aexp Plus e1 e2) = do v1 <- interp3 e1
                               v2 <- interp3 e2
                               Just (v1+v2)
interp3 (Aexp Div e1 e2)  = do v1 <- interp3 e1
                               v2 <- interp3 e2
                               if v2==0
                                 then
                                   Nothing
                                 else
                                   Just (v1+v2)
```

Alternative Notation to do: >>=

```
data Maybe a = Just a | Nothing
```

```
Nothing  >>= f  = Nothing
(Just v) >>= f  = f v
```

>>= is pronounced "bind"

Question: what's the type of >>=?

Alternative Notation to do: >>=

```
data Maybe a = Just a | Nothing
```

```
Nothing  >>= f  = Nothing
(Just v) >>= f  = f v
```

Question: what's the type of >>=?

```
(>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
```

(>>=) is like a "backwards application"

Alternative to "do": bind >>=

do

```
interp3 (Aexp Plus e1 e2) =
           do v1 <- interp3 e1
              v2 <- interp3 e2
              Just (v1+v2)
```

>>=

```
interp4 (Aexp Plus e1 e2)
        = interp4 e1 >>= \ v1 ->
          interp4 e2 >>= \ v2 ->
            Just (v1+v2)
```

A note on notation

```
interp4 (Aexp Plus e1 e2)
        = interp4 e1 >>= \ v1 ->
          interp4 e2 >>= \ v2 ->
            Just (v1+v2)
```

is the same as

```
interp4 (Aexp Plus e1 e2)
        = (interp4 e1) >>= (\ v1 ->
          (interp4 e2) >>= (\ v2 ->
            Just (v1+v2)))
```

Interpreter, v4.0 (Arith4.hs)

```
interp4 :: Exp -> Maybe Int
interp4 (Const v) = Just v

interp4 (Aexp Plus e1 e2)
        = interp4 e1 >>= \ v1 ->
          interp4 e2 >>= \ v2 ->
            Just (v1+v2)

interp4 (Aexp Div e1 e2)
        = interp4 e1 >>= \ v1 ->
          interp4 e2 >>= \ v2 ->
            if v2==0
               then Nothing
               else Just (v1 `div` v2)
```

Alternative Notation to Just: return

```
data Maybe a = Just a | Nothing
```

```
Nothing  >>= f  = Nothing
(Just v) >>= f  = f v
return v        = Just v
```

Question: what's the type of **`return`**?

Alternative Notation to Just: return

```
data Maybe a = Just a | Nothing
```

```
Nothing  >>= f  = Nothing
(Just v) >>= f  = f v
return v        = Just v
```

Question: what's the type of **`return`**?

`return :: a -> Maybe a`

Interpreter, v4.1

```
interp4 :: Exp -> Maybe Int
interp4 (Const v) = return v

interp4 (Aexp Plus e1 e2)
        = interp4 e1 >>= \ v1 ->
          interp4 e2 >>= \ v2 ->
            return (v1+v2)

interp4 (Aexp Div e1 e2)
        = interp4 e1 >>= \ v1 ->
          interp4 e2 >>= \ v2 ->
            if v2==0
               then Nothing
               else return (v1 `div` v2)
```

Bind & return are overloaded

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

instance Monad Maybe where
  Nothing  >>= f  = Nothing
  (Just v) >>= f  = f v
  return v        = Just v
```

Many important instances of **Monad** class:

- IO monad for input/output
- Lists are a monad
- Monads are used to model “side effects”

Ultra-quick Intro to the IO Monad

```
tigerscheme :: IO ()
tigerscheme
    = do
      putStr "TigerScheme> “
      iline <- getLine
      putStr ("    " ++ iline ++ "\n”)
      tigerscheme
```

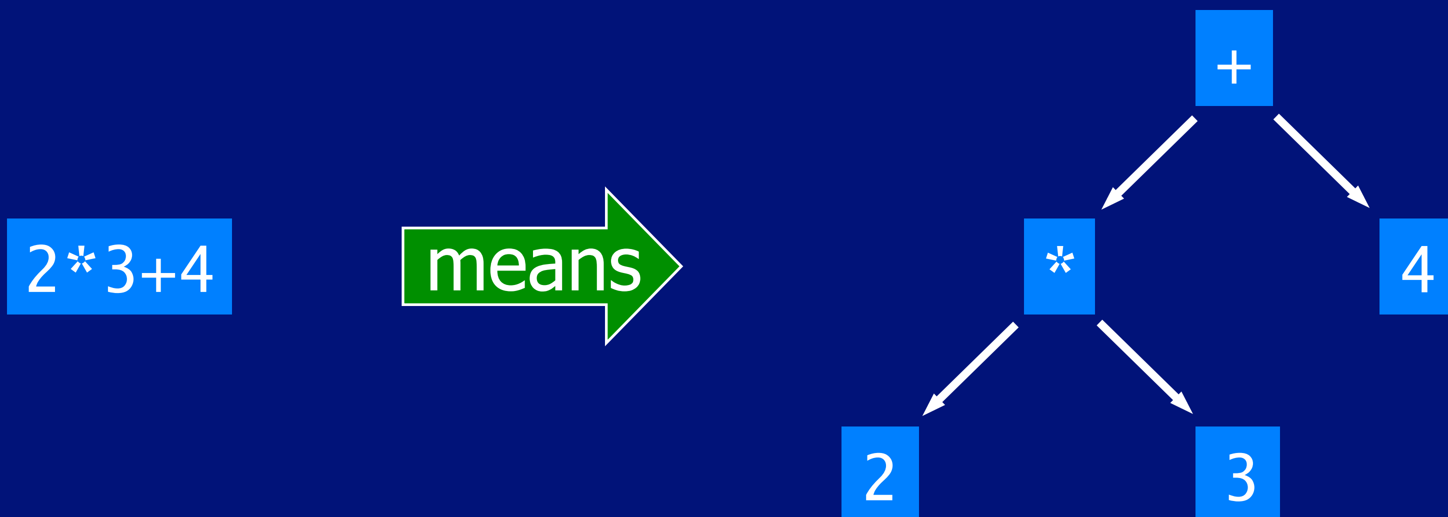
```
Hugs> tigerscheme
TigerScheme> (+ 1 2)
     (+ 1 2)
TigerScheme>
```

user input

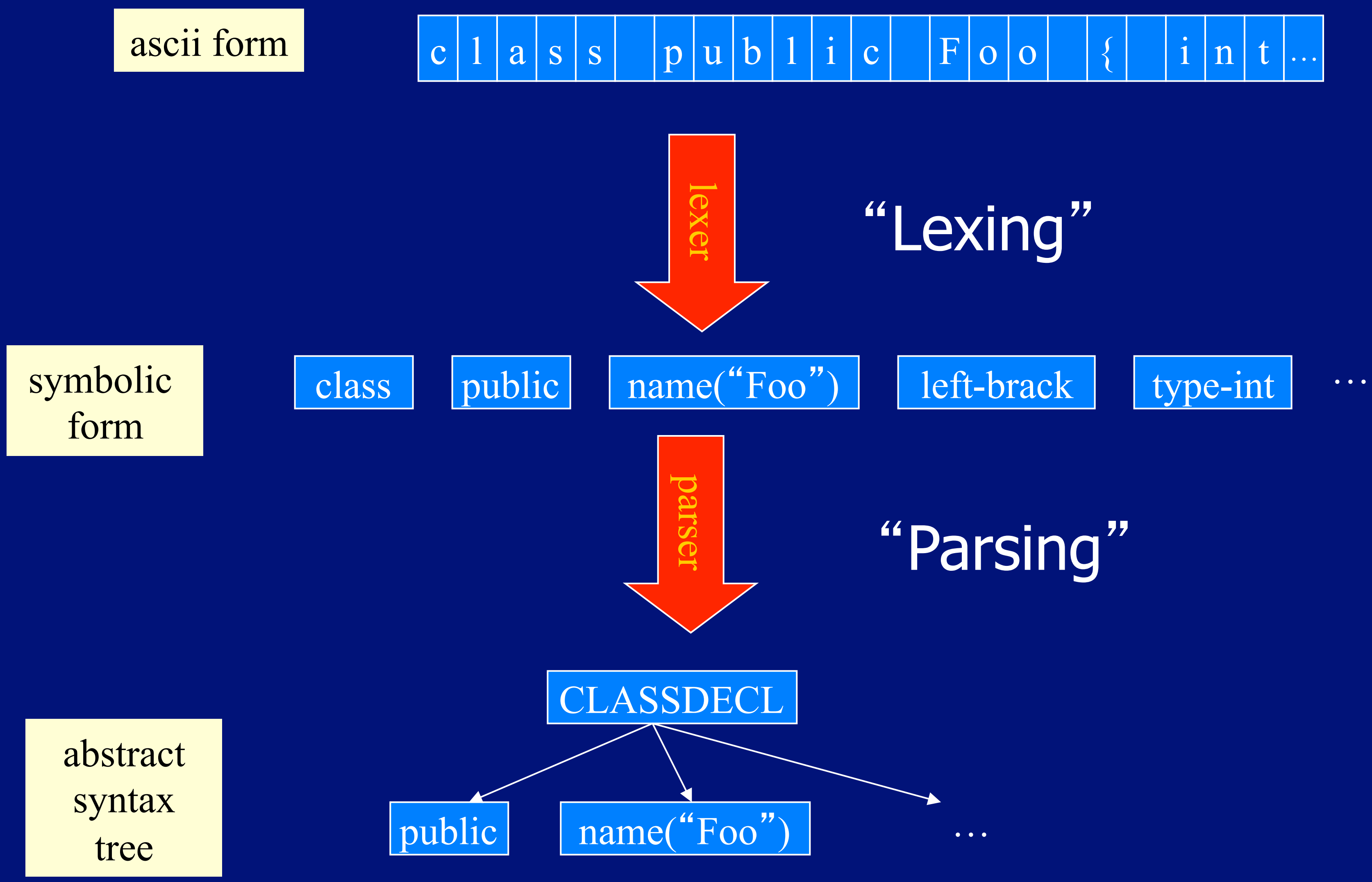
Building Parsers using Monads

What is a Parser?

A parser is a program that analyses a piece of text to determine its syntactic structure.



What a front-end does



Multiple Parse Trees

PLEASE PUT A RED BLOCK ON THE BLOCK IN THE BOX

- could mean "Look on the [previously mentioned] block, find a red block there, and put it in the box"
- or it could mean "Look for a red block and put it on the only block in the box.

"The syntax is ambiguous, but by looking at the positions of the blocks, we can deduce that one of the possible parses is nonsense, and therefore use the other one."

…have Artificial Intelligence applications;
for Programming Languages, not so much

Where Are They Used?

Almost every real life program uses some form of parser to <u>pre-process</u> its input.

GHCi	parses	Haskell programs
Unix	parses	Shell scripts
Explorer	parses	HTML documents

The Parser Type

In a functional language such as Haskell, parsers can naturally be viewed as functions.

```
type Parser = String → Tree
```

A parser is a function that takes a string and returns some form of tree.

However, a parser might not require all of its input string, so we also return any unused input:

```
type Parser = String → (Tree,String)
```

A string might be parsable in many ways, including none, so we generalize to a list of results:

```
type Parser = String → [(Tree,String)]
```

Finally, a parser might not always produce a tree, so we generalize to a value of <u>any type</u>:

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

Note:

- For simplicity, we will only consider parsers that either fail and return the empty list of results, or succeed and return a <u>singleton list</u>.

Basic Parsers

- The parser item fails if the input is empty, and consumes the first character otherwise:

```
String → [(Char,String)]
```

```
item :: Parser Char
item  = P (λinp → case inp of
                    []     → []
                    (x:xs) → [(x,xs)])
```

- The parser <u>failure</u> always fails:

```
failure :: Parser a
failure  = P (λinp → [])
```

- The parser <u>return v</u> always succeeds, returning the value v without consuming any input:

```
return   :: a → Parser a
return v = P (λinp → [(v,inp)])
```

- The parser (p +++ q) behaves as the parser p if it succeeds, and as the parser q otherwise:

```
(+++)  :: Parser a → Parser a → Parser a
p +++ q = P(λinp → case p inp of
                      []         → parse q inp
                      [(v,out)] → [(v,out)])
```

- The function parse applies a parser to a string:

```
parse :: Parser a → String → [(a,String)]
parse (P p) inp = p inp
```

Examples

The behavior of the five parsing primitives can be illustrated with some simple examples:

```
% ghci Parsing

> parse item ""
[]

> parse item "abc"
[('a',"bc")]
```

```
> parse failure "abc"
[]

> parse (return 1) "abc"
[(1,"abc")]

> parse (item +++ return 'd') "abc"
[('a',"bc")]

> parse (failure +++ return 'd') "abc"
[('d',"abc")]
```

Note:

- The library file Parsing is available on the web from the course home page.

- The Parser type is a monad, a mathematical structure that has proved useful for modeling many different kinds of computations.

Sequencing

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)
```

Note:

- Each parser must begin in precisely the same column. That is, the layout rule applies.

- The values returned by intermediate parsers are discarded by default, but if required can be named using the $\leftarrow$ operator.

- The value returned by the last parser is the value returned by the sequence as a whole.

- If any parser in a sequence of parsers fails, then the sequence as a whole fails. For example:

```
> parse p "abcdef"
[(('a','c'),"def")]

> parse p "ab"
[]
```

- The do notation is not specific to the Parser type, but can be used with any monadic type.

Derived Primitives

- Parsing a character that satisfies a predicate:

```
sat  :: (Char → Bool) → Parser Char
sat p = do x ← item
           if p x then
              return x
            else
              failure
```

- Parsing a digit and specific characters:

```
digit :: Parser Char
digit  = sat isDigit

char  :: Char → Parser Char
char x = sat (x ==)
```

- Applying a parser zero or more times:

```
many  :: Parser a → Parser [a]
many p = many1 p +++ return []
```

- Applying a parser one or more times:

```
many1   :: Parser a -> Parser [a]
many1 p = do v  ← p
             vs ← many p
             return (v:vs)
```

- Parsing a specific string of characters:

```
string        :: String → Parser String
string []      = return []
string (x:xs) = do char x
                   string xs
                   return (x:xs)
```

Example

We can now define a parser that consumes a list of one or more digits from a string:

```
p :: Parser String
p  = do char '['
        d  ← digit
        ds ← many (do char ','
                      digit)
        char ']'
        return (d:ds)
```

For example:

```
> parse p "[1,2,3,4]"
[("1234","")]

> parse p "[1,2,3,4"
[]
```

Note:

- More sophisticated parsing libraries can indicate and/or recover from <u>errors</u> in the input string.

Review: from Parsing.lhs

```
> sat    :: (Char -> Bool) -> Parser Char
> sat p = do x <- item
>            if p x then return x else failure
>
> digit :: Parser Char
> digit =  sat isDigit
>
> lower :: Parser Char
> lower =  sat isLower
```

```
> ident      :: Parser String
> ident      =  do x  <- lower
>                  xs <- many alphanum
>                  return (x:xs)
```

Example: ExpParser.hs*

Hutton's Parsing library; also at the course website

```
module ExpParser where
import Parsing

data Op  = Plus | Minus | Times | Div deriving Show

data Exp = Const Int | Aexp Op Exp Exp deriving Show
```

"deriving Show" means automatically define the instance

* Available at the course website

Parsing Ops and Consts

```
parseOp =
    do
      isym <- (symbol "+"
               +++ symbol "-"
               +++ symbol "*"
               +++ symbol "/”)
      return (tr isym)
          where
             tr "+" = Plus
             tr "-" = Minus
             tr "*" = Times
             tr "/" = Div

parseConst = do
               i <- integer
               return (Const i)
```

*Red = defined in Parser.lhs

Running Parsers

```
ExpParser> parse parseOp "*"
  [(Times,"")]

ExpParser> :t parse parseConst "99"
  parse parseConst "99" :: [(Exp,String)]
ExpParser> parse parseConst "99"
  [(Const 99,"")]
```

Parsing Aexps and Exp

```
parseAexp = do
              symbol "("
              op <- parseOp
              space
              e1 <- parseExp
              space
              e2 <- parseExp
              symbol ")"
              return (Aexp op e1 e2)

parseExp = parseConst +++ parseAexp
```

Parsing Exps

```
ExpParser> parse parseExp "(+ 1 2)"
  [(Aexp Plus (Const 1) (Const 2),"")]

ExpParser> parse parseExp "99"
  [(Const 99,"")]
```

Arithmetic Expressions

Consider a simple form of <u>expressions</u> built up from single digits using the operations of addition + and multiplication *, together with parentheses.

We also assume that:

- * and + associate to the right;
- * has higher priority than +.

Formally, the syntax of such expressions is defined by the following context free <u>grammar</u>:

```
expr   → term '+' expr | term

term   → factor '*' term | factor

factor → digit | '(' expr ')‘

digit  → '0' | '1' | … | '9'
```

However, for reasons of efficiency, one might factorise the rules for *expr* and *term*:

$$expr \rightarrow term\ ('+'\ expr \mid \varepsilon)$$

$$term \rightarrow factor\ ('*'\ term \mid \varepsilon)$$

Note:

- The symbol ε denotes the empty string.

It is now easy to translate the grammar into a parser that <u>evaluates</u> expressions, by simply rewriting the grammar rules using the parsing primitives.

That is, we will define:

```
expr :: Parser Int
expr  = …
term :: Parser Int
term = …
factor :: Parser Int
factor = …
```

It is now easy to translate the grammar into a parser that <u>evaluates</u> expressions, by simply rewriting the grammar rules using the parsing primitives.

expr → *term* ('+' *expr* | ε)

That is, we have:

```
expr :: Parser Int
expr  = do t ← term
           (do char '+'
               e ← expr
               return (t + e))
            +++ return t
```

term → factor ('' term | ε)*

```
term :: Parser Int
term  = do f ← factor
           (do char '*'
               t ← term
               return (f * t))
            +++ (return f)
```

factor → digit | '(' expr ')'

```
factor :: Parser Int
factor  = (do d ← digit
              return (digitToInt d))
           +++ (do char '('
                   e ← expr
                   char ')'
                   return e)
```

Finally, if we define

```
eval    :: String → Int
eval xs = fst (head (parse expr xs))
```

then we try out some examples:

```
> eval "2*3+4"
10

> eval "2*(3+4)"
14
```

Exercises

(1) Why does factorising the expression grammar make the resulting parser more efficient?

(2) Extend the expression parser to allow the use of subtraction and division, based upon the following extensions to the grammar:

$$expr \rightarrow term \; ('+' \; expr \mid '-' \; expr \mid \varepsilon)$$

$$term \rightarrow factor \; ('*' \; term \mid '/' \; term \mid \varepsilon)$$