

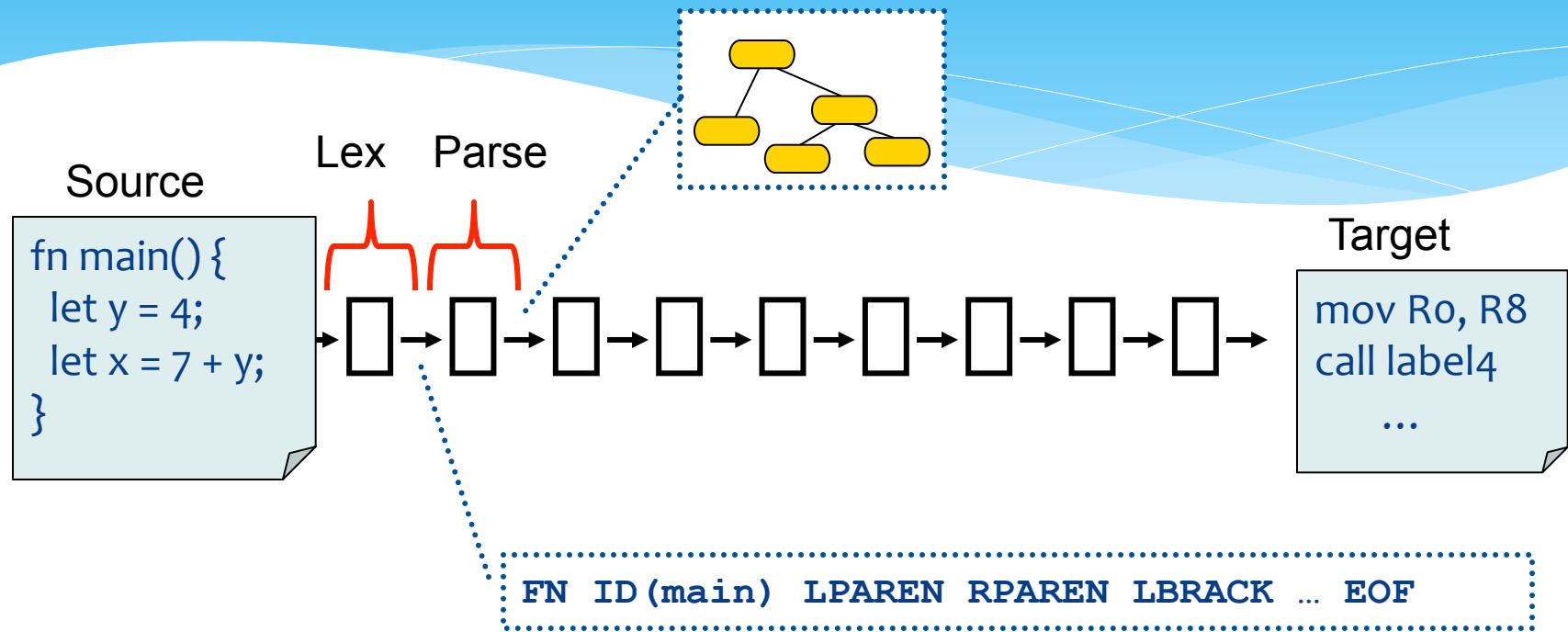
# Parsing Functionally

Bill Harrison  
CS 4430, Spring 2019

# Today

- \* “Parsing” or grammatical analysis
  - \* discovers the real structure of a program and represents it in a computationally useful way
- \* “Predictive” parsing
  - \* also called “recursive descent” parsing

# Compiler phases



Output of lexer is unstructured stream of symbols – how do we recognize the program structure in this stream?

# Thus far...

- \* A **language** is a set of strings
- \* Some languages may be described with a **context-free grammar**
  - \* Terminals: tokens from the lexer
  - \* Non-terminals: have production rules in our grammar
- \* A **parser** for a grammar/language determines whether a string belongs to a language or not
  - \* Parsing discovers a **derivation** (if one exists).
  - \* This derivation will let us build our **parse tree**.
- \* Grammars can be **ambiguous**
  - \* Admit several valid parses
  - \* Can transform grammar to
    - \* remove ambiguity (if necessary)
    - \* make it easier to parse

# Review: Simple CFG

$S \rightarrow \text{if } E \text{ then } S \text{ else } S$

$S \rightarrow \text{begin } S \text{ L}$

$S \rightarrow \text{print } E$

$L \rightarrow \text{end}$

$L \rightarrow ; \text{ S L}$

$E \rightarrow \text{num} = \text{num}$

# Review: Derivation

$S \rightarrow \text{if } E \text{ then } S \text{ else } S$

$S \rightarrow \text{begin } S L$

$S \rightarrow \text{print } E$

$L \rightarrow \text{end}$

$L \rightarrow ; S L$

$E \rightarrow \text{num} = \text{num}$

$S \rightarrow \text{begin } S L$

$\rightarrow \text{begin print } E L$

$\rightarrow \text{begin print } 1=1 L$

$\rightarrow \text{begin print } 1=1 \text{ end}$

# Review: Derivation

$S \rightarrow \text{if } E \text{ then } S \text{ else } S$

$S \rightarrow \text{begin } S L$

$S \rightarrow \text{print } E$

$L \rightarrow \text{end}$

$L \rightarrow ; S L$

$E \rightarrow \text{num} = \text{num}$

$S \rightarrow \text{begin } S L$

$\rightarrow \text{begin print } E L$

$\rightarrow \text{begin print } 1=1 L$

$\rightarrow \text{begin print } 1=1 \text{ end}$

$\therefore$  this string is in language( $S$ )

# Review: Parse Trees from Derivations

$S \rightarrow \text{if } E \text{ then } S \text{ else } S$

$S \rightarrow \text{begin } S \text{ L}$

$S \rightarrow \text{print } E$

$L \rightarrow \text{end}$

$L \rightarrow ; \text{ S L}$

$E \rightarrow \text{num} = \text{num}$

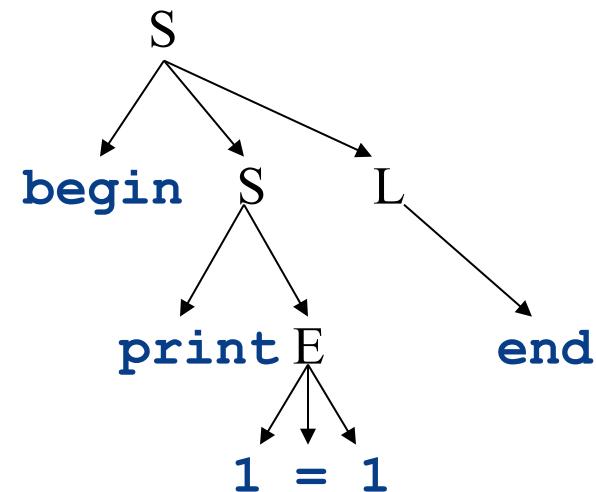
$S \rightarrow \text{begin } S \text{ L}$

$\rightarrow \text{begin print } E \text{ L}$

$\rightarrow \text{begin print } 1=1 \text{ L}$

$\rightarrow \text{begin print } 1=1 \text{ end}$

Parse Tree Associated with Derivation



# Shortcomings of LL Parsers

- \* Recursive descent renders a readable parser.
  - \* depends on the first terminal symbol of each sub-expression providing enough information to choose which production to use.
- \* But consider a rec. des. parser for this grammar

$$\begin{aligned} E &\rightarrow E + T \\ E &\rightarrow T \\ T &\rightarrow \text{id} \end{aligned}$$

```
void E(){switch(tok) {  
    case ?: E(); eat(TIMES); T(); ← no way of choosing production  
    case ?: T();  
    ...  
}  
void T(){eat(ID);}
```

# Example CFG

**Op**  $\rightarrow$  + | \* | - | /

**Exp**  $\rightarrow$  int | ( Exp Op Exp )

- \* Terminal symbols (a.k.a. "tokens")

  - \* + \* - /

- \* Non-terminal symbols

  - \* Op Exp

- \* Productions

  - \* Op  $\rightarrow$  +

  - \* Exp  $\rightarrow$  ( Exp Op Exp )

# Example Derivation

**Op**  $\rightarrow + \mid * \mid - \mid /$   
**Exp**  $\rightarrow \text{int} \mid ( \text{Exp} \text{ Op } \text{Exp} )$

each step  
applies one  
production

**Exp**  $\rightarrow ( \text{Exp} \text{ Op } \text{Exp} )$   
 $\rightarrow ( 1 \text{ Op } \text{Exp} )$   
 $\rightarrow ( 1 \text{ Op } 2 )$   
 $\rightarrow ( 1 * 2 )$

ends in string of  
terminal  
symbols only

Therefore,  $( 1 * 2 )$  is in the language of **Exp**

# Abstract Syntax in Haskell

**Op** → + | \* | - | /  
**Exp** → int | ( Exp Op Exp )

straightforward representation in Haskell

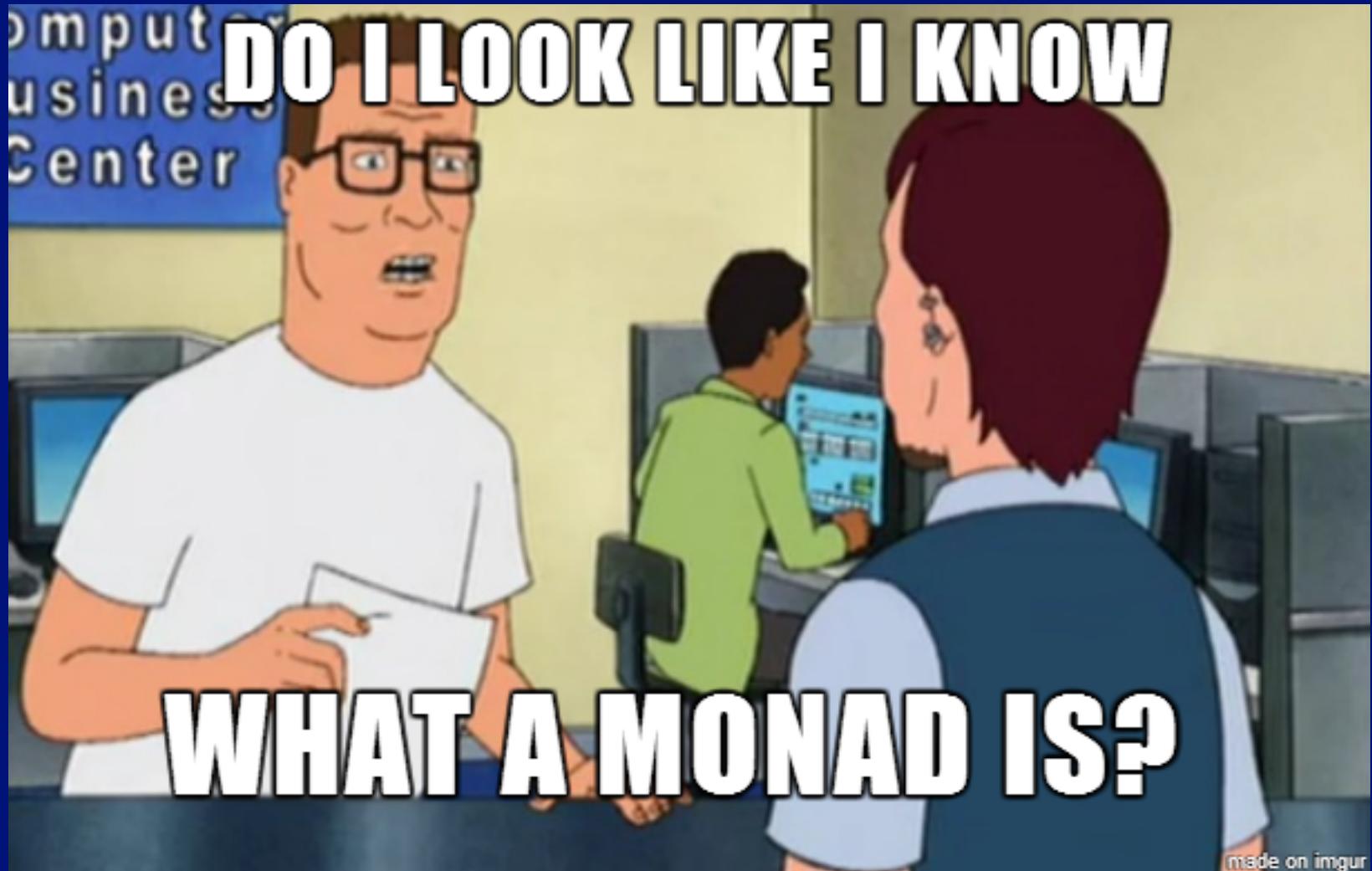
```
data Op = Plus | Times | Minus | Div deriving Show  
data Exp = Const Int | Aexp Op Exp deriving Show
```

Checkout the snapshot SimpleExpParser.zip off of the course page

# **Building Parsers using Monads**

# 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

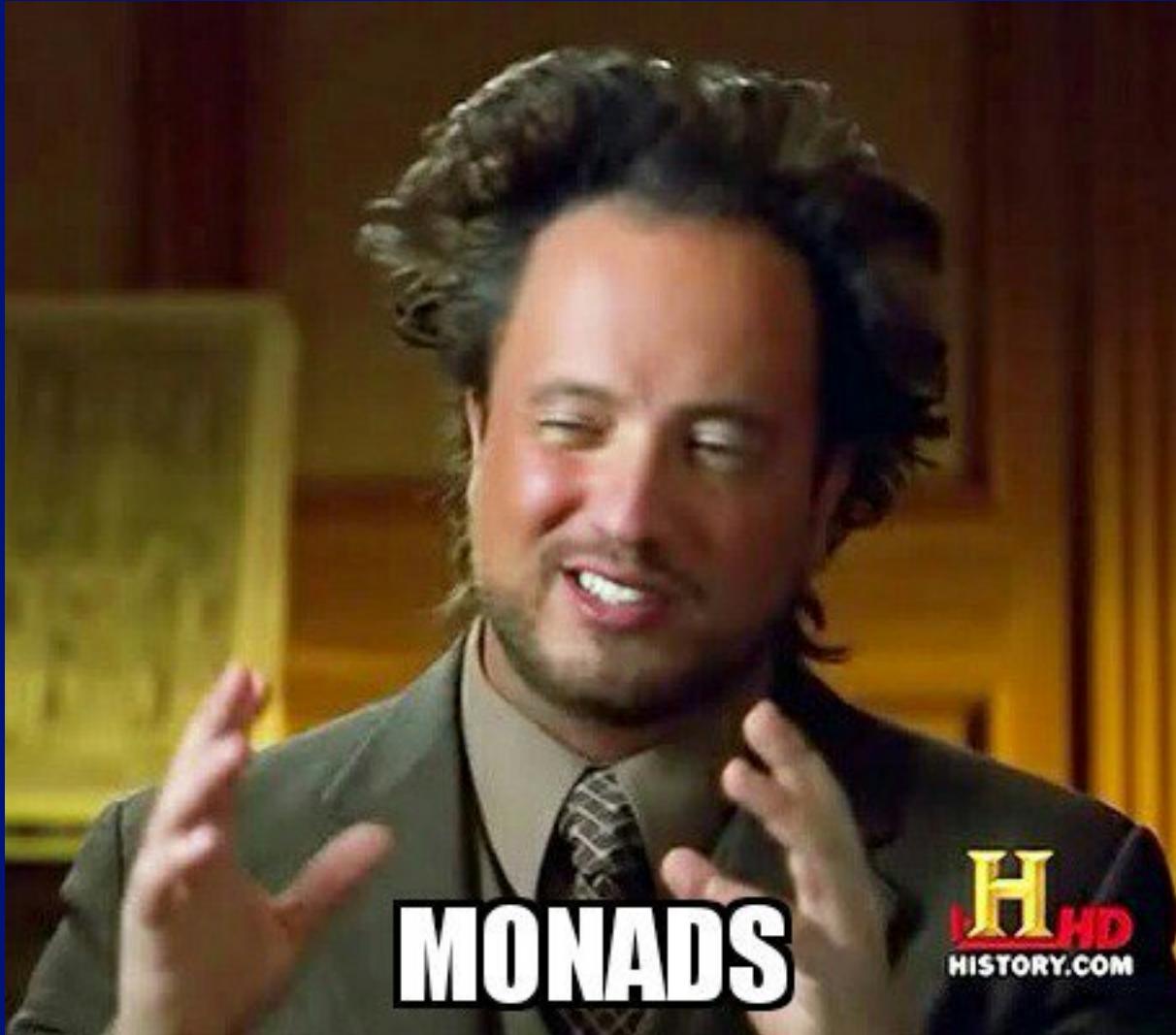


made on imgur

**A MONAD IS JUST A MONOID IN THE  
CATEGORY OF ENDOFUNCTORS**



**WHATS THE PROBLEM?**



**SAY MONAD**



**ONE MORE TIME**

[memegenerator.net](http://memegenerator.net)

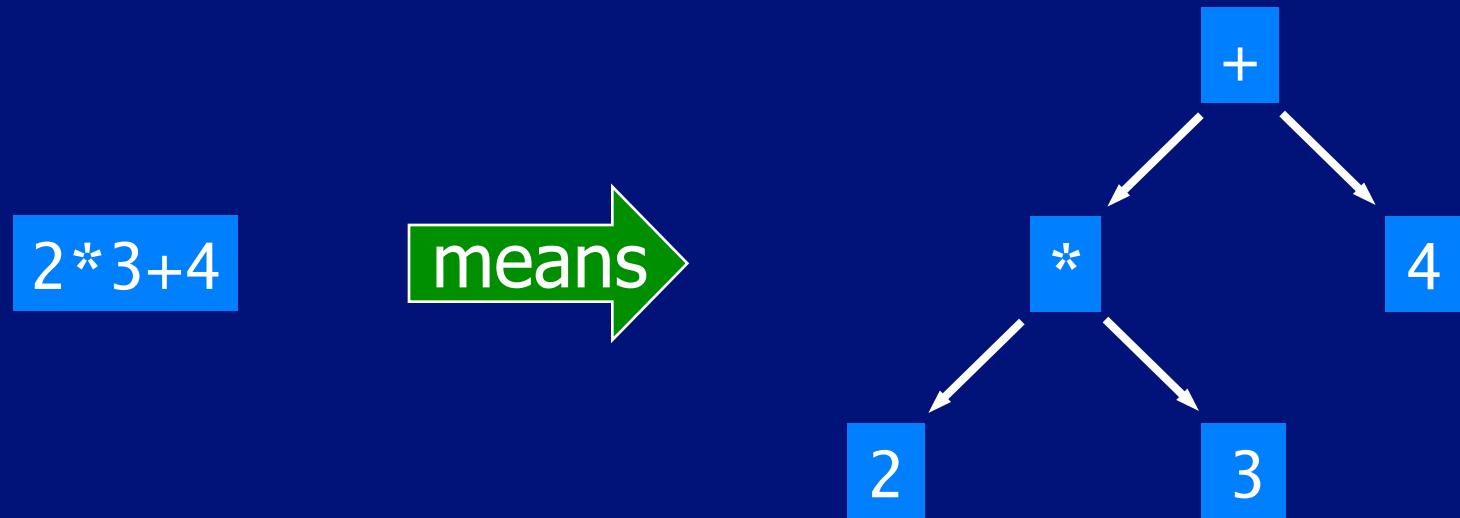
**HEY GIRL.**

**I WROTE YOU A MONAD SO THAT  
YOU CAN ALWAYS 'RETURN' TO  
ME.**

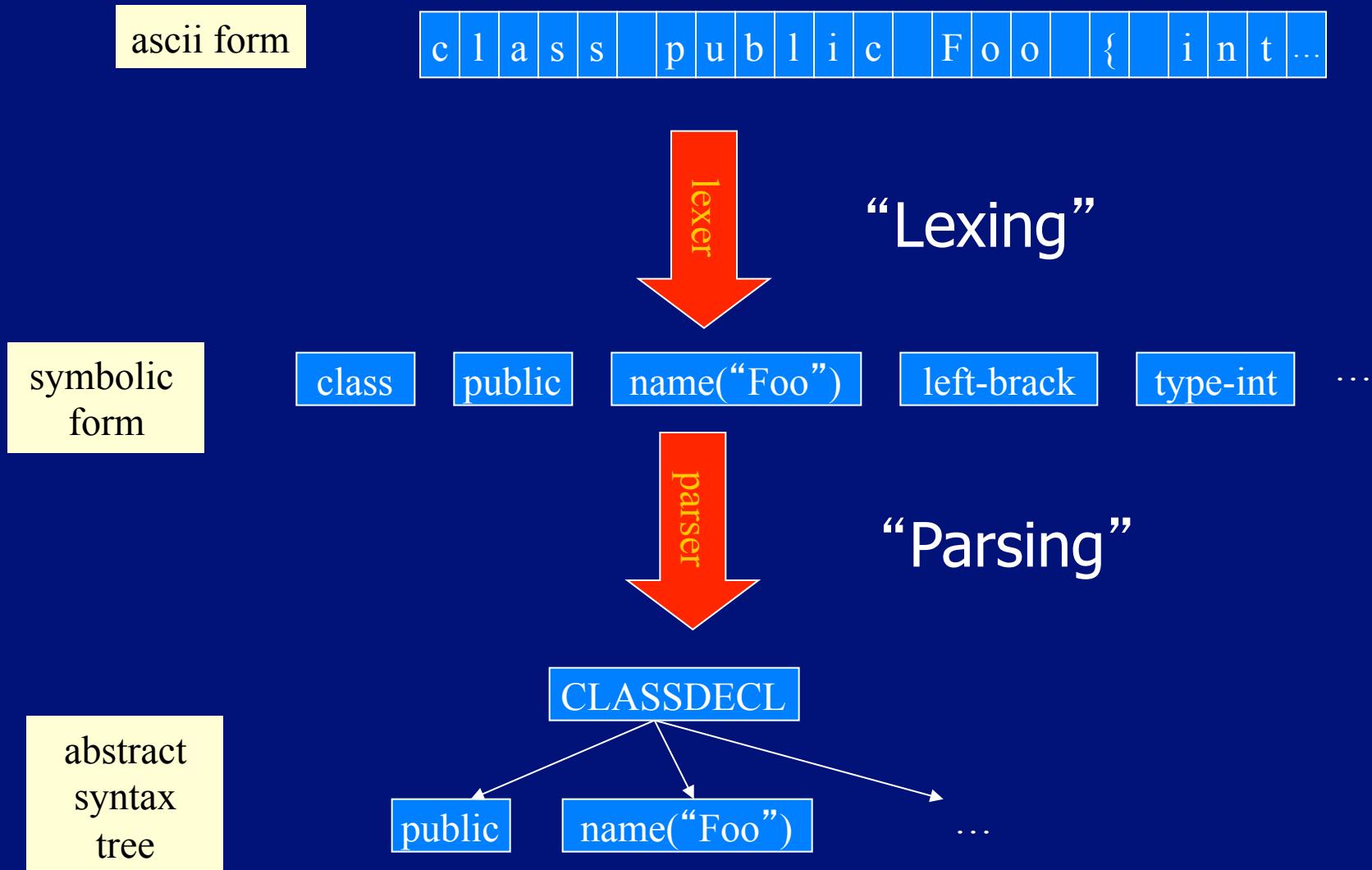
[memegenerator.net](http://memegenerator.net)

# 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 pre-process its input.

GHCi

Unix

Explorer

parses

Haskell programs

Shell scripts

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 any type:

```
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 singleton list.

# 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 failure always fails:

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

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

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

- The parser  $(p \text{ } +++ \text{ } q)$  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 errors 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 Aexprs and Expr

```
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, "")]
```

N.b., SimpleExpParser assumes infix expressions, unlike above

# Arithmetic Expressions

Consider a simple form of expressions 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 grammar:

$$\text{expr} \rightarrow \text{term} \text{ '+' } \text{expr} \mid \text{term}$$
$$\text{term} \rightarrow \text{factor} \text{ '*' } \text{term} \mid \text{factor}$$
$$\text{factor} \rightarrow \text{digit} \mid '(', \text{expr}, ')'$$
$$\text{digit} \rightarrow '0' \mid '1' \mid \dots \mid '9'$$

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

$$\textit{expr} \rightarrow \textit{term} \ ('+' \ \textit{expr} \mid \varepsilon)$$
$$\textit{term} \rightarrow \textit{factor} \ ('*' \ \textit{term} \mid \varepsilon)$$

Note:

- The symbol  $\varepsilon$  denotes the empty string.

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

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

That is, we have:

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

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

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

$$factor \rightarrow digit \mid '(' \ expr \ ')'$$

```
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:

$$expr \rightarrow term ('+' \ expr \mid '-' \ expr \mid \varepsilon)$$
$$term \rightarrow factor ('*' \ term \mid '/' \ term \mid \varepsilon)$$