



CS 115

Functional Programming

Lecture 21: May 23, 2016

Parser Combinators



Functional Programming: Spring 2016



Today

- The problem of parsing
- The usual way of parsing
 - `lex` and `yacc`
 - `alex` and `happy`
- Parsing as a monad
- Primitive parsers
- Parser combinators
- Case study: writing a Scheme parser





What is parsing?

- A *parser* is a program or part of a program that takes a stream of *tokens* (often just characters) and produces a stream of structured datatypes which represent what the stream of tokens means
- Usually thought of in the context of parsing a computer language
- Can also be used for many other tasks:
 - CSV (comma-separated values) files
 - domain-specific languages
 - configuration files





What is parsing?

- The theory of parsing has been well-understood for decades
 - although new ways of parsing are continually being developed, so it's not all old knowledge
- Parsers can be categorized by how they parse a given "language"
- Names like LR_1 , $LALR_1$, LL_k are often used to represent particular parsing strategies
- We won't be dealing with that in this lecture





What is parsing?

- Parsing has a reputation of being a necessary but boring and error-prone part of the process of writing a computer language
- Writing parsers by hand in most computer languages is so tedious that various extra-linguistic tools are generally used to make it easier
- These tools go by the generic names of **lex** and **yacc** programs





lex and yacc

- **lex** and **yacc** are two programs that are written in C and are found on most Unix-derived systems
- The open-source counterparts of **lex** and **yacc** are called **flex** and **bison** (ha ha)
- Most popular computer languages have some kind of **lex/yacc** clones written in that language (e.g. Ocaml has **ocamllex** and **ocamlyacc**)
- Details of the programs naturally differ between languages, but the overall parsing strategy is the same





lex and yacc

- **lex** and **yacc** are both *code generators*
- They read in files which represent high-level descriptions of the parser in special mini-languages written especially for parsing
- They output files of code (in C, or Ocaml, or whatever the language may be) which do the parsing
- Those files are compiled and linked into a program in the target language





lex and yacc

- **lex** is responsible for taking raw string input and outputting a stream of tokens
- Tokens are primitive syntactic elements including
 - punctuation (parentheses, commas, periods, etc.)
 - scalar literals (integers, floats, booleans)
 - strings
 - identifiers
- The **lex** language description file represents tokens by regular expressions which match the tokens
- **lex** compiles these into a very efficient "lexer"





lex and yacc

- **yacc** takes the stream of tokens output by **lex** and converts them into a datatype called an "abstract syntax tree" or AST
- An AST is a datatype which represents the language syntax in a form in which it can be manipulated programmatically
- The **yacc** language description file is written as a context-free grammar (CFG)
- Writing **yacc** grammars is often highly nontrivial!





lex and yacc

- Haskell, like many languages, has its own equivalents to **lex** and **yacc**
- The **lex**-equivalent is called **alex**
- The **yacc**-equivalent is called **happy**
- Used for parsing the Haskell language in the GHC compiler





lex and yacc

- Advantage of **lex** and **yacc**:
 - parsing is typically very efficient
- Disadvantages of **lex** and **yacc**:
 - writing a **yacc** CFG description is difficult and error-prone
 - code generation is a pain (hard to track down errors, for instance)
 - developing the parser is very cumbersome
 - no easy feedback cycle due to code generation





Parser combinators

- Parser combinators are a different approach to parsing
- Everything about it is different!
 - theory is different (LL_k parsers, not $LALR_1$)
 - no code generation required
 - often can totally skip the tokenizing (**lex**) step entirely!
 - much easier to develop and test
 - sometimes less efficient than **lex/yacc**-style parsers
- IMO definitely a superior solution for relatively simple grammars





Parser combinators

- The idea of parser combinators is to define simple parsers (e.g. parse a single character) and build up more complicated parsers from combinations of simple parsers
- Higher-order parsers (*parser combinators*) exist which take one or more parsers as input and produce a parser as output
- E.g. the **many** combinator takes a parser as input and returns a parser that parses the original parser input zero or more times (parsing a character → parsing a string of characters)





Parser combinators

- Parser combinators exist which can express the notion of
 - *alternation*: either parse with parser A, or if that fails, with parser B
 - *sequencing*: parse with parser A, then parser B
 - *backtracking*: parse with parser A, and if that doesn't work, rewind back to the beginning and parse with parser B
- They can also do decent error reporting





Parsing as a monad

- Parser combinators in Haskell are implemented using a special parsing monad called **Parsec**
 - actually a monad transformer called **ParsecT**, but specialized using the **Identity** monad to "plain parsing"
- This monad is quite complicated!
- Conceptually, though, it is doing something straightforward





Parsing as a monad

- A parsing monad is a kind of state monad
- The current input being parsed is the input state
- The input (minus the part successfully parsed) is the output state
- There is also extra state for file/line/character position and extra user-specifiable state
- There also has to be provision for parsing errors (like error-handling monads)





Parsing as a monad

- Most importantly, having a parsing monad means that details of threading the current parse state as well as file position, failure behavior *etc.* are done for us and need no user intervention
- We will not concentrate on the monadic aspects of parsing but will simply use the monad and ignore its internals
- What users need to understand is the different primitive parsers and the parser combinators that combine them





Parsing as a monad

- However, one fundamental kind of parser combinator is handled automatically by the parsing monad: *sequencing*

- An expression of the form

do *<parser1>*
 <parser2>

- represents a parser which will parse whatever *<parser1>* parses followed by whatever *<parser2>* parses
- Can also write this as *<parser1>* **>>** *<parser2>*





Parsing as a monad

- The parsing monad also allows us to grab the input that was successfully parsed using the `>>=` operator (or `<-` in the `do` notation)

- A parser of this form:

```
do p1 <- <parser1>
   p2 <- <parser2>
   return (f p1 p2)
```

- will parse input using `<parser1>`, putting the parse result in `p1`, then continue parsing with `<parser2>` and put that result into `p2`, then combine `p1` and `p2` with `f` and return that as the result





Parsing as a monad

- Naturally, this can also be written as:

```
<parser1> >>= \p1 ->  
  <parser2> >>= \p2 ->  
    return (f p1 p2)
```

- We will usually use `>>` and `>>=` for very short (one-line) parsers, and the `do` notation for longer/more complicated parsers





Text.Parsec

- In GHC, parser combinators are found in the **Text.Parsec** module and in several submodules (e.g. **Text.Parsec.Char**, **Text.Parsec.String** etc.)
- Many of these submodules are imported into **Text.Parsec** and re-exported, so there's usually no need to import them directly
- Sometimes need a function that is not exported from **Text.Parsec**, and then need to import that function's module directly





Datatypes

- Our principal parsing datatype will be **Parser a**
- This is a datatype that parses from a **String** (a list (stream) of characters) and returns a value of type **a** if successful
- If unsuccessful, it yields a **ParseError** which contains the source position (a **SourcePos**) and a list of error messages
- **Parser a** is actually a specialization of more general parsing datatypes





Datatypes

- The definition of **Parser a** is:

```
type Parser a = Parsec String () a
```

- It's defined in **Text.Parsec.String**, so we'll have to **import** that module too
- **Parsec s u a** is a general parsing datatype that
 - uses a "stream" type **s**
 - can manipulate user state **u**
 - returns values of type **a**
- **Parser a** uses **String** for the stream type and has no user state (a common case)





Datatypes

- The definition of `Parsec s u a` itself is:
`type Parsec s u a`
 `= ParsecT s u Identity a`
- `ParsecT s u m a` is like `Parsec`, but in an arbitrary monad `m`
- `ParsecT` is therefore a monad transformer (next lecture!)
- Useful if you want to e.g. print out information during the parsing process (use the `IO` monad for `m` instead of `Identity`)





Datatypes

- We can see that there are many levels of generality we can use for parsing
- For the purposes of this lecture, **Parser a** will be sufficient





Case study

- We will write a simple parser for a subset of the *Scheme* programming language
- The parser will read a file and convert all of the Scheme constructs into an abstract syntax tree (AST)
- We'll describe the parsers and combinators we need as we go along
- First we'll describe the AST itself





Scheme AST

- Scheme syntax is composed of *S-expressions*
- An S-expression is either
 - an *atom*
 - a *list* of S-expressions
 - a *quoted* S-expression
- An atom is either
 - a *literal* boolean, integer, or floating-point number
 - an *identifier* (name)





Scheme AST

- This is a very general syntax!
- Actual Scheme implementations take this AST representation and process it further into something more specific to the Scheme language
 - with specific constructors for lambda expressions etc.
- However, this representation is still very useful
- Scheme uses it because it makes macro processing (programmatic rewrite rules) very easy to implement





Scheme AST

- Here are the S-expression datatypes:

```
data Atom =
```

```
    BoolA Bool
  | IntA Integer
  | FloatA Double
  | IdA String -- identifier
deriving (Show)
```

```
data Sexpr =
```

```
    AtomS Atom
  | ListS [Sexpr]
  | QuoteS Sexpr -- quoted expression
deriving (Show)
```





Scheme AST

- Our goal is to take Scheme code like this:

```
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

- and convert it into values of the **Sexpr** datatype
- Let's start by writing parsers for the **Atom** datatype
- We have to be able to parse booleans, integers, floating-point numbers and identifiers





Booleans

- Our first parser will parse boolean values
- In Scheme, booleans are **#f** (false) and **#t** (true)
- Here is a simple (but incorrect) boolean parser:

```
parseBool :: Parser String
```

```
parseBool = (string "#f") <|> (string "#t")
```

- This has one parser (**string**) and one parser combinator (the **<|>** operator)
- We'll ignore the problems for now and explain what this is trying to accomplish





string

- The **string** parser parses a particular string (sequence of characters)
- So (**string** **"#f"**) parses only the string **"#f"** *i.e.* a Scheme boolean value
- **string** is not really a primitive parser (it can be defined in terms of more primitive parsers) but we'll ignore that here





- Our first parser combinator is the `<|>` operator
- It takes two parsers A and B, tries to parse the input using A, and if that fails, tries to parse it using B
- It therefore implements *alternation*
- Important point: if parser A fails after consuming input, `<|>` *will not backtrack* and parser B will probably fail too
- There is a way to force backtracking, but we'll see that later





parseTest

- It's worthwhile to test parsers while you're developing them
- To do this, use the `parseTest` function
- It has the type `Parser a -> String -> IO ()`
- It runs the parser on the input string, printing the results; so we can test `parseBool` like this:

```
ghci> parseTest parseBool "#f"
```

```
"#f"
```

- So far so good...





parseBool

- Let's try some other tests...

```
ghci> parseTest parseBool "#foo"  
"#f"
```

- This works, because the desired value was found at the beginning of the input

```
ghci> parseTest parseBool "xxx#f"  
parse error at (line 1, column 1):  
unexpected "x"  
expecting "#f" or "#t"
```

- This fails, because the parsed value *wasn't* found at the beginning of the input





parseBool

```
ghci> parseTest parseBool "#t"  
parse error at (line 1, column 1):  
unexpected "t"  
expecting "#f"
```

- What's going on here?
- Why didn't the `<|>` operator try (`string "#t"`) after the parser (`string "#f"`) failed?





parseBool

```
ghci> parseTest parseBool "#t"
```

- Parsec parsers don't automatically backtrack upon failure!
- If a parser fails *without consuming any input*, then `<|>` will work as desired
- But `(string "#f")` failed only *after* successfully consuming the `'#'` character, then failing to consume the `'f'` character because it encountered `'t'` instead
- It didn't backtrack back to the beginning and try `(string "#t")` next





try

- Backtracking is very expensive, and parsers that automatically backtrack as far as needed are not going to be efficient
- Parsec forces you to tell it exactly where it should backtrack using the **try** combinator
- **try** *<parser>* is the same as *<parser>* except that if *<parser>* fails, it backtracks right back to the first character that *<parser>* read
- This will enable us to fix our parser here





parseBool (2nd try)

```
parseBool :: Parser String  
parseBool = (try (string "#f"))  
           <|> (string "#t")
```

- This will work:

```
ghci> parseTest parseBool "#f"  
"#f"
```

```
ghci> parseTest parseBool "#t"  
"#t"
```

- Woo hoo!





parseBool (2nd try)

- We can clean this up by dropping some unnecessary parentheses:

```
parseBool :: Parser String
```

```
parseBool = try (string "#f")  
            <|> string "#t"
```

- There is still one big problem with this definition – can you see it?
- It doesn't parse booleans!
- It just parses strings that *represent* booleans in Scheme





parseBool (3rd try)

- To fix this we have to change the type and **return** the correct values:

```
parseBool :: Parser Bool
```

```
parseBool = try (string "#f" >> return False)  
            <|> (string "#t" >> return True)
```

- The **string** parser returns the parsed string
- Here, we throw it away and return the datatype we want (a **Bool** value, either **False** or **True**)
- Note the monadic **>>** operator used here!





parseBool (3rd try)

- Of course, this could also be written like this:

```
parseBool :: Parser Bool
```

```
parseBool =
```

```
    try (do string "#f"  
            return False)
```

```
  <|> (do string "#t"  
          return True)
```

- The **do** expressions are equivalent to using the **>>** operator





Aside

- What does this code actually mean?

```
do string "#f"  
  return False
```

- This sub-parser has the type **Parser Bool**, just like the larger parser of which it's a part
- A parser of type **Parser Bool** takes in a string, possibly parses part of the string, yielding the remaining string (like the state of a state monad) and a boolean value (like the value part of a state monad)





Aside

- What does this code actually mean?

```
do string "#f"  
  return False
```

- We are actually combining two simple parsers to make a more complicated parser:
 - `string "#f"` parses the string `"#f"` and returns the parsed string if successful
 - `return False` is a trivial parser that does nothing and returns the `False` value always (like `return` used in a state monad)
- The combination gives us the kind of result we want





parseBool (3rd try)

- Let's try it out:

```
ghci> parseTest parseBool "#f"
```

```
False
```

```
ghci> parseTest parseBool "#t"
```

```
True
```

```
ghci> parseTest parseBool "#u"
```

```
parse error at (line 1, column 1):
```

```
unexpected "u"
```

```
expecting "#t"
```





parseBool (3rd try)

```
ghci> parseTest parseBool "#u"
```

```
parse error at (line 1, column 1):
```

```
unexpected "u"
```

```
expecting "#t"
```

- This is OK, but there is an easy way to give this parser more informative error messages
- It will involve the `<?>` combinator
- This combinator specifies part of the error message to use when a parser fails without consuming any input





parseBool (4th try)

```
parseBool :: Parser Bool
```

```
parseBool =
```

```
  try (string "#f" >> return False)
```

```
  <|> try (string "#t" >> return True)
```

```
  <?> "boolean"
```

```
ghci> parseTest parseBool "#u"
```

```
parse error at (line 1, column 1):
```

```
unexpected "u"
```

```
expecting boolean
```





parseBool (4th try)

```
parseBool :: Parser Bool
```

```
parseBool =
```

```
  try (string "#f" >> return False)
```

```
<|> try (string "#t" >> return True)
```

```
<?> "boolean"
```

- We get better error messages now
- We added a **try** to the second case to make sure that when that parser failed, it didn't consume any input





parseBool (4th try)

```
parseBool :: Parser Bool
```

```
parseBool =
```

```
    try (string "#f" >> return False)
```

```
  <|> (string "#t" >> return True)
```

```
  <?> "boolean"
```

- We can also leave out the **try** in the second case
- Check out the error messages in two cases:

```
ghci> parseTest parseBool "foo"
```

```
ghci> parseTest parseBool "#u"
```





parseBool (4th try)

```
ghci> parseTest parseBool "foo"
```

```
parse error at (line 1, column 1):
```

```
unexpected "f"
```

```
expecting boolean
```

```
ghci> parseTest parseBool "#u"
```

```
parse error at (line 1, column 1):
```

```
unexpected "u"
```

```
expecting "#t"
```

- First error message OK, second not so great
- Let's rewrite **parseBool** one last time





parseBool (5th try)

- We note that `#f` and `#t` both start with a `#` character
- Why not factor that out into the first part of the parser? Then wouldn't need all this `try` business and backtracking
- Here you are:

```
parseBool :: Parser Bool
```

```
parseBool =
```

```
  char '#' >>  -- first parse the # character
```

```
    ((char 'f' >> return False)
```

```
    <|> (char 't' >> return True))
```

```
<?> "boolean"
```





parseBool (5th try)

```
parseBool :: Parser Bool
```

```
parseBool =
```

```
  char '#' >>
```

```
    ((char 'f' >> return False)
```

```
    <|> (char 't' >> return True))
```

```
<?> "boolean"
```

- *N.B.* The **char** parser parses a specific character
- This is a more efficient way to write this parser, since backtracking is costly
- But will it produce decent error messages?





parseBool (5th try)

```
ghci> parseTest parseBool "foo"
```

```
parse error at (line 1, column 1):
```

```
unexpected "f"
```

```
expecting boolean
```

- So far, so good...

```
ghci> parseTest parseBool "#u"
```

```
parse error at (line 1, column 2):
```

```
unexpected "u"
```

```
expecting "f" or "t"
```

- Nice! This is the error message we'd expect to have here!





parseInt

- **parseBool** was quite painful to write, but now that we understand how it works, we have most of the tools we need to write the rest of the parser!
- We'll need some new parsers and combinators, but the difficult material is behind us
- Let's now turn to parsing integers
- We'll write a parser called **parseInt** with type **Parser Integer**





parseInt (version 1)

- We'll ignore the sign of the integer for now
- We have:

```
parseInt :: Parser Integer
```

```
parseInt = do
```

```
  digits <- many1 digit
```

```
  return (read digits :: Integer)
```

```
<?> "integer"
```

- Note the use of monadic **do**-notation here!
- Note also the **digit** parser, the **many1** parser combinator, and the **read** function





parseInt (version 1)

- The **digit** parser parses any digit from 0 to 9
- The **many1** parser combinator takes a parser as its argument and returns a parser that parses the original parser 1 or more times
 - So (**many1 digit**) parses one or more **digits**
- The **read** function is the inverse of the **show** function
 - it tries to convert a string into any datatype that is an instance of the **Read** type class (which **Integers** are)
 - it's a very simple parser all by itself (like C's **atoi**)





parseInt (version 1)

- Let's try it out:

```
ghci> parseTest parseInt "1001"  
1001
```

```
ghci> parseTest parseInt "-1001"  
parse error at (line 1, column 1):  
unexpected "-"  
expecting integer
```

- Still need to handle an *optional* initial **-** sign
- We'll need a new combinator: **option**





parseInt (version 2)

```
parseInt :: Parser Integer
```

```
parseInt = do
```

```
    sign <- option "" (string "-")
```

```
    digits <- many1 digit
```

```
    return (read (sign ++ digits) :: Integer)
```

```
<?> "integer"
```

- The **option** combinator will try to parse a minus sign, returning the string `"-"` if successful and the empty string if it fails
- This will enable us to parse positive and negative integers





parseInt (version 2)

- Let's try it out:

```
ghci> parseTest parseInt "1001"
```

```
1001
```

```
ghci> parseTest parseInt "-1001"
```

```
-1001
```

- Error cases also give reasonable error messages
- Let's move on to floating point numbers





parseFloat

- With what we've already done, we can easily define a parser for floating-point numbers (not including explicit exponential notation):

```
parseFloat :: Parser Double
parseFloat = do
    sign <- option "" (string "-")
    digits <- many1 digit
    char '.' -- decimal point
    f <- many1 digit -- at least 1 digit after dec pt
    return (read (sign ++ digits ++ "." ++ f) :: Double)
<?> "floating-point number"
```

- You should be able to figure out why this works





parseId

- We'll write a parser called **parseId** that will parse identifiers (names)
- Identifiers can contain
 - letters or numbers
 - various symbolic characters (at least "**_+-* /=?!**")
- We'll use the **alphaNum** parser that parses any letter from A-Z or a-z or any digit
- We'll also use the **oneOf** combinator that parses any character in a given string





parseId

- Here's the definition:

```
parseId :: Parser String
```

```
parseId =
```

```
  many1 (alphaNum <|> oneOf "_+-*/=?!")
```

```
  <?> "identifier"
```

- There's nothing particularly strange here
- Now that we can parse any of the atom values, we can write an atom parser called **parseAtom**





parseAtom

- **parseAtom** is defined like this:

```
parseAtom :: Parser Atom
```

```
parseAtom =
```

```
    (parseBool >>= return . BoolA)
```

```
<|> try (parseFloat >>= return . FloatA)
```

```
<|> try (parseInt >>= return . IntA)
```

```
<|> (parseId >>= return . IdA)
```

```
<?> "atom"
```

- We have to take the results of the individual parsers and convert them to the **Atom** datatype
- We have to use **try** to implement backtracking where we need it (the order matters!)





Note

- Consider this code fragment:
`(parseBool >>= return . BoolA)`
- You could also write this as follows:
`(do b <- parseBool
 return (BoolA b))`
- Much easier to understand, but less concise





parseAtom

- Let's try it out:

```
ghci> parseTest parseAtom "-1001"  
IntA (-1001)
```

```
ghci> parseTest parseAtom "#f"  
BoolA False
```

```
ghci> parseTest parseAtom "-3.14"  
FloatA (-3.14)
```

```
ghci> parseTest parseAtom "foobar"  
IdA "foobar"
```

- Nice!





Expressions

- Now that we can parse atoms, we need to be able to parse entire expressions
- An expressions can be
 - an atom (done)
 - a parenthesized list of expressions, separated by whitespace and/or comments
 - a quoted expression
- Before we do this, let's parse comments and whitespace





parseComment

- In Scheme, comments start with the semicolon (;) character and go to the end of a line
- A parser for this will discard the parsed string, since it's just a comment
- This parser is easy to write:

```
parseComment :: Parser ()  
parseComment = do  
    char ';'   
    many (noneOf "\n")  
    char '\n'  
    return ()
```





parseComment

```
parseComment :: Parser ()
```

```
parseComment = do
```

```
  char ';' 
```

```
  many (noneOf "\n")
```

```
  char '\n'
```

```
  return ()
```

- This uses two new combinators: **many** and **noneOf**
- **noneOf** parses a character which *isn't* in the supplied string
- **many** parses zero or more of what the argument parser parses





parseWhitespace

- Whitespace is trivial to parse:

```
parseWhitespace :: Parser ()
```

```
parseWhitespace = many1 space >> return ()
```

- This uses the `space` parser which parses any whitespace character
- Since we have to separate expressions in a list with whitespace (and/or comments), we will define a parser for "separators"





parseSep

- This parser will be called **parseSep**:

```
parseSep :: Parser ()
```

```
parseSep =
```

```
  many1 (parseComment <|> parseWhitespace) >> return ()
```

```
  <?> "separator"
```

- Note that this will discard any nonempty sequence of comments and whitespace
- Now that this is done, we can get down to the important task of parsing lists of expressions





parseList

- A (syntactic) list in Scheme is a sequence of expressions inside of parentheses, separated by separators (whitespace or comments)
- The initial and final expression may also be separated from the open/close parentheses by separators, but don't have to be
- We will need to use a couple of new combinators to define our parser





parseList

- Here's the parser:

```
parseList :: Parser [Sexpr]
parseList = do
  char '('
  optional parseSep
  ss <- parseSexpr `sepEndBy` parseSep
  char ')'
  return ss
<?> "list of S-expressions"
```





parseList

```
parseList :: Parser [Sexpr]
parseList = do
  char '('
  optional parseSep
  ss <- parseSexpr `sepEndBy` parseSep
  char ')'
  return ss
<?> "list of S-expressions"
```

- The **optional** parser combinator parses something if it's present and discards it, otherwise does nothing
- We use this to handle whitespace/comments right after the open parenthesis





parseList

```
parseList = do
  char '('
  optional parseSep
  ss <- parseSexpr `sepEndBy` parseSep
  char ')'
  return ss
<?> "list of S-expressions"
```

- The **sepEndBy** parser combinator parses a sequence of what the first parser parses as long as each item in the sequence is separated by (and optionally ended by) what the second parser parses





parseQuote

- Quoted expressions are very simple: they consist of the single quote (') followed by a Scheme expression
- Examples:
 - 'foo ' (a b c) ' ((a 1) (b 2) (c 3))
- The parser is quite simple too:

```
parseQuote :: Parser Sexpr
```

```
parseQuote = char '\'' >> parseSexpr  
          <?> "quoted S-expression"
```





parseSexpr

- Now we have all the parsers we need to write the main S-expression parser
- It will call out to the previously-defined parsers, and when one of them succeeds it will wrap one of the **Sexpr** constructors around the result and return that
- The definition is trivial:

```
parseSexpr :: Parser Sexpr
```

```
parseSexpr =
```

```
  (parseAtom >>= return . AtomS)
```

```
  <|> (parseList >>= return . ListS)
```

```
  <|> (parseQuote >>= return . QuoteS)
```

```
  <?> "S-expression"
```





parseSexpr

```
parseSexpr :: Parser Sexpr
```

```
parseSexpr =
```

```
  (parseAtom >>= return . AtomS)
```

```
  <|> (parseList >>= return . ListS)
```

```
  <|> (parseQuote >>= return . QuoteS)
```

```
  <?> "S-expression"
```

- Question: why don't we have to use **try** anywhere in this parser?
- To parse an entire file's worth of S-expressions, we need one more parser...





parseSexprsFromFile

- We will need the **eof** parser, which matches the end of the input (here, end-of-file or EOF) and returns nothing
- Definition:

```
parseSexprsFromFile :: Parser [Sexpr]
parseSexprsFromFile = do
  optional parseSep
  ss <- parseSexpr `sepEndBy` parseSep
  eof
  return ss
<?> "file of S-expressions"
```





parseSexprsFromFile

```
parseSexprsFromFile :: Parser [Sexpr]
parseSexprsFromFile = do
  optional parseSep
  ss <- parseSexpr `sepEndBy` parseSep
  eof
  return ss
<?> "file of S-expressions"
```

- Note that this parser requires that top-level S-expressions be separated from each other using a separator (e.g. a newline), which is a good idea anyway





parseFromFile

- To run this parser, we will use the `parseFromFile` function defined in `Text.Parsec.String`
- It has this signature:

```
parseFromFile :: Parser a -> String  
              -> IO (Either ParseError a)
```
- The first argument will be the parser (`parseSexprsFromFile`), the second is the file path (the Scheme code's filename) and the return value is the parsed results or an error
- Let's test the parser! (demo)





Summary

- Writing a parser using parser combinators is much more straightforward than using a **lex/yacc**-style toolchain
- Parser combinators thus are heavily used in real-world applications of Haskell
- There are many features in the **Text.Parsec** modules that I haven't covered here, so Hoogle is your friend!





Next time

- More on the **IO** monad
- Monad transformers

