Exploring Languages with Interpreters and Functional Programming
Chapter 44

H. Conrad Cunningham

24 November 2018

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall
P.O. Box 1848
University, MS 38677
(662) 915-5358

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of November 2018 is a recent version of Firefox from Mozilla.

44 Calculator: Parsing

44.1 Chapter Introduction

The *ELI Calculator language* case study examines how we can represent and process simple arithmetic expressions using Haskell.

In Chapter 41, we described two different concrete syntaxes for expressions written as text. In Chapter 42, we defined the abstract syntax represented as an algebraic data type and the language semantics with an evaluation function.

Chapter 40 introduced the general concepts of lexical analysis and parsing. In this chapter, we design and implement a hand-coded lexical analyzer and two hand-coded recursive descent parsers for the two concrete syntaxes given in Chapter 41. The parsers also construct the corresponding abstract syntax trees.

44.2 Parsing

A programming language processor uses a parser to determine whether a program satisfies the grammar for the language's concrete syntax. The parser typically constructs some kind of internal representation of the program to enable further processing.

A common approach to parsing is to divide it into at least two phases:

- A lexical analyzer converts the sequence of characters into a sequence of low-level syntactic units called *tokens*. The grammar describing the tokens is usually a *regular grammar*, which can be processed efficiently using a *finite state machine*.
- A parser converts the sequence of tokens into an initial *semantic model* (e.g. into an *abstract syntax tree* supported by a *symbol table*). The grammar describing the language's full syntax is typically a *context-free grammar*, which requires more complex mechanisms to process.,

If the language has aspects that cannot be described with a context-free grammar, then additional phases may be needed to handle issues such as checking types of variables and expressions and ensuring that variables are declared before they are used.

Of course, regular grammars are context-free grammars, so a separate lexical analyzer is not required. But use of a separate lexical analyzer often leads to a simpler parser and better performance.

However, some approaches to parsing, such as the use of parser combinators, can conveniently handle lexical issues as a part of the parser.

In this chapter, we use the two-stage approach to parsing of the ELI Calculator language. We define a lexical analyzer and parsers constructed using a technique

called recursive descent parsing. The parsers construct abstract syntax trees using the algebraic data type defined in Chapter 42 (i.e. in the Abstract Syntax module).

Chapter 45 generalizes the recursive descent parsers to a set of parsing combinatiors.

44.3 Lexical Analysis

In computing science, *lexical analysis* [Wikipedia 2018a] is typically the process of reading a sequence of characters from a language text and assembling the characters into a sequence of *lexemes*, the smallest meaningful syntactic units. In a natural language like English, the lexemes are typically the words of the language.

The output of lexical analysis is a sequence of *lexical tokens* (usually just called *tokens*). A token associates a syntactic category with a lexeme. In a natural language, the syntactic category may be the word's part of speech (noun, verb, etc.).

We call the program that carries out the lexical analysis a *lexical analyzer*, *lexer*, *tokenizer*, or *scanner*. (However, the latter term actually refers to one phase of the overall process.)

In a programming language, the syntactic categories of tokens consist of entities such as identifiers, integer literals, and operators.

The "whitespace" characters such as blanks, tabs, and newlines are usually not tokens themselves. Instead, they are delimiters which define the boundaries of the other lexemes. However, in some programming languages, the end of a line or the indentation at the beginning of a line have implicit structural meaning in the language.

Consider the ELI Calculator language infix syntax. The character sequence

```
30   +   ( x1 * 2)
```

includes seven tokens:

- integer literal `30`
- addition operator `+`
- left parenthesis symbol `(`
- identifier `x1`
- multiplication operator `*`
- integer literal `2`
- right parenthesis symbol `)`

Tokenization has two stages—a scanner and an evaluator.

A *scanner* processes the character sequence and breaks it into lexeme strings. It usually recognizes a language corresponding to a *regular grammar*, one of the simplest classes of grammars, and is, hence, based on a *finite state machine*. However, in some cases, a scanner may require more complex grammars and processors.

A token *evaluator* determines the syntactic category of the lexeme string and tags the token with this syntactic information.

Sometimes a lexical analyzer program combines the two stages into the same algorithm.

44.3.1 Prefix syntax

Now let's consider a lexical analyzer for the prefix syntax for the ELI Calculator language.

File `LexCalc.hs` gives an example Haskell module that implements a lexical analyzer for this concrete syntax.

The ELI Calculator language's prefix syntax includes the following syntactic categories: identifiers, keywords, integer literals, operators, left parenthesis, and right parenthesis.

The *left* and *right parenthesis* characters are the only lexemes in those two syntactic categories, respectively.

An *identifier* is the name for variable or other entity. We define an identifier to begin with an alphabetic or underscore character and include all contiguous alphabetic, numeric, or underscore characters that follow. It is delimited by a whitespace or another character not allowed in an identifier.

As a sequence of characters, a *keyword* is just an identifier in this language, so the scanner does not distinguish between two categories. The lexical analyzer subsequently separates out keywords by checking each identifier against the list of keywords.

An *integer literal* begins with a numeric character and includes all contiguous numeric characters that follow. It is delimited by a whitespace or nonnumeric character.

We plan to extend this language with additional operators. To enable flexible use of the scanner, we design it to collect all contiguous characters from a list of supported operator characters. Of course, we exclude alphabetic, numeric, underscore, parentheses, and similar characters from the list for the prefix ELI Calculator language.

The lexer subsequently compares each scanned operator against a list of valid operators to remove invalid operators.

The language uses keywords in similar ways to operators, so the lexer also subsequently tags keywords as operators. The current lexical analyzer does not use the `TokKey` token category.

The `LexCalc` module defines a `Token` algebraic data type, defined below, to represent the lexical tokens. The constructors identify the various syntactic categories.

```
import Values ( NumType, Name, toNumType )
-- e.g. NumType = Int , Name = String

data Token = TokLeft           -- left parenthesis
           | TokRight          -- right parenthesis
           | TokNum NumType    -- unsigned integer literal
           | TokId Name        -- names of variables, etc.
           | TokOp Name        -- names of primitive functions
           | TokKey Name       -- keywords (no use currently)
           | TokOther String   -- other characters
             deriving (Show, Eq)
```

The function `lexx`, shown below, incorporates the scanner and most of the lexeme evaluator functionality. It takes a string and returns a list of tokens.

```
import Data.Char ( isSpace, isDigit, isAlpha, isAlphaNum )

lexx :: String -> [Token]
lexx []  = []
lexx xs@(x:xs')
    | isSpace x   = lexx xs'
    | x == ';'    = lexx (dropWhile (/='\n') xs')
    | x == '('    = TokLeft  : lexx xs'
    | x == ')'    = TokRight : lexx xs'
    | isDigit x   = let (num,rest) = span isDigit xs
                    in (TokNum (convertNumType num)) : lexx rest
    | isFirstId x = let (id,rest) = span isRestId xs
                    in  (TokId id) : lexx rest
    | isOpChar x  = let (op,rest) = span isOpChar xs
                    in (TokOp op) : lexx rest
    | otherwise   = (TokOther [x]) : lexx xs'
    where
        isFirstId c = isAlpha c    || c == '_'
        isRestId  c = isAlphaNum c || c == '_'
        isOpChar  c = elem c opchars

opchars = "+-*/~<=>!&|@#$%^?:"  -- not " ' ` ( ) [ ] { } , . ;
```

Function `lexx` pattern matches on the first character of the string and then collects any additional characters of the token using the higher order function

`Data.Char.span`. Function `span` breaks the string into two part—the prefix consisting of all contiguous characters that satisfy its predicate and the suffix beginning with the first character that does not.

Boolean function `isOpChar` returns `True` for characters potentially allowed in operator symbols. These are defined in the string `opchars`{.haskell{, which makes this aspect of the scanner relatively easy to modify.

Function `lexer`, shown below, calls `lexx` and then carries out the following transformations on the list of tokens:

- `TokId` tokens for keywords are transformed into the corresponding `TokOp` tokens (as defined in association list `keywords {.haskell}`)
- `TokOp` tokens for valid operators (as defined in association list `opmap`) are transformed if needed and invalid operators are transformed into `TokOther` tokens

The lexer does not generate error messages. Instead it tags characters that do not fit in any lexeme as a `TokOther` token. The parser can use these as needed (e.g. to generate error messages).

```
lexer :: String -> [Token]
lexer xs = markSpecials (lexx xs)

markSpecials :: [Token] -> [Token]
markSpecials ts = map xformTok ts

xformTok :: Token -> Token
xformTok t@(TokId id)
    | elem id keywords   = TokOp id
    | otherwise          = t
xformTok t@(TokOp op)
    | elem op primitives = t
    | otherwise          = TokOther op
xformTok t               = t

keywords   = []               -- none defined currently
primitives = ["+","-","*","/"]
```

In the above code, the function `xformTok` transforms any identifier that is a defined keyword into an operator token, leaves other identifiers and defined primitive operators alone, and marks everything else with the token type `TokOther`.

44.3.2 Infix syntax

The lexer for the prefix syntax given in the previous subsection can also be used for the simple infix syntax. However, future extensions of the language may

require differences in the lexers.

44.4 Recursive Descent Parsing

A *recursive descent parser* is an approach to parsing languages that have relatively simple grammars [Fowler 2011].

It is a *top-down parser*, a type of parser that begins with start symbol of the grammar and seeks to determine the parse tree by working down the levels of the parse tree toward the program (i.e. sentence).

By contrast, a *bottom-up* parser first recognizes the low-level syntactic units of the grammar and builds the parse tree upward from these leaves toward the root (i.e. start symbol). Bottom-up parsers support a wider range of grammars and tend to be more efficient for production compilers. However, their development tends to be less intuitive and more complex. We leave discussion of these parsers to courses on compiler construction.

A recursive descent parser consists of a set of mutually recursive functions. It typically includes one hand-coded function for each nonterminal of the grammar and one clause for each production for that nonterminal.

The recursive descent approach works well when the grammar can be transformed into an LL(k) (especially LL(1)) grammar. Discussion of these techniques are left to courses on compiler construction.

For an LL(1) grammar, we can write recursive descent parsers that can avoid backtracking to an earlier point in the parse to start down another path.

For example, consider a simple grammar with with rules:

```
S ::= A | B
A ::= C D
B ::= { E }  -- zero or more occurrence of E
C ::= [ F ]  -- zero or one occurrence of F
D ::= '1' | '@' S
E ::= '3'
F ::= '2'
```

Consider the nonterminal `S`, which has alternatives `A` and `B`.

- Alternative `A` can begin with terminal symbols `1`, `2`, or `@`.
- Alternaive `B` can begin with terminal symbol `3` or be empty.

These sets of first symbols are disjoint, so the parser can distinguish among the alternatives based on the first terminal symbol. (Hence, the grammar is backtrack-free.)

44.4.1 Constructing recursive descent parsers

A simple recognizer for the grammar above could include functions similar to those shown below. We consider the five different situations for nonterminals `S`, `A`, `B`, `C`, and `E`.

In the Haskell code, a parsing function takes a `String` with the text of the expression to be processed and returns a tuple `(Bool,String)` where the first component indicates whether or not the parser succeeded (i.e. the output of the parse) and the second component gives the new state of the input.

If the first component is `True`, then the second component holds the input remaining after the parse. If the first component is `False`, then the second component is the remaining part of the input to be processed after the parser failed.

Of course, instead of strings, the parser could work on lists of tokens or other symbols.

1. Alternatives: `S ::= A | B`

```
parseS :: String -> (Bool,String)  -- A | B
parseS xs =
    case parseA xs of              -- try A
        (True,  ys) -> (True, ys)  -- A succeeds
        (False, _ ) ->
            case parseB xs of      -- else try B
                (True, ys) -> (True,  ys) -- B succeeds
                (False, _) -> (False, xs) -- both A & B fail
```

   Function `parseS` succeeds whenever any alternative succeeds. Otherwise, it continues to check subsequent alternatives. It fails if the final alternative fails.

   If there are more than two alternatives, we can nest each additional alternative more deeply within the conditional structure. (That is, we replace the `parseB` failure case value with a `case` expression for the third option. Etc.)

2. Sequencing: `A ::= C D`

```
parseA :: String -> (Bool,String)  -- C D
parseA xs =
    case parseC xs of              -- try C
        (True,  ys) ->
            case parseD ys of      -- then try D
                (True, zs) -> (True,  zs) -- C D succeeds
                (False, _) -> (False, xs) -- D fails
        (False, _ ) -> (False,xs)  -- C fails
```

Function `parseA` fails whenever any component fails. Otherwise, it continues to check subsequent components. It succeeds when the final component succeeds.

If there are more than two components in sequence, we nest each additional component more deeply within the conditional structure. (That is, we replace `parseD xs` with `case parseD xs of ...`.)

3. Repetition zero or more times: `B ::= { E }`

```
parseB :: String -> (Bool,String)  -- { E }
parseB xs =
    case parseE xs of              -- try E
        (True,  ys) -> parseB ys   -- one E, try again
        (False, _)  -> (True,xs)   -- stop, succeeds
```

Function `parseB` always succeeds if `parseE` terminates. However, it may succeed for zero occurrences of `E` or for some positive number of occurrences.

4. Optional elements: `C ::= [ F ]`

```
parseC :: String -> (Bool,String)  -- [ F ]
parseC xs =
    case parseF xs of              -- try F
        (True,  ys) -> (True,ys)
        (False, _ ) -> (True,xs)
```

Function `parseC` always succeeds if `parseF` terminates. However, it may succeed for at most one occurrence of `F`.

5. Base cases to parse low-level syntactic elements: `E ::= '3'`

```
parseE :: String -> (Bool,String)
parseE (x:xs') = (x == '3', xs')
parseE xs      = (False,    xs )
```

On success in any of these cases, the new input state is the string remaining after the successful alternative.

On failure, the input state should be left unchanged by any of the functions.

To use the above templates, it may sometimes be necessary to refactor the rules that involve more than one of the above cases. For example, consider the rule

```
D ::= '1' | '@' S
```

which consists of two alternatives, the second of which is itself a sequence. To see how to apply the templates straightforwardly, we can refactor `D` to be the two rules:

```
D  ::= '1' | DS
DS ::= '@' S
```

In addition to the above parsers for the various rules, we might have a function `parse` that calls the top-level parser (`parseS`) and ensures that all the input is parsed.

```
parse :: String -> Bool
parse xs =
    case parseS xs of
        (True,  []) -> True
        (_,     _ ) -> False
```

See file `ParserS03.hs` for experimental Haskell code for this example recursive descent parser.

To have a useful parser, the above prototype functions likely need to be modified to build the intermediate representation and to return appropriate error messages for unsuccessful parses.

The above prototype functions use Haskell, but a similar technique can be used with any language that supports recursive function calls.

44.4.2 Prefix syntax

This subsection describes an example recursive descent parser for the ELI Calculator language's prefix syntax. The complete code for the `ParsePrefixCalc` module is given in the file ParsePrefixCalc.hs.

As given in Chapter 41, the prefix parser embodies the the following grammar:

```
<expression> ::=  <var> | <val> | <operexpr>
<var>        ::=  <id>
<val>        ::=  [ '-' ] <unsigned>
<operexpr>   ::=  '(' <operator> <operandseq> ')'
<operandseq> ::=  { <expression> }
<operator>   ::=  '+'  |  '*'  |  '-'  |  '/' | ...
```

The `ParserPrefixCalc` module imports and uses the `LexCalc` module) for lexical analysis. In particular, it uses the algebraic data type `Token`, types `NumType` and `Name`, and function `lexer`.

```
import Values ( NumType, Name, toNumType )

data Token = TokLeft          -- left parenthesis
           | TokRight         -- right parenthesis
           | TokNum NumType   -- unsigned integer literal
           | TokId Name       -- names of variables, etc.
           | TokOp Name       -- names of primitive functions
           | TokKey Name      -- keywords
           | TokOther String  -- other characters
             deriving (Show, Eq)
```

```
lexer :: String -> [Token]
```

For the prefix grammar above, the nonterminals `<id>` and `<unsigned>` and the terminals are parsed into their corresponding tokens by the lexical analyzer.

TODO: Update this code and reference. The incomplete module `TestPrefix06` (in file TestPrefix06.hs) provides some testing of the prefix parser.

The output of the parser is an abstract syntax tree constructed with the algebraic data type `Expr` defined in the previous chapter. This is in the Abstract Syntax module.

```
import Values ( ValType, Name )

data Expr = Add Expr Expr
          | Sub Expr Expr
          | Mul Expr Expr
          | Div Expr Expr
          | Var Name
          | Val ValType
```

44.4.2.1 Parse `<expression>`

Now let's build a recursive descent parser using the method described in the previous subsection. We begin with the start symbol `<expression>`.

The parsing function `parseExpression`, shown below, implements the following BNF rule:

```
<expression> ::= <var> | <val> | <operexpr>
```

It uses the recursive descent template #1 with three alternatives.

```
type ParErr = String

parseExpression :: [Token] -> (Either ParErr Expr, [Token])
parseExpression xs =
    case parseVar xs of
        r@(Right _, _) -> r  -- <var>
        _ ->
          case parseVal xs of
              r@(Right _, _) -> r  -- <val>
              _ ->
                  case parseOperExpr xs of
                      r@(Right _, _) -> r  -- <operexpr>
                      (Left m, ts)  -> (missingExpr m ts, ts)

missingExpr m ts  =
```

```
        Left ("Missing expression at  " ++ (showTokens (pref ts))
               ++ "..\n..Nested error { " ++ m ++ " }")
```

Function `parseExpression` takes a `Token` list and attempts to parse an `<expression>`. If the parse succeeds, the function returns a pair consisting of the `Right` value of an `Either` wrapping the corresponding `Expr` abstract syntax tree and the list of input `Token`s remaining after the `Expr`. If the parse fails, then the function returns an error in a `Left` value for the`Either` and the unchanged list of input `Token`s.

We define an auxiliary function `missingExpr` to generate an appropriate error message.

The function `parse`, shown below, is the primary entry point for the `ParsePrefixCalc` module. It first calls the lexical analysis function `lexer` (from the module `LexCalc`) on the input list of characters and then calls the parsing function `parseExpression` with the corresponding list of tokens.

If a parsing error occurs or if there are leftover tokens, then the function returns an appropriate error message.

```
parse :: String -> Either ParErr Expr
parse xs =
    case lexer xs of
        [] -> incompleteExpr xs
        ts ->
            case parseExpression ts of
               (ex@(Right _), []) -> ex
               (ex@(Left  _), _ ) -> ex
               (ex, ss)           -> extraAtEnd ex ss

incompleteExpr xs =
    Left ("Incomplete expression: " ++ xs)

extraAtEnd ex xs =
   Left ("Nonspace token(s) \"" ++ (showTokens xs) ++
         "\" at end of the expression \"" ++ (show ex) ++ "\"")
```

44.4.2.2 Parse `<var>`

Function `parseVar` implements the BNF rule:

```
<var> ::= <id>
```

Variable `<id>` denotes an identifier token recognized by the lexer. So we implement function `parseVar` as a base case of the recursive descent parser (i.e. template #5).

```
parseVar :: [Token] -> (Either ParErr Expr, [Token])
parseVar ((TokId id):ts) = (Right (Var id),ts)
parseVar ts              = (missingVar ts, ts)

missingVar ts =
    Left ("Missing variable at " ++ (showTokens (pref ts)))
```

Function `parseVar` has the same type signature as `parseExpression`. It attempts to match an identifier token at the front of the token sequence. If it finds an identifier, it transforms the token to a `Var` expression and returns it with the remaining token list. Otherwise, it returns an error message and the unchanged token list.

44.4.2.3 Parse <val>

Function `parseVal` implements the BNF rule:

```
<val> ::= [ '-' ] <unsigned>
```

To implement this rule, we can refactor it into two rules that correspond to the recursive descent template functions:

```
<val>      ::= <optminus> <unsigned>
<optminus> ::= [ '-' ]
```

Then `<val>` can be implemented using the sequencing (#2) prototype, `<optminus>` using the optional element (#4) prototype, and `<unsigned>` and `-` using base case (#5) prototypes.

However, `<unsigned>` denotes a numeric token and `-` denotes a single operator token. Thus we can easily implement `parseVal` as a base case of the recursive descent parser.

```
parseVal :: [Token] -> (Either ParErr Expr, [Token])
parseVal ((TokNum i):ts)             = (Right (Val i), ts)
parseVal ((TokOp "-"):(TokNum i):ts) = (Right (Val (-i)), ts)
parseVal ts                          = (missingVal ts, ts)

missingVal ts =
    Left ("Missing value at " ++ (showTokens (pref ts)))
```

Function `parseVal` has the same type signature as `parseExpression`. It attempts to match a numeric token, which is optionally preceded by a negative sign, at the front of the token sequence. If it finds this, it transforms the tokens to a `Val` expression and returns the expression and the remaining token list. Otherwise, it returns an error message and the unchanged token list.

44.4.2.4 Parse <operexpr>

Function `parseOperExpr` implements following BNF rule:

```
<operexpr>   ::=  "(" <operator> <operandseq> ")"
```

It uses a modified version of recursive descent template #2 for sequences of terms.

```
parseOperExpr :: [Token] -> (Either ErrMsg Expr, [Token])
parseOperExpr xs@(TokLeft:(TokOp op):ys) =  -- ( <operator>
    case parseOperandSeq ys of              -- <operandseq>
        (args, zs) ->
            case zs of                      -- )
              (TokRight:zs') -> (makeExpr op args, zs')
              zs'            -> (missingRParen zs, xs)
-- ill-formed <operexpr>s
parseOperExpr (TokLeft:ts)      = (missingOp ts, ts)
parseOperExpr (TokRight:ts)     = (invalidOpExpr ")", ts)
parseOperExpr ((TokOther s):ts) = (invalidOpExpr s, ts)
parseOperExpr ((TokOp op):ts)   = (invalidOpExpr op, ts)
parseOperExpr ((TokId s):ts)    = (invalidOpExpr s, ts)
parseOperExpr ((TokNum i):ts)   = (invalidOpExpr (show i), ts)
parseOperExpr []                = (incompleteExpr, [])

missingRParen ts =
    Left ("Missing `)` at " ++ (show (take 3 ts)))
missingOp ts     =
    Left ("Missing operator at " ++ (show (take 3 ts)))
invalidOpExpr s  =
    Left ("Invalid operation expression beginning with " ++ s)
incompleteExpr   = Left "Incomplete expression"
```

Function `parseOperExpr` has the same type signature as `parseExpression`. It directly matches against the first two tokens to see whether they are a left parenthesis and an operator, respectively, rather than calling separate functions to parse each. If successful, it then parses zero or more operands and examines the last token to see whether it is a right parenthesis.

If the operator expression is ill-formed, the function returns an appropriate error message.

The function `parseOperExpr` delegates the construction of the corresponding `Expr` (i.e. abstract syntax tree) to function `makeExpr`, which we discuss later in the subsection.

The values yielded by the components of `<operexpr>` must be handled differently than the previous components of expressions we have examined. They are not themselves `Expr` values.

- `(` and `)` denote the structure of the expression but do not have any output.

- `<operator>` does not itself yield a complete `Expr`. It must be combined with some number of operands to yield an expression. The number varies depending upon the particular operator. We pass a string to `makeExpr` to denote the operator.
- `<operandseq>` yields a possibly empty list of `Expr` values. We pass an `Expr` list to `makeExpr` to denote the operands.

44.4.2.5 Parse `<operandseq>`

Function `parseOperandSeq` implements the BNF rule:

```
<operandseq> ::=  { <expression> }
```

It uses the recursive descent template #3 for repeated symbols.

```
parseOperandSeq :: [Token] -> ([Expr],[Token])
parseOperandSeq xs =
    case parseExpression xs of
        (Left  _,      _ ) -> ([],xs)
        (Right ex, ys) ->
            let (exs,zs) = parseOperandSeq ys
            in  (ex:exs,zs)
```

The function `parseOperandSeq` takes a token list and collects a list of 0 or more operand `Expr`s. An empty list means that no operands were found.

44.4.2.6 AST construction (`makeExpr`)

Operators in the current abstract syntax take a fixed number of operands. `Add` and `Mul` each take two operands, but a negation operator would take one operand and a conditional "if" operation would take three.

However, the current concrete prefix syntax does not distinguish among the different operators and the number of operands they require. It allows any operator in an `<operexpr>` to have any finite number of operands.

We could, of course, define a grammar that distinguishes among the operators, but we choose to keep the grammar flexible, thus enabling easy extension. We handle the operator-operand matching in the `makeExpr` function using data structures to define the mapping.

Thus, function `makeExpr` takes the operator string and a list of operand `Expr`s and constructs an appropriate `Expr`. It uses function `arity` to determine the number of operands required for the operator and then calls the appropriate `opConsN` function to construct the `Expr`.

```
makeExpr :: String -> [Expr] -> Either ErrMsg Expr
makeExpr op exs =
    case arity op of
```

```
        0 -> opCons0 op exs  -- not implemented
        1 -> opCons1 op exs
        2 -> opCons2 op exs
        3 -> opCons3 op exs
        4 -> opCons4 op exs  -- not implemented
        5 -> opCons5 op exs  -- not implemented
        _ -> opConsX op exs  -- not implemented
```

Function `arity` takes an operator symbol and returns the number of operands that operator requires. It uses the `arityMap` association list to map the operator symbols to the number of arguments expected.

```
import Data.Maybe

arityMap = [ ("+",2), ("-",2), ("*",2), ("/",2) ]
           -- add (operator,arity) pairs as needed
arity :: String -> Int
arity op = fromMaybe (-1) (lookup op arityMap)
```

Function `opCons2` takes a binary operator string and an operand list with two elements and returns the corresponding `Expr` structure wrapped in a `Right`. An error is denoted by passing back an error message wrapped in a `Left`.

```
assocOpCons2 =
    [ ("+",Add), ("-",Sub), ("*",Mul), ("/",Div) ]
     -- add new pairs as needed

opCons2 :: String -> [Expr] -> Either ParErr Expr
opCons2 op exs =
    case length exs of
        2 -> case lookup op assocOpCons2 of
                Just c  -> Right (c (exs!!0) (exs!!1))
                Nothing -> invalidOp op
        n -> arityErr op n

invalidOp op =
    Left ("Invalid operator '" ++ op ++ "'")
arityErr op n =
    Left ("Operator '" ++ op ++ "' incorrectly called with "
          ++ (show n) ++ " operand(s)")
```

Currently, the only supported operators are the binary operators `+`, `-`, `*`, and `/`. These map to the binary `Expr` constructors `Add`, `Sub`,`Mul`, and `Div`. (These are two-argument functions.)

If we extend the supported operators, then we must extend the definitions of `arityMap` and `assocOpCons2` and add new definitions for `opConsN` and `assocOpConsN` for other arities `N`. (We may also need to modify the `LexCalc` module and the definition of `Expr`.)

For now, we respond to unknown operators using function `opConsX` and return an appropriate error message. (In the future, this function may be redefined to support operators with variable numbers of operands.)

```
opConsX :: String -> [Expr] -> Either ErrMsg Expr
opConsX op exs = unsupportedOp op

unsupportedOp op = Left ("Unsupported operator '" ++ op ++ "'")
```

44.4.3 Infix syntax (TODO)

TODO: Update the parser to reflect the grammar change and recursive descent explanation.

TODO: Describe the recursive descent infix parser in module ParseInfixCalc.hs. An incomplete module that does some testing is TestInfix03.hs.

44.5 Exercises

TODO

44.6 Acknowledgements

I initially developed this case study for the Haskell-based offering of CSci 556 (Multiparadigm Programming) in Spring 2017. I continued work on it during Summer and Fall 2017. I based this work, in part, on ideas from:

- the Scala-based Expression Tree Calculator case study from my *Notes on Scala for Java Programmers* [Cunningham 2018]: (which was itself adapted from the tutorial Scala for Java Programmers [Schinz 2017] in Spring 2016)
- the Lua-based Expression Language 1 and Imperative Core interpreters I developed for the Fall 2016 CSci 450 course
- Samuel N. Kamin's textbook *Programming Languages: An Interpreter-based Approach*, Addison Wesley, 1990 [Kamin 1990] and my work to implement three (Core, Lisp, and Scheme) of these interpeters in Lua in 2013
- the Wikipedia articles on Regular Grammar, Context-Free Grammar, Backus-Naur Form, Lexical Analysis, Parsing, LL Parser, Recursive Descent Parser, and Abstract Syntax [Wikipedia 2018, 2018b].
- Chapter 21 (Recursive Descent Parser) of the Martin Fowler and Rebecca Parsons's book *Domain-Specific Languages*, Addison Wesley, 2011 [Fowler 2011].

- Section 3.2 (Predictive Parsing) of Andrew W. Appel's textbook *Modern Compiler Implementation in ML*, Cambridge, 1998 [Appel 1998].
- Chapters 6 (Purely Functional State) and 9 (Parser Combinators) from Paul Chiusano and Runar Bjarnason's *Functional Programming in Scala*, Manning, 2015 [Chiusano 2015].

In Fall 2018, I divided the 2017 Expression Language Parsing chapter into two chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming.* Sections 11.1-11.4 became Chapter 44, Calculator Parsing (this chapter), and sections 11.6-11.7 became Chapter 45, Parsing Combinators. I merged Section 11.5 into Chapter 43, a new chapter on the modular structure of the ELI Calculator language interpreter.

I maintain these notes as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the notes to HTML, PDF, and other forms as needed. The HTML version of this document may require use of a browser that supports the display of MathML.

44.7 References

[Appel 1998]: Andrew W. Appel. *Modern Compiler Implementation in ML*, Cambridge, 1998. (Especially section 3.2 "Predictive Parsing")

[Chiusano 2015]: Paul Chiusano and Runar Bjarnason, *Functional Programming in Scala*, Manning, 2015. (Especially chapters 6 "Purely Functional State" and 9 "Parser Combinators")

[Cunningham 2018]: H. Conrad Cunningham. *Notes on Scala for Java Programmers*, 2018 (which is itself adapted from the tutorial [Schinz 2018] Scala for Java Programmers

[Fowler 2011]: Martin Fowler and Rebecca Parsons. *Domain-Specific Languages*, Addison Wesley, 2011. (Especially chapter 21 "Recursive Descent Parser")

[Kamin 1990] Samuel N. Kamin. *Programming Languages: An Interpreter-Based Approach*, Addison-Wesley, 1990.

[Linz 2017] Peter Linz. *An Introduction to Formal Languages and Automata*, Fifth Edition, Jones and Bartlett, 2017. (Especially sections 1.2, 3.3, and 5.1)

[Schinz 2017]: Michel Schinz and Philipp Haller. A Scala Tutorial for Java Programmers, accessed February 2016.

[Sestoft 2012]: Peter Sestoft. *Programming Language Concepts*, Springer, 2012. (Especially sections 1.3, 2.5, and 2.7 and chapter 8)

[Wikipedia 2018a]: Wikipedia. Articles on Formal Grammar, Regular Grammar, Context-Free Grammar, Backus-Naur Form, Extended Backus-Naur Form, and Parsing. Accessed 9 August 2018.

[Wikipedia 2018b]: Wikipedia. Articles on Abstract Syntax and Associative Array, Accessed 9 August 2018.

44.8 Terms and Concepts

TODO