**CFG Parsing:**

- A parser for a context-free grammar can mostly look like the grammar rules. There are however a few things to watch out for, some tricks, and that lingering issue of initial leading spaces.
- The parsers here will produce abstract syntax trees of this type:

```
data Expr
   = Num Integer
   | Var String
   | Prim2 Op2 Expr Expr       -- Prim2 op operand operand
   | Let [(String, Expr)] Expr -- Let [(name, rhs), ...] body

data Op2 = Add | Mul
```

**Right-associating Operator:**

- Take this simple rule, and suppose we intend the operator to associate to the right:
  **muls ::= natural { "*" natural }** OR **muls ::= natural [ "*" muls ]**.
  The second form uses right recursion to convey right association.
  This is perfect for recursive descent parsing.

```
mulsRv1 :: Parser Expr
mulsRv1 = liftA2 link
        (fmap Num natural)
        (optional (liftA2 (,)
                (operator "*" *> pure (Prim2 Mul))
                mulsRv1))
 where
   link x Nothing = x
   link x (Just (op,y)) = op x y
```

**Note:**

```
*ParserLib> (,) True 'x'
(True,'x')
*ParserLib> (,) True False
(True,False)
```

```
Prelude Control.Applicative> liftA2 (,) [1..3] [1..4]
[(1,1),(1,2),(1,3),(1,4),(2,1),(2,2),(2,3),(2,4),(3,1),(3,2),(3,3),(3,4)]
```

We have the line **fmap Num natural** because we want the return type to be something in Expr. If we get an integer from natural, we want to add the Num tag to it.
E.g.

```
*ParserLib> unParser (fmap Num natural) "33"
Just ("",Num 33)
*ParserLib> runParser (fmap Num natural) "33"
Just (Num 33)
```

We have the line **(operator "*" *> pure (Prim2 Mul))** because if we see a "*", we ignore it and use pure(Prim2 Mul) to represent it.

Lastly, we have the line **mulsRv1** because we want to make a recursive call.

E.g. of running mulsRv1:

```
*ParserLib> runParser mulsRv1 "2*5"
Just (Prim2 Mul (Num 2) (Num 5))
*ParserLib> runParser mulsRv1 "2*5*7"
Just (Prim2 Mul (Num 2) (Prim2 Mul (Num 5) (Num 7)))
*ParserLib> runParser mulsRv1 "2*5*7*9"
Just (Prim2 Mul (Num 2) (Prim2 Mul (Num 5) (Prim2 Mul (Num 7) (Num 9))))
*ParserLib> runParser mulsRv1 "2"
Just (Num 2)
```

- Instead of writing this recursion by hand again for every right-associative operator, we can call a re-factored function and specify just your operand parser and operator parser.

Here is the re-factored general function for right-associative operators.

```
chainr1 :: Parser a                 -- ^ operand parser
        -> Parser (a -> a -> a)     -- ^ operator parser
        -> Parser a                 -- ^ whole answer
chainr1 getArg getOp = liftA2 link getArg
                           (optional
                               (liftA2 (,) getOp (chainr1 getArg getOp)))
 where
   link x Nothing = x
   link x (Just (op,y)) = op x y
```

So here is how we will implement the rule in practice:

```
mulsRv2 :: Parser Expr
mulsRv2 = chainr1 (fmap Num natural) (operator "*" *> pure (Prim2 Mul))
```

E.g.

```
Prelude ParserLib> runParser (chainr1 (fmap Var (identifier [])) (operator "*" *> pure (Prim2 Mul))) "x*y*z"
Just (Prim2 Mul (Var "x") (Prim2 Mul (Var "y") (Var "z")))
```

**Left-associating operator:**

- Suppose we want the operator to associate to the left instead. We cannot code up left recursion directly, but the trick is to implement the other form of the rule.

  Still imagine that the grammar rule is of this form: muls ::= natural { "*" natural }.
  Use many for the "{ "*" natural }" part to get a list of tuples of (operator, number).

  For example if the input string is "2 * 5 * 3 * 7", my plan is to:
  1. read "2" and get Num 2
  2. read "* 5 * 3 * 7" with the help of many and get [(Prim2 Mul, Num 5), (Prim2 Mul, Num 3), (Prim2 Mul, Num 7)]
  3. Then using foldl on the list, starting with Num 2 as the initial accumulator, will build the left-leaning tree

  I.e. The parser still does right-associating recursion, but we will use foldl on the return value to make it left-associating.

  Here's the code:

```
mulsLv1 :: Parser Expr
mulsLv1 = liftA2 link
        (fmap Num natural)
        (many (liftA2 (,)
              (operator "*" *> pure (Prim2 Mul))
              (fmap Num natural)))

  where
    link x opys = foldl (\accum (op,y) -> op accum y) x opys
```

  **fmap Num natural** gets us "Num 2".

```
(many (liftA2 (,)
              (operator "*" *> pure (Prim2 Mul))
              (fmap Num natural)))
```

  gets us [(Prim2 Mul, Num 5), (Prim2 Mul, Num 3), (Prim2 Mul, Num 7)]

  **link x opys = foldl (\accum (op,y) -> op accum y) x opys** combines "Num 2" with [(Prim2 Mul, Num 5), (Prim2 Mul, Num 3), (Prim2 Mul, Num 7)].
  The argument, x, is "Num 2."
  The argument, opys, is [(Prim2 Mul, Num 5), (Prim2 Mul, Num 3), (Prim2 Mul, Num 7)].
  In **foldl (\accum (op,y) -> op accum y) x opys**, accum is "Num 2", op is "Prim2 Mul" and y is "Num _". It's taking the value of **fmap Num natural** and putting "Prim2 Mul" over it and the first "Num _" in the list.

  **Note:** The recursive call is in "many".

E.g.

```
*ParserLib> runParser mulsLv1 "2"
Just (Num 2)
*ParserLib> runParser mulsLv1 "2*5"
Just (Prim2 Mul (Num 2) (Num 5))
*ParserLib> runParser mulsLv1 "2*5*7"
Just (Prim2 Mul (Prim2 Mul (Num 2) (Num 5)) (Num 7))
*ParserLib> runParser mulsLv1 "2*5*7*9"
Just (Prim2 Mul (Prim2 Mul (Prim2 Mul (Num 2) (Num 5)) (Num 7)) (Num 9))
```

- Again in practice we don't write this code again, we re-factor this into a general function:

```
chainl1 :: Parser a                 -- ^ operand parser
        -> Parser (a -> a -> a)     -- ^ operator parser
        -> Parser a                 -- ^ whole answer
chainl1 getArg getOp = liftA2 link
                getArg
                (many (liftA2 (,) getOp getArg))
 where
  link x opys = foldl (\accum (op,y) -> op accum y) x opys
```

Then we use it like:

```
mulsLv2 :: Parser Expr
mulsLv2 = chainl1 (fmap Num natural) (operator "*" *> pure (Prim2 Mul))
```

E.g.

```
Prelude ParserLib> runParser (chainl1 (fmap Var (identifier [])) (operator "*" *> pure (Prim2 Mul))) "x*y*z"
Just (Prim2 Mul (Prim2 Mul (Var "x") (Var "y")) (Var "z"))
```

**Comparing between mulsLv1 and mulsRv1:**

- E.g.

```
Prelude ParserLib> runParser mulsRv1 "2"
Just (Num 2)
Prelude ParserLib> runParser mulsLv1 "2"
Just (Num 2)
```

```
Prelude ParserLib> runParser mulsRv1 "2*5"
Just (Prim2 Mul (Num 2) (Num 5))
Prelude ParserLib> runParser mulsLv1 "2*5"
Just (Prim2 Mul (Num 2) (Num 5))
```

```
Prelude ParserLib> runParser mulsRv1 "2*5*7"
Just (Prim2 Mul (Num 2) (Prim2 Mul (Num 5) (Num 7)))
Prelude ParserLib> runParser mulsLv1 "2*5*7"
Just (Prim2 Mul (Prim2 Mul (Num 2) (Num 5)) (Num 7))
```

```
Prelude ParserLib> runParser mulsRv1 "2*5*7*9"
Just (Prim2 Mul (Num 2) (Prim2 Mul (Num 5) (Prim2 Mul (Num 7) (Num 9))))
Prelude ParserLib> runParser mulsLv1 "2*5*7*9"
Just (Prim2 Mul (Prim2 Mul (Prim2 Mul (Num 2) (Num 5)) (Num 7)) (Num 9))
```

**Initial space, final junk:**

- Token-level parsers assume no leading spaces.
  Notice how if we have spaces in front, natural doesn't work.

```
Prelude ParserLib> unParser natural "      45"
Nothing
Prelude ParserLib> unParser natural "5      "
Just ("",5)
```

  This is because natural is expecting to read numbers and it's reading spaces instead.
- Another problem is that a small parser for a part of the grammar can leave non-space stuff unconsumed, since we anticipate that later a small parser for another part may need it. But the overall combined parser for the whole grammar cannot leave any non-space stuff unconsumed. By the time you're done with the whole grammar, any non-space leftover means the original input string is actually erroneous.
  E.g. We don't consider "2*3*" to be a legal arithmetic expression because our muls parsers can make sense of the prefix "2*3" but leaves the last "*" unconsumed.

```
Prelude ParserLib> unParser mulsLv1 "2*3*"
Just ("*",Prim2 Mul (Num 2) (Num 3))
```

- The trick for solving both is to have a “main” parser whose job is simply to clear initial leading spaces, call the parser for the start symbol, then use eof to check that there is nothing left.

  Here’s the code:

```
lesson2 :: Parser Expr
lesson2 = whitespaces *> muls <* eof
  where
    muls = chainl1 (fmap Num natural) (operator "*" *> pure (Prim2 Mul))
```

```
Prelude ParserLib> unParser lesson2 "2*3*"
Nothing
Prelude ParserLib> unParser lesson2 "2*3"
Just ("",Prim2 Mul (Num 2) (Num 3))
Prelude ParserLib> unParser lesson2 "2*3      "
Just ("",Prim2 Mul (Num 2) (Num 3))
Prelude ParserLib> unParser lesson2 "      2*3"
Just ("",Prim2 Mul (Num 2) (Num 3))
Prelude ParserLib> unParser lesson2 "      2*3      "
Just ("",Prim2 Mul (Num 2) (Num 3))
```

- **Note:** We do *> and <* outside of the parser because if we do it inside, there may be recursive calls to it in the middle of your grammar which may give back the wrong result.

**Operator precedence and parentheses:**

- Suppose I have two operators “*” and “+”, with “+” having lower precedence, and I also support parentheses for overriding precedence.
- In other words, from lowest precedence (binding most loosely) to highest (binding most tightly) is “+”, then “*”, then individual numbers and parentheses (same level without ambiguity).
- The trick is to have lower (looser) rules call higher (tighter) rules, and have the parentheses rule call the lowest rule for recursion. The start symbol is from the lowest rule. This is also how you can write your grammar to convey precedence.

- E.g.

  So my grammar goes like (start symbol is adds):

```
adds ::= muls { "+" muls }
muls ::= atom { "*" atom }
atom ::= natural | "(" adds ")"
```

  And my parser goes like (let's say left-associating operators):

```
lesson3 :: Parser Expr
lesson3 = whitespaces *> adds <* eof
 where
   adds = chainl1 muls (operator "+" *> pure (Prim2 Add))
   muls = chainl1 atom (operator "*" *> pure (Prim2 Mul))
   atom = fmap Num natural <|> (openParen *> adds <* closeParen)
```

  E.g.

```
Prelude ParserLib> unParser lesson3 "2+3*4"
Just ("",Prim2 Add (Num 2) (Prim2 Mul (Num 3) (Num 4)))
Prelude ParserLib> unParser lesson3 "2+3*4+5"
Just ("",Prim2 Add (Prim2 Add (Num 2) (Prim2 Mul (Num 3) (Num 4))) (Num 5))
```

**Keywords and variables:**

- Here is the whole grammar and the start symbol is expr:

```
expr ::= local | adds
local ::= "let" { var "=" expr ";" } "in" expr
adds ::= muls { "+" muls }
muls ::= atom { "*" atom }
atom ::= natural | var | "(" expr ")"
```

  A problem is “let inn+4” should be a syntax error, but a naïve parser implementation sees “let”, “in”, “n”, “+”, “4”.

  One solution is to use a parser for a reserved word should first read as many alphanums as possible, not just the expected letters, and then check that the whole string equals the keyword. This is what keyword does in an earlier section.

  Conversely, the parser for identifiers should read likewise, but then check that the string doesn't clash with reserved words. This is why identifier from earlier takes a parameter for reserved words to avoid.

- Here is the whole parser:

```
lesson4 :: Parser Expr
lesson4 = whitespaces *> expr <* eof
 where
  expr = local <|> adds

  local = pure (\_ eqns _ e -> Let eqns e)
        <*> keyword "let"
        <*> many equation
        <*> keyword "in"
        <*> expr
  -- Basically a liftA4.
  -- Could also be implemented in monadic style, like equation below.

  equation = var
          >>= \v -> operator "="
          >> expr
          >>= \e -> semicolon
          >> return (v, e)
  -- Basically a liftA4.
  -- Recall that liftA4 f a b c d = pure f <*> a <*> b <*> c <*> d
  -- Could also be implemented in applicative style, like local above.

  semicolon = char ';' *> whitespaces
  adds = chainl1 muls (operator "+" *> pure (Prim2 Add))
  muls = chainl1 atom (operator "*" *> pure (Prim2 Mul))
  atom = fmap Num natural
       <|> fmap Var var
       <|> (openParen *> expr <* closeParen)
  var = identifier ["let", "in"]
```

E.g.

```
*ParserLib> unParser lesson4 "let x=5; in x+5"
Just ("",Let [("x",Num 5)] (Prim2 Add (Var "x") (Num 5)))
*ParserLib> unParser lesson4 "let x=5; y=1; in x+5"
Just ("",Let [("x",Num 5),("y",Num 1)] (Prim2 Add (Var "x") (Num 5)))
*ParserLib> unParser lesson4 "let in x+5"
Just ("",Let [] (Prim2 Add (Var "x") (Num 5)))
```