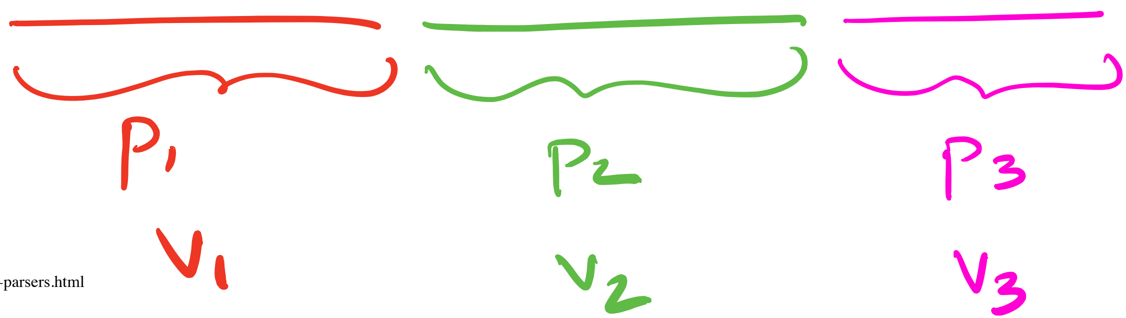
And now, let the *wild rumpus start!*



[ ]

"x + y :) z"

[PLUS (PLUS "x" "y") "z",
PLUS "x" (PLUS "y" "z")]

Parser Combinators

Lets write lots of *high-level* operators to **combine** parsers!

Here's a cleaned up `pairP`

```
pairP :: Parser a -> Parser b -> Parser (a, b)
pairP aP bP = do
  a <- aP
  b <- bP
  return (a, b)
```

A Failure Parser

Surprisingly useful, always *fails*

- i.e. returns `[]` no successful parses

```
failP :: Parser a
failP = P (\_ -> [])
```

QUIZ

Consider the parser

```
satP :: (Char -> Bool) -> Parser Char
satP p = do
  c <- oneChar
  if p c then return c else failP
```

What is the value of

```
quiz1 = runParser (satP (\c -> c == 'h')) "hellow"
quiz2 = runParser (satP (\c -> c == 'h')) "yellow"
```

	quiz1	**quiz2**
A	`[]`	`[]`
B	`[('h', "ellow")]`	`[('y', "ellow")]`
C	`[('h', "ellow")]`	`[]`
D	`[]`	`[('y', "ellow")]`

Parsing Alphabets and Numerics

We can now use `satP` to write

```haskell
-- parse ONLY the Char c
char :: Parser Char
char c = satP (\c' -> c == c')

-- parse ANY ALPHABET
alphaCharP :: Parser Char
alphaCharP = satP isAlpha

-- parse ANY NUMERIC DIGIT
digitChar :: Parser Char
digitChar = satP isDigit
```

QUIZ

We can parse a single `Int` digit

```
digitInt :: Parser Int
digitInt = do
  c <- digitChar       -- parse the Char c
  return (read [c])    -- convert Char to Int
```

What is the result of

```
quiz1 = runParser digitInt "92"
quiz2 = runParser digitInt "cat"
```

	quiz1	**quiz2**
A	`[]`	`[]`
B	`[('9', "2")]`	`[('c', "at")]`
C	`[(9, "2")]`	`[]`
D	`[]`	`[('c', "at")]`

EXERCISE

Write a function

```
strP :: String -> Parser String
strP s = -- parses EXACTLY the String s and nothing else
```

when you are done, we should get the following behavior

```
>>> dogeP = strP "doge"

>>> runParser dogeP "dogerel"
[("doge", "rel")]

>>> runParser dogeP "doggoneit"
[]
```

QUIZ: A Choice Combinator

Lets write a combinator `orElse p1 p2` such that

- returns the results of `p1`

or, else *if those are empty*

- returns the results of `p2`

```
:: Parser a -> Parser a -> Parser a
chooseP p1 p2 = -- produce non-empty results of `p1`
                -- or-else results of `p2`
```

e.g. `chooseP` lets us build a parser that produces an alphabet *OR* a numeric character

```
alphaNumChar :: Parser Char
alphaNumChar = alphaChar `orElse` digitChar
```

Which should produce

```
>>> runParser alphaNumChar "cat"
[('c', "at")]

>>> runParser alphaNumChar "2cat"
[('2', "cat")]

>>> runParser alphaNumChar "230"
[('2', "30")]
```

```haskell
-- a
orElse p1 p2 = do xs <- p1
                  ys <- p2
                  return (x1 ++ x2)
-- b
orElse p1 p2  = do xs <- p1
                   case xs of
                     [] -> p2
                     _  -> return xs

-- c
orElse p1 p2 = P (\cs -> runParser p1 cs ++ runParser p2 cs)

-- d
orElse p1 p2 = P (\cs -> case runParser p1 cs of
                           []  -> runParser p2 cs
                           r1s -> r1s)
```

An "Operator" for `orElse`

It will be convenient to have a short "operator" for `orElse`

```
p1 <|> p2 = orElse p1 p2
```

A Simple Expression Parser

Now, lets write a *tiny* calculator! data

```
-- 1. First, parse the operator
intOp      :: Parser (Int -> Int -> Int)
intOp      = plus <|> minus <|> times <|> divide
  where
    plus   = do { _ <- char '+'; return (+) }
    minus  = do { _ <- char '-'; return (-) }
    times  = do { _ <- char '*'; return (*) }
    divide = do { _ <- char '/'; return div }

-- 2. Now parse the expression!
calc :: Parser Int
calc = do x  <- digitInt
          op <- intOp
          y  <- digitInt
          return (x `op` y)
```

When `calc` is run, it will both parse *and* calculate

```
>>> runParser calc "8/2"
[(4,"")]

>>> runParser calc "8+2cat"
[(10,"cat")]

>>> runParser calc "8/2cat"
[(4,"cat")]

>>> runParser calc "8-2cat"
[(6,"cat")]

>>> runParser calc "8*2cat"
[(16,"cat")]
```

QUIZ

What will `quiz` evaluate to?

```
quiz = runParser calc "99bottles"
```

A. Type error

B. `[]`

C. `[(9, "9bottles")]`

D. `[(99, "bottles")]`

E. Run-time exception

Next: Recursive Parsing

Its cool to parse individual `Char` …

… but *way* more interesting to parse recursive structures!

```
"((2 + 10) * (7 - 4)) * (5 + 2)"
```

EXERCISE: A "Recursive" String Parser

The parser `string s` parses *exactly* the string `s` - fails otherwise

```
>>> runParser (string "mic") "mickeyMouse"
[("mic","keyMouse")]

>>> runParser (string "mic") "donald duck"
[]
```

Here's an implementation

```
string :: String -> Parser String
string ""     = return ""
string (c:cs) = do { _ <- char c; _ <- string cs; return (c:cs) }
```

Which library function will *eliminate* the recursion from `string`?

QUIZ: Parsing Many Times

Often we want to *repeat* parsing some object

```
-- | `manyP p` repeatedly runs `p` to return a list of [a]
manyP  :: Parser a -> Parser [a]
manyP p = m0 <|> m1
  where
    m0  = return []
    m1  = do { x <- p; xs <- manyP p; return (x:xs) }
```

Recall `digitChar :: Parser Char` returned a *single* numeric `Char`

What will `quiz` evaluate to?

```
quiz = runParser (manyP digitChar) "123horse"
```

A. `[("" , "1234horse")]` **B.** `[("1" , "234horse")]` **C.** `[("1", "23horse"), ("12", "3horse"), ("123", "horse")]` **D.** `[("123", "horse")]` **E.** `[]`

failure = \s -> []

vs

return [] = \s -> [([], s)]

Lets fix `manyP`*!*

Run `p` *first* and only `return []` if it fails …

```haskell
-- | `manyP p` repeatedly runs `p` to return a list of [a]
manyP  :: Parser a -> Parser [a]
manyP p = m1 <|> m0
  where
    m0  = return []
    m1  = do { x <- p; xs <- manyP p; return (x:xs) }
```

now, we can write an `Int` parser as

```haskell
int :: Parser Int
int = do { xs <- manyP digitChar; return (read xs) }
```

which will produce

```haskell
>>> runParser oneChar "123horse"
[("123", "horse")]

>>> runParser int "123horse"
[(123, "horse")]
```

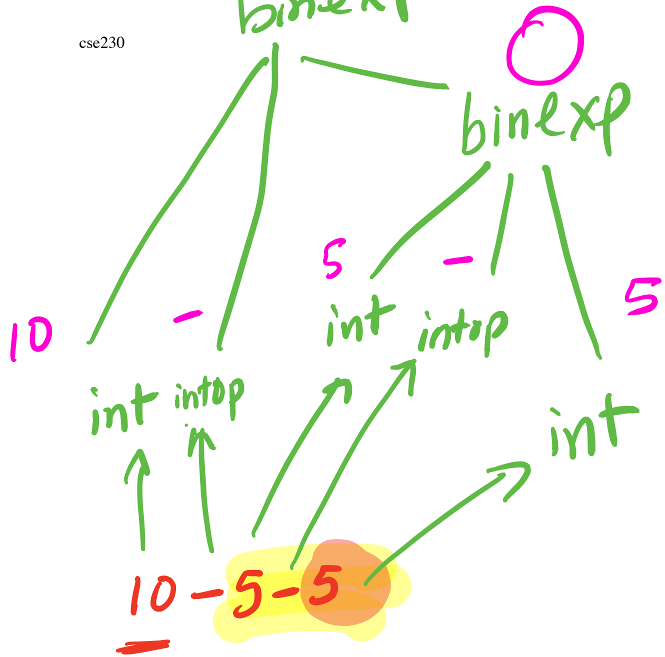
Parsing Arithmetic Expressions

Now we can build a proper calculator!

```
calc0 ::  Parser Int
calc0 = binExp <|> int

int :: Parser Int
int = do
  xs <- many digitChar
  return (read xs)


binExp :: Parser Int
binExp = do
  x <- int
  o <- intOp
  y <- calc0
  return (x `o` y)
```



Works pretty well!

```
>>> runParser calc0 "11+22+33"
[(66,"")]

ghci> doParse calc0 "11+22-33"
[(0,"")]
```

QUIZ

What does `quiz` evaluate to?

```
quiz = runParser calc0 "10-5-5"
```

A. `[(0, "")]`

B. `[]`

C. `[(10, "")]`

D. `[(10, "-5-5")]`

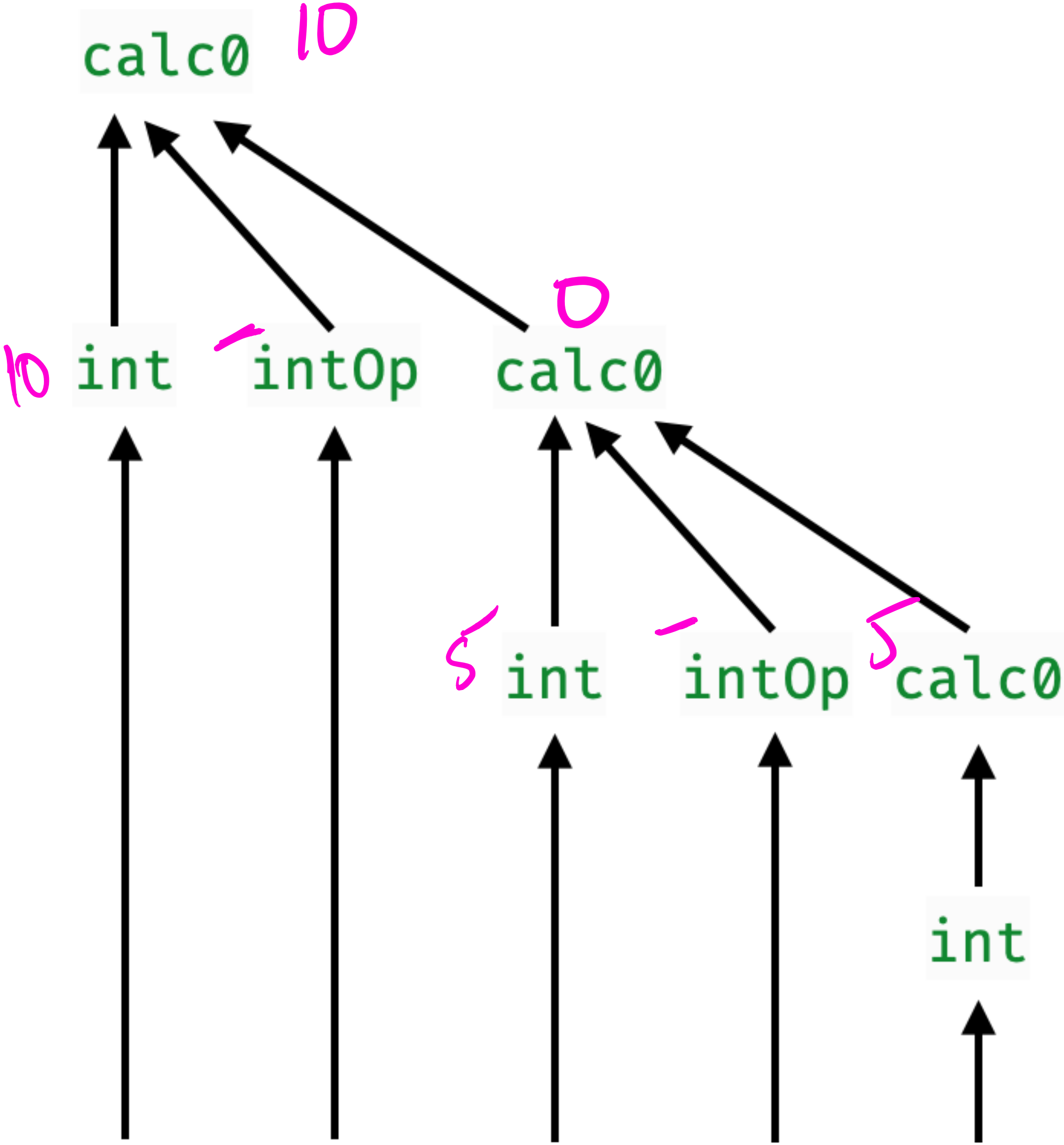
E. `[(5, "-5")]`

Problem: Right-Associativity

Recall

```haskell
binExp :: Parser Int
binExp = do
  x <- int
  o <- intOp
  y <- calc0
  return (x `o` y)
```

`"10-5-5"` gets parsed as `10 - (5 - 5)` because


calc0
10
int
intOp
calc0
0
10
-
int
intOp
calc0
5
-
5
int

10 - 5 - 5

The `calc0` parser implicitly forces each operator to be **right associative**

- doesn't matter for `+`, `*`
- but is incorrect for `-`

QUIZ

Recall


```
binExp :: Parser Int
binExp = do
  x <- int
  o <- intOp
  y <- calc0
  return (x `o` y)
```

What does `quiz` get evaluated to?

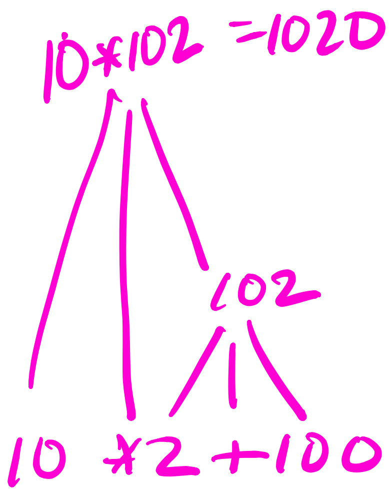
$(n_1 * n_2) + n_3$

```
quiz = runParser calc0 "10*2+100"
```

A. `[(1020,"")]` **B.** `[(120,"")]` **C.** `[(120,""), (1020, "")]` **D.** `[(1020,""), (120, "")]` **E.** `[]`

l-assoc

prec



(10* 100)+2

(10-5)-5

The `calc0` parser implicitly forces *all operators* to be **right associative**

- doesn't matter for `+`, `*`
- but is incorrect for `-`
- does not respect precedence!

Simple Fix: Parentheses!

Lets write a combinator that parses something within `(...)`

```
parensP :: Parser a -> Parser a
parensP p = do
  _ <- char '('
  x <- p
  _ <- char ')'
  return x
```

now we can try

```
calc1 :: Parser Int
calc1 = parens binExp <|> int
```

now the original string wont even parse

```
>>> runParser calc1 "10-5-5"
[]
```

but we can add parentheses to get the right result

```
>>> runParser calc1 "((10-5)-5)"
[(0 ,"")]

>>> runParser calc1 "(10-(5-5))"
[(10 ,"")]

>>> runParser calc1 "((10*2)+100)"
[(120, "")]

>>> runParser calc1 "(10*(2+100))"
[(1020, "")]
```

Left Associativity

But how to make the parser *left associative*

- i.e. parse "10-5-5" as `(10 - 5) - 5` ?

Lets flip the order!

```haskell
calc1      ::  Parser Int
calc1      = binExp <|> oneInt

binExp :: Parser Int
binExp = do
  x <- calc1
  o <- intOp
  y <- int
  return (x `o` y)
```

But …

```
>>> runParser calc1 "2+2"
...
```

Infinite loop! `calc1 --> binExp --> calc1 --> binExp --> ...`

- without *consuming* any input :-(

Solution: Parsing with Multiple Levels

Any expression is a **sum-of-products**

```
 10 * 20 * 30 + 40 * 50 + 60 * 70 * 80
=>
  ((((10 * 20) * 30) + (40 * 50)) + ((60 * 70) * 80))
=>
  ((((base * base) * base)  + (base * base)) + ((base * base) * base))
=>
  (((prod * base) + prod)  + (prod * base))
=>
  ((prod + prod) + prod)
=>
  (sum + prod)
=>
   sum
=>
   expr
```

Parsing with Multiple Levels

So lets **layer** our language as

```
expr :== sum
sum  :== (((prod "+" prod) "+" prod) "+" ... "+" prod)
prod :== (((base "*" base) "*" base) "*" ... "*" base)
base :== "(" expr ")" ORELSE int
```

that is the recursion looks like

```
expr = sum
sum  = oneOrMore prod "+"
prod = oneOrMore base "*"
base = "(" expr ")" <|> int
```

No infinite loop!

- `expr --> prod --> base -->* expr`
- but last step `-->*` *consumes* a `(`

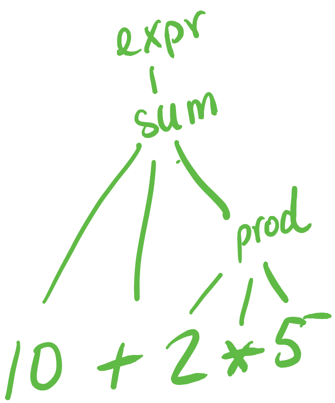
Parsing `oneOrMore`

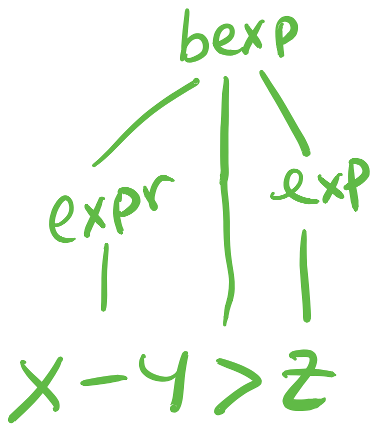
Lets implement `oneOrMore vP oP` as a combinator - `vP` parses a *single* `a` value - `oP` parses an *operator* `a -> a -> a` - `oneOrMore vP oP` parses and returns the result `((v1 o v2) o v3) o v4) o ... o vn)`

But how?

1. *grab* the first `v1` using `vP`
2. *continue* by
   - either trying `oP` then `v2` .. and recursively continue with `v1 o v2`
   - `orElse` (no more `o`) just return `v1`

```
oneOrMore :: Parser a -> Parser (a -> a -> a) -> Parser a
oneOrMore vP oP = do {v1 <- vP; continue v1}
  where
    continue v1 = do { o <- oP; v2 <- vP; continue (v1 `o` v2) }
               <|> return v1
```





"our" "parsec"

p1 <|> p2 try p1 <|> p2

Implementing Layered Parser

Now we can implement the grammar

```
expr = sum
sum  = oneOrMore prod "+"
prod = oneOrMore base "*"
base = "(" expr ")" <|> int
```

simply as

```
expr = sum
sum  = oneOrMore prod addOp
prod = oneOrMore base mulOp
base = parens expr <|> int
```

where `addOp` is `+` or `-` and `mulOp` is `*` or `/`

```
addOp, mulOp :: Parser (Int -> Int -> Int)
addOp = constP "+" (+) <|> constP "-" (-)
mulOp = constP "*" (*) <|> constP "/" div

constP :: String -> a -> Parser a
constP s x = do { _ <- string s; return x }
```

Lets make sure it works!

```
>>> doParse sumE2 "10-1-1"
[(8,"")]

>>> doParse sumE2 "10*2+1"
[(21,"")]

>>> doParse sumE2 "10+2*1"
[(12,"")]
```

Parser combinators

That was a taste of Parser Combinators

- Transferred from Haskell to *many* other languages (http://www.haskell.org/haskellwiki/Parsec).

Many libraries including Parsec (http://www.haskell.org/haskellwiki/Parsec) used in your homework - `oneOrMore` is called `chainl`

Read more about the *theory* - in these recent (http://www.cse.chalmers.se/~nad/publications/danielsson-parser-combinators.html) papers (http://portal.acm.org/citation.cfm?doid=1706299.1706347)

Read more about the *practice* - in this recent post that I like JSON parsing from scratch (https://abhinavsarkar.net/posts/json-parsing-from-scratch-in-haskell/)
