Programación Funcional Avanzada

Reconocedores Monádicos – Parsec

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

UNIVERSIDAD SIMÓN BOLÍVAR

Puedes correr, pero no esconderte...

...siempre hace falta un reconocedor

- Procesar texto estructurado, procesar archivos de configuración...
- ...escribir interpretadores o escribir compiladores.
- Siempre hace falta un reconocedor.

Haskell ofrece varias alternativas

UNIVERSIDAD SIMÓN BOLÍVAR

El problema de la persistencia

Los reconocedores “de pobre”

- Se crean estructuras de datos a tiempo de ejecución.
- Se desea conservarlas en memoria secundaria para futuras ejecuciones.

UNIVERSIDAD SIMÓN BOLÍVAR

El problema de la persistencia

Los reconocedores "de pobre"

- Se crean estructuras de datos a tiempo de ejecución.
- Se desea conservarlas en memoria secundaria para futuras ejecuciones.
- `Show a` – instancia que convierte de dato a `String`
  - El compilador la genera automáticamente.
  - "Imprimir" un dato en un archivo es simple – `print`.
  - El programador puede refinarla de ser necesario.

El problema de la persistencia

Los reconocedores "de pobre"

- Se crean estructuras de datos a tiempo de ejecución.
- Se desea conservarlas en memoria secundaria para futuras ejecuciones.
- `Show a` – instancia que convierte de dato a `String`
  - El compilador la genera automáticamente.
  - "Imprimir" un dato en un archivo es simple – `print`.
  - El programador puede refinarla de ser necesario.
- `Read a` – instancia que convierte de `String` a dato.
  - El compilador la genera automáticamente.
  - "Leer" un dato es simple – `read`
  - Refinarla no es nada sencillo.

UNIVERSIDAD SIMÓN BOLÍVAR

El problema de la persistencia

Los reconocedores "de pobre"

- Se crean estructuras de datos a tiempo de ejecución.
- Se desea conservarlas en memoria secundaria para futuras ejecuciones.
- `Show a` – instancia que convierte de dato a `String`
  - El compilador la genera automáticamente.
  - "Imprimir" un dato en un archivo es simple – `print`.
  - El programador puede refinarla de ser necesario.
- `Read a` – instancia que convierte de `String` a dato.
  - El compilador la genera automáticamente.
  - "Leer" un dato es simple – `read`
  - Refinarla no es nada sencillo.

Para muchos problemas simples, es una solución rápida.

UNIVERSIDAD SIMÓN BOLÍVAR

Parsec – Parser Combinators

- Librería de combinadores monádicos para construir reconocedores
  - Gramáticas sensibles al contexto.
  - *Lookahead* infinito – *backtracking*.
  - Máxima eficiencia en gramáticas predictivas $LL(k)$.

UNIVERSIDAD SIMÓN BOLÍVAR

Parsec – Parser Combinators

- Librería de combinadores monádicos para construir reconocedores
  - Gramáticas sensibles al contexto.
  - *Lookahead* infinito – *backtracking*.
  - Máxima eficiencia en gramáticas predictivas $LL(k)$.
- Ventajas sobre similares ("a mano", Parse::RecDescent, ANTLR, . . .)
  - Un sólo lenguaje para expresar el reconocedor – Haskell.
  - Reconocedores son objetos de primera clase – reconocedores complejos a partir de reconocedores simples.
  - Aprovechar funciones monádicas
    - Son `Monad` – todo lo de `Control.Monad`.
    - Son `Functor` – todo lo de `Data.Functor`.
    - Son `Applicative` – todo lo de `Control.Applicative`
- Debería ser la primera alternativa, a menos que se *necesite LR*.

UNIVERSIDAD SIMÓN BOLÍVAR

Parsec – Parser Combinators

- Librería de combinadores monádicos para construir reconocedores
  - Gramáticas sensibles al contexto.
  - *Lookahead* infinito – *backtracking*.
  - Máxima eficiencia en gramáticas predictivas $LL(k)$.
- Ventajas sobre similares ("a mano", Parse::RecDescent, ANTLR, . . .)
  - Un sólo lenguaje para expresar el reconocedor – Haskell.
  - Reconocedores son objetos de primera clase – reconocedores complejos a partir de reconocedores simples.
  - Aprovechar funciones monádicas
    - Son `Monad` – todo lo de `Control.Monad`.
    - Son `Functor` – todo lo de `Data.Functor`.
    - Son `Applicative` – todo lo de `Control.Applicative`
- Debería ser la primera alternativa, a menos que se *necesite LR*.

Parser monádico aplicativo

UNIVERSIDAD SIMÓN BOLÍVAR

Parsec – Robustez para aplicación práctica

Diseñado para ser aplicable en el "mundo real"

- Velocidad – miles de líneas por segundo.
- Reporte de errores – alto nivel, localizables y contextuales.
- Componentes básicos y complejos "listos para usar"
  - Analizador lexicográfico.
  - Reconocedor general de expresiones.
  - Reconocedor para permutaciones.
- Documentación – usuario y referencia.
- Predictivo por omisión, *backtracking* debe indicarse explícitamente.

UNIVERSIDAD SIMÓN BOLÍVAR

Reconocer un archivo CSV *light*

- Un archivo CSV (*Comma Separated Values*)...
  - Varias líneas, seguidas por el fin de archivo.
  - Cada línea tiene uno o más campos, seguidos por el fin de línea.
  - Campos separados por comas – no contienen comas ni saltos de línea.

Reconocer un archivo CSV *light*

- Un archivo CSV (*Comma Separated Values*)...
  - Varias líneas, seguidas por el fin de archivo.
  - Cada línea tiene uno o más campos, seguidos por el fin de línea.
  - Campos separados por comas – no contienen comas ni saltos de línea.
- Nuestro reconocedor
  - Debe recibir un `String` como entrada.
  - ¿Estructura correcta? – lista de listas de campos `[[String]]`.
  - ¿Estructura incorrecta? – indicar el error.

UNIVERSIDAD SIMÓN BOLÍVAR

Reconocer un archivo CSV – Primer intento

```
import Text.ParserCombinators.Parsec

csv :: Parser [[String]]
csv = do r <- many line
         eof
         return r

line :: Parser [String]
line = do r <- cells
          eol
          return r

cells :: Parser [String]
cells = do f <- content
           n <- moreCells
           return (f:n)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Reconocer un archivo CSV – Primer intento

```
moreCells :: Parser [String]
moreCells =  (char ',' >> cells)
         <|> (return [])

content :: Parser String
content = many $ noneOf ",\n"

eol :: Parser Char
eol = char '\n'

parseCSV :: String -> Either ParseError [[String]]
parseCSV = parse csv "csv"
```

Eso es todo – en serio.

UNIVERSIDAD SIMÓN BOLÍVAR

Combinadores de Parsec

Un API simple, pero constructivo

- `many p` – aplicar el *parser* `p` tantas veces como pueda.
- `eof` – reconoce el fin de archivo.
- `char c` – reconoce un caracter específico.
- `noneOf s` – reconoce un caracter que **no** esté en el conjunto indicado.
- `p <|> q` (*choice*)
  - Intenta aplicar el *parser* `p`
  - Si el *parser* `p` falla pero **no consumió nada de la entrada**, entonces aplica el *parser* `q`.
- `parse p n` – aplica el *parser* `p` sobre la entrada, usando `n` para identificar los mensajes de error.

¿Cómo funcionan?

UNIVERSIDAD SIMÓN BOLÍVAR

El primer éxito o fracaso al final, ¿suena conocido?

Corto-circuito de resultados

- ¿Cuál es la mecánica de un reconocedor (*parser*)?

El primer éxito o fracaso al final, ¿suena conocido?

Corto-circuito de resultados

- ¿Cuál es la mecánica de un reconocedor (*parser*)?
  - Procesa texto (`String`) de entrada.

UNIVERSIDAD SIMÓN BOLÍVAR

El primer éxito o fracaso al final, ¿suena conocido?

Corto-circuito de resultados

- ¿Cuál es la mecánica de un reconocedor (*parser*)?
  - Procesa texto (`String`) de entrada.
  - Consume la parte que cumple con determinado criterio, retornando el resto junto con alguna representación de lo consumido – corresponde a un éxito inmediato.

UNIVERSIDAD SIMÓN BOLÍVAR

El primer éxito o fracaso al final, ¿suena conocido?

Corto-circuito de resultados

- ¿Cuál es la mecánica de un reconocedor (*parser*)?
  - Procesa texto (`String`) de entrada.
  - Consume la parte que cumple con determinado criterio, retornando el resto junto con alguna representación de lo consumido – corresponde a un éxito inmediato.
  - Si no consume nada de la entrada, intenta el siguiente reconocedor – secuencia de alternativas en algún orden práctico.

UNIVERSIDAD SIMÓN BOLÍVAR

El primer éxito o fracaso al final, ¿suena conocido?

Corto-circuito de resultados

- ¿Cuál es la mecánica de un reconocedor (*parser*)?
  - Procesa texto (`String`) de entrada.
  - Consume la parte que cumple con determinado criterio, retornando el resto junto con alguna representación de lo consumido – corresponde a un éxito inmediato.
  - Si no consume nada de la entrada, intenta el siguiente reconocedor – secuencia de alternativas en algún orden práctico.
  - Si ninguno tiene éxito, reporta la falla de forma razonable – fracaso con diagnóstico.

UNIVERSIDAD SIMÓN BOLÍVAR

El primer éxito o fracaso al final, ¿suena conocido?

Corto-circuito de resultados

- ¿Cuál es la mecánica de un reconocedor (*parser*)?
  - Procesa texto (`String`) de entrada.
  - Consume la parte que cumple con determinado criterio, retornando el resto junto con alguna representación de lo consumido – corresponde a un éxito inmediato.
  - Si no consume nada de la entrada, intenta el siguiente reconocedor – secuencia de alternativas en algún orden práctico.
  - Si ninguno tiene éxito, reporta la falla de forma razonable – fracaso con diagnóstico.
- ¡Esto es exactamente lo que ofrece `MonadPlus`!
  - `Parsec` es `Functor` y `Applicative`.
  - `Parsec` es `Monad` y `MonadPlus`.
  - `Parsec` es `MonadTransformer`.

Reconocedor recursivo descendente con *backtracking*.

UNIVERSIDAD SIMÓN BOLÍVAR

Construyamos un reconocedor

. . . aprovechar `Parsec` será más fácil después

- Procesa un `String` como entrada –
  retorna los posibles resultados con el resto de la entrada

```
newtype Parser a = Parser (String -> [(a,String)]
```

  - Fracaso – lista vacía.
  - Múltiples resultados – tuplas con lo reconocido y entrada por procesar.
  - Modelamos la ambigüedad – posibilidad de *backtrack*.

Obviaremos errores y AST por ahora.

UNIVERSIDAD SIMÓN BOLÍVAR

Parser y sus instancias

Parser es Monad – híbrido List y State.

```
instance Monad Parser where
  return a = Parser $ \cs -> [(a,cs)]
  p >>= f  = Parser $ \cs -> concat [
      parse (f a) cs' | (a,cs') <- parse p cs
                                   ]

parse (Parser p) = p
```

Parser y sus instancias

Parser es Monad – híbrido List y State.

```
instance Monad Parser where
  return a = Parser $ \cs -> [(a,cs)]
  p >>= f  = Parser $ \cs -> concat [
      parse (f a) cs' | (a,cs') <- parse p cs
                                   ]

parse (Parser p) = p
```

Parser es MonadPlus.

```
instance MonadPlus Parser where
  mzero       = Parser $ \cs -> []
  p `mplus` q = Parser $ \cs -> parse p cs ++
                                parse q cs
```

UNIVERSIDAD SIMÓN BOLÍVAR

Escribiendo combinadores

Un ejemplo trivial

- `item` – reconocer un caracter cualquiera

```
item :: Parser Char
item = Parser $ \s -> case s of
                        ""     -> []
                        (c:cs) -> [(c,cs)]
```

...ya nos permite escribir cosas como

```
p :: Parser (Char,Char)
p = do
  c <- item
  item
  d <- item
  return (c,d)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Escribiendo combinadores

Reconocer caracteres que cumplan algún predicado

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = do
  c <- item
  if p c then return c else return mzero
```

Escribiendo combinadores

Reconocer caracteres que cumplan algún predicado

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = do
  c <- item
  if p c then return c else return mzero
```

- Reconocer el caracter c en particular

```
char :: Char -> Parser Char
char c = satisfy (c==)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Escribiendo combinadores

Reconocer caracteres que cumplan algún predicado

```
satisfy :: (Char -> Bool) -> Parser Char
satisfy p = do
  c <- item
  if p c then return c else return mzero
```

- Reconocer el caracter c en particular

```
char :: Char -> Parser Char
char c = satisfy (c==)
```

- Reconocer cualquier dígito

```
digit :: Parser Char
digit = satisfy isDigit
```

UNIVERSIDAD SIMÓN BOLÍVAR

Controlando el no-determinismo

. . . porque a veces quiero uno o el otro, excluyente

- `mplus` equivale al no-determinismo
  - Se intentan *ambos* reconocedores.
  - Se combinan sus resultados.
  - En ocasiones sólo interesa si el primero de una secuencia tiene éxito.
- Definimos un combinador de alternativa determinística

```
(+++) :: Parser a -> Parser a -> Parser a
p (+++) q = Parser (
                \s -> case parse (p `mplus` q) of
                            []     -> []
                            (x:xs) -> [x]
            )
```

- `p +++ q` – expresión regular $p + q$

Combinadores recursivos

Una afortunada cadena de eventos...

- Reconocer una cadena particular

```
string :: String -> Parser String
string ""     = return ""
string (c:cs) = do char c
                   string cs
                   return (c:cs)
```

Combinadores recursivos

Una afortunada cadena de eventos...

- Reconocer una cadena particular

```
string :: String -> Parser String
string ""     = return ""
string (c:cs) = do char c
                   string cs
                   return (c:cs)
```

- Aplicar el reconocedor p cero o más veces

```
many, many1 :: Parser a -> Parser [a]
many p = many1 p (+++) return []

many1 p = do a <- p
             as <- many p
             return (a:as)
```

UNIVERSIDAD SIMÓN BOLÍVAR

Combinadores recursivos

Una afortunada cadena de eventos...

- Reconocer una cadena particular

```
string :: String -> Parser String
string ""     = return ""
string (c:cs) = do char c
                   string cs
                   return (c:cs)
```

- Aplicar el reconocedor p cero o más veces

```
many, many1 :: Parser a -> Parser [a]
many p = many1 p (+++) return []

many1 p = do a <- p
             as <- many p
             return (a:as)
```

- many p – expresión regular p^*
- many1 p – expresión regular pp^*

UNIVERSIDAD SIMÓN BOLÍVAR

Combinadores recursivos

Separadores

- Aplicar el reconocedor `p`, intercalando aplicaciones del reconocedor `sep` cuyo resultado no es importante pues actúa como separador

```
sepBy :: Parser a -> Parser b -> Parser [a]
p `sepBy` sep = (p `sepBy1` sep) +++ return []

sepBy1 :: Parser a -> Parser b -> Parser [a]
p `sepBy1` sep = do a <- p
                    as <- many do sep
                                  p
                    return (a:as)
```

Combinadores recursivos

Separadores

- Aplicar el reconocedor p, intercalando aplicaciones del reconocedor sep cuyo resultado no es importante pues actúa como separador

```
sepBy :: Parser a -> Parser b -> Parser [a]
p `sepBy` sep = (p `sepBy1` sep) +++ return []

sepBy1 :: Parser a -> Parser b -> Parser [a]
p `sepBy1` sep = do a <- p
                    as <- many do sep
                                  p
                    return (a:as)
```

- sepBy1 – expresión regular $p(sep\ p)^*$
- sepBy – expresión regular $p(sep\ p)^* + \lambda$

UNIVERSIDAD SIMÓN BOLÍVAR

De regreso en Parsec

- Parsec está construido con el mismo principio
  - Parametriza el producto del reconocedor – AST.
  - Parametriza el tipo de la entrada – *token* genérico.
  - Permite que el usuario incluya estado adicional.

```
GenParser tok st a
```

- En la práctica *tokens* son caracteres y no se incorpora estado.

```
type Parser a = GenParser Char () a
```

- Se aplica a una entrada y se ofrece el resultado en el Monad Error

```
parse :: Parser a
      -> SourceName -> String
      -> Either ParseError a
```

UNIVERSIDAD SIMÓN BOLÍVAR

De regreso en Parsec

Más combinadores

- many, many1, sepBy y sepBy1 – idénticos a los ya descritos.

```
id = do c <- letter
        cs <- many (letter <|> digit <|> char '_')
        return (c:cs)

word = many1 letter

lista p = p `sepBy` (char ',')
```

- endBy y endBy1 – aplica un reconocedor cero o más veces, separado y terminado con otro.

```
instrucciones = instruccion `endBy` (char ';')
```

UNIVERSIDAD SIMÓN BOLÍVAR

De regreso en Parsec

Más combinadores

- `between` – aplicar un reconocedor encerrado entre dos reconocedores, posiblemente diferentes

```
entrellaves = between (char '{') (char '}')

parametros = between (char '(') (char ')') $
               argument `sepBy` (char ',')
```

- `string` – reconoce una cadena específica.

```
program = do string "begin"
             instrucciones
             string "end"
```

UNIVERSIDAD SIMÓN BOLÍVAR

De regreso en `Parsec`

Lookahead infinito

- Si nuestra gramática no es $LL(1)$ habrá ambigüedades.
- Los combinadores siempre consumen entrada hasta fallar.
- Consideremos `p <|> q`
  - Si `p` y `q` tienen prefijos similares, una falla en `p` ocasionará la falla en `q`.
  - Una solución es factorizar por izquierda – a veces no se puede.

```
bad = string "for" <|> string "foreach" <|> ident
```

UNIVERSIDAD SIMÓN BOLÍVAR

De regreso en `Parsec`

Lookahead infinito

- Si nuestra gramática no es $LL(1)$ habrá ambigüedades.
- Los combinadores siempre consumen entrada hasta fallar.
- Consideremos `p <|> q`
  - Si `p` y `q` tienen prefijos similares, una falla en `p` ocasionará la falla en `q`.
  - Una solución es factorizar por izquierda – a veces no se puede.

```
bad = string "for" <|> string "foreach" <|> ident
```

- `try p` – Intenta reconocer `p`.
  - Si tiene éxito, entrega el resultado.
  - Si fracasa, se "devuelve" la entrada consumida.

```
good = try (string "for")
       <|> try (string "foreach")
       <|> try ident
```

UNIVERSIDAD SIMÓN BOLÍVAR

De regreso en `Parsec`

Manejo de errores

- Un reconocedor falla retornando un error (`ParseError`)
  - `errorPos` extrae la posición del error.
  - `errorMessages` extrae los mensajes de error.
- A partir de la posición (`SourcePos`)
  - `sourceName`, `sourceLine` y `sourceCol` obtienen los detalles.
  - Más funciones para control fino de las posiciones.
- <?> – modificación de errores
  - `p <?> msg` – se comporta como `p` en caso de éxito.
  - Pero en caso de fallar **sin** consumir entrada, el mensaje de error estándar de `p` será *reemplazado* con `msg`.

UNIVERSIDAD SIMÓN BOLÍVAR

Reconocer archivos CSV – *Reloaded*

Con todos los jugueticos

El nivel más alto de abstracción

```
import Text.ParserCombinators.Parsec

csvFile = endBy line eol
line = sepBy cell (char ',')
cell = quotedCell <|> many (noneOf ",\n\r")
```

Reconocer archivos CSV – *Reloaded*

Con todos los jugueticos

El nivel más alto de abstracción

```
import Text.ParserCombinators.Parsec

csvFile = endBy line eol
line = sepBy cell (char ',')
cell = quotedCell <|> many (noneOf ",\n\r")
```

Ahora es trivial incluir celdas entre comillas

```
quotedCell = do
  char '"'
  content <- many quotedChar
  char '"' <?> "quote at end of cell"
  return content
```

UNIVERSIDAD SIMÓN BOLÍVAR

Reconocer archivos CSV – *Reloaded*

Con todos los jugueticos

Un caracter “escapado”

```
quotedChar =
  noneOf "\"" <|> try (string "\"\"" >> return '"')
```

Reconocer archivos CSV – *Reloaded*

Con todos los jugueticos

Un caracter "escapado"

```
quotedChar =
  noneOf "\"" <|> try (string "\"\"" >> return '"')
```

El fin de línea en cualquier sistema operativo

```
eol = try (string "\n\r")
      <|> try (string "\r\n")
      <|> string "\n"
      <|> string "\r"
      <?> "end of line"
```

UNIVERSIDAD SIMÓN BOLÍVAR

Reconocer archivos CSV – *Reloaded*

Con todos los jugueticos

El programa principal

```
parseCSV = parse csvFile

main = do input <- getContents
          case parseCSV "(stdin)" input of
            Left c  -> do putStrLn "Error: "
                          print c
            Right r -> mapM_ print r
```

UNIVERSIDAD SIMÓN BOLÍVAR

Un ejemplo más ...

Algunas aplicaciones web (mal hechas) reciben parámetros a través del URL vía la operación HTTP GET.

```
foo=Has+space&bar=%42comillas%42&baz=qux
```

- La codificación se llama `application/x-www-form-urlencoded`.
- Cada pareja clave-valor está separada por un *ampersand*.
- En un valor, los espacios en blanco se codifican con un símbolo +.
- Cualquier otro símbolo está en hexadecimal, precedido por un %.

UNIVERSIDAD SIMÓN BOLÍVAR

Usando `Parsec`

Las parejas están separadas por *ampersand*

```
p_query = p_pair `sepBy` char '&'
```

Usando Parsec

Las parejas están separadas por *ampersand*

```
p_query = p_pair `sepBy` char '&'
```

En la pareja, es posible que haya clave sin valor

```
p_pair = do
  name <- many1 p_char
  value <- optionMaybe (char '=' >> many p_char)
  return (name,value)
```

- `optionMaybe` envuelve el resultado de un reconocedor en `Maybe`.
- Las tuplas resultantes serán `(String,Maybe String)`.

UNIVERSIDAD SIMÓN BOLÍVAR

Usando Parsec

Resta reconocer caracters y dígitos hexadecimales

```
p_char = oneOf urlBaseChars
       <|> (char '+' >> return " ")
       <|> p_hex

urlBaseChars = ['a'..'z'] ++ ['A'..'Z'] ++
               ['0'..'9'] ++
               "$-_.|*'(),"

p_hex = do char '%'
           a <- hexDigit
           b <- hexDigit
           let ((d,_):_) = readHex [a,b]
           return . toEnum $ d
```

- `hexDigit` – combinador que reconoce dígitos hexadecimales.
- `readHex` está en el módulo `Numeric`.

UNIVERSIDAD SIMÓN BOLÍVAR

Aumentando la abstracción

Abreviar aprovechando combinadores monádicos – en lugar de . . .

```
p_pair = do
  name <- many1 p_char
  value <- optionMaybe (char '=' >> many p_char)
  return (name,value)
```

Aumentando la abstracción

Abreviar aprovechando combinadores monádicos – en lugar de . . .

```
p_pair = do
  name <- many1 p_char
  value <- optionMaybe (char '=' >> many p_char)
  return (name,value)
```

. . . puede escribirse

```
p_pair =
  liftM2 (,) (many1 p_char)
             optionMaybe (char '=' >> many p_char)
```

¡*Aplicar* reconocedores y *combinar* resultados!

UNIVERSIDAD SIMÓN BOLÍVAR

Gramática de expresiones

Reconocedor evaluador – Expresiones

```
expr :: Parser Integer
expr =
  do t <- term
     do try (char '+') >> expr >>= return . (t+)
        <|> (try (char '-') >> expr >>= return . (t-))
        <|> return t
     <?> "expr"
```

$$E \rightarrow T + E$$
$$E \rightarrow T - E$$
$$E \rightarrow T$$

UNIVERSIDAD SIMÓN BOLÍVAR

Gramática de expresiones

Reconocedor evaluador – Términos

```
term =
  do p <- power
     do try (char '*') >> term >>= return . (p*)
        <|> (try (char '/') >> term
                            >>= return . (div p))
        <|> return p
     <?> "term"
```

$$T \to P * T$$
$$T \to P / T$$
$$T \to P$$

UNIVERSIDAD SIMÓN BOLÍVAR

Gramática de expresiones

Reconocedor evaluador – Potencias

```
power =
  do b <- factor
     do try (char '^') >> power >>= return . (b^)
        <|> return b
     <?> "power"
```

$$P \rightarrow F \ \wedge \ P$$

$$P \rightarrow F$$

Gramática de expresiones

Sólo el reconocedor – Factores

```
factor = do char '('
            e <- expr
            char ')'
            return e
       <|> number
       <?> "factor"

number = do ds <- many1 digit
            return (read ds)
       <?> "number"
```

$$F \rightarrow (\ E \ )$$

$$F \rightarrow \textbf{num}$$

UNIVERSIDAD SIMÓN BOLÍVAR

Mientras tanto, en `Text.Parsec`

- Funciones de acceso para línea, columna y archivo en proceso – para refinar el reporte de errores.
- `Text.Parsec.Language` – lenguajes listos para usar
  - ¿Necesitas un analizador lexicográfico **de** Haskell?

```
haskell :: TokenParser st
```

  - ¿Necesitas un reconocedor **de** Haskell?

```
haskellDef :: LanguageDef st
```

- `Text.Parsec.Token` – generador de lexicográficos.
- `Text.Parsec.Bytestring` – para el mundo real.

UNIVERSIDAD SIMÓN BOLÍVAR

¿Tienes que implantar un lenguaje completo?

Primero un lexer – `Text.Parsec.Token`

- Definir patrones para comentarios, identificadores, palabras reservadas, operadores reservados...

```
guayoyoDef = emptyDef {
    commentStart   = "/*",
    commentEnd     = "*/",
    identStart     = letter <|> char '_'
    identLetter    = alphaNum <|> char '_'
    opStart        = oneOf "*+-/<=>"
    reservedNames = [ "if", "while", "for", ... ]
}
```

UNIVERSIDAD SIMÓN BOLÍVAR

¿Tienes que implantar un lenguaje completo?

Primero un lexer – `Text.Parsec.Token`

- Definir patrones para comentarios, identificadores, palabras reservadas, operadores reservados...

```
guayoyoDef = emptyDef {
    commentStart   = "/*",
    commentEnd     = "*/",
    identStart     = letter <|> char '_'
    identLetter    = alphaNum <|> char '_'
    opStart        = oneOf "*+-/<=>"
    reservedNames = [ "if", "while", "for", ... ]
}
```

- ...y luego `makeTokenParser` *genera* el reconocedor Parsec.

```
guayoyoTokenParser = makeTokenParser guayoyoDef
```

UNIVERSIDAD SIMÓN BOLÍVAR

¿Tienes que implantar un lenguaje completo?

Seguro hay expresiones – `Text.Parsec.Expr`

- Definir nombres de operadores, asociatividades y precedencias

```
table   = [
  [ prefix "-" negate, prefix "+" id ],
  [ postfix "++" (+1), postfix "--" (-1) ],
  [ binary "*" (*) AssocLeft,
    binary "/" (div) AssocLeft ],
  [ binary "+" (+) AssocLeft,
    binary "-" (-)   AssocLeft ]
]
binary  n f = Infix (reservedOp n >> return f)
prefix  n f = Prefix (eservedOp n >> return f)
postfix n f = Postfix (reservedOp n >> return f)
```

- ...`buildExpressionParser` *genera* el reconocedor Parsec.

```
expr = buildExpressionParser table
```

UNIVERSIDAD SIMÓN BOLÍVAR

Escribir un compilador o interpretador complejo

Cuando *LR* es inevitable

- Cuando el lenguaje requiere un análisis de contexto más complejo – transformarlo a *LL* puede ser imposible o impráctico.

UNIVERSIDAD SIMÓN BOLÍVAR

Escribir un compilador o interpretador complejo

Cuando *LR* es inevitable

- Cuando el lenguaje requiere un análisis de contexto más complejo – transformarlo a *LL* puede ser imposible o impráctico.
- `Alex` – generador de analizadores lexicográficos.
  - Lista de expresiones regulares asociadas con acciones Haskell.
  - Funcional puro o monádico – depende del reconocedor en uso.

UNIVERSIDAD SIMÓN BOLÍVAR

Escribir un compilador o interpretador complejo

Cuando *LR* es inevitable

- Cuando el lenguaje requiere un análisis de contexto más complejo – transformarlo a *LL* puede ser imposible o impráctico.
- `Alex` – generador de analizadores lexicográficos.
  - Lista de expresiones regulares asociadas con acciones Haskell.
  - Funcional puro o monádico – depende del reconocedor en uso.
- `Happy` – generador de reconocedores *LR*.
  - Reglas gramaticales con acciones Haskell asociadas – las acciones se ejecutan al reducir (DFS de izquierda a derecha).
  - Reconocedor puro o monádico – intención es hilvanar un `Monad` a la medida para gestión de estado, manejo de errores y I/O.
  - *LALR*(1) por omisión – *GLR* eficiente para gramáticas ambiguas.
  - Soporte *explícito* para gramáticas de atributos.

Eso es lo que usarías en Traductores o en la cadena de Lenguajes de Programación.

Quiero saber más...

- Sección de `Parsec` en Haskell WiKi
- Documentación de `Parsec`
- Un reconocedor para Scheme usando Parsec
- Alex User Guide
- Happy User Guide

UNIVERSIDAD SIMÓN BOLÍVAR