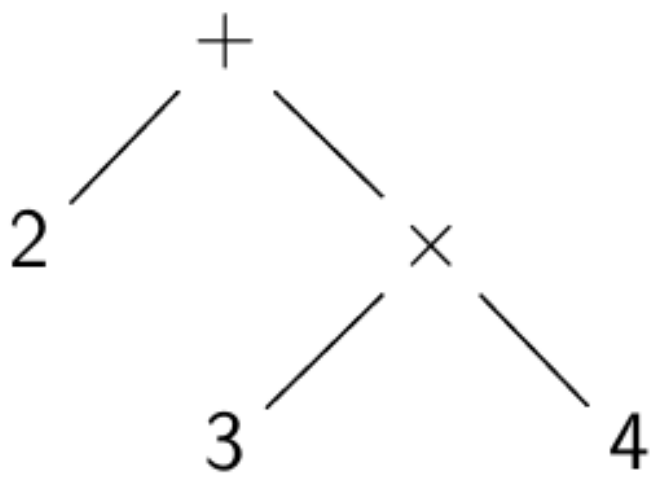
Analizadores Sintácticos (Parsers)

3/9/2025

¿Qué es un parser?

Un **parser** es un programa que toma una cadena de caracteres y devuelve la **estructura sintáctica** de ésta.

$$2 + 3 * 4 \quad \Rightarrow$$



¿Dónde se usan?

Se usan en la mayoría de los programas para preprocesar la entrada de los mismos. Por ejemplo:

- Un programa de calculadora parsea las expresiones numéricas antes de evaluarlas.
- Un compilador parsea los programas antes de ejecutarlos.
- Un navegador, parsea documentos HTML antes de mostrarlos.

El tipo de los parsers

- Éste tipo captura la idea básica de qué es un parser:

```
type Parser = String -> Tree
```

pero no es suficiente para programar un parser.

- Un parser puede no consumir toda la entrada y debe devolver lo que no consumió:

```
type Parser = String -> (Tree, String)
```

El tipo de los parsers

- Un parser puede fallar:

```
type Parser = String ->[(Tree, String)]
```

  la lista vacía representa una falla y una unitaria un éxito.

- Es probable que diferentes parsers devuelvan distintos tipos de árbol, o cualquier tipo de valor.

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

Parsers básicos

Un parser se construye a partir de parsers básicos:

- El parser que siempre tiene éxito, no consume la entrada y devuelve un valor dado:

```
return :: a -> Parser a
return v = \s -> [(v , s)]
```

- El parser que siempre falla:

```
failure :: Parser a
failure = \_ -> []
```

Parsers básicos

- El parser que devuelve el 1° caracter de su entrada o falla si ésta es vacía:

```
item :: Parser Char
item (x:xs) =  [(x, xs)]
item []     =  []
```

- Para aplicar un parser definimos una función:

```
parse :: Parser a->String->[(a,String)]
parse p s = p s
```

Ejemplos

```
> parse (return 1) "abc"
[(1, "abc")]

> parse failure "abc"
[]

> parse item "abc"
[('a', "bc")]

> parse item ""
[]
```

Combinadores de parsers: choice

El parser `p <|> q` se comporta como el parser `p` si éste tiene éxito, y como el parser `q` sino:

```
(p <|> q)::Parser a -> Parser a ->Parser a
(p <|> q) s = case p s of
                [] -> parse q inp
                [(v , out)] -> [(v , out)]
```

Por ejemplo:

```
> (item <|> return 1) "abc"
[('a', "abc")]

> (item <|> return 1) ""
[(1, "")]
```

Combinadores de parsers: sucuenciamiento

Podríamos combinar dos parsers de varias formas, por ejemplo con un operador de tipo:

`Parser a -> Parser b -> Parser (a,b).`

En la práctica es más conveniente usar este operador:

```
(>>=)::Parser a ->(a ->Parser b)->Parser b
(p >>= f) s = case (parse p s) of
                [] -> []
                [(v, out)]->parse (f v) out
```

Utilizaremos este operador con la notación **do**.

Combinadores de parsers: sucuenciamiento

El fragmento de programa:

```
p1 >>= \v1 ->
p2 >>= \v2 ->
...
pn >>= \vn -> return (f v1 v2 ... vn)
```

puede expresarse con la notación **do** como:

```
do v1 <- p1
   v2 <- p2
   ...
   vn <- pn
   return (f v1 v2 ... vn)
```

Comentarios sobre notación **do**

- Se aplica la regla de **layout**, cada parser debe comenzar en la misma columna.
- Si el valor de un parser `pi` intermedio no se utiliza más adelante, se puede omitir la parte: `vi <-`
- El valor devuelto por un secuenciamiento de parsers es el valor devuelto por el último parser de la secuencia.
- Esta notación no es propia del operador >>= dado sino de cualquier mónada.

Ejemplo

Aplica 3 parsers y devuelve el resultado del 1° y el 3° en un par:

```
p :: Parser (Char, Char)
p = do   x <- item
         item
         y <- item
         return (x, y)

> parse p "abcdef"
[(('a', 'c'), "def")]

> parse p "ab"
[]
```

Primitivas derivadas

Parsea un caracter si éste satisface el predicado:

```
sat :: (Char -> Bool) -> Parser Char
sat p = do x <- item
           if p x  then return x
                   else failure
```

Por ejemplo, podemos usar `sat` para parsear un dígito:

```
digit:: Parser Char
digit = sat isDigit
```

Primitivas derivadas

o un caracter dado:

```
char :: Char -> Parser Char
char x = sat (x ==)
```

o una cadena dada:

```
string :: String -> Parser String
string []     = return []
string (x:xs) = do char x
                   string xs
                   return (x:xs)
```

Primitivas derivadas

`many` y `many1` aplican un parser muchas veces hasta que fallan, devolviendo los valores parseados en una lista.

- Aplica un parser 0 o más veces:

```
many :: Parser a -> Parser [a]
many p = many1 p <|> return []
```

- Aplica un parser 1 o más veces:

```
many1 :: Parser a -> Parser [a]
many1 p = do v <- p
             vs <- many p
             return (v:vs)
```

Más Primitivas

- Parsea un número natural:

```
nat :: Parser Int
nat = do xs <- many1 digit
         return (read xs)
```

- Parsea 0 o más espacios:

```
space :: Parser ()
space = do many (sat isSpace)
           return ()
```

Espacios

- Primitiva que aplica un parser ignorando los espacios:

```
token :: Parser a -> Parser a
token p = do space
             v <- p
             space
             return v
```

- Parsea una cadena ignorando espacios:

```
symbol :: String -> Parser String
symbol xs = token (string xs)
```

- Parsea un natural ignorando espacios:

```
natural :: String -> Parser Int
natural xs = token (nat xs)
```

Ejemplo

Parser de lista de naturales:

```
listnat :: Parser [Int]
listnat = do symbol "["
             n <- natural
             ns <- many (do symbol ","
                            natural)
             symbol "]"
             return (n:ns)
```

```
> parse listnat "[1, 2, 3] "
[([1,2,3], "")]

> parse listnat "[1, 2,]"
[]
```

Gramática expresiones aritméticas

- Consideremos la gramática libre de contexto:

$$
\begin{array}{lcl}
\mathit{exp} & ::= & \mathit{term}\ \text{'+'}\ \mathit{exp} \mid \mathit{term} \\
\mathit{term} & ::= & \mathit{atom}\ \text{'*'}\ \mathit{term} \mid \mathit{atom} \\
\mathit{atom} & ::= & \text{'('}\ \mathit{exp}\ \text{')'} \mid n \\
n & ::= & d \mid dn \\
d & ::= & \text{'0'} \mid \text{'1'} \mid \ldots \mid \text{'9'}
\end{array}
$$

- Simplificaremos la gramática factorizando las reglas para *expr* y *term*:

$$
\begin{array}{lcl}
\mathit{exp} & ::= & \mathit{term}\ (\text{'+'}\ \mathit{exp} \mid \epsilon) \\
\mathit{term} & ::= & \mathit{atom}\ (\text{'*'}\ \mathit{term} \mid \epsilon)
\end{array}
$$

Parser de expresiones aritméticas

```
expr :: Parser Int
expr = do t <- term
          do symbol "+"
             e <- expr
             return (t+e)
          <|> return t
```

```
term :: Parser Int
term = do f <- factor
          do symbol "*"
             t <- term
             return (f*t)
          <|> return f
```

Parser de expresiones aritméticas

```
factor :: Parser Int
factor = do symbol "("
            e <- expr
            symbol ")"
            return e
         <|> natural
```

¿Por qué el parser es más eficiente al factorizar la gramática?

Evaluador de expresiones aritméticas

Definimos el evaluador usando el parser:

```
eval :: String -> Int
eval xs = fst (head (parse expr xs))
```

Por ejemplo:

```
> eval "2*3+4"
10
> eval "2*(3+4)"
14
```

Ejercicio

Extender el parser de expresiones según la siguiente extensión de la gramática:

$$
\begin{array}{lcl}
\mathit{exp} & ::= & \mathit{term}\ (\text{'+'}\ \mathit{exp} \mid \text{'-'}\ \mathit{exp} \mid \epsilon) \\
\mathit{term} & ::= & \mathit{atom}\ (\text{'*'}\ \mathit{term} \mid \text{'/'}\ \mathit{term} \mid \epsilon)
\end{array}
$$

Asociatividad de operadores

Para que el operador $-$ asocie a izquierda debemos modificar la gramática:

$$
\begin{array}{lcl}
\mathit{exp} & ::= & \mathit{exp}\ (\text{'+'}\ \mathit{term} \mid \text{'-'}\ \mathit{term} \mid \epsilon) \\
\mathit{term} & ::= & \mathit{term}\ (\text{'*'}\ \mathit{atom} \mid \text{'/'}\ \mathit{atom} \mid \epsilon)
\end{array}
$$

Ésta gramática presenta el problema de la recursión a izquierda: el parser correspondiente no terminará:

```
expr :: Parser Int
expr = do t <- expr
          char '+'
          e <- term
          return (t+e)
       <|> term
```

Eliminando recursión a izquierda

"Transfomamos la gramática en otra equivalente que no tenga recursión a izquierda."

Transformamos:

$$A \to A\alpha \mid \beta$$

donde α es no vacío y β no empieza con A, en la gramática:

$$A \to \beta A'$$
$$A' \to \epsilon \mid \alpha A'$$

Ejercicio

1. Usar la regla dada para eliminar la recursión a izquierda de la gramática de expresiones.

2. Definir el parser a partir de la nueva gramática.

Referencias

- *Programming in Haskell. G. Hutton. Cambridge University Press (2007). Capítulo 8*