Tema 12: Analizadores sintácticos funcionales

Informática (2018–19)

José A. Alonso Jiménez

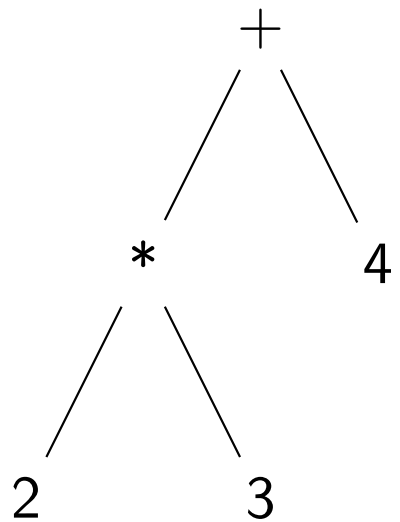
Grupo de Lógica Computacional
Departamento de Ciencias de la Computación e I.A.
Universidad de Sevilla

Tema 12: Analizadores sintácticos funcionales

1. Analizadores sintácticos
2. El tipo de los analizadores sintácticos
3. Analizadores sintácticos básicos
4. Composición de analizadores sintácticos
   - Secuenciación de analizadores sintácticos
   - Elección de analizadores sintácticos
5. Primitivas derivadas
6. Tratamiento de los espacios
7. Analizador de expresiones aritméticas

Analizadores sintácticos

- Un analizador sintáctico es un programa que analiza textos para determinar su estructura sintáctica.
- Ejemplo de análisis sintáctico aritmético: La estructura sintáctica de la cadena "2*3+4" es el árbol



- El análisis sintáctico forma parte del preprocesamiento en la mayoría de las aplicaciones reales.

Opciones para el tipo de los analizadores sintácticos

- Opción inicial:

```
type Analizador = String -> Tree
```

- Con la parte no analizada:

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

- Con todos los análisis:

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

- Con estructuras arbitrarias:

```
type Analizador a = String -> [(a,String)]
```

- Simplificación: analizadores que fallan o sólo dan un análisis.

Analizadores sintácticos básicos: resultado

- `(analiza a cs)` analiza la cadena `cs` mediante el analizador `a`. Por ejemplo,

```
analiza :: Analizador a -> String -> [(a,String)]
analiza a cs = a cs
```

- El analizador `resultado v` siempre tiene éxito, devuelve `v` y no consume nada. Por ejemplo,

```
ghci> analiza (resultado 1) "abc"
[(1,"abc")]
```

```
resultado :: a -> Analizador a
resultado v =  \xs -> [(v,xs)]
```

Analizadores sintácticos básicos: fallo

- El analizador `fallo` siempre falla. Por ejemplo,

```
ghci> analiza fallo "abc"
[]
```

```
fallo :: Analizador a
fallo =  \xs -> []
```

Analizadores sintácticos básicos: elemento

- El analizador `elemento` falla si la cadena es vacía y consume el primer elemento en caso contrario. Por ejemplo,

```
ghci> analiza elemento ""
[]
ghci> analiza elemento "abc"
[('a',"bc")]
```

```
elemento :: Analizador Char
elemento =  \xs -> case xs of
                     [] -> []
                     (x:xs) -> [(x , xs)]
```

Tema 12: Analizadores sintácticos funcionales

1. Analizadores sintácticos

2. El tipo de los analizadores sintácticos

3. Analizadores sintácticos básicos

4. Composición de analizadores sintácticos
   - Secuenciación de analizadores sintácticos
   - Elección de analizadores sintácticos

5. Primitivas derivadas

6. Tratamiento de los espacios

Secuenciación de analizadores sintácticos

- `((p >*> f) e)` falla si el análisis de `e` por `p` falla, en caso contrario, se obtiene un valor (`v`) y una salida (`s`), se aplica la función `f` al valor `v` obteniéndose un nuevo analizador con el que se analiza la salida `s`.

```
infixr 5 >*>
(>*>) :: Analizador a -> (a -> Analizador b) ->
         Analizador b
p >*> f = \ent -> case analiza p ent of
                    []        -> []
                    [(v,sal)] -> analiza (f v) sal
```

Secuenciación de analizadores sintácticos

- `primeroTercero` es un analizador que devuelve los caracteres primero y tercero de la cadena. Por ejemplo,

```
primeroTercero "abel"  ⇝  [(('a','e'),"l")]
primeroTercero "ab"    ⇝  []
```

```
primeroTercero :: Analizador (Char,Char)
primeroTercero =
    elemento >*> \x ->
    elemento >*> \_ ->
    elemento >*> \y ->
    resultado (x,y)
```

Tema 12: Analizadores sintácticos funcionales

1. Analizadores sintácticos

2. El tipo de los analizadores sintácticos

3. Analizadores sintácticos básicos

4. Composición de analizadores sintácticos
   - Secuenciación de analizadores sintácticos
   - Elección de analizadores sintácticos

5. Primitivas derivadas

6. Tratamiento de los espacios

Elección de analizadores sintácticos

- `((p +++ q) e)` analiza e con p y si falla analiza e con q. Por ejemplo,

```
Main*> analiza (elemento +++ resultado 'd') "abc"
[('a',"bc")]
Main*> analiza (fallo +++ resultado 'd') "abc"
[('d',"abc")]
Main*> analiza (fallo +++ fallo) "abc"
[]
```

```
(+++) :: Analizador a -> Analizador a -> Analizador a
p +++ q = \ent -> case analiza p ent of
                    []        -> analiza q ent
                    [(v,sal)] -> [(v,sal)]
```

Primitivas derivadas: sat

- `(sat p)` es el analizador que consume un elemento si dicho elemento cumple la propiedad `p` y falla en caso contrario. Por ejemplo,

```
analiza (sat isLower) "hola"  ⇝  [('h',"ola")]
analiza (sat isLower) "Hola"  ⇝  []
```

```
sat :: (Char -> Bool) -> Analizador Char
sat p = elemento >*> \x ->
        if p x then resultado x else fallo
```

Primitivas derivadas

- `digito` analiza si el primer carácter es un dígito. Por ejemplo,

```
analiza digito "123"  ⇝  [('1',"23")]
analiza digito "uno"  ⇝  []
```

```
digito :: Analizador Char
digito = sat isDigit
```

- `minuscula` analiza si el primer carácter es una letra minúscula. Por ejemplo,

```
analiza minuscula "eva"  ⇝  [('e',"va")]
analiza minuscula "Eva"  ⇝  []
```

```
minuscula :: Analizador Char
minuscula = sat isLower
```

Primitivas derivadas

- `mayuscula` analiza si el primer carácter es una letra mayúscula. Por ejemplo,

```
analiza mayuscula "Eva"  ⇝  [('E',"va")]
analiza mayuscula "eva"  ⇝  []
```

```
mayuscula :: Analizador Char
mayuscula = sat isUpper
```

- `letra` analiza si el primer carácter es una letra. Por ejemplo,

```
analiza letra "Eva"  ⇝  [('E',"va")]
analiza letra "eva"  ⇝  [('e',"va")]
analiza letra "123"  ⇝  []
```

```
letra :: Analizador Char
letra = sat isAlpha
```

Primitivas derivadas

- `alfanumerico` analiza si el primer carácter es una letra o un número. Por ejemplo,

```
analiza alfanumerico "Eva"   ⇝  [('E',"va")]
analiza alfanumerico "eva"   ⇝  [('e',"va")]
analiza alfanumerico "123"   ⇝  [('1',"23")]
analiza alfanumerico " 123"  ⇝  []
```

```
alfanumerico :: Analizador Char
alfanumerico = sat isAlphaNum
```

Primitivas derivadas

- `(caracter x)` analiza si el primer carácter es igual al carácter `x`.
  Por ejemplo,

```
analiza (caracter 'E') "Eva"  ⇝  [('E',"va")]
analiza (caracter 'E') "eva"  ⇝  []
```

```
caracter :: Char -> Analizador Char
caracter x = sat (== x)
```

Primitivas derivadas

- `(cadena c)` analiza si empieza con la cadena `c`. Por ejemplo,

```
analiza (cadena "abc") "abcdef"  ⇝  [("abc","def")]
analiza (cadena "abc") "abdcef"  ⇝  []
```

```
cadena :: String -> Analizador String
cadena []     = resultado []
cadena (x:xs) = caracter x >*> \x  ->
                cadena xs  >*> \xs ->
                resultado (x:xs)
```

Primitivas derivadas

- `varios p` aplica el analizador p cero o más veces. Por ejemplo,

```
analiza (varios digito) "235abc"  ⇝  [("235","abc")]
analiza (varios digito) "abc235"  ⇝  [("","abc235")]
```

```
varios :: Analizador a -> Analizador [a]
varios p  = varios1 p +++ resultado []
```

- `varios1 p` aplica el analizador p una o más veces. Por ejemplo,

```
analiza (varios1 digito) "235abc"  ⇝  [("235","abc")]
analiza (varios1 digito) "abc235"  ⇝  []
```

```
varios1 :: Analizador a -> Analizador [a]
varios1 p = p          >*> \v  ->
            varios p >*> \vs ->
            resultado (v:vs)
```

Primitivas derivadas

- `ident` analiza si comienza con un identificador (i.e. una cadena que comienza con una letra minúscula seguida por caracteres alfanuméricos). Por ejemplo,

```
Main*> analiza ident "lunes12 de Ene"
[("lunes12"," de Ene")]
Main*> analiza ident "Lunes12 de Ene"
[]
```

```
ident :: Analizador String
ident =  minuscula            >*> \x  ->
         varios alfanumerico  >*> \xs ->
         resultado (x:xs)
```

Primitivas derivadas

- `nat` analiza si comienza con un número natural. Por ejemplo,

```
analiza nat "14DeAbril"    ⇝  [(14,"DeAbril")]
analiza nat " 14DeAbril"   ⇝  []
```

```
nat :: Analizador Int
nat = varios1 digito >*> \xs ->
      resultado (read xs)
```

- `espacio` analiza si comienza con espacios en blanco. Por ejemplo,

```
analiza espacio "    a b c"  ⇝  [((),"a b c")]
```

```
espacio :: Analizador ()
espacio = varios (sat isSpace) >*> \_ ->
          resultado ()
```

Tratamiento de los espacios

- `unidad p` ignora los espacios en blanco y aplica el analizador `p`. Por ejemplo,

```
Main*> analiza (unidad nat) " 14DeAbril"
[(14,"DeAbril")]
Main*> analiza (unidad nat) " 14   DeAbril"
[(14,"DeAbril")]
```

```
unidad :: Analizador a -> Analizador a
unidad p = espacio >*> \_ ->
           p       >*> \v ->
           espacio >*> \_ ->
           resultado v
```

Tratamiento de los espacios

- `identificador` analiza un identificador ignorando los espacios delante y detrás. Por ejemplo,

```
Main*> analiza identificador "  lunes12  de Ene"
[("lunes12","de Ene")]
```

```
identificador :: Analizador String
identificador = unidad ident
```

- `natural` analiza un número natural ignorando los espacios delante y detrás. Por ejemplo,

```
analiza natural "  14DeAbril"  ⇝  [(14,"DeAbril")]
```

```
natural :: Analizador Int
natural =  unidad nat
```

Tratamiento de los espacios

- `(simbolo xs)` analiza la cadena `xs` ignorando los espacios delante y detrás. Por ejemplo,

```
Main*> analiza (simbolo "abc") "  abcdef"
[("abc","def")]
```

```
simbolo :: String -> Analizador String
simbolo xs =  unidad (cadena xs)
```

Tratamiento de los espacios

- `listaNat` analiza una lista de naturales ignorando los espacios. Por ejemplo,

```
Main*> analiza listaNat " [  2,  3, 5   ]"
[([2,3,5],"")]
Main*> analiza listaNat " [  2,  3,]"
[]
```

```
listaNat :: Analizador [Int]
listaNat = simbolo "["           >*> \_ ->
           natural               >*> \n ->
           varios (simbolo ","   >*> \_ ->
                   natural)      >*> \ns ->
           simbolo "]"           >*> \_ ->
           resultado (n:ns)
```

Expresiones aritméticas

- Consideramos expresiones aritméticas:
  - construidas con números, operaciones ($+$ y $*$) y paréntesis.
  - $+$ y $*$ asocian por la derecha.
  - $*$ tiene más prioridad que $+$.
- Ejemplos:
  - $2 + 3 + 5$ representa a $2 + (3 + 5)$.
  - $2 * 3 + 5$ representa a $(2 * 3) + 5$.

Gramáticas de las expresiones aritméticas: Gramática 1

- Gramática 1 de las expresiones aritméticas:

$$\begin{array}{lcl} \mathit{expr} & ::= & \mathit{expr} + \mathit{expr} \mid \mathit{expr} * \mathit{expr} \mid (\mathit{expr}) \mid \mathit{nat} \\ \mathit{nat} & ::= & 0 \mid 1 \mid 2 \mid \ldots \end{array}$$

- La gramática 1 no considera prioridad:
  acepta $2 + 3 * 5$ como $(2 + 3) * 5$ y como $2 + (3 * 5)$
- La gramática 1 no considera asociatividad:
  acepta $2 + 3 + 5$ como $(2 + 3) + 5$ y como $2 + (3 + 5)$
- La gramática 1 es ambigua.

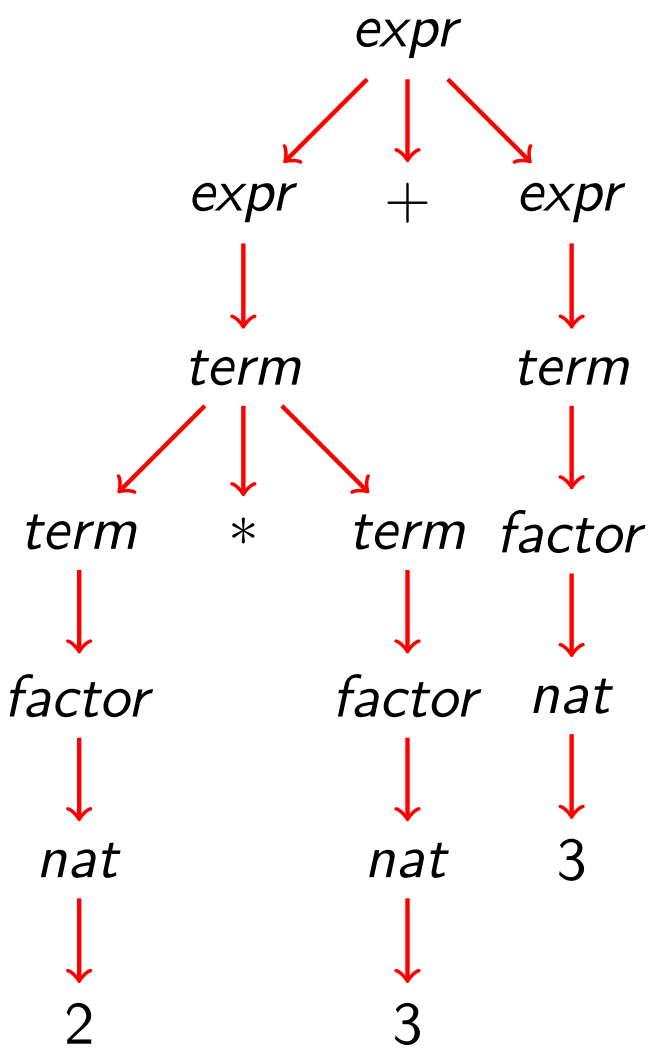
Gramáticas de las expresiones aritméticas: Gramática 2

- Gramática 2 de las expresiones aritméticas (con prioridad):

$$\begin{array}{lcl} \mathit{expr} & ::= & \mathit{expr} + \mathit{expr} \mid \mathit{term} \\ \mathit{term} & ::= & \mathit{term} * \mathit{term} \mid \mathit{factor} \\ \mathit{factor} & ::= & (\mathit{expr}) \mid \mathit{nat} \\ \mathit{nat} & ::= & 0 \mid 1 \mid 2 \mid \ldots \end{array}$$

- La gramática 2 sí considera prioridad:
  acepta $2 + 3 * 5$ sólo como $2 + (3 * 5)$
- La gramática 2 no considera asociatividad:
  acepta $2 + 3 + 5$ como $(2 + 3) + 5$ y como $2 + (3 + 5)$
- La gramática 2 es ambigua.

Árbol de análisis sintáctico de $2 * 3 + 5$ con la gramática 2



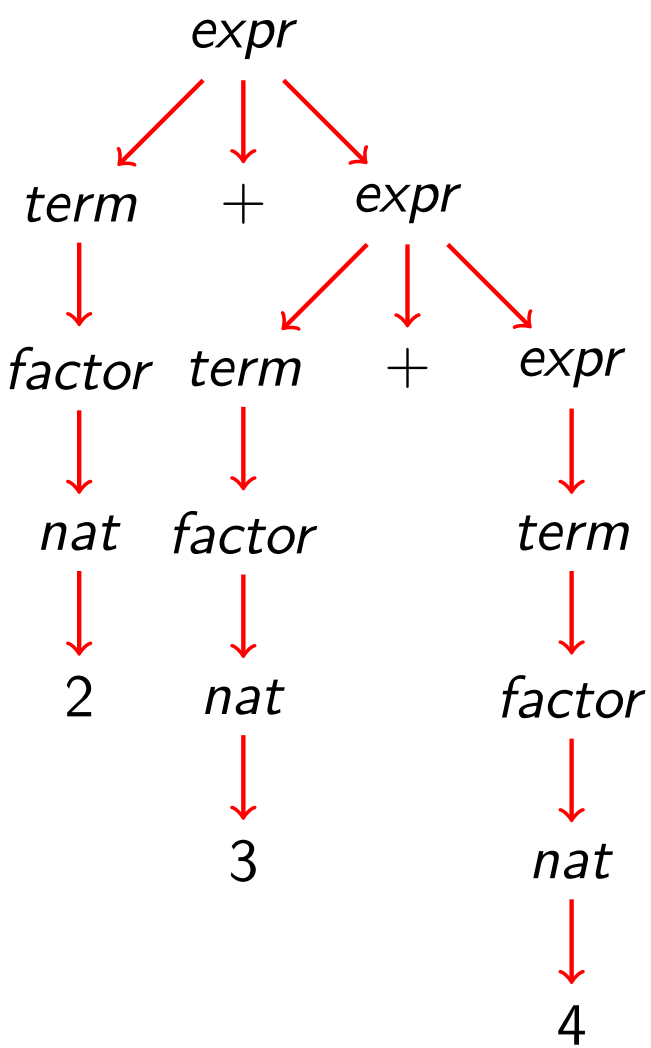
Gramáticas de las expresiones aritméticas: Gramática 3

- Gramática 3 de las expresiones aritméticas:

$$\begin{array}{lcl} \mathit{expr} & ::= & \mathit{term} + \mathit{expr} \mid \mathit{term} \\ \mathit{term} & ::= & \mathit{factor} * \mathit{term} \mid \mathit{factor} \\ \mathit{factor} & ::= & (\mathit{expr}) \mid \mathit{nat} \\ \mathit{nat} & ::= & 0 \mid 1 \mid 2 \mid \ldots \end{array}$$

- La gramática 3 sí considera prioridad:
  acepta $2 + 3 * 5$ sólo como $2 + (3 * 5)$
- La gramática 3 sí considera asociatividad:
  acepta $2 + 3 + 5$ como $2 + (3 + 5)$
- La gramática 3 no es ambigua (i.e. es libre de contexto).

Árbol de análisis sintáctico de $2 + 3 + 5$ con la gramática 3



Gramáticas de las expresiones aritméticas: Gramática 4

- La gramática 4 se obtiene simplificando la gramática 3:

$$\begin{array}{lcl} \mathit{expr} & ::= & \mathit{term}\ (+\ \mathit{expr} \mid \epsilon) \\ \mathit{term} & ::= & \mathit{factor}\ (*\ \mathit{term} \mid \epsilon) \\ \mathit{factor} & ::= & (\mathit{expr}) \mid \mathit{nat} \\ \mathit{nat} & ::= & 0 \mid 1 \mid 2 \mid \ldots \end{array}$$

  donde ϵ es la cadena vacía.
- La gramática 4 no es ambigua.
- La gramática 4 es la que se usará para escribir el analizador de expresiones aritméticas.

Analizador de expresiones aritméticas

- `expr` analiza una expresión aritmética devolviendo su valor. Por ejemplo,

```
analiza expr "2*3+5"     ⇝  [(11,"")]
analiza expr "2*(3+5)"   ⇝  [(16,"")]
analiza expr "2+3*5"     ⇝  [(17,"")]
analiza expr "2*3+5abc"  ⇝  [(11,"abc")]
```

```
expr :: Analizador Int
expr = term                 >*> \t ->
       (simbolo "+"         >*> \_ ->
        expr                >*> \e ->
        resultado (t+e))
       +++ resultado t
```

Analizador de expresiones aritméticas

- averbterm analiza un término de una expresión aritmética devolviendo su valor. Por ejemplo,

```
analiza term "2*3+5"      ⇝  [(6,"+5")]
analiza term "2+3*5"      ⇝  [(2,"+3*5")]
analiza term "(2+3)*5+7"  ⇝  [(25,"+7")]
```

```
term :: Analizador Int
term =  factor              >*> \f ->
        (simbolo "*"        >*> \_ ->
         term               >*> \t ->
         resultado (f*t))
        +++ resultado f
```

Analizador de expresiones aritméticas

- `factor` analiza un factor de una expresión aritmética devolviendo su valor. Por ejemplo,

```
analiza factor "2*3+5"      ⇝  [(2,"*3+5")]
analiza factor "(2+3)*5"    ⇝  [(5,"*5")]
analiza factor "(2+3*7)*5"  ⇝  [(23,"*5")]
```

```
factor :: Analizador Int
factor =  (simbolo "("  >*> \_ ->
           expr         >*> \e ->
           simbolo ")"  >*> \_ ->
           resultado e)
          +++ natural
```

Analizador de expresiones aritméticas

- `(valor cs)` analiza la cadena `cs` devolviendo su valor si es una expresión aritmética y un mensaje de error en caso contrario. Por ejemplo,

```
valor "2*3+5"       ⇝  11
valor "2*(3+5)"     ⇝  16
valor "2 * 3 + 5"   ⇝  11
valor "2*3x"        ⇝  *** Exception: sin usar x
valor "-1"          ⇝  *** Exception: entrada no valida
```

```
valor :: String -> Int
valor xs = case (analiza expr xs) of
             [(n,[])]  -> n
             [(_,sal)] -> error ("sin usar " ++ sal)
             []        -> error "entrada no valida"
```

Bibliografía

1. R. Bird. *Introducción a la programación funcional con Haskell*. Prentice Hall, 2000.
   - Cap. 11: Análisis sintáctico.
2. G. Hutton *Programming in Haskell*. Cambridge University Press, 2007.
   - Cap. 8: Functional parsers.
3. G. Hutton y E. Meijer. Monadic Parser Combinators. Technical Report NOTTCS–TR–96–4, Department of Computer Science, University of Nottingham, 1996.
4. G. Hutton y E. Meijer. Monadic Parsing in Haskell. Journal of Functional Programming, 8(4): 437—444, 1998.
5. B.C. Ruiz, F. Gutiérrez, P. Guerrero y J.E. Gallardo. *Razonando con Haskell*. Thompson, 2004.
   - Cap. 14: Analizadores.