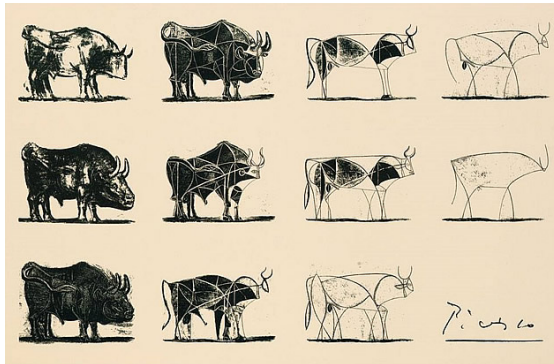


Analizadores Sintácticos - Parsers

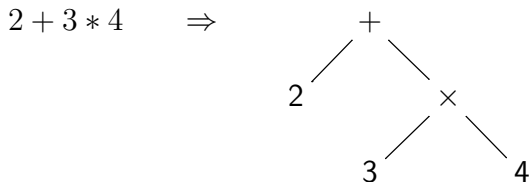
Análisis de Lenguajes de Programación

Mauro Jaskelioff 16/08/2019



¿Qué es un parser?

Un **parser** es un programa que analiza un texto para determinar su **estructura sintáctica**.



¿Dónde se usan?

Casi cualquier programa usa algún parser para **pre-procesar** sus entradas.

GHC	⇒	programas Haskell
Bash	⇒	scripts
Mozilla	⇒	documentos HTML

El Tipo de los Parsers

En un lenguaje funcional como Haskell, los parsers pueden ser vistos naturalmente como **funciones**.

type *Parser* = *String* \rightarrow *Tree*

Un parser es una función que toma una cadena y devuelve algún tipo de árbol.

El Tipo de los Parsers (cont.)

- ▶ Sin embargo, puede suceder que un parser no requiera toda la cadena de entrada, así que también devolvemos la **entrada no usada**:

type *Parser* = *String* \rightarrow (*Tree*, *String*)

- ▶ Una cadena podría ser parseable de varias maneras (o de ninguna) así que generalizamos la respuesta a una **lista de resultados**:

type *Parser* = *String* \rightarrow [(*Tree*, *String*)]

El Tipo de los Parsers (cont.)

- ▶ Por último, un parser podría devolver algo que no es un árbol, así que generalizamos a un valor de cualquier tipo:

type *Parser* *a* = *String* \rightarrow [(*a*, *String*)]

- ▶ Nota: Por simplicidad, consideraremos sólo parsers que, o bien fallan y devuelven la lista vacía, o bien tienen éxito y devuelven una lista con un sólo elemento.

Parsers Básicos

- El parser *item* falla si su entrada es vacía, y consume el primer caracter en otro caso:

item :: *Parser Char*

item = $\lambda inp \rightarrow$ **case** *inp* **of**

 [] \rightarrow []

 (*x* : *xs*) \rightarrow [(*x*, *xs*)]

Parsers Básicos (cont.)

- El parser *failure* siempre falla:

$$\begin{aligned} failure &:: \text{Parser } a \\ failure &= \lambda inp \rightarrow [] \end{aligned}$$

- El parser *return v* siempre tiene éxito. Devuelve el valor *v* sin consumir entrada:

$$\begin{aligned} return &:: a \rightarrow \text{Parser } a \\ return\ v &= \lambda inp \rightarrow [(v, inp)] \end{aligned}$$

Parsers Básicos (cont.)

- El parser $p \langle | \rangle q$ se comporta como el parser p si éste tiene éxito, y como el parser q en otro caso:

$$\begin{aligned} (\langle | \rangle) &:: \text{Parser } a \rightarrow \text{Parser } a \rightarrow \text{Parser } a \\ p \langle | \rangle q &= \lambda inp \rightarrow \text{case } p \text{ inp of} \\ &\quad [] \quad \quad \quad \rightarrow \text{parse } q \text{ inp} \\ &\quad [(v, out)] \rightarrow [(v, out)] \end{aligned}$$

- La función *parse* aplica un parser a una cadena:

$$\begin{aligned} \text{parse} &:: \text{Parser } a \rightarrow \text{String} \rightarrow [(a, \text{String})] \\ \text{parse } p \text{ inp} &= p \text{ inp} \end{aligned}$$

Ejemplos

- El comportamiento de las cinco primitivas puede ser ilustrado con algunos ejemplos simples:

```
% ghci Parsing
```

```
> parse item ""
```

```
[]
```

```
> parse item "abc"
```

```
[('a', "bc")]
```

Ejemplos (cont.)

```
> parse failure "abc"
```

```
[]
```

```
> parse (return 1) "abc"
```

```
[(1, "abc")]
```

```
> parse (item <|> return 'd') "abc"
```

```
[('a', "bc")]
```

```
> parse (failure <|> return 'd') "abc"
```

```
[('d', "abc")]
```

- ▶ La biblioteca *Parsing* está disponible en la página web de la materia (usarla en la práctica).
- ▶ El código es un poco más complejo porque en esta presentación se han hecho algunas simplificaciones.
- ▶ El tipo de *Parser* es una mónada, una estructura matemática que ha resultado útil para modelar diferentes clases de computaciones.

Secuenciamiento

- ▶ Una secuencia de parsers puede ser combinada en un parser compuesto usando la palabra clave **do**.
- ▶ Por ejemplo:

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

- ▶ Cada parser debe comenzar en la misma columna. Es decir, se aplica la **layout rule**.
- ▶ Los valores devueltos por los parsers intermedios son descartados por defecto, pero si se necesitan pueden ser nombrados usando el operador \leftarrow .
- ▶ El valor devuelto por el último parser es el valor que devolverá toda la secuencia.

Comentarios (cont.)

- ▶ Si algún parser en una secuencia falla, toda la secuencia falla. Por ejemplo:

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

- ▶ La notación **do** no es específica del tipo *Parser*, sino que puede ser usada por cualquier tipo monádico.

Primitivas Derivadas

- Parsear un caracter que **satisfice** un predicado:

```
sat    :: (Char → Bool) → Parser Char  
sat p = do x ← item  
          if p x  
          then return x  
          else failure
```


Primitivas Derivadas (cont.)

- Parsear un **dígito** y **caracteres** específicos:

digit :: *Parser Char*

digit = *sat isDigit*

char :: *Char* → *Parser Char*

char *x* = *sat (x ≡)*

- Parsear una **cadena** de caracteres específica:

string :: *String* → *Parser String*

string [] = *return []*

string (*x* : *xs*) = **do** *char* *x*

string *xs*

return (*x* : *xs*)

Primitivas Derivadas (cont.)

- ▶ Aplicar un parser **cero o más veces**:

$$\begin{aligned} \text{many} &:: \text{Parser } a \rightarrow \text{Parser } [a] \\ \text{many } p &= \text{many1 } p \langle | \rangle \text{return } [] \end{aligned}$$

- ▶ Aplicar un parser **una o más veces**:

$$\begin{aligned} \text{many1} &:: \text{Parser } a \rightarrow \text{Parser } [a] \\ \text{many1 } p &= \text{do } v \leftarrow p \\ &\quad vs \leftarrow \text{many } p \\ &\quad \text{return } (v : vs) \end{aligned}$$

Ejemplo

- Definimos un parser que consume una lista de uno o más dígitos a partir de una cadena:

```
p :: Parser String  
p = do char '['  
      d ← digit  
      ds ← many (do char ','  
                     digit)  
      char ']'  
      return (d : ds)
```

Ejemplo (cont.)

- ▶ Por ejemplo

```
> parse p "[1,2,3,4]"  
[("1234", "")]  
  
> parse p "[1,2,3,4"  
[]
```

- ▶ Nota: Las bibliotecas de parsing más sofisticadas pueden indicar y/o recuperarse de errores en la cadena de entrada.

Expresiones Aritméticas

- ▶ Considere una forma simple de expresiones construidas a partir de dígitos usando las operaciones de suma $+$ y multiplicación $*$, además de paréntesis.
- ▶ También suponemos que:
 - ▶ $*$ y $+$ asocian a la derecha;
 - ▶ $*$ tiene prioridad más alta que $+$.

Expresiones Aritméticas (cont.)

- Formalmente, la sintaxis de dichas expresiones se define con la siguiente **gramática libre de contexto**:

$$expr \rightarrow term \text{ '+' } expr \mid term$$
$$term \rightarrow factor \text{ '*' } term \mid factor$$
$$factor \rightarrow digit \mid \text{'(' } expr \text{ ') '}$$
$$digit \rightarrow \text{'0' } \mid \text{'1' } \mid \dots \mid \text{'9'}$$

Expresiones Aritméticas (cont.)

- ▶ Sin embargo, por razones de eficiencia, es importante factorizar las reglas para *expr* y *term*:

$$expr \rightarrow term \ ('+' \ expr \mid \varepsilon)$$

$$term \rightarrow factor \ ('*' \ term \mid \varepsilon)$$

- ▶ Nota: El símbolo ε denota la cadena vacía.

Implementando la gramática

- ▶ Es fácil traducir la gramática a un parser que evalúa expresiones, simplemente reescribiendo las reglas de la gramática usando las primitivas de parseo.
- ▶ Es decir:

```
expr :: Parser Int  
expr = do t ← term  
         (do char '+'  
            e ← expr  
            return (t + e)  
         ⟨|⟩ return t)
```


Implementando la gramática (cont.)

term :: *Parser Int*

```
term = do f ← factor  
      (do char '*'  
        t ← term  
        return (f * t)  
      <|> return f)
```

factor :: *Parser Int*

```
factor = do d ← digit  
          return (digitToInt d)  
      <|> do char '('  
            e ← expr  
            char ')'  
            return e
```

Implementando el parser

- ▶ Por último, definimos:

$$\begin{aligned} eval &:: String \rightarrow Int \\ eval\ xs &= fst\ (head\ (parse\ expr\ xs)) \end{aligned}$$

- ▶ Probamos algunos ejemplos:

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

Ejercicios

1. ¿Por qué factorizar la gramática de expresiones hace que el parser resultante sea más eficiente?
2. Extender el parser de expresiones para permitir el uso de la resta y la división, basándose en la siguiente extensión de la gramática:

$$expr \rightarrow term \ ('+' \ expr \mid '-' \ expr \mid \varepsilon)$$
$$term \rightarrow factor \ ('*' \ term \mid '/' \ term \mid \varepsilon)$$

Asociatividad de Operadores

- ▶ El parser inducido por la gramática del ejercicio anterior asocia a derecha
 - ▶ $9 - 4 - 3$ es parseado como $(9 - (4 - 3))$
 - ▶ Por lo tanto $9 - 4 - 3$ evalúa a 8 y no a 2
- ▶ Para que asocie a izquierda hay que modificar la gramática.

Cambiando la asociatividad

- Probamos con la siguiente gramática

$$expr \rightarrow expr \, ('+' \, term \mid '-' \, term) \mid term$$
$$term \rightarrow term \, ('*' \, factor \mid '/' \, factor) \mid factor$$
$$factor \rightarrow digit \mid '(' \, expr \, ')'$$
$$digit \rightarrow '0' \mid '1' \mid \dots \mid '9'$$

Recursión a Izquierda

- ▶ La gramática presenta el problema de la recursión a izquierda

$$expr \rightarrow \textcolor{red}{expr} \text{'+' } term \mid term$$

- ▶ El parser correspondiente sería:

```
expr :: Parser Int
expr = do t ← 
```

Transformando la gramática

- ▶ Podemos transformar la gramática en otra que no tenga recursión a izquierda.
- ▶ Dada una gramática

$$A \rightarrow A \alpha \mid \beta$$

tal que α es no vacío y β no empieza con A ,

- ▶ podemos transformarla a la gramática

$$A \rightarrow \beta A'$$

$$A' \rightarrow \varepsilon \mid \alpha A'$$

Ejemplo

- ▶ Dada la gramática con recursión a izquierda:

$$e \rightarrow e \text{ 'entonces' } e \mid \text{'esto'}$$

- ▶ la transformamos a

$$e \rightarrow \text{'esto' } e'$$

$$e' \rightarrow \varepsilon \mid \text{'entonces' } e e'$$

Ejercicio

- ▶ Transformar la gramática para eliminar la recursión a izquierda

$$expr \rightarrow expr \text{ ' + ' } term \mid \text{ ' - ' } term \mid term$$
$$term \rightarrow term \text{ ' * ' } factor \mid \text{ ' / ' } factor \mid factor$$
$$factor \rightarrow digit \mid \text{ ' (' } expr \text{ ') '}$$
$$digit \rightarrow \text{ ' 0 ' } \mid \text{ ' 1 ' } \mid \dots \mid \text{ ' 9 '}$$

- ▶ Un parser analiza una cadena de texto y nos da una sintaxis abstracta.
- ▶ Podemos implementar parsers en forma simple a partir de algunos combinadores básicos.
- ▶ A menudo es necesario transformar una gramática para optimizarla o eliminar recursión izquierda.

Referencias

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