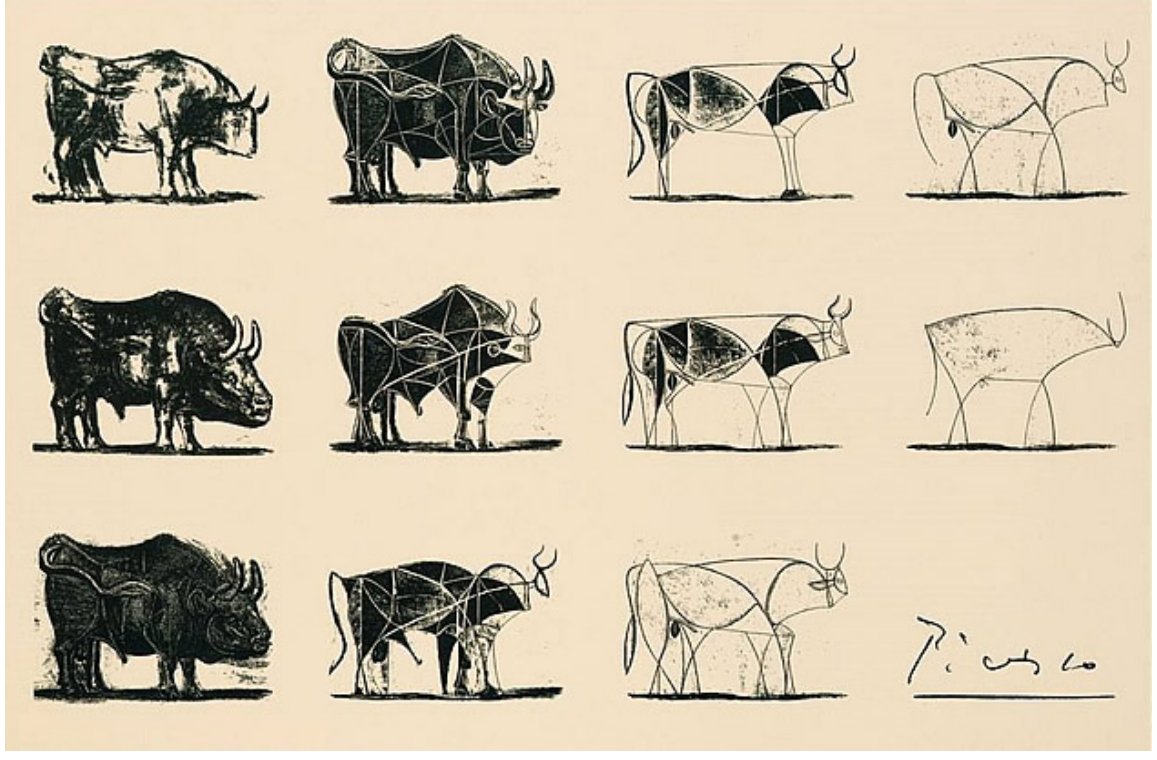
Analizadores Sintácticos - Parsers

Análisis de Lenguajes de Programación

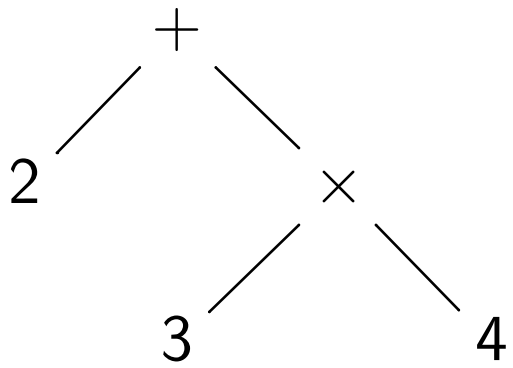
Mauro Jaskelioff 18/08/2017



¿Qué es un parser?

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

$2 + 3 * 4 \quad \Rightarrow$



¿Dónde se usan?

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

GHC	$\Rightarrow$	programas Haskell
Bash	$\Rightarrow$	scripts
Mozilla	$\Rightarrow$	documentos HTML

El Tipo de los Parsers

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

$$\textbf{type } \mathit{Parser} = \mathit{String} \rightarrow \mathit{Tree}$$

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**:

$$\textbf{type } \mathit{Parser} = \mathit{String} \to (\mathit{Tree}, \mathit{String})$$

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

$$\textbf{type } \mathit{Parser} = \mathit{String} \to [(\mathit{Tree}, \mathit{String})]$$

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:

$$\textbf{type}\ \mathit{Parser}\ a = \mathit{String} \rightarrow [(a, \mathit{String})]$$

- 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:

$$
\begin{aligned}
&\mathit{item} :: \mathit{Parser}\ \mathit{Char} \\
&\mathit{item} = \lambda \mathit{inp} \rightarrow \textbf{case}\ \mathit{inp}\ \textbf{of} \\
&\qquad\qquad [\,] \qquad\ \rightarrow [\,] \\
&\qquad\qquad (x : xs) \rightarrow [(x, xs)]
\end{aligned}
$$

Parsers Básicos (cont.)

- El parser *failure* siempre falla:

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

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

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

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:

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

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

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

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")]
```

Nota

- 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:

$$
\begin{aligned}
&p :: \mathit{Parser}\ (\mathit{Char}, \mathit{Char}) \\
&p = \mathbf{do}\ x \leftarrow \mathit{item} \\
&\qquad\quad \mathit{item} \\
&\qquad\quad y \leftarrow \mathit{item} \\
&\qquad\quad \mathit{return}\ (x, y)
\end{aligned}
$$

Comentarios

- 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 **satisface** un predicado:

$$
\begin{aligned}
&\mathit{sat} \quad :: (\mathit{Char} \to \mathit{Bool}) \to \mathit{Parser}\ \mathit{Char} \\
&\mathit{sat}\ p = \mathbf{do}\ x \leftarrow \mathit{item} \\
&\qquad\qquad \texttt{if}\ p\ x \\
&\qquad\qquad\quad \texttt{then}\ \mathit{return}\ x \\
&\qquad\qquad\quad \texttt{else}\ \mathit{failure}
\end{aligned}
$$

Primitivas Derivadas (cont.)

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

$$
\begin{aligned}
&\mathit{digit} :: \mathit{Parser}\ \mathit{Char}\\
&\mathit{digit} = \mathit{sat}\ \mathit{isDigit}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{char} \quad :: \mathit{Char} \to \mathit{Parser}\ \mathit{Char}\\
&\mathit{char}\ x = \mathit{sat}\ (x \equiv)
\end{aligned}
$$

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

$$
\begin{aligned}
&\mathit{string} && :: \mathit{String} \to \mathit{Parser}\ \mathit{String}\\
&\mathit{string}\ [\,] && = \mathit{return}\ [\,]\\
&\mathit{string}\ (x : xs) && = \mathbf{do}\ \mathit{char}\ x\\
&&&\qquad \mathit{string}\ xs\\
&&&\qquad \mathit{return}\ (x : xs)
\end{aligned}
$$

Primitivas Derivadas (cont.)

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

$$
\begin{aligned}
&\mathit{many} \quad :: \mathit{Parser}\ a \to \mathit{Parser}\ [a] \\
&\mathit{many}\ p = \mathit{many1}\ p \langle|\rangle \mathit{return}\ [\,]
\end{aligned}
$$

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

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

Ejemplo

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

$$
\begin{aligned}
&p :: \mathit{Parser\ String}\\
&p = \mathbf{do}\ \mathit{char}\ \text{'['}\\
&\qquad\quad \mathit{d} \leftarrow \mathit{digit}\\
&\qquad\quad \mathit{ds} \leftarrow \mathit{many}\ (\mathbf{do}\ \mathit{char}\ \text{','}\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathit{digit})\\
&\qquad\quad \mathit{char}\ \text{']'}\\
&\qquad\quad \mathit{return}\ (\mathit{d} : \mathit{ds})
\end{aligned}
$$

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**:

$$
\begin{aligned}
\textit{expr} &\rightarrow \textit{term}\ \texttt{'+'}\ \textit{expr} \mid \textit{term} \\
\textit{term} &\rightarrow \textit{factor}\ \texttt{'*'}\ \textit{term} \mid \textit{factor} \\
\textit{factor} &\rightarrow \textit{digit} \mid \texttt{'('}\ \textit{expr}\ \texttt{')'} \\
\textit{digit} &\rightarrow \texttt{'0'} \mid \texttt{'1'} \mid \cdots \mid \texttt{'9'}
\end{aligned}
$$

Expresiones Aritméticas (cont.)

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

$$expr \rightarrow term\ (\text{'+'}\ expr \mid \varepsilon)$$
$$term \rightarrow factor\ (\text{'*'}\ term \mid \varepsilon)$$

- 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:

$$
\begin{aligned}
&\mathit{expr} :: \mathit{Parser}\ \mathit{Int}\\
&\mathit{expr} = \mathbf{do}\ t \leftarrow \mathit{term}\\
&\qquad\qquad (\mathbf{do}\ \mathit{char}\ \text{'+'}\\
&\qquad\qquad\quad e \leftarrow \mathit{expr}\\
&\qquad\qquad\quad \mathit{return}\ (t + e)\\
&\qquad\qquad \langle|\rangle\ \mathit{return}\ t)
\end{aligned}
$$

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:

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

- Probamos algunos ejemplos:

$$\begin{aligned} &> \mathit{eval}\ \texttt{"2*3+4"} \\ &10 \\ &> \mathit{eval}\ \texttt{"2*(3+4)"} \\ &14 \end{aligned}$$

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:

$$expr \rightarrow term \quad (\text{'+'}\ expr \mid \text{'-'}\ expr \mid \varepsilon)$$
$$term \rightarrow factor\ (\text{'*'}\ term \mid \text{'/'}\ term \mid \varepsilon)$$

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

$$
\begin{aligned}
\mathit{expr} &\rightarrow \mathit{expr}\ (\texttt{'+'}\ \mathit{term} \mid \texttt{'-'}\ \mathit{term}) \mid \mathit{term} \\
\mathit{term} &\rightarrow \mathit{term}\ (\texttt{'*'}\ \mathit{factor} \mid \texttt{'/'}\ \mathit{factor}) \mid \mathit{factor} \\
\mathit{factor} &\rightarrow \mathit{digit} \mid \texttt{'('}\ \mathit{expr}\ \texttt{')'} \\
\mathit{digit} &\rightarrow \texttt{'0'} \mid \texttt{'1'} \mid \cdots \mid \texttt{'9'}
\end{aligned}
$$

Recursión a Izquierda

- La gramática presenta el problema de la recursión a izquierda

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

- El parser correspondiente sería:

$$\begin{aligned} &expr :: Parser\ Int \\ &expr = \mathbf{do}\ t \leftarrow expr \\ &\qquad\qquad\ \ char\ \text{'+'} \\ &\qquad\qquad\ \ e \leftarrow term \\ &\qquad\qquad\ \ return\ (t + e) \\ &\qquad\qquad \langle|\rangle\ term \end{aligned}$$

Transformando la gramática

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

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

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

- podemos transformarla a la gramática

$$\begin{aligned} A &\to \beta\ A' \\ A' &\to \varepsilon \mid \alpha\ A' \end{aligned}$$

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

$$
\begin{aligned}
\mathit{expr} &\rightarrow \mathit{expr}\ (\texttt{'+'}\ \mathit{term} \mid \texttt{'-'}\ \mathit{term}) \mid \mathit{term} \\
\mathit{term} &\rightarrow \mathit{term}\ (\texttt{'*'}\ \mathit{factor} \mid \texttt{'/'}\ \mathit{factor}) \mid \mathit{factor} \\
\mathit{factor} &\rightarrow \mathit{digit} \mid \texttt{'('}\ \mathit{expr}\ \texttt{')'} \\
\mathit{digit} &\rightarrow \texttt{'0'} \mid \texttt{'1'} \mid \cdots \mid \texttt{'9'}
\end{aligned}
$$

Resumen

- 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.