Paradigmas de Programación

Práctica 10

Expresiones Regulares

Una expresión regular es una secuencia de caracteres que define un patrón de búsqueda sobre cadenas de símbolos. Para un alfabeto de símbolos Σ las siguientes constantes son expresiones regulares:

$\emptyset$	vacío (no encaja con ninguna cadena)
ε	cadena vacía
a	$a \in \Sigma$

Y dadas dos expresiones regulares r y s, las siguientes operaciones sobre ellas definen nuevas expresiones regulares:

$r \cdot s$	concatenación
r^*	repetición 0 o más veces
$r + s$	or lógico (alternativa)
$r \,\&\, s$	and lógico

Por ejemplo, a^*b encaja con las cadenas b, ab, aab, etc. Para simplificar las expresiones regulares con las que trabajamos vamos a considerar tres casos adicionales:

$\hat{}c$	cualquier símbolo distinto de c
.	cualquier símbolo
$a - b$	cualquier símbolo entre a y b

que serían expresables en términos de los casos anteriores como el or de todos los símbolos de Σ excepto c para $\hat{}c$, el or de todos los símbolos de Σ para $'.'$ y el or de todos los símbolos en el rango para $a - b$.

El estándar Posix especifica una sintaxis para expresiones regulares que utilizan muchas aplicaciones unix. En esta práctica vamos a trabajar con un subconjunto de esta sintaxis:

a	carácter a
.	cualquier carácter
[ ]	cualquiera de los caracteres entre los corchetes. $[abc]$ encaja con a, b o c $[a - z0 - 9]$ encaja con cualquier número o caracter en minúscula
[$\hat{}$]	cualquier caracter excepto los especificados. $[\hat{}ab]$ encaja con cualquier carácter excepto a y b
$*$	0 o más repeticiones
()	agrupa subexpresiones. $(ab)*$ encaja con ε, ab, $abab$, ...
+	1 o más repeticiones. $a+$ equivale a $aa*$
?	0 o 1 veces
|	alternativa. $abc\|def$ encaja con abc o con def

Por ejemplo, grep imprime las lineas de un fichero que encajen con una expresión regular:

```
$ grep -E "let" src.ml
  Lineas con definiciones en un fichero .ml
$ grep -E "let ([^r]|r[^e]|re[^c])" src.ml
  Lineas con definiciones no recursivas
$ grep -E "let.*in"
  Lineas con un let ... in
```

Derivada de una expresión regular

Para calcular si una cadena de texto encaja con una expresión regular vamos a utilizar derivadas de expresiones regulares.

La derivada de un conjunto de cadenas S con respecto a un símbolo a es el conjunto de cadenas de S que empiezan por a sin el símbolo a inicial. Para el conjunto de cadenas aceptadas por una expresión regular, el conjunto de cadenas derivadas repecto a un símbolo es expresable también como expresión regular [1, 2].

Para calcular la derivada de una expresión regular r respecto a un caracter c definimos dos funciones. La función $\nu(r)$, cuyo valor es:

$$\nu(r) = \begin{cases} \varepsilon & \text{si } r \text{ acepta la cadena vacía} \\ \emptyset & \text{si no la acepta} \end{cases}$$

y se define como:

$$\begin{aligned}
\nu(\emptyset) &= \emptyset \\
\nu(\varepsilon) &= \varepsilon \\
\nu(a) &= \emptyset \\
\nu(.) &= \emptyset \\
\nu(\hat{}a) &= \emptyset \\
\nu(a-b) &= \emptyset \\
\nu(r \cdot s) &= \begin{cases} \varepsilon & \text{si } \nu(r) = \varepsilon \text{ y } \nu(s) = \varepsilon \\ \emptyset & \text{en otro caso} \end{cases} \\
\nu(r^*) &= \varepsilon \\
\nu(r+s) &= \begin{cases} \varepsilon & \text{si } \nu(r) = \varepsilon \text{ o } \nu(s) = \varepsilon \\ \emptyset & \text{en otro caso} \end{cases} \\
\nu(r \,\&\, s) &= \begin{cases} \varepsilon & \text{si } \nu(r) = \varepsilon \text{ y } \nu(s) = \varepsilon \\ \emptyset & \text{en otro caso} \end{cases}
\end{aligned}$$

Y la función que calcula la derivada de una expresión regular r con respecto a un carácter a, $\partial_a(r)$:

$$
\begin{array}{rcll}
\partial_a(\emptyset) & = & \emptyset & \\
\partial_a(\varepsilon) & = & \emptyset & \\
\partial_a(a) & = & \varepsilon & \\
\partial_a(c) & = & \emptyset & \text{si } c \neq a \\
\partial_a(\hat{}a) & = & \emptyset & \\
\partial_a(\hat{}c) & = & \varepsilon & \text{si } c \neq a \\
\partial_a(c-d) & = & \varepsilon & \text{si } c \leq a \leq d \\
\partial_a(c-d) & = & \emptyset & \text{si } a < c \text{ o } a > d \\
\partial_a(.) & = & \varepsilon & \\
\partial_a(r \cdot s) & = & \partial_a(r) \cdot s + \nu(r) \cdot \partial_a(s) & \\
\partial_a(r^*) & = & \partial_a(r) \cdot r^* & \\
\partial_a(r + s) & = & \partial_a(r) + \partial_a(s) & \\
\partial_a(r \,\&\, s) & = & \partial_a(r) \,\&\, \partial_a(s) &
\end{array}
$$

Una cadena formada por $c_1 c_2 \ldots c_n$ encaja con una expresión regular r si la derivada de r para toda la cadena acepta ε, esto es:

$$\nu(\partial_{c_n}(\ldots \partial_{c_2}(\partial_{c_1}(r)))) = \varepsilon$$

Instrucciones

Con la práctica se incluye un parser que interpreta expresiones regulares en formato Posix, y construye valores de tipo `Regexp.regexp` utilizando las funciones especificadas en `regexp.mli`:

```
type symbol
type regexp

val symbol_of_char  : char -> symbol
val symbol_of_range : char -> char -> symbol

val empty        : regexp
val empty_string : regexp
val single       : symbol -> regexp
val except       : symbol -> regexp
val any          : regexp
val concat       : regexp -> regexp -> regexp
val repeat       : regexp -> regexp
val alt          : regexp -> regexp -> regexp
val all          : regexp -> regexp -> regexp
```

Defina en `regexp.ml` el tipo symbol para representar carácteres individuales o rangos de caracteres, y el tipo regexp que representa expresiones regulares. Defina las funciones especificadas en `regexp.mli` que construyen valores de esos tipos. Modifique el fichero `regexp.mli` para incluir las definiciones de los tipos symbol y regexp.

Defina en un nuevo fichero `derive.ml` las funciones especificadas en `derive.mli`:

- `regexp_of_string : string ->Regexp.regexp`, que construye un valor de tipo `regexp` a partir de un string con una expresión regular en formato Posix utilizando el parser:

```
let regexp_of_string s = Regexp_parser.main Regexp_lex.token
                                (Lexing.from_string s)
```

- `nullable : Regexp.regexp ->Regexp.regexp`, que para una expresión regular r devuelva $\nu(r)$
- `derive : char ->Regexp.regexp ->Regexp.regexp`, que para un caracter c y una expresión regular r devuelva $\partial_c(r)$
- `matches_regexp : string ->Regexp.regexp ->bool`, que calcula si una cadena encaja con una expresión regular
- `matches : string ->string ->bool`, donde `matches str1 str2` calcula si `str1` encaja con la expresión regular en formato posix `str2`.

Optimización

Varias de las reglas de derivación producen una expresión regular más compleja que la originaria, y el rendimiento con cadenas largas se resiente. Añada a `derive.ml` una función `simplify : Regexp.regexp ->Regexp.regexp` que simplifique recursivamente una expresión regular, teniendo en cuenta las reglas:

- $\emptyset \cdot r = \emptyset$, $r \cdot \emptyset = \emptyset$
- $r \cdot \epsilon = r$, $\epsilon \cdot r = r$
- $\epsilon^* = \epsilon$
- $\emptyset^* = \emptyset$
- $\emptyset + r = r$, $r + \emptyset = r$
- $\emptyset \,\&\, r = \emptyset$, $r \,\&\, \emptyset = \emptyset$

Y modifique las funciones `matches` y `matches_regexp` para que simplifiquen la expresión regular después de cada derivación.

Grep

Con la práctica se incluye un programa `grep` que utilizando los módulos de la práctica implementa una versión simplificada del comando `grep -E`. Una vez implementados `regexp.ml` y `derive.ml` puede compilarlo con `make grep`.

Implementar find (opcional)

Escriba en `find.ml` un programa que dados un directorio y una expresión regular, recorra recursivamente el directorio e imprima todos los elementos cuyo nombre encaje con la expresión regular. Por ejemplo:

```
$ find prueba ".*ml"
prueba/caml
prueba/p9/gtree.ml
prueba/p10/derive.ml
prueba/p10/regexp.ml
```

Para garantizar la portabilidad, utilice el módulo `Sys` para obtener los contenido de los directorios, y el módulo `Filename` para componer las rutas.

Referencias

[1] Janusz A. Brzozowski. Derivatives of regular expressions. In *Journal of the ACM*, 1964.

[2] Scott Owens, John Reppy, and Aaron Turon. Regular-expression derivatives re-examined. *Journal of Functional Programming*, 19(2):173–190, 2009.