

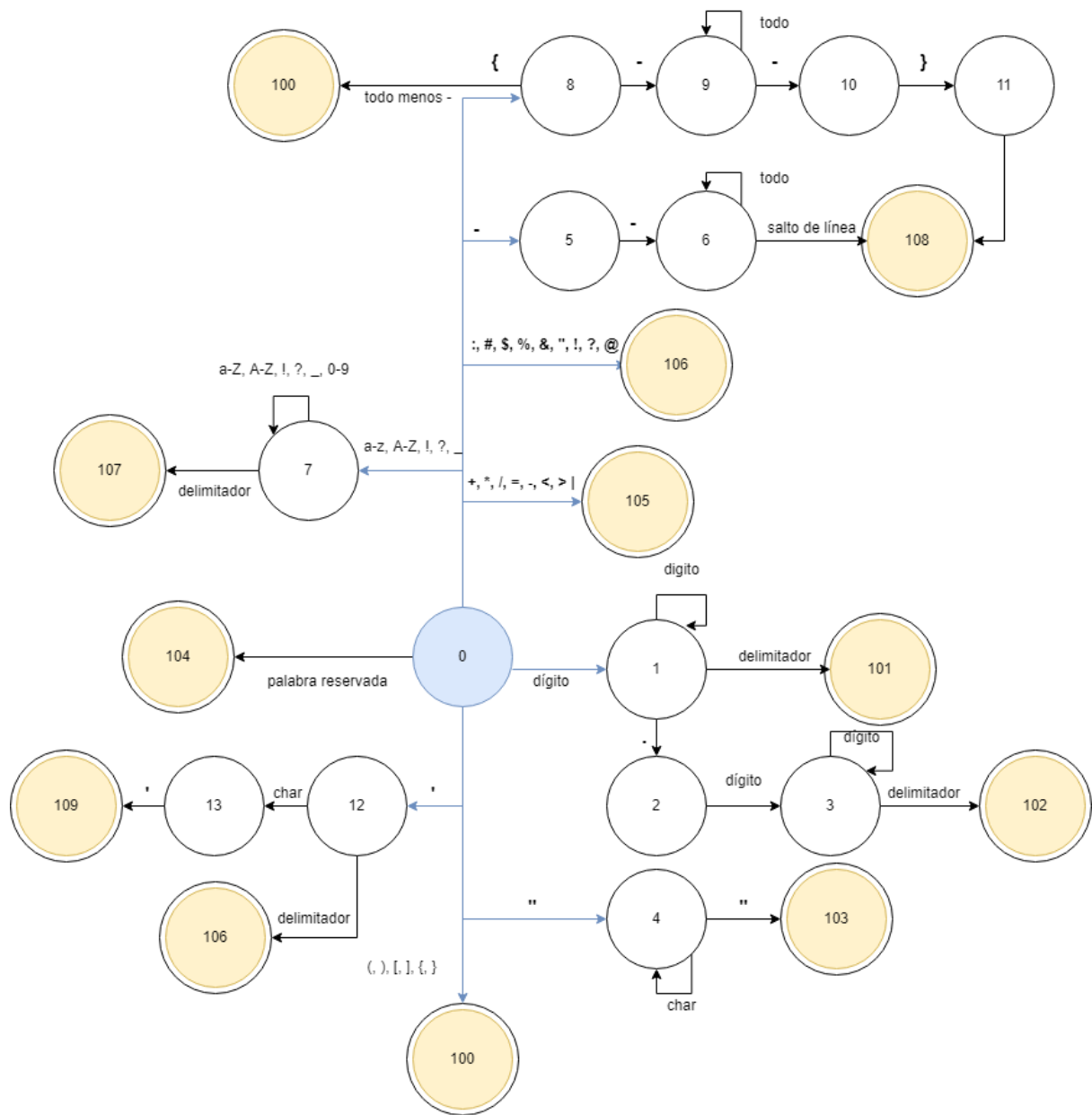
# Implementación de la Solución a la Situación Problema: Resaltador de Sintaxis v3.0 (Paralela)

Daira Adriana Chavarría Rodríguez | A01274745

## Tabla de Contenidos:

|                                    |           |
|------------------------------------|-----------|
| <b>Autómata:</b>                   | <b>2</b>  |
| Estados finales:                   | 2         |
| <b>Código</b>                      | <b>3</b>  |
| Sin paralelizar:                   | 3         |
| Paralelizando:                     | 5         |
| <b>Pruebas y resultados:</b>       | <b>5</b>  |
| <b>Tiempo de ejecución:</b>        | <b>9</b>  |
| Primera versión (sin paralelizar): | 9         |
| Segunda versión (paralelizando):   | 9         |
| Speedup                            | 9         |
| Reflexión de los resultados        | 9         |
| <b>Complejidad</b>                 | <b>10</b> |
| Primera versión (sin paralelizar)  | 10        |
| Segunda versión (paralelizando)    | 10        |
| Reflexión                          | 10        |
| <b>Video:</b>                      | <b>10</b> |

# Autómata:



## Estados finales:

- **100:** paréntesis, corchetes, llaves.
- **101:** números enteros
- **102:** números decimales
- **103:** strings en formato "Hola Mundo".
- **104:** palabras reservadas
- **105:** símbolos especiales
- **106:** operadores
- **107:** identificadores, variables

- **108:** comentarios
- **109:** caracteres en formato 'A'

## Código

### Sin paralelizar

```

ResaltadorVersion3.clj
1 (ns file.test (:require [clojure.java.io :as io]))
2 (require '[clojure.string :as string])
3 (def multilinecomment (atom 0))
4 ;Definición de las listas que ocuparemos para reconocer los tipos de datos (char)
5 (defn myNumbers [] '("0" "1" "2" "3" "4" "5" "6" "7" "8" "9"))
6 (defn myOperators [] '("=" "<" ">" "+" "-" "*" "/" "(")
7 (defn mySymbols [] '(": " #" $" "%" "&" " @" " !" " ?" " _"))
8 (defn myDelim [] '("(" ")" "[" "]" "{" "}" " " " ,")
9 ; Lista de palabras reservadas
10 (defn myReservedWords [] '("do" "forall" "hiding" "if" "let" "in" "then" "else" "qualified" "proc" "newtype" "print"
11 "rec" "mdo" "module" "default" "data" "class" "case" "of" "as" "deriving" "import"
12 "infix" "infixl" "infixr" "False" "True" "return" "Int" "Bool" "instance" "type" "where"))
13 (declare main) (declare escribir) (declare decimales) (declare error) (declare nl)
14 ;Revisamos si un dato está dentro de una de nuestras listas, para verificar su tipo
15 (defn check-If [data lista]
16   (some #(= data %) lista)) ;Regresa true si está dentro de la lista, false si no
17
18 ;Estado de error que termina con un espacio o salto de línea, para no dejar pasar mal uso del léxico
19 (defn error [lista acumulado fileName]
20   (cond (empty? lista) (escribir (apply str acumulado) 110 fileName) ;110 para rrores
21         (check-If (first lista) (myDelim)) (do (escribir (apply str acumulado) 110 fileName) (main lista fileName))
22         :else (error (rest lista) (concat acumulado (first lista)) fileName)))
23 ;Números enteros
24 (defn numbers [lista acumulado fileName]
25   (cond (empty? lista) (escribir (apply str acumulado) 101 fileName) ;101 para números enteros
26         (check-If (first lista) (myNumbers)) (numbers (rest lista) (concat acumulado (first lista)) fileName) ;Si sigue siendo número
27         (check-If (first lista) (myDelim)) (do (escribir (apply str acumulado) 101 fileName) (main lista fileName)) ;Si sigue un delimitador
28         (= (first lista) ".") (decimales (rest lista) (concat acumulado (first lista)) fileName) ;Pasamos a decimales de encontrar un punto
29         :else (error lista acumulado fileName))) ;Si no se cumple nada, error
30
31 ;Números decimales
32 (defn decimales [lista acumulado fileName]
33   (cond (empty? lista) (escribir (apply str acumulado) 102 fileName) ;Salto de línea o final
34         (check-If (first lista) (myDelim)) (do (escribir (apply str acumulado) 102 fileName) (main lista fileName))
35         (check-If (first lista) (myNumbers)) (decimales (rest lista) (concat acumulado (first lista)) fileName)
36         :else (error lista acumulado fileName)))
37 ;Identificar si es una variable o una palabra reservada
38 (defn varHelper [var fileName]
39   (cond (check-If var (myReservedWords)) (escribir var 104 fileName) ;104 de ser reservada
40         :else (escribir var 107 fileName))) ;107 para identificadores
41 ;Identificadores de variables
42 (defn variable [lista acumulado fileName]
43   (cond (empty? lista) (varHelper (apply str acumulado) fileName) ;Salto de línea o final
44         (or (check-If (first lista) (myOperators)) (check-If (first lista) (myDelim))) (do (varHelper (apply str acumulado) fileName) (main lista fi
45         :else (variable (rest lista) (concat acumulado (first lista)) fileName)))
46 ;Comentarios, iniciando por --
47 (defn comentarios [lista acumulado fileName]
48   (cond (empty? lista) (escribir (apply str acumulado) 108 fileName) ;Salto de línea o final, siendo que termina el comentario
49         :else (comentarios (rest lista) (concat acumulado (first lista)) fileName))) ;Todo lo demás es válido
50 ;String en formato "Hola Mundo"
51 (defn getString [lista acumulado fileName]
52   (cond (= (first lista) "\"") (do (escribir (apply str (concat acumulado "\"")) 103 fileName) (main (rest lista) fileName)) ;Hasta encontrar de
53         (empty? lista) (escribir (apply str acumulado) 110 fileName) ;Error
54         :else (getString (rest lista) (concat acumulado (first lista)) fileName))) ;Repetir hasta encontrar error o dobles comillas
55 ;Comentarios de múltiples líneas
56 (defn mlComment [lista acumulado fileName]
57   (cond (empty? lista) (escribir (apply str acumulado) 108 fileName)

```

```

57 ;Hasta encontrar el cierre, se sigue detectando como comentario (variable global)
58 (and (= (first lista) "-") (= (first (rest lista)) "]")) (do (reset! multipleLineComment 0) (escribir (apply str (concat acumulado '("-" "]")
59 :else (mlComment (rest lista) (concat acumulado (first lista) fileName)))
60 ;Char en formato 'A'
61 (defn character [lista fileName]
62   (cond (= (first (rest lista)) "'") (do (escribir (apply str (concat '("'") (first lista) '("'")) 109 fileName)
63     (main (rest (rest lista)) fileName)) ;Formato tradicional de los char 'A', 'I', etc
64 :else (error lista '("'") fileName))) ;De no cumplirse, error
65 ;Estado inicial 0, función principal
66 (defn main [lista fileName]
67   (cond
68     (empty? (first lista)) '() ;Terminó la lectura
69     (= @multipleLineComment 1) (mlComment lista () fileName) ;Verificamos si nuestra variable global marca comentario de múltiples líneas
70     (and (= (first lista) "{") (= (first (rest lista)) "-")) (do (swap! multipleLineComment inc) (mlComment lista () fileName)) ;Inicio de comentario
71     (check-if (first lista) (myNumbers)) (numbers lista () fileName) ;Si es un número
72     (and (= (first lista) "-" (= (first (rest lista)) "-")) (comentarios (rest (rest lista)) '("-" "-") fileName) ;Comentario en formato --comment
73     (check-if (first lista) (myDelim)) (do (escribir (first lista) 100 fileName) (main (rest lista) fileName)) ;Delimitadores
74     (check-if (first lista) (myOperators)) (do (escribir (first lista) 105 fileName) (main (rest lista) fileName)) ;Operadores
75     (check-if (first lista) (mySymbols)) (do (escribir (first lista) 106 fileName) (main (rest lista) fileName)) ;Símbolos
76     (= (first lista) "'") (character (rest lista) fileName) ;Char en formato 'A'
77     (= "\" (first lista)) (getString (rest lista) (first lista) fileName) ;Inicio de string "Hola Mundo"
78     :else (variable (rest lista) (first lista) fileName)) ;Todo lo demás es aceptado como identificador

```

```

80 ;Generación del nombre de archivo de salida, teniendo en cuenta el nombre del archivo de entrada (HTML)
81 (defn outputName [name]
82   (str (.substring (java.lang.String. name) 0 (- (count name) 4)) ".html"))
83
84 (defn get-code [tipo]
85   (cond
86     (= tipo 100) "#eb8c34\">" ;DELIMITADORES
87     (= tipo 101) "#d000ff\">" ;Números ENTEROS
88     (= tipo 102) "#e6a627\">" ;Números DECIMALES
89     (= tipo 103) "#FFF233\">" ;String "Hola Mundo"
90     (= tipo 104) "#51f542\">" ;palabra RESERVADA
91     (= tipo 105) "#ffcb78\">" ;Operadores
92     (= tipo 106) "#34ceeb\">" ;SÍMBOLOS especiales
93     (= tipo 107) "#ffffff\">" ;IDENTIFICADORES
94     (= tipo 108) "#9c92a8\">" ;COMENTARIOS
95     (= tipo 109) "#ED33FF\">" ;Char en formato 'A'
96     (= tipo 110) "#ff0000\">")) ;ERRORES
97
98 ; <span style="color: #ff0000">January 30, 2011</span>
99 (defn escribir-h [data codigo fileName]
100   (let [wrtr (io/writer fileName :append true)]
101     (.write wrtr "<span style =\"color: ")
102     (.write wrtr codigo)
103     (.write wrtr data)
104     (.write wrtr "</span>")
105     (.close wrtr)))

```

```

107 (defn escribir [data tipo fileName]
108   (escribir-h data (get-code tipo) fileName))
109 ;Encabezado HTML
110 (defn beginHTML [fileName]
111   (let [wrtr (io/writer fileName :append true)]
112     (.write wrtr "<!DOCTYPE html> <html> <body style=\"background-color:black;\")
113     (.close wrtr)))
114 ;Cierre del HTML
115 (defn endHTML [fileName]
116   (let [wrtr (io/writer fileName :append true)]
117     (.write wrtr "</body> </html>\")
118     (.close wrtr)))
119 ;Nueva línea en HTML
120 (defn nl [fileName]
121   (let [wrtr (io/writer fileName :append true)]
122     (.write wrtr "<br>")
123     (.close wrtr)))
124 ;Todo lo leído se parte (string -> char)
125 (defn begin [data fileName]
126   (main (apply list (string/split data #")) fileName) (nl fileName))
127 ;Inicia el resaltador
128 (defn resaltador [nombre]
129   (beginHTML (outputName nombre))
130   (with-open [rdr (io/reader nombre)]
131     (doseq [line (line-seq rdr)] ;Se lee línea por línea de cada archivo
132       (begin line (outputName nombre)))) ;Y se inicia el autómata con dicha cadena de caracteres
133   (reset! multipleLineComment 0)
134   (endHTML (outputName nombre)))

```

```

135 ;Se crea la lista de los nombres de los archivos a leer
136 (defn fileNames [] (string/split-lines (slurp "nombres.txt")))
137 ;Paralelizando, iniciando el resaltador para cada archivo .txt
138 (time (println (pmap (fn [name] (resaltador name)) (apply list (fileNames))))))

```

## Paralelizando:

Se utilizó el mismo código, salvo que se implementó **pmap** para poder hacer las operaciones de cada archivo al mismo tiempo.

```

(time (println (pmap (fn [name] (resaltador name)) (apply list (fileNames))))

```

## Pruebas y resultados:



A screenshot of a text editor showing a file named 'nombres.txt'. The file contains a list of 10 .txt files, numbered 1 to 10. The files are: 1. codigoCanva.txt, 2. prueba.txt, 3. prueba2.txt, 4. StateMonad.txt, 5. wikiHaskell.txt, 6. wikiHaskell2.txt, 7. practicalTemplate.txt, 8. pT.txt, 9. pT2.txt, and 10. pT3.txt.

Se hicieron pruebas usando 10 archivos .txt (los mostrados en la imagen a la izquierda).

Ambos códigos (ambas versiones) generaron los **mismos resultados**, así que se muestran de forma general a continuación:

\*Muchos .txt son versiones repetidas del .txt original

\*Cada archivo tiene un .html de salida

\* Cada archivo, en general, tuvo un largo de 200 líneas.

```

-- Operaciones básicas:
1.1239 + 5443
(if (<= x y))
-- Operadores
+ - * = <= >=
-- Símbolos
& % $ #
"Hola Mundo" --comentario
-- Delimitadores...
{} []
-- Después se realizarán más ejemplos
-- con el debido respeto al léxico
{- comentario de muchas líneas...
línea 2
línea 3
-}
'A' 'B' ("A" "ASDSA") 'C'
"Hola mundo ... de nuevo unu"

```

```

{-# LANGUAGE TemplateHaskell #-}
import Control.Monad
import Language.Haskell.TH

curryN :: Int -> Q Exp
curryN n = do
  f <- newName "f"
  xs <- replicateM n (newName "x")
  let args = map VarP (f:xs)
  ntup = TupE (map VarE xs)
  return $ LamE args (AppE (VarE f) ntup)

genCurries :: Int -> Q [Dec]
genCurries n = forM [1..n] mkCurryDec
  where mkCurryDec ith = do
    cury <- curryN ith
    let name = mkName $ "curry" ++ show ith
    return $ FunD name [Clause [] (NormalB cury) []]

curry1 = \ f x1 -> f (x1)
curry2 = \ f x1 x2 -> f (x1, x2)
curry3 = \ f x1 x2 x3 -> f (x1, x2, x3)
curry4 = \ f x1 x2 x3 x4 -> f (x1, x2, x3, x4)
...
curry20 = \ f x1 x2 ... x20 -> f (x1, x2, ..., x20)

genCurries :: Int -> Q [Dec]
genCurries n = forM [1..n] mkCurryDec
  where mkCurryDec ith = do
    cury <- curryN ith
    let name = mkName $ "curry" ++ show ith
    return $ FunD name [Clause [] (NormalB cury) []]

genCurries :: Int -> Q [Dec]
genCurries n = forM [1..n] mkCurryDec
  where mkCurryDec ith = funD name [clause [] (normalB (curryN ith)) []]
  where name = mkName $ "curry" ++ show ith

```

```

genId :: Q Exp
genId = do
  x <- newName "x"
  lamE [varP x] (varE x)

mapN :: Int -> Q Dec
mapN n
  | n >= 1 = funD name [cl1, cl2]
  | otherwise = fail "mapN: argument n may not be <= 0."
  where
    name = mkName $ "map" ++ show n
    cl1 = do f <- newName "f"
            xs <- replicateM n (newName "x")
            ys <- replicateM n (newName "ys")
            let argPatts = varP f : consPatts
                consPatts = [ [p] $(varP x) : $(varP ys) [] ]
                (x.ys) <- xs `zip` ys ]
            apply = foldl (\ g x -> [] $g $(varE x) [])
            first = apply (varE f) xs
            rest = apply (varE name) (f:ys)
            clause argPatts (normalB [ $(first) : $rest [] ]) []
    cl2 = clause (replicate (n+1) wildP) (normalB (conE '[])) []

x :: Int
x = 42

static :: Q Exp
static = [ x ]

plus42 :: Int -> Int
plus42 x = $static + x

x :: Int
x = 42

dynamic :: Q Exp
dynamic = VarE (mkName "x")

```

```

times2 :: Int -> Int
times2 x = $dynamic + x

data Deriving = Deriving { tyCon :: Name, tyVar :: Name }

deriveFunctor :: Name -> Q [Dec]
deriveFunctor ty
= do (TyConI tyCon) <- reify ty
    (tyConName, tyVars, cs) <- case tyCon of
    DataD _ nm tyVars cs _ -> return (nm, tyVars, cs)
    NewtypeD _ nm tyVars c _ -> return (nm, tyVars, [c])
    _ -> fail "deriveFunctor: tyCon may not be a type synonym."

let (KindedTV tyVar StarT) = last tyVars
instanceType = conT "Functor" `appT`
(foldl apply (conT tyConName) (init tyVars))

putQ $ Deriving tyConName tyVar
sequence [instanceD (return []) instanceType [genFmap cs]]
where
apply t (PlainTV name) = appT t (varT name)
apply t (KindedTV name _) = appT t (varT name)

genFmap :: [Con] -> Q Dec
genFmap cs
= do funD 'fmap' (map genFmapClause cs)

genFmapClause :: Con -> Q Clause
genFmapClause c@(NormalC name fieldTypes)
= do f <- newName "f"
    fieldNames <- replicateM (length fieldTypes) (newName "x")

let pats = varP f:[conP name (map varP fieldNames)]
body = normalB $ appsE $
conE name : map (newField f) (zip fieldNames fieldTypes)

clause pats body []

```

```

newField :: Name -> (Name, StrictType) -> Q Exp
newField f (x, (_, fieldType))
= do Just (Deriving typeCon typeVar) <- getQ
case fieldType of
VarT typeVar' | typeVar' == typeVar ->
[[ $(varE f) $(varE x) ]]
ty `AppT` VarT typeVar' |
leftmost ty == (ConT typeCon) && typeVar' == typeVar ->
[[ fmap $(varE f) $(varE x) ]]
_ -> [[ $(varE x) ]]

leftmost :: Type -> Type
leftmost (AppT ty1 _) = leftmost ty1
leftmost ty = ty

data RegExp
= Char (Set Char) -- [a], [abc], [a-z]; matches a single character from the specified class
| Alt RegExp RegExp -- r1 | r2 (alternation); matches either r1 or r2
| Seq RegExp RegExp -- r1 r2 (concatenation); matches r1 followed by r2
| Star RegExp -- r* (Kleene star); matches r zero or more times
| Empty -- matches only the empty string
| Void -- matches nothing (always fails)
| Var String -- a variable holding another regexp (explained later)
deriving Show

match :: RegExp -> String -> Bool
match r s = nullable (foldl deriv r s)

nullable :: RegExp -> Bool
nullable (Char _) = False
nullable (Alt r1 r2) = nullable r1 || nullable r2
nullable (Seq r1 r2) = nullable r1 && nullable r2
nullable (Star _) = True
nullable Empty = True
nullable Void = False
nullable (Var _) = False

```

```

deriv :: RegExp -> Char -> RegExp
deriv (Char cs) c
| c `Set.member` cs = Empty
| otherwise = Void
deriv (Alt r1 r2) c = Alt (deriv r1 c) (deriv r2 c)
deriv (Seq r1 r2) c
| nullable r1 = Alt (Seq (deriv r1 c) r2) (deriv r2 c)
| otherwise = Seq (deriv r1 c) r2
deriv (Star r) c = deriv (Alt Empty (Seq r (Star r))) c
deriv Empty _ = Void
deriv Void _ = Void
deriv (Var _) _ = Void

{-# LANGUAGE QuasiQuotes #-}

validDotComMail :: RegExp
validDotComMail = [regex|([a-z]([0-9])*)@([a-z]([0-9])*)\.com|]

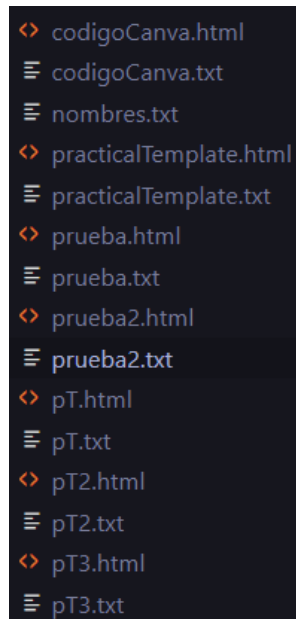
alphaNum, validDotComMail' :: RegExp
alphaNum = [regex|[a-z]([0-9])|]
validDotComMail' = [regex|${alphaNum} * @$ {alphaNum} *\.com|]

regex :: QuasiQuoter
regex = QuasiQuoter {
quoteExp = compile
, quotePat = notHandled "patterns"
, quoteType = notHandled "types"
, quoteDec = notHandled "declarations"
}
where notHandled things = error $
things ++ " are not handled by the regex quasiquoter."

compile :: String -> Q Exp
compile s =
case P.parse regexParser "" s of
Left err -> fail (show err)
Right regexp -> [e| regexp |]

```





```
<> codigoCanva.html
≡ codigoCanva.txt
≡ nombres.txt
<> practicalTemplate.html
≡ practicalTemplate.txt
<> prueba.html
≡ prueba.txt
<> prueba2.html
≡ prueba2.txt
<> pT.html
≡ pT.txt
<> pT2.html
≡ pT2.txt
<> pT3.html
≡ pT3.txt
```

\*La imagen de arriba tiene como propósito comprobar que se generaron archivos de salida (HTML) para cada archivo de entrada, es decir, no se juntó todo en uno solo.

## Tiempo de ejecución:

Primera versión (sin paralelizar):

```
"Elapsed time: 10136.3575 msecs"
```

```
"Elapsed time: 10244.8645 msecs"
```

```
"Elapsed time: 10953.2324 msecs"
```

Se tardó alrededor de 10444.81813 milisegundos en terminar de procesar los 10 archivos, es decir, aproximadamente **10.44 segundos**.

Segunda versión (paralelizando):

```
"Elapsed time: 2484.845 msecs"
```

```
"Elapsed time: 3463.4617 msecs"
```

```
"Elapsed time: 3004.7034 msecs"
```

Se tardó alrededor de 2984.3367 milisegundos en terminar de procesar los 10 archivos, es decir, aproximadamente **2.98 segundos**.

## Speedup

La versión paralela resulta ser **3.5 veces más rápida** que la versión anterior.

## Complejidad

### Primera versión (sin paralelizar)

**$O(n*m)$**

Debido a que se recorre **para cada caracter de cada archivo**.

n = cantidad de archivos

m = cantidad de caracteres

### Segunda versión (paralelizando)

**$O(n)$**

Gracias a que estamos paralelizando, no repetimos el ciclo dentro de la misma "línea".

n = cantidad de caracteres

## Reflexión

Comparando con los resultados obtenidos en cuanto al tiempo de ejecución, **tiene sentido debido a que con menor complejidad se esperaría un resultado más rápido para los mismos casos prueba**.

Este tipo de herramientas resultan de especial impacto en profesiones como la nuestra, debido a que mejorar la complejidad (y por tanto el tiempo de ejecución) de los códigos resulta ser esencial al momento de incorporar soluciones de mejora y utilidad al público (como, por ejemplo, motores de búsqueda).

El mundo tecnológico está en constante evolución, es por ello que adaptarse a lo que hoy se está usando resulta ser fundamental para entender (o crear) lo que mañana se implementará.

## Video:

<https://youtu.be/OJD1nuZpiTg><https://youtu.be/OJD1nuZpiTg>