Compiladores
Unidad 2: Análisis Léxico

Caso de estudio 01: La biblioteca Alex de Haskell

Manuel Soto Romero

Semestre 2026-1
Facultad de Ciencias UNAM

En esta nota revisaremos el funcionamiento de la biblioteca Alex de Haskell, un generador de analizadores léxicos que permite automatizar la traducción de expresiones regulares a autómatas deterministas ejecutables. A lo largo de la nota exploraremos su estructura general, los fundamentos teóricos que la sostienen y ejemplos prácticos que muestran cómo integrar Alex en el diseño de compiladores.

La construcción de analizadores léxicos es un paso fundamental en el proceso de compilación. Sin embargo, implementarlos manualmente suele ser costoso y propenso a errores. En este caso de estudio se presenta Alex, una herramienta que transforma especificaciones en forma de expresiones regulares en código eficiente de Haskell, sustentado en la teoría de autómatas finitos. Se discuten sus fases internas, desde la generación de AFN-ε hasta la obtención de autómatas deterministas minimizados, y se muestra cómo este proceso se traduce en un analizador práctico capaz de producir *tokens* con atributos útiles para el compilador.

1. ¿Qué es Alex?

Alex es un generador de analizadores léxicos para Haskell, análogo a la herramienta Lex en C. A partir de un archivo de especificación, Alex produce automáticamente un módulo Haskell que implementa una función de análisis léxico. Este archivo describe los *tokens* mediante expresiones regulares, asignando a cada una una acción en Haskll (normalmente la construcción de un *token*).

El compilador del lenguaje (en este caso Haskell) no necesita recorrer a mano la cadena ni clasificar símbolos; Alex se encarga de transformar las especificaciones en autómatas deterministas eficientes que reconocen los *tokens* en tiempo lineal.

2. Estructura general de un archivo en Alex y funcionamiento

Un archivo típio de Alex, como el mostrado en el Ejemplo 1 de una versión de MiniLisp, se divide en tres secciones:

1. Preámbulo Código Haskell entre llaves { ... } que se copia directamente al archivo generado. Suele incluir importaciones y definiciones de tipos.
2. Reglas Definiciones de macros (clases de símbolos) y reglas léxicas con expresiones regulares asociadas a acciones.
3. Epílogo Definiciones adicionales en Haskell, como el tipo `Token` o funciones auxiliares.

Ejemplo 1

Vamos un ejemplo de archivo de especificación para ALEX, correspondiente a una versión sencilla del lenguaje MINILISP.

Preámbulo

En el preábulo se establece el módulo HASKELL y se importan bibliotecas necesarias para el manejo de símbolos.

```
{
  module Lex (Token(...), lexer) where
  import Data.Char (isSpace)
}
```

Macros y Reglas

Después del preámbulo, ALEX requiere indicar un `%wrapper`, que define el modo de funcionamiento del analizador léxico. En este caso se utiliza `%wrapper "basic"`, que genera un analizador simple en forma de función pura (veremos otras variantes en el siguiente ejemplo):

```
%wrapper "basic"
```

Aquí se definen clases de símbolos (llamadas macros) y, posteriormente, las reglas léxicas que describen qué *tokens* denem gemerarse para ciertos patrones. Observemos que algunas acciones simplemente ignoras símbolos (como los espacios), mientras que otras construyen *tokens* con atributos (como los números y los identificadores).

```
-- Usamos codigos hex para los espacios en blanco Unicode mas comunes:
$white = [\x20\x09\x0A\x0D\x0C\x0B]
$digit = 0-9
$letter = [A-Za-z_]
$idrest = [A-Za-z0-9_]

tokens :-

  $white+               ;
  \(                    { \_ -> TokenPA }
  \)                    { \_ -> TokenPC }
  \+                    { \_ -> TokenSuma }
  \-                    { \_ -> TokenResta }
  not                   { \_ -> TokenNot }
  let                   { \_ -> TokenLet }

  "#t"                  { \_ -> TokenBool True }
  "#f"                  { \_ -> TokenBool False }

  $digit+               { \s -> TokenNum (read s) }
  $letter$idrest*       { \s -> TokenId s }

  . { \s -> error ("Lexical error: caracter no reconocido = "
```

```
                ++ show s
                ++ " | codepoints = "
                ++ show (map fromEnum s)) }
```

Epílogo

Finalmente, en el epílogo se define el tipo de datos `Token`, junto con constructores para cada *token*. Además se agrega una función auxiliar para normalizar espacios (en este lenguaje en específico) y se define la función `lexer` para facilitar su uso dado que Alex por defecto genera una función llamada `alexScanTokens`.

```
{
data Token
  = TokenId String
  | TokenNum Int
  | TokenBool Bool
  | TokenSuma
  | TokenResta
  | TokenNot
  | TokenPA
  | TokenPC
  | TokenLet
  deriving (Show)

-- Normaliza cualquier espacios en blanco Unicode a ' ' para que $white+ lo consuma
normalizeSpaces :: String -> String
normalizeSpaces = map (\c -> if isSpace c then '\x20' else c)

-- Alias: Alex define alexScanTokens (String -> [Token])
lexer :: String -> [Token]
lexer = alexScanTokens . normalizeSpaces
}
```

Cuando Alex procesa un archivo de especificación no se limita a emparejar expresiones regulares con cadenas de símbolos. Internamente sigue un flujo bien establecido que refleja los fundamentos de la teoría de lenguajes regulares:

1. Cada expresión regular definida en las reglas se transforma en un autómata finito no determinista con transiciones ε (AFN-ε). Luego, todos estos autómatas se unen en un único AFN-ε que representa la totalidad de las reglas. Este AFN-ε se convierte después en un AFN sin transiciones ε. Cabe aclarar que existen algoritmos que permiten ir directamente de las expresiones regulares a un AFN sin necesidad de pasar por las ε-transiciones, pero la construcción con ε es la más común y pedagógicamente clara.

2. El AFN resultante se transforma en un autómata finito determinista (AFD) mediante el procesimiento estándar de contrucción por subconjuntos.

3. El AFD obtenido obtenido se otpimiza reduciendo la cantidad de estados redundantes (minimización). Esto asegura que el analizador generado sea compacto y eficiente.

4. Finalmente, el AFD minimizado se convierte en código Haskell. Desde el punto de vista de las personas programadoras, este código corresponde a lo que en el curso llamaremos una Máquina Discriminadora Determinista (MDD), es decir, una representación ejecutable del autómata determinista.

Observación 1

El término **MDD** no forma tal cual parte de la implementación de Alex. En realidad, Alex produce funciones y tablas en Haskell que implementan directamente el AFD minimizado. Sin embargo, en el curso usaremos la noción de MDD como recurso didáctico para enfatizar que el resultado final es un **AFD ejecutable** en el sentido de que ya no se trata simplemente de un AFD abstracto que reconoce cadenas, sino de un mecanismo que sirve como **núcleo del analizador léxico**, capaz de **generar *tokens* con su cateogría léxica y atributos** según las reglas definidas en el archivo de especificación.

El uso de generadores de analizadores léxicos como Alex ofrece beneficios inmediatos para quienes construyen compiladores, pero también abre la puerta a reflexionar sobre el diseño de los algoritmos que los hacen posibles. Desde ambas perspectivas, se pueden señalar algunas ventajas claras:

- Las personas que implementan un compilador no necesitan implementar manualmente el analizador léxico. Basta con describir los patrones en forma de expresiones regulares, y los generadores se encargan de construir automáticamente código eficiente.
- Al basarse en teoría de autómatas, los analizadores generados reconocen *tokens* de manera determinista en tiempo lineal respecto al tamaño de la entrada, garantizando precisión y rapidez.
- Si cambian las reglas de un lenguaje (por ejemplo, al añadir operadores o palabras reservadas), basta con modificar el archivo de especificación y generar el analizador sin reescribir código complejo.
- Desde la perspectiva de las ciencias de la computación, entender y construir estos algoritmos es un ejercicio fundamental. El diseño de un generador como Alex involucra el dominio de la teoría de lenguajes regulares, autómatas, algoritmos de determinización y minimizació, así como su traducción a estructuras de datos eficientes en un lenguaje de programación real.
- Desde las ciencias de la computación no sólo usamos la herramienta, sino que estamos en posición de mejorarla o crear nuevas variantes. Esto puede incluir optimizaciones de los algoritmos de construcción e AFD, adaptaciones para soportar Unicode de manera más eficiente o incluso la integración de técnicas modernas de análisis incremental y paralelización.

3. Ejercicios

Para consolidad los temas que revisamos en esta nota, se proponen un par de ejercicios breves usando Alex con el fin de experimentar.

Ejercicio 1

1. Escribe un archivo de especificación llamado `SimpleLexer.x` que reconozca únicamente:
   - Números enteros.
   - El operador +
   - Espacios en blanco (que se ignoran).

```
$digit = 0-9
tokens :-
  $white+      ;
  $digit+      { \s -> TInt (read s) }
  \+           { \_ -> TOpPlus }

{
data Token = TInt Int | TOpPlus
  deriving (Show)

lexer :: String -> [Token]
lexer = alexScanTokens
}
```

2. Compila y ejecuta

```
alex SimpleLexer.x
ghci SimpleLexer.hs
> lexer "3 + 42"
[TInt 3, TOpPlus, TInt 42]
```

Ejercicio 2

Ahora crea un archivo `SimpleLexerPosn.x` que use el *wrapper* `posn`. Este *wrapper* hace que la acción reciba, además del *lexema*, la **posición** (línea y columna) del *token*. Con ello, el *token* puede cargar **más de un atributo** (por ejemplo, valor y posición).

```
{
module SimpleLexerPosn (Token(..), lexer) where
}

%wrapper "posn"

$digit = 0-9

tokens :-

  $white+        ;

  $digit+        { \pos s -> TIntPos (read s) pos }
  \+             { \pos _ -> TOpPlusPos pos }

{
data Token
  = TIntPos    Int  AlexPosn   -- valor y posicion
```

```
    | TOpPlusPos     AlexPosn   -- solo posicion
    deriving (Show)

lexer :: String -> [Token]
lexer = alexScanTokens
}
```

Prueba en GHCi:

```
> lexer "3 + 42"
[TIntPos 3 (AlexPosn 0 1 1), ...]
```

Esto muestra cómo los ***tokens*** **pueden tener más de un atributo**, algo esencial en compiladores reales donde además del **valor** se suele incluir información como la **posición**, el **tipo** o referencias a la **tabla de símbolos**.

4. Conclusión

El análisis realizado nos permite comprender que el verdadero valor de Alex no reside únicamente en su utilidad práctica como generador de analizadores léxicos, sino también en la forma en que encarna los principios teóricos de los lenguajes regulares y la teoría de autómatas. De este modo, su estudio aporta tanto a la formación práctica de quienes implementan compiladores como a la reflexión académica sobre el diseño y mejora de algoritmos. En conclusión, aprender a usar y analizar herramientas como Alex constituye un ejercicio esencial para toda persona interesada en la ciencia de la computación y en el arte de los lenguajes de programación.

Referencias

[1] A. V. Aho, M. S. Lam, R. Sethi y J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2.ª ed. Boston, MA, USA: Addison-Wesley, 2006.

[2] Haskell Community, *Alex*, Repositorio en GitHub, [En línea]. Disponible en: `https://github.com/haskell/alex`, 2025.

[3] Haskell.org. "Alex: A Lexical Analyser Generator for Haskell." [En línea]. Disponible en: `https://hackage.haskell.org/package/alex`. (2025).

[4] M. S. Paterson y D. R. Musser, "Lex: A Lexical Analyzer Generator," Bell Laboratories, Murray Hill, NJ, inf. téc., 1975.

[5] M. S. Romero, *MiniLisp*, Repositorio en GitHub, [En línea]. Disponible en: `https://github.com/lambdasspace/MiniLisp`, 2025.