REPUBLICA BOLIVARIANA DE VENEZUELA

MINISTERIO DEL PODER POPULAR PARA LA EDUCACION SUPERIOR

U.N.E.R.G. UNIVERSIDAD NACIONAL EXPERIMENTAL ROMULO

GALLEGOS

**Prof.** Dehivis Pérez

**INTEGRANTE:**

Franco Quatrini

**C.I:** 19418050

Analizador léxico

Un analizador léxico y/o analizador lexicográfico (en inglés scanner) es la primera fase de un compilador consistente en un programa que recibe como entrada el código fuente de otro programa (secuencia de caracteres) y produce una salida compuesta de [tokens](https://es.wikipedia.org/wiki/Token_(programaci%C3%B3n)) (componentes léxicos) o símbolos. Estos tokens sirven para una posterior etapa del proceso de traducción, siendo la entrada para el analizador sintáctico (en inglés parser).

La especificación de un lenguaje de programación a menudo incluye un conjunto de reglas que definen el léxico. Estas reglas consisten comúnmente en expresiones regulares que indican el conjunto de posibles secuencias de caracteres que definen un token o lexema.

En algunos lenguajes de programación es necesario establecer patrones para caracteres especiales (como el espacio en blanco) que la gramática pueda reconocer sin que constituya un token en sí.

Comenzaremos con la parte mas sencilla del compilador: el analizador léxico. Habitualmente el término ``análisis léxico'' se refiere al tratamiento de la entrada que produce como salida la lista de tokens. Un token hace alusión a las unidades mas simples que tiene significado. Habitualmente un token o lexema queda descrito por una expresión regular. Léxico viene del griego lexis, que significa ``palabra''. Perl es, sobra decirlo, una herramienta eficaz para encontrar en que lugar de la cadena se produce un emparejamiento. Sin embargo, en el análisis léxico, el problema es encontrar la sub cadena a partir de la última posición en la que se produjo un emparejamiento y que es aceptada por una de las expresiones regulares que definen los lexemas del lenguaje dado.

La estructura general del analizador léxico consiste en un bucle en el que se va recorriendo la entrada, buscando por un emparejamiento con uno de los patrones/lexemas especificados y, cuando se encuentra, se retorna esa información al analizador sintáctico. Como no tenemos escrito el analizador sintáctico simplemente iremos añadiendo los terminales al final de una lista.

Una iteración del bucle tiene la forma de una secuencia de condicionales en las que se va comprobando si la entrada casa con cada una de las expresiones regulares que definen los terminales del lenguaje. Las condiciones tienen un aspecto similar a este:

...

if (m{\G\s\*(\d+)}gc) {

push @tokens, 'NUM', $1;

}

elsif (m{\G\s\*([a-z\_]\w\*)\b}igc) {

push @tokens, 'ID', $1;

}

...

Una expresión common {\G\s\*(\d+)}gc es una expresión regular. Es conveniente que en este punto repase la introducción a las expresiones regulares en[[*]](http://nereida.deioc.ull.es/~pl/perlexamples/node71.html#section:introregexp)[2].

#### El Operador de Binding

Nótese que, puesto que no se menciona sobre que variable se hace el binding (no se usa ninguno de los operadores de binding =~ y !~): se entiende que es sobre la variable por defecto $\_ sobre la que se hace el matching.

#### Casando a partir del Último Emparejamiento

El ancla \G casa con el punto en la cadena en el que terminó el último emparejamiento.

La expresión regular describiendo el patrón de interés se pone entre paréntesis para usar la estrategia de los paréntesis con memoria.

Las opciones c y g son especialmente útiles para la construcción del analizador.

#### La opción g

Como lo usamos en un contexto escalar, la opción g itera sobre la cadena, devolviendo cierto cada vez que casa, y falso cuando deja de casar. Se puede averiguar la posición del emparejamiento utilizando la función pos.

#### La opción c

La opción /c afecta a las operaciones de emparejamiento con /g en un contexto escalar. Normalmente, cuando una búsqueda global escalar tiene lugar y no ocurre casamiento, la posición de comienzo de búsqueda es restablecida al comienzo de la cadena. La opción /c hace que la posición inicial de emparejamiento permanezca donde la dejó el último emparejamiento con éxito y no se vaya al comienzo. Al combinar esto con el ancla \G, la cuál casa con el final del último emparejamiento, obtenemos que la combinación

m{\G\s\*(...)}gc

logra el efecto deseado: Si la primera expresión regular en la cadena elsif fracasa, la posición de búsqueda no es inicializada de nuevo gracias a la opción c y el ancla \G sigue recordando donde terminó el ultimo casamiento.

#### La opción i

Por último, la opción i permite ignorar el tipo de letra (mayúsculas o minúsculas).

#### Código del Analizador Léxico

Este es el código completo de la subrutina scanner que se encarga del análisis léxico:

1 package Lexical::Analysis;

2 sub scanner {

3 local $\_ = shift;

4 { # Con el redo del final hacemos un bucle "infinito"

5 if (m{\G\s\*(\d+)}gc) {

6 push @tokens, 'NUM', $1;

7 }

8 elsif (m{\G\s\*int\b}igc) {

9 push @tokens, 'INT', 'INT';

10 }

11 elsif (m{\G\s\*string\b}igc) {

12 push @tokens, 'STRING', 'STRING';

13 }

14 elsif (m{\G\s\*p\b}igc) {

15 push @tokens, 'P', 'P'; # P para imprimir

16 }

17 elsif (m{\G\s\*([a-z\_]\w\*)\b}igc) {

18 push @tokens, 'ID', $1;

19 }

20 elsif (m{\G\s\*\"([^"]\*)\"}igc) {

21 push @tokens, 'STR', $1;

22 }

23 elsif (m{\G\s\*([+\*()=;,])}gc) {

24 push @tokens, 'PUN', $1;

25 }

26 elsif (m{\G\s\*(\S)}gc) { # Hay un caracter "no blanco"

27 Error::fatal "Caracter invalido: $1\n";

28 }

29 else {

30 last;

31 }

32 redo;

33 }

34 }

#### Relación de co rutina con el Analizador Sintáctico

Si decidiéramos establecer una relación de co rutina con el analizador léxico los condicionales se pueden programar siguiendo secuencias con esta estructura:

return ('INT', 'INT') if (m{\G\s\*int\b}igc);

#### Manejo de Errores

Para completar el analizador solo quedan declarar las variables usadas y las subrutinas de manejo de errores:

######## global scope variables

our @tokens = ();

our $errorflag = 0;

package Error;

sub error($) {

my $msg = shift;

if (!$errorflag) {

warn "Error: $msg\n";

$errorflag = 1;

}

}

sub fatal($) {

my $msg = shift;

die "Error: $msg\n";

}

El uso de our es necesario porque hemos declarado al comienzo del módulo use strict. El pragma use strict le indica al compilador Perl que debe considerar como obligatorias un conjunto de reglas de buen estilo de programación. Entre otras restricciones, el uso del pragma implica que todas las variables (no-mágicas) deben ser declaradas explícitamente (uso de my, our, etc.) La declaración our se describe en

Análisis

Esta etapa está basada usualmente en una máquina de estados finitos. Esta máquina contiene la información de las posibles secuencias de caracteres que puede conformar cualquier token que sea parte del lenguaje (las instancias individuales de estas secuencias de caracteres son denominados lexemas). Por ejemplo, un token de naturaleza entero puede contener cualquier secuencia de caracteres numéricos.