INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

GARZÓN DOMÍNGUEZ GERARDO ISMAEL

3CV18

COMPILADORES

PRÁCTICA 3:
FLEX

INTRODUCCIÓN

A continuación se presentan conceptos básicos para el desarrollo de esta práctica.

Intuitivamente, una gramática es un conjunto de reglas que generan un subconjunto L de $\Sigma*$,para algún un alfabeto Σ, esto es: $L \subseteq \Sigma *$. Estas reglas permiten reemplazar símbolos o cadenas de símbolos hasta que finalmente se puede probar que alguna cadena $w \in L$, a este proceso se le conoce como derivación. Bajo esta descripción, los símbolos que son reemplazados por otros símbolos son llamados no terminales, mientras que los símbolos que no pueden ser reemplazados por otros símbolos son referidos como terminales, por último, las reglas bajo las cuales está permitida la sustitución de símbolos son llamadas producciones.[1]

Formalmente, una gramática es una 4-tupla: G = (N, Σ, S, P), donde:
N: Conjunto finito de símbolos no terminales, Σ: Conjunto finito de símbolos terminales,
S: Símbolo inicial, además $S \in N$, P: Conjunto finito de reglas de producción. [1]

Formalmente P es una relación en $(N \cup \Sigma) *$, tal que cada elemento de P, es un par ordenado, donde el primer elemento pertenece a N, y al menos un par tiene como primer elemento a S. [1]

A continuación, se explicará brevemente que es una expresión regular, ya que el concepto principal a notar en esta práctica es como las *regex*, son parte importante del análisis léxico para un compilador.

Las expresiones regulares pueden usarse para describir ciertos lenguajes. Si r es una expresión regular, entonces $L(r)$ denota un lenguaje asociado con r partiendo de las siguientes afirmaciones [2]:

1. ϕ es una expresión regular que denota el conjunto vacío.
2. λ es una expresión regular que denota el conjunto: $\{\lambda\}$.
3. Para todo $\alpha \in \Sigma$, α es una expresión regular que denota a $\{\alpha\}$.

Si r 1 , r 2 son expresiones regulares, entonces [2]:

4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$,
5. $L(r_1 \cdot r_2) = L(r_1)\, L(r_2)$,
6. $L((r_1)) = L(r_1)$,
7. $L(r_1^*) = (L(r_1))^*$.

Son lenguajes regulares asociados con las expresiones regulares r 1 , r 2 .

La idea detrás de las expresiones regulares es describir un patrón o regularidad que cumplen las cadenas pertenecientes a un lenguaje, que por extensión se le llama lenguaje regular.

DESARROLLO

En esta sección se muestra como se resuelven los problemas dados mediante expresiones regulares usando el software FLEX (Fast lexycal analyzer generator). Los problemas a resolver son validar:

- **Nombres de variables al estilo *C***, o sea, pueden contener _ en cualquier posición, iniciar con cualquier letra a-z, o A-Z, y pueden precederle cadenas numéricas, pero las cadenas numéricas están prohibidas al inicio, además no se pueden tener espacios.
- **Números enteros con y sin signo,** esto es, cualquier cadena numérica que contenga un + o – opcional en su inicio, precedida por uno o más dígitos decimales.
- **Números flotantes con y sin signo,** una cadena de dígitos que contenga un + o – opcional al inicio, precedida por uno o más de dígitos en base 10, el punto decimal es forzoso (para distinguir entre enteros y flotantes / reales), y debe ser precedido por uno o más dígitos decimales.
- **Operaciones binarias**, para este caso, se validarán operaciones binarias de la forma $AopB$, donde A, B pueden ser identificadores, flotantes o enteros, y op, es una cadena que representa una operación binaria conocida (suma, resta, multiplicación, módulo, potenciación): $+, -, \star, /, \%, \star\star$ respectivamente. Como se puede notar, se validará una potenciación al estilo *python*.

A continuación se muestra la definición de cada expresión regular para cada tipo de validación mostrada anteriormente y una breve explicación:

$$ID[a-zA-Z\_]+[\_a-zA-Z0-9]* \qquad \text{(Regex 1)}$$

Expresión nombrada *ID*, debe comenzar con algún carácter dentro del rango a y z minúsculas y mayúsculas (vea cerradura más, que excluye a la cadena vacía), así como guion bajo, a esta parte de la cadena le *puede* preceder (vea cerradura estrella) el mismo tipo de caracteres que la primera parte de la cadena, pero esta vez incluyendo los dígitos de 0 al 9, esto impide que los nombres de identificador comiencen con dígitos de base 10.

En la siguiente figura se muestra como esta expresión, como se espera, valida identificadores al estilo C, o mayoría de lenguajes de programación populares en su defecto.

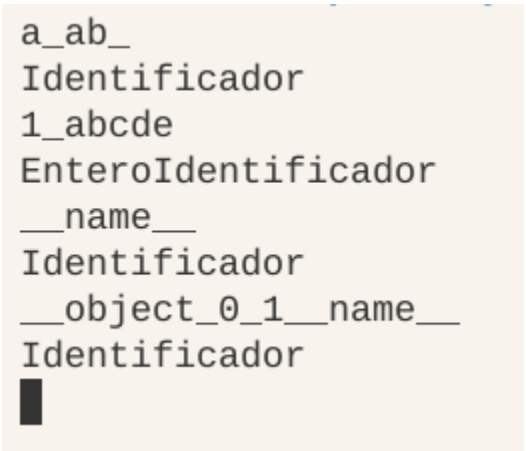

Figure 1: Prueba de validación de múltiples cadenas con (Regex 1)

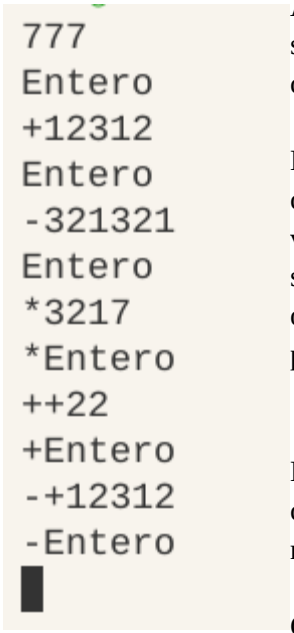

Figure 2: Prueba de validación de múltiples cadenas con (Regex 2)

Al analizar la salida, se puede notar que la segunda cadena introducida “1_abcde”, solo tiene una parte válida, esta es la parte después del dígito 1, el cual es validado como un entero y no como parte del identificador.

La expresión regular (Regex 2) permitirá validar cualquier número entero en base 10, con o sin signo al inició. Cabe mencionar que la comprobación de si es un número válido para enteros de n-bits puede ser o no ser necesaria según el lenguaje para el que se necesite la expresión, ya que algunos lenguajes tienen enteros de precisión arbitraria o también llamados enteros grandes (*bigint*), en este caso se omite tal validación, y podría incluso no formar parte del análisis léxico.

$$INTEGER[\backslash + |\backslash -]?[0-9]+ \quad \text{(Regex 2)}$$

La expresión checa que haya 0 o 1 símbolo de signo, ya sea – o +, esta parte de la cadena deberá estar precedida por uno o más dígitos decimales. En la Figure 2 se muestra una serie de pruebas para la expresión.

Como se puede notar, las primeras 3 cadenas introducidas son válidas. Sin embargo, en los últimos casos solo una parte de la cadena es válida, la parte inválida es que hay un signo de más, de tal manera que el programa generado por flex solo tomará en cuenta la cadena desde el segundo signo. Se pueden realizar pruebas adicionales para comprobarlo

Figure 3: Prueba de validación de múltiples cadenas con (Regex 3)

La siguiente expresión validará números reales, pero no números enteros, esto debido a que en esta implementación la expresión regular para enteros se encuentra en el mismo archivo que la expresión regular para reales, que por esta razón, es más adecuado llamarlos flotantes, a pesar de que tampoco se hagan comprobaciones de rango para algún tipo de dato primitivo específico como *float, double, long double,* etc.

$$FLOATING[\backslash + |\backslash -]?[0-9] + \text{“.”} 1[0-9]+ \quad \text{(Regex 3)}$$

Esta comprobación es similar a la anterior, con excepción de que las cadenas que pertenezcan al lenguaje de esta expresión deberán tener uno o más dígitos decimales que representen su parte fraccionaria. La muestra las cadenas de prueba para esta expresión

Por último, se muestra la expresión regular que valida operaciones binarias, en este caso, no se valida cada caso por separado, bastará con que se cumpla la forma *A{operador}B*. Los operadores posibles son: +,-,*,%,/,**, este último es potenciación al estilo *python.*

$$OP([\backslash + |\backslash -]?\{ID\}|\{INTEGER\}|\{FLOATING\})\{BINOP\}([\backslash + |\backslash -]?\{ID\}|\{INTEGER\}|\{FLOATING\})$$

(Regex 4)

Por ahora no se permiten espacios entre los operandos y el operador, sin embargo, se validarán operaciones binarias entre identificadores, y números enteros y reales, esto sería el equivalente a realizar operaciones

matemáticas entre identificadores y constantes en cualquier lenguaje que lo permita, como C. Note además, que los identificadores pueden ir prefijados con un signo. La Figure 4 Muestra las pruebas realizadas para esta expresión regular.

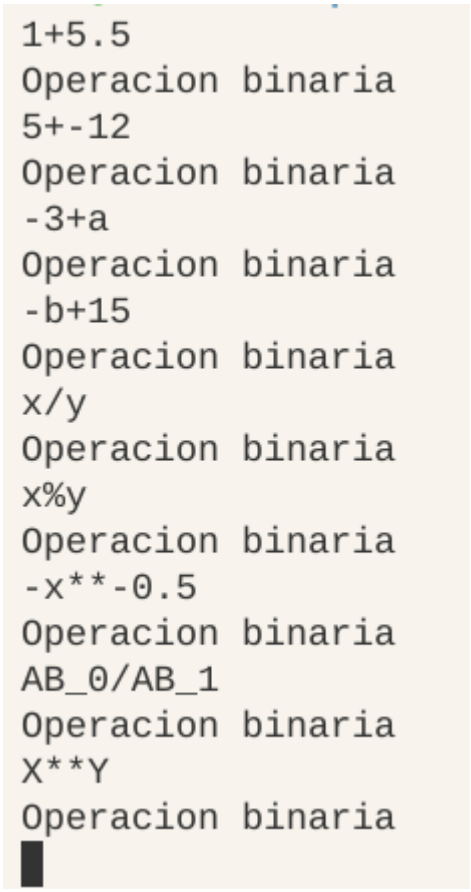

Figure 4: Prueba de validación de múltiples cadenas con (Regex 4)

CONCLUSIÓN

En esta práctica pude adentrar un poco más en el tema de expresiones regulares, y noté la relación que tienen con los compiladores, además, se nota la relación que tienen los compiladores con las gramáticas, esto es, por lo que noté, diferentes etapas del compilador utilizarán distintos tipos de gramáticas, por ejemplo, aquí usamos gramáticas regulares, que pueden ser descritas mediante autómatas finitos deterministas y no deterministas, pero imagino que hay partes del compilador que pueden necesitar de autómatas más sofisticados, como uno que determine si cada paréntesis abierto en una expresión corresponde a un paréntesis cerrado. Por último debo concluir que pude entender el propósito de Flex, a pesar de que su página oficial lo indica, pude visualizar por que y como se puede usar este programa para crear analizadores léxicos, es decir, se le indican ciertas reglas mediante expresiones regulares, y cuando una regla se cumple, se puede ejecutar código C, entonces, cada que una cadena sea aceptada por una regex, escribiremos un token que identifica el significado “léxico” de esa cadena, esto es, saber que es, identificador, constante, una operación, etc. De tal manera que al finalizar se podrá tener un programa que a partir de un archivo de entrada genere una salida que consista en una secuencia de tokens, básicamente un analizador léxico [3].

REFERENCIAS

[1] J. Anderson, Automata theory with modern applications, 1ra ed. Cambridge, New York, Melbourne: Cambridge University Press, 2006, p. 114.

[2] P. Linz, An Introduction to Formal Languages and Automata, 6ta ed. Davis, California: Jones & Bartlett, 2017, pp. 75-103.

[3] Julie Zelenski, flex In A Nutshell, 27 Jun 2012. Accedido: 20 Oct, 2021. [Online]. Disponible en: https://web.stanford.edu/class/archive/cs/cs143/cs143.1128/handouts/050%20Flex%20In%20A%20Nutshell.pdf.