Universidad Nacional Autónoma de México

Facultad de Ciencias

Compiladores

Grupo 7006

Analizador léxico

Integrantes:

Miguel Ángel Escamilla Monroy

Sara Doris Montes Incin

Carlos Eduardo Orozco Viveros

Edgar Quiroz Castañeda

Proyecto

Índice

Objetivo

Realizar un analizador léxico para la gramática dada utilizando la herramienta Flex.

Problema

Desarrollar un analizador léxico que, dada una gramática, reconozca los tokens que conforman el lenguaje.

Diseño de la solución

Se identificaron las expresiones regulares para todas las cadenas del lenguaje. Una vez dadas las expresiones, se puede usar *Lex* para construir un autómata finito determinista (AFD) que reconozca al lenguaje buscado.

Este proceso es largo y no trivial. Para ejemplificar, se presentan algunas de las expresiones regulares y se construye un AFD usando elementos punteados que reconozca un subconjunto del lenguaje.

Las expresiones usadas son las siguientes

id [a-zA-Z_]([a-zA-Z_] | [0-9])*

int (0(_0)* | [1-9]((_)?[0-9])*)

sf (int.int? | .int)

ext ([0-9]([_]?[0-9])*

exp ([0-9] | [0-9].int? | .int)[Ee][+-]?ext

float (sf | exp)

string (“[^\n”]*” | `[^`]*`)

Y a continuación se construye el AFD que las reconozca usando elementos punteados.

cerradura = {
    id -> •[a-zA-Z_]([a-zA-Z_] | [0-9])*
    int -> (•0(_0)* | [1-9]((_)?[0-9])*)
    int -> (0(_0)* | •[1-9]((_)?[0-9])*)
    sf -> (•int.int? | .int)
    sf -> (int.int? | •.int)
    exp -> (•[0-9] | [0-9].int? | .int)[Ee][+-]?ext
    exp -> ([0-9] | •[0-9].int? | .int)[Ee][+-]?ext
    exp -> ([0-9] | [0-9].int? | •.int)[Ee][+-]?ext
    float -> (•sf | exp)
    float -> (sf | •exp)
    string -> (•“[^\n”]*” | `[^`]*`)
    string -> (“[^\n”]*” | •`[^`]*`)
} = q_0

Transiciones de q_0

goto(q_0, [a-zA-Z_])= {
    id -> [a-zA-Z_] (•[a-zA-Z_] | [0-9])*
    id -> [a-zA-Z_] ([a-zA-Z_] | •[0-9])*
    id -> [a-zA-Z_] ([a-zA-Z_] |[0-9])*•
} = q_1

goto(q_0, 0) = {
    int -> (0(•_0)* | [1-9]((_)?[0-9])*)
    int -> (0(_0)* | [1-9]((_)?[0-9])*)•
} = q_2

goto(q_0, [1-9]) = {
    exp -> ([0-9] | [0-9].int? | .int)[Ee][+-]?ext•
    exp -> ([0-9] | [0-9]•.int? | .int)[Ee][+-]?ext
} = q_3

goto(q_0, int) = {
    sf -> (int•.int? | .int)

} = q_4

goto(q_0, .) = {
    sf -> (int.int? | .•int)
    exp -> ([0-9] | [0-9].int? | .•int)[Ee][+-]?ext
} = q_5

goto(q_0, sf) = {
    float -> (sf | exp)•
} = q_6

goto(q_0, exp) = {
    float -> (sf | exp)•
} = q_6

goto(q_0, “) = {
    string -> (“•[^\n”]*” | `[^`]*`)
    string -> (“[^\n”]*•” | `[^`]*`)
} = q_7

goto(q_0, ‘) = {
    string -> (“[^\n”]*” |`•[^`]*`)
    string -> (“[^\n”]*” |`[^`]*•`)
} = q_8

Transiciones de q_1

goto(q_1, [a-zA-Z_]) = {
    id -> [a-zA-Z_] (•[a-zA-Z_] | [0-9])*
    id -> [a-zA-Z_] ([a-zA-Z_] | •[0-9])*
    id -> [a-zA-Z_] ([a-zA-Z_] | [0-9])*•
} = q_1

goto(q_1, [0-9]) = {
    id -> [a-zA-Z_] (•[a-zA-Z_] | [0-9])*
    id -> [a-zA-Z_] ([a-zA-Z_] | •[0-9])*
    id -> [a-zA-Z_] ([a-zA-Z_] |[0-9])*•
} = q_1

Transiciones de q_2

goto(q_2, _) = {
    int -> (0(_•0)* | [1-9]((_)?[0-9])*)
} = q_9

Transiciones de q_3

goto(q_3, .) = {
exp -> ([0-9] | [0-9].•int? | .int)[Ee][+-]?ext
exp -> ([0-9] | [0-9].int? | .int)•[Ee][+-]?ext
} = q_{10}

Transiciones de q_4

goto(q_4, .) = {
sf -> (int.•int? | .int)
sf -> (int.int? | .int)•
} = q_{11}

Transiciones de q_5

goto(q_5, int) = {
sf -> (int.int? | .int)•
exp -> ([0-9] | [0-9].int? | .int)•[Ee][+-]?ext
} = q_{12}

Transiciones de q_7

goto(q_7, [^\n"]) = {
string -> ("•[^\n"]*" | `[^`]*`)
string -> ("[^\n"]*•" | `[^`]*`)
} = q_7

Transiciones de q_8

goto(q_8, [^`]) = {
string -> ("[^\n"]*" |`•[^`]*`)
string -> ("[^\n"]*" |`[^`]*•`)
} = q_8

goto(q_8, `) = {
string -> ("[^\n"]*" |`[^`]*`)•
} = q_{13}

Transiciones de q_9

goto(q_9, 0) = {
int -> (0(•_0)* | [1-9]((_)?[0-9])*)
int -> (0(_0)* | [1-9]((_)?[0-9])*)•
} = q_2

Transiciones de q_{10}

goto(q_{10}, int) =
{
exp -> ([0-9] | [0-9].int? | .int)•[Ee][+-]?ext
} = q_{14}

Transiciones de q_{11}

goto(q_{11}, int) = {
sf -> (int.int? | .int)•
} = q_{15}

Transiciones de q_{12}

goto(q_{12}, [Ee]) = {
exp -> ([0-9] | [0-9].int? | .int)[Ee]•[+-]?ext
exp -> ([0-9] | [0-9].int? | .int)[Ee][+-]?•ext
} = q_{16}

Transiciones de q_{14}

goto(q_{14}, [Ee]) = {
exp -> ([0-9] | [0-9].int? | .int)[Ee]•[+-]?ext
exp -> ([0-9] | [0-9].int? | .int)[Ee][+-]?•ext
} = q_{16}

Transiciones de q_{16}

goto(q_{16}, [+-]) = {
exp -> ([0-9] | [0-9].int? | .int)[Ee][+-]?•ext
} = q_{17}

goto($q_{16,}$ ext) = {
exp -> ([0-9] | [0-9].int? | .int)[Ee][+-]?ext•
} = q_{18}

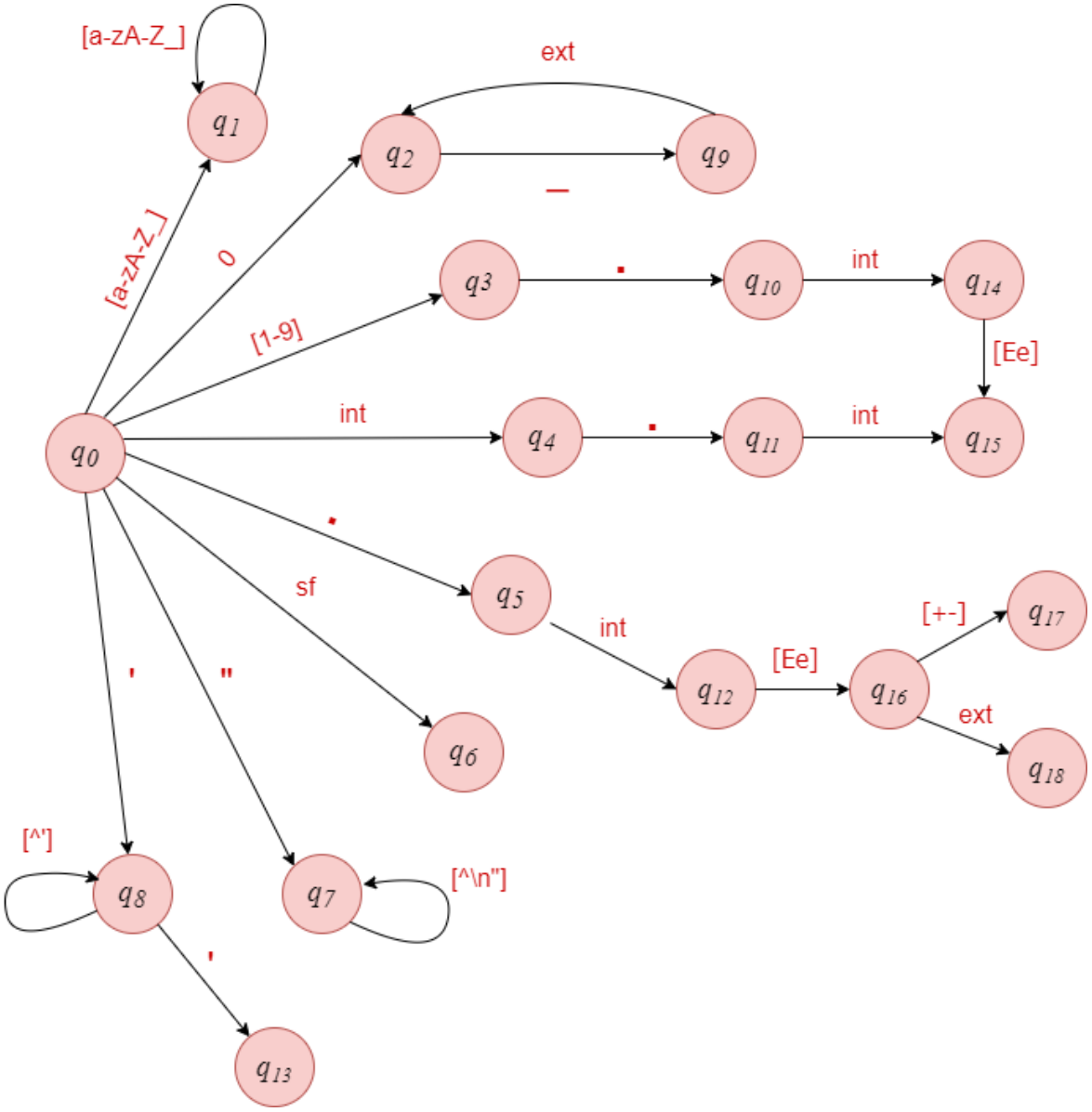
Transiciones de q_{17}

goto(q_{17}, ext) = {
exp -> ([0-9] | [0-9].int? | .int)[Ee][+-]?ext•
} = q_{18}

La función de transición recién construida se puede visualizar más fácilmente en forma de una tabla de transiciones.

Estado	[a-zA-Z_]	[Ee]	[1-9]	[^\n"]	[^`]	[0-9]	[-+]	int	sf	exp	ext	0	_	.	"	'
q_0	q_1		q_3					q_4	q_6	q_6		q_2		q_5	q_7	q_8
q_1	q_1					q_1										
q_2													q_9			
q_3														q_{10}		
q_4														q_{11}		
q_5								q_{12}								
q_6																
q_7				q_7												
q_8					q_8											q_{13}
q_9												q_2				
q_{10}								q_{14}								
q_{11}								q_{15}								
q_{12}		q_{16}														
q_{13}																
q_{14}		q_{15}														
q_{15}																
q_{16}							q_{17}				q_{18}					
q_{17}											q_{18}					
q_{18}																

Otra manera de visualizar la funcioń de transición es con un dibujo del AFD resultante.



Análisis de la solución

En el caso de los tokens descritos en el documento usamos las reglas de expresiones regulares vistas a lo largo del curso para diseñar las expresiones regulares correspondientes a cada token descrito.

El rango como prefijo de la expresión de los identificadores nos asegura que ningún identificador inicie con algo diferente a una letra o un guión bajo, como los identificadores pueden contener guiones, números y letras, entonces la disyunción nos permite iterar entre estos, por lo que la cerradura estrella es necesaria para esto.

Los tipos numéricos fueron casos más complejos, aquí observamos el comportamiento de otros lenguajes de programación y su representación se tipos numéricos para definir nuestras expresiones; los enteros permiten los guiones bajos intermedios, en el caso de cero podemos tener una cadena del estilo *0_0_0_0….* o *1_5_4_9*. Sin embargo, fuera del cero, no permitimos que existan ceros a la izquierda.

Los flotantes fueron divididos en subexpresiones para facilitar su lectura y su manejo al aplicar los elementos punteados. Debido a que los números flotantes soportan tanto representaciones enteras, números decimales, así como notación exponencial los dividimos en estas categorías; hicimos una expresión regular para los números decimales y otra para la notación exponencial, finalmente la expresión principal es la disyunción de estas dos. El caso de los números imaginarios fue más sencillo, ya que sólo tuvimos que usar la expresión definida de los flotantes y añadir la parte imaginaria.

Las cadenas se representan de la manera estándar, por lo que no deben contener comillas en medio de una cadena y, en el caso de las comillas dobles, tampoco se soporta el salto de línea, mientras que en las cadenas de comillas sencillas si es soportado.

Una vez con las expresiones definidas, sólo aplicamos los elementos punteados para obtener el autómata de nuestras expresiones regulares como se ha hecho a lo largo del curso.

El resto de tokens fueron identificados de la gramática dada. Para esto bastó con reconocer las secuencias de caracteres terminales dentro de las reglas.
Las constantes booleanas, los símbolos de puntuación, los operadores y las palabras reservadas todas son cadenas constantes. Así que sus respectivas expresiones regulares son directas.

Implementación

Una vez obtenidas las expresiones regulares para todas las cadenas del lenguaje, hay que asignar un identificador a cada una.

En la implementación, esto se hizo en un archivo *classes.h* que se incluyó en el programa principal. Cabe notar que el analizador solo se regresa estos identificadores, ignorando la sección del lexemas. Esto para simplificar todo el proceso.

Ya con las expresiones regulares y sus clasificaciones, se puede traducir todo a *Lex* directamente. El resto de la implementación son detalles para mejorar la usabilidad del programa.

Entrada y salida

Se agregó un sistema de entrada y salida. Esto para leer el código a analizar desde un archivo, con extensión *.art*, y escribir los tokens resultantes en otro, con extensión *.tokens*.

Errores

Luego, hay que definir un mecanismo para detectar errores. Por omisión, *Lex* simplemente escribe cualquier carácter que no pueda reconocer, y sigue analizando el resto del texto.

Primero, para detectar errores se añadió una regla con la menor prioridad que reconozca cualquier carácter. Esta regla solo se activa cuando se encuentra una cadena que no se puede reconocer.

Se decidió que los errores se mostrarán en la consola, indicando la cadena que causó el problema y la línea en la que sucedió. Para esto último se añadió una regla que reconoce el salto de línea, y cuya acción es incrementar una variable donde se lleva la cuenta de la línea actual.

Cadenas

Las cadenas son un tanto diferentes del resto de las expresiones. Esto porque tienen reglas que sólo son válidas dentro de ellas y que ignoran las reglas generales. Así que para estas expresiones se utilizó el mecanismo de estados de *Lex*.

Las cadenas con comillas simples pueden contener cualquier carácter, inclusive saltos de línea. Por lo que se necesita redefinir la regla para incrementar la cuenta de las líneas.

Para las cadenas con comillas dobles, además tiene un mecanismo para escapar caracteres. Esto provoca que cadenas como *"\"* sean inválidas. Esto se procesa agregando una regla para que siempre se lea y se ignore el carácter siguiente a una diagonal invertida.

Para ambos tipos de cadenas, si se llega al final del archivo leyendo una cadena, significa que nunca se cerró. Es una cadena mal formada y se imprime el error apropiado.

Repositorio:

https://github.com/QuirozE/projectoCompiladores

Conclusiones

Al desarrollar esta práctica, pudimos notar que la implementación de nuestro analizador léxico, sí cumple con la definición de un analizador léxico, ya que con las clases léxicas que definimos dada la gramática y con la herramienta flex, al leer una secuencia de caracteres, estos se agruparon a su clase léxica con base en el token correspondiente.