Facultad de Ingeniería UNAM

Universidad Nacional Autónoma de México

Analizador Sintáctico

M.I.A Laura Sandoval Montaño

Compiladores

Morales Ortega Carlos
Vélez Grande Cinthya

Fecha de entrega: 07/11/2023
Semestre 2024-1
Grupo 03

Universidad Nacional Autónoma de México

Objetivo

Elaborar un analizador léxico y Sintáctico Descendente Recursivo que revisen programas escritos en el lenguaje definido por la gramática acordada en clase.

Descripción del Problema

Se deberá realizar un analizador léxico y sintáctico descendente recursivo que reconozca la siguiente gramática:

1. Sentencia declarativa:

   Gramática:

   <Decl> → <Tipo>a<valorIni><listaVar>;
   <Tipo>→ f
   <Tipo>→ i
   <valorIni>→=<tipoVal>
   <valorIni> → ξ
   <listaVar>→ ,a<valorIni><listaVar>
   <listaVar> → ξ
   <tipoVal> → n
   <tipoVal> → v

   Ejemplos:

   Enteiro _val=2, _num1, _num2;
   Flutuador _cad1=”cadeana#1”, _cad2;

2. Sentencias de asignación:

   Gramática:

   <Asig> → a<opArit>E;
   <opArit> → =
   <opArit> → x
   <opArit> → y
   <opArit>→ k
   <opArit>→ r
   <opArit>→ o

   Ejemplos:

   _num1 = _val+15*(12-(+93));
   _cad2=_cad1 like “cadena #2”
   _num2 = (-285);

3. Expresión aritmética
   Gramática

   E → T E’

E’ → + T E’
E’ → - T E’
E’ → ξ
T → F T’
T’ → %FT’
T’ → ξ
T’ → *F T’
T’ → /FT’
F → (E)
F → a
F → n
F → <Llama>

Ejemplos:
34*_num1 + ((-12)+_val)/[calcula(63)]

4. Expresión Relacional

Gramática:
R → E<opRel>E
R → (E<opRel>E)
<opRel> → >
<opRel> → <
<opRel> → e
<opRel> → d
<opRel> → m
<opRel> → w

Ejemplos:
2 <= (_val+4)
_num1 ==18

5. Sentencias
Gramática:
<Sent> → <Asig>
<Sent> → Q
<Sent> → <Ret>
<Sent> → <Troc>
<Sent> → <Enq>
<Sent> → <Faz>
<Sent> → <Se>
<Sent> → <Para>
<Sent> → <Llama>
6. Lista de 0 más sentencias
Gramática:
<listaS> → <Sent><listaS>

<listaS> → ξ

7. Expresión lógica
   Gramática:
   <expLogica> → !<expRel>
   <expLogica> → R<expLog>
   <expLog> → <opLog>R
   <expLog> → ξ
   <expRel> → (R)
   <expRel> → E
   <opLog> → h
   <opLog> → j

   Ejemplos:
   !_num1
   !(_num2+3<18)
   _val>8 && (3<_num1)

8. Sentencias enquanto (mientras)
   Gramática:
   <Enq> → q(<expLogica>)#<listaS>#

   Ejemplo:
   enquanto (_val>6)
   #
   <listaS>
   #

9. Sentencia Bifurcación
   Gramática:
   <Se> → s(<expLogica>)#<listaS>#

   Ejemplo:
   se (_num1 >=31)
   #
   <listaS>
   #

10. Sentencia Quebrar
    Gramática:
    Q → u;

    Ejemplo:
    quebrar;
11. Sentencia Fazer-Enquanto
    Gramática:
    <Faz>→ z#<listas>#q(<expLogica>);

Ejemplo:

fazer
#
<listaS>
enquanto(<expLogica>);

12. Sentencia Retorno

Gramática:

<Ret> → t<valRet>;
<valRet> → E
<valRet> → [expCadena]
<valRet> → ξ

Ejemplos:

regresa _val*5;
regresa[“cadena %#1”];
regresa;

13. Expresión cadena

Gramática:

<expCad> → a<opCad>
<expCad> → v<opCad>
<opCad> → l <expCad>
<opCad> → h <expCad>
<opCad> → ξ

Ejemplos:

“cadena #1”
_cad1
_cad1 & _cad2
_cad2 like “otra cadena?”

14. Sentencia Para

Gramática:

<Para> → pa[n,n]#<listaS>#

Ejemplo:

para _val [1,8]
#
<listaS>
#

15. Sentencia Trocar-Caso

Gramática:

<Trocar> → b(a):#c(n){<listaS>}<casos>#

<casos> → ξ
<casos> → c(n){<listaS>}<casos>
<casos> → (){<listaS>}<casos>

Ejemplo:

trocar(_num1):
#
caso(n) { <listaS>}
….
( ) { <listaS>}
#

16. Llamada a una función

Gramática:

<Llama> → [a(<listP>)]
<listP> → ξ
<listP> → E<Param>
<listP> → [<expCad>]<Param>
<Param> →,<listP>
<Param> → ξ

Ejemplo:

[fun_uno(x,8)]

17. Funciones

Gramática:

<Func> → <TipoF>a(<listArg>){<Cuerpo >}
<TipoF> → <Tipo>
<TipoF> → g
<listArg> → <Tipo>a<otroArg>
<listArg> → ξ
<otroArg>→,<Tipo>a<otroArg>
<otroArg> → ξ
<Cuerpo> → <listDecl><listaS>
<listDecl> → ξ
<listDecl> → <Decl><listDecl>

Ejemplo:

enteiro _fun1 (flutuador _string1)
{
<listaDecl>
<listaS>
}

El analizador tendrá como entrada un archivo con el programa fuente, dicho programa será indicado desde la línea de comandos al momento de ejecutar el analizador.

El programa realizará el análisis léxico y sintáctico. El analizador léxico deberá generar además de los tokens, la cadena de átomos que será la entrada del analizador sintáctico. Los átomos se pueden ir generando a la par que los tokens, para irlos almacenando en una sola cadena.

Los tokens generados tendrán la siguiente estructura con dos campos:

Campo 1	Campo 2
CLASE	VALOR

Los átomos están definidos para cada componente léxico y corresponden a los elementos terminales de la gramática.

CLASE	DESCRIPCIÓN	ÁTOMO
0	Operadores aritméticos	Mismo símbolo
1	Operadores lógicos	Tabla átomos Op. Lógicos
2	Operadores relacionales	Tabla átomos Op. Relacionales
3	Constantes numéricas enteras	n
4	Palabras reservadas	Tabla átomos Reservadas
5	Identificadores	a
6	Símbolos especiales	Mismo símbolo
7	Operadores de asignación	Tabla átomos Op. Asignación
8	Constantes de cadenas	v
9	Operadores sobre cadenas	Tabla átomos Op. Cadenas

VALOR	PALABRA RESERVADA	ÁTOMO
0	assinado	g
1	caso	c
2	enquanto	q
3	fazer	z
4	flutuador	f
5	inteiro	i
6	para	p
7	quebrar	u
8	retorno	t
9	se	s
10	trocar	b

VALOR	OP. RELACIONAL	ÁTOMO
0	==	e
1	!=	d
2	>	<
3	<	>
4	>=	m
5	<=	w

VALOR	OP. ASIGNACIÓN	ÁTOMO
0	=	=

1	+=	x
2	-=	y
3	*=	k
4	/=	r
5	%=	o

VALOR	OP. LÓGICO	ÁTOMO
0	&&	h
1	||	j
2	!	!

VALOR	OP. CADENAS	ÁTOMO
0	&	&
1	like	l
2		

Cada que se encuentre un error sintáctico, deberá mostrarse indicando qué es lo que se esperaba. Además, se indicará en que átomo de la cadena se encontró.

Al finalizar, el analizador léxico-sintáctico deberá mostrar tabla de símbolos, literales, tokens y cadena de átomos.

Propuesta de solución

Se plantea un programa en lex/flex que con base en las expresiones regulares que satisfacen cada clase de componente léxico, al momento de reconocer un componente léxico, en las reglas de operación realice ciertas funciones de búsqueda e inserción. De búsqueda, si se trata de un componente léxico que se encuentra en un catálogo o tabla, de tal forma que devuelva su posición; de inserción, si se trata de un componente léxico que deba agregarse a la tabla de literales o símbolos.

Por otro lado, cuando un componente léxico es reconocido, luego de ser agregado o encontrado su valor en su respectiva tabla, catálogo o de acuerdo con lo planteado en el problema; se agrega su clase y valor a una lista que contendrá todos los tokens de acuerdo con el orden en que fueron encontrados; por otro lado, su átomo se agregará a una cadena de átomos final.

Lo anterior se logró a partir del uso de **estructuras, listas ligadas, arreglos de estructuras, funciones inserción y búsqueda, constructores, funciones auxiliares, variables globales, definición de expresiones regulares y reglas de traducción.**

Análisis

De forma conjunta, ambos integrantes del equipo realizaron el análisis de la problemática a resolver, una vez comprendida, propusimos soluciones para su resolución de forma modular,

subdividiendo el problema de forma que pudiésemos realizar avances y modificaciones en el programa de manera independiente hasta cierto punto.

Carlos se encargó de hacer las modificaciones necesarias al analizador léxico que ya se había trabajado para poder generar la cadena de átomos. Y al mismo tiempo en las reglas de traducción, se agregaron las funciones necesarias para generar la cadena de átomos.

De forma conjunta ambos integrantes realizaron el cálculo de los conjuntos de selección de cada producción acordada en clase para determinar si se trataba de una gramática LL(1).

Cinthya realizó las modificaciones a las producciones en dónde se habían encontrado errores, de tal forma que si fuera una gramática (LL).

Una vez se determinó que se podía construir el analizador sintáctico descendente, se analizó que tipo de producción se tenía y se realizó la codificación de la función de cada una de estas producciones, tomando en cuenta sus conjuntos de selección. De igual forma, se creó la función para mostrar el mensaje de error que se solicita.

Diseño e implementación

Para el diseño del analizador sintáctico, se toma en cuenta la cadena de átomos entregada y generada durante la etapa de análisis léxico.

Para la generación de esta cadena de átomos, primero, se agregó a la estructura de los componentes léxicos el miembro **char átomo**, se creó la función *addCadenaFinal()*, la cual recibe el átomo del componente léxico encontrado y lo agrega a un arreglo de caracteres, este arreglo se va redimensionando de tal manera que ocupa únicamente el espacio necesario de acuerdo al número de átomos.

En base a esta cadena, se crearon las funciones para las producciones para ellos se tomó en cuenta los conjuntos de selección.

Conjuntos de selección.

Los pasos para el cálculo de los conjuntos de selección de las producciones fueron:

1. Identificación de no-terminales y producciones anulables

Producciones anulables	**No-terminales anulables**
3	<otraFunc>
8	<listArg>
10	<otroArg>
12	<listDecl>
18	<valorIni>
20	<listaVar>

33	E’
38	T’
62	<listaS>
66	<expLog>
78	<valRet>
83	<opCad>
86	<casos>
90	<listP>
94	<Param>

2. Cálculo del conjunto First para cada producción y no-terminal

Conjunto first de la producción	**Conjunto first de los no terminales**
First(1)=First(<Func>)={f i g}	First(<Program>)={f i g}
First(2)=First(<Func>)={f i g}	First(<otraFunc>)={f i g}
First(3)={ }	
First(4)=First(<TipoF>)={f i g}	First(<Func>)={f i g}
First(5)=First(<Tipo>)={f i}	First(<TipoF>)={f i g}
First(6)={g}	
First(7)=First(<Tipo>)={f i}	First(<listArg>)={f i}
First(8)={ }	
First(9)={ , }	First(<otroArg>)={ , }
First(10)={ }	
First(11)=First(<listDecl>)={f i}	First(<Cuerpo>)={f i}
First(12)={ }	First(<listDecl>)={f i}
First(13)=First(<Decl>)={f i}	
First(14)=First(<Tipo>)={f i}	First(<Decl>)={f i}
First(15)={f}	First(<Tipo>)={f i}
First(16)={i}	
First(17)={ = }	First(<valorIni>)={ = }
First(18)={ }	
First(19)={ , }	First(<listaVar>)={ , }
First(20)={ }	
First(21)={n}	First(<tipoVal>)={n v}
First(22)={v}	
First(23)={a}	First(<Asig>)={a}
First(24)={=}	First(<opArit>)={ = x y k r o }
First(25)={x}	
First(26)={y}	

First(27)={k}	
First(28)={r}	
First(29)={o}	
First(30)=First(T)={ (a n x [}	First(E)={ (a n x [}
First(31)={+}	First(E')={+ - }
First(32)={-}	
First(33)={ }	
First(34)=First(F)={ (a n x [}	First(T)={ (a n x [}
First(35)={ * }	First(T')={ * / %}
First(36)={ / }	
First(37)={%}	
First(38)={ }	
First(39)={ (}	First(F)={ (a n x [}
First(40)={a}	
First(41)={n}	
First(43)=First(<Llama>)={ [}	
First(44)=First(E)={ (a n x [}	First(R)={ (a n x [}
First(46)={>}	First(<opRel>)={> < e d m w}
First(47)={<}	
First(48)={e}	
First(49)={d}	
First(50)={m}	
First(51)={w}	
First(52)=First(<Asig>)={a}	First(<Sent>)={a u t b q z s p [}
First(53)=First(Q)={u}	
First(54)=First(<Ret>)={t}	
First(55)=First(<Troc>)={b}	
First(56)=First(<Enq>)={q}	
First(57)=First(<Faz>)={z}	
First(58)=First(<Se>)={s}	
First(59)=First(<Para>)={p}	
First(60)=First(<Llama>)={ [}	
First(61)=First(<Sent>)={a u t b q z s p [}	First(<listaS>)={a u t b q z s p [}
First(62)={ }	
First(63)={ ! }	First(<expLogica>)={ ! (a n x [}
First(64)=First(R)={ (a n x [}	
First(65)=First(<opLog>)={ h j }	First(<expLog>)={ h j }
First(66)={ }	

First(67)={ { }	First(<expRel>)={ (a n x [}
First(68)=First(E)={ (a n x [}	
First(69)={h}	First(<opLog>)={ h j }
First(70)={ j }	
First(71)={q}	First(<Enq>)={q}
First(72)={s}	First(<Se>)={s}
First(73)={u}	First(Q)={u}
First(74)={z}	First(<Faz>)={z}
First(75)={ t }	First(<Ret>)={ t }
First(76)=First(E)={ (a n x [}	First(<valRet>)={ (a n x [v }
First(77)={ { }	
First(78)={ }	
First(79)={a}	First(<expCad>)={a v}
First(80)={v}	
First(81)={ l }	First(<opCad>)={ l h }
First(80)={h}	
First(83)={ }	
First(84)={p}	First(<Para>)={p}
First(85)={b}	First(<Trocar>)={b}
First(86)={ }	First(<casos>)={ c (}
First(87)={c}	
First(88)={ (}	
First(89)={ [}	First(<Llama>)={ [}
First(90)={ }	First(<listP>)={ (a n x [}
First(91)=First(E)={ (a n x [}	
First(92)={ { }	
First(93)={ , }	First(<Param>)={ , }
First(94)={ }	

3. Cálculo del conjunto Follow para cada no-terminal anulable

No-terminales anulables	**Conjuntos follow de los no terminales anulables**
<otraFunc>	Follow(<otraFunc>)=Follow(Program)U{⊣}={ }U{⊣}={⊣}
<listArg>	Follow(<listArg>)={) }
<otroArg>	Follow(<otroArg>)=Follow(<listArg>)={) }
<listDecl>	Follow(<listDecl>)=First(<listaS>)UFollow(<Cuerpo>)={a u t b q z s p [}U{ } }={a u t b q z s p [} }
<valorIni>	Follow(<valorIni>)=First(<listaVar>)U{ ; } ={ , }U{ ; }={ , ; }

<listaVar>	Follow(<listaVar>)={ ; }
E'	Follow(E')=Follow(E)={ ;) > < e d m w h j , }
T'	Follow(T')=Follow(T)=First(E')UFollow(E)UFollow(E')=First(E')UFollow(E) ={+ -}U{ ;) > < e d m w h j , } = {+ - ;) > < e d m w h j , }
<listaS>	Follow(<listaS>)=Follow(<Cuerpo>)U{#}U{ } }={ } }U{#}U{ } }={ } # }
<expLog>	Follow(<expLog>)=Follow(<expLogica>)U{) }={) }U{) }={) }
<valRet>	Follow(<valRet>)={ ; }
<opCad>	Follow(<opCad>)=Follow(<expCad>)={] }
<casos>	Follow(<casos>)={ # }
<listP>	Follow(<listP>)={) }UFollow(<Param>)={) }
<Param>	Follow(<Param>)=Follow(<listP>)={) }

Otros conjuntos follow calculados
Follow(Program)={ }
Follow(<Cuerpo>)={ } }
Follow(E)={ ; }U{) }UFirst(<opRel>)UFollow(R)UFollow(<expRel>)UFollow(<valRet>)UFirst(<Param>)UFollow(<listP>)
Follow(E)={ ; }U{) }U{> < e d m w}U{h j) }U{) }U{ ; }U{ , }U{) }={ ;) > < e d m w h j , }
Follow(R)=First(<expLog>)UFollow(<expLogica>)UFollow(<expLog>)U{) }={h j }U{) }U{) }={h j) }
Follow(<expLogica>)={) }
Follow(<expRel>)=Follow(<expLogica>)={) }
Follow(<expCad>)={] }

4. Obtención de los conjuntos de selección

No.	**Producción**	**Conjunto solución de la producción**	**¿Disjuntos?**
1	<Program> → <Func><otraFunc>	c.s.(1)={f i g}	✓
2	<otraFunc> →<Func><otraFunc>	c.s.(2)={f i g}	✓
3	<otraFunc> → ξ	c.s.(3)={ ⊣ }	
4	<Func> → <TipoF>a(<listArg>){<Cuerpo >}	c.s.(4)={f i g}	✓
5	<TipoF> → <Tipo>	c.s.(5)={f i}	✓
6	<TipoF> → g	c.s.(6)={ g }	
7	<listArg> → <Tipo>a<otroArg>	c.s.(7)={f i}	✓
8	<listArg> → ξ	c.s.(8)={) }	
9	<otroArg>→,<Tipo>a<otroArg>	c.s.(9)={ , }	✓
10	<otroArg> → ξ	c.s.(10)={) }	
11	<Cuerpo> → <listDecl><listaS>	c.s.(11)={f i}	✓

12	<listDecl> → ξ	c.s.(12)={a u t b q z s p [} }	✓
13	<listDecl> → <Decl><listDecl>	c.s.(13)={f i}	
14	<Decl> → <Tipo>a<valorIni><listaVar>;	c.s.(14)={f i}	✓
15	<Tipo> → f	c.s.(15)={ f }	✓
16	<Tipo> → i	c.s.(16)={ i }	
17	<valorIni> →=<tipoVal>	c.s.(17)={ = }	✓
18	<valorIni> → ξ	c.s.(18)={ , ; }	
19	<listaVar>→ ,a <valorIni><listaVar>	c.s.(19)={ , }	✓
20	<listaVar> → ξ	c.s.(20)={ ; }	
21	<tipoVal> → n	c.s.(21)={n}	✓
22	<tipoVal> → v	c.s.(22)={v}	
23	<Asig> → a<opArit>E;	c.s.(23)={a}	✓
24	<opArit> → =	c.s.(24)={ = }	✓
25	<opArit> → x	c.s.(25)={x}	
26	<opArit> → y	c.s.(26)={y}	
27	<opArit> → k	c.s.(27)={k}	
28	<opArit> → r	c.s.(28)={r}	
29	<opArit> → o	c.s.(29)={o}	
30	E → T E’	c.s.(30)={ (a n x [}	✓
31	E’ → + T E’	c.s.(31)={+}	✓
32	E’ → - T E’	c.s.(32)={ - }	
33	E’ → ξ	c.s.(33)={ ;) > < e d m w h j , }	
34	T → F T’	c.s.(34)={ (a n x [}	✓
35	T’ → *FT’	c.s.(35)={ * }	✓
36	T’ → /FT’	c.s.(36)={ / }	
37	T’ → %FT’	c.s.(37)={%}	
38	T’ → ξ	c.s.(38)={+ - ;) > < e d m w h j , }	
39	F → (E)	c.s.(39)={ (}	✓
40	F → a	c.s.(40)={a}	
41	F → n	c.s.(41)={n}	
42	F → <Llama>	c.s.(42)={ [}	
43	R → E<opRel>E	c.s.(43)={ (a n x [}	✓
44	<opRel> → >	c.s.(44)={>}	✓
45	<opRel> → <	c.s.(45)={<}	
46	<opRel> → e	c.s.(46)={e}	
47	<opRel> → d	c.s.(47)={d}	
48	<opRel> → m	c.s.(48)={m}	

49	<opRel> → w	c.s.(49)={w}	
50	<Sent> → <Asig>	c.s.(50)={a}	✓
51	<Sent> → Q	c.s.(51)={u}	
52	<Sent> → <Ret>	c.s.(52)={t}	
53	<Sent> → <Troc>	c.s.(53)={b}	
54	<Sent> → <Enq>	c.s.(54)={q}	
55	<Sent> → <Faz>	c.s.(55)={z}	
56	<Sent> → <Se>	c.s.(56)={s}	
57	<Sent> → <Para>	c.s.(57)={p}	
58	<Sent> → <Llama>	c.s.(58)={ [}	
59	<listaS> → <Sent><listaS>	c.s.(59)={a u t b q z s p [}	✓
60	<listaS> → ξ	c.s.(60)={ } # }	
61	<expLogica> → !<expRel>	c.s.(61)={ ! }	✓
62	<expLogica> → R<expLog>	c.s.(62)={ (a n x [}	
63	<expLog> → <opLog>R	c.s.(63)={ h j }	✓
64	<expLog> → ξ	c.s.(64)={) }	
65	<expRel> → {R}	c.s.(65)={ { }	✓
66	<expRel> → E	c.s.(66)={ (a n x [}	
67	<opLog> → h	c.s.(67)={h}	✓
68	<opLog> → j	c.s.(68)={ j }	
69	<Enq> → q(<expLogica>)#<listaS>#	c.s.(69)={q}	✓
70	<Se> → s(<expLogica>)#<listaS>#	c.s.(70)={s}	✓
71	Q → u;	c.s.(71)={u}	✓
72	<Faz>→ z#<listas>#q(<expLogica>);	c.s.(72)={z}	✓
73	<Ret> → t<valRet>;	c.s.(73)={ t }	✓
74	<valRet> → E	c.s.(74)={ (a n x [}	✓
75	<valRet> → {<expCad>}	c.s.(75)={ { }	
76	<valRet> → ξ	c.s.(76)={ ; }	
77	<expCad> → a<opCad>	c.s.(77)={a}	✓
78	<expCad> → v<opCad>	c.s.(78)={v}	
79	<opCad> → l <expCad>	c.s.(79)={ l }	✓
80	<opCad> → h <expCad>	c.s.(80)={h}	
81	<opCad> → ξ	c.s.(81)={] }	
82	<Para> → pa[n,n]#<listaS>#	c.s.(82)={p}	✓
83	<Trocar> → b(a):#c(n){<listaS>}<casos>#	c.s.(83)={b}	✓
84	<casos> → ξ	c.s.(84)={ # }	✓
85	<casos> → c(n){<listaS>}<casos>	c.s.(85)={c}	
86	<casos> → (){<listaS>}<casos>	c.s.(86)={ (}	

87	<Llama> → [a(<listP>)]	c.s.(87)={ [}	✓
88	<listP> → ξ	c.s.(88)={) }	✓
89	<listP> → E<Param>	c.s.(89)={ (a n x [}	
90	<listP> → {<expCad>}<Param>	c.s.(90)={ { }	
91	<Param> → ,<listP>	c.s.(91)={ , }	✓
92	<Param> → ξ	c.s.(92)={) }	

Construcción del Analizador Sintáctico Descendente Recursivo

Se elaboraron funciones escritas en C, para cada no-terminal. Cada función será codificada de acuerdo con el tipo de producción, a continuación, se presentan los 4 casos:

Caso	Tipo de producción	Código
1	i : A -> b	c= getchar() return
2	i : A -> bα	c= getchar() procesar(α) return
3	i : A -> ξ	return
4	i : A -> Bα	procesar(Bα) return

Estructura del Programa

La estructura de nuestro programa es:

<Serie de funciones>

Cuya gramática está dada por:

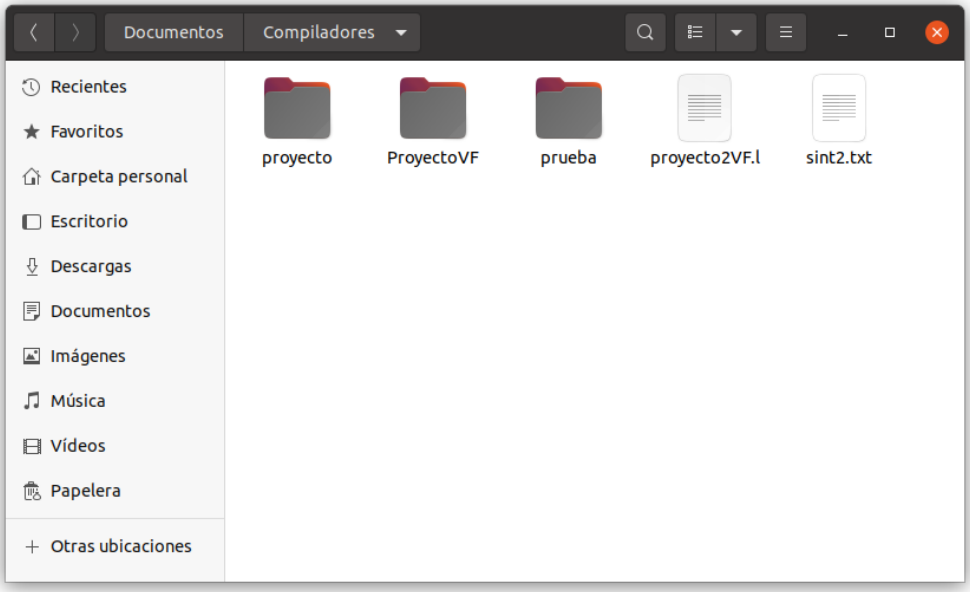
<Program> □ <Func><otraFunc>
<otraFunc> □<Func><otraFunc>
<otraFunc> □ ξ

En base a esto, una vez concluida la etapa de análisis léxico, solo queda llamar a nuestra función "AnalizadorSintactico( )", la cual empieza a leer la cadena de átomos entregada, y llama a la función "Program ( )", que es como tal la función que empieza el análisis sintáctico.

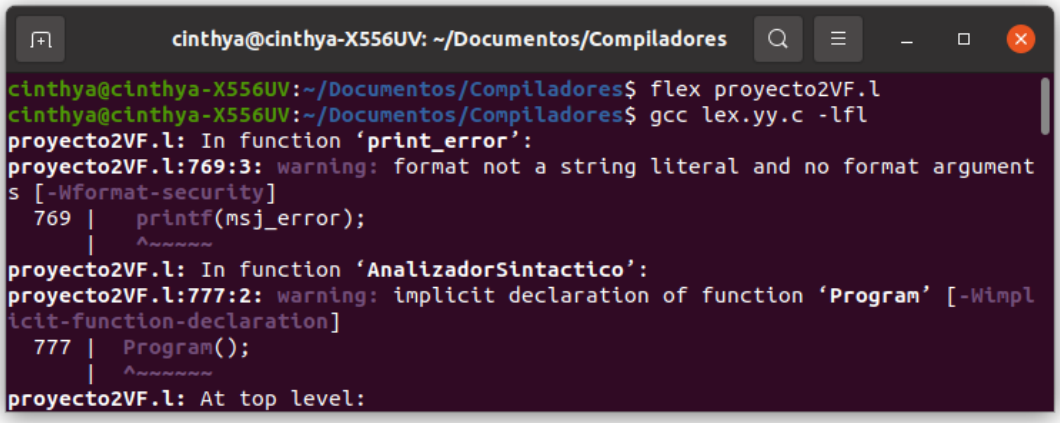
Finalmente, una vez concluido este análisis, se muestra si el programa es sintácticamente correcto o no lo es.

Instrucciones de ejecución

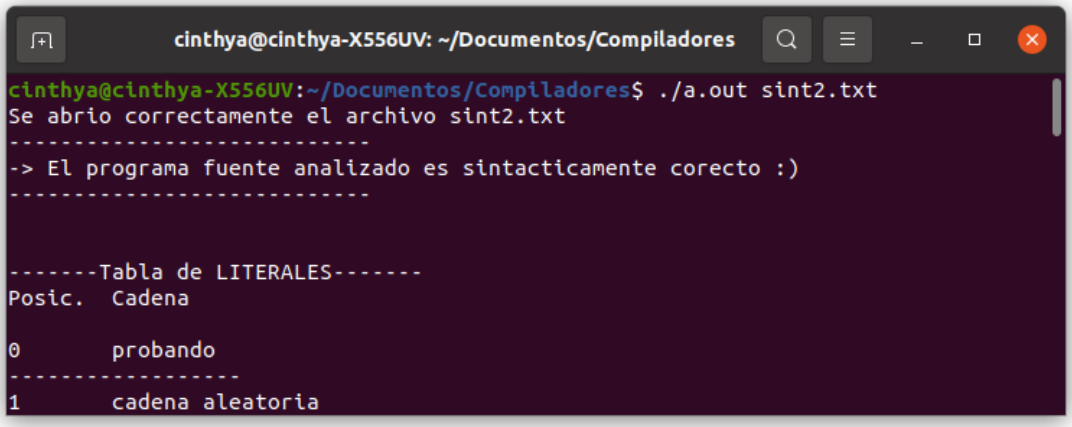
Para ejecutar el programa, se debe ubicar tanto el programa "proyecto2VF.l", correspondiente al analizador sintáctico, como el programa a analizar con extensión ".txt" dentro del mismo directorio.



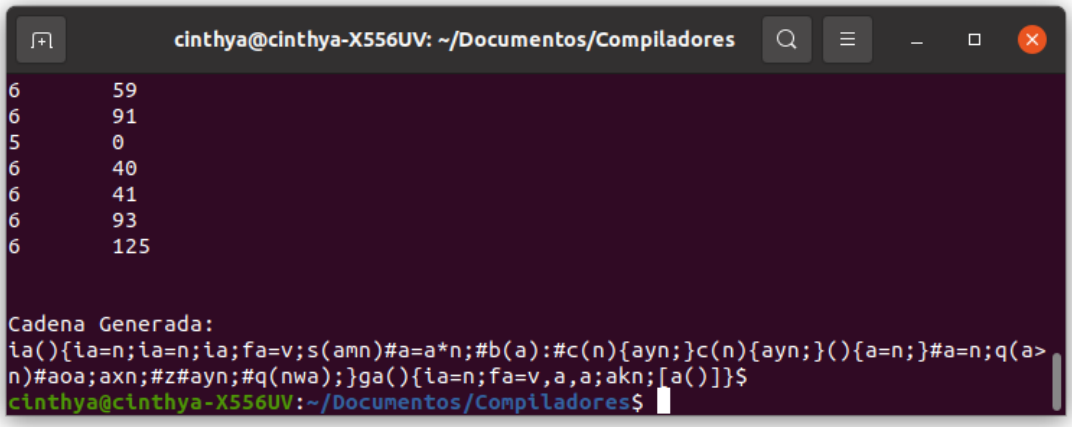
Desde la terminal se accede al directorio en el que se ubica el programa y se ingresa el comando "flex proyecto2VF.l" para realizar la compilación del programa ".l". Posteriormente se ingresa el comando "gcc lex.yy.c -lfl" para la compilación del programa en C.



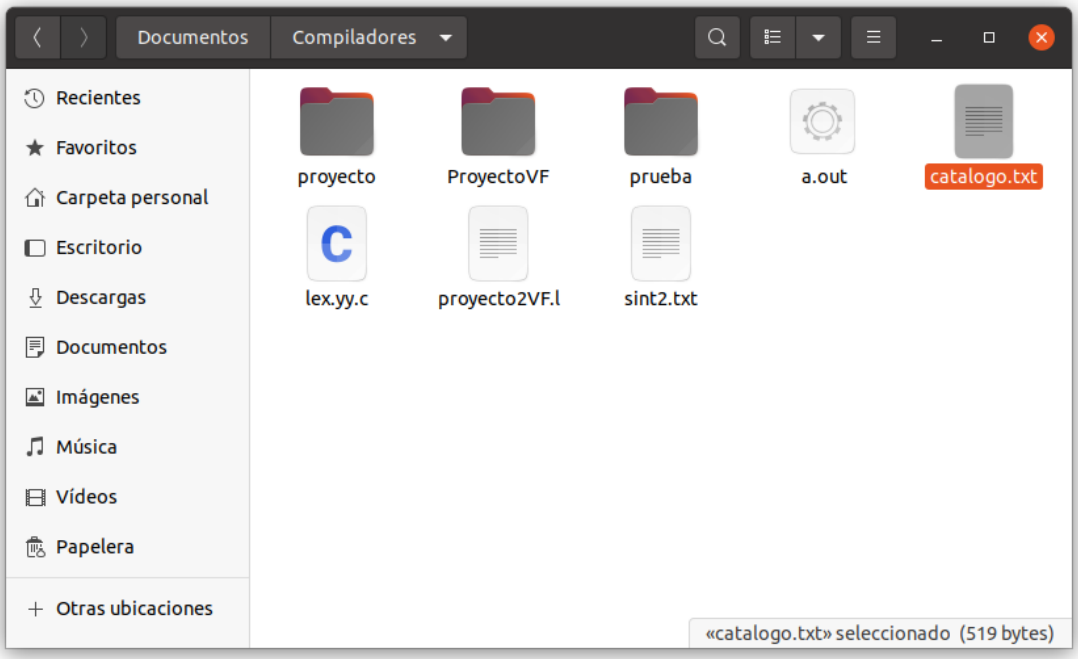
Para su ejecución se deberá ingresar el comando "./a.out <nombre_del_programa>.txt", donde <nombre_del_programa> se sustituye por el nombre del archivo que contenga el programa que se desea analizar.



Se mostrará en pantalla la salida correspondiente a los errores tanto léxicos como sintácticos, la indicación de si el programa está correcto sintácticamente, la tabla de literales, de símbolos y la de tokens, así como la cadena de átomos.



Adicionalmente, en la carpeta en la que se ubique el programa se creara un archivo llamado “catalogo.txt” que contiene los catálogos creados para los componentes léxicos, con los que trabaja el programa.



Conclusiones

Morales Ortega Carlos

Gracias al presente proyecto se ha aprendido a realizar un analizador sintáctico descendente recursivo, además el hecho de desarrollarlo, implico saber a detalle cómo funciona, y más que eso el análisis para saber si se tiene una gramática adecuada, ya que si no se tenía una gramática LL o los conjuntos de selección eran incorrectos, entonces el analizador sintáctico no serviría.

Por lo tanto, se reforzó el conocimiento adquirido en clase y lo teórico lo plasmamos de manera teórica, esto resultó bastante interesante, ya que, al construir nuestro propio analizador, recurrimos a nuestros conocimientos acerca de este tema y de materias anteriores.

La parte compleja y medular del proyecto, consiste en la construcción de los conjuntos de selección era esencial, una vez asegurándonos que, si pertenece a una gramática LL, la elaboración de las funciones para cada producción no llevo mayor complejidad.

Finalmente, el proyecto obligó a investigar más allá de lo que ya hemos aprendido y además refleja nuestra creatividad para poder solucionar el problema propuesto o bien abordar el problema y entregar una solución.

Vélez Grande Cinthya

Mediante el desarrollo del presente proyecto fue posible aplicar todos los conocimientos adquiridos en teoría con respecto a la implementación de un analizador sintáctico descendente recursivo. Fue fundamental tener los conocimientos referentes al tipo de gramática con el que se realizaría la implementación del analizador, en este caso, la gramática LL(1), la cual fue adecuadamente analizada para determinar si podría ser útil para implementar el analizador.

Adicionalmente, con el análisis de la gramática pudimos obtener los conjuntos de selección, la correcta obtención de estos sería crucial para la codificación del analizador, puesto que, de acuerdo a los símbolos que conformaran los conjuntos se sabría qué camino tomar dentro de las funciones asociadas a las producciones para cada elemento no terminal de la gramática.

El desarrollo de este analizador nos permitió entender la razón por la cual el analizador es recursivo, puesto que en el diseño de las funciones se emplean funciones recursivas, que tendrán como casos base aquellas producciones que conducen a la cadena vacía, es decir, las que hacen anulables las producciones. Con base en ello, se puede decir que fue fundamental aplicar conocimientos de asignaturas previas relacionadas con programación.

Con esto, se concluye que el proyecto de desarrolló de manera satisfactoria, cumpliendo el objetivo inicial planteado de manera adecuada, pues se hizo uso de todos los conocimientos adquiridos en el curso para un adecuado diseño e implementación de un analizador sintáctico descendente recursivo.