Universidad Nacional Autónoma de México

Facultad de Ingeniería

Segundo Programa: Analizador Sintactico.

Alumnos:
- García López Erik
- Serapio Hernández Alexis Arturo

Materia: Compiladores

Profesora: M.C. Laura Sandoval Montaño

Semestre 2024-1

Fecha de Entrega: 09/11/2023

● Objetivo:

Construir, en un mismo programa, los analizadores Léxico y Sintáctico Descendente Recursivo que revisen programas escritos en el lenguaje definido por la gramática del Anexo A de este documento.

● Descripción del Programa:

- La entrada es un archivo con el programa fuente a analizar que deberá estar escrito en el lenguaje definido por la gramática del Anexo A de este documento. Este archivo de entrada se indicará desde la línea de comandos.
- El programa realizará tanto el análisis léxico como el 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, pero irlos almacenando en una sola cadena.
- Los átomos están definidos en este documento por cada componente léxico y corresponden a los elementos terminales de la gramática.
- La tabla de clases de componentes léxicos con sus correspondientes átomos es:

Clase	Descripción	átomo
0	Operadores aritméticos + - / * %	mismo símbolo
1	Operadores lógicos (ver tabla).	(ver tabla)
2	Operadores relacionales (ver tabla).	(ver tabla)
3	Constantes numéricas enteras. Sólo en base 10. Si está con signo, encerrarlo entre ( ). Ejemplos: 0 672 (-265) (+49)	n
4	Palabras reservadas (ver tabla).	(ver tabla)
5	Identificadores. Inician con _ y le sigue una letra minúscula o mayúscula, después pueden contener letras minúsculas o mayúsculas, dígitos y _	a
6	Símbolos especiales ( ) { } ; , [ ] : #	mismo símbolo
7	Operadores de asignación (ver tabla).	(ver tabla)
8	Constantes cadenas. Encerradas entre comillas (“) cualquier secuencia de caracteres incluye salto de línea.	v
9	Operadores sobre cadenas (ver tabla).	(ver tabla)

- El valor en los tokens y los átomos se indican en las siguientes tablas.

Valor	Palabra reservada	Equiv. en C	átomo
0	assinado	void	g
1	caso	case	c
2	enquanto	while	q
3	fazer	do	z
4	flutuador	float	f
5	inteiro	int	i
6	para	for	p
7	quebrar	break	u
8	retorno	return	t
9	se	if	s
10	trocar	switch	b

Valor	Op. relacional	átomo
0	==	e
1	!=	d
2	>	>
3	<	<
4	>=	m
5	<=	w

Valor	Op. Asig.	átomo
0	=	=
1	+=	x
2	-=	y
3	*=	k
4	/=	r
5	%=	o

Valor	Op. Sobre cadenas	átomo
0	&	&
1	like	l

Valor	Op. lógico	átomo
0	&&	h
1	||	j
2	!	!

- El analizador sintáctico deberá mostrar todos los errores sintácticos que encuentre, indicando qué se esperaba.

- Como resultados, el analizador léxico-sintáctico deberá mostrar el contenido de la tabla de símbolos, las tablas de literales, los tokens y la cadena de átomos. Finalmente deberá indicar si está sintácticamente correcto el programa fuente.

- Los errores que vaya encontrando el analizador léxico, los podrá ir mostrando en pantalla o escribirlos en un archivo, así como él o los errores sintácticos. Es conveniente que cuando encuentre un error sintáctico se indique en qué átomo de la cadena se encontró (con ubicación).

- El programa deberá estar comentado, con una descripción breve de lo que hace (puede ser el objetivo indicado en este documento), el nombre de quienes elaboraron el programa y fecha de elaboración.

- ## Descripción del Problema:

Para este segundo programa se pedía elaborar un analizador sintáctico, específicamente un analizador sintáctico descendente recursivo. Lo primero que hace es recibir una cadena de átomos que proviene del analizador léxico que fue creada a partir de los tokens, debido a esto fue lo primero que se modificó en el programa. Los átomos están definidos por cada uno de los componentes léxicos de la gramática.

Además de lo anterior, son imprescindibles los conjuntos de selección de cada una de las producciones de la gramática; para calcular los conjuntos de selección es recomendable primero calcular todos los conjuntos first y los conjuntos follow para tenerlos a la mano cuando se necesiten, posterior a eso hay que verificar que los conjuntos de selección pertenecientes a una gramática sean conjuntos disjuntos para que se cumpla que sea una gramática LL(1) y así poder hacer el recorrido recursivo del analizador sintáctico descendente recursivo.

- **Sintaxis de las diferentes estructuras del lenguaje**

Sintaxis de las diferentes estructuras del lenguaje

1) Sentencia declarativa:

Ejemplos:

enteiro _val=2, _num1, _num2;

flutuador _cad1="cadena #1", _cad2;

Gramática:

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

2) Sentencias de asignación:

Ejemplos:

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

Gramática:

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

3) Expresión aritmética

Ejemplos:

34*_num1 + ((-12)+_val)/[calcula(63)]
18

Gramática:

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

4) Expresión relacional

Ejemplos:

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

Gramática:

R → E<opRel>E
<opRel> → >
<opRel> → <
<opRel> → e
<opRel> → d
<opRel> → m
<opRel> → w

5) Sentencias

<Sent> → <Asig>
<Sent> → Q
<Sent> → <Ret>
<Sent> → <Trocar>
<Sent> → <Enq>
<Sent> → <Faz>
<Sent> → <Se>
<Sent> → <Para>
<Sent> → <Llama>

6) Lista de 0 más sentencias:

<listaS> → <Sent><listaS>
<listaS> → ξ

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

<expLogica> → !<expRel>
<expLogica> → R<expLog>
<expLog> → <opLog>R
<expLog> → ξ
<expRel> → {R}
<expRel> → E
<opLog> → h
<opLog> → j

8) Sentencia enquanto (mientras)
Ejemplo:
enquanto (_val>6)
#
<listaS>
#

Gramática:
<Enq> → q(<expLogica>)#<listaS>#

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

Gramática:
<Se> → s(<expLogica>)#<listaS>#

10) Sentencia quebrar
Ejemplo:
quebrar;

Gramática
Q → u;

11) Sentencia fazer-enquanto
Ejemplo:
fazer
#
<listaS>
enquanto(<expLogica>);

Gramática:
<Faz>→ z#<listas>#q(<expLogica>);

12) Sentencia retorno
Ejemplos:
regresa _val*5;
regresa[“cadena %#1”];
regresa;

Gramática:
<Ret> → t<valRet>;
<valRet> → E
<valRet> → {<expCad>}
<valRet> → ξ

13) Expresión cadena
Ejemplos:
“cadena #1”
_cad1
_cad1 & _cad2
_cad2 like “otra cadena?”

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

14) Sentencia para
Ejemplo:
para _val [1,8]
#
<listaS>
#
Gramática:
<Para> → pa[n,n]#<listaS>#

15) Sentencia trocar-caso

Ejemplo:

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

Gramática:

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

16) Llamada a una función

Ejemplo:

```
[fun_uno(x,8)]
```

Gramática:

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

17) Funciones

Ejemplo:

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

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>

Estructura del Programa:

<Serie de funciones>

Gramática:

<Program> → <Func><otraFunc>
<otraFunc> →<Func><otraFunc>
<otraFunc> → ξ

- **Gramática del lenguaje**

1:	<Program> →<Func><otraFunc>
2:	<otraFunc> →<Func><otraFunc>
3:	<otraFunc> → ξ
4:	<Func> → <TipoF>a(<listArg>){<Cuerpo >}
5:	<TipoF> → <Tipo>
6:	<TipoF> → g
7:	<listArg> → <Tipo>a<otroArg>
8:	<listArg> → ξ
9:	<otroArg>→,<Tipo>a<otroArg>
10:	<otroArg> → ξ
11:	<Cuerpo> → <listDecl><listaS>
12:	<listDecl> → ξ
13:	<listDecl> → <Decl><listDecl>
14:	<Decl> → <Tipo>a<valorIni><listaVar>;
15:	<Tipo>→ f
16:	<Tipo>→ i
17:	<valorIni>→=<tipoVal>
18:	<valorIni> → ξ
19:	<listaVar>→ ,a<valorIni><listaVar>
20:	<listaVar> → ξ
21:	<tipoVal> → n
22:	<tipoVal> → v
23:	<Asig> → a<opArit>E;
24:	<opArit> → =
25:	<opArit> → x
26:	<opArit> → y
27:	<opArit> → k
28:	<opArit> → r
29:	<opArit> → o
30:	E → T E'
31:	E' → + T E'
32:	E' → - T E'
33:	E' → ξ
34:	T → F T'
35:	T' → *F T'
36:	T' → /FT'
37:	T' → %FT'
38:	T' → ξ
39:	F → (E)
40:	F → a
41:	F → n

42:	F → <Llama>
43:	R → E<opRel>E
44:	<opRel> → >
45:	<opRel> → <
46:	<opRel> → e
47:	<opRel> → d
48:	<opRel> → m
49:	<opRel> → w
50:	<Sent> → <Asig>
51:	<Sent> → Q
52:	<Sent> → <Ret>
53:	<Sent> → <Trocar>
54:	<Sent> → <Enq>
55:	<Sent> → <Faz>
56:	<Sent> → <Se>
57:	<Sent> → <Para>
58:	<Sent> → <Llama>
59:	<listaS> → <Sent><listaS>
60:	<listaS> → ξ
61:	<expLogica> → !<expRel>
62:	<expLogica> → R<expLog>
63:	<expLog> → <opLog>R
64:	<expLog> → ξ
65:	<expRel> → {R}
66:	<expRel> → E
67:	<opLog> → h
68:	<opLog> → j
69:	<Enq> → q(<expLogica>)#<listaS>#
70:	<Se> → s(<expLogica>)#<listaS>#
71:	Q → u;
72:	<Faz>→ z#<listas>#q(<expLogica>);
73:	<Ret> → t<valRet>;
74:	<valRet> → E
75:	<valRet> → {<expCad>}
76:	<valRet> → ξ
77:	<expCad> → a<opCad>
78:	<expCad> → v<opCad>
79:	<opCad> → l <expCad>
80:	<opCad> → h <expCad>
81:	<opCad> → ξ
82:	<Para> → pa[n,n]#<listaS>#

83:	<Trocar> → b(a):#c(n){<listaS>}<casos>#
84:	<casos> → ξ
85:	<casos> → c(n){<listaS>}<casos>
86:	<casos> → (){<listaS>}<casos>
87:	<Llama> →[a(<listP>)]

88:	<listP> → ξ
89:	<listP> → E<Param>
90:	<listP> →{<expCad>}<Param>
91:	<Param> → ,<listP>
92:	<Param> → ξ

- **Conjunto First de cada producción**

First(1) = { fig }
First(2) = { fig }
First(3) = { }
First(4) = { fig }
First(5) = { fi }
First(6) = { g }
First(7) = { fi }
First(8) = { }
First(9) = { , }
First(10) = { }
First(11) = { fi }
First(12) = { }
First(13) = { fi }
First(14) = { fi }
First(15) = { f }
First(16) = { i }
First(17) = { = }
First(18) = { }
First(19) = { , }
First(20) = { }
First(21) = { n }
First(22) = { v }
First(23) = { a }
First(24) = { = }
First(25) = { x }
First(26) = { y }
First(27) = { k }
First(28) = { r }
First(29) = { o }
First(30) = { (an[}
First(31) = { + }
First(32) = { - }
First(33) = { }
First(34) = { (an[}
First(35) = { * }
First(36) = { / }
First(37) = { % }
First(38) = { }

First(39) = { (}
First(40) = { a }
First(41) = { n }
First(42) = { [}
First(43) = { (an[}
First(44) = { > }
First(45) = { < }
First(46) = { e }
First(47) = { d }
First(48) = { m }
First(49) = { w }
First(50) = { a }
First(51) = { u }
First(52) = { t }
First(53) = { b }
First(54) = { q }
First(55) = { z }
First(56) = { s }
First(57) = { p }
First(58) = { [}
First(59) = { autbqzsp[}
First(60) = { }
First(61) = { ! }
First(62) = { (an[}
First(63) = { hj }
First(64) = { }
First(65) = { { }
First(66) = { (an[}
First(67) = { h }
First(68) = { j }
First(69) = { q }
First(70) = { s }
First(71) = { u }
First(72) = { z }
First(73) = { t }
First(74) = { (an[}
First(75) = { { }
First(76) = { }
First(77) = { a }
First(78) = { v }

First(79) = { l }
First(80) = { h }
First(81) = { }
First(82) = { p }
First(83) = { b }
First(84) = { }
First(85) = { c }
First(86) = { (}
First(87) = { [}
First(88) = { }
First(89) = { (an[}
First(90) = { { }
First(91) = { , }
First(92) = { }

- **Conjunto First de cada No Terminal**

First(<Program>) = { fig }
First(<otraFunc>) = { fig }
First(<Func>) = { fig }
First(<TipoF>) = { fi }
First(<listArg>) = { fi }
First(<otroArg>) = { , }
First(<Cuerpo>) = { fi }
First(<listDecl>) = { fi }
First(<Decl>) = { fi }
First(<Tipo>) = { fi }
First(<valorIni>) = { = }
First(<listaVar>) = { , }
First(<tipoVal>) = { nv }
First(<Asig>) = { a }
First(<opArit>) = { =xykro }
First(E) = { (an[}
First(E') = { +- }
First(T) = { (an[}
First(T') = { */% }
First(F) = { (an[}
First(R) = { (an[}
First(<opRel>) = { ><edmw }
First(<Sent>) = { autbqzsp[}
First(<listaS>) = { autbqzsp[}

First(<expLogica>) = { !(an[}
First(<expLog>) = { hj }
First(<expRel>) = { (an[}
First(<opLog>) = { hj }
First(<Enq>) = { q }
First(<Se>) = { s }
First(Q) = { u }
First(<Faz>) = { z }
First(<Ret>) = { t }
First(<valRet>) = { (an[}
First(<expCad>) = { av }
First(<opCad>) = { lh }
First(<Para>) = { p }
First(<Trocar>) = { b }
First(<casos>) = { c(}
First(<Llama>) = { [}
First(<listP>) = { (an[{ }
First(<Param>) = { , }

- **Conjuntos follow de terminales anulables y otros requeridos:**

Follow(<Program>) = {}
Follow(<otraFunc>) = { Follow(Program) U ⊣ } = { ⊣ }
Follow(<listArg>) = {) }
Follow(<otroArg>) = { Follow(<listArg>) } = {) }
Follow(<listDecl>) = { First(<listaS>) } = { autbqzsp[}
Follow(<Cuerpo>) = { } }
Follow(<Func>) = { First(<otraFunc>) U Follow(Program) U Follow(<otraFunc>) }
= { {fig} U {} U {⊣} } = { fig⊣ }
Follow(<valorIni>) = { First(<listaVar>) U ; U Follow(<listaVar>)} = { ,; }
Follow(<listaVar>) = { ; }
Follow(<expLogica>) = {) }
Follow(<expLog>) = { Follow(<expLogica>) } = {) }
Follow(R) = { First(<expLog>) U Follow(<expLogica>) U Follow(<expLog>) U }
= {hj} U {)} U {)} U {}} = { hj)} }
Follow(<expRel>) = { Follow(<expLogica>) } = {) }
Follow(<valRet>) = { ; }
Follow(<listP>) = {) }
Follow(<Param>) = { Follow(<ListP>) } = {) }

Follow(E) = { ; U) U First(<opRel>) U Follow(R) U Follow(<expRel>) U
Follow(<valRet>) U First(<Param>) U Follow(<listP>) }
= { {;} U {)} U {><edmw} U {hj)} U {)} U {;} U {,} U{)}} = { ;)><edmwhj, }
Follow(E') = { Follow(E) } = { ;)><edmwhj, }
Follow(T) = { First(E') U Follow (E) U Follow(E') } = {+-} U {;)><edmwhj,} U
{;)><edmwhj,}
= {+-;)><edmwhj,}
Follow(T') = { Follow(T) } = {+-;)><edmwhj,}
Follow(<listaS>) = { Follow(<Cuerpo>) U { # } U { } } U } = { }# }
Follow(<expCad>) = { } }
Follow(<opCad>) = { }
Follow(<casos>) = { # }

- **Conjuntos de selección:**

Producciones anulables: <otraFunc>, <listArg>, <otroArg>, <listDecl>, <valorIni>, <listaVar>, E', T', <listaS>, <expLog>, <valRet>, <opCad>, <casos>, <listP>, <Param>

C.S.(1)={fig}
C.S.(2)={fig}
C.S.(3)={ ł }
C.S.(4)={fig}
C.S.(5)={fi}
C.S.(6)={g}
C.S.(7)={fi}
C.S.(8)={)}
C.S.(9)={,}
C.S.(10)={)}
C.S.(11)={fiautbqzsp[}
C.S.(12)={autbqzsp[}
C.S.(13)={fi}
C.S.(14)={fi}
C.S.(15)={f}
C.S.(16)={i}
C.S.(17)={=}
C.S.(18)={,;}
C.S.(19)={,}
C.S.(20)={;}
C.S.(21)={n}

C.S.(22)={v}
C.S.(23)={a}
C.S.(24)={=}
C.S.(25)={x}
C.S.(26)={y}
C.S.(27)={k}
C.S.(28)={r}
C.S.(29)={o}
C.S.(30)={(an[}
C.S.(31)={+}
C.S.(32)={-}
C.S.(33)={;)><edmwhj,}
C.S.(34)={(an[}
C.S.(35)={*}
C.S.(36)={/}
C.S.(37)={%}
C.S.(38)={+-;)><edmwhj,}
C.S.(39)={(}
C.S.(40)={a}
C.S.(41)={n}
C.S.(42)={[}
C.S.(43)={(an[}
C.S.(44)={>}
C.S.(45)={<}
C.S.(46)={e}
C.S.(47)={d}
C.S.(48)={m}
C.S.(49)={w}
C.S.(50)={a}
C.S.(51)={u}
C.S.(52)={t}
C.S.(53)={b}
C.S.(54)={q}
C.S.(55)={z}
C.S.(56)={s}
C.S.(57)={p}
C.S.(58)={[}
C.S.(59)={autbqzsp[}
C.S.(60)={}#}
C.S.(61)={!}

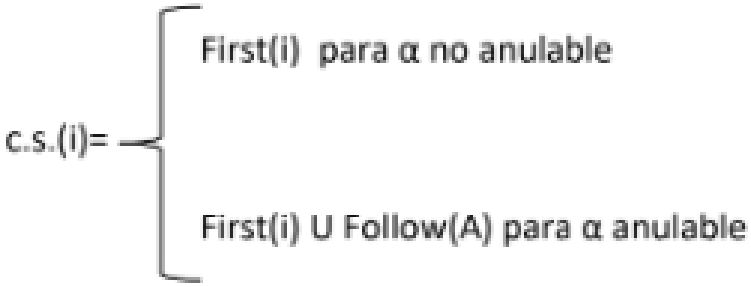
C.S.(62)={(an[}
C.S.(63)={hj}
C.S.(64)={)}
C.S.(65)={{}
C.S.(66)={(an[}
C.S.(67)={h}
C.S.(68)={j}
C.S.(69)={q}
C.S.(70)={s}
C.S.(71)={u}
C.S.(72)={z}
C.S.(73)={t}
C.S.(74)={(an[}
C.S.(75)={{}
C.S.(76)={;}
C.S.(77)={a}
C.S.(78)={v}
C.S.(79)={l}
C.S.(80)={h}
C.S.(81)={] }
C.S.(82)={p}
C.S.(83)={b}
C.S.(84)={#}
C.S.(85)={c}
C.S.(86)={(}
C.S.(87)={[}
C.S.(88)={)}
C.S.(89)={(an[}
C.S.(90)={{}
C.S.(91)={,}
C.S.(92)={)}

Observando podemos ver que se cumple que el conjunto de selección sea disjunto de un mismo no terminal, por lo que sí es una gramática LL(1)

● Propuesta de solución y fases del desarrollo:

- **Obtener conjuntos de selección (Garcia Lopez Erik)**

Antes de construir el analizador sintáctico, se debe realizar una serie de pasos. En primer lugar, es esencial calcular los conjuntos de selección de la gramática que define la estructura del lenguaje que estamos analizando. Para llevar a cabo este proceso, se requiere identificar las producciones que pueden ser nulas, los no terminales que pueden ser nulos, los conjuntos First de todas las producciones y no terminales, así como los conjuntos Follow de todos los no terminales que pueden ser nulos. Luego, se aplica la regla siguiente a los conjuntos de selección.



Una producción anulable se refiere a una producción que genera cadenas vacías, es decir, cadenas epsilon, de forma directa o indirecta. Una vez que los conjuntos de selección han sido calculados, es fundamental asegurarse de que estos sean mutuamente excluyentes en las producciones de un mismo no terminal. Dado que esta condición se cumple, podemos concluir que la gramática es LL(1), lo que nos permite avanzar en la construcción del analizador sintáctico.

- **Obtener cadena de átomos (Garcia Lopez Erik)**

Para llevar a cabo este proceso, se inicia creando un archivo específico para almacenar exclusivamente la secuencia de átomos. Dependiendo de la expresión regular detectada en el archivo de entrada, se genera el átomo correspondiente. La generación de átomos se realiza de dos maneras diferentes en las reglas de traducción. Por ejemplo, si nos encontramos en la sección de palabras reservadas o de símbolos especiales, se crean funciones que determinan el tipo de entrada y, en función de eso, añaden el átomo correspondiente al archivo de secuencia de átomos. En algunas ocasiones, los átomos se generan en las mismas funciones donde se crean los tokens, mientras que en otras situaciones, se desarrollan funciones independientes para incorporar los átomos. Por otro lado, si los átomos son únicos, como en el caso de los identificadores, constantes numéricas enteras o constantes cadenas, se insertan los átomos directamente en el archivo dentro de las mismas acciones de las expresiones regulares.

- **Creación de funciones para cada no terminal (Serapio Hernández Alexis Arturo)**

Para la creación de funciones en nuestro analizador sintáctico, hicimos uso de la siguiente guía proporcionada por la profesora:

Caso	Tipo 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

Donde, procesar(α) y procesar(Bα) consiste en:

1) Por cada no-terminal en α y para B, llamar a la función que le corresponde.
2) Por cada terminal:

```
if (c == 'x')
    c= getchar( );
else
    rechaza();
```

donde $x \in \Sigma_p$

Dado un tipo de producción de acuerdo a los tipos que observamos en la tabla anterior podemos avanzar de una u otra forma en la cadena de átomos.

El getchar(); que se usa de ejemplo en la imagen, puede sustituirse por algunas funciones adicionales o una estructura de datos.
En nuestro caso, hicimos uso de un arreglo que almacena caracter por caracter al momento de recorrer el analizador

Las funciones las declaramos de la manera siguiente:

```
579   void Program();
580   void otraFunc();
581   void Func();
582   void TipoF();
583   void listArg();
584   void otroArg();
585   void Cuerpo();
586   void listDecl();
587   void Decl();
588   void Tipo();
589   void valorIni();
590   void listaVar();
591   void tipoVal();
592   void Asig();
593   void opArit();
594   void E();
595   void EP();
596   void T();
597   void TP();
598   void F();
599   void R();
600   void opRel();
601   void Sent();
602   void listaS();
603   void expLogica();
604   void expLog();
605   void expRel();
606   void opLog();
607   void Enq();
608   void Se();
609   void Q();
610   void Faz();
611   void Ret();
```

Finalmente estas funciones retornan a la primera función declarada llamada “program” dentro de la cual podemos observar las salidas de nuestro analizador sintáctico.
Se nos arrojará un mensaje diciendo que la cadena fue aceptada o en su defecto un mensaje que muestra que la cadena fue rechazada y los errores que esta tuvo al momento de su análisis.

En la salida podemos observar entonces algo como lo siguiente:

```
------------------------Analizador sintactico------------------------

Se genero la cadena de atomos en el archivo cadena.txt
La cadena de atomos es: ia(fa){fa=n;}$

Error sintactico
Caracter: =   Posicion: 9   Esperado: a | u | t | b | q | z | s | p | [ | f | i
```

- **Detección de errores sintácticos (Serapio Hernández Alexis Arturo):**

Los errores sintácticos se detectan gracias a nuestro arreglo llamado atomos[].

En este arreglo, se almacenan uno a uno los caracteres que encuentra el analizador sintáctico.

Cuando un elemento dentro de este arreglo no coincide con lo que se esperaba en la cadena de átomos, llamamos a la función rechaza(), al llamar a esta función automáticamente toma el elemento que no coincide con nuestra cadena de átomos y lo retorna imprimiendo en pantalla las características que tenía el analizador, la cadena a analizar y la posición dentro de la cadena de átomos indicando que existe un error en dicha declaración.

Como podemos ver en la siguiente función opLog(), si el arreglo átomos[] contiene un carácter que coincide con el conjunto de selección indicado (h, j), el analizador continua su ejecución. En caso de que el analizador caiga en esta función y no encuentre un carácter que coincida, se manda a llamar a la función rechaza y se imprimen los valores esperados.

```
void opLog(){
    if(c=='h'||c=='j'){
        posicion++;
        c=atomos[posicion];
        return;
    }
    else{
        rechaza("h || j ");
        return;
    }
}
```

Sin embargo, la función rechaza() permite que el analizador se siga ejecutando para detectar todos los errores sintácticos e imprimirlos en conjunto, por lo tanto gracias a esta función podemos observar una salida como la siguiente:

```
------------------------Analizador sintactico------------------------

Se genero la cadena de atomos en el archivo cadena.txt
La cadena de atomos es: ia(fa){fa=n;}$

Error sintactico
Caracter: =   Posicion: 9   Esperado: a | u | t | b | q | z | s | p | [ | f | i
```

Cuando una cadena de átomos es aceptada, es decir que encuentra el símbolo de fin de cadena, nos arroja como resultado un print que nos indica que la cadena es aceptada tanto lexica como sintacticamente y el programa concluye exitosamente.

```
Los tokens muestran en el archivo tokens.txt
-----------------------Analizador sintactico-----------------------

Se genero la cadena de atomos en el archivo cadena.txt
La cadena de atomos es: fa(fa,fa){fa=n;a=a+n*(n-n);a=n;a=a+a;}$

La cadena es aceptada (esta bien lexicamente y sintacticamente)
Es sintácticamente correcto
erik@erik-VirtualBox:~/Escritorio/Sintactico$
```

Por lo tanto para la detección de errores tenemos estas dos posibilidades donde el programa se acepta solo si recorre toda la cadena de átomos de manera correcta.

- **Indicaciones de como correr el programa**

Dentro de una terminal de linux debemos estar en la ruta donde se encuentra nuestro archivo. Una vez hecho lo anterior escribimos los siguientes comandos:

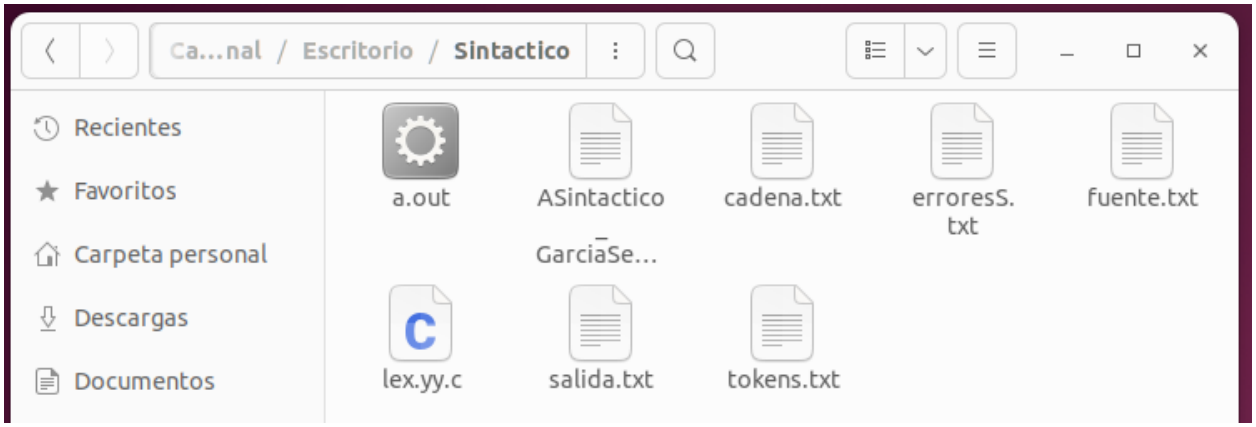
flex ASintactico_GarciaSerapio.l
gcc lex.yy.c -lfl -o salida.out
./a.out <entrada.txt

entrada.txt es el nombre del archivo que va a analizar, puede llevar cualquier nombre siempre y cuando tenga la extensión “.txt”.

Ejemplo:

```
erik@erik-VirtualBox:~/Escritorio/Sintactico$ flex ASintactico_GarciaSerapio.l
erik@erik-VirtualBox:~/Escritorio/Sintactico$ gcc lex.yy.c -lfl
erik@erik-VirtualBox:~/Escritorio/Sintactico$ ./a.out <fuente.txt
```

Despues apareceran los archivos en la carpeta:



- ## Conclusiones

    - ### García López Erik:
      Realizar este programa ayudó a que practicara mucho a sacar los conjuntos first, follow y los conjuntos de selección ya que la gramática estaba larga, pienso que este proceso fue muy tardado ya que había que ser muy cuidadoso, de lo contrario todo el programa iba a estar mal.
      En cuanto a la programación para generar la cadena de átomos fue sencilla, pero para las funciones recursivas se complicó un poco ya que a veces colocamos returns que no iban o a veces nos hacía falta ponerlos, lo cual hacía que el programa no funcionara como debía y hacía dudar de su funcionamiento, a veces pensaba que era porque habia algun conjunto de seleccion mal calculado, finalmente nos dimos cuenta que ese era el error.
      De igual forma tuvimos que realizar muchas pruebas de escritorio cuando el programa no funcionaba como queríamos pero resultaba laborioso ya que las funciones son recursivas

    - ### Serapio Hernández Alexis Arturo:
      Gracias a este proyecto pude comprender de mejor manera la manera en la que funciona un compilador y específicamente un analizador sintáctico. Comprendí la importancia de la obtención de los conjuntos de selección para este analizador ya que son los que “restringen” lo que el analizador realmente quiere dar a entender y permite la correcta lógica tanto del código como de la programación del usuario. Implemente y aplique la importancia de la generación de la cadena de átomos debido a que es importante que el programa analizador sepa lo que debe de esperar para evitar errores en la mala escritura del mismo.Personalmente se nos complicó la implementación de las funciones en el código y esto mismo nos orillo a tener que detenernos y realizar una prueba de escritorio paso a paso para entender completamente si habia algun error en las conexiones de nuestras funciones debido a la complejidad de la gramática, sin embargo considero que se llegó a un nivel deseable.