Documentación Analizador Sintáctico

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 problema.

Para este segundo proyecto/programa de la materia de compiladores, tuvimos que elaborar un analizador sintáctico. Éste debería poder usar una cadena de átomos, generada por el analizador léxico a través de los tokens, y determinar si la cadena representa un programa válido para la gramática definida. Los átomos representan a los elementos terminales de la gramática y están definidos con cada uno de los componentes léxicos del lenguaje. Para el analizador sintáctico debemos utilizar el método de analizado sintáctico descendente recursivo e implementarlo a nuestro programa del analizador léxico previamente elaborado.

Para poder crear un analizador sintáctico descendente recursivo, lo primero que requerimos es conocer el conjunto de selección de cada producción de la gramática (la obtención se muestra en las siguientes páginas del documento). Para obtenerlas a su vez se tendrá que obtener el conjunto First de cada producción y el conjunto Follow de los no terminales anulables. Una vez se tengan los conjuntos de selección y se haya probado que la gramática es LL(1) se puede pasar a codificar. La forma de pasar el analizador a programación es utilizando la pila de la memoria del programa. Aquí se puede modelar un autómata finito determinista con pila mediante funciones recursivas. Habrá una función por cada no terminal, y dentro de la función una sección de código por cada producción del no terminal.

Las especificaciones de entrada definidas para el analizador léxico se siguen manteniendo. El programa deberá recibir un archivo a través de la línea de comandos al ser ejecutado. Por otro lado, para la salida le aumentaremos un archivo extra, el de “CadenaAtomos.txt” quedando un total de 6 archivos de salida de la ejecución:

- CadenaAtomos.txt
- Tokens.txt
- TablaSimbolos.txt
- TablaLiteralesCadenas.txt
- TablaLiteralesNumeros.txt
- Errores.txt (tanto léxicos como sintácticos)

Gramática del Lenguaje:

1:	<Programa> → <ListaD>[<SerieF>]
2:	<ListaD> → D<ListaD>
3:	<ListaD> → ξ
4:	<SerieF> → <Func><otraF>
5:	<otraF> → <Func><otraF>
6:	<otraF> → ξ
7:	D → <Tipo>L:
8:	<Tipo> → g
9:	<Tipo> → n
10:	<Tipo> → d
11:	<Tipo> → h
12:	L → a<Valor> l’
13:	l’ → ,a<Valor> l’
14:	l’ → ξ
15:	<Valor> → = V
16:	<Valor> → ξ
17:	V → c
18:	V → s
19:	V → z
20:	V → r
21:	A → a A’:
22:	A’ → = E
23:	A’ → m E
24:	A’ → k E
25:	A’ → p E
26:	A’ → t E
27:	A’ → u E
28:	A → a A’:
29:	I → i [R] F’
30:	F’ → (S)B
31:	B → e(S)
32:	B→ ξ
33:	I → i [R] F’
34:	W → w[R](S)
35:	<For> → f [E](S)

36:	<Return> → bZ
37:	Z → [E]:
38:	Z → :
39:	E → TE’
40:	E’ → +TE’
41:	E’ → -TE’
42:	E’ → ξ
43:	T → FT’
44:	T’ → *FT’
45:	T’ → /FT’
46:	T’ → $FT’
47:	T’ → ξ
48:	F → (E)
49:	F → a
50:	F → z
51:	F → r
52:	F → [a]
53:	R → ER’
54:	R’ → >E
55:	R’ → <E
56:	R’ → ?E
57:	R’ → yE
58:	R’ → lE
59:	R’ → ¿E
60:	S → S’<otraS>
61:	S’ → A
62:	S’ → I
63:	S’ → W
64:	S’ → <For>
65:	S’ → <Return>
66:	S’ → [a]:
67:	<otraS> → S’<otraS>
68:	<otraS> → ξ
69:	<Func> → <TipoFun>a(<ListaD>S)
70:	<TipoFun> → <Tipo>
71:	<TipoFun> → v

Proceso de obtención de los conjuntos de selección de la gramática

a) Obtener producciones anulables.

No terminales con producción ξ :

3:	<ListaD> → ξ
6:	<otraF> → ξ
14:	I' → ξ
16:	<Valor> → ξ
32:	B→ ξ
42:	E' → ξ
47:	T' → ξ
68:	<otraS> → ξ

No terminales con producción de puros no terminales:

2:	<ListaD> → D<ListaD>
4:	<SerieF> → <Func><otraF>
5:	<otraF> → <Func><otraF>
39:	E → TE'
43:	T → FT'
53:	R → ER'
60:	S → S'<otraS>
61:	S' → A
62:	S' → I
63:	S' → W
64:	S' → <For>
65:	S' → <Return>
67:	<otraS> → S'<otraS>
70:	<TipoFun> → <Tipo>

Sin embargo, ninguno de estas últimas producciones está formada únicamente por no terminales anulables. Por lo que no se incluirán en la lista de anulables.

Los no terminales anulables son:

<ListaD>
<otraF>
I'
<Valor>
B
E'
T'
<otraS>

b) Obtener el conjunto First de cada producción y cada no terminal.

Conjuntos First inconcluso de cada producción:

First(1)=	***First(2,3) + { [}***
First(2)=	***First(7)***
First(3)=	{ }
First(4)=	***First(69)***
First(5)=	***First(69)***
First(6)=	{ }
First(7)=	***First(8, 9, 10, 11)***
First(8)=	{ g }
First(9)=	{ n }
First(10)=	{ d }
First(11)=	{ h }
First(12)=	{ a }
First(13)=	{ , }
First(14)=	{ }
First(15)=	{ = }
First(16)=	{ }
First(17)=	{ c }
First(18)=	{ s }
First(19)=	{ z }
First(20)=	{ r }
First(21)=	{ a }
First(22)=	{ = }
First(23)=	{ m }
First(24)=	{ k }
First(25)=	{ p }
First(26)=	{ t }
First(27)=	{ u }
First(28)=	{ a }
First(29)=	{ i }
First(30)=	{ (}
First(31)=	{ e }
First(32)=	{ }
First(33)=	{ i }
First(34)=	{ w }
First(35)=	{ f }

First(36)=	{ b }
First(37)=	{ [}
First(38)=	{ : }
First(39)=	***First(43)***
First(40)=	{ + }
First(41)=	{ - }
First(42)=	{ }
First(43)=	***First(48, 49, 50, 51, 52)***
First(44)=	{ * }
First(45)=	{ / }
First(46)=	{ $ }
First(47)=	{ }
First(48)=	{ (}
First(49)=	{ a }
First(50)=	{ z }
First(51)=	{ r }
First(52)=	{ [}
First(53)=	***First(39)***
First(54)=	{ > }
First(55)=	{ < }
First(56)=	{ ? }
First(57)=	{ y }
First(58)=	{ l }
First(59)=	{ ¿ }
First(60)=	***First(61, 62, 63, 64, 65, 66)***
First(61)=	***First(21)***
First(62)=	***First(29)***
First(63)=	***First(34)***
First(64)=	***First(35)***
First(65)=	***First(36)***
First(66)=	{ [}
First(67)=	***First(61, 62, 63, 64, 65, 66)***
First(68)=	{ }
First(69)=	***First(70, 71)***
First(70)=	***First(8, 9, 10, 11)***
First(71)=	{ v }

Resolviendo para los que son en función de otro First obtenemos los First para todas las producciones:

Conjuntos First de cada producción:

First(1)=	{ g n d h [}
First(2)=	{ g n d h }
First(3)=	{ }
First(4)=	{ g n d h v }
First(5)=	{ g n d h v }
First(6)=	{ }
First(7)=	{ g n d h }
First(8)=	{ g }
First(9)=	{ n }
First(10)=	{ d }
First(11)=	{ h }
First(12)=	{ a }
First(13)=	{ , }
First(14)=	{ }
First(15)=	{ = }
First(16)=	{ }
First(17)=	{ c }
First(18)=	{ s }
First(19)=	{ z }
First(20)=	{ r }
First(21)=	{ a }
First(22)=	{ = }
First(23)=	{ m }
First(24)=	{ k }
First(25)=	{ p }
First(26)=	{ t }
First(27)=	{ u }
First(28)=	{ a }
First(29)=	{ i }
First(30)=	{ (}
First(31)=	{ e }
First(32)=	{ }
First(33)=	{ i }
First(34)=	{ w }
First(35)=	{ f }

First(36)=	{ b }
First(37)=	{ [}
First(38)=	{ : }
First(39)=	{ (a z r [}
First(40)=	{ + }
First(41)=	{ - }
First(42)=	{ }
First(43)=	{ (a z r [}
First(44)=	{ * }
First(45)=	{ / }
First(46)=	{ $ }
First(47)=	{ }
First(48)=	{ (}
First(49)=	{ a }
First(50)=	{ z }
First(51)=	{ r }
First(52)=	{ [}
First(53)=	{ (a z r [}
First(54)=	{ > }
First(55)=	{ < }
First(56)=	{ ? }
First(57)=	{ y }
First(58)=	{ l }
First(59)=	{ ¿ }
First(60)=	{ a i w f b [}
First(61)=	{ a }
First(62)=	{ i }
First(63)=	{ w }
First(64)=	{ f }
First(65)=	{ b }
First(66)=	{ [}
First(67)=	{ a i w f b [}
First(68)=	{ }
First(69)=	{ g n d h v }
First(70)=	{ g n d h }
First(71)=	{ v }

Conjuntos First de cada no terminal de la gramática:

First(<Programa>)=	{ g n d h [}
First(<ListaD>)=	{ g n d h }
First(<SerieF>)=	{ g n d h v }
First(<otraF>)=	{ g n d h v }
First(D)=	{ g n d h }
First(<Tipo>)=	{ g n d h }
First(L)=	{ a }
First(l')=	{ , }
First(<Valor>)=	{ = }
First(V)=	{ c s z r }
First(A)=	{ a }
First(A')=	{ = m k p t u }
First(I)=	{ i }
First(F')=	{ (}
First(B)=	{ e }
First(W)=	{ w }
First(<For>)=	{ f }
First(<Return>)=	{ b }
First(Z)=	{ [: }
First(E)=	{ (a z r [}
First(E')=	{ + - }
First(T)=	{ (a z r [}
First(T')=	{ * / $ }
First(F)=	{ (a z r [}
First(R)=	{ (a z r [}
First(R')=	{ > < ? y l ¿ }
First(S)=	{ a i w f b [}
First(S')=	{ a i w f b [}
First(<otraS>)=	{ a i w f b [}
First(<Func>)=	{ g n d h v }
First(<TipoFun>)=	{ g n d h v }

c) Obtener el conjunto Follow de cada no-terminal anulable.

Recordando, los no terminales anulables son:

<ListaD>
<otraF>
l'
<Valor>
B
E'
T'
<otraS>

Para <ListaD>:

1:	<Programa> → <ListaD>[<SerieF>]
2:	<ListaD> → D<ListaD>
69:	<Func> → <TipoFun>a(<ListaD>S)

Follow(<listaD>) = { [} + First(S)

Follow(<listaD>) = { [a i w f b }

Para <otraF>:

4:	<SerieF> → <Func><otraF>
5:	<otraF> → <Func><otraF>

Follow(<otraF>) = Follow(<SerieF>)

Para <SerieF>:

1:	<Programa> → <ListaD>[<SerieF>]

Follow(<serieF>) = {] }

Follow(<otraF>) = {] }

Para l' :

12:	L → a<Valor> l'
13:	l' → ,a<Valor> l'

Follow(l') = Follow (L)

Para L:

7:	D → <Tipo>L:

Follow(L) = { : }

Follow (l') = { : }

Para <Valor>:

12:	L → a<Valor> l'
13:	l' → ,a<Valor> l'

Follow(< Valor>) = First(l') + Follow(l')

Follow (<Valor>) = { , : }

Para B:

30:	F' → (S)B

Follow(B) = Follow (F')

Para F':

29:	I → i [R] F'

Follow(F') = Follow(I)

Para I:

62:	S' → I

Follow(I) = Follow(S')

Para S':

60:	S → S'<otraS>
67:	<otraS> → S'<otraS>

Follow(S') = First(<otraS>) + Follow (S) + Follow (<otraS>)

Para S:

30:	F' → (S)B
31:	B → e(S)
34:	W → w[R](S)
35:	<For> → f [E](S)
69:	<Func> → <TipoFun>a(<listaD>S)

Follow(S) = {) }

Para <otraS>:

60:	S → S'<otraS>
67:	<otraS> → S'<otraS>

Follow(<otraS>) = Follow(S) = {) }

Follow(S') = { a i w f b [ ) }

Follow(I) = { a i w f b [ ) }

Follow(F') = { a i w f b [ ) }

Follow(B) = { a i w f b [ ) }

Para E':

39:	E → TE'
40:	E' → +TE'
41:	E' → -TE'

Para E':

Follow(E') = Follow (E)

Para E:

22:	A' → = E
23:	A' → m E
24:	A' → k E
25:	A' → p E
26:	A' → t E
27:	A' → u E
35:	<For> → f [E](S)
37:	Z → [E]:
48:	F → (E)
53:	R → ER'
54:	R' → >E
55:	R' → <E
56:	R' → ?E
57:	R' → yE
58:	R' → lE
59:	R' → ¿E

Follow(E) = Follow(A') + {] } + {) } + First(R') + Follow(R')

Para A':

21:	A → a A':
28:	A → a A':

Follow(A') = { : }

Para R':

53:	R → ER'

Follow(R') = Follow (R)

Para R:

29:	I → i [R] F'
33:	I → i [R] F'
34:	W → w[R](S)

Follow (R) = {] }

Follow(R') = {] }

Follow(E) = { :]) > < ? y I ¿ }

Follow(E') = { :]) > < ? y I ¿ }

Para T':

43:	T → FT'
44:	T' → *FT'
45:	T' → /FT'
46:	T' → $FT'

Follow(T') = Follow(T)

Para T:

39:	E → TE'
40:	E' → +TE'
41:	E' → -TE'

Follow(T) = First(E') + Follow(E')

Follow(T) = { + - }

Follow(T') = { + - :]) > < ? y l ¿ }

Para <otraS>:

60:	S → S'<otraS>
67:	<otraS> → S'<otraS>

Follow(<otraS>) = Follow(S)

Para S:

30:	F' → (S)B
31:	B → e(S)
34:	W → w[R](S)
35:	<For> → f [E](S)
69:	<Func> → <TipoFun>a(<listaD>S)

Follow(S) = {) }

Follow(<otraS>) = {) }

En la siguiente table se tienen los conjuntos Follow de todos los no terminables anulables:

Follow(<ListaD>) =	{ [a i w f b }
Follow(<otraF>) =	{] }
Follow(I') =	{ : }
Follow(<Valor>) =	{ , : }
Follow(B) =	{ a i w f b [ ) }
Follow(E') =	{ :]) > < ? y l ¿ }
Follow(T') =	{ + - :]) > < ? y l ¿ }
Follow(<otraS>) =	{) }

Conjunto de selección para todas las producciones de la gramática.

c.s.(1)=	{ g n d h [}
c.s.(2)=	{ g n d h }
c.s. (3)=	{ [a i w f b }
c.s. (4)=	{ g n d h v }
c.s. (5)=	{ g n d h v }
c.s. (6)=	{] }
c.s. (7)=	{ g n d h }
c.s. (8)=	{ g }
c.s. (9)=	{ n }
c.s. (10)=	{ d }
c.s. (11)=	{ h }
c.s. (12)=	{ a }
c.s. (13)=	{ , }
c.s. (14)=	{ : }
c.s. (15)=	{ = }
c.s. (16)=	{ , : }
c.s. (17)=	{ c }
c.s. (18)=	{ s }
c.s. (19)=	{ z }
c.s. (20)=	{ r }
c.s. (21)=	{ a }
c.s. (22)=	{ = }
c.s. (23)=	{ m }
c.s. (24)=	{ k }
c.s. (25)=	{ p }
c.s. (26)=	{ t }
c.s. (27)=	{ u }
c.s. (28)=	{ a }
c.s. (29)=	{ i }
c.s. (30)=	{ (}
c.s. (31)=	{ e }
c.s. (32)=	{ a i w f b [ ) }
c.s. (33)=	{ i }
c.s. (34)=	{ w }
c.s. (35)=	{ f }

c.s. (36)=	{ b }
c.s. (37)=	{ [}
c.s. (38)=	{ : }
c.s. (39)=	{ (a z r [}
c.s. (40)=	{ + }
c.s. (41)=	{ - }
c.s. (42)=	{ :]) > < ? y l ¿ }
c.s. (43)=	{ (a z r [}
c.s. (44)=	{ * }
c.s. (45)=	{ / }
c.s. (46)=	{ $ }
c.s. (47)=	{ + - :]) > < ? y l ¿ }
c.s. (48)=	{ (}
c.s. (49)=	{ a }
c.s. (50)=	{ z }
c.s. (51)=	{ r }
c.s. (52)=	{ [}
c.s. (53)=	{ (a z r [}
c.s. (54)=	{ > }
c.s. (55)=	{ < }
c.s. (56)=	{ ? }
c.s. (57)=	{ y }
c.s. (58)=	{ l }
c.s. (59)=	{ ¿ }
c.s. (60)=	{ a i w f b [}
c.s. (61)=	{ a }
c.s. (62)=	{ i }
c.s. (63)=	{ w }
c.s. (64)=	{ f }
c.s. (65)=	{ b }
c.s. (66)=	{ [}
c.s. (67)=	{ a i w f b [}
c.s. (68)=	{) }
c.s. (69)=	{ g n d h v }
c.s. (70)=	{ g n d h }
c.s. (71)=	{ v }

Siguiendo la tabla en orden, las producciones que están junto a una del mismo color son porque son producciones del mismo no terminal y se debe cumplir que su conjunto de selección sea disjunto. Por otro lado, las filas en morado son producciones repetidas, pero para no tener que cambiar todo el orden se conservaron sin ser consideradas. Podemos comprobar que para un mismo no terminal, el conjunto de selección de todas sus producciones es disjunto.

Sí cumple con ser una gramática LL(1).

¿Cómo correr el programa?

Para correr el programa lo que debemos hacer es los siguiente:

1. Abrir la consola o terminal de su sistema Linux de preferencia. (Debe contar con flex/lex yacc instalado, al igual que gcc)

2. Moverse al directorio donde tenga el código fuente (cd “directorio”) y preferentemente el archivo que se analizará.

3. Ejecutar el siguiente comando de compilación flex:

```
flex lexicoSintactico.l
```

4. Luego ejecutar el siguiente comando de compilación de C para el archivo previamente compilado por flex que se llamará lex.yy.c:

```
gcc -lfl lex.yy.c
```

5. Finalmente ejecutar el comando de ejecución del archivo con el código objeto generado y pasando como argumento el nombre del archivo a analizar:

```
./a.out nombreArchivoParaAnalizar.txt
```

6. Con esto habrá logrado ejecutar exitosamente el programa. Se habrán mostrado los resultados en pantalla y se le habrán generado un total de 8 archivos nuevos a su carpeta, 6 de los cuales son los resultados:
    - lex.yy.c
    - a.out
    - CadenaAtomos.txt
    - Tokens.txt
    - TablaSimbolos.txt
    - TablaLiteralesCadenas.txt
    - TablaLiteralesNumeros.txt
    - Errores.txt

Conclusiones

La elaboración de esta segunda etapa de un compilador, el analizador sintáctico, ha sido un proyecto muy interesante que me ha ayudado a comprender mejor el funcionamiento de los analizadores y de la importancia de cada etapa de compilación. Al haber hecho el analizador léxico, comprendía de dónde salían los tokens y cómo se realizaba su identificación. Sin embargo, no fue hasta hacer el analizador sintáctico que comprendí su funcionalidad para crear la cadena de átomos y poder reconocer una gramática. Supongo que en el analizador semántico ya se usarán los tokens y no sólo la cadena de átomos para checar que sea correcto el programa.

A lo largo de la elaboración no me enfrenté a muchas dificultades, pero sí tuve que repasar los conceptos para comprenderlos bien y poderlos implementar. El sacar el conjunto de selección no es complicado ya que son una serie de pasos definidos, pero lo que me preocupa un poco es que en muchas ocasiones los diferentes resultados están relacionados entre sí. Por esto es posible que haya cometido un pequeño error a causa de un descuido y que éste se vaya arrastrando en los cálculos. Sin embargo, para minimizar esta posibilidad chequé múltiples veces mi procedimiento y comprobé que sí se trata de una gramática LL(1). La otra complicación que sufrí fue el comprender cuándo se debía avanzar en la cadena de átomos dentro de las funciones. En un principio estaba un poco confundido al respecto y había puesto la lectura al inicio de todas las funciones, pero finalmente pude darme cuenta de que no debía ser así y lo corregí. Sólo leyendo un nuevo carácter en determinadas secciones del código de las funciones.

Viendo el programa terminado y funcionando, estoy muy satisfecho con él. Considero que los objetivos planteados en un inicio se cumplen exitosamente y lo hace de una manera ordenada y medianamente eficiente. Si bien creo que tiene aspectos donde el código se podría optimizar, creo es una buena primera aproximación a la creación de un compilador léxico-sintáctio. Con la base que he programado hasta el momento, creo que no tendré problemas en un futuro para incluir el analizador semántico.