Universidad Nacional Autónoma de México
Facultad de Ingeniería
Compiladores
Grupo 03

Implementación de un Analizador Sintáctico Recursivo Descendente

Profesora: M.C. Laura Sandoval Montaño
Alumno: Zuriel Zárate García
Semestre 2023-1

Ciudad de México, México 8 de diciembre de 2022

Índice

1. Objetivos

- 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 del documento: "ProgAnalizadorSintactico2023-1.pdf"

2. Introducción

En el trabajo anterior, pudimos observar el funcionamiento del Análisis Léxico utilizando una poderosa herramienta: "Flex/Lex". En esta ocasión, nos enfocaremos en trabajar el análisis sintáctico, especialmente descendente y recursivo, a partir de la generación de una cadena de átomos durante la fase de análisis léxico y de una gramática libre de contexto de tipo *LL(1)*.

2.1. Planteamiento del problema

Clase	Descripción	átomo
0	Palabras reservadas (ver Tabla 2).	(ver Tabla 2)
1	Identificadores. Iniciar con '$' y le sigue al menos una letra minúscula o mayúscula. Ejemplos: $ejemplo, $Varible, $OtraVariable, $XYZ	i
2	Constantes numéricas enteras. En base 10 (secuencia de dígitos del 0-9 sin 0's a la izquierda, excepto para el número 0), en base 8 (inicien con O u o y le sigan dígitos del 0 al 7).	n
3	Constantes numéricas reales. Siempre deben llevar parte decimal y es opcional la parte entera. Ejemplos: 73.0, .0, 10.2 No aceptados: . , 12 , 4.	r
4	Constantes cadenas. Encerrado entre comillas (") cualquier secuencia de más de un carácter que no contenga " ni '. Para cadenas de un solo carácter, encerrarlo entre apóstrofos ('). La cadena de unas comillas debe ser encerrada entre apóstrofos: "'" . La cadena de un apóstrofo debe ser encerrada por comillas: "'" . No se aceptan cadenas vacías. Ejemplos NO válidos: "ejemplo no "valido" , "" , """ , "hola 'mundo"	s
5	Símbolos especiales [ ] ( ) { } , : ;	mismo símbolo
6	Operadores aritméticos + - * / % \^	mismo símbolo
7	Operadores relacionales (ver Tabla 3).	(ver Tabla 3)
8	Operador de asignación =	=

Cuadro 1: Tabla de átomos correspondientes a cada clase

Una gramática es una "máquina" capaz de producir cadenas correspondientes a un lenguaje determinado. Dicha máquina es un *autómata de pila* o *autómata push-down*. Para que esta máquina pueda reconocer cadenas, necesita generarlas utilizando producciones muy específicas. Si, utilizando

estas producciones, el autómata no es capaz de llegar a una determinada cadena, la cadena no se reconoce, es decir, no pertenece al lenguaje deseado.

El análisis sintáctico descendente recursivo se implementa partiendo del diseño de una gramática libre de contexto de la familia *LL(1)*. Este tipo de gramática debe cumplir, para todas sus variantes y todos sus subconjuntos, con tener conjuntos de selección **disjuntos** para todas las producciones de un mismo **no terminal**. Es aquí donde comienza el reto, ya que un lenguaje de programación es muy extenso y, para todas las formas en las que se puede declarar una sentencia, se necesitan muchas producciones, por ende, mucho análisis y mucho cuidado con el cálculo de los conjuntos de selección.

Para evitar escribir demasiado, se usa el concepto de "atomo", que viene a ser un caracter que representa una cadena completa, lo cual ayuda a facilitar mucho el análisis y la implementación en código. Puede observar en la Tabla (1) los átomos que se definieron en clase para representar a todas las clases de cadenas posibles.

Valor	**Palabra reservada**	**Equivale en C**	**átomo**
0	alternative	case	a
1	big	long	b
2	evaluate	if	f
3	instead	else	t
4	large	double	g
5	loop	while	w
6	make	do	m
7	number	int	#
8	other	default	o
9	real	float	x
10	repeat	for	j
11	select	switch	h
12	small	short	p
13	step	continue	c
14	stop	break	q
15	symbol	char	y
16	throw	return	z

Cuadro 2: Átomos correspondientes a las palabras reservadas

Valor	**Op. relacional**	**átomo**
0	<	<
1	>	>
2	<=	l
3	>=	u
4	==	e
5	!=	d

Cuadro 3: Átomos correspondientes a los operadores relacionales

A continuación, se muestra la gramática diseñada en clase para el análisis sintáctico descrito anteriormente:

1:	<Program> →<Func><otraFunc>
2:	<otraFunc> →<Func><otraFunc>
3:	<otraFunc> → ξ
4:	<Func> → <Tipo>i(<Param>){<Cuerpo >}
5:	<Param> → <Tipo>i<otroParam>
6:	<Param> → ξ
7:	<otroParam>→,<Tipo>i<otroParam>
8:	<otroParam> → ξ
9:	<Cuerpo> → <Decl><listaP>
10:	<Decl> → ξ
11:	<Decl> → D<Decl>
12:	D → <Tipo>K;
13:	<Tipo>→ b
14:	<Tipo>→ g
15:	<Tipo>→ #
16:	<Tipo>→ y
17:	<Tipo>→ x
18:	K → iQ
19:	Q → ξ
20:	Q → =NC
21:	Q → ,K
22:	N → n
23:	N → r
24:	N → s
25:	C → ξ
26:	C → ,K
27:	A → i=A';
28:	A' → s
29:	A' → E
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' → %FT'
39:	T' → ^FT'
40:	T' → ξ
41:	F → (E)

42:	F → i
43:	F → n
44:	F → r
45:	F → <Llama>
46:	R → iR'V
47:	R → nR'V'
48:	R → rR'V''
49:	R → sR'V'''
50:	R' → >
51:	R' → <
52:	R' → l
53:	R' → e
54:	R' → d
55:	R' → u
56:	V → i
57:	V → n
58:	V → r
59:	V → s
60:	V' → n
61:	V' → i
62:	V'' → r
63:	V'' → i
64:	V''' → s
65:	V''' → i
66:	P → A
67:	P → I
68:	P → H
69:	P → W
70:	P → J
71:	P → <Llama>
72:	P → <Devuelve>
73:	P → c;
74:	<listaP> → ξ
75:	<listaP> → P<listaP>
76:	W → w(R)m{<listaP>}
77:	I → f(R)<listaP>I':
78:	I' → t<listaP>
79:	I' → ξ
80:	J → j(YXZ{<listaP>}
81:	Y → i=E;
82:	Y → ;

83:	X → R;
84:	X → ;
85:	Z → i=E)
86:	Z →)
87:	H→ h(i){C'O'}
88:	C' → an:<listaP>UC'
89:	C' → ξ
90:	O' → o:<listaP>
91:	O' → ξ
92:	U → q

93:	U → ξ
94:	<Devuelve> → z(<valor>);
95:	<valor> → V
96:	<valor> → ξ
97:	<Llama> → [i(<arg>)]
98:	<arg> → ξ
99:	<arg> → V<otroArg>
100:	<otroArg> → ,V<otroArg>
101:	<otroArg> → ξ

3. Desarrollo

3.1. Corrección de errores de analizador léxico

Lo primero que tuvimos que hacer, antes de cualquier cosa, fue corregir los errores que tuve en el analizador léxico. En este caso, olvidé la conversión de octales a decimales, por lo que agregué la función correspondiente para lograrlo:

```
/*
     Convierte de octal a decimal. Recibe la cadena
     con la forma o<numeros> y devuelve el valor
     decimal.
*/
int octadec(char* octal){
     int i = strlen(octal)-2;
     char c;
     int ent;
     int decim = 0;
     while(i>=0){
          octal = octal+1;
          c = *octal;
          ent = c-'0';
          printf("\nn: %d*pow(8,%d)",ent,i);
          decim = decim + ent*pow(8,i);
          i--;
     }
     return decim;
}
```

Se indicó colocar las constantes reales en una tabla de literales, pero eso ya se hacía, entonces no se necesitó algún otro cambio.

3.2. Conjuntos de selección

Antes de codificar, necesitaba los conjuntos de selección, ya que era imposible sacar todos de manera mental debido a la cantidad de producciones que se tienen en la gramática. A continuación,

se muestran los conjuntos de selección que calculamos:

C.S.(1)= bg#yx
C.S.(2)= bg#yx
C.S.(3)= -|
C.S.(4)= bg#yx
C.S.(5)= bg#yx
C.S.(6)=)
C.S.(7)= ,
C.S.(8)=)
C.S.(9)= i}bg#yxcwfjhz[
C.S.(10)= i}cwfjhz[
C.S.(11)= bg#yx
C.S.(12)= bg#yx
C.S.(13)= b
C.S.(14)= g
C.S.(15)= #
C.S.(16)= y
C.S.(17)= x
C.S.(18)= i
C.S.(19)= ;
C.S.(20)= =
C.S.(21)= ,
C.S.(22)= n
C.S.(23)= r
C.S.(24)= s
C.S.(25)= ;
C.S.(26)= ,
C.S.(27)= i
C.S.(28)= s
C.S.(29)= (inr[
C.S.(30)= (inr[
C.S.(31)= +
C.S.(32)= -
C.S.(33)= ;)
C.S.(34)= (inr[
C.S.(35)= *
C.S.(36)= /
C.S.(37)=
C.S.(38)= %
C.S.(39)= ^
C.S.(40)= ;)+-
C.S.(41)= (
C.S.(42)= i
C.S.(43)= n
C.S.(44)= r

C.S.(45)= [
C.S.(46)= i
C.S.(47)= n
C.S.(48)= r
C.S.(49)= s
C.S.(50)= >
C.S.(51)= <
C.S.(52)= l
C.S.(53)= e
C.S.(54)= d
C.S.(55)= u
C.S.(56)= i
C.S.(57)= n
C.S.(58)= r
C.S.(59)= s
C.S.(60)= n
C.S.(61)= i
C.S.(62)= r
C.S.(63)= i
C.S.(64)= s
C.S.(65)= i
C.S.(66)= i
C.S.(67)= p
C.S.(68)= h
C.S.(69)= w
C.S.(70)= j
C.S.(71)= [
C.S.(72)= z
C.S.(73)= c
C.S.(74)= qta};o
C.S.(75)= ifhwj[zc
C.S.(76)= w
C.S.(77)= f
C.S.(78)= t
C.S.(79)= ioqz[
C.S.(80)= j
C.S.(81)= i
C.S.(82)= ;
C.S.(83)= inrs
C.S.(84)= ;
C.S.(86)=)
C.S.(87)= h
C.S.(88)= a
C.S.(89)= }o
C.S.(90)= o
C.S.(91)= }

C.S.(92)= q
C.S.(93)= }ao
C.S.(94)= z
C.S.(95)= inrs
C.S.(96)=)
C.S.(97)= [
C.S.(98)=)
C.S.(99)= inrs
C.S.(100)= ,
C.S.(101)=)

3.3. Diseño e implementación

En este caso, hice uso de la estructura definida como "Componente" en el diseño del analizador léxico:

- Clase: Entero que almacena, como tal, la clase a la que corresponde el componente.
- Valor: Esto puede ser la posición, para los Componentes de catálogos o tablas de símbolos o, tal cual, el valor numérico de la cadena (utilizado en constantes numéricas enteras, caracteres especiales y operadores aritméticos).
- Cadena: Almacena la cadena reconocida.
- Tipo: Para los identificadores, se especifica el tipo de dato con la posición de la palabra reservada (empleada en su declaración) en el catálogo de palabras reservadas. Esto sucede una vez que se han llenado los catálogos.

La lista de tokens usa esta estructura también, aunque no utilizaba el campo "tipo". En esta ocasión, agregamos un contador de líneas al análisis léxico y decidimos colocar el número de línea en este campo "tipo" en la estructura de los tokens, para poder encontrar las lineas donde se generan los errores sintácticos.

Posteriormente, definimos una cadena, inicalmente vacía, y colocamos las condiciones necesarias, en las acciones que se realizan cuando se reconoce una cadena, para generar el átomo correspondiente y concatenarlo a dicha cadena.

```
void concatena(char atomo, char* cadena){
    char nuevo[2];
    nuevo[0] = atomo;
    nuevo[1] = '\0';
    strcat(cadena,nuevo);
    printf("CADENA DE ATOMOS ACTUAL: %s \n",cadena);
}
```

Después de esto, me dispuse a implementar las reglas del análisis sintáctico descendente recursivo con cada una de las producciones, considerando los ya calculados conjuntos de selección.

3.3.1. Función getchar

Esta función fue renombrada como "sigAtomo()" para evitar conflictos con la sintáxis de C. En esta función, se avanza un elemento en la cadena con ayuda de un apuntador, el contenido se pasa a una variable **char** llamada *atomo* y se incrementa un contador de átomos, que servirá para la búsqueda de un token cuando se encuentre un error léxico. A continuación, se muestra dicha función.

```
void sigAtomo(){
    apStr++;
    atomo = *apStr;
    numAtomo++;
}
```

3.3.2. Posición del error sintáctico

Con ayuda de la siguiente función, se obtiene el token del átomo que se tiene, que incluye el número de línea donde se colocó y la cadena que se utilizó. Simplemente toma el token que está en la n posición donde nos quedamos en la lectura de la cadena y lo retorna.

```
Nodo* buscarToken(Tabla tabla, int numAtomo){
    Nodo *q = tabla.H;
    for(int i=0; i<numAtomo; i++){
        q = q->sig;
    }
    return q;
}
```

Cada que se detecta un error, se busca el token y se despliega un mensaje con la línea donde está el error, así como la cadena que se usó y la que se esperaba usar. Por poner un ejemplo:

```
    void arg(){
    if(atomo==')'){
        return;
    }
    else if(atomo=='i'||atomo=='n'||atomo=='r'||atomo=='s'){
        V();
        otroArg();
        return;
    }
    else{
        aux = buscarToken(*listAtom,numAtomo);
        strcpy(cadEsperada,aux->info->cadena);
        numLinea = aux->info->tipo;
        rechaza();
        printf("(Linea %d): Se esperaba ')', '$<nombre_identificador
           >' o una constante pero se obtuvo '%s'",numLinea,
           cadEsperada);
```

```
            return;
        }
    }
```

4. Resultados

A continuación, se muestran 3 ejemplos de ejecución:

```
Cadena de atomos generada: yi(){xi=r,i;yi,i=s;bi=n,i,i;f(iur)w(iln)m{i=i+n\n;i=(n+i)^n;f(i>n)c;:}i=s;:j
(i=n;iln;i=i+n){[i(i)]}f(iei)z(s);tz(s);:}#i(bi){yi;#i;h(i){an:an:an:i=s;qan:an:an:i=s;qan:an:an:i=s;qo
:i=s;}i=i*r%n-i;z(i);}

Tamanio de la cadena: 201

Cadena aceptada!
Intento terminado. Numero de errores: 0

Fin de la ejecucion.
Revise el archivo 'salida.txt'

┌──(zuriel㉿zuri)-[~/Escritorio/Compiladores/PROYECTO/AnalizadorSintactico]
└─$
```

Figura 1: Ejecución del programa original

```
1   symbol  $funUno( )
2   {
3       real  $NumReal=27.5, $OtroReal;
4       symbol $cadUno,$cadDos= "Cadena #2";
5       big $numUno=5, $numDos, $num;
6
7        evaluate($NumReal>=28.5)
8               loop( $numUno<=20) make
9               {
10                  $numUno=$numUno+3\4;
11                  $numDos= (3+$numUno)^2;
12                  evaluate ($numDos> 200)
13                      step;
14
15              }
```

Figura 2: Modificación 1 a la prueba

```
Cadena de atomos generada: yi(){xi=r,i;yi,i=s;bi=n,i,i;f(iur)w(iln)m{i=i+n\n;i=(n+i)^n;f(i>n)c;}i=s;:j(
i=n;iln;i=i+n){[i(i)]}f(iei)z(s);tz(s);:}#i(bi){yi;#i;h(i){an:an:an:i=s;qan:an:an:i=s;qan:an:an:i=s;qo:
i=s;}i=i*r%n-i;z(i);}

Tamanio de la cadena: 200

Syntax Error (Linea 15): Se esperaba 'instead' o ':' pero se obtuvo '}'
Syntax Error (Linea 15): Se esperaba ':' pero se obtuvo '}'

Intento terminado. Numero de errores: 2

Fin de la ejecucion.
Revise el archivo 'salida.txt'

┌──(zuriel㉿zuri)-[~/Escritorio/Compiladores/PROYECTO/AnalizadorSintactico]
└─$
```

Figura 3: Ejecución del programa con modificación 1

```
29  number  $funDos(big $Num)
30  {
31      symbol  $cadResult;
32      number $valor;
33
34      select(192)
35      {
36          alternative 2:
```

Figura 4: Modificación 2 a la prueba

```
Cadena de atomos generada: yi(){xi=r,i;yi,i=s;bi=n,i,i;f(iur)w(iln)m{i=i+n\n;i=(n+i)^n;f(i>n)c;}i=s;:j(
i=n;iln;i=i+n){[i(i)]}f(iei)z(s);tz(s);:}#i(bi){yi;#i;h(n){an:an:an:i=s;qan:an:an:i=s;qan:an:an:i=s;qo:
i=s;}i=i*r%n-i;z(i);}

Tamanio de la cadena: 200

Syntax Error (Linea 15): Se esperaba 'instead' o ':' pero se obtuvo '}'
Syntax Error (Linea 15): Se esperaba ':' pero se obtuvo '}'
Syntax Error (Linea 34): Se esperaba '$<nombre_identificador>' pero se obtuvo '192'
Syntax Error (Linea 34): Se esperaba '$<nombre_identificador>', '[', '}' o una estructura de control pe
ro se obtuvo '192'
Syntax Error (Linea 34): Se esperaba '}' pero se obtuvo '192'
Syntax Error (Linea 34): Se esperaba un tipo de dato o fin de cadena pero se obtuvo '192'

Intento terminado. Numero de errores: 6

Fin de la ejecucion.
Revise el archivo 'salida.txt'

┌──(zuriel㉿zuri)-[~/Escritorio/Compiladores/PROYECTO/AnalizadorSintactico]
└─$
```

Figura 5: Ejecución del programa con modificación 2

5. README.md

Las instrucciones de instalación y ejecución del programa se muestran en el archivo README.md. Para más información, favor de consultarlo. Puede descargar el proyecto completo aquí: https://github.com/delionsgate/ZZG_AXIN.

6. Conclusiones

A lo largo de este proyecto, me di cuenta de lo realmente complicado que puede ser el proceso de compilación. En el análisis sintáctico, es extremadamente importante el diseño correcto de la gramática y el cuidado que se debe tener a la hora de calcular los conjuntos de selección, ya que muchas veces me trabé y no sabía cómo continuar debido a la gran cantidad de producciones y llamadas recursivas que el sistema hacía.

A pesar de lo anterior, me parece que cumplí con los objetivos de esta implementación satisfactoriamente, no solamente logré lo que se pedía, sino que acabe aprendido a calcular rápidamente los conjuntos de selección de una gramática $LL(1)$.

Así como en el trabajo anterior, estoy muy orgulloso del trabajo que hice ahora, me siento profundamente satisfecho (ya que lo hice prácticamente sólo) y siento que mis aprendizajes han sido enriquecidos. Dado lo anterior, concluyo que los objetivos de este proyeto se cumplieron.

7. Referencias

Chi, R. I. G. (2019). Teoría de lenguajes y autómatas [Consultado el 31 de octubre de 2022. Recuperado de: https://issuu.com/ingrosychi/docs/analizador_lexico_con_afd]. *ISSUU*.

de la Cruz, M., & Ortega, A. (2007). Construcción de un analizador léxico para ALFA con LEX [Consultado el 31 de octubre de 2022. Recuperado de: http://arantxa.ii.uam.es/~mdlcruz/docencia/compiladores/2007_2008/lexico_07_08.pdf]. *Prácticas de Procesadores de Lenguaje*.

Pérez, E. S. (2022). Compiladores [Consultado el 1 de noviembre de 2022. Recuperado de: https://hopelchen.tecnm.mx/principal/sylabus/fpdb/recursos/r94256.PDF]. *UAA - Sistemas Electrónicos*.