Analizador Léxico Sintáctico

Desarrollador

Juan Cardona

Facultad de Ingeniería

Departamento de ingeniería de sistemas

Teoría de Lenguajes

2021-1

Introducción

Para el desarrollo de este proyecto, se tiene como propósito la construcción de 2 de los componentes más importantes de un compilador, un analizador léxico y un analizador sintáctico de un lenguaje, en este caso, de programación.

La primera fase para la construcción de un compilador es la creación de un Analizador léxico, su función es leer una cadena de caracteres de entrada e identificar cada componente léxico o lexema, que se lea correctamente, asignándole un token único.

La segunda fase es la creación de un Analizador sintáctico, su función es la de analizar cadenas de tokens, haciendo un proceso de reconocimiento a la gramática que genera el lenguaje de programación, y determinar si las cadenas pertenecen o no a dicho lenguaje.

Para la primera fale el método que se usará para construir el analizador léxico consiste en los siguientes pasos:

- Creación de la lista de componentes léxicos
- AFD: Diagramas de burbujas para cada clase de lexema
- AFD: Tablas de transiciones para cada clase de lexema
- Tabla de lexemas y tokens

Para la segunda fale el método que se usará para construir el analizador sintáctico consiste en los siguientes pasos:

- Identificar las secuencias de tokens que componen el lenguaje de programación
- G: Gramática LL(1) que genere el lenguaje
- Construcción del reconocedor descendente de pila para la gramática
- Inclusión de manejo de errores en el reconocedor

Por último, se hará el desarrollo de la aplicación que implementa los analizadores.

Primera Fase

Lista de componentes léxicos

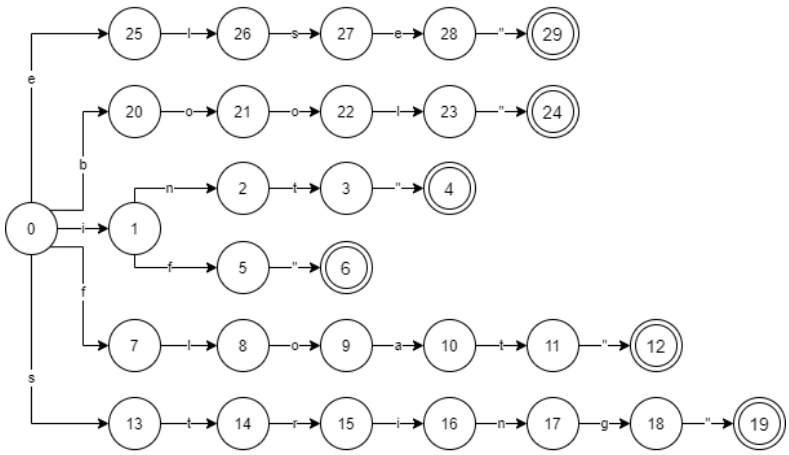
El analizador léxico acepta palabras reservadas, identificadores, constantes numéricas, constantes alfabéticas, operaciones aritméticas y separadores.

- Palabras reservadas (keyword):
  - int
  - float
  - string
  - bool
  - if
  - else
- Identificadores (identify): nombres de variables, funciones o clases creadas por el usuario. Constantes alfanuméricas, no inician con un número.
- Constantes numéricas (numConst): números enteros positivos y números flotantes.
  - D (cualquier caracter numérico o dígito)
  - ,
- Constantes alfabéticas (charConst): cadenas de caracteres
  - L (cualquier carácter alfabético, en minúscula o mayúscula, o letra)
- Operaciones aritméticas (operator):
  - +
  - -
  - *
  - /
  - =
  - <
  - <=
  - <>
  - >
  - >=
  - ==
- Separadores (separator):
  - (
  -)
  - {
  - }
  - ;

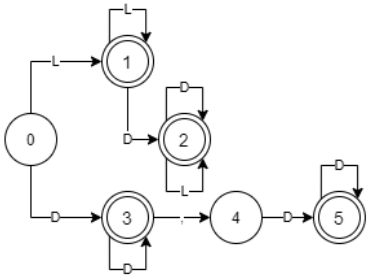
AFD: Diagramas de burbujas para cada clase de lexema

Para los AFD se consideraron una lista de caracteres de ingreso partiendo de la lista de componentes léxicos y se seleccionaron, mediante los caracteres necesarios para asignarle a cada tipo de lexema, un token especifico correspondiente en los AFD a los estados de aceptación.

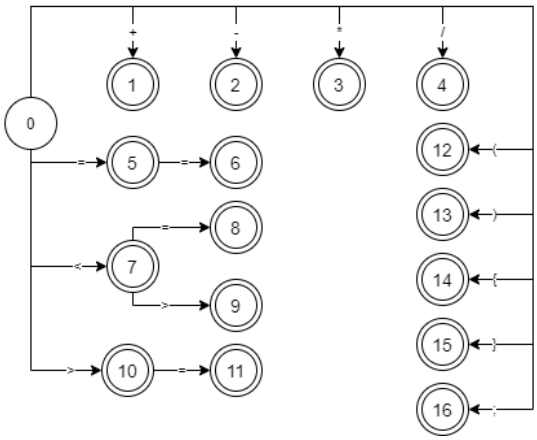
AFD1: keyword



AFD2: identify, numConst, charConst



AFD3: operator, separator



AFD: Tablas de transiciones para cada clase de lexema

Partiendo de cada diagrama de burbujas se determinaron las siguientes tablas de transiciones y se le asignaron a cada estado de aceptación un token correspondiente al lexema que se identifica en las secuencias de símbolos de entrada.

ADF1: keyword

	e	s	l	b	o	g	r	n	t	f	a	d	u	i		
0	20	10		16						5				1	0	
1								2		4					0	
2									3						0	
3															**1**	INT
4															**1**	IF
5			6												0	
6					7										0	
7											8				0	
8									9						0	
9															**1**	FLOAT
10									11						0	
11							12								0	
12														13	0	
13								14							0	
14						15									0	
15															**1**	STRING
16					17										0	
17					18										0	
18			19												0	
19															**1**	BOOL
20			21												0	
21		22													0	
22	23														0	
23															**1**	ELSE

AFD2: identify, numConst, charConst

- D (cualquier caracter numérico o dígito)
- L (cualquier carácter alfabético, en minúscula o mayúscula, o letra)

	L	D	,		
0	1	3		0	
1	1	2		**1**	CHARCONST
2	2	2		**1**	IDENTIFY
3		3	4	**1**	INTCONST
4		5		0	
5		5		**1**	FLOATCONST

ADF3: operator, separator

	+	-	*	/	=	<	>	(	)	{	}	;		
0	1	2	3	4	5	7	10	12	13	14	15	16	0	
1													**1**	PLUS
2													**1**	MINUS
3													**1**	MULTIPLICATION
4													**1**	DIVISION
5					6								**1**	ASIGNATION
6													**1**	BOOLCOPARISON
7					8		9						**1**	MINOR
8													**1**	MINOREQUAL
9													**1**	DIFFERENT
10					11								**1**	HIGHER
11													**1**	HIGHEREQUAL
12													**1**	OPENPARENTHESIS
13													**1**	CLOSEPARENTHESIS
14													**1**	OPENBRACKETS
15													**1**	CLOSEBRACKETS
16													**1**	SEMICOLON

Tabla de lexemas y tokens

De cada estado de aceptación obtenemos la lista de tokens para cada lexema que reconoce el analizador léxico.

Lexema	Token	
esle	ELSE	
bool	BOOL	
int	INT	
if	IF	
float	FLOAT	
string	STRING	
L^+	CHARCONST	Estos lexemas se representan por una expresión regular
$L^+(L+D)^*$	IDENTIFY	
D^+	INTCONST	
D^+,D^+	FLOATCONST	
+	PLUS	
-	MINUS	
*	MULTIPLICATION	
/	DIVISION	
=	ASIGNATION	
==	BOOLCOPARISON	
<	MINOR	
<=	MINOREQUAL	
<>	DIFFERENT	
>	HIGHER	
>=	HIGHEREQUAL	
(	OPENPARENTHESIS	
)	CLOSEPARENTHESIS	
{	OPENBRACKETS	
}	CLOSEBRACKETS	
;	SEMICOLON	

Segunda Fase

Secuencias de tokens

Para definir las secuencias de tokens validas para nuestro lenguaje, primero se agruparán tokens por similitud lógica en conjuntos, estos conjuntos representarán los terminales de la gramática.

Lexema	Token	Terminales
esle	ELSE	ELSE
if	IF	IF
int	INT	T
float	FLOAT	
bool	BOOL	
string	STRING	
L+	CHARCONST	V
L+(L+D)*	IDENTIFY	
D+	INTCONST	D
D+,D+	FLOATCONST	
+	PLUS	O
-	MINUS	
*	MULTIPLICATION	
/	DIVISION	
=	ASIGNATION	=
==	BOOLCOPARISON	B
<	MINOR	
<=	MINOREQUAL	
<>	DIFFERENT	
>	HIGHER	
>=	HIGHEREQUAL	
(	OPENPARENTHESIS	(
)	CLOSEPARENTHESIS	)
{	OPENBRACKETS	{
}	CLOSEBRACKETS	}
;	SEMICOLON	;

Para el lenguaje de programación se necesita que la gramática identifique las siguientes secuencias de tokens:

- **Definición de variables:**
  - T V;
  - T V=V;
  - T V=D;
- **Asignación de valores:**

  Todas las 'V' a la derecha de un '=' son recursivas con cualquier secuencia que tenga la 'V' al lado izquierdo sin incluir el ';'.

  Los '[ ]' no están incluidos en los lexemas, pero son usados para denotar multiplicad en la aparición de múltiples ocurrencias de una secuencia de tokens.

  - V=V;
  - V=D;
  - V=(VOV);
  - V=(VBV);
  - **Estructuración condicional:**
    - IF(VBV){[V;]$^{+}$}ELSE{[V;]$^{+}$};

G: Gramática LL(1) que genere el lenguaje

Partiendo de las secuencias de tokens anteriores, del lenguaje de programación, se escribe las siguientes gramáticas con el fin de comenzar a construir el reconocedor para dichas gramáticas y por consiguiente del lenguaje de programación.

- **Definición de variables:**

1. <S> → TV<A>;<S>
2. <S> → λ
3. <A> → =<B>
4. <A> → λ
5. <B> → V
6. <B> → D

- **Asignación de valores + estructuración condicional:**

1. <R> → V=<X>;<R>
2. <R> → λ
3. <X> → V
4. <X> → D
5. <X> →(<X><Z><X>)
6. <Z> → O
7. <Z> → B
8. <R> → IF(<X>B<X>){<R>}ELSE{<R>};<R>

Ahora con estas dos gramáticas que generan por separado la definición y asignación de variables, podemos unirlas para crear la gramática que genera el lenguaje de programación.

- **Gramática que genera el lenguaje:**

1. <S> → TV<A>;<S>
2. <S> → V=<X>;<S>
3. <S> → IF(<X>B<X>){<S>}ELSE{<S>};<S>
4. <S> → λ
5. <A> → =<B>
6. <A> → λ
7. <B> → V
8. <B> → D
9. <X> → V
10. <X> → D
11. <X> →(<X><Z><X>)
12. <Z> → O
13. <Z> → B

Analizaremos si la gramática está en la forma LL(1), determinando sus conjuntos de selección y viendo si estos son disjuntos para cada No Terminal.

$N_{anulables}$: {<S>, <A>}

$P/nes_{anulables}$: {4, 6}

Primeros(<S>): {T, V, IF}

Primeros(<A>): {=}

Primeros(<B>): {V, D}

Primeros(<X>): {V, D, (}

Primeros(<Z>): {O, B}

Siguientes(<S>): {┤ , }}

Siguientes(<A>): {;}

Siguientes(<B>): {;}

Siguientes(<X>): {B,), O}

Siguientes(<Z>): {V, D, (}

Primeros(1): {T}

Primeros(2): {V}

Primeros(3): {IF}

Primeros(4): { }

Primeros(5): {=}

Primeros(6): { }

Primeros(7): {V}

Primeros(8): {D}

Primeros(9): {V}

Primeros(10): {D}

Primeros(11): {(}

Primeros(12): {O}

Primeros(13): {B}

Selección(1): {T}

Selección(2): {V}

Selección(3): {IF}

Selección(4): {┤ , }}

Selección(5): {=}

Selección(6): {B,), O}

Selección(7): {V}

Selección(8): {D}

Selección(9): {V}

Selección(10): {D}

Selección(11): {(}

Selección(12): {O}

Selección(13): {B}

Conjuntos de selección disjuntos, por las producciones del tipo:

1. <N> → Tα
2. <N> → λ
3. <N> → β

La gramática es LL(1)

Reconocedor descendente de pila para la gramática

Ahora que tenemos nuestra gramática LL(1) y sus conjuntos de selección, construiremos el reconocedor descendente de pila y se determinarán las transiciones correspondientes al token de entrada respecto al token en el tope de la pila.

Símbolos de entrada = {T, V, ;, =, IF, (, B,), {, }, ELSE, D, O, ┤}

Símbolos en la pila = {<S>, <A>, <B>, <X>, <Z>, V, ;, =, (, B,), {, }, ELSE, ▼}

Configuración inicial = ▼<S>

	T	V	;	=	IF	(	B	)	{	}	ELSE	D	O	┤
<S>	1	2			3					4				4
<A>				5			4	4					4	
<B>		6										6		
<X>		6				7						6		
<Z>							6						6	
V		6												
;			6											
=				6										
(						6								
B							6							
)								6						
{									6					
}										6				
ELSE											6			
▼														A

Transiciones:

1: replace (<S>;<A>V), avance
2: replace (<S>;<X>=), avance
3: replace (<S>;}>S>{ELSE}<S>{)<X>B<X>(), avance
4: desapile, retenga
5: replace (<B>), avance
6: desapile, avance
7: replace ()<X><Z><X>), avance

Inclusión de manejo de errores en el reconocedor

Por último, realizaremos el manejo de errores para el reconocedor descendente de pila, analizando los posibles fallos al recibir un token de entrada no especificado para el token en el tope de la pila y escribir su mensaje pertinente. Para el manejo de errores no se considerará tratamiento en la pila, el usuario deberá, una ves reciba un resultado de error, corregir el código para volver a analizarlo.

	T	V	;	=	IF	(	B	)	{	}	ELSE	D	O	┤
<S>	1	2	E5	E6	3	E8	E9	E10	E11	4	E13	E14	E15	4
<A>	E3	E4	E5	5	E7	E8	4	4	E11	E12	E13	E14	4	E2
<B>	E3	6	E5	E6	E7	E8	E9	E10	E11	E12	E13	6	E15	E2
<X>	E3	6	E5	E6	E7	7	E9	E10	E11	E12	E13	6	E15	E2
<Z>	E3	E4	E5	E6	E7	E8	6	E10	E11	E12	E13	E14	6	E2
V	E3	6	E5	E6	E7	E8	E9	E10	E11	E12	E13	E14	E15	E2
;	E3	E4	6	E6	E7	E8	E9	E10	E11	E12	E13	E14	E15	E2
=	E3	E4	E5	6	E7	E8	E9	E10	E11	E12	E13	E14	E15	E2
(	E3	E4	E5	E6	E7	6	E9	E10	E11	E12	E13	E14	E15	E2
B	E3	E4	E5	E6	E7	E8	6	E10	E11	E12	E13	E14	E15	E2
)	E3	E4	E5	E6	E7	E8	E9	6	E11	E12	E13	E14	E15	E2
{	E3	E4	E5	E6	E7	E8	E9	E10	6	E12	E13	E14	E15	E2
}	E3	E4	E5	E6	E7	E8	E9	E10	E11	6	E13	E14	E15	E2
ELSE	E3	E4	E5	E6	E7	E8	E9	E10	E11	E12	6	E14	E15	E2
▼	E1	E1	E1	E1	E1	E1	E1	E1	E1	E1	E1	E1	E1	A

Errores:
E1: write(expresión ya es correcta), avance
E2: write(expresión incompleta), termine
E3: write(token T no esperado), termine
E4: write(token V no esperado), termine
E5: write(token ; no esperado), termine
E6: write(token = no esperado), termine
E7: write(token IF no esperado), termine
E8: write(token (no esperado), termine
E9: write(token B no esperado), termine
E10: write(token) no esperado), termine
E11: write(token { no esperado), termine

E12: write(token } no esperado), termine
E13: write(token ELSE no esperado), termine
E14: write(token D no esperado), termine
E15: write(token O no esperado), termine

De esta manera se puede continuar con el desarrollo de el compilador con los analizadores léxico y sintáctico.

Desarrollo de la aplicación

Para ver el código, la documentación y el archivo ejecutable con el analizador léxico visitar el siguiente repositorio: https://github.com/ohmono/analizador-lexico

Bibliografía

- Flórez Rueda, R. (2010). Introducción a los compiladores. Medellín, Colombia: Universidad de Antioquia