Construcción de un analizador léxico y sintáctico para un sublenguaje de Java

Iván Illán Barraya Javier Monescillo Buitrón

12 de marzo de 2019

Índice

1. Introducción **3**

2. Descripción del problema **3**

3. Solución propuesta **3**

4. Diseño del sistema **3**
4.1. Lenguaje fuente 4
4.2. Tabla de tokens 4
4.3. EBNF 5

5. Análisis léxico **6**
5.1. JFlex 6

6. Análisis sintáctico **6**
6.1. Análisis sintáctico ascendente 7
6.1.1. Control de errores sintácticos en Cup 7
6.2. Cup 8

7. Analizador semántico **8**
7.1. Generación de código en análisis ascendente 8

1. Introducción

Hablar un poco de Java

2. Descripción del problema

El problema que se presenta es la construcción de un analizador léxico y sintáctico de un lenguaje especificado previamente, usando las herramientas Jflex y Cup.

3. Solución propuesta

Para diseñar dichos analizadores, utilizaremos los conocimientos de la materia durante las distintas etapas del proceso.

- Diseño del sistema
- Diseño del Analizador Léxico
  - Identificar Tokens
  - Construcción del Analizador Léxico
- Diseño del Analizador Sintáctico
  - Identificar producciones
- Diseño del Analizador Semántico

La primera etapa consta del diseño referente al lenguaje fuente, procesador de lenguajes, o arquitectura general del sistema. Se proporcionará un diagrama total del problema.

4. Diseño del sistema

BLA BLA BLA + Diagramilla

4.1. Lenguaje fuente

HABLAR LENGUAJE FUENTE

Elementos del lenguaje	Ejemplo
Tipos de datos	int o boolean
Instrucción de asignación	type_var a = b; | type_var a = 0; | c = b;
Decremento	var–;
Incremento	var++;
Bucles	for(int i = 0; i<hola.length; i++) /* Ops*/
Llamadas a métodos	hola = calcularCosas(); | metodoVoid();
Retorno de valores	return x;
Secuencia de instrucciones	a = b; b * 2; //etc
Operaciones aritméticas	a + b; a - b; a * b; a / b;
Operaciones relacionales	a <b; a <= b; a >= b; a >b; a == b; a!=b;
Operaciones lógicas	a && b a || b y !a
Cabecera de los métodos	public static
Tipo devuelto de un método	int, boolean, void

Notese, que para cerrar una sentencia es necesario de indicar al final de dicha sentencia el carácter ';' como se hace típicamente en Java.

4.2. Tabla de tokens

En la siguiente tabla de tokens, se muestran los lexemas de ejemplo, los tokens y las expresiones regulares asociadas a cada token.

Token	Lexema	Patrón
return	return	r·e·t·u·r· n
for	for	f·o·r
int	int	i·n·t
boolean	boolean	b·o·o·l·e·a·n
void	void	v·o·i·d
public static	public static	p·u·b·l·i·c· ·s·t·a·t·i·c
true	true	t·r·u·e
false	false	f·a·l·s·e
Asignación	=	=
Negación	!	!
Lógicos binarios	&&	&·& | |·|
Incrementos	++	+·+ | -·-
Relacionales	<	<| <·= | >| >·= | =·= | !·=
Paréntesis abierto	(	(
Paréntesis cerrado	)	)
Llave abierta	{	{
Llave cerrada	}	}
Punto y coma	;	;
Coma	,	,
Punto	.	.
Asterisco barra	*/	*·/
Barra asterisco	/*	/·*
ID	hola	[A-Za-z][A-Zaz0-9_]*
NUM	4	0 | [1-9][0-9]*
Valor nulo	Null	N·u·l·l

4.3. EBNF

HABLAR DE LA NOTACION EXTENDED BACKUS NAUR FORM poner la gramática nueva Por último, en la siguiente figura puede ver una representación de la gramática del lenguaje en EBNF.

PROGRAMA	::=	DEC_COMP AUTOMATA {AUTOMATA}
DEC_COMP	::=	CMP CODIGO { CMP CODIGO }
CODIGO	::=	'#' ASCII '#'
AUTOMATA	::=	**moore** ID CUERPO_AUTOMATA
CUERPO_AUTOMATA	::=	'{' ESTADOS ESTADO_INI ALF_IN ALF_OUT TRANSICION COMPORTAMIENTOS '}'
ESTADOS	::=	**estados** { ID ',' } ID ';'
ESTADO_INI	::=	**estado_in** ID ';'
ALF_IN	::=	**alf_in** { EVENTOS ',' } EVENTOS ';'
ALF_OUT	::=	**alf_out** { CMP ',' } CMP ';'
TRANSICION	::=	**transicion** '{' TRANSICION_DEF { ',' TRANSICION_DEF } ';' '}'
TRANSICION_DEF	::=	'(' ID ',' ID ',' ID ')'
COMPORTAMIENTOS	::=	**comportamientos** '{' COMP_DEF { ',' COMP_DEF } ';' '}'
COMP_DEF	::=	'(' ID ',' ID ',' ID ')'
CMP	::=	'c'NUMEROS
NUMEROS	::=	0 | 1 | .. | 9
COMENTARIOS	::=	'/*' ASCII '*/'

5. Análisis léxico

El analizador léxico tiene varias funciones:

- Reconocer los símbolos / tokens que componen el texto fuente.
- Eliminar comentarios del texto fuente.
- Eliminar espacios en blanco, saltos de línea, tabulaciones, etc.
- Informar de los errores léxicos detectados

Como salida del analizador léxico, se obtiene una representación de la cadena de entrada en forma de cadena de **tokens**, que será posteriormente utilizada en la fase de análisis sintáctico. Para construir el analizador léxico se ha utilizado JFlex.

Por un lado *JFlex* es una herramienta software en la que se declaran los tokens que componen nuestro lenguaje fuente, así como las expresiones regulares asociadas a los mismos para poder generar el analizador léxico.

5.1. JFlex

En esta sección se muestra el código en JFlex donde se construye el analizador léxico del lenguaje Jjava.

Listing 1: Analizador Léxico en JFlex

6. Análisis sintáctico

Las principales funciones del analizador sintáctico son las siguientes:

- Analizar la secuencia de **tokens** y verificar si son correctos sintácticamente.
- Obtener una representación interna del texto.
- Informar de los errores sintácticos detectados.

En resumen, dada una secuencia de **tokens** obtenida como resultado de la fase de análisis léxico, se comprueba que dicha secuencia está escrita correctamente y se obtiene una representación interna de la misma, que servirá como entrada para el Análisis semántico.

Existen dos estrategias en el *Análisis sintáctico*

- Análisis sintáctico ascendente
- Análisis sintáctico descendente

6.1. Análisis sintáctico ascendente

CUP significa Construcción de analizadores útiles y es un generador de analizadores LALR para Java. Fue desarrollado por C. Scott Ananian, Frank Flannery, Dan Wang, Andrew W. Appel y Michael Petter. Implementa la generación de analizadores LALR(1) estándar.

La estrategia del análisis sintáctico ascendente funciona construyendo el árbol sintáctico desde las hojas hasta la raíz. Se busca en la cadena de tokens una subcadena que pueda ser reducida a uno de los símbolos no terminales que forman la gramática.

El analizador LALR(1) nace de la simplificación de estados del analizador LR(1). No se entra en detalle de la construcción del AFD reconocedor de prefijos viables ni del analizador sintáctico LR(1) ya que la herramienta Cup realiza el proceso de simplificación de forma autónoma, así como la inclusión de marcadores.

6.2. Control de errores sintácticos en Cup

Para controlar los errores sintácticos en Cup se han utilizado producciones de error. Las producciones de error utilizan un símbolo terminal error que pertenece a la clase *Symbol* propia de CUP.
De tal manera que cuando se produce una reducción al mismo, se invoca a una rutina de error asociada a este símbolo. En esta rutina de error, se invoca al método report_error de la clase *Parser*.

En la sección *Cup* puede ver el código asociado a esta fase de la construcción del procesador de lenguajes.

6.3. Cup

En esta sección se muestran tanto las producciones asociadas a la gramática y utilizadas para realizar el análisis sintáctico, como las reglas semánticas asociadas a cada producción; que permiten el análisis semántico.

Listing 2: Analizador Sintáctico y Semántico en CUP

```
Meter el codigo gramatica
```

7. Analizador semántico

El analizador semántico tiene varias funciones:

- Dar significado a las construcciones del lenguaje fuente.
- Generación de código.
- Acabar de completar el lenguaje fuente.

7.1. Generación de código en análisis ascendente

SI LO HUBIERA

Referencias

[1] Autómatas de Moore https://es.wikipedia.org/wiki/M%C3%A1quina_de_Moore

[2] METER MAS COSAS O QUITAR