TRABAJO FINAL - FUNDAMENTOS TEÓRICOS DE INFORMÁTICA

Analizador Léxico-Sintáctico LR

AUTOR: Nuñez, Gustavo Marcelo.

DOCENTES: Moreno, Leonardo. Tidona, Fernando.

Diciembre 2022

Introducción

El presente documento pertenece al trabajo final correspondiente a la materia *Fundamentos Teóricos de Informática*.

En el mismo se describe el trabajo realizado: un analizador léxico-sintáctico validador de un conjunto de sentencias típicas de un lenguaje, como por ejemplo:

- sentencias de inicio/fin programa
- while
- if
- sentencias de asignación
- constantes
- caracteres literales
- operadores matemáticos (+, -, /, *)
- operadores comparativos (==, <, >, >=, <=, !=)

Análisis léxico-sintáctico

Un **analizador léxico** o *lexer* es un módulo destinado a leer caracteres de entrada (ya sea entrada estándar o de un archivo donde se encuentra la cadena a analizar), reconocer subcadenas que correspondan a símbolos del lenguaje y retornar los tokens correspondientes y sus atributos.

Un **analizador sintáctico** o *parser* es un programa que normalmente es parte de un compilador. El compilador se asegura de que el código se traduce correctamente a un lenguaje ejecutable. La tarea del analizador es, en este caso, la descomposición y transformación de las entradas en un formato utilizable para su posterior procesamiento. Se analiza una cadena de instrucciones en un lenguaje de programación y luego se descompone en sus componentes individuales.

La combinación de estas herramientas son el comienzo del desarrollo de un compilador.

A continuación, se detallan las características del lenguaje desarrollado para este proyecto.

Gramática

program	→	BEGIN statement_list END
statement_list	→	statement
	→	statement_list statement
statement	→	assignment
	→	expression
	→	while
	→	if
	→	switch
	→	for
assignment	→	ID EQUAL expression
while	→	WHILE '(' condition ')' '{' statement_list '}'
if	→	IF '(' condition ')' '{' statement_list '}'
switch	→	SWITCH '(' operator ')' '{' case_list '}'
case_list	→	case
	→	case_list case
case	→	CASE CONSTANT ':' '{' statement_list BREAK '}'
for	→	FOR '(' operator TO operator ')' '{' statement_list '}'
condition	→	expression EQUAL EQUAL expression
	→	expression '>' EQUAL expression
	→	expression '<' EQUAL expression
	→	expression '>' expression
	→	expression '<' expression
	→	expression DISTINCT expression
expression	→	expression OP_ADD_SUB term
	→	term
term	→	term OP_MUL_DIV operator
	→	operator
operator	→	CONSTANT
	→	ID

Tokens

- **BEGIN**: Cadena literal '*begin*'.
- **END**: Cadena literal '*end*'.
- **CONSTANT**: Caracteres numéricos.
- **ID**: Cadenas literales '*x*', '*y*' ó '*z*'.
- **WHILE**: Cadena literal '*while*'.
- **IF**: Cadena literal '*if*'.
- **SWITCH**: Cadena literal '*switch*'.
- **CASE**: Cadena literal '*case*'.
- **FOR**: Cadena literal '*for*'.
- **TO**: Cadena literal '*to*'.
- **BREAK**: Cadena literal '*break*'.
- **EQUAL**: Cadena literal '*=*'.
- **DISTINCT**: Cadena literal '*!=*'.
- **OP_ADD_SUB**: Cadenas literales '*+*' ó '*-*'.
- **OP_MUL_DIV**: Cadenas literales '***' ó '*/*'.
- Símbolos literales: <>(){}:

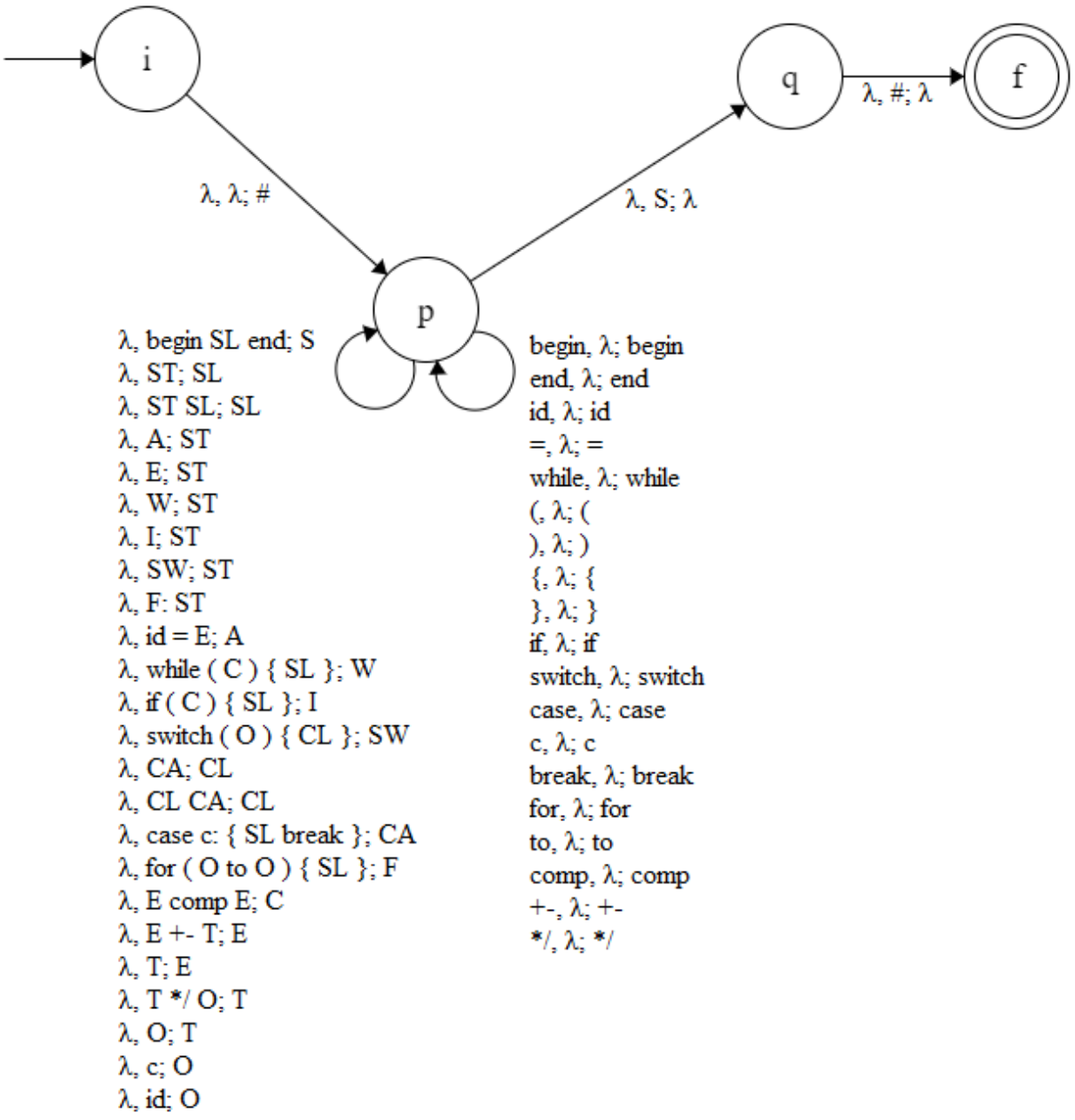
Gramática simplificada y aumentada

			S'	-->	S
program	-->	begin statement_list end	S	-->	begin SL end
statement_list	-->	statement | statement_list statement	SL	-->	ST | SL ST
statement	-->	assignment | expression | while | if | switch | for	ST	-->	A | E | W | I | SW | F
assignment	-->	id = expression	A	-->	id = E
while	-->	while (condition) {statement_list}	W	-->	while (C) { SL }
if	-->	if (condition) {statement_list}	I	-->	if (C) { SL }
switch	-->	switch (operator) {case_list}	SW	-->	switch (O) { CL }
case_list	-->	case | case_list case	CL	-->	CA | CL CA
case	-->	case constant : {statement_list break}	CA	-->	case c : { SL break }
for	-->	for (operator to operator) {statement_list}	F	-->	for (O to O) { SL }
condition	-->	expression comparator(*) expression	C	-->	E comp E
expression	-->	expression +- term | term	E	-->	E +- T | T
term	-->	term */ operator | operator	T	-->	T */ O | O
operator	-->	constant | id	O	-->	c | id(**)

(*) comparator: {'==', '>=', '<=', '>', '<', '!=' }

(**) id: {'x', 'y', 'z'}

Diagrama de transiciones



Análisis de ejemplo

Cadena de entrada: x=2+2.

Estado actual	Contenido de la pila	Resto de la entrada
p	#	x=2+2
p	#x	=2+2
p	#id	=2+2
p	#id=	2+2
p	#id=2	+2
p	#id=c	+2
p	#id=O	+2
p	#id=T	+2
p	#id=E	+2
p	#id=E+	2
p	#id=E+-	2
p	#id=E+-2	
p	#id=E+-c	
p	#id=E+-O	
p	#id=E+-T	
p	#id=E	
p	#A	
p	#ST	
p	#SL	
f	vacío	

Derivación: SL --> ST --> A --> id=E --> id=E+-T --> id=E+-O --> id=E+-c --> id=E+-2 --> id=E+2 --> id=T+2 --> id=O+2 --> id=c+2 --> id=2+2 --> x=2+2

Stack tecnológico

El proyecto fue desarrollado en lenguaje **Java**, utilizando las librerías **JFlex** y **byaccj**.

JFlex es una herramienta que se basa en la definición de una colección de reglas patrón - acción. Para determinar los distintos patrones (en este caso, la definición y coincidencia de los tokens) se utilizó la herramienta **byaccj**.

Con la combinación de estas herramientas se genera un programa que analiza la entrada utilizando los tokens identificados por el analizador y realiza las acciones especificadas, como por ejemplo marcar sintaxis incorrecta.

Posibles mejoras

Como posibles mejoras se puede definir operadores ternarios, sintaxis para la declaración de funciones, constructores, definir tipos de datos, manejo de excepciones, etc. con el propósito de concluir con el desarrollo de un compilador.