Diseño de compliadores

Analizador Léxico

Daniel Charua A01017419

23/02/19

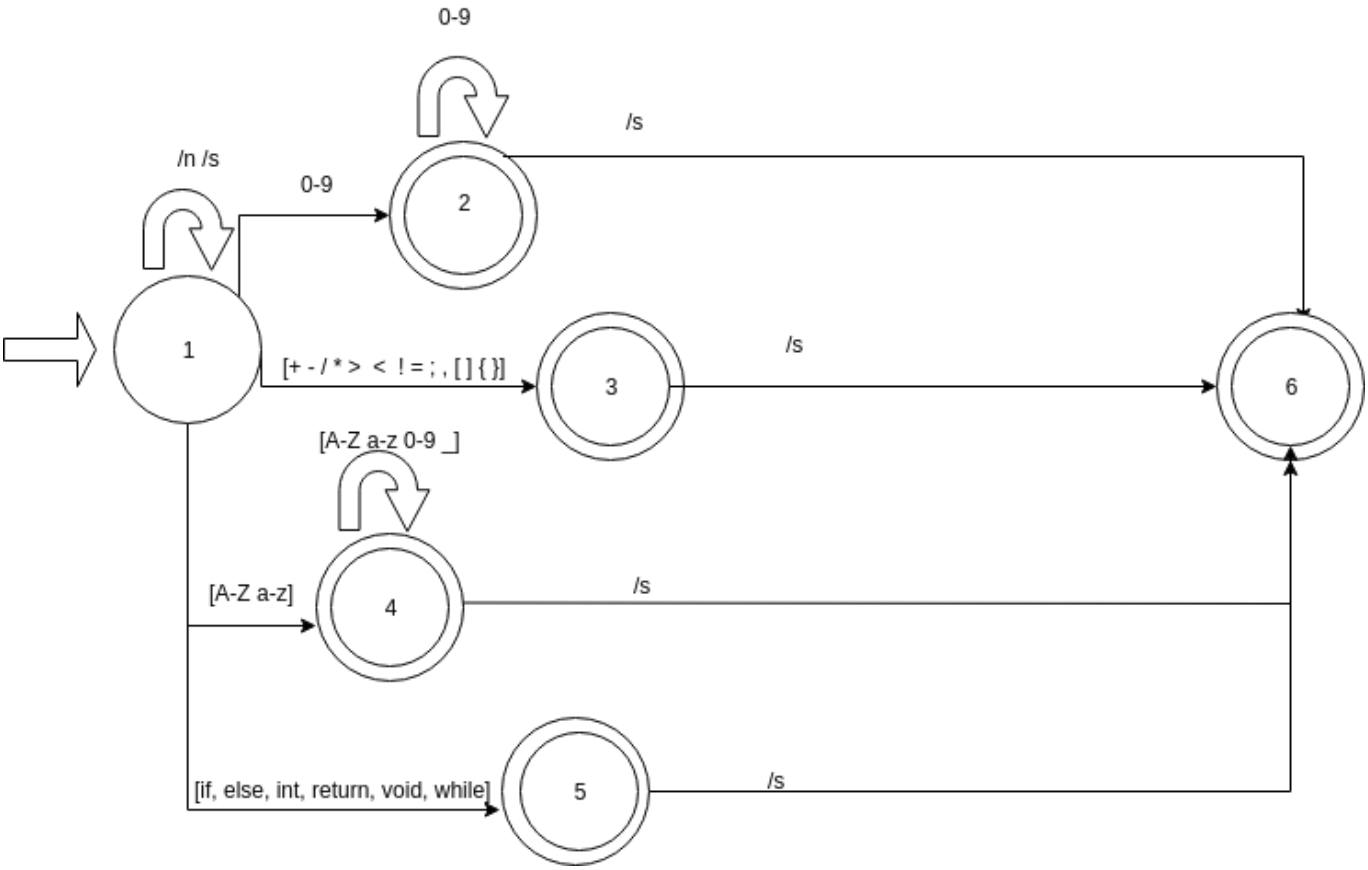
Expresiones regulares

Se implementó del analizador léxico con expresiones regulares, los siguientes expresiones fueron usadas:

```
(r'[ \s\n\t]+',   TokenType.SPACE),
(r'',             TokenType.SPACE),
(r'#[^\n]*',      TokenType.SPACE),
(r'\$',           TokenType.ENDFILE),
(r'\{',           TokenType.SPECIAL),
(r'\}',           TokenType.SPECIAL),
(r'\[',           TokenType.SPECIAL),
(r'\]',           TokenType.SPECIAL),
(r'\(',           TokenType.SPECIAL),
(r'\)',           TokenType.SPECIAL),
(r'\/\*(\*(?!\/)|[^*])*\*\/', TokenType.COMMENT),
(r';',            TokenType.SPECIAL),
(r',',            TokenType.SPECIAL),
(r'\+',           TokenType.SPECIAL),
(r'-',            TokenType.SPECIAL),
(r'\*',           TokenType.SPECIAL),
(r'/',            TokenType.SPECIAL),
(r'<=',           TokenType.SPECIAL),
(r'<',            TokenType.SPECIAL),
(r'>=',           TokenType.SPECIAL),
(r'>',            TokenType.SPECIAL),
```

```
(r'==',            TokenType.SPECIAL),
(r'!=',            TokenType.SPECIAL),
(r'=',             TokenType.SPECIAL),
(r'else',          TokenType.RESERVED),
(r'if',            TokenType.RESERVED),
(r'int',           TokenType.RESERVED),
(r'return',        TokenType.RESERVED),
(r'void',          TokenType.RESERVED),
(r'while',         TokenType.RESERVED),
(r'[0-9]+[a-zA-Z]+', TokenType.ERROR),
(r'[0-9]+',        TokenType.NUM),
(r'[a-zA-Z_][0-9a-zA-Z_]*', TokenType.ID)]
```

Automata



Gramática:

program -> declaration-list

declaration-list -> declaration {declaration}

declaration -> var-declaration | fun-declaration

var-declaration -> type-specifier [ID ; | ID [NUM] ;]

type-specifier -> int | void

fun-declaration-> type-specifier ID (params) compound-stmt

params-> param-list | void

param-list -> param {, param}

param -> type-specifier [ID | ID [ ]]

compount-stmt -> { local-declarations statement-list }

local-declarations -> empty {var-declaration}

statement-list -> empty {statement}

statement -> expression-stmt | compound-stmt | selection-stmt | iteration-stmt | return-stmt

expression-stmt -> expression ; | ;

selection-stmt -> if (expression) statement | if (expression) statement else statement

iteration-stmt -> while (expression) statement

return-stmt -> return ; | return expression ;

expression -> var = expression | simple-expression

var -> ID | ID [expression]

simple-expression -> additive-expression [relop additive-expression]

relop -> <= | < | > | >= | = | !=

additive-expression -> term {addop term}

addop -> + | -

term -> factor {mulop factor}

mulop -> * | /

factor -> (expression) | var | call | NUM

call -> ID (args)

args -> arg-list | empty

arg-list -> expression {, expression}