Universidad de San Carlos de Guatemala

Facultad de Ingenieria

Lenguajes Y compiladores 1

Primer Semestre 2020

Manual Tecnico Proyecto 2

Nombre: Juan José Ramos

Carnet: 201801262

Fecha: 22/5/2020

CONTENIDO

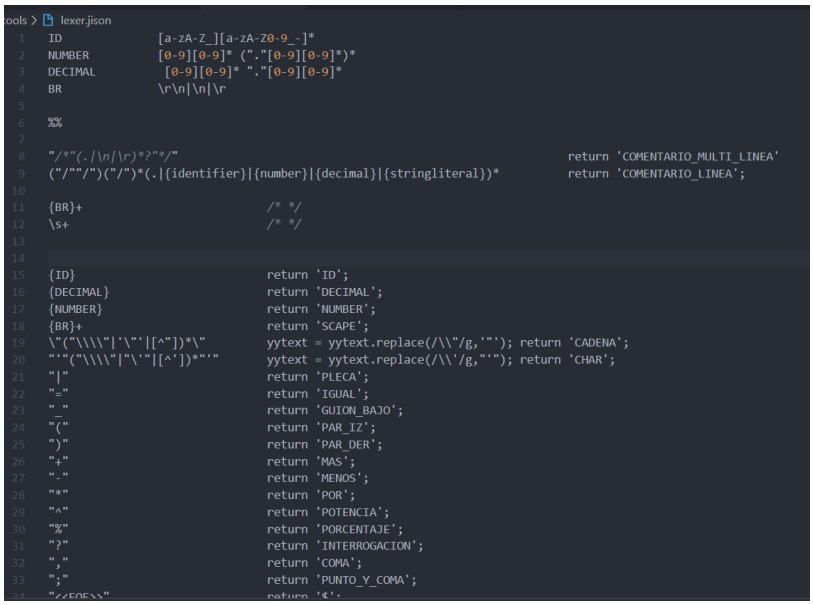
Servidor

Para el desarrollo del servidor, se utilizo Express, el servidor se genero mediante la herramienta, y esta creo las carpetas necesarias para la funcionalidad de dicho servidor.

bin	5/12/2020 9:05 PM	File folder	
controller	5/12/2020 9:05 PM	File folder	
lib	5/12/2020 9:05 PM	File folder	
model	5/12/2020 9:05 PM	File folder	
node_modules	5/22/2020 9:43 AM	File folder	
public	5/12/2020 9:05 PM	File folder	
routes	5/12/2020 9:05 PM	File folder	
tools	5/16/2020 9:06 PM	File folder	
views	5/12/2020 9:05 PM	File folder	
app	5/22/2020 9:59 AM	JavaScript File	2 KB
package	5/20/2020 9:26 AM	JSON File	1 KB
package-lock	5/9/2020 9:45 PM	JSON File	80 KB

Análisis Léxico

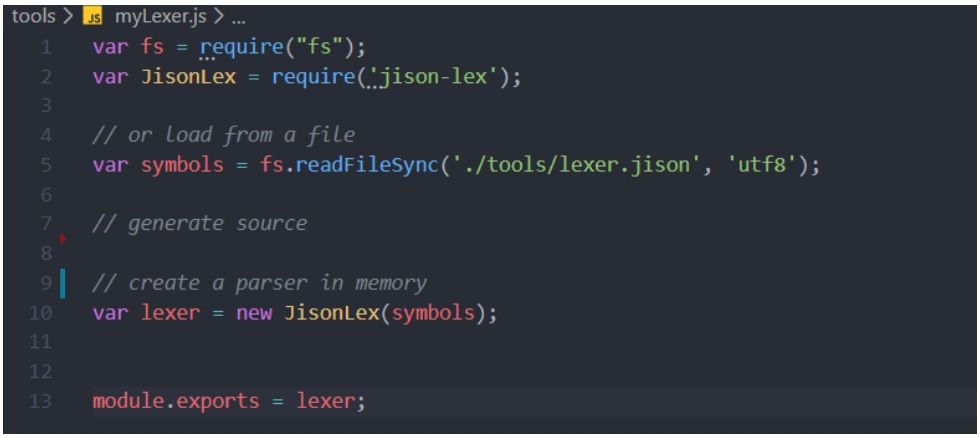
Tanto para el análisis léxico como el sintactico, se utilizó la herramienta Jison, para el análisis léxico, se genero un archivo llamado *lexer.jison* dicho archivo, contenia los simbolo comunes del lenguaje



```
tools > lexer.jison
ID              [a-zA-Z_][a-zA-Z0-9_-]*
NUMBER          [0-9][0-9]* ("."[0-9][0-9]*)*
DECIMAL          [0-9][0-9]* "."[0-9][0-9]*
BR              \r\n|\n|\r

%%

"/*"(.|\n|\r)*?"*/"                                                    return 'COMENTARIO_MULTI_LINEA'
("/""/")("/")*(.|{identifier}|{number}|{decimal}|{stringliteral})*      return 'COMENTARIO_LINEA';

{BR}+                       /* */
\s+                         /* */


{ID}                        return 'ID';
{DECIMAL}                   return 'DECIMAL';
{NUMBER}                    return 'NUMBER';
{BR}+                       return 'SCAPE';
\"("\\\\"|'\"'|[^"])*\"     yytext = yytext.replace(/\\"/g,'"'); return 'CADENA';
"'"("\\\\"|"\'"|[^'])*"'"   yytext = yytext.replace(/\\'/g,"'"); return 'CHAR';
"|"                         return 'PLECA';
"="                         return 'IGUAL';
"_"                         return 'GUION_BAJO';
"("                         return 'PAR_IZ';
")"                         return 'PAR_DER';
"+"                         return 'MAS';
"-"                         return 'MENOS';
"*"                         return 'POR';
"^"                         return 'POTENCIA';
"%"                         return 'PORCENTAJE';
"?"                         return 'INTERROGACION';
","                         return 'COMA';
";"                         return 'PUNTO_Y_COMA';
"<<EOF>>"                   return '$';
```

Cada vez que el archivo encontraba una coincidencia retornaba la descripcion de dicho token, por ejemplo, si encontraba "=" retorna "Igual", y asi con cada token, por el contario, si no encontraba coincidencia alguna, el token era un error lexico y retornaba "LEXICAL_ERROR" indicando que el token no era reconocido por el lenguaje.

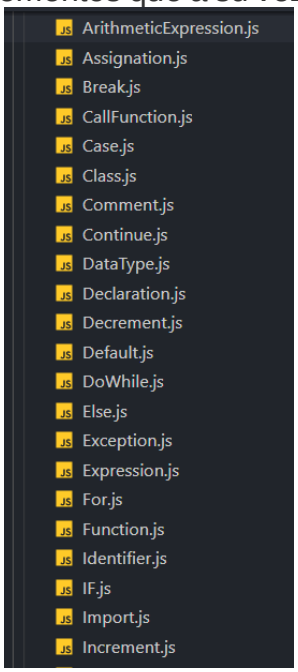
Para hacer uso del arhivo de analisis lexico, se genero la siguiente clase, llamada $myLexer.js$



Donde haciendo uso de la herramienta jison-lex, se importaba el archivo jison del análisis léxico y se exporta la clase para poder utilizar el scanner.

Análisis Sintáctico

El desarrollo del análisis sintáctico es más complicado, primero se procedió a generar los archivos o elementos necesarios para armar el árbol de análisis sintáctico, dichos elementos iban a ser los elementos que a su vez, iban a ser parte de la gramática.



Las clases anteriores, como se mencionó, son objetos representan a los elementos de la gramática y cada clase, recibe los parámetros necesarios que va a contener dentro del AST, por ejemplo, la clase IF tiene la siguiente estructura

```
class If  {
    /**
     * @constructor
     * @param condition Condicion que debe ser tipo boolean
     * @param list Lista de instrucciones a ejecutar en caso la condicion sea verdadera
     * @param ElseList Lista de instrucciones a ejecutar en caso la condicion sea falsa
     * @param line Linea de la sentencia if
     * @param column Columna de la sentencia if
     * @param name
     *      */
    constructor(cond, lst, el, l, c) {
        this.name = "If"
        this.line = l;
        this.column = c;
        this.condition = cond;
        this.list = lst;
        this.ElseList = el;
    }

}
exports.If = If;
```

El constructor de la clase recibe los parámetros que debe tener una sentencia if, por ejemplo la condición, la línea y columna, la lista de instrucciones internas, es decir, si dentro del if hay otro if, un for, una variable, etc. Asi como también recibe una lista de “else if” o “else ” en caso de que la condición sea falsa.

Al igual que con el if, el resto de clases (for, while, dowhile, etc) recibe los parámetros que hacen que la estructura sea correcta y pueda verse ordenada y limpia dentro del AST.

Gramática

Una vez generada cada estructura, se procede a armar la gramática, para ello primero se importan los elementos a utilizar

```
%{
    const {Tree}                = require('./components/Tree');
    const {DataType}            = require('./components/DataType');
    const {Declaration}         = require('./components/Declaration');
    const {Assignation}         = require('./components/Assignation');
    const {Expression}          = require('./components/Expression');
    const {Class}               = require('./components/Class');
    const {Import}              = require('./components/Import');
    const {If}                  = require('./components/If');
    const {Else}                = require('./components/Else');
    const {Identifier}          = require('./components/Identifier');
    const {Switch}              = require('./components/Switch');
    const {Case}                = require('./components/Case');
    const {Default}             = require('./components/Default');
    const {While}               = require('./components/While');
    const {DoWhile}             = require('./components/DoWhile');
    const {For}                 = require('./components/For');
    const {Increment}           = require('./components/Increment');
    const {Decrement}           = require('./components/Decrement');
    const {Break}               = require('./components/Break');
    const {Continue}            = require('./components/Continue');
    const {Return}              = require('./components/Return');
    const {Comment}             = require('./components/Comment');
    const {Function}            = require('./components/Function');
    const {Print}               = require('./components/Print');
    const {CallFunction}        = require('./components/CallFunction');
    const {ArithmeticExpression}= require('./components/ArithmeticExpression');
    const {LogicExpression}     = require('./components/LogicExpression');
    const {RelationalExpression}= require('./components/RelationalExpression');
    const {Exception}           = require('./components/Exception');
    const {Method}              = require('./components/Method');

%}
```

Y se estructura la gramática, la primera producción es INIT, esta llama a las instrucciones y a su vez crea el árbol de análisis sintáctico.

```
%start INIT

%%

INIT
    : INSTRUCCIONS EOF      {$$ = new Tree($1); return $$;}
    ;

//INSTRUCCIONES GLOBALES
INSTRUCCIONS
    : INSTRUCCIONS DECLARATION_TYPE   { $$ = $1; $$.push($2); }
    | DECLARATION_TYPE                { $$ = [$1]; }
    ;


//INSTRUCCIONES EN FUNCION O METODO
INSTRUCCIONS2
    : INSTRUCCIONS2 DECLARATION_TYPE_FUNCTION   { $$ = $1; $$.push($2); }
    | DECLARATION_TYPE_FUNCTION                 { $$ = [$1]; }
    ;
```

Las instrucciones principales son la siguientes

```
//GLOBALES
DECLARATION_TYPE
    : CLASS             {$$ = $1;}
    | DECLARATION       {$$ = $1;}
    | ASIGNATION        {$$ = $1;}
    | IF                {$$ = $1;}
    | COMMENTS          {$$ = $1;}
    | WHILE             {$$ = $1;}
    | DOWHILE           {$$ = $1;}
    | PRINT             {$$ = $1;}
    | FOR               {$$ = $1;}
    | SWITCH            {$$ = $1;}
    | VOID_METHOD       {$$ = $1;}
    | error             {$$ = new Exception($1, "ERROR SINTACTICO: " + $1 + " en linea: " + (this._$.first_line)
                                                + ", columna: " +  this._$.first_column, this._$.first_line, this._$.first_column)}

    ;
```

Al decir principales, se refiere a que estas están a nivel de clase, es decir, son instrucciones globales. Como los métodos o funciones. A su vez, también hay instrucciones “internas”, es decir, las que pueden ir dentro de los métodos, funciones y otras instrucciones.

```
//NIVEL DE METODO O FUNCION
DECLARATION_TYPE_FUNCTION
    : DECLARATION2      {$$ = $1;}
    | CALL_FUNCTION     {$$ = $1;}
    | ASIGNATION3       {$$ = $1;}
    | IF                {$$ = $1;}
    | COMMENTS          {$$ = $1;}
    | WHILE             {$$ = $1;}
    | DOWHILE           {$$ = $1;}
    | PRINT             {$$ = $1;}
    | FOR               {$$ = $1;}
    | SWITCH            {$$ = $1;}
    | 'continue' ';'    {$$ = new Continue($1, this._$.first_line, this._$.first_column);}
    | 'break' ';'       {$$ = new Break($1, this._$.first_line, this._$.first_column);}
    | 'return' ';'          {$$ = new Return( $1, $2, this._$.first_line, this._$.first_column);}
    | error             {$$ = new Exception($1, "ERROR SINTACTICO: " + $1 + " en linea: " + (this._$.first_line)
                                                + ", columna: " +  this._$.first_column, this._$.first_line, this._$.first_column)}
    ;
```

Cada vez que se declara una instrucción nueva, esta hace una llamada al archivo js que le corresponde, por ejemplo

```
/* SECCION METODO*/
VOID_METHOD
    : 'void' 'main' '('  ')'      '{' INSTRUCCIONS2 '}'    {$$ = new Method('void', 'main', [], $6, this._$.first_line, this._$.first_col
    | 'void' identifier '(' PARAMETERS ')' '{' INSTRUCCIONS2 '}'    {$$ = new Method('void', $2, $4, $7, this._$.first_line, this._$.fir
    ;

/*SECCION IF*/

IF : 'if' IF_CONDITION INSTRUCCIONS_BLOCK                       {$$ = new If($2, $3, [], this._$.first_line, this._$.first_column);}
   | 'if' IF_CONDITION INSTRUCCIONS_BLOCK 'else' INSTRUCCIONS_BLOCK {$$ = new If($2, $3, [new Else($5,this._$.first_line, this._$.first_
   | 'if' IF_CONDITION INSTRUCCIONS_BLOCK 'else' IF             {$$ = new If($2, $3, [$5], this._$.first_line, this._$.first_column)
   ;

IF_CONDITION
    : '(' EXPRESION ')' {$$ = $2;}
    ;

INSTRUCCIONS_BLOCK
    : '{' INSTRUCCIONS2 '}' {$$ = $2;}
    | '{' '}' {$$ = [];}
    ;

INSTRUCCIONS_BLOCK_CLASS
    : '{' INSTRUCCIONS '}' {$$ = $2;}
    | '{' '}' {$$ = [];}
    ;
```

Obsérvese que la gramática del if, al terminar de ejecutar la instrucción llama al la calse IF, que se genero con anterioridad y le envia los parámetros necesarios para su funcionamiento quedando un ast similar a este

```
If {
  name: 'If',
  line: 23,
  column: 4,
  condition: [Identifier],
  list: [Array],
  ElseList: []
}
```

Para las demás instrucciones (for, while, etc), se procede de la misma forma, se crea la estructura gramatical y al final, se llama a la clase correspondiente y se envían los parámetros para su ejecución

```
/* SECCION WHILE*/

WHILE : 'while' IF_CONDITION INSTRUCCIONS_BLOCK {$$ = new While($2, $3, this._$.first_line, this._$.first_column);}
      ;

/* SECCION DO WHILE*/

DOWHILE
    : 'do'  INSTRUCCIONS_BLOCK 'while' IF_CONDITION ';' {$$ = new DoWhile($2, $4, this._$.first_line, this._$.first_column);}
    ;

/* SECCION FOR*/

FOR
    : 'for' '(' TYPE_FOR ';' EXPRESION ';' INCREMENT_DECREMENT  ')' INSTRUCCIONS_BLOCK  {$$ = new For($3, $5, $7, $9, this._$.fir
    ;

TYPE_FOR
    : ASIGNATION_FOR  {$$ = $1;}
    | DECLARATION_FOR {$$ = $1;}
    ;

DECLARATION_FOR
    : TYPE identifier  '=' EXPRESION        {$$ = new Declaration($1, $2, $4, this._$.first_line, this._$.first_column);}
    ;
ASIGNATION_FOR : identifier '=' EXPRESION   {$$ = new Assignation($1, $3, this._$.first_line, this._$.first_column);}
    ;
```

Reporte de Copias

Para el reporte de copias, se generaron varios métodos, uno de ellos es el siguiente

```
compararClases(tree, treeCopy){

    /* VERIFICACION DE CLASES REPETIDAS*/

    //Verifico si el nombre es igual
    if(tree.id == treeCopy.id){

        //Busco los metodos y funciones de la clase 1
        this.getMethodsAndFunctions(tree.list, this.methodsClass1, this.functionClass1);

        //Busco los metodos y funciones de la clase 2
        this.getMethodsAndFunctions(treeCopy.list, this.methodsClass2, this.functionClass2);

        //verificar si la cantidad de elementos es la misma
        if(this.functionClass1.length == this.functionClass2.length
            && this.methodsClass1.length == this.methodsClass2.length){

                //Se ordenan los arreglos
                this.functionClass1.sort();
                this.functionClass2.sort();

                this.methodsClass1.sort();
                this.methodsClass2.sort();

                //verificar si las funciones se llaman igual
                var contadorFunciones = 0;
                for (let index = 0; index < this.functionClass1.length; index++) {
                    const element = this.functionClass1[index];
                    const element2 = this.functionClass2[index]
                    if(element == element2){
                        contadorFunciones++;
                    }
                }
```

Dicho método se utiliza para verificar si las clases son copias, primero recibe los ast generados de las clases a evaluar, luego verifica si las clases se llaman igual, si cumple, avanza si no, automáticamente las clases no son copia, si se llaman igual, se envían los ast a un metodo que busca los métodos y funciones para verificar la cantidad y si se llaman igual, si todo cumple la clase es copia, de lo contrario, indica que la clase no es copia y el por qué.

Al igual que con las clases, para el reporte de métodos y variables se generaron varios métodos que hacen funcional los reportes.

Cliente

Para el desarrollo del Cliente, se utilizo el lenguaje GO, así como JavaScript. Iniciando con GO, se desarrollo un “servidor” para lanzar la pagina y poder realizar las peticiones correspondientes.

Estructura Inicial

Se importan los elementos necesarios de GO que se utilizaran en la aplicación, como por ejemplo net/http que permitirá realizar las peticiones GET, POST necesarias

```
import (
    "encoding/json"
    "fmt"
    "log"
    "net/http"
    "github.com/gorilla/mux"
    "bytes"
    "io/ioutil"
    "bufio"
    "os"
    "strconv"
    "text/template"
)
```

Se procede a realizar una función llamada *indexHandler* que se encargara de lanzar la pagina html

```
func indexHandler(w http.ResponseWriter, req *http.Request){
    p := Page{ Tittle : "Proyecto 2", Text : "", Console: ""}
    t, _ := template.ParseFiles("index.html")
    t.Execute(w, p)
}
```

Por ultimo, el método main para levantar el server, en dicho método, se colocan los métodos http que serán utilizados en la aplicación (GET y POST) Y se indica el puerto por donde se accede al servidor.

```
func main() {
    router := mux.NewRouter()

    //endpoints
    router.HandleFunc("/", indexHandler)
    router.HandleFunc("/analizar", EndPoint).Methods("POST")
    router.HandleFunc("/lexerror", GetError).Methods("GET")
    router.HandleFunc("/sinteror", GetError).Methods("GET")


    log.Fatal(http.ListenAndServe(":8080", router))

}
```

Una vez realizada la estructura inicial del servidor, se procede a desarrollar las funciones necesarias para ejecutar el programa

Métodos HTTP

Se inicia con una función llamada EndPoint, dicha función se encargara de hacer las peticiones POST al servidor de Express

```go
func EndPoint(w http.ResponseWriter, r *http.Request) {

    if r.URL.Path != "/analizar" {
        http.Error(w, "404 not found.", http.StatusNotFound)
        return
    }
    switch r.Method {
    case "GET":
        //http.ServeFile(w, r, "index.html")
    case "POST":
        // Call ParseForm() to parse the raw query and update r.PostForm and r.Form.
        if err := r.ParseForm(); err != nil {
            fmt.Fprintf(w, "ParseForm() err: %v", err)
            return
        }
        //OBTENTO LA DATA DEL TEXT AREA
        content := r.FormValue("textAreaAnalizar")
        textoActual = content
        url := "http://localhost:3000/api/v1/server/analizar"

        //SE CREA EL JSON QUE SE VA A ENVIAR
        requestBody, err := json.Marshal(map[string]string{
            "entrada"   :   content,
            "consola"   :   "consola"})

        if err != nil {
            fmt.Println(err)
        }

        ////METODO POST
        resp, err := http.Post(url, "application/json", bytes.NewBuffer(requestBody))
```

Dicho método, recibe la dirección de la petición y el texto a analizar, se crea un json que contendrá el texto a analizar y se envía. El request resultante, se parsea a json nuevamente y se verifica si contiene errores léxicos o sintácticos

```go
    ///SE HACE UN JSON PARSE AL BODY DEL REQUEST PARA OBTENER LOS DATOS
    //OBTENIDOS AL HACER POST
    body, err := ioutil.ReadAll(resp.Body)
    if err != nil {
        panic(err)
    }
    var t Respuesta
    err = json.Unmarshal(body, &t)
    if err != nil {
        panic(err)
    }
    fmt.Println(t.AST)
    text := ""
    for i, s := range t.Console {
        fmt.Println(i)
        if t.Type == "lexico" {
            tipoError = "lexico";
            arrayLexico = append(arrayLexico, s)
        } else {
            tipoError = "sintactico"
            arraySintactico = append(arraySintactico, s)
        }
        text = text + s + "\n"
    }
    textoConsola = text
    //SE HACE UNA LLAMADA A LA PAGINA NUEVAMENTE
    p := Page{ Tittle : "Proyecto 2", Text : content, Console: text}
    temp, _ := template.ParseFiles("index.html")
    temp.Execute(w, p)
```

Dichos errores se guardan en unos arreglos declarados con anterioridad para ser exportados en los reportes.

Para obtener los errores, se hace una petición GET al servidor, para ello se creo una función llamada “GetError” que hace la petición y el request de la petición lo itera buscando los errores y armando el html de errores

```
func GetError(w http.ResponseWriter, r *http.Request) {
    url := "http://localhost:3000/api/v1/server/getError"


    ////METODO POST
    resp, err := http.Get(url)



    //CREA EL ARCHIVO HTML
    file, err :=os.Create("error.html")
    if err != nil {
        panic(err)
    }
    buf := bufio.NewWriter(file)

    var cadena = ""
    if tipoError == "lexico" {
        for i, s := range arrayLexico {
            a := strconv.Itoa(i)
            cadena = cadena + "<tr>\n" +
            "      <th scope=\"row\">" + a + "</th>\n" +
            "      <td>" + s + "</td>\n" +
            "</tr>"
        }
    } else {
        for i, s := range arraySintactico {
            a := strconv.Itoa(i)
            cadena = cadena + "<tr>\n" +
            "      <th scope=\"row\">" + a + "</th>\n" +
            "      <td>" + s + "</td>\n" +
            "</tr>"
        }
```