Manual Técnico

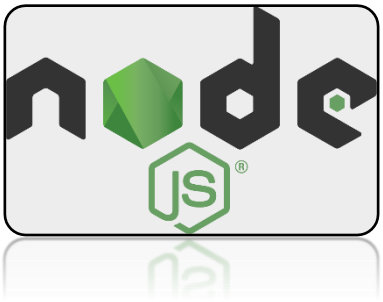
Organización de Lenguajes y Compiladores 1

José Rafael Morente González | 201801237
UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

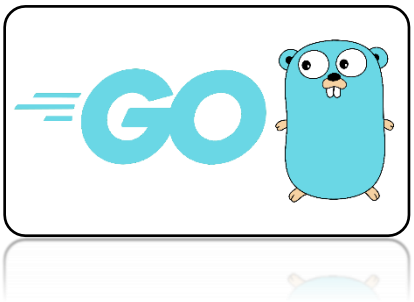
Contenido

Node JS

Node JS es utilizado para construir una API RESTful para poder acceder desde el frontend a partir de servicios los cuales nos ayudaran a determinar si un archivo es copia o no, para este proyecto la aplicación de Node JS fue creado en base a javascript. La aplicación fue creada para que corra en el puerto 3000.



Golang



Golang fue utilizado como frontend para consumir los servicios de Node JS. Y así mostrar los reportes y análisis realizados. La aplicación fue creada para que corra en el puerto 8100.

Análisis Léxico

Para el análisis léxico se utilizo la herramienta llamada Jison esta es utilizada para análisis léxico y sintáctico. El analizador léxico define los patrones para los tokens que deseamos reconocer. Hacemos uso de expresiones regulares para identificar números, cadenas y comentarios.

```
DIGITO          [0-9]
DIGITOTWO         [1-9]
DIGITOS         ("0"|{DIGITOTWO}{DIGITO}*)
EXPONENTE       ([Ee][+-]?{DIGITOS})
```

Nótese que los comentarios son tratados de la misma manera que los espacios en blanco, no retornamos ningún valor.

```
"//".*                  /* skip comments */
"/*"                    this.begin('comment');
<comment>"*/"           this.popState();
<comment>.              /* skip comment content*/
\s+                     /* skip whitespace */
```

Nótese también que para retornar los símbolos comunes del lenguaje se realizó lo siguiente en un archivo llamado lexico.jison para compilar este archivo se utilizó el siguiente comando **jison-lex lexico.jison** para ejecutar este comando tenemos que tener instalado jison-lex. Se puede realizar con el siguiente comando **npm jison-lex –save**

```
"*"                     return 'TK_Multiplicacion'
"/"                     return 'TK_Division'
"%"                     return 'TK_Modulo'
"-"                     return 'TK_Sustraccion'
"^"                     return 'TK_Potencia'
"+"                     return 'TK_Adicion'
";"                     return 'TK_PuntoComa'
"."                     return 'TK_Punto'
"<"                     return 'TK_Menor'
">"                     return 'TK_Mayor'
"!"                     return 'TK_Exclamacion'
"="                     return 'TK_Igual'
":"                     return 'TK_DosPuntos'
","                     return 'TK_Coma'
```

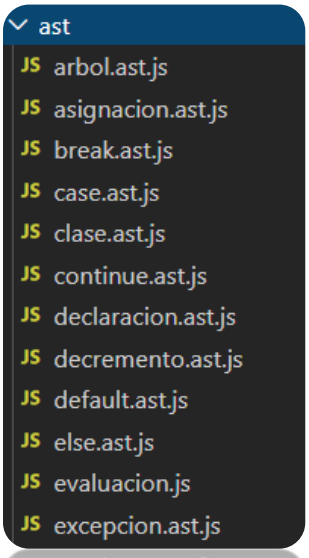
Análisis Sintáctico

El desarrollo del análisis sintáctico es más complicado, primero se procedió a generar las clases necesarias para armar el árbol de análisis sintáctico. Una vez generada cada estructura para el árbol AST, se procede a armar 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
    : INICIO_INSTRUCCION EOF                {$$ = new Arbol($1); return $$;}
    ;

INICIO_INSTRUCCION
    : INICIO_INSTRUCCION DECLARACION_TIPO   { $$ = $1; $$.push($2); }
    | DECLARACION_TIPO                      { $$ = [$1]; }
    ;

DECLARACION_TIPO
    : DECLARACION_CLASE             {$$ = $1;}
    | DECLARACION_VARIABLES        {$$ = $1;}
    | ASIGNACION_VARIABLES         {$$ = $1;}
    | SENTENCIA_IF                 {$$ = $1;}
```

Árbol AST

En lenguajes formales y lingüística computacional, un árbol de sintaxis abstracta (AST), o simplemente un árbol de sintaxis, es una representación de árbol de la estructura sintáctica simplificada del código fuente escrito en cierto lenguaje de programación. Cada nodo del árbol denota una construcción que ocurre en el código fuente. La sintaxis es abstracta en el sentido que no representa cada detalle que aparezca en la sintaxis verdadera.



Para armar este árbol se crearon varios objetos los cuales se introdujeron a la gramática de Jison para que sea fácil de armarlo. La raíz del árbol AST es el árbol.ast.js en el cual se van introduciendo todas las reglas de la gramática.

Ejemplo:

Para armar el AST de la **Clase** el constructor pide un identificador, la lista de instrucciones que esta misma tiene, los importes, fila y columna.

```
/**
* CONSTRUCTOR
*/
constructor(identificador, lista, importes, fila, columna) {
    this.nombre = "Clase";
    this.identificador = identificador;
    this.lista  = lista;
    this.importes  = importes;
    this.fila  = fila;
    this.columna = columna;
}
```

Desde Jison se realiza de la siguiente manera, {$$ = new Clase($3, $4, $1)}

Donde $ hace referencia a la posición que quiero guardar, y con this._$.first_line, this._$.first_column recupero la fila y columna.

```
DECLARACION_CLASE
    : DECLARACION_IMPORTE 'PR_class' identificador BLOQUE_CLASE {$$ = new Clase($3, $4, $1, this._$.first_line, this._$.first_column);}
    ;
```

Para obtener el árbol completo al final se realiza lo siguiente. La raíz apunta a la producción INIT entonces cuando realizo el análisis de parse se iguala a una variable para obtener el árbol AST.

```
/**
 * GENERAR ANALISIS SINTACTICO
 */
var arbolAST = analizadorSintactico.parse(data.entrada);
var arbolASTCopia = analizadorSintactico.parse(data.entradaCopia);
```