Universidad de San Carlos de Guatemala

Facultad de Ingeniería

Ciencias y Sistemas

Lab. Organización de Lenguajes y Compiladores 1

MANUAL TECNICO

Justin Josue Aguirre Román

202004734

INTRODUCCION

El presente documento describe los aspectos técnicos informáticos del sistema de información. El documento familiariza al personal técnico especializado encargado de las actividades de mantenimiento, revisión, solución de problemas, instalación y configuración del sistema, árboles y herramientas para su desarrollo.

OBJETIVO DE ESTE MANUAL

El objetivo primordial de este Manual es ayudar y guiar al técnico a informarse y utilizar herramientas utilizadas en el software auxiliar para la empresa “CompScript”, para de esa manera poder hacer uso de la información deseada para poder despejar todas las dudas existentes y para poder comprender:

• Guía para gestión de herramientas para poner en funcionamiento el sistema de manejo para impresiones.

• Conocer cómo utilizar el sistema, mediante una descripción detallada e ilustrada de las opciones.

• Conocer el alcance de toda la información por medio de una explicación detallada e ilustrada de cada una de las páginas que lo conforman el manual técnico.

REQUERIMIENTOS

El sistema pude ser instalado en cualquier sistema operativo que cumpla con los siguientes requerimientos:

a. Sistema Operativo: Cualquiera con una fecha de salida del 2014 en adelante.

a. Windows

i. Windows 10 (8u51 y superiores)

ii. Windows 8.x (escritorio)

iii. Windows 7 SP1

iv. Windows Vista SP2

v. Windows Server 2008 R2 SP1 (64 bits)

vi. Windows Server 2012 y 2012 R2 (64 bits)

vii. RAM: 128 MB

viii. Espacio en disco: 124 MB para JRE; 2 MB para Java Update

ix. Procesador: Mínimo Pentium 2 a 266 MHz

x. Exploradores: Internet Explorer 9 y superior, Firefox

b. Linux

i. Oracle Linux 5.5+1

ii. Oracle Linux 6.x (32 bits), 6.x (64 bits)2

iii. Oracle Linux 7.x (64 bits)2 (8u20 y superiores)

iv. Red Hat Enterprise Linux 5.5+1 6.x (32 bits), 6.x (64 bits)2

v. Red Hat Enterprise Linux 7.x (64 bits)2 (8u20 y superiores)

vi. Suse Linux Enterprise Server 10 SP2+, 11.x

vii. Suse Linux Enterprise Server 12.x (64 bits)2 (8u31 y superiores)

viii. Ubuntu Linux 12.04 LTS, 13.x

ix. Ubuntu Linux 14.x (8u25 y superiores)

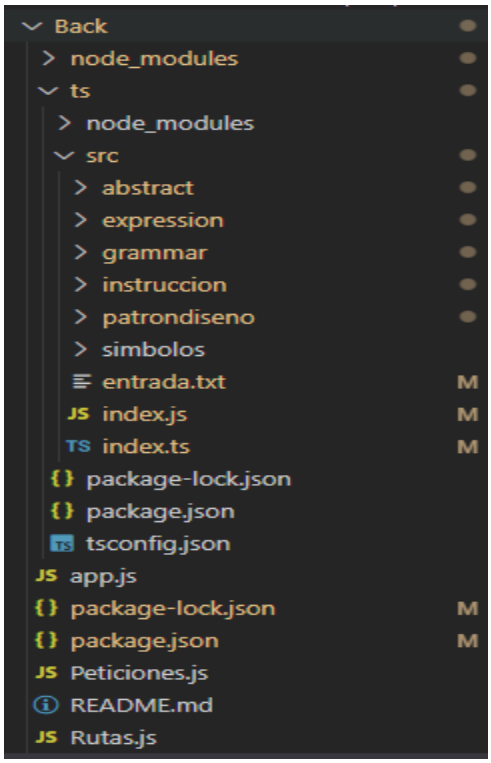
x. Ubuntu Linux 15.04 (8u45 y superiores)

xi. Ubuntu Linux 15.10 (8u65 y superiores

METODOS UTILIZADOS

El desarrollo del software se realizo se le manejo una arquitectura Cliente-Servidor, con el objetivo de que pueda separar los servicios administrados por el intérprete, de la aplicación cliente que se mostrará al usuario final.

Para el funcionamiento del servidor se trabajo en un lenguaje JavaScript, creando diversos archivos (principal y acciones realizables) y a su vez se instalaron distintas dependencias (node, jison, express, cors, etc).



Se estipula con el archivo Ruta.js y app.js la configuración del servidor, manejo de información y peticiones posibles que podrá realizar el cliente.

Para el desarrollo del software se dispuso de la creación de distintas variables globales para el intercambio de información entre los distintos métodos funcionamiento del software.

Comenzando con las bases para la realización del proceso de análisis tenemos como primer paso establecer comunicación entre los diversos archivos que se enlazan durante el tiempo de ejecución del software. El recibimiento de una orden nos lleva al archivo app.js que al detectar y reconocer una petición nos llevara a Rutas.js; este mismo es el primer paso de comunicación entre la gramática y el mensaje de entrada.

```
routes.get('/Ejecutar',function(req,res){
    salida="";
    const Environment_1 = require("./ts/src/simbolos/Environment");
    const parser = require("./ts/src/grammar/gramatica");
    const ast = parser.parse(entrada.toString());
    const env = new Environment_1.Environment(null);
    var salida = new singleton_1.Singleton();
```

Podemos observar que la constante **Environment_1** hace de su requerimiento un archivo externo al igual que **parser,** este último el encargado de conectar el archivo de gramática.js y gramática.jison que serán explicados en el documento en esta misma carpeta titulado como “GRAMATICA.PDF”. Pero que a grandes rasgos deberían mantener el siguiente formato:

COMENTARIO: Los comentarios son una forma elegante de indicar que función tiene cierta sección del código que se ha escrito simplemente para dejar algún mensaje en específico.

Estructura:

// Este es un comentario de una línea

/* Este es un comentario
Multilínea Para este lenguaje */

DECLARACION: Una variable deberá de ser declarada antes de poder ser utilizada. Todas las variables tendrán un tipo de dato y un nombre de identificador.

Estructura:

<Tipo> id;
<Tipo> id1, id2, id3, id4;
<Tipo> identificador = **<Expresión>**;
<Tipo> id1, id2, id3, id4 = **<Expresión>**;

ASIGNACION: Una variable deberá de ser declarada antes de poder ser utilizada. Todas las variables tendrán un tipo de dato y un nombre de identificador.

Estructura:

identificador = **<Expresión>**;
id1, id2, id3, id4 = **<Expresión>**;

CASTEOS: Los casteos son una forma de indicar al lenguaje que convierta un tipo de dato en otro, por lo que, si queremos cambiar un valor a otro tipo, es la forma adecuada de hacerlo.

Estructura:

(**<Tipo>**) **<Expresión>**;

INCREMENTO: si incrementamos una variable, se incrementará de uno en uno su valor.

DECREMENTO: si decrementamos una variable, se disminuirá de uno en uno su valor.

Estructura:

<Expresion>'+''+'';

<Expresion>' '-''-'';'

VECTORES: Los vectores son una estructura de datos de tamaño fijo que pueden almacenar valores de forma limitada, y los valores que pueden almacenar son de un único tipo; int, double, boolean, char o string.

Estructura:

DECLARACION TIPO 1 (una dimensión)

<Tipo> 'Id' '[' ']' = new **<Tipo>** '[' **<Expresion>** ']' ';'

<Tipo> 'Id' '['']''['']'= new **<Tipo>**'[' **<Expresion>** ']''['**<Expresion>**']' ';'

DECLARACION TIPO 2

<Tipo> 'Id' '[' ']' = '[' **<LISTAVALORES>** ']' ';'

VECTORES: La sentencia if ejecuta las instrucciones sólo si se cumple una condición. Si la condición es falsa, se omiten las sentencias dentro de la sentencia.

Estructura:

'if' '(' **<Expresion>** ')' '{' **<Instrucciones>** '}'

| 'if' '('**<Expresion>**')''{'**<Instrucciones>**'}''else''{' **<Instrucciones>** '}'

| 'if' '(' **<Expresion>** ')' '{' **<Instrucciones>** '}' 'else' **<IF>**

SWITCH: Estructura principal del switch, donde se indica la expresión a evaluar.

Estructura:

'switch' '(' **<Expresion>** ')' '{' **<CaseList> <Default>** '}'

| ‘switch’ ‘(’ **<Expresion>** ‘)’ ‘{‘ **<CaseList>** ‘}’

| ‘switch’ ‘(’ **<Expresion>** ‘)’ ‘{‘ **<Default>** ‘}’

CASE: Estructura que contiene las diversas opciones a evaluar con la expresión establecida en el switch.

Estructura:

‘case’ **<Expresion>** ‘:’ **<Instrucciones>**

WHILE: El ciclo o bucle While, es una sentencia que ejecuta una secuencia de instrucciones mientras la condición de ejecución se mantenga verdadera.

Estructura:

‘while’ ‘(’ **<Expresion>** ‘)’ ‘{‘ **<Instrucciones>** ‘}’

FOR El ciclo o bucle for, es una sentencia que nos permite ejecutar N cantidad de veces la secuencia de instrucciones que se encuentra dentro de ella.

Estructura:

‘for‘(’<Declaracion>|<Asignacion>’;’<Condicion>’;’<Actualizacion>‘)’‘{‘<Instruccion> ‘}’

DO_WHILE: El ciclo o bucle Do-While, es una sentencia que ejecuta al menos una vez el conjunto de instrucciones que se encuentran dentro de ella y que se sigue ejecutando mientras la condición sea verdadera

Estructura:

‘do’ ‘{‘ <Instrucciones> ‘}’ ‘while’ ‘(’ <Expresion> ‘)’ ‘;’

BREAK: La sentencia break hace que se salga del ciclo inmediatamente, es decir que el código que se encuentre después del break en la misma iteración no se ejecutara y este se saldrá del ciclo.

Estructura:

‘break**’ ;**

CONTINUE: La sentencia continue puede detener la ejecución de la iteración actual y saltar a la siguiente. La sentencia continue siempre debe de estar dentro de un ciclo, de lo contrario será un error.

Estructura:

‘continue**’ ;**

RETURN: La sentencia return finaliza la ejecución de un método o función y puede especificar un valor para ser devuelto a quien llama a la función.

Estructura:

‘return**’ ;**

| ‘return**’ <Expresion> ;**

FUNCIONES: Una función es una subrutina de código que se identifica con un nombre, tipo y un conjunto de parámetros. Para las funciones es obligatorio que las mismas posean un valor de retorno que coincida con el tipo con el que se declaró la función.

Estructura:

‘Id’ ‘(‘ **<Parametros>** ‘)’ ’:’ <Tipo> ‘{‘ **<Instrucciones>** ‘}’

Parametros -> **<Parametros>** , **<Tipo>** ‘id’

| **<Tipo>** ‘id’

METODOS: Un método también es una subrutina de código que se identifica con un nombre, tipo y un conjunto de parámetros, aunque a diferencia de las funciones estas subrutinas no deben de retornar un valor.

Estructura:

‘Id’ ‘(‘ **<Parametros>** ‘)’ ’:’ ‘void’ ‘{‘ **<Instrucciones>** ‘}’

Parametros -> **<Parametros>** , **<Tipo>** ‘id’

| **<Tipo>** ‘id’

LLAMADA: La llamada a una función específica la relación entre los parámetros reales y los formales y ejecuta la función. Los parámetros se asocian normalmente por posición, aunque, opcionalmente, también se pueden asociar por nombre. Si la función tiene parámetros formales por omisión, no es necesario asociarles un parámetro real.

Estructura:

LLAMADA ->‘Id’ ‘(‘ **<Parametros>** ‘)’

|‘Id’ ‘(‘ ‘)’

Parametros -> **<Parametros>** , ‘id’

| ‘id’

PRINT: Esta función nos permite imprimir expresiones con valores únicamente de tipo entero, doble, booleano, cadena y carácter. Esta función no concatena un salto de línea al final del contenido que recibe como parámetro.

Estructura:

‘Print’ ‘(‘ <Expresion>‘)’;

PRINTLN: Esta función nos permite imprimir expresiones con valores únicamente de tipo entero,doble, booleano, cadena y carácter. Esta función concatena un salto de línea al finalizar el contenido que recibe como parámetro.

Estructura:

‘Println’ ‘(‘ <Expresion>‘)’;

TO_LOWER: Esta función recibe como parámetro una expresión de tipo cadena y retorna una nueva cadena con todas las letras minúsculas.

Estructura:

‘toLower’ ‘(‘ <Expresion>‘)’;

TO_UPPER: Esta función recibe como parámetro una expresión de tipo cadena retorna una nueva cadena con todas las letras mayúsculas.

Estructura:

‘toUpper’ ‘(‘ <Expresion>‘)’;

ROUND: Esta función recibe como parámetro un valor numérico.

Estructura:

‘round ‘(‘ <Valor>‘)’;

LENGTH: Esta función recibe como parámetro un vector, una lista o una cadena y devuelve el tamaño deeste.

Estructura:

‘length ‘(‘ <Valor>‘)’;

TYPE_OF: Esta función retorna una cadena con el nombre del tipo de dato evaluado.

Estructura:

‘typeof ‘(‘ <Valor>‘)’;

TO_STRING: Esta función permite convertir un valor de tipo numérico o booleano en texto.

Estructura:

‘toString ‘(‘ <Valor>‘)’;

TO_CHAR_ARRAY: Esta función permite convertir una cadena en un vector de caracteres

Estructura:

‘toCharArray ‘(‘ <Valor>‘)’;

RUN: Para poder ejecutar todo el código generado dentro del lenguaje, se utilizará la sentencia RUN para indicar que método o función es la que iniciará con la lógica del programa.

Estructura:

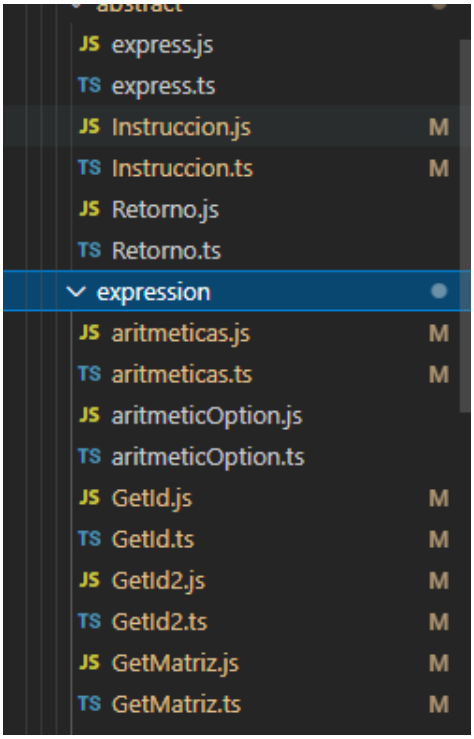
‘run ‘(’ ’)’ ‘;’

‘run’ ‘(’ <LISTAVALORES> ’)’ ‘;’

LISTAVALORES-> LISTAVALORES ‘,’ EXPRESION

| EXPRESION

Luego de fase de análisis por medio de archivos de lectura léxica y sintáctica pasamos a la ejecución y análisis semántico del software. Usando diversas clases, una para cada tipo de instrucción que podría dar lugar el archivo de entrada inicial, pues estas también vienen en sub-grupos. Tenemos clases que derivan de clases abstractas a modo de molde que seguirán todas las demás, permitiendo la ejecución de un mismo método con distintas sentencias.




```typescript
import { Singleton } from "../patrondiseno/singleton"
import { Environment } from "../simbolos/Environment"

export abstract class Instruccion{
    public i=0;
    constructor(public line:number,public column:number){
        this.line=line
        this.column=column
    }

    public abstract execute(env:Environment,sn:Singleton):any
    public abstract execute2(env:Environment,sn:Singleton):any
}
```

DEVOLUCION DE DATOS

Para cada tipo de instrucción solicitada habrá dos tipos de ejecución y cada uno solicitara distintos tipos de información. Pero los únicos capaces de devolver información hasta el servidor cliente son las instrucciones Print y Println que guardaran su valor indicado para que al momento de su ejecución este se guarde en forma de texto plano que será transmitido por medio de un objeto de tipo Singleton que tendrá la capacidad de guardar información en alguna posición de un Array dependiendo del caso.

El tipo Singleton tiene propiedades llamadas getMsg y SetMsg que son las encargadas de como su nombre lo indica realizar un set o un get al mensaje almacenado hasta ese momento.

SENTENCIAS DE TRANSFERENCIA

Estas son las instrucciones break, continue y return que cortaran el proceso de un ciclo, un método o una función. En todos estos casos se siguió la misma regla: el retorno de un objeto de tipo Retorno que nos dirá cual de las 3 opciones ha ocurrido y si poseía un valor (en caso del return). Además que el ambiente del cual han ocurrido estas sentencias hace la verificación de si la sentencia realizada es unos de estos casos y de serlo hará nuevamente lo mismo, retornar según el caso.

```
export class BREAK extends Instruccion {
    constructor(
        line: number,
        column: number
    ) {
        super(line, column);
    }

    public execute2(env: Environment) {
    }
    public execute(env: Environment,sn:Singleton):Retorno {
        let val={
            value:true,
            type:Type.BREAK
        }
        return val;
    }
```

SENTENCIAS CICLICAS

Refiriendonos únicamente al for, while y do while que mantendrán una estructura muy similar: una condicionante que al hacer cada uno de los recorridos por el ciclo se volverá a correr para la verificación actualizada.

Unicamente para el for se contara con un asignación o una declaración en su sentencia inicial.

```
export class FOR extends Instruccion {
    constructor(
        public Param1: Instruccion,
        public expresionB: Expression,
        public Param2: Instruccion,
        public Intrucciones: Array<Instruccion> | null,
        line: number,
        column: number
    ) {
        super(line, column);
    }
    public execute2(env: Environment) {
    }
    public execute(env: Environment,sn:Singleton) {
        const envFor = new Environment(env);
        this.Param1.execute(envFor,sn);
        let exp = this.expresionB.execute(envFor,sn);
        if (exp.type == Type.BOOLEAN) {

            while (Boolean(exp.value)) {
                const envFor2 = new Environment(envFor);
                var breakOp = false
                if (this.Intrucciones != null) {
                    for (const x of this.Intrucciones) {
                        var corte: Retorno = x.execute(envFor2,sn);
                        if (corte != undefined) {
                            if (corte.type == Type.BREAK) {
                                breakOp = true;
                                break;
                            } else if (corte.type == Type.CONTINUE) {
                                break;
```

REPORTES

Para los tipos de reportes lo que se ejecuto fue únicamente una tabla con la información obtenida de la ejecución del software, esta tabla será en formato HTML y se abrirá al momento de seleccionarlo.

FRONTED

Para esta parte del software únicamente se desarrollo una pestaña con distintas sub divisiones para mantener un control mas marcado sobre que parte de la interfaz desarrolla ciertas funciones.

Se utilizo por ejemplo un tipo de función especial para la actualización, en cualquier momento, de contenido dentro de la caja de texto Entry. Funcion que también será la encargada de leer el texto de un archivo seleccionado y colocarlo en la caja Entry.

```
const handleChange = e => {
    setMessageE({
        [e.target.name]: e.target.value
    })
    console.log(MessageE.msg)
  }
  let {msg} = MessageE


const refresh = () =>{
    setMessageE({msg : ruta.toString()})
}
```