Natalia Mariel Calderón Echeverría
202200007
Organización de Lenguajes y Compiladores 1
Primer semestre 2024

Manual de Tecnico

Requerimientos básicos:

- **Lenguaje programado:** Javascript
    - uso de html para reportes y el **front-end** del proyecto.
- **Librerías:**
    - **Jison**

      Java cup es la librería encargada de la creación del analizador, léxico y sintáctico. Es la herramienta que permite generar la gramática
    - **Node.js**

      Node.js es un entorno de ejecución de JavaScript que facilita la ejecución y por ende el uso de Javascript al momento de programar.
    - **Nodemon**

      A lo largo del proyecto fue utilizado para facilitar el desarrollo y las pruebas controladas de los distintos aspectos del lenguaje.

Requerimientos del equipo:

Nombre	Estado	2% CPU	26% Memoria	1% Disco	0% Red	0% GPU	Motor de la GPU
Aplicaciones (5)							
Administrador de tareas		0.4%	26.2 MB	0 MB/s	0 Mbps	0%	
Brave Browser (21)		0.1%	1,146.0 MB	0.1 MB/s	0 Mbps	0%	GPU 0 - 3D
Explorador de Windows		0.1%	49.4 MB	0 MB/s	0 Mbps	0%	
Microsoft Edge (6)		0%	115.7 MB	0 MB/s	0 Mbps	0%	
Visual Studio Code (20)		0.9%	773.0 MB	0.1 MB/s	0 Mbps	0%	GPU 0 - 3D

Funcionalidades importantes del programa

Este programa se desarrolló con ayuda de la herramienta Jison y haciendo uso del patrón interpreter. Esto con el fin de abordar de mejor manera la complejidad del programa y las distintas clases que interactúan.

Si bien es cierto, a lo largo de la gramática existen distintas producciones vitales y el programa hace uso de todas y cada una de ellas, considero que la base de este proyecto se centra en la producción de expresiones, en donde interactúan distintos tipos de clases como lo es la clase: Aritméticas, Dato, Relacionales, Lógicas, funciones nativas, entre otros.

```
expresion
        : MENOS expresion %prec UMENOS  { $$ = new Negativo($2, @1.first_line, @1.first_column); }
    | exclamacion expresion %prec UEX {$$ = new Not($2, @1.first_line, @1.first_column);}
    | CADENA    { $$ = new Dato($1, TipoDato.CADENA, @1.first_line, @1.first_column); }
    | CHAR  { $$ = new Dato($1, TipoDato.CHAR, @1.first_line, @1.first_column); }
    | BOOLEAN   { $$ = new Dato($1, TipoDato.BOOLEAN, @1.first_line, @1.first_column); }
    | DOUBLE    { $$ = new Dato($1, TipoDato.DOUBLE, @1.first_line, @1.first_column); }
    | INT   { $$ = new Dato($1, TipoDato.INT, @1.first_line, @1.first_column); }
    | expresion mayorIgual expresion       {$$ = new Relacionales($1, $2, $3, @1.first_line, @1.first_column);}
    | expresion mayorQue expresion       {$$ = new Relacionales($1, $2, $3, @1.first_line, @1.first_column);}
    | expresion menorIgual expresion       {$$ = new Relacionales($1, $2, $3, @1.first_line, @1.first_column);}
    | expresion menorQue expresion       {$$ = new Relacionales($1, $2, $3, @1.first_line, @1.first_column);}
    | expresion diferente expresion       {$$ = new Relacionales($1, $2, $3, @1.first_line, @1.first_column);}
    | expresion dosIgual expresion       {$$ = new Relacionales($1, $2, $3, @1.first_line, @1.first_column);}
    | expresion modulo expresion       {$$ = new Aritmetica($1, $2, $3, @1.first_line, @1.first_column);}
    | expresion DIVIDIR expresion       {$$ = new Aritmetica($1, $2, $3, @1.first_line, @1.first_column);}
    | expresion MENOS expresion       {$$ = new Aritmetica($1, $2, $3, @1.first_line, @1.first_column);}
    | expresion MAS expresion       {$$ = new Aritmetica($1, $2, $3, @1.first_line, @1.first_column);}
    | expresion POR expresion       {$$ = new Aritmetica($1, $2, $3, @1.first_line, @1.first_column);}
    | expresion oSigno expresion       {$$ = new Logicos($1, $2, $3, @1.first_line, @1.first_column);}
    | expresion And expresion       {$$ = new Logicos($1, $2, $3, @1.first_line, @1.first_column);}
    | expresion interrogracion expresion  dosPuntos expresion   {$$ = new Ternario($1, $3, $5, @1.first_line, @1.first_column);}
    | expresion MASmas   {$$ = new IncDec($1, $2, @1.first_line, @1.first_column);}
    | expresion MENOSmenos {$$ = new IncDec($1, $2, @1.first_line, @1.first_column);}
    | TOLOWER abrirPar expresion cerrarPar {$$= new lower($3,@1.first_line, @1.first_column);}
    | TOUPPER abrirPar expresion cerrarPar {$$= new upper($3,@1.first_line, @1.first_column);}
    | ROUND abrirPar expresion cerrarPar {$$= new round($3,@1.first_line, @1.first_column);}
    | TOSTRING abrirPar expresion cerrarPar {$$= new toString($3,@1.first_line, @1.first_column);}
    | TYPEOF abrirPar PALABRA_I cerrarPar {$$= new TypeOf($3,@1.first_line, @1.first_column);}
    | potencia abrirPar expresion coma expresion cerrarPar {$$=new Potencia($3,$5,@1.first_line, @1.first_column);}
    | abrirPar parentesis_exp {$$=$2;}
    | PALABRA_I palabras {
```

La expresión expresión se usa a lo largo de la gramática declarada, ya que es la base de los resultados que se piensan obtener. Es decir, también se hace referencia a ella en instrucciones vitales como lo es el declarar variables, arreglo, ciclos, cout, entre otras.

```
print
    : COUT CorImp expresion tipos_print  { $$ = new Print($3,$4, @1.first_line, @1.first_column); }
;
```

Como se ha mostrado previamente, la producción de expresiones presenta recursividad y, en muchos casos, ambigüedad, ya que la mayoría de las expresiones comienzan con una estructura similar seguida de un token distintivo. Esta ambigüedad se ha resuelto mediante la declaración de una jerarquía de operaciones al inicio de la gramática. Esta jerarquía no solo elimina la ambigüedad, sino que también facilita la ejecución de operaciones matemáticas sin contratiempos.

```
%left  'MASmas', 'MENOSmenos'
%left   'interrogracion'
%left   'oSigno'
%left   'And'
%right UEX
%left 'dosIgual','menorQue','mayorQue','menorIgual','mayorIgual','diferente'
%left 'MAS','MENOS'
%left 'POR','DIVIDIR', 'modulo'
%left 'abrirPar' 'cerrarPar' 'abrirCor' 'cerrarCor'

%right UMENOS
```

Esta fue una breve explicación de la columna vertebral de la gramática que permite posteriormente hacer un uso correcto del patrón interpreter. El patrón Interpreter es una de las herramientas de diseño que permiten abordar lenguajes complejos de mejor manera, pero a su vez el mismo patrón puede resultar algo confuso para alguien que recién está siendo introducido a él, esto se debe a que incorpora conceptos clase como la Herencia, Polimorfismo y la Recursividad.

Uno de los principales aciertos a la hora de realizar este proyecto fue la separación entre instrucción y expresión y el correcto tipado de las mismas. Esta es una de las clases principales de expresión que permite identificar el tipo de dato con el que se está trabajando.

```
class Instruccion{
    constructor(tipo, fila, columna){
        this.tipo = tipo;
        this.fila = fila;
        this.columna = columna;
    }

    interpretar(entorno,tablaDeSimbolos,sb){}
}

const TipoInstr = {
    PRINT: 'PRINT',
    IF: 'IF',
    Variable: 'Variable',
    INC_DEC: 'INC_DEC',
    ARREGLO: 'Arreglo',
    FUNCION: 'FUNCION',
    METODO: 'METODOS',
    EXECUTE: 'EXECUTE',
    IF_ELSE: 'IF_ELSE',
    ELSE: 'ELSE',
    WHILE: 'WHILE',
    DOWHILE: 'DOWHILE',
    FOR: 'FOR',
    BREAK: 'BREAK',
    CONTINUE: 'CONTINUE',
    RETURN: 'RETURN',
    SWITCH: 'SWITCH'
}

module.exports = {Instruccion, TipoInstr}
```

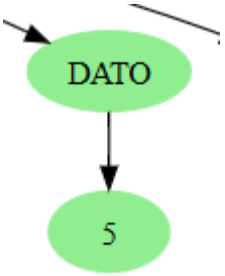
```
class Expresion{

    constructor(valor, tipo, fila, columna){
        this.valor = valor;
        this.tipo = tipo;
        this.fila = fila;
        this.columna = columna;
    }

    interpretar(entorno,tablaDeSimbolos,sb){

    }

}

const TipoDato = {
    INT: 'INT',
    DOUBLE: 'DOUBLE',
    BOOLEAN: 'BOOL',
    CHAR: 'CHAR',
    CADENA: 'STD::STRING',
    ERROR: 'ERROR'
}

module.exports = {Expresion, TipoDato}
```

En base a estas clases previas principales, se hereda de ella y se general las distintas clases que serán utilizadas a lo largo de la gramática. Una de las clases más importantes relacionadas con las expresiones es la clase dato y las distintas clases operacionales (lógicas, relacionales, aritméticas, Incrementar, entre otras.)

```
const {Expresion} = require("../Expresion.js")
const StringBuilder = require('../StringBuilder.js');
const NodoAst_1 = require("../Simbolos/NodoAst");

class Dato extends Expresion{
    constructor(valor, tipo,linea, columna){
        super(valor, tipo,linea, columna);
        //this.tipo = tipo;
        this.valor = valor;
        //this.linea = linea;
        //this.columna = columna;
    }

    getNodo() {
        let nodo = new NodoAst_1.NodoAst('DATO');
        nodo.agregarHijo(this.valor);
        return nodo;
    }

    interpretar(entorno,tablaDeSimbolos,sb){
        switch(this.tipo){
            case 'INT': return Number(this.valor);
            case 'STD::STRING': return this.valor;
            case 'DOUBLE':return parseFloat(this.valor);
            case 'CHAR': return this.valor;
            case 'ERROR': return this.valor;
            case 'BOOL':
                    let lowerCaseValue = this.valor.toLowerCase();
                    if (lowerCaseValue === 'true') {
                        return this.valor = true;
                    } else if (lowerCaseValue === 'false') {
                        return this.valor = false;
                    } else {
                        sb.append("Error Semántico -> VALOR BOOLEANO NO VALIDO");
                        throw new Error("Error Semántico -> VALOR BOOLEANO NO VALIDO: " + this.valor);

                    }
                    break;
```

Esta clase, llamada Dato, extiende la clase Expresión e implementa funcionalidades para interpretar diferentes tipos de datos. Su constructor recibe un valor, un tipo, y coordenadas de línea y columna.

La función getNodo() devuelve un nodo del árbol abstracto sintáctico con el valor del dato como hijo, esta función es la que realiza todo lo relacionado con la representación de un dato en el AST. La función interpretar() analiza el tipo del dato y realiza la conversión adecuada, como convertir a número entero, cadena, doble, caracter o booleano. En caso de un valor booleano, verifica y convierte el texto en booleano y maneja errores semánticos.



Otra clase importante relacionada con las expresiones, es la clase Lógicos, que maneja todo lo relacionado con instrucciones logicas, siendo su

respuesta True o False. Su buen funcionamiento es la base del buen funcionamiento de las instrucciones, Do while, while, if, if else, else y for.

```
const { Expresion, TipoDato } = require("../Expresion");
const StringBuilder = require('../StringBuilder.js');
const NodoAst_1 = require("../Simbolos/NodoAst");

class Logico extends Expresion{

    constructor(expIzq, operador, expDer, fila, columna){
        super("BOOL", TipoDato.BOOLEAN, fila, columna);
        this.expIzq = expIzq;
        this.expDer = expDer;
        this.operador = operador;
    }

    getNodo() {
        let nodo = new NodoAst_1.NodoAst('EXPRESION - LOGICA');
        nodo.agregarHijoAST(this.expIzq.getNodo());
        if (this.operador == "&&") {
            nodo.agregarHijo("&&");
        }
        else if (this.operador == "||") {
            nodo.agregarHijo("||");
        }
        nodo.agregarHijoAST(this.expDer.getNodo());
        return nodo;
    }

   interpretar(entorno,tablaDeSimbolos,sb){
        let expDer = this.expDer.interpretar(entorno,tablaDeSimbolos,sb);
        let expIzq = this.expIzq.interpretar(entorno,tablaDeSimbolos,sb);



        if(this.operador == "&&"){
            if (this.expDer.tipo=="BOOL" && this.expIzq.tipo=="BOOL" ) {
                let resultado = expIzq && expDer;
                return resultado;
            }
            else{
                console.log("Error Semántico: dentro operador logico && - recuerde ambas exp debe dar un resultado booleano (true/false)");
                sb.append("\n");
                sb.append("Error Semántico: dentro operador logico && - recuerde ambas exp debe dar un resultado booleano (true/false)");
                sb.append("\n");
                this.tipo=TipoDato.ERROR;
                return this;
            }
```

La clase Lógico extiende de la clase Expresion, lo que significa que hereda sus propiedades y métodos. En su constructor, recibe tres parámetros que representan las expresiones izquierda y derecha, así como el operador lógico que las une. Luego, tiene dos métodos principales: getNodo() y interpretar().

El método getNodo() se encarga de construir un nodo del Árbol de Sintaxis Abstracta (AST) para representar la expresión lógica. Dependiendo del operador (&& o ||), agrega los hijos correspondientes al nodo, que son los nodos AST de las expresiones izquierda y derecha.

Por otro lado, el método interpretar() es responsable de evaluar la expresión lógica en tiempo de ejecución. Primero, interpreta las expresiones izquierda y derecha recursivamente para obtener sus valores. Luego, verifica si ambas expresiones tienen el tipo de dato booleano (BOOL). Si es así, realiza la operación lógica (&& o ||) y devuelve el resultado. En caso contrario, registra un error semántico indicando que las expresiones deben ser booleanas y establece el tipo de dato de la expresión como un error. Finalmente, retorna el resultado de la operación lógica o la expresión con error, según corresponda.

```
class tablaSimbolos {
    constructor() {
        this.tablaSimbolos = [];
    }

    agregarSimbolo(simb) {
        this.tablaSimbolos.push(simb);
        //console.log(this.tablaSimbolos);
    }

    limpiarVariablesTemporales() {
        this.tablaSimbolos = this.tablaSimbolos.filter(variable => variable.tipoVar !== "temporal");
        console.log("Variables temporales eliminadas.");
    }

    actualizarValor1D(id, indice, nuevoValor, entorno) {
        const index = this.getSimboloIndex(id, entorno);
        console.log("valor que entra a la func ", nuevoValor);
        if (index !== -1) {
            const symbol = this.tablaSimbolos[index];
            if (Array.isArray(symbol.valor) && indice >= 0 && indice < symbol.valor.length) {
                symbol.valor[indice] = nuevoValor;
                console.log(`Valor actualizado en la posición ${indice} del símbolo ${id}`);
                this.reporteTabla(); // actualizar el reporte
            } else {
                console.log(`Índice ${indice} fuera de rango`);
            }
        } else {
            console.log(`No se puede actualizar el valor, el símbolo ${id} no existe`);
        }
    }
```

Otra clase que vale la pena explicar el funcionamiento y es vital para el buen funcionamiento de este programa es la clase TablaSimbolos.js

En su constructor, inicializa una matriz vacía tablaSimbolos que almacenará los símbolos creados durante la ejecución del programa.

Los métodos de la clase incluyen funciones para agregar un símbolo a la tabla ***agregarSimbolo***, limpiar variables temporales de la tabla ***limpiarVariablesTemporales***., actualizar el valor de un símbolo unidimensional ***actualizarValor1D*** y bidimensional ***modificarValor2D***, y remover los símbolos más recientes de un entorno específico ***removerRecientesPorEntorno.***

Además, hay métodos para obtener un símbolo por su nombre y entorno ***getSimbolo***, obtener toda la tabla de símbolos ***getTabla***, obtener el índice de un símbolo en la tabla ***getSimboloIndex***, actualizar el valor de un símbolo por su índice ***actualizarValor***, y generar un reporte HTML de la tabla de símbolos ***reporteTabla.***

ID	Tipo	Valor	Fila	Columna	ENTORNO	estructura
var1	INT	0	3	0	GLOBAL	VARIABLE
arreglo1	INT	0,0,0,0,0	5	0	GLOBAL	ARREGLO
arreglo2	INT	0,0,1,2,0,0,5,1,0,0,8,0,0	6	0	GLOBAL	ARREGLO
pr	INT	13	43	0	GLOBAL	VARIABLE
var1	INT	10	12	4	main	VARIABLE
cadenaSalida	STD::STRING	Final de la tabla de multiplicar	33	4	tablaMultiplicar	VARIABLE
temporal	INT	0	46	4	AnalizarArreglo	VARIABLE
suma	INT	0	47	0	AnalizarArreglo	VARIABLE
ceros	INT	0	48	0	AnalizarArreglo	VARIABLE
resultado	INT	12	73	4	recursividadBasica	VARIABLE