Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Monterrey

Diseño de Compiladores
Ing. Elda Quiroga

Documentación Final
Lenguaje de Programación
'Japtor'
https://github.com/jorgemoya/japtor

Jorge A. Moya A00618575

_____
Firma

Monterrey, Nuevo León a 6 de mayo del 2015

Descripción del Proyecto

Visión

Hacer un compilador en un navegador para que se pueda codificar y desplegar resultados de manera inmediata y sencilla en la ventana por medio del internet. La visión del proyecto es que esta herramienta sea utilizado como recurso para aprender programación de manera accesible. Planeo mantener la aplicación open source en Github para continuar desarrollandola o que alguien más pueda modificar el proyecto.

Objetivo

El objetivo del proyecto es aprender las distintas etapas de un compilador, como funciona, y que hay que tomar en cuenta para codificarlo. Aprender a utilizar memoria, una máquina virtual y flujo de instrucciones. Lo mejor que puedo aprender de este proyecto es poder analizar que lenguajes son los mejores para ciertas tareas.

Alcance

El lenguaje va a contar con estructuras básicas de cómo:

- Declaración de variables
- Declaración de funciones
- Asignación de variables
- Ciclos
- Condiciones
- Desplegar valores
- Recursividad
- Parametros
- Retorno
- Vectores/Matrices

El proyecto también cuenta con una interfaz gráfica web que es donde se ingresa el código, se compila, y se despliegan los resultados.

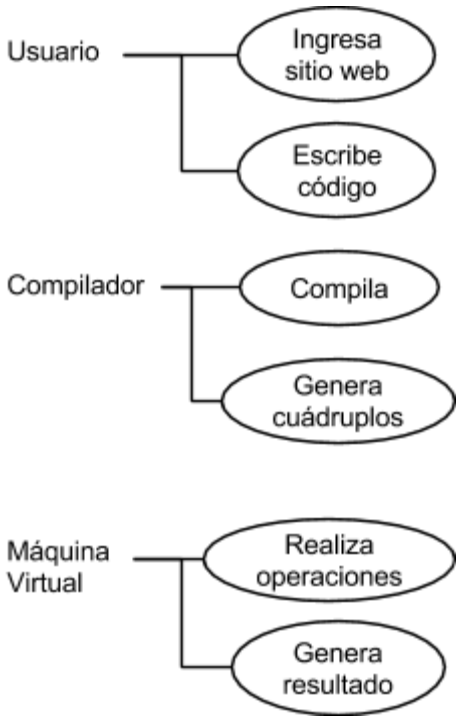
Análisis de Requerimientos

- Requerimientos Funcionales
    - Reconocer las palabras reservadas
    - Reconocer los tipos de variables
    - Identificar errores de léxico, sintaxis y semántica.
    - Traducir correctamente el program escrito por el usuario.
    - Realizar operaciones en el orden de prioridad establecido.

- La máquina virtual debe leer y ejecutar los cuádruplos regresando el valor correcto para cada expresión.
- Reconocer las distintas funciones, sus parámetros y valores de retorno.
- Manejar memoria correctamente para los distintos ‘scopes’ dentro de la ejecución.
- Manejar arreglos.

- Requerimientos No Funcionales
  - Ejecución rápida
  - Lenguaje simple
  - Manejo de arreglos eficientes

Casos de Uso generales



Descripción de los principales Test Cases

Los test cases se encuentran en dentro de la carpeta test en el proyecto, y si siguen el orden de como los fuí creando. Se puede observar como se fueron de algo sencillo, como usar asignaciones, declaraciones de variables, tanto locales como globales. Se fueron comprobando los ciclos y condiciones, que funcionaran las comparaciones tanto lógicas como normales. Después entramos en las funciones, que se pudieran declarar parámetros y regresar valores. Por último entramos en arreglos y matrices y como se asignaban y ejecutaban. Fueron construidos etapa por etapa para poder verificar el funcionamiento del programa.

Descripción del Proceso General

El proceso general para generar este proyecto fue apoyarme con la maestra para definir el concepto que quería codificar de mi compilador, codificarlo y probarlo. Subir mis cambios a GitHub.

Bitácora

La bitácora del desarrollo de mi proyecto se puede encontrar en https://github.com/jorgemoya/japtor/commits/master.

Reflexión

El proyecto me ayudó mucho para entender cómo funcionan algunos de los lenguajes que se usan y así poder entender cuál es mejor para algún uso específico. Fue interesante ver como se maneja el léxico y la sintaxis, como verificar la semántica y, especialmente, cómo se genera cuádruples para su ejecución. Manejar memoria fue un desafío interesante, pero una vez resuelto, me da una mayor reflexión del uso de memoria en los lenguajes. Fue un proyecto difícil, de muchas horas, lleno de situaciones complejas, pero al final, me entretuvo bastante. Muy buen proyecto para la carrera.

_____

Jorge Moya

Descripción del Lenguaje

Nombre del Lenguaje

Japtor

Descripción

Es un lenguaje básico, combinación de Javascript/Java, que permite declarar variables de distintos tipos (int, float, string, boolean) y cuenta con un par de estatutos, como la asignación, los ciclos, las condiciones, las expresiones, las funciones (con parámetros) y los arreglos. El lenguaje cuenta con una función clave llamada “main” de tipo void que se debe de incluir para correr el programa. Utiliza corchetes y semicolon para cerrar los bloques. El lenguaje y la máquina virtual fueron codificados en Javascript.

Descripción de los errores

- SYNTAX ERROR: error al introducir el código. La estructura está mal, falta algún punto y como o un bloque de léxico.
- NO MAIN DECLARED: falta la función ‘main’.
- INCOMPATIBLE TYPES: cuando se intenta ejecutar alguna expresión o asignación con tipos incompatibles.
- INCORRECT ARRAY DIMENSION: cuando se le intenta de llamar un arreglo con mas o menos de una dimensión.
- ARRAY POINTERS ONLY HANDLE INTS: cuando la expresión dentro de un arreglo no es igual a un ‘int’.
- INVALID IF STATEMENT: cuando la expresión dentro de un if no es booleana.
- INVALID WHILE STATEMENT: cuando la expresión dentro de un while no es booleana.
- EXPECTED RETURN: cuando una función espera un retorno y no se regresa nada.
- ILLOGICAL COMPARISON: cuando se usan operadores lógicos sin resultado de tipo booleano.
- INVALID TYPES: cuando los tipos no son compatibles.
- NEED PARAMETERS: cuando se esperan parámetros en una funcion.
- INCORRECT PARAMETERS: cuando los parámetros no son adecuados.
- OUT OF MEMORY: se acaba la memoria.
- ID NOT DECLARED: se usa una ID que no ha sido declarada.
- OUT OF BOUNDS: cuando se está fuera de los límites de un arreglo.
- NOT FOUND: no se encuentra un valor.

Descripción del Compilador

Equipo de Cómputo, lenguaje y utilerías especiales usadas en el desarrollo del proyecto

La codificación entera del proyecto se llevó a cabo en una Macbook Pro. No es necesario algún equipo fuera del individual para utilizar el proyecto.

El lenguaje y la máquina virtual fueron desarrolladas en Javascript. Para la parte del léxico/sintaxis/semántica utilicé una librería llamada Jison, muy parecida a Bison pero para Javascript. La librería te acepta un léxico y una sintaxis y genera código en Javascript. Más información en http://zaach.github.io/jison/.

Para la interfaz web se utilizaron varios frameworks. En la parte visual contamos con HTML y CSS usando el framework de Bootstrap (http://getbootstrap.com/). El editor de código integrado es un plugin de Javascript llamado Ace (http://ace.c9.io/#nav=about). Se usó también Javascript puro para hacer llamadas a las funciones necesarias y la librería de jQuery para manejar las animaciones.

Descripción del Análisis de Léxico

Patrones de Construcción de los Elementos Principales

- IDs
  - ([a-zA-Z][a-zA-Z0-9]*)(-|_)*([a-zA-Z][a-zA-Z0-9]*)*
- Ints
  - [0-9]+
- Floats
  - [0-9]*"."[0-9]+
- Strings
  - \"[^\"]*\"|\'[^\']*\
- Booleans
  - "true"|"false"
- Palábras Reservadas
  - var, int, float, string, boolean, void, write, if else, while, return, assign, program, function
- Operadores Aritméticos
  - +, -, *, /
- Operadores Lógicos
  - ||, &&, <=, >=, ==, >, <
- Operadores de Asignación

- =
- Separadores
  - [ ], { }, ( ), ;

Enumeración de Tokens

```
%%

program
        : EOF
                                {return null;}
        | program_declaration program_block
        ;

/**
        PRORGRAM
**/

program_declaration
        : PROGRAM ID ';'
                                {
                                        var proc = new Proc("global", "void", dirProc(), [], [],
null); // Creates global proc
                                        yy.procs.push(proc); // Pushes scope to procs
                                        scope.push("global"); // Makes global current scope
                                        yy.quads.push(["goto", null, null, null]); // Expects goto
main
                                        jumps.push(yy.quads.length - 1); // Expects main return
                                }
        ;

program_block
        : vars functions
        ;

/**
        VARS
**/

vars
        : VAR var_declaration vars ';' vars
        | ',' var_declaration vars
        |
        ;

var_declaration
        : type ID var_array
                                {
                                        var currentScope = scope.stackTop(); // Looks for scope
(global, main, function1..)
                                        var proc = findProc(yy, currentScope); // Finds current
process
                                        var variable = {
                                                dir: assignMemory($type, false, false, $var_array),
// Assigns memory depending on type and if is an array
                                                id: $ID,
                                                type: $type,
                                                dim: $var_array // Array with dimensions of array or
empty
                                        }
```

```
                                        proc.vars.push(variable); // Pushes to process
                                }
        ;

var_array
        : "[" I "]"
                        {
                                yy.consts.push([parseInt($I), assignMemory("int", false, true,
[])]); // Adds I to constants

                                // Pushes I to stacks
                                types.push("int");
                                ids.push(parseInt($I));

                                $$ = [$I]; // Returns an array with I for var_declaration
                        }
        | "[" I "]" "[" I "]"
                        {
                                yy.consts.push([parseInt($2), assignMemory("int", false, true,
[])]); // Adds I to constants
                                // Pushes I to stacks
                                types.push("int");
                                ids.push(parseInt($2)); // Returns an array with I for
var_declaration

                                yy.consts.push([parseInt($5), assignMemory("int", false, true,
[])]); // Adds I to constants
                                // Pushes I to stacks
                                types.push("int");
                                ids.push(parseInt($5)); // Returns an array with I for
var_declaration
                                $$ = [$2,$5];
                        }
        |
                        {
                                $$ = []; // Returns an empty array
                        }
        ;

/**
        TYPE
**/

type
        : INT
        | FLOAT
        | STRING
        | BOOLEAN
        | VOID
        ;

/**
        FUNCTION
**/

functions
        : FUNCTION funct functions
                                {
                                        // After the creation of all functions, if no main was
declared, return error.
                                        var main = findProc(yy, "main");
                                        if (main === "undefined") {
                                                throw new Error("NO MAIN DECLARED.");
                                        }
                                }
```

```
        | EOF
        ;

funct
        : function_declaration function_params function_block
                                {
                                        // Functions always generate a return unless main
                                        if (scope.stackTop().id !== "main") { // Scope is a stack
with functions
                                                yy.quads.push(["return", null, null, null]);
                                        }
                                }
        ;

function_declaration
        : type ID
                                {
                                        var dir = dirProc(); // Returns the next avaiable Proc dir
                                        var proc = new Proc($ID, $type, dir, [], [],
yy.quads.length); // Created a new Process with ID, type, and dir
                                        yy.procs.push(proc); // Pushes to procs array
                                        scope.push($ID); // Pushes to scope stack

                                        // If id == main, returns the position of the quads for the
initial goto
                                        if ($ID === "main")     {
                                                var jump = jumps.pop();
                                                yy.quads[jump][3] = yy.quads.length;
                                        }
                                }
        ;

function_params
        : '(' vars_params ')'
        ;

function_block
        :  '{' vars block '}'
                                {
                                        // Main must be the last declared function. No other function
will run after main.
                                        if (scope.pop() === "main") {
                                                return null;
                                        }
                                }
        ;

/**
        FUNCTION PARAMS
**/

vars_params
        : vars_params_declaration vars_params
        |
        ;

vars_params_declaration
        : type ID var_array
                                {
                                        var currentScope = scope.stackTop(); // Looks for scope
(global, main, function1..)
                                        var proc = findProc(yy, currentScope); // Finds current
process
                                        var variable = {
                                                dir: assignMemory($type, false, false, $var_array),
```

```
// Assigns memory depending on type and if is an array
                                                id: $ID,
                                                type: $type,
                                                dim: $var_array // Array with dimensions of array or
empty
                                        }
                                        proc.vars.push(variable); // Pushes to process vars
                                        proc.params.push(variable); // Pushes to params to know it is
a param
                                }
        | ',' type ID var_array
                                {
                                        var currentScope = scope.stackTop(); // Looks for scope
(global, main, function1..)
                                        var proc = findProc(yy, currentScope); // Finds current
process
                                        var variable = {
                                                dir: assignMemory($type, false, false, $var_array),
// Assigns memory depending on type and if is an array
                                                id: $ID,
                                                type: $type,
                                                dim: $var_array  // Array with dimensions of array or
empty
                                        }
                                        proc.vars.push(variable); // Pushes to process vars
                                        proc.params.push(variable); // Pushes to params to know it is
a param
                                }
        ;

/**
        BLOCK
**/

block
        : statutes
        ;

/**
        STATUTES
**/

statutes
        : statute statutes
        |
        ;

statute
        : assignment_statute
        | write_statute
        | if_statute
        | while_statute
        | return_statute
        | expression_statute
        ;

/**
        ASSIGNMENT
**/

assignment_statute
        : ASSIGN ID '=' expression ';'
                                {
                                        var var1 = ids.pop(); // Pops from stack expression id
                                        var var1t = types.pop(); // Pops from stack expression type
```

```
                                        var id = $ID; // ID
                                        var idt = findTypeId(yy, $ID); //Type of ID
                                        if (var1t === idt || (var1t === "int" && idt === "float")) {
// If equals types or int && float
                                                var op = yy.quads.push([$3, findDir(yy, var1), null,
findDir(yy, id)]); // Creates quad and finds the dir of each of the vars
                                        } else {
                                                throw new Error("INCOMPATIBLE TYPES");
                                        }
                                }
        | ASSIGN ID '[' expression ']' '=' expression ';'
                                {
                                        var var1 = ids.pop(); // Pops from stack second expression id
                                        var var1t = types.pop(); // Pops from stack second expression
type
                                        var var2 = ids.pop(); // Pops from stack first expression id
                                        var var2t = types.pop(); // Pops from stack first expression
type
                                        var id = $ID; // ID
                                        var idt = findTypeId(yy, $ID); // Type of ID

                                        var dims = findDim(yy, id); // Returns the dimension of ID
                                        if (dims.length != 1) { // If Not ID[]
                                                throw new Error("INCORRECT ARRAY DIMENSION")
                                        }

                                        if (var2t != "int") { // Type of the first expression must be
int
                                                throw new Error("ARRAY POINTERS ONLY HANDLE INTS");
                                        }

                                        yy.quads.push(["verify", findDir(yy, var2), 0, dims[0]-1];
// Adds verify to quads with dir of vars, and the limit from 0 to dim[0]
                                        yy.quads.push(["++", findDir(yy, id), findDir(yy, var2), "("
+ findDir(yy, createTemp(yy, idt)) + ")"]); // DirBase + S1

                                        var pointer = ids.pop(); types.pop();

                                        if (var1t === idt || (var1t === "int" && idt === "float")) {
                                                var op = yy.quads.push([$6, findDir(yy, var1), null,
findDir(yy, pointer)]); // Assign
                                        } else {
                                                throw new Error("INCOMPATIBLE TYPES");
                                        }
                                }
        | ASSIGN ID '[' expression ']' '[' expression ']' '=' expression ';'
                                {
                                        var var1 = ids.pop(); // Pops from stack third expression id
                                        var var1t = types.pop();// Pops from stack third expression
type
                                        var var2 = ids.pop(); // Pops from stack second expression id
                                        var var2t = types.pop(); // Pops from stack second expression
type
                                        var var3 = ids.pop(); // Pops from stack first expression id
                                        var var3t = types.pop(); // Pops from stack first expression
type
                                        var id = $ID; // ID
                                        var idt = findTypeId(yy, $ID); // Type of ID

                                        var dims = findDim(yy, id); // Returns the dimension of ID
                                        if (dims.length != 2) { // If Not ID[][]
                                                throw new Error("INCORRECT ARRAY DIMENSION")
                                        }

                                        if (var2t != "int") { // Type of the second expression must
```

```
be int
                                                throw new Error("ARRAY POINTERS ONLY HANDLE INTS");
                                        }

                                        yy.quads.push(["verify", findDir(yy, var3), 0, dims[0]-1]);
// Adds verify to quads with dir of vars, and the limit from 0 to dim[0]
                                        yy.quads.push(["*", findDir(yy, var3), findDir(yy,
parseInt(dims[0])), findDir(yy, createTemp(yy, var3t))]); // m1 * s1

                                        var multpointer = ids.pop();
                                        var multpointertype = types.pop();

                                        yy.quads.push(["verify", findDir(yy, var2), 0, dims[1]-1]);
// Adds verify to quads with dir of vars, and the limit from 0 to dim[1]
                                        yy.quads.push(["+", findDir(yy, multpointer), findDir(yy,
var2), findDir(yy, createTemp(yy, multpointertype))]); // (m1 * s1) + s2

                                        var sumpointer = ids.pop();
                                        var sumpointertype = types.pop();

                                        yy.quads.push(["++", findDir(yy, id), findDir(yy,
sumpointer), "(" + findDir(yy, createTemp(yy, idt)) + ")"]); // DirBase + S

                                        var pointer = ids.pop(); types.pop();

                                        if (var1t === idt || (var1t === "int" && idt === "float")) {
                                                var op = yy.quads.push([$9, findDir(yy, var1), null,
findDir(yy, pointer)]); // Assign
                                        } else {
                                                throw new Error("INCOMPATIBLE TYPES");
                                        }
                                }
        ;

assignment_declaration
        : ASSIGN ID
                                {
                                        ids.push($ID); // Pushes ID to stack
                                        types.push(findTypeId(yy, $ID)); // Pushes type to stack
                                }
        ;

/**
        WRITE
**/

write_statute
        : WRITE '(' expression ')' ';'
                                {
                                        yy.quads.push(["write", null, null, findDir(yy, ids.pop())]);
// Quad that prints the ID in dir
                                }
        ;

/**
        IF
**/

if_statute
        : IF if_condition if_block else_statute
        ;

if_condition
        : '(' expression ')'
                                {
```

```
                                        var type = types.pop(); // Pops from stack
                                        var id = ids.pop(); // Pops from stack
                                        if (type === "boolean") { // Verify expression is boolean
                                                yy.quads.push(["gotof", findDir(yy, id), null,
null]); // GotoF
                                                jumps.push(yy.quads.length - 1); // Adds position to
jump stack
                                        } else {
                                                throw new Error("INVALID IF STATEMENT");
                                        }
                                }
        ;

if_block
        : '{' block '}'
        ;

else_statute
        : else_declaration else_block
        |
                                {
                                        var jump = jumps.pop(); // Pops from stack
                                        yy.quads[jump][3] = yy.quads.length; // Adds position to jump
value
                                }
        ;

else_declaration
        : ELSE
                                {
                                        var jump = jumps.pop(); // Pops from stack
                                        yy.quads.push(["goto", null, null, null]);
                                        yy.quads[jump][3] = yy.quads.length; // Adds position to jump
value
                                        jumps.push(yy.quads.length - 1);
                                }
        ;

else_block
        : '{' block '}'
                                {
                                        var jump = jumps.pop(); // Pops from stack
                                        yy.quads[jump][3] = yy.quads.length;  // Adds position to
jump value
                                }
        ;

while_statute
        : while_declaration while_condition while_block
                                {
                                        var jump = jumps.pop(); // Pops from stack
                                        yy.quads[jump][3] = yy.quads.length; // Adds position to jump
value
                                }
        ;

while_declaration
        : WHILE
                                {
                                        jumps.push(yy.quads.length);
                                }
        ;

while_condition
        : '(' expression ')'
```

```
                                {
                                        var type = types.pop();
                                        var id = ids.pop();
                                        if(type === "boolean") {
                                                yy.quads.push(["gotof", findDir(yy,id), null, null]);
                                                jumps.push(yy.quads.length - 1);
                                        } else {
                                                throw new Error("INVALID WHILE STATEMENT");
                                        }
                                }
        ;

/**
        WHILE
**/

while_block
        : '{' block '}'
                                {
                                        var jump = jumps.pop(); // Pops from stack
                                        yy.quads.push(["goto",null,null,jumps.pop()]); //Goto quad
jump value
                                        jumps.push(jump); // Readds jump to stack
                                }
        ;

/**
        RETURN
**/

return_statute
        : RETURN expression ';'
                                {
                                        proc = findProc(yy, scope.stackTop()); // Find proc being
used (first in stack)
                                        var id = ids.pop(); // Pop from stack
                                        var type = types.pop(); // Pop from stack
                                        if (proc.type !== "void" && proc.type === type) { // If not
void and if equal types
                                                yy.quads.push(["return", null, null,
findDir(yy,id)]); // Return the result of te function to the dir of the id
                                        } else {
                                                throw new Error("EXPECTED RETURN");
                                        }
                                }
        ;

/**
        EXPRESSION
**/

expression_statute
        : expression ';'
        ;

expression
        : comparison
        | comparison logical_ops comparison
                                {
                                        var var2 = ids.pop(); // Pop comparison2 id
                                        var var2t = types.pop(); // Pop comparison2 type
                                        var var1 = ids.pop(); // Pop comparison1 id
                                        var var1t = types.pop(); // Pop comparison1 type
                                        var op = ops.pop(); // Pop op
                                        var type = validateSem(op, var1t, var2t); // Validates types
```

```
are compatible
                                        if (type !== "x") { // If compatible
                                                var op = [op, findDir(yy, var1), findDir(yy, var2),
findDir(yy, createTemp(yy, type))]; // Adds logical_ops quad
                                        } else {
                                                throw new Error("ILLOGICAL COMPARISON");
                                        }
                                        yy.quads.push(op);
                                }
        ;

logical_ops
        : '&' '&'
                                {ops.push("&&");} // Pushes to stack
        | '|' '|'
                                {ops.push("||");} // Pushes to stack
        ;

comparison
        : exp
        | exp comparison_ops exp
                                {
                                        var var2 = ids.pop(); // Pop exp2 id
                                        var var2t = types.pop(); // Pop exp2 type
                                        var var1 = ids.pop(); // Pop exp1 id
                                        var var1t = types.pop(); // Pop exp1 type
                                        var op = ops.pop(); // Pop op
                                        var type = validateSem(op, var1t, var2t); // Validates types
are compatible
                                        if (type !== "x") { // If compatible
                                                var op = [op, findDir(yy, var1), findDir(yy, var2),
findDir(yy, createTemp(yy, type))]; // Adds comparison quad
                                        } else {
                                                throw new Error("ILLOGICAL COMPARISON");
                                        }
                                        yy.quads.push(op);
                                }
        ;

comparison_ops
        : '<' '='
                                {ops.push("<=");} // Pushes to stack
        | '>' '='
                                {ops.push(">=");} // Pushes to stack
        | '!' '='
                                {ops.push("!=");} // Pushes to stack
        | '=' '='
                                {ops.push("==");} // Pushes to stack
        | '>'
                                {ops.push(">");} // Pushes to stack
        | '<'
                                {ops.push("<");} // Pushes to stack
        ;

exp
        : term exp_exit
        ;

exp_exit
        : exp_validation sum_or_minus exp
        |       exp_validation
        ;

exp_validation
        :
```

```
                                {
                                        if (ops.stackTop() === "+" || ops.stackTop() === "-") { // If
first from ops stack is + or -
                                                var var2 = ids.pop(); // Pop value2 id
                                                var var2t = types.pop(); // Pop value2 type
                                                var var1 = ids.pop(); // Pop value1 id
                                                var var1t = types.pop(); // Pop value1 type
                                                var op = ops.pop(); // Pop ops
                                                var type = validateSem(op, var1t, var2t); //
Validates types are compatible
                                                if(type !== "x") {
                                                        var op = [op, findDir(yy, var1), findDir(yy,
var2), findDir(yy, createTemp(yy, type))];
                                                } else {
                                                        throw new Error("INVALID TYPES");
                                                }
                                                yy.quads.push(op); // Adds + or - quad
                                        }
                                }
        ;

sum_or_minus
        : '+'
                                {ops.push($1);} // Push to stack
        | '-'
                                {ops.push($1);} // Push to stack
        ;

term
        : factor term_exit
        ;

term_exit
        : term_validation mult_or_divi term
        | term_validation
        ;

term_validation
        :
                                {
                                        if (ops.stackTop() === "*" || ops.stackTop() === "/") { // If
first from ops stack is * or /
                                                var var2 = ids.pop(); // Pop value2 id
                                                var var2t = types.pop(); // Pop value2 type
                                                var var1 = ids.pop(); // Pop value1 id
                                                var var1t = types.pop(); // Pop value1 type
                                                var op = ops.pop(); // Pop ops
                                                var type = validateSem(op, var1t, var2t); //
Validates types are compatible
                                                if (type !== "x") {
                                                        var op = [op, findDir(yy, var1), findDir(yy,
var2), findDir(yy, createTemp(yy, type))];
                                                } else {
                                                        throw new Error("INVALID TYPES");
                                                }
                                                yy.quads.push(op); // Adds * or / quad
                                        }
                                }
        ;

mult_or_divi
        : '*'
                        {ops.push($1);} // Push to stack
        | '/'
                        {ops.push($1);} // Push to stack
```

```
        ;

factor
        : constant
        | id options
        | "(" add_closure expression end_closure")"
        ;

id
        : ID
                                {
                                        var proc = findProc(yy, $ID);
                                        if (proc !== "undefined") {
                                                ids.push($ID);
                                                types.push(proc.type);
                                                expectingParams = true;
                                        } else {
                                                ids.push($ID);
                                                types.push(findTypeId(yy, $ID));
                                                expectingParams = false;
                                        }
                                }
        ;

options
        : params
        | array
        |
                                {
                                        if (expectingParams) {
                                                throw new Error("NEED PARAMETERS");
                                        }
                                }
        ;

params
        : "(" find_proc params_input ")"
                                {
                                        if (tempProc.type !== "void") {
                                                var temp = createTemp(yy, tempProc.type);

yy.quads.push(["gosub",tempProc.init,null,findDir(yy,temp)]);
                                        } else {
                                                yy.quads.push(["gosub",tempProc.init,null,null]);
                                        }

                                        ops.pop();
                                        tempProc = null;
                                        expectingParams = false;
                                }
        ;

find_proc
        :
                                {
                                                var id = ids.pop();
                                                tempProc = findProc(yy, id);
                                                yy.quads.push(["era",tempProc.dir,null,null]);
                                                types.pop();
                                                ops.push("|");
                                                paramTemp = 0;
                                }
        ;

params_input
```

```
        : param_expression
        |       param_expression ',' params_input
        |
        ;

param_expression
        : expression
                                {
                                        var id = ids.pop(); // Pop from stack
                                        var type = types.pop(); // Pop from stack
                                        if (paramTemp >= tempProc.numParams()) { // If the numbers of
params is higher than what is expected
                                                throw new Error("INCORRECT PARAMETERS");
                                        }

                                        if (tempProc.params[paramTemp].type === type ||
(tempProc.params[paramTemp].type === "float" && type === "int") ) {
                                                if (tempProc.params[paramTemp].dim > 0) { // if there
are many params
                                                        yy.quads.push(["param", "(" + findDir(yy, id)
+ "," + tempProc.params[paramTemp].dim + ")", null, ++paramTemp]);
                                                } else {
                                                        yy.quads.push(["param", findDir(yy, id),
null, ++paramTemp]);
                                                }
                                        } else {
                                                throw new Error("INVALID TYPES");
                                        }
                                        // ops.pop();
                                }
        ;

/**
        ARRAY
**/

array
        : vector
        | matrix
        ;

vector
        : "[" add_closure expression end_closure "]"
                                {
                                        var id = ids.pop(); // Pop exp id
                                        var type = types.pop(); // Pop exp type
                                        var id_array = ids.pop();  // Pop array id
                                        var type_array = types.pop(); // Pop array type

                                        var dims = findDim(yy, id_array); // Find dim size
                                        if (dims.length == 2 || dims.length == 0) {
                                                throw new Error("INCORRECT ARRAY DIMENSION"); //
Incorrect size
                                        }

                                        if (type != "int") { // Must be int
                                                throw new Error("ARRAY POINTERS ONLY HANDLE INTS");
                                        }

                                        yy.quads.push(["verify", findDir(yy, id), 0, dims[0]-1]); //
Pushes verify to quad from 0 to dims[0]-1
                                        yy.quads.push(["++", findDir(yy, id_array), findDir(yy, id),
"(" + findDir(yy, createTemp(yy, type_array)) + ")"]); // DirBase + s1
                                }
        ;
```

```
matrix
        : "[" add_closure expression end_closure "]" "[" add_closure expression end_closure"]"
                                {
                                        var var1 = ids.pop(); // Pop exp2 id
                                        var var1t = types.pop(); // Pop exp2 type
                                        var var2 = ids.pop(); // Pop exp1 id
                                        var var2t = types.pop(); // Pop exp1 type
                                        var id = ids.pop(); // Pop array id
                                        var idt = types.pop();; // Pop array type

                                        var dims = findDim(yy, id); // Find dim size
                                        if (dims.length != 2) {
                                                throw new Error("INCORRECT ARRAY DIMENSION") //
Incorrect size
                                        }

                                        if (var2t != "int") { // Must be int
                                                throw new Error("ARRAY POINTERS ONLY HANDLE INTS");
                                        }

                                        yy.quads.push(["verify", findDir(yy, var2), 0, dims[0]-1]);
// Pushes verify to quad from 0 to dims[0]-1
                                        yy.quads.push(["*", findDir(yy, var2), findDir(yy,
parseInt(dims[0])), findDir(yy, createTemp(yy, var2t))]); // s1 * m1

                                        var multpointer = ids.pop();
                                        var multpointertype = types.pop();

                                        yy.quads.push(["verify", findDir(yy, var1), 0, dims[1]-1]);
// Pushes verify to quad from 0 to dims[1]-1
                                        yy.quads.push(["+", findDir(yy, multpointer), findDir(yy,
var1), findDir(yy, createTemp(yy, multpointertype))]); // (s1 * m1) + s2

                                        var sumpointer = ids.pop();
                                        var sumpointertype = types.pop();

                                        yy.quads.push(["++", findDir(yy, id), findDir(yy,
sumpointer), "(" + findDir(yy, createTemp(yy, idt)) + ")"]); // DirBase + s
                                }
        ;

add_closure
        :
                                {ops.push("|");} // Adds fondo
        ;

end_closure
        :
                                {ops.pop();} // Removes fondo
        ;

constant
        : I
                                {
                                        // Add INT to constant
                                        yy.consts.push([parseInt($I), assignMemory("int", false,
true, [])]);
                                        // Pushes to stack
                                        types.push("int");
                                        ids.push(parseInt($I));
                                }
        | F
                                {
                                        // Add FLOAT to constant
```

```
                                        yy.consts.push([parseFloat($F), assignMemory("float", false,
true, [])]);
                                        types.push("float");
                                        ids.push(parseFloat($F));
                                }
        | B
                                {
                                        // Add BOOLEAN to constant
                                        yy.consts.push([$B, assignMemory("boolean", false, true,
[])]);
                                        // Pushes to stack
                                        types.push("boolean");
                                        ids.push($B);
                                }
        | S
                                {
                                        // Add STRING to constant
                                        yy.consts.push([$S, assignMemory("string", false, true,
[])]);
                                        // Pushes to stack
                                        types.push("string");
                                        ids.push($S);
                                }
        ;

%%
```

Descripción del Análisis Sintáctico

```
<program> -> <program_declaration program_block>
<program> -> EOF
<program_declaration> -> PROGRAM ID ;
<program_block> -> vars functions
<vars> -> VAR var_declarations vars ; vars
<vars> -> , var_declaration vars
<vars> -> E
<var_declaration> -> <type> ID <var_array>
<var_array> -> [ <I> ]
<var_array> -> [ I ] [ I ]
<var_array> -> E
<type> -> INT
<type> -> FLOAT
<type> -> STRING
<type> -> BOOLEAN
<type> -> VOID
<functions> -> FUNCTION <funct> <functions>
<functions> -> EOF
```

<funct> -> <function_declaration> <function_params> <function_block>
<function_declaration> -> <type> ID
<function_params> -> (<vars_params>)
<function_block> -> { <vars> <block> }
<vars_params> -> <vars_params_declaration> <vars_params>
<vars_params> -> E
<vars_params_declaration> -> <type> ID <var_array>
<vars_params_declaration> -> , <type> ID <var_array>
<block> -> <statutes>
<statutes> -> <statute> <statutes>
<statutes> -> E
<statute> -> <assignment_statute>
<statute> -> <write_statute>
<statute> -> <if_statute>
<statute> -> <while_statute>
<statute> -> <return_statute>
<statute> -> <expression_statute>
<assignment_statute> -> ASSIGN ID = <expression> ;
<assignment_statute> -> ASSIGN ID [<expression>] = <expression> ;
<assignment_statute> -> [<expression>] [<expression>] =
<expression> ;
<assignment_declaration> -> ASSIGN ID
<write_statute> -> (<expression>) ;
<if_statute> -> IF <if_condition> <if_block> <else_statute>
<if_condition> -> (<expression>)
<if_block> -> { <block> }
<else_statute> -> <else_declaration> <else_block>
<else_statute> -> E
<else_declaration> -> ELSE
<else_block> -> { <block> }
<while_statute> -> <while_declaration> <while_condition> <while_block>
<while_declaration> -> WHILE
<while_condition> -> (<expression>)
<while_block> -> { <block> }
<return_statute> -> RETURN <expression> ;
<expression_statute> -> <expression> ;
<expression> -> <comparison>
<expression> -> <comparison> <logical_ops> <comparison>
<logical_ops> -> &&

```
<logical_ops> -> ||
<comparison> -> <exp>
<comparison> -> <exp> <comparison_ops> <exp>
<comparison_ops> -> <=
<comparison_ops> -> >=
<comparison_ops> -> !=
<comparison_ops> -> ==
<comparison_ops> -> >
<comparison_ops> -> <
<exp> -> <term> <exp_exit>
<exp_exit> -> <exp_validation> <sum_or_minus> <exp>
<exp_exit> -> <exp_validation>
<exp_validation> -> E
<sum_or_minus> -> +
<sum_or_minus> -> -
<term> -> <factor> <term_exit>
<term_exit> -> <term_validation> <mult_or_divi> <term>
<term_exit> -> <term_validation>
<term_validation> -> E
<mult_or_divi> -> *
<mult_or_divi> -> /
<factor> -> <constant>
<factor> -> <id> <options>
<factor> -> ( <add_closure> <expression> <end_closure> )
<id> -> ID
<options> -> <params>
<options> -> <array>
<options> -> E
<params> -> ( <find_proc> <params_input> )
<find_proc> -> E
<params_input> -> <param_expression>
<params_input> -> <param_expression> , <params_input>
<params_input> -> E
<param_expression> -> <expression>
<array> -> <vector>
<array> -> <matrix>
<vector> -> [ <add_closure> <expression> <end_closure> ]
<matrix> -> [ <add_cloure> <expression> <end_cloure> ] [ <add_closure>
<expression> <end_closure> ]
```

```
<add_closure> -> E
<end_closure> -> E
<constant> -> I
<constant> -> F
<constant> -> B
<constant> -> S
```

Dirección virtuales asociadas

Directorios: 2000 a 4999
Global Ints: 5000 a 6999
Global Floats: 7000 a 8999
Global Strings: 9000 a 10999
Global Booleans: 11000 a 11999
Local Ints: 12000 a 13999
Local Floats: 14000 a 15999
Local Strings: 16000 a 17999
Local Booleans: 18000 a 18999
Temporal Ints: 19000 a 20999
Temporal Floats: 21000 a 22999
Temporal Strings: 23000 a 24999
Temporal Booleans: 25000 a 25999
Constant Ints: 26000 a 27999
Constant Floats: 28000 a 29999
Constant Strings: 30000 a 31999
Constant Boolean: 32000 a 32999

Descripción de Acciones Semánticas y Código

La manera en la que se corre el codigo es empieza por %start, luego va recorriendo LEFT to RIGHT los tokens. Una vez que se lee un token, y antes de salir, se corre código. En Jison no existen los mid grammar functions, se tiene que dividir la sintaxis para poder correr código antes de ejecutar algún otro token. En el código que se ejecuta antes de salir de un token es en donde se hacen las validaciones y en algunos casos se generarán cuadruplos apoyadas de otras funciones fuera del código. En esta generación de semántica es donde se crean los Procs, se guardan las variables, se les asignan direcciones.

Tabla de Consideraciones Semánticas

		+	-	/	*	==	<	<=	>	>=	&&	||	!=
i	i	i	i	i	i	b	b	b	b	b			b
f	f	f	f	f	f	b	b	b	b	b			b
s	s	s											
b	b					b					b	b	b
i	f	f	f	f	f	b	b	b	b	b			b
i	s												
i	b												
f	i	f	f	f	f	b	b	b	b	b			b
f	s												
f	b												
s	i												
s	f												
s	b												
b	i												
b	f												
b	s												

Descripción Detallada del Proceso de Administración de Memoria en Compilación

Se creó una clase llamada Procs que se inicializa cuando se crea una nueva función. Estos procs estan en un stack, y cuando se asignan variables locales, se guardan dentro del arreglo correspondiente del primer proc. Los procs cuentan con un tipo y dirección. Para guardar direcciones se tiene un contador y simplemente se van sumando cuando se encuentra un tipo. También se llena un arreglo llamado consts con el valor de una constante y una dirección.

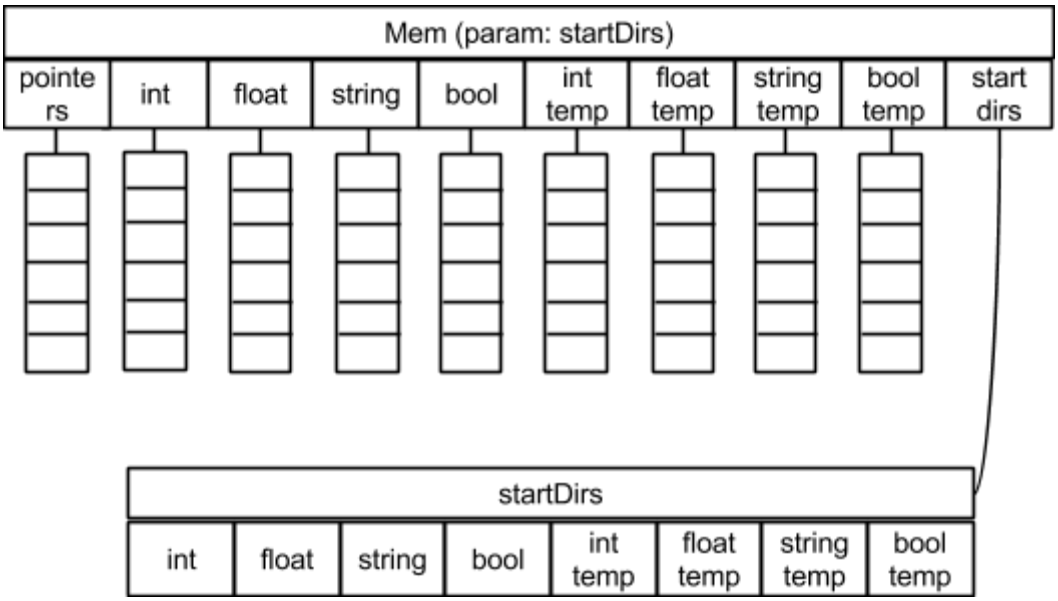
Descripción de la Máquina Virtual

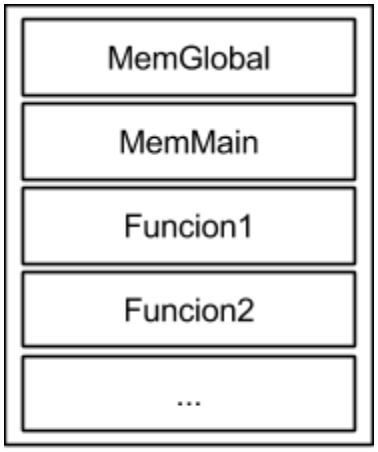
Equipo de Cómputo, lenguaje y utilerías especiales usadas en el desarrollo del proyecto

La codificación entera del proyecto se llevó a cabo en una Macbook Pro. No es necesario algún equipo fuera del individual para utilizar el proyecto. La máquina virtual fue desarrollada 100% en Javascript y no se usó ninguna librería adicional.

Descripción detallada del proceso de Administración de Memoria usado en la compilación

Se tiene una 'clase' llamada Mem que se instancia al principio de la ejecución para crear la memoria global, y una segunda vez para crear la memoria del 'main'. A cada tipo se le cuelga un array vacío. Cada vez que se llame una función se instancia otra memoria. Se le pasa como parámetros a la instanciación de una memoria, todas las direcciones de inicio de la función. Cuando una función termina, se termina la memoria. Toda las memorias se meten a un arreglo llamado Mems.

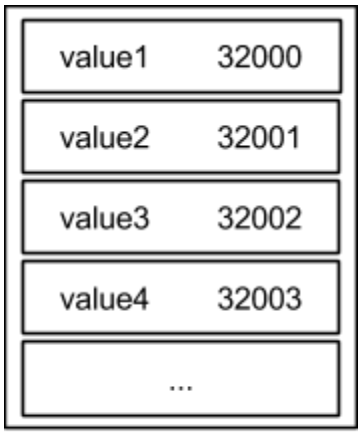




Mems Array

Cada vez que se quiere insertar un valor de una variable, se busca si es 'local' o 'global', si es local se inserta en `mems[mems.length - 1]`, si es global se inserta en `mems[0]`. Para saber en que fila del array se tiene que insertar, se usa startDirs para calcular la posición con la fórmula `mems[mems.length-1].int[dir - mems[mems.length-1].startDirs[0]] = value;` Dependiendo del tipo de la variable a la que se le quiere insertar un valor, cambia el `.int` y la apuntador de `startDirs`. Para buscar el valor se utiliza el mismo algoritmo.

También cuenta con un arreglo llamado consts que se genera desde la compilación que es el que se usa para guardar el valor y la memoria de todo valor constante. Si un valor es de tipo constante, se busca en esta tabla. `return consts[i][0];`



Consts Array

Pruebas del Funcionamiento del Lenguaje

Incluir pruebas que comprueben el funcionamiento del proyecto.

Factorial

```
program factorial;
function int factorial(int x) {
	if(x==0) {
		return 1;
	}
	return x * factorial(x-1);
}
function void main() {
	write(factorial(7));
}
```

Resultado: 5040

Fibonnaci

```
program fibonacci;
function int fibonacci(int n) {
  if(n < 2) {
    return n;
  } else {
    return fibonacci(n-2) + fibonacci(n-1);
  }
}
function void main() {
  write(fibonacci(7));
}
```

Resultado: 13

Search

```
program search;
function int search(int number, int array[10]) {
    var int x;
    assign x = 0;
    while (x < 10) {
        if (array[x] == number) {
            return x;
        }
        assign x = x + 1;
    }
```

```
}
function void main() {
    var int number, int x, int array[10];

    assign number = 30;
    assign x = 0;

    while (x < 10) {
        assign array[x] = x * 5;
        assign x = x + 1;
    }

    write(search(number, array));
}
```

Resultado: 6

Sort

```
program sort;
function void sort(int array[10]) {
    var int x, int j, boolean flag, int temp;
    assign flag = true;
    assign j = 0;

    while (flag == true) {
        assign flag = false;
        while (j < 10 - 1) {
            if (array[j] > array[j+1]) {
                assign temp = array[j];
                assign array[j] = array[j+1];
                assign array[j+1] = temp;
                assign flag = true;
            }
            assign j = j + 1;
        }
        assign j = 0;
    }

    assign x = 0;
    while (x < 10) {
        write(array[x]);
        assign x = x + 1;
    }
}
function void main() {
    var int array[10];
```

```
    assign array[0] = 43;
    assign array[1] = 32;
    assign array[2] = 4;
    assign array[3] = 60;
   assign array[4] = 10;
    assign array[5] = 23;
    assign array[6] = 24;
    assign array[7] = 2;
    assign array[8] = 13;
    assign array[9] = 51;

    sort(array);
}
```

Result: 2 4 10 13 23 24 32 43 51 60

Multiplicar Matrices

```
program multiply;
function void main() {
    var int m1[4][3], int m2[3][3], int c, int d, int k, int m, int n, int p, int q,
int mf[4][3], int sum;

    assign m1[0][0] = 1;
    assign m1[0][1] = 2;
    assign m1[0][2] = 3;
    assign m1[1][0] = 4;
    assign m1[1][1] = 5;
    assign m1[1][2] = 6;
    assign m1[2][0] = 7;
    assign m1[2][1] = 8;
    assign m1[2][2] = 9;
    assign m1[3][0] = 10;
    assign m1[3][1] = 11;
    assign m1[3][2] = 12;

    assign m2[0][0] = 1;
    assign m2[0][1] = 2;
    assign m2[0][2] = 3;
    assign m2[1][0] = 4;
    assign m2[1][1] = 5;
    assign m2[1][2] = 6;
    assign m2[2][0] = 7;
    assign m2[2][1] = 8;
    assign m2[2][2] = 9;

    assign m = 4;
```

```
    assign n = 3;
    assign p = 3;
    assign q = 3;

    assign c = 0;
    assign d = 0;
    assign k = 0;
    assign sum = 0;

    while(c < m) {
        assign d = 0;
        while(d < q) {
            assign k = 0;
            while(k < p) {
                assign sum = sum + m1[c][k] * m2[k][d];
                assign k = k + 1;
            }
            assign mf[c][d] = sum;
            assign sum = 0;
            assign d = d + 1;
        }
        assign c = c + 1;
    }

    assign c = 0;
    while (c < m) {
        assign d = 0;
        while(d < q) {
            write(mf[c][d]);
            assign d = d + 1;
        }
        assign c = c + 1;
    }
}
```

Resultado: 30 36 42 66 81 96 102 126 150 138 171 204