Campus Monterrey

Marcelo Hernandez - A01194283

Zososcript - Compilador

Desarrollo de aplicaciones avanzadas de ciencias computacionales (Gpo 505)

Monterrey, N.L. a 24 de mayo de 2024

Indice

Herramientas

CoCo/r:

CoCo/r (Compiler-Compiler for Recursive Descent parsers) es una herramienta utilizada para generar analizadores sintácticos recursivos descendentes. Específicamente, CoCo/r toma una gramática libre de contexto escrita en una forma específica y produce el código fuente del analizador en lenguajes como C# o Java.

El propósito principal de CoCo/r es facilitar la creación de compiladores e intérpretes para nuevos lenguajes de programación o para lenguajes de dominio específico. Al automatizar la generación del analizador sintáctico, los desarrolladores pueden centrarse en la lógica del análisis semántico y la generación de código en lugar de en los detalles de la implementación del analizador sintáctico.

Características principales de CoCo/r:

- A partir de una gramática formal, genera el código del analizador.
- Parsers Recursivos Descendentes.

En resumen, CoCo/r es una herramienta poderosa para construir analizadores sintácticos de manera eficiente y precisa, siendo especialmente útil en el contexto de desarrollo de compiladores e intérpretes.

C++:

Escogí C++ debido a que este lenguaje también es compilado y me interesaba que mi compilador cree un archivo ejecutable del código que se le entregó.

GitHub:

Escogí git para el control de versiones.
Repo: [marcelo8hdz/zososcript]

Gramatica

Expresiones Regulares:

Caracteres

```
C/C++
CHARACTERS
    letter = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz".
    digit = "0123456789".
    quote = '"'.
    cr  = '\r'.
    lf  = '\n'.
    tab = '\t'.
    space = ' '.
    anyButQuote = ANY - '"'.
```

caracter	**explicacion**
letter	Cualquier letra del alfabeto, minúscula o mayúscula.
digit	Cualquier dígito
quote	Comillas, para definir strings
cr (carriage return)	Final de la línea
lf	Final de la linea
tab	tabulacion
space	espacio
anyButQuote	Cualquiera menos comillas (“”)

Tokens

```
C/C++
TOKENS
    ident = letter { letter | digit }.
    string = quote { anyButQuote } quote.
    number = ['-'] digit {digit}.
    float = ['-'] digit {digit} '.' digit {digit}.
```

Automata	**explicacion**
ident	Comienza con una letra y después puede ser seguido por cualquier dígito o letra.
string	Comilla seguido por cualquier cosa menos comilla, y termina cuando se topa otra comilla.
number	Simbolo de negativo (opcional) seguido por uno o más dígitos.
float	Símbolo de negativo (opcional) seguido por uno o más dígitos, subsecuentemente seguido por un punto y uno o más dígitos.

Gramatica:

```
C/C++
PRODUCTIONS
    Ident<wchar_t* &name> = ident.

    AddOp<int& op> = '+' | '-' .

    MulOp<int& op> = '*' | '/'.

    RelOp<int& op> = "==" | '<' | '>' | "!=" .

    Type<int &type> = "int" | "boolean" | "float" | "void".

    VariableDeclaration = Type<type> Ident<name> {',' Ident<name>} ';'.

    FunctionDeclaration =
        "function" Type<type> Ident<name> '(' ')'
        '{'{ Statement }'}'
    .
```

```
VariableAssignation =
    Ident<name> '=' LogicalExpresion<newtype> ';'
.

IfCase =
    "if" '(' LogicalExpresion<type> ')' '{'
         { Statement }
    '}'
    [ "else" '{'
        { Statement }
    '}']
.

WhileLoop =
    "while" '(' LogicalExpresion<type> ')'
    '{' { Statement } '}'
.

Print =
    "print" '('
         ( string | LogicalExpresion<type> )
         {','( string  | LogicalExpresion<type> )}
    ')'';'
.

Statement =
    VariableAssignation
    | IfCase
    | WhileLoop
    | Print
.

LogicalExpresion<int& type> =
    SimExpr<type> { RelOp<op> SimExpr<nextType> }
.

SimExpr<int& type> =
    Term<type> { AddOp<op> Term<nextType> }
.

Term<int& type> =
    Factor<type> { MulOp<op> Factor<nextType>}
.
```

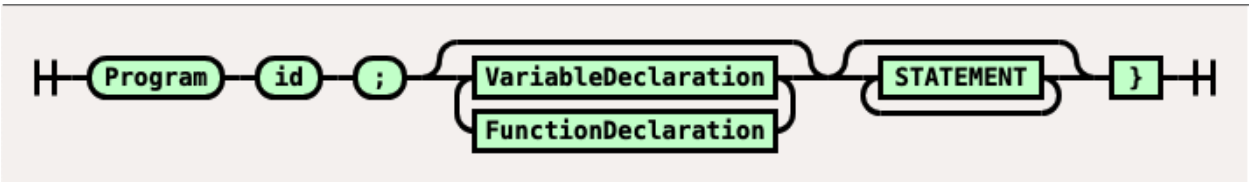
```
Factor<int& type> =
    float
    | number
    | "false"
    | "true"
    | Ident<name>
    |  '(' SimExpr<type> ')'
    | '-' Factor<type>
.

Zoso (. wchar_t* name; InitDeclarations(); .) =
    "Program" Ident<name> ';'
    { FunctionDeclaration | VariableDeclaration }
    { Statement }
.
```

Automata	**explicacion**
Ident	Un id de una variable.
AddOp	Un operador de suma o resta.
MulOp	Un operador de multiplicación o división.
RelOp	Un operador logico.
Type	Una palabra reservada de tipo, int, float, boolean, void.
VariableDeclaration	Type seguido por uno o más Idents, termina con un punto y coma.
FunctionDeclaration	Palabra reservada ‘function’ seguida por Type seguida por un Ident seguida por un paréntesis abierto, después uno cerrado, después un corchete abierto seguido por 0 o más Statements.
VariableAssignation	Ident seguido de un signo de igual y después una expresión Logica.
IfCase	Palabra reservada ‘if’, después un paréntesis abierto, después una expresión lógica y un paréntesis que cierra.
WhileLoop	Palabra reservada ‘while’, después un paréntesis abierto, después una expresión lógica y un paréntesis que cierra. Después un corchete que abre seguido por 0 o más Statements y un corchete que cierra.

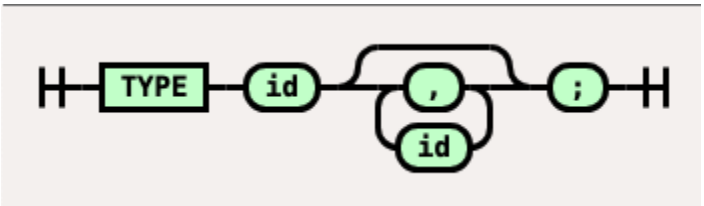
Print	Palabra reservada ‘print’ seguida por un paréntesis que abre, después un string o una expresión lógica, seguida por 0 o más comas y string o expresión lógica.
Statement	Un VariableAssignation o IfCase o WhileLoop o Print.
LogicalExpr	Una expresión simple que puede o no ser seguida por un operador lógico y otra expresión simple.
SimExpr	Un término que puede o no ser seguido por un AddOp y otro término.
Term	Un factor que puede o no ser seguido por un MulOp y otro factor.
Factor	Una constante (int, float o bool) o un id, o un paréntesis seguido por otro factor y un paréntesis que cierra o un signo de negativo y otro factor.
Zoso	Palabra reservada ‘Program’ seguido por el nombre del programa (un Ident) seguido por un punto y coma. Después de esto siguen 0 o más declaraciones de variables o funciones y subsecuentemente 0 o más Statements

Autómatas Finitos Definidos:

Zoso:

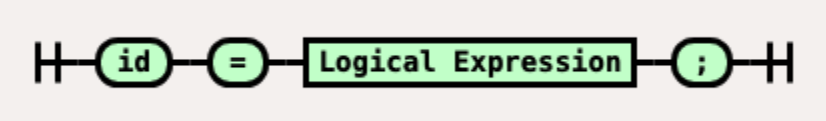


VariableDeclaration:

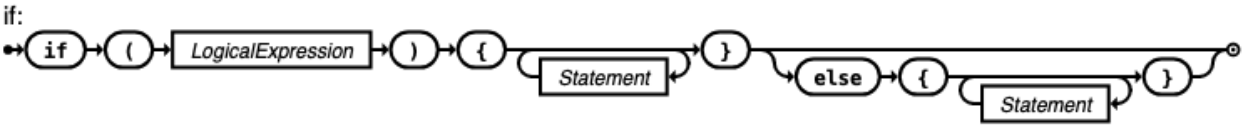


FunctionDeclaration:

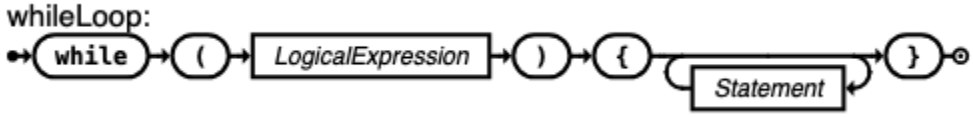
VariableAssignation:



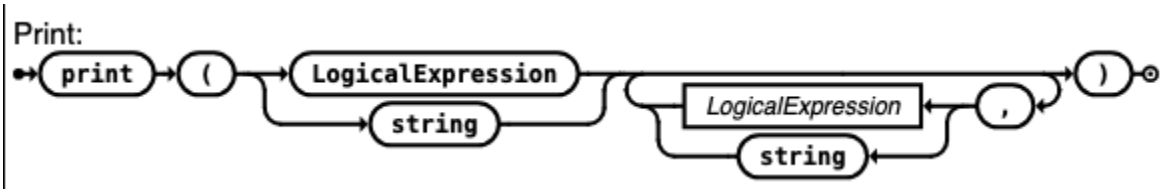
IfCase:



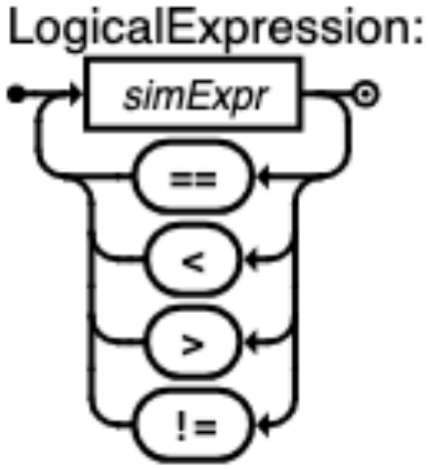
WhileLoop:



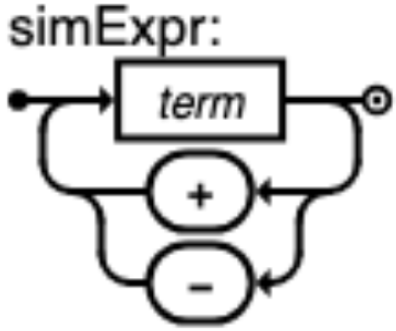
Print:



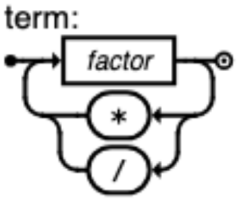
Logical Expression:



Simple Expression:



Term:



Factor:

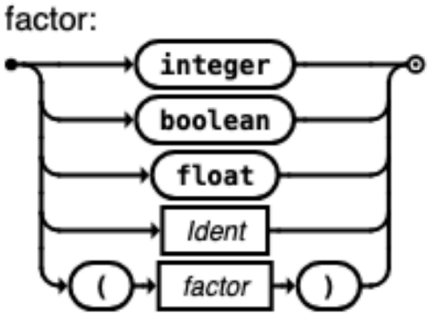


Tabla de Variables

Codigo:

```C/C++
#if !defined(SYMBOLTABLE_H__)
#define SYMBOLTABLE_H__

#include "Scanner.h"
#include <string>
#include <map>

namespace Zoso {

class Parser;
class Errors;

struct Obj {  // variable in symbol table
	wchar_t* name;			// name of the object
	int address;		// address in memory or start of function
	int type;			// type of the object
	int kind;			// var, function
};

class SymbolTable {
public:
	std::map<std::wstring, Obj> variables;

	const int undef, integer, boolean, decimal;
	const int var, function;

	int integerAddress, booleanAddress, floatAddress, voidAddress;

	SymbolTable(Parser *parser);

	Obj NewObj (wchar_t* name, int kind, int type);

	Obj Find (wchar_t* name);
};

}; // namespace

#endif // !defined(SYMBOLTABLE_H__)
```

Estructura de Datos:

Para la tabla de variables escogí utilizar un hashmap con la intención de hacer la búsqueda lo más eficiente posible. En esta guardo como llave el nombre de la variable y como valor el objeto de la variable.

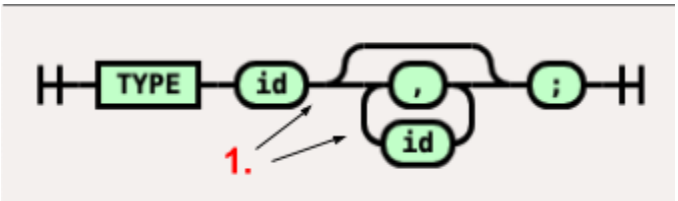
Tiene un contador para los lugares en memoria de cada tipo de variable, y unas constantes para asignar a los tipos y clases a cada nuevo objeto.

Usa el nombre de la variable para buscar el valor del objeto en el hashmap, con un valor de O(1) constante.

Para el objeto de la variable decidí guardar el nombre, su lugar en memoria, su tipo y su clase (variable o función). Su clase fue guardada debido a que al principio de la planeación del código pensé que agregaría la opción de utilizar funciones.

Puntos Neuralgicos:

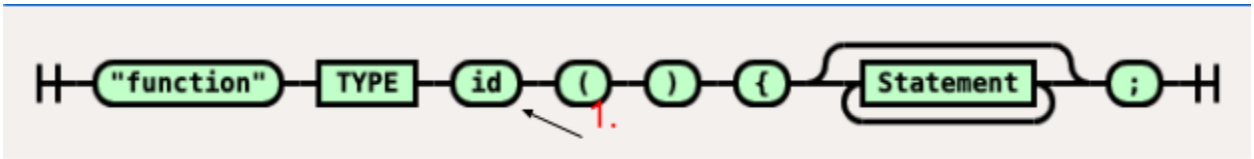
VariableDeclaration:



Punto Neuralgico 1:

- Tirar un error si el Id ya esta declarado.
- Crear nuevo objeto para symbol table con su tipo, id y de clase variable.

FunctionDeclaration:



Punto Neuralgico 1:

- Tirar un error si el Id ya esta declarado.
- Crear nuevo objeto para symbol table con su tipo, id y de clase funcion.

Cubo Semantico

Codigo:

```C/C++
#include <unordered_map>
#include <string>
#include <iostream>

using TypeMap = std::unordered_map<int, int>;
using OperationMap = std::unordered_map<int, TypeMap>;
std::unordered_map<int, OperationMap> SemanticCube;


void initializeSemanticCube() {
    int undef, integer, boolean, decimal, error; // types
    undef = 0, integer = 1, boolean = 2, decimal = 3, error = 404;

    int ADD, SUB, MUL, DIV, EQU, LSS, GTR, ASSIGN, NEQU;
    ADD = 0, SUB = 1, MUL = 2, DIV = 3, EQU = 4, LSS = 5, GTR = 6, ASSIGN = 7,
NEQU = 18;


    // Populate SemanticCube for "int"
    SemanticCube[integer][ASSIGN] = {{integer, integer}, {decimal, error},
{boolean, error}, {undef, error}};
    SemanticCube[integer][EQU] = {{integer, boolean}, {decimal, error},
{boolean, error}, {undef, error}};
    SemanticCube[integer][GTR] = {{integer, boolean}, {decimal, boolean},
{boolean, error}, {undef, error}};
    SemanticCube[integer][LSS] = {{integer, boolean}, {decimal, boolean},
{boolean, error}, {undef, error}};
    SemanticCube[integer][NEQU] = {{integer, boolean}, {decimal, boolean},
{boolean, error}, {undef, error}};
    SemanticCube[integer][MUL] = {{integer, integer}, {decimal, decimal},
{boolean, error}, {undef, error}};
    SemanticCube[integer][DIV] = {{integer, integer}, {decimal, decimal},
{boolean, error}, {undef, error}};
    SemanticCube[integer][ADD] = {{integer, integer}, {decimal, decimal},
{boolean, error}, {undef, error}};
    SemanticCube[integer][SUB] = {{integer, integer}, {decimal, decimal},
{boolean, error}, {undef, error}};

    // Populate SemanticCube for "float"
```

```
    SemanticCube[decimal][ASSIGN] = {{integer, decimal}, {decimal, decimal},
{boolean, error}, {undef, error}};
    SemanticCube[decimal][EQU] = {{integer, error}, {decimal, boolean},
{boolean, error}, {undef, error}};
    SemanticCube[decimal][GTR] = {{integer, boolean}, {decimal, boolean},
{boolean, error}, {undef, error}};
    SemanticCube[decimal][LSS] = {{integer, boolean}, {decimal, boolean},
{boolean, error}, {undef, error}};
    SemanticCube[decimal][NEQU] = {{integer, boolean}, {decimal, boolean},
{boolean, error}, {undef, error}};
    SemanticCube[decimal][MUL] = {{integer, decimal}, {decimal, decimal},
{boolean, error}, {undef, error}};
    SemanticCube[decimal][DIV] = {{integer, decimal}, {decimal, decimal},
{boolean, error}, {undef, error}};
    SemanticCube[decimal][ADD] = {{integer, decimal}, {decimal, decimal},
{boolean, error}, {undef, error}};
    SemanticCube[decimal][SUB] = {{integer, decimal}, {decimal, decimal},
{boolean, error}, {undef, error}};

    // Populate SemanticCube for "boolean"
    SemanticCube[boolean][ASSIGN] = {{integer, error}, {decimal, error},
{boolean, boolean}, {undef, error}};
    SemanticCube[boolean][EQU] = {{integer, error}, {decimal, error}, {boolean,
boolean}, {undef, error}};
    SemanticCube[boolean][GTR] = {{integer, boolean}, {decimal, error},
{boolean, boolean}, {undef, error}};
    SemanticCube[boolean][LSS] = {{integer, error}, {decimal, error}, {boolean,
boolean}, {undef, error}};
    SemanticCube[boolean][NEQU] = {{integer, error}, {decimal, error},
{boolean, error}, {undef, error}};
    SemanticCube[boolean][MUL] = {{integer, error}, {decimal, error}, {boolean,
error}, {undef, error}};
    SemanticCube[boolean][DIV] = {{integer, error}, {decimal, error}, {boolean,
error}, {undef, error}};
    SemanticCube[boolean][ADD] = {{integer, error}, {decimal, error}, {boolean,
error}, {undef, error}};
    SemanticCube[boolean][SUB] = {{integer, error}, {decimal, error}, {boolean,
error}, {undef, error}};
}
```

Explicacion:

Es una estructura de datos para la etapa de análisis semántico de un compilador. Permite determinar cómo interactúan diferentes tipos de datos cuando se utilizan en operaciones como suma, resta, multiplicación, etc.

Primero, se definen los tipos de datos que nuestro lenguaje de programación puede manejar: "undef" (indefinido), "integer" (entero), "boolean" (booleano) y "decimal" (decimal).

Luego, se declaran las operaciones que se pueden realizar entre estos tipos de datos: suma, resta, multiplicación, división, comparaciones, etc.

Para hacer esto eficiente, utilizamos mapas desordenados anidados en C++. Para cuando sea utilizado en búsqueda su complejidad sea constante O(1)

Cada entrada del Cubo Semántico especifica el tipo resultante cuando se realiza una operación entre dos tipos dados.

En resumen, el Cubo Semántico es como una guía que nos dice qué esperar cuando operamos con diferentes tipos de datos en nuestro código, lo que es fundamental para garantizar la coherencia y la corrección en el análisis de programas.

Es un mapa desordenado para buscar el tipo en complejidad constante O(1). El Cubo Semántico proporciona una manera eficiente de determinar la validez de operaciones entre tipos de datos y el tipo resultante de esas operaciones en un compilador o durante el análisis semántico de un programa.

Generador de Codigo

Codigo:

```C/C++
#if !defined(CODEGENERATOR_H__)
#define CODEGENERATOR_H__

#include "Scanner.h"
#include "Avail.h"
#include <stdlib.h>
#include <stdio.h>
#include <wchar.h>
#include <vector>
#include <stack>
#include <map>
#include <iostream>
#include "boost/variant.hpp"

namespace Zoso {

class CodeGenerator {
    public:
        // opcodes
        int ADD, SUB, MUL, DIV, EQU, LSS, GTR, ASSIGN,
            LOAD, CONST, FCALL, RETURN, GOTO, GOTOF, STORE, READ, WRITE, PRINT,
             NEQU;

        int ERROR;
        int undef, integer, boolean, decimal; // types

        wchar_t* opcode[21];

        Avail* avail;
        std::stack<int> operandStack;
        std::stack<int> operatorStack;
        std::stack<int> typeStack;
        std::stack<int> jumpStack;

        std::vector<std::vector<int> > code;
        std::map<int, boost::variant<int, float, bool, std::wstring> >
constantMap;
```

```
        CodeGenerator();
        ~CodeGenerator();

        void getAddOpResultType(int& resultType);
        void getRelOpResultType(int& resultType);
        void getMulOpResultType(int& resultType);
        void getAssignResultType(int& resultType);

        void printConstantMapToFile(const std::string& filename);
        void printCodeVectorToFile(const std::string& filename);


        void printQuads();
        void printConstantMap();
};

}; // namespace

#endif // !defined(CODEGENERATOR_H__)
```

Explicacion:

Descripción General
El generador de código es encargado de transformar el árbol de sintaxis abstracta (AST) o las instrucciones intermedias en código de máquina o código intermedio que puede ser ejecutado directamente por la máquina o una máquina virtual.

Funcionalidad Principal
El generador de código toma las instrucciones intermedias y las convierte en una representación ejecutable en el vector de código. Este proceso incluye la administración de constantes y la generación de los archivos para correr la máquina virtual.

Estructura de Datos

- **code (vector de instrucciones):**
  - **Descripción:** Este vector almacena las instrucciones intermedias generadas durante el análisis de la entrada. Cada instrucción contiene información sobre la operación y los operandos involucrados.
- **constantMap (mapa de constantes):**

Descripción: Este mapa almacena las constantes usadas en el código, indexadas por una clave entera. Las constantes pueden ser de tipo entero, flotante, booleano o cadena de caracteres ancha (wstring).

operandStack (pila de operandos):

Descripción: Esta pila se utiliza para manejar los operandos durante la generación de código. Los operandos se empujan y se sacan de la pila a medida que se generan las instrucciones.

operatorStack (pila de operadores):

Descripción: Esta pila se utiliza para manejar los operadores durante la generación de código. Los operadores se empujan y se sacan de la pila a medida que se generan las instrucciones.

jumpStack (pila de operadores):

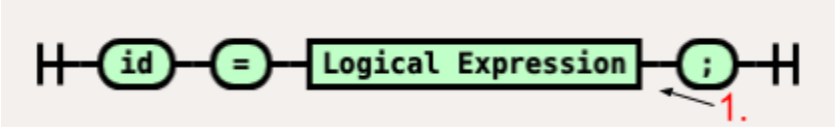
Descripción: Esta pila se utiliza para manejar los brincos de los GOTOss durante la generación de código. Los brincos se empujan y se sacan de la pila a medida que se generan las instrucciones.

Avail

Descripción: Gestiona la memoria temporal disponible para el almacenamiento de resultados intermedios.

Puntos Neuralgicos:

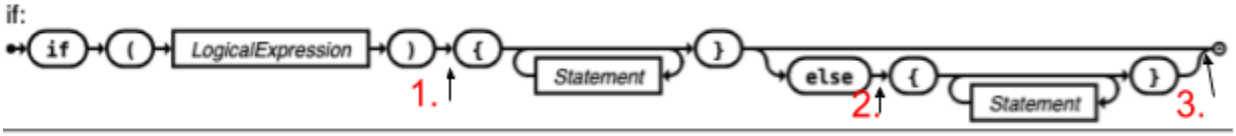
VariableAssignation:



Punto Neuralgico 1:

- Tirar un error si el Id no esta declarado.
- Consigue los tipos haciendo 2 pops al stack de tipos.
- Tirar un error si el tipo del resultado de la expresion no es asignable al tipo de la variable.
- Crear un cuádruplo de ASSIGN con el resultado de la expresion. Usando un pop del stack de operandos para conseguir los operandos derecho e izquierdo.

IfCase:



Punto Neuralgico 1:

- Pop del stack de tipos.
- Pop del stack de operandos.
- Crear un cuádruplo de GOTOF con el resultado de la expresión.
- Pushear al stack de brincos el número de instrucción en el que vas.

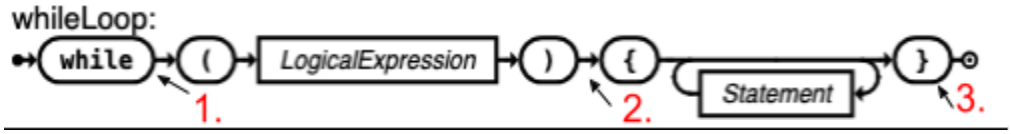
Punto Neuralgico 2:

- Crear un Quad de GOTO.
- Pop del stack de brincos.
- Empujar al stack de brincos el número de instrucción en el que estás.
- Llenar la instrucción del Pop de stack de brincos con el numero de la siguiente instrucción.

Punto Neuralgico 3:.

- Pop del stack de brincos.
- Llenar la instrucción del Pop de stack de brincos con el numero de la siguiente instrucción.

WhileLoop:



Punto Neuralgico 1:

- Push al stack de broncos la siguiente instrucción.

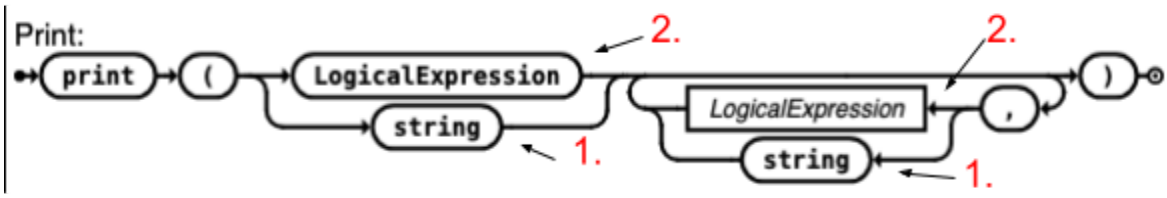
Punto Neuralgico 2:

- Pop al stack de tipos.
- Pop al stack de operandos.
- Crear un Quad de GOTOF con el resultado del pop al stack de operandos.
- Push al stack de brincos con el numero de la siguiente instrucción.

Punto Neuralgico 3:.

- Pop del stack de brincos.
- Pop del stack de brincos.
- Crear un Quad de GOTO hacia el segundo Pop
- Llenar el Quad del primer pop con la siguiente instrucción.

Print:



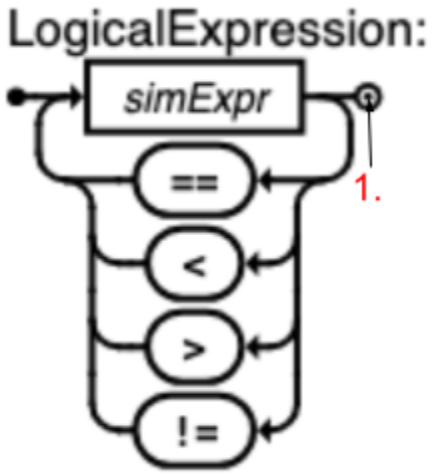
Punto Neuralgico 1:

- Conseguir la siguiente memoria temporal
- Leer valor.
- Llenar la siguiente memoria temporal con el valor.
- Crear quad de imprimir con la memoria temporal.

Punto Neuralgico 2:

- Pop al stack de operandos.
- Crear Quad de imprimir con el stack de operandos.

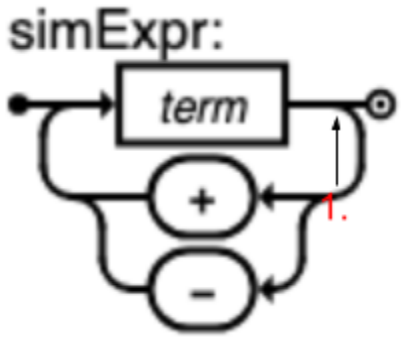
Logical Expression:



Punto Neuralgico 1:

- Comprueba si typeStack, operandStack u operatorStack están vacíos, retorna temprano si es el caso.
- Comprueba si el operador en la cima de operatorStack es uno de los operadores relacionales, retorna si no lo es.
- Extrae el operando derecho del stack de operandos.
- Extrae el tipo del operando derecho del stack de tipos.
- Extrae el operando izquierdo del stack de operandos.
- Extrae el tipo del operando izquierdo del stack de tipos.
- Extrae el operador relacional del stack de operadores.
- Usa una estructura SemanticCube para obtener el tipo de resultado basado en los tipos de los operandos y el operador.
- Comprueba si el tipo de resultado es ERROR, lanza una excepción si es el caso.
- Obtiene un nuevo espacio temporal de avail.
- Crea el Quad de la operacion
- Empuja el resultado en el stack de operandos.
- Empuja el tipo de resultado en stack de tipos.

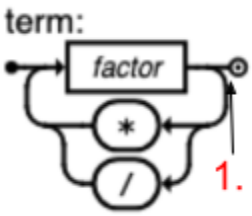
Simple Expression:



Punto Neuralgico 1:

- Comprueba si typeStack, operandStack u operatorStack están vacíos, retorna temprano si es el caso.
- Comprueba si el operador en la cima de operatorStack es uno de los operadores de suma o resta, retorna si no lo es.
- Extrae el operando derecho del stack de operandos.
- Extrae el tipo del operando derecho del stack de tipos.
- Extrae el operando izquierdo del stack de operandos.
- Extrae el tipo del operando izquierdo del stack de tipos.
- Extrae el operador relacional del stack de operadores.
- Usa una estructura SemanticCube para obtener el tipo de resultado basado en los tipos de los operandos y el operador.
- Comprueba si el tipo de resultado es ERROR, lanza una excepción si es el caso.
- Obtiene un nuevo espacio temporal de avail.
- Crea el Quad de la operacion
- Empuja el resultado en el stack de operandos.

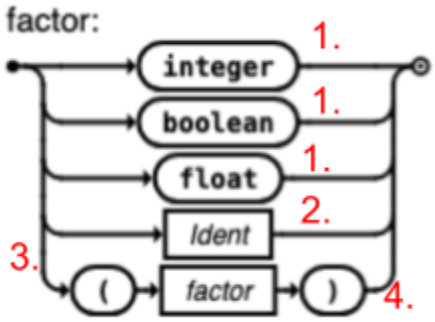
Term:



Punto Neuralgico 1:

- Comprueba si typeStack, operandStack u operatorStack están vacíos, retorna temprano si es el caso.
- Comprueba si el operador en la cima de operatorStack es uno de los operadores de multiplicacion o division, retorna si no lo es.
- Extrae el operando derecho del stack de operandos.
- Extrae el tipo del operando derecho del stack de tipos.
- Extrae el operando izquierdo del stack de operandos.
- Extrae el tipo del operando izquierdo del stack de tipos.
- Extrae el operador relacional del stack de operadores.
- Usa una estructura SemanticCube para obtener el tipo de resultado basado en los tipos de los operandos y el operador.
- Comprueba si el tipo de resultado es ERROR, lanza una excepción si es el caso.
- Obtiene un nuevo espacio temporal de avail.
- Crea el Quad de la operacion
- Empuja el resultado en el stack de operandos.

Factor:



Punto Neuralgico 1:

- Leer valor.
- Abrir memoria temporal.
- Empujar Memoria al stack de operandos
- Empujar tipo al stack de tipos
- Llenar memoria con el valor

Punto Neuralgico 2:

- Buscar variable
- Empujar tipo al stack de tipos
- Empujar Memoria al stack de operandos

Punto Neuralgico 3:

- Empujar paréntesis al stack de operadores

Punto Neuralgico 4:

- Popear stack de operadores

Maquina Virtual

Codigo:

```C/C++
#include "CodeVector.h"
#include "ConstantMap.h"
#include "Operations.hpp"
#include <vector>
#include <map>
#include <iostream>
#include <stdexcept>
#include <wchar.h>

int main() {
    int instructionPointer = 0;
    while (instructionPointer < code.size()) {
        int op = code[instructionPointer][0];
        int arg1 = code[instructionPointer][1];
        int arg2 = code[instructionPointer][2];
        int result = code[instructionPointer][3];

        switch (op) {
            case 0:  // ADD
                constantMap[result] = add(constantMap[arg1],
constantMap[arg2]);;
                // std::cout << "ADD" << constantMap[arg1] << '+' <<
constantMap[arg2] << '=' << constantMap[result] << '\n';
                break;
            case 1:  // SUB
                constantMap[result] = sub(constantMap[arg1],
constantMap[arg2]);
                // std::cout << "SUB" << constantMap[arg1] << '-' <<
constantMap[arg2] << '=' << constantMap[result] << '\n';
                break;
            case 2:  // MUL
                constantMap[result] = mul(constantMap[arg1],
constantMap[arg2]);
                break;
            case 3:  // DIV
                // if (constantMap[arg2] == 0) {
                //     throw std::runtime_error("Division by zero");
                // }
```

```
                constantMap[result] = div(constantMap[arg1],
constantMap[arg2]);
                // std::cout << "DIV" << constantMap[arg1] << '/' <<
constantMap[arg2] << '=' << constantMap[result] << '\n';
                break;
            case 4:  // EQU
                constantMap[result] = constantMap[arg1] == constantMap[arg2];
                break;
            case 5:  // LSS
                constantMap[result] = lessThan(constantMap[arg1],
constantMap[arg2]);
                break;
            case 6:  // GTR
                constantMap[result] = greaterThan(constantMap[arg1],
constantMap[arg2]);
                break;
            case 7:  // ASSIGN
                // std::cout << "ASSIGN " <<  arg2 << '=' << constantMap[arg1]
<< ' ';
                constantMap[arg2] = constantMap[arg1];
                // std::cout << constantMap[arg2] << '\n';
                break;
            case 8:  // LOAD
                constantMap[result] = arg1;  // Assuming arg1 is a value to be
loaded
                break;
            case 9:  // CONST
                constantMap[result] = arg1;  // Assuming arg1 is a constant
value
                break;
            case 10:  // FCALL
                std::cout << "Function call not implemented\n";
                break;
            case 11:  // RETURN
                std::cout << "Return not implemented\n";
                break;
            case 12:  // GOTO
                instructionPointer = result;
                continue;
            case 13:  // GOTOF
                if (const bool* condition =
boost::get<bool>(&constantMap[arg1])) {
                    if (!*condition) {
                        instructionPointer = result;
```

```
                    continue;
                }
            }
            break;
        case 14:  // STORE
            std::cout << "STORE not implemented\n";
            break;
        case 15:  // READ
            std::cout << "Read not implemented\n";
            break;
        case 16:  // WRITE
            std::cout << "Write not implemented\n";
            break;
        case 17:  // PRINT
            print(constantMap[result]);
            break;
        case 18:  // NEQU
            constantMap[result] = (constantMap[arg1] != constantMap[arg2]);
            break;
        default:
            throw std::runtime_error("Unknown opcode");
    }
    instructionPointer++;
  }
  return 0;
}
```

Explicacion:

Ejecuta un conjunto de instrucciones almacenadas en un vector (*code*) y utiliza un mapa de memoria (*constantMap*) para almacenar y manipular valores durante la ejecución. La VM soporta varias operaciones aritméticas, de comparación, de asignación, y control de flujo, entre otras.

Componentes Principales

Vector de Instrucciones (code)

Almacena las instrucciones que la VM debe ejecutar. Cada instrucción es un vector de enteros que contiene:

op: Código de operación que indica el tipo de instrucción.

arg1, arg2: Argumentos para la instrucción.

result: Índice donde se almacenará el resultado de la operación.

Mapa de Memoria (constantMap)

Almacena los valores constantes y los resultados de las operaciones. Los valores pueden ser de tipo entero, flotante, booleano o cadena de caracteres ancha (wstring).

Puntero de Instrucción (instructionPointer)

Mantiene la posición actual en el vector de instrucciones.

Estructura del Código

El código principal de la VM es un bucle while que itera sobre el vector de instrucciones y ejecuta la instrucción correspondiente basada en el código de operación (op). La ejecución se realiza mediante un switch que maneja diferentes códigos de operación.