Manual Técnico (TAC_Assembly_Generator)

Lenguaje: Java 11
Herramientas Secundarias: jflex, cup
Fecha De Desarrollo: Septiembre-Octubre 2020
Desarrollador:

Nombre: Sergio Daniel CIfuentes Argueta
GitHub: https://github.com/SergioCifuentes/
E-Mail: sergiodaniel-cifuentesargueta@cunoc.edu.gt

El objetivo de la aplicación es convertir varios tipos de lenguajes combinados (Visual Basic, Python, Java, C) a código tres dimensiones, para esto se necesita un archivo de entrada según estos lenguajes. Para analizar el archivo de entrada léxicamente y sintácticamente se utilizó jflex y cup respectivamente. Se recomienda que pase por el archivo de GramicaMlg que se encuentra en la misma carpeta de DocumentacionMLG.

Cuádruplos:
Para la generación de código en tres dimensiones se utilizó la herramienta de cuádruplos. Estos cuadrupulos se generaban durante la compilacion para ser transformados al final a TAC.

```
public class Quadruple {
    private Integer op;
    private Object arg1;
    private Object arg2;
    private String result;
    private boolean constante;

    public Quadruple(Integer op, Object arg1, Object arg2, String result) {
        this.op = op;
        this.arg1 = arg1;
        this.arg2 = arg2;
        this.result = result;
        constante=false;
    }
```

Este objeto está ubicado en *tac_assembly_generator.TAC.quadruple* para que lo pases viendo. El control de estos elementos lo llevaban a cabo las tablas de cuadruplos que representaban el bloqueo o ambito de código en el que se encuentra la compilación, al momento de llegar al final de un bloque de codigo se la Tabla con atributos sintetizados le pasaba el arreglo de quadruples a su padre para que al final todos los cuadruplos estubieran juntos.

```java
public class QuadrupleTable {

    private ArrayList<Object> quadruples;
    private ArrayList<ArrayList<Object>> idQuads;
    private QuadrupleTable father;
    private Switch switchAsst;
    private For forAsst;
    private TempGenerator tem;

    public QuadrupleTable(QuadrupleTable father,TempGenerator temp) {
        this.father=father;
        tem=temp;
        this.quadruples = new ArrayList<>();
        idQuads = new ArrayList<>();
        idQuads.add(new ArrayList<>());
    }
```

Este es un ejemplo de cómo se ven los Quadruples juntos al final del archivo:

```
Quadruple{op=1, arg1='y', arg2=null, result=t8}
Quadruple{op=1, arg1=4, arg2=null, result=t9}
Quadruple{op=1, arg1=3, arg2=null, result=t10}
Quadruple{op=1, arg1=2, arg2=null, result=t11}
Quadruple{op=3, arg1=3, arg2=2, result=t10}
Quadruple{op=5, arg1=t9, arg2=t10, result=t11}
Quadruple{op=1, arg1=2, arg2=null, result=t12}
Quadruple{op=1, arg1=4, arg2=null, result=t13}
Quadruple{op=1, arg1=1, arg2=null, result=t14}
Quadruple{op=1, arg1=4, arg2=null, result=t15}
```

Como puede notar el objeto tiene números como op, que representan la operación que va a llevar a cabo el cuádruplos, estas operaciones se encuentran en la clase Operation en el paquete: *tac_assembly_generator.TAC.quadruple*.
Con todos los elementos generados al final se hace una simple traducción como el siguiente algoritmo que se encuentra en *tac_assembly_generator.TAC.TAC*.

```java
if (quadAsst.getOp() == null) {
    OutputText.appendToPane(mainFrame.getTACPannel(), quadAsst.getResult() + ":\n", Color.blue, false);
} else if (quadAsst.getOp().equals(Operation.GO_TO)) {
    OutputText.appendToPane(mainFrame.getTACPannel(), Operation.getIntOpOutput(quadAsst.getOp()) + " " + quadAsst.getResult() + "\n", Color.w
} else if (quadAsst.getOp() == Operation.EQUAL) {
    if (quadAsst.isConstante()) {
        OutputText.appendToPane(mainFrame.getTACPannel(), "const ", Color.white, false);
    }
    String tacQuad = quadAsst.getResult() + Operation.getIntOpOutput(quadAsst.getOp()) + quadAsst.getArg1();
    OutputText.appendToPane(mainFrame.getTACPannel(), tacQuad + "\n", Color.white, false);
} else if (quadAsst.getOp() <= Operation.MINUS) {
    String tacQuad = quadAsst.getResult() + "=" + quadAsst.getArg1() + Operation.getIntOpOutput(quadAsst.getOp()) + quadAsst.getArg2();
    OutputText.appendToPane(mainFrame.getTACPannel(), tacQuad + "\n", Color.white, false);
```

Esto solo son algunos casos de operaciones que pueden llevar los cuádruplos.
Antes de llegar a este punto en donde se genera los cuádruplos, primero se debe llevar a cabo pruebas de semántica, que principalmente se encarga la clase TestManager (*tac_assembly_generator.languages.semantic.verification*), esta clase es la conexión del analizador sintáctico y todas las pruebas semánticas que se aplican, además de tener una muy fuerte conexión con la tabla de símbolos, en donde los terminaran al momento de comprobar su estado.

TypeManager:
Uno de los elementos que tiene este TestManager es el TypeManager (*tac_assembly_generator.languages.semantic.type*) el cual llevará a cabo el control de los tipos según el lenguaje analizando.

```
public class TypeManager {

    private static final String INTEGER_NAME = "int";
    public static final int INTEGER_TYPE = 2;
    private static final String FLOAT_NAME = "float";
    public static final int FLOAT_TYPE = 1;
    private static final String CHAR_NAME = "char";
    public static final int CHAR_TYPE = 3;
    private static final String BOOL_NAME = "boolean";
    public static final int BOOL_TYPE = 4;
    private static final String VAR_NAME = "var";
    public static final int VAR_TYPE = 5;
    private static final String CLASS_NAME = "class";
    public static final int CLASS_TYPE = 6;
    public static final Type CLASS= new Type(CLASS_NAME, CLASS_TYPE,null);
```

El principal método es el de loadnextType() el cual será llamado al momento de terminar un lenguaje de archivo de entrada y pasar al siguiente. Esto lo que hará es cambiar los tipos que se están manejando y los padres de dichos tipos

```
public void loadnextType() {


    switch (languageType) {
        case VB_TYPES:
            loadTypes(JAVA_TYPES);
            break;
        case JAVA_TYPES:
            loadTypes(PYTHON_TYPES);
            break;
        case PYTHON_TYPES:
            loadTypes(C_TYPES);
            break;

        default:
            throw new AssertionError();
    }
}
```

```
public void loadTypes(int typeOfTypes) {
    types = new Type[5];
    languageType = typeOfTypes;
    switch (typeOfTypes) {
        case VB_TYPES:
            types[0] = new Type(FLOAT_NAME, FLOAT_TYPE, null);
            types[1] = new Type(INTEGER_NAME, INTEGER_TYPE, types[0]);
            types[2] = new Type(CHAR_NAME, CHAR_TYPE, null);
            types[3] = new Type(BOOL_NAME, BOOL_TYPE, null);
            break;
        case JAVA_TYPES:
            types[0] = new Type(FLOAT_NAME, FLOAT_TYPE, null);
            types[1] = new Type(INTEGER_NAME, INTEGER_TYPE, types[0]);
            types[2] = new Type(CHAR_NAME, CHAR_TYPE, types[1]);
            types[3] = new Type(BOOL_NAME, BOOL_TYPE, null);

               break;
        case PYTHON_TYPES:
            types[0] = new Type(FLOAT_NAME, FLOAT_TYPE, null);
            types[1] = new Type(INTEGER_NAME, INTEGER_TYPE, types[0]);
            types[2] = new Type(CHAR_NAME, CHAR_TYPE, types[1]);
            types[3] = new Type(BOOL_NAME, BOOL_TYPE, null);
            types[4] = new Type(VAR_NAME, VAR_TYPE, null);
            break;
        case C_TYPES:
```

Estos tipos se utilizaran para realizar operaciones y comprobar los tipos al momento de compilación.

AmbitControler:

El controlador de ámbito (*tac_assembly_generator.languages.semantic*) se encarga de conjuntamente con la tabla de símbolos, controlar las variables con las que se están trabajando en un lenguaje en específico. Para esto utilizamos un sistema de jerarquía para etiquetar cada bloque de codigo con su respectivo ámbito, esta etiqueta es la clase *Ambit.*

```
public class Ambit {
    private int id;
    private Ambit father;

    public Ambit(int id, Ambit father) {
        this.id = id;
        this.father = father;
    }
```

Que es una simple clase con un id y otra instancia father que representa su padre, si el ámbito es el principal el father será null. Al momento de llamar a una variable o de buscarlos en la tabla de símbolos, el símbolo debe de estar en el mismo ámbito o padre en el que se está llamando.

SymbolTable:

La tabla de símbolos (tac_assembly_generator.languages.semantic), como sabemos controla las variables con las que estamos trabajando, en este se guarda cada información para cada tuple, esto para asegurar el ámbito perteneciente y tipo del symbolo.

```
public class Tuple {
    private String name;
    private String functionName;
    private Type type;
    private Object value;
    private Symbol symbol;
    private Ambit ambit;
    private ArrayList<Tuple> parameters;
    private ArrayList<Object> dimensions;
    private Tuple classFather;
    private String language;
    private boolean constante;
    //0 normal type
    //1 one dimension array ...
    private Integer dimension;

    public Tuple(String name, Type type, Object value,Integer dimension, Symbol symbol,Ambit ambit) {

        this.name = name;
        this.type = type;
        this.value = value;
        this.symbol=symbol;
        this.dimension=dimension;
        this.ambit=ambit;
        constante=false;
```