Trabajo Práctico N°3 y N°4
Diseño de Compiladores I
Grupo 4

Integrantes:
Freire, Agustín
Loguercio, Giorgio
Sottile, Gianni

agustinfreire2002@outlook.com.ar
loguerciogiorgioivan@gmail.com
giannisottile7@gmail.com

Facultad de Cs. Exactas
Universidad Nacional del Centro de la Provincia de Buenos Aires

Introducción

Durante el presente trabajo se desarrollará las últimas etapas de un compilador siguiendo las instrucciones de la cátedra, contando con las etapas de generación de código intermedio y generación de código assembler. Se desarrollará en JAVA utilizando la herramienta de yacc, con representación de Tercetos y con código assembler de WebAssembly.

El objetivo de estas etapas será la generación de código intermedio para las sentencias ejecutables y, finalmente, la generación de código assembler.

Decisiones de diseño e implementación

Rango representable de enteros en webassembly

Mientras probamos casos, nos encontramos con que en webassembly se utiliza un rango de representación menor al que nos pedía la cátedra para los enteros.
Razón por la cual, tuvimos que cambiar el máximo representable que permite nuestro compilador.

Representación intermedia

Para la representación de Tercetos se creó una clase Terceto con la siguiente estructura:

```
public class Terceto {
    private String operador;
    private String op1;
    private String op2;
    private String tipo;
    private boolean hecho;
```

Donde tipo sirve para ir llevando el tipo durante la generación de tercetos en las expresiones matemáticas.
El boolean “hecho” es necesario para pattern matching (se explica más tarde)

Esta clase es utilizada dentro de un ArrayList para ir llevando todos los tercetos generados junto a su índice.

Se creó una clase GeneracionCodigo la cual se encarga de generar el código intermedio y realizar los chequeos necesarios.

```
public class GeneradorCodigo {
    private ArrayList<Terceto> tercetos = new ArrayList();
    private Stack<Integer> flujoControl = new Stack();
    private boolean huboError = false;
```

Esta clase es instanciada para cada ámbito, ya que las funciones deben tener sus propios tercetos asociados, manteniendo todo en una pila de generadores de código en el parser.

Cada terceto generado es asignado a la regla de la forma: $$.sval = terceto; para que luego sea accesible de ser necesario mediante la notación posicional.

Se generan tercetos para:

- Asignación

- Expresión Matemática
- Comparaciones
- Pattern Matching (explicado más adelante)
- Invocación a función
- OUTF
- RET
- Flujos de control (IF, IF ELSE, FOR)
- Conversión en parámetro real
- Acceso a triplas
- Adicionales explicados en su respectiva sección

Donde cada operando es el respectivo a su semántica.

Ámbito

Para posibilitar la existencia de distintos ámbitos y sus chequeos correspondientes a las reglas de alcance, se creó una variable dentro del Parser que lleva el ámbito actual, inicializandose “global”.

Al declarar variables y funciones, se las agrega a la tabla de símbolos con el ámbito actual como prefijo.

Dado que en el anterior trabajo práctico se agregaban los identificadores directamente a la tabla de símbolos, debimos quitar esa funcionalidad ya que la implementación de los ámbitos nos obligó a llevarla a la gramática.

Los ámbitos nuevos se crean solamente en el cuerpo de las funciones declaradas.

Al terminar el cuerpo de una función, se llama a este método del parser el cual modifica el ámbito actual eliminando el de la función que ha sido finalizada.

```
void cambiarAmbito(){
        int index = this.ambitoActual.lastIndexOf(":");
        if (index != -1) {
            this.ambitoActual = this.ambitoActual.substring(0, index);
        }
}
```

Chequeos de declaración

```
boolean esEmbebido(String sval){
    char firstC=sval.charAt(0);
    if ( firstC =='x' || firstC =='y' || firstC =='z' || firstC == 'd' ) {
        return true;
    }
    return false;
}

boolean redeclaracionTipoErroneo(){
    char firstC=yylval.sval.charAt(0);
    if ( (tipoVar.equals(AccionSemantica.DOUBLE) && (firstC =='x' || firstC =='y' || firstC =='z')) ||
    (tipoVar.equals(AccionSemantica.ULONGINT) && (firstC == 'd')) ) {
        return true;
    }
    return false;
}
```

```
void checkRedeclaracion(String val){
    String varAmbito = ambitoActual +":"+val;
    if (esEmbebido(val)){
        ErrorHandler.addErrorSemantico("Se redeclaro el tipo de la variable", lex.getLineaInicial());
    }
    else if (redeclaracionTipoErroneo()){
        ErrorHandler.addErrorSemantico("Se redeclaro el tipo de la variable y con tipos erroneos", lex.
        getLineaInicial());
    }
    else if (this.ts.estaEnTablaSimbolos(varAmbito)){
        ErrorHandler.addErrorSemantico("Se redeclaro la variable en el ambito: " +ambitoActual + " " , lex.
        getLineaInicial());
    }
    else {
        ts.addClave(varAmbito);
        ts.addAtributo(varAmbito,AccionSemantica.TIPO,tipoVar);
        ts.addAtributo(varAmbito,AccionSemantica.USO,"nombre variable");
        if(!tipoVar.equals(AccionSemantica.DOUBLE) && !tipoVar.equals(AccionSemantica.ULONGINT)){
            if(!this.ts.getAtributo(tipoVar, AccionSemantica.TIPO).equals("")){
                ts.addAtributo(varAmbito, AccionSemantica.TIPO_BASICO, this.ts.getAtributo(tipoVar,
                AccionSemantica.TIPO));
            } else {
                ErrorHandler.addErrorSemantico("No existe la tripla con el ID " + tipoVar, lex.getLineaInicial
                ());
            }
        }
    }
}
```

La función esEmbebido se fija que el identificador pasado por parámetro no sea de tipo embebido ya que son globales y no requieren declaración, simplemente se utilizan.

La función redeclaracionTipoErroneo si bien se fija que sean embebidos, sirve para dar mayor información en el mensaje de error diciéndole al programador que está intentando declarando una embebida la cual no es declarable y, a su vez, intenta declararle un tipo erróneo.

El método checkRedeclaracion utiliza los métodos nombrados anteriormente y, si no hubo ningún error se fija si fue redeclarada buscándola en la tabla de símbolos.

En caso de cumplir estos chequeos, se la agrega a la tabla de símbolos, con sus respectivos atributos de Tipo y Uso.

A su vez, si el tipo no es double ni ulongint, implica que fue definido por el usuario. Si el mismo no existe en la tabla de símbolos se informa con un error, y sino se añade el tipo básico a la variable declarada de tipo definido por el usuario.

Para las funciones

```
void checkRedFuncion(String nombre, String tipo){
    String funcionAmbito = ambitoActual+":"+nombre;
    if(this.ts.estaEnTablaSimbolos(funcionAmbito)){
        ErrorHandler.addErrorSemantico("Funcion redeclarada "+nombre, lex.getLineaInicial());
    }
    else {
        this.ts.addClave(funcionAmbito);
        ts.addAtributo(funcionAmbito,AccionSemantica.USO,"nombre funcion");
        this.ts.addAtributo(funcionAmbito, AccionSemantica.TIPO, tipo);
    }
}
```

Se realiza chequeo de redeclaración, de lo contrario simplemente se agrega.

Para los factores de una expresion_matematica, realizamos el chequeo de la declaración si es un ID en nivel factor:

```
factor      : ID {  gc.checkDeclaracion($1.sval, lex.getLineaInicial(), this.ts,ambitoActual);}
        | constante
        | invoc_fun
        | triple
        ;
```

Asignaciones

```
asignacion  : triple_asig ASIGN expresion_matematica {estructurasSintacticas("Se realizó una asignación a
la variable: " + $1.sval + " en la linea: " + lex.getLineaInicial());
                        $$.sval = gc.checkTipoAsignacion($1.sval, lex.getLineaInicial(), $3.sval, this.ts,
                        ambitoActual);}

        | ID ASIGN expresion_matematica {  estructurasSintacticas("Se realizó una asignación a la
        variable: " + $1.sval + " en la linea: " + lex.getLineaInicial());
                    $$.sval = gc.checkTipoAsignacion($1.sval, lex.getLineaInicial(), $3.sval, this.ts,
                    ambitoActual);
        }
        ;
```

Para las asignaciones creamos un método checkTipoAsignacion que chequea los tipos y retorna el terceto ya añadido.

```
    public String checkTipoAsignacion(String id, int lineaActual, String opAsig, TablaSimbolos ts, String
ambitoActual) {
        Pattern pattern = Pattern.compile("\\[(\\d+)\\]");
        Matcher matcher = pattern.matcher(opAsig);

        Matcher matcher_id = pattern.matcher(id);
        boolean matchres_id = matcher_id.find();
        boolean matchres = matcher.find();
        boolean declarado = true;

        String id_izq="";
        String id_der="";
        String tipo_der="";
        String tipo_izq = "";

        if(matchres_id) {
            Terceto t = this.getTerceto(Integer.parseInt(matcher_id.group(1)));
            id_izq = t.getOp1();
            id_izq = checkDeclaracion(id_izq, lineaActual, ts,ambitoActual);
            tipo_izq = ts.getAtributo(id_izq, AccionSemantica.TIPO_BASICO);
            id_izq = id;
        } else {
            id_izq = id;
            id_izq = checkDeclaracion(id_izq, lineaActual, ts,ambitoActual);
            tipo_izq = ts.getAtributo(id_izq, AccionSemantica.TIPO);
        }

        if(id_izq == null) {
            declarado = false;
        }
```

Primero nos fijamos si en el lado izquierdo de la asignación tenemos un elemento de la tripla o una variable. Para esto, será un acceso a tripla si hay un entero entre corchetes, lo cual indica un terceto de acceso a tripla, donde debemos retornar el terceto del acceso. En base a esto, se obtiene el tipo de forma distinta y se chequea la declaración de la variable.

Realizamos lo mismo para el lado derecho

```
if(matchres) {
    Terceto t = this.getTerceto(Integer.parseInt(matcher.group(1)));
    if(t.getOperador().equals("ACCESOTRIPLE")) {
        id_der = t.getOp1();
        id_der = checkDeclaracion(id_der, lineaActual, ts,ambitoActual);
        id_der = opAsig;
        tipo_der = t.getTipo();
    } else {
        id_der = opAsig;
        tipo_der = t.getTipo();
    }
    if(tipo_der.equals("error")) {
        declarado = false;
    }
} else {
    id_der = opAsig;
    id_der = checkDeclaracion(id_der, lineaActual, ts,ambitoActual);
    tipo_der = ts.getAtributo(id_der, AccionSemantica.TIPO);
    if(id_der == null) {
        declarado = false;
    }
}
```

Como se puede asignar, además de una variable o un elemento de una tripla, una expresión matemática, se debe diferenciar todos los casos para obtener el tipo y chequear las respectivas declaraciones.
A su vez, si se asigna un acceso a tripla debemos retornar el terceto asociado al acceso.

La declaración viene del método checkDeclaracion:

```
public String checkDeclaracion(String id, int lineaInicial, TablaSimbolos ts, String ambito) {
   Pattern pattern = Pattern.compile("\\[(\\d+)\\]");
   Matcher matcher = pattern.matcher(id);
   if (!id.matches("^[0-9].*") && !id.matches("^-.*") && !matcher.find()) {
      String var = this.obtenerVariableSinAmbito(id);
      String[] partes = ambito.split(":");

      for(int i = partes.length - 1; i >= 0; --i) {
         String nuevaCadena = String.join(":", (CharSequence[])Arrays.copyOfRange(partes, 0, i + 1));
         String claveTs = nuevaCadena + ":" + var;
         if (ts.estaEnTablaSimbolos(claveTs)) {
            return claveTs;
         }
      }

      ErrorHandler.addErrorSemantico("La variable " + id + " no esta al alcance o no fue declarada.",
      lineaInicial);
      return null;
   } else {
      return id;
   }
}
```

Si estamos chequeando una variable, la buscaremos en la tabla de símbolos obteniendo la declaración más cercana al ámbito y la retornamos.

De lo contrario, se retorna el id ya que si era una constante no se debe chequear el ámbito, y si era un terceto tampoco.
Si no se encuentra ninguna, se da un error ya que no estaba al alcance o no fue declarada.

Siguiendo con el código de checkTipoAsignacion,

```
        if(declarado) {
            if(!tipo_der.equals(tipo_izq)) {
                if(id_der.equals("")) {
                    ErrorHandler.addErrorSemantico("Tipo inesperado en asignacion. La variable izquierda " +
id_izq + " es " + tipo_izq + " y lo que se asigna es " + tipo_der, lineaActual);
                }
                else {
                    ErrorHandler.addErrorSemantico("Tipo inesperado en asignacion. La variable izquierda " +
id_izq + " es " + tipo_izq + " y la variable derecha " + id_der +  " es " + tipo_der, lineaActual);
                }
            }
        }else {
            ErrorHandler.addErrorSemantico("La variable no esta declarada en la Asignacion",lineaActual);
        }

        return this.addTerceto(":=", id_izq, id_der, tipo_der);
    }
```

Si no hubo errores de declaración, se fija que no haya errores de tipos distintos. Caso contrario, se avisa del error de que no estaba declarada.

Independientemente de que haya habido errores, se retorna el terceto.

Expresiones matemáticas

Se realiza un chequeo con el método checkTipoExpresion, tanto en expresion_matematica como en termino

```
expresion_matematica    : expresion_matematica '+' termino {$$.sval = gc.checkTipoExpresion($1.sval, $3.sval,
lex.getLineaInicial(), this.ts, "+",ambitoActual);}

                | expresion_matematica '-' termino {$$.sval = gc.checkTipoExpresion($1.sval, $3.sval,
lex.getLineaInicial(), this.ts, "-",ambitoActual);}

                | termino { $$.sval = $1.sval;}
```

```
termino         : termino '*' factor {$$.sval = gc.checkTipoExpresion($1.sval, $3.sval, lex.getLineaInicial(),
this.ts, "*",ambitoActual);}
                | termino '/' factor {$$.sval = gc.checkTipoExpresion($1.sval, $3.sval, lex.getLineaInicial(),
this.ts, "/",ambitoActual);}
                | factor {$$.sval = $1.sval;}
```

El método es:

```
    public String checkTipoExpresion(String op_izq, String op_der, int
lineaActual, TablaSimbolos ts, String operando,String ambitoActual)
```

Se utiliza la misma lógica anterior para chequear si fue declarada o no, primero fijandonos si es un terceto.
En caso de que la expresión esté declarada, se chequean los tipos de los operandos.

```
if(declarado) {
    if(!tipo_der.equals(tipo_izq)) {
            ErrorHandler.addErrorSemantico("Tipo inesperado en la expresion. La variable izquierda " +
            id_izq + " es " + tipo_izq + " y la variable derecha " + id_der +  " es " + tipo_der,
            lineaActual);
    }
}
else {
    ErrorHandler.addErrorSemantico("alguna de las variables no esta declarada en la expresion",
    lineaActual);
}

String retorno = this.addTerceto(operando, id_izq, id_der, tipo_izq);
return retorno;
```

Pattern Matching

Para la regla de : '(' patron_izq ')' comparador '(' patron_der ')'
Antes teníamos solo un no terminal “patron” pero debimos realizar la distinción.

Para controlar la longitud de los patrones deberemos tener en cuenta la cantidad de variables de cada patrón y la posibilidad de que se utilicen expresiones matemáticas ya que las mismas generan tercetos.

Nuestro primer enfoque era contar la cantidad de tercetos, pero al encontrarnos con que las expresiones matematicas generaban más, resultó ser erróneo y nos obligó a distinguir el patrón izquierdo del derecho y llevar la cuenta de la posición del patrón.

```
Integer inicioPatron;
Integer posPatron;
int cantPatronIzq;
int cantPatronDer;
```

patron_izq : lista_patron_izq ',' expresion_matematica
Luego de leerse el patrón izquierdo, se inicializan las variables necesarias

```
this.iniciarPatron();
this.cantPatronIzq++;
$$.sval = gc.addTerceto("COMP", gc.checkDeclaracion($3.sval, lex.getLineaInicial(), this.ts, this.ambitoActual), "");
```

```
void iniciarPatron(){
        if(this.inicioPatron > gc.getCantTercetos()){
                this.inicioPatron = gc.getCantTercetos();
                this.posPatron = this.inicioPatron;
        }
}
```

Se aumenta la cantidad de elementos del patrón izquierdo y se genera el terceto con operador “COMP” y operando 2 en vacío, que luego se completará, y se chequea la declaración del elemento.

Para la parte recursiva del patrón izquierdo funciona igual.

lista_patron_izq : lista_patron_izq ',' expresion_matematica

| expresion_matematica {this.iniciarPatron(); this.cantPatronIzq=1; $$.sval = gc.addTerceto("COMP", gc.checkDeclaracion($1.sval, lex.getLineaInicial(), this.ts, this.ambitoActual), "");}
;

Para el lado derecho

patron_der : lista_patron_der ',' expresion_matematica

```
this.cantPatronDer++;
posPatron = gc.updateAndCheckSize(this.posPatron, gc.checkDeclaracion($3.sval, lex.getLineaInicial(),
this.ts, this.ambitoActual), lex.getLineaInicial(), this.ts, this.ambitoActual);
this.posPatron++;
```

Se aumenta la cantidad de elementos del patrón derecho
Se obtiene la posición actual, a su vez chequeando la declaración del elemento.
Se aumenta la posición actual que se está recorriendo.

Lo mismo se realiza para la parte recursiva.

El método update and check size funciona de la siguiente manera:

```
public int updateAndCheckSize(int pos, String op2, int lineaActual, TablaSimbolos ts, String ambitoActual) {
    if(pos >= this.getCantTercetos()) {
        ErrorHandler.addErrorSemantico("La longitud de los patrones a matchear es distinta.", lineaActual);
    } else {
        Terceto t = this.getTerceto(pos);
        int newPos = pos;
        while(newPos < this.getPosActual() && !t.getOperador().equals("COMP")) {
            newPos++;
            t = this.getTerceto(newPos);
        }
        if(t.getOperador().equals("COMP")) {
            t.setOp2(op2);
        }
        this.checkTipo(newPos, lineaActual, ts, ambitoActual, operando:"");
        pos = newPos;
    }
    return pos;
}
```

Se fija si la posición en que se recorre el patrón es mayor a la cantidad de tercetos, tal caso significa que la longitud de los patrones a matchear es distinta.

De lo contrario, se obtiene el terceto de la posición actual del patrón y avanza, si es necesario, hasta encontrar un terceto donde el operando sea “COMP” y reemplaza el segundo operando que habíamos dejado en blanco al principio.

A su vez, se chequea el tipo de ambos elementos que se compararon.

Tras todo esto, se llega a la regla del nivel más alto del patrón:

'(' patron_izq ')' comparador '(' patron_der ')'

La acción tras leer esta regla es fijarse si el segundo operando del último terceto generado esta vacío (lo cual significa que la longitud de los patrones es distinta)

```
if(gc.getTerceto(gc.getPosActual()).getOp2().isEmpty()){
    ErrorHandler.addErrorSemantico("La longitud de los patrones a matchear es distinta.", lex.getLineaInicial());}
else { $$.sval = gc.updateCompAndGenerate(this.inicioPatron, $4.sval, this.cantPatronIzq, this.cantPatronDer, lex.getLineaInicial());}

this.inicioPatron = Integer.MAX_VALUE;
$$.sval = "[" + this.gc.getPosActual() + "]";
```

De lo contrario, se actualizan

```
public String updateCompAndGenerate(int pos, String comp, int sizePatronIzq, int sizePatronDer, int lineaActual) {
    if(sizePatronIzq < sizePatronDer) {
        ErrorHandler.addErrorSemantico("La cantidad de elementos del patron izquierdo es menor a la del patron derecho", lineaActual);
    } else if (sizePatronIzq > sizePatronDer){
        ErrorHandler.addErrorSemantico("La cantidad de elementos del patron derecho es menor a la del patron izquierdo", lineaActual);
    } else {
        ArrayList<Integer> comparadores = new ArrayList<>();
        for (int i = pos; i < this.tercetos.size(); i++) {
            Terceto t = this.getTerceto(i);
            if(t.getOperador().equals("COMP")) {
                t.setOperador(comp);
                comparadores.add(i);
            }
        }
        int ultimoTercetoGenerado = pos;
        this.addTerceto(op:"AND", "[" + comparadores.getFirst() + "]", "[" + comparadores.get(1) + "]");
        ultimoTercetoGenerado = this.tercetos.size() - 1;
        int size = this.tercetos.size();
        for (int i = 2; i < comparadores.size(); i++) {
            this.addTerceto(op:"AND", "[" + ultimoTercetoGenerado + "]", "[" + comparadores.get(i) + "]");
            ultimoTercetoGenerado = this.tercetos.size() - 1;
        }
        return "[" + ultimoTercetoGenerado + "]";
    }
    return "error";
}
```

Se chequea cual patrón tenía una cantidad menor.
Si no hubo errores, se procede a la generación de los tercetos que representan la lógica final.
Se crea un terceto AND con los dos primeros elementos a comparar, y luego se van creando AND entre el último terceto generado y el próximo a comparar.
Dado n comparadores, se crean n-1 tercetos AND

Finalmente, se retorna el terceto.

Sentencias de control

Para la generación de Tercetos en flujos de control IF, IF-ELSE y FOR, se utilizó una pila que guarda el índice de los tercetos a completar. Si bien los casos pueden ser distintos, la solución es mediante la misma pila.

Para esto, creamos reglas para cada punto de control, ya que si ponemos código directamente entre los no-terminales, generaba muchos errores.

Para el IF y IF-ELSE tenemos reglas distintas, ya que si no se tiene else, la bifurcación por falso es luego del cuerpo del IF y no se genera una bifurcación incondicional.

seleccion : IF condicion_punto_control THEN cuerpo_control sinelse_punto_control
| IF condicion_punto_control THEN cuerpo_control else_punto_control cuerpo_control endif_punto_control

```
sinelse_punto_control : END_IF {
                        gc.actualizarBF(gc.getCantTercetos());
                        gc.pop();
                        }
                        ;

condicion_punto_control : condicion {
                        gc.addTerceto("BF", $1.sval, "");
                        gc.push(gc.getPosActual());
                }
                ;
else_punto_control : ELSE {
                        gc.addTerceto("BI", "", "-");
                        int posSig = gc.getCantTercetos();
                        gc.actualizarBF(posSig);
                        gc.pop();
                        gc.push(gc.getPosActual());
                        this.gc.addTerceto("Label" + posSig, "-", "-");
                }
                ;
endif_punto_control : END_IF {
                        int posSig = gc.getCantTercetos();
                        gc.actualizarBI(posSig);
                        this.gc.addTerceto("Label" + posSig, "-", "-");
                        gc.pop();
                        estructurasSintacticas("Se definió una sentencia de control con else, en la linea: " + lex.getLineaInicial());
                }
                ;
```

Para el flujo de control del bucle FOR

```
for             : FOR '(' asignacion_for ';' condicion_for ';' foravanc CTE ')' cuerpo_iteracion {      estructurasSintacticas("Se declaró un bucle FOR en la linea: " + lex.getLineaInicial());
                        String var = this.varFors.get(this.varFors.size()-1);
                        if(!this.ts.getAtributo($8.sval, AccionSemantica.TIPO).equals(AccionSemantica.ULONGINT)){
                                ErrorHandler.addErrorSemantico("La constante de avance no es de tipo entero.", lex.getLineaInicial());
                                gc.addTerceto("+", gc.checkDeclaracion(var, lex.getLineaInicial(), this.ts, this.ambitoActual), String.valueOf($7.ival * Double.parseDouble($8.sval)));
                        } else {
                                gc.addTerceto("+", gc.checkDeclaracion(var, lex.getLineaInicial(), this.ts, this.ambitoActual), String.valueOf($7.ival * Integer.parseInt($8.sval)));
                        }
                        this.varFors.remove(this.varFors.size()-1);
                        gc.addTerceto("BI", $5.sval, "");
                        gc.actualizarBF(gc.getCantTercetos());
                        gc.pop();
                        this.gc.addTerceto("Label" + this.gc.getCantTercetos(), "-", "-");
                }
```

Se utiliza un ArrayList de variables de for, ya que si hay fors anidados se pierde la variable del for anterior.
Se realiza el chequeo de tipo sobre la constante de avance.

Con los puntos de control:

```
condicion_for   : condicion { $$.sval = $1.sval;
                              gc.addTerceto("BF", $1.sval, "");
                              gc.push(gc.getPosActual());
                        }
                ;

asignacion_for  : ID ASIGN CTE {String varFor = gc.checkDeclaracion($1.sval,lex.getLineaInicial(),this.ts,ambitoActual);
                              if (varFor != null){
                                      if(!this.ts.getAtributo(varFor, AccionSemantica.TIPO).equals(AccionSemantica.ULONGINT))
{ErrorHandler.addErrorSemantico("La variable " + $1.sval + " no es de tipo entero.", lex.getLineaInicial());}
                                      gc.addTerceto(":=",varFor, $3.sval);
                              }
                              else{
                                      gc.addTerceto(":=", $1.sval, $3.sval);
                              }
                              if(!this.ts.getAtributo($3.sval, AccionSemantica.TIPO).equals(AccionSemantica.ULONGINT))
{ErrorHandler.addErrorSemantico("La constante " + $3.sval + " no es de tipo entero.", lex.getLineaInicial());}
                              this.varFors.add($1.sval);
                              this.gc.addTerceto("Label" + this.gc.getCantTercetos(), "-", "-");
                              }
                ;

foravanc        : UP {$$.ival = 1;}
                | DOWN {$$.ival = -1;}
                ;
```

Se utiliza la pila de flujo de control para llevar correctamente las bifurcaciones asociadas a este tipo de sentencia de control.

Se realiza el chequeo de redeclaración o que no esté declarada la variable.

Se realiza el chequeo de tipo sobre la variable a asignar y la constante asignada, ya que ambas deben ser de tipo entero, en nuestro caso ulongint.

Además, una decisión de implementación fue que el UP nos diera una constante 1 y el DOWN -1 para luego multiplicarla por la constante de avance y sumarla a la variable, manejando de esta manera correctamente la semántica de avance.

Tags

Para los tags, creamos una clase ControlTagAmbito que se encarga de controlar la declaración de los tags en un ámbito dado, ya que el uso de tags y saltos está restringido a un mismo ámbito.

```java
public class ControlTagAmbito {
   private HashMap<String, Boolean> tags = new HashMap();

   public ControlTagAmbito() {
   }

   public void huboGoto(String tag) {
      if (!this.tags.containsKey(tag)) {
         this.tags.put(tag, false);
      }

   }

   public void declaracionTag(String tag, int linea) {
      if (!this.tags.containsKey(tag)) {
         ErrorHandler.addErrorSemantico("No se pueden declarar etiquetas antes de un GOTO (no se permite saltar hacia arriba).", linea);
      } else {
         this.tags.replace(tag, true);
      }

   }

   public void tagsValidos(int linea) {
      Iterator var3 = this.tags.keySet().iterator();

      while(var3.hasNext()) {
         String s = (String)var3.next();
         if (!(Boolean)this.tags.get(s)) {
            ErrorHandler.addErrorSemantico("Se hace un goto a la etiqueta: " + s + " la cual no ha sido de declarada en el ambito.", linea);
         }
      }

   }
}
```

Esto mantiene para cada etiqueta un booleano de si fue declarada o no.

Decisión de diseño (RESTRICCION)
Debido a la complejidad de poder realizar GOTO en ejecución hacia arriba, pusimos la única restricción de que no se pueda saltar hacia arriba, por lo que en esta misma clase restringimos tal comportamiento.

En el parser, utilizamos un arraylist de ControlTagAmbito para mantener el control asociado a cada ámbito, así como se realizó con el generador de código de cada ámbito.

En la utilización del goto:

```
goto        : GOTO TAG { $$.sval = gc.addTerceto("GOTO", ambitoActual + ":" + $2.sval,"");
                         this.ts.addAtributo($2.sval,AccionSemantica.USO,"nombre etiqueta");
                         this.tags.get(tags.size()-1).huboGoto(this.ambitoActual+":"+$2.sval);
                       }
```

Se crea el terceto, se añade la entrada de USO a la tabla de simbolos
Y se llama al método de huboGoto, que si no se encuentra la etiqueta en el mapa del ControlTagAmbito, se la crea y se la pone en falso.

En la declaración:

```
TAG { if(esEmbebido($1.sval)){
  ErrorHandler.addErrorSemantico("No se puede declarar una etiqueta que tenga tipos embebidos", lex.getLineaInicial());
  } else {
      estructurasSintacticas("Se declaro una etiqueta goto, en linea: " + lex.getLineaInicial());
      String etiquetaAmbito=ambitoActual+":"+$1.sval;
      this.ts.addClave(etiquetaAmbito);
      this.ts.addAtributo(etiquetaAmbito,AccionSemantica.USO,"nombre de tag");
      this.tags.get(tags.size()-1).declaracionTag(etiquetaAmbito, lex.getLineaInicial());
      gc.addTerceto("TAG", etiquetaAmbito, "-");
  }
```

Se crea la entrada en la tabla de simbolos, se añade el USO y se llama a declaracionTag en ControlTagAmbito, que si no encuentra, debido a la restricción incorporada tira un error, y si la encuentra la setea true.

Luego, al cambiar de ámbito (al salir de una función) o cuando termina la regla del programa, se llama a tagsValidos del ámbito actual para controlar la declaración de las tags:

tags.get(this.tags.size()-1).tagsValidos(lex.getLineaInicial());

Y luego se elimina el último elemento del arraylist de ControlTagAmbito.

Tripla

Para las triplas se agregó un terceto de operando “ACCESOTRIPLA” que tiene el id y la posición a acceder de la tripla. El chequeo del acceso a las triplas se realiza en ejecución ya que el índice puede calcularse con resultados de operaciones aritmeticas, funciones o accesos.

Chequeos de tipo del parámetro en invocaciones a funciones

Se permite que el tipo del parámetro real sea diferente al del parámetro formal, siempre que se anteponga al nombre del parámetro real el tipo del parámetro formal, para convertirlo al momento de la invocación.

Para esto añadimos un terceto de conversión TODOUBLE/TOULONGINT según sea el caso. Luego, se chequean los tipos utilizando el

```
param_real      : expresion_matematica {$$.sval = $1.sval; gc.checkParamReal($1.sval, lex.getLineaInicial(), this.ts, ambitoActual,idFuncion);}
                | tipo ID {
                        String id = gc.checkDeclaracion($2.sval, lex.getLineaInicial(), this.ts, this.ambitoActual);
                        String tipoId = this.ts.getAtributo(id, AccionSemantica.TIPO);
                if ((!tipoId.equals("ulongint")) && (!tipoId.equals("double"))){
                        ErrorHandler.addErrorSemantico("no se puede castear una tripla, ya que no se puede pasar como parametro", lex.getLineaInicial());
                }

if(!tipoId.equals($1.sval)){$$.sval = gc.addTerceto("TO".concat($1.sval), gc.checkDeclaracion($2.sval, lex.getLineaInicial(), this.ts, this.ambitoActual),
"");} else { ErrorHandler.addWarningSemantico("Se intenta realizar una conversion innecesaria. No se realizará.", lex.getLineaInicial()); $$.sval = $2.sval;
gc.checkParamReal($2.sval, lex.getLineaInicial(), this.ts, ambitoActual,idFuncion);}

if(!this.ts.getAtributo(this.ts.getAtributo(idFuncion, AccionSemantica.PARAMETRO), AccionSemantica.TIPO).equals($1.sval)){ ErrorHandler.addErrorSemantico("El
tipo del parametro real no coincide con el tipo del parametro formal.", lex.getLineaInicial());}}
                ;
```

Primero chequeamos que el tipo del casteo sea distinto al de la variable original para no generar un terceto de conversión innecesario. A su vez también chequeamos que no se

pueda pasar por parámetro un tipo tripla,una expresión de triplas, ni tampoco se puede castear una tripla a otro tipo.

Luego chequeamos que el tipo del casteo del parámetro formal sea igual al del parámetro real

Chequeos del tipo de retorno de la función

```
retorno     : RET '('expresion_matematica')' {
    this.cantRetornos.set(this.cantRetornos.size()-1, this.cantRetornos.get(this.
    cantRetornos.size()-1) + 1);
    String expresion = gc.checkDeclaracion($3.sval, lex.getLineaInicial(), this.ts,
    this.ambitoActual);
    gc.checkTipoRetorno(expresion, ambitoActual,this.ts,lex.getLineaInicial());
    $$.sval = gc.addTerceto("RET", expresion, "");
    }
    ;
```

Se realiza el chequeo del tipo del retorno de la función, con el metodo checkTipoRetorno:

```
public void checkTipoRetorno(String retorno, String funcion, TablaSimbolos ts, int linea) {
   Pattern pattern = Pattern.compile("\\[(\\d+)\\]");
   Matcher matcher = pattern.matcher(retorno);
   if (matcher.find()) {
      String tipo = this.getTerceto(Integer.parseInt(retorno.substring(1, retorno.length() - 1))).getTipo();
      if (!tipo.equals(ts.getAtributo(funcion, "tipo"))) {
         ErrorHandler.addErrorSemantico("El tipo del retorno es distinto al de la funcion.", linea);
      }
   } else if (!ts.getAtributo(retorno, "tipo").equals(ts.getAtributo(funcion, "tipo"))) {
      ErrorHandler.addErrorSemantico("El tipo del retorno es distinto al de la funcion.", linea);
   }

}
```

Buscamos si hay un terceto, que en tal caso nos fijamos en su tipo.
Si no hay terceto, buscamos el tipo del atributo en la tabla de símbolos.

Octales

Ya que el trabajo estaba dividido en dos partes, no pensamos en la ejecución de los enteros representados en octales hasta ahora.

Lo que hicimos fue que, ya que son una representación, y no un tipo, sólo se utilizan como constantes y eso nos permitió guardarlos como ulongint en la tabla de símbolos en representación decimal.

Esto se realizó en la etapa del análisis léxico, en las acciones semánticas al detectar un octal lo que hicimos fue reemplazar el concatenado actual por su representación entera en base decimal.

```java
public void checkOctal(AnalizadorLexico analizador) {
    String number = analizador.getConcatActual();
    BigInteger actual = new BigInteger(number, 8);
    if(actual.compareTo(MAX_INT) >= 0){
        analizador.addWarning("Constante Ulongint base 8 fuera de rango, se trunco al maximo representable");
        BigInteger maxOctalInt = new BigInteger("2147483648");
        String octalString = maxOctalInt.toString(8);
        analizador.setConcatActual(maxOctalInt.toString());
    }else {
        analizador.setConcatActual(actual.toString());
    }
}
```

Ya que previamente lo convertíamos en entero base decimal para chequear su rango, lo que hicimos fue guardar el valor entero en el concatenado actual del lexico.

Nos pareció una buena idea realizarlo de esta manera, ya que la idea de utilizar una representación octal que sólo se utiliza en constantes, buscaba que el programador ingresara estas de manera más fácil, y dejando su manejo de manera normal como un entero.

Generación Código Assembler

Se creó una clase GenerarCodigoWasm para lograr tal objetivo.

```
public class GeneradorWasm {
    private TablaSimbolos ts;
    private GeneradorCodigo gc_main;
    private String filePath;
    private String funcionActual;
    private int posicionActual;
    private String ident;
    // Pilas que manejan anidaciones de flujos de control
    private Stack<String> tipoFlujoActual = new Stack<String>();
    private Stack<String> ifs;
    private Stack<String> fors;
    private Stack<String> bloquesGOTO;
    private boolean hayPatternMatching = false;
    //AUXILIARES PARA OPERACIONES DE TRIPLAS
    private String aux1= new String(""); //guardo datos de operaciones de triplas
    private String aux2=new String("");
    private String aux3=new String("");
    private String aux1Tripla= new String(""); //guardo el dato de acceder a una tripla
    private String aux2Tripla=new String("");
    private String aux3Tripla=new String("");
    // Estructuras
    private Set<String> accesoTriplas = new HashSet<String>();
    private Set<String> gotos = new HashSet<String>();
    private Map<String, Integer> funcionesId= new HashMap<String,Integer>();
    private ManejadorErroresEjecucion erroresEj = new ManejadorErroresEjecucion();
    // Variables que arman partes del archivo .wta
    private StringBuilder variablesGlobales = new StringBuilder();
    private StringBuilder funciones = new StringBuilder();
    private StringBuilder inicioMain = new StringBuilder();
    private StringBuilder cuerpoMain = new StringBuilder();
    private StringBuilder variablesActual = new StringBuilder();
    private StringBuilder cuerpoActual = new StringBuilder();
```

Se utilizan clases StringBuilder para ir generando el código por secciones y luego unirlos, ya que ciertas variables se reconocen en momentos distintos por lo que no se puede declarar en cualquier lugar. A su vez, esto nos brinda un código más limpio, para lo cual también agregamos una indentación dinámica.

Se utilizan strings auxiliares que realizan el seguimiento de cuáles variables auxiliares están siendo utilizadas para asignar dinámicamente las mismas. Esto es necesario ya que dentro de cada flujo de control webassembly utiliza una pila distinta y se pierden valores. A su vez, cuando se calculan condiciones o se realiza el pattern matching se utilizan hasta 3 variables auxiliares al mismo tiempo.
Todo esto se explica en profundidad en la sección de triplas.

En el parser, luego de generarse el código intermedio se llama a el método traducir.
Primero escribimos los imports necesarios.

```
public void traducir() {
    this.escribir(this.variablesGlobales,"(module");
    this.escribir(this.variablesGlobales,"(import \"console\" \"log\" (func $log (param i32 i32)))");
    this.escribir(this.variablesGlobales,"(import \"js\" \"mem\" (memory 1))");
    this.escribir(this.variablesGlobales, "(import \"env\" \"console_log\" (func $console_log_i32 (param i32)))");
    this.escribir(this.variablesGlobales, "(import \"env\" \"console_log\" (func $console_log_f64 (param f64)))");
    this.escribir(this.variablesGlobales, "(import \"env\" \"exit\" (func $exit))");
    this.cargarVariables();
    this.cargarErrores();
    this.cargarCadmuls();
    this.escribir(this.variablesGlobales, "");
    this.reducirIdentacion();
    this.funcionesId.put("global", 0);
    int contador = 1;
```

Cargamos las variables:

```
    public void cargarVariables() {
        Map<String, Map<String, String>> tabla = this.ts.getTabla();
        for (Map.Entry<String, Map<String, String>> entry : tabla.entrySet()) {
            String key = entry.getKey();
            Map<String, String> val = entry.getValue();
            if("nombre variable".equals(val.get(AccionSemantica.USO))) {
                String tipoVar = val.get(AccionSemantica.TIPO);
                switch(tipoVar) {
                case "ulongint": tipoVar = "i32";break;
                case "double": tipoVar = "f64";break;
                default: tipoVar =
val.get(AccionSemantica.TIPO_BASICO).equals("ulongint")?"i32":"f64";
                    String key1 = key + "V1";
                    this.escribir(this.variablesGlobales,"(global $"+key1.replace(':', 'A')+"
(mut " +tipoVar+")" + "(" + tipoVar + ".const 0))");
                    String key2 = key + "V2";
                    this.escribir(this.variablesGlobales,"(global $"+key2.replace(':', 'A')+"
(mut " +tipoVar+")" + "(" + tipoVar + ".const 0))");
                    key = key + "V3";
                break;
                }
                this.escribir(this.variablesGlobales,"(global $"+key.replace(':', 'A')+" (mut "
+tipoVar+")" + "(" + tipoVar + ".const 0))");
            }
        }
```

Se recorre la tabla de símbolos buscando variables mediante el atributo USO, declarándose todas globales, debido a que el ámbito se controla implícitamente mediante el name mangling.

Luego, se crean todas las variables auxiliares necesarias que serán explicadas en sus respectivas secciones.

```
this.escribir(this.variablesGlobales, "(global $funcionLlamadora (mut i32) (i32.const 0))");
this.escribir(this.variablesGlobales, "(global $AUXOVERFLOW (mut i32) (i32.const 0))");
this.escribir(this.variablesGlobales, "(global $f64auxTripla (mut f64) (f64.const 0))");
this.escribir(this.variablesGlobales, "(global $i32auxTripla (mut i32) (i32.const 0))");
this.escribir(this.variablesGlobales, "(global $f64aux2Tripla (mut f64) (f64.const 0))");
this.escribir(this.variablesGlobales, "(global $i32aux2Tripla (mut i32) (i32.const 0))");
this.escribir(this.variablesGlobales, "(global $f64aux3Tripla (mut f64) (f64.const 0))");
this.escribir(this.variablesGlobales, "(global $i32aux3Tripla (mut i32) (i32.const 0))");
this.escribir(this.variablesGlobales, "(global $AUX1V1i32 (mut i32) (i32.const 0))");
this.escribir(this.variablesGlobales, "(global $AUX1V2i32 (mut i32) (i32.const 0))");
this.escribir(this.variablesGlobales, "(global $AUX1V3i32 (mut i32) (i32.const 0))");
this.escribir(this.variablesGlobales, "(global $AUX2V1i32 (mut i32) (i32.const 0))");
this.escribir(this.variablesGlobales, "(global $AUX2V2i32 (mut i32) (i32.const 0))");
this.escribir(this.variablesGlobales, "(global $AUX2V3i32 (mut i32) (i32.const 0))");
this.escribir(this.variablesGlobales, "(global $AUX1V1f64 (mut f64) (f64.const 0))");
this.escribir(this.variablesGlobales, "(global $AUX1V2f64 (mut f64) (f64.const 0))");
this.escribir(this.variablesGlobales, "(global $AUX1V3f64 (mut f64) (f64.const 0))");
this.escribir(this.variablesGlobales, "(global $AUX2V1f64 (mut f64) (f64.const 0))");
this.escribir(this.variablesGlobales, "(global $AUX2V2f64 (mut f64) (f64.const 0))");
this.escribir(this.variablesGlobales, "(global $AUX2V3f64 (mut f64) (f64.const 0))");
this.escribir(this.variablesGlobales, "(global $AUX3V1f64 (mut f64) (f64.const 0))");
this.escribir(this.variablesGlobales, "(global $AUX3V2f64 (mut f64) (f64.const 0))");
this.escribir(this.variablesGlobales, "(global $AUX3V3f64 (mut f64) (f64.const 0))");
this.escribir(this.variablesGlobales, "(global $AUX3V1i32 (mut i32) (i32.const 0))");
this.escribir(this.variablesGlobales, "(global $AUX3V2i32 (mut i32) (i32.const 0))");
this.escribir(this.variablesGlobales, "(global $AUX3V3i32 (mut i32) (i32.const 0))");
```

Ahora se cargan los errores que ocurren en ejecución:

```
    public void cargarErrores() {
        erroresEj.agregarError("negativo", "\"Error en ejecucion: El resultado de una operacion sin signo dio
negativo.\")");
        erroresEj.agregarError("overflow", "\"Error en ejecucion: El resultado de una suma de enteros genero
overflow.\")");
        erroresEj.agregarError("recursion", "\"Error en ejecucion: se realizo una recursion sobre una funcion.
\")");
        erroresEj.agregarError("rango", "\"Error en ejecucion: indice fuera de rango.\")");
        erroresEj.agregarError("conversionErronea", "\"Error en ejecucion: se intenta realizar una conversion de
flotante negativo a entero sin signo.\")");
        for (String error : erroresEj.getErrores().keySet()) {
            this.escribir(this.variablesGlobales, "(data (i32.const " + erroresEj.getDir(error) + ")" +
erroresEj.getMsj(error));
        }
    }
```

Se creó una clase ManejadorErroresEjecucion que maneja la generación de errores generando la dirección dinámicamente, ya que para guardar strings en webassembly se deben guardar en una posición de memoria y luego para su acceso se requiere cargar la posición y el tamaño.

```java
public class ManejadorErroresEjecucion {
    private int direccionMemoriaActual = 101;
    private Map<String, MensajeError> erroresEjecucion = new HashMap<>();

    public void agregarError(String clave, String mensaje) {
        MensajeError mensajeError = new MensajeError(direccionMemoriaActual, mensaje);
        erroresEjecucion.put(clave, mensajeError);
        direccionMemoriaActual += mensaje.length();
    }

    public Map<String, MensajeError> getErrores() {
        return new HashMap<String, MensajeError>(erroresEjecucion);
    }

    public int getDirMax() {
        return this.direccionMemoriaActual;
    }

    public int getDir(String clave) {
        MensajeError mensajeError = erroresEjecucion.get(clave);
        return (mensajeError != null) ? mensajeError.getDireccionMemoria() : -1;
    }

    public String getMsj(String clave) {
        MensajeError mensajeError = erroresEjecucion.get(clave);
        return (mensajeError != null) ? mensajeError.getMensaje() : null;
    }

    public String getSizeMsj(String clave) {
        return String.valueOf(erroresEjecucion.get(clave).size());
    }
}
```

La misma utiliza una clase MensajeError que guarda el mensaje, el tamaño y la dirección que le otorga el ManejadorErroresEjecucion:

```java
public class MensajeError {
    private int direccionMemoria;
    private String mensaje;
    private int size;

    public MensajeError(int direccionMemoria, String mensaje) {
        this.direccionMemoria = direccionMemoria;
        this.mensaje = mensaje;
        this.size = mensaje.length();
    }

    public int getDireccionMemoria() {
        return direccionMemoria;
    }

    public String getMensaje() {
        return mensaje;
    }

    public int size() {
        return size;
    }
}
```

Luego de cargar los errores, se cargan las cadenas múltiples que también requieren de una posición de memoria y tamaño:

```java
    public void cargarCadmuls() {
        Map<String, Integer> cadmuls = this.ts.getCadenas();
        Integer dirMem = erroresEj.getDirMax();
        for(String key : cadmuls.keySet()) {
            this.escribir(this.variablesGlobales,"(data (i32.const " + dirMem + ") \"" + key.substring("CADMUL:".length()) +
"\")");
            this.ts.setPosicionMemoria(key, dirMem);
            dirMem+= key.length();
        }
    }
```

Tales cadenas de texto comenzarán a partir de la dirección que tenga disponible el ManejadorErroresEjecucion.

Ahora, por cada generador de código de cada función (que guarda el código intermedio de cada una) generamos el código webassembly de las mismas.

```
210         GeneradorCodigo aux = gc_main;
211         Map<String, GeneradorCodigo> funciones = this.ts.getFunciones();
212         for (String funcion : funciones.keySet()) {
213             this.funcionesId.put(funcion, contador);
214             cuerpoActual = new StringBuilder();
215             bloquesGOTO = new Stack<String>();
216             this.funcionActual = funcion;
217             gc_main = funciones.get(funcion);
218             String parametro = this.ts.getAtributo(funcion, AccionSemantica.PARAMETRO);
219             String tipoParam = this.ts.getAtributo(parametro, AccionSemantica.TIPO).equals("double")?
    "f64":"i32";
220             String tipoFuncion = this.ts.getAtributo(funcion, AccionSemantica.TIPO).equals("double")?
    "f64":"i32";
221             parametro = parametro.replace(':', 'A');
222             this.escribir(this.funciones, "( func $"+funcion.replace(':', 'A')  + " (param $"+ parametro  + " " +
    tipoParam+") (result "+ tipoFuncion + ")");
223             this.escribir(variablesGlobales, "(global $"+ parametro +" (mut "+tipoParam+") ("+tipoParam+".const
    0))");
224             this.escribir(cuerpoActual, "local.get $" + parametro);
225             this.escribir(cuerpoActual, "global.set $" + parametro);
226             variablesActual = new StringBuilder();
227             this.escribir(variablesActual,"(local $" + funcion.replace(':', 'A') +"retorno " + tipoFuncion +
    ")");
228             this.aumentarIdentacion();
```

Guardamos el generadorCodigo main (código intermedio main) en una variable auxiliar para ir cambiando el código intermedio que se está traduciendo, ya que toda la clase utiliza el atributo gc_main para obtener los tercetos.
Reseteamos los bloques que se están escribiendo de cada función antes de comenzar a escribir.

Se obtiene el parámetro, su tipo y el tipo de la función para poder escribir el encabezado de la misma en webassembly.

Dado que las funciones en webassembly crean una variable local implícitamente, y nosotros tenemos creada una variable global previamente por la tabla de símbolos, hacemos un get de la variable local (que se recibe por parámetro en el llamado a la función) y lo seteamos en la variable global para luego poder utilizarla normalmente.

Además, se crea una variable de retorno para asignar un retorno default (con un 0) en el caso en que el retorno dependa de un flujo de control y no se ejecute.

Ahora, se realiza el chequeo de recursión

```
this.escribir(cuerpoActual, "i32.const " + contador);
this.escribir(cuerpoActual, "global.get $funcionLlamadora");
this.escribir(cuerpoActual, "i32.eq");
this.escribir(cuerpoActual, "(if");
this.aumentarIdentacion();
this.escribir(cuerpoActual,"(then");
this.aumentarIdentacion();
this.escribir(cuerpoActual, "i32.const "+erroresEj.getDir("recursion"));
this.escribir(cuerpoActual, "i32.const "+erroresEj.getSizeMsj("recursion"));
this.escribir(cuerpoActual,"call $log");
this.escribir(cuerpoActual, "call $exit");
this.reducirIdentacion();
this.escribir(cuerpoActual,")");
this.reducirIdentacion();
this.escribir(cuerpoActual,")");
```

Para controlar la recursión creamos un mapa asociando un ID a cada función

```
private Map<String, Integer> funcionesId= new HashMap<String,Integer>();
```

El mecanismo consiste en que al entrar a una función se guarde el ID de la misma, se cargue el ID de la función que la llama (el cual se setea antes de hacer el llamado a la función) y se chequea que no sea el mismo.

Mecanismo para la generación

Ya que WebAssembly funciona con una pila de la cual obtiene los operandos, siempre que se realice alguna operación se deben cargar primero en la pila.

Para dicho objetivo, se creó un método “obtenerGets”

```
    private void obtenerGets(Terceto t, boolean quieroIzquierdo) {
        int pos = this.posicionActual;
        String op1 = t.getOp1();
        String op2 = t.getOp2();
        Pattern pattern = Pattern.compile("\\[(\\d+)\\]");
        Matcher matcher1 = pattern.matcher(op1);
        Matcher matcher2 = pattern.matcher(op2);
        boolean find1 = matcher1.find();
        boolean find2 = matcher2.find();
```

```
        if(quieroIzquierdo && !find1) {
            if(!op1.equals("")) {
                if(op1.matches("^[0-9].*") || op1.matches("^-.*")) {

                 this.escribir(cuerpoActual,this.ts.getAtributo(op1, AccionSemantica.TIPO).equals("double") ? "f64.const "+op1 :
"i32.const "+op1);
                } else {
                    this.escribir(cuerpoActual,"global.get $"+op1.replace(':', 'A'));
                }
            }
        }else if(find1){ //o una expreion comun o un acceso a tripla
            int indiceTerceto = Integer.parseInt(matcher1.group(1)); //el terceto posta
            Terceto t_op1 = this.gc_main.getTerceto(indiceTerceto);
            if(this.posicionActual < indiceTerceto) {
                this.posicionActual = indiceTerceto;
                this.ejecutarTraduccion(t_op1);
            }else if(this.posicionActual > indiceTerceto && !t_op1.isHecho()) {
                this.posicionActual = indiceTerceto;
                this.ejecutarTraduccion(t_op1);
            }
            if(t_op1.getOperador().equals("ACCESOTRIPLE")) {
                String AuxTripla;
                String tipoOp1 = t_op1.getTipo().equals("ulongint")?"i32":"f64";
                if(aux1Tripla.equals(t.getOp1()))
                    AuxTripla="auxTripla";
                else if(aux2Tripla.equals(t.getOp1()))
                    AuxTripla="aux2Tripla";
                else
                    AuxTripla="aux3Tripla";
                this.escribir(cuerpoActual, "global.get $"+tipoOp1+AuxTripla);
                if(AuxTripla.equals("auxTripla"))
                    this.aux1Tripla="";
                else if (AuxTripla.equals("aux2Tripla"))
                    this.aux2Tripla="";
                else
                    this.aux3Tripla="";
            }

        }
```

Obtiene los operandos requeridos de la manera adecuada según si es un terceto, una constante, una variable o un acceso a tripla. En caso de ser una invocación a función o resultado de una expresión matemática, ya se encuentra en la pila por lo que no se debe obtener ningún operando.

Lo mismo para el segundo operando:

```
        if(!find2) {
            if(!op2.equals("")) {
                if(op2.matches("^[0-9].*") || op2.matches("^-.*")) {
                    this.escribir(cuerpoActual,this.ts.getAtributo(op2, AccionSemantica.TIPO).equals("double") ? "f64.const
"+t.getOp2() : "i32.const "+t.getOp2());
                } else {
                    this.escribir(cuerpoActual,"global.get $"+op2.replace(':', 'A'));
                }
            }
        } else {
            int indiceTerceto = Integer.parseInt(matcher2.group(1));
            Terceto t_op2 = this.gc_main.getTerceto(indiceTerceto);
            if(pos < indiceTerceto) {
                this.posicionActual = indiceTerceto;
                this.ejecutarTraduccion(t_op2);

            }else if(pos > indiceTerceto && !t_op2.isHecho()) {
                this.posicionActual = indiceTerceto;
                this.ejecutarTraduccion(t_op2);
            }
            if(t_op2.getOperador().equals("ACCESOTRIPLE")) {
                String AuxTripla;
                String tipoOp2 = t_op2.getTipo().equals("ulongint")?"i32":"f64";
                if(aux1Tripla.equals(t.getOp2()))
                    AuxTripla="auxTripla";
                else if(aux2Tripla.equals(t.getOp2()))
                    AuxTripla="aux2Tripla";
                else
                    AuxTripla="aux3Tripla";
                this.escribir(cuerpoActual, "global.get $"+tipoOp2+AuxTripla);
                if(AuxTripla.equals("auxTripla"))
                    this.aux1Tripla="";
                else if (AuxTripla.equals("aux2Tripla"))
                    this.aux2Tripla="";
                else
                    this.aux3Tripla="";
            }
        }
        this.posicionActual = pos;
```

En ambos casos, si el operando es un terceto que aún no se tradujo, se debe realizar la respectiva traducción. Esto se debe a que es necesario para el Pattern Matching, donde se realizan comparaciones entre tercetos en posiciones no consecutivas.
Para esto, debemos guardar la posición actual en un auxiliar para cambiarla y ejecutar la traducción de los tercetos de manera recursiva.

En ambos casos, para el acceso a tripla se debe obtener el resultado del acceso desde el auxiliar donde se encuentre el mismo. Luego de obtener tal resultado, se libera el auxiliar correspondiente.

Chequeo de errores en Ejecución

Overflow

Se utiliza el mismo mecanismo para detectar el overflow de suma en ulongint, y el de resta con resultado negativo.
El mecanismo es el mismo, ya que en webassembly se utiliza complemento a la base 2 por lo que el overflow convierte el resultado en negativo. Por esta razón, se utiliza un sólo método pasando el tipo de error por parámetro:

```
private void checkOverflow(String get, String error) {
    this.escribir(cuerpoActual, get);
    this.escribir(cuerpoActual, "i32.const 0");
    this.escribir(cuerpoActual, "i32.lt_s");
    this.escribir(cuerpoActual,"(if");
    this.aumentarIdentacion();
    this.escribir(cuerpoActual,"(then");
    this.aumentarIdentacion();
    this.escribir(cuerpoActual, "i32.const "+erroresEj.getDir(error));
    this.escribir(cuerpoActual, "i32.const "+erroresEj.getSizeMsj(error));
    this.escribir(cuerpoActual, "call $log");
    this.escribir(cuerpoActual, "call $exit");
    this.reducirIdentacion();
    this.escribir(cuerpoActual,")");
    this.reducirIdentacion();
    this.escribir(cuerpoActual,")");
}
```

Conversión Errónea

Al convertir de double a ulongint, puede que se convierta en un entero negativo. Razón por la cual, si bien no se nos pide el chequeo, lo realizamos.

```
private void toUlongint(Terceto t) {
    this.obtenerGets(t);
    this.escribir(cuerpoActual, "f64.const 0");
    this.escribir(cuerpoActual, "f64.lt");
    this.escribir(cuerpoActual, "(if");
    this.aumentarIdentacion();
    this.escribir(cuerpoActual, "(then");
    this.aumentarIdentacion();
    this.escribir(cuerpoActual, "i32.const "+erroresEj.getDir("conversionErronea"));
    this.escribir(cuerpoActual, "i32.const "+erroresEj.getSizeMsj("conversionErronea"));
    this.escribir(cuerpoActual, "call $log");
    this.escribir(cuerpoActual, "call $exit");
    this.reducirIdentacion();
    this.escribir(cuerpoActual, ")");
    this.reducirIdentacion();
    this.escribir(cuerpoActual, ")");
    this.obtenerGets(t);
    this.escribir(cuerpoActual,"i32.trunc_f64_u");
}
```

Indice fuera de rango

Este chequeo tampoco se pide pero es necesario ya que tenemos una variable por elemento de la tripla y genera errores al asignar a una tripla fuera de rango o al acceder a la misma, ya que no existe variables para ello.
El mecanismo consiste en fijarse tanto para la asignación como para el acceso, que el índice que se quiera acceder sea 1, 2 o 3. De lo contrario, se genera un error.

En caso de que sea una asignación, se realiza la asignación correspondiente sobre la variable auxiliar que esté libre (dado que al entrar al if, webassembly usa otra pila y necesitamos una variable externa).
En caso de ser un acceso se guarda el elemento correspondiente sobre la variable auxiliar que esté libre, por la misma razón.

Ya que el índice de acceso puede generarse en ejecución, decidimos realizar el chequeo en ejecución para el índice tanto para las constantes como para los otros posibles índices (operacion aritmetica, invocación a función, acceso a un elemento de una tripla)

Operaciones aritméticas

La regla general para las operaciones aritméticas, es fijarse el tipo de la variable de la operación a realizar (i32, f64), se obtienen los operandos y se genera el código de la operación.

```
private void suma(Terceto t) {
    String tipo = t.getTipo();
    if(tipo.equals("double")) {
        this.obtenerGets(t);
        this.escribir(cuerpoActual,"f64.add");
    } else if(tipo.equals("ulongint")) {
        this.obtenerGets(t);
        this.escribir(cuerpoActual,"i32.add");
        this.escribir(cuerpoActual, "global.set $AUXOVERFLOW");
        this.checkOverflow("global.get $AUXOVERFLOW", "overflow");
        this.escribir(cuerpoActual, "global.get $AUXOVERFLOW");
    } else {
        this.aritmeticaTripla(t, ".add");
    }
}
```

Para la suma y para la resta, se llama al checkOverflow.
Ya que el resultado aritmético quedó en la pila, se setea el auxiliar del overflow y es pasado por parámetro al checkOverflow.
Luego de que se chequee, lo obtenemos de nuevo desde la variable auxiliar, ya que se pierde en el chequeo y la operación correspondiente lo necesita dejar en la pila.

En caso de ser una tripla, se realiza la operación elemento a elemento y su chequeo.
En caso de ser un acceso a tripla, se obtiene o setea la variable auxiliar.
Ambos casos se tienen en cuenta en el método de aritmética asociada a triplas:

```java
private void aritmeticaTripla(Terceto t, String sufijoOperacion) {
    String op1 = t.getOp1();
    String op2 = t.getOp2().replace(':', 'A');
    Pattern pattern = Pattern.compile("\\[(\\d+)\\]");
    Matcher matcher1 = pattern.matcher(op1);
    Matcher matcher2 = pattern.matcher(op2);
    boolean find1 = matcher1.find();
    boolean find2 = matcher2.find();
    int pos = this.posicionActual;
    String tipo;
    if(!find1) {
        tipo = this.ts.getAtributo(op1, AccionSemantica.TIPO_BASICO).equals("ulongint")?"i32":"f64";
    }else {
        String tipoTripla = t.getTipo();
        tipo = ts.getAtributo(tipoTripla,"tipotripla").equals("ulongint")?"i32":"f64";
    }
    String operacion = tipo + sufijoOperacion;
    String aux = "";
    String aux2 = "";
    op1 = op1.replace(':', 'A');
```

Primero se obtiene el tipo de distinta forma según sea un acceso a tripla o una tripla.

```
if(find1 && !find2) {
    int indiceTerceto = Integer.parseInt(matcher1.group(1)); //operando1
    Terceto t_op1 = gc_main.getTerceto(indiceTerceto);
    if(this.posicionActual < indiceTerceto) {
        this.posicionActual = indiceTerceto;
        this.ejecutarTraduccion(t_op1);
    }else if(this.posicionActual > indiceTerceto && !t_op1.isHecho()) {
        this.posicionActual = indiceTerceto;
        this.ejecutarTraduccion(t_op1);
    }
    this.posicionActual=pos;

    if(this.aux1.equals(t.getOp1()))
        aux="AUX1";
    else if(this.aux2.equals(t.getOp1()))
        aux="AUX2";
    else
        aux="AUX3";
    this.escribir(cuerpoActual, "global.get $"+aux+"V1"+tipo);
    this.escribir(cuerpoActual, "global.get $"+op2+"V1");
    this.escribir(cuerpoActual, operacion);

    this.escribir(cuerpoActual, "global.set $"+aux+"V1"+tipo);

    this.escribir(cuerpoActual, "global.get $"+aux+"V2"+tipo);
    this.escribir(cuerpoActual, "global.get $"+op2+"V2");
    this.escribir(cuerpoActual, operacion);
    this.escribir(cuerpoActual, "global.set $"+aux+"V2"+tipo);

    this.escribir(cuerpoActual, "global.get $"+aux+"V3"+tipo);
    this.escribir(cuerpoActual, "global.get $"+op2+"V3");
    this.escribir(cuerpoActual, operacion);
    this.escribir(cuerpoActual, "global.set $"+aux+"V3"+tipo);
    if(aux.equals("AUX1")) {
        this.aux1="["+this.posicionActual+"]";
    }else if(aux.equals("AUX2")){
        this.aux2="["+this.posicionActual+"]";
    }else {
        this.aux3="["+this.posicionActual+"]";
    }
```

Primero, si se tiene un terceto del lado izquierdo y no del derecho, implica que del lado izquierdo se encuentra en un auxiliar el valor asociado.
Dado que es una tripla, se realiza la operación element-wise (elemento a elemento).

La operación se guarda en el mismo auxiliar que se tenía el valor para aprovechar la memoria. Luego, se actualiza el terceto en el atributo de la clase que mantiene el seguimiento de las variables auxiliares con sus resultados guardados.
De esta manera, se tiene el registro de lo que guarda cada variable auxiliar, para que cuando se tenga un operando ver en cual auxiliar se encuentra.

En caso de que ambos sean tercetos, se utiliza la misma lógica pero se busca en cual auxiliar se encuentra el resultado de cada uno, para luego operar.

```
} else if(find1 && find2) {
    int indiceTerceto = Integer.parseInt(matcher1.group(1)); //operando1
    Terceto t_op1 = gc_main.getTerceto(indiceTerceto);
    if(this.posicionActual < indiceTerceto) {
        this.posicionActual = indiceTerceto;
        this.ejecutarTraduccion(t_op1);
    }else if(this.posicionActual > indiceTerceto && !t_op1.isHecho()) {
        this.posicionActual = indiceTerceto;
        this.ejecutarTraduccion(t_op1);
    }
    this.posicionActual=pos;

    int indiceTerceto2 = Integer.parseInt(matcher2.group(1)); //operando2
    Terceto t_op2 = gc_main.getTerceto(indiceTerceto2);
    if(this.posicionActual < indiceTerceto2) {
        this.posicionActual = indiceTerceto2;
        this.ejecutarTraduccion(t_op2);
    }else if(this.posicionActual > indiceTerceto2 && !t_op2.isHecho()) {
        this.posicionActual = indiceTerceto2;
        this.ejecutarTraduccion(t_op2);
    }
    this.posicionActual=pos;

    if(this.aux1.equals(t.getOp1()))
        aux="AUX1";
    if(this.aux2.equals(t.getOp1()))
        aux="AUX2";
    if(this.aux3.equals(t.getOp1()))
        aux="AUX3";
    if(this.aux1.equals(t.getOp2()))
        aux2="AUX1";
    if(this.aux2.equals(t.getOp2()))
        aux2="AUX2";
    if(this.aux3.equals(t.getOp2()))
        aux2="AUX3";
```

```
    this.escribir(cuerpoActual, "global.get $"+aux+"V1"+tipo);
    this.escribir(cuerpoActual, "global.get $"+aux2+"V1"+tipo);
    this.escribir(cuerpoActual, operacion);
    this.escribir(cuerpoActual, "global.set $"+aux+"V1"+tipo);

    this.escribir(cuerpoActual, "global.get $"+aux+"V2"+tipo);
    this.escribir(cuerpoActual, "global.get $"+aux2+"V2"+tipo);
    this.escribir(cuerpoActual, operacion);
    this.escribir(cuerpoActual, "global.set $"+aux+"V2"+tipo);

    this.escribir(cuerpoActual, "global.get $"+aux+"V3"+tipo);
    this.escribir(cuerpoActual, "global.get $"+aux2+"V3"+tipo);
    this.escribir(cuerpoActual, operacion);
    this.escribir(cuerpoActual, "global.set $"+aux+"V3"+tipo);
    if(aux.equals("AUX1"))
        this.aux1="["+this.posicionActual+"]";
    if(aux.equals("AUX2"))
        this.aux2="["+this.posicionActual+"]";
    if(aux.equals("AUX3"))
        this.aux3="["+this.posicionActual+"]";
    if(aux2.equals("AUX1"))
        this.aux1=new String("");
    if(aux2.equals("AUX2"))
        this.aux2=new String("");
    if(aux2.equals("AUX3"))
        this.aux3=new String("");
```

Al finalizar la operación, se libera el registro en el cual se encontraba el resultado, y también se actualiza cual resultado posee ahora.

La misma lógica se sigue para cuando el operando 2 es un terceto y el primero no lo es.

```
} else if(!find1 && find2){

    int indiceTerceto2 = Integer.parseInt(matcher2.group(1)); //operando1
    Terceto t_op2 = gc_main.getTerceto(indiceTerceto2);
    if(this.posicionActual < indiceTerceto2) {
        this.posicionActual = indiceTerceto2;
        this.ejecutarTraduccion(t_op2);
    }else if(this.posicionActual > indiceTerceto2 && !t_op2.isHecho()) {
        this.posicionActual = indiceTerceto2;
        this.ejecutarTraduccion(t_op2);
    }
    this.posicionActual=pos;

    if(this.aux1.equals(t.getOp2()))
        aux="AUX1";
    else if(this.aux2.equals(t.getOp2()))
        aux="AUX2";
    else
        aux="AUX3";
    this.escribir(cuerpoActual, "global.get $"+aux+"V1"+tipo);
    this.escribir(cuerpoActual, "global.get $"+op1+"V1");
    this.escribir(cuerpoActual, operacion);

    this.escribir(cuerpoActual, "global.set $"+aux+"V1"+tipo);

    this.escribir(cuerpoActual, "global.get $"+aux+"V2"+tipo);
    this.escribir(cuerpoActual, "global.get $"+op1+"V2");
    this.escribir(cuerpoActual, operacion);
    this.escribir(cuerpoActual, "global.set $"+aux+"V2"+tipo);

    this.escribir(cuerpoActual, "global.get $"+aux+"V3"+tipo);
    this.escribir(cuerpoActual, "global.get $"+op1+"V3");
    this.escribir(cuerpoActual, operacion);
    this.escribir(cuerpoActual, "global.set $"+aux+"V3"+tipo);
    if(aux.equals("AUX1")) {
        this.aux1="["+this.posicionActual+"]";
    }else if(aux.equals("AUX2")){
        this.aux2="["+this.posicionActual+"]";
    }else {
        this.aux3="["+this.posicionActual+"]";
    }
```

En caso de que ambos sean triplas, se realiza la operación elemento a elemento.

```
} else {

    if(this.aux1.equals(new String("")) ) {
        aux = "AUX1";
    }else if(this.aux2.equals(new String("")) ) {
        aux = "AUX2";
    }else {
        aux = "AUX3";
    }

    this.escribir(cuerpoActual, "global.get $"+op1+"V1");
    this.escribir(cuerpoActual, "global.get $"+op2+"V1");
    this.escribir(cuerpoActual, operacion);
    this.escribir(cuerpoActual, "global.set $"+aux+"V1"+tipo);

    this.escribir(cuerpoActual, "global.get $"+op1+"V2");
    this.escribir(cuerpoActual, "global.get $"+op2+"V2");
    this.escribir(cuerpoActual, operacion);
    this.escribir(cuerpoActual, "global.set $"+aux+"V2"+tipo);

    this.escribir(cuerpoActual, "global.get $"+op1+"V3");
    this.escribir(cuerpoActual, "global.get $"+op2+"V3");
    this.escribir(cuerpoActual, operacion);
    this.escribir(cuerpoActual, "global.set $"+aux+"V3"+tipo);
    if(aux.equals("AUX1")) {
        this.aux1="["+this.posicionActual+"]";
    }else if(aux.equals("AUX2")){
        this.aux2="["+this.posicionActual+"]";
    }else {
        this.aux3="["+this.posicionActual+"]";
    }
}
```

Se guarda en el auxiliar disponible y se actualiza el nuevo.

En caso de que la operación haya sido de tipo ulongint

```
if(tipo.equals("ulongint") ) {
    if(sufijoOperacion.equals("add")) {
        this.checkOverflow("global.get $"+aux+"V1"+tipo, "overflow");
        this.checkOverflow("global.get $"+aux+"V2"+tipo, "overflow");
        this.checkOverflow("global.get $"+aux+"V3"+tipo, "overflow");
    }else if(sufijoOperacion.equals("sub")) {
        this.checkOverflow("global.get $"+aux+"V1"+tipo, "negativo");
        this.checkOverflow("global.get $"+aux+"V2"+tipo, "negativo");
        this.checkOverflow("global.get $"+aux+"V3"+tipo, "negativo");
    }

}
```

Se realiza el chequeo de overflow de suma o resta según corresponda.

Asignaciones

```
private void asignacion(Terceto t) {
    String tipo = t.getTipo();
    String op2 = t.getOp2();
    String op1 = t.getOp1();
    Pattern pattern = Pattern.compile("\\[(\\d+)\\]");
    Matcher matcher2 = pattern.matcher(op2);
    Matcher matcher1 = pattern.matcher(op1);
    String aux="";
    boolean findOp2 = matcher2.find();
    boolean findOp1 = matcher1.find();
    if(!tipo.equals("ulongint") && !tipo.equals("double")) {
        tipo = this.ts.getAtributo(t.getOp1(), AccionSemantica.TIPO_BASICO).equals("ulongint")?"i32":"f64";
        if(findOp2) {
            if(this.aux1.equals(op2))
                aux="AUX1";
            else
                aux="AUX2";
            this.escribir(cuerpoActual, "global.get $"+aux+"V1"+tipo);
            this.escribir(cuerpoActual, "global.set $" + op1.replace(':', 'A') + "V1");
            this.escribir(cuerpoActual, "global.get $"+aux+"V2"+tipo);
            this.escribir(cuerpoActual, "global.set $" + op1.replace(':', 'A') + "V2");
            this.escribir(cuerpoActual, "global.get $"+aux+"V3"+tipo);
            this.escribir(cuerpoActual, "global.set $" + op1.replace(':', 'A') + "V3");
            if(aux.equals("AUX1")) {
                this.aux1= new String("");
            }else {
                this.aux2= new String("");
            }

        }else {
            this.escribir(cuerpoActual, "global.get $" + op2.replace(':', 'A') + "V1");
            this.escribir(cuerpoActual, "global.set $" + op1.replace(':', 'A') + "V1");
            this.escribir(cuerpoActual, "global.get $" + op2.replace(':', 'A') + "V2");
            this.escribir(cuerpoActual, "global.set $" + op1.replace(':', 'A') + "V2");
            this.escribir(cuerpoActual, "global.get $" + op2.replace(':', 'A') + "V3");
            this.escribir(cuerpoActual, "global.set $" + op1.replace(':', 'A') + "V3");
        }
    }else {
```

El primer caso de la asignación es que sea una tripla. Si el operando 2 es un terceto, quiere decir que se realizó una operación aritmética entre triplas (dado que a una tripla solo se puede asignar una tripla) y su resultado está en algún registro auxiliar.
Se realiza la asignación elemento a elemento y se libera el registro asociado a la operación.

Si la asignación era de tipos primitivos, se realiza lo siguiente:

```
}else {
    String loQueAsigno = "";
    String AuxTripla="";
    if(findOp2) {
        String  posicionTercetoOP2 = t.getOp2().substring(1,t.getOp2().length()-1);
        Terceto tOp2 = gc_main.getTerceto(Integer.parseInt(posicionTercetoOP2));
        String operadorOp2 = tOp2.getOperador();
        String tipoOp2 = tOp2.getTipo().equals("ulongint")?"i32":"f64";
        if(operadorOp2.equals("ACCESOTRIPLE")) {
            if(aux1Tripla.equals(t.getOp2()))
                AuxTripla="auxTripla";
            else if(aux2Tripla.equals(t.getOp2()))
                AuxTripla="aux2Tripla";
            else
                AuxTripla="aux3Tripla";
            loQueAsigno = "global.get $"+tipoOp2+AuxTripla;
        }
        else if(findOp1) {
            if(aux1Tripla.equals("")) {
                AuxTripla="auxTripla";
                aux1Tripla=t.getOp2();
            }else if(aux2Tripla.equals("")) {
                AuxTripla="aux2Tripla";
                aux2Tripla=t.getOp2();
            }else {
                AuxTripla="aux3Tripla";
                aux3Tripla=t.getOp2();
            }
            loQueAsigno = "global.get $"+tipoOp2+AuxTripla;
            this.escribir(cuerpoActual,"global.set $"+tipoOp2+AuxTripla);
        }
    }else {
```

En caso de que lo asignado provenga de un terceto, si es un acceso a tripla se lo obtiene de la variable auxiliar donde se encuentre.
Si no era un acceso a tripla y el primer operando es un terceto, quiere decir que se está asignando algo a un acceso a tripla, por lo que se setea la variable auxiliar que contendrá el acceso a la tripla a la que se le va a asignar algo.

Para el resto de los casos en que el segundo operando es un terceto, será una invocación a función o un resultado de una operación aritmética, por lo que ya estará en la pila.

Si el segundo operando no era un terceto:

```
}else {
    String tipoOp2 = this.ts.getAtributo(op2, AccionSemantica.TIPO).equals("ulongint")?"i32":"f64";
    if(this.ts.getAtributo(op2, AccionSemantica.USO).equals("nombre variable")){
        loQueAsigno = "global.get $"+op2.replace(':', 'A');
    }else{
        loQueAsigno =  tipoOp2+".const "+op2;
    }
}
```

Sea una variable o una constante, se la pone en la pila de su manera correspondiente.

Finalmente, si del lado izquierdo se encuentra un terceto, quiere decir que la asignación es a un elemento de la tripla.

Ya que se tiene el índice de acceso, se lo obtiene y se chequea a cual índice se quiere acceder.
De esta manera, al encontrar el índice se setea el valor que se quiere asignar al elemento.
En caso de no encontrarlo, se avisa de un error de índice fuera de rango.

```
if(findOp1) { //asignacion a tripla
    String  posicionTerceto = t.getOp1().substring(1,t.getOp1().length()-1);
    Terceto asignacion = gc_main.getTerceto(Integer.parseInt(posicionTerceto));
    String IdAsignacion =asignacion.getOp1().replace(':', 'A');
    this.escribir(cuerpoActual, "global.get $accesoAsig"+IdAsignacion);
    this.escribir(cuerpoActual, "i32.const 1");
    this.escribir(cuerpoActual, "i32.eq");
    this.escribir(cuerpoActual, "(if");
    this.aumentarIdentacion();
    this.escribir(cuerpoActual, "(then");
    this.aumentarIdentacion();
    this.escribir(cuerpoActual, loQueAsigno);
    this.escribir(cuerpoActual, "global.set $"+IdAsignacion+"V1");
    this.reducirIdentacion();
    this.escribir(cuerpoActual, ")");
    this.escribir(cuerpoActual, "(else");
    this.aumentarIdentacion();
    this.escribir(cuerpoActual, "global.get $accesoAsig"+IdAsignacion);
    this.escribir(cuerpoActual, "i32.const 2");
    this.escribir(cuerpoActual, "i32.eq");
    this.escribir(cuerpoActual, "(if");
    this.aumentarIdentacion();
    this.escribir(cuerpoActual, "(then");
    this.aumentarIdentacion();
    this.escribir(cuerpoActual, loQueAsigno);
    this.escribir(cuerpoActual, "global.set $"+IdAsignacion+"V2");
    this.reducirIdentacion();
    this.escribir(cuerpoActual, ")");
    this.escribir(cuerpoActual, "(else");
    this.aumentarIdentacion();
    this.escribir(cuerpoActual, "global.get $accesoAsig"+IdAsignacion);
    this.escribir(cuerpoActual, "i32.const 3");
    this.escribir(cuerpoActual, "i32.eq");
    this.escribir(cuerpoActual, "(if");
    this.aumentarIdentacion();
    this.escribir(cuerpoActual, "(then");
```

```
this.aumentarIdentacion();
this.escribir(cuerpoActual, loQueAsigno);
this.escribir(cuerpoActual, "global.set $"+IdAsignacion+"V3");
this.reducirIdentacion();
this.escribir(cuerpoActual, ")");
this.escribir(cuerpoActual, "(else");
this.aumentarIdentacion();
this.escribir(cuerpoActual, "i32.const "+erroresEj.getDir("rango"));
this.escribir(cuerpoActual, "i32.const "+erroresEj.getSizeMsj("rango"));
this.escribir(cuerpoActual, "call $log");
this.escribir(cuerpoActual, "call $exit");
this.reducirIdentacion();
this.escribir(cuerpoActual, ")");
this.reducirIdentacion();
this.escribir(cuerpoActual, ")");
this.reducirIdentacion();
this.escribir(cuerpoActual, ")");
this.reducirIdentacion();
this.escribir(cuerpoActual, ")");
this.reducirIdentacion();
this.escribir(cuerpoActual, ")");
this.reducirIdentacion();
this.escribir(cuerpoActual, ")");
if(AuxTripla.equals("auxTripla"))
    this.aux1Tripla="";
else if (AuxTripla.equals("aux2Tripla"))
    this.aux2Tripla="";
else
    this.aux3Tripla="";
```

Luego, se libera el auxiliar que contenía el acceso a tripla.
En caso que del lado izquierda fuera un ID, se obtiene el operando del lado derecho y se lo setea.

```
}else { //id
    this.obtenerGets(t, false);
    this.escribir(cuerpoActual,"global.set $" + t.getOp1().replace(':', 'A'));
}
```

Asignación a Triplas

Al momento de saber que se va a realizar una asignación a un elemento de la tripla, si es una variable o una constante se la obtiene.

```
private void asigTripla(Terceto t) {
    String op1 = t.getOp1();
    String op2 = t.getOp2();
    Pattern pattern = Pattern.compile("\\[(\\d+)\\]");

    Matcher matcher1 = pattern.matcher(op2);
    boolean find = matcher1.find();
    if(!find) {
        if(op2.matches("^[0-9].*")) {
            this.escribir(cuerpoActual, "i32.const "+op2);
        }else {
            this.escribir(cuerpoActual,"global.get $"+op2.replace(':', 'A'));
        }
    }
```

En caso de que el operando derecho un terceto y sea de accesotriple, se lo debe obtener de la variable auxiliar en la que se encuentre alojado, siguiendo la misma lógica utilizada en otros métodos:

```
if(tercetoOp2.getOperador().equals("ACCESOTRIPLE")) {
    if(aux1Tripla.equals(t.getOp2()))
        AuxTripla="auxTripla";
    else if(aux2Tripla.equals(t.getOp2()))
        AuxTripla="aux2Tripla";
    else
        AuxTripla="aux3Tripla";
    this.escribir(cuerpoActual, "global.get $"+tipoOp2+AuxTripla);
    if(AuxTripla.equals("auxTripla"))
        this.aux1Tripla="";
    else if (AuxTripla.equals("aux2Tripla"))
        this.aux2Tripla="";
    else
        this.aux3Tripla="";
}
```

Si el operando derecho fuera terceto pero no de un acceso a tripla, se sabe que es una invocación a función o una expresión matemática, por lo que ya se encuentra en la pila.

Luego, se debe utilizar un SET que contiene todas las variables de accesoAsig y acceso generadas hasta el momento para poder declarar las mismas.

```
if(!this.accesoTriplas.contains(op1)) {
    this.escribir(variablesGlobales, "(global $accesoAsig"+op1.replace(':', 'A')+ " (mut i32) (i32.const 1))");
    this.escribir(variablesGlobales, "(global $acceso"+op1.replace(':', 'A')+  " (mut i32) (i32.const 1))");
    this.accesoTriplas.add(op1);
}
this.escribir(cuerpoActual, "global.set $accesoAsig"+op1.replace(':', 'A'));
```

Estas son necesarias debido a que cuando se quiere acceder a una tripla, para asignación o para utilizar su valor, se debe chequear el índice y eso se realiza con un flujo de control el cual utiliza una pila aparte.

Flujos de control

Ya que tanto los IF como los FOR se pueden anidar, se deberá llevar un control del tipo del flujo actual en el que se encuentra el programa, mediante una pila.
Esto se utiliza para interpretar de la manera adecuada las bifurcaciones generadas en los tercetos, ya que tiene distinta semántica según el flujo en el que se encuentre.

Selección

Al encontrarse una bifurcación por falso

```
 private void bifurcacionPorFalso(Terceto t) {
     boolean esIf = tipoFlujoActual.peek().equals("IF");
     if(esIf) {
         this.escribir(cuerpoActual,"(if");
         this.aumentarIdentacion();
         this.escribir(cuerpoActual,"(then");
         this.aumentarIdentacion();
     }else {
         this.escribir(cuerpoActual,"br_if $"+ "endforA"+this.fors.peek());
     }

 }
```

Se genera el código del IF y el THEN.

Al encontrarse una bifurcación incondicional

```
private void bifurcacionIncondicional(Terceto t) {
    if(this.tipoFlujoActual.peek().equals("IF")) {
        this.reducirIdentacion();
        this.escribir(cuerpoActual,")");
        this.escribir(cuerpoActual,"(else");
        this.aumentarIdentacion();
    }else if(this.tipoFlujoActual.peek().equals("FOR")) {
        this.escribir(cuerpoActual,"br $"+ this.fors.peek());
        this.reducirIdentacion();
        this.escribir(cuerpoActual,"end");
        this.escribir(cuerpoActual,"end");
    }
}
```

Se genera la rama del else.

Al encontrarse una Label

```
private void generarLabel(Terceto t) {
    String op1 = t.getOp1().replace(':', 'A');
    switch(op1) {
    case "else":break;
    case "endif": this.tipoFlujoActual.pop();
                  this.reducirIdentacion();
                  this.escribir(cuerpoActual,")");
                  this.reducirIdentacion();
                  this.escribir(cuerpoActual,")");
                  break;
    case "FIN_IF_SOLO" : this.reducirIdentacion();
                  this.escribir(cuerpoActual,")");
                  this.reducirIdentacion();
                  this.escribir(cuerpoActual,")");
                  break;
    case "endfor": this.tipoFlujoActual.pop(); this.fors.pop(); break;
    default:
        this.tipoFlujoActual.push("FOR");
        this.escribir(cuerpoActual,"block $endforA"+ t.getOp1());
        this.escribir(cuerpoActual,"loop $"+op1);
        this.aumentarIdentacion();
        this.fors.push(op1);
        break;
    }
}
```

De ser una etiqueta con operando endif, se regresa al flujo anterior.
Si la etiqueta representa el fin de un if que no tenía else, se cierran los paréntesis del IF y del THEN.
En caso de que la etiqueta sea un else, no debe realizarse nada. Esto quedó así ya que no debe realizar nada, pero al quitarlo del case se va por el caso de default.

Bucle

Para los bucles se utilizó un bloque de tipo loop, que al realizar un br a su etiqueta se comienza a ejecutar desde el comienzo del bloque.
De esta manera, generamos el loop al encontrarnos una etiqueta de for

```
private void generarLabel(Terceto t) {
    String op1 = t.getOp1().replace(':', 'A');
    switch(op1) {
    case "else":break;
    case "endif": this.tipoFlujoActual.pop();
                this.reducirIdentacion();
                this.escribir(cuerpoActual,")");
                this.reducirIdentacion();
                this.escribir(cuerpoActual,")");
                break;
    case "FIN_IF_SOLO" : this.reducirIdentacion();
                this.escribir(cuerpoActual,")");
                this.reducirIdentacion();
                this.escribir(cuerpoActual,")");
                break;
    case "endfor": this.tipoFlujoActual.pop(); this.fors.pop(); break;
    default:
        this.tipoFlujoActual.push("FOR");
        this.escribir(cuerpoActual,"block $endforA"+ t.getOp1());
        this.escribir(cuerpoActual,"loop $"+op1);
        this.aumentarIdentacion();
        this.fors.push(op1);
        break;
    }
}
```

En tal caso, se actualiza el tipo del flujo, se genera el bloque para salir del for (ya que al saltar a un bloque se salta a su fin) y un bloque de tipo loop.

A su vez, se añade el for actual a una pila para controlar que se utilice el salto al bloque con la etiqueta adecuada para salir del bucle (ya que si se anidan los FOR, se pierde la etiqueta del bloque para salir.)
Si se encuentra un endfor, se regresa al tipo del flujo anterior y se desapila el for actual.

```
private void bifurcacionIncondicional(Terceto t) {
    if(this.tipoFlujoActual.peek().equals("IF")) {
        this.reducirIdentacion();
        this.escribir(cuerpoActual,")");
        this.escribir(cuerpoActual,"(else");
        this.aumentarIdentacion();
    }else if(this.tipoFlujoActual.peek().equals("FOR")) {
        this.escribir(cuerpoActual,"br $"+ this.fors.peek());
        this.reducirIdentacion();
        this.escribir(cuerpoActual,"end");
        this.escribir(cuerpoActual,"end");
    }
}

private void bifurcacionPorFalso(Terceto t) {
    boolean esIf = tipoFlujoActual.peek().equals("IF");
    if(esIf) {
        this.escribir(cuerpoActual,"(if");
        this.aumentarIdentacion();
        this.escribir(cuerpoActual,"(then");
        this.aumentarIdentacion();
    }else {
        this.escribir(cuerpoActual,"br_if $"+ "endforA"+this.fors.peek());
    }

}
```

Si se encuentra una bifurcación incondicional, generamos el salto incondicional al bloque para salir del for.
Si se encuentra una bifurcación por falso, generamos el br_if que es un salto condicional al for en caso de que no se cumpla la condición del mismo.
En ambos saltos se utiliza como etiqueta del salto el nombre del for que está en la pila.

GOTO

Dada la restricción asociada a la complejidad de los saltos hacia arriba, realizamos los GOTO con bloques anidados que contienen al ambito en el que se encuentran (funcion y main)

```
// Tags y saltos
public void agregarBloquesGoto(StringBuilder sb) {
    while (!bloquesGOTO.isEmpty()) {
        sb.append(bloquesGOTO.pop()).append("\n");
    }
}

private void generarTag(Terceto t) {
    String tag = t.getOp1();
    if(!gotos.contains(tag)) {
        this.bloquesGOTO.push(this.ident + "(block $"+tag.replace(':', 'A'));
        this.escribir(cuerpoActual, ") ;; fin de tag: " + tag);
        this.gotos.add(tag);
    }
}
```

Al encontrarse un TAG, se genera el bloque en una estructura que contiene cada declaración de bloque y se cierra el bloque cuando se encuentra la etiqueta.
El método agregarBloquesGoto se encarga de añadir el bloque al ámbito donde se encuentre (el cual es el parámetro StringBuilder)

```
private void saltoIncondicional(Terceto t) {
    this.escribir(cuerpoActual,"br $"+t.getOp1().replace(':', 'A'));
}
```

Al encontrarse un GOTO se realiza el salto incondicional a la etiqueta del bloque.

La declaración de los bloques GOTO se debe insertar en el bloque del ámbito correspondiente.
Si se genera en una función, se agregará al StringBuilder de declaración de variables (ya que se encuentra arriba del cuerpo de las mismas)

```
this.escribir(cuerpoActual,")\n");
this.agregarBloquesGoto(this.variablesActual);
this.funciones.append(variablesActual);
this.funciones.append(cuerpoActual.toString());
```

Luego, para agregar los bloques goto al ámbito del main, se realiza antes de escribir el archivo .wat:

```
private void escribeArchivo() {
    try (FileWriter fw = new FileWriter(this.filePath, true);
        BufferedWriter bw = new BufferedWriter(fw)) {
        bw.write(variablesGlobales.toString() + "\n" );
        bw.write(funciones.toString()+ "\n");
        this.agregarBloquesGoto(inicioMain);
        bw.write(inicioMain.toString() + "\n");
        bw.write(cuerpoMain.toString() + "\n");
        bw.newLine();
    } catch (IOException e) {
        System.err.println("Error al escribir en el archivo: " + e.getMessage());
    }
}
```

Comparaciones

La regla general para las comparaciones, es fijarse el tipo de la variable de la operación a realizar (i32, f64), se obtienen los operandos y se genera el código de la comparación.

```
private void menor(Terceto t) {
    String tipo = t.getTipo();
    if(tipo.equals("double")) {
        this.obtenerGets(t);
        this.escribir(cuerpoActual,"f64.lt");
        this.escribir(variablesActual,"(local $comp"+this.posicionActual + " i32)");
        this.escribir(cuerpoActual,"local.set $comp"+this.posicionActual);
        if(!this.hayPatternMatching)this.escribir(cuerpoActual,"local.get $comp"+this.posicionActual);
    } else if(tipo.equals("ulongint")) {
        this.obtenerGets(t);
        this.escribir(cuerpoActual,"i32.lt_u");
        this.escribir(variablesActual,"(local $comp"+this.posicionActual + " i32)");
        this.escribir(cuerpoActual,"local.set $comp"+this.posicionActual);
        if(!this.hayPatternMatching)this.escribir(cuerpoActual,"local.get $comp"+this.posicionActual);
    } else {
        String tipo_basico = this.ts.getAtributo(t.getTipo(),"tipotripla");
        this.comparacionTripla(tipo_basico.equals("ulongint")? "i32.lt_u":"f64.lt", "i32.eq", t.getOp1(), t.getOp2());
    }
}
```

Luego de esto, se crea la variable comp que siempre es i32, y esta se debe guardar siempre , debido a que en los pattern matching estas se utilizan luego para hacer los ands, debido a que se hacen muchas comparaciones y no quedan en la pila.
En el caso que sea una tripla, se debe hacer una comparaciónTripla

```
private void comparacionTripla(String comp, String eq, String op1, String op2) {

    String op1v1 = "";
    String op1v2 = "";
    String op1v3 = "";
    String op2v1 = "";
    String op2v2 = "";
    String op2v3 = "";
    String aux= "";
    String aux2= "";
    int pos = this.posicionActual;
    Pattern pattern = Pattern.compile("\\[(\\d+)\\]");
    Matcher matcher1 = pattern.matcher(op1);
    Matcher matcher2 = pattern.matcher(op2);
    boolean find1 = matcher1.find();
    boolean find2 = matcher2.find();
```

Se declaran las variables de los respectivos elementos de las triplas y su auxiliar.

```
if(find1) { //expresion
    int indiceTerceto = Integer.parseInt(matcher1.group(1)); //operando1
    Terceto t_op1 = gc_main.getTerceto(indiceTerceto);

    if(this.posicionActual < indiceTerceto) {
        this.posicionActual = indiceTerceto;
        this.ejecutarTraduccion(t_op1);
    }else if(this.posicionActual > indiceTerceto && !t_op1.isHecho()) {
        this.posicionActual = indiceTerceto;
        this.ejecutarTraduccion(t_op1);
    }
    this.posicionActual=pos;

    if(this.aux1.equals(op1))
        aux="AUX1";
    else if(this.aux2.equals(op1))
        aux="AUX2";
    else
        aux="AUX3";
    String tipo = t_op1.getTipo();
    tipo = this.ts.getAtributo(tipo,"tipotripla");
    tipo = tipo.equals("ulongint")?"i32":"f64";
    op1v1 = aux+"V1"+tipo;
    op1v2 = aux+"V2"+tipo;
    op1v3 = aux+"V3"+tipo;

}else { //id de tripla
    op1 = op1.replace(':', 'A');
    op1v1 = op1+"V1";
    op1v2 = op1+"V2";
    op1v3 = op1+"V3";

}
```

Si es una expresión del lado izquierdo (es decir una operación entre triplas) el resultado se encuentra en un auxiliar, se debe acceder al correspondiente y guardar en las variables op el valor de esta expresión(elemento a elemento).

en el caso contrario esque se esta asignando una tripla y se guardan sus respectivos valores.

```
if(find2) {

    int indiceTerceto = Integer.parseInt(matcher2.group(1)); //operando1
    Terceto t_op2 = gc_main.getTerceto(indiceTerceto);

    if(this.posicionActual < indiceTerceto) {
        this.posicionActual = indiceTerceto;
        this.ejecutarTraduccion(t_op2);
    }else if(this.posicionActual > indiceTerceto && !t_op2.isHecho()) {
        this.posicionActual = indiceTerceto;
        this.ejecutarTraduccion(t_op2);
    }
    this.posicionActual=pos;

    if(this.aux1.equals(op2))
        aux2="AUX1";
    else if(this.aux2.equals(op2))
        aux2="AUX2";
    else
        aux2="AUX3";
    String tipo = t_op2.getTipo();
    if(!tipo.equals("double") && !tipo.equals("ulongint")) {
        tipo = this.ts.getAtributo(tipo,"tipotripla");
    }
    tipo = tipo.equals("ulongint")?"i32":"f64";
    op2v1 = aux2+"V1"+tipo;
    op2v2 = aux2+"V2"+tipo;
    op2v3 = aux2+"V3"+tipo;

} else {
    op2 = op2.replace(':', 'A');
    op2v1 = op2+"V1";
    op2v2 = op2+"V2";
    op2v3 = op2+"V3";

}
```

Ya obtenido el primero operando, se busca el segundo operando de la misma manera.
Si el segundo operando tiene un terceto no calculado ya que su número de terceto es mayor, quiere decir que está del lado derecho en un pattern matching y debe calcularse.
Se utiliza la posición actual adecuada para el mismo y al volver se setea la original.

Finalmente, se realiza la comparación elemento a elemento, creando y guardando cada comparación individual.
Luego se libera la variable auxiliar que estaba ocupada.

Finalmente, si había un pattern matching se debe generar el resultado final de la comparación y guardarlo ya que se utilizará próximamente.

```
this.escribir(cuerpoActual, "global.get $"+op1v1);
this.escribir(cuerpoActual, "global.get $"+op2v1);
this.escribir(cuerpoActual, comp);
this.escribir(variablesActual, "(local $comp"+this.posicionActual+"V1" + " i32)");
this.escribir(cuerpoActual,"local.set $comp"+this.posicionActual+"V1");
this.escribir(cuerpoActual, "global.get $"+op1v2);
this.escribir(cuerpoActual, "global.get $"+op2v2);
this.escribir(cuerpoActual, comp);
this.escribir(variablesActual, "(local $comp"+this.posicionActual+"V2" + " i32)");
this.escribir(cuerpoActual,"local.set $comp"+this.posicionActual+"V2");
this.escribir(cuerpoActual, "global.get $"+op1v3);
this.escribir(cuerpoActual, "global.get $"+op2v3);
this.escribir(cuerpoActual, comp);
this.escribir(variablesActual, "(local $comp"+this.posicionActual+"V3" + " i32)");
this.escribir(cuerpoActual,"local.set $comp"+this.posicionActual+"V3");
this.escribir(cuerpoActual,"local.get $comp"+this.posicionActual+"V3");
this.escribir(cuerpoActual,"local.get $comp"+this.posicionActual+"V2");
this.escribir(cuerpoActual, eq);
this.escribir(cuerpoActual,"local.get $comp"+this.posicionActual+"V1");
this.escribir(cuerpoActual, eq);

if (aux.equals("AUX1") || aux2.equals("AUX1"))
    this.aux1=new String("");
else if(aux.equals("AUX2") || aux2.equals("AUX2"))
    this.aux2=new String("");
else if (aux.equals("AUX3") || aux2.equals("AUX3"))
        this.aux3=new String("");

if(this.hayPatternMatching) {
    this.escribir(variablesActual, "(local $comp"+this.posicionActual + " i32)");
    this.escribir(cuerpoActual,"local.set $comp"+this.posicionActual);
}
```

Pattern Matching

Inicio Patron
Hicimos recursivamente la ejecución de los tercetos necesarios para la comparación que están más adelante en la lista de tercetos.
Se utiliza un boolean

```
private boolean hayPatternMatching = false;
```

Que se setea en true cuando se encuentra el terceto Patron y cuando se encuentra un AND o NAND se setea en false (tercetos exclusivamente generados para pattern matching).
Este booleano se utiliza porque si hay pattern matching no necesitamos que el elemento quede en la pila, ya que primero se realizan todas las comparaciones (guardandose en sus

respectivas variables de comparación), y luego cuando ya se hicieron todas las comparaciones individuales, se hace la operación and entre ellas. Que luego sera consumida por el if.

NAND
Este método es utilizado para invertir la condición del for, debido a que para salir del for se debe utilizar la condición negada.

```
private void nand(Terceto t) {
    this.obtenerComparaciones(t);
    this.escribir(cuerpoActual,"i32.and");
    this.escribir(cuerpoActual,"i32.eqz");

}
```

Se realiza un and y luego con el eqz se invierte su resultado. Esto se usa para invertir el valor del eq.

El método obtener comparaciones

```
private void obtenerComparaciones(Terceto t) {
    int comp1 = Integer.parseInt(t.getOp1().split("\\[|\\]")[1]);
    int comp2 = Integer.parseInt(t.getOp2().split("\\[|\\]")[1]);

    if(!gc_main.getTerceto(this.posicionActual-1).getOperador().equals("AND"))
        this.escribir(cuerpoActual,"local.get $comp"+comp1);

    this.escribir(cuerpoActual,"local.get $comp"+comp2);
}
```

Se encarga de obtener las comparaciones necesarias para realizar el and, o nand según corresponda si el pattern se encuentra en un FOR.

Conversión de parámetro formal

Se realiza la conversión adecuada, chequeando en ejecución el caso de que se convierta un flotante negativo a un entero sin signo.

```
private void toDouble(Terceto t) {
    this.obtenerGets(t);
    this.escribir(cuerpoActual,"f64.convert_i32_s");
}

private void toUlongint(Terceto t) {
    this.obtenerGets(t);
    this.escribir(cuerpoActual, "f64.const 0");
    this.escribir(cuerpoActual, "f64.lt");
    this.escribir(cuerpoActual, "(if");
    this.aumentarIdentacion();
    this.escribir(cuerpoActual, "(then");
    this.aumentarIdentacion();
    this.escribir(cuerpoActual, "i32.const "+erroresEj.getDir("conversionErronea"));
    this.escribir(cuerpoActual, "i32.const "+erroresEj.getSizeMsj("conversionErronea"));
    this.escribir(cuerpoActual, "call $log");
    this.escribir(cuerpoActual, "call $exit");
    this.reducirIdentacion();
    this.escribir(cuerpoActual, ")");
    this.reducirIdentacion();
    this.escribir(cuerpoActual, ")");
    this.obtenerGets(t);
    this.escribir(cuerpoActual,"i32.trunc_f64_u");
}
```

OUTF

Para realizar la impresión se utilizan funciones distintas para imprimir valores numéricos y cadenas multilínea. Estas son importadas desde el javascript embebido en el .html.
Se realiza el import de las mismas

```
this.escribir(this.variablesGlobales,"(module");
this.escribir(this.variablesGlobales,"(import \"console\" \"log\" (func $log (param i32 i32)))");
this.escribir(this.variablesGlobales,"(import \"js\" \"mem\" (memory 1))");
this.escribir(this.variablesGlobales, "(import \"env\" \"console_log\" (func $console_log_i32 (param i32)))");
this.escribir(this.variablesGlobales, "(import \"env\" \"console_log\" (func $console_log_f64 (param f64)))");
this.escribir(this.variablesGlobales, "(import \"env\" \"exit\" (func $exit))");
```

Se tiene el método

```
    private void outf(Terceto t) {
        String cadmul = t.getOp1();
        if (cadmul.startsWith("CADMUL:")) { //mensaje comun
            this.escribir(cuerpoActual,"i32.const "+ this.ts.getPosicionMemoria(cadmul));
            this.escribir(cuerpoActual,"i32.const "+ cadmul.substring("CADMUL:".length()).length());
            this.escribir(cuerpoActual,"call $log");
        } else {
            Pattern pattern = Pattern.compile("\\[(\\d+)\\]");
            String op1 = t.getOp1();
            Matcher matcher1 = pattern.matcher(op1);
            boolean find = matcher1.find();
            if(find) { //hay terceto, si hay operacion tiene que estar, salvo acceso o terceto de operacion de tripla
                String terceto = matcher1.group(1);
                Terceto tercetoExpresion = gc_main.getTerceto(Integer.parseInt(terceto));
                String tipo = tercetoExpresion.getTipo().equals("ulongint")?"i32":"f64"; //tipo del terceto(si es un acceso a
tripla) o constante comun

                if(tercetoExpresion.getOperador().equals("ACCESOTRIPLE")) { //es un acceso a tripla y tengo que hacer get al dato
                    String AuxTripla="";
                    if(aux1Tripla.equals(t.getOp1()))
                        AuxTripla="auxTripla";
                    else if(aux2Tripla.equals(t.getOp1()))
                        AuxTripla="aux2Tripla";
                    else
                        AuxTripla="aux3Tripla";
                    this.escribir(cuerpoActual, "global.get $"+tipo+AuxTripla);
                    if(AuxTripla.equals("auxTripla"))
                        this.aux1Tripla="";
                    else if (AuxTripla.equals("aux2Tripla"))
                        this.aux2Tripla="";
                    else
                        this.aux3Tripla="";
                    this.escribir(cuerpoActual, "call $console_log_"+tipo);

                }else { // no es acceso a tripla
```

El primer caso, si es una cadena multilinea (que se tiene tal prefijo en el String del mensaje) Se debe cargar la dirección de memoria, el tamaño de la cadena y luego llamar al método que imprime.

Si fuera un terceto, en caso de ser un acceso a tripla se encuentra el valor en un auxiliar por lo que se detecta cual lo contiene y luego se lo obtiene y se llama a la impresión.

En caso de no ser un acceso a tripla

```
                }else { // no es acceso a tripla

                    if(!tercetoExpresion.getTipo().equals("ulongint") && !tercetoExpresion.getTipo().equals("double")) { //es una
expresion de triplas
                        tipo= this.ts.getAtributo(tercetoExpresion.getTipo(), "tipotripla"); //busco el tipo real de la tripla
                        tipo = tipo.equals("ulongint")?"i32":"f64";
                        String DatoGuardado;
                        if(this.aux1.equals(t.getOp1())) //guardo los datos de el terceto op1 en aux1
                            DatoGuardado="AUX1";
                        else
                            DatoGuardado="AUX2";
                        imprimirTriplas(DatoGuardado,tipo,tipo);
                        if(DatoGuardado.equals("AUX1")) {//Libero la variable auxiliar
                            this.aux1=""; //entonces use lo de aux 1 y libero esa variable
                        }else {
                            this.aux2="";
                        }
                    }else {
                        this.escribir(cuerpoActual, "call $console_log_"+tipo); //es una cosntante comun
                    }
                }
```

Si es un tipo primitivo, ya está en la pila por lo que se llama a la impresión.

Si no es un tipo primitivo, se quiere imprimir un resultado de una operación entre triplas por lo que se llama al módulo que imprime cada elemento

```
// Impresiones por pantalla

private void imprimirTriplas(String var, String tipo, String aux)
    this.escribir(cuerpoActual, "global.get $"+var+"V1"+aux);
    this.escribir(cuerpoActual, "call $console_log_"+tipo);
    this.escribir(cuerpoActual, "global.get $"+var+"V2"+aux);
    this.escribir(cuerpoActual, "call $console_log_"+tipo);
    this.escribir(cuerpoActual, "global.get $"+var+"V3"+aux);
    this.escribir(cuerpoActual, "call $console_log_"+tipo);
}
```

En caso de no ser un terceto lo que se quiere imprimir, si es un ID el cual no pertenece a una tripla entonces se lo obtiene y se llama a la impresión.

Sino, se tiene una constante o un ID de una tripla, por lo que se genera el código asociado a su correspondiente impresión.

```
        }else { //no es un terceto
            String tipo = this.ts.getAtributo(op1, AccionSemantica.TIPO);
            if(this.ts.getAtributo(op1, AccionSemantica.USO).contains("nombre") && ( this.ts.getAtributo(op1,
AccionSemantica.TIPO_BASICO).equals("") )) {//ID Y NO ES TRIPLA
                op1 = op1.replace(':', 'A');
                this.escribir(cuerpoActual, "global.get $"+op1);
                tipo = tipo.equals("ulongint")?"i32":"f64";
                this.escribir(cuerpoActual, "call $console_log_"+tipo);
            }else {
                op1 = op1.replace(':', 'A');
                switch(tipo) { //constantes
                case "ulongint":this.escribir(cuerpoActual, "i32.const "+op1);
                            this.escribir(cuerpoActual, "call $console_log_i32");
                            break;
                case "double":this.escribir(cuerpoActual, "f64.const "+op1);
                            this.escribir(cuerpoActual, "call $console_log_f64");
                            break;
                default: //TRIPLA
                    tipo = this.ts.getAtributo(t.getOp1(), AccionSemantica.TIPO_BASICO).equals("ulongint")?"i32":"f64";
                    imprimirTriplas(op1,tipo);
                    break;
                }
            }
        }

    }
}
```

Errores nuevos considerados

Se agregó la restricción de que no se pueda realizar un salto hacia atrás con un GOTO, razón por la cual se lo considera un error.

- No se puede declarar variables con tipos embebidos.
- Se chequea la longitud de los patrones de pattern matching.
- Se chequea el tipo en las asignaciones.
- Se chequea el tipo en las expresiones.
- Se chequea declaración en las expresiones.

- No se puede declarar una función que su ID implique un tipo embebido.
- Se chequea la declaración de las variables y funciones.
- Se arroja un **warning** de conversiones innecesarias.
- Se chequea el tipo entre el parámetro real y el formal.
- Se controla que la constante de avance del FOR sea de tipo entero.
- Se controla que la variable de avance del FOR sea de tipo entero.
- Se controla que la constante de inicialización de la variable de avance del FOR sea de tipo entero.
- Se chequea la redeclaración del tipo de una variable, con y sin tipo erróneo (que si era de tipo embebido se la redeclare y con otro tipo).
- Se chequea la redeclaración de variables y funciones en un mismo ámbito.
- Se chequea el tipo de retorno de la función.
- Se chequea el tipo en las comparaciones.
- Se chequea la declaración en las comparaciones.

Correcciones de TP 1 y TP2

Normalización de doubles

Se realizó una normalización de los doubles que tienen el mismo valor, aunque se ingresen de forma distinta, para que se guarden una sola vez. Elegimos normalizar con un exponente “e” en lugar de “d” ya que es el que utiliza webAssembly para flotantes de 64 bits.

Cadenas de texto

Se guardaron en un Set<String>, en la clase TablaSimbolos pero no en su estructura principal. Luego fue cambiado por un Map<String, Integer> debido a que las cadenas multilinea deben tener una posición de memoria asociada para guardarse en webassembly y luego llamarse para su respectiva impresión.

Triplas

A la entrada del tipo definido por el usuario en la tabla de simbolos, se le añade una entrada “tipotripla” para guardar su tipo primitivo y otra entrada USO con “nombre de tipo tripla”.
A su vez, en las variables del tipo tripla en la tabla de símbolos, se agrega una entrada “tipo básico” para guardar su tipo primitivo, ya que su entrada “tipo” es del tipo definido por el usuario.

Cambios del TP3 necesarios para TP4

Se añadió un terceto de Inicio de patrón para poder setear el booleano que controla la generación de los resultados en las operaciones para no dejarlos en la pila.

```
lista_patron_izq    : lista_patron_izq ',' expresion_matematica {

this.iniciarPatron(); this.cantPatronIzq++;$$.sval = gc.addTerceto('
"");}
                | expresion_matematica {

                gc.addTerceto("PATRON", "-", "-");
                this.iniciarPatron(); this.cantPatronIzq=1; $$.sval
this.ambitoActual), "");}
                ;
```

Se genera un terceto etiqueta con un operando endfor, para la generación de partes específicas del for en webassembly.

```
   gc.actualizarBF(gc.getCantTercetos());
   gc.pop();
   this.gc.addTerceto("Label" + this.gc.getCantTercetos(), "endfor", "FOR"+this.cantFors);
```

Se genera otro terceto para el inicio del for.

```
   this.varFors.add($1.sval);
   this.cantFors++;
   this.gc.addTerceto("Label" + this.gc.getCantTercetos(), "FOR"+this.cantFors, "-");
   }
```

Si bien antes se generaban, no contenían información en los operandos, la cual necesitamos posteriormente para la generación de cada parte del for.

Se genera un terceto de “inicio if” ya que es necesario para mantener el flujo actual del programa.

```
condicion_punto_control : condicion {
                        gc.addTerceto("inicioif", "-", "-");
                        gc.addTerceto("BF", $1.sval, "");
                        gc.push(gc.getPosActual());
                }
                ;
```

Se distinguió las reglas de acceso a tripla, para generar tercetos distintos para acceso de asignación y acceso para obtener el elemento.

```
triple_asig     : ID '{' expresion_matematica '}' {String tipo = gc.getTipoAccesoTripla($3.sval, this.ts);
                                                   if(tipo != "ulongint"){ErrorHandler.addErrorSintactico("Se intento acceder con un tipo distinto a entero a
la tripla.", lex.getLineaInicial());}

                                                        String idTripla=gc.checkDeclaracion($1.sval,lex.getLineaInicial(),this.ts,ambitoActual);
                                                        if (idTripla != null) {
                                                               tipo = this.ts.getAtributo(idTripla, AccionSemantica.TIPO_BASICO);
                                                         }else{
                                                               ErrorHandler.addErrorSemantico( "La tripla " + idTripla + " nunca fue declarada.",
lex.getLineaInicial()) ;
                                                               tipo = "error";
                                                          }
                                                        $$.sval = gc.addTerceto("ASIGTRIPLA", idTripla, $3.sval, tipo);
                                                       }
                ;

triple          : ID '{' expresion_matematica '}' {String tipo = gc.getTipoAccesoTripla($3.sval, this.ts);
                                                   if(tipo != "ulongint"){ErrorHandler.addErrorSintactico("Se intento acceder con un tipo distinto a entero a
la tripla.", lex.getLineaInicial());}                                                                                    String
idTripla=gc.checkDeclaracion($1.sval,lex.getLineaInicial(),this.ts,ambitoActual);
                                                        if (idTripla != null) {
                                                               tipo = this.ts.getAtributo(idTripla, AccionSemantica.TIPO_BASICO);
                                                         }else{
                                                               ErrorHandler.addErrorSemantico( "La tripla " + idTripla + " nunca fue declarada.",
lex.getLineaInicial()) ;
                                                               tipo = "error";
                                                          }
                                                       $$.sval = gc.addTerceto("ACCESOTRIPLE", idTripla, $3.sval, tipo);
                                                       }
                .
```

Se creó un método que invierte condiciones, utilizado para la condición del for ya que se debe romper el bucle cuando la condición NO se cumple

```java
public void invertirCondicion(String terceto) {
    if(terceto != null) {
        Terceto t = this.getTerceto(Integer.parseInt(terceto.substring(1, terceto.length()-1)));
        String operador = t.getOperador();
        switch (operador) {
        case "<": t.setOperador(">="); break;
        case "<=": t.setOperador(">"); break;
        case ">": t.setOperador("<="); break;
        case ">=": t.setOperador("<"); break;
        case "=": t.setOperador("!="); break;
        case "!=": t.setOperador("="); break;
        case "AND": t.setOperador("NAND"); break;
        default:
        }
    }
}
```

Conclusión

En esta segunda parte del trabajo práctico, nos enfocamos en la generación de código intermedio y su traducción a WebAssembly.

Implementamos un sistema de tercetos para representar las operaciones y flujos del programa, permitiendo manejar expresiones, asignaciones y sentencias de control de manera eficiente.

Además, se introdujeron mejoras en el manejo de ámbitos y chequeos de tipos, lo que redujo errores y aumentó la robustez del compilador.

Esta segunda etapa consolidó las bases sentadas en la primera parte, donde desarrollamos el análisis léxico y sintáctico. Integrando ambas partes, logramos un compilador funcional capaz de traducir programas de un lenguaje de alto nivel a código ejecutable en WebAssembly.

Se logró una muy buena introducción al diseño de compiladores, y se obtuvo una mejor comprensión del proceso de desarrollo e implementación de estos, junto a sus respectivos desafíos técnicos dados los requisitos del compilador y las restricciones del lenguaje.

En nuestro caso, decidimos utilizar webassembly ya que es una tecnología bastante nueva que está creciendo y tiene un buen enfoque a futuro, lo cual consideramos una buena incorporación a nuestro aprendizaje. Sin embargo, la adopción de esta tecnología nos generó ciertos desafíos que, en gran medida, logramos solucionar.

En resumen, el trabajo realizado en esta segunda parte del proyecto permitió cerrar el ciclo completo del compilador, desde el análisis léxico y sintáctico hasta la generación final de código ejecutable. Se logró construir un compilador funcional para un subconjunto del lenguaje objetivo, demostrando la capacidad de traducir de manera precisa un lenguaje de alto nivel a código de bajo nivel para WebAssembly.