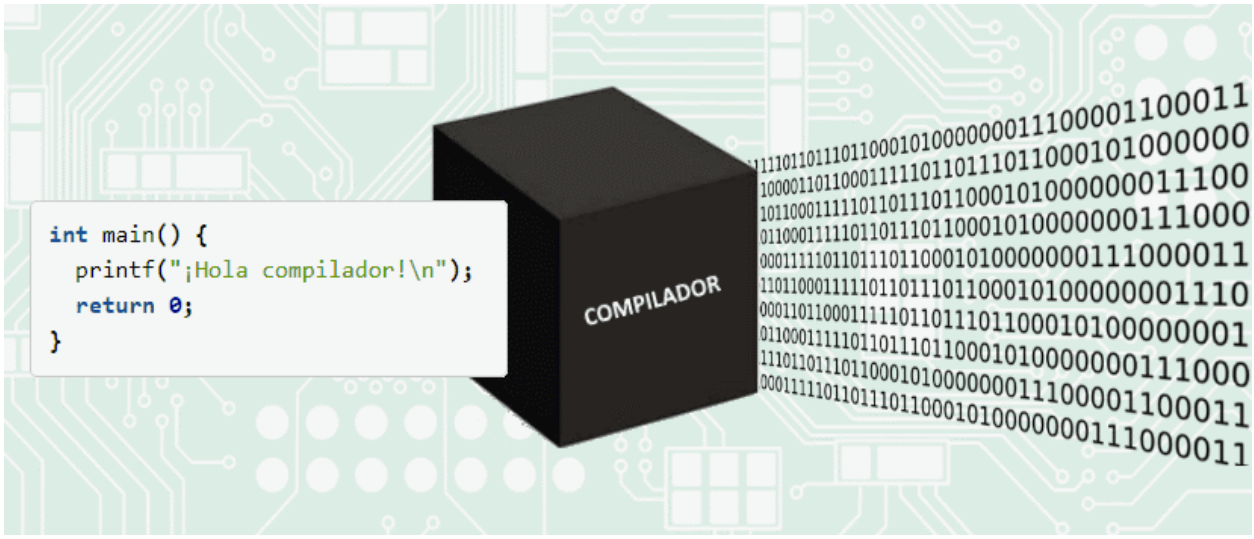
Diseño de Compiladores 1

Grupo 10 - Tp 3 y 4



Integrantes

Medioli Nicolas - mediolifnicolas@gmail.com

Arias Facundo - facundoarias1231@gmail.com

Docente asignado

Jose Fernandez Leon

Temas asignados

6 8 12 14 18 20 21 24 28 29 32 35

índice

Introducción

En este informe se detalla el desarrollo y la implementación de las dos últimas etapas de un compilador: Análisis Semántico y Generación de código. Las especificaciones de las mismas, fueron dadas por la cátedra.

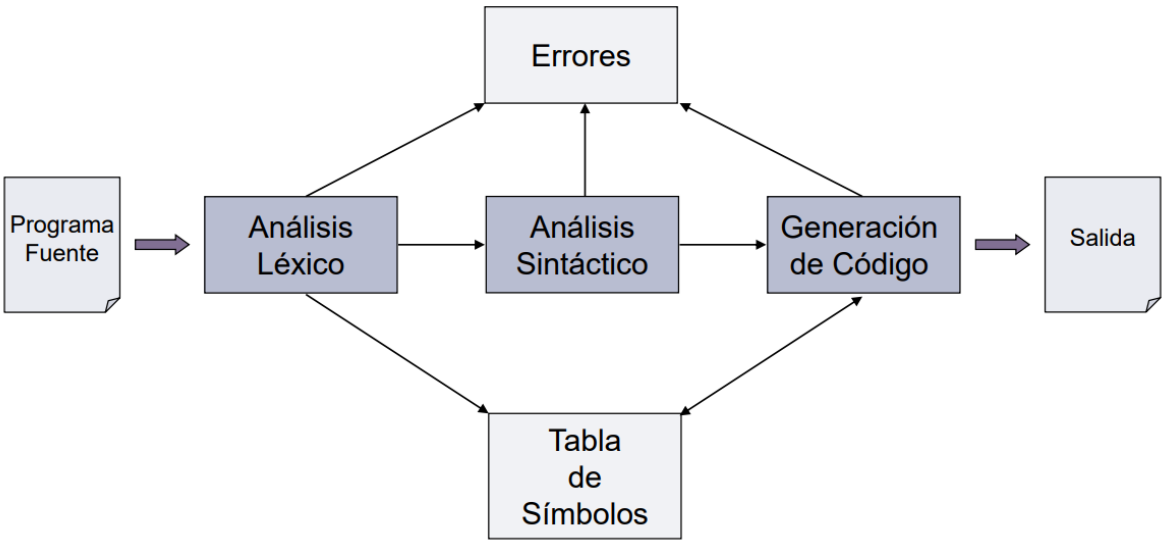


Figura 1: "Estructura general del compilador"

Ejecución del compilador

El ejecutable del compilador se ejecuta abriendo una consola dentro de la carpeta "entrega grupo 10" y escribiendo el siguiente comando de ejemplo:

- *java -jar compilador.jar tests\lexico\case1.txt*

Especificaciones asignadas (tp3 y tp4)

Análisis semántico:

- Generacion de codigo intermedio: ***Tercetos***
- Operadores:
    - ***Tema 12:***

      Ante una sentencia de asignación que utilice el operador -=, se deberá generar código para asignar a la

      variable su valor actual - el resultado del lado derecho de la asignación:

      Por ejemplo:

      a -= 10_i, // El código generado debe efectuar la asignación a = a - 10_i,

- Sentencias de control:
    - ***Tema 14: DO UNTIL***

      El bloque de sentencias ejecutables se ejecutará hasta que la condición sea verdadera. La evaluación de la condición se efectuará luego de la ejecución del bloque.

- *Objetos:*
    - ***Tema 18: Herencia por composición - Uso con nombre***

Ejemplos de declaración de clases:

```
CLASS ca {
   INT a;c,     // declaración de atributos
   VOID m() {   // declaración de método
     …
   },
}

CLASS cb{
   FLOAT b,     // declaración de atributo
   FLOAT a,     // declaración de atributo
   VOID n() {   // declaración de método
      …
   },
   ca,          // nombre de clase
}
…
```

Semántica de la herencia: La clase cb hereda de la clase ca.

Ejemplos de declaración de objetos para las clases ca y cb:

```
ca a1; a2,
cb b1; b2; b3,
```

Ejemplos de referencias a métodos y atributos de los objetos declarados:

```
a1.a = 3_i,     // atributo declarado en ca
b1.ca.a = 3_i, // atributo heredado de ca
b1.a = 2.3,     // atributo declarado en cb
b1.b = 1.2,     // atributo declarado en cb
b1.ca.c = 1_i, // atributo heredado de ca
b1.ca.m(),      // método heredado de ca
b1.n(),         // método declarado en cb
a2.m(),         // método declarado en ca
```

Se indica explícitamente si el atributo o método corresponden a la clase heredada por composición.

El compilador debe efectuar todos los chequeos semánticos necesarios para comprobar el correcto uso de atributos y métodos.

- ***Tema 20: Declaracion de metodos distribuida***

Los métodos pueden declararse dentro de la declaración de la clase a la que pertenecen, o mediante una cláusula IMPL. Ejemplo de implementación de métodos mediante cláusula IMPL:

```
CLASS ca {
   INT a;c,     // declaración de atributos
   VOID m() {   // declaración de método
     …
   },
   VOID p(),    // prototipo de método
}

ca x; y; z,

x.p();          // Debe ser informado como error

IMPL FOR ca: {
   VOID p() {   // implementación del método p de la clase ca
      ….
    },
   …
}
```

El compilador debe efectuar los chequeos necesarios para controlar el correcto acceso a los métodos declarados. En el ejemplo, el método p no podrá ser utilizado hasta su implementación.

- ***Tema 21: Forward declaration***

Incorporar la posibilidad de declarar las clases con posterioridad a su uso.

Por ejemplo:

```
CLASS ca {      // Declaración de clase ca
   VOID m() {
      …
      }
   ...
   }
CLASS cc,
CLASS cd {
   cc c1,
   ca a1,
   VOID j() {
     c1.p(),   // Debe ser informado como error por el compilador
     a1.m(),   // Invocación correcta
   }
}
CLASS cc {      // forward declaration para clase cc
  INT z,
  VOID p() {
     …
     }
  …
}
```

El compilador debe efectuar todos los chequeos semánticos necesarios para comprobar el correcto uso de atributos y métodos.

- ***Tema 24: Sobreescritura de atributos***

Una clase que hereda de otra puede sobreescribir atributos declarados en la clase de la que hereda, pero no puede sobreescribir métodos. El compilador debe efectuar todos los chequeos semánticos necesarios para comprobar el correcto uso de atributos y métodos

- *Comprobación de uso de variables*
  - ***Tema 28:***

    El compilador debe chequear e informar, para cada variable declarada, si no fue usada del lado derecho de una asignación en ningún ámbito.

- Conversiones

El chequeo de tipos y la incorporación de conversiones implícitas se efectuará durante la generación de código intermedio para Tercetos y Árbol Sintáctico, y durante la generación de código Assembler para Polaca Inversa

- ***Tema 29: Conversiones explícitas***

- El compilador debe reconocer el uso de conversiones explícitas en el código fuente, que serán indicadas mediante la palabra reservada asignada en el TP02.

```
TOF (<expresión>)      // para grupos que tienen asignado el tema 7
TOD (<expresión>)      // para grupos que tienen asignado el tema 8
```

- Se debe considerar que cuando se indique la conversión **TOD**(expresión) o **TOF**(expresión), el compilador deberá generar código para efectuar una conversión del tipo del argumento al tipo indicado por la palabra reservada. (**DOUBLE** o **FLOAT** respectivamente). Dado que los otros tipos asignados al grupo son enteros (1-2-3-4-5-6), el argumento de una conversión, debe ser de alguno de esos dos tipos.
- Sólo se podrán efectuar operaciones entre dos operandos de distinto tipo (uno entero y otro flotante), si se convierte el operando de tipo entero (1-2-3-4-5-6) al tipo de punto flotante mediante la conversión explícita que corresponda. En otro caso, se debe informar error.
- En el caso de asignaciones, si el lado izquierdo es de uno de los tipos enteros asignados (1-2-3-4-5-6), y la expresión del lado derecho es de tipo diferente, se deberá informar error por incompatibilidad de tipos.

Generacion de codigo

- Chequeo en tiempo de ejecución

c) Overflow en sumas de datos de punto flotante El código Assembler deberá controlar el resultado de la operación indicada, para el tipo de datos de punto flotante asignado al grupo. Si el mismo excede el rango del tipo del resultado, deberá emitir un mensaje de error y terminar.

d) Overflow en productos de enteros: El código Assembler deberá controlar el resultado de la operación indicada, para los tipos de datos enteros asignados al grupo. Si el mismo excede el rango del tipo del resultado, deberá emitir un mensaje de error y terminar.

f) Resultados negativos en restas de enteros sin signo:

El código Assembler deberá controlar el resultado de la operación indicada. Este control se aplicará a operaciones entre enteros sin signo. En caso que una resta entre datos de este tipo arroje un resultado negativo, deberá emitir un mensaje de error y terminar.

- chequeo de tipos y conversiones
    - **Tema 29: Conversiones explícitas**

      Los grupos cuyos lenguajes incluyen conversiones explícitas, deberán traducir el código de las conversiones a código Assembler.

Análisis semántico - Generación de código

En esta etapa, la forma de representación implementada son los tercetos, debido a esto, debíamos guardar el "tipo" de los identificadores y de las constantes en la tabla de símbolos, además, debíamos incorporar un atributo Uso, indicando el uso de un identificador en el programa, entre otros atributos adicionales. Debido a esto, se hicieron algunos cambios en la tabla de símbolos:

Cambios en la tabla de símbolos

La tabla tiene como clave un dato de tipo String asociado a un "SymbolTableEntry".

```
public class SymbolTable {

    private HashMap<String, SymbolTableEntry> table;
```

Este último tipo contiene varios atributos necesarios para la implementación de todas las estructuras que utilizaremos a lo largo del trabajo.

```java
public class SymbolTableEntry {

    private final int tokenID;
    private String lexeme;
    private final boolean predefined;

    private final HashMap<AttribKey, Object> attribs;
```

El “tokenID” hace referencia a la clave numérica del lexema detectado en el código fuente.

El atributo “lexeme” tiene como valor el lexema, “predefined” guarda si es una entrada precargada (por ejemplo el lexema IF ya está predefinida, por lo tanto es “true”) y por último, “attribs” es un HashMap de atributos que contienen información útil, además, “AttribKey” es un tipo de dato enum que ayuda a identificar el tipo de atributo que se guarda:

```java
public enum AttribKey {

    ID_TYPE(str:"Tipo de identificador"),
    DATA_TYPE(str:"Tipo de dato"),
    INSTANCE_OF(str:"Instancia de clase"),
    ARG_TYPE(str:"Tipo de argumento");

    private final String str;
    AttribKey(String str)
    {
        this.str = str;
    }

    @Override
    public String toString()
    {
        return this.str;
    }
}
```

Anteriormente, para detectar la ubicación donde se encuentra el token en el código, teníamos un atributo “*location*” que guardaba el número de línea donde se encontraba. Para esta entrega, decidimos modificarlo y separarlo en una clase a parte que se llama

LocatedSymbolTableEntry, que funciona como un decorador para la clase *SymbolTableEtnry,* en la implementacion que se muestra a continuacion se puede ver facilmente:

```
3   public class LocatedSymbolTableEntry {
4
5       private SymbolTableEntry entry;
6       private TokenLocation location;
7
8       public LocatedSymbolTableEntry(SymbolTableEntry entry, TokenLocation location)
9       {
10          this.entry = entry;
11          this.location = location;
12      }
13
14      public SymbolTableEntry getSTEntry()
15      {
16          return this.entry;
17      }
18
19      public TokenLocation getLocation()
20      {
21          return this.location;
22      }
23  }
24
```

Decidimos implementarlo de esta manera porque queda un código más organizado.

YACCDataUnit y SemanticHelper

Antes de explicar cómo implementamos lo pedido en el práctico, es necesario hacer una mención a estas dos estructuras. Ambas sirven como apoyo a la hora de agregar cambios y funcionalidades a la gramática del compilador.

YACCDataUnit es una clase que tiene como atributos todos los tipos de datos necesarios a la hora de utilizar la gramática, esto es muy útil ya que hace más fácil y maleable su uso, además, como muchas reglas gramaticales están anidadas entre sí, todas tienen que retornar y utilizar el mismo tipo de dato.

```java
public class YACCDataUnit {

    public DataType dataType;
    public int firstTriplet;
    public int tripletQuantity;
    public String lexeme;
    public Vector<LocatedSymbolTableEntry> tokensData;
    public TripletOperand tripletOperand;
    public int reservedTriplet;
    public String referencedEntryKey;
    protected boolean valid;
```

Lo anterior mencionado, hace referencia a la notación posicional que ofrece YACC para permitirnos manipular el argumento de una regla sintáctica o el dato que queremos retornar. Como esto nos causaba muchos problemas a la hora de retornar algo que no era compatible con el uso que queríamos dar, decidimos hacer este nuevo tipo de dato general que utiliza la gramática. Un ejemplo de este uso es el siguiente:

```
| ID
    {
        // Chequear que exista en el ambito local

        LocatedSymbolTableEntry tokenData = (LocatedSymbolTableEntry)$1.obj;
        String lexeme = tokenData.getSTEntry().getLexeme();

        String referencedEntryKey = semanticHelper.getEntryKeyByScope(lexeme, getCurrentScopeStr());

        if (referencedEntryKey == null)
        {
            compiler.reportSemanticError(
                String.format("ID no encontrado: %s", lexeme),
                getTokenLocation($1)
            );

            YACCDataUnit data = new YACCInvalidDataUnit();
            data.tokensData.add((LocatedSymbolTableEntry)$1.obj);
            $$ = new ParserVal(data);
            break;
        }

        YACCDataUnit data = new YACCDataUnit();
        data.tokensData.add((LocatedSymbolTableEntry)$1.obj);
        data.referencedEntryKey = referencedEntryKey;

        $$ = new ParserVal(data);
    }
;
```

En este ejemplo se observa cómo se crea un objeto de tipo *YACCDataUnit llamado "data" y se* manipula el primer argumento de la regla ($1) casteando como un *LocatedSymbolTableEntry,despues* se lo agrega al atributo "tokenData" del objeto "data" y luego se guarda la referencia a la entrada de la tabla de símbolos. Por último se retorna un *ParserVal* que contiene un objeto de tipo *YACCDataUnit* utilizando la notación "$$".

Hay muchos ejemplos de esta utilización a lo largo de la gramática, pero siempre tiene la misma estructura de manipulación y retorno. No explicamos todos los casos porque se haría interminable el informe

Por otro lado, SemanticHelper también es una clase, pero contiene funciones y métodos que se utilizan varias veces en la gramática, además contiene la lógica para saber el entorno en el que se está declarando una variable, entre otras funcionalidades. Más adelante se pondrá un ejemplo del uso de la misma.

Tablas de compatibilidad de tipos

Para hacer la conversión explícita, utilizamos una estructura que contenga todos los tipos de las variables y dependiendo la operación que se realice entre ellas devolvemos un resultado booleano "true" si es compatible, de lo contrario, se imprime por pantalla un error indicando que las variables no son compatibles.

Por ejemplo, para el código:

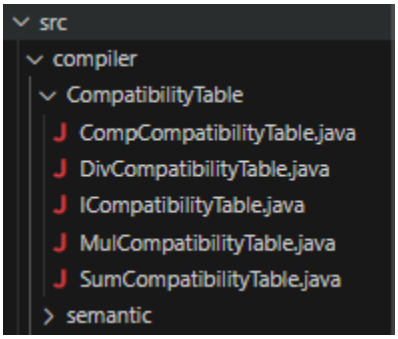
```
1  {
2
3      UINT var1,
4      UINT var2,
5      LONG var3,
6
7
8      var2 = 7_ui,
9      var3 = 8_l,
10
11
12     var1 =  var2 + var3,
13
14     }
```

Se imprime el siguiente error:

```
[Semantico: Linea 12] No se pueden sumar variables de distinto tipo

Hubo errores en el parsing
```

En un principio hubo una confusión y implementamos una conversión implícita (por eso la solución de la conversión explícita no es la mas optima), utilizando las tablas anteriormente dichas, en vez de devolver un booleano, retornamos el tipo que tiene que dar el resultado de la operación, luego de un simple cambio, pudimos solucionarlo.

Las clases utilizadas son :



Implementando la interfaz "*ICompatibilityTavle.jav*", un ejemplo de utilización de la misma, es la clase *MulCompatibilitytable* :

```java
public class MulCompatibilityTable implements ICompatibilityTable {

    private static DataType[][] matrix;

    private static synchronized void initializeMatrix()
    {
        int dataTypesQuantity = DataType.values().length;
        matrix = new DataType[dataTypesQuantity][dataTypesQuantity];

        setSymmetric(_uint, _uint, DataType.LONG);
        setSymmetric(_long, _long, DataType.LONG);
        setSymmetric(_double, _double, DataType.DOUBLE);
    }

```

La misma, tiene un método para calcular el resultado de una operación (en este caso multiplicación) entre dos operandos pasados como parámetro:

```
public DataType calcDataType(DataType a, DataType b)
{
    return matrix[a.ordinal()][b.ordinal()];
}
```

Por ejemplo, si recibe como parámetro dos "*UINT*" devuelve "*true*". Estas estructuras fueron utilizadas para la creación de tercetos detectando si es válida la operación.

En el caso que la operación no sea válida, se mostrará un mensaje de error por pantalla.

Tercetos

Antes de crear los tercetos, necesitábamos una estructura dinámica que los contenga, por eso, creamos una clase "ListOfTriplet" que tiene como principal atributo un Vector de tercetos.

```
public class ListOfTriplets {
    private final Vector<Triplet> tripletList;
```

Luego, para realizar un terceto, creamos una clase que guarda la operación que se está efectuando, los dos operandos involucrados, y el tipo de dato que debe retornar:

```
public class Triplet {
    private final String operation;
    private final TripletOperand o1;
    private final TripletOperand o2;
    private DataType type;
```

"TripletOperand" es un tipo de dato especial que contiene el operando, el cual puede ser una entrada a la tabla de símbolos o el índice vinculado al terceto que está involucrado.

```
public class TripletOperand {

    public final Integer index;

    public final SymbolTableEntry stEntry;
```

El terceto se crea y se agrega a la lista en el código de la gramática (compiler.y) dependiendo la regla gramatical que se esté ejecutando, estas ocurren cuando hay una operación de asignación, selección, sentencia de control, sentencia PRINT, invocación a funciones y métodos de clase y sentencia RETURN.

Ejemplo de creación y agregación de terceto a la lista utilizando la clase *SemanticHelper*:

```
954  basic_expr
955      : expr '+' term
956          {
957              YACCDataUnit data1 = (YACCDataUnit)$1.obj;
958              YACCDataUnit data3 = (YACCDataUnit)$3.obj;
959
960              if (!data1.isValid() || !data3.isValid())
961              {
962                  $$ = new ParserVal(new YACCInvalidDataUnit());
963                  break;
964              }
965
966              TripletOperand operand1 = data1.tripletOperand;
967              TripletOperand operand2 = data3.tripletOperand;
968
969              Triplet t = semanticHelper.getTriplet(operand1, operand2, "+", listOfTriplets, sumCompatibilityTable);
970
971              if (t == null) {
972                  compiler.reportSemanticError("No se pueden sumar variables de distinto tipo", getTokenLocation($2));
973                  $$ = new ParserVal(new YACCInvalidDataUnit());
974                  break;
975              }
976
```

Hicimos un método en *SemanticHelper* llamado *getTriplet*, en pocas palabras, le damos dos operandos, una operación, la lista de triplets y la tabla de compatibilidad que corresponda (en el caso anterior *SumCompatibilityTable*). Luego, esta función crea el terceto y retorna null si no pudo crearlo, en ese caso se imprime el error por pantalla.

A continuación se muestra y detalla la lógica del método *getTriplet*:

```
423     public Triplet getTriplet(TripletOperand operand1, TripletOperand operand2, String oper
424
425         Triplet t = new Triplet(operation, operand1, operand2);
426
427         DataType type1, type2;
428
429         if (operand1.isFinal()) {
430             type1 = (DataType)operand1.getstEntry().getAttrib(AttribKey.DATA_TYPE);
431         } else {
432             type1 = listOfTriplets.getTriplet(operand1.getIndex()).getType();
433         }
434
435         if (operand2.isFinal()) {
436             type2 = (DataType)operand2.getstEntry().getAttrib(AttribKey.DATA_TYPE);
437         } else {
438             type2 = listOfTriplets.getTriplet(operand2.getIndex()).getType();
439         }
440
441         DataType isValid = compatibilityTable.calcDataType(type1, type2);
442
443
444         if (isValid == null)
445         {
446             return null;
447         }else{
448             if (operation == "/")
449                 t.setDataType(DataType.DOUBLE);
450             else
451                 t.setDataType(type1);
452         }
453         return t;
454     }
```

Luego de recibir por parámetro los operandos, la operación, la lista de triplets y la tabla de compatibilidad correspondiente (suma, resta multiplicacion o division), crea un terceto con la operación y los operandos, luego, en los if, pregunta si los operandos son “finales” ,osea si son entradas a la tabla de símbolos, en caso contrario, son índices a otro terceto, en dichos casos se le setea el tipo de diferentes formas.

Luego para saber si la operación es válida, usando el método *calcDataType* de la tabla de compatibilidad, se retorna un valor “BOOLEAN” si es posible operar los datos, en caso contrario se retorna null.

Por último, el *if* de la línea 444 revisa si la operación es válida, si es null retorna null, si no, se chequea si la operación es una división, en ese caso, se setea el tipo de terceto con *DOUBLE* (Ya que el resultado de todas las divisiones es double), si la operación no es de división se le setea el tipo de terceto con el tipo del primer operando, ya que como ambos operandos son del mismo tipo, es indiferente cual se setea.

Para las operaciones de control *IF*, se crea un terceto de la condición:

```
sentencia_if
    : IF '(' condicion_if_reserva ')' cuerpo_if END_IF
        {
            YACCDataUnit data3 = (YACCDataUnit)$3.obj;
            YACCDataUnit data5 = (YACCDataUnit)$5.obj;

            int jzToBackpatch = data3.reservedTriplet;

            String comp = data3.lexeme;

            int end_if = 1 + jzToBackpatch + data5.tripletQuantity;
            listOfTriplets.replaceTriplet(
                jzToBackpatch,
                new Triplet(
                    "NEG_CJUMP",
                    new TripletOperand(end_if, listOfTriplets),
                    null
                )
            );

            listOfTriplets.addTag(end_if, listOfTriplets.getNewIfTag());

            YACCDataUnit data = new YACCDataUnit();
            data.tripletQuantity = 1 + data3.tripletQuantity + data5.tripletQuantity;
            data.tokensData.add((LocatedSymbolTableEntry)$1.obj);
            data.firstTriplet = data3.reservedTriplet;

            $$ = new ParserVal(data);
        }
```

```
condicion_if_reserva
    : condicion
        {
            // Agrega el terceto del jump condicional

            Triplet triplet = new Triplet("JZ", null, null);
            int tripletID = listOfTriplets.addTriplet(triplet);

            YACCDataUnit data = (YACCDataUnit)$1.obj;
            data.reservedTriplet = tripletID;

            $$ = new ParserVal(data);
        }
    ;
```

La regla *condición* crea un terceto con los dos operadores y el comparador utilizado.

Luego de crear el terceto correspondiente, se calcula en que linea de codigo termina el cuerpo del if, para saber hacia dónde hay que saltar y se crea un nuevo terceto con el índice al cual hay que saltar si no se cumple la condición (Por eso se utiliza *NEG_CJUMP*) :

```
sentencia_if
    : IF '(' condicion_if_reserva ')' cuerpo_if END_IF
        {
            YACCDataUnit data3 = (YACCDataUnit)$3.obj;
            YACCDataUnit data5 = (YACCDataUnit)$5.obj;

            int jzToBackpatch = data3.reservedTriplet;

            String comp = data3.lexeme;

            int end_if = 1 + jzToBackpatch + data5.tripletQuantity;
            listOfTriplets.replaceTriplet(
                jzToBackpatch,
                new Triplet(
                    "NEG_CJUMP",
                    new TripletOperand(end_if, listOfTriplets),
                    null
                )
            );

            listOfTriplets.addTag(end_if, listOfTriplets.getNewIfTag());

            YACCDataUnit data = new YACCDataUnit();
            data.tripletQuantity = 1 + data3.tripletQuantity + data5.tripletQuantity;
            data.tokensData.add((LocatedSymbolTableEntry)$1.obj);
            data.firstTriplet = data3.reservedTriplet;

            $$ = new ParserVal(data);
        }
```

El *DO UNTIL* se maneja similar al *IF*. Se guarda el indice del terceto donde comienza el cuerpo del DO, Luego, si se cumple la condición del UNTIL, se vuelve a ejecutar el cuerpo anteriormente mencionado:

```
do_until
    : DO cuerpo_do UNTIL '(' condicion ')'
        {
            YACCDataUnit data2 = (YACCDataUnit)$2.obj;
            YACCDataUnit data5 = (YACCDataUnit)$5.obj;

            // Agregar salto condicional

            int bodyStartTriplet = listOfTriplets.getSize() - data2.tripletQuantity - data5.tripletQuantity;

            Triplet triplet = new Triplet("NEG_CJUMP", new TripletOperand(bodyStartTriplet, listOfTriplets), null);
            int tripletID = listOfTriplets.addTriplet(triplet);

            YACCDataUnit data = new YACCDataUnit();
            data.tokensData.add((LocatedSymbolTableEntry)$1.obj);
            data.tripletQuantity = 1 + data2.tripletQuantity + data5.tripletQuantity;
            data.firstTriplet = tripletID;

            listOfTriplets.addTag(bodyStartTriplet, "do_until_" + listOfTriplets.getIncrementalNum());

            $$ = new ParserVal(data);
        }
    ;
```

bodyStartTriplet almacena el valor del índice donde comienza el cuerpo del Do, luego se agrega el terceto con la operación *NEG_CJUMP.* De esta forma si la condición se cumple avanza, de lo contrario se ejecuta nuevamente la lógica del bucle.

Con respecto a las etiquetas (tags) se va a hablar en la parte correspondiente a la generación de código assembler del respectivo informe.

Chequeos semánticos

Para identificar el ámbito al que pertenecen las variables, funciones, clases, etc, utilizamos “name mangling”, es decir, el nombre de una variable llevará luego de su nombre original, la identificación del ámbito al que pertenece.

Las reglas de alcance del lenguaje proporcionadas por la cátedra son las siguientes:

- Cada variable, función, clase, método será visible dentro del ámbito en el que fue declarada/o y por los ámbitos contenidos en el ámbito de la declaración.
- Cada variable, función, clase, método será visible a partir de su declaración, con la restricción indicada en el ítem anterior.
- Se permiten variables con el mismo nombre, siempre que sean declaradas en diferentes ámbitos.
- Se permiten funciones con el mismo nombre, siempre que sean declarados en diferentes ámbitos.
- Se permiten clases con el mismo nombre, siempre que sean declaradas en diferentes ámbitos.
- No se permiten variables, funciones, clases con el mismo nombre dentro de un mismo ámbito.
- Otras reglas que correspondan a temas particulares (presentados al principio del presente informe).

Para detectar el ámbito actual y agregar el id del mismo al nombre de la variable (función, clase, etc), en la gramática, creamos una “LinkedList” que contiene una lista de ámbitos que se van generando de forma dinámica, además, utilizamos una variable llamada “_currentID” que contiene el identificador del entorno actual, esto nos sirve para detectar rápidamente el alcance de dicha variable.

```
String _currentID;
LinkedList<String> _currentScope;
Compiler compiler;
SemanticHelper semanticHelper;
SymbolTable symbolTable;
String implementationMethodScope;
LinkedList<String> scopeCopy;
ListOfTriplets listOfTriplets;
SumCompatibilityTable sumCompatibilityTable;
MulCompatibilityTable mulCompatibilityTable;
DivCompatibilityTable divCompatibilityTable;
CompCompatibilityTable compCompatibilityTable;
```

Aca se muestra un ejemplo de cómo se chequea el alcance de un identificador:

```
factor
    : ID
        {
            // Chequear alcance y tipo de ID

            String varLexeme = getSTEntry($1).getLexeme();
            SymbolTableEntry varEntry = semanticHelper.getEntryByScope(varLexeme, getCurrentScopeStr());

            if (varEntry == null)
            {
                compiler.reportSemanticError("Variable no alcanzable", getTokenLocation($1));
                break;
            }

            if (varEntry.getAttrib(AttribKey.ID_TYPE) != IDType.VAR_ATTRIB)
            {
                compiler.reportSemanticError("El identificador " + varEntry.getLexeme() + "no es de tipo var/attribute", getTokenLocation($1));
                break;
            }

            YACCDataUnit data = new YACCDataUnit();
            data.tripletOperand = new TripletOperand(varEntry);
            $$ = new ParserVal(data);
        }
```

Aquí se vuelve a utilizar la clase *SemanticHelper* invocando el método ".getEntryByScope(String id, String scope)", este método retorna una entrada a la tabla de símbolos si se encuentra al alcance, sino retorna null.

Si la variable no es alcanzable, la condición "varEntry == null" resuelve como verdadera y reporta el error. Luego, si el identificador no es una variable o un atributo, también imprime un error. Por último devuelve un valor de tipo *YACCDataUnit.*

Detección e impresión de errores semánticos

Variables

1) Variable no declarada en el ámbito. Para el código:

```
1   {
2       VOID m()
3       {
4           UINT var1,
5           var1 = 7_ui,
6
7           PRINT %HOLA%,
8
9           var2 = 1_ui,
10      },
11
12      UINT var2,
13      var2 = 3_ui,
14  }
```

Se puede ver como en la línea 9 se usa una variable que no está declarada en ese ámbito, el error se muestra de la siguiente manera:

```
[Semantico: Linea 9] ID no encontrado:

Hubo errores en el parsing

Lista de tokens leidos:
'{' VOID ID '(' ')' '{' UINT ID ',' ID '=' CTE_UI

Estructuras sintacticas encontradas:
Linea 2, estructura: Definición de función
Linea 4, estructura: Declaración de variable/s
Linea 7, estructura: Sentencia PRINT
Linea 12, estructura: Declaración de variable/s
```

2) Variable redeclarada. Para el código:

```
{
    VOID m()
    {
        UINT var1,
        var1 = 7_ui,
        UINT var2,

        PRINT %HOLA%,

        var2 = 1_ui,
    },

    UINT var2,
    var2 = 3_ui,
}
```

En este caso declaramos una variable del mismo nombre en un ámbito diferente, no se considera un error ya que se está redeclarando, por lo tanto imprime:

```
Parsing correcto
```

3) Variables con mismo nombres pero declaradas en ámbitos diferentes. Para el código:

```
{
    VOID m()
    {
        UINT var1,
    },

    UINT var1,

    var1 = 4_ui,
}
```

El compilador detecta como válido este código, ya que las variables están declaradas en ámbitos diferentes:

```
Parsing correcto
```

4) Variables con el mismo nombre y en el mismo ámbito. Para el código:

```
1   {
2       VOID m(){
3           UINT var1,
4           UINT var2,
5           UINT var2,
6       },
7
8       UINT var1,
9
10  }
```

Acá se puede ver como las variables *var2* están declaradas en el mismo entorno, por lo tanto falla:

```
[Semantico] La variable: var2 ya ha sido declarada

Hubo errores en el parsing
```

Funciones

5) Funciones no declaradas en el mismo entorno. Para el código:

```
{
    VOID m()
    {
        UINT var1,

        VOID t()
        {
            PRINT %HOLA%,
        },
    },

    m(),
    t(),

}
```

Imprime un error, ya que t() no está al alcance:

```
[Semantico: Linea 13] ID no encontrado:

Hubo errores en el parsing
```

6) Funciones con el mismo nombre declaradas en diferentes ambitos. Para el código:

```
{
    VOID m(){
        VOID a(){

        },
    },

    VOID a(){
        PRINT %HOLA%,
    },

    a(),
}
```

Esto es válido ya que utiliza una función que está declarada y es alcanzable:

```
Parsing correcto
```

7) Funciones con el mismo nombre declaradas en el mismo ámbito. Para el código:

```
1   {
2       VOID a(){
3
4       },
5
6       VOID a(){
7           PRINT %HOLA%,
8       },
9
10      a(),
11  }
```

Se muestra el siguiente error:

```
[Semantico: Linea 6] Identificador ya declarado en el ámbito local
Error en yyparse()
```

Clases

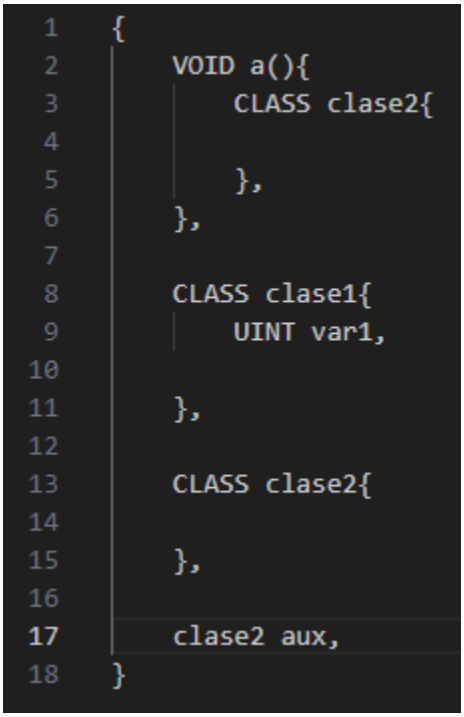
8) Clase visible dentro del ámbito donde fue declarada. para el código:

```
1   {
2       CLASS clase1{
3           UINT var1,
4       },
5
6       clase1 a,
7   }
```

El compilador ejecuta sin errores:

```
Parsing correcto
```

9) Clases con el mismo nombre pero declaradas en ámbitos diferentes. Para el código:



```
{
    VOID a(){
        CLASS clase2{

        },
    },

    CLASS clase1{
        UINT var1,

    },

    CLASS clase2{

    },

    clase2 aux,
}
```

Como *clase2* está declarada en dos entornos diferentes (global y función respectivamente), el código es válido:

```
Parsing correcto
```

10) Clase con el mismo nombre declaradas en el mismo ámbito. Para el código:

```
1   {
2       CLASS clase1{
3           UINT var1,
4
5       },
6
7       CLASS clase1{
8
9       },
10
11      clase1 aux,
12  }
```

Esto no es válido y se muestra por pantalla:

```
[Semantico: Linea 7] Clase ya definida en el ambito local: clase1
Error en yyparse()
```

Generación de código - Código assembler

Para esta última etapa, elegimos generar código assembler para Pentium de 32 bits utilizando la herramienta dada por la cátedra “MASM32”.

Se debe aclarar que el compilador puede imprimir por pantalla variables (Esto no fue pedido por la cátedra, pero decidimos incluirlo ya que se nos hacia mas comodo verificar algunos resultados y además no fue complicado agregarlo).

Modificaciones agregadas

Antes de poder implementar la generación a código assembler, tuvimos que hacer algunos cambios.

Necesitamos que cada terceto tenga asociado una estructura que guarde el nombre de la variable auxiliar utilizada en la sección de “data” del código assembler, también, dicha estructura debe guardar el espacio que ocupa el tipo de dato que tiene. Para esto, creamos una clase nueva llamada MemoryAssociation con los siguientes atributos:

```
3  public class MemoryAssociation {
4
5      private String tag = null;
6      private int offset = -1;
7      private int size = -1;
8      private DataType dataType = null;
9      private boolean isConstant = false;
10
```

y los constructores:

```java
public MemoryAssociation(String tag, int size, DataType dataType)
{
    this.tag = SymbolTable.encodeString(tag);
    this.size = size;
    this.dataType = dataType;
}

public MemoryAssociation(int offset, int size, DataType dataType)
{
    this.offset = offset;
    this.size = size;
    this.dataType = dataType;
}

public MemoryAssociation(String tag)
{
    this.tag = SymbolTable.encodeString(tag);
}

public MemoryAssociation(int size)
{
    this.size = size;
}
```

En la gramática, luego de crear un terceto, le seteamos una “memoria” que contiene todo lo anteriormente dicho:

```
Triplet t = semanticHelper.getTriplet(operand1, operand2, "+", listOfTriplets, sumCompatibilityTable);

if (t == null) {
    compiler.reportSemanticError("No se pueden sumar variables de distinto tipo", getTokenLocation($2));
    $$ = new ParserVal(new YACCInvalidDataUnit());
    break;
}

t.setMemoryAssociation(new MemoryAssociation(symbolTable.createAuxVar(t.getType()), t.getType().getSize(), t.getType()));
```

En la imagen anterior se puede ver en la última línea la sentencia *symbolTable.createAuxVar(t.getType())...* Este es un método nuevo de la clase *SymbolTable,* el cual, tiene como funcionalidad crear y agregar a dicha tabla la variable auxiliar asociada a cada terceto (@aux1, @aux2, @aux3, etc).

En la clase *SymbolTable* se agregó un atributo llamado *auxVarCounter* que funciona como un contador de las variables auxiliares que se van creando por cada terceto, ademas, se agrego el método mencionado en el párrafo anterior:

```
128     public String createAuxVar(DataType dataType)
129     {
130         String name = encodeString("@aux" + auxVarCounter++);
131         addNewEntry(new SymbolTableEntry(), name)
132             .setAttrib(AttribKey.DATA_TYPE, dataType)
133             .setAttrib(AttribKey.IS_AUX_VAR, value:true)
134             .setAttrib(AttribKey.MEMORY_ASSOCIATION, new MemoryAssociation(name, dataType.getSize(), dataType));
135         return name;
136     }
137
```

Este método agrega a la tabla de símbolos una nueva entrada que hace referencia a la variable auxiliar y además, la retorna (*encodeString()* es un método que simplemente le hace una modificación al String para que sea reconocible por el programa assembler).

Por último, en la lista de tercetos (*ListOfTriplets*), se agrego un atributo de tipo Hashmap llamado *tags,* que almacena un número entero correspondiente a la posición (índice) de un terceto y una lista de Strings que corresponden a las tag de ese terceto:

```
public class ListOfTriplets implements Iterable<Triplet> {
    private final Vector<Triplet> tripletList;
    private final HashMap<Integer, LinkedList<String>> tags;
```

Estas tags, se le asocian a un terceto en la gramática cuando hay un *IF* o un *DO UNTIL*. En el caso del primero, se le asocia el índice donde termina el cuerpo del if, junto con una nueva tag *IF:*

```
sentencia_if
    : IF '(' condicion_if_reserva ')' cuerpo_if END_IF
        {
            YACCDataUnit data3 = (YACCDataUnit)$3.obj;
            YACCDataUnit data5 = (YACCDataUnit)$5.obj;

            int jzToBackpatch = data3.reservedTriplet;

            String comp = data3.lexeme;

            int end_if = 1 + jzToBackpatch + data5.tripletQuantity;
            listOfTriplets.replaceTriplet(
                jzToBackpatch,
                new Triplet(
                    "NEG_CJUMP",
                    new TripletOperand(end_if, listOfTriplets),
                    null
                )
            );

            listOfTriplets.addTag(end_if, listOfTriplets.getNewIfTag());

            YACCDataUnit data = new YACCDataUnit();
            data.tripletQuantity = 1 + data3.tripletQuantity + data5.tripletQuantity;
            data.tokensData.add((LocatedSymbolTableEntry)$1.obj);
            data.firstTriplet = data3.reservedTriplet;

            $$ = new ParserVal(data);
        }
```

listofTriplets.getNewIfTag(), simplemente devuelve una nueva tag (@@@if_end_*NUMERONUEVO*):

```
public String getNewIfTag()
{
    return String.format(format:"@@@if_end_%d", incrementalCounter++);
}
```

mientras que en el *DO UNTIL*;

```
do_until
    : DO cuerpo_do UNTIL '(' condicion ')'
        {
            YACCDataUnit data2 = (YACCDataUnit)$2.obj;
            YACCDataUnit data5 = (YACCDataUnit)$5.obj;

            // Agregar salto condicional

            int bodyStartTriplet = listOfTriplets.getSize() - data2.tripletQuantity - data5.tripletQuantity;

            Triplet triplet = new Triplet("NEG_CJUMP", new TripletOperand(bodyStartTriplet, listOfTriplets), null);
            int tripletID = listOfTriplets.addTriplet(triplet);

            YACCDataUnit data = new YACCDataUnit();
            data.tokensData.add((LocatedSymbolTableEntry)$1.obj);
            data.tripletQuantity = 1 + data2.tripletQuantity + data5.tripletQuantity;
            data.firstTriplet = tripletID;

            listOfTriplets.addTag(bodyStartTriplet, "do_until_" + listOfTriplets.getIncrementalNum());

            $$ = new ParserVal(data);
        }
    ;
```

En los dos casos se utiliza el metodo *addTag* de la clase *ListOfTriplets,* la misma, se implementa de dicha forma:

```
public void addTag(int index, String tag)
{
    if (!tags.containsKey(index))
        tags.put(index, new LinkedList<>());
    tags.get(index).add(tag);
}
```

Mecanismo utilizado

En la carpeta x86 se encuentran dos archivos .java, “Translator.java” y “TripletTranslator.java”. El primero, se encarga de construir el “esqueleto” del programa .asm utilizando los métodos de la clase para hacer cada seccion del codigo ensamblador:

```
public String getAssemblyCode()
{
    return craftHeaderSection() +
           craftDataSection() +
           craftCodeSection();
}
```

Decidimos modularizar de esta manera, ya que el encabezado siempre es el mismo y las otras dos secciones asociadas a la declaración de variables y generación de código cambian en cada código.

craftHeaderSection() se encargan de construir el encabezado:

```
public String craftHeaderSection()
{
    return
        ".586\n" +
        ".model flat, stdcall\n\n" +
        "option casemap :none\n" +
        "include \\masm32\\include\\windows.inc\n" +
        "include \\masm32\\include\\kernel32.inc\n" +
        "include \\masm32\\include\\user32.inc\n" +
        "include \\masm32\\include\\masm32.inc\n" +
        "includelib \\masm32\\lib\\kernel32.lib\n" +
        "includelib \\masm32\\lib\\user32.lib\n" +
        "includelib \\masm32\\lib\\masm32.lib\n\n";
}
```

craftDataSection() ensambla las variables y las constantes de la tabla de símbolos al código assembler:

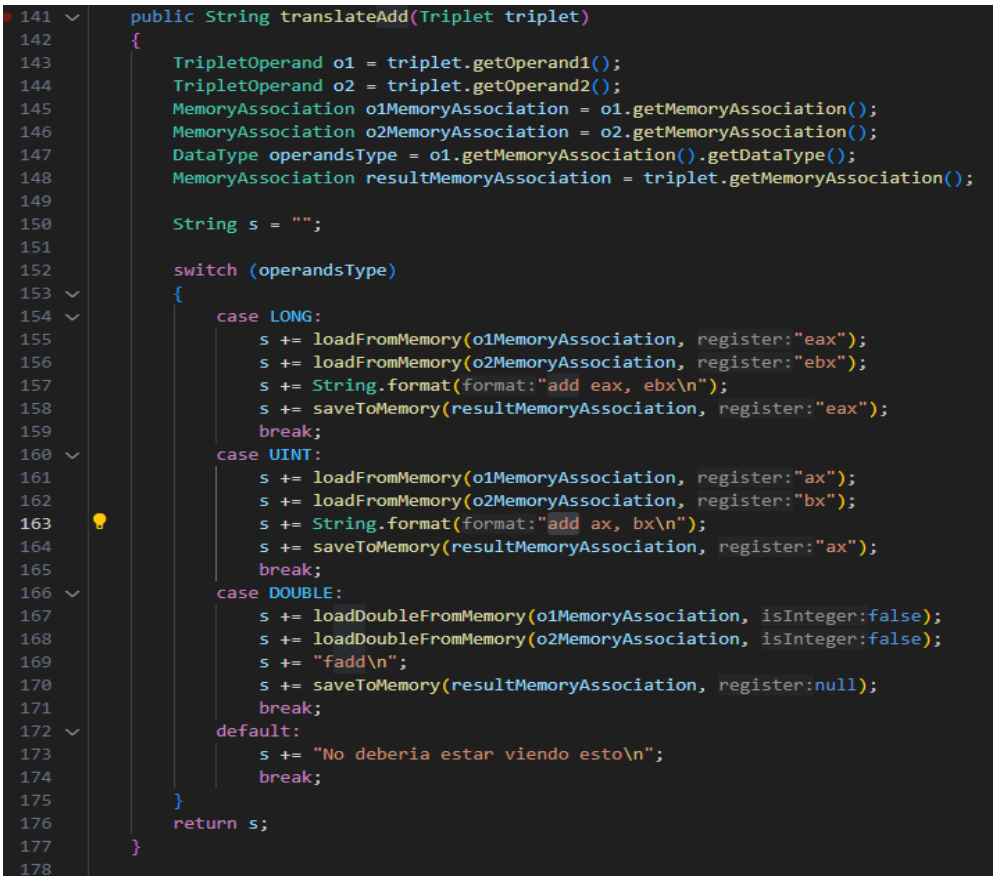
```java
public String craftDataSection()
{
    StringBuilder sb = new StringBuilder();

    sb.append(str:".data\n");

    sb.append(str:"___temp_double___ dq ?\n");
    sb.append(str:"__overflow_msg__ db 'Overflow detectado. Finaliza la ejecucion', 10, 0\n");

    List<String> constantsKeys = symbolTable.getConstantList();

    for (String constantKey : constantsKeys)
        sb.append(getConstantDeclarationLine(constantKey));

    List<String> varsKeys = symbolTable.getVarList();

    for (String varKey : varsKeys)
        sb.append(getVarDeclarationLine(varKey));

    sb.append(str:"\n");

    return sb.toString();
}
```

craftCodeSection() utiliza la clase TripletTranslator.java para hacer una traducción de los tercetos a código assembler dependiendo de la operación a realizar.

A continuación se muestra una porción de la función mencionada anteriormente para operaciones aritméticas, esta llama al método correspondiente de la clase *TripletTranslator* dependiendo de la operación del terceto:

```java
switch (triplet.getOperation())
{
    case "+":
        sb.append(tripletTranslator.translateAdd(triplet));
        break;
    case "=":
        sb.append(tripletTranslator.translateAssign(triplet));
        break;
    case "-":
        sb.append(tripletTranslator.translateSub(triplet));
        break;
    case "/":
        sb.append(tripletTranslator.translateDiv(triplet));
        break;
    case "*":
        sb.append(tripletTranslator.translateMul(triplet));
        break;
    case "PRINT":
        sb.append(tripletTranslator.translatePrint(triplet));
        break;
    case "RETURN":
        sb.append(str:"ret\n");
        break;
    case "JMP":
```

Suponiendo que el terceto tiene una operación de suma, se invoca :



```
141  public String translateAdd(Triplet triplet)
142  {
143      TripletOperand o1 = triplet.getOperand1();
144      TripletOperand o2 = triplet.getOperand2();
145      MemoryAssociation o1MemoryAssociation = o1.getMemoryAssociation();
146      MemoryAssociation o2MemoryAssociation = o2.getMemoryAssociation();
147      DataType operandsType = o1.getMemoryAssociation().getDataType();
148      MemoryAssociation resultMemoryAssociation = triplet.getMemoryAssociation();
149
150      String s = "";
151
152      switch (operandsType)
153      {
154          case LONG:
155              s += loadFromMemory(o1MemoryAssociation, register:"eax");
156              s += loadFromMemory(o2MemoryAssociation, register:"ebx");
157              s += String.format(format:"add eax, ebx\n");
158              s += saveToMemory(resultMemoryAssociation, register:"eax");
159              break;
160          case UINT:
161              s += loadFromMemory(o1MemoryAssociation, register:"ax");
162              s += loadFromMemory(o2MemoryAssociation, register:"bx");
163              s += String.format(format:"add ax, bx\n");
164              s += saveToMemory(resultMemoryAssociation, register:"ax");
165              break;
166          case DOUBLE:
167              s += loadDoubleFromMemory(o1MemoryAssociation, isInteger:false);
168              s += loadDoubleFromMemory(o2MemoryAssociation, isInteger:false);
169              s += "fadd\n";
170              s += saveToMemory(resultMemoryAssociation, register:null);
171              break;
172          default:
173              s += "No deberia estar viendo esto\n";
174              break;
175      }
176      return s;
177  }
178
```

Esta función (*translateAdd(t: Triplet)*) compara el tipo de datos que se están sumando, para traducir la suma al código assembler correspondiente.

Este funcionamiento se aplica a todas las demás operaciones (resta, multiplicación y división), asignaciones, sentencias de control, etc.

Las funciones *LoadFromMemory()* cargan en el registro dado la variable auxiliar del terceto, mientras que las funciones *saveToMemory()*, cargan en la variable auxiliar del terceto, el registro que guarda el resultado de la operación.

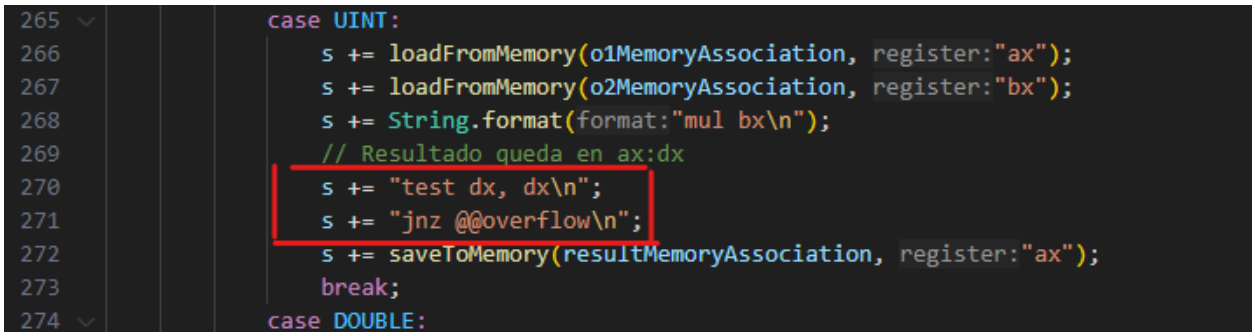
Controles en tiempo de ejecución

Recordando los chequeos en tiempo de ejecución que nos tocaron:

- **Overflow en productos de enteros:**

  " El código Assembler deberá controlar el resultado de la operación indicada, para los tipos de datos enteros asignados al grupo. Si el mismo excede el rango del tipo del resultado, deberá emitir un mensaje de error y terminar "

  Para realizar este chequeo, utilizamos dos instrucciones en código assembler a la hora de multiplicar dos enteros. En la clase *TripletTranslator.java,* en la sección que se encarga de las multiplicaciones se incorporó lo siguiente:



```
265         case UINT:
266             s += loadFromMemory(o1MemoryAssociation, register:"ax");
267             s += loadFromMemory(o2MemoryAssociation, register:"bx");
268             s += String.format(format:"mul bx\n");
269             // Resultado queda en ax:dx
270             s += "test dx, dx\n";
271             s += "jnz @@overflow\n";
272             s += saveToMemory(resultMemoryAssociation, register:"ax");
273             break;
274         case DOUBLE:
```

  La primera instrucción (*test dx, dx*) realiza una operación AND entre el registro dx y sí mismo, ya que su propósito es actualizar la flag del procesador según el resultado de la operación lógica (bit 0 o 1). Luego la segunda instrucción (*jnz @@overflow*) significa "Jump is not zero", básicamente, si la flag no está establecida (osea que el resultado de la operación AND no fue 0), el programa salta a la etiqueta de *@@overflow*, en caso contrario, el programa continúa normalmente.

  El caso en el que la multiplicación exceda el límite dado, se mostrará por pantalla el siguiente error para el código:

```
{
    UINT var1,
    UINT var2,
    UINT var3,

    var2 = 10000_ui,

    var1 = 20000_ui,

    var3 = var1 * var2,

}
```

```
Salida del programa:

Overflow detectado. Finaliza la ejecucion
```

- **Resultados negativos en restas de enteros sin signo:**

  " El código Assembler deberá controlar el resultado de la operación indicada. Este control se aplicará a operaciones entre enteros sin signo. En caso que una resta entre datos de este tipo arroje un resultado negativo, deberá emitir un mensaje de error y termina"

  Se controla de forma similar al chequeo anterior, a la hora de restar enteros sin signos (UINT), en vez de hacer un *test,* hacemos un *cmp,* tal que, si el segundo valor es mayor que el primero, se salta a la etiqueta *@@overflow:*

```
case UINT:
    s += loadFromMemory(o1MemoryAssociation, register:"ax");
    s += loadFromMemory(o2MemoryAssociation, register:"bx");
    s += String.format(format:"cmp ax, bx\n");
    s += String.format(format:"jb @@overflow_resta\n");
    s += String.format(format:"sub ax, bx\n");
    s += saveToMemory(resultMemoryAssociation, register:"ax");
    break;
case DOUBLE:
```

El caso en el que la resta de negativa, se mostrará por pantalla el siguiente error, para el siguiente código:

```
{
    UINT var1,
    UINT var2,
    UINT var3,

    var2 = 5_ui,

    var1 = 2_ui,

    var3 = var1 - var2,

}
```

```
Salida del programa:

Overflow en resta de UINTs detectado. Finaliza la ejecucion
```

Casos de prueba en tiempo de ejecución

- **Hello world:** Programa básico para los compiladores. Dado el siguiente código, imprimir por pantalla “Hello world”.

```
{
    PRINT %HELLO WORLD%,
}
```

```
Salida del programa:

HELLO WORLD
```

- **TOD:** Para el siguiente código:

```
1   {
2       UINT var1,
3       var1 = 7_ui,
4
5       DOUBLE var2,
6       var2 = 1.51,
7
8       DOUBLE var3,
9
10      var3 = var1 + var2,
11  }
```

Se espera que se muestre el siguiente error, ya que se suman variables de diferentes tipos:

```
[Semantico: Linea 10] No se pueden sumar variables de distinto tipo

Hubo errores en el parsing
```

Luego, se corrige, incluyendo la cláusula *TOD:*

```
1   {
2       UINT var1,
3       var1 = 7_ui,
4
5       DOUBLE var2,
6       var2 = 1.51,
7
8       DOUBLE var3,
9       DOUBLE var4,
10
11      var4 = TOD(var1),
12
13      var3 = var4 + var2,
14
15      PRINT var3,
16  }
```

```
Salida del programa:

8.510000
```

- **Ejemplo de composición:** Para el código:

```
** Se espera que ande BIEN
{

    CLASS trabajo
    {
        UINT id,
    },

    CLASS persona
    {
        STRING nombre,
        trabajo,

        VOID imprimir_persona()
        {
            PRINT %nombre%,
            PRINT trabajo.id,
        },
    },

    persona p1,

    p1.trabajo.id = 25_ui,
    p1.imprimir_persona(),
}
```

La salida es:

```
Lista de tercetos:
[0] 'JMP' [Triplet: 4] [null] Type: null [@aux5] Mem: null | Info: null
[1] 'PRINT' [Entrada en la tabla de simbolos: %nombre%] [null] Type: null [@aux2] Mem: null | Info: ProcedureInit: true | ProcedureEnd: false
[2] 'PRINT' [Entrada en la tabla de simbolos: id] [null] Type: null [@aux3] Mem: null | Info: null
[3] 'RETURN' [null] [null] Type: null [@aux4] Mem: null | Info: null
[4] 'THIS' [Entrada en la tabla de simbolos: p1] [null] Type: null [@aux6] Mem: null | Info: null
[5] '=' [Entrada en la tabla de simbolos: id] [Entrada en la tabla de simbolos: 25_ui] Type: null [@aux7] Mem: null | Info: null
[6] 'THIS' [Entrada en la tabla de simbolos: p1] [null] Type: null [@aux8] Mem: null | Info: null
[7] 'CALL' [Entrada en la tabla de simbolos: imprimir_persona] [null] Type: null [@aux9] Mem: null | Info: null
[8] 'END' [null] [null] Type: null [@aux10] Mem: null | Info: null

Memory association de o1:
offsetRespectEBP: false Offset: '4' | DataType -> 'UINT'
Microsoft (R) Macro Assembler Version 6.14.8444
Copyright (C) Microsoft Corp 1981-1997.  All rights reserved.

 Assembling: program.asm
Microsoft (R) Incremental Linker Version 5.12.8078
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

Salida del programa:

nombre25
```

- **Overflow en sumas de datos de punto flotante:**

Para resolver este chequeo, incluimos las siguientes líneas de código cuando se detecta una suma de datos de tipo punto flotante o double:

```
case DOUBLE:
    s += loadDoubleFromMemory(o1MemoryAssociation, isInteger:false);
    s += loadDoubleFromMemory(o2MemoryAssociation, isInteger:false);
    s += "fadd\n";

    s += "fst qword ptr [___temp_double___]\n";
    s += "mov eax, dword ptr [___temp_double___]\n";
    s += "mov edx, dword ptr [___temp_double___ + 4]\n";

    s += "cmp eax, 0\n";
    s += "jne @@no_overflow_suma\n";
    s += "and edx, 7FF00000h\n";
    s += "cmp edx, 7FF00000h\n";
    s += "jne @@no_overflow_suma\n";
    s += "jmp @@overflow_suma\n";
    s += "@@no_overflow_suma:\n";

    s += saveToMemory(resultMemoryAssociation, register:null);
    break;
default:
```

Esto nos permite chequear que el resultado de la suma no exceda los límites establecidos por la cátedra.

Para el código:

```
{
    DOUBLE a; b; c,

    a = 1.7976931348623157D+308,

    b = 1.2325346436436234D+300,

    c = a + b,

    PRINT %c es igual a %,
    PRINT c,
}
```

La salida es:

```
Salida del programa:

Overflow en suma de DOUBLEs detectado. Finaliza la ejecucion
```

En la siguiente imagen se aprecia el código generado por el compilador para el chequeo de overflow en tiempo de ejecución:

```
78   fadd
79   fst qword ptr [___temp_double___]
80   mov eax, dword ptr [___temp_double___]
81   mov edx, dword ptr [___temp_double___ + 4]
82   cmp eax, 0
83   jne @@no_overflow_suma
84   and edx, 7FF00000h
85   cmp edx, 7FF00000h
86   jne @@no_overflow_suma
87   jmp @@overflow_suma
88   @@no_overflow_suma:
89   fstp qword ptr [__@aux0]
```

- **Detección de división por cero:**

En estos casos, se muestra el código generado para chequear la división por cero en enteros sin signo. (Para long es lo mismo, pero usando los registros eax).

```
{
    DOUBLE a; b; c,

    a = 1.5,
    b = 0.0,

    c = a / b,

    PRINT c,
}
```

```
fdiv
fstsw ax
and eax, 4h
cmp eax, 4h
jz @@zero_div_error
```

```
{
    UINT a; b,
    DOUBLE c,

    a = 2_ui,
    b = 0_ui,

    c = a / b,

    PRINT %esto nunca se ejecuta%,
}
```

```
mov ax, word ptr [_b_global]
cmp ax, 0
jz @@zero_div_error
```

El error que se imprime por pantalla para ambos casos es el siguiente:

```
Salida del programa:

Division por cero detectada. Finaliza la ejecucion
```

Se tuvo que agregar las siguientes lineas de codigo cuando se detecta una división:

```
switch (operandsType)
{
    case LONG:
    case UINT:

        String reg = (operandsType == DataType.LONG) ? "eax" : "ax";
        s += loadFromMemory(o2MemoryAssociation, reg);
        s += String.format(format:"cmp %s, 0\n", reg);
        s += "jz @@zero_div_error\n";
        // Primero el dividendo (va en ST1)
        s += loadDoubleFromMemory(o1MemoryAssociation, isInteger:true);
        s += loadDoubleFromMemory(o2MemoryAssociation, isInteger:true);
        s += "fdiv\n";
        s += saveToMemory(resultMemoryAssociation, register:null);
        break;
    case DOUBLE:
        s += loadDoubleFromMemory(o1MemoryAssociation, isInteger:false);
        s += loadDoubleFromMemory(o2MemoryAssociation, isInteger:false);
        s += "fdiv\n";
        s += "fstsw ax\n";
        s += "and eax, 4h\n";
        s += "cmp eax, 4h\n";
        s += "jz @@zero_div_error\n";
        s += saveToMemory(resultMemoryAssociation, register:null);
        break;
    default:
        s += "No deberia estar viendo esto\n";
        break;
}
```

- **Tema 21: Forward declaration**

Este chequeo lo resolvimos haciendo uso de una lista que almacena las referencias a clases que no se encontraron en el momento de definir una variable. Estas referencias se almacenan en una estructura de tipo "*ForwardData*".

En el momento en que se declara una clase, también se actualiza la lista para buscar referencias a la posible nueva clase declarada, y si se encuentran usos "*forward*", se quitan de la lista.

Al final del proceso semántico y de parsing, se comprueba si la lista está vacía; en el caso de no estarlo significa que hay un error (referencia a una clase que nunca se declaró).

A continuación se muestran ejemplos del funcionamiento de este mecanismo:

En este primer ejemplo vemos el funcionamiento básico de este mecanismo, donde se declara una clase pero se define después, por lo tanto, no deberia haber ningun problema:

```
{
    cc c1,

    CLASS cc{
        UINT a,
    },

}
```

```
 Assembling: program.asm
Microsoft (R) Incremental Linker Version 5.12.8078
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

Salida del programa:


PS C:\Users\facun\OneDrive\Escritorio\compilador>
```

Ahora vemos que error imprime si nunca se define la clase:

```
{
    cc c1,

}
```

```
Clave: >= | TokenID: 272
Hubo referencias forward no encontradas
Corrija los errores para continuar con la compilacion
PS C:\Users\facun\OneDrive\Escritorio\compilador>
```

En este ejemplo se puede ver como se declara *clase_laburo* dentro de *clase_persona* pero *clase_laburo* todavía no está definida, el programa no lo detecta como un problema y compila normalmente.

```
{
    CLASS clase_persona
    {
        STRING nombre,
        clase_laburo laburo,
    },

    CLASS clase_laburo
    {
        UINT id_empleado,
    },

    PRINT %ejecucion correcta%,
}
```

```
Salida del programa:

ejecucion correcta


Process finished with exit code 0
```

Por último, vemos como se declara la *clase_inexistente,* pero no se define nunca, por lo tanto genera un error de forward declaration:

```
{
    CLASS clase_persona
    {
        STRING nombre,
        clase_inexistente inex,
    },

    PRINT %esto nunca se ejecuta%,
}
```

```
Clave: >= | TokenID: 272
Hubo referencias forward no encontradas
Corrija los errores para continuar con la compilacion
```

Conclusión

Al principio de la cursada no sabíamos cómo se podría traducir un código fuente a un código ejecutable, pero a medida que avanzabamos en este trabajo empezamos a comprender cómo se implementa un compilador, profundizando en su estructura y sus partes, tuvimos que informarnos sobre ciertos temas, tomar decisiones de implementación y aprender a usar herramientas nuevas. Luego de terminar las dos últimas etapas del compilador, logramos interiorizar más el funcionamiento de un compilador. Además, tuvimos que sortear muchos obstáculos, ya sea corrigiendo la primera entrega o haciendo que funcione correctamente la segunda.

Para finalizar, aunque no hayamos hecho un compilador capaz de soportar la totalidad de operaciones requeridas, la experiencia de crear un software complejo como este, nos puso a prueba y hemos podido mejorar nuestras habilidades y capacidades adquiriendo conocimientos nuevos, desde ya, estamos agradecidos por el feedback recibido y los temas nuevos aprendidos.