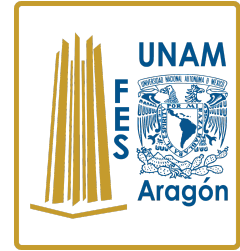




UNIVERSIDAD NACIONAL
AUTÓNOMA DE MÉXICO



UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO
FACULTAD DE ESTUDIOS SUPERIORES ARAGÓN

Ingeniería en Computación

Compiladores

Proyecto Compiladores

Prof. Martin Romero Ugalde

Integrantes:

ESTRADA BARBOSA ERICK EDUARDO

HERNANDEZ SOLEDAD ANGEL AGUSTÍN

IBIETA OLVERA JAZMIN SARAI

PRADO GONZÁLEZ JOSÉ LEONARDO

ÍNDICE

INTRODUCCIÓN	4
CAPÍTULO 1: GENERALIDADES	4
Sobre:	4
Objetivo:	4
Requerimientos del sistema:	4
Ayuda adicional:	5
Aho, A. V. (1990). COMPILADORES PRINCIPIOS TÉCNICAS Y HERRAMIENTAS (1a. ed.). WILMINGTON: ADDISON WESLEY IBEROAMERICANA.	5
CAPÍTULO 2: LENGUAJE	5
Sobre el lenguaje:	5
Palabras reservadas y detalles específicos:	5
Estructura general de un programa:	8
CAPÍTULO 3: COMPILADOR	12
Sobre el compilador:	12
Vista general:	12
Estructura del proyecto	12
Capítulo 4 Análisis Semántico.	13
¿Qué es el análisis semántico?	13
Clase del analizador semantico.java	13
CAPÍTULO 5, Código Intermedio	14
¿Qué es el código intermedio?	14
Generación de Código Intermedio	14
Clase que Genera Código Intermedio	14
Botón Tripletas	17
Resultados	17
CAPÍTULO 6, Optimización	18
¿Qué es la Optimización de Código?	18
Optimización de Código	19
Código que Optimiza Código	19

CAPÍTULO 7, Código Objeto	23
¿Qué es el código Objeto?	23
Generación de código Objeto	23
Código que Genera Código Objeto	23
Botón Código Objeto	26
Resultados	26
Apéndices:	29
Apéndice A:	29
Apéndice B:	31
B.1: Ejemplo Funcional:	31
B.2: VideoTutorial:	37
Apéndice C. Para descargar e instalar.	38

INTRODUCCIÓN

Este documento contiene las primeras especificaciones para el desarrollo del anteproyecto “CodeChair”, con el objetivo principal de mejorar la calidad de vida general de las personas con discapacidad física o aquejadas. Aspira a hacerlo implementando ciertas características y capacidades a una silla de ruedas. Estas características incluyen la capacidad de mover la silla de ruedas de forma autónoma a un lugar específico, proporcionando y recibiendo información sobre el entorno del usuario, así como facilitar las interacciones con los electrodomésticos con la ayuda de dispositivos tecnológicos inalámbricos.

CAPÍTULO 1: GENERALIDADES

Sobre:

El principal enfoque de nuestro proyecto consta en la automatización de una silla de ruedas común y corriente, con lo que lograremos conseguir que el usuario de la silla de ruedas tenga una experiencia más cómoda al permitirle ciertas funciones que le facilitarán la realización de necesidades básicas, tales como: movimiento de la silla, activación de freno, activación de alertas para facilitar el traslado del usuario, equipo de iluminación para facilitar la movilidad, envío de mensaje de auxilio a contacto en caso de incidente, desplazamiento a ciertas áreas definidas por el usuario entre otras funciones.

Objetivo:

Diseñar una guía para usuarios comunes, que sea capaz de sintetizar las instrucciones y explicaciones para el correcto funcionamiento del proyecto presentado anteriormente, el cual emplea un lenguaje de programación creado por nosotros mismos.

Requerimientos del sistema:

Procesador	Intel Pentium 2 226MHz.
RAM	4 GB
Espacio disponible en disco	300 MB
Sistema Operativo	Windows 7, Windows 10
Ambiente de Ejecución (JRE + Update)	126 MB de espacio disponible en disco.

JDK (Herramientas de desarrollo + código fuente)	272 MB de espacio disponible en disco.

Ayuda adicional:

Aho, A. V. (1990). *COMPILADORES PRINCIPIOS TÉCNICAS Y HERRAMIENTAS* (1a. ed.). WILMINGTON: ADDISON WESLEY IBEROAMERICANA.

CAPÍTULO 2: LENGUAJE**Sobre el lenguaje:**

El compilador maneja un lenguaje simple y poco preciso el cual nos permite principalmente llevar a cabo la programación del sistema utilizando estructuras de control y operaciones que nos sirvan para este propósito.

Palabras reservadas y detalles específicos:

El programa cuenta con 11 tipos de tokens y 25 palabras reservadas.

Tokens.

Token	Descripción	Ejemplo
Palabras Reserva	Son las cadenas que identifican las funciones reservadas	IMPORT, STRING, INT
Operadores	Caracteres +, -, /, *, =	+ ó - ó / ó * ó =
Comparación	Caracteres ==, !=, <, >, >=, <=	== ó != ó < ó > ó >= ó <=
Separadores	Caracteres ., :, ;, ' "	. ó , ó : ó ; ó ' ó "
AbiertoCerrado	Caracteres (,), {, }, [,]	(ó) ó { ó } ó [ó]

COMPILADOR CHAIRCODE

Pasos	Caracteres ++, -- (incremento decremento)	++ ó --
id	Letra seguida por letras y numeros	Contador1, ResultadoMul
número	Cualquier dígito	1, 4892, 57, 192
SaltoLinea	Caracteres \n indicando el salto de línea	\n
Comentario	Comentario de línea indicado por //	//Comentario, // Ola
Error	Cualquier caracter no valido	ñ, , ~

Palabras Reservadas:

Palabra reservada	Descripción	Ejemplo
IMPORT	Cadena mayus y min I,M,P,O,R,T	IMPORT ó import

DEF	Cadena mayus y min D, E, F	DEF ó def
CLASS	Cadena mayus y min C, L, A, S, S	CLASS ó class
IF	Cadena mayus y min I, F	IF ó if

COMPILADOR CHAIRCODE

ELSE	Cadena mayus y min E, L, S, E	ELSE ó else
FOR	Cadena mayus y min F, O, R	FOR ó for
IN	Cadena mayus y min I, N	IN ó in
RANGE	Cadena mayus y min R, A, N, G, E	RANGE ó range
SELF	Cadena mayus y min S, E, L, F	SELF ó self
WHILE	Cadena mayus y min W, H, I, L, E	WHILE ó while
TRY	Cadena mayus y min T, R, Y	TRY ó try
EXCEPT	Cadena mayus y min E, X, C, E, P, T	EXCEPT ó except
RETURN	Cadena mayus y min R, E, T, U, R, N	RETURN ó return
BREAK	Cadena mayus y min B, R, E, A, K	BREAK ó break
NEXT	Cadena mayus y min N, E, X, T	NEXT ó next
INPUT	Cadena mayus y min I, N, P, U, T	INPUT ó input
PRINT	Cadena mayus y min P, R, I, N, T	PRINT ó print
INT	Cadena mayus y min I, N, T	INT ó int
FLOAT	Cadena mayus y min F, L, O, A, T	FLOAT ó float

STRING	Cadena mayus y min S, T, R, I, N, G	STRING ó string
POWER	Cadena mayus y min P, O, W, E, R	POWER ó power
SQRT	Cadena mayus y min S, Q, R, T	SQRT ó sqrt
AND	Cadena mayus y min A, N, D	AND ó and
OR	Cadena mayus y min O, R	OR ó or
NOT	Cadena mayus y min N, O, T	NOT ó not

Estructura general de un programa:

Reglas del Lenguaje:

1. Las palabras reservadas deben ser rigurosamente escritas ó todas en mayúsculas ó todas en minúsculas para considerarse como tal.
2. Los caracteres inválidos (No añadidos al alfabeto) son catalogados como caracteres inválidos
3. Los identificadores deben comenzar por una letra mayúscula o minúscula, seguida de letras y/o números.
4. Todo carácter abierto ((, {, [) deberá tener su carácter para cerrar.
5. Los operadores, deben de ser seguidos de números menos el ++(incremento) y el -- (decremento).

Patrones Válidos:

```

/* Variables básicas de comentarios y espacios */
TerminadorDeLinea = \r|\n|\r\n
EntradaDeCaracter = [^\r\n]
EspacioEnBlanco = {TerminadorDeLinea} | [ \t\f]
ComentarioTradicional = "/*" [^*] ~"*/" | "/*" "*" + "/"
FinDeLineaComentario = "//" {EntradaDeCaracter}* {TerminadorDeLinea}?
ContenidoComentario = ( [^*] | \*+ [^/*] ) *
ComentarioDeDocumentacion = "/*" {ContenidoComentario} "*" + "/"

```


COMPILADOR CHAIRCODE

```
/* Comentario */  
Comentario = {ComentarioTradicional} | {FinDeLineaComentario} |  
{ComentarioDeDocumentacion}
```

```
/* Identificador */  
Letra = [A-Za-zÑñ_ÁÉÍÓÚáéíóúÜü]  
Digito = [0-9]  
Identificador = {Letra}{Letra}|{Digito})*  
Numero = {Digito} ({Digito})*
```

```
/* Número */  
Numero = 0 | [1-9][0-9]*  
%%
```

```
/* Comentarios o espacios en blanco */  
{Comentario}|{EspacioEnBlanco} { /*Ignorar*/ }
```

```
/*Numero*/  
REAL {Numero} "." {Numero}  
NUMERO {Numero}
```

```
/*Operadores*/  
"+" "SUMA"  
"- " "RESTA"  
"/" "DIVISION"  
"*" "MULTIPLICACION"
```

```
/*Logicos*/  
"==" "IGUAL"  
"!=" "DIFERENTE"  
">" "MAYORQUE"  
"<" "MENORQUE"  
">=" "MAYORIGUALQUE"  
"<=" "MENORIGUALQUE"
```

```
/*puntuacion*/  
"." "PUNTO"  
"," "COMA"  
":" "DOS PUNTOS"  
";" "PUNTOCOMA"  
\" "COMILLASIMPLE"
```

COMPILADOR CHAIRCODE

```
\ " [a-zA-Z0-9_.-]* \ " "CADENA"  
"=" "ASIGNACION"
```

```
\ " "COMILLADOBLE"  
\ ( "PARENTESISABIERTO"  
\ ) "PARENTESIS CERRADO"  
\ { "LLAVEABIERTO"  
\ } "LLAVECERRADO"  
\ [ "CORCHETEABIERTO"  
\ ] "CORCHETECERRADO"
```

```
"++" "INCREMENTO"  
"--" "DECREMENTO"
```

```
/*Palabras reservadas*/  
"IMPORT" | "import" | "Import"  
"DEF" | "def" | "Def"  
"CLASS" | "class" | "Class"  
"IF" | "if" | "If"  
"ELSE" | "else" | "Else"  
"FOR" | "for" | "For"  
"IN" | "in" | "In"  
"RANGE" | "range" | "Range"  
"SELF" | "self" | "Self"  
"WHILE" | "while" | "While"  
"TRY" | "try" | "Try"  
"EXCEPT" | "except" | "Except"  
"RETURN" | "return" | "Return"  
"BREAK" | "break" | "Break"  
"NEXT" | "next" | "Next"  
"INPUT" | "input" | "Input"  
"OUTPUT" | "output" | "Output"  
"PRINT" | "print" | "Print"  
"INT" | "int" | "Int"  
"FLOAT" | "float" | "Float"  
"STRING" | "string" | "String"  
"TRUE" | "true" | "True"  
"FALSE" | "false" | "False"  
"POWER" | "power" | "Power"  
"SQRT" | "sqrt" | "Sqrt"
```

COMPILADOR CHAIRCODE

"AND" | "and" | "And"

"OR" | "or" | "Or"

"NOT" | "not" | "Not"

"BEGIN" | "begin" | "Begin"

"END" | "end" | "End"

/* IDs */

{Identificador} "ID"

CAPÍTULO 3: COMPILADOR

Sobre el compilador:

Un compilador es un programa que traduce un programa escrito en lenguaje fuente y produce otro equivalente escrito en un lenguaje destino. Lenguaje de alto nivel. Por ejemplo: C, Pascal, C++.

Vista general:

Estructura del proyecto

El compilador se compone de 4 carpetas Source Packages, Test Packages, Libraries y Test Librares. En la carpeta Source Packages, se cuenta con dos paquetes java: <default package> y Multimedia. En el segundo, son guardadas todas las imágenes que se emplean en el compilador para ser más visualmente atractivo al usuario.

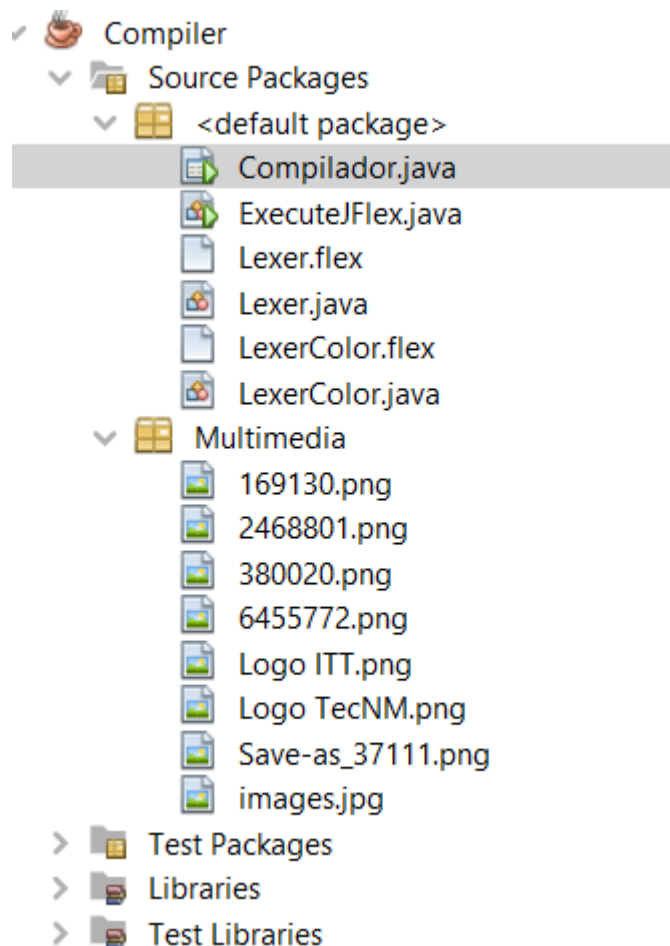


Figura 1 Estructura del Proyecto

Por el otro lado, en el paquete <default package> son guardadas las clases con extensión .java y .flex las cuales corresponden al compilador y permiten que funcione de manera correcta. En total,

contando aquellas que fueron programadas tanto para el análisis semántico como para el código intermedio, se cuentan con veintisiete clases.

Dentro de la carpeta Libraries se encuentran las librerías necesarias para el buen funcionamiento del compilador. Finalmente las carpetas Test Package y Test Libraries se encuentran vacías.

Capítulo 4 Análisis Semántico.

¿Qué es el análisis semántico?

La fase de análisis semántico revisa el programa fuente para tratar de encontrar errores semánticos y reúne la información sobre los tipos para la fase posterior de generación de código. En ella se utiliza la estructura jerárquica determinada por la clase de análisis sintáctico para identificar los operadores y operandos de expresiones y proposiciones.

Clase del analizador semantico.java

Después de haber guardado anteriormente las producciones generadas en el análisis sintáctico, estas se tratan de diferente manera dependiendo de las instrucciones que guardan.

Análisis de Identificadores:

Lo primero que hace esta parte es generar un HashMap que contiene los tipos de dato que manejamos en el lenguaje, además de los valores que aceptan estos tipos de datos; esto para comparar las asignaciones más adelante.

Después, se va iterando el arreglo de producciones **identProd**, que es el que contiene las producciones que se generan al declarar una variable. Los valores que están en la posición 1 son los ID (nombres de las variables), y los que están en la posición 0 son los tipos de datos. Estos se van agregando al HashMap de identificadores, excepto en los casos donde ya haya una variable con el mismo nombre en este HashMap, lo que genera un error.

Cuando este proceso termina, se itera el arreglo **asigProd**, que almacena las líneas de código donde hay una asignación de valores a una variable. Esta vez, los ID están en la posición 0, ya que estas instrucciones no especifican el tipo de dato. Entonces, se revisa en el HashMap de identificadores mencionado anteriormente si esta variable está declarada, además de que si el tipo de dato asignado coincide con el que debe almacenar la variable, usando el HashMap tiposDatos del principio. El arreglo **asigProdConID** tiene la misma función, con la excepción de que el valor asignado es también una variable, por lo que se debe revisar el tipo de dato de ambas variables.

Cabe destacar que las producciones en identProd están en la forma:

```
TIPODATO ID [ = VALOR ] ; //(valor opcional)
```

COMPILADOR CHAIRCODE

Y las de asigProd:

ID = VALOR ;

```
HashMap<String,String> tiposDatos = new HashMap<>();
tiposDatos.put("NUMERO", "INT");
tiposDatos.put("REAL", "FLOAT");
tiposDatos.put("CADENA", "STRING");
tiposDatos.put("TRUE", "BOOLEAN");
tiposDatos.put("FALSE", "BOOLEAN");
int i = 0;
for(Production id: identProd){
    if (!identificadores.containsKey(id.lexemeRank(1))){
        identificadores.put(id.lexemeRank(1), id.lexicalCompRank(0));
        i++;
    }
    else {
        errors.add(new ErrorLSSL(1,"Error semántico: Ya existe un
identificador llamado "+id.lexemeRank(1),id,true));
    }
}
System.out.println(Arrays.asList(identificadores)); // muestra
identificadores
for (Production id: asigProd){
    if (!identificadores.containsKey(id.lexemeRank(0))){
        errors.add(new ErrorLSSL(1,"Error semántico: Variable
\""+id.lexemeRank(0)+"\" no declarada. [#, %]",id,true));
    }
    else{
        if
(!identificadores.get(id.lexemeRank(0)).equals(tiposDatos.get(id.lexicalCompRa
nk(2)))){
            errors.add(new ErrorLSSL(1,"Error semántico : Variable
\""+id.lexemeRank(0)+"\" es de tipo "+identificadores.get(id.lexemeRank(0)) +
" [#, %]",id,true));
        }
    }
}
for (Production id: asigProdConID){
    if
(!identificadores.containsKey(id.lexemeRank(0))||!identificadores.containsKey(
id.lexemeRank(2))){
        errors.add(new ErrorLSSL(1,"Error semántico: Variable no
declarada. [#, %]",id,true));
    }
    else{
        if
(!identificadores.get(id.lexemeRank(0)).equals(identificadores.get(id.lexemeRa
nk(2)))){
            errors.add(new ErrorLSSL(1,"Error semántico : Variable
```

```

\""+id.lexemeRank(0)+"\" es de tipo "+identificadores.get(id.lexemeRank(0)) +
" [#, %]",id,true));

    }
}

}

```

Análisis de condiciones:

Las siguientes instrucciones que revisa el analizador son las condiciones, donde se comparan unos valores con otros. Estas producciones se almacenan en los arreglos ***compaProdIzq***, ***compaProdDer*** y ***compaProdDoble***, de acuerdo con la posición que tiene la variable en la instrucción: a la izquierda del operador, a la derecha o en ambos lados. Estas vienen en la forma VALOR1 OPERADORLOGICO VALOR2. Cabe destacar que los errores de comparación entre dos valores que no son variables (por ejemplo, "HOLA" > 1;) se marcan como error desde el análisis sintáctico.

En caso de que solo haya una variable, se revisa si esta fue declarada, y si es así, podrá ser analizada. Primero revisa si contiene un valor tipo cadena, lo que en nuestro lenguaje haría imposible su comparación con cualquier otro dato. De manera similar, los valores booleanos sólo pueden ser comparados con otros del mismo tipo, y con operadores de Igual o Diferente.

El compilador también valida que los valores numéricos sólo se puedan comparar con otros valores numéricos u operaciones.

En caso de que ambos valores a comparar sean variables, se realiza el mismo procedimiento, sólo que se validan y comparan los tipos de ambos datos desde el HashMap de identificadores.

```

for (Production id: compaProdIzq){

    if (!identificadores.containsKey(id.lexemeRank(0))){
        errors.add(new ErrorLSSL(1,"Error semántico: Variable
\""+id.lexemeRank(0)+"\" no declarada. [#, %]",id,true));
    }
    else{
        if (identificadores.get(id.lexemeRank(0)).matches("STRING")){
            errors.add(new ErrorLSSL(1,"Error semántico : Variable
\""+id.lexemeRank(0)+"\" es de tipo STRING, imposible comparar [#,
%]",id,true));
        }
        if
(identificadores.get(id.lexemeRank(0)).matches("BOOLEAN")&&!id.lexicalCompRank
(1).matches("IGUAL|DIFERENTE")){
            errors.add(new ErrorLSSL(1,"Error semántico : Variable
\""+id.lexemeRank(0)+"\" es de tipo BOOLEAN, sólo posible comparar con
operadores IGUAL y DIFERENTE [#, %]",id,true));
        }
        if
(identificadores.get(id.lexemeRank(0)).matches("BOOLEAN")&&!id.lexicalCompRank
(2).matches("TRUE|FALSE")){
            errors.add(new ErrorLSSL(1,"Error semántico : Variable

```

```

\""+id.lexemeRank(0)+"\" es de tipo BOOLEAN, sólo posible comparar con valores
booleanos [#, %]",id,true));
    }
    if
    (identificadores.get(id.lexemeRank(0)).matches("INT|FLOAT")){
        if(!id.lexicalCompRank(2).matches("NUMERO|REAL|ID")){
            errors.add(new ErrorLSSL(1,"Error semántico : Valor
numérico de variable \""+id.lexemeRank(0)+"\" no se puede comparar con valor
no numérico [#, %]",id,true));
        }
    }

}

}

}

// FOR COMPAPRODIZQ

for (Production id: compaProdDer){

    if (!identificadores.containsKey(id.lexemeRank(2))){
        errors.add(new ErrorLSSL(1,"Error semántico: Variable
"+id.lexemeRank(2)+" no declarada. [#, %]",id,true));
    }
    else{
        if (identificadores.get(id.lexemeRank(2)).matches("STRING")){
            errors.add(new ErrorLSSL(1,"Error semántico : Variable
\""+id.lexemeRank(2)+"\" es de tipo STRING, imposible comparar [#,
%]",id,true));
        }
        if
        (identificadores.get(id.lexemeRank(2)).matches("BOOLEAN")&&!id.lexicalCompRank
(1).matches("IGUAL|DIFERENTE")){
            errors.add(new ErrorLSSL(1,"Error semántico : Variable
\""+id.lexemeRank(2)+"\" es de tipo BOOLEAN, sólo posible comparar con
operadores IGUAL y DIFERENTE [#, %]",id,true));
        }
        if
        (identificadores.get(id.lexemeRank(2)).matches("BOOLEAN")&&!id.lexicalCompRank
(0).matches("TRUE|FALSE")){
            errors.add(new ErrorLSSL(1,"Error semántico : Variable
\""+id.lexemeRank(2)+"\" es de tipo BOOLEAN, sólo posible comparar con valores
booleanos [#, %]",id,true));
        }
        if
        (identificadores.get(id.lexemeRank(2)).matches("INT|FLOAT")){
            if(!id.lexicalCompRank(0).matches("NUMERO|REAL")){
                errors.add(new ErrorLSSL(1,"Error semántico : Valor
numérico de variable \""+id.lexemeRank(2)+"\" no se puede comparar con valor
no numérico [#, %]",id,true));
            }
        }
    }
}

```



```

    }

    }// FOR COMPAPRODDER
    for (Production id: compaProdDoble){

        if
        (!identificadores.containsKey(id.lexemeRank(0))||!identificadores.containsKey(
        id.lexemeRank(2))){
            errors.add(new ErrorLSSL(1,"Error semántico: Variable
            "+id.lexemeRank(0)+" no declarada. [#, %]",id,true));
        }
        else{
            if
            (identificadores.get(id.lexemeRank(0)).matches("STRING")||identificadores.get(
            id.lexemeRank(2)).matches("STRING")){
                errors.add(new ErrorLSSL(1,"Error semántico : Variable
                \"+id.lexemeRank(0)+"\" es de tipo STRING, imposible comparar [#,
                %]",id,true));
            }
            if
            (identificadores.get(id.lexemeRank(0)).matches("BOOLEAN")&&!id.lexicalCompRank
            (1).matches("IGUAL|DIFERENTE")){
                errors.add(new ErrorLSSL(1,"Error semántico : Variable
                \"+id.lexemeRank(0)+"\" es de tipo BOOLEAN, sólo posible comparar con
                operadores IGUAL y DIFERENTE [#, %]",id,true));
            }
            if
            (identificadores.get(id.lexemeRank(2)).matches("BOOLEAN")&&!id.lexicalCompRank
            (1).matches("IGUAL|DIFERENTE")){
                errors.add(new ErrorLSSL(1,"Error semántico : Variable
                \"+id.lexemeRank(2)+"\" es de tipo BOOLEAN, sólo posible comparar con
                operadores IGUAL y DIFERENTE [#, %]",id,true));
            }
            if
            (identificadores.get(id.lexemeRank(0)).matches("BOOLEAN")&&!identificadores.ge
            t(id.lexemeRank(2)).matches("BOOLEAN")){
                errors.add(new ErrorLSSL(1,"Error semántico : Variable
                \"+id.lexemeRank(0)+"\" es de tipo BOOLEAN, sólo posible comparar con valores
                booleanos [#, %]",id,true));
            }
            if
            (identificadores.get(id.lexemeRank(2)).matches("BOOLEAN")&&!identificadores.ge
            t(id.lexemeRank(0)).matches("BOOLEAN")){
                errors.add(new ErrorLSSL(1,"Error semántico : Variable
                \"+id.lexemeRank(2)+"\" es de tipo BOOLEAN, sólo posible comparar con valores
                booleanos [#, %]",id,true));
            }
            if
            (identificadores.get(id.lexemeRank(0)).matches("INT|FLOAT")){

            if(!identificadores.get(id.lexemeRank(2)).matches("INT|FLOAT")){
                errors.add(new ErrorLSSL(1,"Error semántico : Valor
                numérico de variable \"+id.lexemeRank(0)+"\" no se puede comparar con valor

```

```

no numérico [#, %]",id,true));
    }
    }
    if
    (identificadores.get(id.lexemeRank(2)).matches("INT|FLOAT")){

    if(!identificadores.get(id.lexemeRank(0)).matches("INT|FLOAT")){
        errors.add(new ErrorLSSL(1,"Error semántico : Valor
numérico de variable \""+id.lexemeRank(0)+"\" no se puede comparar con valor
no numérico [#, %]",id,true));
    }
    }
}

```

Análisis de operaciones:

Por último, se revisa el contenido de cada una de las producciones generadas al realizar operaciones. Estos ciclos de iteración tienen una estructura similar a los del punto anterior, ya que las instrucciones se guardan en producciones diferentes dependiendo de la posición de la variable, o si hay más de una variable en la operación.

Además de que revisa si las variables usadas han sido declaradas, este análisis revisa si estas son de tipo String o Booleano, lo que hace que hace imposible realizar operaciones aritméticas con estos valores.

Otro error semántico que manejamos al realizar operaciones es la asignación del producto de una división a una variable de tipo entero, lo que causaría una discrepancia entre los tipos de datos.

```

for (Production id: operProdIzq){

    if (!identificadores.containsKey(id.lexemeRank(0))){
        errors.add(new ErrorLSSL(1,"Error semántico: Variable
"+id.lexemeRank(0)+" no declarada. [#, %]",id,true));
    }
    else{
        if
        (identificadores.get(id.lexemeRank(0)).matches("STRING|BOOLEAN")){
            errors.add(new ErrorLSSL(1,"Error semántico : Variable
\""+id.lexemeRank(0)+"\" es de tipo "+identificadores.get(id.lexemeRank(0))
+", imposible hacer operaciones aritméticas [#, %]",id,true));
        }

    }

    if (identificadores.get(id.lexemeRank(0)).matches("INT")&&
id.lexicalCompRank(1).matches("DIVISION")){
        errors.add(new ErrorLSSL(1,"Error semántico : División en
valor entero [#, %]",id,true));
    }
}
} // FOR OPERPRODIZQ
for (Production id: operProdDer){

    if (!identificadores.containsKey(id.lexemeRank(2))){

```

```

        errors.add(new ErrorLSSL(1,"Error semántico: Variable
"+id.lexemeRank(2)+" no declarada. [#, %]",id,true));
    }
    else{
        if
        (identificadores.get(id.lexemeRank(2)).matches("STRING|BOOLEAN")){
            errors.add(new ErrorLSSL(1,"Error semántico : Variable
\""+id.lexemeRank(2)+"\" es de tipo "+identificadores.get(id.lexemeRank(0))
+", imposible hacer operaciones aritméticas [#, %]",id,true));
        }

    }

    }

    }// FOR OPERPRODDEDER
    for (Production id: operProdDoble){

        if
        (!identificadores.containsKey(id.lexemeRank(0))||!identificadores.containsKey(
id.lexemeRank(2))){
            errors.add(new ErrorLSSL(1,"Error semántico: Variable
"+id.lexemeRank(0)+" no declarada. [#, %]",id,true));
        }
        else{
            if
            (identificadores.get(id.lexemeRank(0)).matches("STRING|BOOLEAN"))||identificad
res.get(id.lexemeRank(2)).matches("STRING|BOOLEAN")){
                errors.add(new ErrorLSSL(1,"Error semántico : Variable
\""+id.lexemeRank(0)+"\" es de tipo "+identificadores.get(id.lexemeRank(0))
+", imposible hacer operaciones aritméticas [#, %]",id,true));
                errors.add(new ErrorLSSL(1,"Error semántico : Variable
\""+id.lexemeRank(2)+"\" es de tipo "+identificadores.get(id.lexemeRank(2))
+", imposible hacer operaciones aritméticas [#, %]",id,true));
            }

        }

    }

    }// FOR OPERPRODDOUBLE

```

Botón Compilar

Al presionar el botón compilar el evento limpiara los parámetros codObjComp y variables, para posteriormente llamar al método compile.

```
private void btnCompilarActionPerformed(java.awt.event.ActionEvent evt) {
    codObjComp.clear();
    variables.clear();
    compile();
}
```

Resultados

Al compilar exitosamente el compilador nos mostrará el mensaje de compilación terminada en la consola y la tabla de tokens mostrará la información relevante de los mismos.

The screenshot shows the CHAIRCODE compiler interface. On the left, the source code is displayed with syntax highlighting. Below the code, the console shows the message "pilación terminada...". On the right, the "Tabla de Tokens" (Token Table) is shown, which lists the tokens extracted from the source code.

Componente léxico	Lexema	[Línea, Columna]
INT	INT	[1, 1]
ID	EDAD	[1, 5]
ASIGNACION	=	[1, 10]
NUMERO	34	[1, 12]
MULTIPlicACION	*	[1, 15]
NUMERO	8	[1, 17]
SUMA	+	[1, 19]
NUMERO	5	[1, 21]
DIVISION	/	[1, 23]
NUMERO	8	[1, 25]
PUNTOCOMA	;	[1, 26]
FLOAT	FLOAT	[2, 1]
ID	B	[2, 7]
ASIGNACION	=	[2, 9]
NUMERO	8	[2, 11]
MULTIPlicACION	*	[2, 13]
NUMERO	0	[2, 15]
SUMA	+	[2, 17]
NUMERO	1	[2, 19]
MULTIPlicACION	*	[2, 21]

Figura 2 Resultados de la Compilación

CAPÍTULO 5, Código Intermedio

¿Qué es el código intermedio?

Un lenguaje intermedio es el lenguaje de una máquina abstracta diseñada para ayudar a realizar el análisis de un programa informático. El término proviene de su uso en compiladores, donde el código fuente de un programa es traducido a un modo más apropiado para transformaciones de mejora de código antes de generar el código objeto o código máquina para una máquina determinada. El diseño del lenguaje intermedio difiere típicamente del lenguaje de máquina de tres maneras fundamentales:

COMPILADOR CHAIRCODE

Cada instrucción representa exactamente una operación fundamental suma, resta, etc.

La información de la estructura de control puede no estar incluida en el juego de instrucciones.

El número de registros disponibles puede ser grande, incluso ilimitado.

Dos de los tipos comunes de los códigos intermedios son el código de 3 direcciones y las cuádruplas, para ambos es requerido un destino, argumento 1, argumento 2 y operador

Código tres direcciones: *dest = arg1 op arg2*

Cuádruplas: *op arg1 arg2 res*

Este compilador hace uso del código de tres direcciones como código intermedio.

Generación de Código Intermedio

Clase que Genera Código Intermedio

El Código Intermedio se genera con el código que se muestra más adelante, para poder crearlo se hace uso de un apila, que va ingresando tokens para poder tener un orden en las operaciones, también utiliza temporales para realizar una recursividad en los token, luego evalúa los token para asignarles su tipo de operación correspondiente, se declara una arraylist para ingresar los tokens, una cadena para guardar el código intermedio, un valor para manipular los temporales, enseguida revisa las asignaciones de código entonces envía a la cadena el código al cual se transformara en tripletas.

```
ArrayList<Token> toks = new ArrayList<Token>();
codigoIntermedio = ("--Código intermedio--\n");
int temp;
//revisa las declaraciones
for (Production id: identProd){
    temp = 1;
    if(id.lexicalCompRank(2).equals("ASIGNACION")&&id.getSizeTokens()>5){
        codigoIntermedio = codigoIntermedio +
("\n\n=====\\n"+id.lexemeRank(0, -1)+"\\n=====");
        codObj.add("\\n\n=====\\n"+id.lexemeRank(0,
-1)+"\\n=====");
    }
}
```

Lo siguiente que se hace es identificar las operaciones de mayor prioridad las cuales son division y multiplicacion, tomando los lexemas de los tokens se construye el codigo de tres direcciones haciendo uso de los temporales, luego quita los tokens del arreglo y agrega los temporales, este mismo metodo se utiliza para las otras operaciones, suma, resta.

```
toks = id.getTokens();
for (int i = 0; i < toks.size(); i++){
    if(toks.get(i).getLexeme().equals("*") || toks.get(i).getLexeme().equals("/")){
        i--;
        codigoIntermedio = codigoIntermedio + ("\\nT"+temp+" =
"+toks.get(i).getLexeme()+toks.get(i+1).getLexeme()+toks.get(i+2).getLexeme());
    }
}
```

```

        codObj.add("\nT"+temp+ " =
"+toks.get(i).getLexeme()+toks.get(i+1).getLexeme()+toks.get(i+2).getLexeme());

        toks.remove(i);
        toks.remove(i);
        toks.remove(i);
        toks.add(i,new Token("T"+temp, "ID",i,i));
        temp++;
    }//if token = * /

    }//for cada
    for (int i = 0; i<toks.size();i++){
if(toks.get(i).getLexeme().equals("+")||toks.get(i).getLexeme().equals("-")){
    i--;
    codigoIntermedio = codigoIntermedio + ("\nT"+temp+ " =
"+toks.get(i).getLexeme()+toks.get(i+1).getLexeme()+toks.get(i+2).getLexeme());

    codObj.add("\nT"+temp+ " =
"+toks.get(i).getLexeme()+toks.get(i+1).getLexeme()+toks.get(i+2).getLexeme());

    toks.remove(i);
    toks.remove(i);
    toks.remove(i);
    toks.add(i,new Token("T"+temp, "ID",i,i));
    temp++;
    }//if token = + -
    }
    codigoIntermedio = codigoIntermedio + ("\n"+id.lexemeRank(1)+" =
"+"T"+(temp-1));
    //para guardar las variables declaradas para posteriormente utilizarlo en el
metodo ensamblador
    variables.add(id.lexemeRank(1));
    //codigo Objeto
    codObjComp.add(objectCode(codObj));
    System.out.println(codObjComp.get(0));
    codObj.clear();

    }//if hay asignacion
    }//for producciones
////////////////////////////////////
////////////////////////////////////7
    //codigo Objeto
    codObjComp.add(objectCode(codObj));
    System.out.println(codObjComp.get(0));
    codObj.clear();
    //System.out.println(objectCode(codObj));
    }//if hay asignacion

}

```

Después sigue con la creación del código intermedio de para las producciones de las funciones y las producciones de las estructuras condicionales en este solo comprueba si es del tipo entonces toma los parámetros asignados a los métodos y transforma a código intermedio el código.

```

        for (Production id: funcProd){
            codigoIntermedio = codigoIntermedio +
(" \n \n ===== \n " + id.lexemeRank(0, -1) + " \n ===== ");
            codigoIntermedio = codigoIntermedio + (" \n param " + id.lexemeRank(2) + " \n call
" + id.lexemeRank(0) + ", 1");

            } // FOR FUNCPROD
        // IF
        for (Production id: ifProd){
            codigoIntermedio = codigoIntermedio +
(" \n \n ===== \n " + id.lexemeRank(0, -1) + " \n ===== ");
            codigoIntermedio = codigoIntermedio + (" \n T1 =
" + id.lexemeRank(2, -3) + " \n if_false T1 goto L1 " + " \n . \n . \n . \n label L1");

            } // FOR IFPROD
        // WHILE
        for (Production id: whileProd){
            codigoIntermedio = codigoIntermedio +
(" \n \n ===== \n " + id.lexemeRank(0, -1) + " \n ===== ");
            codigoIntermedio = codigoIntermedio + (" \n label L1 \n T1 =
" + id.lexemeRank(2, -3) + " \n if_false T1 goto L2 " + " \n . \n . \n . \n goto L1 \n label L2");

            } // FOR WHILEPROD
        // System.out.print(codigoIntermedio);
    } // codigoIntermedio

```

Botón Tripletas

El botón tripletas se encarga de mostrar el código en tres direcciones generado.

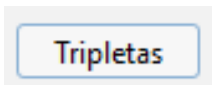


Figura 3 Botón Tripletas

Resultados

Al generar el código intermedio del siguiente código:

```
1  INT EDAD = 34;
2  FLOAT B = 1.7;
3  STRING M = "ERES_MAYOR";
4  STRING H = "HOLA_MUNDO";
5
6  IF(EDAD >= 34){
7      WHILE(EDAD>70){
8          INPUT("3RA_EDAD");
9          EDAD++;
10     }
11 }
12
13 EDAD = 1 + 4 * 6;
14
15 WHILE(100>56){
16     FOR(i IN RANGE(1,400)){
17         B = 1.0;
18     }
```

Figura 4 Código a Transformar en Tripletas

Después de compilarlo y darle en el botón tripletas nos da como resultado lo siguiente:

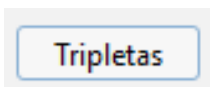


Figura 5 Botón Tripletas

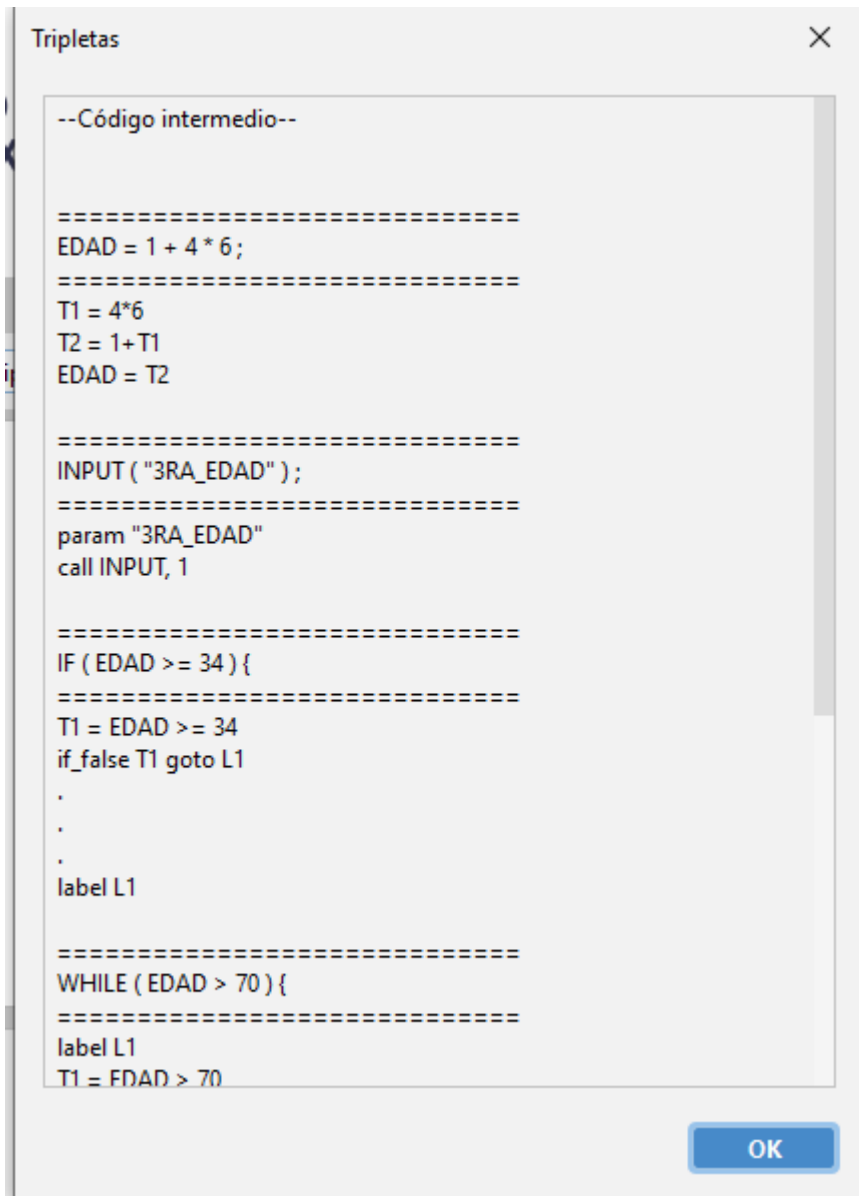


Figura 5 Resultado de Generacion deCodigo Intermedio

CAPÍTULO 6, Optimización

¿Qué es la Optimización de Código?

La optimización de código es el conjunto de fases de un compilador que transforma un fragmento de código en otro fragmento con un comportamiento equivalente y que se ejecuta de forma más eficiente, es decir, usando menos recursos de cálculo como memoria o tiempo de ejecución.

Optimización de Código

Código que Optimiza Código

Para optimizar el código se requiere del método optimización que se ejecuta junto con los otros métodos.

El Funcionamiento del método comienza con declarar las variables que se utilizaran para realizar la optimización.

```
private void optimizacion(){
    ArrayList<Token> toks = new ArrayList<Token>();
    codigoIntermedio = ("--Código intermedio--\n");
    int temp;
    int divisor;
    float frac;
```

Lo siguiente que realiza el compilador es analizar todas las producciones.

```
for (Production id: identProd){
    temp = 1;
    System.out.println(id.lexemeRank(0, -1));
```

A continuación utiliza varios ciclos para optimizar ciertos códigos el primero es intercambiar las divisiones por multiplicaciones de números reales.

```
if(id.lexicalCompRank(2).equals("ASIGNACION")&&id.getSizeTokens()>5){
    toks = id.getTokens();
    for (int i = 0;i<toks.size();i++){
        if(toks.get(i).getLexeme().equals("/")){
            if(toks.get(i+1).getLexicalComp().equals("NUMERO")){
                divisor =
Integer.parseInt(toks.get(i+1).getLexeme());
                System.out.println("DIVISOR: "+divisor);
                frac = 1 / (float)divisor;
                System.out.println("DIVISOR: "+frac);
                toks.remove(i);
                toks.remove(i);
```

```

                                toks.add(i, new Token("*", "MULTIPLICACION",i,i));
                                toks.add(i+1, new Token(Float.toString(frac),
"REAL",i,i));
                                }
                                }//if token = /
                                }//for elimina divisiones

```

El segundo en optimizar es es eliminar las multiplicaciones innecesarias.

```

for (int i = 0;i<toks.size();i++){
    if(toks.get(i).getLexeme().equals("*")){
        if(toks.get(i-1).getLexeme().equals("0")){
            toks.remove(i);
            toks.remove(i);
        }
        if(toks.get(i+1).getLexeme().equals("0")){
            i--;
            toks.remove(i);
            toks.remove(i);
        }
    }

    }//if token = *
    }//for multiplicaciones por cero

for (int i = 0;i<toks.size();i++){
    if(toks.get(i).getLexeme().equals("*")){
        if(toks.get(i+1).getLexeme().equals("2")){
            toks.remove(i);
            toks.remove(i);
            toks.add(i,new Token("+", "SUMA",i,i));
            toks.add(i+1,new Token(toks.get(i-1).getLexeme(),
"NUMERO",i,i));
        }

        }//if token = 2

        else if(toks.get(i+1).getLexeme().equals("1")){
            toks.remove(i);
            toks.remove(i);
        }// if token = 1
        }//if token multiplicacion
    }//for multiplicaciones por dos

```

Lo tercero en optimizar son las sumas con resultado de cero.

```

for (int i = 0;i<toks.size();i++){
    if(toks.get(i).getLexicalComp().matches("SUMA|RESTA")){
        if(toks.get(i-1).getLexeme().equals("0")){

```

```

        i--;
        toks.remove(i);
        toks.remove(i);
    }else
        if(toks.get(i+1).getLexeme().equals("0")){
            toks.remove(i);
            toks.remove(i);

        }

    }//if token = 0

    }//if token suma
} //for sumas cero

```

El siguiente paso es la creación del código intermedio

```

codigoIntermedio = codigoIntermedio +
("\n\n===== \n"+id.lexemeRank(0,
-1)+"\n=====");

codObj.add("\n\n===== \n"+id.lexemeRank(0,
-1)+"\n=====");

    for (int i = 0;i<toks.size();i++){

if(toks.get(i).getLexeme().equals("*")||toks.get(i).getLexeme().equals("/")){
    i--;
    codigoIntermedio = codigoIntermedio + ("\nT"+temp+" =
"+toks.get(i).getLexeme()+toks.get(i+1).getLexeme()+toks.get(i+2).getLexeme())
;

    codObj.add("\nT"+temp+" =
"+toks.get(i).getLexeme()+toks.get(i+1).getLexeme()+toks.get(i+2).getLexeme())
;

    toks.remove(i);
    toks.remove(i);
    toks.remove(i);
    toks.add(i,new Token("T"+temp, "ID",i,i));
    temp++;
} //if token = * /

} //for cada
for (int i = 0; i<toks.size();i++){

if(toks.get(i).getLexeme().equals("+")||toks.get(i).getLexeme().equals("-")){
    i--;
    codigoIntermedio = codigoIntermedio + ("\nT"+temp+" =
"+toks.get(i).getLexeme()+toks.get(i+1).getLexeme()+toks.get(i+2).getLexeme())
;

```

```

        codObj.add("\nT"+temp+ " =
"+toks.get(i).getLexeme()+toks.get(i+1).getLexeme()+toks.get(i+2).getLexeme())
;

        toks.remove(i);
        toks.remove(i);
        toks.remove(i);
        toks.add(i,new Token("T"+temp, "ID",i,i));
        temp++;
    }//if token = + -
}
if (temp==1)
    codigoIntermedio = codigoIntermedio + ("\n"+id.lexemeRank(0)+"
= " + id.lexemeRank(2));
else
    codigoIntermedio = codigoIntermedio + ("\n"+id.lexemeRank(0)+"
= "+ "T"+(temp-1));
//para guardar las variables declaradas para posteriormente
utilizarlo en el metodo ensamblador
variables.add(id.lexemeRank(0));

////////////////////////////////////
////////////////////////////////////7
//codigo Objeto
codObjComp.add(objectCode(codObj));
System.out.println(codObjComp.get(0));
codObj.clear();
//System.out.println(objectCode(codObj));
} //if hay asignacion

}
//INPUT Y OUTPUT
for (Production id: funcProd){
    codigoIntermedio = codigoIntermedio +
("\n\n===== \n"+id.lexemeRank(0,
-1)+"\n=====");
    codigoIntermedio = codigoIntermedio + ("\nparam
"+id.lexemeRank(2)+"\ncall "+id.lexemeRank(0)+" , 1");

    }//FOR FUNCPROD
//IF
for (Production id: ifProd){
    codigoIntermedio = codigoIntermedio +
("\n\n===== \n"+id.lexemeRank(0,
-1)+"\n=====");
    codigoIntermedio = codigoIntermedio + ("\nT1 =
"+id.lexemeRank(2,-3)+"\nif_false T1 goto L1 "+ "\n.\n.\n.\nlabel L1");

    }//FOR IFPROD
//WHILE
for (Production id: whileProd){
    codigoIntermedio = codigoIntermedio +

```

```

("\n\n===== \n"+id.lexemeRank(0,
-1)+"\n=====");

if(id.lexicalCompRank(5).matches("MULTIPLICACION|DIVISION|SUMA|RESTA")){
    codigoIntermedio = codigoIntermedio + ("\nT1 =
"+id.lexemeRank(4,-3) + "\nlabel L1\nif_false " + id.lexemeRank(2,3) + " T1
goto L2 "+" \n.\n.\n.\ngoto L1\nlabel L2");
    }
    else{
        codigoIntermedio = codigoIntermedio + ("\nlabel
L1\nif_false " + id.lexemeRank(2,-3) +" goto L2 "+" \n.\n.\n.\ngoto L1\nlabel
L2");
    }
}
}

```

CAPÍTULO 7, Código Objeto

¿Qué es el código Objeto?

El código objeto es el código que resulta de la compilación del código fuente. tiene como objetivo ser analizado por la máquina para que ésta realice las instrucciones que se le indican al compilador

Generación de código Objeto

Código que Genera Código Objeto

Para generar el código objeto se utiliza un método llamado object code que lo primero que hace es tomar el código intermedio para poder utilizarlo.

```

private String objectCode(ArrayList<String> tripletas1) {
    ArrayList<String> tripletas = new ArrayList<String>();
    tripletas = tripletas1;
    String tl=tripletas.get(0)+"\n";tripletas.remove(0);

```

Enseguida se declaran las variables que se utilizarán los registros R's, los operadores op, un índice y un control de casos.

```

String inst, R0, R1, R2, R3, op, m;
int caso = 0;
inst = R1 = R0 = R2 = R3 = op = m = "";
int index = 0;

```

Lo siguiente es eliminar texto de las tripletas que no se necesite y luego intercambiar los operadores por sus equivalencias en código objeto.

```

for (String tripleta : tripletas) {

    tripleta = tripleta.replaceAll("T[1-9] = ", "").replaceAll("\\\\n",
    "");

    //JOptionPane.showMessageDialog(null, tripleta);
    // Definimos que operacion es
    if (tripleta.contains("*")) {
        inst = "MUL";
        op = "*";
    }
    if (tripleta.contains("/")) {
        inst = "DIV";
        op = "/";
    }
    if (tripleta.contains("-")) {
        inst = "SUB";
        op = "-";
    }
    else if (tripleta.contains("+")) {
        inst = "ADD";
        op = "+";
    }
}

```

Enseguida analiza las tripletas para saber a qué parte de la operación corresponden.

```

// Definimos que operacion es

    //Condicionales para ver el orden de la operacion
    if (R0.isEmpty() && R1.isEmpty()) {
        //JOptionPane.showMessageDialog(null, "caso 0");
        R0 = (tripleta.substring(0,
tripleta.indexOf(op))).replaceAll(" ", "");
        R1 = (tripleta.substring(tripleta.indexOf(op) +
1)).replaceAll(" ", "");

        }else if ((tripleta.substring(0,
tripleta.indexOf(op))).contains("T") &&
(tripleta.substring(tripleta.indexOf(op) + 1)).contains("T") ) { //2 TEMPORALES
        R0 = "R0";
        R1="R1";
        caso =4 ;
        //JOptionPane.showMessageDialog(null, "caso 4");
        } else if ((tripleta.substring(0,
tripleta.indexOf(op))).contains("T")) { //temporal izquierdo
        R1 = (tripleta.substring(tripleta.indexOf(op) +
1)).replaceAll(" ", "");
        caso =1 ;
        //JOptionPane.showMessageDialog(null, "caso 1");
        } else if ((tripleta.substring(tripleta.indexOf(op) +
1)).contains("T")) { //temporal derecho
        R1 = (tripleta.substring(0,

```

```

tripleta.indexOf(op))).replaceAll(" ", "");
    caso =2;
    //JOptionPane.showMessageDialog(null,"caso 2");
}
else{
    R1 = (tripleta.substring(0,
tripleta.indexOf(op))).replaceAll(" ", "");
    R2 = (tripleta.substring(tripleta.indexOf(op) +
1)).replaceAll(" ", "");
    caso=5;
    //JOptionPane.showMessageDialog(null,"caso 5");
}

```

Después de identificar el tipo de operación y su orden ahora comienza a generar el código objeto.

```

//Condicionales para ver el orden de la operacion
switch (caso) {
    case 1:
        m+="LD R1," + R1+"\n";
        m+=inst + " R0,R0,R1"+" \n";
        break;
    case 2:
        m+="LD R1," + R1+"\n";
        m+=inst + " R0,R1,R0"+" \n";
        break;
    case 3:
        m+="LD R1," + R2+"\n";
        m+="LD R2," + R2+"\n";
        m+=inst + " R1,R1,R2"+" \n";
        break;
    case 4:
        m+=inst+" "+R0+", "+R0+", "+R1;
        break;
    case 5:
        m+="LD R1," + R1+"\n";
        m+="LD R2," + R2+"\n";
        m+=inst + " R1,R1,R2"+" \n";
        caso =0;
        break;
    default:
        m+="LD R0," + R0+"\n";
        m+="LD R1," + R1+"\n";
        m+=inst + " R0,R0,R1"+" \n";
        caso =0;
}

```

Mediante el código Ensamblador se adapta al código tomando el código objeto y añadiendo segmentos, inicio y final de código así como las declaraciones de los datos.


```

int i=0;
    String vr = "";
    String STRUC="";
    objeto = objeto.replaceAll("=====",
";=====").replaceAll("FLOAT",";FLOAT").replaceAll("INT",";INT");
    for(String str: variables){vr+="    "+str+" DB 0,0\n";}
    //JOptionPane.showMessageDialog(null, vr);
    //creamos la estructura base de ensamblador
    STRUC+="model small\n.stack\n.data \n"+vr+".code\nINICIO: MOV AX,
@DATA\n        MOV DS, AX\n        MOV ES, AX\n\n";

    while(i<4){objeto = objeto.replaceAll("MUL R"+i+",R"+i+",R"+(i+1), "MUL
R"+(i+1)).replaceAll("DIV R"+i+",R"+i+",R"+(i+1), "DIV R"+(i+1));i++;}
    //System.out.println(objeto);
    objeto = objeto.replaceAll("LD", "MOV");
    objeto = objeto.replaceAll("R0,R0", "AX");
    objeto = objeto.replaceAll("R1,R1", "BX");
    objeto = objeto.replaceAll("R2,R2", "CX");
    objeto = objeto.replaceAll("R3,R3", "DX");
    objeto = objeto.replaceAll("R0", "AX");
    objeto = objeto.replaceAll("R1", "BX");
    objeto = objeto.replaceAll("R2", "CX");
    objeto = objeto.replaceAll("R3", "DX");
    STRUC+=objeto;

    STRUC+="\nFIN: MOV AX,4C00H\n        INT 21H\n        END\n";
    return STRUC;

```

Botón Código Objeto

El botón código objeto muestra el código objeto que se genera al presionar el botón.

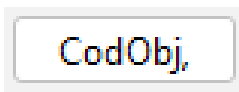


Figura 6 Botón Código Objeto

El botón Ensamblador adapta el código objeto al código en lenguaje ensamblador para el emu8086 y abre el código en el emu.

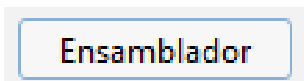


Figura 7 Botón Ensamblador

Resultados

Al presionar el código objeto nos muestra el siguiente código:

```
Codigo Objeto

-----CODIGO OBJETO-----

T1 = 4*6
LD R0,1
LD R1,T1
ADD R0,R0,R1
```

Figura 8 Código Objeto Generado

Después al presionar el ensamblador nos muestra el código adaptado para el emu y enseguida lo abre en el emu.

```
.model small
.stack
.data
    EDAD DB 0,0
.code
INICIO: MOV AX, @DATA
        MOV DS, AX
        MOV ES, AX

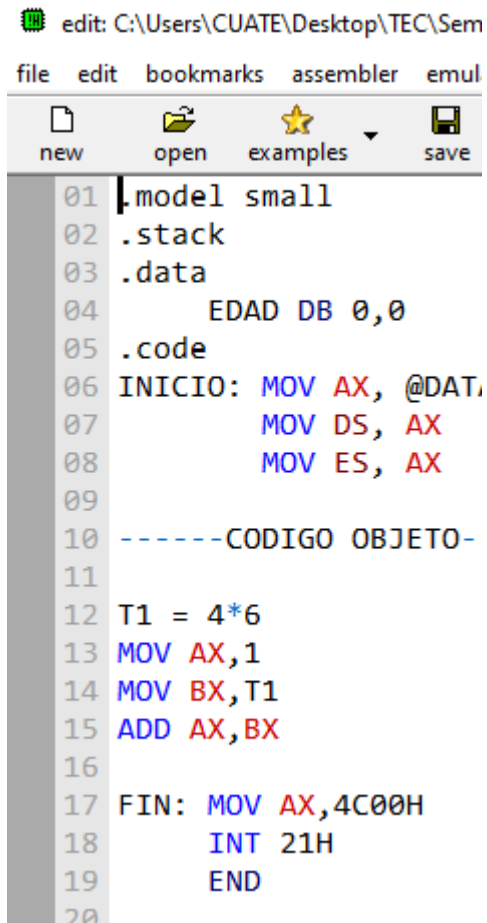
-----CODIGO OBJETO-----

T1 = 4*6
MOV AX,1
MOV BX,T1
ADD AX,BX

FIN: MOV AX,4C00H
    INT 21H
    END
```

Figura 9 Código Lenguaje Ensamblador

COMPILADOR CHAIRCODE



The screenshot shows the EMU8086 IDE interface. The title bar indicates the file path: `edit: C:\Users\CUATE\Desktop\TEC\Sem`. The menu bar includes `file`, `edit`, `bookmarks`, `assembler`, and `emul`. The toolbar contains icons for `new`, `open`, `examples`, and `save`. The main text area displays an assembly program with line numbers 01 through 20. The code defines a small model, stack, and data segment, then proceeds to the code segment with instructions for moving data, setting segment registers, and performing arithmetic operations.

```
01 |.model small
02 |.stack
03 |.data
04 |    EDAD DB 0,0
05 |.code
06 |INICIO: MOV AX, @DAT
07 |        MOV DS, AX
08 |        MOV ES, AX
09 |
10 |-----CODIGO OBJETO-----
11 |
12 |T1 = 4*6
13 |MOV AX,1
14 |MOV BX,T1
15 |ADD AX,BX
16 |
17 |FIN: MOV AX,4C00H
18 |    INT 21H
19 |    END
20 |
```

Figura 10 Lenguaje Ensamblador en EMU8086

CAPÍTULO 7, Lenguaje Ensamblador

¿Qué es el Lenguaje Ensamblador?

El lenguaje ensamblador expresa las instrucciones de una forma más natural al hombre a la vez que muy cercana al microcontrolador, ya que cada una de esas instrucciones se corresponde con otra en código máquina.

Generación Lenguaje Ensamblador

Botón ensamblador

Se genera el TextArea para crear la ventana emergente donde posteriormente se mostrará el lenguaje ensamblador creado por nuestro compilador. Después se creará el archivo donde se almacenará el código en ensamblador y abrirá dicho archivo en la ventana que creamos.

```
JTextArea textArea = new JTextArea(ensamblador);
JScrollPane scrollPane = new JScrollPane(textArea);
textArea.setLineWrap(true);
textArea.setWrapStyleWord(true);
textArea.setEditable(false);
scrollPane.setPreferredSize( new Dimension( 400, 500 ) );
JOptionPane.showMessageDialog(null, scrollPane,
"Tripletas",JOptionPane.PLAIN_MESSAGE);

archivoT(".\\COD_OBJ.asm",ensamblador);
abrirarchivo(".\\COD_OBJ.asm");
```

Apéndices:

Apéndice A:

A.1: Palabras reservadas:

Palabra reservada	Descripción	Ejemplo
IMPORT	Cadena mayus y min I,M,P,O,R,T	IMPORT ó import

DEF	Cadena mayus y min D, E, F	DEF ó def
CLASS	Cadena mayus y min C, L, A, S, S	CLASS ó class
IF	Cadena mayus y min I, F	IF ó if
ELSE	Cadena mayus y min E, L, S, E	ELSE ó else
FOR	Cadena mayus y min F, O, R	FOR ó for
IN	Cadena mayus y min I, N	IN ó in
RANGE	Cadena mayus y min R, A, N, G, E	RANGE ó range
SELF	Cadena mayus y min S, E, L, F	SELF ó self
WHILE	Cadena mayus y min W, H, I, L, E	WHILE ó while

COMPILADOR CHAIRCODE

TRY	Cadena mayus y min T, R, Y	TRY ó try
EXCEPT	Cadena mayus y min E, X, C, E, P, T	EXCEPT ó except
RETURN	Cadena mayus y min R, E, T, U, R, N	RETURN ó return
BREAK	Cadena mayus y min B, R, E, A, K	BREAK ó break
NEXT	Cadena mayus y min N, E, X, T	NEXT ó next
INPUT	Cadena mayus y min I, N, P, U, T	INPUT ó input
PRINT	Cadena mayus y min P, R, I, N, T	PRINT ó print
INT	Cadena mayus y min I, N, T	INT ó int
FLOAT	Cadena mayus y min F, L, O, A, T	FLOAT ó float
STRING	Cadena mayus y min S, T, R, I, N, G	STRING ó string
POWER	Cadena mayus y min P, O, W, E, R	POWER ó power
SQRT	Cadena mayus y min S, Q, R, T	SQRT ó sqrt
AND	Cadena mayus y min A, N, D	AND ó and
OR	Cadena mayus y min O, R	OR ó or
NOT	Cadena mayus y min N, O, T	NOT ó not

Apéndice B:

B.1: Ejemplo Funcional:

- Primero abriremos o crearemos un nuevo Archivo.

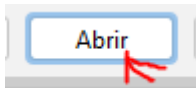


Figura 11 Botón Abrir

- Seleccionaremos el archivo que queremos abrir y lo abrimos.

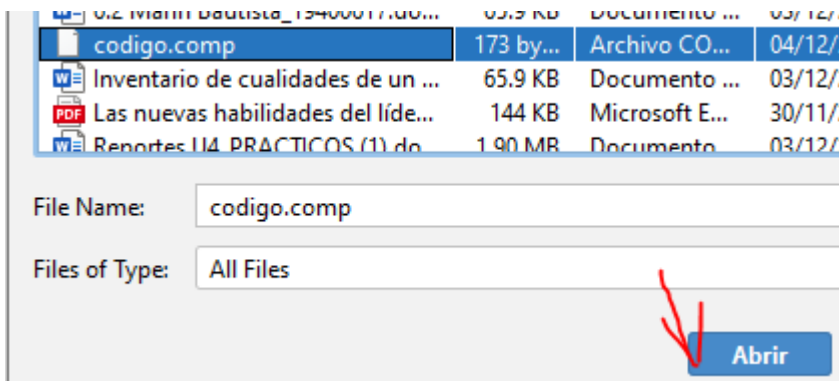


Figura 12 Selección de Archivo

- Se mostrará el código en el área de texto.

```

1  INT EDAD = 34;
2  FLOAT B = 5.7
3  STRING M = "ERES_MAYOR";
4  STRING H = "HOLA_MUNDO";
5
6  IF(EDAD >= 34){
7      WHILE(EDAD>70){
8          INPUT("3RA_EDAD");
9          EDAD++;
10     }
11 }
12
13 WHILE(100>56){
14     FOR(i IN RANGE(1,400)){
15         B = 50*8;
16     }
17 }
    
```

Figura 13 Código Fuente

- Ahora compilamos para detectar los tokens, errores léxicos, errores semánticos y errores sintácticos.
- En caso de aparecer algún error nos dirá cual es y donde se encuentra para que podamos solucionarlo.

```

Compilación terminada...

ERROR_SINTACTICO: PUNTOCOMA(;) NO AGREGADO EN LA DECLARACION [2, 1]
Error semántico: Variable "B" no declarada. [15, 11]

La compilación terminó con errores...

1 | INT EDAD = 34;
2 | FLOAT B = 5.7
3 | STRING M = "ERES_MAYOR";
  | .....

```

Figura 14 Error Sintactico

- La tabla de tokens nos mostrará todos los elementos del código.

Componente léxico	Lexema	[Línea, Columna]
INT	INT	[1, 1]
ID	EDAD	[1, 6]
ASIGNACION	=	[1, 11]
NUMERO	34	[1, 13]
PUNTOCOMA	;	[1, 15]
FLOAT	FLOAT	[2, 1]
ID	B	[2, 7]
ASIGNACION	=	[2, 9]
REAL	1.7	[2, 11]
PUNTOCOMA	;	[2, 14]
STRING	STRING	[3, 1]
ID	M	[3, 8]
ASIGNACION	=	[3, 10]
CADENA	"ERES_MAYOR"	[3, 12]
PUNTOCOMA	;	[3, 24]
STRING	STRING	[4, 1]
ID	H	[4, 8]
ASIGNACION	=	[4, 10]
CADENA	"HOLA_MUNDO"	[4, 12]

Figura 15 Tabla de Tokens

- Ya compilado sin errores podemos generar el código intermedio.

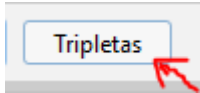


Figura 16 Botón Tripletas

- En este podremos ver las líneas de código y su equivalente en 3 direcciones.

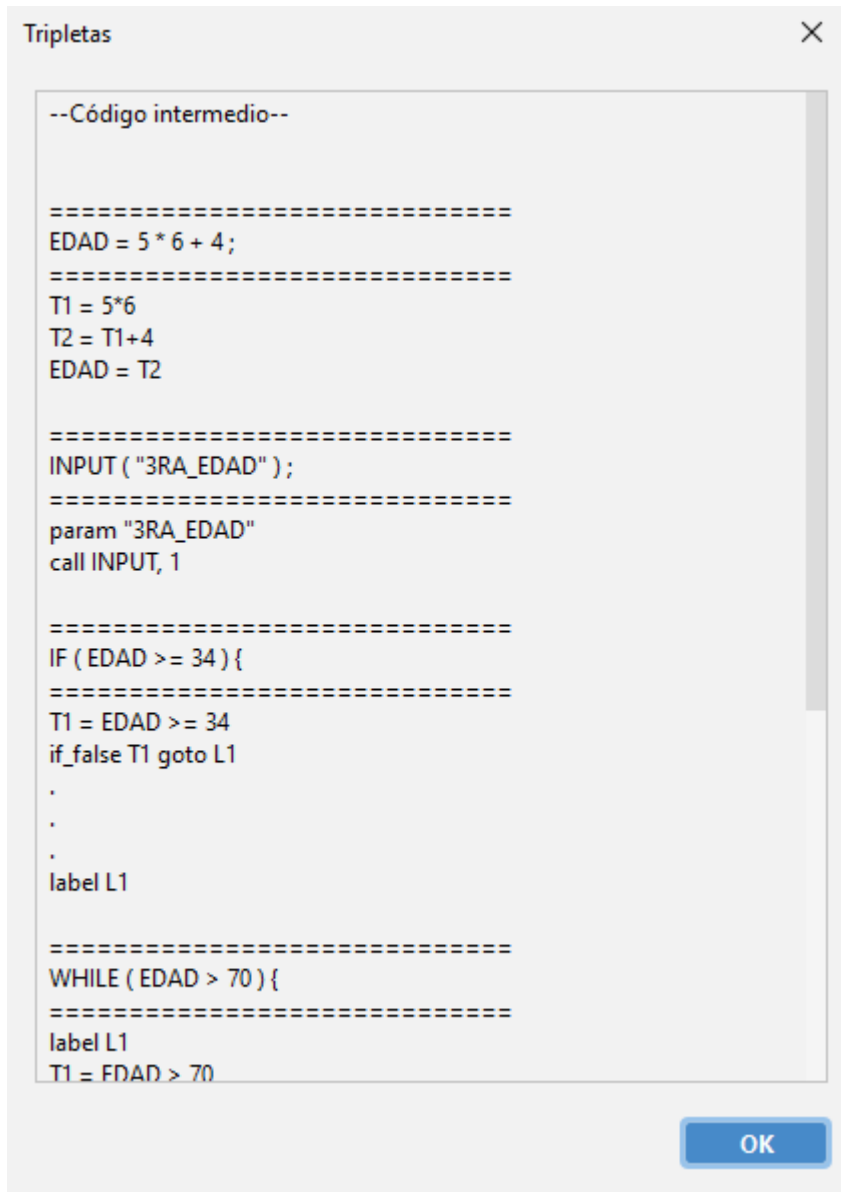


Figura 17 Código Intermedio

- También podemos generar el código objeto y que lo muestre en el emulador EMU 8086.

COMPILADOR CHAIRCODE

-----CODIGO OBJETO-----

```
T1 = 5*6
LD R0,T1
LD R1,4
ADD R0,R0,R1
```

Figura 18 Código Objeto

```
.model small
.stack
.data
    EDAD DB 0,0
.code
INICIO: MOV AX, @DATA
        MOV DS, AX
        MOV ES, AX

-----CODIGO OBJETO-----

T1 = 5*6
MOV AX,T1
MOV BX,4
ADD AX,BX

FIN: MOV AX,4C00H
    INT 21H
    END
```

Figura 19 Lenguaje Ensamblador

COMPILADOR CHAIRCODE

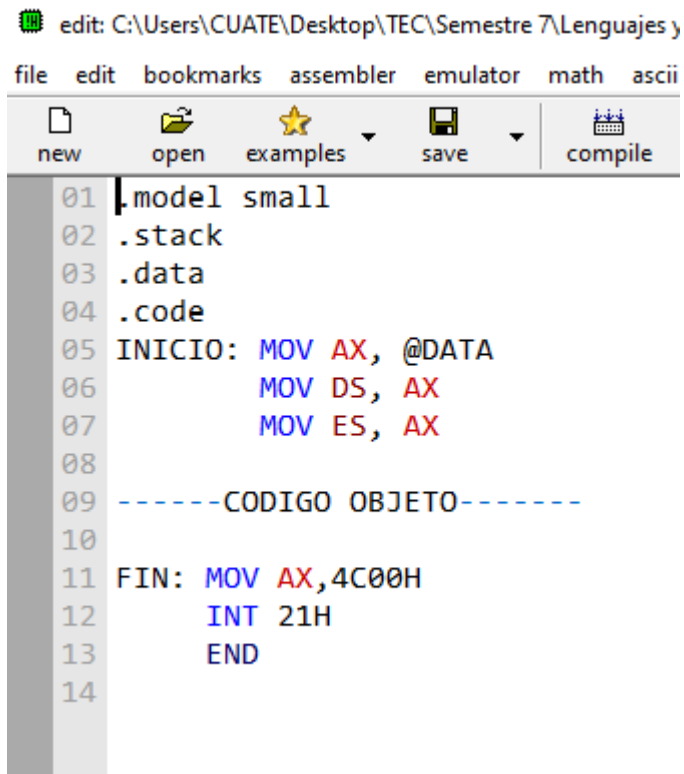


Figura 20 Lenguaje Ensamblador en EMU8086

El compilador también realiza optimización de código, para esto colocamos código que se pueda optimizar.

```
INT EDAD = 34 * 8 + 5 / 8;
FLOAT B = 8 * 0 + 1 * 2 * 3;
EDAD = 10 - 0 + 4 * 5;
INT H = 7 - 3 / 8 + 5 * 1;
INT x = 5 * 0 + 5 / 2;
STRING M = "ERES_MAYOR";

INPUT("KIUBO");

IF (EDAD > 5 * 8){}

WHILE(100 > 50 - 1){
    //instrucciones
}
```

Figura 21 Código a Optimizar.

Después realizamos los mismos pasos que anteriormente.

Pulsamos en compilar.

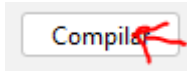


Figura 22 Botón Compilar

Después de eso nos preguntará si queremos optimizar el código, en este caso diremos que si.

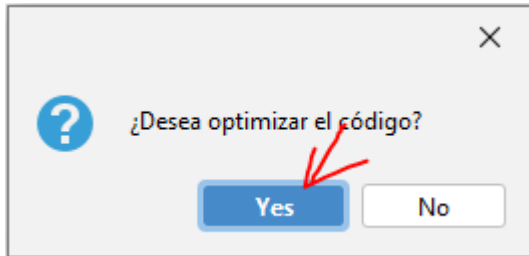


Figura 23 Selección de Optimización.

Con esto nos mostrará la tabla de tokens como normalmente lo hace.

Tabla de Tokens		
Componente léxico	Lexema	[Línea, Columna]
INT	INT	[1, 1]
ID	EDAD	[1, 5]
ASIGNACION	=	[1, 10]
NUMERO	34	[1, 12]
MULTIPLICACION	*	[1, 15]
NUMERO	8	[1, 17]
SUMA	+	[1, 19]
NUMERO	5	[1, 21]
DIVISION	/	[1, 23]
NUMERO	8	[1, 25]
PUNTOCOMA	;	[1, 26]
FLOAT	FLOAT	[2, 1]
ID	B	[2, 7]
ASIGNACION	=	[2, 9]
NUMERO	8	[2, 11]
MULTIPLICACION	*	[2, 13]
NUMERO	0	[2, 15]
SUMA	+	[2, 17]
NUMERO	1	[2, 19]

Figura 24 Tabla de Tokens

Enseguida debemos crear las tripletas con el botón tripletas.

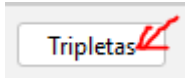


Figura 25 Botón Tripletas

Esto generará un código intermedio optimizado.

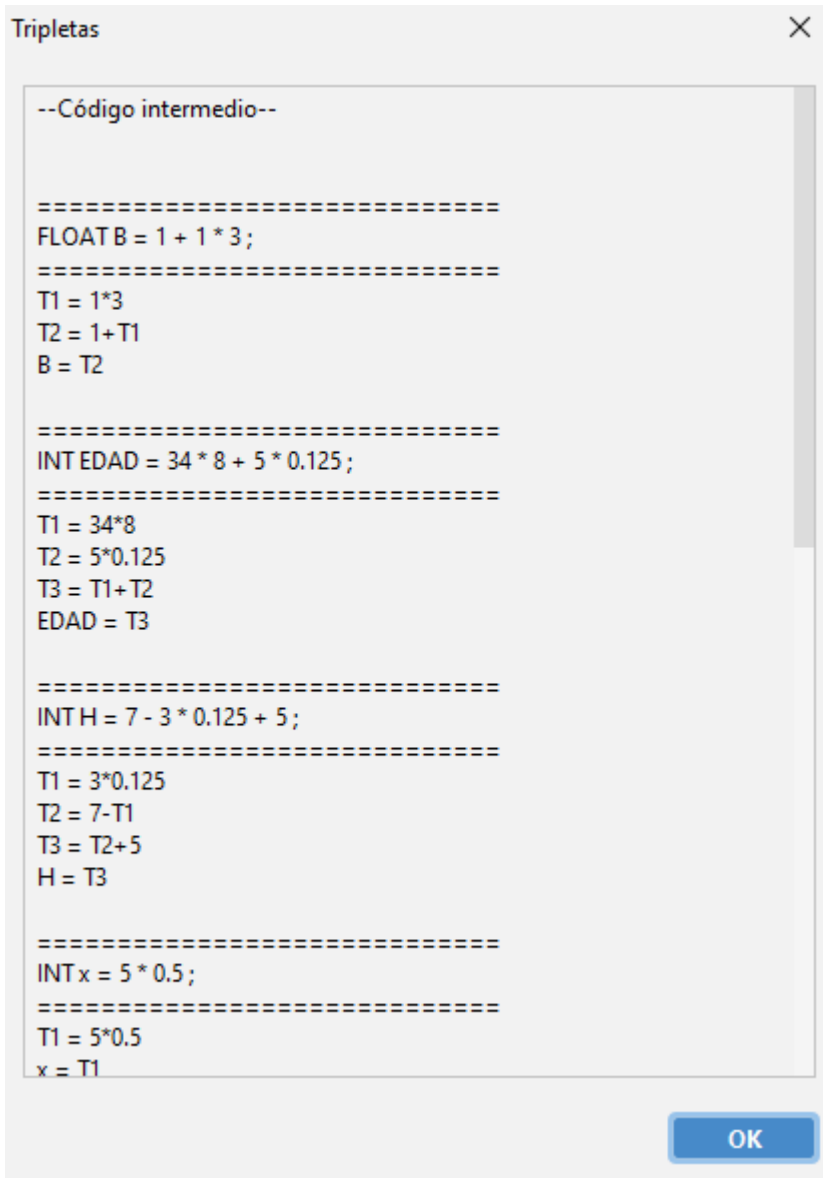


Figura 26 Código de Tripletas

B.3: Índice de Figuras:

Figura 1 Estructura del Proyecto

Figura 2 Resultados de la Compilación

Figura 3 Botón Tripletas

Figura 4 Código a Transformar en Tripletas

Figura 5 Botón Tripletas

Figura 6 Botón Código Objeto

Figura 7 Botón Ensamblador

Figura 8 Código Objeto Generado

Figura 9 Código Lenguaje Ensamblador

Figura 10 Lenguaje Ensamblador en EMU8086

Figura 11 Botón Abrir

Figura 12 Selección de Archivo

Figura 13 Código Fuente

Figura 14 Error Sintactico

Figura 15 Tabla de Tokens

Figura 16 Botón Tripletas

Figura 17 Código Intermedio

Figura 18 Código Objeto

Figura 19 Lenguaje Ensamblador

Figura 20 Lenguaje Ensamblador en EMU8086

Figura 21 Código a Optimizar.

Figura 22 Botón Compilar

Figura 23 Selección de Optimización

Figura 24 Tabla de Tokens

Figura 25 Botón Tripletas

Figura 26 Código de Tripletas

Apéndice C. Para descargar e instalar.

Requerimiento	Enlace de Descarga
JFlex 1.4.3	http://www.java2s.com/Code/Jar/j/Downloadjflex143jar.htm
JDK 1.8	https://www.oracle.com/mx/java/technologies/javase/javase8-archive-downloads.html
NetBeans 12.4	https://netbeans.apache.org/download/nb124/nb124.html