

Nombres de los estudiantes	Carné	Sección
Jonnathan Alejandro Juárez Velásquez	15377	10
Nombre de la tarea	Documentación Proyecto #2	

Diagrama de Flujo:

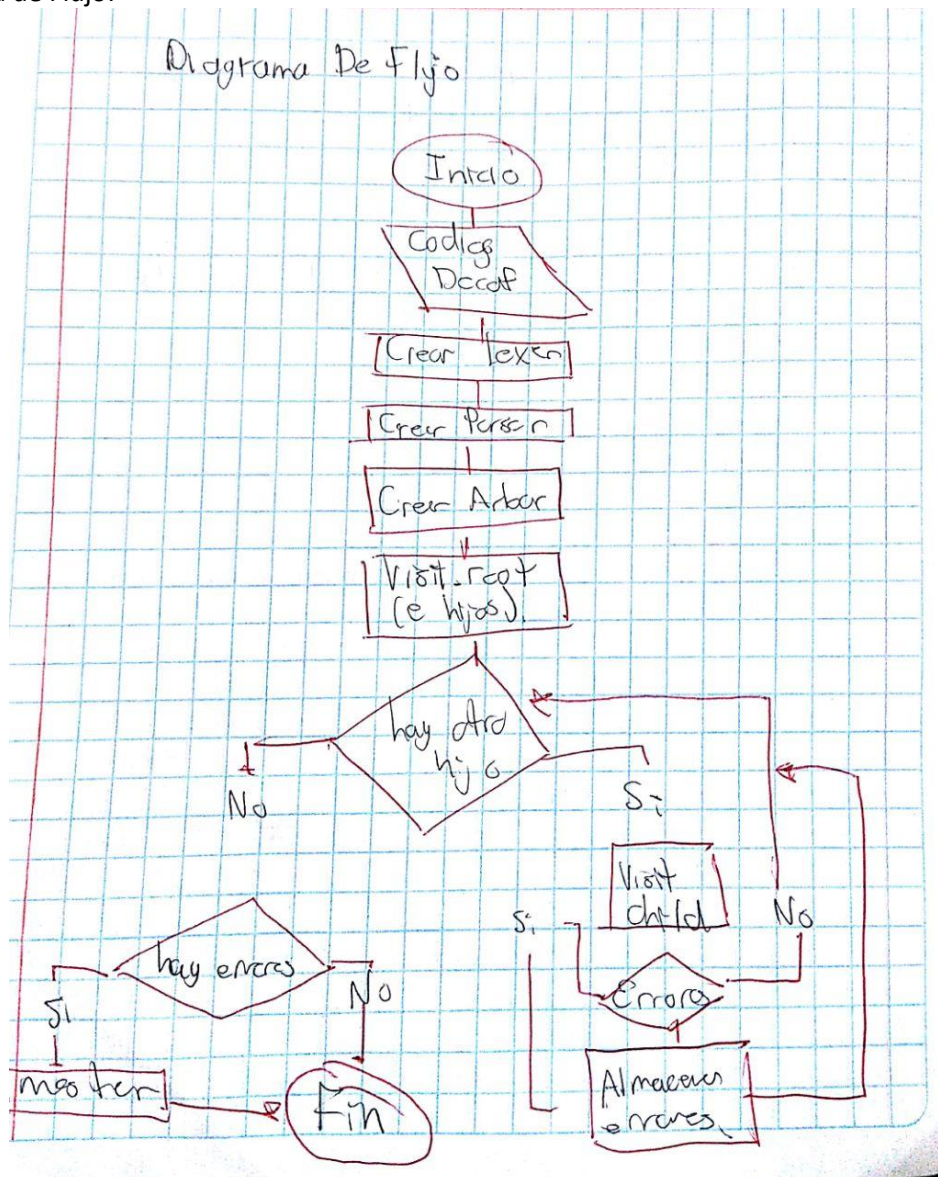
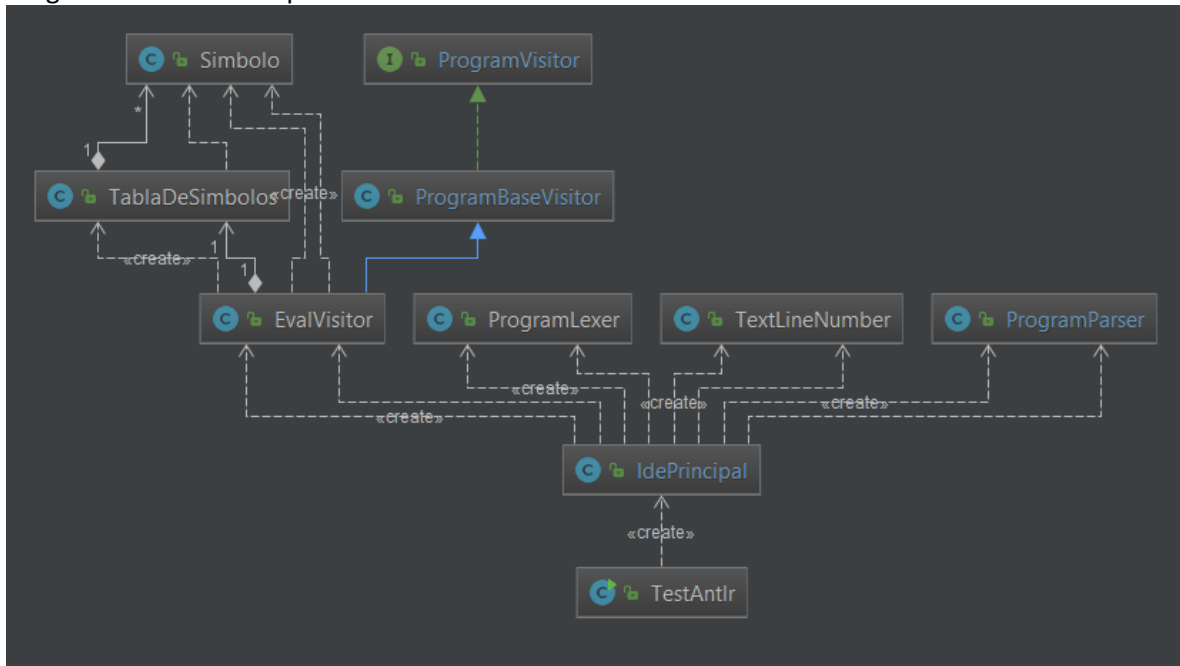


Diagrama de Clases simplificado



Esta versión únicamente incluye las relaciones entre las clases del programa, esto debido a que agregar parámetros y métodos aumenta de manera considerable el tamaño y sería complicado visualizarlo.

Sistema de tipos:

proram

```

: 'class' ID '{' (declaration)* '}'
;
#progDeclarattion

```

declaration

```

: structDeclaration
#aStructDec
| varDeclaration
| methodDeclaration
#aMethodDec
;
#aVarDec

```

varDeclaration

```

: varType ID ';'
| varType ID '[' NUM ']' ';'
;
#simpleVarTyoe
#arrayVarType

```

structDeclaration

```

: STRUCT ID '{' (varDeclaration)* '}'
;
#declarationStruct

```

```

varType
:      INT                      #vartypeInt
|      CHAR                    #vartypeChar
|      BOOLEAN                 #vartypeBoolean
|      STRUCT ID               #vartypestrucID
|      structDeclaration      #vartypestruc
|      VOID                    #vartypeVoid
;

```

```

methodDeclaration
:      methodType ID '(' (parameter | ( parameter (',' parameter)* ))? ')' block
#methodDecl
;

```

```

methodType
:      INT                      #methodTypeInt
|      CHAR                    #methodTypeChar
|      BOOLEAN                 #methodTypeBoolean
|      VOID                    #methodTypeIntVoid
;

```

```

parameter
:      parameterType ID        #parameterID
|      parameterType ID '[' NUM ']' #parameterArray
;

```

```

parameterType
:      INT                      #parameterTypeInt
|      CHAR                    #parameterTypeChar
|      BOOLEAN                 #parameterTypeBoolean
;

```

```

block
:      '{' (varDeclaration | statement)* '}'      #blockDeclaration
;

```

```

statement
:      'if' '(' expression ')' block (statementElse)?      #statementIF
|      WHILE '(' expression ')' block                      #statementWhile
|      'return' (expression | ) ';'                        #statementReturn
|      methodCall ';'                                      #statementMethodCall
|      block                                                #statementBlock

|      location '=' expression ';'                          #statementLocation

```

```

        |      (expression)?';'                #statementExpression
        ;

statementElse
:
    ELSE block      #statemElse
;

location
: ID | ID '[' expression ']'
  | '.' location
;

expression
:
    andExpr
  |
    expression cond_op_or andExpr
;

andExpr
:
    eqExpr          #andExprEqExpr
  |
    andExpr cond_op_and eqExpr  #andExprCondOpAnd
;

eqExpr
:
    relationExpr      #eqExprRelationExpr
  |
    eqExpr eq_op relationExpr  #eqExprEqOp
;

relationExpr
:
    addExpr          #relExprAddExpre
  |
    relationExpr rel_op addExpr  #relExprRelOp
;

addExpr
:
    multExpr          #addExprMultExpr
  |
    addExpr minusplus_op multExpr  #addExprMinusPlusOp
;

multExpr
:
    unaryExpr          #multExprUnary
  |
    multExpr multdiv_op unaryExpr  #multExprMultDivOp
;

unaryExpr

```

```

:      '('(INT|CHAR)')' value
|      '-' value
|      '!' value
|      value
;

value
:      location                #valueLocation
|      methodCall             #valueMethodCall
|      literal                 #valueLiteral
|      '(' expression ')'      #valueExprWithParent
;

methodCall
:      ID '(' (parameter | ( parameter (',' parameter)*)) ? ')' #methodCallDecl
;

arg
:      expression
;

minusplus_op
:      '+'
|      '-'
;

multdiv_op
:      '*'
|      '/'
|      '%'
;

rel_op
:      '<'
|      '>'
|      '<='
|      '>='
;

eq_op
:      '=='
|      '!='
;

```

```
cond_op_or : '||';  
cond_op_and: '&&';
```

```
literal  
  :    int_literal  
  |    char_literal  
  |    boolean_literal  
  ;
```

```
int_literal  
  :    NUM  
  ;
```

```
char_literal  
  :    Char  
  ;
```

```
boolean_literal  
  :    'true'  
  |    'false'  
  ;
```

Reglas Semánticas validadas

- Ningún identificador es declarado dos veces en el mismo ámbito
- Ningún identificador es utilizado antes de ser declarado.
- El programa contiene una definición de un método main sin parámetros, en donde se empezará la ejecución del programa.
- num en la declaración de un arreglo debe de ser mayor a 0.
- El número y tipos de argumentos en la llamada a un método deben de ser los mismos que los argumentos formales, es decir las firmas deben de ser idénticas.
- Si un método es utilizado en una expresión, este debe de devolver un resultado.
- La instrucción return no debe de tener ningún valor de retorno, si el método se declara del tipo void.
- El valor de retorno de un método debe de ser del mismo tipo con que fue declarado el método.
- Si se tiene la expresión id[<expr>], id debe de ser un arreglo y el tipo de <expr> debe de ser int.
- El tipo de <expr> en la estructura if y while, debe de ser del tipo boolean.
- Los tipos de operandos para los operadores <arith_op> y <rel_op> deben de ser int.
- Los tipos de operandos para los operadores <eq_ops> deben de ser int, char o boolean, y además ambos operandos deben de ser del mismo tipo.
- Los tipos de operandos para los operadores <cond_ops> y el operador ! deben de ser del tipo boolean
- El valor del lado derecho de una asignación, debe de ser del mismo tipo que la locación del lado izquierdo.

Oportunidades de mejora

- Validar estructuras mas complejas.
- En los tipos, validar el resto de estructuras de expresiones. En los la evaluaciond e retorno
- Comprobar tipos en los argumentos

Comentarios personales

Considero que el análisis semítico es una parte del compilador muy importante para prevenir que se ingresen acciones que causaran problemas en la generación de código intermedio. Adema vale la pena mencionar que es una parte bastante tediosa en la parte del diseño de un compilador.

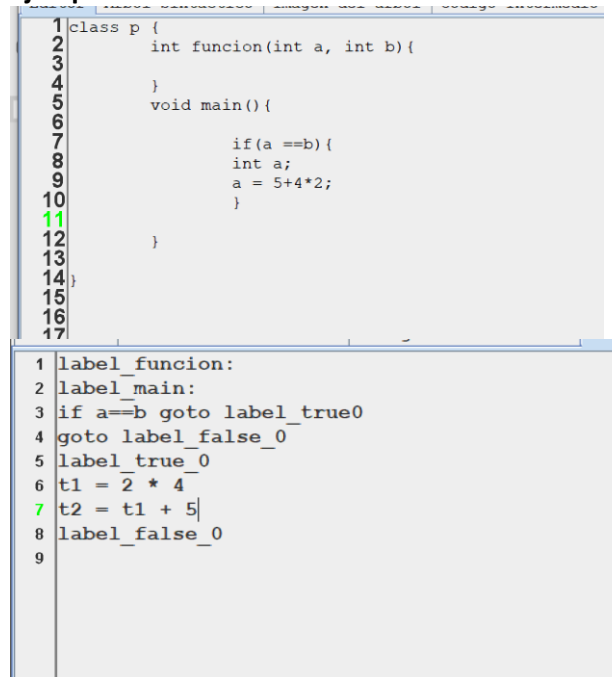
Fase #2: código intermedio:

Lenguaje utilizado: código de tres direcciones. Esto debido a su similitud con el lenguaje de Assembler en ARM. Este consiste en una dirección de destino, y 2 que son las temporales. En el caso de sentencias if, se genera etiquetas de salto condicional.

Ejemplos de funcionamiento:

A continuación, se presenta como se generan algunas estructuras de código intermedio

Ejemplos de if:



The image shows a code editor with two panels. The top panel displays C code with line numbers 1 through 17. The code defines a function 'funcion' and a 'main' function. Inside 'main', there is an 'if' statement: 'if (a == b) { int a; a = 5 + 4 * 2; }'. The bottom panel displays the corresponding assembly code with line numbers 1 through 9. It uses labels like 'label_funcion:', 'label_main:', 'label_true_0', and 'label_false_0' to implement the conditional logic using 'goto' and arithmetic instructions.

```
1 class p {  
2     int funcion(int a, int b){  
3  
4     }  
5     void main() {  
6  
7         if (a == b) {  
8             int a;  
9             a = 5 + 4 * 2;  
10        }  
11    }  
12 }  
13  
14 }  
15  
16  
17 }  
  
1 label_funcion:  
2 label_main:  
3 if a==b goto label_true0  
4 goto label_false_0  
5 label_true_0  
6 t1 = 2 * 4  
7 t2 = t1 + 5  
8 label_false_0  
9
```

Ejemplos de if y else:

```
1 class p {
2
3     void main() {
4         if(true) {
5             int a;
6             a = 5+4*2;
7         }
8         else {
9             int a;
10            a = 8;
11        }
12    }
13 }
14
15
16
17
18
```

Error en línea:8, 2. "true" no declarado

```
1 class p {
2
3     void main() {
4         if(true) {
5             int a;
6             a = 5+4*2;
7         }
8         else {
9             int a;
10            a = 8;
11        }
12    }
13 }
14
15
16
17
18
```

Error en línea:8, 2. "true" no declarado

Llamda a métodos

```
    }
    while(true) {
        funcion(5,3);
    }
```



```

12 goto label_while_false_0
13 label_while_true_0
14 param 5
15 param 3
16 call function
17 label_while_false_0
18

```

While

```

    a = 0,
    }
    while(true){
        function(5,3);
    }
}

```

```

12 goto label_while_false_
13 label_while_true_0
14 param 5
15 param 3
16 call function
17 label_while_false_0
18

```

Instrucciones aritméticas

```
void main() {
```

```
    int a;
```

```
    a = 5+4*2;
```

```
}
```

```
}
```

Archivo | intermedio#2.txt

Editor Arbol Sintactico Imagen del arbol Cod

```
1 label_funcion:
2 label_main:
3 t1 = 2 * 4
4 t2 = t1 + 5
5
6
```

Etiquetas de métodos

Editor Arbol Sintactico Imagen del arbol Co

```
1 label_funcion:
2 label_main:
3
```

Generación de código objeto (Assembler ARM)

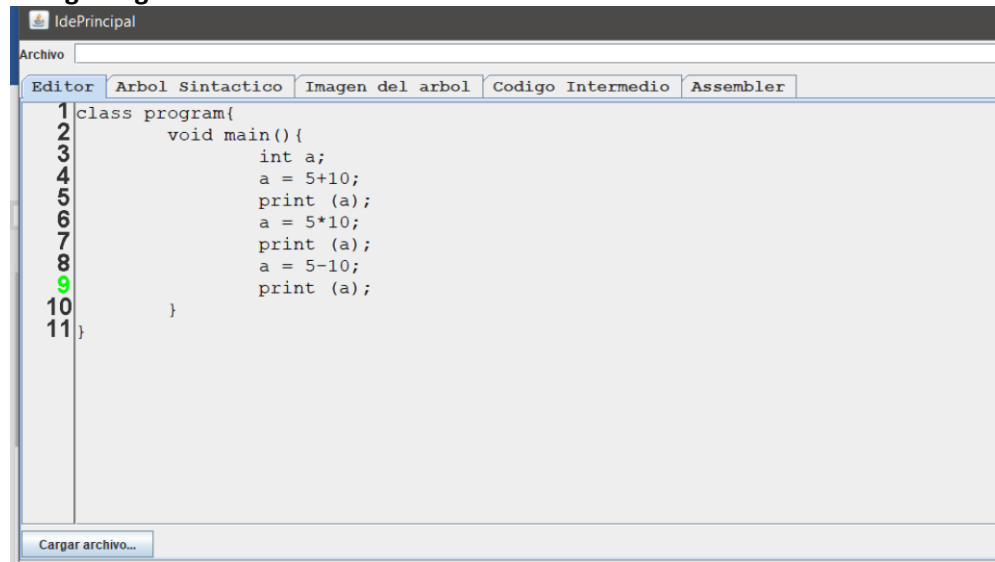
Es un lenguaje de bajo nivel desarrollado para procesadores móviles, cuenta con 15 registros de los cuales son utilizables 12 (R0-R12) con un stack pointer (R14), un link register (R13) y un program counter (R15). Cuenta con un espacio de memoria cuyo tamaño de localidad máxima es un word de 32bits.

Complemento del diseño de la aplicación.

En esta última fase únicamente se habilito el panel de generación de código Assembler, en donde se muestra el resultado de la compilación.

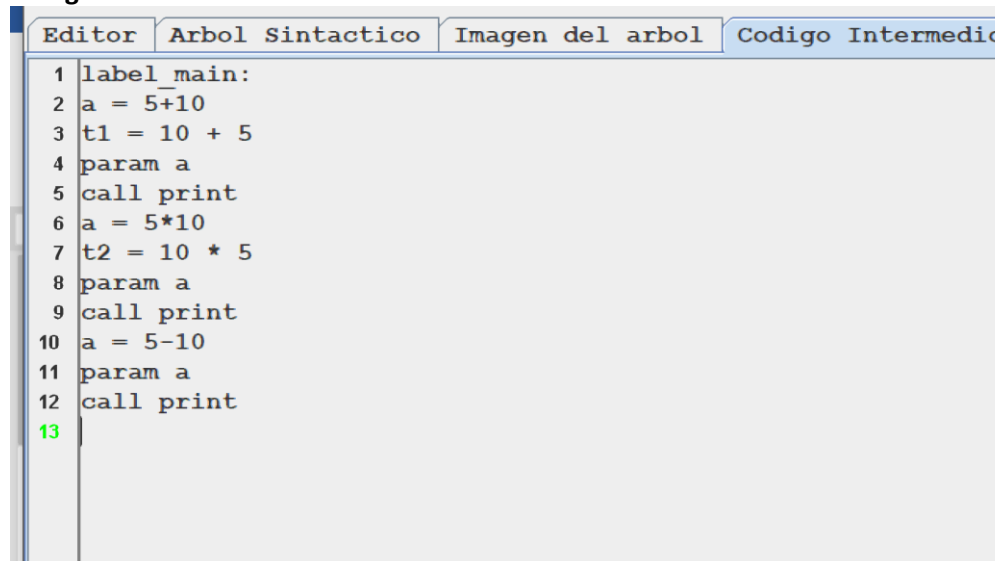
Pruebas de ejecución de código.

Código original



```
1 class program{
2     void main(){
3         int a;
4         a = 5+10;
5         print (a);
6         a = 5*10;
7         print (a);
8         a = 5-10;
9         print (a);
10    }
11 }
```

Código Intermedio



```
1 label_main:
2 a = 5+10
3 t1 = 10 + 5
4 param a
5 call print
6 a = 5*10
7 t2 = 10 * 5
8 param a
9 call print
10 a = 5-10
11 param a
12 call print
13
```

Código Assembler ARM

```
9  .type main,@function
10
11 main:
12     stmfd sp!, {lr}
13     MOV R3,# 10
14     MOV R4,# 5
15     ADD R1 , R3, R4
16     MOV R1, R1
17     LDR R0, =formatoDecimal
18     BL printf
19
20     MOV R3,# 10
21     MOV R4,# 5
22     MUL R2 , R3, R4
23     MOV R1, R2
24     LDR R0, =formatoDecimal
25     BL printf
26     MOV R1, R2
27     LDR R0, =formatoDecimal
28     BL printf
29     /* salida correcta */
30 fin:
31     mov r0, #0
```