



Instituto Tecnológico de Costa Rica

Escuela de Ingeniería en Computación

Curso: IC-5701 Compiladores e intérpretes

1° semestre 2023

Proyecto I

Estudiantes:

Celina Madrigal Murillo - 2020059364

Gabriel Mora Estriquí - 2019048848

María José Porras Maroto - 2019066056

Profesor:

Ignacio Trejos Zelaya

Grupo: 2

Fecha de entrega:

Martes 02 de mayo del 2023

Índice

1. Esquema para el manejo del texto fuente	3
2. Modificaciones hechas al analizador de léxico	3
2.1 Parser	3
2.2 Scanner	3
2.3 Token	6
3. Cambios hechos a los tokens y a cualquier otra estructura de datos	7
4. Estrategia para generar la versión HTML.	11
5. Cambios realizados a las reglas sintácticas de Axt.	20
6. Nuevas rutinas de reconocimiento sintáctico	43
7. Lista de nuevos errores sintácticos detectados	44
8. Modelaje realizado para los árboles de sintaxis abstracta	45
9. Extensión realizada a los métodos que permiten visualizar los árboles de sintaxis abstracta	45
10. Extensión realizada a los métodos que permiten representar los árboles de sintaxis abstracta como texto en XML.	46
11. Plan de pruebas para validar el compilador	59
11.2 Pruebas del Análisis Léxico	108
Prueba For while do end	111
12. Reflexión	112
13. Descripción resumida de las tareas realizadas por cada miembro del grupo de trabajo.	113
14. Indicaciones de uso del programa	114
13.1 Cómo compilar el programa.	114
13.2 Cómo ejecutar el programa.	119
15. Anexos	120
16. Referencias	122

1. Esquema para el manejo del texto fuente

No se realizaron cambios en la forma en que el compilador lee los archivos de entrada, por lo tanto, se mantuvo el mismo esquema anteriormente utilizado.

2. Modificaciones hechas al analizador de léxico

Se llevaron a cabo varios cambios en distintas partes del analizador léxico, por lo que a continuación se detallan las modificaciones realizadas en cada una de las clases pertenecientes al paquete "Syntactic Analyzer".

2.1 Parser

En primer lugar, se eliminaron del single command las alternativas de comando vacío y begin, y se agregaron nuevas opciones como skip, repeat y for. Además, se realizaron modificaciones en alternativas ya existentes como let e if.

Asimismo, se hicieron cambios en la sintaxis de declaration y se añadió una nueva regla al single declaration.

Por otro lado, se introdujeron nuevas reglas para proc funcs, y se realizó una modificación en la regla de proc funcs dentro de single declaration, agregando un comando "end" al final. Por último, se incluyó un nuevo tipo de declaración dentro de la single declaration.

2.2 Scanner

En la clase de Scanner, el primer cambio fue la consideración de los nuevos tokens agregados al lenguaje. Esta modificación afectaba el método de scanToken() para que pudiese reconocer los nuevos símbolos léxicos. A continuación, se muestran las adiciones al método.

```

case ',':
    takeIt();
    if (currentChar == ',') {
        takeIt();
        archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> "+ "," +" </tt></font> ");
        return Token.DOTDOT; //Se agregó el ..
    } else
        archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> "+ "." +" </tt></font> ");
        return Token.DOT;

case ':':
    takeIt();
    if (currentChar == '=') {
        takeIt();
        archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> "+ "=" +" </tt></font> ");
        return Token.BECOMES; //Se agregó :=
    } else
        archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> "+ ":" +" </tt></font> ");
        return Token.COLON;

case ';':
    takeIt();
    archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> "+ ";" +" </tt></font> ");
    return Token.SEMICOLON;

case ',':
    takeIt();
    archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> "+ "," +" </tt></font> ");
    return Token.COMMA;

case '~':
    takeIt();
    archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> "+ "~" +" </tt></font> ");
    return Token.IS;

//Se agregó BAR
case '|':
    takeIt();
    archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> "+ "|" +" </tt></font> ");
    return Token.BAR;

```

```

//Se agregó DOLLAR
case '$':
    takeIt();
    archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> +" + "$" +" </tt></font> ");
    return Token.DOLLAR;

case '(':
    takeIt();
    archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> +" + "(" +" </tt></font> ");
    return Token.LPAREN;

case ')':
    takeIt();
    archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> +" + ")" +" </tt></font> ");
    return Token.RPAREN;

case '[':
    takeIt();
    archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> +" + "[" +" </tt></font> ");
    return Token.LBRACKET;

case ']':
    takeIt();
    archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> +" + "]" +" </tt></font> ");
    return Token.RBRACKET;

case '{':
    takeIt();
    archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> +" + "{" +" </tt></font> ");
    return Token.LCURLY;

case '}':
    takeIt();
    archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> +" + "}" +" </tt></font> ");
    return Token.RCURLY;

case SourceFile.EOT:
    return Token.EOT;

```

Por otra parte, el scanner resulta imprescindible para la generación del archivo HTML, por lo que esto significó algunas modificaciones y adiciones. En primer lugar se agregó un constructor al Scanner que tome en cuenta la generación del archivo HTML según el programa fuente a probar.

```

public Scanner(SourceFile source, ArchivoHTML archivo) {
    sourceFile = source;
    currentChar = sourceFile.getSource();
    this.archivoHTML = archivo;
    debug = false;
}
public void enableDebugging() {
    debug = true;
}

```

La incorporación del archivo a las propiedades del Scanner permitirán más adelante la escritura de los tokens según vayan siendo identificados. Lo que se relaciona al siguiente cambio.

```

public Token scan () {
    Token tok;
    SourcePosition pos;
    int kind;
    currentlyScanningToken = false;
    while (currentChar == '!'
        || currentChar == ' '
        || currentChar == '\n'
        || currentChar == '\r'
        || currentChar == '\t')
        scanSeparator();

    currentlyScanningToken = true;
    currentSpelling = new StringBuffer("");
    pos = new SourcePosition();
    pos.start = sourceFile.getCurrentLine();

    kind = scanToken();

    pos.finish = sourceFile.getCurrentLine();
    tok = new Token(kind, currentSpelling.toString(), pos);
    archivoHTML.HTML(tok);
    if (debug)
        System.out.println(tok);
    return tok;
}

```

La función de scan() es utilizada con frecuencia en los métodos de la clase Parser para la identificación de los distintos componentes del programa fuente ingresado. Tomando en cuenta la aplicación del scan() se agregó la creación del archivo HTML mediante la función HTML cada vez que se use el método.

2.3 Token

En primer lugar, se eliminó la palabra reservada "begin", la cual ya no es necesaria en el programa. Además, se agregaron nuevas palabras reservadas, tales como "for", "from", "package", "private", "rec", "repeat", "select", "skip", "times", "to", "until" y "when" como nuevas alternativas en la especificación de Token.

Asimismo, se incluyeron bar ("|"), dollar (\$) y dotdot (..) como nuevos símbolos léxicos (de puntuación) en la especificación de Token.

Para que el programa reconociera correctamente todas estas nuevas palabras, se llevó a cabo un proceso de re-indexación.

3. Cambios hechos a los tokens y a cualquier otra estructura de datos

Como se mencionó anteriormente, la clase "token" fue modificada principalmente para incorporar nuevas palabras reservadas y símbolos léxicos. En primer lugar, se eliminó la palabra reservada "begin". Posteriormente, se agregaron nuevas palabras reservadas como "for", "from", "package", "private", "rec", "repeat", "select", "skip", "times", "to", "until" y "when", así como también nuevos símbolos léxicos como bar ("|"), dollar (\$) y dotdot ("..").

En las siguientes imágenes se puede apreciar el aspecto final del código:

```
// Token classes...

public static final int

// literals, identifiers, operators...
INTLITERAL = 0,
CHARLITERAL = 1,
IDENTIFIER = 2,
LONG_IDENTIFIER = 3,
OPERATOR = 4,

// reserved words - must be in alphabetical order...
ARRAY = 5,
//BEGIN = 5, Eliminamos begin
CONST = 6,
DO = 7,
ELSE = 8,
END = 9,
FOR = 10, //Agregamos FOR
FROM = 11, //Agregamos FROM
FUNC = 12,
IF = 13,
IN = 14,
LET = 15,
OF = 16,
PACKAGE = 17, //Agregamos PACKAGE
PRIVATE = 18, //Agregamos PRIVATE
PROC = 19,
REC = 20, //Agregamos REC
RECORD = 21,
REPEAT = 22, //Agregamos REPEAT
SELECT = 23, //Agregamos SELECT
SKIP = 24, //Agregamos SKIP
THEN = 25,
TIMES = 26, //Agregamos TIMES
TO = 27, //Agregamos TO
TYPE = 28,
UNTIL = 29, //Agregamos UNTIL
VAR = 30,
WHEN = 31, //Agregamos WHEN
WHILE = 32,
```

```
// punctuation...
DOT = 33,
COLON = 34,
SEMICOLON = 35,
COMMA = 36,
BECOMES = 37,
IS = 38,
BAR = 39, //Agregamos BAR " | "
DOLLAR = 40, //Agregamos DOLLAR
DOTDOT = 41, //Agregamos DOTDOT

// brackets...
LPAREN = 42,
RPAREN = 43,
LBRACKET = 44,
RBRACKET = 45,
LCURLY = 46,
RCURLY = 47,

// special tokens...
EOT = 48,
ERROR = 49;
```

Luego de efectuar las modificaciones correspondientes, se incorporaron las nuevas palabras y símbolos a la tabla de tokens. En las siguientes imágenes se presenta el resultado final de la tabla:

```
private static String[] tokenTable = new String[] {  
    "<int>",  
    "<char>",  
    "<identifier>",  
    "<long_identifier>",  
    "<operator>",  
    "array",  
    // "begin",  
    "const",  
    "do",  
    "else",  
    "end",  
    "for",           //Agregamos FOR  
    "from",          //Agregamos FROM  
    "func",  
    "if",  
    "in",  
    "let",  
    "of",  
    "package",      //Agregamos PACKAGE  
    "private",       //Agregamos PRIVATE  
    "proc",  
    "rec",           // //Agregamos REC  
    "record",  
    "repeat",        //Agregamos REPEAT  
    "select",        //Agregamos SELECT  
    "skip",  
    "then",  
    "times",         //Agregamos TIMES  
    "to",            //Agregamos TO  
    "type",  
    "until",         //Agregamos UNTIL  
    "var",  
    "when",          //Agregamos WHEN  
    "while",  
    "..."};
```

```

    WHILE ,
    " " ,
    ":" ,
    ";" ,
    "=" ,
    "~" ,
    "|", //Agregamos BAR ||
    "$", //Agregamos DOLLAR
    "...", //Agregamos DOTDOT
    "(" ,
    ")" ,
    "[" ,
    "]",
    "{" ,
    "}" ,
    "" ,
    "<error>"

};


```

4. Estrategia para generar la versión HTML.

- La generación de la versión de HTML involucra varias clases y métodos en el proceso de generación. En primer lugar, tomando en cuenta la generación de archivos como especificación del proyecto, se generó un nuevo paquete llamado ArchivosSalida donde se incorporaron clases nuevas para el manejo de xml y html a crear. Dentro de este paquete se encuentra la clase ArchivoHTML.

```

public class ArchivoHTML {
    FileWriter writerHTML;
    String nombreHTML;

```

Esta clase está acompañada de un constructor que al ser llamado establece el nombre del archivo en el atributo nombreHTML. El establecer el nombre del archivo en el constructor resulta importante para el funcionamiento del método escribir.

```
public ArchivoHTML(String nombreArchivo) {
    this.nombreHTML = nombreArchivo;
}
```

El método de escribir busca agregar nuevas líneas al archivo html. El proceso para escribir en el HTML consiste en que se busca crear un objeto FileWriter sobre un archivo, que espera que se haya creado, y una condición que determina si es posible o no anexar datos que se hayan enviado como parte de los parámetros. Si no hay problemas escribe el contenido y unos cuantos saltos de línea, únicamente para cuestión de formato interno del archivo.

```
public void escribir(String content){
    try (FileWriter writerHTML = new FileWriter(nombreHTML, true)){
        writerHTML.write("\n");
        writerHTML.write(content);
        writerHTML.write("\n");
        writerHTML.close();
    } catch (IOException e) {
        System.out.println("Error: no se pudo escribir la nueva linea la HTML");
    }
}
```

Al haber comprendido estas clases, constructores y métodos es necesario explicar el proceso de creación, escritura y finalización del archivo HTML de salida. Para comenzar, una vez se ha confirmado que se desea compilar un programa fuente este inicia el método compileProgram, localizado en la clase de IDECompiler del paquete Triangle, donde se recupera el nombre del archivo del programa fuente, se crea el archivo html con el nombre esperado y se escribe la primera línea del archivo.

```
public boolean compileProgram(String sourceName) throws IOException{
    System.out.println("*****" +
        "Triangle Compiler (IDE-Triangle 1.0)" +
        "*****");

    System.out.println("Syntactic Analysis ...");
    SourceFile source = new SourceFile(sourceName);

    String nombreArchivo = sourceName.substring(0, sourceName.length()-3); // Guarda el nombre del archivo sin el tri
    ArchivoHTML archivoHTML = new ArchivoHTML(nombreArchivo+"html"); //Agrega html para que se lea como "<nombreArchivo>.html"
    archivoHTML.escribir("<!DOCTYPE html>\n<html>\n<p style='font-family: 'DejaVu Sans', monospace;'>\n"); //Iniciar formato html para el archivo

    Scanner scanner = new Scanner(source,archivoHTML);
    report = new IDEReporter();
    Parser parser = new Parser(scanner, report);
    boolean success = false;
    rootAST = parser.parseProgram();

    archivoHTML.escribir("</p>\n</html>"); //cierra el formato del html para el archivo
}
```

Además, en este punto se crea el objeto parser que permitirá la revisión sintáctica del programa fuente utilizando el método de parseProgram(). Como fue

mencionado brevemente en la sección 2 del documento, cada vez que en las funcionalidades de la clase Parser se utilice scan() se estarían revisando y generando nuevas líneas para el archivo de salida.

La escritura en el archivo se da de dos maneras distintas dependiendo del Token identificado en el scanner. Si en el método de scanToken() se reconocen separadores, estos serán escritos en la misma función.

```

case '.':
    takeIt();
    if (currentChar == '..') {
        takeIt();
        archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> "+ ".." +" </tt></font> ");
        return Token.DOTDOT; //Se agrego el ..
    } else
        archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> "+ "." +" </tt></font> ");
        return Token.DOT;

case ':':
    takeIt();
    if (currentChar == '=') {
        takeIt();
        archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> "+ "=" +" </tt></font> ");
        return Token.BECOMES; //Se agregó :=
    } else
        archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> "+ ":" +" </tt></font> ");
        return Token.COLON;

case ';':
    takeIt();
    archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> "+ ";" +" </tt></font> ");
    return Token.SEMICOLON;

case ',':
    takeIt();
    archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> "+ "," +" </tt></font> ");
    return Token.COMMA;

case '~':
    takeIt();
    archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> "+ "~" +" </tt></font> ");
    return Token.IS;

//Se agregó BAR
case '|':
    takeIt();
    archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> "+ "|" +" </tt></font> ");
    return Token.BAR;

```

```

//Se agregó DOLLAR
case '$':
    takeIt();
    archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> +" + "$" +" </tt></font> ");
    return Token.DOLLAR;

case '(':
    takeIt();
    archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> +" + "(" +" </tt></font> ");
    return Token.LPAREN;

case ')':
    takeIt();
    archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> +" + ")" +" </tt></font> ");
    return Token.RPAREN;

case '[':
    takeIt();
    archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> +" + "[" +" </tt></font> ");
    return Token.LBRACKET;

case ']':
    takeIt();
    archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> +" + "]" +" </tt></font> ");
    return Token.RBRACKET;

case '{':
    takeIt();
    archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> +" + "{" +" </tt></font> ");
    return Token.LCURLY;

case '}':
    takeIt();
    archivoHTML.escribir(" <font style='padding-left:1em' color:'black'><tt> +" + "}" +" </tt></font> ");
    return Token.RCURLY;

case SourceFile.EOT:
    return Token.EOT;

```

El reconocimiento de las palabras reservadas, así como de los operadores, identificadores y literales ocurren en scan().

```

public Token scan () {
    Token tok;
    SourcePosition pos;
    int kind;
    currentlyScanningToken = false;
    while (currentChar == '!'
        || currentChar == ' '
        || currentChar == '\n'
        || currentChar == '\r'
        || currentChar == '\t')
        scanSeparator();

    currentlyScanningToken = true;
    currentSpelling = new StringBuffer("");
    pos = new SourcePosition();
    pos.start = sourceFile.getCurrentLine();

    kind = scanToken();

    pos.finish = sourceFile.getCurrentLine();
    tok = new Token(kind, currentSpelling.toString(), pos);
    archivoHTML.HTML(tok);
    if (debug)
        System.out.println(tok);
    return tok;
}

```

La variable tok almacena la información del Token escaneado, por lo que se envía al método de HTML para su escritura.

```
public void HTML(Token currentToken){  
    switch(currentToken.kind){  
  
        //Casos para palabras reservadas, identificadores y literales  
        //Únicamente las escribe en negrita en el archivo HTML  
        case Token.ARRAY:  
        {  
            escribir("<b>"+ currentToken.spelling +"</b>");  
            break;  
        }  
        case Token.CONST:  
        {  
            escribir("<b>"+ currentToken.spelling +"</b>");  
            break;  
        }  
        case Token.DO:  
        {  
            escribir("<b>"+ currentToken.spelling +"</b>");  
            break;  
        }  
        case Token.ELSE:  
        {  
            escribir("<b>"+ currentToken.spelling +"</b>");  
            break;  
        }  
        case Token.END:  
        {  
            escribir("<b>"+ currentToken.spelling +"</b>");  
            break;  
        }  
        case Token.FOR:  
        {  
            escribir("<b>"+ currentToken.spelling +"</b>");  
            break;  
        }  
    }  
}
```

```
case Token.FROM:  
{  
    escribir("<b>"+ currentToken.spelling +"</b>");  
    break;  
}  
case Token.FUNC:  
{  
    escribir("<b>"+ currentToken.spelling +"</b>");  
    break;  
}  
case Token.IF:  
{  
    escribir("<b>"+ currentToken.spelling +"</b>");  
    break;  
}  
case Token.IN:  
{  
    escribir("<b>"+ currentToken.spelling +"</b>");  
    break;  
}  
case Token.LET:  
{  
    escribir("<b>"+ currentToken.spelling +"</b>");  
    break;  
}  
case Token.OF:  
{  
    escribir("<b>"+ currentToken.spelling +"</b>");  
    break;  
}  
case Token.PRIVATE:  
{  
    escribir("<b>"+ currentToken.spelling +"</b>");  
    break;  
}  
case Token.PROC:  
{  
    escribir("<b>"+ currentToken.spelling +"</b>");  
    break;  
}
```

```
        case Token.REC:
    {
        escribir("<b>" + currentToken.spelling + "</b>");
        break;
    }
    case Token.RECORD:
    {
        escribir("<b>" + currentToken.spelling + "</b>");
        break;
    }
    case Token.REPEAT:
    {
        escribir("<b>" + currentToken.spelling + "</b>");
        break;
    }
    case Token.SELECT:
    {
        escribir("<b>" + currentToken.spelling + "</b>");
        break;
    }
    case Token.SKIP:
    {
        escribir("<b>" + currentToken.spelling + "</b>");
        break;
    }
    case Token.THEN:
    {
        escribir("<b>" + currentToken.spelling + "</b>");
        break;
    }
    case Token.TIMES:
    {
        escribir("<b>" + currentToken.spelling + "</b>");
        break;
    }
    case Token.TO:
    {
        escribir("<b>" + currentToken.spelling + "</b>");
        break;
    }
}
```

```

case Token.TYPE:
{
    escribir("<b>" + currentToken.spelling + "</b>");
    break;
}
case Token.UNTIL:
{
    escribir("<b>" + currentToken.spelling + "</b>");
    break;
}
case Token.VAR:
{
    escribir("<b>" + currentToken.spelling + "</b>");
    break;
}
case Token.WHEN:
{
    escribir("<b>" + currentToken.spelling + "</b>");
    break;
}
case Token.WHILE:
{
    escribir("<b>" + currentToken.spelling + "</b>");
    break;
}

//Literales en azul
case Token.INTLITERAL:
{
    escribir(" <font style='padding-left:1em' color=\"blue\"><tt> " + currentToken.spelling + " </tt></font> ");
    break;
}
case Token.CHARLITERAL:
{
    escribir(" <font style='padding-left:1em' color=\"blue\"><tt> " + currentToken.spelling + " </tt></font> ");
    break;
}

```

```

//Van escritos en negro sin resaltar
case Token.IDENTIFIER:
{
    escribir(" <font style='padding-left:1em' color:'black'><tt> " + currentToken.spelling + " </tt></font> ");
    break;
}
case Token.OPERATOR:
{
    escribir(" <font style='padding-left:1em' color:'black'><tt> " + currentToken.spelling + " </tt></font> ");
    break;
}

case Token.EOT: { //el encontrar un token.EOT significa que sale de la generacion del html
    break;
}
default:
    break;
}

```

Según el tipo de Token identificado, se escribirá con las especificaciones del formato de cada uno. Por último, en el IDECompiler se escribe la línea final que da cierre a la generación del archivo HTML de salida.

```
archivoHTML.escribir("</p>\n</html>"); //cierra el formato del html para el archivo
```

Dando como resultado final un tipo de documento como el que se muestra a continuación.

```

<!DOCTYPE html>
<html>
<p style="font-family: 'DejaVu Sans', monospace;">

<b>for</b>

<font style='padding-left:1em' color:'black'><tt> x </tt></font>
<font style='padding-left:1em' color:'black'><tt> := </tt></font>
<font style='padding-left:1em' color="blue"><tt> 1 </tt></font>
<font style='padding-left:1em' color:'black'><tt> .. </tt></font>
<font style='padding-left:1em' color="blue"><tt> 10 </tt></font>

<b>while</b>

<font style='padding-left:1em' color:'black'><tt> x </tt></font>
<font style='padding-left:1em' color:'black'><tt> < </tt></font>
<font style='padding-left:1em' color="blue"><tt> 10 </tt></font>

<b>do</b>

<font style='padding-left:1em' color:'black'><tt> x </tt></font>
<font style='padding-left:1em' color:'black'><tt> := </tt></font>
<font style='padding-left:1em' color:'black'><tt> x </tt></font>
<font style='padding-left:1em' color:'black'><tt> + </tt></font>
<font style='padding-left:1em' color="blue"><tt> 1 </tt></font>

<b>end</b>

</p>
</html>

```

Con la vista de salida desde el navegador de la siguiente manera:

```
for x := 1 .. 10 while x < 10 do x := x + 1 end
```

Anotaciones sobre implementación:

La única instrucción no aplicada fue la hora de agregar comentarios al HTML final. Los comentarios, los saltos de línea y espacios generados por los tabuladores no se lograron implementar a la hora de escribir en el HTML.

5. Cambios realizados a las reglas sintácticas de Δxt.

Primeramente como ya fue mencionado en puntos anteriores se eliminó de single-Command la primera alternativa (ϵ , el comando vacío). También fueron eliminadas estas alternativas:

- | "begin" Command "end"
- | "let" Declaration "in" single-Command
- | "if" Expression "then" single-Command "else" single-Command
- | "while" Expression "do" single-Command

```
//Eliminamos begin

/* case Token.BEGIN:
   acceptIt();
   commandAST = parseCommand();
   accept(Token.END);
   break;*/

//Eliminamos let

/*case Token.LET:
{
   acceptIt();
   Declaration dAST = parseDeclaration();
   accept(Token.IN);
   Command cAST = parseSingleCommand();
   finish(commandPos);
   commandAST = new LetCommand(dAST, cAST, commandPos);
}
break;*/
```

```

//Eliminamos if

/*case Token.IF:
{
    acceptIt();
    Expression eAST = parseExpression();
    accept(Token.THEN);
    Command c1AST = parseSingleCommand();
    accept(Token.ELSE);
    Command c2AST = parseSingleCommand();
    finish(commandPos);
    commandAST = new IfCommand(eAST, c1AST, c2AST, commandPos);
}
break;*/

/*case Token.WHILE:
{
    acceptIt();
    Expression eAST = parseExpression();
    accept(Token.DO);
    Command cAST = parseSingleCommand();
    finish(commandPos);
    commandAST = new WhileCommand(eAST, cAST, commandPos);
}
break;

case Token.SEMICOLON:
case Token.END:
case Token.ELSE:
case Token.IN:
case Token.EOT:

    finish(commandPos);
    commandAST = new EmptyCommand(commandPos);
}
break;*/

```

Luego de estas eliminaciones se añadieron nuevas alternativas a single-Command. En primer lugar, se ha introducido el comando "**skip**", que cumple la función de un comando vacío. Este solamente acepta el token "SKIP", termina la posición y crea un nuevo "EmptyCommand". Se ve de la siguiente manera:

```

case Token.SKIP:
{
    acceptIt();
    finish(position: commandPos);
    commandAST = new EmptyCommand(thePosition: commandPos);
}
break;

```

Posteriormente se añadió "**let**" **Declaration** "in" **Command** "end" el cual ya existía previamente, pero fue modificado para requerir obligatoriamente la presencia de un "end". El proceso comienza por aceptar el token "LET", seguido del análisis sintáctico de la declaración. Posteriormente, se acepta el token "IN" y se realiza el análisis sintáctico del comando. Una vez finalizado esto, se acepta el token "END", finaliza la posición y se crea un nuevo "LetCommand". El código se ve de la siguiente manera:

```

case Token.LET:
{
    acceptIt();
    Declaration dAST = parseDeclaration();
    accept(tokenExpected: Token.IN);
    Command cAST = parseCommand();
    accept(tokenExpected: Token.END);
    finish(position: commandPos);
    commandAST = new LetCommand(dAST, cAST, thePosition: commandPos);

}
break;

```

La siguiente alternativa añadida fue "if" **Expression** "then" **Command** ("|" **Expression** "then" **Command**)* "else" **Command** "end". En la primera parte del proceso, se comienza por aceptar el comando "IF". A continuación, se procede a realizar el análisis sintáctico de la expresión y se acepta el token "THEN". Luego, se realiza el análisis sintáctico del primer comando recibido, mientras que para el segundo se implementa un nuevo proceso de análisis llamado `parseBarThen` que incluye tanto lo que se encuentra dentro de los paréntesis como el comando "END". Finalmente, se concluye con la creación de un nuevo "ifCommand" al finalizar la posición del comando.

```
case Token.IF: {
    acceptIt(); //Se acepta el if
    Expression eAST = parseExpression(); //Se acepta el expression.
    accept(tokenExpected: Token.THEN);
    Command c1AST = parseCommand();
    Command c2AST = parseBarThen(); //
    finish(position: commandPos);
    commandAST = new IfCommand(eAST, c1AST, c2AST, thePosition: commandPos);
}
break;
```

El proceso de análisis sintáctico en el método "parseBarThen" se desarrolla de la siguiente manera: se inicia el comando en nulo y se crea una nueva posición para comenzar el análisis. A continuación, se realiza una verificación mediante una instrucción "if" para comprobar si el token recibido es "ELSE". Si es así, se acepta el token, se realiza el análisis sintáctico del comando, se acepta el token "END" y se asigna el comando analizado al comando anteriormente declarado como nulo. Si no entra en el primer "if", se procede a verificar mediante otra instrucción "if" si el token es "BAR". En caso afirmativo, se realiza una recursión en el método "parseBarThen", se acepta el token "BAR", se realiza el análisis sintáctico de la expresión, se acepta el token "THEN", se realiza el análisis sintáctico del comando y se crea un nuevo comando que llama nuevamente al método "parseBarThen". Finalmente, se concluye la posición actual, se crea un nuevo "IfCommand" y, si no se ha entrado en ninguno de los dos "if" anteriores, se muestra un mensaje de error indicando que se espera un "|" o un "ELSE".

```

Command parseBarThen() throws SyntaxError {

    Command commandAST = null;
    SourcePosition commandPos = new SourcePosition();
    start(position: commandPos);

    if (currentToken.kind == Token.ELSE) {
        acceptIt();
        Command cAST = parseCommand();
        accept(tokenExpected: Token.END);
        commandAST = cAST;
    } else if (currentToken.kind == Token.BAR) {
        acceptIt();
        Expression eAST = parseExpression();
        accept(tokenExpected: Token.THEN);
        Command cAST = parseCommand();
        Command c2AST = parseBarThen();
        finish(position: commandPos);
        commandAST = new IfCommand(eAST, c1AST: cAST, c2AST: c2AST, thePosition: commandPos);
    } else {
        syntacticError(messageTemplate:"| or else expected here", tokenQuoted:@"");
    }

    return commandAST;
}

```

Otra alternativa añadida es el **repeat**. Para este comando, se comienza por aceptar el token "REPEAT" y se crea una expresión nula junto con una posición inicial para el análisis sintáctico.

```

case Token.REPEAT: {
    acceptIt();
    Expression expressionAST = null;
    SourcePosition expressionPos = new SourcePosition();
    start(position: expressionPos);
}

```

Para el repeat existen 5 casos distintos los cuales se manejaron con un switch. El primer caso sería "**repeat**" "**while**" **Expression** "**do**" **Command** "**end**". Para este caso se verifica si el siguiente token es "WHILE". Si es así, se acepta y se crea un comando "WhileCommand" utilizando la función "whileDo". Esta función se encarga de analizar tanto el token "DO" como el token "END" del comando, así como parsear la expresión. Finalmente, se crea el comando "RepeatCommand".

```
case Token.WHILE: {
    // Crear el primer arbol
    acceptIt();
    WhileCommand While = whileDo(commandPos);
    // Crear el arbol final

    commandAST = new RepeatCommand( While, thePosition:commandPos);

    break;
}
```

La función `whileDo` se encarga de analizar la sintaxis de la estructura "WHILE...DO" utilizada en el caso 1 del comando "REPEAT". El proceso comienza en la posición especificada como parámetro, se crea un comando "WhileCommand" inicializado en nulo y se procede a analizar la expresión correspondiente al condicional del while. A continuación, se verifica si el siguiente token es el token "DO". Si es así, se acepta el token y se crea un nuevo comando "DoCommand" utilizando una nueva clase propia creada en el AbstractSyntax Trees. Este comando incluye el comando analizado previamente y la posición actual.

Luego se verifica si el siguiente token es el token "END". Si es así, se acepta, se finaliza la posición y se crea un nuevo comando "WhileCommand" que incluye la expresión, el comando "DoCommand" y la posición actual.

```

private WhileCommand whileDo(SourcePosition commandPos) throws SyntaxError{
    start(position:commandPos);
    WhileCommand commandAST = null;

    // Obtener la expresion
    Expression eAST = parseExpression();

    // Aceptar el command
    accept(tokenExpected: Token.DO);
    Command cAST = parseCommand();

    // Crear AST del Do
    DoCommand DoAST;
    DoAST = new DoCommand(cAST, thePosition:commandPos);

    if(currentToken.kind == Token.END){

        acceptIt();
        finish(position:commandPos);
        commandAST = new WhileCommand (eAST, cAST:DoAST, thePosition:commandPos);

    }

    // Error
    else{
        syntacticError(messageTemplate:"Expected END here", tokenQuoted:currentToken.spelling);
    }

    // Retornar el arbol

    return commandAST;
}

```

```

*/
public class DoCommand extends Command{
    public DoCommand (Command cAST, SourcePosition thePosition) {
        super (thePosition);
        C = cAST;
    }

    public Object visit(Visitor v, Object o) {
        return v.visitDoCommandAST(@This: this, o);
    }

    public Command C;
}

```

El segundo caso del repeat sería el "repeat" "until" **Expression "do"** **Command "end"**. En este caso Si se encuentra el token "UNTIL", se procede a crear una instancia de la clase UntilCommand utilizando la función UntilDo del módulo AbstractSyntaxTrees. Esta función se encarga de analizar el comando en su totalidad, incluyendo los tokens "DO" y "END", así como de analizar la expresión.

Luego se crea un nuevo objeto RepeatUntilAST, el cual es una clase nueva que también está definida en el módulo AbstractSyntaxTrees.

```

case Token.UNTIL: {
    acceptIt();
    UntilCommand UntilAST = UntilDo(commandPos);
    commandAST = new RepeatUntilAST( UntilAST, thePosition:commandPos);
    break;
}

```

```

public class UntilCommand extends Command{
    public Expression I;
    public Command C;

    public UntilCommand (Expression iAST, Command cAST, SourcePosition thePosition) {
        super (thePosition);
        I = iAST;
        C = cAST;
    }

    public Object visit(Visitor v, Object o) {
        return v.visitUntilCommand(this, o);
    }
}

```

La función UntilDo comienza analizando el comando a partir de la posición que se le ha pasado como parámetro. Luego, inicializa un objeto UntilCommand con valor nulo. A continuación, analiza la expresión del comando, acepta el token "DO" y analiza el comando que sigue, creando un nuevo objeto DoCommand. Luego, comprueba si el token actual es "END". Si es así, acepta el token, finaliza la posición y crea un nuevo objeto UntilCommand, utilizando la expresión, el DoCommand y la posición.

```

private UntilCommand UntilDo(SourcePosition commandPos) throws SyntaxError {
    start(position: commandPos);
    UntilCommand commandAST = null;

    Expression eAST = parseExpression();

    accept(tokenExpected: Token.DO);
    Command cAST = parseCommand();

    DoCommand DoAST;
    DoAST = new DoCommand(cAST, thePosition: commandPos);

    if(currentToken.kind == Token.END){
        acceptIt();
        finish(position: commandPos);
        commandAST = new UntilCommand(iAST: eAST, cAST: DoAST, thePosition: commandPos);
    }

    else{
        syntacticError(messageTemplate:"Expected END here", tokenQuoted:currentToken.spelling);
    }

    return commandAST;
}

```

En los casos 3 y 4, la función primero verifica que el token actual sea "DO". Si es así, acepta el token y analiza el comando que sigue. Luego, crea un nuevo objeto DoCommand, utilizando el comando analizado y la posición actual.

Para determinar si se trata del caso 3 o del caso 4, se utiliza una estructura switch. Para el caso 3, que es el comando "**repeat**" "**do**" **Command** "**while**" **Expression** "**end**", se verifica si el token actual es "WHILE". Si es así, se acepta el token y se crea un nuevo objeto DoWhileCommand utilizando la función DoWhile del módulo AbstractSyntaxTrees. Esta función se encarga de analizar tanto la expresión como el token "END". Por último, se crea un nuevo objeto RepeatDoWhileAST utilizando el DoCommand, el DoWhileCommand y la posición actual. Ambos objetos son nuevas clases definidas en el módulo AbstractSyntaxTrees.

Para el caso 4, que corresponde al comando "**repeat**" "**do**" **Command** "**until**" **Expression** "**end**", la función primero verifica si el token actual es "UNTIL". Si es así, se acepta el token y se crea un nuevo objeto DoUntilCommand utilizando la función DoUntil del módulo AbstractSyntaxTrees. Esta función se encarga de analizar tanto la expresión como el token "END". Finalmente, se crea un nuevo objeto RepeatDoUntilAST, también definido en el módulo AbstractSyntaxTrees, que utiliza el DoCommand, el DoUntilCommand y la posición actual.

```

case Token.DO: {
    acceptIt();
    Command cAST = parseCommand();

    DoCommand DoAST;
    DoAST = new DoCommand(cAST, thePosition.commandPos);

    //Verificar si es mejor cambiar a if
    switch (currentToken.kind) {
        // "repeat" "do" Command "while" Expression "end"
        case Token.WHILE:
            acceptIt();
            DoWhileCommand WhileAST = DoWhile(commandPos);
            commandAST = new RepeatDoWhileAST(cAST: DoAST, DWhile: WhileAST, commandPos);
            break;
        // "repeat" "do" Command "until" Expression "end"
        case Token.UNTIL:
            acceptIt();
            DoUntilCommand UntilAST = DoUntil(commandPos);
            commandAST = new RepeatDoUntilAST( cAST: DoAST, UntilAST, commandPos);
            break;
        default:
            syntacticError(messageTemplate:"Expected 'while' or 'until' after the command", tokenQuoted:currentToken.spelling);
            break;
    }
    break;
}

```

```

public class DoWhileCommand extends Command{
    public Expression E;

    public DoWhileCommand(Expression eAST, SourcePosition commandPos) {
        super(thePosition:commandPos);
        E = eAST;
    }

    public Object visit(Visitor v, Object o) {
        return v.visitDoWhileCommand(@This: this, o);
    }

}

```

```

public class RepeatDoWhileAST extends Command{
    public Command C;
    public Identifier I;
    public DoWhileCommand DoWhile;

    public RepeatDoWhileAST(Identifier iAST, Command cAST, DoWhileCommand DWhile, SourcePosition commandPos) {
        super(thePosition:commandPos);
        C = cAST;
        I = iAST;
        DoWhile = DWhile;
    }

    public RepeatDoWhileAST(Command cAST, DoWhileCommand DWhile, SourcePosition commandPos) {
        super(thePosition:commandPos);
        C = cAST;
        I = null;
        DoWhile = DWhile;
    }

    public Object visit(Visitor v, Object o) {
        return v.visitRepeatDoWhileCommand(@This: this, o);
    }
}

```

```

/*
public class DoUntilCommand extends Command{
    public Expression E;

    public DoUntilCommand (Expression eAST, SourcePosition thePosition) {
        super (thePosition);
        E = eAST;
    }

    public Object visit(Visitor v, Object o) {
        return v.visitDoUntilCommand(@This: this, o);
    }

}

```

```

public class RepeatDoUntilAST extends Command{

    public Command C;
    public Identifier I;
    public DoUntilCommand DoUntil;

    public RepeatDoUntilAST(Identifier iAST, Command cAST, DoUntilCommand UntilAST, SourcePosition commandPos) {
        super(thePosition.commandPos);
        C = cAST;
        I = iAST;
        DoUntil = UntilAST;
    }

    public RepeatDoUntilAST(Command cAST, DoUntilCommand UntilAST, SourcePosition commandPos) {
        super(thePosition.commandPos);
        C = cAST;
        I = null;
        DoUntil = UntilAST;
    }

    public Object visit(Visitor v, Object o) {
        return v.visitRepeatDoUntilCommand(@This: this, o);
    }
}

```

La función DoWhile comienza analizando el comando a partir de la posición que se le ha pasado como parámetro. Luego, inicializa un objeto DoWhileCommand con valor nulo. A continuación, analiza la expresión del comando y comprueba si el token actual es "END". Si es así, acepta el token, finaliza la posición actual y crea un nuevo objeto DoWhileCommand con la expresión y la posición analizadas.

```
private DoWhileCommand DoWhile(SourcePosition commandPos) throws SyntaxError {
    start(position: commandPos);
    DoWhileCommand commandAST = null;
    Expression eAST = parseExpression();

    if(currentToken.kind == Token.END) {
        acceptIt();
        finish(position: commandPos);
        commandAST = new DoWhileCommand (eAST, commandPos);
    }

    else{
        syntacticError(messageTemplate:"Expected END here", tokenQuoted:currentToken.spelling);
    }

    // Devuelve el arbol
    return commandAST;
}
```

La función DoUntil comienza analizando el comando a partir de la posición que se le ha pasado como parámetro. Luego, inicializa un objeto DoUntilCommand con valor nulo. A continuación, analiza la expresión del comando y comprueba si el token actual es "END". Si es así, acepta el token, finaliza la posición actual y crea un nuevo objeto DoUntilCommand con la expresión y la posición analizadas.

```
private DoUntilCommand DoUntil(SourcePosition commandPos) throws SyntaxError {
    start(position: commandPos);
    DoUntilCommand commandAST = null;

    // Obtener la expresion
    Expression eAST = parseExpression();

    // End
    if(currentToken.kind == Token.END){
        acceptIt();
        finish(position: commandPos);
        commandAST = new DoUntilCommand (eAST, thePosition:commandPos);
    }

    // Error
    else{
        syntacticError(messageTemplate:"Expected END here", tokenQuoted:currentToken.spelling);
    }

    // Retornar el arbol
    return commandAST;}
```

En el caso 5, correspondiente al comando "**repeat**" Expression "times" "do" **Command "end"**, se inició el análisis examinando todas las posibles declaraciones para la expresión que sigue al "repeat". Se parsearon y se verificó la presencia del token "TIMES". Luego, se declaró un TimesCommand utilizando la función TimesDo y se creó un nuevo RepeatTimesCommand.

```

    case Token.INTLITERAL:
    {
        IntegerLiteral ilAST = parseIntegerLiteral();
        finish(position:expressionPos);
        expressionAST = new IntegerExpression(ilAST, thePosition:expressionPos);

        if(currentToken.kind == Token.TIMES){
            acceptIt();

            TimesCommand times = TimesDo(commandPos);
            // Crear el arbol final

            commandAST = new RepeatTimesCommand(eAST:expressionAST, Times: times, thePosition:commandPos);

        }
        break;
    }

    case Token.CHARLITERAL:
    {
        CharacterLiteral clAST= parseCharacterLiteral();
        finish(position:expressionPos);
        expressionAST = new CharacterExpression(clAST, thePosition:expressionPos);
        if(currentToken.kind == Token.TIMES){
            acceptIt();
            TimesCommand times = TimesDo(commandPos);
            // Crear el arbol final

            commandAST = new RepeatTimesCommand( eAST:expressionAST,Times: times, thePosition:commandPos);

        }
        break;
    }
}

```

```
case Token.LBRACKET:
{
    acceptIt();
    ArrayAggregate aaAST = parseArrayAggregate();
    accept(tokenExpected: Token.RBRACKET);
    finish(position: expressionPos);
    expressionAST = new ArrayExpression(aaAST, thePosition:expressionPos);
    if(currentToken.kind == Token.TIMES) {
        acceptIt();
        TimesCommand times = TimesDo(commandPos);
        // Crear el arbol final
        commandAST = new RepeatTimesCommand( eAST:expressionAST,Times: times, thePosition:COMMANDPos);

    }
    break;
}

case Token.LCURLY:
{
    acceptIt();
    RecordAggregate raAST = parseRecordAggregate();
    accept(tokenExpected: Token.RCURLY);
    finish(position: expressionPos);
    expressionAST = new RecordExpression(raAST, thePosition:expressionPos);
    if(currentToken.kind == Token.TIMES) {
        acceptIt();

        TimesCommand times = TimesDo(commandPos);
        // Crear el arbol final

        commandAST = new RepeatTimesCommand( eAST:expressionAST,Times: times, thePosition:COMMANDPos);

    }
    break;
}
```

```

        case Token.OPERATOR:
    {
        Operator opAST = parseOperator();
        Expression eAST = parsePrimaryExpression();
        finish(position: expressionPos);
        expressionAST = new UnaryExpression(eAST, opAST, eAST, thePosition: expressionPos);
        if(currentToken.kind == Token.TIMES){
            acceptIt();
            TimesCommand times = TimesDo(commandPos);
            // Crear el arbol final

            commandAST = new RepeatTimesCommand( eAST: expressionAST, Times: times, thePosition: commandPos);

        }
        break;
    }

    case Token.LPAREN:{

        acceptIt();
        expressionAST = parseExpression();
        accept(tokenExpected:Token.RPAREN);
        if(currentToken.kind == Token.TIMES){
            acceptIt();
            TimesCommand times = TimesDo(commandPos);
            // Crear el arbol final
            commandAST = new RepeatTimesCommand( eAST: expressionAST, Times: times, thePosition: commandPos);

        }
        break;
    }

    //////////

    default:
        syntacticError(messageTemplate: "Expected 'while', 'do', 'until' or an expression after repeat", tokenQuoted: currentToken.spelling);
        break;
    }
}

```

```

/*
public class TimesCommand extends Command {

    public TimesCommand (Command cAST, SourcePosition thePosition) {
        super (thePosition);
        C = cAST;
    }

    public Object visit(Visitor v, Object o) {
        return v.visitTimesCommand(ast:this, o);
    }

    public Command C;
}

```

La función TimesDo comienza en la posición proporcionada como parámetro, inicializa un TimesCommand nulo, acepta el token "DO" y analiza el comando que le sigue. A continuación, crea un nuevo DoCommand y verifica si el token actual es

"END". Si lo es, termina la posición y crea un nuevo TimesCommand con el DoCommand y la posición actual.

```

    }

    private TimesCommand TimesDo(SourcePosition commandPos) throws SyntaxError{

        start(position: commandPos);
        TimesCommand TcommandAST = null;

        // Aceptar el command
        accept(tokenExpected:Token.DO);
        Command cAST = parseCommand();

        // Crear AST del Do
        DoCommand DoAST;
        DoAST = new DoCommand(cAST, thePosition: commandPos);

        if(currentToken.kind == Token.END){

            acceptIt();
            finish(position: commandPos);
            TcommandAST = new TimesCommand ( cAST: DoAST, thePosition: commandPos);

        }

        // Error
        else{
            syntacticError(messageTemplate: "Expected END here", tokenQuoted: currentToken.spelling);
        }

        // Retornar el arbol}

        return TcommandAST;
    }
}

```

Se añadió otra alternativa al lenguaje que es el comando "for", que tiene tres casos distintos. En cada uno de ellos, lo primero que se hace es verificar si el token actual es "FOR" y, en caso afirmativo, se acepta y se procede a analizar el identificador asociado al ciclo. Luego pregunta si el token actual es "BECOMES", si es así lo acepta y crea un ForBecomesCommand con la función ParseForBecomesCommand, la cual empieza en la posición pasada por parámetro, luego inicializa un ForBecomesCommand en nulo, parsea la expresión, termina la posición y crea un nuevo ForBecomesCommand en el inicializado anteriormente. Volviendo al for, luego de hacer todo lo anterior se acepta el token "DOTDOT", se parsea la expresión y se crea un nuevo DotDCommand, el cual es una nueva clase creada en AbstractSyntaxTrees.

En el primer caso del comando "for", que tiene la forma "**for Identifier ":= Expression .. Expression "do Command "end"**", se verifica si el token actual es "DO". Si es así, se acepta y se utiliza la función ParseDoCommand para crear un DoCommand. En dicha función, se comienza en la posición proporcionada como parámetro, se inicializa un DoCommand nulo, se analiza el comando, se acepta el token "END", se termina la posición y se crea un nuevo DoCommand. Finalmente, se crea un nuevo ForBecomesAST utilizando la nueva clase definida en AbstractSyntaxTrees.

En el segundo caso del comando "for", que tiene la forma "**for Identifier ":= Expression .. Expression "while Expression "do Command "end"**" primero se verifica si el token actual es "WHILE", si es así se acepta y se utiliza la función whileDo para crear un WhileCommand. Esta función ya fue explicada en el primer caso del "repeat". Por último, se crea un nuevo RepeatForWhile, la cual es una nueva clase creada en AbstractSyntaxTrees.

En el tercer caso, que tiene la forma "for Identifier ":= Expression .. Expression "until Expression "do Command "end", primero se verifica si el token actual es "UNTIL" si es así se acepta y se crea un UntilCommand con la función UntilDo, la cual ya fue explicada en el segundo caso del "repeat". Por último, se crea un nuevo RepeatForUntil, el cual es una nueva clase creada en AbstractSyntaxTrees.

```

case Token.FOR: {
    acceptIt();
    Identifier iASTF = parseIdentifier();
    if (currentToken.kind == Token.BECOMES) {
        acceptIt();
        ForBecomesCommand ForBecomesAST = ParseForBecomesCommand(commandPos, iASTF);

        accept(tokenExpected:Token.DOTDOT);

        Expression eASTF = parseExpression();

        DotDCommand DDoAST ;
        DDoAST = new DotDCommand(eAST: eASTF, thePosition: commandPos);

        if(currentToken.kind == Token.DO) {
            acceptIt();
            DoCommand ForDoAST = ParseDoCommand(commandPos);
            commandAST = new ForBecomesAST( ForBecomesV: ForBecomesAST, eAST: DDoAST, Dovar: ForDoAST, thePosition: commandPos);
        }
        else if (currentToken.kind == Token.WHILE) {
            acceptIt();

            WhileCommand WhileAST = whileDo(commandPos);

            commandAST = new RepeatForWhile( ForFromVar: ForBecomesAST, eAST: DDoAST, whileAST: WhileAST, thePosition: commandPos);
        }
        else if (currentToken.kind == Token.UNTIL) {
            acceptIt();

            UntilCommand UntilAST = UntilDo(commandPos);

            commandAST = new RepeatForUntil(ForFromVar: ForBecomesAST, eAST: DDoAST, whileAST: UntilAST, thePosition: commandPos);
        }
        else{
            syntacticError(messageTemplate: "Expected 'do', 'while' or 'until' ", tokenQuoted: currentToken.spelling);
        }
        break;
}

```

```

private ForBecomesCommand ParseForBecomesCommand(SourcePosition commandPos, Identifier iASTF) throws SyntaxError {
    start(position: commandPos);
    ForBecomesCommand commandAST = null;
    Expression eAST = parseExpression();
    finish(position: commandPos);
    commandAST = new ForBecomesCommand(iAST: iASTF, eAST, thePosition: commandPos);
    return commandAST;
}

```

```

public class DotDCommand extends Command{

    public DotDCommand(Expression eAST, SourcePosition thePosition) {
        super(thePosition);
        CLC = eAST;
    }

    public Object visit(Visitor v, Object object) {
        return v.visitDotDCommandAST(@This: this, @: object);
    }

    public Expression CLC;

}

```

```

private DoCommand ParseDoCommand(SourcePosition commandPos) throws SyntaxError {
    start(position: commandPos);
    DoCommand commandAST = null;
    Command cAST = parseCommand();
    accept(tokenExpected:Token.END);
    finish(position: commandPos);
    commandAST = new DoCommand(cAST, thePosition: commandPos);
    return commandAST;
}

```

```

public class RepeatForWhile extends Command{
    public ForBecomesCommand ForBecomes;
    public DotDCommand E;
    public WhileCommand whileC;
    public Identifier I;

    public RepeatForWhile(Identifier iAST, ForBecomesCommand ForFromVar, DotDCommand eAST, WhileCommand whileAST, SourcePosition thePosition) {
        super(thePosition);
        I = iAST;
        ForBecomes = ForFromVar;
        E = eAST;
        whileC = whileAST;
    }
    public RepeatForWhile(ForBecomesCommand ForFromVar, DotDCommand eAST, WhileCommand whileAST, SourcePosition thePosition) {
        super(thePosition);
        I = null;
        ForBecomes = ForFromVar;
        E = eAST;
        whileC = whileAST;
    }
    @Override
    public Object visit(Visitor v, Object o) {
        return v.visitRepeatForWhile(this, o);
    }
}

```

```

public class RepeatForUntil extends Command{
    public ForBecomesCommand ForBecomes;
    public DotDCommand E;
    public UntilCommand UntilC;
    public Identifier I;

    public RepeatForUntil(Identifier iAST, ForBecomesCommand ForFromVar, DotDCommand eAST, UntilCommand whileAST, SourcePosition thePosition) {
        super(thePosition);
        I = iAST;
        ForBecomes = ForFromVar;
        E = eAST;
        UntilC = whileAST;
    }
    public RepeatForUntil(ForBecomesCommand ForFromVar, DotDCommand eAST, UntilCommand whileAST, SourcePosition thePosition) {
        super(thePosition);
        I = null;
        ForBecomes = ForFromVar;
        E = eAST;
        UntilC = whileAST;
    }
    @Override
    public Object visit(Visitor v, Object o) {
        return v.visitRepeatForUntil(this, o);
    }
}

```

Otra de las modificaciones realizadas fue al **declaration**, el cual ahora luce del a siguiente manera:

```
Declaration parseDeclaration() throws SyntaxError {
    Declaration declarationAST = null; // in case there's a syntactic error

    SourcePosition declarationPos = new SourcePosition();
    start(position:declarationPos);
    declarationAST = parseCompoundDeclaration();

    //declarationAST = parseSingleDeclaration();
    while (currentToken.kind == Token.SEMICOLON) {
        acceptIt();
        Declaration d2AST = parseSingleDeclaration();
        finish(position:declarationPos);
        declarationAST = new SequentialDeclaration(d1AST: declarationAST, d2AST,
            thePosition:declarationPos);
    }
    return declarationAST;
}
```

Dentro del `declaration` se realiza un análisis sintáctico de una "CompoundDeclaration". Luego, se verifica si el token actual es un punto y coma ("SEMICOLON"). Si se cumple esta condición, se acepta, se realiza otro análisis sintáctico de una "CompoundDeclaration", se finaliza la posición y se crea un nuevo objeto de la clase "SequentialDeclaration". Por último, se devuelve el valor de la variable "declarationAST".

```
// REVISAR
Declaration parseCompoundDeclaration() throws SyntaxError{

    Declaration declarationAST = null;
    SourcePosition position = new SourcePosition();
    start(position);
    switch(currentToken.kind) {
        case Token.CONST:
        case Token.VAR:
        case Token.PROC:
        case Token.FUNC:
        case Token.TYPE:
            declarationAST = parseSingleDeclaration();
            finish(position);
            break;
        case Token.REC:
            acceptIt();
            declarationAST = parseProcFunc();
            accept(tokenExpected: Token.END);
            finish(position);
            break;
        case Token.PRIVATE:
            acceptIt();
            Declaration dAST = parseDeclaration();
            accept(tokenExpected: Token.IN);
            Declaration dAST2 = parseDeclaration();
            accept(tokenExpected: Token.END);
            finish(position);
            declarationAST = new PrivateDeclaration(dAST: dAST, dAST: dAST2, thePosition:position);
            break;
        default:
            syntacticError(messageTemplate:"\"%\" cannot start a declaration.",
                           tokenQuoted:CurrentToken.spelling);
            break;
    }
    return declarationAST;
}
```

También se agregaron nuevas reglas las cuales son las siguientes:

```

Declaration parseProcFunc() throws SyntaxError {
    Declaration procFuncAST = null;
    SourcePosition position = new SourcePosition();
    start(position);

    switch (currentToken.kind) {
        case Token.PROC:
            accept(tokenExpected: Token.PROC);
            Identifier iAST = parseIdentifier();
            accept(tokenExpected: Token.LPAREN);
            FormalParameterSequence formalAST = parseFormalParameterSequence();
            accept(tokenExpected: Token.RPAREN);
            accept(tokenExpected: Token.IS);
            Command cAST = parseCommand();
            accept(tokenExpected: Token.END);
            finish(position);
            procFuncAST = new ProcDeclaration(iAST, fpsAST, formalAST, cAST, thePosition:position);
            break;

        case Token.FUNC:
            accept(tokenExpected: Token.FUNC);
            Identifier identifierAST = parseIdentifier();
            accept(tokenExpected: Token.LPAREN);
            FormalParameterSequence fpsAST = parseFormalParameterSequence();
            accept(tokenExpected: Token.RPAREN);
            accept(tokenExpected: Token.COLON);
            TypeDenoter tAST = parseTypeDenoter();
            accept(tokenExpected: Token.IS);
            Expression eAST = parseExpression();
            finish(position);
            procFuncAST = new FuncDeclaration(identifierAST, fpsAST, tAST, eAST, thePosition:position);
            break;

        default:
            syntacticError(messageTemplate:"Expected here a proc or func", tokenQuoted:currentToken.spelling);
    }
}

```

Se ha realizado una modificación en "single declaration", específicamente en la sección de "PROC". En esta parte, se ha reemplazado el "single command" por un "command" y se ha agregado el token "END".

```

case Token.PROC:
{
    acceptIt();
    Identifier iAST = parseIdentifier();
    accept(tokenExpected: Token.LPAREN);
    FormalParameterSequence fpsAST = parseFormalParameterSequence();
    accept(tokenExpected: Token.RPAREN);
    accept(tokenExpected: Token.IS);
    Command cAST = parseSingleCommand();

    if (currentToken.kind == Token.END) {
        acceptIt();
        finish(position:declarationPos);
        declarationAST = new ProcDeclaration(iAST, fpsAST, cAST, thePosition:declarationPos);
    }
    else {
        syntacticError(messageTemplate:"Expected end after the command", tokenQuoted:currentToken.spelling);
    }
}

```

Y por último también se modificó el single declaration en la parte de “VAR” para que se pudiera inicializar una variable tanto con el token “COLON” como con el “BECOMES”.

```

case Token.VAR:
{
    acceptIt();
    Identifier iAST = parseIdentifier();
    if(currentToken.kind == Token.COLON || currentToken.kind == Token.BECOMES) {
        acceptIt();
        TypeDenoter tAST = parseTypeDenoter();
        finish(position:declarationPos);
        declarationAST = new VarDeclaration(iAST, tAST, thePosition:declarationPos);
    }
    else{
        syntacticError(messageTemplate:"Expected ':' or ':='", tokenQuoted:currentToken.spelling);
    }
}
break;

```

6. Nuevas rutinas de reconocimiento sintáctico

- Se eliminó de single-Command la primera alternativa (epsilon, comando vacío).
- Se eliminó del single-Command las siguientes alternativas:
 | "begin" Command "end"
 | "let" Declaration "in" single-Command
 | "if" Expression "then" single-Command "else" single-Command
 | "while" Expression "do" single-Command
- Se conservaron las siguientes alternativas en single-Command: |V-name ":=" Expression
- Se añadió al single-Command las siguientes reglas:
 "skip"
 | "let" Declaration "in" Command "end"
 | "if" Expression "then" Command "else" Command "end"
 | "repeat" "while" Expression "do" Command "end"
 | "repeat" "until" Expression "do" Command "end"
 | "repeat" "do" Command "while" Expression "end"
 | "repeat" "do" Command "until" Expression "end"
 | "repeat" Expression "times" "do" Command "end"
 | "for" Identifier ":=" Expression ".." Expression "do" Command "end"
 | "for" Identifier ":=" Expression ".." Expression "while" Expression "do" Command "end"
 | "for" Identifier ":=" Expression ".." Expression "until" Expression "do" Command "end"

- Se modificaron las siguientes reglas:

$$\begin{aligned} \text{Declaration} ::= & \text{compound-Declaration} \\ | & \text{Declaration ";" compound-Declaration} \end{aligned}$$
- Se añadieron estas nuevas reglas en compound-Declaration

$$\begin{aligned} \text{compound-Declaration} ::= & \text{single-Declaration} \\ | & \text{"rec" Proc-Funcs "end"} \\ | & \text{"private" Declaration "in" Declaration "end"} \end{aligned}$$
- Se añadieron las siguientes reglas:

$$\begin{aligned} \text{Proc-Func} ::= & \text{"proc" Identifier "(" Formal-Parameter-Sequence ")" "~" Command "end"} \\ | & \text{"func" Identifier "(" Formal-Parameter-Sequence ")" ":" Type-denoter} \\ | & \text{"~" Expression} \\ \text{Proc-Funcs} ::= & \text{Proc-Func ("|" Proc-Func)+} \end{aligned}$$
- En single-Declaration, se modificó la opción referente a proc:

$$\begin{aligned} | & \text{"proc" Identifier "(" Formal-Parameter-Sequence ")" "~" Command} \\ & \text{"end"} \\ | & \text{...} \end{aligned}$$
- Se añadió a single-Declaration la declaración de variable inicializada :

$$\begin{aligned} | & \text{"var" Identifier ":=" Expression} \end{aligned}$$

7. Lista de nuevos errores sintácticos detectados

- Token.Repeat.Do (dentro de SingleCommand): Va a salir un error en caso de que no reciba un WHILE o un UNTIL.
- Token.For (dentro de Single Command): Va a mandar un error en caso de que no reciba un WHILE, un DO, o un UNTIL

```
syntacticError(messageTemplate:"Expected 'while' or 'until' after the command", tokenQuoted:currentToken.spelling);
```

```
syntacticError(messageTemplate:"Expected 'do', 'while' or 'until' ", tokenQuoted:currentToken.spelling);
```

- V_name: Va a mandar un error en caso de que no encuentre “:=” después del nombre de una variable

```
syntacticError(messageTemplate:"Expected ':=' ", tokenQuoted:"");
```

8. Modelaje realizado para los árboles de sintaxis abstracta

El modelaje de los AST del programa se planteó tomando en cuenta la forma en la que se trabaja para Triangle la estructura de los árboles. Los AST poseen nodos que permiten identificar distintos componentes (literales, operadores e identificadores) y también muestran subárboles para las frases que puedan encontrarse en el programa fuente (declaraciones y/o expresiones). Considerando esta descripción, el plan de acción para la creación de los AST en este programa se basa en aceptar y crear cada uno de los nodos principales del programa y una vez todos los nodos se hayan creado se incorporan estos en la creación del árbol final del programa fuente en cuestión.

A mayor detalle, el programa es analizado sintácticamente por medio del parser, escaneando y reconociendo las palabras reservadas y componentes identificados para el lenguaje. Esta revisión permite reconocer la raíz del árbol a crear y los posibles casos esperados para la elaboración de los subárboles de los nodos asociados. En el paquete de AbstractSyntaxTree se encuentran las clases de las estructuras de los árboles asociados al proceso de análisis sintáctico, donde los constructores esperan, identifican y asignan los componentes individuales (expresiones, declaraciones o comandos) que componen el subárbol o árbol sintáctico principal.

9. Extensión realizada a los métodos que permiten visualizar los árboles de sintaxis abstracta

Retomando la idea de la sección, los nuevos AST implementados siguieron el orden de creación donde primero se completan los nodos y luego se finaliza la creación del árbol. En este caso, cada AST nuevo se agregó al paquete de AbstractSyntaxTrees, modificación de métodos ya incorporados en ciertas clases, así como la incorporación de métodos auxiliares en TableVisitor y TreeVisitor.

En primer lugar, debido a la adición, modificación y eliminación de reglas según lo indican las instrucciones del proyecto era necesario reflejar esos cambios en el programa. Por lo que los primeros cambios fueron agregar las reglas en los métodos

de la clase Parser, para que cuando se analice sintácticamente el programa fuente este pueda identificar y aceptar, no solo los cambios propios del lenguaje, sino que también las nuevas reglas y/o patrones que se esperan de los comandos y declaraciones.

Al permitir el reconocimiento de estas alteraciones el siguiente paso fue el agregar los nuevos AST al paquete de AbstractSyntaxTrees que se acomodan a las nuevas especificaciones de los subárboles y árboles esperados para el proyecto. Este último punto también significaba la incorporación de métodos en la clase de TableVisitor y TreeVisitor por cada uno de los árboles esperados. Los métodos agregados al TableVisitor se agregaron métodos para recorrer los nuevos árboles y permita visitar los nodos asociados al mismo. Por su parte en la clase de TreeVisitor se agregaron métodos que permiten recorrer el nuevo árbol, identificar su tipo (Nullary, Unary, Binary, Ternary o Quaternary) y definir el texto que deberá aparecer al momento en que este sea generado.

10. Extensión realizada a los métodos que permiten representar los árboles de sintaxis abstracta como texto en XML.

Como se comentó en la sección 4, para la creación de los archivos de salida esperados según las especificaciones del proyecto, las clases y métodos necesarios para el proceso se encuentran agrupados dentro de un nuevo paquete llamado ArchivosSalida. Para la elaboración del archivo XML se crearon dos clases dentro de este paquete, ArchivoXML y CreadorXML, que serán explicados a mayor detalle a continuación.

Para la creación del archivo se tiene la clase ArchivoXML, la cual directamente tiene como función generar el archivo con el nombre esperado según el programa fuente ingresado, escribir la línea inicial del archivo, generar un CreadorXML y permitir la visita del programa fuente. Por su parte, CreadorXML contiene la sobreescritura de los métodos implementados en Visitor que permiten reconocer y recorrer los subárboles y sus componentes. Este recorrido a su vez permite que por cada componente identificado este pueda ser escrito en el .xml esperado y dirigir el programa al siguiente componente para que también sea escrito.

La extensión de los métodos de visita para la creación del XML son los siguientes.

```
@Override  
public Object visitDoCommandAST(DoCommand aThis, Object o) {  
    addLine("<DoCommand>");  
    aThis.C.visit(this, null);  
    addLine("</DoCommand>");  
    return(null);  
}  
  
@Override  
public Object visitRepeatCommand(RepeatCommand aThis, Object o) {  
    addLine("<RepeatCommand>");  
    aThis.WhileC.visit(this, null);  
    addLine("</RepeatCommand>");  
    return(null);  
}  
  
@Override  
public Object visitRepeatUntilAST(RepeatUntilAST aThis, Object o) {  
    addLine("<RepeatUntil>");  
  
    if(aThis.I != null){  
        aThis.I.visit(this, null);  
    }  
    aThis.UntilC.visit(this, null);  
  
    addLine("</RepeatUntil>");  
    return(null);  
}
```

```
@Override
public Object visitUntilCommand(UntilCommand aThis, Object o) {
    addLine("<UntilCommand>");
    aThis.I.visit(this, null);
    aThis.C.visit(this, null);
    addLine("</UntilCommand>");
    return(null);
}

@Override
public Object visitDoWhileCommand(DoWhileCommand aThis, Object o) {
    addLine("<DoWhileCommand>");
    aThis.E.visit(this, null);
    addLine("</DoWhileCommand>");
    return(null);
}

@Override
public Object visitRepeatDoWhileCommand(RepeatDoWhileAST aThis, Object o) {
    addLine("<RepeatDoWhileCommand>");

    aThis.DoWhile.visit(this, null);
    aThis.C.visit(this, null);
    addLine("</RepeatDoWhileCommand>");
    return(null);}

@Override
public Object visitDoUntilCommand(DoUntilCommand aThis, Object o) {
    addLine("<DoUntilCommand>");
    aThis.E.visit(this, null);
    addLine("</DoUntilCommand>");
    return(null);
}
```

```
@Override
public Object visitRepeatDoUntilCommand(RepeatDoUntilAST aThis, Object o) {
    addLine("<RepeatDoUntilCommand>");
    aThis.DoUntil.visit(this, null);
    aThis.C.visit(this, null);
    addLine("</RepeatDoUntilCommand>");
    return(null);}

@Override
public Object visitForBecomesCommand(ForBecomesCommand aThis, Object o) {
    addLine("<ForBecomesCommand>");
    aThis.I.visit(this, null);
    aThis.E.visit(this, null);
    addLine("</ForBecomesCommand>");
    return(null);
}

@Override
public Object visitForBecomesAST(ForBecomesAST aThis, Object o) {
    addLine("<ForBecomes>");
    aThis.E.visit(this, null);
    aThis.ForBecomes.visit(this, null);
    aThis.DoC.visit(this, null);
    addLine("</ForBecomes>");
    return(null);
}
```

```
@Override
public Object visitRepeatForWhile(RepeatForWhile aThis, Object o) {
    addLine("<RepeatForWhile>");
    if(aThis.I != null){
        aThis.I.visit(this, null);
    }
    aThis.E.visit(this, null);
    aThis.ForBecomes.visit(this, null);
    aThis.whileC.visit(this, null);

    addLine("</RepeatForWhile>");

    return(null);
}

public Object visitRepeatForUntil(RepeatForUntil aThis, Object o) {
    addLine("<RepeatForUntil>");
    if(aThis.I != null){
        aThis.I.visit(this, null);
    }
    aThis.E.visit(this, null);
    aThis.ForBecomes.visit(this, null);
    aThis.UntilC.visit(this, null);
    addLine("</RepeatForUntil>");

    return(null);
}
```

```
    @Override
    public Object visitEmptyCommand(EmptyCommand ast, Object o) {
        addLine("<EmptyCommand>");
        addLine("</EmptyCommand>");
        return(null);
    }
    @Override
    public Object visitIfCommand(IfCommand ast, Object o) {
        addLine("<IfCommand>");
        ast.E.visit(this, null);
        ast.C1.visit(this, null);
        ast.C2.visit(this, null);
        addLine("</IfCommand>");
        return(null);
    }
    @Override
    public Object visitLetCommand(LetCommand ast, Object o) {
        addLine("<LetCommand>");
        ast.D.visit(this, null);
        ast.C.visit(this, null);
        addLine("</LetCommand>");

        return(null);
    }
    @Override
    public Object visitSequentialCommand(SequentialCommand ast, Object o) {
        addLine("<SequentialCommand>");
        ast.C1.visit(this, null);
        ast.C2.visit(this, null);
        addLine("</SequentialCommand>");

        return(null);
    }
}
```

```

@Override
public Object visitWhileCommand(WhileCommand ast, Object o) {
    addLine("<WhileCommand>");
    ast.E.visit(this, null);
    ast.C.visit(this, null);
    addLine("</WhileCommand>");
    return(null);
}
@Override
public Object visitTimesCommand(TimesCommand ast, Object o) {
    addLine("<TimesCommand>");
    ast.C.visit(this, null);
    addLine("</TimesCommand>");
    return(null);
}

```

```

public Object visitDotDCommandLiteralAST(DotDCommandLiteral ast, Object o){
    addLine("<DotDCommandLiteral>");
    ast.CLC.visit(this, null);
    addLine("</DotDCommandLiteral>");
    return(null);
}
@Override
public Object visitDotDCommand2(DotDCommand2 ast, Object o){
    addLine("<DotDCommand>");
    ast.CLCT.visit(this, null);
    addLine("</DotDCommand>");
    return(null);
}
@Override
public Object visitBarCommandCaseRange(BarCommandCaseRange ast, Object obj){
    addLine("<BarCommandCaseRange>");
    ast.CRC.visit(this, null);
    addLine("</BarCommandCaseRange>");
    return(null);
}

```

```

@Override
public Object visitThenCommandAST(ThenCommand aThis, Object o) { //
    addLine("<ThenCommand>");
    aThis.c.visit(this, null);
    addLine("</ThenCommand>");
    return(null);
}

@Override
public Object visitVarDeclarationBecomes(VarDeclarationBecomes aThis, Object o) { //
    addLine("<VarDeclarationBecomes>");
    aThis.E.visit(this, null);
    aThis.I.visit(this, null);
    addLine("</VarDeclarationBecomes>");
    return(null);
}

@Override
public Object visitArrayExpression(ArrayExpression ast, Object o) {
    addLine("<ArrayExpresssion>");
    ast.AA.visit(this, null);
    addLine("</ArrayExpresssion>");
    return(null);
}

@Override
public Object visitBinaryExpression(BinaryExpression ast, Object o) {
    addLine("<BinaryExpresssion>");
    ast.E1.visit(this, null);
    ast.E2.visit(this, null);
    ast.O.visit(this, null);
    addLine("</BinaryExpresssion>");
    return(null);
}

@Override
public Object visitPrivateDeclaration(PrivateDeclaration ast, Object o) {
    addLine("<PrivateDeclaration>");
    ast.D1.visit(this, null);
    ast.D2.visit(this, null);
    addLine("</PrivateDeclaration>");
    return(null);
}

```

```
@Override
public Object visitConstFormalParameter(ConstFormalParameter ast, Object o) {
    addLine("<ConstFormalParameter>");
    ast.I.visit(this, null);
    ast.T.visit(this, null);
    addLine("</ConstFormalParameter>");
    return(null);
}
@Override
public Object visitFuncFormalParameter(FuncFormalParameter ast, Object o) {
    addLine("<FuncFormalParameter>");
    ast.FPS.visit(this, null);
    ast.T.visit(this, null);
    addLine("</FuncFormalParameter>");
    return(null);
}
@Override
public Object visitProcFormalParameter(ProcFormalParameter ast, Object o) {
    addLine("<ProcFormalParameter>");
    ast.FPS.visit(this, null);
    addLine("</ProcFormalParameter>");
    return(null);
}
@Override
public Object visitVarFormalParameter(VarFormalParameter ast, Object o) {
    addLine("<ProcFormalParameter>");
    ast.T.visit(this, null);
    addLine("</ProcFormalParameter>");
    return(null);
}
```

```
public Object visitCharacterLiteral(CharacterLiteral ast, Object o) {
    addLine("<CharacterLiteral>");
    addLine("</CharacterLiteral>");
    return(null);
}
@Override
public Object visitIdentifier(Identifier ast, Object o) {
    addLine("<Identifier>");
    addLine("</Identifier>");
    return(null);
}
@Override
public Object visitIntegerLiteral(IntegerLiteral ast, Object o) {
    addLine("<IntegerLiteral>");
    addLine("</IntegerLiteral>");
    return(null);
}
@Override
public Object visitOperator(Operator ast, Object o) {
    if("<".equals(ast.spelling))
        addLine("<Operator value='&lt;'>");
    else if(">".equals(ast.spelling))
        addLine("<Operator value='&gt;'>");
    else
        addLine("<Operator value= '" + ast.spelling + "'>");
    if(ast.decl != null){
        ast.decl.visit(this, null);
    }
    addLine("</Operator>");
    return(null);
}
```

```
@Override
public Object visitCallExpression(CallExpression ast, Object o) {
    addLine("<CallExpression>");
    ast.I.visit(this, null);
    ast.APS.visit(this, null);
    addLine("</CallExpression>");
    return(null);
}
@Override
public Object visitCharacterExpression(CharacterExpression ast, Object o) {
    addLine("<CharacterExpression>");
    ast.CL.visit(this, null);
    addLine("</CharacterExpression>");
    return(null);
}
@Override
public Object visitEmptyExpression(EmptyExpression ast, Object o) {
    addLine("<EmptyExpression>");
    addLine("</EmptyExpression>");
    return(null);
}
@Override
public Object visitIfExpression(IfExpression ast, Object o) {
    addLine("<IfExpression>");
    ast.E1.visit(this, null);
    ast.E2.visit(this, null);
    ast.E3.visit(this, null);
    addLine("</IfExpression>");
    return(null);
}
.
```

```
@Override
public Object visitIntegerExpression(IntegerExpression ast, Object o) {
    addLine("<IntegerExpression>");
    addLine("</IntegerExpression>");
    return(null);
}
@Override
public Object visitLetExpression(LetExpression ast, Object o) {
    addLine("<LetExpression>");
    ast.D.visit(this, null);
    ast.E.visit(this, null);
    addLine("</LetExpression>");
    return(null);
}
@Override
public Object visitRecordExpression(RecordExpression ast, Object o) {
    addLine("<RecordExpression>");
    ast.RA.visit(this, null);
    addLine("</RecordExpression>");
    return(null);
}
@Override
public Object visitUnaryExpression(UnaryExpression ast, Object o) {
    addLine("<UnaryExpression>");
    ast.E.visit(this, null);
    ast.O.visit(this, null);
    addLine("</UnaryExpression>");
    return(null);
}
```

```
@Override
public Object visitDotDCommandAST(DotDCommand aThis, Object o) {
    addLine("<DotDCommand>");
    aThis.CLC.visit(this, null);
    addLine("</DotDCommand>");
    return(null);
}

@Override
public Object visitRepeatTimesCommand(RepeatTimesCommand aThis, Object o) {
    addLine("<RepeatTimesCommand>");
    aThis.TimesC.visit(this, null);
    addLine("</RepeatTimesCommand>");
    return(null);
}

@Override
public Object visitAssignCommand(AssignCommand ast, Object o) {
    addLine("<AssingCommand>");
    ast.V.visit(this, null);
    ast.E.visit(this, null);
    addLine("</AssingCommand>");
    return(null);
}

public Object visitCallCommand(CallCommand ast, Object o) {
    addLine("<CallCommand>");
    ast.I.visit(this, null);
    ast.APS.visit(this, null);
    addLine("</CallCommand>");
    return(null);
}
```

11. Plan de pruebas para validar el compilador

Se presentará una prueba con una sintaxis correcta y otra prueba con un error, de modo que se aprecie el error esperado por el compilador. Las pruebas por realizar son de las nuevas modificaciones realizadas. En la carpeta Pruebas se podrán encontrar más ejemplos para probar nuestro compilador.

Prueba del Comando Skip:

Objetivo del caso de prueba:

El objetivo de esta prueba es ver el funcionamiento de dicho comando. En este caso las pruebas responden al funcionamiento esperado para el comando skip. Primero vamos a ver el funcionamiento del comando correctamente y después otra prueba con un error.

Diseño del caso de prueba

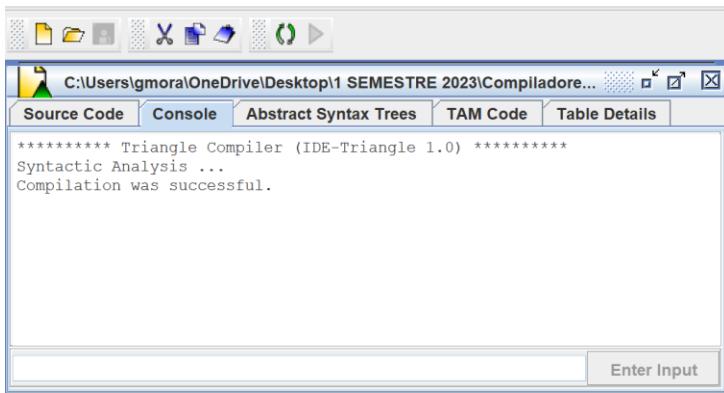
```
skip ;
puteol() ;
skip;
puteol();
skip|
! OK
```

Esta prueba funciona correctamente

Resultados esperados

El resultado esperado es que el programa compile de manera satisfactoria, construya el árbol, despliegue el XML y el HTML de manera correcta.

Resultados observados



XML

```

▼<Program>
  ▼<SequentialCommand>
    ▼<SequentialCommand>
      ▼<SequentialCommand>
        ▼<SequentialCommand>
          <EmptyCommand> </EmptyCommand>
        ▼<CallCommand>
          <Identifier> </Identifier>
          <EmptyActualParameterSequence> </EmptyActualParameterSequence>
        </CallCommand>
      </SequentialCommand>
      <EmptyCommand> </EmptyCommand>
    </SequentialCommand>
  ▼<CallCommand>
    <Identifier> </Identifier>
    <EmptyActualParameterSequence> </EmptyActualParameterSequence>
  </CallCommand>
</SequentialCommand>
<EmptyCommand> </EmptyCommand>
</SequentialCommand>
</Program>

```

HTML

```
skip ; puteol ( ) ; skip ; puteol ( ) ; skip
```

Discusión y análisis de los resultados obtenidos.

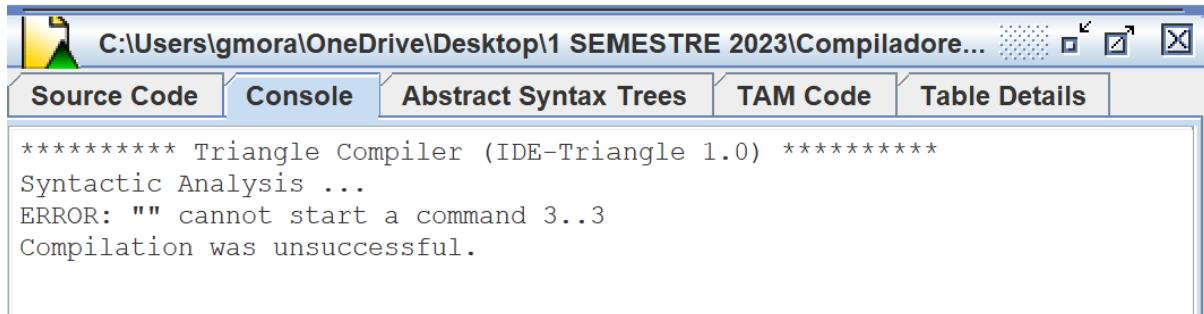
Como esperábamos a partir de estos resultados podemos ver que el comando skip funciona de manera correcta en todas sus respectivas partes.

Prueba para evidenciar la capacidad del compilador para detectar errores

Diseño de la prueba negativa:

```
puteol();  
! Error: Después del ";" faltaría un skip. ";" no es un terminador|
```

Resultado:



The screenshot shows the Triangle Compiler IDE interface. The title bar reads "C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladores\triangle\triangle.exe". The tabs at the top are "Source Code" (which is selected), "Console", "Abstract Syntax Trees", "TAM Code", and "Table Details". The console window displays the following text:

```
***** Triangle Compiler (IDE-Triangle 1.0) *****  
Syntactic Analysis ...  
ERROR: "" cannot start a command 3..3  
Compilation was unsuccessful.
```

Prueba comando let y de compound declaration

Objetivo del caso de prueba

El objetivo de esta prueba es probar el comando let junto una declaración compuesta para ver que estas funcionen correctamente.

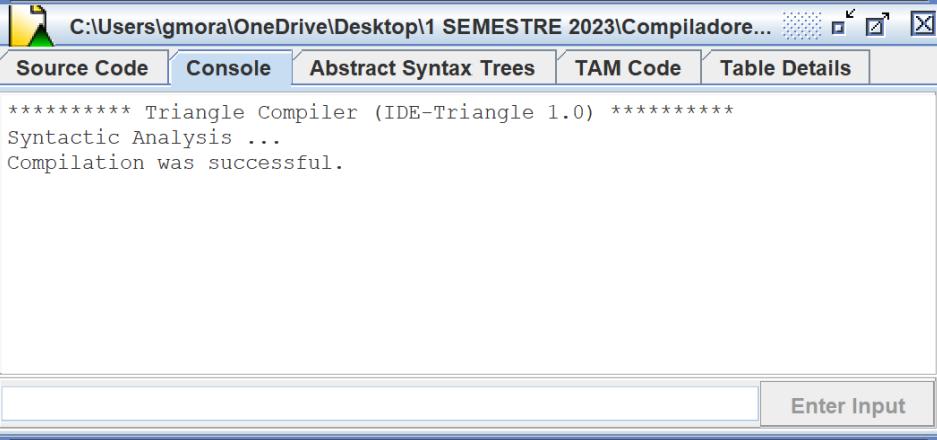
Diseño del caso de prueba

```
let
    var i : Char;
    var j : Integer;
    const k ~ 8;
    type dia ~ Integer
in
    skip
end
```

Resultados esperados

Los resultados esperados de esta prueba es que estos funcionen adecuadamente, que compile, que se haga el árbol, que se creen el HTML y el XML.

Resultados observados



The screenshot shows the Triangle Compiler IDE interface. The title bar indicates the path: C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladores... . The tabs at the top are Source Code, Console, Abstract Syntax Trees, TAM Code, and Table Details. The Abstract Syntax Trees tab is active, displaying a hierarchical tree structure of the program's declarations. The tree starts with a Program node, which contains a Let Command node. This Let Command node has several children: Sequential Declaration, Variable Declaration (with children i and Char), Variable Declaration (with children j and Integer), Constant Declaration (with children k and Integer Expression 8), Type Declaration (with children dia and Integer), and an Empty Command node.

HTML

```
let  var  i  :  Char  ;  var  j  :  Integer  ;  const  k  ~  8  ;  type  dia  ~  Integer  in  skip  end
```

XML

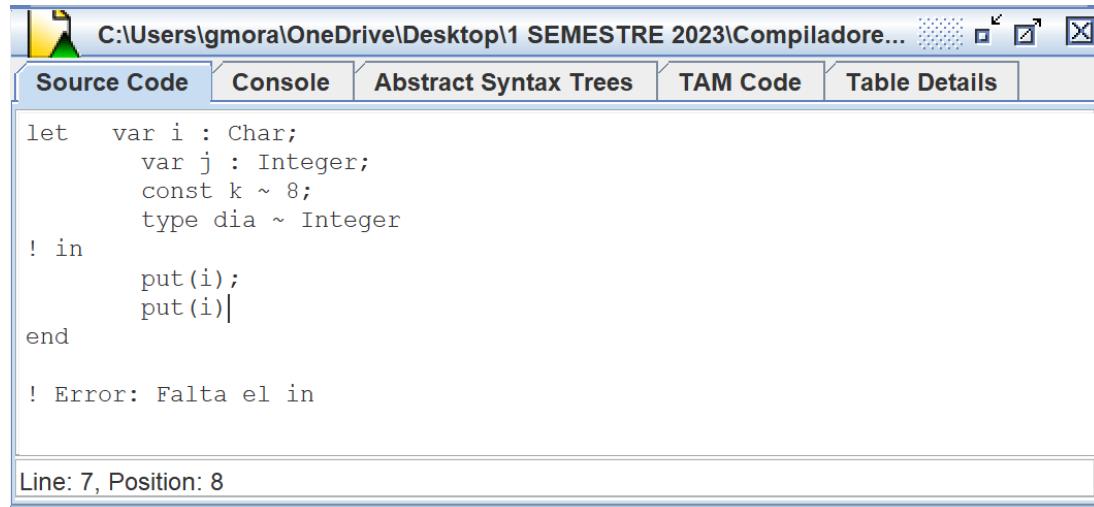
```

▼<Program>
  ▼<LetCommand>
    ▼<SequentialDeclaration>
      ▼<SequentialDeclaration>
        ▼<SequentialDeclaration>
          <VarDeclaration> </VarDeclaration>
          <SimpleTypeDenoter> </SimpleTypeDenoter>
          <Identifier> </Identifier>
          <VarDeclaration> </VarDeclaration>
          <SimpleTypeDenoter> </SimpleTypeDenoter>
          <Identifier> </Identifier>
        </SequentialDeclaration>
      ▼<ConstDeclaration>
        <IntegerExpression> </IntegerExpression>
        <Identifier> </Identifier>
      </ConstDeclaration>
    </SequentialDeclaration>
  ▼<TypeDeclaration>
    <SimpleTypeDenoter> </SimpleTypeDenoter>
    <Identifier> </Identifier>
  </TypeDeclaration>
</SequentialDeclaration>
<EmptyCommand> </EmptyCommand>
</LetCommand>
</Program>

```

Prueba para evidenciar la capacidad del compilador para detectar errores

Diseño de la Prueba



The screenshot shows a software interface for compiler development. The title bar indicates the path: C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladore... . Below the title bar is a toolbar with icons for file operations. The main window contains five tabs: Source Code (selected), Console, Abstract Syntax Trees, TAM Code, and Table Details. The Source Code tab displays the following code:

```

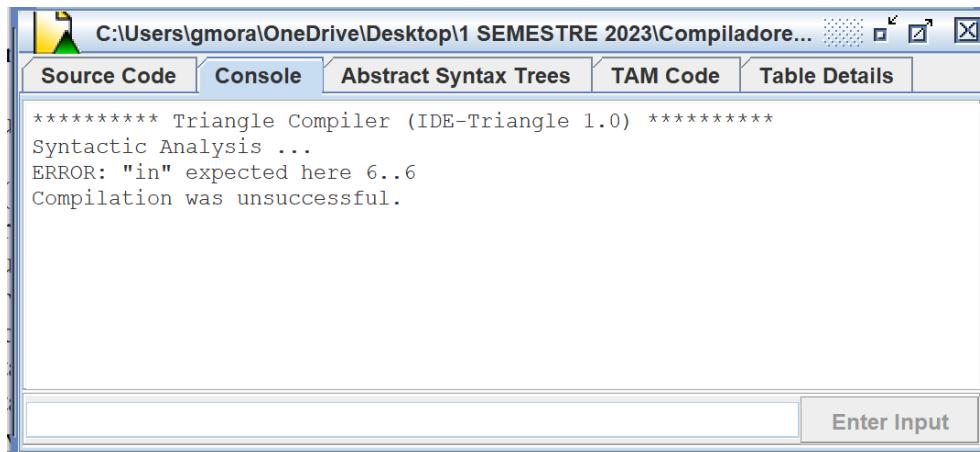
let  var i : Char;
     var j : Integer;
     const k ~ 8;
     type dia ~ Integer
! in
     put(i);
     put(i)
end

! Error: Falta el in

```

The code includes several syntax errors, such as undeclared identifiers and type declarations. At the bottom of the code area, the message "Error: Falta el in" is displayed, indicating a missing "in" keyword. The status bar at the bottom left shows "Line: 7, Position: 8".

Resultado



The screenshot shows the Triangle Compiler IDE interface. The title bar reads "C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladore...". The tabs at the top are "Source Code", "Console", "Abstract Syntax Trees", "TAM Code", and "Table Details". The "Console" tab is selected. The console window displays the following text:

```
***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
ERROR: "in" expected here 6..6
Compilation was unsuccessful.
```

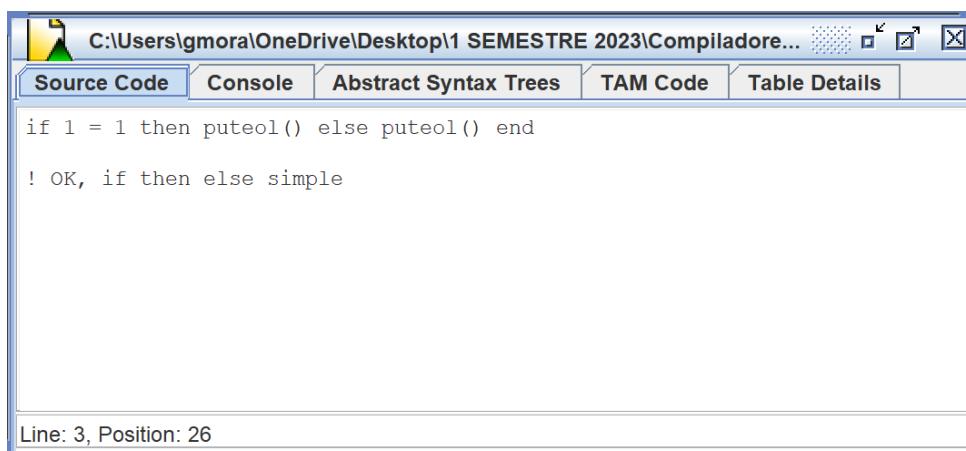
At the bottom right of the console window is a button labeled "Enter Input".

Pruebas comando If

Objetivo del caso de prueba

El objetivo de esta prueba es ver el funcionamiento de dicho comando. En este caso las pruebas responden al funcionamiento esperado para el comando if. Primero vamos a ver el funcionamiento del comando correctamente y después otra prueba con un error.

Diseño del caso de prueba



The screenshot shows the Triangle Compiler IDE interface. The title bar reads "C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladore...". The tabs at the top are "Source Code", "Console", "Abstract Syntax Trees", "TAM Code", and "Table Details". The "Source Code" tab is selected. The code editor window contains the following code:

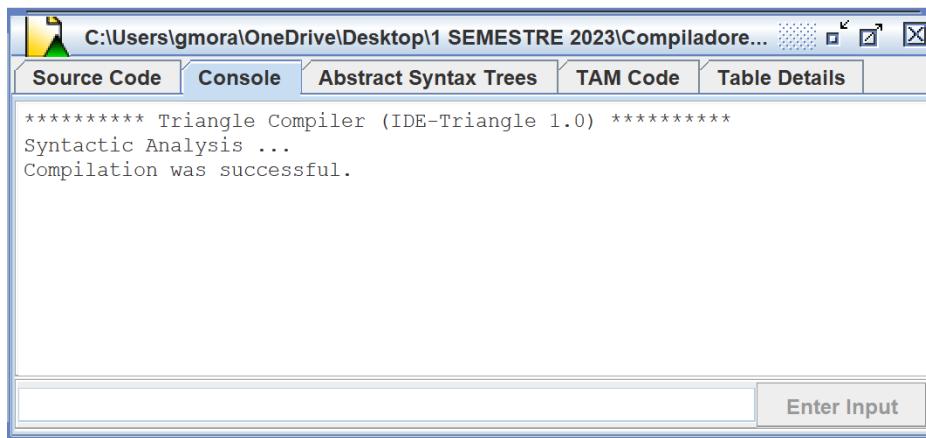
```
if 1 = 1 then puteol() else puteol() end
```

Below the code editor, the status bar shows "Line: 3, Position: 26".

Resultados esperados

El resultado esperado es que el programa compile de manera satisfactoria, construya el árbol, despliegue el xml y el html de manera correcta.

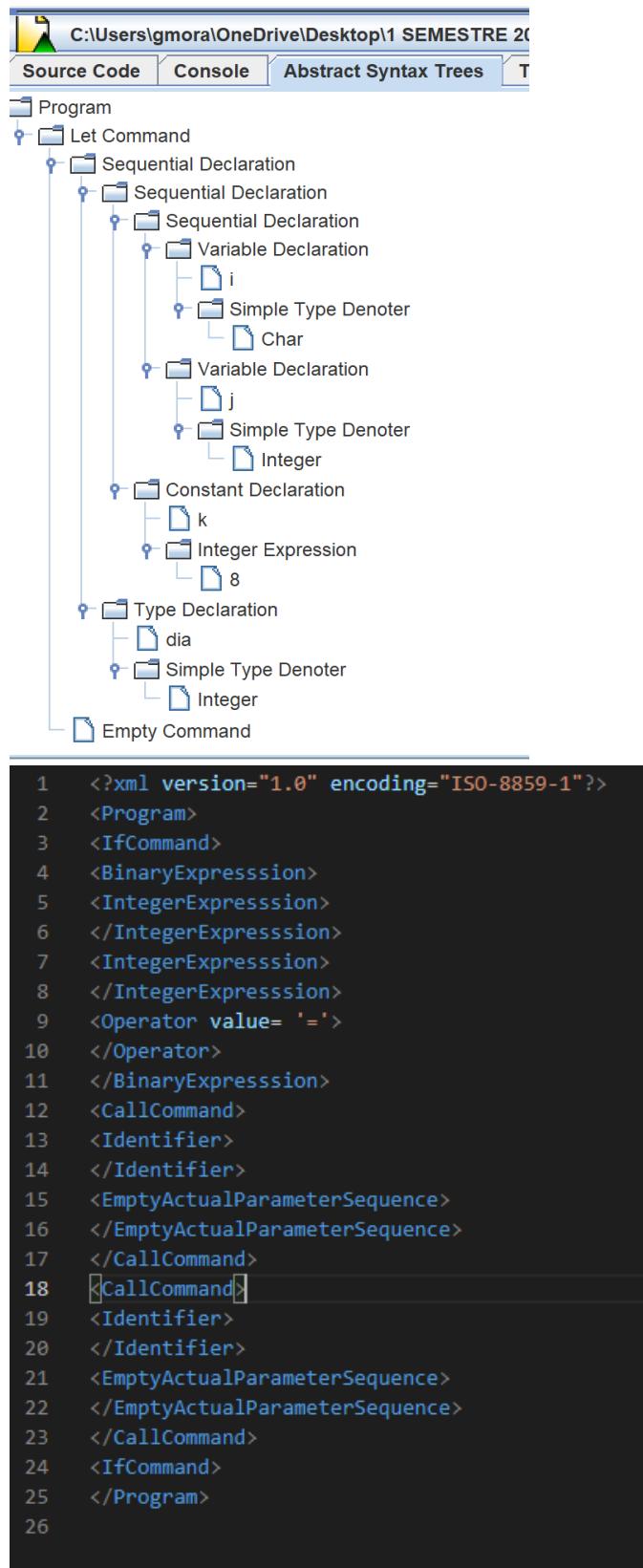
Resultados observados



The screenshot shows a window titled "C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladore...". The window has tabs at the top: "Source Code", "Console", "Abstract Syntax Trees", "TAM Code", and "Table Details". The "Console" tab is selected, displaying the following text:

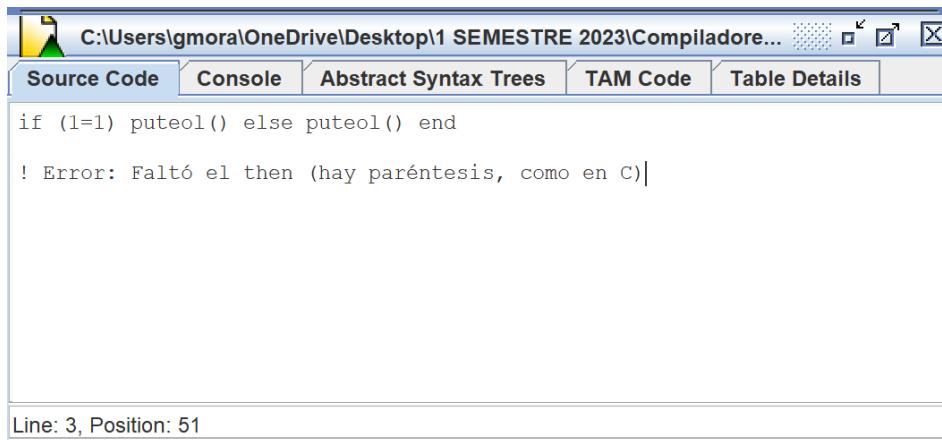
```
***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
Compilation was successful.
```

At the bottom of the window, there is an "Enter Input" button.



Prueba para evidenciar la capacidad del compilador para detectar errores

Diseño de la Prueba Negativa

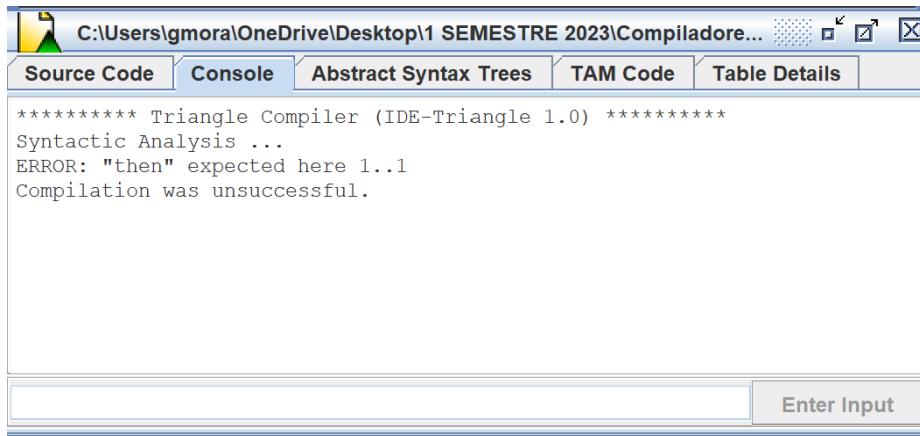


The screenshot shows the Triangle Compiler IDE interface. The title bar reads "C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladore...". The tabs at the top are "Source Code", "Console", "Abstract Syntax Trees", "TAM Code", and "Table Details". The "Source Code" tab is active, displaying the following code:

```
if (l=1) puteol() else puteol() end
! Error: Faltó el then (hay paréntesis, como en C)|
```

Below the code editor, a status bar indicates "Line: 3, Position: 51".

Resultado



The screenshot shows the Triangle Compiler IDE interface with the "Console" tab active. The output window displays the following text:

```
***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
ERROR: "then" expected here 1..1
Compilation was unsuccessful.
```

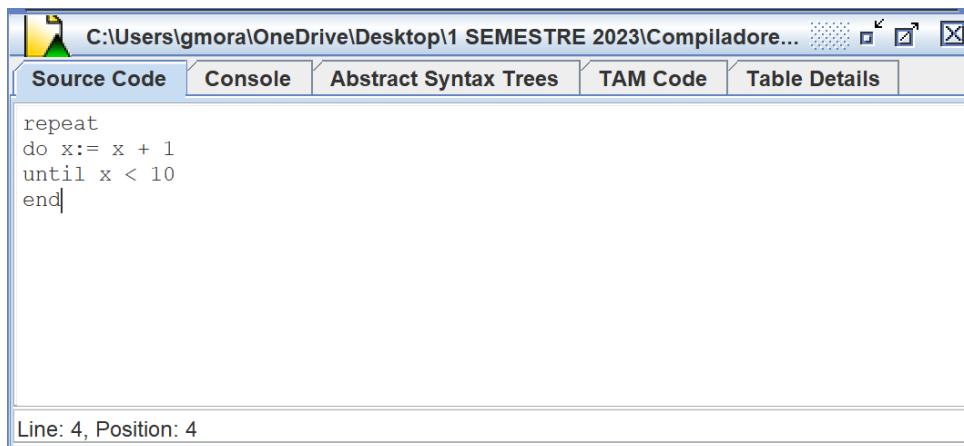
At the bottom right of the console window, there is an "Enter Input" button.

Pruebas comando Repeat do until end

Objetivo del caso de prueba

El objetivo de esta prueba es ver el funcionamiento de dicho comando. En este caso las pruebas responden al funcionamiento esperado para el comando repeat seguido de la palabra reservada until. Primero vamos a ver el funcionamiento del comando correctamente y después otra prueba con un error.

Diseño del caso de prueba



The screenshot shows the Triangle Compiler IDE interface. The title bar reads "C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladore...". The tabs at the top are "Source Code", "Console", "Abstract Syntax Trees", "TAM Code", and "Table Details". The "Source Code" tab is selected, displaying the following code:

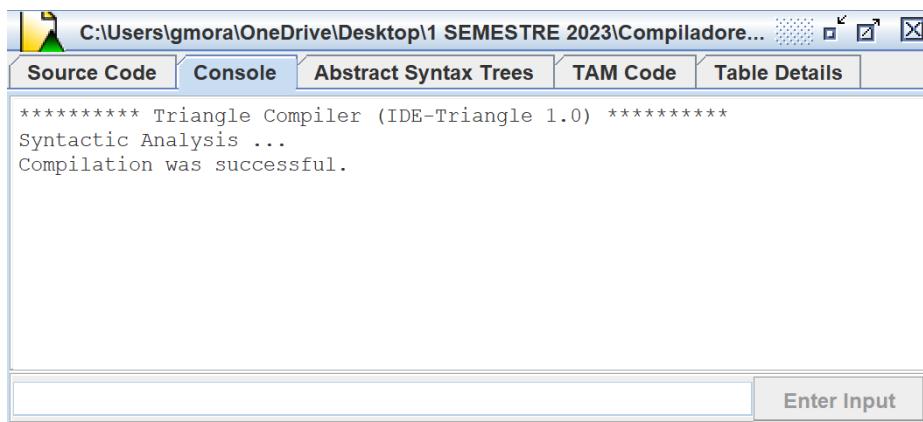
```
repeat
do x:= x + 1
until x < 10
end
```

Below the code editor, it says "Line: 4, Position: 4".

Resultados esperados

El resultado esperado es que el programa compile de manera satisfactoria, construya el árbol, despliegue el xml y el html de manera correcta.

Resultados observados



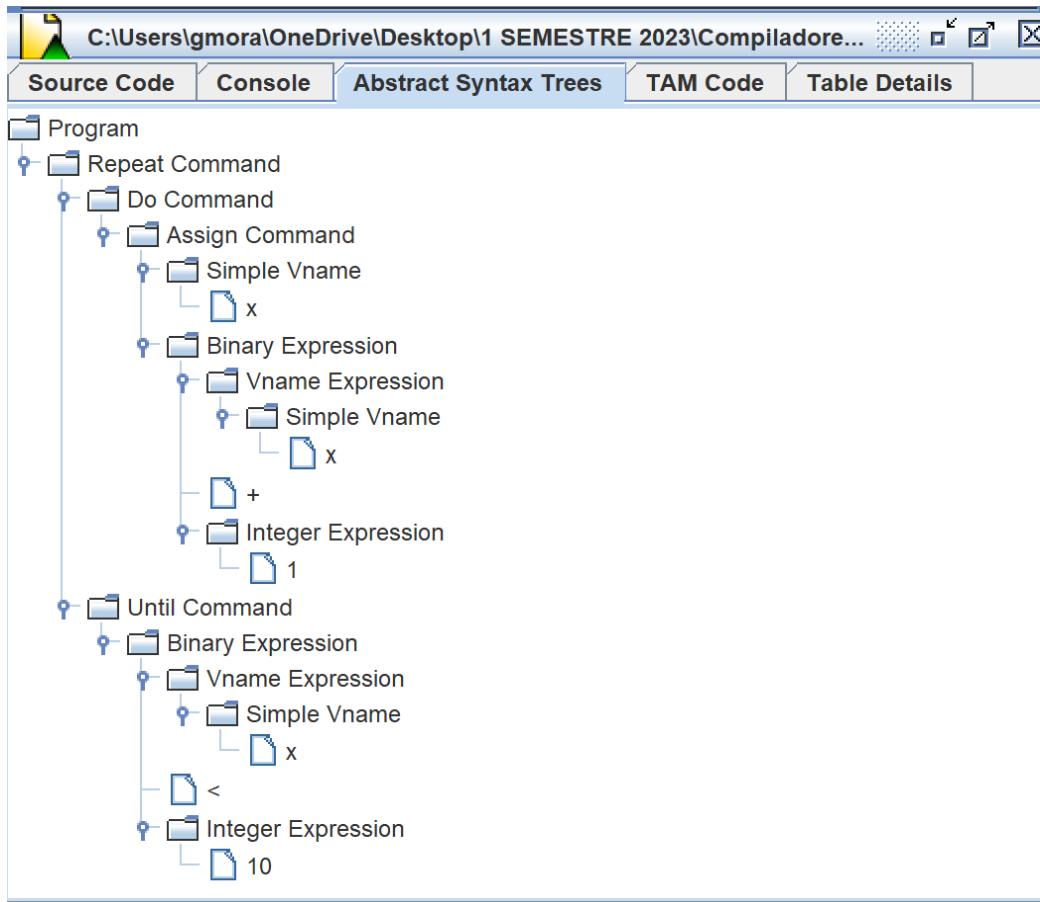
The screenshot shows the Triangle Compiler IDE interface. The title bar reads "C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladore...". The tabs at the top are "Source Code", "Console", "Abstract Syntax Trees", "TAM Code", and "Table Details". The "Console" tab is selected, displaying the following output:

```
***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
Compilation was successful.
```

At the bottom right of the console window is a button labeled "Enter Input".

XML

```
▼<Program>
  ▼<RepeatDoUntilCommand>
    ▼<DoUntilCommand>
      ▼<BinaryExpression>
        ▼<VnameExpression>
          ▼<SimpleVname>
            <Identifier> </Identifier>
            </SimpleVname>
          </VnameExpression>
          <IntegerExpression> </IntegerExpression>
          <Operator value="<"> </Operator>
        </BinaryExpression>
      </DoUntilCommand>
    ▼<DoCommand>
      ▼<AssingCommand>
        ▼<SimpleVname>
          <Identifier> </Identifier>
          </SimpleVname>
        ▼<BinaryExpression>
          ▼<VnameExpression>
            ▼<SimpleVname>
              <Identifier> </Identifier>
              </SimpleVname>
            </VnameExpression>
            <IntegerExpression> </IntegerExpression>
            <Operator value="+"> </Operator>
          </BinaryExpression>
        </AssingCommand>
      </DoCommand>
    </RepeatDoUntilCommand>
</Program>
```

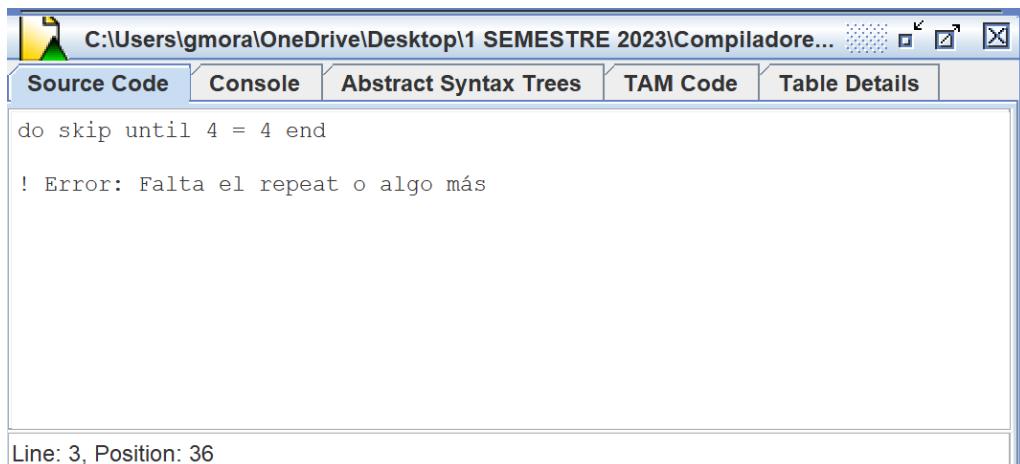


HTML

```
repeat    do      x      :=      x      +      1      until      x      <      10      end
```

Prueba para evidenciar la capacidad del compilador para detectar errores

Diseño de la Prueba Negativa

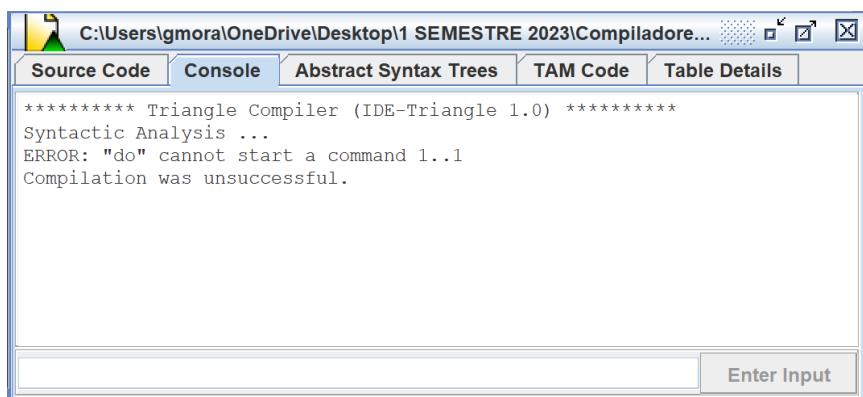


The screenshot shows a window titled 'C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladore...'. The 'Source Code' tab is active, displaying the following code:

```
do skip until 4 = 4 end
! Error: Falta el repeat o algo más
```

At the bottom of the window, it says 'Line: 3, Position: 36'.

Resultado



The screenshot shows a window titled 'C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladore...'. The 'Console' tab is active, displaying the following output:

```
***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
ERROR: "do" cannot start a command 1..1
Compilation was unsuccessful.
```

At the bottom right of the window, there is an 'Enter Input' button.

Pruebas comando Repeat do While End

Objetivo del caso de prueba

El objetivo de esta prueba es ver el funcionamiento de dicho comando. En este caso las pruebas responden al funcionamiento esperado para el comando repeat seguido de las palabras reservadas do y while. Primero vamos a ver el funcionamiento del comando correctamente y después otra prueba con un error.

Diseño del caso de prueba

repeat do skip while 3=3 end
! OK

Line: 3, Position: 5

Resultados esperados

El resultado esperado es que el programa compile de manera satisfactoria, construya el árbol, despliegue el XML y el HTML de manera correcta.

Resultados observados

Program

- Repeat Command
 - Do Command
 - Empty Command
 - While Command
 - Binary Expression
 - Integer Expression
 - 3
 - =
 - Integer Expression
 - 3

***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
Compilation was successful.

Enter Input

XML

```

▼ <Program>
  ▼ <RepeatDoWhileCommand>
    ▼ <DoWhileCommand>
      ▼ <BinaryExpression>
        <IntegerExpression> </IntegerExpression>
        <IntegerExpression> </IntegerExpression>
        <Operator value="="> </Operator>
      </BinaryExpression>
    </DoWhileCommand>
    ▼ <DoCommand>
      <EmptyCommand> </EmptyCommand>
    </DoCommand>
  </RepeatDoWhileCommand>
</Program>

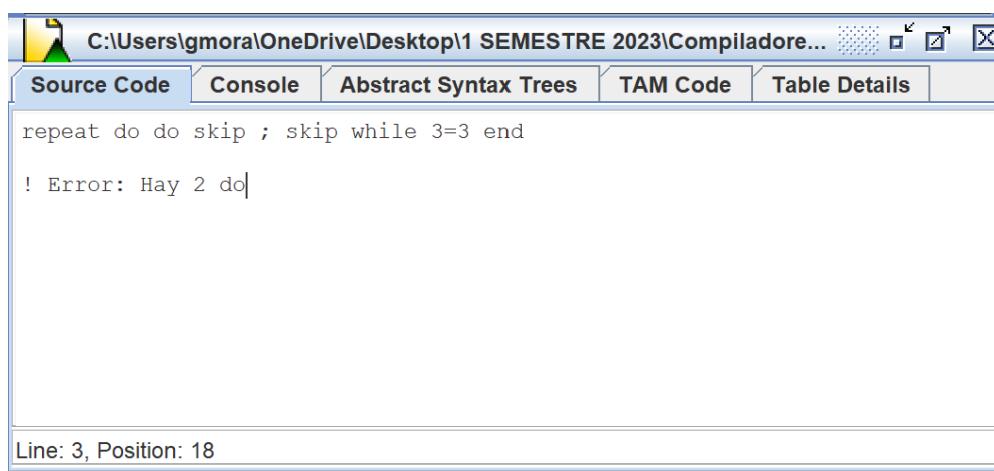
```

HTML

```
repeat do skip while 3 = 3 end
```

Prueba para evidenciar la capacidad del compilador para detectar errores

Diseño de la Prueba Negativa

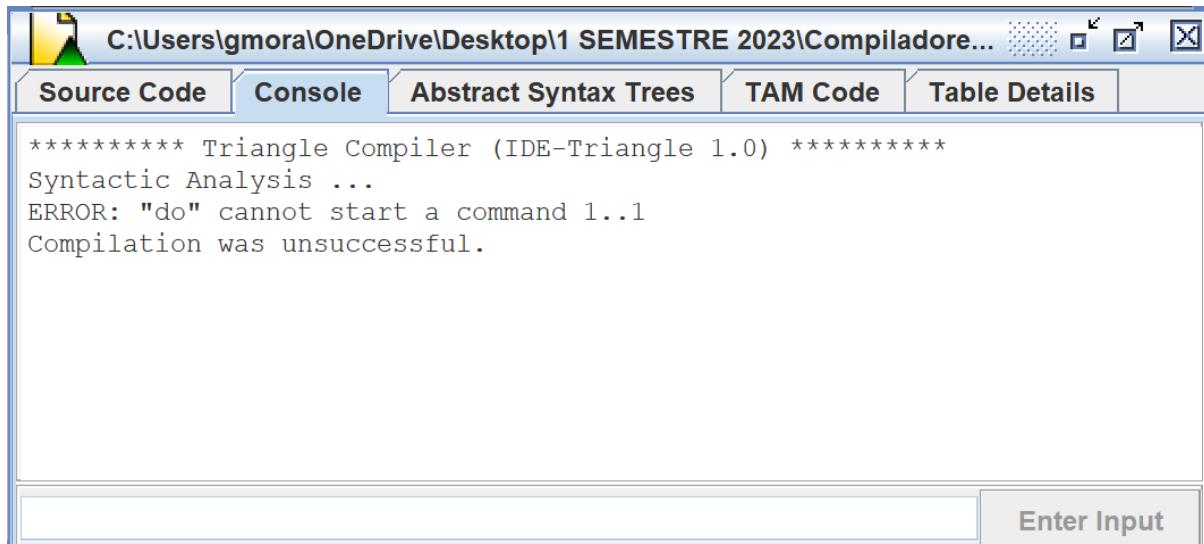


The screenshot shows a software interface with a title bar 'C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladore...' and several tabs: 'Source Code' (selected), 'Console', 'Abstract Syntax Trees', 'TAM Code', and 'Table Details'. The 'Console' tab displays the following text:

```
repeat do do skip ; skip while 3=3 end
! Error: Hay 2 do|
```

At the bottom of the console window, it says 'Line: 3, Position: 18'.

Resultado



The screenshot shows the Triangle Compiler IDE interface. The title bar reads "C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladore...". The tabs at the top are "Source Code", "Console", "Abstract Syntax Trees", "TAM Code", and "Table Details". The "Console" tab is selected. The console window displays the following text:

```
***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
ERROR: "do" cannot start a command 1..1
Compilation was unsuccessful.
```

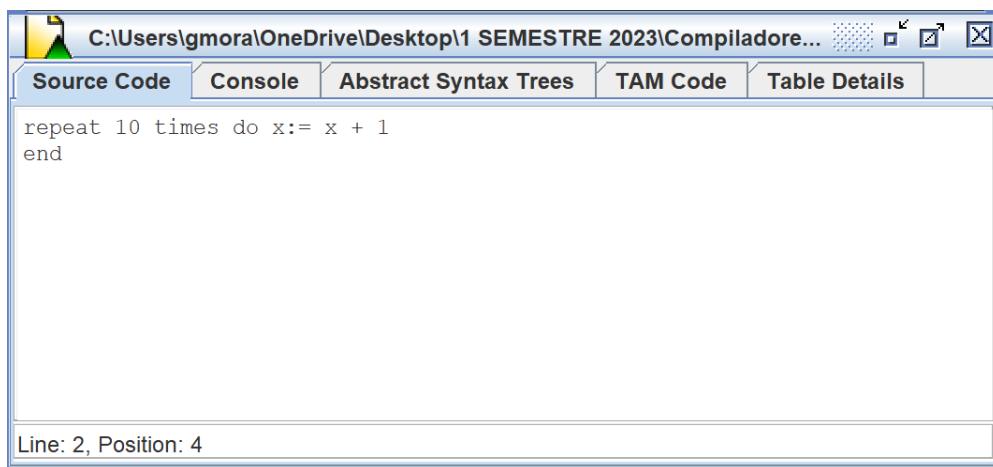
At the bottom right of the console window is a button labeled "Enter Input".

Pruebas comando Repeat times

Objetivo del caso de prueba

Esta prueba busca ver el funcionamiento de dicho comando. En este caso las pruebas responden al funcionamiento esperado para el comando repeat seguido de la palabra reservada times. Primero vamos a ver el funcionamiento del comando correctamente y después otra prueba con un error.

Diseño del caso de prueba

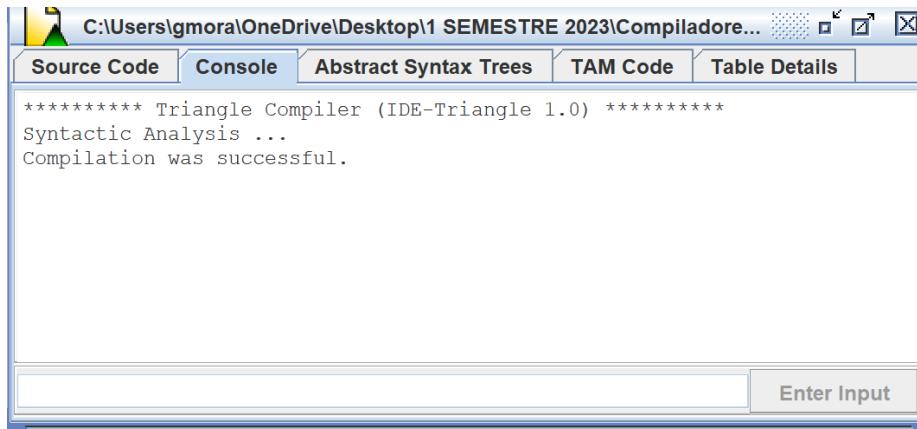


The screenshot shows the Triangle Compiler IDE interface. The title bar reads "C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladore...". The tabs at the top are "Source Code", "Console", "Abstract Syntax Trees", "TAM Code", and "Table Details". The "Source Code" tab is selected. The source code window contains the following code:

```
repeat 10 times do x:= x + 1
end
```

At the bottom left of the source code window is a status bar with the text "Line: 2, Position: 4".

Resultados esperados

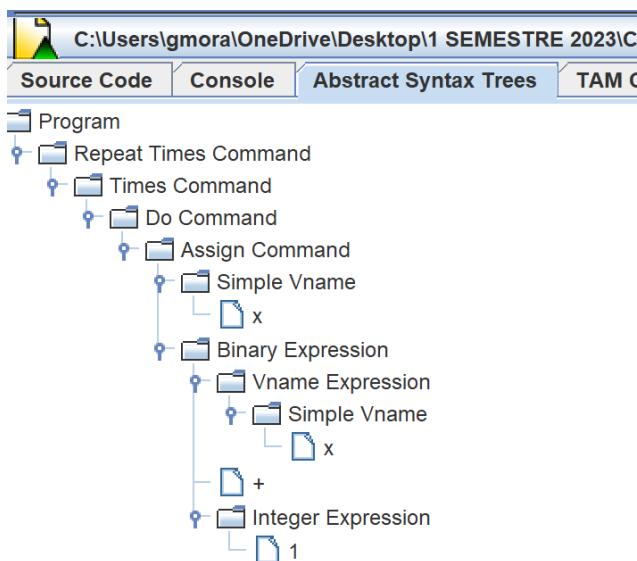


XML

```

▼ <Program>
  ▼ <RepeatTimesCommand>
    ▼ <TimesCommand>
      ▼ <DoCommand>
        ▼ <AssingCommand>
          ▼ <SimpleVname>
            <Identifier> </Identifier>
          </SimpleVname>
          ▼ <BinaryExpression>
            ▼ <VnameExpression>
              ▼ <SimpleVname>
                <Identifier> </Identifier>
              </SimpleVname>
              </VnameExpression>
              <IntegerExpression> </IntegerExpression>
              <Operator value="+"> </Operator>
            </BinaryExpression>
          </AssingCommand>
        </DoCommand>
      </TimesCommand>
    </RepeatTimesCommand>
  </Program>

```

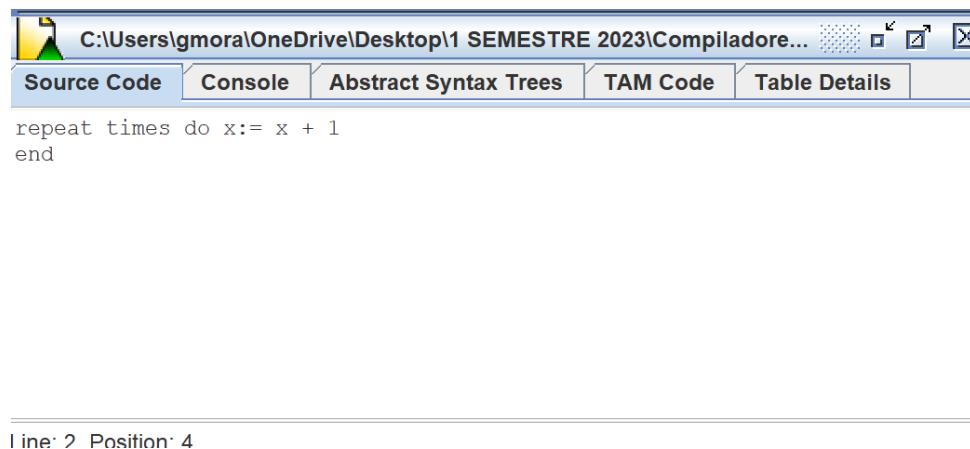


HTML

```
repeat 10 times do x := x + 1 end
```

Prueba para evidenciar la capacidad del compilador para detectar errores

Diseño de la Prueba Negativa

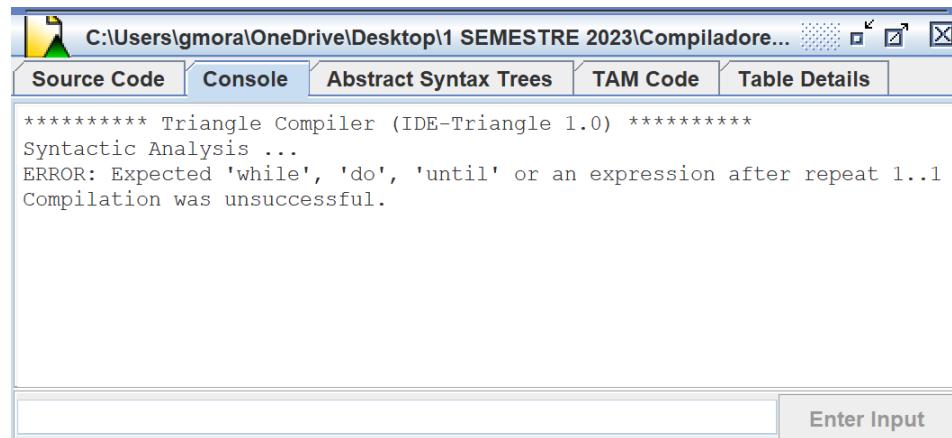


The screenshot shows the Triangle Compiler IDE interface. The title bar reads "C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladore...". The tabs at the top are "Source Code", "Console", "Abstract Syntax Trees", "TAM Code", and "Table Details". The "Source Code" tab is active, displaying the following code:

```
repeat times do x:= x + 1
end.
```

Below the code editor, a status bar indicates "Line: 2 Position: 4".

Resultado



The screenshot shows the Triangle Compiler IDE interface with the "Console" tab active. The output window displays the following error message:

```
***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
ERROR: Expected 'while', 'do', 'until' or an expression after repeat 1..1
Compilation was unsuccessful.
```

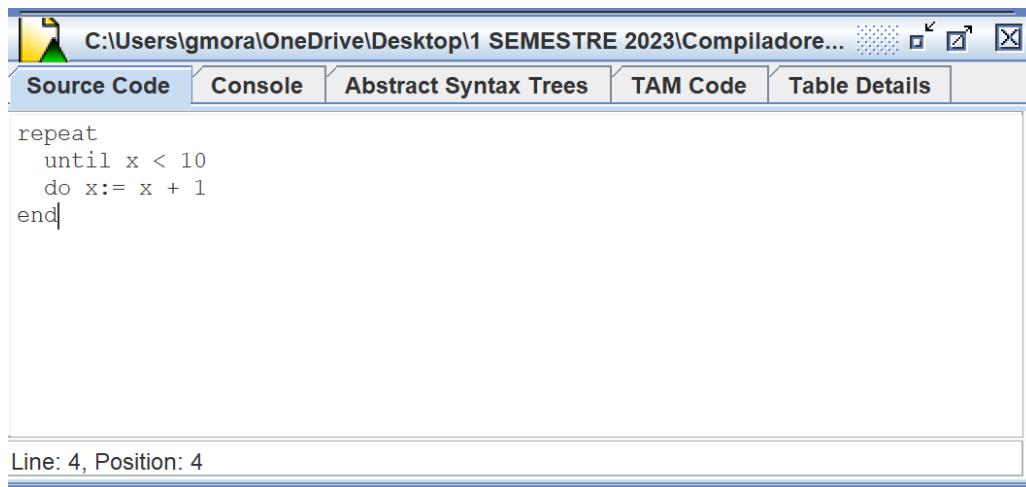
At the bottom of the console window, there is an "Enter Input" text input field.

Pruebas comando Repeat until do end

Objetivo del caso de prueba

El objetivo de esta prueba es ver el funcionamiento de dicho comando. En este caso las pruebas responden al funcionamiento esperado para el comando repeat seguido de la palabra reservada until y do. Primero vamos a ver el funcionamiento del comando correctamente y después otra prueba con un error.

Diseño del caso de prueba



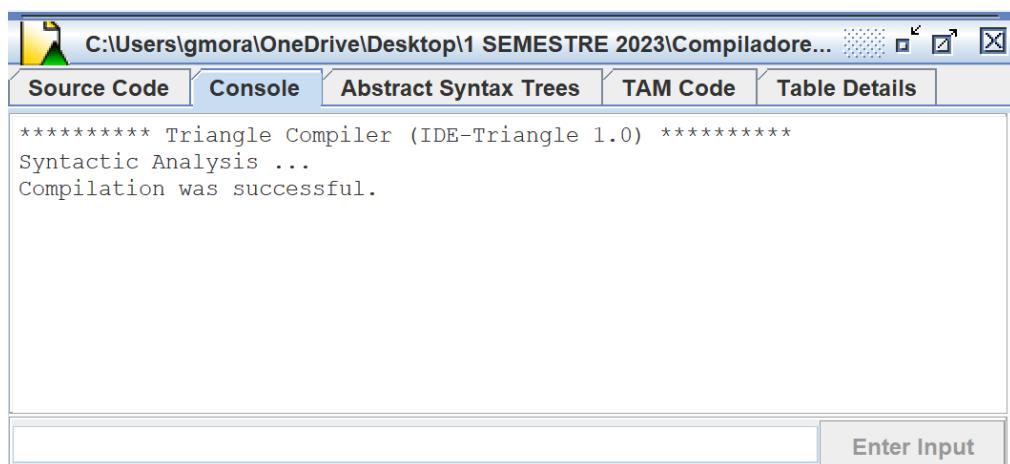
```
repeat
  until x < 10
  do x := x + 1
end|
```

Line: 4, Position: 4

Resultados esperados

El resultado esperado es que el programa compile de manera satisfactoria, construya el árbol, despliegue el xml y el html de manera correcta.

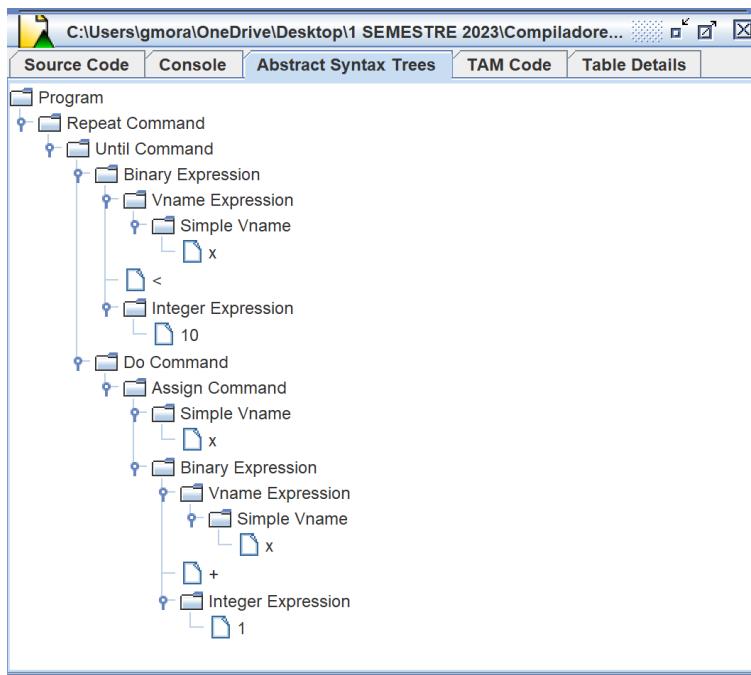
Resultados observados



```
***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
Compilation was successful.
```

Enter Input

Resultados observados



This XML file does not appear to have any style information associated with it.

```

▼<Program>
  ▼<RepeatUntil>
    ▼<UntilCommand>
      ▼<BinaryExpression>
        ▼<VnameExpression>
          ▼<SimpleVname>
            <Identifier> </Identifier>
            </SimpleVname>
          </VnameExpression>
          <IntegerExpression> </IntegerExpression>
          <Operator value="<"> </Operator>
        </BinaryExpression>
      ▼<DoCommand>
        ▼<AssingCommand>
          ▼<SimpleVname>
            <Identifier> </Identifier>
            </SimpleVname>
          ▼<BinaryExpression>
            ▼<VnameExpression>
              ▼<SimpleVname>
                <Identifier> </Identifier>
                </SimpleVname>
              </VnameExpression>
              <IntegerExpression> </IntegerExpression>
              <Operator value="+"> </Operator>
            </BinaryExpression>
          </AssingCommand>
        </DoCommand>
      </UntilCommand>
    </RepeatUntil>
  </Program>

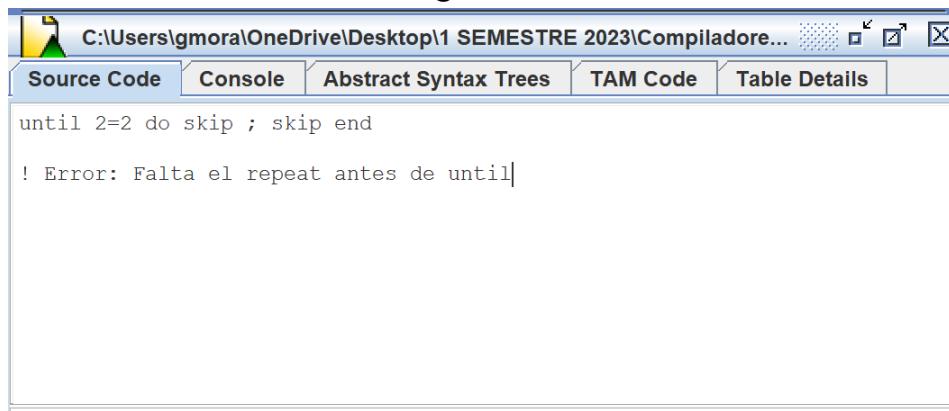
```

HTML

```
repeat until x < 10 do x := x + 1 end
```

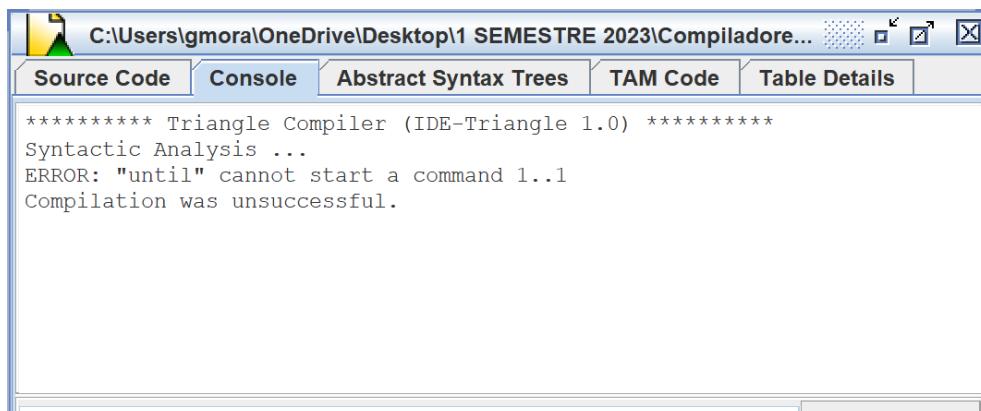
Prueba para evidenciar la capacidad del compilador para detectar errores

Diseño de la Prueba Negativa



The screenshot shows the Triangle Compiler IDE interface. The title bar reads "C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladore...". The tabs at the top are "Source Code" (selected), "Console", "Abstract Syntax Trees", "TAM Code", and "Table Details". In the "Source Code" tab, the code "until 2=2 do skip ; skip end" is entered. Below the code, an error message is displayed: "! Error: Falta el repeat antes de until". At the bottom of the window, it says "Line: 3, Position: 40".

Resultados



The screenshot shows the "Console" tab of the Triangle Compiler IDE. The output text is as follows:

```
***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
ERROR: "until" cannot start a command 1..1
Compilation was unsuccessful.
```

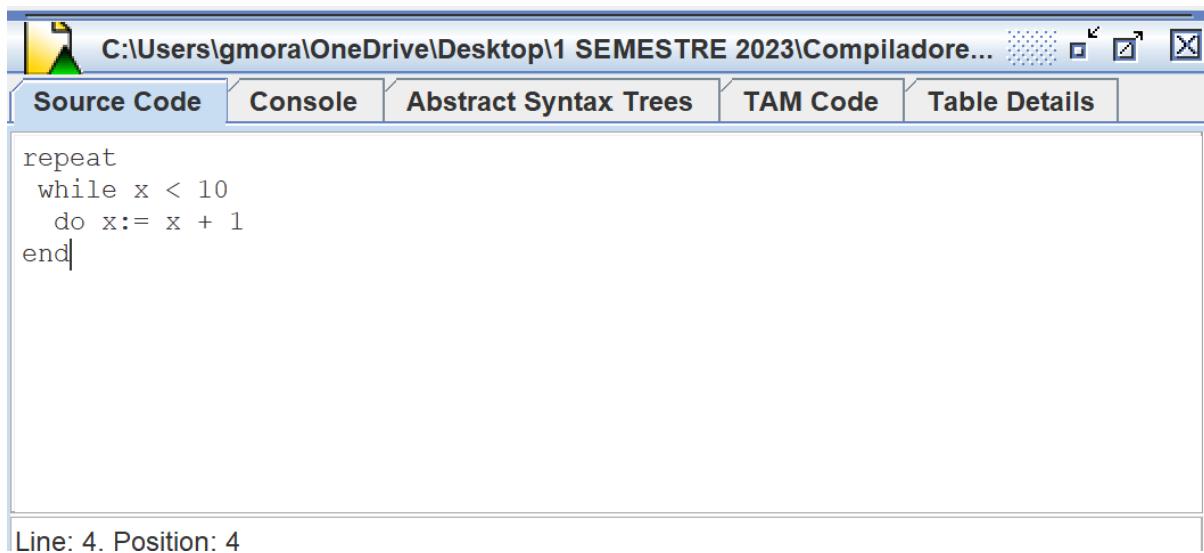
At the bottom right of the console window, there is an "Enter Input" button.

Pruebas para el comando Repeat while do end

Objetivo del caso de prueba

El objetivo de esta prueba es ver el funcionamiento de dicho comando. En este caso las pruebas responden al funcionamiento esperado para el comando repeat seguido de las palabras reservadas while y do. Primero vamos a ver el funcionamiento del comando correctamente y después otra prueba con un error.

Diseño del caso de prueba



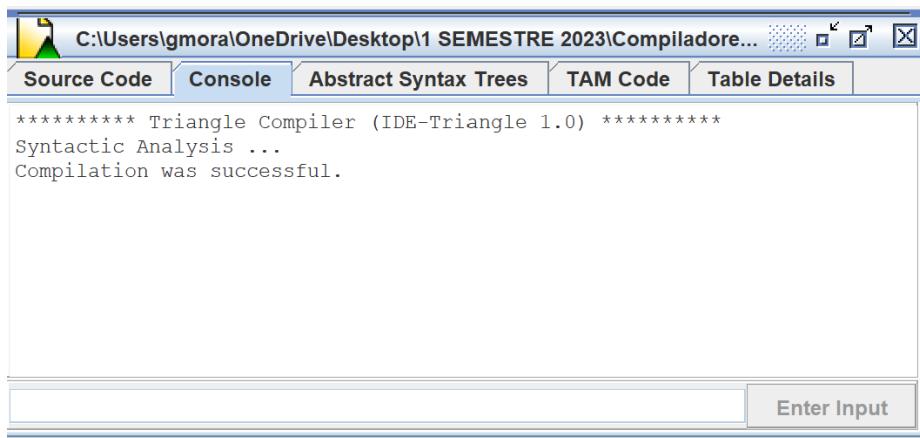
```
repeat
  while x < 10
    do x:= x + 1
  end|
```

Line: 4. Position: 4

Resultados esperados

El resultado esperado es que el programa compile de manera satisfactoria, construya el árbol, despliegue el xml y el html de manera correcta.

Resultados observados

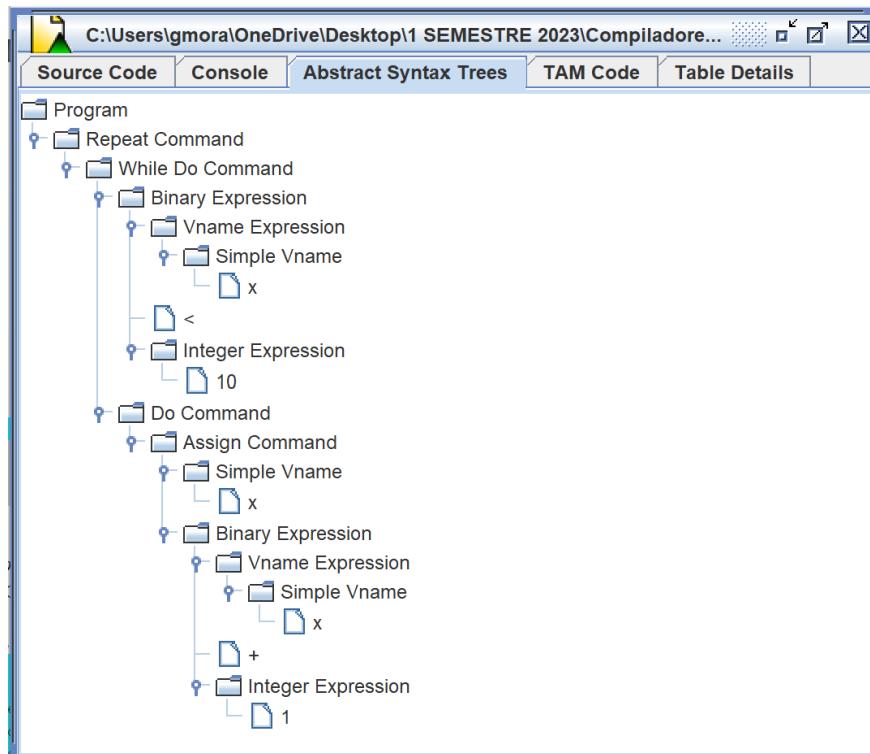


XML

```

▼<Program>
  ▼<RepeatCommand>
    ▼<WhileCommand>
      ▼<BinaryExpression>
        ▼<VnameExpression>
          ▼<SimpleVname>
            <Identifier> </Identifier>
            </SimpleVname>
          </VnameExpression>
          <IntegerExpression> </IntegerExpression>
          <Operator value="<"> </Operator>
        </BinaryExpression>
    ▼<DoCommand>
      ▼<AssingCommand>
        ▼<SimpleVname>
          <Identifier> </Identifier>
          </SimpleVname>
        ▼<BinaryExpression>
          ▼<VnameExpression>
            ▼<SimpleVname>
              <Identifier> </Identifier>
              </SimpleVname>
            </VnameExpression>
            <IntegerExpression> </IntegerExpression>
            <Operator value="+>" </Operator>
          </BinaryExpression>
        </AssingCommand>
      </DoCommand>
    </WhileCommand>
  </RepeatCommand>
</Program>

```

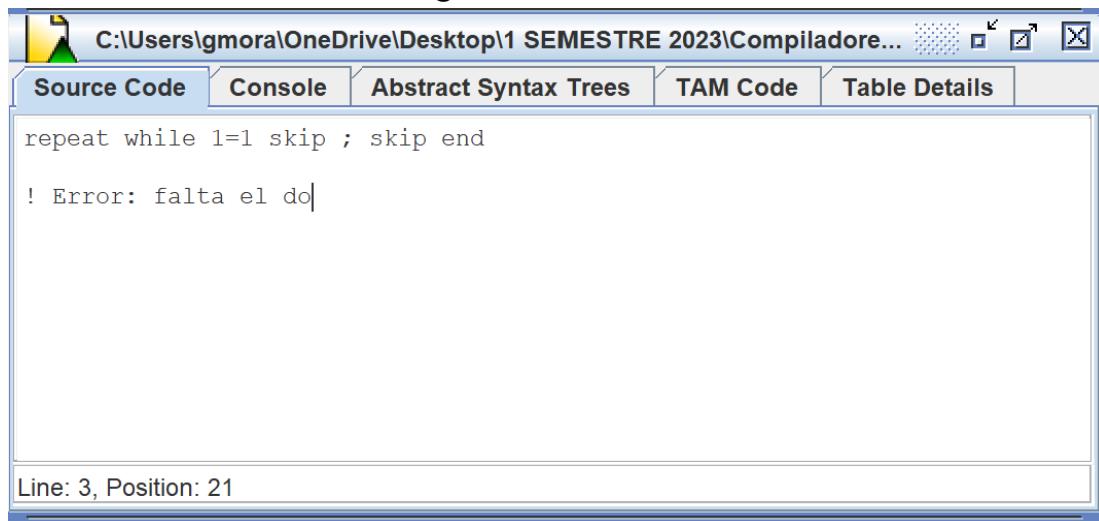


HTML

```
repeat while x < 10 do x := x + 1 end
```

Prueba para evidenciar la capacidad del compilador para detectar errores

Diseño de la Prueba Negativa



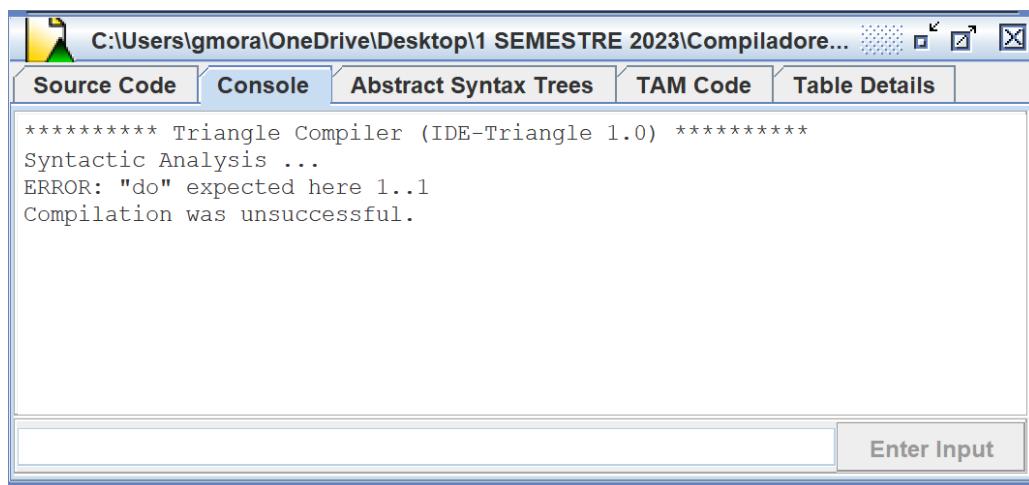
The screenshot shows the Triangle Compiler IDE interface. The title bar reads "C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladore...". The tabs at the top are "Source Code", "Console", "Abstract Syntax Trees", "TAM Code", and "Table Details". The "Source Code" tab is active, displaying the following code:

```
repeat while 1=1 skip ; skip end
! Error: falta el do|
```

A tooltip at the bottom left says "Line: 3, Position: 21". The "Console" tab shows the following output:

```
***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
ERROR: "do" expected here 1..1
Compilation was unsuccessful.
```

Resultados



The screenshot shows the Triangle Compiler IDE interface. The title bar reads "C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladore...". The tabs at the top are "Source Code", "Console", "Abstract Syntax Trees", "TAM Code", and "Table Details". The "Console" tab is active, displaying the following output:

```
***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
ERROR: "do" expected here 1..1
Compilation was unsuccessful.
```

An "Enter Input" button is visible at the bottom right of the console window.

Pruebas para el comando For Becomes do End

Objetivo del caso de prueba

El objetivo de esta prueba es ver el funcionamiento esperado para el comando For. Primero vamos a ver el funcionamiento del comando correctamente y después otra prueba con un error.

Diseño del caso de prueba

```
for x := 1 .. 10 do x:= x + 1 end
```

Line: 2, Position: 34

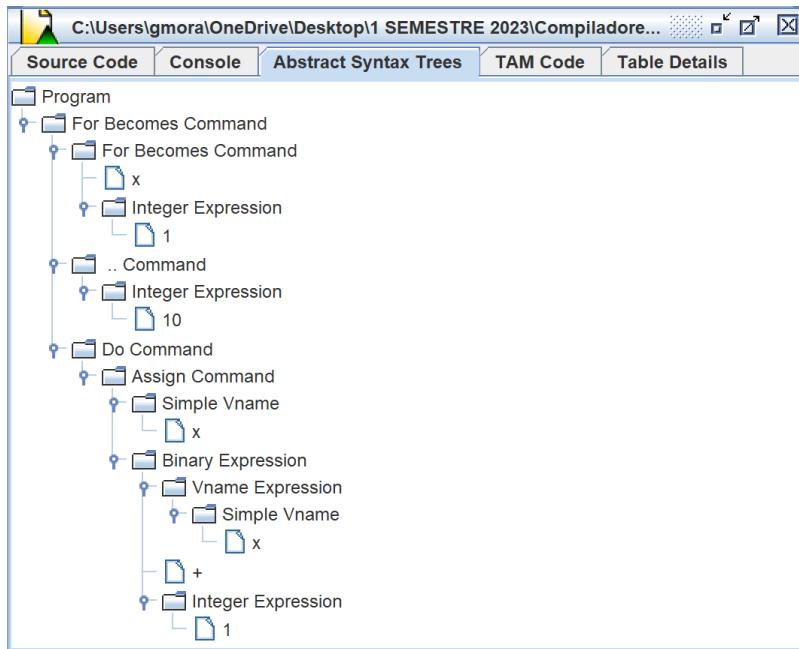
Resultados esperados

El resultado esperado es que el programa compile de manera satisfactoria, construya el árbol, despliegue el XML y el HTML de manera correcta.

Resultados observados

```
***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
Compilation was successful.
```

Enter Input



HTML

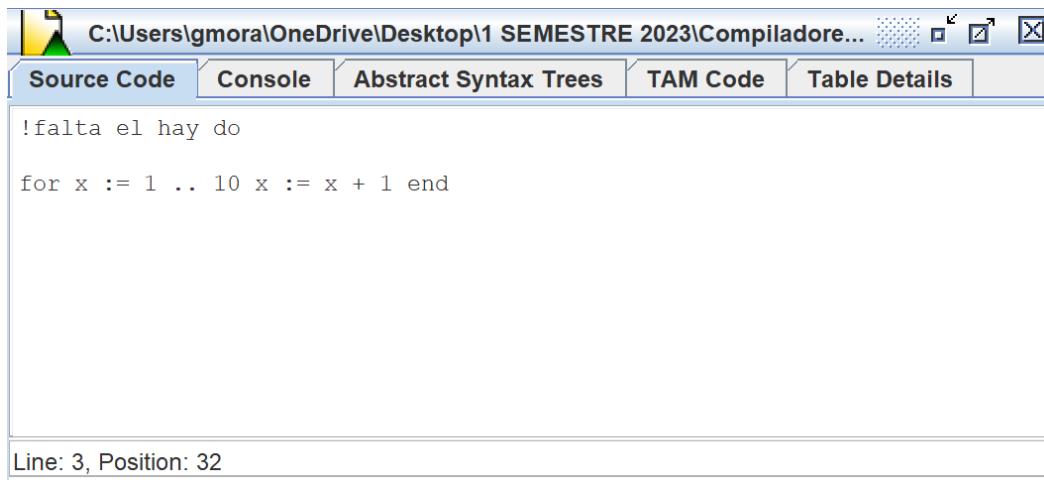
```
for x := 1 .. 10 do x := x + 1 end
```

XML

```
<?xml version="1.0" encoding="UTF-8"?>
<Program>
    <ForBecomes>
        <DotDCommand>
            <IntegerExpression> </IntegerExpression>
        </DotDCommand>
        <ForBecomesCommand>
            <Identifier> </Identifier>
            <IntegerExpression> </IntegerExpression>
        </ForBecomesCommand>
    <DoCommand>
        <AssingCommand>
            <SimpleVname>
                <Identifier> </Identifier>
            </SimpleVname>
            <BinaryExpression>
                <VnameExpression>
                    <SimpleVname>
                        <Identifier> </Identifier>
                    </SimpleVname>
                </VnameExpression>
                <IntegerExpression> </IntegerExpression>
                <Operator value="+"> </Operator>
            </BinaryExpression>
        </AssingCommand>
    </DoCommand>
</ForBecomes>
</Program>
```

Prueba para evidenciar la capacidad del compilador para detectar errores

Diseño de la Prueba Negativa

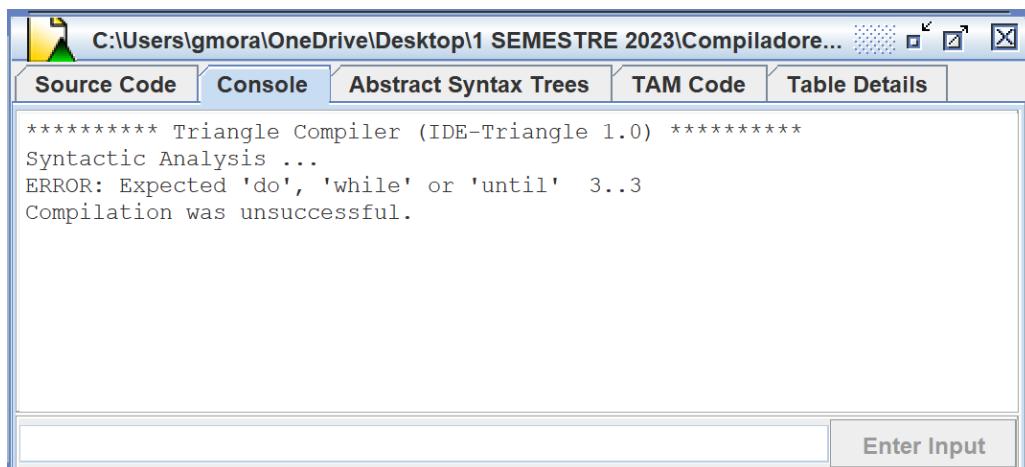


The screenshot shows the Triangle Compiler IDE interface. The title bar reads "C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladore...". The tabs at the top are "Source Code", "Console", "Abstract Syntax Trees", "TAM Code", and "Table Details". The "Source Code" tab is active, displaying the following code:

```
!falta el hay do
for x := 1 .. 10 x := x + 1 end
```

Below the code editor, a status bar indicates "Line: 3, Position: 32".

Resultados



The screenshot shows the Triangle Compiler IDE interface with the "Console" tab active. The output window displays the following text:

```
***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
ERROR: Expected 'do', 'while' or 'until' 3..3
Compilation was unsuccessful.
```

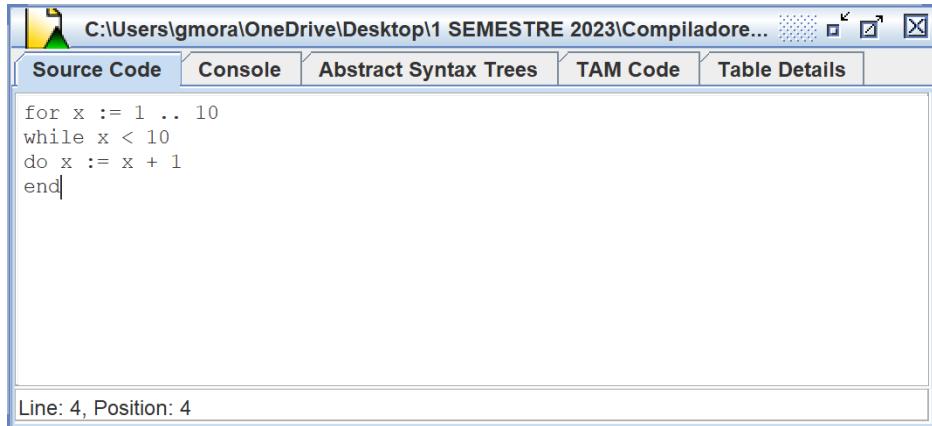
At the bottom right of the console window, there is an "Enter Input" button.

Pruebas para el comando For do while end

Objetivo del caso de prueba

El objetivo de esta prueba es ver el funcionamiento del segundo flujo probable para el comando For que toma en cuenta la palabra reservada While. Primero vamos a ver el funcionamiento del comando correctamente y después otra prueba con un error para ver cómo es que el compilador se comporta.

Diseño del caso de prueba



A screenshot of the Triangle Compiler IDE interface. The title bar shows the path: C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladore... . The window has several tabs at the top: Source Code (selected), Console, Abstract Syntax Trees, TAM Code, and Table Details. The main area contains the following Pseudocode:

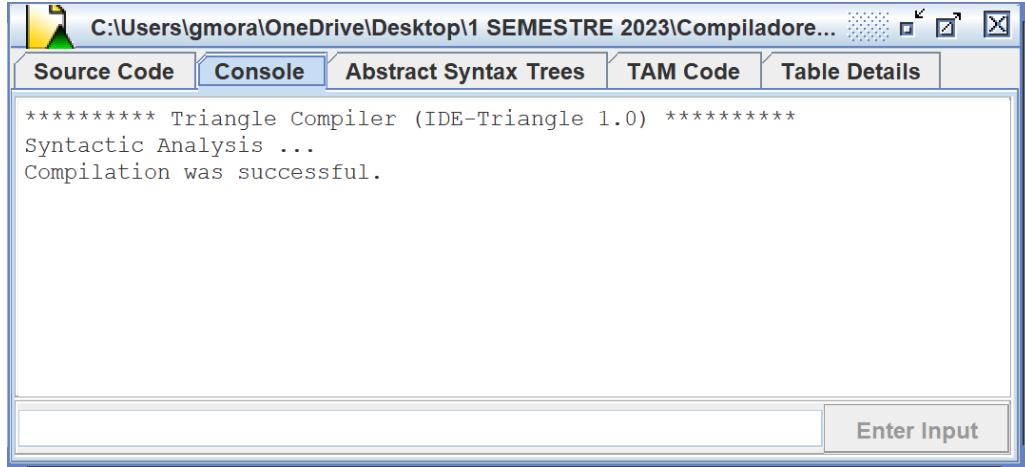
```
for x := 1 .. 10
while x < 10
do x := x + 1
end|
```

The status bar at the bottom indicates "Line: 4, Position: 4".

Resultados esperados

El resultado esperado es que el programa compile de manera satisfactoria, construya el árbol, despliegue el xml y el html de manera correcta.

Resultados observados



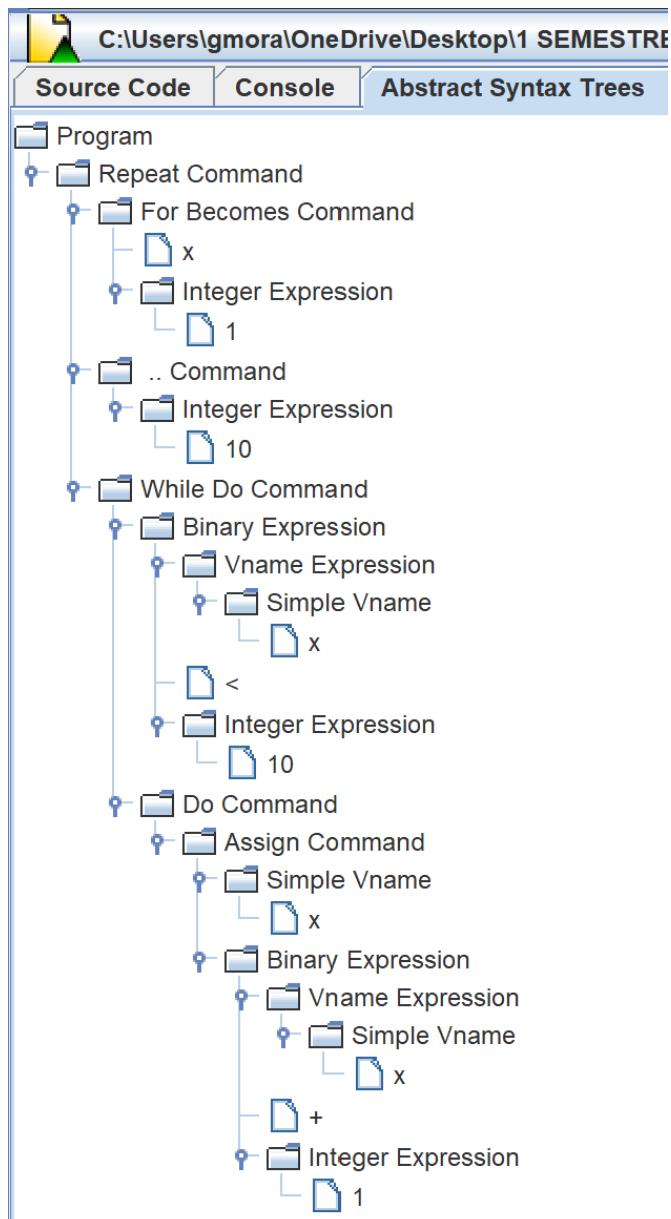
A screenshot of the Triangle Compiler IDE interface. The title bar shows the path: C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladore... . The window has several tabs at the top: Source Code, Console (selected), Abstract Syntax Trees, TAM Code, and Table Details. The main area displays the following output from the compiler:

```
***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
Compilation was successful.
```

The status bar at the bottom indicates "Enter Input".

XML

```
▼<Program>
  ▼<RepeatForWhile>
    ▼<DotDCommand>
      <IntegerExpression> </IntegerExpression>
    </DotDCommand>
    ▼<ForBecomesCommand>
      <Identifier> </Identifier>
      <IntegerExpression> </IntegerExpression>
    </ForBecomesCommand>
    ▼<WhileCommand>
      ▼<BinaryExpression>
        ▼<VnameExpression>
          ▼<SimpleVname>
            <Identifier> </Identifier>
          </SimpleVname>
        </VnameExpression>
        <IntegerExpression> </IntegerExpression>
        <Operator value="<"> </Operator>
      </BinaryExpression>
    ▼<DoCommand>
      ▼<AssingCommand>
        ▼<SimpleVname>
          <Identifier> </Identifier>
        </SimpleVname>
        ▼<BinaryExpression>
          ▼<VnameExpression>
            ▼<SimpleVname>
              <Identifier> </Identifier>
            </SimpleVname>
          </VnameExpression>
          <IntegerExpression> </IntegerExpression>
          <Operator value="+"> </Operator>
        </BinaryExpression>
      </AssingCommand>
    </DoCommand>
  </WhileCommand>
</RepeatForWhile>
</Program>
```

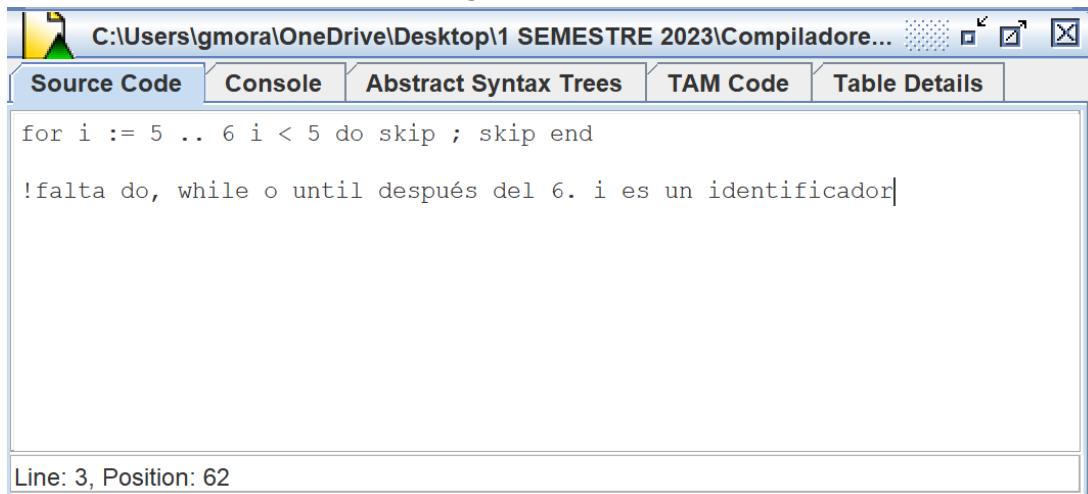


HTML

```
for x := 1 .. 10 while x < 10 do x := x + 1 end
```

Prueba para evidenciar la capacidad del compilador para detectar errores

Diseño de la Prueba Negativa

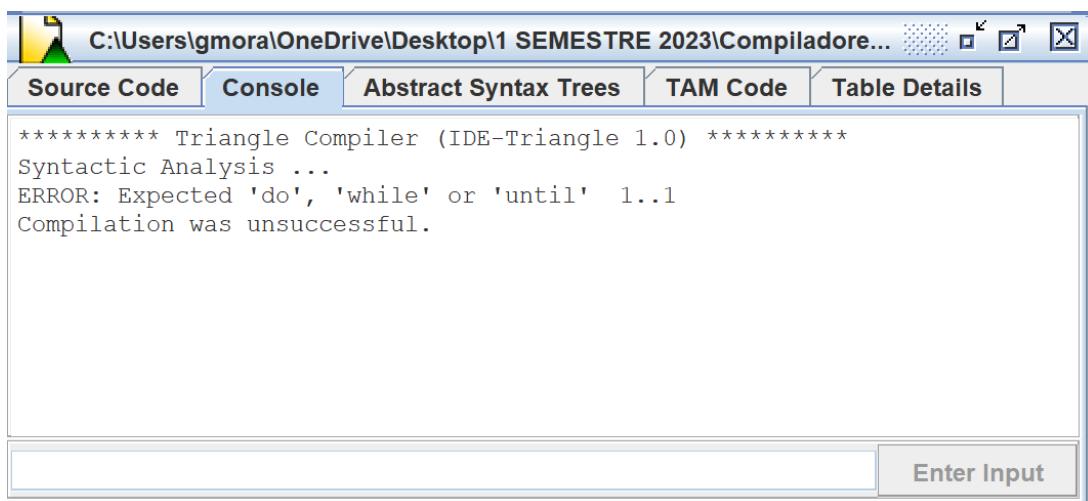


The screenshot shows the Triangle Compiler IDE interface. The title bar reads "C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladore...". The tabs at the top are "Source Code", "Console", "Abstract Syntax Trees", "TAM Code", and "Table Details". The "Source Code" tab is active, displaying the following code:

```
for i := 5 .. 6 i < 5 do skip ; skip end
!falta do, while o until después del 6. i es un identificador
```

Below the code editor, a status bar shows "Line: 3, Position: 62".

Resultados



The screenshot shows the "Console" tab of the Triangle Compiler IDE. The output window displays the following text:

```
***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
ERROR: Expected 'do', 'while' or 'until' 1..1
Compilation was unsuccessful.
```

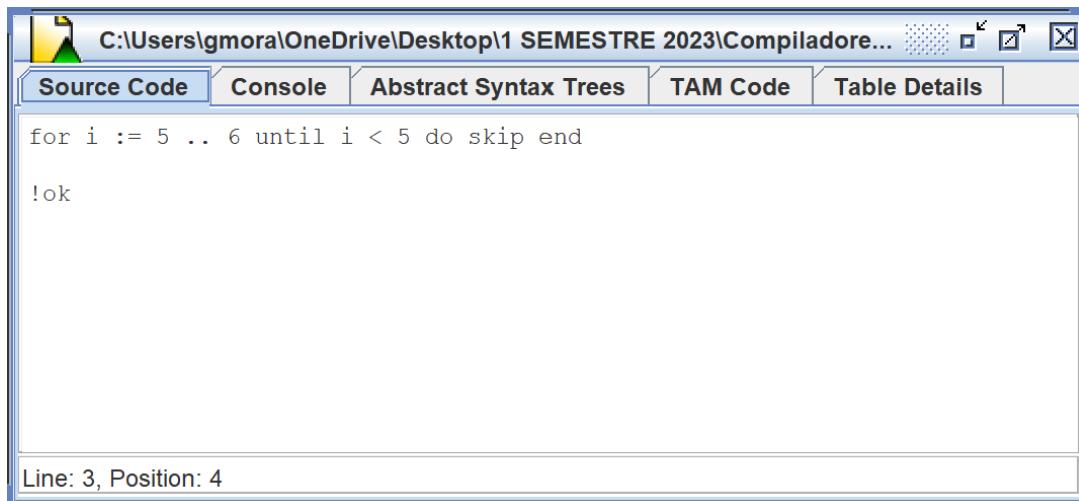
At the bottom right of the console window, there is an "Enter Input" button.

Pruebas para el comando For until do end

Objetivo del caso de prueba

El objetivo de esta prueba es ver el funcionamiento del segundo flujo probable para el comando For que toma en cuenta la palabra reservada until. Primero vamos a ver el funcionamiento del comando correctamente y después otra prueba con un error para ver cómo es que el compilador se comporta.

Diseño del caso de prueba



The screenshot shows a software window titled 'C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladore...'. The 'Source Code' tab is active, displaying the following pseudocode:

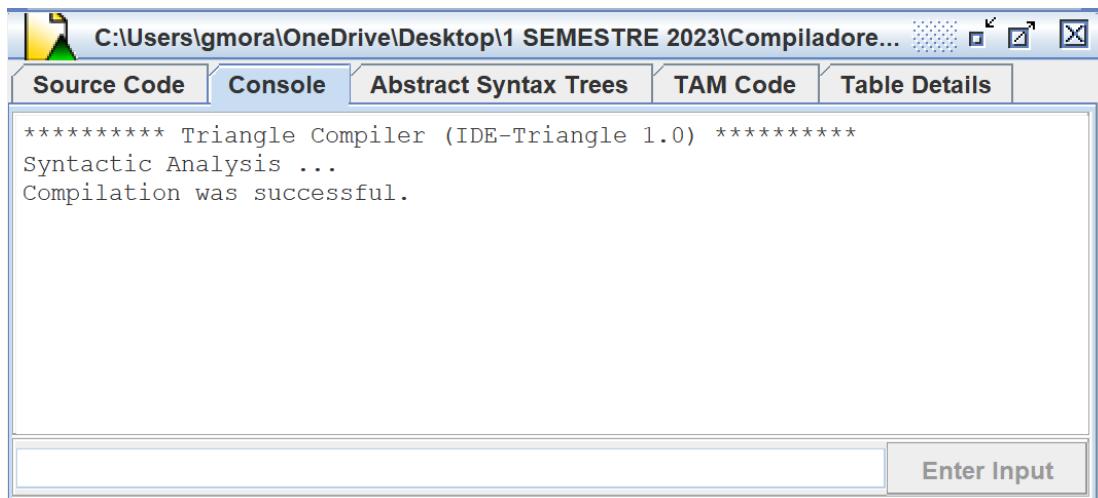
```
for i := 5 .. 6 until i < 5 do skip end
!ok
```

Below the code editor, a status bar indicates 'Line: 3, Position: 4'.

Resultados esperados

El resultado esperado es que el programa compile de manera satisfactoria, construya el árbol, despliegue el xml y el html de manera correcta.

Resultados observados



The screenshot shows a software window titled 'C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladore...'. The 'Console' tab is active, displaying the following output from the Triangle Compiler:

```
***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
Compilation was successful.
```

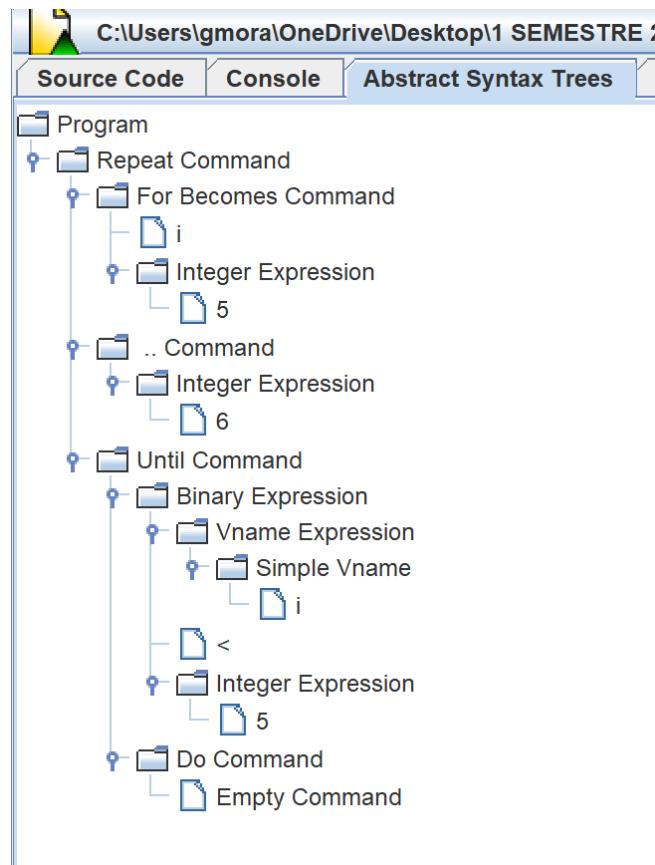
At the bottom right of the console window, there is an 'Enter Input' button.

XML

```

▼<Program>
  ▼<RepeatForUntil>
    ▼<DotDCommand>
      <IntegerExpression> </IntegerExpression>
    </DotDCommand>
    ▼<ForBecomesCommand>
      <Identifier> </Identifier>
      <IntegerExpression> </IntegerExpression>
    </ForBecomesCommand>
    ▼<UntilCommand>
      ▼<BinaryExpression>
        ▼<VnameExpression>
          ▼<SimpleVname>
            <Identifier> </Identifier>
          </SimpleVname>
        </VnameExpression>
        <IntegerExpression> </IntegerExpression>
        <Operator value="<"> </Operator>
      </BinaryExpression>
    ▼<DoCommand>
      <EmptyCommand> </EmptyCommand>
    </DoCommand>
  </UntilCommand>
</RepeatForUntil>
</Program>

```

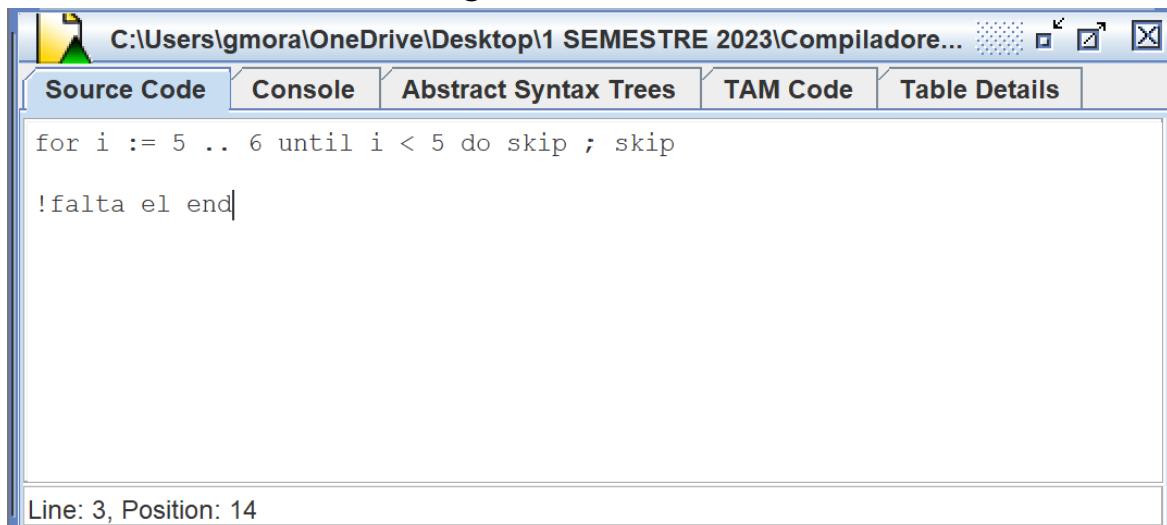


HTML

```
for i := 5 .. 6 until i < 5 do skip end
```

Prueba para evidenciar la capacidad del compilador para detectar errores

Diseño de la Prueba Negativa

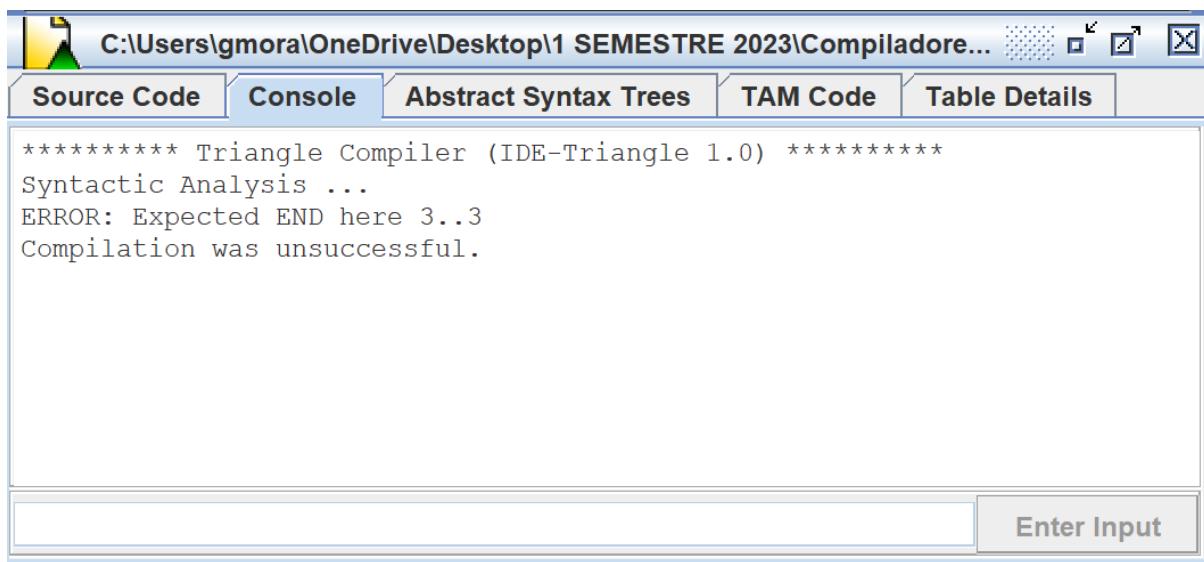


The screenshot shows the Triangle Compiler IDE interface. The title bar reads "C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladores\triangle\triangle\triangle.exe". The tabs at the top are "Source Code", "Console", "Abstract Syntax Trees", "TAM Code", and "Table Details". The "Source Code" tab is active, displaying the following code:

```
for i := 5 .. 6 until i < 5 do skip ; skip
!falta el end|
```

A cursor is positioned at the end of the line "falta el end|". A status bar at the bottom indicates "Line: 3, Position: 14".

Resultados



The screenshot shows the "Console" tab of the Triangle Compiler IDE. The output window displays the following text:

```
***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
ERROR: Expected END here 3..3
Compilation was unsuccessful.
```

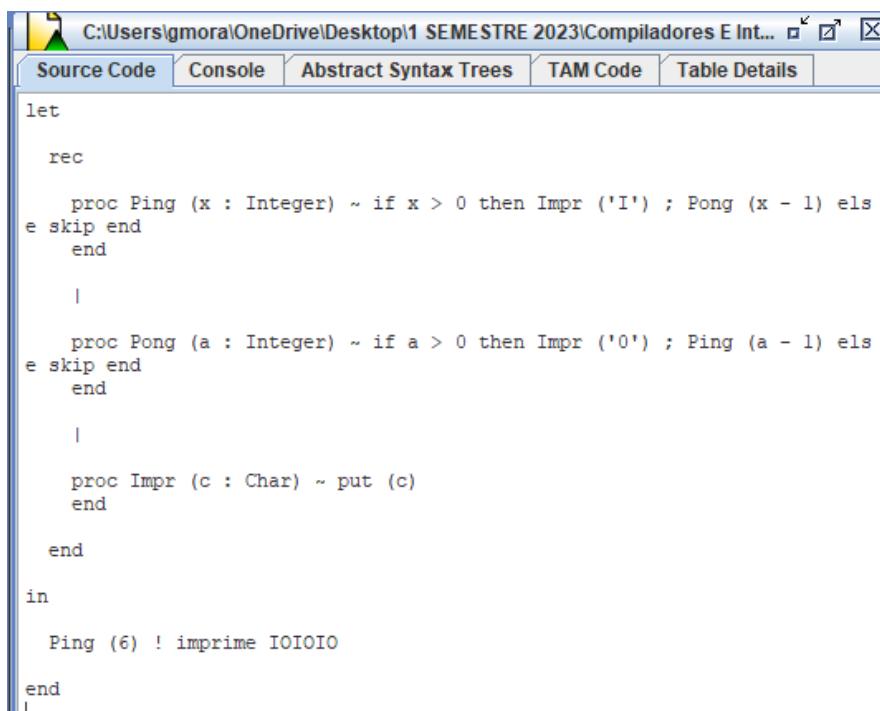
At the bottom right of the console window is a button labeled "Enter Input".

Pruebas para el comando Rec

Objetivo del caso de prueba

El objetivo de esta prueba es ver el funcionamiento del comando que incluye la palabra reservada Rec en las declaraciones. Primero vamos a ver el funcionamiento del comando correctamente y después otra prueba con un error para ver cómo es que el compilador se comporta.

Diseño del caso de prueba



```

C:\Users\gmoral\OneDrive\Desktop\1 SEMESTRE 2023\Compiladores E Int...
Source Code Console Abstract Syntax Trees TAM Code Table Details

let
  rec
    proc Ping (x : Integer) ~ if x > 0 then Impr ('I') ; Pong (x - 1) else skip end
    end
  |
  proc Pong (a : Integer) ~ if a > 0 then Impr ('O') ; Ping (a - 1) else skip end
  end
  |
  proc Impr (c : Char) ~ put (c)
  end
end
in
Ping (6) ! imprime IOIOIO
end
|

```

Resultados esperados

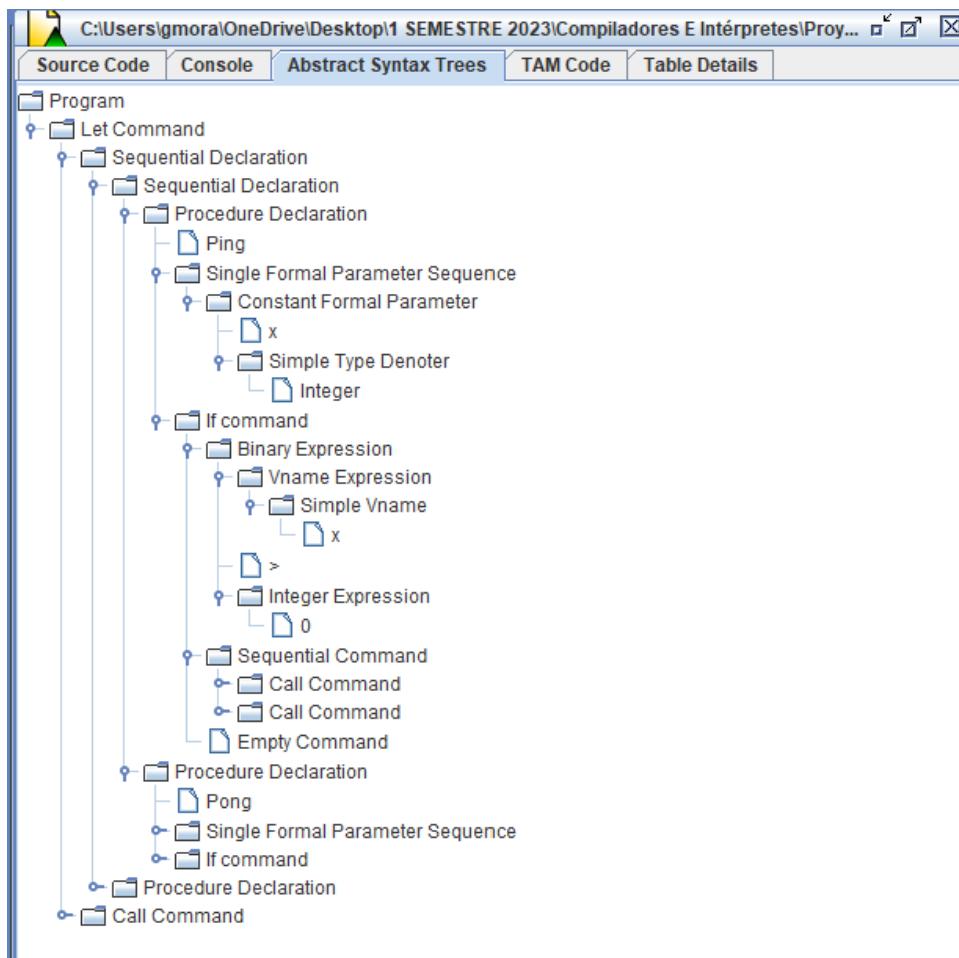
El resultado esperado es que el programa compile de manera satisfactoria, construya el árbol, despliegue el XML y el HTML de manera correcta.

Resultados observados

C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladores E Int... □ □

Source Code Console Abstract Syntax Trees TAM Code Table Details

```
***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
Compilation was successful.
```



HTML

```
let rec proc Ping ( x : Integer ) ~if x > 0 then Impr ( 'I' ) ; Pong ( x - 1 ) else skip end end | proc Impr ( c : Char ) ~ put ( c ) end end in Ping ( 6 ) end
```

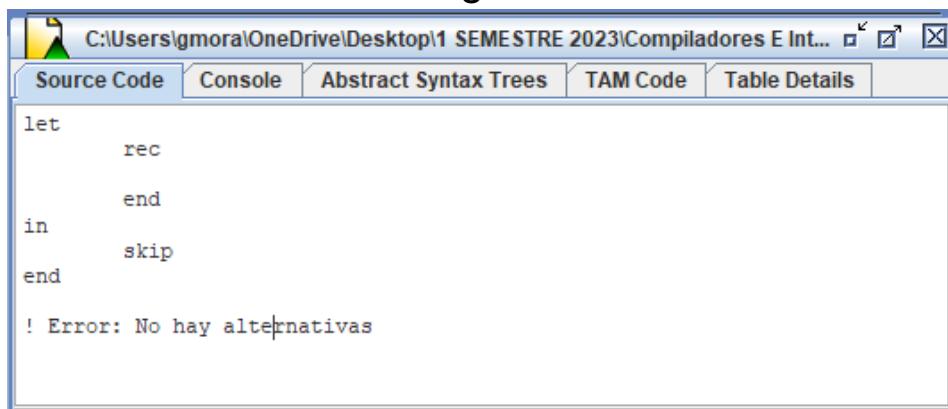
XML

```

<Program>
<LetCommand>
<SequentialDeclaration>
<SequentialDeclaration>
<ProcDeclaration>
<SingleFormalParameterSequence>
<ConstFormalParameter>
<Identifier>
</Identifier>
<SimpleTypeDenoter>
</SimpleTypeDenoter>
<Identifier>
</Identifier>
</ConstFormalParameter>
</SingleFormalParameterSequence>
<IfCommand>
<BinaryExpression>
<VnameExpression>
<SimpleVname>
<Identifier>
</Identifier>
</SimpleVname>
</VnameExpression>
<IntegerExpression>
</IntegerExpression>
<Operator value='>'>
</Operator>
</BinaryExpression>
<SequentialCommand>
<CallCommand>
<Identifier>
</Identifier>
<SingleActualParameterSequence>
<ConstActualParameter>
```

Prueba para evidenciar la capacidad del compilador para detectar errores

Diseño de la Prueba Negativa



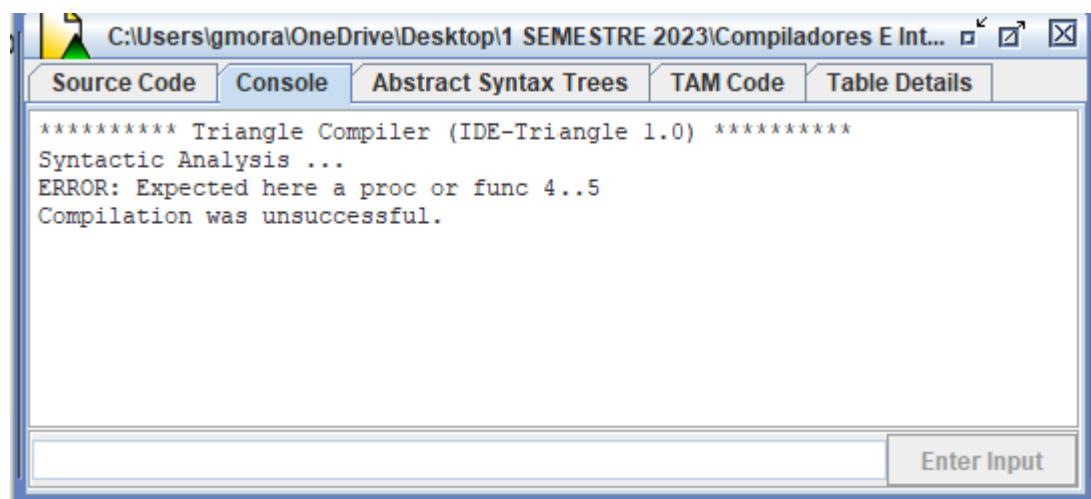
The screenshot shows a software interface with a title bar "C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladores E Int...". Below the title bar are five tabs: "Source Code" (selected), "Console", "Abstract Syntax Trees", "TAM Code", and "Table Details". The "Source Code" tab contains the following text:

```

let
    rec
    end
in
    skip
end
```

Below this, an exclamation mark followed by the error message "Error: No hay alternativas" is displayed.

Resultados



The screenshot shows the Triangle Compiler IDE interface. The title bar reads "C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladores E Int...". The "Console" tab is selected, displaying the following output:

```
***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
ERROR: Expected here a proc or func 4..5
Compilation was unsuccessful.
```

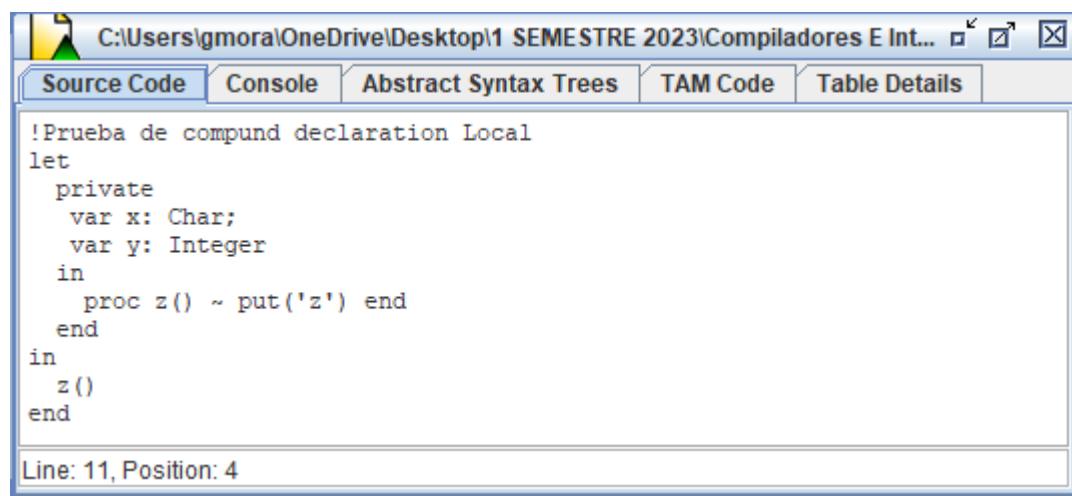
Below the console window is a text input field labeled "Enter Input".

Pruebas del comando Private

Objetivo del caso de prueba

El objetivo de esta prueba es ver el funcionamiento del comando que incluye la palabra reservada `Private` en las declaraciones. Primero vamos a ver el funcionamiento del comando correctamente y después otra prueba con un error para ver cómo es que el compilador se comporta.

Diseño del caso de prueba



The screenshot shows the Triangle Compiler IDE interface. The title bar reads "C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladores E Int...". The "Source Code" tab is selected, displaying the following code:

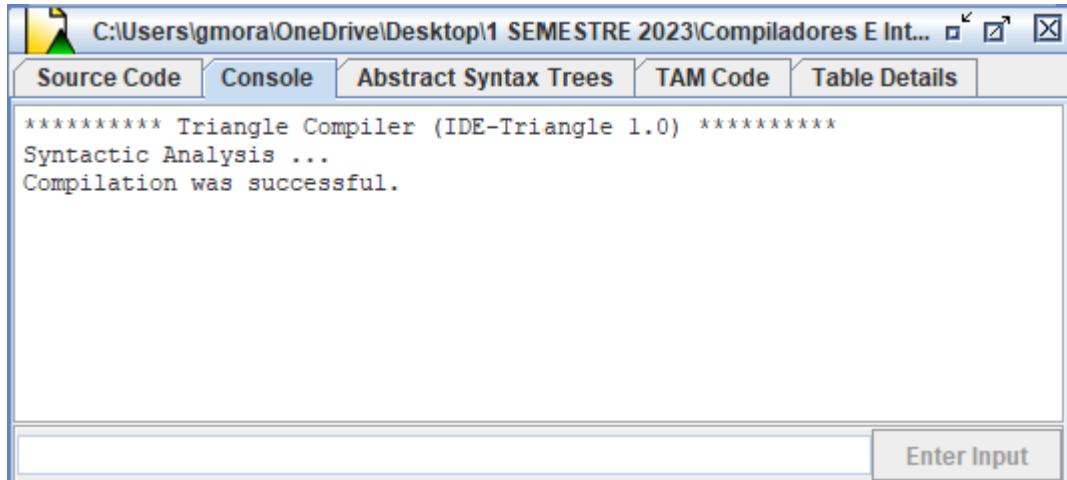
```
!Prueba de compund declaration Local
let
  private
    var x: Char;
    var y: Integer
  in
    proc z() ~ put('z') end
  end
in
  z()
end
```

At the bottom of the source code window, it says "Line: 11, Position: 4".

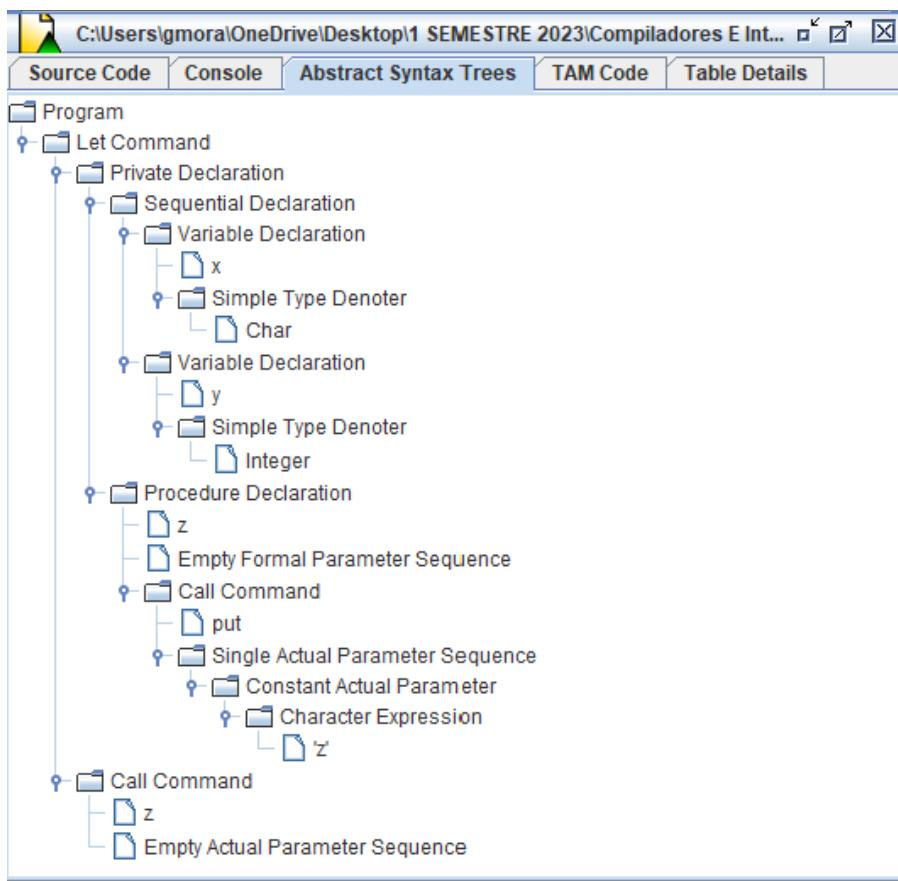
Resultados esperados

El resultado esperado es que el programa compile de manera satisfactoria, construya el árbol, despliegue el XML y el HTML de manera correcta.

Resultados observados



***** Triangle Compiler (IDE-Triangle 1.0) *****
 Syntactic Analysis ...
 Compilation was successful.



XML

```

<Program>
  <LetCommand>
    <PrivateDeclaration>
      <SequentialDeclaration>
        <VarDeclaration>
          </VarDeclaration>
        <SimpleTypeDenoter>
          </SimpleTypeDenoter>
        <Identifier>
          </Identifier>
        <VarDeclaration>
          </VarDeclaration>
        <SimpleTypeDenoter>
          </SimpleTypeDenoter>
        <Identifier>
          </Identifier>
        </SequentialDeclaration>
      <ProcDeclaration>
        <EmptyFormalParameterSequence>
        </EmptyFormalParameter>
      <CallCommand>
        <Identifier>
          </Identifier>
        <SingleActualParameterSequence>
          <ConstActualParameter>
            <CharacterExpression>
              <CharacterLiteral>
                </CharacterLiteral>
              </CharacterExpression>
            </ConstActualParameter>
          </SingleActualParameterSequence>
        </CallCommand>
      </ProcDeclaration>
    </PrivateDeclaration>
  <CallCommand>
    <Identifier>
      </Identifier>
    <EmptyActualParameterSequence>
    </EmptyActualParameterSequence>
  </CallCommand>
</LetCommand>
</Program>

```

HTML

```

let private var  x  :  Char  ; var  y  :  Integer  in proc  z  (  )  ~  put  (  'z'  )  end  end  in  z  (  )  end

```

Prueba para evidenciar la capacidad del compilador para detectar errores

Diseño de la Prueba Negativa

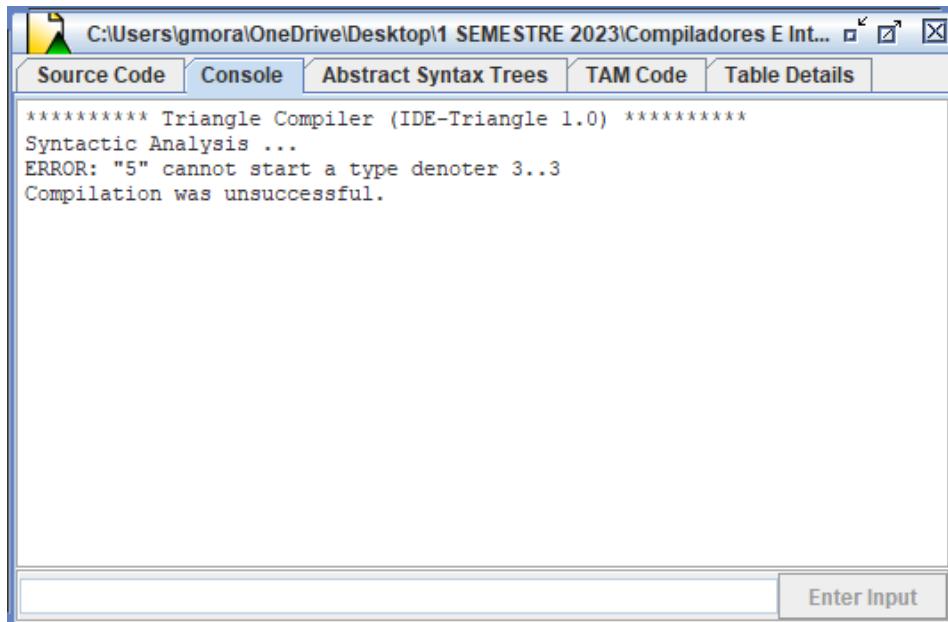
```

let
    private
        var a := 5;
        var b := 15
    ! in falta, reporta error porque espera ; o in
        var c := a * b *2
    end ; ! composición secuencial de declaraciones
    var d := 5
in
    putint(c)
end

! Error: Falta el in del private

```

Resultados



The screenshot shows the Triangle Compiler IDE interface. The title bar reads "C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladores E Int...". The main window has tabs for "Source Code", "Console", "Abstract Syntax Trees", "TAM Code", and "Table Details". The "Console" tab is active, displaying the following output:

```

***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
ERROR: "5" cannot start a type denoter 3..3
Compilation was unsuccessful.

```

At the bottom of the console window, there is an "Enter Input" field.

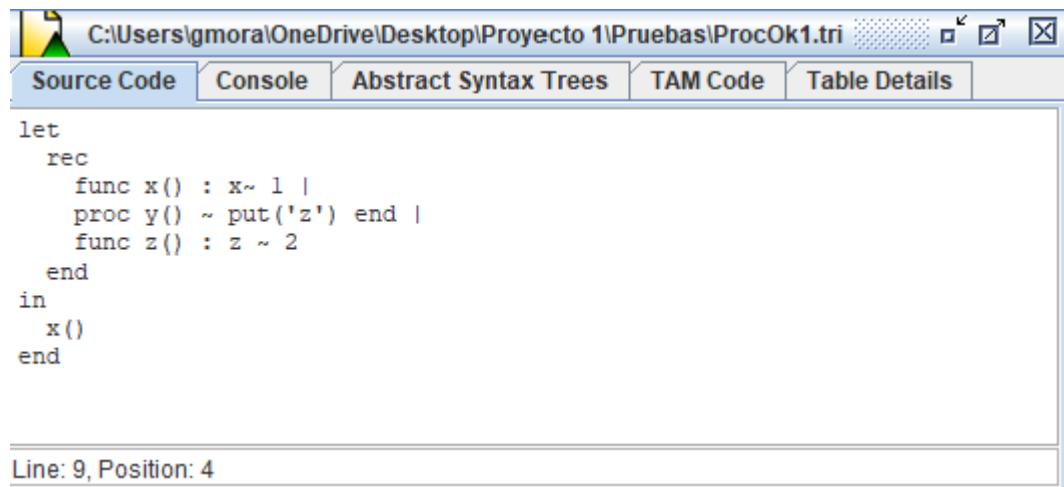
Pruebas del comando Proc

Objetivo del caso de prueba

El objetivo de Proc prueba es ver el funcionamiento del comando que incluye la palabra reservada Rec en las declaraciones. Primero vamos a ver el

funcionamiento del comando correctamente y después otra prueba con un error para ver cómo es que el compilador se comporta.

Diseño del caso de prueba

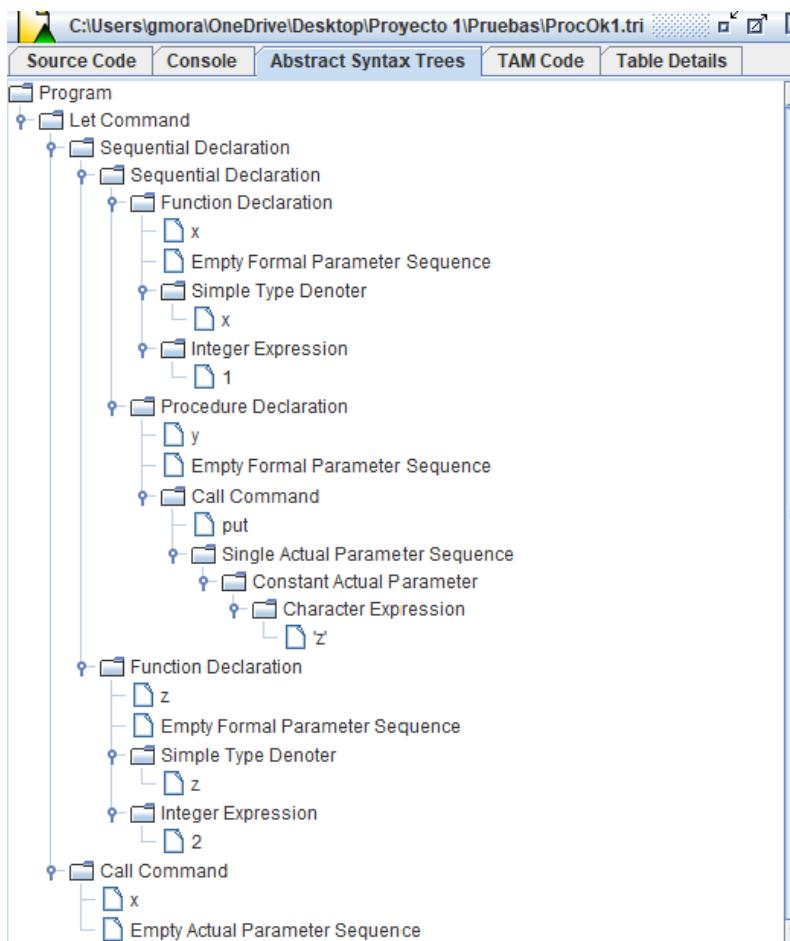


```
let
rec
  func x() : x~ 1 |
  proc y() ~ put('z') end |
  func z() : z ~ 2
end
in
x()
end
```

Line: 9, Position: 4

Resultados esperados

El resultado esperado es que el programa compile de manera satisfactoria, construya el árbol, despliegue el XML y el HTML de manera correcta.



Resultados observados

```

*****
Triangle Compiler (IDE-Triangle 1.0)
*****
Syntactic Analysis ...
Compilation was successful.
  
```

HTML

```

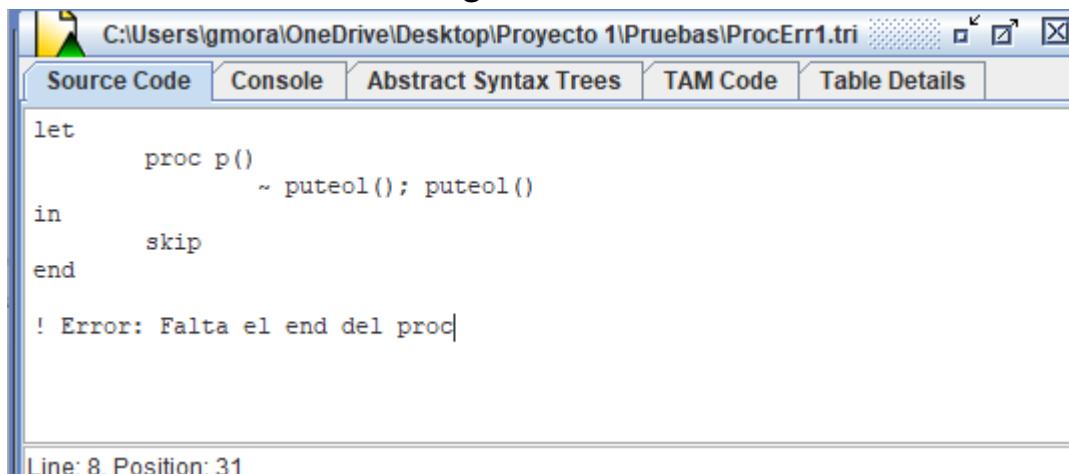
let rec func  x  ( )  : x ~ 1  | proc  y  ( )  ~  put  ( 'z' ) end  | func  z  ( )  : z ~ 2 end in  x  ( ) end
  
```

XML

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Program>
<LetCommand>
<SequentialDeclaration>
<SequentialDeclaration>
<FuncDeclaration>
<SimpleTypeDenoter>
</SimpleTypeDenoter>
<Identifier>
</Identifier>
<EmptyFormalParameterSequence>
</EmptyFormalParameter>
<IntegerExpression>
</IntegerExpression>
</FuncDeclaration>
<ProcDeclaration>
<EmptyFormalParameterSequence>
</EmptyFormalParameter>
<CallCommand>
<Identifier>
</Identifier>
<SingleActualParameterSequence>
<ConstActualParameter>
<CharacterExpression>
<CharacterLiteral>
</CharacterLiteral>
</CharacterExpression>
</ConstActualParameter>
</SingleActualParameterSequence>
</CallCommand>
</ProcDeclaration>
</SequentialDeclaration>
<FuncDeclaration>
<SimpleTypeDenoter>
</SimpleTypeDenoter>
<Identifier>
</Identifier>
<EmptyFormalParameterSequence>
</EmptyFormalParameter>
<IntegerExpression>
</IntegerExpression>
```

Prueba para evidenciar la capacidad del compilador para detectar errores

Diseño de la Prueba Negativa



The screenshot shows the Triangle Compiler IDE interface. The title bar reads "C:\Users\gmora\OneDrive\Desktop\Proyecto 1\Pruebas\ProcErr1.tri". The "Source Code" tab is selected, displaying the following code:

```

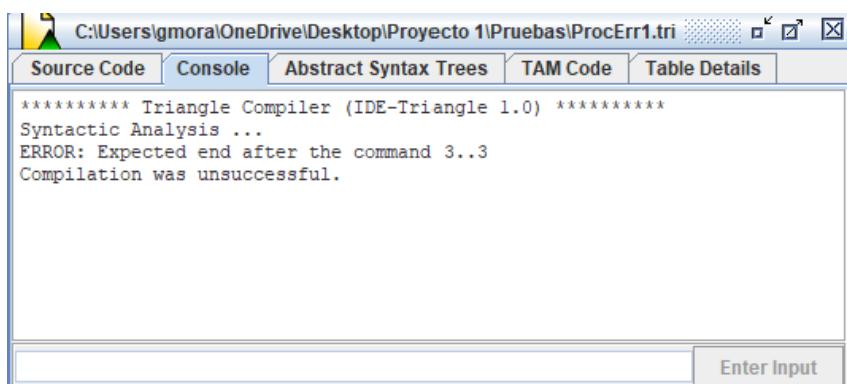
let
    proc p()
        ~ puteol(); puteol()
in
    skip
end

! Error: Falta el end del proc

```

A red error highlight is present under the word "proc". A tooltip at the bottom left says "Line: 8, Position: 31".

Resultados



The screenshot shows the "Console" tab of the Triangle Compiler IDE. The output is:

```

***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
ERROR: Expected end after the command 3..3
Compilation was unsuccessful.

```

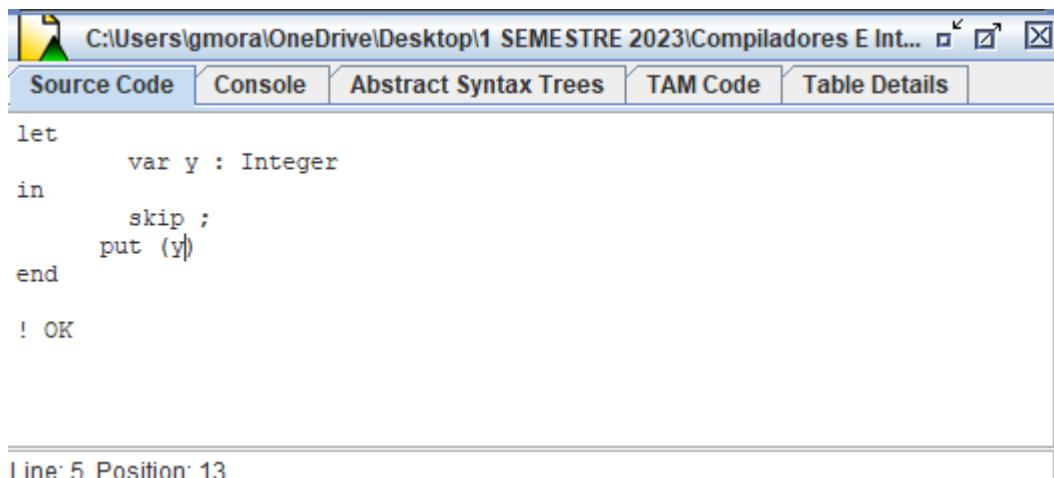
An "Enter Input" button is visible at the bottom right.

Pruebas del comando Var

Objetivo del caso de prueba

El objetivo de la prueba es ver el funcionamiento del comando que incluye la palabra reservada Var en las declaraciones. Primero vamos a ver el funcionamiento del comando correctamente y después otra prueba con un error para ver cómo es que el compilador se comporta.

Diseño del caso de prueba



The screenshot shows the Triangle Compiler IDE interface. The title bar reads "C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladores E Int...". The tabs at the top are "Source Code" (selected), "Console", "Abstract Syntax Trees", "TAM Code", and "Table Details". The code area contains the following Pascal-like code:

```
let
    var y : Integer
in
    skip ;
    put (y)
end

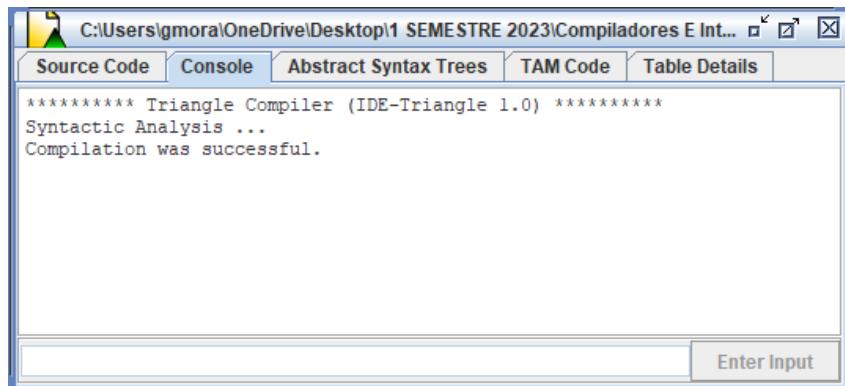
! OK
```

Line: 5 Position: 13

Resultados esperados

El resultado esperado es que el programa compile de manera satisfactoria, construya el árbol, despliegue el XML y el HTML de manera correcta.

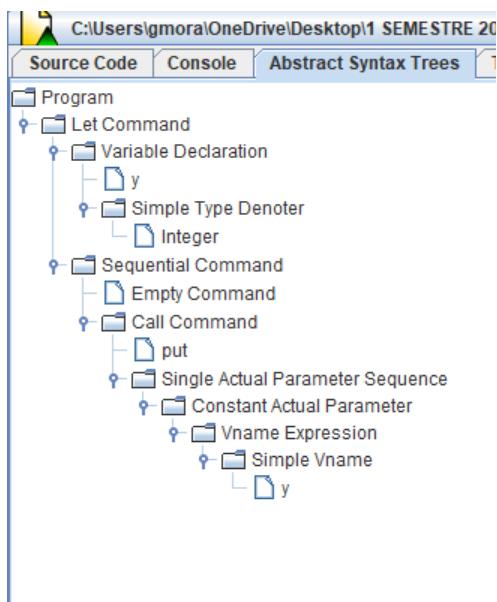
Resultados observados



The screenshot shows the Triangle Compiler IDE interface. The title bar reads "C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladores E Int...". The tabs at the top are "Source Code", "Console" (selected), "Abstract Syntax Trees", "TAM Code", and "Table Details". The console window displays the following output:

```
***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
Compilation was successful.
```

An "Enter Input" button is visible at the bottom right of the console window.



HTML

```
let var y : Integer in skip ; put ( y ) end
```

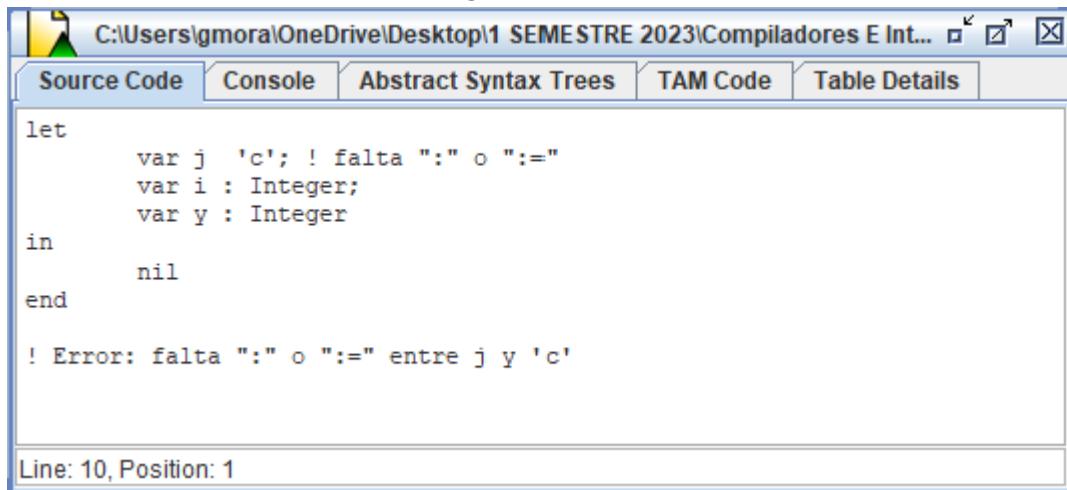
XML

```

<?xml version="1.0" encoding="UTF-8"?>
<Program>
  <LetCommand>
    <VarDeclaration> </VarDeclaration>
    <SimpleTypeDenoter> </SimpleTypeDenoter>
    <Identifier> </Identifier>
  <SequentialCommand>
    <EmptyCommand> </EmptyCommand>
    <CallCommand>
      <Identifier> </Identifier>
      <SingleActualParameterSequence>
        <ConstActualParameter>
          <VnameExpression>
            <SimpleVname>
              <Identifier> </Identifier>
            </SimpleVname>
          </VnameExpression>
        </ConstActualParameter>
      <SingleActualParameterSequence>
        <CallCommand>
          </CallCommand>
        </SingleActualParameterSequence>
      </CallCommand>
    </SequentialCommand>
  </LetCommand>
</Program>
  
```

Prueba para evidenciar la capacidad del compilador para detectar errores

Diseño de la Prueba Negativa



The screenshot shows a window titled 'C:\Users\gmora\OneDrive\Desktop\1 SEMESTRE 2023\Compiladores E Int...'. The 'Source Code' tab is selected, displaying the following pseudocode:

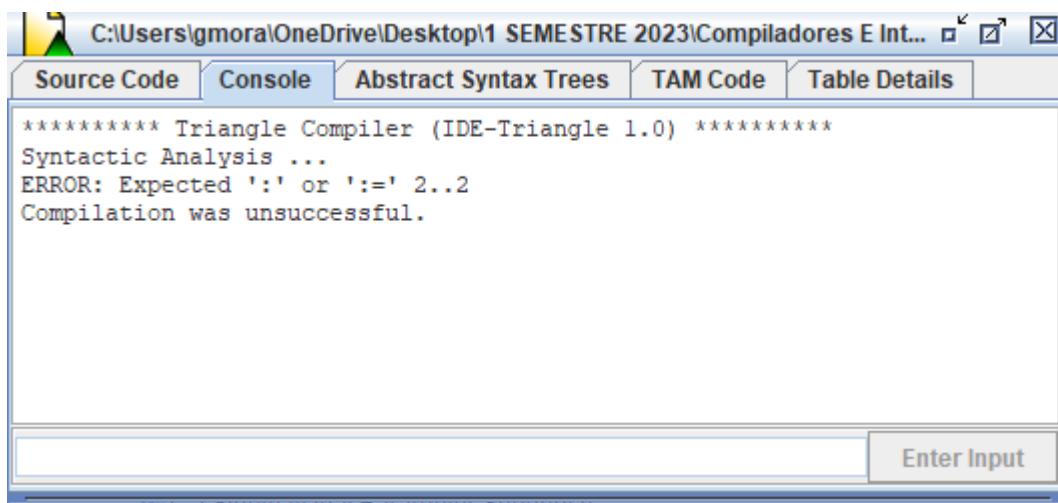
```

let
    var j 'c'; ! falta ":" o ":="
    var i : Integer;
    var y : Integer
in
    nil
end

```

Below the code, an error message is displayed: '! Error: falta ":" o ":=' entre j y 'c''. At the bottom of the window, it says 'Line: 10, Position: 1'.

Resultados



The screenshot shows the 'Console' tab of the IDE. The output is as follows:

```

***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
ERROR: Expected ':' or ':=' 2..2
Compilation was unsuccessful.

```

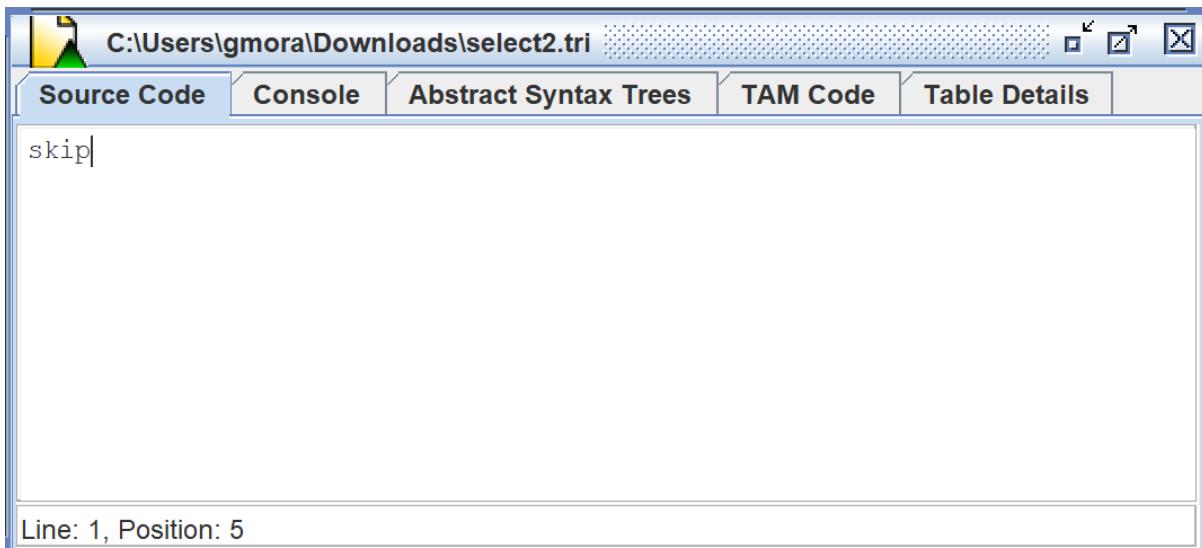
At the bottom right of the console window, there is a button labeled 'Enter Input'.

11.2 Pruebas del Análisis Léxico

El propósito de las pruebas que se presentan es verificar si el compilador es capaz de reconocer y procesar correctamente los diferentes tokens utilizados durante la compilación de los archivos. Estas pruebas no están diseñadas para demostrar la funcionalidad real del programa, sino para comprobar si el compilador reconoce correctamente la naturaleza del token y lo acepta o rechaza según su estructura. En

consecuencia, se mostrará únicamente el código y el resultado obtenido del compilador o el AST generado en caso de ser necesario.

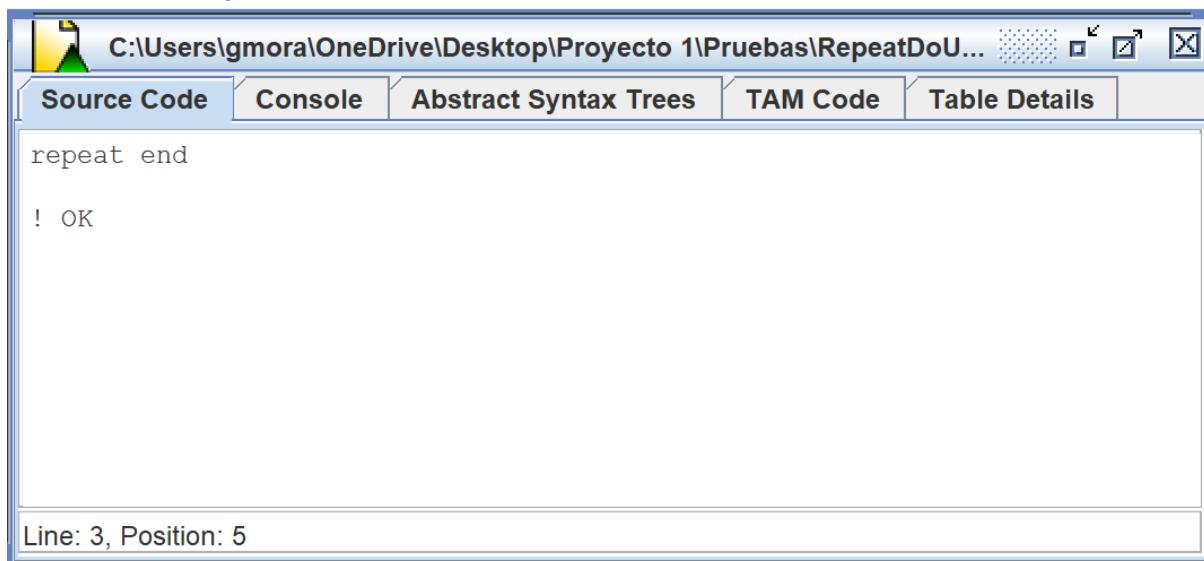
Prueba Skip



Resultado

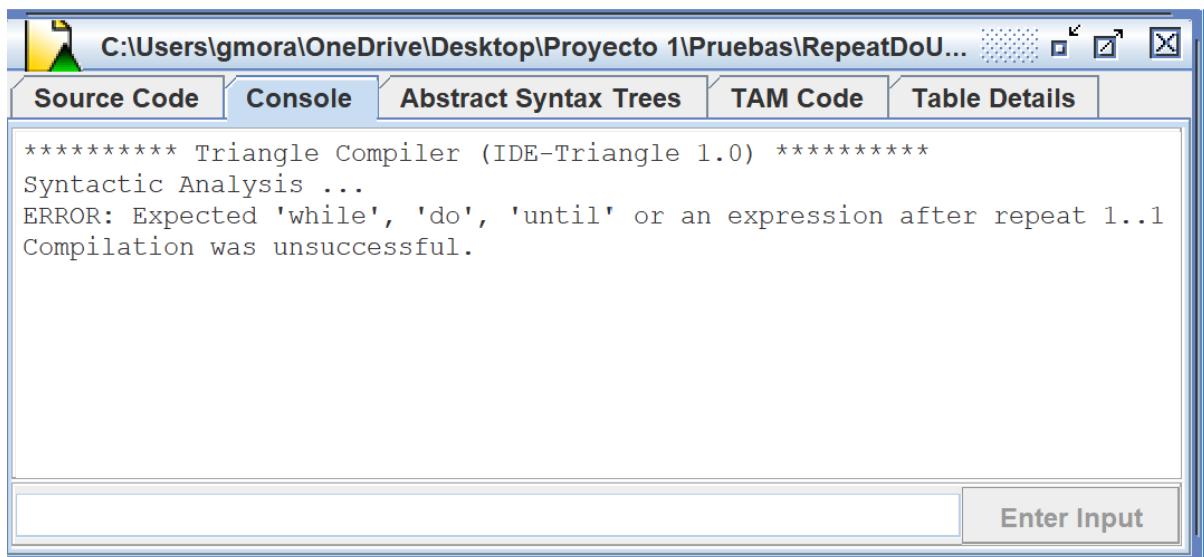


Prueba Repeat



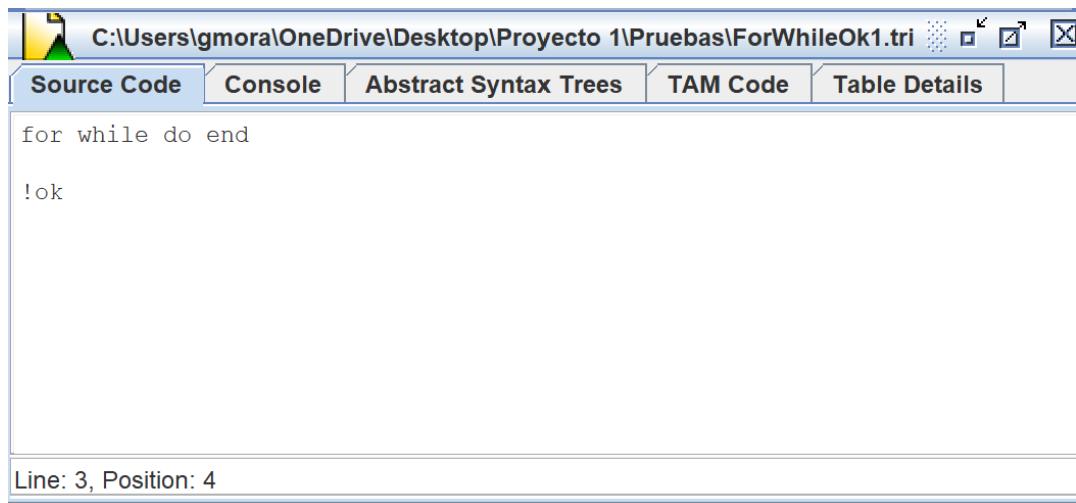
The screenshot shows a window titled 'C:\Users\gmora\OneDrive\Desktop\Proyecto 1\Pruebas\RepeatDoU...'. The 'Source Code' tab is selected, displaying the code 'repeat end'. Below the code, the message '! OK' is shown. A status bar at the bottom indicates 'Line: 3, Position: 5'.

Resultado



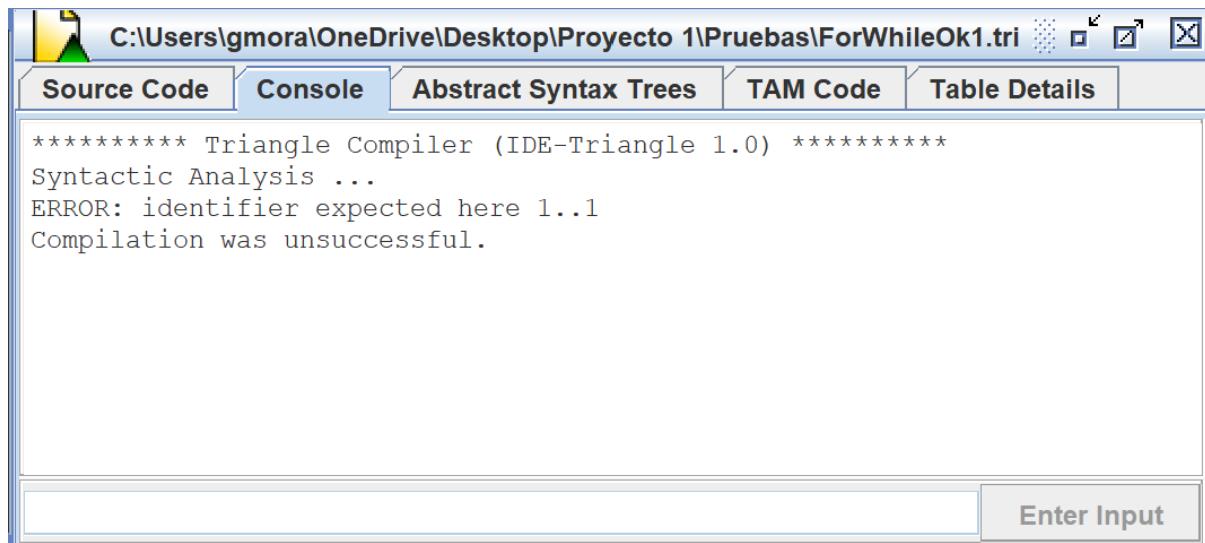
The screenshot shows a window titled 'C:\Users\gmora\OneDrive\Desktop\Proyecto 1\Pruebas\RepeatDoU...'. The 'Console' tab is selected, displaying the output of the compiler. The output includes the message '***** Triangle Compiler (IDE-Triangle 1.0) *****', 'Syntactic Analysis ...', 'ERROR: Expected 'while', 'do', 'until' or an expression after repeat 1..1', and 'Compilation was unsuccessful.' At the bottom right, there is an 'Enter Input' button.

Prueba For while do end



The screenshot shows the Triangle Compiler IDE interface. The title bar displays the file path: C:\Users\gmora\OneDrive\Desktop\Proyecto 1\Pruebas\ForWhileOk1.tri. The tabs at the top are Source Code, Console, Abstract Syntax Trees, TAM Code, and Table Details. The Source Code tab is selected, showing the code 'for while do end' on the first line and '!ok' on the second line. A status bar at the bottom indicates 'Line: 3, Position: 4'.

Resultado



The screenshot shows the Triangle Compiler IDE interface. The title bar displays the file path: C:\Users\gmora\OneDrive\Desktop\Proyecto 1\Pruebas\ForWhileOk1.tri. The tabs at the top are Source Code, Console, Abstract Syntax Trees, TAM Code, and Table Details. The Console tab is selected, displaying the following error message:
***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
ERROR: identifier expected here 1..1
Compilation was unsuccessful.
A text input field labeled 'Enter Input' is visible at the bottom right.

12. Reflexión

La realización del proyecto representó dos retos principales para el grupo de trabajo, la comprensión del lenguaje y la modificación de un programa creado por terceras personas. Cada uno de estos retos trajo consigo situaciones y consideraciones a tomar en cuenta a futuro por el equipo en futuros trabajos.

El primer reto recae en que, para cumplir con las indicaciones del proyecto, se debían agregar, modificar y/o eliminar reglas del lenguaje Triangle lo que tenía implicaciones en el análisis sintáctico que debía lograr realizar el programa. Esto significó para el equipo tener que destinar tiempo de las etapas iniciales del proyecto para repasar y estudiar la sintaxis y léxico del lenguaje para asegurar un mejor control de las variaciones a generar. Para suerte del grupo, gracias a la materia repasada en las lecciones y los materiales adicionales, este proceso resultó mucho más fácil y rápido para los integrantes.

En cuanto a la segunda situación mencionada, la modificación de un código ajeno es algo con lo que pocos de los integrantes no tenían experiencia, por lo que el abordaje para el desarrollo del proyecto no siguió el mismo ritmo habitual. Antes de empezar con el trabajo era necesario familiarizarse con el código del programa y comprender, no sólo las distintas estructuras generadas por los creadores, sino que también era importante comprender la organización del proyecto y el flujo que seguían las clases y métodos implementados hasta el momento para entender las implicaciones que resultarían las adiciones a generar. Este punto, a pesar de que en un inicio consumió tiempo de trabajo del grupo, una vez se pasó esta etapa el proceso de trabajo fue más sencillo.

A pesar de estos dos desafíos, el grupo reconoce que representaron una oportunidad para conocer nuevas formas para trabajar y exponerse a circunstancias diferentes, que requieran de una manera alternativa a la que, hasta este punto de la carrera, muchos han tenido que explorar. De igual forma, permitió adquirir un mejor manejo del lenguaje Triangle y de los conceptos que hasta el momento se han estudiado en el curso.

13. Descripción resumida de las tareas realizadas por cada miembro del grupo de trabajo.

Tarea	Realizado por
Eliminar comando vacío	Celina Madrigal Murillo
Eliminar begin end, let in, if then else y while do	Celina Madrigal Murillo
Añadir skip	Celina Madrigal Murillo
Añadir let in end	Celina Madrigal Murillo
Añadir if then then else end	Celina Madrigal Murillo
Añadir repeat while do end	María José Porras Maroto y Gabriel Mora Estripí
Añadir repeat until do end	María José Porras Maroto
Añadir repeat do while end	María José Porras Maroto
Añadir repeat times do end	Gabriel Mora Estripí
Añadir for := .. do end	María José Porras Maroto
Añadir for := .. while end	María José Porras Maroto
Añadir for := .. until end	María José Porras Maroto
Modificar Declaration	Celina Madrigal Murillo
Añadir regla compound-Declaration	Celina Madrigal Murillo
Añadir regla Proc-Func	Celina Madrigal Murillo
Agregar regla Proc-Funcs	Gabriel Mora Estripí
Modificar Single Declaration	Celina Madrigal Murillo
Añadir a single-Declaration otra declaración de variable	Celina Madrigal Murillo
Eliminar la palabra reservada begin	Celina Madrigal Murillo
Añadir las nuevas palabras reservadas	Celina Madrigal Murillo, María José Porras Maroto y Gabriel Mora Estripí.
Añadir nuevos simbolos lexicos	Gabriel Mora Estripí
Generación de un archivo HTML	María José Porras Maroto
Generación de un archivo XML	Celina Madrigal Murillo

Documentación	Celina Madrigal Murillo, María José Porras Maroto y Gabriel Mora Etribí
---------------	--

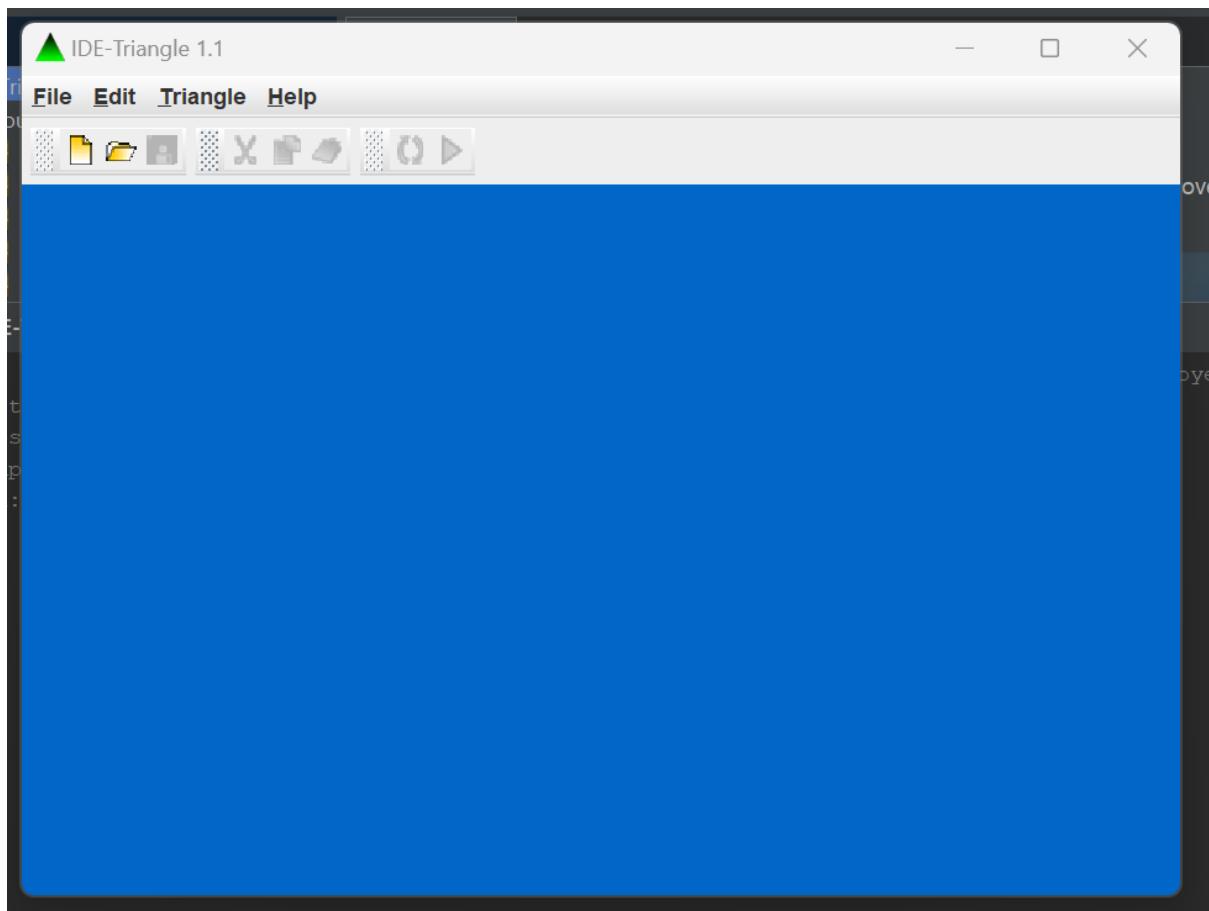
14. Indicaciones de uso del programa

Durante la edición de este programa, se empleó el entorno de desarrollo integrado (IDE) de Apache NetBeans, en su versión 17. Esta versión fue utilizada de forma constante y consistente a lo largo de todo el proyecto, tanto en la edición como en las pruebas y demás etapas del proceso. Por esta razón, se recomienda encarecidamente su uso para compilar y ejecutar el programa sin contratiempos.

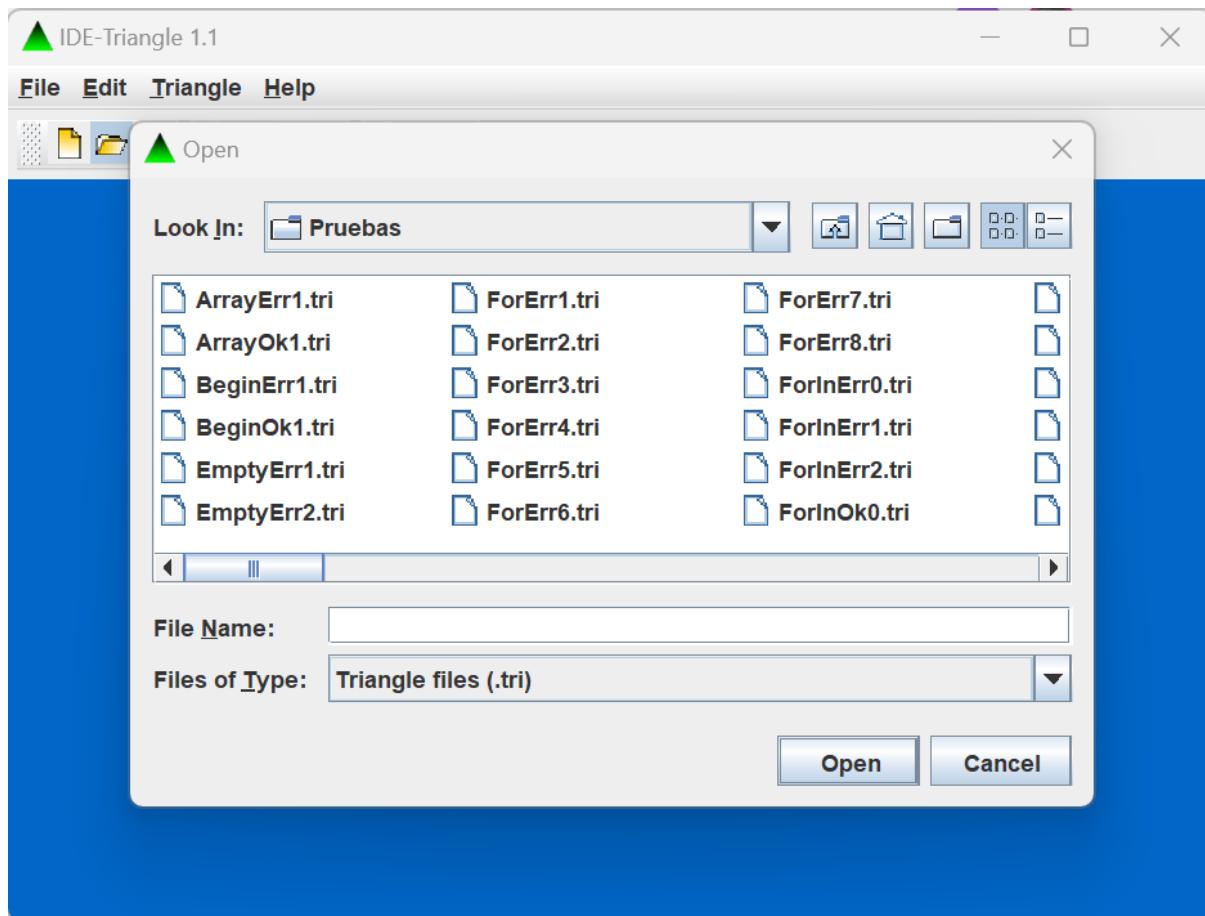
Cabe destacar que, para evitar errores inesperados durante la ejecución, es necesario llevar a cabo la descompresión del archivo .zip del proyecto con el programa WinRAR. De lo contrario, pueden surgir problemas imprevistos que podrían afectar el correcto funcionamiento del programa.

13.1 Cómo compilar el programa.

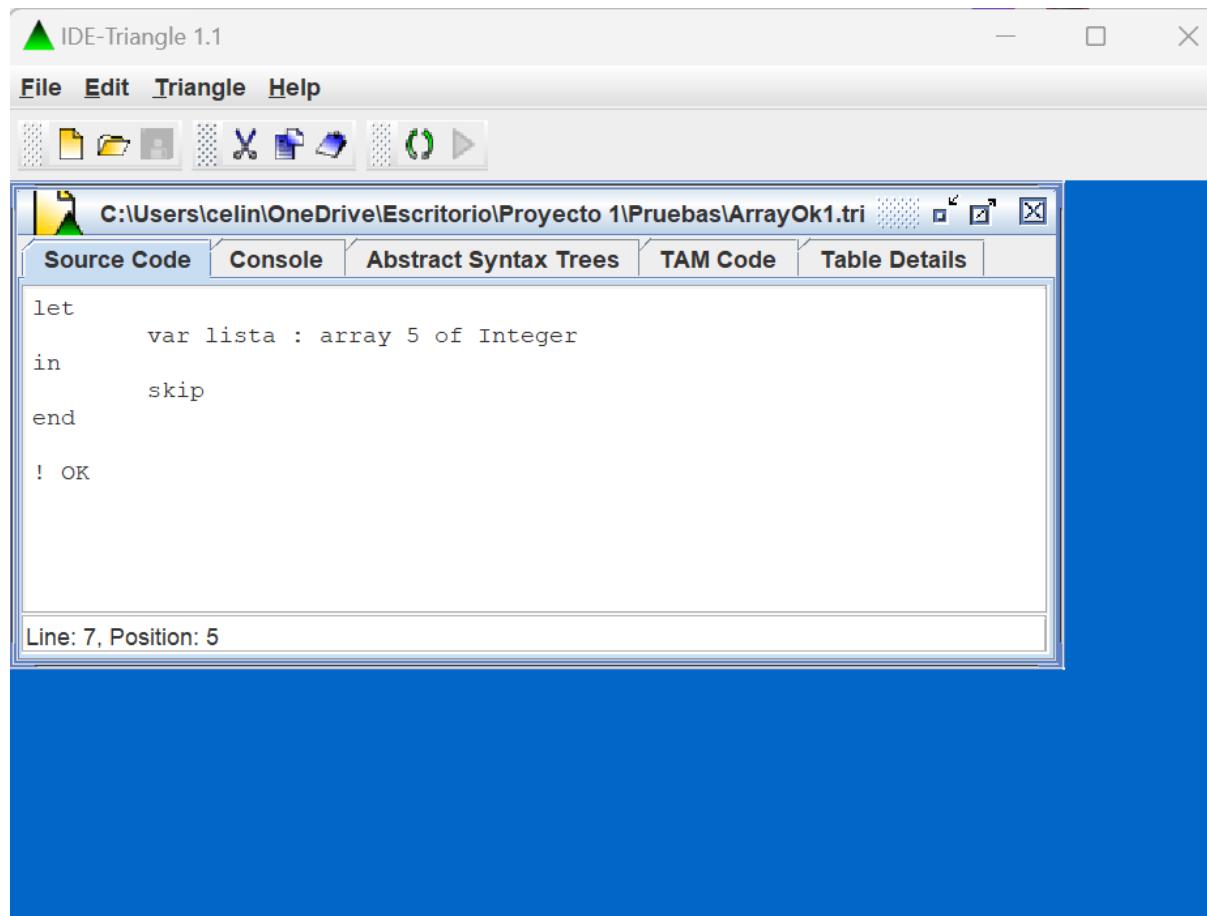
Utilizando el IDE de Apache NetBeans 17 abrimos el proyecto, damos click derecho y damos click en Run. Nos debería aparecer la siguiente pantalla:



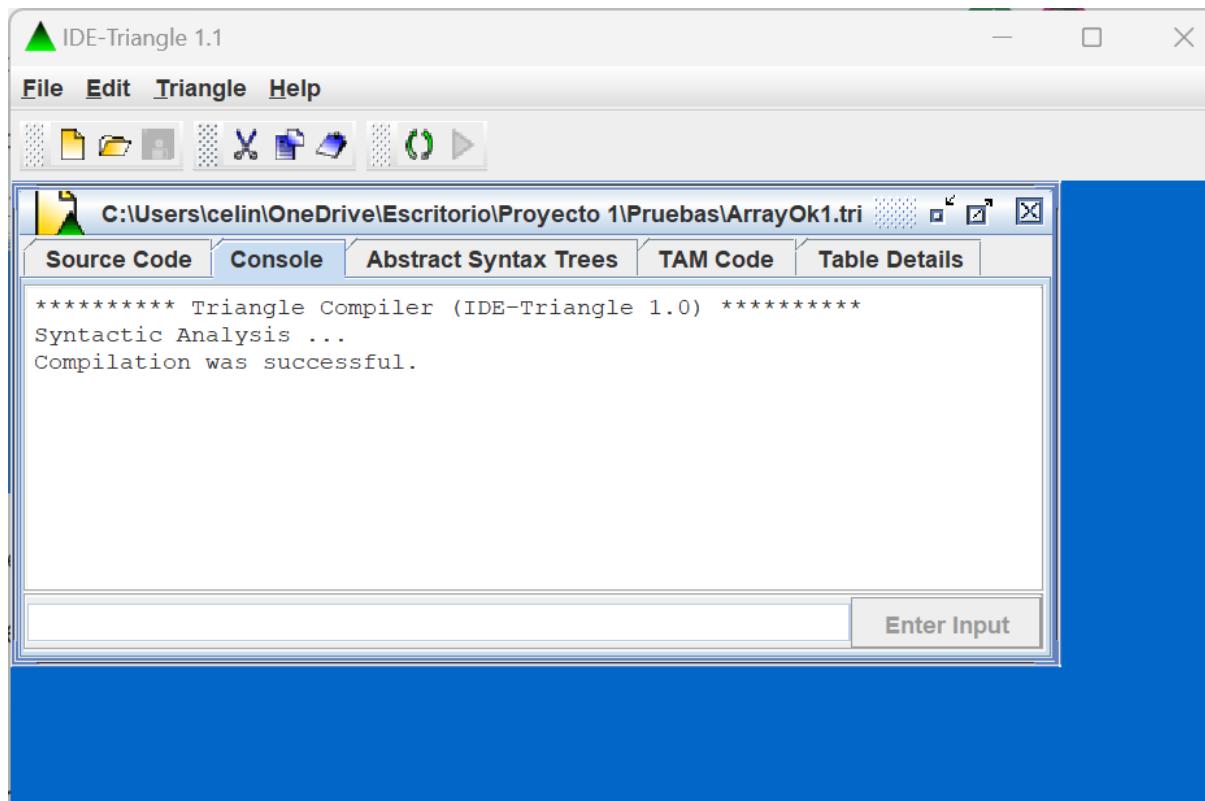
Una vez en esta pantalla le damos al folder amarillo para buscar un archivo de prueba



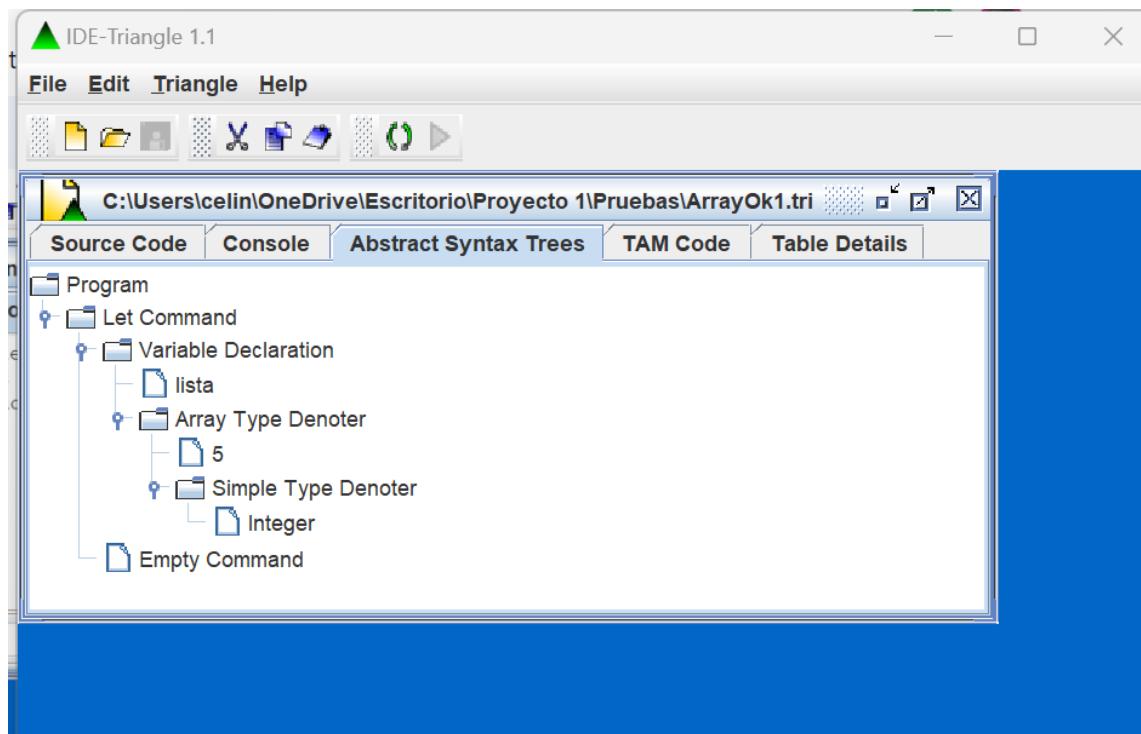
Una vez seleccionada la prueba deseada le damos open y nos debería de aparecer de la siguiente manera:



Para correr la prueba le damos a las flechas verdes en la parte de arriba. Al realizar esto nos debería aparecer la siguiente pantalla con un mensaje indicando el resultado de la prueba



Para ver el resultado del árbol le damos a Abstract Syntax Trees



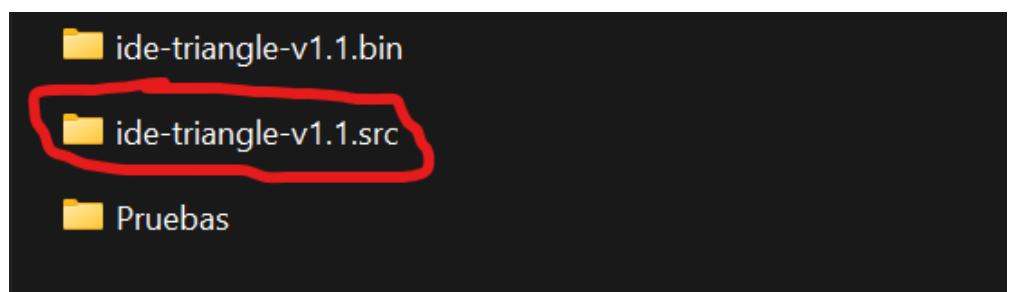
Para ver el Archivo HTML y XML generados vamos a la carpeta donde se encuentra el archivo .tri de la prueba realizada.

Nombre	Estado	Fecha de modificación	Tipo	Tamaño
ArrayErr1.tri	✓	1/5/2023 18:46	Archivo TRI	1 KB
ArrayOk1	✓	1/5/2023 18:53	Microsoft Edge HTM...	1 KB
ArrayOk1.tri	✓	1/5/2023 18:46	Archivo TRI	1 KB
ArrayOk1	✓	1/5/2023 18:53	Archivo de origen XML	1 KB

13.2 Cómo ejecutar el programa.

Para ejecutar el programa tenemos 2 opciones:

1. Como fue mencionado en el punto anterior, podemos ir al IDE de Apache NetBeans 17, abrimos el proyecto, damos click derecho y damos click en Run.
2. También podemos ejecutarlo mediante el archivo IDE-Triangle.jar, el cual lo encontramos siguiendo estos pasos:
 - En la carpeta del proyecto abrimos el .src



- Luego abrimos la carpeta llamada dist

build	✓	1/5/2023 18:36	Carpeta de archivos
dist	✓	1/5/2023 18:37	Carpeta de archivos
nbproject	✓	30/4/2023 12:03	Carpeta de archivos
src	✓	30/4/2023 12:03	Carpeta de archivos
test	✓	23/4/2023 21:56	Carpeta de archivos
build	✓	23/4/2023 16:56	Archivo de origen XML
manifest.mf	✓	23/4/2023 16:56	Archivo MF

- En esta carpeta encontraremos el archivo IDE-Triangle.jar el cual para ejecutarlo solo tendremos que dar doble click y ya podremos hacer uso del proyecto



15. Anexos

Nueva sintaxis y léxico del lenguaje Triángulo extendido tras modificaciones

Program	::= Command
Command	::= single-Command Command ; single-Command
single-Command	::= V-name := Expression Identifier (Actual-Parameter-Sequence) "skip" "let" Declaration in single-Command "if" Expression "then" Command (" " Expression Command) "else" Command "end" "repeat" "while" Expression "do" Command "end" "repeat" "until" Expression "do" Command "end" "repeat" "do" Command "while" Expression "end" "repeat" "do" Command "until" Expression "end" "repeat" Expression "times" "do" Command "end" "for" Identifier ":" Expression .. Expression "do" Command "end" "for" Identifier ":" Expression .. Expression "while" Expression "do" Command "end" "for" Identifier ":" Expression .. Expression "until" Expression "do" Command "end"
Expression	::= secondary-Expression "let" Declaration "in" Expression "if" Expression "then" Expression "else" Expression
secondary-Expression	::= primary-Expression secondary-Expression Operator primary-Expression
primary-Expression	::= Integer-Literal Character-Literal V-name Identifier (Actual-Parameter-Sequence) Operator primary-Expression (Expression) { Record-Aggregate } [Array-Aggregate]
Record-Aggregate	::= Identifier ~ Expression Identifier ~ Expression , Record-Aggregate
Array-Aggregate	::= Expression Expression , Array-Aggregate
V-name	::= Identifier V-name . Identifier V-name [Expression]
Declaration	::= single-Declaration Declaration ; compound-Declaration

```

single-Declaration ::= "const" Identifier ~ Expression
                    | "var" Identifier : Type-denoter
                    | "var" Identifier ":" Expression
                    | "proc" Identifier ( Formal-Parameter-Sequence ) ~
                      Command end
                    | func Identifier ( Formal-Parameter-Sequence )
                      : Type-denoter ~ Expression
compound-Declaration ::= single-Declaration
                      | "rec" Proc-Funcs "end"
                      | "private" Declaration "in" Declaration "end"
Proc-Func ::= "proc" Identifier "(" Formal-Parameter-Sequence ")"
              ":" Command "end"
            | "func" Identifier "(" Formal-Parameter-Sequence ")"
              ":" Type-denoter "~" Expression
Proc-Funcs ::= Proc-Func ("|" Proc-Func)*
Formal-Parameter-Sequence ::= proper-Formal-Parameter-Sequence
proper-Formal-Parameter-Sequence ::= Formal-Parameter
                                  | Formal-Parameter , proper-Formal-Parameter-Sequence
Formal-Parameter ::= Identifier : Type-denoter
                   | var Identifier : Type-denoter
                   | proc Identifier ( Formal-Parameter-Sequence )
                   | func Identifier ( Formal-Parameter-Sequence )
                     : Type-denoter
Actual-Parameter-Sequence ::= proper-Actual-Parameter-Sequence
proper-Actual-Parameter-Sequence ::= Actual-Parameter
                                  | Actual-Parameter , proper-Actual-Parameter-Sequence
Actual-Parameter ::= Expression

```

```

Program ::= (Token|Comment|Blank)*
Token ::= Integer-Literal|Character-Literal|Identifier|Operator|
         array|const|do|else|end|
         func|if|in|let|of|proc|record|
         then|type|var|while|repeat|for|until|from|
         package| private| rec| select| skip| times| when|
         .|:|;|,|:=|~|(|)|[]|{}|||$|...
Integer-Literal ::= Digit Digit*
Character-Literal ::= 'Graphic'
Identifier ::= Letter(Letter|Digit)*
Operator ::= Op-character Op-Character*
Comment ::= ! Graphic* end-of-line
Blank ::= space|tab|end-of-line
Graphic ::= Letter|Digit|Op-character|space|tab|.|:|;|,|~|(|)|[]|{}|_||||!|`|^|#$| |
Letter ::= a|b|c|d|e|f|g|h|i|j|k|l|m|
          n|o|p|q|r|s|t|u|v|w|x|y|z|
          A|B|C|D|E|F|G|H|I|J|K|L|M|
          N|O|P|Q|R|S|T|U|V|W|X|Y|Z
Digit ::= 0|1|2|3|4|5|6|7|8|9
Op-character ::= +|-|*|/|=|<|>|\\|&|@|^|?

```

16. Referencias

Pérez, L. (2005). IDE-Triangle. Guía de Implementación Rápida [Archivo PDF]. Universidad Latina de Costa Rica

Ramírez, D. (2018). Manual para integración del IDE y el compilador de Δ. IDE-Triangle – Integración con el compilador de Δ [Archivo PDF]. Tecnológico de Costa Rica

Trejos, I (2023) Casos de prueba. Tecnológico de Costa Rica

Watt D. y Brown D. (2000). *Programming Language Processors in Java*. Pearson Education