



**Instituto Tecnológico de Costa Rica**  
**Escuela de Ingeniería en Computación**

**Curso:** IC-5701 Compiladores e intérpretes

**1° semestre 2023**

**Proyecto II**

**Estudiantes:**

Celina Madrigal Murillo - 2020059364

Gabriel Mora Estribí - 2019048848

María José Porras Maroto - 2019066056

**Profesor:**

Ignacio Trejos Zelaya

**Grupo:** 2

**Fecha de entrega:**

Viernes 02 de junio del 2023

# Índice

1. Comprobación de los tipos para todas las variantes de los comandos repeat ... end	4
1.1 visitWhileCommand	4
1.2 visitUntilCommand	5
1.3 visitRepeatDoWhileCommand	6
1.4 visitRepeatDoUntilCommand	7
2. Manejo de alcance, tipo y protección de for_:=_._do_end y sus variantes condicionadas (while y until)	9
2.1 visitForBecomesAST	9
2.2 visitRepeatForWhile	10
2.3 visitRepeatForUntil	10
3. Procesamiento de var Id := Exp	11
4. Validación de la unicidad de nombres de parámetros en las declaraciones	12
5. Procesamiento de la declaración compuesta private	12
6. Procesamiento de la declaración compuesta rec	13
7. Nuevas rutinas de análisis contextual	16
7.1 VisitWhileCommand	16
7.2 VisitUntilCommand	16
7.3 VisitDoWhileCommand	17
7.4 VisitRepeatDoWhileCommand	17
7.5 VisitVarDeclarationBecomes	18
7.6 VisitDoUntilCommand	18
7.7 VisitRepeatDoUntilCommand	19
7.8 VisitForBecomesCommand	19
7.9 VisitForBecomesAST	19
7.1.1 VisitRepeatForWhile	20
7.1.2 VisitRepeatForUntil	20
7.1.3 VisitDotDCommandAST	20
8. Lista de nuevos errores contextuales detectados	21
9. Plan de pruebas	21
9.1 Array	21
9.2 For	24
9.3 For Until	27
9.4 For while	29
9.5 If	32
9.6 Let	34
9.7 Private	37

9.8 Rec	39
9.9 Repeat Do Until	42
9.10 Repeat Do While	45
9.11 Repeat Until Do	47
9.12 Repeat While Do	50
9.13 Skip	52
9.14 Var	55
9.15 procParamProc	57
10. Discusión y análisis de los resultados obtenidos	60
11. Conclusiones	60
12. Reflexion	60
13. Descripción resumida de las tareas realizadas por cada miembro del grupo de trabajo	61
14. Cómo debe compilarse el programa	62
15. Cómo debe ejecutarse el programa	65
16. Referencias	67

## 1. Comprobación de los tipos para todas las variantes de los comandos repeat ... end

### 1.1 visitWhileCommand

El flujo de acción que sigue el método visitWhileCommand es el siguiente:

1. Primero, se visita el subárbol ast.E del nodo WhileCommand para evaluar la expresión del bucle while.
2. A continuación, se verifica si el tipo resultante de ast.E es igual al tipo booleanType definido en StdEnvironment. Si no es así, se genera un error.
3. Luego, se comprueba si el objeto o pasado al método no es nulo. Esto indica que el comando while está dentro de un comando repeat. Si es así, se realiza lo siguiente:
  - a. Se crea una nueva instancia de RepeatDeclaration llamada repeat.
  - b. Se abre el scope.
  - c. Si el comando repeat tiene una variable de control (I), se agrega al idTable con el nombre y la instancia repeat. De lo contrario, se agrega una entrada vacía.
  - d. Se verifica si hay duplicados en el idTable. Si hay duplicados, se genera un error.
  - e. Luego, se visita el subárbol ast.C, que representa el cuerpo del comando while.
  - f. Finalmente, se cierra el scope.
4. Si el objeto o es nulo, significa que el comando while no está dentro de un comando repeat. En este caso, simplemente se visita el subárbol ast.C.

```

public Object visitWhileCommand(WhileCommand ast, Object o) {
    TypeDenoter eType = (TypeDenoter) ast.E.visit(v: this, o: null);
    if (!eType.equals(obj:StdEnvironment.booleanType))
        reporter.reportError(message:"Boolean expression expected here", tokenName: "", pos:ast.E.position);

    if(o != null){
        RepeatDeclaration repeat = new RepeatDeclaration(thePosition:dummyPos);
        idTable.openScope();
        if(((RepeatCommand) o).I != null)
            idTable.enter(id: ((RepeatCommand) o).I.spelling, attr:repeat);
        else
            idTable.enter(id: "", attr:repeat);

        if(repeat.duplicated)
            reporter.reportError(message:"identifier \"%\" already declared",
                tokenName: ((RepeatCommand) o).I.spelling, pos:((RepeatCommand) o).position);
        ast.C.visit(v: this, o: null);
        idTable.closeScope();
    }
    else
        ast.C.visit(v: this, o: null);

    return null;
}

```

## 1.2 visitUntilCommand

El flujo de acción que sigue el método visitUntilCommand es el siguiente:

1. Primero, se visita el subárbol aThis.I del nodo UntilCommand para evaluar la expresión de control del bucle until.
2. A continuación, se verifica si el tipo resultante de aThis.I es igual al tipo booleanType definido en StdEnvironment. Si no es así, se genera un error de informe.
3. Luego, se comprueba si el objeto o pasado al método no es nulo. Esto indica que el comando until está dentro de un comando repeat. Si es así, se realiza lo siguiente:
  - a. Se crea una nueva instancia de RepeatDeclaration llamada repeat.
  - b. Se abre el scope.
  - c. Si el comando repeat no tiene una variable de control (I), se agrega una entrada vacía al idTable. De lo contrario, se agrega la variable de control al idTable con el nombre y la instancia repeat.

- d. Se verifica si hay duplicados en el idTable. Si hay duplicados, se genera un error.
  - e. Luego, se visita el subárbol aThis.C, que representa el cuerpo del comando until.
  - f. Finalmente, se cierra el scope.
4. Si el objeto o es nulo, significa que el comando until no está dentro de un comando repeat. En este caso, simplemente se visita el subárbol aThis.C.

```

public Object visitUntilCommand(UntilCommand aThis, Object o) {
    TypeDenoter eType = (TypeDenoter) aThis.I.visit(v: this, o: null);
    if (!eType.equals(obj.StdEnvironment.booleanType)) {
        reporter.reportError(message: "Boolean expression expected here", tokenName: "", pos: aThis.I.position);
    }
    if (o != null) {
        RepeatDeclaration repeat = new RepeatDeclaration(thePosition: dummyPos);
        idTable.openScope();
        if ((RepeatUntilAST) o).I == null)
            idTable.enter(id: "", attr: repeat);
        else
            idTable.enter(id: ((RepeatUntilAST) o).I.spelling, attr: repeat);

        if (repeat.duplicated)
            reporter.reportError(message: "identifier \"%\" already declared",
                tokenName: ((RepeatUntilAST) o).I.spelling, pos: ((RepeatUntilAST) o).position);

        aThis.C.visit(v: this, o: null);
        idTable.closeScope();
    }
    else
        aThis.C.visit(v: this, o: null);
    return null;
}

```

### 1.3 visitRepeatDoWhileCommand

El flujo de acción que sigue el método visitRepeatDoWhileCommand es el siguiente:

1. Se crea una nueva instancia de RepeatDeclaration llamada repeat utilizando la posición dummyPos.
2. Se verifica si aThis.I no es nulo. Esto indica que el comando repeat tiene una variable de control. Si es así, se realiza lo siguiente:
  - a. Se abre el scope.

- b. Se agrega la variable de control aThis.I al idTable con el nombre y la instancia repeat.
  - c. Se verifica si hay duplicados en el idTable. Si hay duplicados, se genera un error.
  - d. Luego, se visita el subárbol aThis.C, que representa el cuerpo del bucle repeat...do.
  - e. Finalmente, se cierra el scope.
3. Si aThis.I es nulo, significa que el comando repeat no tiene una variable de control. En este caso, se realiza lo siguiente:
  - a. Se abre el scope.
  - b. Se agrega una entrada vacía al idTable.
  - c. Se visita el subárbol aThis.C, que representa el cuerpo del bucle repeat...do.
  - d. Finalmente, se cierra el scope.
4. A continuación, se visita el subárbol aThis.DoWhile, que representa la expresión de control del bucle do...while.

```
public Object visitRepeatDoWhileCommand(RepeatDoWhileAST aThis, Object o) {
    RepeatDeclaration repeat = new RepeatDeclaration(thePosition:dummyPos);
    if(aThis.I != null){
        idTable.openScope();
        idTable.enter(id: aThis.I.spelling, attr: repeat);
        if(repeat.duplicated)
            reporter.reportError(message:"identifier \"%\" already declared", tokenName: aThis.I.spelling, pos:aThis.position);

        aThis.C.visit(v: this, o: null);
        idTable.closeScope();
    }
    else{
        idTable.openScope();
        idTable.enter(id: "", attr: repeat);
        aThis.C.visit(v: this, o: null);
        idTable.closeScope();
    }

    aThis.DoWhile.visit(v: this, o: null);
    return null;
}
```

## 1.4 visitRepeatDoUntilCommand

El flujo de acción que sigue el método visitRepeatDoUntilCommand es el siguiente:

1. Se crea una nueva instancia de RepeatDeclaration llamada repeat utilizando la posición dummyPos.
2. Se verifica si aThis.I no es nulo. Esto indica que el comando repeat tiene una variable de control. Si es así, se realiza lo siguiente:
  - a. Se abre el scope.
  - b. Se agrega la variable de control aThis.I al idTable con el nombre y la instancia repeat.
  - c. Se verifica si hay duplicados en el idTable. Si hay duplicados, se genera un error.
  - d. Luego, se visita el subárbol aThis.C, que representa el cuerpo del bucle repeat...do.
  - e. Finalmente, se cierra el scope.
3. Si aThis.I es nulo, significa que el comando repeat no tiene una variable de control. En este caso, se realiza lo siguiente:
  - a. Se abre el scope.
  - b. Se agrega una entrada vacía al idTable.
  - c. Se visita el subárbol aThis.C, que representa el cuerpo del bucle repeat...do.
  - d. Finalmente, se cierra el scope.
4. A continuación, se visita el subárbol aThis.DoUntil, que representa la expresión de control del bucle do...until.



```

public Object visitRepeatDoUntilCommand(RepeatDoUntilAST aThis, Object o) {
    RepeatDeclaration repeat = new RepeatDeclaration(thePosition:dummyPos);
    if(aThis.I != null){
        idTable.openScope();
        idTable.enter(id: aThis.I.spelling, attr:repeat);
        if(repeat.duplicated)
            reporter.reportError(message:"identifier \"%\" already declared", tokenName: aThis.I.spelling, pos:aThis.position);

        aThis.C.visit(v: this, o: null);
        idTable.closeScope();
    }
    else{
        idTable.openScope();
        idTable.enter(id: "", attr:repeat);
        aThis.C.visit(v: this, o: null);
        idTable.closeScope();
    }

    aThis.DoUntil.visit(v: this, o: null);
    return null;
}

```

## 2. Manejo de alcance, tipo y protección de `for_:=__do_end` y sus variantes condicionadas (while y until)

### 2.1 visitForBecomesAST

El flujo de acción que sigue el método visitForBecomesAST es el siguiente:

1. Se llaman a los métodos visit en los objetos aThis.ForBecomes y aThis.E respectivamente.
2. Se abre el scope.
3. Se agrega la variable de control aThis.ForBecomes.I al idTable
4. Se verifica si hay duplicados en el idTable. Si hay duplicados, se genera un error.
5. Se llama al método visit en el objeto aThis.DoC. Esto visita el subárbol aThis.DoC, que representa el comando ejecutado en cada iteración del bucle for.
6. Se cierra el scope.

```

public Object visitForBecomesAST(ForBecomesAST aThis, Object o) {
    aThis.ForBecomes.visit(v: this, o: null); // expl
    aThis.E.visit(v: this, object: null); // exp2
    idTable.openScope(); // Se inicia el scope para el com y ids.

    idTable.enter(id: aThis.ForBecomes.I.spelling, attr: aThis.ForBecomes); // ingresa el id del for
    if (aThis.ForBecomes.duplicated)
        reporter.reportError(message: "identifier \"%s\" already declared", tokenName: aThis.I.spelling, pos: aThis.position);

    aThis.DoC.visit(v: this, o: null); // command
    idTable.closeScope(); // Se cierra el scope.
    return null;
}

```

## 2.2 visitRepeatForWhile

El flujo de acción que sigue el método visitRepeatForWhile es el siguiente:

1. Se llaman a los métodos visit en los objetos aThis.ForBecomes y aThis.E, respectivamente.
2. Se abre el scope.
3. Se agrega la variable de control del bucle repeat...for...while al idTable.
4. Se verifica si hay duplicados en el idTable. Si hay duplicados, se genera un error.
5. Se llaman a los métodos visit en los objetos aThis.WhileC.E y aThis.WhileC.C, respectivamente.
6. Se cierra el scope.

```

@Override
public Object visitRepeatForWhile(RepeatForWhile aThis, Object o) {
    aThis.ForBecomes.visit(v: this, o: null);
    aThis.E.visit(v: this, object: null);

    idTable.openScope();
    idTable.enter(id: aThis.ForBecomes.I.spelling, attr: aThis.ForBecomes);
    if (aThis.ForBecomes.duplicated)
        reporter.reportError(message: "identifier \"%s\" already declared", tokenName: aThis.I.spelling, pos: aThis.position);

    aThis.WhileC.E.visit(v: this, o: null);
    aThis.WhileC.C.visit(v: this, o: null);

    idTable.closeScope();
    return null;
}

```

## 2.3 visitRepeatForUntil

El flujo de acción que sigue el método visitRepeatForUntil es el siguiente:

1. Se llaman a los métodos visit en los objetos aThis.ForBecomes y aThis.E, respectivamente.
2. Se abre el scope.
3. Se agrega la variable de control del bucle repeat...for...until al idTable.
4. Se verifica si hay duplicados en el idTable. Si hay duplicados, se genera un error.
5. Se llama al método visit en el objeto aThis.UntilC, Esto visita el subárbol aThis.UntilC.
6. Se cierra el scope.

### 3. Procesamiento de var Id := Exp

El flujo de acción que sigue el procesamiento de var Id := Exp es el siguiente:

1. Se llama al método visit en el objeto ast.T. El resultado de la visita se asigna nuevamente a ast.T.
2. Se agrega la variable al idTable.
3. Se verifica si hay duplicados en el idTable. Si hay duplicados, se genera un error.

```
@Override
public Object visitVarDeclarationBecomes(VarDeclaration ast, Object o) {
    ast.T = (TypeDenoter) ast.T.visit(v: this, o: null);
    idTable.enter(id: ast.I.spelling, attr: ast);
    if (ast.duplicated)
        reporter.reportError(message: "identifier \"%\" already declared",
                               tokenName: ast.I.spelling, pos: ast.position);

    return null;
}
```

## 4. Validación de la unicidad de nombres de parámetros en las declaraciones

Para la validación de la unicidad de nombres de parámetros en las declaraciones se hace uso del método `enter` encontrado en *IdentificationTable.java*. Este método permite realizar una nueva entrada en la tabla de identificación para el identificador especificado y atributo. La nueva entrada pertenece al nivel actual duplicado se establece en `true` iff ya hay una entrada para el mismo identificador en el nivel actual.

```
public void enter (String id, Declaration attr) {

    IdEntry entry = this.latest;
    boolean present = false, searching = true;

    // Check for duplicate entry ...
    while (searching) {
        if (entry == null || entry.level < this.level)
            searching = false;
        else if (entry.id.equals(anObject.id) && !"".equals(anObject.id) ) {
            present = true;
            searching = false;
        } else
            entry = entry.previous;
    }

    attr.duplicated = present;
    // Add new entry ...
    entry = new IdEntry(id, attr, level: this.level, previous: this.latest);
    this.latest = entry;
}
```

## 5. Procesamiento de la declaración compuesta `private`

Para el procesamiento de `private` primero Se llama al método `pushPublic()` en el objeto `idTable`, esto indica que se está cambiando al alcance público, donde las declaraciones de variables y comandos son visibles para todo el programa. Luego se llama al método `visit` en el objeto `ast.D1`. Aquí se están visitando y procesando las declaraciones públicas. Después se llama al método `pushPrivate()` en el objeto `idTable`.

Esto indica que se está cambiando al alcance privado, donde las declaraciones de variables y comandos son visibles sólo dentro del bloque actual o del bloque privado. Luego se llama al método visit en el objeto ast.D2 Aquí se están visitando y procesando las declaraciones privadas. Y por último, se llama al método closePrivate() en el objeto idTable. Esto indica que se está cerrando el alcance privado actual, lo que implica que las variables y comandos definidos en ese alcance ya no son accesibles.

```
public Object visitPrivateDeclaration(PrivateDeclaration ast, Object o) {

    idTable.pushPublic();
    ast.D1.visit(v: this, o: null);

    idTable.pushPrivate();
    ast.D2.visit(v: this, o: null);

    idTable.closePrivate();
    return null;
}
```

## 6. Procesamiento de la declaración compuesta rec

La declaración "rec" se compone de un "Proc Funcs". Para analizar esta declaración, se sigue un proceso secuencial. Primero se visita la función "visitRecDeclaration", donde se examina su único hijo, que es una "SequentialDeclarationProcFuncs". Después de analizar la función "rec", se procede a examinar la función "Proc Funcs".

Es importante destacar que la función "Proc Funcs" es recursiva, lo que significa que se agregan dos casos base que detienen la recursión. Estos casos base son similares, uno se aplica a las instancias de "procs" y el otro a las instancias de "funcs". Los casos base indican que la recursión termina cuando los árboles han sido visitados completamente.

Después de los casos base, se encuentra la lógica del programa. En primer lugar, se determina si hay una secuencia de "procfuns" juntos. En caso afirmativo, se llama a la función "visitarProcFuncs". Esta función determina si el segundo hijo del árbol es un "proc" o un "func". Una vez determinado, se llama a las funciones "agregarProc" o "agregarFunc" y se visita el primer hijo del árbol.

Antes de explicar las funciones "agregarProc" y "agregarFunc", se mostrará el código de la función "visitarProcFuncs".

```
private void visitarProcFuncs(SequentialDeclarationProcFuncs aThis) {
    if (aThis.D2 instanceof FuncDeclaration) {
        agregarFunc((FuncDeclaration) aThis.D2);
        aThis.D1.visit(v: this, o: null);
    } else {
        if (aThis.D2 instanceof ProcDeclaration) {
            agregarProc((ProcDeclaration) aThis.D2);
            aThis.D1.visit(v: this, o: null);
        }
    }
}
```

Las funciones "agregarProc" y "agregarFunc" tienen un papel importante en el proceso. Su objetivo principal es agregar identificadores a una tabla, verificar si están duplicados y luego visitar los hijos del árbol correspondiente. Además, al finalizar, se marca la función o procedimiento como visitado.

Sin embargo, aún no hemos terminado de analizar la función "visitSequentialDeclarationProcFuncs". En caso de que el árbol sintáctico recibido no sea una secuencia de "procfuns", la función realiza otra acción. En este caso, se trata de agregar los "func" o "proc" de los hijos del árbol. Para lograr esto, se determina el tipo de instancia y se vuelven a utilizar las funciones "agregarProc" y "agregarFunc". Por último, se visitan los hijos del árbol. A continuación, se muestra el código completo de esta función.

```

@Override
public Object visitSequentialDeclarationProcFuncs(SequentialDeclarationProcFuncs aThis, Object o) {

    if (aThis.D2 instanceof ProcDeclaration && ((ProcDeclaration) aThis.D2).visited) {
        return null;
    }
    if (aThis.D2 instanceof FuncDeclaration && ((FuncDeclaration) aThis.D2).visited) {
        return null;
    }
    if (aThis.D1 instanceof SequentialDeclarationProcFuncs) {
        visitarProcFuncs(aThis);
    }
    else {
        if (aThis.D1 instanceof ProcDeclaration) {
            agregarProc((ProcDeclaration) aThis.D1);
        } else {
            if (aThis.D1 instanceof FuncDeclaration) {
                agregarFunc((FuncDeclaration) aThis.D1);
            }
        }
        if (aThis.D2 instanceof ProcDeclaration) {
            agregarProc((ProcDeclaration) aThis.D2);
        } else {
            if (aThis.D2 instanceof FuncDeclaration) {
                agregarFunc((FuncDeclaration) aThis.D2);
            }
        }
    }
}

aThis.D1.visit(v: this, o: null);
aThis.D2.visit(v: this, o: null);
return null;
}

```

En las funciones "agregarProc" y "agregarFunc", se implementó una estrategia para evitar la duplicación de identificadores. Primero, se agregaban los identificadores a una tabla y luego se verificaba si ya existían duplicados en ella. Para lograr que los identificadores fueran accesibles en otros contextos, se utilizó la recursión, una tabla de identificadores y se agregó el atributo "visitado" en las clases "ProcDeclaration" y "FuncDeclaration".

De esta manera, al agregar los identificadores a la tabla y verificar su duplicación, se garantizó que no hubiera nombres repetidos. Además, gracias a la recursión, se pudieron propagar los identificadores a través de diferentes niveles del árbol de análisis. La tabla de identificadores sirvió como una estructura de datos centralizada para almacenar y acceder a los identificadores. El atributo "visitado" en las clases "ProcDeclaration" y "FuncDeclaration" permitió marcar las funciones y procedimientos como visitados, lo que pudo ser útil en otros aspectos del análisis.

## 7. Nuevas rutinas de análisis contextual

Entre las nuevas rutinas que fueron adicionadas o modificadas para crear el análisis contextual esperado se encuentran en el Checker.java las siguientes funciones:

### 7.1 VisitWhileCommand

```
@Override
public Object visitWhileEndCommand(WhileEndCommand aThis, Object o) {
    TypeDenoter eType = (TypeDenoter) aThis.E.visit(v: this, o: null);
    if (!eType.equals(obj:StdEnvironment.booleanType)) {
        reporter.reportError(message:"Boolean expression expected here", tokenName: "", pos:aThis.E.position);
    }
    return null;
}
```

### 7.2 VisitUntilCommand

```
@Override
public Object visitUntilCommand(UntilCommand aThis, Object o) {
    TypeDenoter eType = (TypeDenoter) aThis.I.visit(v: this, o: null);
    if (!eType.equals(obj:StdEnvironment.booleanType)) {
        reporter.reportError(message:"Boolean expression expected here", tokenName: "", pos:aThis.I.position);
    }
    if(o != null){
        RepeatDeclaration repeat = new RepeatDeclaration(thePosition:dummyPos);
        idTable.openScope();
        if(((RepeatUntilAST) o).I == null)
            idTable.enter(id: "", attr: repeat);
        else
            idTable.enter(id: ((RepeatUntilAST) o).I.spelling, attr: repeat);

        if(repeat.duplicated)
            reporter.reportError(message:"identifier \"%\" already declared",
                tokenName: ((RepeatUntilAST) o).I.spelling, pos:((RepeatUntilAST) o).position);

        aThis.C.visit(v: this, o: null);
        idTable.closeScope();
    }
    else
        aThis.C.visit(v: this, o: null);
    return null;
}
```



## 7.3 VisitDoWhileCommand

```
@Override
public Object visitDoWhileCommand(DoWhileCommand aThis, Object o) {
    TypeDenoter eType = (TypeDenoter) aThis.E.visit(v: this, o: null);
    if (!eType.equals(obj:StdEnvironment.booleanType)) {
        reporter.reportError(message:"Boolean expression expected here", tokenName: "", pos:aThis.E.position);
    }
    return null;
}
```

## 7.4 VisitRepeatDoWhileCommand

```
@Override
public Object visitRepeatDoWhileCommand(RepeatDoWhileAST aThis, Object o) {
    RepeatDeclaration repeat = new RepeatDeclaration(thePosition:dummyPos);
    if(aThis.I != null){
        idTable.openScope();
        idTable.enter(id: aThis.I.spelling, attr: repeat);
        if(repeat.duplicated)
            reporter.reportError(message:"identifier \"%s\" already declared", tokenName: aThis.I.spelling, pos:aThis.position);

        aThis.C.visit(v: this, o: null);
        idTable.closeScope();
    }
    else{
        idTable.openScope();
        idTable.enter(id: "", attr: repeat);
        aThis.C.visit(v: this, o: null);
        idTable.closeScope();
    }

    aThis.DoWhile.visit(v: this, o: null);
    return null; }
}
```

## 7.5 VisitVarDeclarationBecomes

```

@Override
public Object visitVarDeclarationBecomes(VarDeclarationBecomes aThis, Object o) {
    TypeDenoter eType = (TypeDenoter) aThis.Ex.visit(v: this, o: null);

    aThis.E = eType;

    idTable.enter(id: aThis.I.spelling, attr: aThis);

    if (aThis.duplicated)
        reporter.reportError(message: "identifier \"%\" already declared",
            tokenName: aThis.I.spelling, pos: aThis.position);

    return null;
}

```

## 7.6 VisitDoUntilCommand

```

@Override
public Object visitDoUntilCommand(DoUntilCommand aThis, Object o) {
    TypeDenoter eType = (TypeDenoter) aThis.E.visit(v: this, o: null);
    if (!eType.equals(obj: StdEnvironment.booleanType)) {
        reporter.reportError(message: "Boolean expression expected here", tokenName: "", pos: aThis.E.position);
    }
    return null;
}

```

## 7.7 VisitRepeatDoUntilCommand

```

@Override
public Object visitRepeatDoUntilCommand(RepeatDoUntilAST aThis, Object o) {
    RepeatDeclaration repeat = new RepeatDeclaration(thePosition:dummyPos);
    if(aThis.I != null){
        idTable.openScope();
        idTable.enter(id: aThis.I.spelling, attr: repeat);
        if(repeat.duplicated)
            reporter.reportError(message:"identifier \"%\" already declared", tokenName: aThis.I.spelling, pos:aThis.position);

        aThis.C.visit(v: this, o: null);
        idTable.closeScope();
    }
    else{
        idTable.openScope();
        idTable.enter(id: "", attr: repeat);
        aThis.C.visit(v: this, o: null);
        idTable.closeScope();
    }

    aThis.DoUntil.visit(v: this, o: null);
    return null;
}

```

## 7.8 VisitForBecomesCommand

```

@Override
public Object visitForBecomesCommand(ForBecomesCommand aThis, Object o) {
    TypeDenoter eType = (TypeDenoter) aThis.E.visit(v: this, o: null);
    if (!eType.equals(obj:StdEnvironment.integerType))
        reporter.reportError(message:"Integer Expression expected here", tokenName: "", pos:aThis.E.position);

    return null;
}

```

## 7.9 VisitForBecomesAST

```

@Override
public Object visitForBecomesAST(ForBecomesAST aThis, Object o) {
    aThis.ForBecomes.visit(v: this, o: null);
    aThis.E.visit(v: this, object: null);
    idTable.openScope();

    idTable.enter(id: aThis.ForBecomes.I.spelling, attr:aThis.ForBecomes);
    if (aThis.ForBecomes.duplicated)
        reporter.reportError(message:"identifier \"%\" already declared", tokenName: aThis.I.spelling, pos:aThis.position);

    aThis.DoC.visit(v: this, o: null); // command
    idTable.closeScope();
    return null;
}

```

### 7.1.1 VisitRepeatForWhile

```

@Override
public Object visitRepeatForWhile(RepeatForWhile aThis, Object o) {
    aThis.ForBecomes.visit(v: this, o: null);
    aThis.E.visit(v: this, object: null);

    idTable.openScope();
    idTable.enter(id: aThis.ForBecomes.I.spelling, attr: aThis.ForBecomes);
    if (aThis.ForBecomes.duplicated)
        reporter.reportError(message: "identifier \"%\" already declared", tokenName: aThis.I.spelling, pos: aThis.position);

    aThis.WhileC.E.visit(v: this, o: null);
    aThis.WhileC.C.visit(v: this, o: null);

    idTable.closeScope();
    return null;
}

```

### 7.1.2 VisitRepeatForUntil

```

@Override
public Object visitRepeatForUntil(RepeatForUntil aThis, Object o) {
    aThis.ForBecomes.visit(v: this, o: null);
    aThis.E.visit(v: this, object: null);
    idTable.openScope();

    idTable.enter(id: aThis.ForBecomes.I.spelling, attr: aThis.ForBecomes);
    if (aThis.ForBecomes.duplicated)
        reporter.reportError(message: "identifier \"%\" already declared", tokenName: aThis.I.spelling, pos: aThis.position);

    aThis.UntilC.visit(v: this, o: null);
    idTable.closeScope();
    return null;
}

```

### 7.1.3 VisitDotDCommandAST

```

@Override
public Object visitDotDCommandAST(DotDCommand aThis, Object o) {
    TypeDenoter eType = (TypeDenoter) aThis.CLC.visit(v: this, o: null);
    if (!eType.equals(obj: StdEnvironment.integerType)) {
        reporter.reportError(message: "Integer expression expected here", tokenName: "", pos: aThis.CLC.position);
    }
    return null;
}

```

## 8. Lista de nuevos errores contextuales detectados

Caso	Error
repeat while <b>Exp</b> do Com end repeat until <b>Exp</b> do Com end repeat do Com while <b>Exp</b> end repeat do Com until <b>Exp</b> end	Mensaje de error por: <b>Exp</b> que debe ser de tipo Boolean.
for Id := <b>Exp1</b> .. <b>Exp2</b> do Com end	Mensaje de error en caso de que <b>Exp1</b> y <b>Exp2</b> no sean de tipo entero Mensaje de error en caso de que <b>Id</b> ya haya sido declarado.
for <b>Id</b> := Exp1 .. Exp2 do Com end	Mensaje de error en caso de que <b>Id</b> no sea entero, aparezca a la izquierda de una asignación, se pase como parámetro var o ya haya sido declarada.
for Id := Exp1 to Exp2 while <b>Exp3</b> do Com end for Id := Exp1 to Exp2 until <b>Exp3</b> do Com end	Mensaje de error en caso de que <b>Exp3</b> no sea de tipo booleano
if <b>Exp</b> then Com1 (   <b>Exp1</b> then Com1 ) * else Com2 end	Mensaje de error en caso de que las expresiones <b>Exp</b> y <b>Exp1</b> no sean de tipo booleano

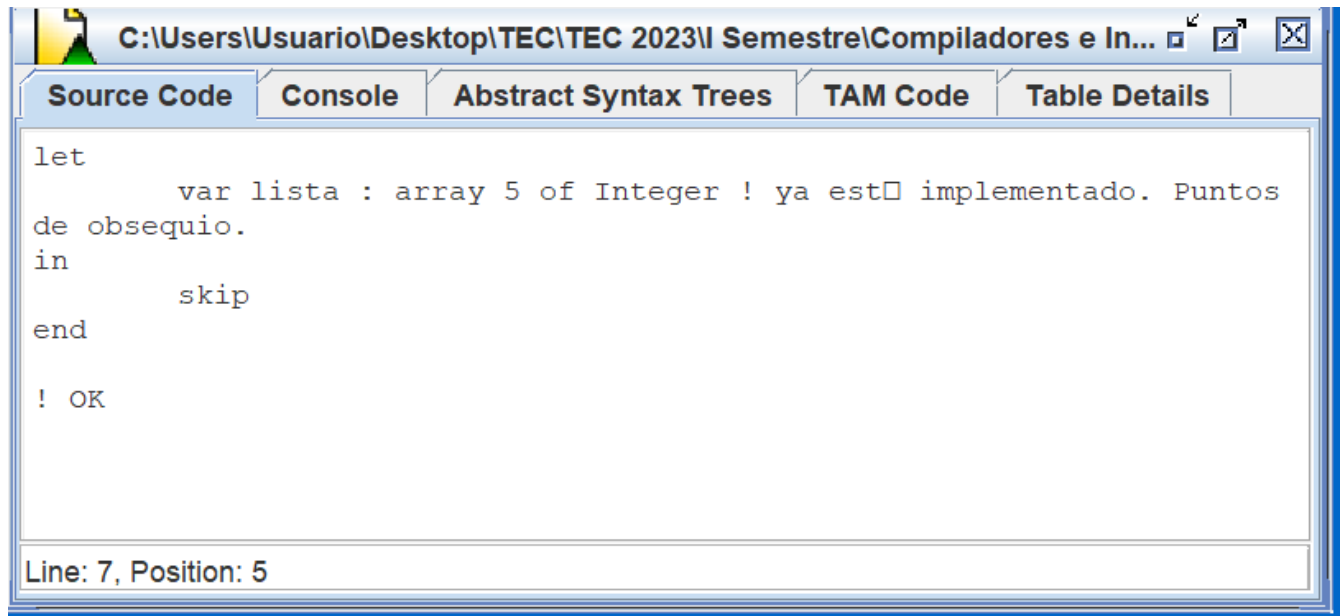
## 9. Plan de pruebas

### 9.1 Array

#### Objetivo

El objetivo de esta prueba es ver el funcionamiento contextual correcto esperado por el comando array. En estas pruebas se espera representar el funcionamiento del compilador ante un caso de prueba correcto y uno que presente un error.

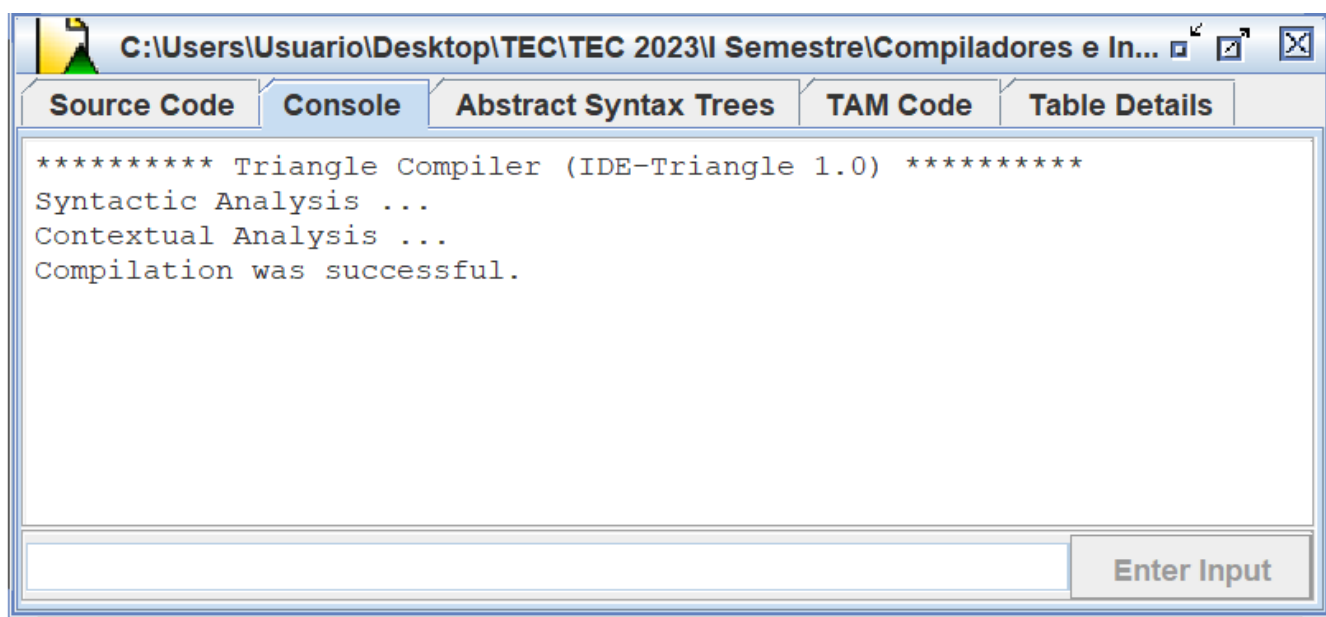
## Diseño de la prueba Correcta



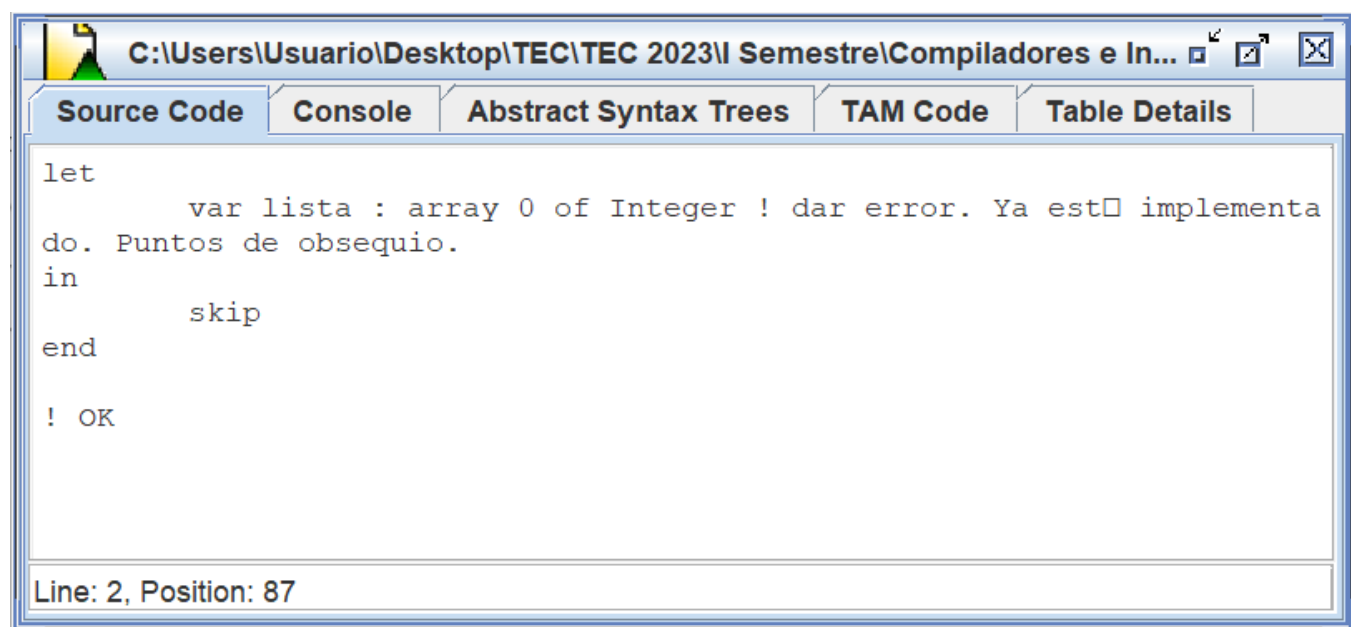
### Resultados esperados

El resultado esperado es que la prueba sea compilada de manera satisfactoria generando los respectivos archivos de salida del proyecto anterior.

### Resultados observados



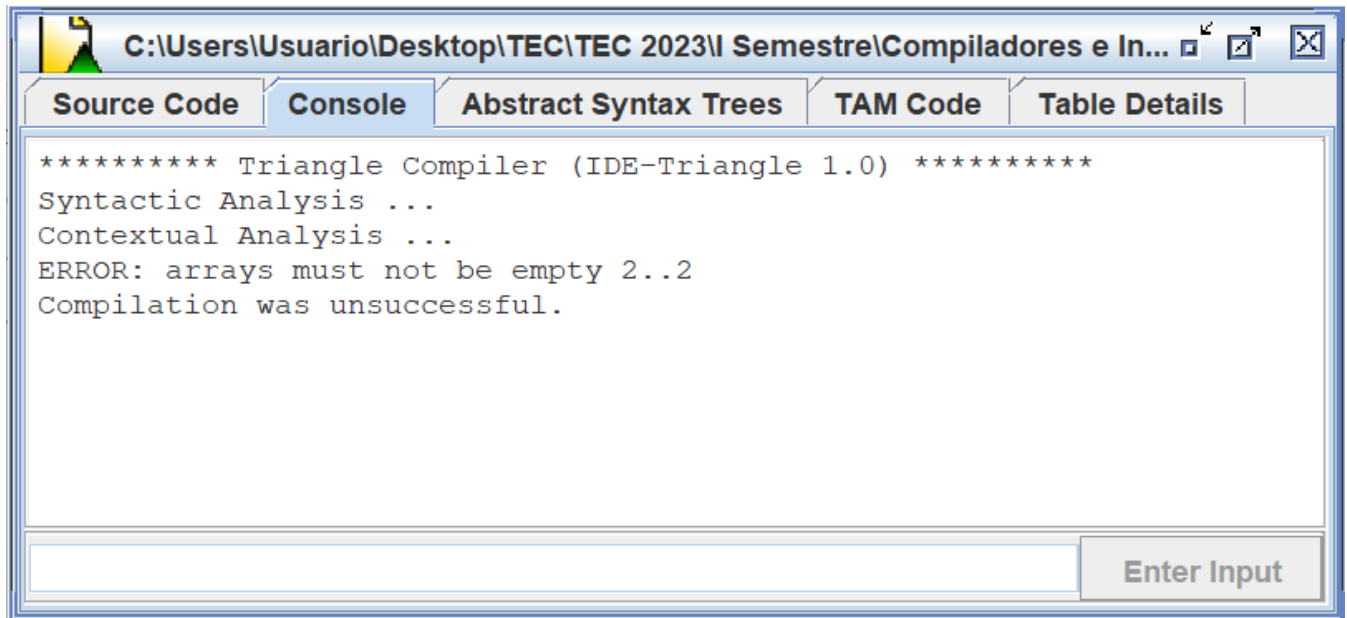
### Diseño de la prueba Errónea



### Resultados esperados

El resultado esperado es que la prueba a la hora de compilar es que encuentre el error de contexto y que lo indique la pestaña de consola.

## Resultados observados



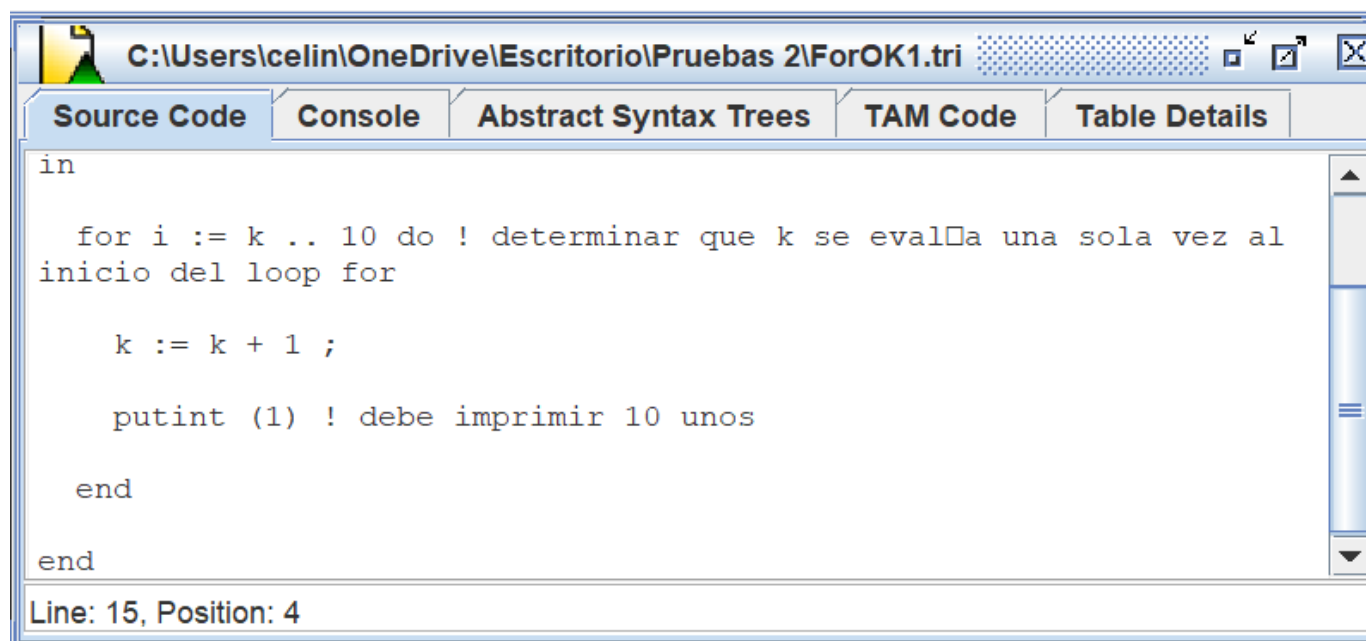
### 9.2 For

#### Objetivo

El objetivo de esta prueba es ver el funcionamiento contextual correcto esperado por el comando for. En estas pruebas se espera representar el funcionamiento del compilador ante un caso de prueba correcto y uno que presente un error.

#### Diseño de la prueba Correcta





```

in

  for i := k .. 10 do ! determinar que k se evalúa una sola vez al
  inicio del loop for

    k := k + 1 ;

    putint (1) ! debe imprimir 10 unos

  end

end

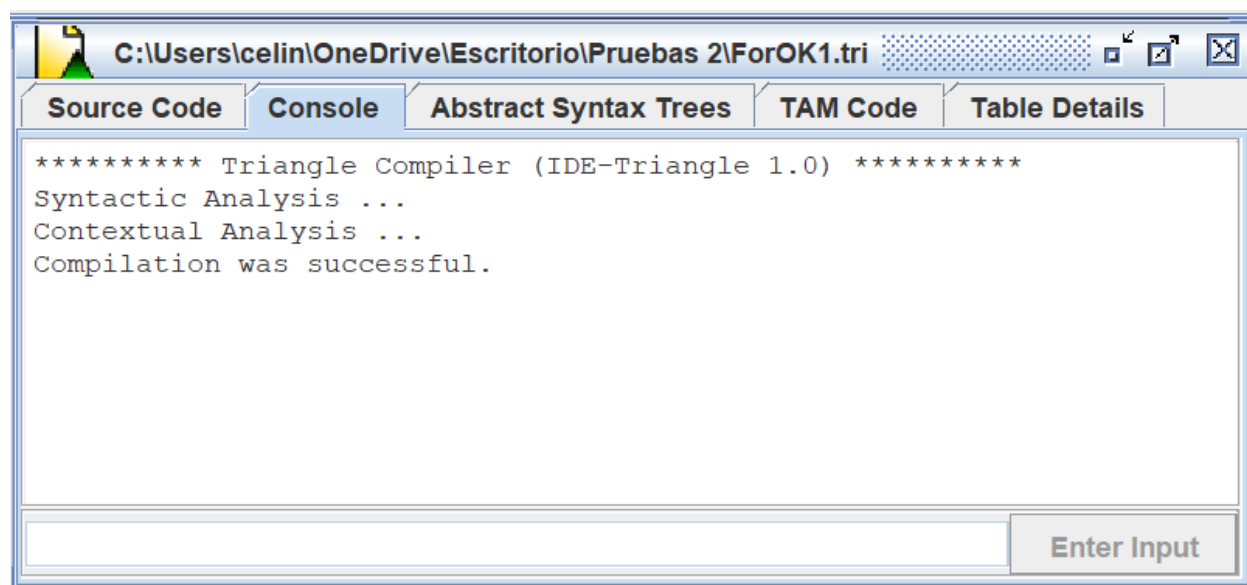
```

Line: 15, Position: 4

### Resultados esperados

El resultado esperado es que la prueba a la hora de compilar es que encuentre el error de contexto y que lo indique la pestaña de consola.

### Resultados observados



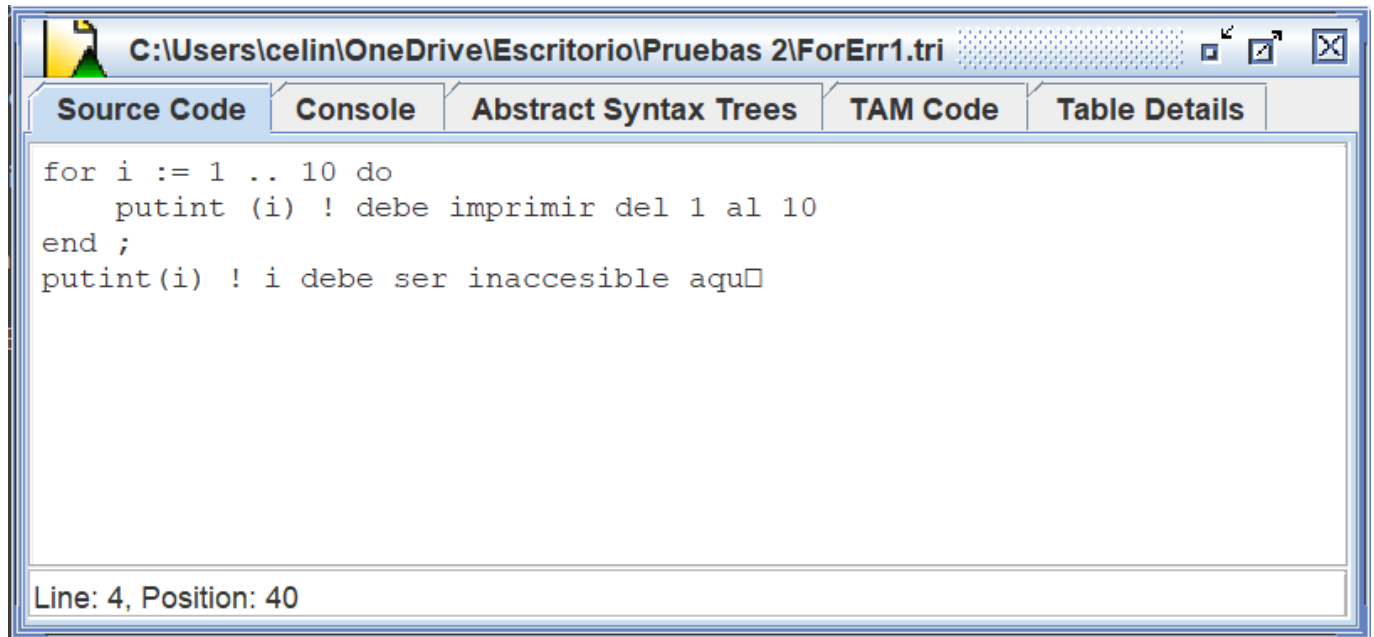
```

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

```

Enter Input

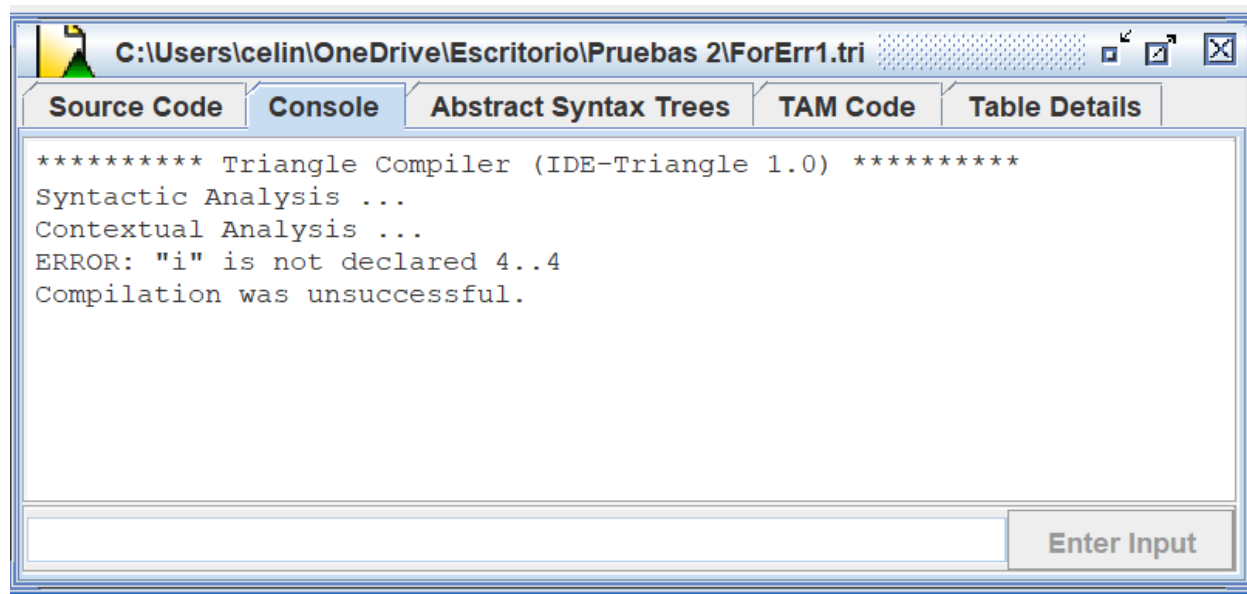
## Diseño de la prueba Errónea



### Resultados esperados

El resultado esperado es que la prueba a la hora de compilar es que encuentre el error de contexto y que lo indique la pestaña de consola.

### Resultados observados

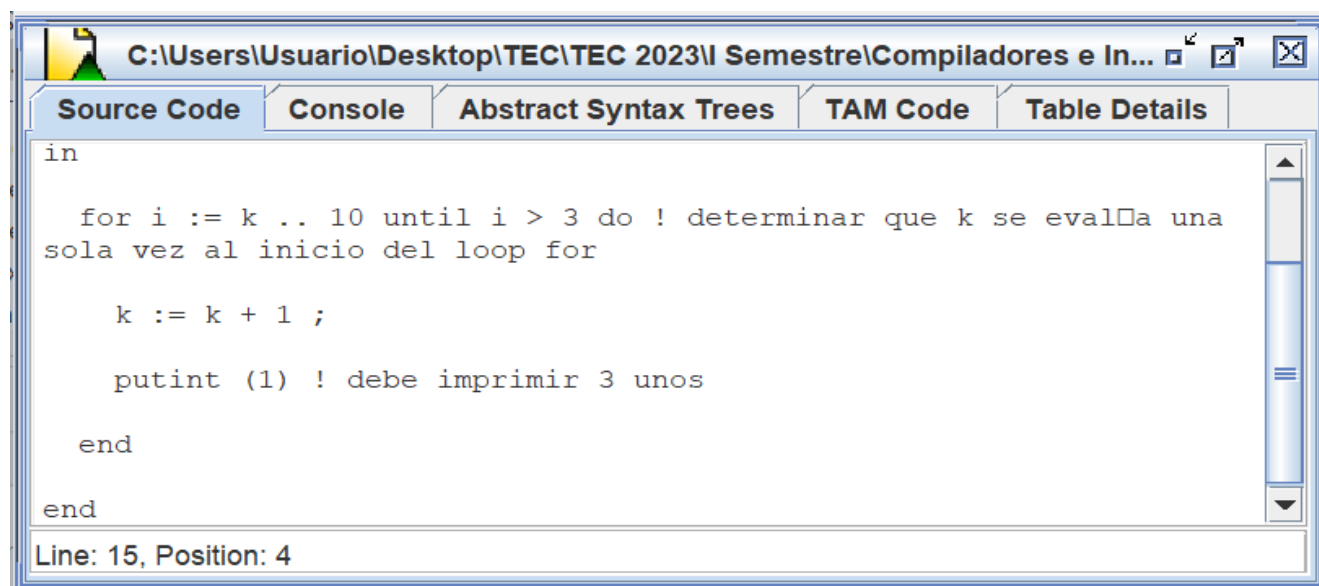


### 9.3 For Until

#### Objetivo

El objetivo de esta prueba es ver el funcionamiento contextual correcto esperado por el comando for until. En estas pruebas se espera representar el funcionamiento del compilador ante un caso de prueba correcto y uno que presente un error.

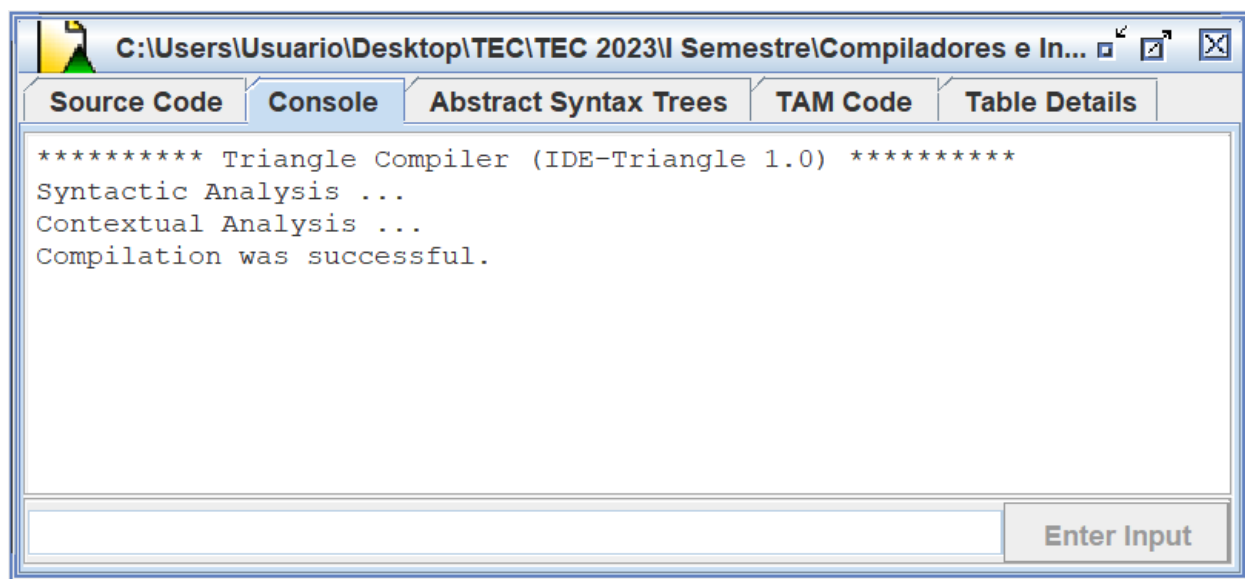
#### Diseño de la prueba Correcta



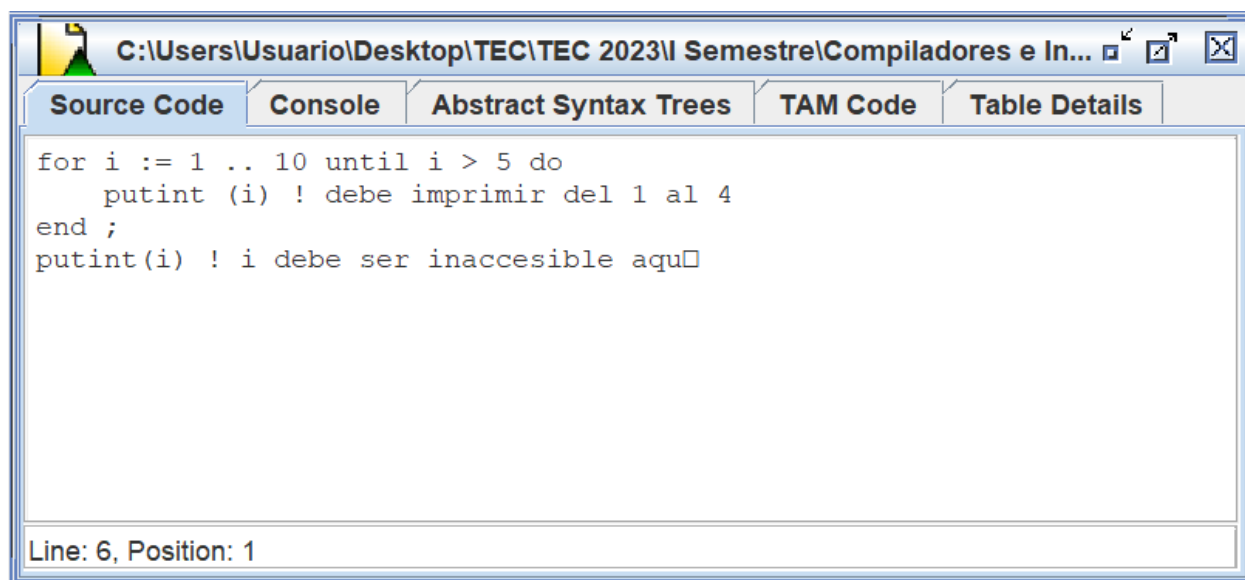
## Resultados esperados

El resultado esperado es que la prueba sea compilada de manera satisfactoria generando los respectivos archivos de salida del proyecto anterior.

## Resultados observados



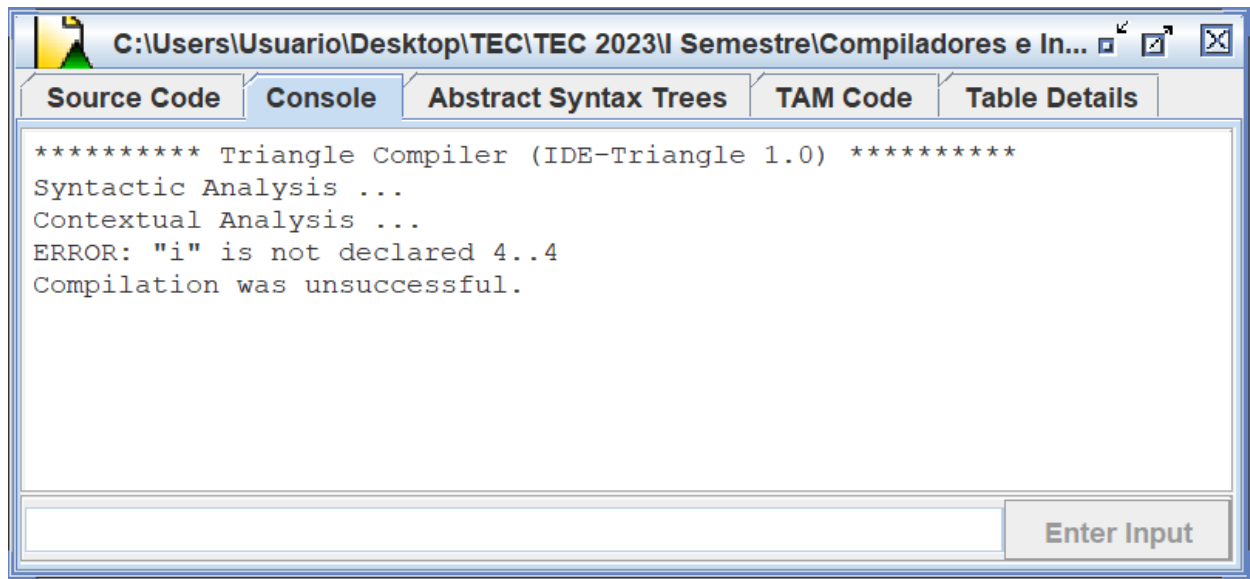
## Diseño de la prueba Errónea



## Resultados esperados

El resultado esperado es que la prueba a la hora de compilar es que encuentre el error de contexto y que lo indique la pestaña de consola.

## Resultados observados

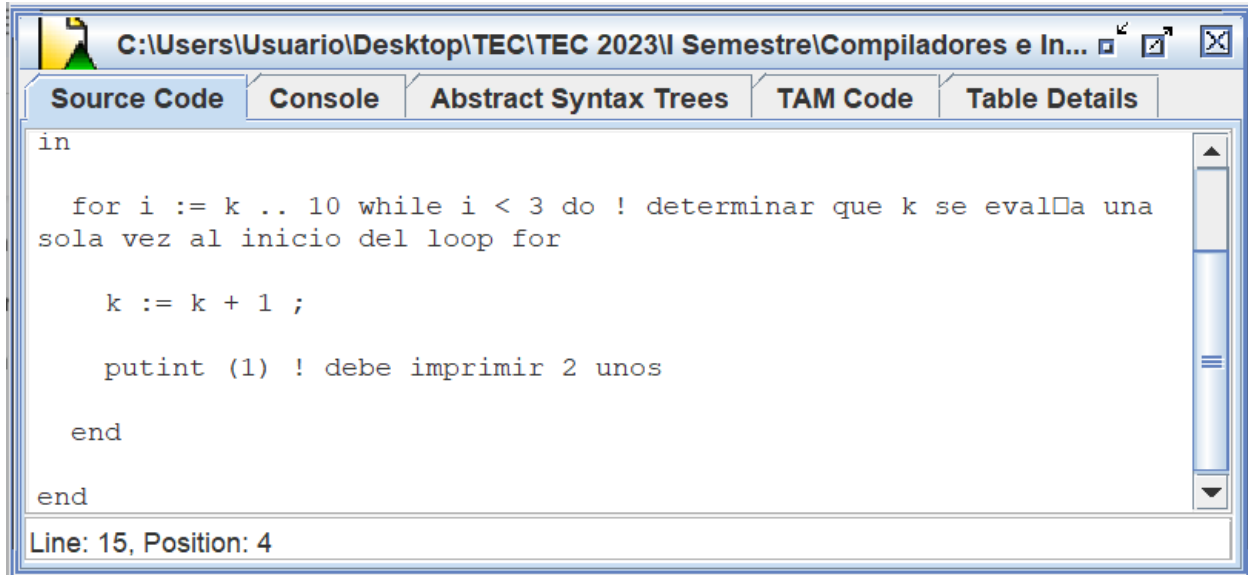


## 9.4 For while

### Objetivo

El objetivo de esta prueba es ver el funcionamiento contextual correcto esperado por el comando for while. En estas pruebas se espera representar el funcionamiento del compilador ante un caso de prueba correcto y uno que presente un error.

### Diseño de la prueba Correcta



```

in

  for i := k .. 10 while i < 3 do ! determinar que k se evalúa una
sola vez al inicio del loop for

    k := k + 1 ;

    putint (1) ! debe imprimir 2 unos

  end

end

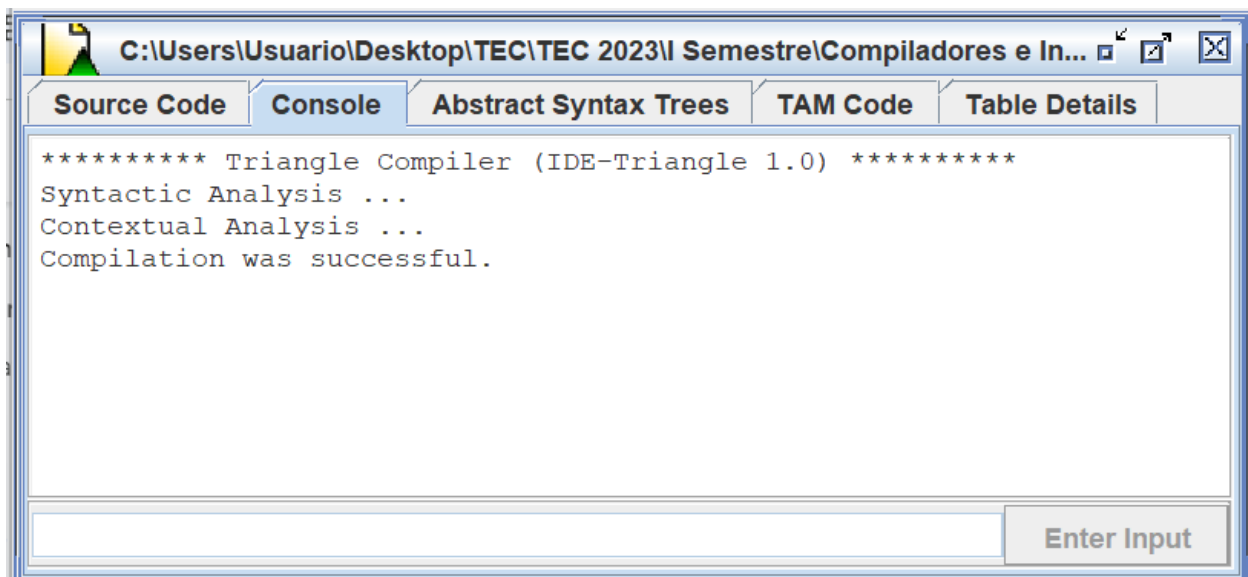
Line: 15, Position: 4

```

### Resultados esperados

El resultado esperado es que la prueba sea compilada de manera satisfactoria generando los respectivos archivos de salida del proyecto anterior.

### Resultados observados



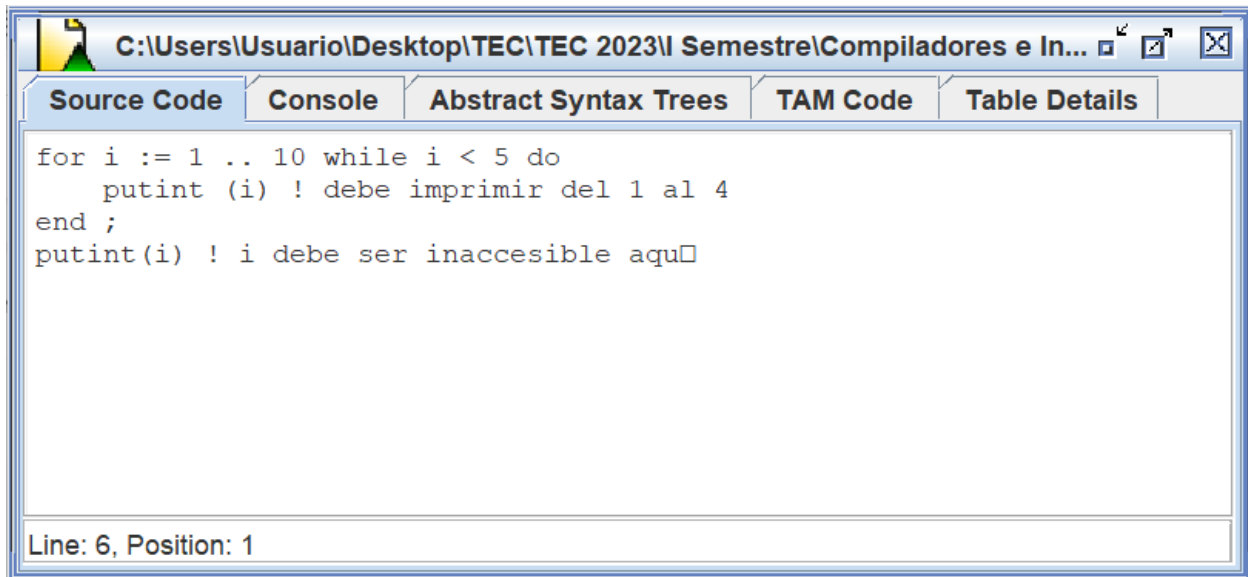
```

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

```

Enter Input

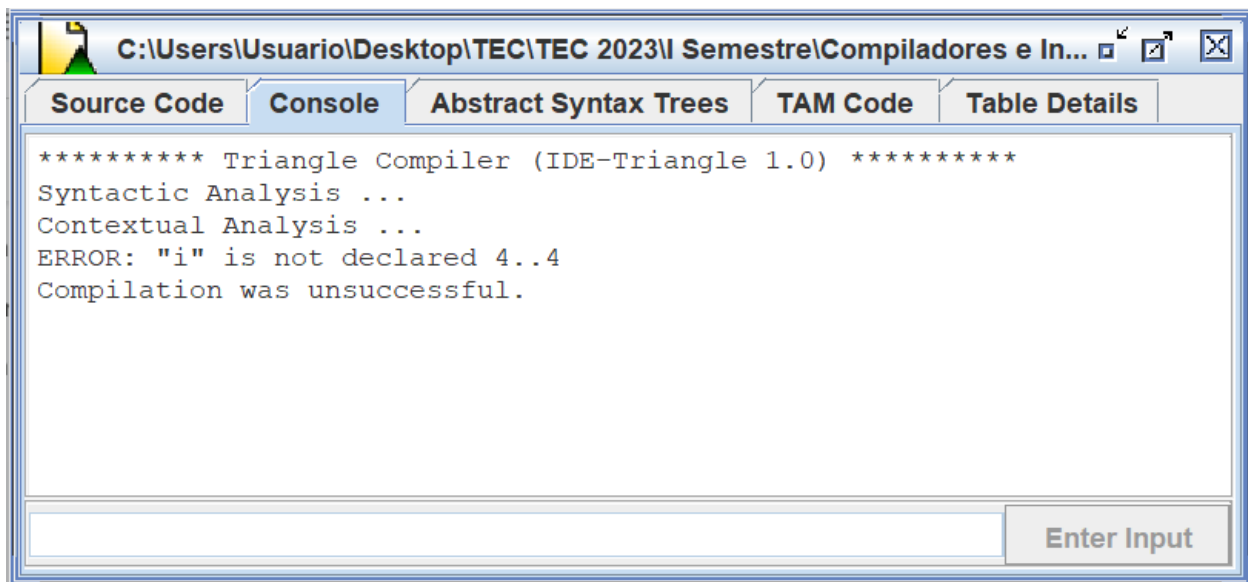
## Diseño de la prueba Errónea



## Resultados esperados

El resultado esperado es que la prueba a la hora de compilar es que encuentre el error de contexto y que lo indique la pestaña de consola.

## Resultados observados

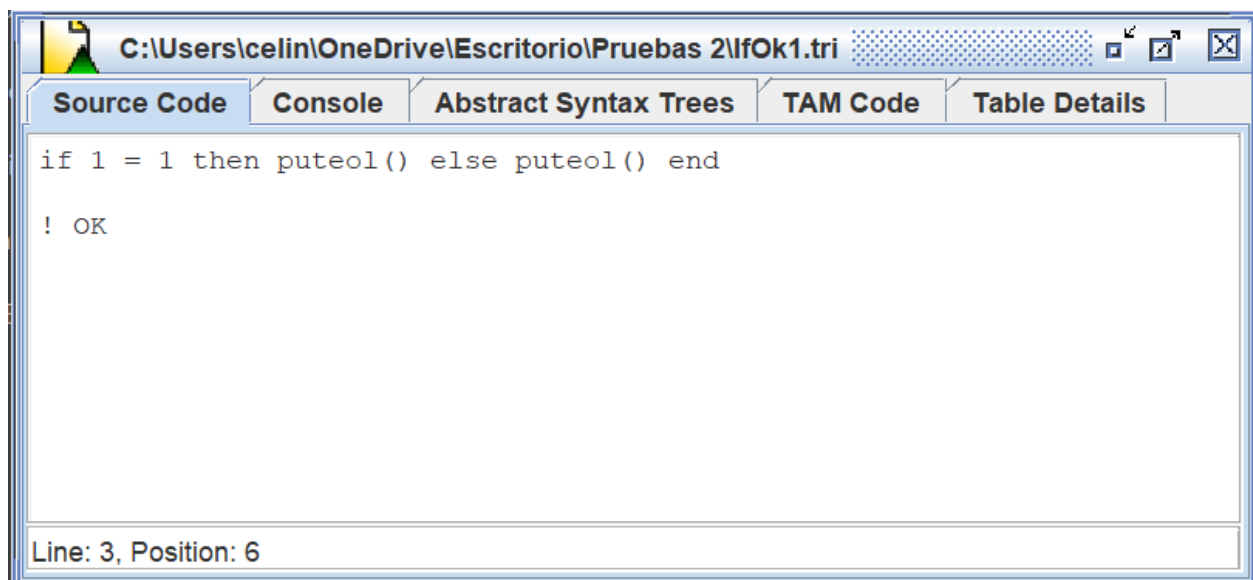


## 9.5 If

### Objetivo

El objetivo de esta prueba es ver el funcionamiento contextual correcto esperado por el comando if. En estas pruebas se espera representar el funcionamiento del compilador ante un caso de prueba correcto y uno que presente un error.

### Diseño de la prueba Correcta

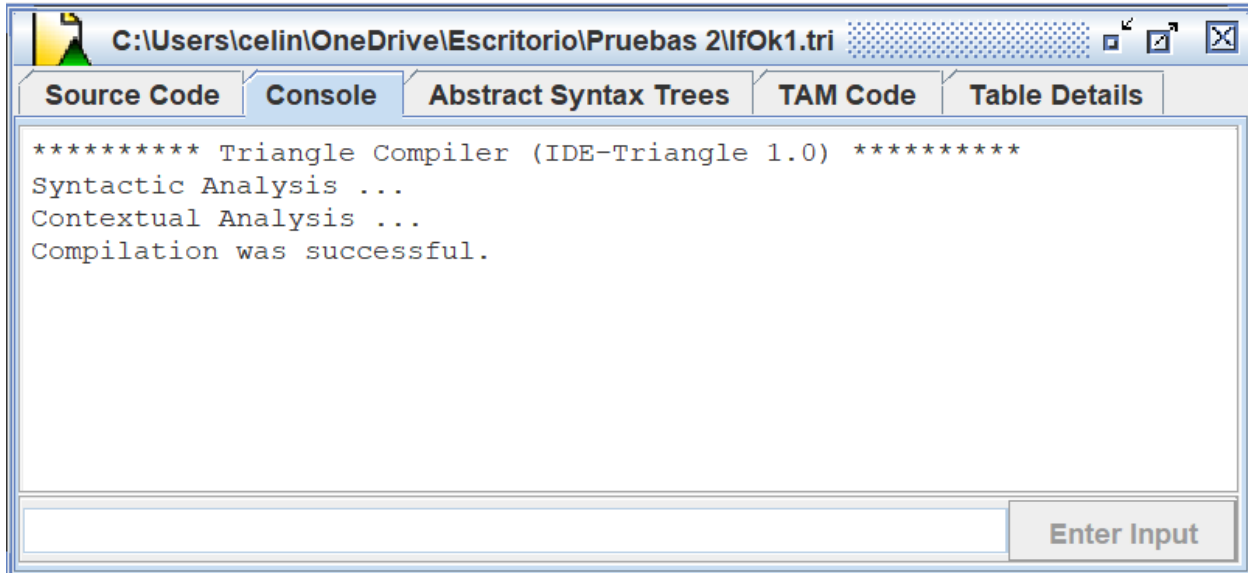


### Resultados esperados

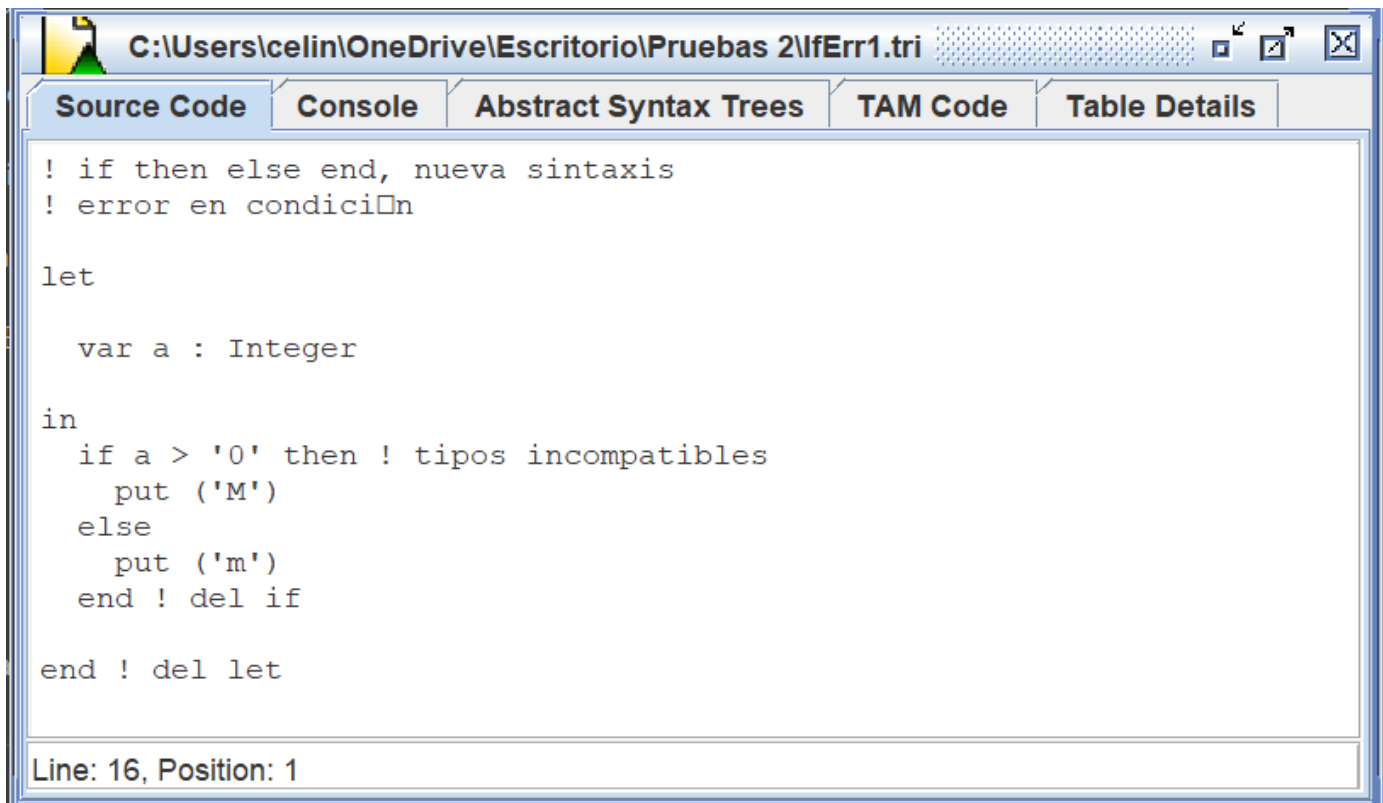
El resultado esperado es que la prueba sea compilada de manera satisfactoria generando los respectivos archivos de salida del proyecto anterior.

### Resultados observados





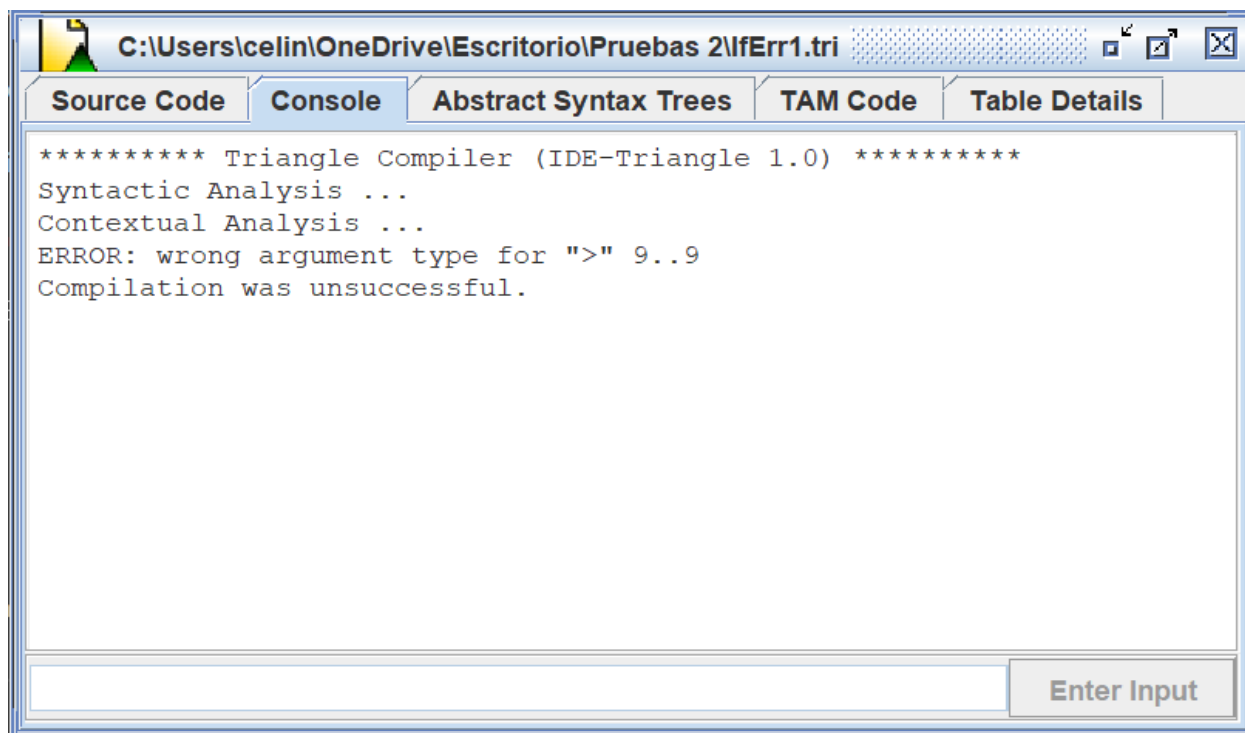
### Diseño de la prueba Errónea



## Resultados esperados

El resultado esperado es que la prueba a la hora de compilar es que encuentre el error de contexto y que lo indique la pestaña de consola.

## Resultados observados

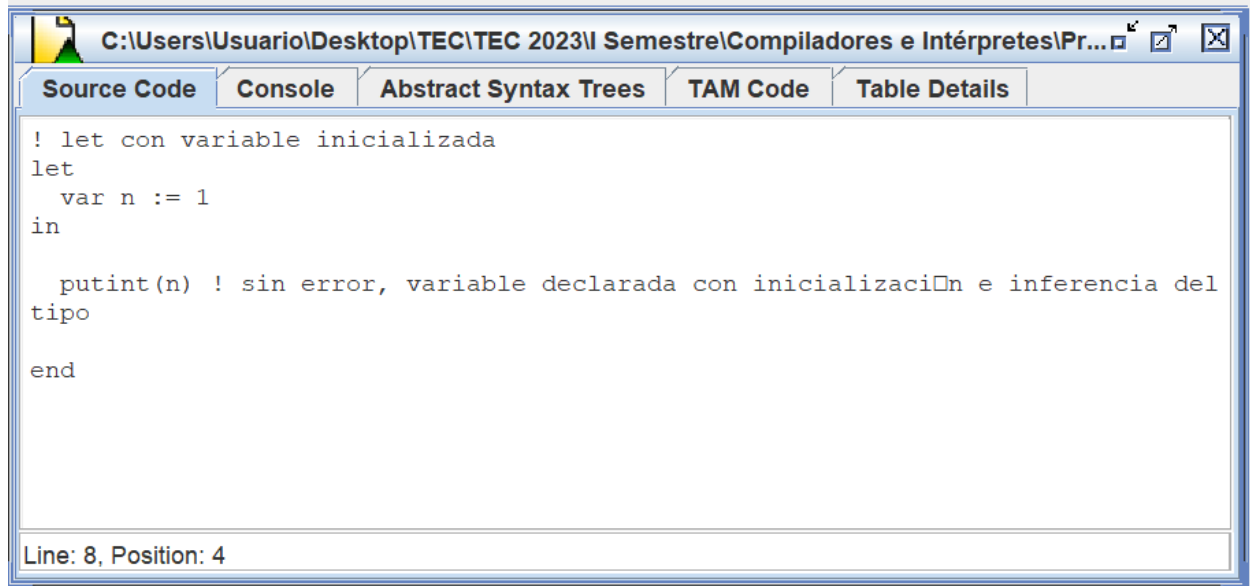


## 9.6 Let

### Objetivo

El objetivo de esta prueba es ver el funcionamiento contextual correcto esperado por el comando let. En estas pruebas se espera representar el funcionamiento del compilador ante un caso de prueba correcto y uno que presente un error.

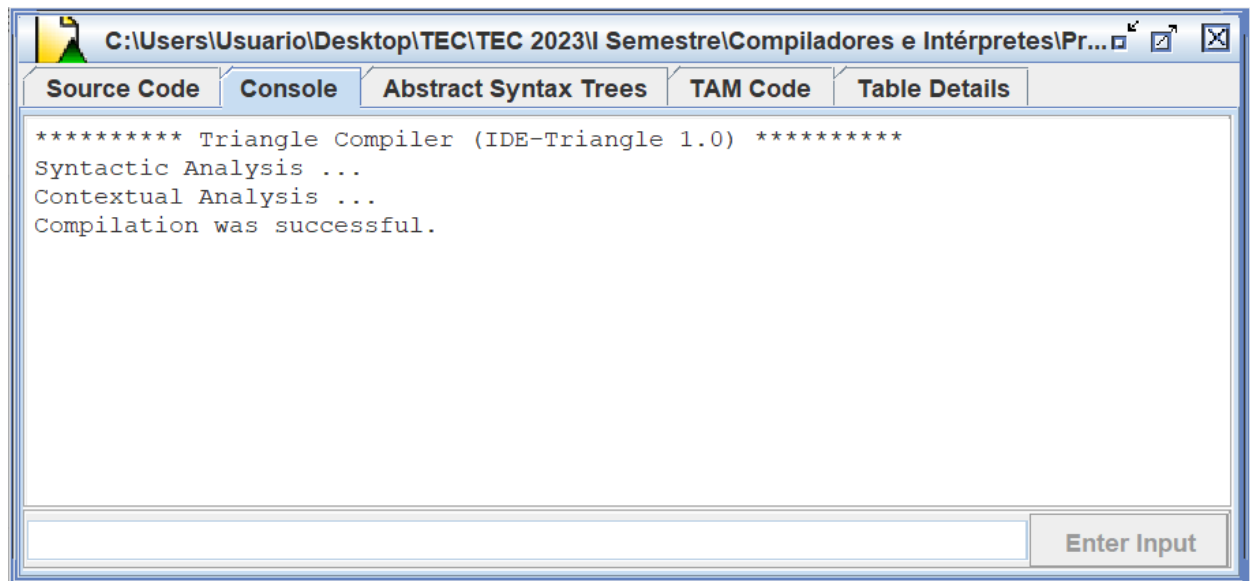
### Diseño de la prueba Correcta



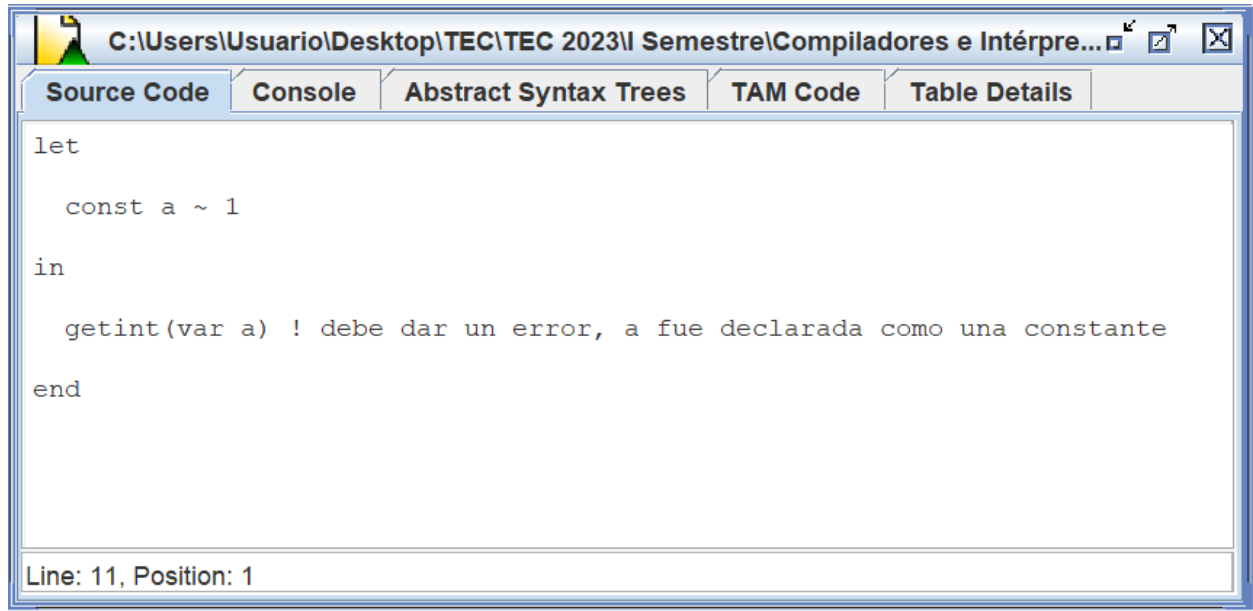
## Resultados esperados

El resultado esperado es que la prueba sea compilada de manera satisfactoria generando los respectivos archivos de salida del proyecto anterior.

## Resultados observados



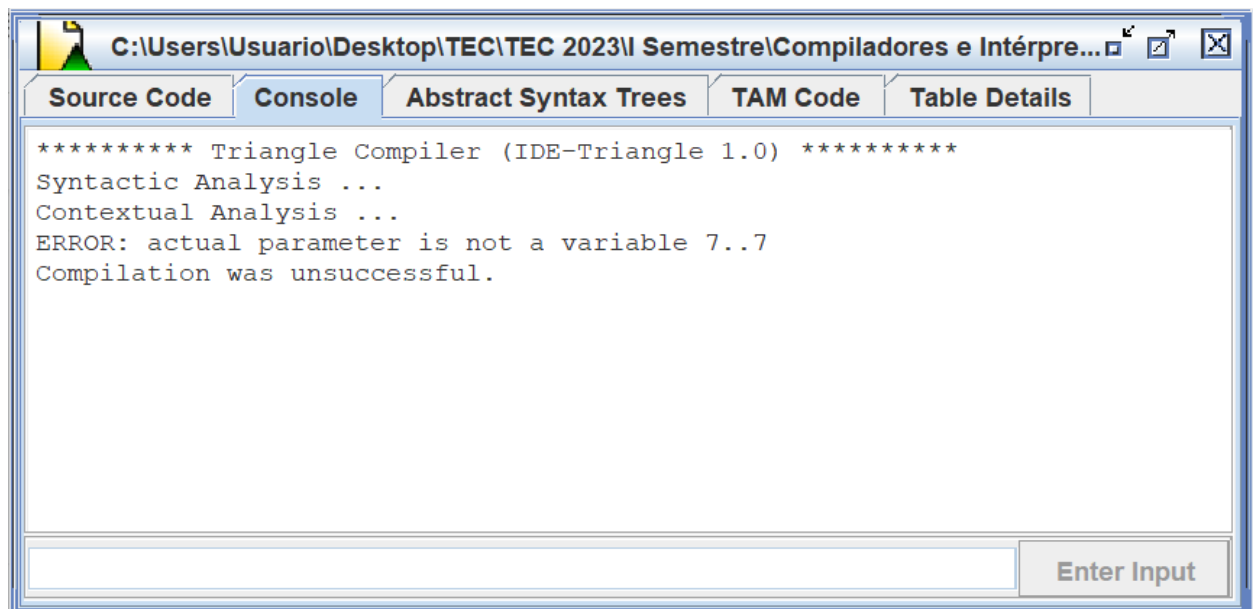
## Dise o de la prueba Err nea



### Resultados esperados

El resultado esperado es que la prueba a la hora de compilar es que encuentre el error de contexto y que lo indique la pestaña de consola.

### Resultados observados

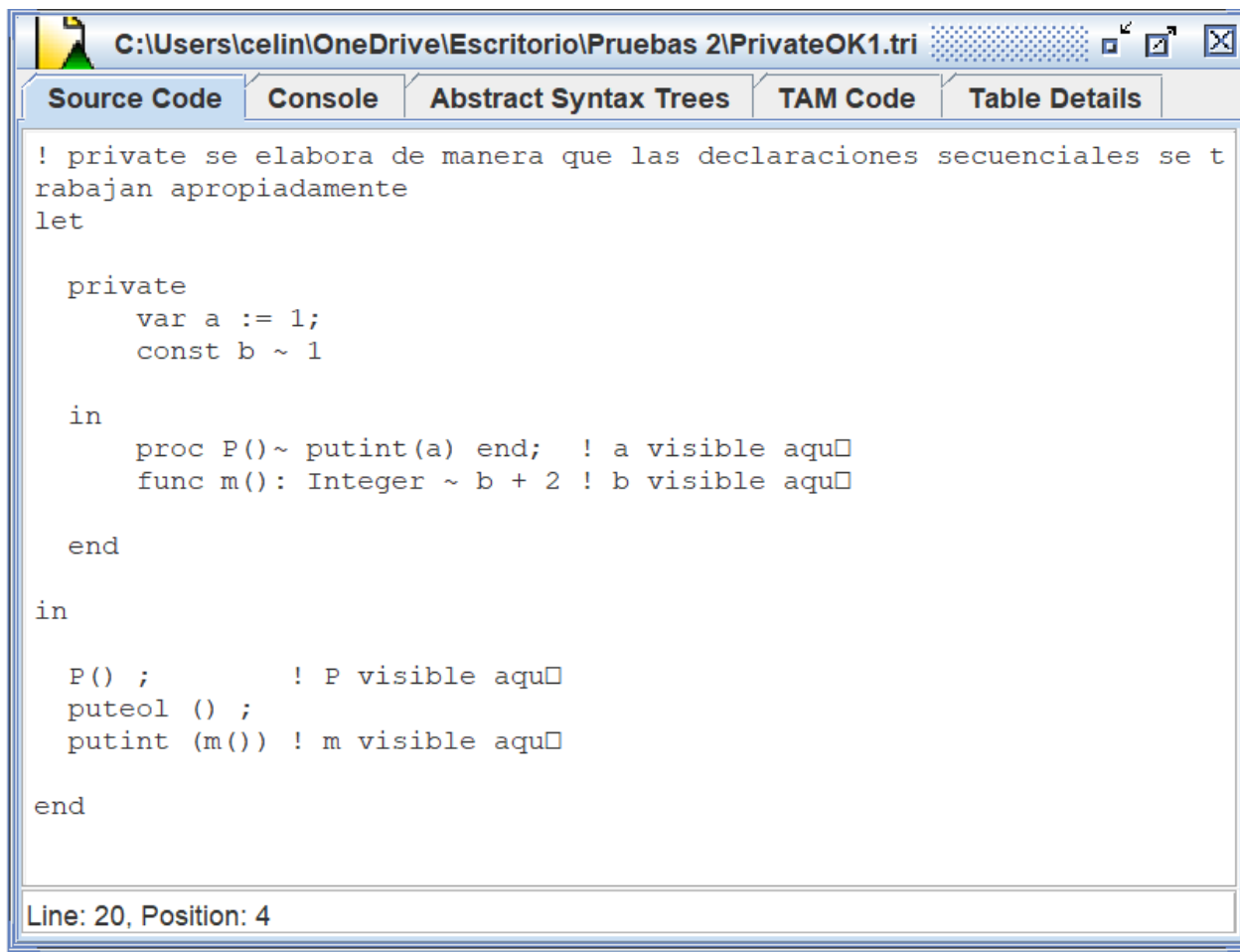


## 9.7 Private

### Objetivo

El objetivo de esta prueba es ver el funcionamiento contextual correcto esperado por el comando private. En estas pruebas se espera representar el funcionamiento del compilador ante un caso de prueba correcto y uno que presente un error.

### Diseño de la prueba Correcta



```

! private se elabora de manera que las declaraciones secuenciales se t
rabajan apropiadamente
let

  private
    var a := 1;
    const b ~ 1

  in
    proc P()~ putint(a) end;  ! a visible aqua
    func m(): Integer ~ b + 2 ! b visible aqua

  end

in

  P() ;          ! P visible aqua
  puteol () ;
  putint (m()) ! m visible aqua

end

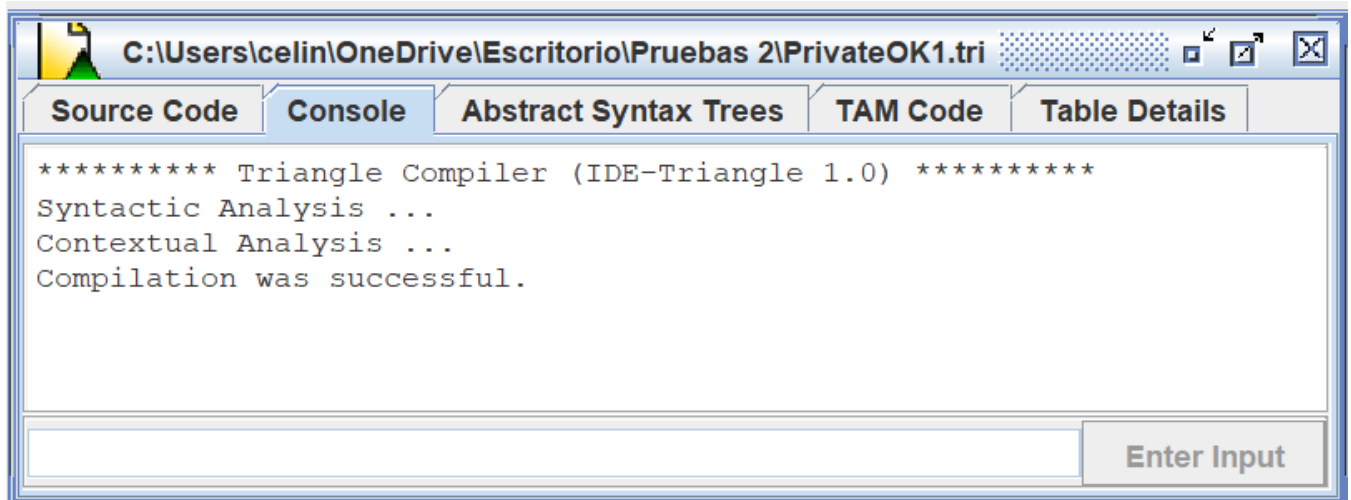
```

Line: 20, Position: 4

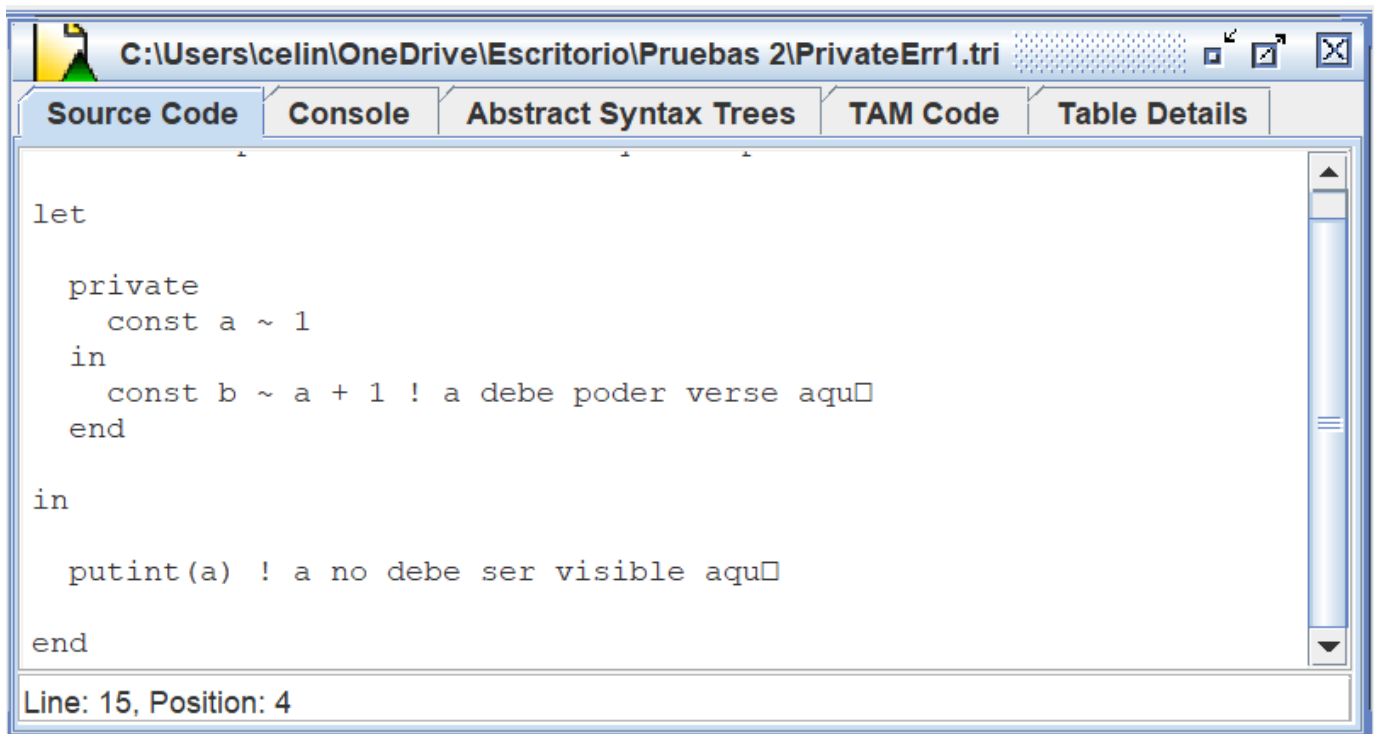
### Resultados esperados

El resultado esperado es que la prueba sea compilada de manera satisfactoria generando los respectivos archivos de salida del proyecto anterior.

## Resultados observados



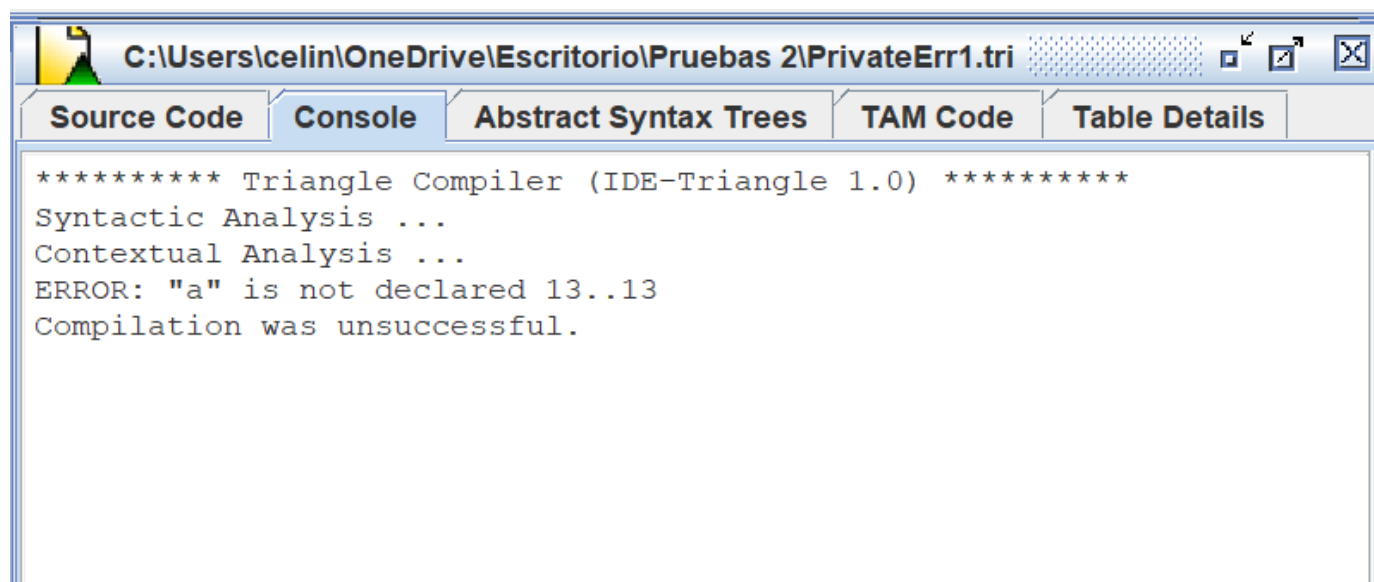
## Diseño de la prueba Errónea



## Resultados esperados

El resultado esperado es que la prueba a la hora de compilar es que encuentre el error de contexto y que lo indique la pestaña de consola.

### Resultados observados



The screenshot shows a window titled "C:\Users\celin\OneDrive\Escritorio\Pruebas 2\PrivateErr1.tri". It has five tabs: "Source Code", "Console", "Abstract Syntax Trees", "TAM Code", and "Table Details". The "Console" tab is active, displaying the following text:

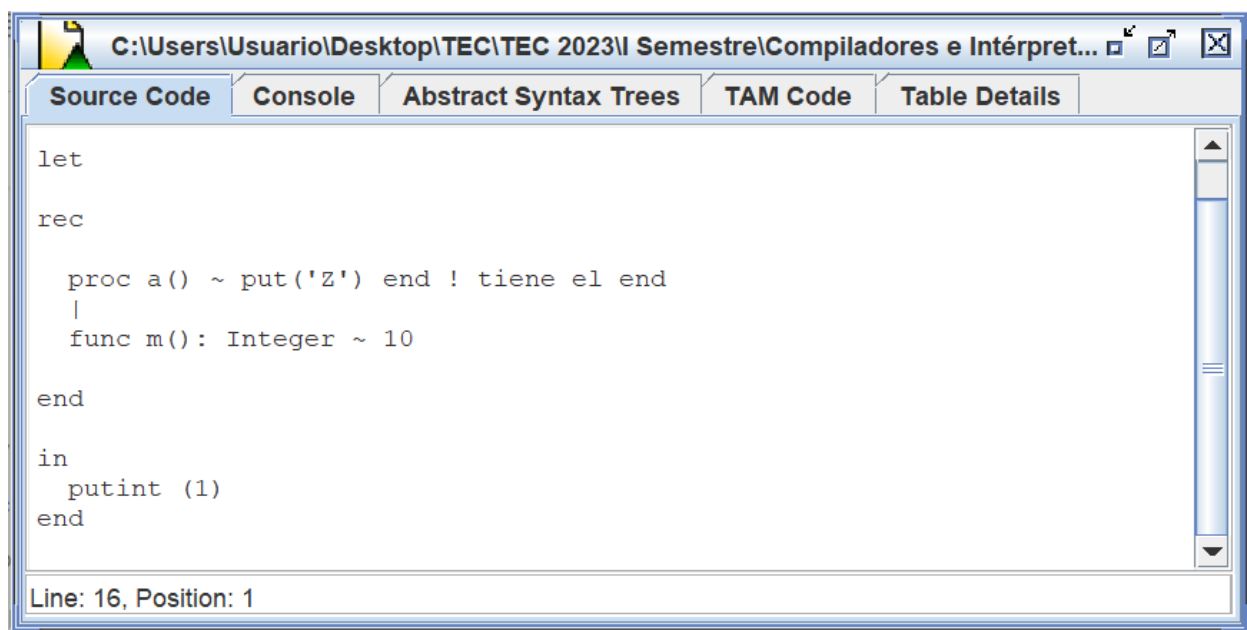
```
***** Triangle Compiler (IDE-Triangle 1.0) *****  
Syntactic Analysis ...  
Contextual Analysis ...  
ERROR: "a" is not declared 13..13  
Compilation was unsuccessful.
```

## 9.8 Rec

### Objetivo

El objetivo de esta prueba es ver el funcionamiento contextual correcto esperado por el comando rec. En estas pruebas se espera representar el funcionamiento del compilador ante un caso de prueba correcto y uno que presente un error.

### Diseño de la prueba Correcta



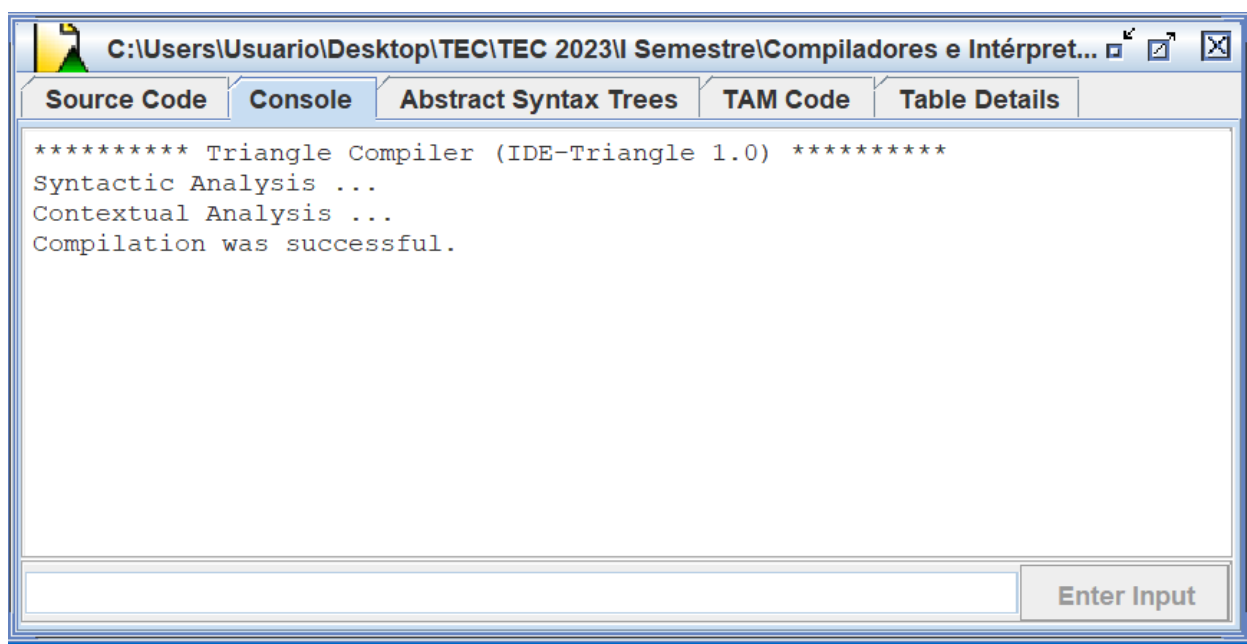
```
let
rec
    proc a() ~ put('Z') end ! tiene el end
    |
    func m(): Integer ~ 10
end
in
    putint (1)
end
```

Line: 16, Position: 1

## Resultados esperados

El resultado esperado es que la prueba sea compilada de manera satisfactoria generando los respectivos archivos de salida del proyecto anterior.

## Resultados observados

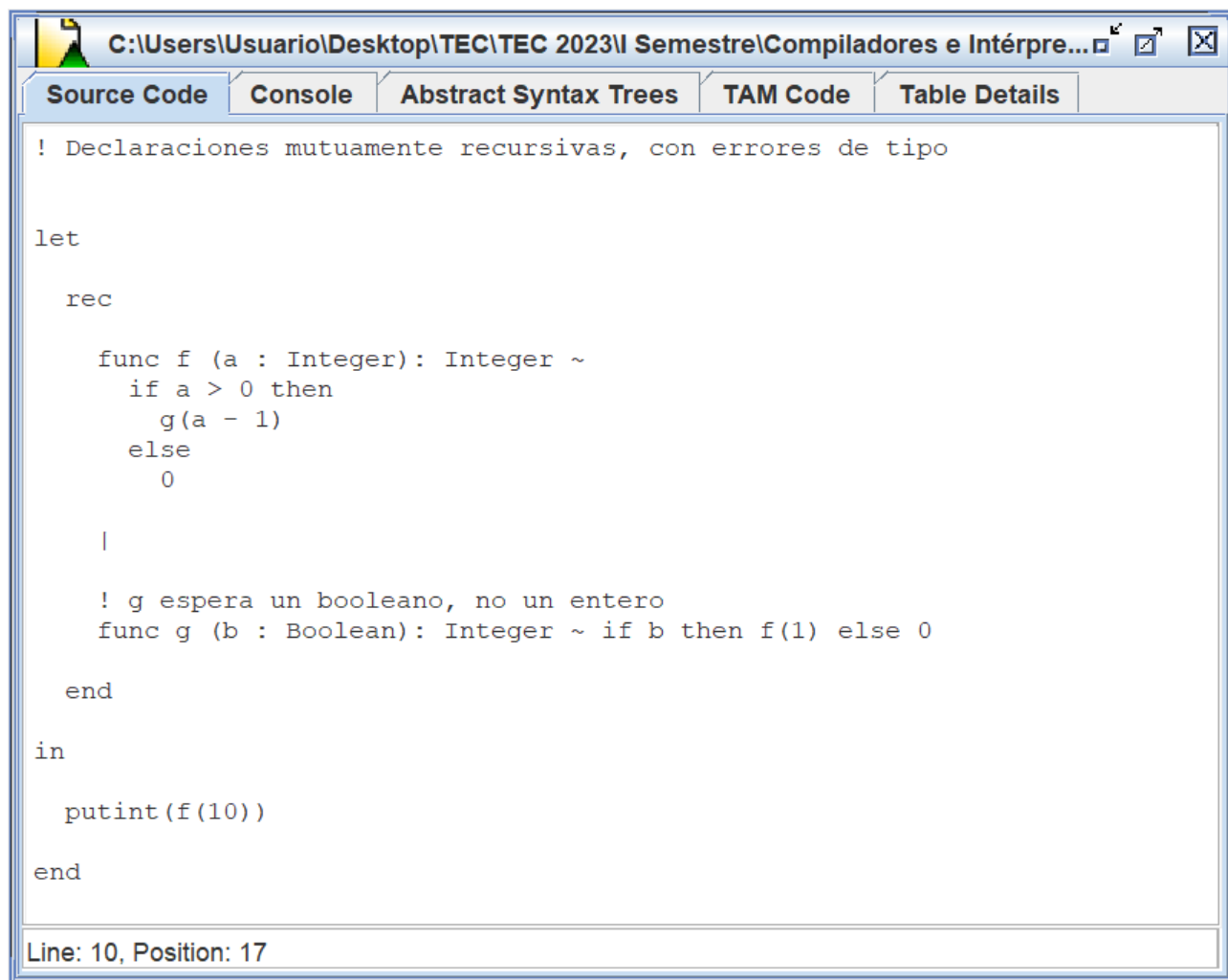


```
***** Triangle Compiler (IDE-Triangle 1.0) *****
Syntactic Analysis ...
Contextual Analysis ...
Compilation was successful.
```

Enter Input



## Diseño de la prueba Errónea



The screenshot shows a window titled "C:\Users\Usuario\Desktop\TEC\TEC 2023\I Semestre\Compiladores e Intérpre...". It has five tabs: "Source Code", "Console", "Abstract Syntax Trees", "TAM Code", and "Table Details". The "Source Code" tab is active, displaying the following code:

```
! Declaraciones mutuamente recursivas, con errores de tipo

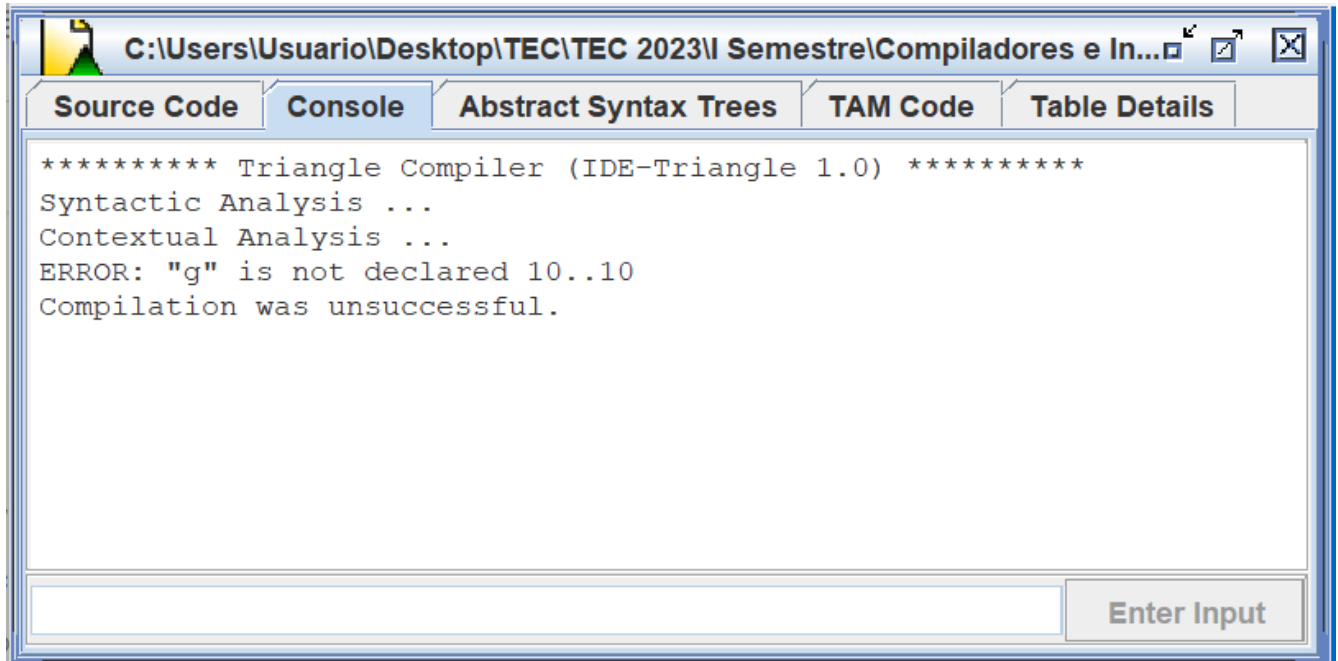
let
  rec
    func f (a : Integer): Integer ~
      if a > 0 then
        g(a - 1)
      else
        0
    |
    ! g espera un booleano, no un entero
    func g (b : Boolean): Integer ~ if b then f(1) else 0
  end
in
  putint(f(10))
end
```

At the bottom of the window, a status bar indicates "Line: 10, Position: 17".

### Resultados esperados

El resultado esperado es que la prueba a la hora de compilar es que encuentre el error de contexto y que lo indique la pestaña de consola.

### Resultados observados

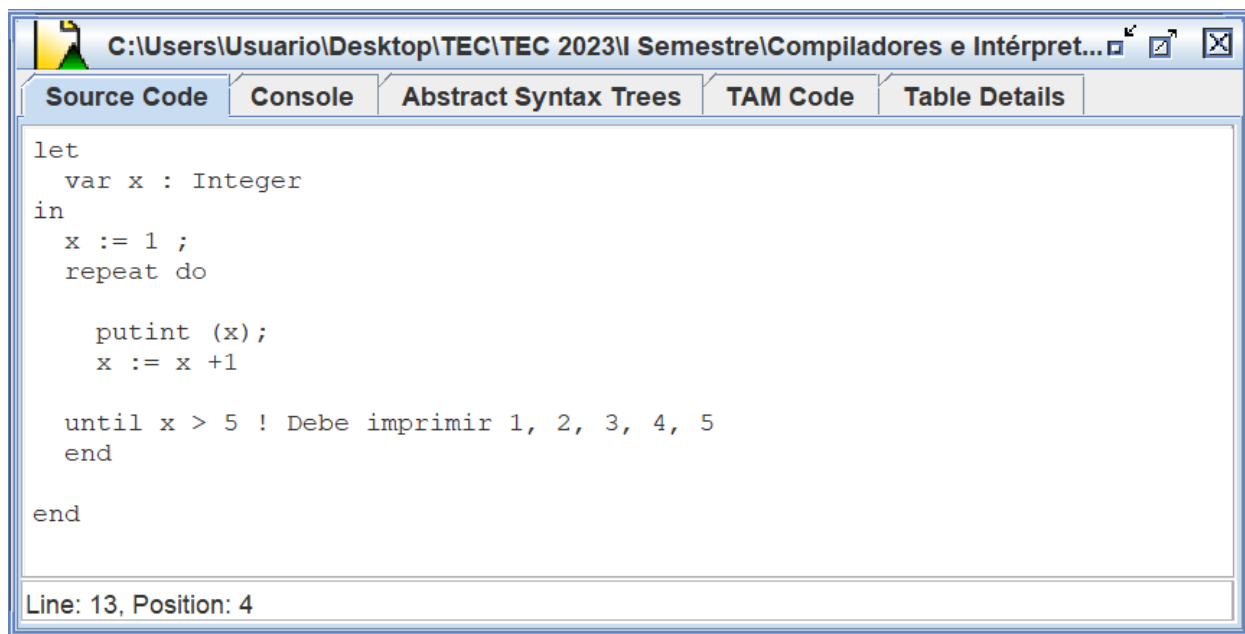


## 9.9 Repeat Do Until

### Objetivo

El objetivo de esta prueba es ver el funcionamiento contextual correcto esperado por el comando for. En estas pruebas se espera representar el funcionamiento del compilador ante un caso de prueba correcto y uno que presente un error.

### Diseño de la prueba Correcta



```

let
  var x : Integer
in
  x := 1 ;
  repeat do

    putint (x);
    x := x +1

  until x > 5 ! Debe imprimir 1, 2, 3, 4, 5
end

end

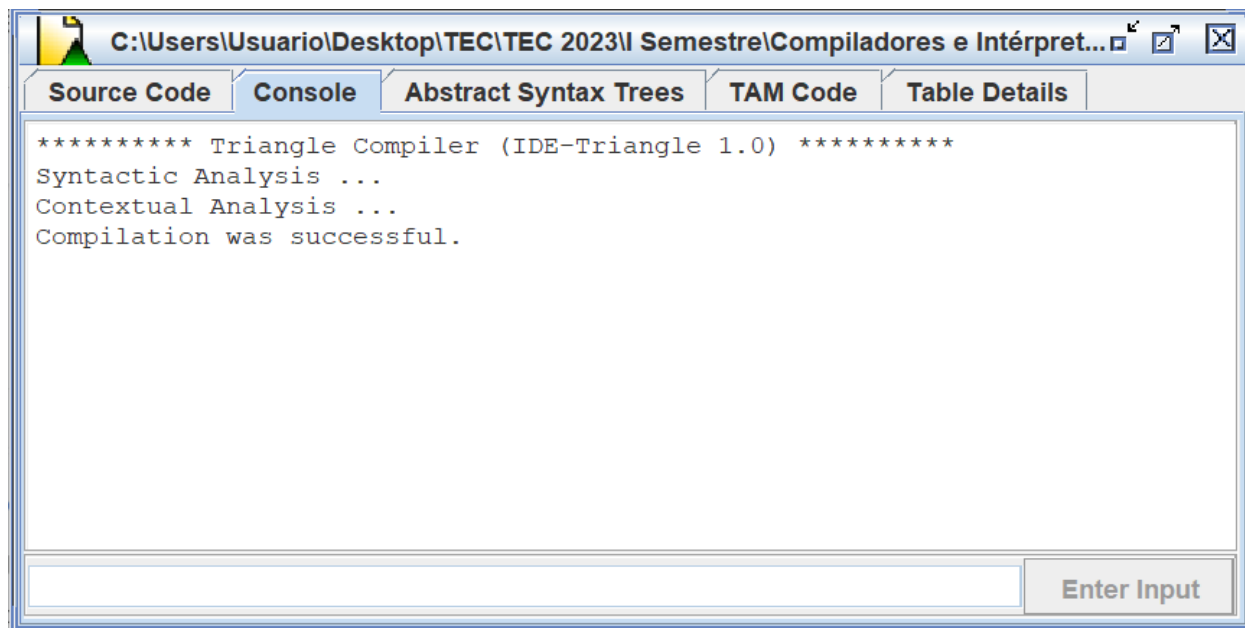
```

Line: 13, Position: 4

## Resultados esperados

El resultado esperado es que la prueba sea compilada de manera satisfactoria generando los respectivos archivos de salida del proyecto anterior.

## Resultados observados



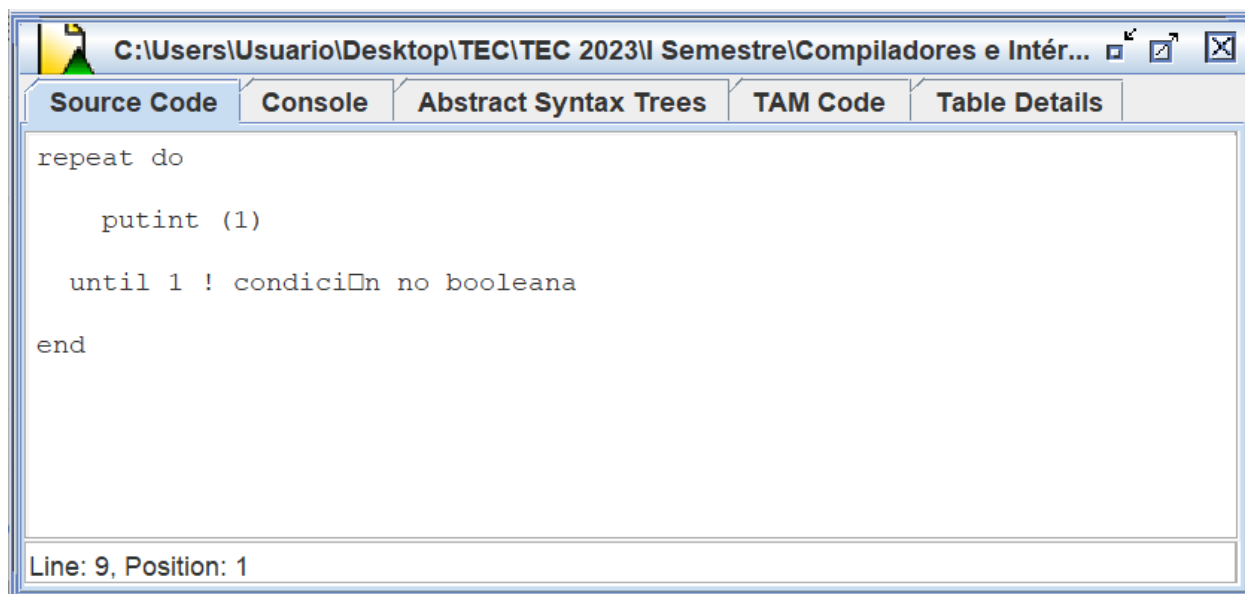
```

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

```

Enter Input

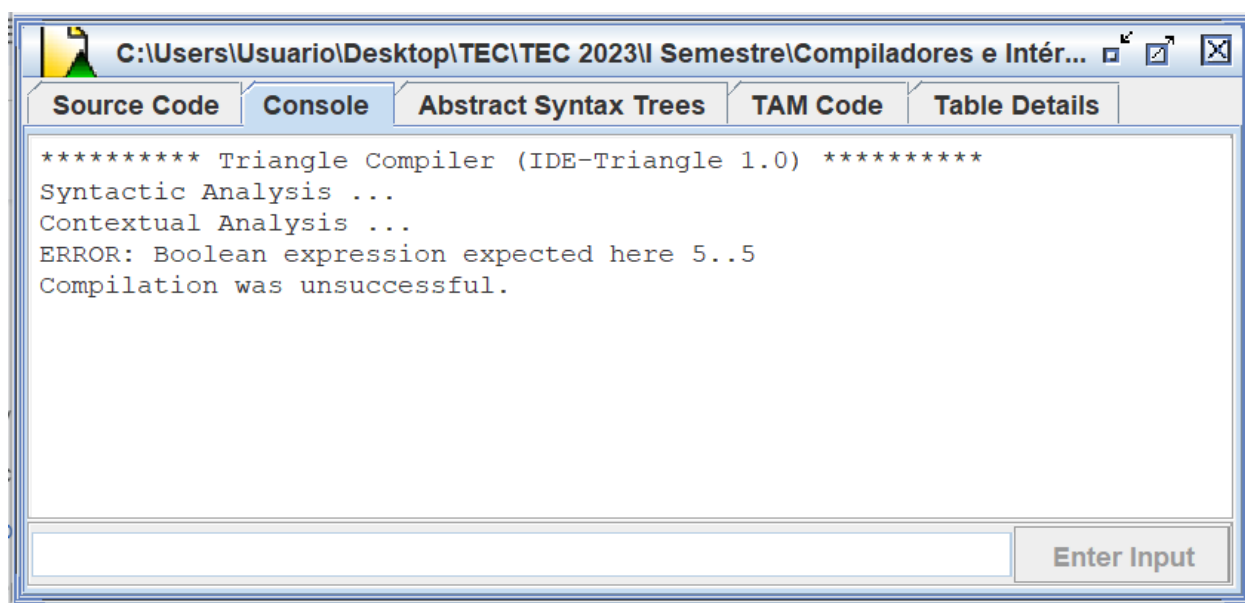
## Diseño de la prueba Errónea



## Resultados esperados

El resultado esperado es que la prueba a la hora de compilar es que encuentre el error de contexto y que lo indique la pesta a de consola.

## Resultados observados

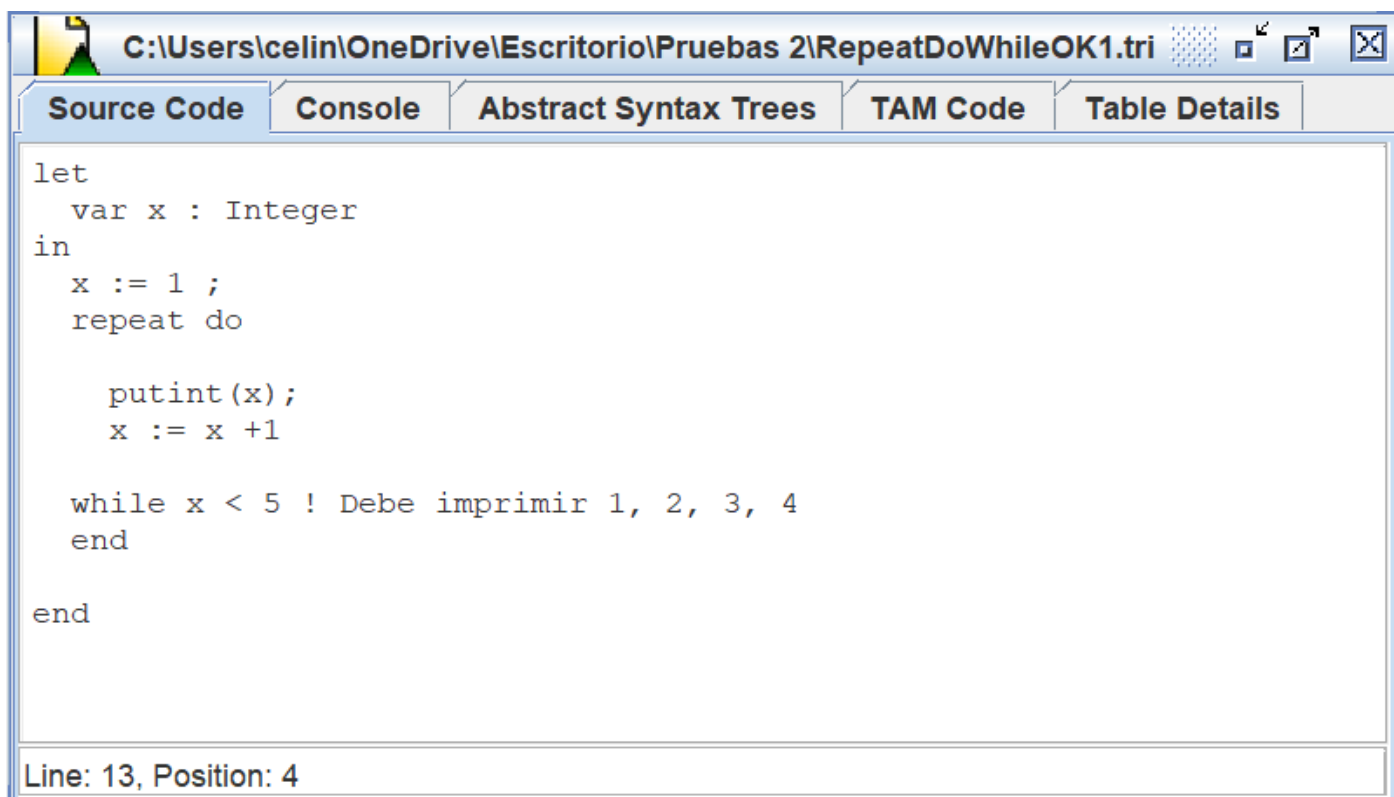


## 9.10 Repeat Do While

### Objetivo

El objetivo de esta prueba es ver el funcionamiento contextual correcto esperado por el comando repeat do while. En estas pruebas se espera representar el funcionamiento del compilador ante un caso de prueba correcto y uno que presente un error.

### Diseño de la prueba Correcta



The screenshot shows a code editor window with the title bar "C:\Users\celin\OneDrive\Escritorio\Pruebas 2\RepeatDoWhileOK1.tri". The editor has tabs for "Source Code", "Console", "Abstract Syntax Trees", "TAM Code", and "Table Details". The "Source Code" tab is active, displaying the following code:

```
let
  var x : Integer
in
  x := 1 ;
  repeat do

    putint(x);
    x := x +1

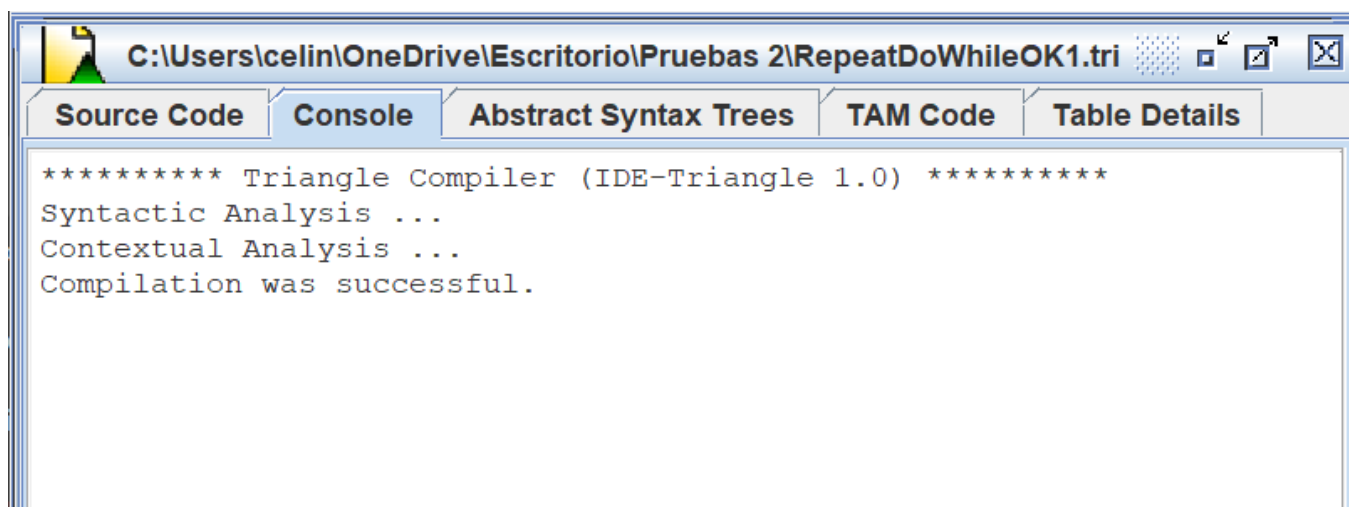
  while x < 5 ! Debe imprimir 1, 2, 3, 4
end
end
```

At the bottom of the editor, a status bar indicates "Line: 13, Position: 4".

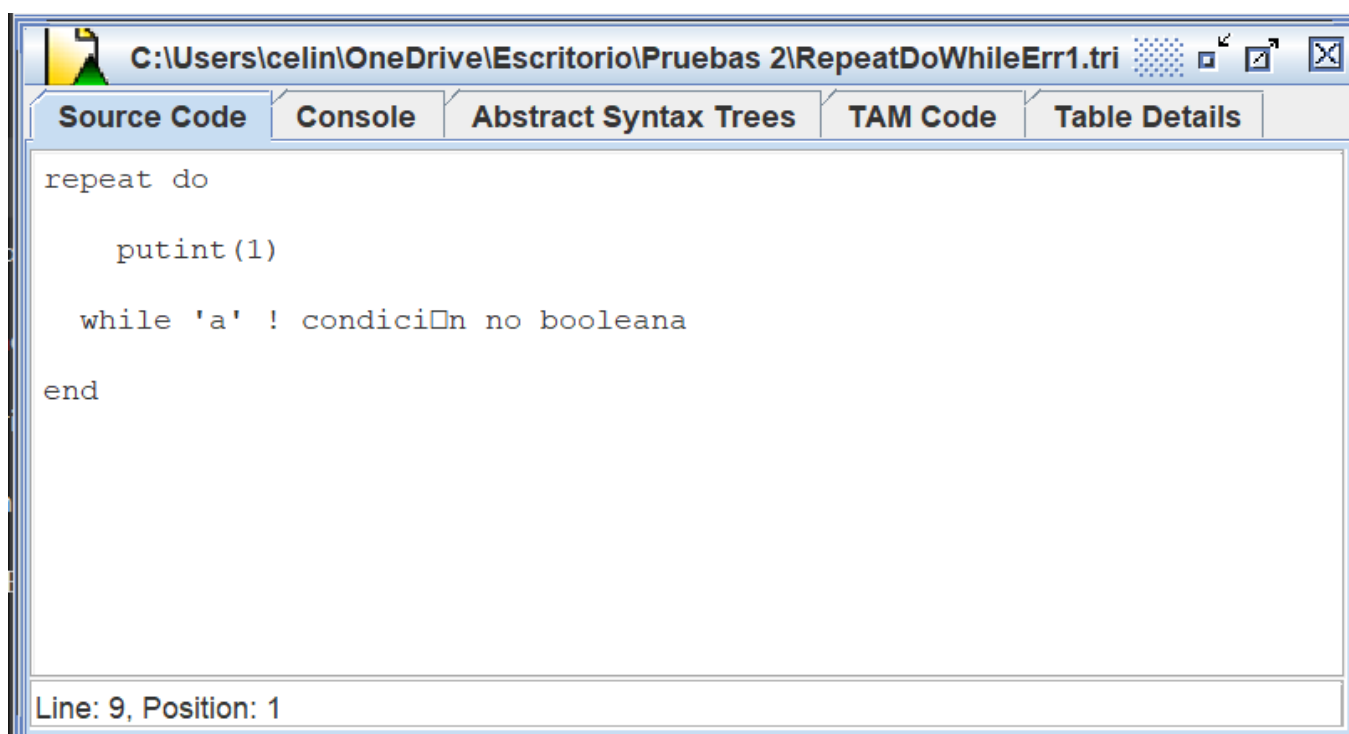
### Resultados esperados

El resultado esperado es que la prueba sea compilada de manera satisfactoria generando los respectivos archivos de salida del proyecto anterior.

### Resultados observados



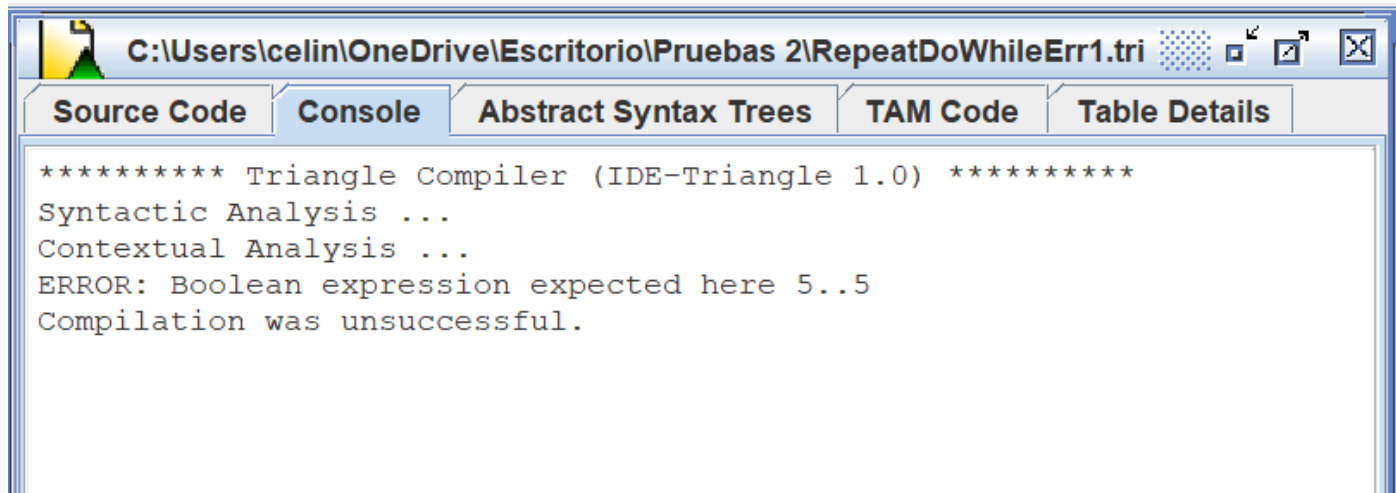
### Diseño de la prueba Errónea



### Resultados esperados

El resultado esperado es que la prueba a la hora de compilar es que encuentre el error de contexto y que lo indique la pestaña de consola.

## Resultados observados

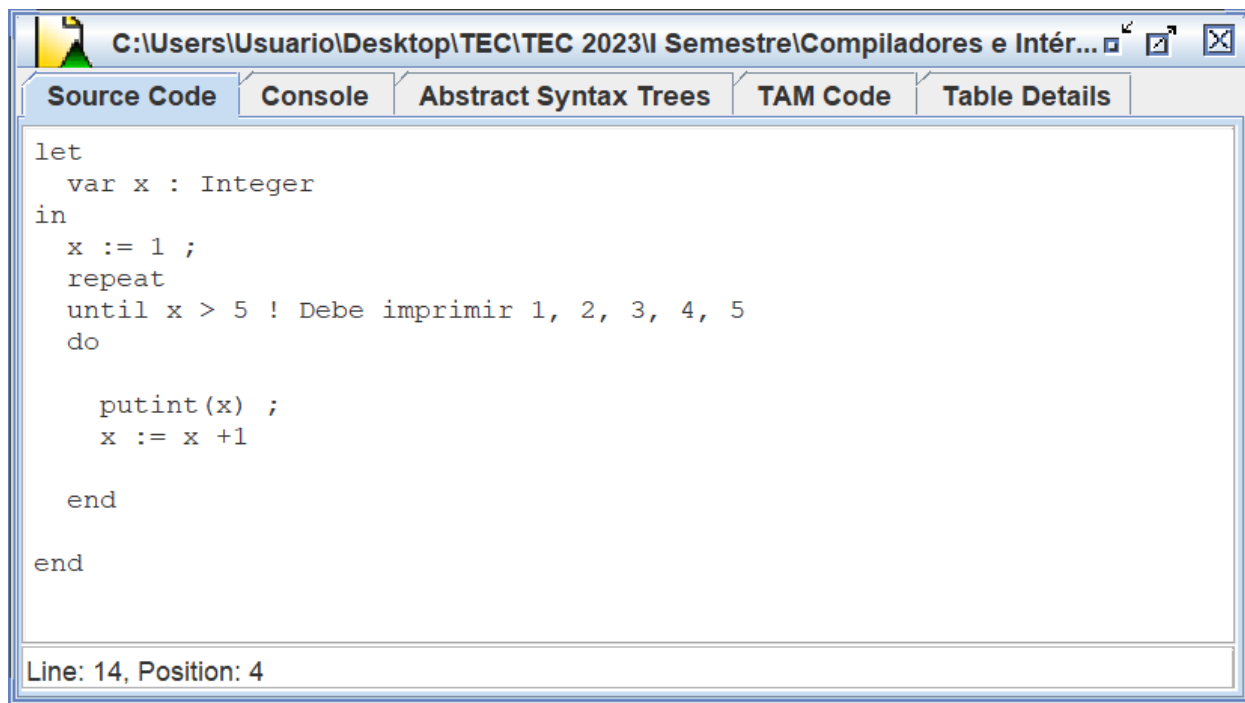


### 9.11 Repeat Until Do

#### Objetivo

El objetivo de esta prueba es ver el funcionamiento contextual correcto esperado por el comando repeat until do. En estas pruebas se espera representar el funcionamiento del compilador ante un caso de prueba correcto y uno que presente un error.

#### Diseño de la prueba Correcta



The screenshot shows the Triangle Compiler IDE window. The title bar indicates the file path: C:\Users\Usuario\Desktop\TEC\TEC 2023\I Semestre\Compiladores e Intér... The window has five tabs: Source Code, Console, Abstract Syntax Trees, TAM Code, and Table Details. The Source Code tab is active, displaying the following code:

```
let
  var x : Integer
in
  x := 1 ;
  repeat
    until x > 5 ! Debe imprimir 1, 2, 3, 4, 5
  do

    putint(x) ;
    x := x +1

  end

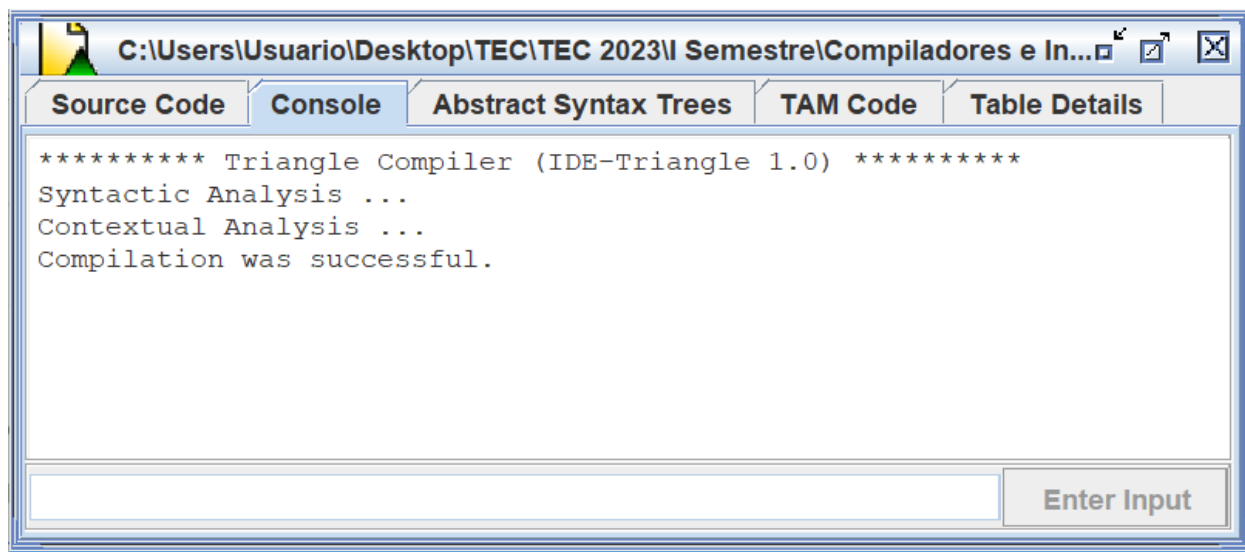
end
```

At the bottom of the window, the status bar shows "Line: 14, Position: 4".

### Resultados esperados

El resultado esperado es que la prueba sea compilada de manera satisfactoria generando los respectivos archivos de salida del proyecto anterior.

### Resultados observados



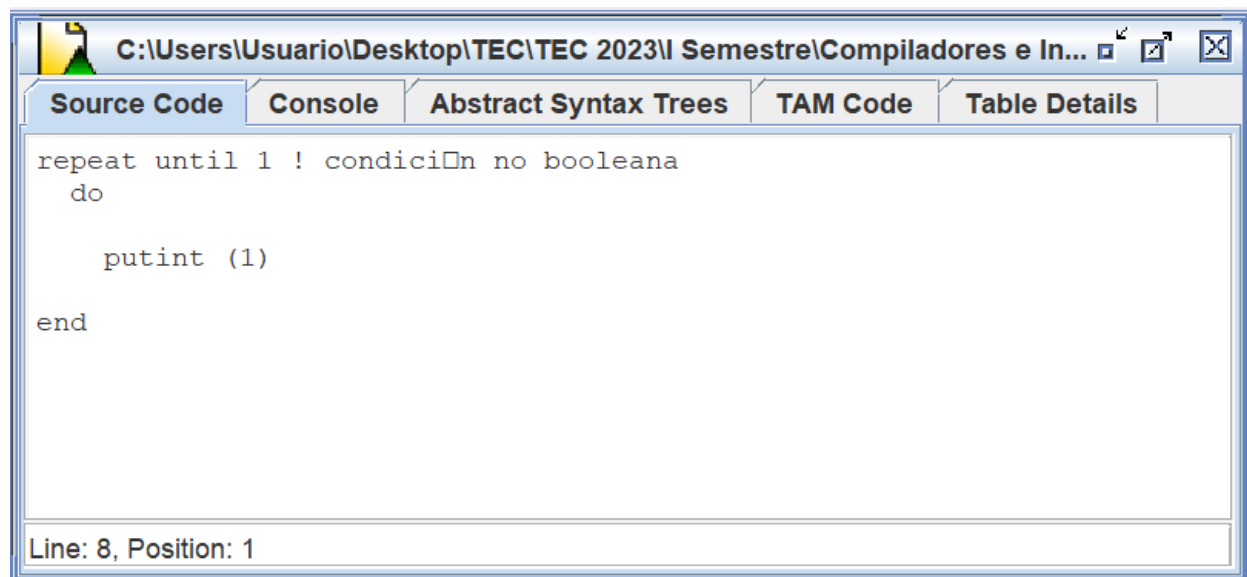
The screenshot shows the Triangle Compiler IDE window with the Console tab active. The console output displays the following messages:

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

At the bottom of the window, there is an input field and a button labeled "Enter Input".



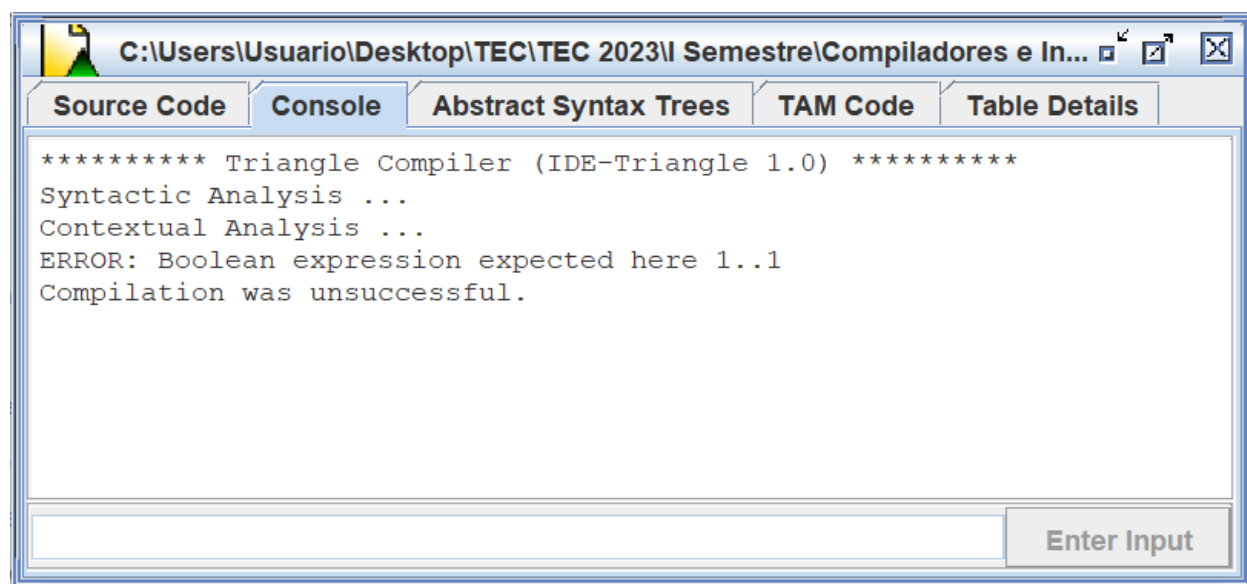
## Diseño de la prueba Errónea



## Resultados esperados

El resultado esperado es que la prueba a la hora de compilar es que encuentre el error de contexto y que lo indique la pesta a de consola.

## Resultados observados

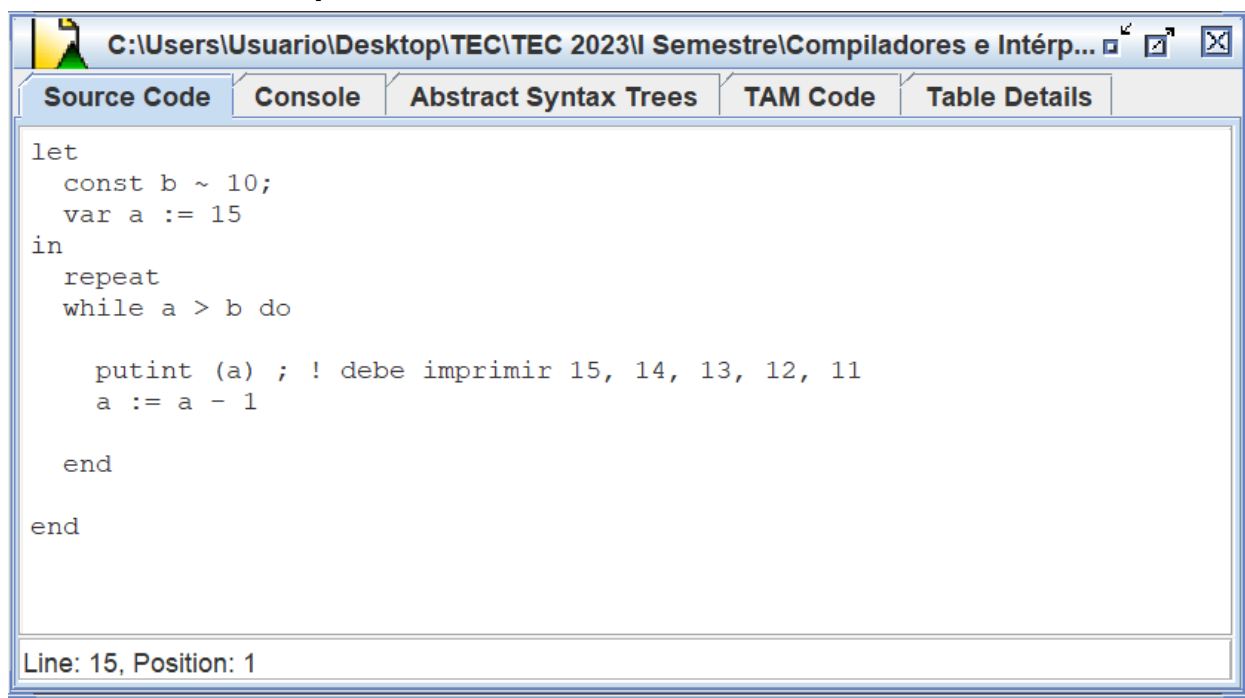


## 9.12 Repeat While Do

### Objetivo

El objetivo de esta prueba es ver el funcionamiento contextual correcto esperado por el comando repeat while do. En estas pruebas se espera representar el funcionamiento del compilador ante un caso de prueba correcto y uno que presente un error.

### Diseño de la prueba Correcta



```
let
  const b ~ 10;
  var a := 15
in
  repeat
    while a > b do

      putint (a) ; ! debe imprimir 15, 14, 13, 12, 11
      a := a - 1

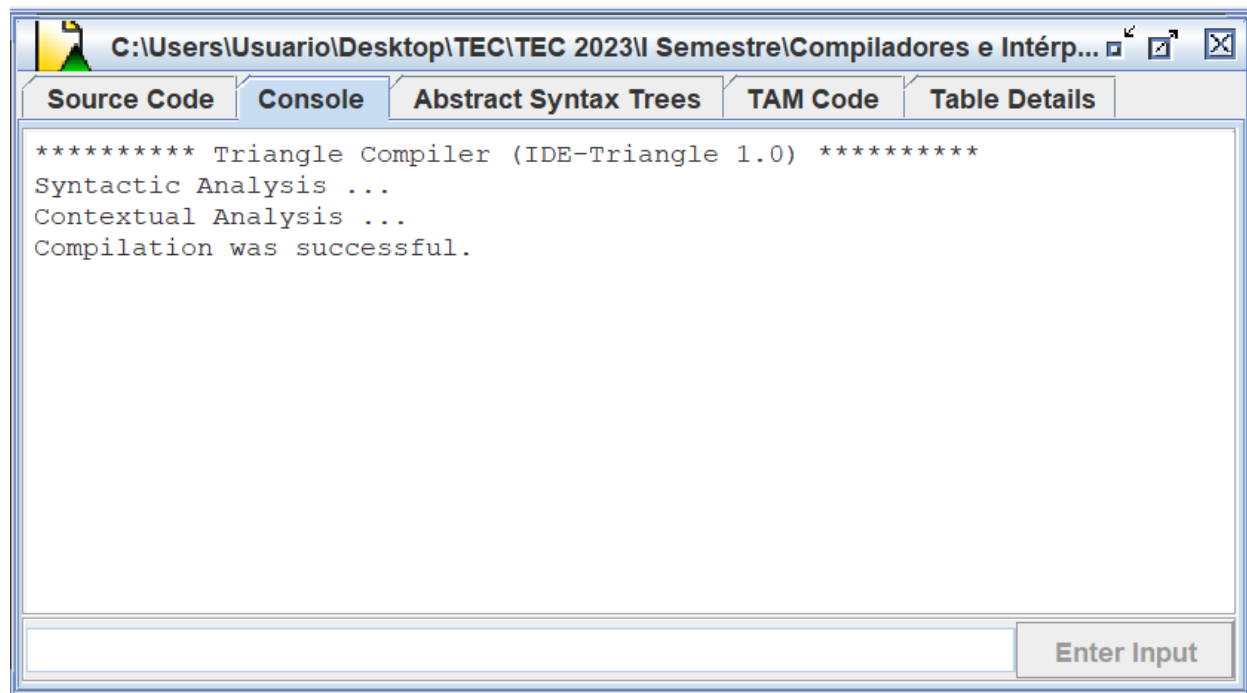
    end
  end
end
```

Line: 15, Position: 1

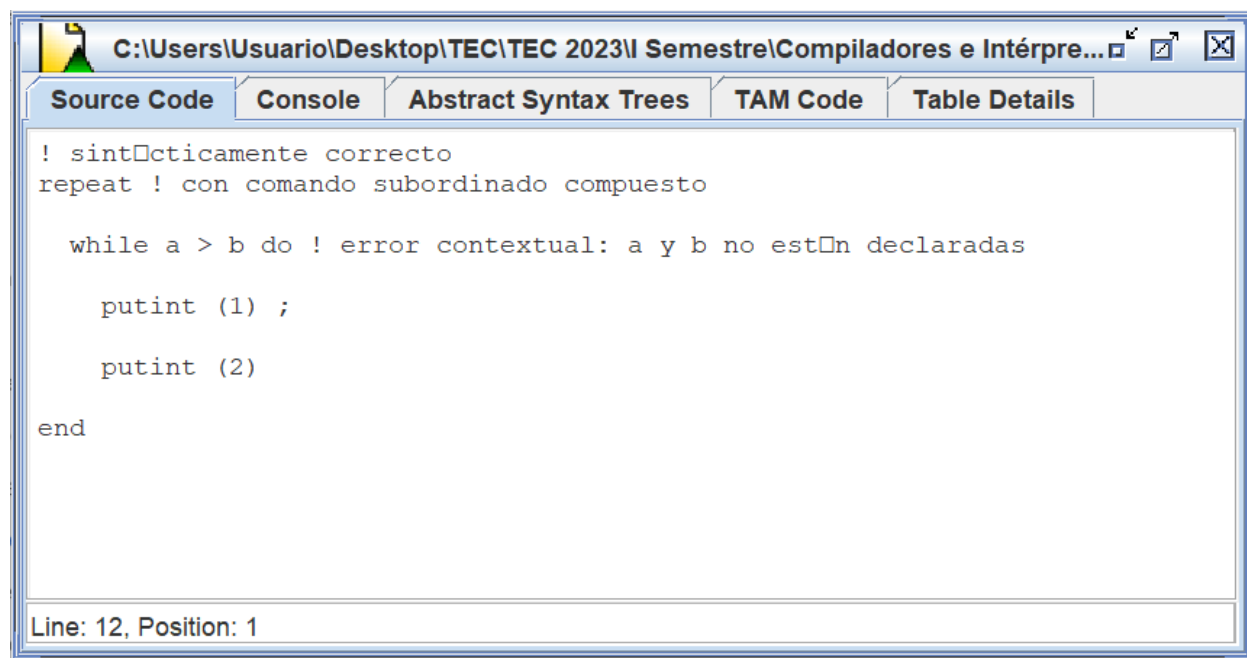
### Resultados esperados

El resultado esperado es que la prueba sea compilada de manera satisfactoria generando los respectivos archivos de salida del proyecto anterior.

### Resultados observados



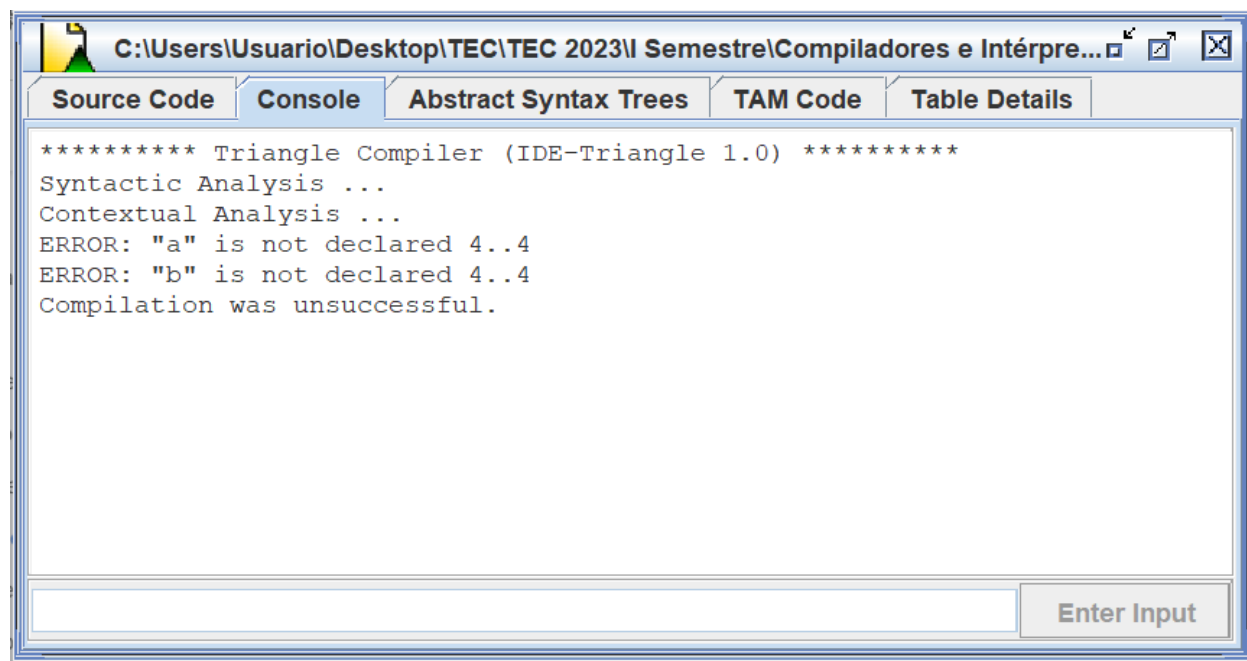
### Diseño de la prueba Errónea



### Resultados esperados

El resultado esperado es que la prueba a la hora de compilar es que encuentre el error de contexto y que lo indique la pestaña de consola.

## Resultados observados

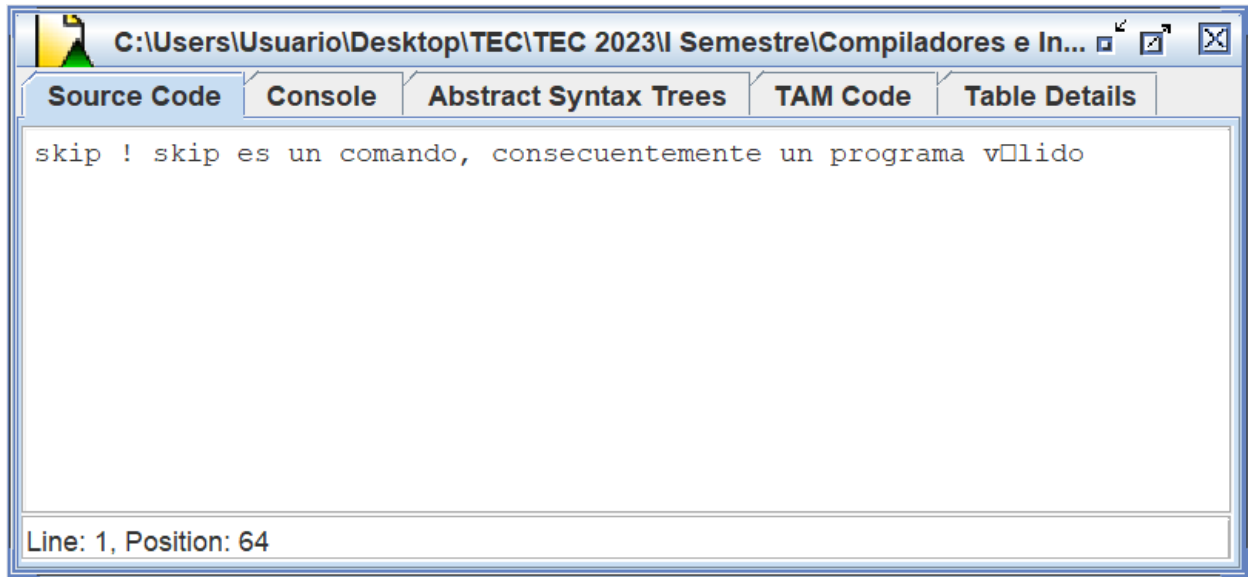


### 9.13 Skip

#### Objetivo

El objetivo de esta prueba es ver la forma sintáctica correcta esperada por el comando skip, ya que este no requiere de revisión contextual, pues no depende de tipos ni de declaraciones de identificadores. En estas pruebas se espera representar el funcionamiento del compilador ante un caso de prueba correcto y uno que presente un error.

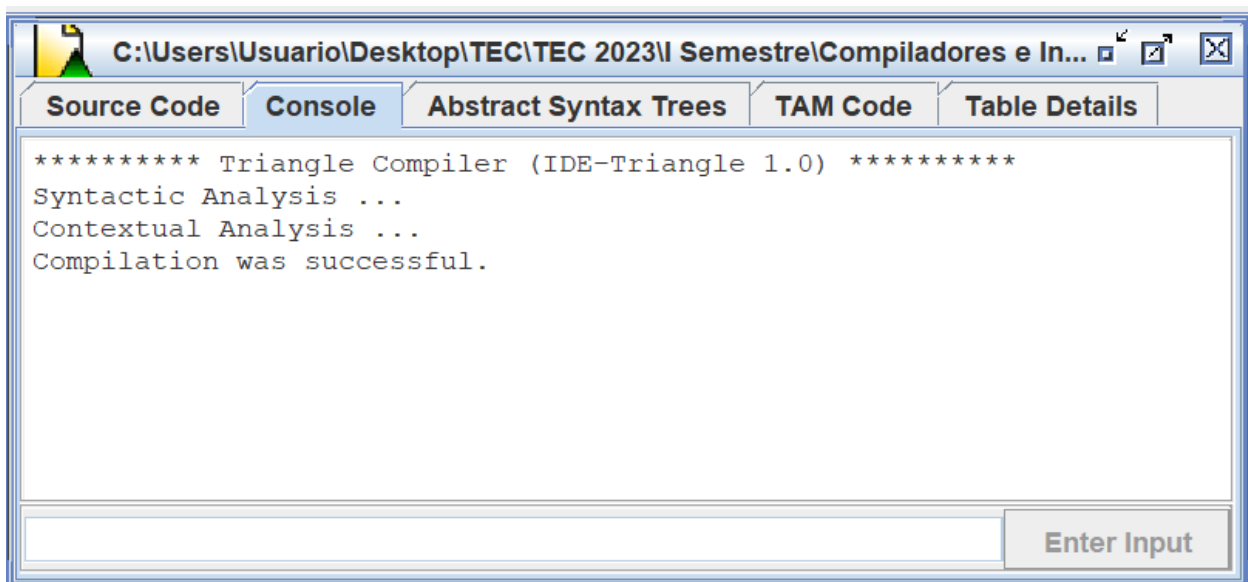
#### Diseño de la prueba Correcta



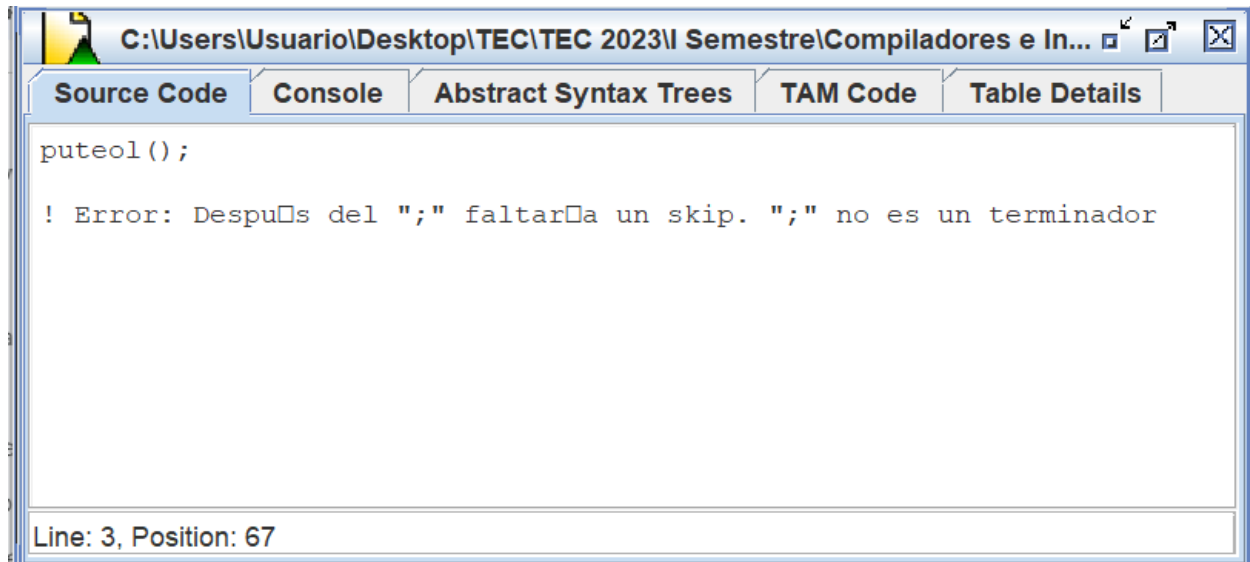
### Resultados esperados

El resultado esperado es que la prueba sea compilada de manera satisfactoria generando los respectivos archivos de salida del proyecto anterior.

### Resultados observados



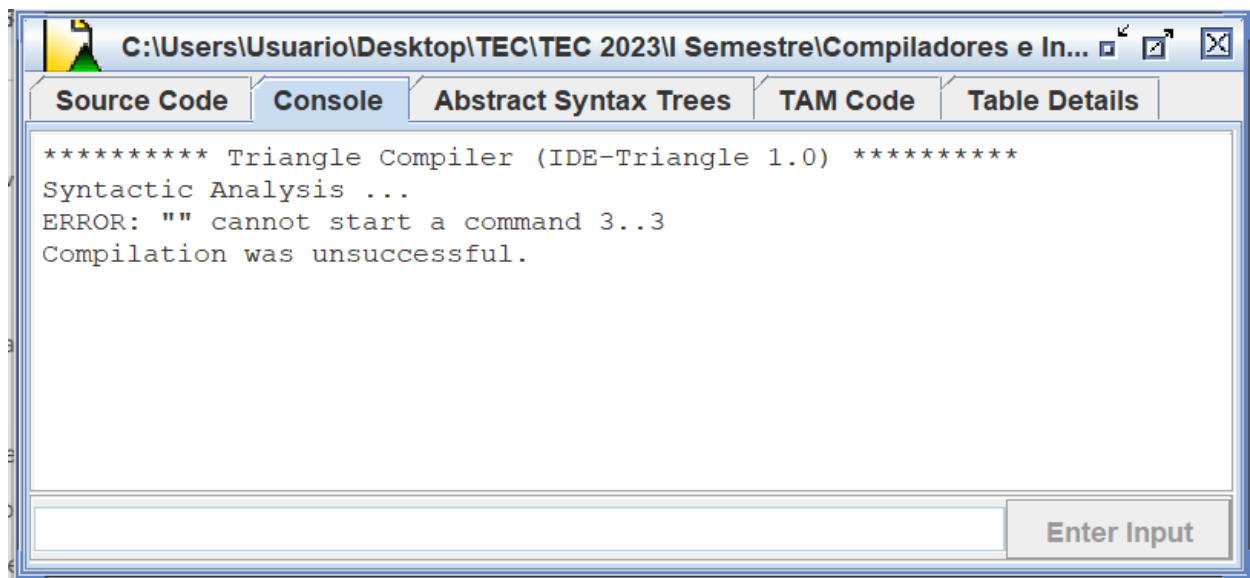
### Dise o de la prueba Err nea



### Resultados esperados

El resultado esperado es que la prueba a la hora de compilar es que encuentre el error de sintaxis y que lo indique la pestaña de consola.

### Resultados observados

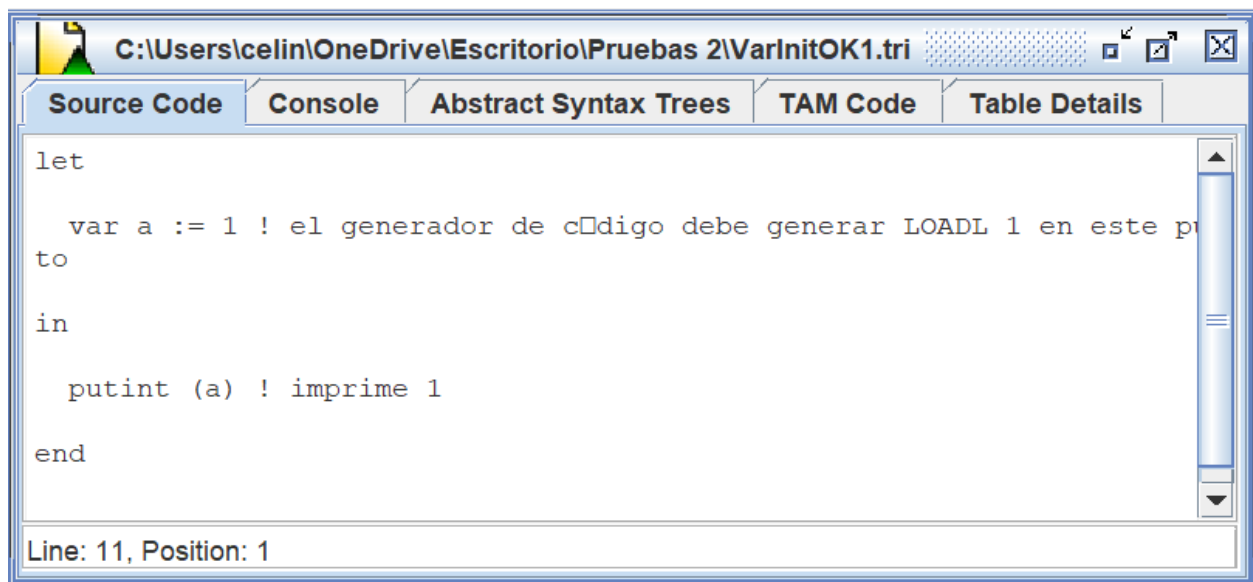


## 9.14 Var

### Objetivo

El objetivo de esta prueba es ver el funcionamiento contextual correcto esperado por el comando var. En estas pruebas se espera representar el funcionamiento del compilador ante un caso de prueba correcto y uno que presente un error.

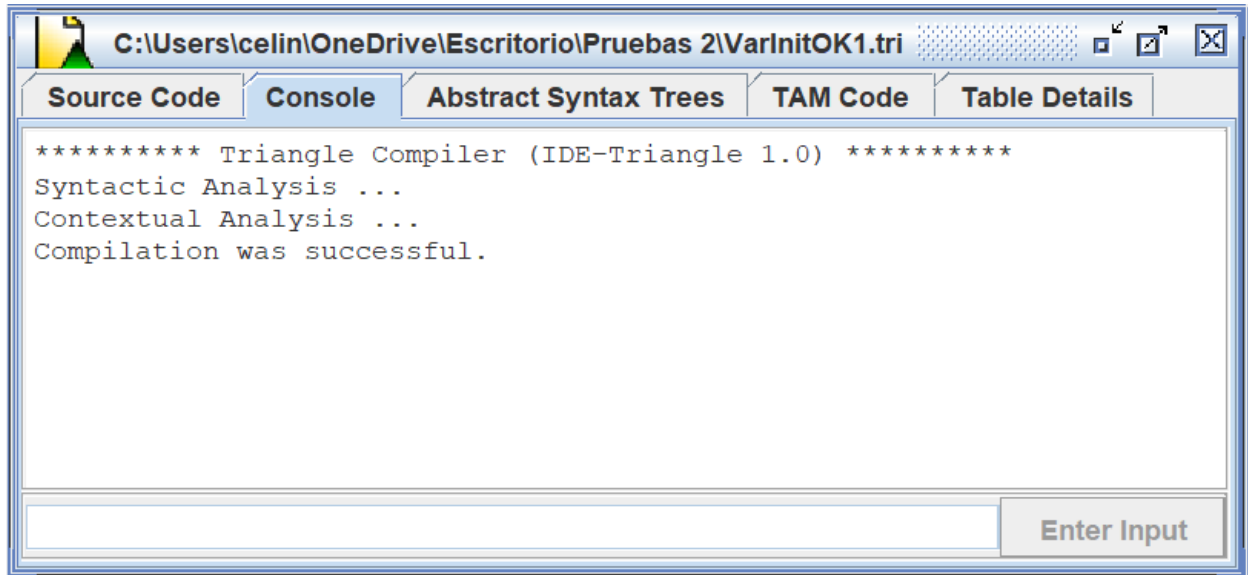
### Diseño de la prueba Correcta



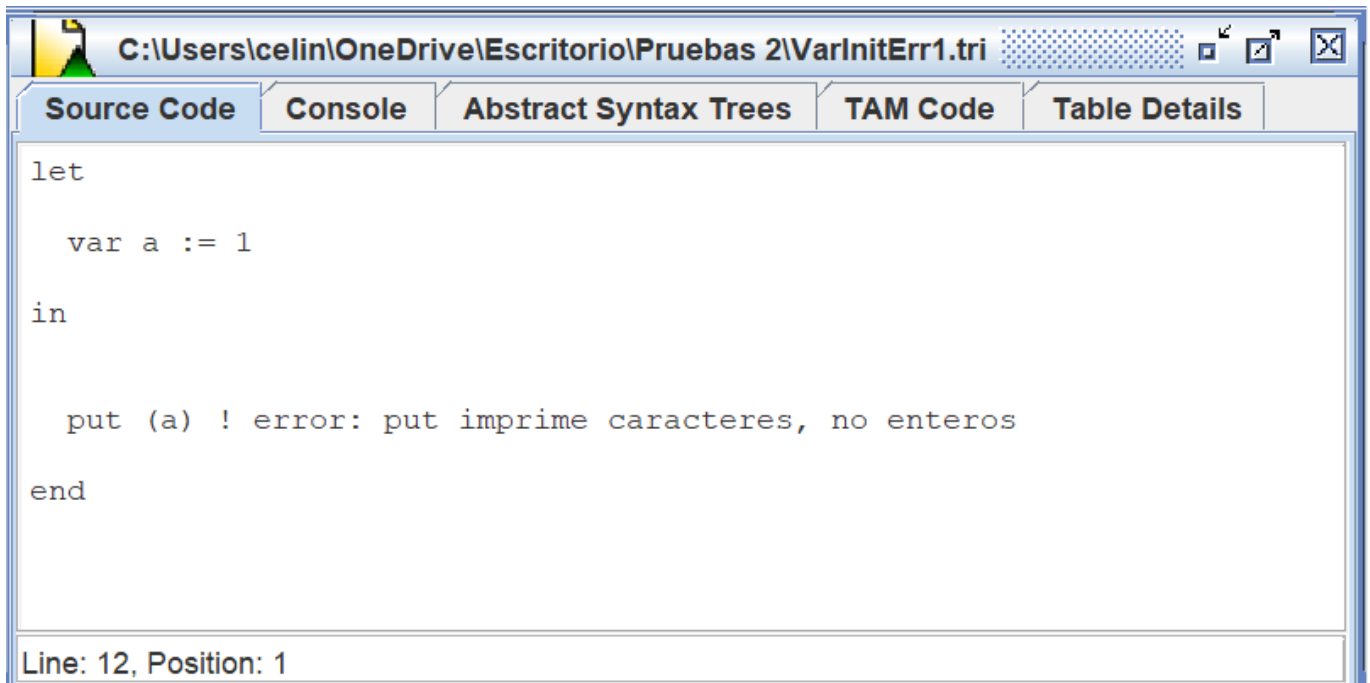
### Resultados esperados

El resultado esperado es que la prueba sea compilada de manera satisfactoria generando los respectivos archivos de salida del proyecto anterior.

### Resultados observados



### Diseño de la prueba Errónea

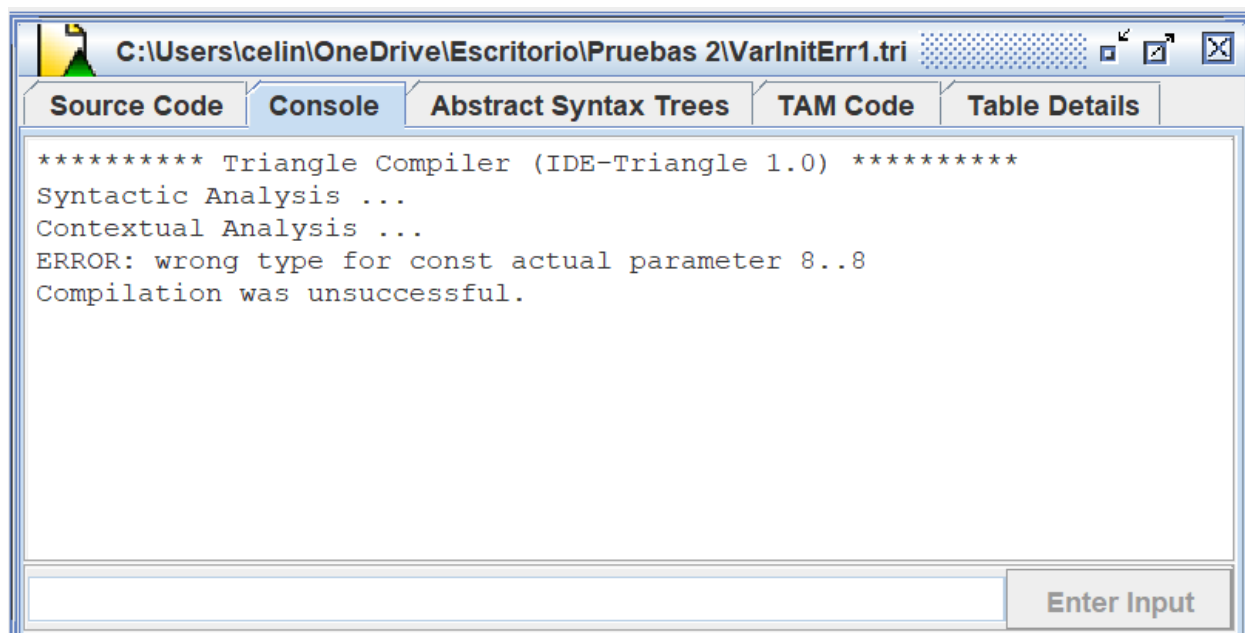


### Resultados esperados

El resultado esperado es que la prueba a la hora de compilar es que encuentre el error de contexto y que lo indique la pestaña de consola.



## Resultados observados

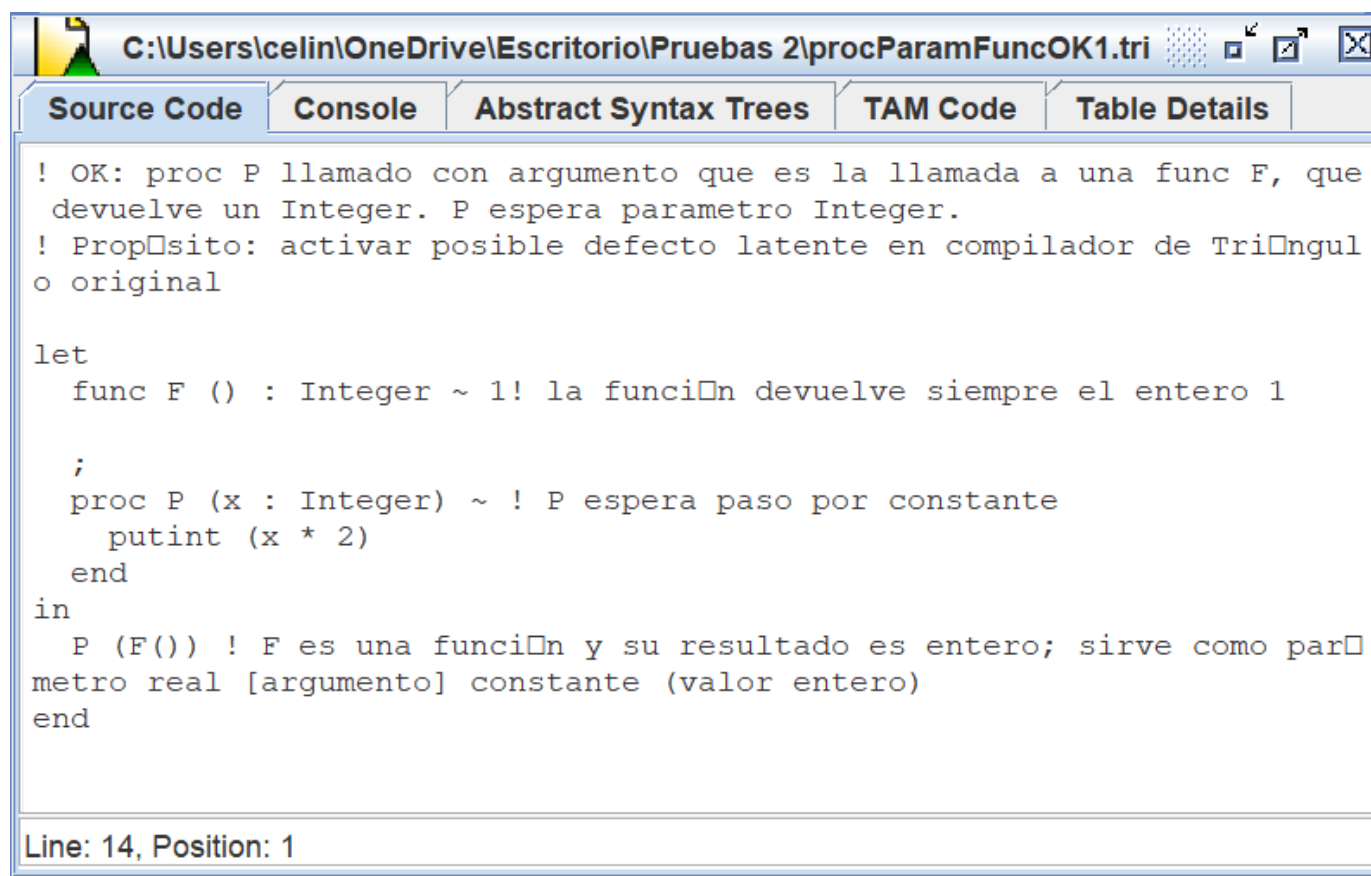


### 9.15 procParamProc

#### Objetivo

El objetivo de esta prueba es ver el funcionamiento contextual correcto esperado por el comando procParamProc. En estas pruebas se espera representar el funcionamiento del compilador ante un caso de prueba correcto y uno que presente un error.

#### Diseño de la prueba Correcta



The screenshot shows a code editor window with the title bar 'C:\Users\celin\OneDrive\Escritorio\Pruebas 2\procParamFuncOK1.tri'. The window has five tabs: 'Source Code' (selected), 'Console', 'Abstract Syntax Trees', 'TAM Code', and 'Table Details'. The 'Source Code' tab displays the following text:

```

! OK: proc P llamado con argumento que es la llamada a una func F, que
  devuelve un Integer. P espera parametro Integer.
! Propósito: activar posible defecto latente en compilador de Triángulo
original

let
  func F () : Integer ~ 1! la función devuelve siempre el entero 1

  ;
  proc P (x : Integer) ~ ! P espera paso por constante
    putint (x * 2)
  end
in
  P (F()) ! F es una función y su resultado es entero; sirve como parámetro
metro real [argumento] constante (valor entero)
end

```

At the bottom of the window, a status bar indicates 'Line: 14, Position: 1'.

### Resultados esperados

El resultado esperado es que la prueba sea compilada de manera satisfactoria generando los respectivos archivos de salida del proyecto anterior.

### Resultados observados

The screenshot shows the Triangle Compiler IDE window titled "C:\Users\celin\OneDrive\Escritorio\Pruebas 2\procParamFuncOK1.tri". The "Console" tab is active, displaying the following output:

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

### Diseño de la prueba Errónea

The screenshot shows the Triangle Compiler IDE window titled "C:\Users\celin\OneDrive\Escritorio\Pruebas 2\procParamProcErr1.tri". The "Console" tab is active, displaying the following output:

```
! Error: proc P llamado con argumento que es la llamada a un proc F. P
  espera parametro Integer
! Propósito: activar posible defecto latente en compilador de Triángulo
  original

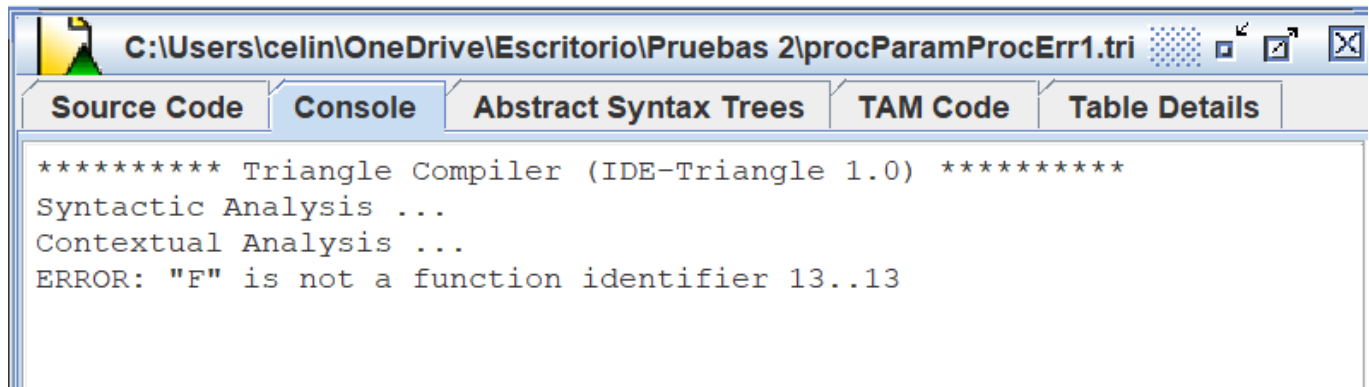
let
  proc F () ~ ! un proc, no es una función
    putint(1)
  end ! Los comandos no entregan valores. Solamente completan su ejecución
    sin retornar valores.
;
  proc P (x : Integer) ~ ! P espera paso por constante
    putint (x * 2)
  end
in
  P (F()) ! Error: los procedimientos no retornan valores. F no es una
    función; es un proc. P espera un argumento de tipo Integer
end
```

Line: 15, Position: 1

### Resultados esperados

El resultado esperado es que la prueba a la hora de compilar es que encuentre el error de contexto y que lo indique la pestaña de consola.

## Resultados observados



```
***** Triangle Compiler (IDE-Triangle 1.0) *****  
Syntactic Analysis ...  
Contextual Analysis ...  
ERROR: "F" is not a function identifier 13..13
```

## 10. Discusión y análisis de los resultados obtenidos

Tomando en cuenta que el objetivo del compilador para esta implementación del proyecto recae en el análisis contextual de los programas, se considera como grupo que se logró esa meta. Lo anterior considerando que según el plan de pruebas suministrado en la sección anterior se logra apreciar la concordancia entre los resultados esperados y los observados, tanto en las pruebas para el flujo correcto y la detección de errores. De igual forma, en la detección de errores, se demuestra, mediante los mensajes de error vistos en la consola del IDE, que el error esperado fue correctamente detectado por el compilador.

## 11. Conclusiones

El planeamiento y desarrollo de este proyecto permitió al equipo de trabajo comprender mucho mejor los detalles relacionados con el análisis contextual del compilador previsto. Este proyecto resultó como buen ejercicio para estudiar, repasar y realmente comprender las distintas reglas contextuales a las que debían de apegarse los programas desarrollados en lenguaje  $\Delta$ tx. Asimismo, el planteamiento de los casos de prueba y su ejecución le permitieron al grupo validar y verificar la correcta aplicación de las reglas sintácticas y contextuales de los distintos casos propuestos.

## 12. Reflexion

La edición de un programa realizada por terceros puede ser complicada y confusa. Es necesario comenzar por comprender la lógica utilizada, la organización de los archivos, los métodos y las estructuras disponibles, así como la forma en que se relacionan entre sí. Las referencias bibliográficas son una gran ayuda para obtener una mejor comprensión de la herramienta. Si hay dudas, este material de apoyo puede proporcionar respuestas o servir como guía. Es fundamental tener claros los objetivos que se desean lograr, y es muy importante realizar pruebas para verificar el funcionamiento de la herramienta durante todo el desarrollo del proyecto.

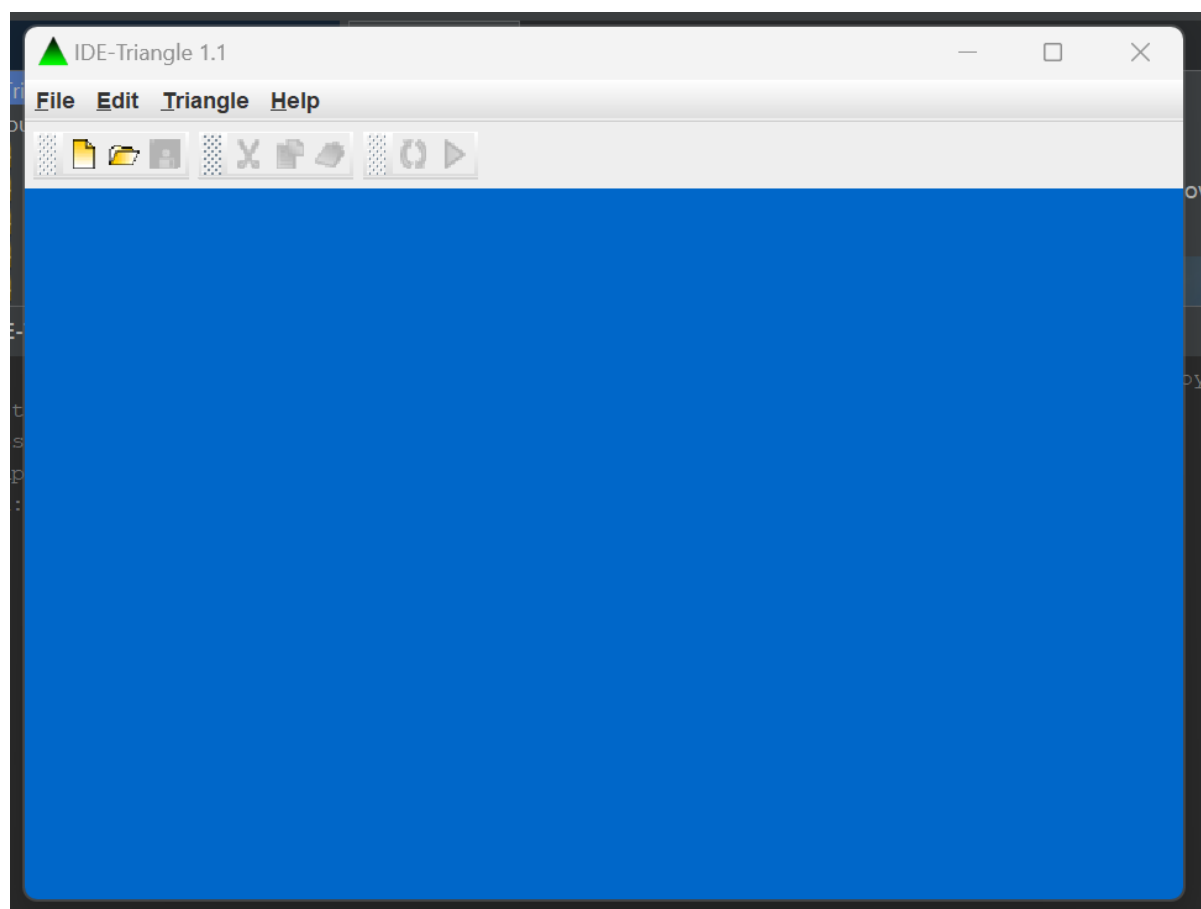
## 13. Descripción resumida de las tareas realizadas por cada miembro del grupo de trabajo

Tarea	Realizado por
Comprobación de todos los tipos para todas las variantes de los comandos repeat ... end.	Gabriel Mora
Manejo de alcance, tipo y protección de for_:=_.._do_end y sus variantes condicionadas (while y until)	María José Porras
Procesamiento de var Id := Exp	Celina Madrigal
Validación de la unicidad de nombres de parámetros en las declaraciones	Gabriel Mora
Procesamiento de la declaración compuesta private	María José Porras
Procesamiento de la declaración compuesta rec	Celina Madrigal
Nuevas rutinas de análisis contextual	Gabriel Mora

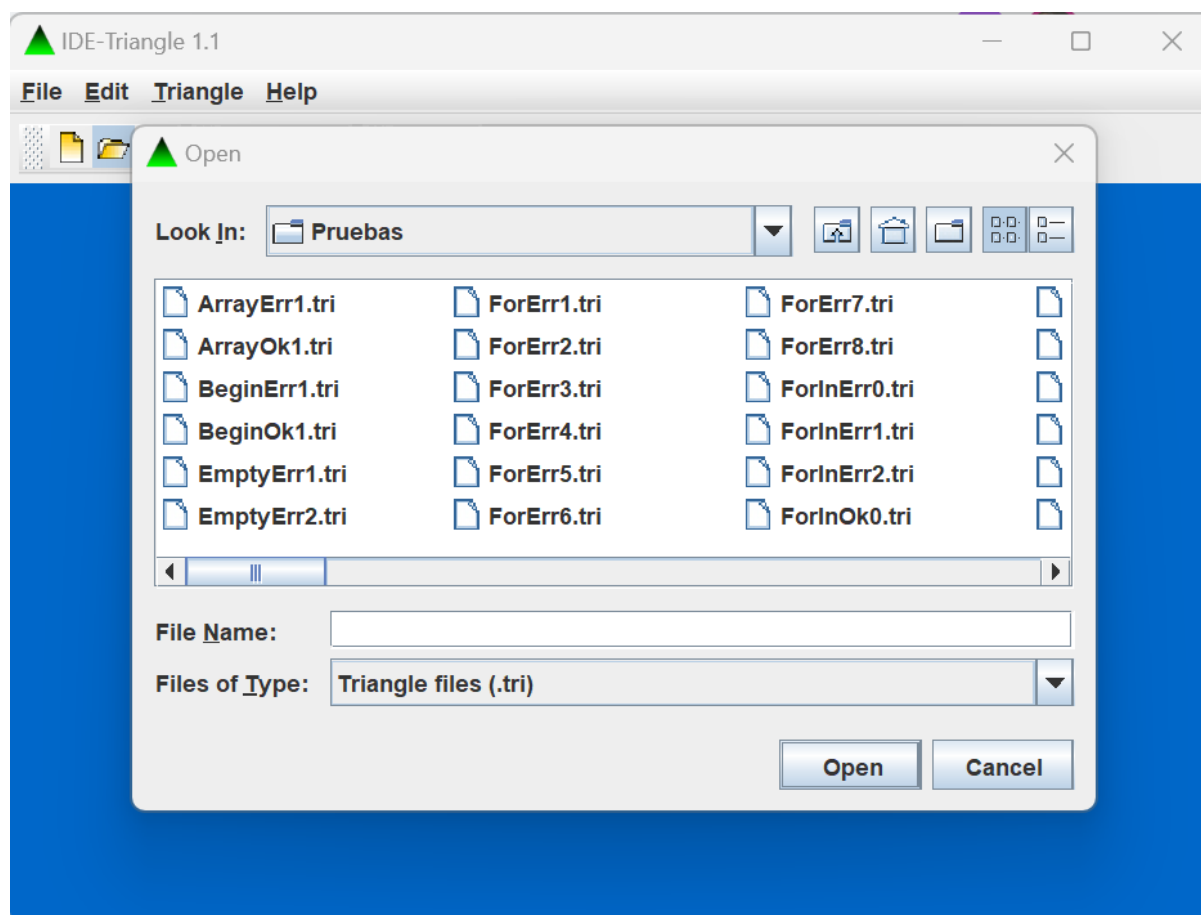
Lista de nuevos errores contextuales detectados	María José Porras
Plan de pruebas para validar el compilador	Celina Madrigal
Documentación	Gabriel Mora, María José Porras y Celina Madrigal

## 14. Cómo debe compilarse el programa

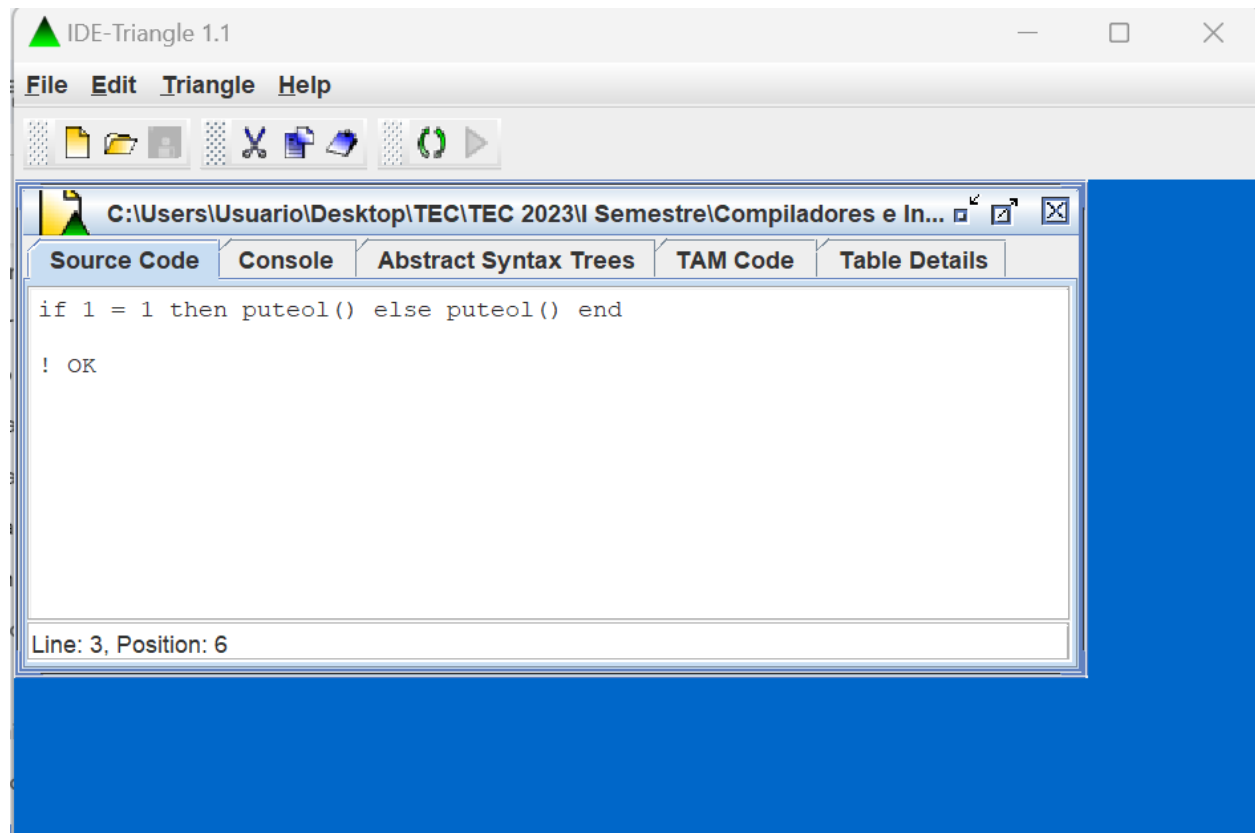
Utilizando el IDE de Apache NetBeans 17 abrimos el proyecto, damos click derecho y damos click en Run. Nos debería aparecer la siguiente pantalla:



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

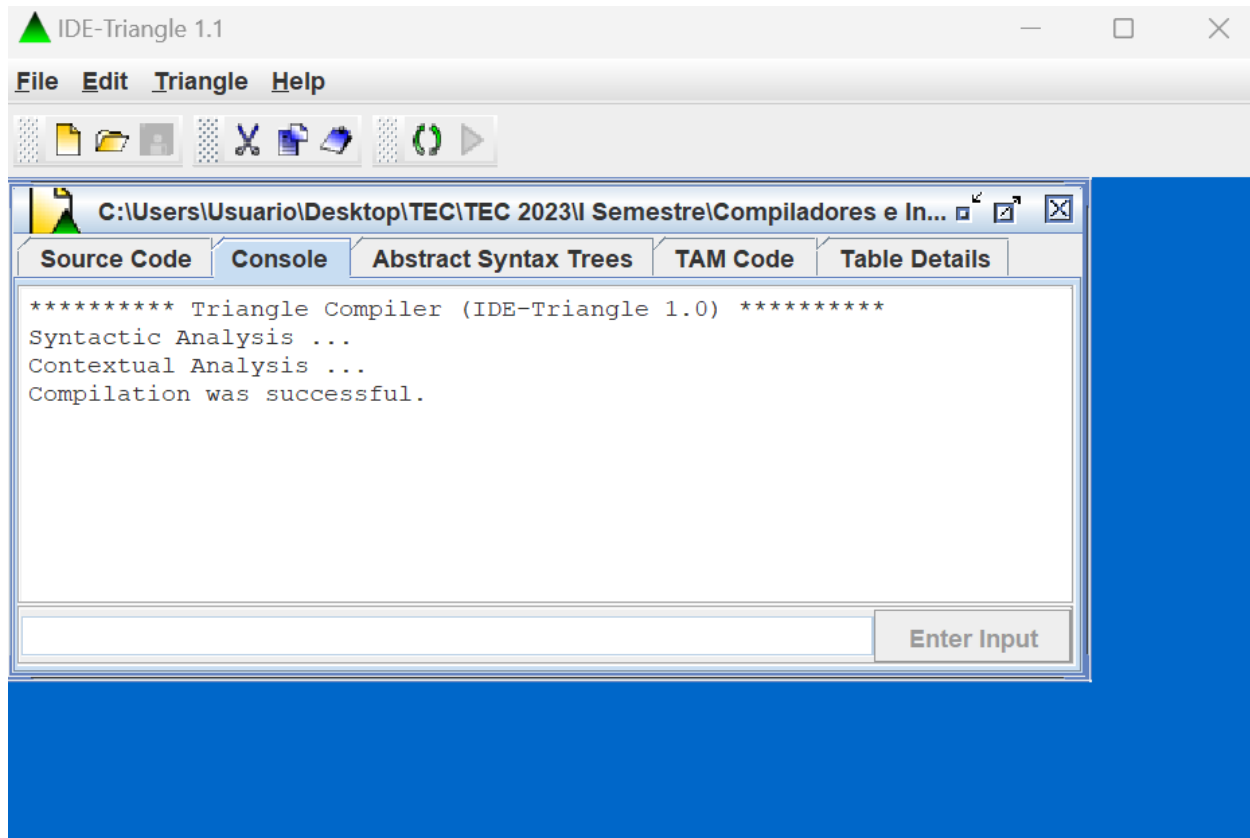


Una vez seleccionada la prueba deseada le damos open y nos debería de aparecer de la siguiente manera:



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

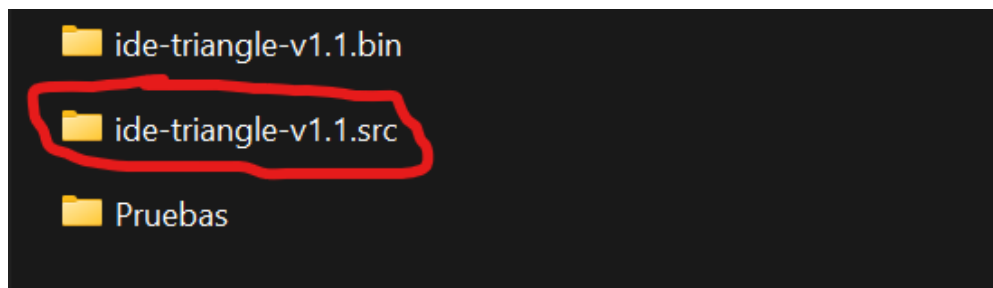




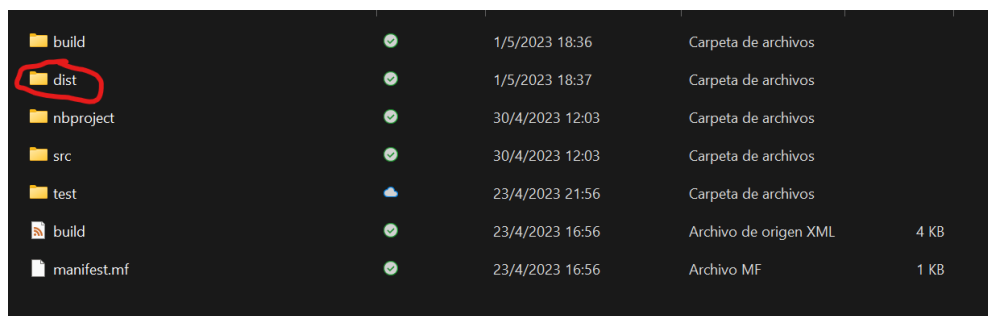
## 15. Cómo debe ejecutarse el programa

Para ejecutar el programa tenemos 2 opciones:

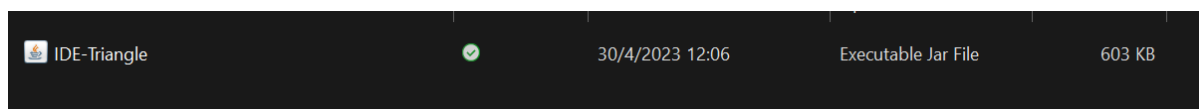
1. Como fue mencionado en el punto anterior, podemos ir al IDE de Apache NetBeans 17, abrimos el proyecto, damos click derecho y damos click en Run.
2. También podemos ejecutarlo mediante el archivo IDE-Triangle.jar, el cual lo encontramos siguiendo estos pasos:
  - En la carpeta del proyecto abrimos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



- Luego abrimos la carpeta llamada dist



- 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



## 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  $\Delta$ . IDE-Triangle – Integración con el compilador de  $\Delta$  [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