Instituto Tecnológico de Costa Rica

Escuela de Ingeniería en Computación

IC-5701: Compiladores e Intérpretes

Ing. Ignacio Trejos Zelaya

Proyecto #1 - Compilador Triángulo Extendido

Analizador Léxico

Analizador Sintáctico

Analizador Contextual

Fecha de entrega: 23 de setiembre del 2017

Estudiantes:

Calvo Navarro, Marvin. 200753649

Salas Bonilla, Jose Antonio. 2015013633

Viquez Murillo, Julio. 2015013680

Sede Central, Cartago

Tabla de Contenidos

1. Introducción

El siguiente documento es una descripción breve de las modificaciones realizadas al lenguaje Triángulo. Se utilizó como base el compilador del lenguaje y el intérprete de la máquina abstracta TAM desarrollado en Java por los profesores David Watt y Deryck Brown. El compilador es una modificación de uno existente, de manera que es capaz de procesar el lenguaje extendido. Además, el compilador coexiste con el ambiente de edición, compilación y ejecución (“IDE”) IDE-Triángulo, desarrollado por el Ing. Luis Leopoldo Pérez (implementado en Java).

Los programas de entrada están escritos en archivos de texto. El usuario selecciona el archivo que contiene el texto del programa fuente desde el ambiente de programación (IDE) base suministrado. Los archivos fuente deben tener terminación .tri.

Importante resaltar que el esquema para el manejo del texto fuente no ha sido modificado, se cuenta con el esquema original.

2. Analizador Léxico.

2.1. Tokens

Se agregaron los siguientes Tokens como palabras reservadas:

- and.
- for.
- local.
- par.
- recursive.
- repeat.
- skip.
- to.
- until.

Los Tokens fueron agregados en orden alfabético, lo cual es importante. Esto implicó en la modificación del orden de las palabras reservadas. De ahora en adelante, la primer palabra reservada será “and” y la última palabra reservada sigue siendo “while”.

Por otra parte, se eliminó la palabra reservada “begin”.

Las adiciones de las nuevas palabras reservadas, tanto como la eliminación de “begin”, se ve reflejado también en la tabla de tokens, donde también se agregaron en orden alfabético las nuevas palabras reservadas y se eliminó “begin”.

3. Analizador Sintáctico.

El lenguaje fuente es una extensión del lenguaje Triángulo, un pequeño lenguaje imperativo con estructura de bloques anidados, que desciende de Algol 60 y de Pascal. Esta extensión de Triángulo agrega diversas formas de comando iterativo, declaración de variables inicializadas, declaración de procedimientos o funciones mutuamente recursivos y otras declaraciones compuestas (colateral y local).

3.1. Cambios a la sintáxis.

3.1.1 Single Command

Se eliminó las siguientes reglas:

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

Se agregaron las siguientes reglas:

```
  "skip"
| "repeat" "while" Expression "do" Command "end"
| "repeat" "until" Expression "do" Command "end"
| "repeat" "do" Command "while" Expression "end"
| "repeat" "do" Command "until" Expression "end"
| "for" "var" Identifier ":=" Expression "to" Expression "do" Command "end"
| "for" "var" Identifier ":=" Expression "to" Expression
      "while" Expression "do" Command "end"
| "for" "var" Identifier ":=" Expression "to" Expression
      "until" Expression "do" Command "end"
| "let" Declaration "in" Command "end"
| "if" Expression "then" Command "else" Command "end"
```

El código fuente del single command se encuentra en la línea 279 de la clase Parser.java

3.1.2 Declaration

Se modifica, de forma que ahora se lea de la siguiente forma:

```
Declaration
        ::=  compound-Declaration
        |    Declaration ";" compound-Declaration
```

La implementación del declaration se encuentra en la línea 691 de la clase Parser.java

3.1.3 Compound Declaration

Se añadió la siguiente regla:

```
compound-Declaration
        ::=  single-Declaration
        |    "recursive" Proc-Funcs "end"
        |    "local" Declaration "in" Declaration "end"
        |    "par" single-Declaration ("and" single-Declaration)+ "end"
```

La implementación del compound declaration se encuentra en la línea 707 de la clase Parser.java

3.1.4 Proc-Func

Se añadió la siguiente regla:

```
Proc-Func
        ::=  "proc" Identifier "(" Formal-Parameter-Sequence ")"
                    "~" Command "end"
        |    "func" Identifier "(" Formal-Parameter-Sequence ")"
                    ":" Type-denoter "~" Expression

Proc-Funcs
        ::=  Proc-Func ("and" Proc-Func)+
```

La implementación del Proc-Func se encuentra en la línea 770 de la clase Parser.java

3.1.5 Single Declaration

Se añadió la siguiente regla:

```
| "var" Identifier ":=" Expression
```

Se modificó la opción referente a proc para que se lea:

```
    …
    |    "proc" Identifier "(" Formal-Parameter-Sequence ")"
                "~" Command "end"
  | …
```

La implementación del single declaration se encuentra en la línea 809 de la clase Parser.java

3.2. Modelajes de árboles sintácticos:

En las declaraciones con múltiples declaraciones en su estructura (véase Proc-Funcs y Par) se implementó una estructura de árboles que permitiese agregar desde 2 declaraciones hasta potencialmente infinitas declaraciones. Para ello se usó el recurso del comando Do{}While(); de Java que permite la inserción de al menos dos declaraciones antes de evaluar

la expresión de condición, hasta que la condición no se cumpla el árbol sigue creciendo añadiendo nodos a la izquierda. Ejemplo (clase: parser línea: 732):

```
case Token.PAR:
{
   acceptIt();
   declarationAST = parseSingleDeclaration();
   do{
       accept(Token.AND);
       Declaration d2AST = parseSingleDeclaration();
       finish(declarationPos);
       declarationAST = new ParDeclaration(declarationAST, d2AST,declarationPos);
   }while (currentToken.kind == Token.AND);
   accept(Token.END);
   finish(declarationPos);
}
break;
```

3.2.1. Recorrido del Visitor

Debido a la agregación de nuevas estructuras sintácticas, se debieron modificar y agregar funcionalidad al Visitor para garantizar el debido recorrido y representación de las estructuras del AST. Para cada nueva estructura del CompoundDeclaration y las nuevas reglas del SingleCommand (for y repeat), se creó una clase que hereda el comportamiento de su tipo, ya sea Declaration ó Command, así por ejemplo para la estructura del local se implementó la siguiente clase (clase: LocalDeclaration):

```
public class LocalDeclaration extends Declaration {

   public LocalDeclaration(Declaration d1AST, Declaration d2AST, SourcePosition thePosition){
       super (thePosition);
       D1 = d1AST;
       D2 = d2AST;
   }
  public Object visit(Visitor v, Object o) {
    return v.visitLocalDeclaration(this, o);
  }
   public Declaration D1;
   public Declaration D2;
}
```

Cada clase nueva del AST requiere una intervención en la clase LayoutVisitor para que esta sea capaz de representar el árbol dentro del IDE. Ejemplo (clase: LayoutVisitor línea: 242):

```
@Override
public Object visitParDeclaration(ParDeclaration ast, Object o) {
    return layoutBinary("Par.Decl.", ast.D1, ast.D2);
}

@Override
public Object visitProcFuncDeclaration(ProcFuncDeclaration ast, Object o) {
    return layoutBinary("Recursive.Decl.", ast.D1, ast.D2);
}

@Override
public Object visitCompoundDeclaration(CompoundDeclaration ast, Object o) {
    return layoutBinary("Compound.Decl.", ast.D1, ast.D2);
}
```

Para la nueva regla de variable inicializada, se procede a crear una clase que hereda el comportamiento de su tipo, en este caso Declaration (clase: VarInitialiteDeclaration), la cual cuenta con dos atributos de tipo: identifier y expression. La implementación de esta clase, requiere de una modificación a la clase LayoutVisitor del paquete TreeDrawer. Ejemplo:

```
@Override
public Object visitVarInitializeDeclaration(VarInitializeDeclaration ast, Object obj) {
   return layoutBinary("VarInit.Decl.", ast.I, ast.E);
}
```

3.3 Reconocimiento de errores

Al agregar nuevos comandos como “for” y “repeat” con tres y cuatros variaciones distintas, su estructura cambia sutilmente, así por ejemplo luego de un “repeat” se pueden esperar tres tokens distintos:”while”, “until” y “do”, a su vez este último posee otras dos variaciones con “while” y “until” luego de la expresión. Esto desencadena nuevos errores no encontrados en la versión original de triángulo. El tratamiento a estos errores se manejó de la siguiente forma (clase: Parser.java línea: 433):

```
default:
    syntacticError("\"%\" invalid syntax, spected \"while\", \"until\" or \"do\" token on",
    currentToken.spelling);
break;
```

4. Analizador Contextual

Basados en las reglas sintácticas del lenguaje Triángulo extendido se definen las restricciones de contexto. Se interpreta las reglas sintácticas desde una perspectiva contextual, abstrayendo estas reglas como árboles de sintaxis abstracta. Para describir los árboles de sintaxis abstracta se utilizan las siguientes meta-variables de las clases indicadas:

• *V*: V-name • *Exp*, Exp_1, Exp_2, Exp_3, Exp_i : Expression • *Com*, Com_1, Com_2 : Command • *Dec*, Dec_1, Dec_2, Dec_3 : Declaration • *Id* : Identifier	• PF_1, PF_2, PF_i, PF_n : Proc-Func • *PFs* : Proc-Func* • *FPS* : Formal-Parameter-Sequence • *TD* : Type-denoter

NOTA: El comando skip no requiere de evaluación contextual.

4.1 Comprobación contextual de las nuevas reglas

4.1.1 repeat … end

repeat while *Exp* **do** *Com* **end**
repeat until *Exp* **do** *Com* **end**
repeat do *Com* **while** *Exp* **end**
repeat do *Com* **until** *Exp* **end**

La comprobación contextual de este comando consiste en verificar que su Exp, sin importar que la Exp pertenezca a un “while” o a un “until”, sea de tipo booleana.

Codigo:

```
public Object visitWhileCommand(WhileCommand ast, Object o) {
  TypeDenoter eType = (TypeDenoter) ast.E.visit(this, null);
  if (! eType.equals(StdEnvironment.booleanType))
    reporter.reportError("Boolean expression expected here", "", ast.E.position);
  ast.C.visit(this, null);
  return null;
}

public Object visitUntilCommand(UntilCommand ast, Object o) {
  TypeDenoter eType = (TypeDenoter) ast.E.visit(this, null);
  if (! eType.equals(StdEnvironment.booleanType))
    reporter.reportError("Boolean expression expected here", "", ast.E.position);
  ast.C.visit(this, null);
  return null;
}
```

4.1.2 for … end

for var *Id* **:=** *Exp1* **to** *Exp2* **do** *Com* **end**

for var *Id* **:=** *Exp1* **to** *Exp2* **while** *Exp3* **do** *Com* **end**
for var *Id* **:=** *Exp1* **to** *Exp2* **until** *Exp3* **do** *Com* **end**

El análisis contextual de este comando consiste en verificar que la Exp1 y la Exp2 sean de tipo enteras, y en caso de que el comando también posea un while o un until, se comprueba que la Exp3 sea de tipo booleana. El manejo del alcance de la variable de control Id se resuelve abriendo un nuevo scope en el For para que esta variable sólo pueda ser visible dentro del mismo.

Codigo:

```
public Object visitForCommand(ForCommand ast, Object o) {
  idTable.openScope();
  ast.D.visit(this, null);
  TypeDenoter eType = (TypeDenoter) ast.E.visit(this, null);
  if (! eType.equals(StdEnvironment.integerType)){
      reporter.reportError("Integer expression expected here", "", ast.E.position);
  }
  ast.C.visit(this, null);
  idTable.closeScope();
  return null;
}

public Object visitForWhileCommand(ForWhileCommand ast, Object o) {
  idTable.openScope();
  ast.D.visit(this, null);
  TypeDenoter eType = (TypeDenoter) ast.E1.visit(this, null);;
  if (! eType.equals(StdEnvironment.integerType)){
      reporter.reportError("Integer expression expected here", "", ast.E1.position);
  } else{
      TypeDenoter e2Type = (TypeDenoter) ast.E2.visit(this, o);
      if (! e2Type.equals(StdEnvironment.booleanType)){
          reporter.reportError("Boolean expression expected here", "", ast.E2.position);
      }
  }
  ast.C.visit(this, null);
  idTable.closeScope();
  return null;
}

public Object visitForUntilCommand(ForUntilCommand ast, Object o) {
  idTable.openScope();
  ast.D.visit(this, null);
  TypeDenoter eType = (TypeDenoter) ast.E1.visit(this, null);;
  if (! eType.equals(StdEnvironment.integerType)){
      reporter.reportError("Integer expression expected here", "", ast.E1.position);
  } else{
      TypeDenoter e2Type = (TypeDenoter) ast.E2.visit(this, o);
      if (! e2Type.equals(StdEnvironment.booleanType)){
          reporter.reportError("Boolean expression expected here", "", ast.E2.position);
      }
  }
  ast.C.visit(this, null);
  idTable.closeScope();
  return null;
}
```

```
public Object visitForVarDeclaration(ForVarDeclaration ast, Object o) {
  idTable.enter (ast.I.spelling, ast);
  if (ast.duplicated)
    reporter.reportError ("identifier \"%\" already declared",
                          ast.I.spelling, ast.position);
  else {
       TypeDenoter eType = (TypeDenoter) ast.E.visit(this, o);
       if (! eType.equals(StdEnvironment.integerType))
            reporter.reportError ("Integer expression expected here", "",
                          ast.E.position);
  }
  return null;
}
```

4.1.3 if … end

if *Exp* **then** Com_1 **else** Com_2 **end**

El análisis contextual de este comando consiste en verificar que la *Exp* debe ser de tipo booleana. Seguidamente, se verifica que *Com1* y *Com2* satisface las restricciones contextuales correspondientes.

Código:

```
public Object visitIfCommand(IfCommand ast, Object o) {
  TypeDenoter eType = (TypeDenoter) ast.E.visit(this, null);
  if (! eType.equals(StdEnvironment.booleanType))
    reporter.reportError("Boolean expression expected here", "", ast.E.position);
  ast.C1.visit(this, null);
  ast.C2.visit(this, null);
  return null;
}
```

4.1.4 Variable Inicializada

var *Id* **:=** *Exp*

El análisis de la declaración de una variable inicializada, el tipo del identificador *Id* se deduce a partir del tipo de la expresión inicializadora *Exp*. El identificador *Id* se exporta al resto del bloque donde aparece esta declaración.

Código:

```
public Object visitVarInitializeDeclaration(VarInitializeDeclaration ast, Object o) {
  ast.I.type = (TypeDenoter) ast.E.visit(this, null);
  idTable.enter(ast.I.spelling, ast);
  if (ast.duplicated)
    reporter.reportError ("identifier \"%\" already declared",
                          ast.I.spelling, ast.position);
  return null;
}
```

4.1.5 Local

local Dec_1 **in** Dec_2 **end**

El análisis contextual de esta declaración consiste en verificar que lo que está declarado en Dec1 sea visible únicamente en la declaración Dec2. La estrategia a aplicar fue ingresar los identificadores del Dec1 en un nuevo scope, decrementar el nivel para ingresar los identificadores de Dec2 y por último cerrar el scope de los identificadores Dec1.

Código:

```
public Object visitLocalDeclaration(LocalDeclaration ast, Object o) {
    idTable.openScope();
    ast.D1.visit(this,null);
    idTable.level--;
    ast.D2.visit(this,null);
    idTable.closeMiddleLevels();
    return null;
}
```

closeMiddleLevels:

```
public void closeMiddleLevels(){
    IdEntry entry = this.latest;
    while(entry.previous != null && entry.previous.level <= this.level){
        entry = entry.previous;
    }
    while(entry.previous!=null && entry.previous.level>this.level){
        entry.previous = entry.previous.previous;
    } }
```

4.1.5 Par

par Dec_1 **and** Dec_2 **end**

El análisis contextual de esta declaración consiste en verificar que los identificadores declarados en Dec1 no sean conocidos en Dec2 ni viceversa, además de limitar a que no se pueden declarar identificadores idénticos. Para resolver este problema se agregó un atributo nuevo a la tabla de identificadores y a cada entry que indique si el identificador pertenece a una

declaración par. Con esta estrategia se marcan los identificadores de par y al momento de realizar la comparación en el retrieve se verifica si son par.

Código:

```
public Object visitParDeclaration(ParDeclaration ast, Object o) {
    State aux = idTable.state;
    idTable.state = State.PAR;
    ast.D1.visit(this, null);
    ast.D2.visit(this, null);
    idTable.state = aux;
    return null;
}
```

Clase State:

```
public enum State {
    NONE,
    PAR,
    RECURSIVE1,
    RECURSIVE2;
}
```

Modificación IdentificationTable:

```
public final class IdentificationTable {

  protected int level;
  protected IdEntry latest;
  protected State state; //0 default, 1 par

  public IdentificationTable () {
    level = 0;
    latest = null;
    state = State.NONE;
  }
```

Modificación IdEntry:

```
public class IdEntry {

  protected String id;
  protected Declaration attr;
  protected int level;
  protected State state;
  protected IdEntry previous;

  IdEntry (String id, Declaration attr, int level, State state, IdEntry previous) {
    this.id = id;
    this.attr = attr;
    this.level = level;
    this.state = state;
    this.previous = previous;
  }
```

Modificación al retrieve:

```
entry = this.latest;
while (searching) {
  if (entry == null)
    searching = false;
  else if (entry.id.equals(id) && !(entry.state == this.state && entry.state==State.PAR)) {
    present = true;
    searching = false;
    attr = entry.attr;
  } else
    entry = entry.previous;
```

4.1.6 Recursive

recursive *PFs* **end**

El análisis contextual de esta declaración consiste en verificar que todas las declaraciones de PFn sean visibles entre ellas. La estrategia para realizar la verificación fue realizar dos pasadas, para ello, se agregó un atributo estado a la tabla de identificadores para saber en qué pasada se encuentra actualmente la tabla, los estados agregados son RECURSIVE1 que indica que se encuentra en la primera pasada y RECURSIVE2 para la segunda pasada. En la primera pasada se agregan a la tabla todos los Id de los PFs y en la segunda se visitan los cuerpos de los PFs.

Codigo:

```
public Object visitProcFuncDeclaration(ProcFuncDeclaration ast, Object o) {
  State currentState = idTable.state;
  if (idTable.state != State.RECURSIVE2){
    idTable.state = State.RECURSIVE1;
    ast.D1.visit(this, null);
    ast.D2.visit(this,null);
  }
  if (currentState != State.RECURSIVE1){
      idTable.state = State.RECURSIVE2;
      ast.D1.visit(this,null);
      ast.D2.visit(this,null);
      idTable.state = currentState;
  }
  return null;
}
```

```
public Object visitFuncDeclaration(FuncDeclaration ast, Object o) {
  State currentState = idTable.state;
  if (idTable.state != State.RECURSIVE2){
      ast.T = (TypeDenoter) ast.T.visit(this, null);
      idTable.enter (ast.I.spelling, ast); // permits recursion
      if (ast.duplicated)
        reporter.reportError ("identifier \"%\" already declared",
                              ast.I.spelling, ast.position);
      if(idTable.state == State.RECURSIVE1){
          idTable.openScope();
          ast.FPS.visit(this, null);
          idTable.closeScope();
      }
  } else idTable.state = State.NONE;
  if (idTable.state != State.RECURSIVE1){
      idTable.openScope();
      ast.FPS.visit(this, null);
      TypeDenoter eType = (TypeDenoter) ast.E.visit(this, null);
      idTable.closeScope();
      if (! ast.T.equals(eType))
        reporter.reportError ("body of function \"%\" has wrong type",
                              ast.I.spelling, ast.E.position);
  }
  idTable.state = currentState;
  return null;
}
```

```
public Object visitProcDeclaration(ProcDeclaration ast, Object o) {
  State currentState = idTable.state;
  if (idTable.state != State.RECURSIVE2){
      idTable.enter (ast.I.spelling, ast); // permits recursion
      if (ast.duplicated)
        reporter.reportError ("identifier \"%\" already declared",
                              ast.I.spelling, ast.position);
      if(idTable.state == State.RECURSIVE1){
          idTable.openScope();
          ast.FPS.visit(this, null);
          idTable.closeScope();
      }
  }else idTable.state = State.NONE;

  if (idTable.state != State.RECURSIVE1){
      idTable.openScope();
      ast.FPS.visit(this, null);
      ast.C.visit(this, null);
      idTable.closeScope();
  }
  idTable.state = currentState;
  return null;
}
```

Enum de state:

```
public enum State {
    NONE,
    PAR,
    RECURSIVE1,
    RECURSIVE2;
}
```

5. Plan de pruebas

Para probar el funcionamiento del compilador, se utilizaron los ejemplos brindados por el profesor Ignacio Trejos Zelaya.

5.1 Léxico

5.1.1 Begin

- Objetivo: Probar la correcta eliminación del Token Begin
- Diseño: BeginErr1.tri

- Resultados esperados: reconocer el “begin” como identificador y no como parte de la gramática establecida del lenguaje
- Resultados observados: Se reconoce el begin como identificador, se espera un token de tipo becomes inmediatamente después de su invocación.

5.2 Sintáctico

4.2.1 ForVarFromToErr1

- Objetivo: Probar la correcta implementación de las nuevas reglas agregadas, del tipo "for var from to".
- Diseño: ForVarFromToErr1.tri
- Resultados esperados: Validar la correcta declaración de la variable , detectando que falta la palabra reservada “var”.
- Resultados observados: Detecta el error, indicando lo siguiente: ERROR: "var" expected here.

5.2.2 ForVarFromToOK2

- Objetivo: Probar la identificación de dos más comandos en la instrucción “for var from to”
- Diseño: ForVarFromToOK2.tri
- Resultados esperados: Análisis sintáctico correcto.
- Resultados observados: Se realiza correctamente el análisis sintáctico identificándose ambos comandos sin error.

5.2.3 ForVarFromToUntilErr2

- Objetivo: Probar la estructura sintáctica del “for var from to until” con un “do” de sobra
- Diseño: ForVarFromToUntilErr2
- Resultados esperados: El analizador sintáctico reconoce un segundo “do”, esperando un comando.
- Resultados Observados: Se advierte de la presencia de un segundo “do” cuando se espera un comando.

5.2.4 ForVarFromToUntilOK1

- Objetivo: Probar la identificación de dos más comandos en la instrucción “for var from to until”
- Diseño: ForVarFromToUntilOK1.tri
- Resultados esperados: Identificación de los dos comandos expresados.

- Resultados observados: Se realiza correctamente el análisis sintáctico identificándose ambos comandos sin error.

5.2.5 ForVarFromToWhileErr1

- Objetivo: Probar la estructura sintáctica del “for var from to while”, con la faltante de una condición.
- Diseño: ForVarFromToWhileErr1.tri
- Resultados esperados: El analizador sintáctico reconoce un “do”, esperando una expresión.
- Resultados observados: Se advierte de la presencia de un segundo “do” cuando se espera una expresión.

5.2.6 ForVarFromToWhileOK1

- Objetivo: Probar el correcto funcionamiento a nivel sintáctico del comando “for var from to while”
- Diseño: ForVarFromToWhileOK1.tri
- Resultados esperados: El analizador sintáctico reconoce sin falta la estructura del “for var from to while”
- Resultados observados: No se evidencia error, la estructura es perfectamente reconocida.

5.2.7 IfErr1

- Objetivo: Probar la estructura sintáctica del “if else”, con el faltante de un “end” (palabra reservada) .
- Diseño: IfErr1.tri
- Resultados esperados: El analizador sintáctico reconoce un “end”, ya sea del if o del else, sin embargo indica que espera otro “end” más.
- Resultados observados: Se advierte que se espera un “end”.

5.2.8 IfOK1

- Objetivo: Poner a prueba el analizador sintáctico con el manejo de los end en un comando if
- Diseño: Ifok1.tri
- Resultados esperados: El analizador sintáctico reconoce exitosamente la estructura de ambos end.
- Resultados observados: No se evidencian errores y la estructura de los end es identificada y correspondida a su respectivo comando.

5.2.9 LetErr1

- Objetivo: Probar la estructura sintáctica del “let”, con la faltante de una declaración.
- Diseño: IfErr1.tri
- Resultados esperados: El analizador sintáctico reconoce un “in”, esperando una declaración.
- Resultados observados: Se advierte de la presencia de un “in”, cuando se espera una declaración. Indica que “in” no puede inicializar una declaración.

5.2.10 LetErrOK2

- Objetivo: Probar la identificación de 2 comandos, uno de ellos el skip
- Diseño: LetErrOK2.tri
- Resultados esperados: El analizador sintáctico identifica sin problemas ambos comandos y ejecuta bien la función del “;”
- Resultados obtenidos: El comando skip es identificado como primer comando, el “;” procede a advertir de otro comando y el putint(a) es identificado correctamente.

5.2.11 LocalErr2

- Objetivo: Probar la estructura sintáctica de “local”, con la faltante de la palabra reservada “end”, en la declaración de local.
- Diseño: LocalErr2.tri
- Resultados esperados: El analizador sintáctico reconoce la palabra “local”, sin embargo notifica que falta la palabra “end” como delimitador de la declaración “local”.
- Resultados observados: Se advierte que se espera un “end”.

5.2.12 LocalOk1

- Objetivo: Probar la identificación de 2 declaraciones, la estructura sintáctica de “local”.
- Diseño: LocalOK1.tri
- Resultados esperados: El analizador sintáctico identifica sin problemas ambas declaraciones.
- Resultados obtenidos: No se evidencian errores y la estructura del local es identificada junto a sus respectivas declaraciones.

5.2.13 RecursiveOk1

- Objetivo: Evaluar el funcionamiento sintáctico de una declaración recursive.

- Diseño: RecursiveOk1.tri
- Resultados esperados: El analizador sintáctico reconoce la estructura del recursive sin problema alguno.
- Resultados obtenidos: No se encontraron errores y el análisis fue exitoso.

5.2.14 RecursiveErr1

- Objetivo: Evaluar el funcionamiento sintáctico de una declaración recursive.
- Diseño: RecursiveErr1.tri
- Resultados esperados: El analizador sintáctico reconoce la estructura del recursive, sin embargo notifica la faltante de la palabra reservada “end” en la segunda llamada de un procedimiento.
- Resultados obtenidos: Se advierte que se espera un “end”.

5.3 Contextual

5.3.1 ForVarFromToErr6

- Objetivo: Evaluar el análisis contextual del comando de repetición for controlado por contador.
- Diseño: ForVarFromToErr6.tri
- Resultados esperados: El analizador contextual evalúa los tipos y el alcance declarados en el comando for. Notifica del uso de una variable no declarada, por lo que no es accesible.
- Resultados obtenidos: Se advierte que la variable no ha sido declarada.

5.3.2 ForVarFromToOK3

- Objetivo: Evaluar el análisis contextual del comando de repetición for controlado por contador.
- Diseño: ForVarFromToOK3.tri
- Resultados esperados: El analizador contextual evalúa los tipos de ambas expresiones (*Exp1* y *Exp2*), así como el tipo de la declaración del identificador *Id* y su alcance. El analizador contextual debe determinar que el identificador *k* sea evaluado una sola vez.
- Resultados obtenidos: No se evidencian errores y la estructura del for es identificada junto a sus respectivas expresiones y comandos.

5.3.3 ForVarFromToUntilErr3

- Objetivo: Evaluar el análisis contextual del comando de repetición for controlado por contador y condición (Until).
- Diseño: ForVarFromToUntilErr3.tri

- Resultados esperados: El analizador contextual evalúa los tipos de las expresiones (*Exp1*, *Exp2* y *Exp3*), así como el tipo de la declaración del identificador *Id* y su alcance (como el de la *Exp3* y *Com*). El analizador contextual debe determinar que la Exp3 no es de tipo booleana.
- Resultados obtenidos: Se advierte que se espera una expresión booleana. Exp3 no es de tipo booleana.

5.3.4 ForVarFromToUntilOK2

- Objetivo: Evaluar el análisis contextual del comando de repetición for controlado por contador y condición (Until y While).
- Diseño: ForVarFromToUntilOK2.tri
- Resultados esperados: El analizador contextual evalúa los tipos de las expresiones (*Exp1*, *Exp2* y *Exp3*), así como el tipo de la declaración del identificador *Id* y su alcance (como el de la *Exp3* y *Com*).
- Resultados obtenidos: No se evidencian errores y la estructura del for es identificada junto a sus respectivas expresiones y comandos.

5.3.5 IfErr5

- Objetivo: Evaluar el análisis contextual del condicional *if*.
- Diseño: IfErr5.tri
- Resultados esperados: El analizador contextual evalúa el tipo de la expresión (*Exp*), así como que *Com1* y *Com2* cumplan con las restricciones contextuales correspondiente. El analizador contextual debe determinar que la Exp no es de tipo booleana.
- Resultados obtenidos: Se advierte que se espera una expresión booleana. Exp no es de tipo booleana.

5.3.6 IfOK1

- Objetivo: Evaluar el análisis contextual del condicional *if*.
- Diseño: IfOK1.tri
- Resultados esperados: El analizador contextual evalúa el tipo de la expresión (*Exp*), así como que *Com1* y *Com2* cumplan con las restricciones contextuales correspondiente.
- Resultados obtenidos: No se evidencian errores y la estructura del if es identificada junto a su respectiva expresión y comandos.

5.3.7 LocalErr3

- Objetivo: Evaluar qué identificadores locales no sean visibles afuera.
- Diseño: LocalErr3.tri
- Resultados esperados: El identificador a no debe ser visible en el comando dentro del “in” del Let

- Resultados obtenidos: Error contextual, el identificador a no es conocido por el putint()

5.3.8 LocalOk1

- Objetivo: Evaluar el análisis contextual de la expresión local y la exportación de la segunda declaración.
- Diseño: LocalOK1.tri
- Resultados esperados: Exportación de la variable “b” en el in del let.
- Resultados obtenidos: El putint(b) es capaz de identificar la variable b declarada en el Local

5.3.9 ParErr3

- Objetivo: Evaluar contextualmente que los identificadores no se repitan.
- Diseño: parErr3.tri
- Resultados esperados: Error contextual, identificador ya declarado en una de las declaraciones del par.
- Resultados obtenidos: El identificador z ya fue declarado en una de las declaraciones del par.

5.3.10 ParOK2

- Objetivo: Evaluar la exportación de identificadores del par.
- Diseño: parOK2.tri
- Resultados esperados: Los comandos del “in” conocen los identificadores del par.
- Resultados obtenidos: Análisis contextual satisfactorio, tanto el if como el putint conocen los identificadores del par.

5.3.11 RecursiveErr5

- Objetivo: Probar en niveles anidados del recursive la visibilidad de los identificadores.
- Diseño: recursiveErr5.tri
- Resultados esperados: Los identificadores d1 y d2 no deben ser vistos por el segundo procedimiento del recursive anidado.
- Resultados obtenidos: El segundo procedimiento no identifica el d1 y d2 por lo cual se da error de identificadores no declarados.

5.3.12 RecursiveOk5

- Objetivo: Evaluar sintácticamente el recursive y su comportamiento con declaraciones anidadas.
- Diseño: recursiveOK5.tri

- Resultados esperados: El análisis contextual se ejecuta satisfactoriamente, tomando en cuenta el alcance de los recursive anidados.
- Resultados obtenidos: No hubo problemas de alcance con los procedimientos “despedirse” y “saludar”.

5.3.13 RepeatDoUntilErr1

- Objetivo: Evaluar contextualmente el comando repetitivo *repeat* y sus variaciones.
- Diseño: RepeatDoUntilErr1.tri
- Resultados esperados: El análisis contextual no se ejecuta satisfactoriamente. El análisis contextual evalúa la expresión *Exp* y su tipo, el cual debe de ser booleano.
- Resultados obtenidos: Se notifica de que se espera una expresión *Exp* de tipo booleana. *Exp* no es booleana.

5.3.14 RepeatDoUntilOK1

- Objetivo: Evaluar contextualmente el comando repetitivo *repeat* y sus variaciones.
- Diseño: RepeatDoUntilOK1.tri
- Resultados esperados: El análisis contextual se ejecuta satisfactoriamente. El análisis contextual evalúa la expresión *Exp* y su tipo, el cual debe de ser booleano.
- Resultados obtenidos: Análisis contextual satisfactorio. La estructura del comando repetitivo *repeat* en la variación específica cumple con las restricciones contextuales declaradas.

5.3.15 RepeatDoWhileErr3

- Objetivo: Evaluar contextualmente el comando repetitivo *repeat* y sus variaciones.
- Diseño: RepeatDoWhileErr3.tri
- Resultados esperados: El análisis contextual no se ejecuta satisfactoriamente. El análisis contextual evalúa la expresión *Exp* y su tipo, el cual debe de ser booleano.
- Resultados obtenidos: Se notifica de que se espera una expresión *Exp* de tipo booleana. *Exp* no es booleana.

5.3.16 RepeatDoWhileOK1

- Objetivo: Evaluar contextualmente el comando repetitivo *repeat* y sus variaciones.
- Diseño: RepeatDoWhileOK1.tri

- Resultados esperados: El análisis contextual se ejecuta satisfactoriamente. El análisis contextual evalúa la expresión *Exp* y su tipo, el cual debe de ser booleano.
- Resultados obtenidos: Análisis contextual satisfactorio. La estructura del comando repetitivo *repeat* en la variación específica cumple con las restricciones contextuales declaradas.

5.3.17 RepeatWhileDoErr4

- Objetivo: Evaluar contextualmente el comando repetitivo *repeat* y sus variaciones.
- Diseño: RepeatWhileDoErr4.tri
- Resultados esperados: El análisis contextual no se ejecuta satisfactoriamente. El análisis contextual evalúa la expresión *Exp* y su tipo, el cual debe de ser booleano.
- Resultados obtenidos: Se notifica de que se espera una expresión *Exp* de tipo booleana. *Exp* no es booleana. Por lo tanto se presenta un error de tipo.

5.3.18 RepeatWhileDoOK2

- Objetivo: Evaluar contextualmente el comando repetitivo *repeat* y sus variaciones.
- Diseño: RepeatWhileDoOK2.tri
- Resultados esperados: El análisis contextual se ejecuta satisfactoriamente. El análisis contextual evalúa la expresión *Exp* y su tipo, el cual debe de ser booleano.
- Resultados obtenidos: Análisis contextual satisfactorio. La estructura del comando repetitivo *repeat* en la variación específica cumple con las restricciones contextuales declaradas.

5.3.19 RepeatUntilDoErr1

- Objetivo: Evaluar contextualmente el comando repetitivo *repeat* y sus variaciones.
- Diseño: RepeatUntilDoErr1.tri
- Resultados esperados: El análisis contextual no se ejecuta satisfactoriamente. La estructura del comando *repeat* es incorrecta, falta la palabra reservada “*end*”.
- Resultados obtenidos: Se notifica de que se espera la palabra reservada “*end*”.

5.3.20 RepeatUntilDoOK2

- Objetivo: Evaluar contextualmente el comando repetitivo *repeat* y sus variaciones.
- Diseño: RepeatUntilDoOK2.tri
- Resultados esperados: El análisis contextual se ejecuta satisfactoriamente. El análisis contextual evalúa la expresión *Exp* y su tipo, el cual debe de ser

booleano. Permite declarar múltiples comandos separados por punto y coma (“;”).

- Resultados obtenidos: Análisis contextual satisfactorio. La estructura del comando repetitivo *repeat* en la variación específica cumple con las restricciones contextuales declaradas.

6. Conclusión

Las modificaciones al analizador léxico, sintáctico y contextual, así como la adición de nuevas reglas gramaticales al lenguaje Triangle fueron realizadas en su totalidad. Con esto, se logró potenciar las posibilidades que Triangle puede ofrecer al programador mediante el uso de nuevos comandos, como el “for” y “until”, así como nuevas formas de declaraciones como el “recursive”, “local” y “par”. Todo esto brinda a un programador de Triangle mayor ventaja al utilizar menor cantidad de instrucciones para programar un algoritmo.

7. Indicaciones de ejecución

Para iniciar el IDE modificado de Triángulo se debe ingresar a la carpeta IDE-Triangle++, luego ingresar a la carpeta bin y ejecutar “IDE-Triangle.jar”. La carpeta src contiene todo el código fuente del IDE-Triangle distribuido en sus carpetas correspondientes.

El código fuente del intérprete de triángulo se encuentra en la carpeta Triangle++ distribuido en su respectivas carpetas.

En la carpeta Pruebas se encuentran todos los archivos de prueba oficiales y no oficiales con los que se probó el compilador.

8. Reflexiones

Se pudo admirar toda la complejidad que conlleva la implementación de un intérprete o compilador, inclusive siendo este implementado en un lenguaje de programación de alto nivel como es el caso del intérprete de Triangle, que esta implementado en el lenguaje Java, esto nos hace pensar el gran reto ingenieril que supuso la creación del primer compilador para un lenguaje de alto nivel en 1954, el compilador de FORTRAN.

9. Tareas Realizadas

Nombre	Tareas

Marvin Calvo Navarro	- Modificación de la clase Token.
	  - Se agregó nuevas palabras reservadas.
	  - Se eliminó la palabra reservada “begin”.
	- Modificación de la clase singleDeclaration, específicamente en el token.VAR.
	  - Se agregó la regla variable inicializada.
	  - Se modificó la regla referente a “proc”.
	- Modificación de la interfaz Visitor.
	  - Se agregó la declaración del método abstracto visitVarInitializeDeclaration (correspondiente a la nueva regla).
	- Modificación del paquete Triangle.AbstractSyntaxTrees.
	  - Se agregó la clase VarInitializeDeclaration que hereda de la clase abstracta Declaration (correspondiente a la nueva regla).
	- Implementación del método visitVarInitializeDeclaration en la clase LayoutVisitor.
	- Implementación de los métodos abstractos agregados a la interfaz Visitor, en la clase TreeVisitor del IDE-Triangle.
	- Implementación del método abstracto agregado visitVarInitializeDeclaration a la interfaz Visitor, en la clase TableVisitor del IDE-Triangle.
	- Implementación de pruebas varias al compilador.
	- Análisis contextual
	  - Declaración de variable inicializada.
	  - Modificación a visitSimpleVname, para agregar los procesamientos de:
	  - VarInitializeDeclaration.
	  - ForVarDeclaration

	- Creación y ejecución de casos de prueba para verificar el análisis contextual de las reglas agregadas.

José Salas	- Modificación de la clase Declaration
	  - Invocación de los compound declaration en lugar de single declaration
	- Creación de compound Declaration
	  - Implementación sintáctica del Recursive
	  - Implementación sintáctica del Local
	  - Implementación sintáctica del Par
	  - Implementación de la multiplicidad de los single declaration
	- Creación de Proc Funcs
	  - Implementación de la multiplicidad de ProcFunc
	  - Implementación del Proc
	  - Implementación del Func
	- Creación de clases hijas del AST
	  - Implementación de LocalDeclaration.java
	  - Implementación de ParDeclaration.java
	  - Implementación ProcFuncDeclaration.java
	- Implementación de los visit de las estructuras anteriormente mencionadas en el LayoutVisitor.java
	- Implementación contextual del Par
	- Modificación a la IdentificationTable y IdEntry

Julio Víquez	- Modificación de la clase single-command
	  - Eliminación de comandos: vacío y begin
	  - Modificación de comandos: while, let, if
	  - Adición de comandos: skip, for, until, forWhile, forUntil
	- Implementación de los métodos visit en la clase LayoutVisitor
	- Creación del Triangle.jar
	- Implementación de los métodos abstractos agregados a la interfaz Visitor en la clase

	TreeVisitor del IDE-Triangle.
	- Implementacion de metodos abstractos en el TreeDrawer del IDE-Triangle
	- Análisis contextual
	  - Repeat command
	  - For command
	  - For While command
	  - For Until command
	  - For Var declaration
	  - Local declaration
	  - Recursive declaration
	- Creación de algunos ejemplos para verificar el análisis contextual de las reglas agregadas
	- Parte de la modificación a la tabla de identificadores