Triangle^{+} Compiler
Lexical and Syntactical Analyzer

1st Campos-Espinoza, Jose
2016098975
Student
Costa Rica Institute of Technology
Cartago, Costa Rica
Email: jocampos@ic-itcr.ac.cr

2nd Miranda-Arias, Andres
2017075170
Student
Costa Rica Institute of Technology
Cartago, Costa Rica
Email: anmiranda@ic-itcr.ac.cr

3rd Molina-Brenes, Ricardo
2017239524
Student
Costa Rica Institute of Technology
Cartago, Costa Rica
Email: rmolina@ic-itcr.ac.cr

4th Padilla-Jenkins, Israel
2017097351
Student
Costa Rica Institute of Technology
Cartago, Costa Rica
Email: ipadilla@ic-itcr.ac.cr

***Abstract*—Based on a previously existing Triangle Δ compiler, an extension to the language was made, resulting in Δ^{+}. Different changes were made to the syntactical and lexical analyzer of Watt and Brown's compiler. The different test cases and results are shown as implemented, as well as the different modifications to the compiler in Java. Different syntactical rules were deleted, others modified or added, particularly in the *Declaration* and *Command* parsing. All changes made were done respecting the existing coding format and style, implemented by Watt and Brown originally, as well as the changes made by Dr. Luis Pérez. Overall, all the syntactic and lexical rules deleted, added and modified were made successfully; the compiler fails to compile where needed, and compiles successfully when correct syntax is used. As well, the abstract syntax trees (ASTs) were drawn correctly in Dr. Pérez's IDE, thus correctly completing the project's scope.**

***Index Terms*—Compiler, Triangle, Triangle Extended, TAM, Triangle Abstract Machine, Abstract Syntax Trees**

Contents

I. INTRODUCTION

The objective of this project is to understand the details of the lexical and syntactical analysis phases of a compiler according to the techniques exposed by David A. Watt and Deryck F. Brown in their book *Programming Language Processors in Java: Compilers and Interpreters*[1]. The compiler base used was for the Triangle Δ language, developed by Watt and Brown. The aim and scope of this project is to modify an existing compiler, in order to process and extended version of Triangle, Δ^+. In addition, the final result of the modified compiler must be able to coexist with an integrated development environment (IDE), developed in Java by Dr. Luis Leopoldo Pérez[1], and previously adjusted by Computer Engineering students from Tecnológico de Costa Rica. [2]

II. METHODOLOGY

A. Changes made to Syntactical Analyzer

The first changes made to the compiler provided were in the language syntax. These changes were made to the keywords; some were deleted and others were added. The keyword deleted was ”begin”. On the other hand, the keywords added were the following:

```
and, for, init, local, loop, recursive,
   repeat, skip, to, until
```

These words were added as new alternatives for the Δ^+ extension of the language, specifically in the class *Token*. Also the indexes for the words in the dictionary were adjust and added in alphabetical order to achieve the rules of the class.

In addition, some changes were made properly in the syntax of the language, as described on the project specification [2]. The first change made was to eliminate from *single-Command* the empty command option. Second, the following alternatives were deleted from *single-Command*:

[1]Dr. Luis Leopoldo Pérez, luiperpe@ns.isi.ulatina.ac.cr

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

As well, the following alternatives of *single-Command* remained the same:

```
single-Command ::=
    | V-name ":=" Expression
    | Identifier "("
      Actual-Parameter-Sequence ")"
```

To implement different changes in the alternatives for *single-Command*, the following Java function was implemented with an internal switch case according to the current token in the parser

```
Command parseSingleCommand() throws
    SyntaxError {
   Command commandAST = null; // in case
      there's a syntactic error

   SourcePosition commandPos = new
      SourcePosition();
   start(commandPos);

   switch (currentToken.kind)
```

The following additions were made to the *single-Command* alternatives:

```
single-Command ::=
   "skip"
```

The Java implementation of this specific case was the following:

```
//SKIP case,skips the command.
case Token.SKIP:
{
   acceptIt();
   finish(commandPos);
   commandAST = new
      EmptyCommand(commandPos);
}
```

```
break;
```

Since skip is a keyword for this extended version of Triangle, the parser must accept the token using the acceptIt() function, and finish parsing the command, evaluating it as an EmptyCommand().

The next addition to *single-Command* was:

```
  | "loop" "while" Expression "do"
    Command "repeat"
```

with its respective Java implementation:

```
case Token.LOOP:
{
   acceptIt();
   switch(currentToken.kind)
   {
      //while, new according to
          specification
      case Token.WHILE:
      {
         acceptIt();
         Expression e1AST =
             parseExpression();
         accept(Token.DO);
         Command c1AST = parseCommand();
         accept(Token.REPEAT);
         finish(commandPos);
         commandAST = new
             WhileCommand(e1AST, c1AST,
             commandPos);
      }
      break;
```

In this particular alternative, the "loop" token is accepted by acceptIt(), then if the next token is a "while", it enters that particular case. It is accepted, the first expression is parsed by parseExpression(), the token "do" is accepted, the command is parsed by parseCommand(), the token "repeat" is accepted as well and the command is established as a WhileCommand.

The next addition was:

```
  | "loop" "until" Expression "do"
    Command "repeat"
```

with its respective Java implementation:

```
//until, new according to specification
case Token.UNTIL:
{
   acceptIt();
   Expression e1AST = parseExpression();
   accept(Token.DO);
   Command c1AST = parseCommand();
   accept(Token.REPEAT);
   finish(commandPos);
   commandAST = new UntilCommand(e1AST,
       c1AST, commandPos);
}
break;
```

Here, the "loop" token is accepted, then the "until" token. Following, the expression is parsed, the token "do" is accepted and the command is parsed. Finally, the command is established as an UntilCommand.

The next addition made was:

```
  | "loop" "do" Command "while"
    Expression "repeat
```

with its respective Java implementation:

```
//DO case
case Token.DO:
{
   acceptIt();
   Command c1AST = parseCommand();
   switch(currentToken.kind)
   {
      case Token.WHILE:
      {
         acceptIt();
         Expression e1AST =
             parseExpression();
         accept(Token.REPEAT);
         finish(commandPos);
         commandAST = new
             DoWhileCommand(c1AST, e1AST,
             commandPos); //DoWhileCommand
      }
      break;
```

Here, since "loop" "do" has two different options, another switch case is implemented, but first, the command is parsed. In the first case, "loop" "do", both tokens are accepted, the command is parsed and the "while" token is accepted. The expression is parsed as well and the "repeat" token is accepted. The final command established is a DoWhileCommand.

The next addition made was:

```
  | "loop" "do" Command "until"
     Expression "repeat"
```

with its respective Java implementation:

```
case Token.UNTIL:
{
   acceptIt();
   Expression e1AST = parseExpression();
   accept(Token.REPEAT);
   finish(commandPos);
   commandAST = new DoUntilCommand(c1AST,
      e1AST, commandPos);
      //DoUntilCommand
}
break;
```

Here, the ”loop”, ”do” and ”until” tokens are accepted, and the command parsed. Following, the expression is parsed and the ”repeat” token is accepted. The final command established is a DoUntilCommand.

The next addition made was:

```
  | "loop" "for" Identifier "~"
     Expression "to" Expression "do"
     Command "repeat"
```

with its respective Java implementation:

```
case Token.FOR:
{
   acceptIt();
   Identifier iAST = parseIdentifier();
   accept(Token.IS);
   Expression eAST= parseExpression();
   accept(Token.TO);
   Expression e2AST=parseExpression();
   accept(Token.DO);
   commandAST = parseCommand();
   accept(Token.REPEAT);
   finish(commandPos);
   commandAST = new ForDoCommand(iAST,
      eAST, e2AST, commandAST,
      commandPos);//for command
}
break;
```

Here, the ”loop” and ”for” tokens are accepted. The identifier is parsed with parseIdentifier(), the ”is” token (∼) is accepted as well. Both expressions are parsed into eAST and e2AST, and the ”to” token is accepted. Last, the ”do” token is accepted and the command is parsed. The final command established is a ForDoCommand.

The next addition made was:

```
  | "let" Declaration "in" Command "end"
```

with its respective Java implementation:

```
//LET case, modified according to
   specification
case Token.LET:
{
   acceptIt();
   Declaration dAST = parseDeclaration();
   accept(Token.IN);
   Command cAST = parseCommand();
   accept(Token.END);
   finish(commandPos);
   commandAST = new LetCommand(dAST,
      cAST, commandPos);
}
break;
```

Here, the ”let”, ”in” and ”end” tokens are accepted. The declaration is parsed with parseDeclaration(), the command is parsed and the final command established is a LetCommand.

The final addition made to *single-Command* was:

```
  | "if" Expression "then" Command
     "else" Command "end" |
```

with its respective Java implementation:

```
//IF case, modified according to
   specification
case Token.IF:
{
   acceptIt();
   Expression eAST = parseExpression();
   accept(Token.THEN);
   Command c1AST = parseCommand();
   accept(Token.ELSE);
   Command c2AST = parseCommand();
   accept(Token.END);
   finish(commandPos);
   commandAST = new IfCommand(eAST,
      c1AST, c2AST, commandPos);
}
break;
```

This final case within the initial switch, established that the ”if”, ”then”, ”else” and ”end” tokens are accepted. An expression and two different commands are parsed, and the final command established into the commandAST is an IfCommand.

After updating the *single-Command*, the next big modification was made to *Declaration*. Here, *Declaration* was updated to read the following:

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

and it was implemented in Java as follows:

```
// parseDeclaration modified on 10/14/19
   by andres.mirandaarias@gmail.com
Declaration parseDeclaration() throws
   SyntaxError {
   Declaration declarationAST = null; //
      in case there's a syntactic error

   SourcePosition declarationPos = new
      SourcePosition();
   start(declarationPos);
   declarationAST =
      parseCompoundDeclaration();
   while (currentToken.kind ==
      Token.SEMICOLON) {
      acceptIt();
      Declaration dAST2 =
         parseCompoundDeclaration();
      finish(declarationPos);
      declarationAST =
      new
      SequentialDeclaration(declarationAST,
      dAST2, declarationPos);
   }
   return declarationAST;
}
```

This modification states that the *Declaration* can be one or more *compound-Declaration*, creating a new SequentialDeclaration abstract syntax trees, with the first *compound-Declaration* in one branch, and the *Declaration* in the other.

Since *compound-Declaration* was added in the Δ^+ compiler, it had to be implemented in the parser.

```
compound-Declaration
   ::= single-Declaration
   |  "recursive" Proc-Funcs "end"
   |  "local" Declaration "in"
      Declaration "end"
```

The function parseCompoundDeclaration() was implemented in Java as follows:

```
// parseCompoundDeclaration added on
   10/14/19 by
   andres.mirandaarias@gmail.com
Declaration parseCompoundDeclaration()
   throws SyntaxError{
   Declaration declarationAST = null;
   SourcePosition declarationPos = new
      SourcePosition();
   start(declarationPos);
   switch(currentToken.kind){
      case Token.CONST:
      case Token.VAR:
      case Token.PROC:
      case Token.FUNC:
      case Token.TYPE:
         declarationAST =
            parseSingleDeclaration();
      break;
      case Token.RECURSIVE:
      {
         acceptIt();
         declarationAST =
            parseProcFuncs();
         accept(Token.END);
         finish(declarationPos);
         declarationAST = new
            RecursiveDeclaration(
            declarationAST,
            declarationPos);
      }
      break;
      case Token.LOCAL:
      {
         acceptIt();
         Declaration dAST1 =
            parseDeclaration();
         accept(Token.IN);
         Declaration dAST2 =
            parseDeclaration();
         accept(Token.END);
         finish(declarationPos);
         declarationAST = new
            LocalDeclaration(dAST1,
            dAST2, declarationPos);
      }
      break;
      default:
        syntacticError("\"%\" error parsing
            compound-Declaration",
            currentToken.spelling);
        break;
    }
    return declarationAST;
}
```

Since *compound-Declaration* has three different alternatives, and *single-Declaration* was already implemented, only the option for

```
|   "recursive" Proc-Funcs "end"
|   "local" Declaration "in" Declaration
    "end
```

had to be added and implemented. In these particular cases, RecursiveDeclaration and LocalDeclaration were not implemented as abstract syntax trees, thus, it was required to be implemented from zero. Specifically, for RecursiveDeclaration, the following Java class was added under the package Triangle.AbstractSyntaxTrees:

```
//Class added on 10/14/19 by
    andres.mirandaarias@gmail.com
package Triangle.AbstractSyntaxTrees;
import Triangle.SyntacticAnalyzer.
    SourcePosition;

public class RecursiveDeclaration extends
    Declaration {
    public RecursiveDeclaration
        (Declaration procFuncAST,
                     SourcePosition
                         thePosition) {
        super (thePosition);
        this.procFuncAST = procFuncAST;
    }

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

    public Declaration procFuncAST;
}
```

As for the LocalDeclaration class, the following was implemented, under the same package:

```
//Class added on 10/14/19 by
    andres.mirandaarias@gmail.com
package Triangle.AbstractSyntaxTrees;
import Triangle.SyntacticAnalyzer.
    SourcePosition;

public class LocalDeclaration extends
    Declaration {
    public LocalDeclaration (Declaration
        dcl1, Declaration dcl2,
                     SourcePosition
                         thePosition) {
        super (thePosition);
        this.dcl1 = dcl1;
        this.dcl2 = dcl2;
    }

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

    public Declaration dcl1, dcl2;
}
```

Given that *Declaration* was already implemented, for the *LocalDeclaration* alternative, but *Proc-Funcs* was not, for *RecursiveDeclaration*, this new alternative had to be added to be *Declaration* parser.

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

In order to implement the *Proc-Funcs* parsing into the Parser, the following code was added:

```
// parseProcFuncs added on 10/14/19 by
    andres.mirandaarias@gmail.com
SequentialDeclaration parseProcFuncs()
    throws SyntaxError{
    SequentialDeclaration declarationAST =
        null;
    SourcePosition declarationPos = new
        SourcePosition();
    start(declarationPos);
    Declaration dAST1 = null;
    if(currentToken.kind == Token.PROC ||
        currentToken.kind == Token.FUNC){
        dAST1 = parseProcFunc();
        finish(declarationPos);
```

```
   } else{
      syntacticError("\"%\" error parsing
         proc-funcs",
         currentToken.spelling);
   }

   if(currentToken.kind == Token.AND){
      acceptIt();
      start(declarationPos);
      Declaration dAST2 = parseProcFunc();
      finish(declarationPos);
      declarationAST = new
         SequentialDeclaration
         (dAST1, dAST2, declarationPos);
      while(currentToken.kind ==
         Token.AND){
         acceptIt();
         start(declarationPos);
         dAST1 = parseProcFunc();
         finish(declarationPos);
         declarationAST = new
            SequentialDeclaration
               (declarationAST, dAST1,
                  declarationPos);
      }
   } else{
      syntacticError("\"%\" error parsing
         proc-funcs, expected AND",
         currentToken.spelling);
   }
   return declarationAST;
}
```

Since, once again, *Proc-Func* is not implemented, it had to be added to the parser as follows:

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

To implement this alternative in the *Parser* class, the following code was added:

```
// parseProcFunc added on 10/14/19 by
    andres.mirandaarias@gmail.com
Declaration parseProcFunc() throws
    SyntaxError{
   Declaration declarationAST = null;
   SourcePosition declarationPos = new
       SourcePosition();
   start(declarationPos);
   switch(currentToken.kind){
      case Token.PROC:
      {
         acceptIt();
         Identifier iAST =
            parseIdentifier();
         accept(Token.LPAREN);
         FormalParameterSequence fAST =
            parseFormalParameterSequence();
         accept(Token.RPAREN);
         accept(Token.IS);
         Command cAST = parseCommand();
         accept(Token.END);
         finish(declarationPos);
         declarationAST = new
            ProcDeclaration(iAST, fAST,
            cAST, declarationPos);
      }
      break;
      case Token.FUNC:
      {
         acceptIt();
         Identifier iAST =
            parseIdentifier();
         accept(Token.LPAREN);
         FormalParameterSequence fAST =
           parseFormalParameterSequence();
         accept(Token.RPAREN);
         accept(Token.COLON);
         TypeDenoter tAST =
            parseTypeDenoter();
         accept(Token.IS);
         Expression eAST =
            parseExpression();
         finish(declarationPos);
         declarationAST = new
            FuncDeclaration(iAST, fAST,
            tAST, eAST, declarationPos);
      }
      break;
      default:
         syntacticError("\"%\" error
            parsing proc-func",
           currentToken.spelling);
      break;
   }
   return declarationAST;
}
```

The final overall change in the syntactic rules of the compiler that were made was in *single-Declaration*. Initially, this rule was as follows:

```
single-Declaration
   ::= const Identifier ~ Expression
   |  var Identifier : Type-denoter
   |  proc Identifier (
       Formal-Parameter-Sequence ) ~
          single-Command
   |  func Identifier (
       Formal-Parameter-Sequence )
          : Type-denoter ~ Expression
   |  type Identifier ~ Type-denoter
```

For this particular project, there were two updates made to this *single-Declaration* rule:

```
...
|  "proc" Identifier "("
   Formal-Parameter-Sequence ")"
      "~" Command "end"
...
|  "var" Identifier "init" Expression
```

Specifically for the ”proc” update, the following code was modified:

```
//command and end was added, according to
   proyect specification
case Token.PROC:
{
   acceptIt();
   Identifier iAST = parseIdentifier();
   accept(Token.LPAREN);
   FormalParameterSequence fpsAST =
      parseFormalParameterSequence();
   accept(Token.RPAREN);
   accept(Token.IS);
   Command cAST = parseCommand();
   accept(Token.END);
   finish(declarationPos);
   declarationAST = new
      ProcDeclaration(iAST, fpsAST,
      cAST, declarationPos);
}
break;
```

On the other hand, since the new alternative is for initialized variable declarations, the *VarDeclarationInit* class had to be added after modifying that specific section in the parser as follows:

```
//case Var.Init was added
case Token.VAR:
{
   acceptIt();
   Identifier iAST = parseIdentifier();
   switch(currentToken.kind){
      case Token.COLON:
      // implementation for this case
      break;
      case Token.INIT:
      {
         acceptIt();
         Expression eAST =
            parseExpression();
         finish(declarationPos);
         declarationAST = new
            VarDeclarationInit
            (iAST,eAST,declarationPos);
      }
      break;
   }
}
break;
```

After modifying the parser, the class *VarDeclarationInit* was added in order to create this particular abstract syntax tree, as well as implementing the respective visitor methods in the respective classes.

```
//Class added on 10/14/19 by
   andres.mirandaarias@gmail.com
package Triangle.AbstractSyntaxTrees;
import Triangle.SyntacticAnalyzer.
   SourcePosition;
public class VarDeclarationInit extends
    Declaration {
   public VarDeclarationInit (Identifier
       iAST, Expression eAST,
                   SourcePosition
                      thePosition) {
      super (thePosition);
      I = iAST;
      E = eAST;
   }

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

   public Identifier I;
   public Expression E;
}
```

B. HTML representation of the source code

This feature takes the source code and outputs a file with the code but formatted, to assign a specific font and size to the text, reserved words bold, comments color green, and literals color blue.

To achieve this the Scanner class was modified, the reason why this class was chosen is because it goes over the source code and gets spaces, tabs, new lines and comments, those are required to be in the HTML file to get the code to look alike the source code entered on the IDE.

In the Scanner class a variable to store the HTML content was added, then this variable makes its way on the output file, also methods were modified those methods are *scanSeparator* and *scan*, the first one saves the comments in a temporal variable and then when the comment is completely scanned appends it to the HTML variable, on the other hand scan method just sends the information of the scanned tokens all of this strings and variables are handled by a new method added call *insertHTML(type, token or comment)* this method is the responsible for appending the text of the source code into the HTML variable with a classification of type reserved word, comment or literal to format the way it was described before.

C. Changes made to the syntactic rules of Triangle to achieve the language to be LL(1)

This compiler uses a Recursive-descent parser which requires the grammar to be LL(1), these means the grammar for the language has to satisfy the following two conditions:

- If the grammar contains X|Y, starters[X] and starter[Y] must be disjoint.
- If the grammar contains X*, starters[X] must be disjoint from the set of tokens that follow X* in this particular order.

To make Triangle$^+$ suitable for this compiler simple transformations should be done, those transformations are left factorization and substitution that will be shown below.

The first and most obvious changes are located on *Command* and *single-Command* those does not satisfy the conditions above.

```
Command        ::= single-Command
                 | Command; single-Command

single-Command ::= skip
               | V-name := Expression
               | Identifier
                 (Actual-Parameter-Sequence)
               ...
```

Firstly *Command* has left recursion so it has to change, then *V-name Identifier* produce the same tokens which gives the problem of its starters been not disjoint. After the left recursion was eliminated and left factorization was made the syntactic rules become the following.

```
Command        ::= single-Command(;
    single-Command)*

single-Command ::= ...
               | Identifier(:= Expression|
                 (Actual-Parameter-Sequence))
               ...
```

Another syntactic change was made in *Declaration* it has left recursion and it should be eliminated. The following is the original syntax:

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

The syntax after changes is:

```
Declaration ::= compound-Declaration(;
    compound-Declaration)*
```

Lastly because of the integration of a new rule on single-Declaration, it has to be modified with left factorization before it can be introduced. The new rule gives problems because it has the same starters as var so the grammar should be modified as follows:

```
single-Declaration ::= ...
                 | "var" Identifier (":"
                   Type-denoter | "init"
                   Expression)
```

D. Abstract Syntax Trees Modeling

The Abstract Syntax Tree modeling in the compiler was left practically the same. Nonetheless, three different trees were added and had to be modeled. This was made modifying the *Visitor* class as follows:

```
package Triangle.AbstractSyntaxTrees;
public interface Visitor {
   /*
   Other methods
   */
   public abstract Object
       visitRecursiveDeclaration(
      RecursiveDeclaration ast, Object o);
      //RECURSIVE DECL. VISITOR ADDED.
   public abstract Object
       visitVarDeclarationInit(
      VarDeclarationInit ast, Object o);
      //VAR DECL. INIT VISITOR ADDED.
   public abstract Object
       visitLocalDeclaration(
      LocalDeclaration ast, Object o);
      //LOCAL DECLARATION VISITOR ADDED.
   /*
   Other methods
   */
}
```

Since *Visitor* is an interface, different classes implement this interface throughout the compiler. Specifically, the *TableVisitor, TreeVisitor, LayoutVisitor, Cheker* and *Encoder* implement the *Visitor* interface. Given that the scope of this project is limited to the syntactical and lexical analyzer, meaning the contextual analyzer and code generation is not modified yet, the classes *Encoder* and Checker were modified, but the visit methods were not implemented yet.

Specifically for the TableVisitor class, the visit methods for all three new ASTs were implemented as follows:

```
public class TableVisitor implements
    Visitor {
   /*
   Other methods
   */
   //RECURSIVE DEC. TABLE VISITOR ADDED.
   public Object
       visitRecursiveDeclaration(
          RecursiveDeclaration ast, Object
              o){
      ast.procFuncAST.visit(this,null);
      return(null);
   }

   //LOCAL DEC. TABLE VISITOR ADDED.
   public Object visitLocalDeclaration(
          LocalDeclaration ast, Object o){
      ast.dcl1.visit(this,null);
      ast.dcl2.visit(this,null);
      return(null);
   }

   //VARDECLARATIONINIT ADDED, on
      10/14/19 by
      andres.mirandaarias@gmail.com
   public Object visitVarDeclarationInit(
          VarDeclarationInit ast, Object
              o){
      try{
         String type = "";
         int value = -1, displacement =
             -1, level = -1;
         if(ast.entity instanceof
             KnownValue){
            type = "KnownAddress";
            value =
                ((KnownValue)ast.entity)
               .value;
         }
         else if(ast.entity instanceof
             UnknownValue){
            type = "UnknownAddress";
            level =
                ((UnknownValue)ast.entity)
               .address.level;
            displacement =
                ((UnknownValue)ast.entity)
               .address.displacement;
         }

         addIdentifier(ast.I.spelling,
                   type,
                   (ast.entity!=null?
                      ast.entity.size:0),
                   level,
                   displacement,
                   value);
      } catch(NullPointerException e){}
      ast.E.visit(this,null);
      ast.I.visit(this,null);
      return(null);
   }
   /*
   Other methods
   */
}
```

After implementing the *TableVisitor* class, the next modification made was in the *TreeDrawer* class, which, as the name suggests, is the one in charge of ”drawing” the ASTs. These changes were made as follows:

```
public class TreeVisitor implements
    Visitor {
   /*
   Other methods
   */
   //RECURSIVE DECLARATION TREE VISITOR
      ADDED.
   public Object
      visitRecursiveDeclaration(
         RecursiveDeclaration ast, Object
            obj){
      return(createUnary("Recursive
         Declaration", ast.procFuncAST));
   }

   //LOCAL DECL. TREE VISITOR ADDED
   public Object visitLocalDeclaration(
         LocalDeclaration ast, Object
            obj){
      return(createBinary("Local
         Declaration", ast.dcl1,
         ast.dcl2));
   }

   //VAR DECL. INIT TREE VISITOR ADDED
   public Object visitVarDeclarationInit(
         VarDeclarationInit ast, Object
            obj){
      return(createBinary("Variable
         Declaration Init", ast.I,
         ast.E));
   }
   /*
   Other methods
   */
}
```

Finally, the last class updated to add the visitor methods defined on the Visitor interface, was the *LayoutVisitor* class. The modifications were made as follows, adding the visitor methods for *VarDeclarationInit*, *RecursiveDeclaration* and *LocalDeclaration*:

```
public class LayoutVisitor implements
    Visitor {
   /*
   Other methods
   */
   //LOCAL DECL. LAYOUT ADDED
   public Object visitLocalDeclaration(
         LocalDeclaration ast, Object
            obj){
      return layoutBinary("Local Decl.",
         ast.dcl1, ast.dcl2);
   }

   //RECURSIVE DECL. LAYOYT VISITOR.
   public Object
      visitRecursiveDeclaration(
         RecursiveDeclaration ast, Object
            obj){
      return layoutUnary("Rec. Decl.",
         ast.procFuncAST);
   }

   //VAR DECL INIT LAYOUT ADDED, on
      10/14/19 by
      andres.mirandaarias@gmail.com
    public Object
      visitVarDeclarationInit(
         VarDeclarationInit ast, Object
            obj){
      return layoutBinary("VarDeclInit.",
         ast.I, ast.E);
    }
}
```

As previously mentioned, both classes *Checker* and Encoder had to implement these different visitor methods, since they implement the *Visitor* interface. However, the methods were added but not yet implemented, and example can be seen bellow, and all three methods in both classes followed the same format, varying only in their respective names and parameters:

```
//LOCAL DECL. CHECKER ADDED
public Object visitLocalDeclaration(
      LocalDeclaration ast,Object o){
   return null;
}
```

E. Extension made to the methods that allow abstract syntax trees to be represented as text in XML

The methods that came in the XML generator were not modified, only the methods needed to write the new rules created that are detailed below:

Two classes were created whose names are: WriterVisitor.java and Writer.java

```
public interface Visitor {

  .
  . //Other methods
  .
```

```
 public abstract Object
     visitDoUntilCommand(DoUntilCommand
     ast, Object o);
 public abstract Object
     visitDoWhileCommand(DoWhileCommand
     ast, Object o); //Visit for DoWhile
     command added

 .
 . //Other methods
 .
 }
```

You can find these classes from the TreeWriterXML package. On the other hand there is an interface within the AST package called visitor, so the WriterVisitor.java class is responsible for implementing the methods of this Visitor.java class.

```
private void writeLineXML(String line) {
      try {
         fileWriter.write(line);
         fileWriter.write('\n');
      } catch (IOException e) {
         System.err.println("Error while
             writing file for print the
             AST");
         e.printStackTrace();
      }
   }
end
```

Within the WriterVisitor.java class there is a method called "writeLineXML" which is responsible for writing without problems in the xml file line by line.

```
private String transformOperator(String
    operator) {
      if (operator.compareTo("<") == 0)
         return "&lt;";
      else if (operator.compareTo("<=")
          == 0)
         return "&lt;=";
      else
         return operator;
   }
end
```

Another method called "transformOperator" was created which converts the characters "¡" "¡=" to their equivalents in HTML

F. Extension made to the methods that allow to visualize AST's on an IDE

In this section is explained what changes were made to achieve the extension of the AST's. The class extended is the *LayoutVisitor* class which is part of the *TreeDrawer Package* and implements the "visitor" Class. The changes made are in the following section.

Commands modified or extended in the LayoutVisitor class:

```
//DO UNTIL LAYOUT ADDED
 public Object
     visitDoUntilCommand(DoUntilCommand
     ast, Object obj){
    return layoutBinary("DoUntilCom.",
       ast.cAST, ast.eAST);
 }

 //DO WHILE LAYOUT ADDED
 public Object
     visitDoWhileCommand(DoWhileCommand
     ast, Object obj){
    return layoutBinary("DoWhileCom.",
       ast.cAST, ast.eAST);
 }

 //FOR CMD LAYOUT ADDED
 public Object
     visitForDoCommand(ForDoCommand ast,
     Object o) {
   return layoutQuaternary("ForDoCom.",
     ast.I, ast.E1, ast.E2, ast.C);
 }

 //UNTIL CMD LAYOUT ADDED
 public Object
     visitUntilCommand(UntilCommand ast,
     Object obj){
   return layoutBinary("UntilCom.",
     ast.E,ast.C);
 }
```

The other part of the code modified is in the section of Declarations:

```
//LOCAL DECL. LAYOUT ADDED
 public Object
     visitLocalDeclaration(LocalDeclaration
     ast, Object obj){
   return layoutBinary("Local Decl.",
     ast.dcl1, ast.dcl2);
```

```
  }

  //RECURSIVE DECL. LAYOYT VISITOR.
  public Object visitRecursiveDeclaration
    (RecursiveDeclaration ast, Object obj){
    return layoutUnary("Rec. Decl.",
        ast.procFuncAST);
  }

  //VAR DECL INIT LAYOUT ADDED.
  public Object visitVarDeclarationInit
    (VarDeclarationInit ast, Object obj){
    return layoutBinary("VarDeclInit.",
        ast.I, ast.E);
  }
```

Expressions, Aggregates, Parameters, Type denoter, literals, variables and program were not modified or extended.

G. Test plan to validate the compiler

Case Objective: ”begin” Command ”end”
Case Design

```
begin puteol() end
```

Expected Results: It is expected to fail compilation since begin is not a keyword anymore
Observed Results: Syntactic Analysis ... ERROR: ”:=” expected here 2..2
Compilation was unsuccessful.

Case Objective: ”begin” Command ”end”
Case Design:

```
let
  var begin: Integer
in
  putint (begin);
  puteol ()
end
```

Expected Results: It is expected to compile successfully since begin is used as a variable name, not as a keyword.
Observed Results: Syntactic Analysis ...
Compilation was successful.

Case Objective: ”let” Declaration ”in” Command ”end”
Case Design:

```
let var i : Char;
   var j : Integer;
   const k ::= 8;
   type dia ˜ Integer

   put(i);
   put(i)
end
```

Expected Results: It is expected to fail compilation since the ”in” keyword is missing
Observed Results: Syntactic Analysis ... ERROR: ” ” expected here 3..3
Compilation was unsuccessful.

Case Objective: ”let” Declaration ”in” Command ”end”
Case Design:

```
let in
   put(i);
   put(i)
end
```

Expected Results: It is expected to fail compilation since it has an empty Declaration
Observed Results:Syntactic Analysis ... ERROR: ”in” error parsing proc-func 1..1
Compilation was unsuccessful.

Case Objective: ”let” Declaration ”in” Command ”end”
Case Design:

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

Expected Results: It is expected to fail compilation since it is missing single-Command
Observed Results: Syntactic Analysis ... ERROR: ”end” cannot start a command 7..7
Compilation was unsuccessful.

Case Objective: ”let” Declaration ”in” Command ”end”

Case Design:

```
let var i : Char;
    var j : Integer;
    const k ::= 8;
    type dia ~ Integer

in
    put(i);
    put(i)
```

Expected Results: It is expected to fail compilation since the keyword ”end” is missing

Observed Results: Syntactic Analysis ... ERROR: ” ” expected here 3..3

Compilation was unsuccessful.

Case Objective: ”let” Declaration ”in” Command ”end”

Case Design:

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

Expected Results: It is expected to compile successfully since the correct syntax is used

Observed Results: Syntactic Analysis ...

Compilation was successful.

Case Objective: ”if” Expression ”then” Command ”else” Command ”end”

Case Design:

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

Expected Results: It is expected to fail compilation since the keyword ”then” is missing

Observed Results: Syntactic Analysis ... ERROR: ”then” expected here 1..1

Compilation was unsuccessful.

Case Objective: ”if” Expression ”then” Command ”else” Command ”end”

Case Design:

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

Expected Results: It is expected to fail compilation since the keyword ”else” is missing

Observed Results: Syntactic Analysis ... ERROR: ”else” expected here 1..1

Compilation was unsuccessful.

Case Objective: ”if” Expression ”then” Command ”else” Command ”end”

Case Design:

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

Expected Results: It is expected to fail compilation since the keyword ”do” is used rather than ”then”

Observed Results: Syntactic Analysis ... ERROR: ”then” expected here 1..1

Compilation was unsuccessful.

Case Objective: ”if” Expression ”then” Command ”else” Command ”end”

Case Design:

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

Expected Results: It is expected to fail compilation since the keyword ”end” is missing

Observed Results: Syntactic Analysis ... ERROR: ”end” expected here 1..1

Compilation was unsuccessful.

Case Objective: ”if” Expression ”then” Command ”else” Command ”end”

Case Design:

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

Expected Results: It is expected to compile successfully since correct syntax was used

Observed Results: Syntactic Analysis ...

Compilation was successful.

Case Objective: emptyCommand $\rightarrow$ ”skip”

Case Design:

```
! no instructions
```

Expected Results: It is expected to fail compilation since empty commands are not supported anymore

Observed Results: Syntactic Analysis ... ERROR: ”” cannot start a command 1..1

Compilation was unsuccessful.

Case Objective: emptyCommand → ”skip”

Case Design:

```
puteol();
```

Expected Results: It is expected to fail compilation since empty commands are not supported anymore

Observed Results: Syntactic Analysis ... ERROR: ”” cannot start a command 1..1

Compilation was unsuccessful.

Case Objective: emptyCommand → ”skip”

Case Design:

```
skip
```

Expected Results: It is expected to compile succesfully since correct syntax is implemented

Observed Results: Syntactic Analysis ...

Compilation was successful.

Case Objective: ”loop” ”while” Expression ”do” Command ”repeat”

Case Design:

```
loop while 1=1 skip repeat
```

Expected Results: It is expected to fail compilation since keyword ”do” is missing

Observed Results: Syntactic Analysis ... ERROR: ”do” expected here 1..1

Compilation was unsuccessful.

Case Objective: ”loop” ”while” Expression ”do” Command ”repeat”

Case Design:

```
loop while 1=1 do skip
```

Expected Results: It is expected to fail compilation since keyword ”repeat” is missing

Observed Results: Syntactic Analysis ... ERROR: ”repeat” expected here 1..1

Compilation was unsuccessful.

Case Objective: ”loop” ”while” Expression ”do” Command ”repeat”

Case Design:

```
loop while 1=1 do puteol(); skip repeat
```

Expected Results: It is expected to compile successfully since correct syntax was implemented

Observed Results: Syntactic Analysis ...

Compilation was successful.

Case Objective: ”while” Expression ”do” singleCommand

Case Design:

```
while 1=1 do skip
```

Expected Results: It is expected to fail compilation since ”while” is not a keyword anymore

Observed Results: Syntactic Analysis ... ERROR: ”while” cannot start a command 1..1

Compilation was unsuccessful.

Case Objective: ”loop” ”until” Expression ”do” Command ”repeat”

Case Design:

```
until 2=2 do skip repeat
```

Expected Results: It is expected to fail compilation since the keyword ”loop” is missing

Observed Results: Syntactic Analysis ... ERROR: ”until” cannot start a command 1..1

Compilation was unsuccessful.

Case Objective: ”loop” ”until” Expression ”do” Command ”repeat”

Case Design:

```
loop until 2=2 do skip
```

Expected Results: It is expected to fail compilation since the keyword ”repeat” is missing

Observed Results: Syntactic Analysis ... ERROR: ”repeat” expected here 1..1

Compilation was unsuccessful.

Case Objective: ”loop” ”until” Expression ”do” Command ”repeat”

Case Design:

```
loop until 2=2 skip repeat
```

Expected Results: It is expected to fail compilation since the keyword ”do” is missing

Observed Results: Syntactic Analysis ... ERROR: ”do” expected here 1..1

Compilation was unsuccessful.

Case Objective: ”loop” ”until” Expression ”do” Command ”repeat”

Case Design:

```
loop until 2=2 do skip repeat
```

Expected Results: It is expected to compile successfully since correct syntax was implemented

Observed Results: Syntactic Analysis ...

Compilation was successful.

Case Objective: ”loop” ”do” Command ”until” Expression ”end”

Case Design:

```
loop
do
a := a + 2
until
a < 2
repeat
```

Expected Results: It is expected to compile because it complies with the established syntax

Observed Results: Syntactic Analysis ...

Compilation was successful.

Case Objective: ”loop” ”do” Command ”until” Expression ”end”

Case Design:

```
loop
do
a < 2
until
a < 2
repeat
```

Expected Results: It is expected not to compile because “a ¡ 2” is not a command

Observed Results: Syntactic Analysis ... ERROR: ”:=” expected here 3..3

Compilation was unsuccessful.

Case Objective: ”loop” ”do” Command ”until” Expression ”end”

Case Design:

```
loop
do
skip
until
skip
repeat
```

Expected Results: It is expected not to compile because ”skip” is a command not an expression

Observed Results: Syntactic Analysis ... ERROR: ”skip” cannot start an expression 5..5

Compilation was unsuccessful.

Case Objective: loop” ”for” Identifier ”∼” Expression ”to” Expression ”do” Command ”repeat”

Case Design:

```
loop
for
a
~
1
to
10
do
skip
repeat
```

Expected Results: The compilation is expected to be successful as it complies with the established syntax

Observed Results: Syntactic Analysis ...

Compilation was successful.

Case Objective: loop" "for" Identifier "~" Expression "to" Expression "do" Command "repeat"

Case Design:

```
loop
for
3
~
1
to
10
do
skip
repeat
```

Expected Results: The compilation is expected to fail because an identifier is expected not an expression

Observed Results: Syntactic Analysis ... ERROR: identifier expected here 3..3 Compilation was unsuccessful.

Case Objective: loop" "for" Identifier "~" Expression "to" Expression "do" Command "repeat"

Case Design:

```
loop
for
a
~
skip
to
10
do
skip
repeat
```

Expected Results: The compilation is expected to fail because an expression is expected not a command

Observed Results: Syntactic Analysis ... ERROR: "pass" cannot start an expression 5..5

Compilation was unsuccessful.

Case Objective: loop" "for" Identifier "~" Expression "to" Expression "do" Command "repeat"

Case Design:

```
loop
for
a
frm
1
to
10
do
10
repeat
```

Expected Results: The compilation is expected to fail because it is expected ~

Observed Results: Syntactic Analysis ... ERROR: "~" expected here 4..4

Compilation was unsuccessful.

Case Objective: "if" Expression "then" Command "else" Command "end"

Case Design:

```
let
  var a : Integer
in
  a:=3;
  if a > 0 then
    put ('M')
  else
    put ('m')
  end
```

Expected Results: You are expected to compile successfully as it has no errors

Observed Results: Syntactic Analysis ...

Compilation was successful.

Case Objective: "if" Expression "then" Command "else" Command "end"

Case Design:

```
let
  var a : Integer
in
  a:=3;
  if a > 0 then
    put ('M')
  els
    put ('m')
  end
end
```

Expected Results: It is expected that it does not compile since it has errors is else and not els

Observed Results: Syntactic Analysis ... ERROR: "else" expected here 7..8

Compilation was unsuccessful.

Case Objective: ”if” Expression ”then” Command ”else” Command ”end”
Case Design:

```
let
  var a : Integer
in
  a:=3;
  if a > 0 then
    put ('M')
  else
    put ('m')
  edn
end
```

Expected Results: It is expected that it does not compile since it has errors is end and not edn
Observed Results: Syntactic Analysis ... ERROR: ”end” expected here 9..10
Compilation was unsuccessful.

Case Objective: ”let” Declaration ”in” Command ”end”
Case Design:

```
let
  var a : Integer
ni
  a:=3;
  if a > 0 then
    put ('M')
  else
    put ('m')
  end
end
```

Expected Results: It is expected that it does not compile since it has errors is in and not ni
Observed Results: Syntactic Analysis ... ERROR: ”in” expected here 3..3
Compilation was unsuccessful.

Case Objective: Declaration ”;” compound-Declaration
Case Design:

```
let
  var a : Integer
in
  a:=3.
  if a > 0 then
    put ('M')
  else
    put ('m')
  end
end
```

Expected Results: It is expected to fail because it is missing ”;” in the statement
Observed Results: Syntactic Analysis ... ERROR: ”end” expected here 4..5
Compilation was unsuccessful.

Case Objective: compund-Declaration ::= single-Declaration
Case Design:

```
let
    func f (a : Integer): Integer ~ if a >
       0 then g(a - 1) else 0
    in
      putint(f(10))
    end
end
```

Expected Results: The compilation is expected to be successful.
Observed Results: Syntactic Analysis ...
Compilation was successful.

Case Objective: compound-Declaration ::= recursive end
Case Design:

```
let
  recursive

  end
in
  skip
end
```

Expected Results: compilation error because recursive needs at least one pro-func Observed Results: ERROR: ”end” error parsing proc-funcs 4..5

Case Objective: Proc-Func ::= proc Identifier(Formal-Parameter-Sequence) Command
Case Design:

```
let
  proc dos(var i : Integer) ~
    i := i + uno(i)

in
  skip
end
```

Expected Results: compilation error because proc needs reserved word end at the end Observed Results: ERROR: ”end” expected here 5..6

Case Objective: compound-Declaration ::= recursive Proc-Funcs end
Case Design:

```
let
  recursive
    func uno(var i : Integer) : Integer ~
      dos(i)/3*2
      and
    proc dos(var i : Integer) ~
      i := i + uno(i)
    end
  end
in
  skip
end
```

Expected Results: The compilation is expected to be successful. Observed Results: Compilation was successful.

Case Objective: compound-Declaration ::= local Declaration in Declaration end
Case Design:

```
let
  local
    var num: Integer
  in
    var othernum: Integer
  end
in
  skip
end
```

Expected Results: The compilation is expected to be successful. Observed Results: Compilation was successful.

Case Objective: single-Declaration ::= var Identifier init Expression
Case Design:

```
let
  var doble init num*2
in
  skip
end
```

Expected Results: The compilation is expected to be successful. Observed Results: Compilation was successful.

For tokens test cases, consist to prove that the token cannot be a variable name, declaration or identifier. Also, this test case is applied to the ten new reserved words, to prove that they are now tokens.

Case Objective: Token Error
Case Design:

```
let
  var skip: Integer
in
  putint (skip);
  puteol ()
end
```

Expected Result: Compilation is expected to fail because skip is a token word and cannot be an identifier
Observed Result: Syntactic Analysis ... ERROR: identifier expected here 2..2 Compilation was unsuccessful.

Case Objective: Token Skip Success
Case Design:

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

Expected Result: Compilation successful, two examples of the use of command skip
Observed Result: Syntactic Analysis ... Compilation was successful.

III. Results and Discussion

A. *Discussion and analysis of the results obtained*

In general the results obtained were satisfactory, the project was completely full filled in every request made in its specification. The following are the main achievements reached:

1) Syntactic changes
2) Lexical changes
3) Generate XML and HTML
4) Modified the lexical analyzer to work with extended language full
5) Modified the parser so that it can recognize the language extended complete and build abstract syntax trees

B. A reflection on the experience of modifying fragments of a compiler / environment written by third parties

We face a challenging task due to the difficulty of understanding code developed by a person outside the work team, especially if it is a new topic in which one is acquiring knowledge is more complicated. Understanding the work of someone else is a process were, getting the idea behind of the methods or variables is just there in text, even if there are comments the idea can be misunderstood and lead to errors on the program.

Nonetheless, thanks to the fact that code development followed a good line of programming practices and its comments are descriptive enough, the understanding has made easier. Besides that, the project helped consolidate the knowledge transmitted in class by the professor since in this context the practice helps to make the differences clearer between the parts of the compiler and the work that each one does.

C. Summary description of the tasks performed by each member of the group of job

1) Ricardo Molina-Brenes: Generation of all classes related to the creation of the XML file, implementation of methods of the Visitor interface and creation of packages. Collaborative contribution in teamwork helping all members in present difficulties and work in the documentation.

2) Andres Miranda-Arias: Made modifications to the lexical and syntactical analyzer, specifically all the changes made to parsing *Declaration*; adding the new *compound-Declaration, Proc-Func, Proc-Funcs* and modifying *single-Declaration*, as well as implementing all the required classes and visitor methods were needed. Helped team members were required, as well as extensive work in the final project documentation and test cases.

3) Israel Padilla-Jenkins: Added, modified and delete cases to achieve the implementations of the new tokens words: *and, for, init, local, loop, recursive, repeat, skip, to, until*. Worked on major lexical and syntactical changes in parser class, in Single-Command and Command cases: *skip, loop cases, let-in case, if case*. Added required class and methods of specific commands: *Dowhile, DoUntil, ForCommand, Until*. Communication with teamwork when needed and help on creating testing cases and working in project's documentation.

4) Jose Campos-Espinoza: Responsible for the HTML output file generated out of the source code. Helped team members when required and worked testing the compiler and in the documentation for the project.

D. Program compilation

First of all, the project .zip file must be unzipped. The main project should already be compiled and ready to execute.

Nonetheless, in order to actually compile the program, the main project should be opened. The recommendation previously made by Diego Ramirez in his user manual[3], and the one made by this team, is to open the project using NetBeans 8.2 IDE.

Once the project is opened, depending on the user's NetBeans' configuration, the project must be compiled by clicking on the hammer button, or if in Windows, press F11 button. If no changes were made, the project should and **must** compile successfully. On the other hand, as mentioned, depending on the user's NetBeans configuration, if green right arrow button is pressed, the project could be compiled and executed at once.

E. Program execution

Once compiled, the execution of the program can take place two different ways. First way is to open the following path: *"ide-triangle-v1.1.scr\dist"*. In this path, the user can find the .jar of the project, named *IDE-Triangle.jar*.

If double clicked, the main IDE will appear in the user's main screen. Now, in order to test the compiler, the user must select the new file option (blank page icon) and type in the desired Triangle code. Next, to compile this written code and test the compiler, the user must select the double green arrow icon.

The compilation process will begin, and its outputs can be seen in the "Console" tab. Here, the user can see if compilation was successful or not, and in the later case, the user can see the errors and why did compilation fail, as previously seen in the test cases.

IV. Conclusion

For a project this complex it is important to study and know the most relevant aspects of the compiler and how it works, before even start to see the source code. There are things that have to be done in order to make the compiler succeed, for example knowing the strategy to parse the source codes, what it needs like the language to be LL(1) and if it does not satisfy that to make the appropriate changes to ensure that the compiler will not fail after all the work is done. Also the advantage of knowing how the compiler works is that working on it is much more easier, to find what it needs to be modified and fix bugs became simpler just because you know were everything is located.

References

[1] D. Watt and D. Brown, Programming Language Processors in Java: Compilers and Interpreters, 1st ed. Essex: Pearson Education Limited, 2000.

[2] I. Trejos, Proyecto #1, las fases de análisis léxico y sintáctico de un compilador, 2nd ed. Cartago, 2019.

[3] D. Ramirez, Manual para integración del IDE y el compilador de Δ, na, Cartago, 2018.

V. Annex

In order to locate all of the following files, the user must first unzip the project .zip file.

A. Compiler Source Code

The user will locate the compiler source code in the following path: *"ide-triangle-v1.1.scr\src"*. In this particular folder, the user will see four different folders, *Core, GUI, TAM* and *Triangle*. Within this last folder, the user will see every file of the compiler's source code, organized in different folders.

B. Compiler Object Code

The user will locate the compiler object code executable file for Windows OS in the following path: *"ide-triangle-v1.1.scr\dist"*. Here, the user will find the *IDE-Triangle.jar* file to execute the program without directly opening the project.

C. Test Case .tri Files

The user will locate the all test cases in the following path: *"Test Cases"*. Given that the test case generation was divided amongst all four team members, the user will find the test cases divided in four different folders, each with its respective author.

D. GitHub Project Link

If required, necessary or important, the user can locate the project's GitHub link at https://github.com/andresm07/Triangle_Lexical_Syntactical_Analyzer