Triangle$^+$ Compiler Contextual 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 of the language was performed, which resulted in Δ^+. Different changes were made to the contextual analyzer of the Watt and Brown compiler. The different test cases and results are shown as implemented, as well as the different modifications to the compiler in Java. Mainly the changes were made in the Checker class. All changes done were made respecting the existing coding format and style, originally implemented by Watt and Brown, as well as the changes made by Dr. Luis Pérez. In general, all requested changes were made successfully; the compilation process will fail when contextual or syntactical rules are missed, though it will compile when correct syntax and context is used, thus completing the scope of the project correctly.**

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

CONTENTS

I. Introduction

The objective of this project is to understand the details of contextual 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. Type Checking for *loop ... repeat* Variants

To ensure that ***loop ... repeat*** command satisfies the contextual restrictions, the Contextual Analyzer was modified to identify contextual errors, more specifically the Checker class got some new features to make the compiler able to detect those errors, the features or additions are going to be described below.

There are two main restrictions to take into consideration the first one is, the expression that ***loop ... repeat*** command receives has to be Boolean type, and the other one is that the command itself has to comply with the contextual restrictions. That said is time to apply the restrictions in the Checker class for all the variants:

- **loop while** *Exp* **do** *Com* **repeat**
- **loop until** *Exp* **do** *Com* **repeat**
- **loop do** *Com* **while** *Exp* **repeat**
- **loop do** *Com* **until** *Exp* **repeat**

As mentioned above the expression has to be Boolean type and command has to comply with contextual restrictions and this restrictions are verified by the following code:

```
TypeDenoter eType = (TypeDenoter)
    ast.eAST.visit(this,null);
if(!eType
    .equals(StdEnvironment.booleanType)){
  reporter.reportError("Boolean
      expression expected", "",
      ast.eAST.position);
}
idTable.openScope();
ast.cAST.visit(this,null);
idTable.closeScope();
return null;
```

Where the first step is to verify if *Exp* is boolean type, if not it generates the report error, then it opens a new scope in the table because *Com* is like a different block on the code, and its contextual restrictions have to be checked apart.

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

*B. Solution to the Scope, Type and Protection of the Control Variable for the Command **loop for ... repeat***

First of all to explain how the solution was implemented, this command got a change in its AST structure, at first glance it would look like its AST is in quaternary form but with some tweaks it can be converted to ternary form, to gain a little advantage when Generating code because the AST will be more simple and easy to read. The structure of this command is the following:

loop for $Id \sim Exp_1$ **to** Exp_2 **do** Com **repeat**

The obvious way to insert this into the AST representation is to take the Id, Exp_1, Exp_2 and Com and put it on the AST. Given that $Id \sim Exp_1$ is a declaration itself and it has its contextual restrictions like apart from the **for** command, this is the key to convert this command AST into ternary form, the steps required to make this possible are the following:

- Create a new AST **for** declaration this should receive the Id and $Exp1$ a make it into one single declaration.
- Modify the Parser class to parse the **for** declaration and make a ternary AST based on the **for** command, this new AST has as parameters $Decl$, Exp_2 and Com which are declaration, the ending expression and the command respectively.
- Add a new method in Checker class to inspect the command for contextual errors.
- Adapt the Visitor Classes in charge of representing the information of the source to now work with the new form of the **for** command.

Having explained that it is more simple to explain how the Checker class make sure the **for** command comply with its contextual restrictions, firstly as mentioned above now the code handles a **for declaration** so is required to check this one contextually for errors, the code below is in charge of that:

```
public Object
    visitForDeclaration(ForDeclaration
    ast, Object o){
   TypeDenoter eType = (TypeDenoter)
       ast.E.visit(this, null);
   ConstDeclaration cAst = new
       ConstDeclaration(ast.I, ast.E,
       ast.position);
   idTable.enter(cAst.I.spelling, cAst);
   if (cAst.duplicated)
      reporter.reportError ("identifier
          \"%\" already declared",
                                cAst.I.spelling,
                                    cAst.position);
   if(!(eType instanceof IntTypeDenoter))
      reporter.reportError ("wrong
          expression type, must be an
          integer type",
                          "",
                              ast.E.position);
   return null;
 }
```

The above code do the following actions, first gets type of expression, then initialize a new const declaration and checks for its contextual restrictions, lastly checks type of the expression that has to be type integer.
Once the checking for the declaration is ready the checking for the **for** command has all of his components ready to be checked and here is how it does it:

```
public Object
    visitForDoCommand(ForDoCommand ast,
    Object o){
   TypeDenoter eType = (TypeDenoter)
       ast.E.visit(this,null);
   if(!(eType instanceof IntTypeDenoter)){
      reporter.reportError("Wrong
          expression type. Integer type
          expected", "", ast.E.position);
   }
   idTable.openScope();
   ast.D.visit(this, null);
   ast.C.visit(this, null);
   idTable.closeScope();
   return null;
 }
```

The first step is to check for the type of Exp_2, remember Id and Exp_1 are checked in the method explained above, then it's opened a new scope in the table the declaration and command are checked there and the scope closes, note here that these way the contextual is decorating the AST and simultaneously is protecting the declaration, and as simple as that the **for** command gets checked for its contextual restrictions.

C. Initialized Variable Declaration Processing Solution

Firstly, when processing initialized variables, their expressions must be evaluated to determine and statically assign the variable type denoter. Once made, the variable name is stored in the identification table, and can be reassigned a different value. If the variable name is already previously stored within the identification table, the contextual analyzer must report an error that the variable name is already existent.

D. Validation of the Uniqueness of the Parameter Names in the Declarations of Functions or Procedures

In order to validate parameters when calling either a function or a procedure, the first thing to do is to open a scope and enter all the parameter names. These names are verified to check for repeated names within the parameter variable names; if such case occurs, the reporter will raise such error. When all the parameters within both functions and procedures have been processed and visited, the scope is closed and the remaining AST tree validation takes place.

E. ***Local*** *Compound-Declaration Processing*

In order to process local compound declarations, ***"local"*** *Declaration* ***"in"*** *Declaration* ***"end"***, the first thing to do is to start marking and processing the AST nodes as local (a.k.a. private). To be able to perform such action, a function is implemented in the Identification Table which opens a local scope; simply sets a new attribute (localScope) as true. By doing so, it enables the compiler to understand that the following declarations will be local to other declarations. If the first declaration of the sentence is local, it is processed as such, followed by a function closing the local scope in the Identification Table; changing the localScope to false. Afterwards, the remaining declarations are processed normally within the sentence. Once the complete verification of the **local** compound-declaration is processed, the nodes within the AST marked as local are removed and the rest of the tree is processed.

F. ***Recursive*** *Compound Statement Processing*

Processing a recursive call, with either procedures or functions, was a complex task. In order to achieve a correct contextual analysis, different methods were implemented until a final one succeeded. To process such sentences, three attributes were added to the Checker class: a list of ASTs, a level of recursion, and a boolean flag indicating if recursive processing is taking place. First, when visiting a recursive declaration, the flag is set to true. Regardless if it is a procedure or function, if such flag is true, the ASTs are added to the AST list declared, and when all the sub trees are added, they are visited with an auxiliary method. It is important to clearly state that this way to implement such processing was first given to the team by Soledad Kopper Gamboa[2], thus the respective credit is being given to her by acknowledging her help.

[2]Computer Engineering Student at ITCR.

G. New Contextual Analysis Routines Added

1) ***skip****:*

```
public Object
   visitEmptyCommand(EmptyCommand ast,
   Object o) {
   return null;
}
```

2) ***"loop" "while"*** *Expression* ***"do"*** *Command* ***"repeat"****:*

```
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);
   }
   idTable.openScope();
   ast.C.visit(this, null);
   idTable.closeScope();
   return null;
  }
```

3) ***"loop" "until"*** *Expression* ***"do"*** *Command* ***"repeat"****:*

```
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", "",
         ast.E.position);
   }
   idTable.openScope();
   ast.C.visit(this,null);
   idTable.closeScope();
   return null;
  }
```

4) ***"loop" "do"*** *Command* ***"while"*** *Expression* ***"repeat"****:*

```
public Object
    visitDoWhileCommand(DoWhileCommand
    ast, Object o){
   TypeDenoter eType = (TypeDenoter)
       ast.eAST.visit(this,null);
   if(!eType.equals
          (StdEnvironment.booleanType)){
      reporter.reportError("Boolean
          expression expected", "",
          ast.eAST.position);
   }
   idTable.openScope();
   ast.cAST.visit(this,null);
   idTable.closeScope();
   ast.eAST.visit(this,null);
   return null;
 }
```

*5) "**loop**" "**do**" Command "**until**" Expression "**repeat**":*

```
public Object
    visitDoUntilCommand(DoUntilCommand
    ast, Object o){
   TypeDenoter eType = (TypeDenoter)
       ast.eAST.visit(this,null);
   if(!eType.equals
          (StdEnvironment.booleanType)){
      reporter.reportError("Boolean
          expression expected", "",
          ast.eAST.position);
   }
   idTable.openScope();
   ast.cAST.visit(this,null);
   idTable.closeScope();
   ast.eAST.visit(this,null);
   return null;
 }
```

*6) "**loop**" "**for**" Identifier "**~**" Expression "**to**" Expression "**do**" Command "**repeat**":*

```
public Object
    visitForDeclaration(ForDeclaration
    ast, Object o){
   TypeDenoter eType = (TypeDenoter)
       ast.E.visit(this, null);
   ConstDeclaration cAst = new
       ConstDeclaration(ast.I, ast.E,
       ast.position);
   idTable.enter(cAst.I.spelling, cAst);
   if (cAst.duplicated)
      reporter.reportError ("identifier
          \"%\" already declared",
                    cAst.I.spelling,
                                   cAst.position);
   if(!(eType instanceof IntTypeDenoter))
      reporter.reportError ("wrong
          expression type, must be an
          integer type",
                         "",
                                    ast.E.position);
   return null;
 }
```

*7) "**let**" Declaration "**in**" Command "**end**":*

```
public Object visitLetCommand(LetCommand
    ast, Object o) {
   idTable.openScope();
   ast.D.visit(this, null);
   ast.C.visit(this, null);
   idTable.closeScope();
   return null;
 }
```

*8) "**if**" Expression "**then**" Command "**else**" Command "**end**":*

```
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;
 }
```

*9) "**recursive**" Proc-Funcs "**end**":*

```
public final class Checker implements
    Visitor {
   //ATTRIBUTES IMPLEMENTED FOR RECURSIVE
       DECLARATION CONTEXTUAL ANALYSIS
   private ArrayList<Object> astList =
       new ArrayList<>();
   private boolean visitRecursive = false;
   private int nestedLevel = 0;

   public Object visitRecursiveDeclaration
          (RecursiveDeclaration ast,
             Object o){
   visitRecursive = true;
   nestedLevel++;
   ast.procFuncAST.visit(this, null);
   nestedLevel--;
```

```
    if(nestedLevel == 0){
      visitRecursive = false;
      visitRecursiveDeclarationNested();
    }
    return null;
  }

  public void
      visitRecursiveDeclarationNested(){
      for(Object ast : astList){
        if(ast instanceof
            ProcDeclaration){
          idTable.openScope();
          ((ProcDeclaration)ast)
              .FPS.visit(this, null);
          ((ProcDeclaration)ast)
              .C.visit(this, null);
          idTable.closeScope();
        }
        else if(ast instanceof
            FuncDeclaration){
            idTable.openScope();
            ((FuncDeclaration)ast).
              FPS.visit(this, null);
            TypeDenoter eType =
                (TypeDenoter)
              ((FuncDeclaration)ast)
                  .T.visit(this, null);
            idTable.closeScope();
            if(!((FuncDeclaration)ast)
              .T.equals(eType)){
                reporter.reportError("body
                    of function \"%\" has
                    wrong type",
                    ((FuncDeclaration)ast)
                      .I.spelling,
                      ((FuncDeclaration)ast)
                        .E.position);
            }
        }
      }
      astList.clear();
  }
}
```

```
public Object
    visitProcDeclaration(ProcDeclaration
    ast, Object o) {
    idTable.enter (ast.I.spelling, ast);
        // permits recursion
    if (ast.duplicated){
        reporter.reportError ("identifier
            \"%\" already declared",
            ast.I.spelling, ast.position);
    }
    //CHANGES IMPLEMENTED
    if(visitRecursive){
        astList.add(ast);
    }
    else{
        idTable.openScope();
        ast.FPS.visit(this, null);
        ast.C.visit(this, null);
        idTable.closeScope();
    }
    return null;
}

public Object
    visitFuncDeclaration(FuncDeclaration
    ast, Object o) {
    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);
    }
    //CHANGES IMPLEMENTED
    if(visitRecursive){
        astList.add(ast);
    }
    else{
        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);
    }
    return null;
}
```

*10) ”**local**” Declaration ”**in**” Declaration ”**end**”:*

```
public Object
    visitLocalDeclaration(LocalDeclaration
    ast,Object o){
    idTable.openLocalScope(); //Define
        declarations as local
    if(ast.dcl1 instanceof
        LocalDeclaration){
         visitLocalDeclarationNested(
         (LocalDeclaration)ast.dcl1,o);
            //Validate if asignments are
            nested
      }
      else{
         ast.dcl1.visit(this, null);
      }
      idTable.closeLocalScope();
      ast.dcl2.visit(this, null);
      idTable.clearLocalScope();
      if(ast.dcl1 instanceof
          LocalDeclaration){
         idTable.clearLocalScope();
      }
      return null;
   }

   public Object visitLocalDeclarationNested
         (LocalDeclaration dcl, Object o){
      dcl.dcl1.visit(this, null);
      dcl.dcl2.visit(this, null);
      return null;
   }
```

```
public final class IdentificationTable{
   //...
   protected boolean localScope = false;

   public void openLocalScope(){
      localScope = true; //Start marking
         nodes as local
   }

   public void closeLocalScope(){
      localScope = false; //Stop marking
         nodes as local
   }

   public void clearLocalScope(){
      //Eliminate last level of nodes
         marked as local
      IdEntry entry, local,
         localDeclaration;
      entry = this.latest;
      localDeclaration =
         this.latest.previous;
      while(localDeclaration.localLevel
         != true){
         local = entry;
         entry = local.previous;
         localDeclaration =
            local.previous;
      }
      entry.previous =
         localDeclaration.previous;
      this.latest = entry;
   }
}
```

```
public class IdEntry{
   //...
   protected boolean localLevel; //MARKS
      IF NODE IS LOCAL
}
```

*11) **"var"** Identifier **"init"** Expression:*

```
public Object visitVarDeclarationInit(
      VarDeclarationInit ast, Object o){
   TypeDenoter eType = (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;
}
```

H. Contextual Errors Detected

- \"%\" is not a procedure identifier
- Boolean expression expected
- Wrong expression type. Integer type expected
- \"%\" is not a binary operator
- incompatible argument types for \"%\"
- wrong argument type for \"%\"
- \"%\" is not a function identifier
- incompatible limbs in if-expression
- \"%\" is not a unary operator
- identifier \"%\" already declared
- body of function \"%\" has wrong type
- incompatible array-aggregate element
- duplicate field \"%\" in record
- duplicated formal parameter \"%\"
- const actual parameter not expected here
- wrong type for const actual parameter
- func actual parameter not expected here
- wrong signature for function \"%\"
- wrong type for function \"%\"
- proc actual parameter not expected here
- wrong signature for procedure \"%\"

- actual parameter is not a variable
- var actual parameter not expected here
- wrong type for var actual parameter
- too few actual parameters
- too many actual parameters
- incorrect number of actual parameters
- arrays must not be empty
- \"%\" is not a type identifier
- record expected here
- no field \"%\" in this record type
- \"%\" is not a const or var identifier
- array expected here
- \"%\" is not declared
- LHS of assignment is not a variable
- assignment incompatibilty

I. Test Plan to Validate the Compiler

Case Objective: loop do Com **until** Exp **repeat**
Case Design:

```
loop do skip until 3=3 repeat
!OK
```

Expected Results:Compilation successful Observed Results: Syntactic Analysis ... Contextual Analysis ... Compilation was successful.

Case Objective: loop do Com **until** Exp **repeat** error
Case Design:

```
loop do skip until 5 repeat
!ERROR
```

Expected Results: error on expression Observed Results: Syntactic Analysis ... Contextual Analysis ... ERROR: Boolean expression expected 1..1 Compilation was unsuccessful.

Case Objective: loop do Com **while** Exp **repeat**
Case Design:

```
loop do skip until 5=5 repeat
!OK
```

Expected Results:compilation successful Observed Results: Syntactic Analysis ... Contextual Analysis ... Compilation was successful.

Case Objective: "loop do" Com **while** Exp **repeat** error
Case Design:

```
loop do skip until 5 repeat
!ERROR
```

Expected Results: error on expression type Observed Results: Syntactic Analysis ... Contextual Analysis ... ERROR: Boolean expression expected 1..1 Compilation was unsuccessful.

Case Objective: "loop until" Exp **do** Com **repeat**
Case Design:

```
loop until 5=5 do skip repeat
!OK
```

Expected Results: compilation successful Observed Results: Syntactic Analysis ... Contextual Analysis ... Compilation was successful.

Case Objective: "loop until" Exp **do** Com **repeat** error
Case Design:

```
loop until 5 do skip repeat
!ERROR
```

Expected Results: error on expression type Observed Results: Syntactic Analysis ... Contextual Analysis ... ERROR: Boolean expression expected 1..1 Compilation was unsuccessful.

Case Objective: "loop while" Exp **do** Com **repeat**
Case Design:

```
loop while 5=5 do skip repeat
!OK
```

Expected Results: compilation successful Observed Results: Syntactic Analysis ... Contextual Analysis ... Compilation was successful.

Case Objective: "loop while" Exp **do** Com **repeat** error Case Design:

```
loop while 5 do skip repeat
!ERROR
```

Expected Results: error on expression type Observed Results: Syntactic Analysis ... Contextual Analysis ... ERROR: Boolean expression expected 1..1 Compilation was unsuccessful.

Case Objective: loop for $Id \sim Exp_1$ **to** Exp_2 **do** Com **repeat**

Case Design:

```
loop for id ˜ 3 to 5 do skip repeat
!OK
```

Expected Results: compilation successful Observed Results: Syntactic Analysis ... Contextual Analysis ... Compilation was successful.

Case Objective: loop for $Id \sim Exp_1$ **to** Exp_2 **do** Com **repeat** Exp_1 type error

Case Design:

```
loop for id ˜ 3=3 to 5 do skip repeat
!ERROR
```

Expected Results: type error Observed Results: Syntactic Analysis ... Contextual Analysis ... ERROR: wrong expression type, must be an integer type 1..0 Compilation was unsuccessful.

Case Objective: loop for $Id \sim Exp_1$ **to** Exp_2 **do** Com **repeat** Exp_2 type error

Case Design:

```
loop for id ˜ 3 to '5' do skip repeat
!ERROR
```

Expected Results: type error Observed Results: Syntactic Analysis ... Contextual Analysis ... ERROR: wrong expression type, must be an integer type 1..0 Compilation was unsuccessful.

Case Objective: loop for $Id \sim Exp_1$ **to** Exp_2 **do** Com **repeat** Id error

Case Design:

```
loop for id ˜ 3 to id do skip repeat
!ERROR
```

Expected Results: error, id is not known to expressions Observed Results: Syntactic Analysis ... Contextual Analysis ... ERROR: "id" is not declared 1..1 ERROR: Wrong expression type. Integer type expected 1..1 Compilation was unsuccessful.

Case Objective: loop for $Id \sim Exp_1$ **to** Exp_2 **do** Com **repeat**

Case Design:

```
loop for id ˜ 3 to 5 do
let
   const i ˜ id
in
   skip
end repeat
!OK
```

Expected Results: compilation successful Observed Results: Syntactic Analysis ... Contextual Analysis ... Compilation was successful.

Case Objective: loop for $Id \sim Exp_1$ **to** Exp_2 **do** Com **repeat**

Case Design:

```
let
  var d : Boolean
in
  loop for i ˜ 1 to d do
    skip
  repeat
end
!Error d is not integer
```

Expected Results: Error, Contextual Error, D is not a integer. Observed Results: Syntactic Analysis ... Contextual Analysis ... ERROR: wrong expression type, must be an integer type 4..4 Compilation was unsuccessful.

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

Case Design:

```
let var a:Integer
in
a := 2
end
```

Expected Results: It is expected a successful compilation Observed Results: Syntactic Analysis ... Contextual Analysis ... Compilation was successful.

Case Objective: var structure validation

Case Design:

```
let var a init Integer
in
a := 2
end
```

Expected Results: It is expected a successful compilation Observed Results: Syntactic Analysis ... Contextual Analysis ... Compilation was successful.

Case Objective: var structure validation

Case Design:

```
let var a init Integer
in
a := 2
end
```

Expected Results: It is expected a successful compilation Observed Results: Syntactic Analysis ... Contextual Analysis ... Compilation was successful.

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

Case Design:

```
let type fecha˜record
d:Integer,
m:Integer,
y:Integer
end
in
let var a:fecha
in
skip
end
end
```

Expected Results: It is expected a successful compilation Observed Results: Syntactic Analysis ... Contextual Analysis ... Compilation was successful.

Case Objective: proc structure validation

Case Design:

```
let proc
hola(mensaje:Char)˜
put(mensaje)
end;
var msj:Char
in
msj := 'h';
hola(msj)
end
```

Expected Results: It is expected a successful compilation Observed Results: Syntactic Analysis ... Contextual Analysis ... Compilation was successful.

Case Objective: let ERROR Statement now exists

Case Design:

```
let var a:Integer; var
a:Integer
in
skip
end
```

Expected Results: The identifier ”a” already is declared Observed Results: Syntactic Analysis ... Contextual Analysis ... ERROR: identifier ”a” already declared 1..1 Compilation was unsuccessful.

Case Objective: var ERROR Types do not match

Case Design:

```
let var a:Integer
in
a := '2'
end
```

Expected Results: The types are not compatible Observed Results: Syntactic Analysis ... Contextual Analysis ... ERROR: assignment incompatibility 3..3 Compilation was unsuccessful.

Case Objective: proc ERROR Parameter type bad

Case Design:

```
let proc
hola(mensaje:Char)˜
put(mensaje)
end;
var msj:Integer
in
msj := 1;
hola(msj)
end
```

Expected Results: Parameter type not in agreement with the one passed Observed Results: Syntactic Analysis ... Contextual Analysis ... ERROR: wrong type for const actual parameter 7..7 Compilation was unsuccessful.

Case Objective: func ERROR Type of return no agrees with the statement

Case Design:

```
let func
suma(a:Integer,
b:Integer):Integer˜
'a';
var a:Integer
in
a := suma(12,12)
end
```

Expected Results: The type of return does not match the declaration Observed Results: Syntactic Analysis ... Contextual Analysis ... ERROR: body of function ”suma” has wrong type 2..2 Compilation was unsuccessful.

Case Objective: var init exp

Case Design:

```
let var a:Integer; var
a::= 0
in
skip
end
```

Expected Results: Don’t work because in the code we use ::= and not init Observed Results: Syntactic Analysis ... ERROR: ”:=” cannot start a type denoter 2..2 Compilation was unsuccessful.

Case Objective: recursive compound declaration OK

Case Design:

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

Expected Results: Successful compilation Observed Results: Syntactic Analysis ... Contextual Analysis ... Compilation was successful.

Case Objective: recursive compound declaration error

Case Design:

```
let
 recursive
  func f (a : Integer): Integer ~ 1
  and
  func g (b : Boolean , x : Integer , b
      : Char): Integer ~ 2 ! b repetido
 end
in
 putint(f(10))
end
```

Expected Results: Error in compilation process since variable b is declared twice Observed Results: Syntactic Analysis ... Contextual Analysis ... ERROR: duplicated formal parameter ”b” 9..9 Compilation was unsuccessful.

Case Objective: recursive compound declaration error 2

Case Design:

```
let
 recursive
  func f (): Integer ~ 1
  and
  func g (b : Integer): Integer ~ 3
  and
  func f (a : Integer): Integer ~ 2
 end
in
 putint(f(10))
end
```

Expected Results: Error in compilation process since function f is declared twice Observed Results: Syntactic Analysis ... Contextual Analysis ... ERROR: identifier ”f” already declared 13..13 Compilation was unsuccessful.

Case Objective: local compound declaration OK

Case Design:

```
let
 local
  const a ~ 1
 in
  const b ~ a + 1
 end
in
 putint(b)
end
```

Expected Results: Successful compilation Observed Results: Syntactic Analysis ... Contextual Analysis ... Compilation was successful.

Case Objective: local compound declaration error

Case Design:

```
let
 local
  const a ~ 1
 in
  const b ~ a + 1
 end
in
 put(a)
```

```
end
```

Expected Results: Compilation error since a should not be visible Observed Results: Syntactic Analysis ... Contextual Analysis ... ERROR: ”a” is not declared 12..12 Compilation was unsuccessful.

III. Results and Discussion

A. Discussion and Analysis of Results Obtained

As can be seen in the test cases the results are quite the expected. However, this does not mean it was an easy and simple process to get there, there were several simple changes that make a complex solution, that is the case of the conversion from quaternary form of the AST to ternary form in the **for** command and in the **recursive** command as well; these changes made look pretty simple but it took several hours to think how to implement those changes and get the compiler working as intended.

B. Reflection on the Experience of Modifying Fragments of a Compiler / Environment Written by Third Parties

At the end of the requested instructions, we as students are satisfied to overcome the different challenges present, both having to understand the code already designed by other people as well as being able to include in it the necessary restrictions so that by including new sentences these are executed correctly by performing a satisfactory contextual analysis. The knowledge acquired when passing from the theory to the practice is many, and very enriching.

C. Summary Description of the Tasks Performed by Each Member of the Group of Job

1) Ricardo Molina-Brenes: Responsible for the validation of the uniqueness of the parameter names in the declarations of functions or procedures, initialized variable declaration processing solution, both positive and negative test generator and documentation.

2) Andres Miranda-Arias: Implementation of different visitor methods in the Checker class, as well as responsible for the implementation of the recursive and compound declaration sentences completely. Helped modify the quaternary structure of the For Do Command and implement it as a ternary structure. Also in charge of developing the test cases for the recursive and local compound declaration sentences.

3) Israel Padilla-Jenkins: Implementation of the ternary form for the **For Do** command, added **For Declaration** class, visitors and functions needed in order to make work the **For Do** command in ternary form. Work in **For Do** command test cases. Worked in documentation.

4) Jose Campos-Espinoza: Collaborate in Checker class and enabling the compiler to use the Contextual analyzer and table details properly. In charge of testing **loop ... do** variants and **loop for ... do** command, contribute to document the Contextual analysis on the commands mentioned previously.

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.

The other way to execute, is to open the source code project in an IDE (NetBeans 8.2 is recommended), and manually execute the program by clicking the green right arrow. The process is as described previously above.

IV. Conclusion

The project was completed to its full extension, and in correct functionality. The results obtained throughout the test plan were satisfactory, compilation failed or succeeded when expected and the program behaved as it should. In general, the most complicated section to do was implementing the correct contextual analysis for the recursive compound declaration sentences, since the recursive declaration of several ASTs and its processing was a difficult task to do; once again, we are grateful with Soledad Kopper for her assistance in this task. Overall, the project took every member to new limits in their thinking process, and the expectations for the new upcoming third project, Code Generation and Interpretation, are high.

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 #2, la fase de análisis contextual de un compilador, 1st 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_Contextual_Analyzer