Triangle^{+} Compiler
Code Generation and Interpretation

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, a language extension was performed, which resulted in Δ^{+}. Different changes were made to the code generator and interpretation 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. All changes made were made respecting the existing format and coding 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 compiler cannot compile when necessary and is compiled correctly when using the correct syntax and the correct context finally generating the specific code. Thus completing the scope of the project correctly.**

***Index Terms*—**

CONTENTS

I. Introduction

The objective of this project is to understand the details of the code generation and interpretation of a compiler according to the techniques outlined by David A. Watt and Deryck F. Brown in his book *Java programming language processors: compilers and interpreters* [1] The compiler base used was for the Triangle Δ language, developed by Watt and Brown. The purpose and scope of this project is to modify an existing compiler, in order to process an 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 students of Computer Engineering of Costa Rica Technology. [2]

II. Methodology

*A. Description of the code generation scheme for the command **skip**.*

The **skip** command should not generate anything but is taken into consideration by the contextual analyzer, when it gets to the Encoder it just simply return null as the visitor method for the Encoder do it. In this case it does not generate or call anything else it just pass by like ignoring it.

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

*B. Description of the code generation scheme for the command **if then else end**.*

*C. Description of the code generation scheme for the command **loop ... repeat** and its variants.*

As we describe in the previous documentations for both Lexical and Contextual analyzers the variants of these command are:

- loop while E do C repeat
- loop until E do C repeat
- loop do C while E repeat
- loop do C until E repeat

In that order we are going to explain what the process is to generate code for the TAM machine and its result as well.

1) loop while E do C repeat: Firstly a review of how **while** command works. It first evaluates its expression denoted by E, if the evaluation gives a true value executes the command, then it will reevaluate if true again it will execute the command, if not the execution of the command ends. The command can be executed 0 times if the first evaluation gives a true value.

```
public Object
    visitWhileCommand(WhileCommand ast,
    Object o) {
  Frame frame = (Frame) o;
  int jumpAddr, loopAddr;

  jumpAddr = nextInstrAddr;
  emit(Machine.JUMPop, 0, Machine.CBr, 0);
  loopAddr = nextInstrAddr;
  ast.C.visit(this, frame);
  patch(jumpAddr, nextInstrAddr);
  ast.E.visit(this, frame);
  emit(Machine.JUMPIFop, Machine.trueRep,
      Machine.CBr, loopAddr);
  return null;
}
```

The above code is the responsible to generate TAM code for this command, as explained on Watt and Brown's book it uses a technique called back-patching, in a nutshell it generates a jump instruction whit a given value, it doesn't matter the value because it will be changed, this jump instruction to go to the evaluation part of the code, note at the patch call just after ast.C.visit(...), it updates the position in which the jump instruction has to go. To give a more visual example below is going to be the TAM code generated by this simple command **loop while** $i > 1$ **do** $i := i - 2$.

```
30: JUMP 35
31: LOAD i
32: LOADL 2
33: CALL sub
34: STORE i
35: LOAD i
36: LOADL 1
37: CALL gt
38: JUMPIF(1) 30
```

Note the first action is evaluate the expression, then it depends on the value given by the evaluation, if true the code is executed if not the program continues or halts.

2) loop until E do C repeat: Until has a slightly difference with while, is like the opposite. Here is how it works, the evaluation is first as in the while command but if the result value of these evaluation is false the command denoted by C is executed then it reevaluates and determine if it can be executed again based on the value given by the expression, if this value is true the execution ends. Below you will see the changes on the code responsible to generate TAM machine code for these command.

```
public Object
    visitUntilCommand(UntilCommand ast,
    Object o) {
  Frame frame = (Frame) o;
  int jumpAddr, loopAddr;
  ...
  emit(Machine.JUMPIFop,
      Machine.falseRep, Machine.CBr,
      loopAddr);
  return null;
}
```

And yes, that's it, just change Machine.trueRep with Machine.falseRep, now the **until** command works as is supposed to. To prove it here is the TAM code generated by a command nearly identical as the while command used before. **loop until** $i > 1$ **do** $i := i - 2$.

```
30: JUMP 35
31: LOAD i
32: LOADL 2
33: CALL sub
34: STORE i
35: LOAD i
36: LOADL 1
37: CALL gt
38: JUMPIF(0) 30
```

3) loop do C while E repeat: These command is simpler than the previous ones explained, the reason is that it does the evaluation after the code is executed so no back-patching required, the evaluation works as the while command before, if the evaluation gives a true value the code is executed and if not the execution ends, note that even if the first time the evaluation gives a false value the code is executed. Below is the code responsible to generate TAM machine code.

```
public Object
    visitDoWhileCommand(DoWhileCommand
    ast, Object o){
  Frame frame = (Frame) o;
  int loopAddr;

  loopAddr = nextInstrAddr;
  ast.cAST.visit(this, frame);
  ast.eAST.visit(this, frame);
  emit(Machine.JUMPIFop, Machine.trueRep,
      Machine.CBr, loopAddr);
  return null;
}
```

Is visible that it is simpler and has less elements, this method generates the code to work as intended. Here is the TAM code generated for a simple command **loop do** $i := i - 2$ **while** $i > 1$.

```
31: LOAD i
32: LOADL 2
33: CALL sub
34: STORE i
35: LOAD i
36: LOADL 1
37: CALL gt
38: JUMPIF(1) 31
```

4) loop do C until E repeat: At this point there is not much that has to be explained, this method works just as the one above and the evaluation as the other until command, anyway here is the code that generates the TAM and its generated code.

```
public Object
    visitDoUntilCommand(DoUntilCommand
    ast, Object o){
  Frame frame = (Frame) o;
  int loopAddr;
```

```
  loopAddr = nextInstrAddr;
  ast.cAST.visit(this, frame);
  ast.eAST.visit(this, frame);
  emit(Machine.JUMPIFop,
      Machine.falseRep, Machine.CBr,
      loopAddr);
  return null;
}
```

```
31: LOAD i
32: LOADL 2
33: CALL sub
34: STORE i
35: LOAD i
36: LOADL 1
37: CALL gt
38: JUMPIF(0) 31
```

*D. Description of the code generation scheme for the command **loop for ... repeat**.*

First, the For Do Command checker was improved by opening the scope first and making the visits after:

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

The implementation of the For Do Command encoder is the next:

```
public Object
    visitForDoCommand(ForDoCommand ast,
    Object o)
  {
    Frame frame = (Frame) o;
    int expressionSize,loopRepeat,loopCond;
    expressionSize =
        (Integer)ast.E.visit(this, frame);

    ast.D.visit(this, new
        Frame(frame.level, frame.size +
        expressionSize));
    loopCond = nextInstrAddr;
    emit(Machine.JUMPop, 0, Machine.CBr,
        0);
    loopRepeat = nextInstrAddr;
    ast.C.visit(this, frame);
    emit(Machine.CALLop, frame.level,
        Machine.PBr,
        Machine.succDisplacement);
    patch(loopCond, nextInstrAddr);
    emit(Machine.LOADop,2, Machine.STr,-2);
    emit(Machine.CALLop, frame.level,
        Machine.PBr,
        Machine.geDisplacement);
    emit(Machine.JUMPIFop,
        Machine.trueRep, Machine.CBr,
        loopRepeat);
    emit(Machine.POPop, 0, 0, 2);

    return null;
  }
```

Which can be explained as How the declaration and the expression are the loop limits and how the code proceeds to execute the command contained in the AST within its limits. Also the TAM code generated in one example is the next:

```
0: LOADL    10
1: LOADL    1
2: JUMP     4[CB]
3: CALL     succ
4: LOAD (2) -2[ST]
5: CALL     ge
6: JUMPIF(1) 3[CB]
7: POP  (0) 2
8: HALT
```

In this case the example used is a loop from 1 to 10 with the skip command used as Command.

*E. Solution given to introduce private declarations (**local**).*

A tour of the trees of abstract syntax of both declarations is made, which is done by means of the visit () method followed to that route we take the numeric value of this object through the intValue () function. After the

representation both values are added and the total of the sum of that numerical representation is returned.

```
    Frame frame = (Frame) o;
    int extraSize1 = ((Integer)
        ast.dcl1.visit(this,
        frame)).intValue();
    int extraSize2 = ((Integer)
        ast.dcl2.visit(this,
        frame)).intValue();
    int total = extraSize1+extraSize2;
    return total;
```

*F. Solution to introduce procedures and functions mutually recursive (**recursive**).*

G. New routines added to code generation or interpretation, and modifications made to the existing ones.

In order to generated the TAM code of the **loop** command variants the following methods were modified in the Encoder class. **visitDoWhileCommand** from

```
public Object
    visitDoWhileCommand(DoWhileCommand
    ast, Object o){
  ast.cAST.visit(this, frame);
  ast.eAST.visit(this, frame);
  return null;
}
```

To

```
public Object
    visitDoWhileCommand(DoWhileCommand
    ast, Object o){
  Frame frame = (Frame) o;
  int loopAddr;

  loopAddr = nextInstrAddr;
  ast.cAST.visit(this, frame);
  ast.eAST.visit(this, frame);
  emit(Machine.JUMPIFop, Machine.trueRep,
      Machine.CBr, loopAddr);
  return null;
}
```

visitDoUntilCommand from:

```
public Object
    visitDoUntilCommand(DoUntilCommand
    ast, Object o){
  ast.cAST.visit(this, frame);
  ast.eAST.visit(this, frame);
  return null;
}
```

To

```
public Object
    visitDoUntilCommand(DoUntilCommand
    ast, Object o){
  Frame frame = (Frame) o;
  int loopAddr;

  loopAddr = nextInstrAddr;
  ast.cAST.visit(this, frame);
  ast.eAST.visit(this, frame);
  emit(Machine.JUMPIFop,
      Machine.falseRep, Machine.CBr,
      loopAddr);
  return null;
}
```

visitUntilCommand from:

```
public Object
    visitUntilCommand(UntilCommand
    ast,Object o){
  ast.C.visit(this, frame);
  ast.E.visit(this, frame);
  return null;
}
```

To

```
public Object
    visitUntilCommand(UntilCommand
    ast,Object o){
    Frame frame = (Frame) o;
    int jumpAddr, loopAddr;

    jumpAddr = nextInstrAddr;
    emit(Machine.JUMPop, 0, Machine.CBr,
        0);
    loopAddr = nextInstrAddr;
    ast.C.visit(this, frame);
    patch(jumpAddr, nextInstrAddr);
    ast.E.visit(this, frame);
    emit(Machine.JUMPIFop,
        Machine.falseRep, Machine.CBr,
        loopAddr);
    return null;
  }
```

H. List of errors detected in code generation or execution.

- length of operand can't exceed 255 words
- too many instructions for code segment
- can't access data more than 6 levels out
- can't store values larger than 255 words
- can't load values larger than 255 words
- can't nest routines more than 7 deep
- can't nest routines so deeply

I. Describe any modification made to the IDE.

In the first project, some modifications were made in order to avoid the program from crashing due to the expectation of contextual analysis and code generation taking place. For the second project, contextual analysis was enabled, while for this final project, code generation and interpretation was enabled; the compiler generates code and, if successful, the **run** button is enabled to allow the user to actually run the program and verify if it does what is is supposed to do if written correctly.

J. Test Plan to Validate the Compiler

Case Objective: Test **loop while** E **do** C **repeat**.
Case Design:

```
let
  var a : Integer
in
  a := 9;
  loop
  while a > 10 do

    putint(a) ;
    a := a - 2

  repeat

end
```

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

Case Objective: Test **loop while** E **do** C **repeat**.
Case Design:

```
let
  const b ~ 5;
  var a init 10
in
  loop
  while a > b do

    putint (a) ; ! debe imprimir 10, 9, 8,
       7, 6
    a := a - 1

  repeat

end
```

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

Case Objective: Test **loop until** E **do** C **repeat**.
Case Design:

```
let
  var x : Integer
in
  x := 1 ;
  loop
  until x > 0 ! NO debe imprimir
  do
    putint (x);
    x := x +1
  repeat
end
```

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

Case Objective: Test **loop until** E **do** C **repeat**.
Case Design:

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

    putint (x);
    x := x +1

  repeat

end
```

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

Case Objective: Test **loop do** C **while** E **repeat**.
Case Design:

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

    putint(x);
    x := x +1

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

end
```

Expected Results: successful compilation Observed Results: Syntactic Analysis ... Contextual Analysis ... Code Generation ... Compilation was successful.
Case Objective: Test **loop do** C **until** E **repeat**.
Case Design:

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

    putint (x);
    x := x +1

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

end
```

Expected Results: successful compilation Observed Results: Syntactic Analysis ... Contextual Analysis ... Code Generation ... Compilation was successful.
Case Objective: Loop For Do Command OK
Case Design:

```
let
  var k : Integer;
  const b ~ 10
in
  k := 1;
  loop for i ~ k to b do
    putint (k); ! debe imprimir 1,2,3...10
    k:=k+1
  repeat
end
```

Expected Results: Successful Compilation. Observed Results: Syntactic Analysis ... Contextual Analysis ... Code Generation ... Compilation was successful. Code Generated: 1 2 3 4 5 6 7 8 9 10 Program has halted normally.
Case Objective: Loop For Do Command OK2
Case Design:

```
let
  var k : Integer
in
  k := 1;
  loop for i ~ k to 10 do
    putint (1); ! debe imprimir 10 unos
    k:=k+1
  repeat
end
```

Expected Results: Successful Compilation. Observed Results: Syntactic Analysis ... Contextual Analysis ... Code Generation ... Compilation was successful. Code Generated: 1 1 1 1 1 1 1 1 1 1 Program has halted normally.
Case Objective: Loop For Do Command Error1
Case Design:

```
loop for h ~ 1 to 10 do
  h := h + 1 ; ! debe dar un error
  putint (h) ! debe imprimir del 1 al 10
repeat
```

Expected Results: Unsuccessful Compilation. Observed Results: Syntactic Analysis ... Contextual Analysis ... ERROR: LHS of assignment is not a variable 2..2 Compilation was unsuccessful.
Case Objective: Loop For Do Command Error2
Case Design:

```
let
  var d : Boolean
in
  loop for i ~ d to 10 do ! d debe ser
     entera
    putint (d)
  repeat
end
```

Expected Results: Unsuccessful Compilation. Observed Results: Syntactic Analysis ... Contextual Analysis ... ERROR: wrong expression type, must be an integer type 4..4 ERROR: wrong type for const actual parameter 5..5 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 ... Code Generation ... 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.

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: var init exp

Case Design:

```
let
  var x init 1 * 3 + 2

in
  putint (x) ! imprime 5
end
```

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

Case Objective: var init exp

Case Design:

```
let
  var x init 1
in
  putint (x); ! debe imprimir 1
  puteol ();
  getint (var x); ! variable puede ser
     pasada por referencia
  putint (x) ! debe imprimir lo que se
     ingrese
end
```

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

Case Objective: var init exp

Case Design:

```
let
  var x :) 1 * 3 + 2

in
  putint (x) ! imprime 5
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: var init exp

Case Design:

```
let
  var x == 1
in
  putint (x); ! debe imprimir 1
  puteol ();
  getint (var x); ! variable puede ser
     pasada por referencia
  putint (x) ! debe imprimir lo que se
     ingrese
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.

III. Results and Discussion

A. Discussion and Analysis of Results Obtained

The implementation of **loop** command and its variants can be considered successful, at first it wasn't there were problems in the logic of the code generation due to a misunderstanding in the concepts of back-patching and even a direction that was left alone and generate a jump instruction with a wrong direction. Those problems were identified in testing and corrected, then we do more extensive testing and get the expected results.

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

Taking a look at the past reflections for the Lexical and Contextual Analyzers, in that order, the first one was kind of simple, then in the second one the difficulty increased in a way that we had to understand better what we did in the first one to understand the second one. In general, the lexical analyzer can be implemented just by looking at what you have to do. Contextual analysis requires more attention and study of the compiler model. Finally, in the code generation phase, we really have to understand the concepts used in the compiler, and more importantly, what should be the code generated to ensure the compiler is doing the work right.

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

1) Jose Campos-Espinoza: Implemented **loop** command variants, test and made sure it works properly. Enable code generation and visualization in the IDE.

2) Ricardo Molina-Brenes: Implemented initialized variable declaration processing solution, both positive and negative test generator and documentation. Also in charge of developing the test cases for local compound declaration sentences and solution in the code.

3) Israel Padilla-Jenkins: Implemented **loop for do** command generation code in Encoder-class and modified the method in checker-class due to improvements. Generation of test cases for **loop for do**. Worked in Documentation.

4) Andres Miranda-Arias: Implemented **recursive** compound declaration in the encoder class to allow code generation. In charge of test plan for such sentence as well; helped with final project documentation.

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

Overall, the project was completed to its full extent, with the exception of the recursive compound declaration. The biggest difficulty was understanding the TAM code, and being able to extrapolate that code into our project, to be able to do the triangle code generation correctly and thus running successfully the programs in the test cases. This final project, Code Generation and Interpretation, was the most complicated of the three projects performed. It is believed that, with more time, a better approach could have been implemented, and maybe the recursive compound declaration code generation could have been implemented correctly. As well, different syntactic and contextual rules could have been added to elevate the complexity of the triangle compiler; extending even more the Δ^+ compiler.

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 #3, Las fases de síntesis: generación e interpretación de código, 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_Code_Generation_and_Interpretation