Proyecto #1
Fase de análisis léxico y sintáctico

IC-5701 Compiladores e Intérpretes

versión 1.0

Preparado por:

Melanie Bermúdez Campos
2016077991

Alejandro Tapia Barboza
2016167784

Carlos Vega López
2016200200

8 de Junio del 2020

Introducción

Este documento tiene la intención de evidenciar todo el proceso de planteamiento y desarrollo para efectuar la primera fase correspondiente al análisis léxico y sintáctico de un compilador. El código de ejecución del compilador fue originalmente desarrollado por Watt y Brown en el libro *Programming Lenguage Processor in Java* y proporionado por el profesor Ignacio Trejos Zelaya quien a su vez suministra un IDE para la compilación y ejecución, desarrollado por el Dr Luis Leopoldo Perez.

Esquema para el manejo de texto fuente

El texto fuente se recibe por medio de archivos de extensión .tri, no se hicieron modificaciones al esquema de lectura de archivos de entrada o de estructura de datos al IDE, se trabajó bajo la recomendación de uso del profesor.

Modificaciones hechas al analizador léxico

Al analizador léxico se le añadieron nuevas palabras reservadas alternativas en la especificación de Tokens entre las cuales se incluyen: *and, elsif, exit, nil, private, req, repeat, return, to, until.* Además, se elimina la palabra reservada *begin.* El formato de escritura sigue las normas de letras mayúsculas y minúsculas para identificadores y palabras significativas.

Archivo Token.java

```
// Token classes...

public static final int

  // literals, identifiers, operators...
  INTLITERAL  = 0,
  CHARLITERAL = 1,
  IDENTIFIER  = 2,
  OPERATOR  = 3,

  // reserved words - must be in alphabetical order...
  AND     = 4, //nueva palabra reservada
  ARRAY   = 5,
  //BEGIN = 5, se elimina palabra reservada
  CONST   = 6,
  DO      = 7,
  ELSE    = 8,
  ELSIF   = 9, //nueva palabra reservada
  END     = 10,
  EXIT    = 11, //nueva palabra reservada
  FUNC    = 12,
  IF      = 13,
  IN      = 14,
  LET     = 15,
  NEXT    = 16 ,
  NIL     = 17 , //nueva palabra reservada
  OF      = 18 ,
  PRIVATE = 19, //nueva palabra reservada
  PROC    = 20,
  REC     = 21, //nueva palabra reservada
  RECORD  = 22,
  REPEAT  = 23, //nueva palabra reservada
  RETURN  = 24, //nueva palabra reservada
  THEN    = 25,
  TO      = 26, //nueva palabra reservada
  TYPE    = 27,
  UNTIL   = 28, //nueva palabra reservada
  VAR     = 29,
  WHILE =   30,
```

```
private static String[] tokenTable = new String[] {
  "<int>",
  "<char>",
  "<identifier>",
  "<operator>",
  "and", //nueva palabra reservada
  "array",
  "const",
  "do",
  "else",
  "elsif", //nueva palabra reservada
  "end",
  "exit", //nueva palabra reservada
  "func",
  "if",
  "in",
  "let",
  "nil", //nueva palabra reservada
  "of",
  "private", //nueva palabra reservada
  "proc",
  "rec", //nueva palabra reservada
  "record",
  "repeat", //nueva palabra reservada
  "return", //nueva palabra reservada
  "then",
  "to", //nueva palabra reservada
  "type",
  "until", //nueva palabra reservada
  "var",
  "while",
  ".",
  ":",
  ";",
  ",",
```

Explicación del esquema para generar la versión HTML

Para el esquema de la creación del archivo de formato HTML se utiliza un formato basado en la utilización de la lectura del archivo que se carga o crea a la hora de querer realizar una compilación, mediante un método en la clase HtmlWriter el cual se encarga de generar la primera parte del html la cual contiene el tipo de documento y el header seguidamente para el body se llama una función que lee el archivo y va guardando en formato html en un string para luego incorporarlo al archivo de salida que se está escribiendo.

```java
public class HtmlWriter {
    private String fileName;

    public HtmlWriter(String fileName) {
        this.fileName = fileName.replace(".tri", ".html");
    }

    // Draw the AST representing a complete program.
    //public void write(Program ast) {
    public void write(String data) {
        // Prepare the file to write
        try {
            FileWriter fileWriter = new FileWriter(fileName);

            //HTML header
            fileWriter.write("<html><body>"+data+"</body></html>");

            //WriterVisitor layout = new WriterVisitor(fileWriter);
            //ast.visit(layout, null);

            fileWriter.close();

        } catch (IOException e) {
            System.err.println("Error while creating file for print the AST");
            e.printStackTrace();
        }
    }
}
```

Además de esta clase, se hicieron un par de extensiones en la clase scanner.java que permiten la generación del HTML de manera correcta.

```
public String getHtmlBuffer() {
  return htmlBuffer;
}
private void addHtmlReservedWord() {
  htmlBuffer+="<FONT FACE= \"monospace\" SIZE =3 COLOR=#000000><strong>"+htmlSpaces+currentSpelling.toString()+"</stron
  htmlSpaces = "";
  counter++;

}
private void addHtmlIdentifier() {
  String token = currentSpelling.toString();
  if(Token.isReservedWord(token)) {
    addHtmlReservedWord();
  }else {
    htmlBuffer+="<FONT FACE= \"monospace\" SIZE =3 COLOR=#000000>"+htmlSpaces+token+"</FONT>";
    htmlSpaces = "";
    counter++;
  }
}

//agrega texto con formato para las literales
private void addHtmlLiteral() {
  htmlBuffer+="<FONT FACE= \"monospace\" SIZE =3 COLOR=#000099>"+htmlSpaces+currentSpelling.toString()+"</FONT>";
  htmlSpaces = "";
  counter++;
}
private void addToHtmlComment(String newChar) {
  htmlBufferComment+=newChar;
}
//agrega texto con formato para las comentarios
private void addHtmlComment() {
  htmlBuffer+="<FONT FACE= \"monospace\" SIZE =3 COLOR=#008000>"+htmlSpaces+htmlBufferComment+"</FONT>";
  htmlBufferComment = "";
  htmlSpaces = "";
  counter++;
}

//agrega los espacios que se van acumulando cuando se lee
private void addHtmlSpace() {
  htmlBuffer+="<FONT FACE= \"monospace\" SIZE =3 COLOR=#000000>"+htmlSpaces+currentSpelling.toString()+"</FONT>";
  htmlSpaces = "";
```

Nuevas rutinas de reconocimiento sintáctico

La nuevas rutinas agregadas al analizador sintáctico son las siguientes:

Nil

Esta rutina únicamente recibe un comando vacío no genera nada en el árbol sintáctico.

```
//Agregando el caso de Nil
case Token.NIL: {
  acceptIt();
  finish(commandPos);
  commandAST = new EmptyCommand(commandPos);
  break;

}
```

Let Declaration in Command end

Esta rutina recibe la nueva palabra reservada let, seguidamente una declaración, seguidamente la palabra reservada in, un comando y finalmente la palabra reservada end.

```
// Agregando caso de LET IN END
 case Token.LET: {
  acceptIt();
  Declaration dAST = parseDeclaration();
  accept(Token.IN);
  Command cAST = parseCommand();
  accept(Token.END);
  finish(commandPos);
  commandAST = new LetCommand(dAST, cAST, commandPos);
```

If Expression then Command (elsif Expression then Command)* else Command end

Esta rutina recibe el token if, una expresión, una palabra reservada then, un comando, un paréntesis de apertura, una palabra reservada elsif, una expresión, una palabra reservada then, un comando, un paréntesis de cierre. Estas se pueden iterar cero o más veces. Seguidamente recibe una palabra reservada else, un comando y la palabra reservada end.

```
// Modificando caso IF
case Token.IF:{
  acceptIt();
  Expression eAST = parseExpression();
  accept(Token.THEN);
  Command cAST = parseCommand();
  Command c2AST = parseRestOfIf();
  finish(commandPos);
  commandAST = new IfCommand(eAST, cAST, c2AST, commandPos);
  break;


}
```

```
// Parse Rest of IF
Command parseRestOfIf() throws SyntaxError {
  Command commandAST = null; // in case there's a syntactic error
  SourcePosition commandPos = new SourcePosition();
  start(commandPos);
  switch (currentToken.kind) {

    case Token.ELSE: {
      acceptIt();
      Command cAST = parseCommand();
      accept(Token.END);
      commandAST = cAST;
    }
      break;

    case Token.ELSIF: {
      acceptIt();
      Expression eAST = parseExpression();
      accept(Token.THEN);
      Command cAST = parseCommand();
      Command c2AST = parseRestOfIf();
      finish(commandPos);
      commandAST = new IfCommand(eAST, cAST, c2AST, commandPos);
    }
      break;

    default:
      syntacticError("\"%\" cannot start a command", currentToken.spelling);
      break;
  }
  // default
  return commandAST;
}
```

Repeat while Expression do Command end

Este comando recibe la palabra reservada repeat, la palabra reservada while, una expresión, una palabra reservada do, un comando y la palabra reservada end.

```
// Agregando caso Repeat
case Token.REPEAT: {
  acceptIt();

  switch (currentToken.kind) {

    //WHILE DO END
    case Token.WHILE: {
      acceptIt();
      Expression eAST = parseExpression();
      accept(Token.DO);
      Command cAST = parseCommand();
      accept(Token.END);
      finish(commandPos);
      commandAST = new RepeatWhileCommand(eAST, cAST, commandPos);
      break;
    }
```

Repeat until Expression do Command end

Este comando recibe la palabra reservada repeat, la palabra reservada until, una expresión, una palabra reservada do, un comando y la palabra reservada end.

```
  // UNTIL DO END
  case Token.UNTIL: {
    acceptIt();
    Expression eAST = parseExpression();
    accept(Token.DO);
    Command cAST = parseCommand();
    accept(Token.END);
    finish(commandPos);
    commandAST = new RepeatUntilCommand(eAST, cAST, commandPos);
  }
  break;
```

Repeat do Command while Expression end

Este comando recibe la palabra reservada repeat, la palabra reservada do, un comando, la palabra reservada while, una expresión y la palabra reservada end.

```
//Agredando caso Repeat DO
case Token.DO: {
  acceptIt();
  Command cAST = parseCommand();
  switch (currentToken.kind) {

    //DO WHILE END
    case Token.WHILE: {
      acceptIt();
      Expression eAST = parseExpression();
      accept(Token.END);
      finish(commandPos);
      commandAST = new RepeatDoWhileCommand( cAST, eAST, commandPos);
      break;
    }
```

Repeat do Command until Expression end

Este comando recibe la palabra reservada repeat, la palabra reservada do, un comando, la palabra reservada until, una expresión y la palabra reservada end.

```
//DO  UNTIL END
case Token.UNTIL: {
  acceptIt();
  Expression eAST = parseExpression();
  accept(Token.END);
  finish(commandPos);
  commandAST = new RepeatDoUntilCommand( cAST, eAST, commandPos);
  break;
}
```

Repeat var Identifier in Expression to Expression do Command end

Este comando recibe la palabra reservada repeat, la palabra reservada var, un identificador, la palabra reservada in, una expresión y la palabra reservada to, una expresión, la palabra reservada do, un comando y la palabra reservada end.

```
// Agredando caso VAR
case Token.VAR: {
  acceptIt();
  Identifier iAST = parseIdentifier();
  accept(Token.IN);
  Expression e1AST = parseExpression();
  accept(Token.TO);
  Expression e2AST = parseExpression();
  accept(Token.DO);
  Command cAST = parseCommand();
  accept(Token.END);
  Declaration cVarDeclaration = new InExVarDeclaration(iAST,e1AST,commandPos);
  finish(commandPos);
  commandAST = new RepeatVarCommand( cVarDeclaration, e2AST,cAST, commandPos);
  break;
}
```

Return

Esta rutina únicamente recibe un comando vacío no genera nada en el árbol sintáctico.

```
case Token.RETURN: {
  acceptIt();
  finish(commandPos);
  commandAST = new EmptyCommand(commandPos);
  break;
}
```

Compound Declaration

Se añade la declaración compound- declaration como una modificación al declaration existente. Dentro de esta declaración se crea la rutina para aceptar la palabra reservada rec, que instancia una declaración nueva llamada proc func y finalmente recibe la palabra reservada end. Además, se crea la rutina para reconocer la palabra reservada private, la cual recibe una declaración, la palabra reservada in, una declaración y finalmente la palabra reservada end.

```
//Agregando declaracion compuesta
Declaration parseCompoundDeclaration() throws SyntaxError {
  Declaration declarationAST = null; // in case there's a syntactic error
  SourcePosition declarationPos = new SourcePosition();
  start(declarationPos);

  switch (currentToken.kind) {

    case Token.REC: {
      acceptIt();
      declarationAST = parseProcFuncs();
      accept(Token.END);
      finish(declarationPos);
      break;
    }
    case Token.PRIVATE: {
      acceptIt();
      Declaration d1AST = parseDeclaration();
      accept(Token.IN);
      Declaration d2AST = parseDeclaration();
      accept(Token.END);
      finish(declarationPos);
      declarationAST = new SequentialDeclaration(d1AST,d2AST,declarationPos);
      break;
    }
    default:
      declarationAST = parseSingleDeclaration();

  }
  return declarationAST; // revisar return
}
```

Proc Identifier (Formal-Parameter-Sequence) ~ Command end

Dentro de la declaración proc-func se añade la rutina para aceptar el token proc, seguidamente un identificador, un paréntesis de apertura, una secuencia de parámetros, un paréntesis de cierre, un carácter reservado is, un comando y la palabra reservada end.

```
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;
  }
```

Func Identifier (Formal-Parameter-Sequence) : Type denoter ~ Expression

Dentro de la declaración proc-func se añade la rutina para aceptar el token func, seguidamente un identificador, un paréntesis de apertura, una secuencia de parámetros, un paréntesis de cierre, un carácter reservado colon, un comando y la palabra reservada end.

```
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;
}
```

Proc Func (end Proc Func) +

Dentro de la declaración proc-funcs se añade la regla que reconoce la declaración proc-func, seguidamente un paréntesis de apertura, una palabra reservada end, la declaración proc- func, un paréntesis de cierre. Esta se puede iterar una o más veces.

```
do{

  Declaration p1AST = parseProcFunc();
   if(currentToken.kind == Token.AND){
     acceptIt();
     Declaration p2AST = parseProcFunc();
     finish(declarationPos);
     declarationAST = new SequentialDeclaration (p1AST, p2AST, declarationPos );
     return declarationAST;
   }
   else{
      syntacticError("\"%\" cannot start a declaration", currentToken.spelling);
      break;
   }
}
while(currentToken.kind == Token.AND);

return declarationAST;
}
```

Var Identifier := Expression

A single declaration se le añade una modificación a la rutina de var para que reconozca un identificador, la palabra reservada become y una expresión.

```
case Token.VAR: {
  acceptIt();
  Identifier iAST = parseIdentifier();
   switch (currentToken.kind) {
     case Token.COLON: {
        acceptIt();
        TypeDenoter tAST = parseTypeDenoter();
        finish(declarationPos);
        declarationAST = new VarDeclaration(iAST, tAST, declarationPos);
     }
     break;
     case Token.BECOMES: {
        acceptIt();
        Expression eAST = parseExpression();
        finish(declarationPos);
        declarationAST= new VarExpresionDeclaration(iAST, eAST, declarationPos);
    }
   }
}
```

Lista de errores sintácticos detectados

Para esta primera fase no se detectaron errores sintácticos.

Modelaje realizado para los árboles sintácticos

Para la generación de árboles sintácticos se tuvieron que realizar modificaciones para la creación de las nuevas rutinas. Primero se crearon las clases necesarias para cada una de las rutinas nuevas, seguidamente se importan dentro de la clase parser.java. Las clases que además fueron afectadas son las siguientes:

Visitor.java

Dentro de esta clase únicamente se definen los objetos nuevos, que en este caso serian las clases nuevas de comandos que se crearon.

```
public interface Visitor {

  // Commands
  public abstract Object visitAssignCommand(AssignCommand ast, Object o);
  public abstract Object visitCallCommand(CallCommand ast, Object o);
  public abstract Object visitEmptyCommand(EmptyCommand ast, Object o);
  public abstract Object visitIfCommand(IfCommand ast, Object o);
  public abstract Object visitLetCommand(LetCommand ast, Object o);
  public abstract Object visitSequentialCommand(SequentialCommand ast, Object o);
  public abstract Object visitWhileCommand(WhileCommand ast, Object o);
  public abstract Object visitRepeatWhileCommand (RepeatWhileCommand ast, Object o);
  public abstract Object visitRepeatUntilCommand (RepeatUntilCommand ast, Object o);
  public abstract Object visitRepeatDoWhileCommand (RepeatDoWhileCommand ast, Object o);
  public abstract Object visitRepeatDoUntilCommand (RepeatDoUntilCommand ast, Object o);
  public abstract Object visitRepeatVarCommand (RepeatVarCommand ast, Object o);
  public abstract Object visitRestOfIfElseCommand (RestOfIfElseCommand ast, Object o);
  public abstract Object visitRestOfIfElsifCommand (RestOfIfElsifCommand ast, Object o);
  public abstract Object visitNextCommand (NextCommand ast, Object o);
  public abstract Object visitLoopIdentifierCommand (LoopIdentifierCommand ast, Object o);
```

TableVisitor.java

En esta clase se implementan las interfaces previamente definidas en el visitor.java que se utilizan para revisar el AST y generar el modelo default de la tabla.

```
public Object visitRepeatWhileCommand(RepeatWhileCommand ast, Object o) {
    ast.E.visit(this, null);
    ast.C.visit(this, null);

    return (null);
}

public Object visitRepeatUntilCommand(RepeatUntilCommand ast, Object o) {
    ast.E.visit(this, null);
    ast.C.visit(this, null);

    return (null);
}

public Object visitRepeatDoWhileCommand(RepeatDoWhileCommand ast, Object o) {
    ast.C.visit(this, null);
    ast.E.visit(this, null);

    return (null);
}

public Object visitRepeatDoUntilCommand(RepeatDoUntilCommand ast, Object o) {
    ast.C.visit(this, null);
    ast.E.visit(this, null);

    return (null);
}

public Object visitRepeatVarCommand(RepeatVarCommand ast, Object o) {
    ast.D.visit(this, null);
    ast.E1.visit(this, null);
    ast.C.visit(this, null);

    return (null);
}

public Object visitRestOfIfElseCommand(RestOfIfElseCommand ast, Object o) {
    ast.C.visit(this, null);
    return (null);
}

public Object visitRestOfIfElsifCommand(RestOfIfElsifCommand ast, Object o) {
    ast.E.visit(this, null);
    ast.C.visit(this, null);
    ast.C2.visit(this, null);
```

TreeVisitor.java

Esta clase implementa las interfaces definidas en visitor,java para generar los nodos al árbol sintáctico.

```
import Triangle.AbstractSyntaxTrees.RepeatUntilCommand;
import Triangle.AbstractSyntaxTrees.RepeatWhileCommand;
import Triangle.AbstractSyntaxTrees.RepeatDoWhileCommand;
import Triangle.AbstractSyntaxTrees.RepeatDoUntilCommand;
import Triangle.AbstractSyntaxTrees.RepeatVarCommand;
import Triangle.AbstractSyntaxTrees.RestOfIfElseCommand;
import Triangle.AbstractSyntaxTrees.RestOfIfElsifCommand;
import Triangle.AbstractSyntaxTrees.NextCommand;
import Triangle.AbstractSyntaxTrees.LoopIdentifierCommand;
import Triangle.AbstractSyntaxTrees.CompoundDeclaration;
import Triangle.AbstractSyntaxTrees.InExVarDeclaration;
import Triangle.AbstractSyntaxTrees.VarExpresionDeclaration;
// import Triangle.AbstractSyntaxTrees.
//import Triangle.AbstractSyntaxTrees.RepeatLoopCommand;
```

```
public Object visitRepeatWhileCommand(RepeatWhileCommand ast, Object obj) {
    return (createBinary("Repeat While Do Command", ast.E, ast.C));
}

public Object visitRepeatUntilCommand(RepeatUntilCommand ast, Object obj) {
    return (createBinary("Repeat Until Do Command", ast.C, ast.E));
}
public Object visitRepeatDoWhileCommand(RepeatDoWhileCommand ast, Object obj) {
    return (createBinary("Repeat Do While Command", ast.C, ast.E));
}

public Object visitRepeatDoUntilCommand(RepeatDoUntilCommand ast, Object obj) {
    return (createBinary("Repeat Do Until Command", ast.E, ast.C));
}
public Object visitRepeatVarCommand(RepeatVarCommand ast, Object obj) {
    return (createTernary("Repeat Var", ast.D, ast.E1,ast.C));
}

public Object visitRestOfIfElseCommand(RestOfIfElseCommand ast, Object obj) {
    return (createUnary("Rest of If Else",ast.C));
}
public Object visitRestOfIfElsifCommand(RestOfIfElsifCommand ast, Object obj) {
    return (createTernary("Rest of If Elsif",ast.E, ast.C,ast.C2));
}
 public Object visitNextCommand(NextCommand ast, Object obj) {
    return (createUnary("Next Command",ast.I));
}
public Object visitLoopIdentifierCommand(LoopIdentifierCommand ast, Object obj) {
    return (createUnary("Loop Identiifer Command",ast.I));
}
```

WriterVisitor.java

Esta clase implementa las interfaces declaradas en visitor para escribirlas dentro de la función FileWriter.

```
// Implementar nuevos visitors

public Object visitRepeatWhileCommand(RepeatWhileCommand ast, Object o) {
    writeLineHTML("<RepeatWhileCommand>");
    ast.E.visit(this,null);
    ast.C.visit(this,null);
    writeLineHTML("</RepeatWhileCommand>");
    return null;
}

public Object visitRepeatUntilCommand(RepeatUntilCommand ast, Object o) {
    writeLineHTML("<RepeatUntilCommand>");
    ast.E.visit(this,null);
    ast.C.visit(this,null);
    writeLineHTML("</RepeatUntilCommand>");
    return null;
}

public Object visitRepeatDoWhileCommand(RepeatDoWhileCommand ast, Object o) {
    writeLineHTML("<RepeatDoWhileCommand>");
    ast.C.visit(this,null);
    ast.E.visit(this,null);
    writeLineHTML("</RepeatDoWhileCommand>");
    return null;
}

public Object visitRepeatDoUntilCommand(RepeatDoUntilCommand ast, Object o) {
    writeLineHTML("<RepeatDoUntilCommand>");
    ast.C.visit(this,null);
    ast.E.visit(this,null);
    writeLineHTML("</RepeatDoUntilCommand>");
    return null;
}

public Object visitRepeatVarCommand(RepeatVarCommand ast, Object o) {
    writeLineHTML("<RepeatVarCommand>");
    ast.D.visit(this,null);
    ast.E1.visit(this,null);
    ast.C.visit(this,null);
    writeLineHTML("</RepeatVarCommand>");
    return null;
}
```

LayoutVisitor.java

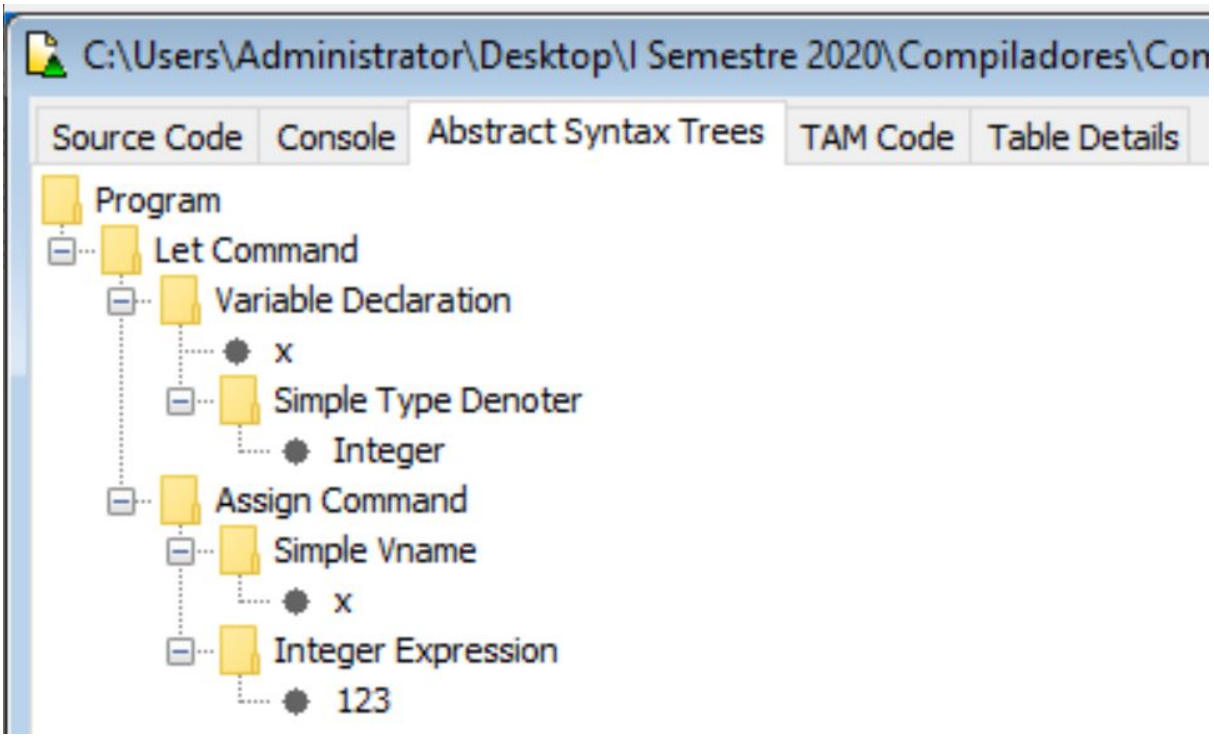
Esta clase implementa la clase visitor y permite formar el diseño para los objetos creados.

```
import Triangle.AbstractSyntaxTrees.RepeatUntilCommand;
import Triangle.AbstractSyntaxTrees.RepeatWhileCommand;
import Triangle.AbstractSyntaxTrees.RepeatDoUntilCommand;
import Triangle.AbstractSyntaxTrees.RepeatDoWhileCommand;
import Triangle.AbstractSyntaxTrees.RepeatVarCommand;
import Triangle.AbstractSyntaxTrees.RepeatLoopCommand;
import Triangle.AbstractSyntaxTrees.LoopIdentifierCommand;
import Triangle.AbstractSyntaxTrees.NextCommand;
import Triangle.AbstractSyntaxTrees.RestOfIfElseCommand;
import Triangle.AbstractSyntaxTrees.CompoundDeclaration;
import Triangle.AbstractSyntaxTrees.VarExpresionDeclaration;
import Triangle.AbstractSyntaxTrees.RestOfIfElsifCommand;
```

```
public Object visitRepeatWhileCommand(RepeatWhileCommand ast, Object obj) {
  return layoutBinary("RepeatWhileCom.", ast.E, ast.C);
}

public Object visitRepeatUntilCommand(RepeatUntilCommand ast, Object obj) {
  return layoutBinary("RepeatUntilCom.", ast.E, ast.C);
}

public Object visitRepeatDoWhileCommand(RepeatDoWhileCommand ast, Object obj) {
  return layoutBinary("RepeatDoWhileCom.", ast.C, ast.E);
}

public Object visitRepeatDoUntilCommand(RepeatDoUntilCommand ast, Object obj) {
  return layoutBinary("RepeatDoUntilCom.", ast.C, ast.E);
}

public Object visitRepeatVarCommand(RepeatVarCommand ast, Object obj) {
  return layoutTernary("RepeatVarCom.", ast.D, ast.E1, ast.C);
}

public Object visitLoopIdentifierCommand(LoopIdentifierCommand ast, Object obj) {
  return layoutBinary("LoopIdentifierCommand.", ast.I, ast.C);
}

public Object visitNextCommand(NextCommand ast, Object obj) {
  return layoutUnary("NextCommand.", ast.I);
}
```

Extensión para visualización de árboles

El IDE del compilador de Triángulo cuenta con una pestaña para poder visualizar el AST generado después de la compilación del código que se ingresa, cabe mencionar que si se encuentra un error en la sintaxis el árbol no se generará. Para visualizar el Árbol de Sintaxis Abstracto se debe ingresar a la pestaña que se encuentra en el IDE llamada “Abstract Syntax Tree”, ubicada en la parte superior.



Extensión para visualización de árboles en XML.

Para la visualización del Árbol de Sintaxis Abstracto en un archivo con formato xml, fue implementado mediante una clase que hace uso del patrón de diseño Visitor, implementado en el compilador, esto para recorrer el árbol y mediante la clase “Writer.java” escribe en el archivo de salida, para luego guardarlo con formato xml.

Un ejemplo de métodos utilizados para la creación del archivo XML es el siguiente

```
public Object visitProcDeclaration(ProcDeclaration ast, Object obj) {
    writeLineHTML("<ProcDeclaration>");
    ast.I.visit(this, null);
    ast.FPS.visit(this, null);
    ast.C.visit(this, null);
    writeLineHTML("</ProcDeclaration>");
    return null;
}
```

Se puede observar que mediante la visita de los diferentes comandos, se utiliza el método writeLineHTML para escribir en el archivo de salida las etiquetas según lo que lee en el árbol.

Plan de pruebas

Se hacen las siguientes pruebas para evidenciar el funcionamiento del compilador de Triángulo extendido. Se incorpora en los archivos adjuntos los casos de prueba utilizados.

Casos de pruebas	Objetivo	Diseño	Resultado esperado	Resultado obtenido
Verificar que “nil” funciona correctamente	Evidenciar que se agrega el Token “nil” a la gramática de Triángulo extendido	Nill.tri	Que el compilador reconozca el token “nil” como parte de la gramática del lenguaje	No hay error en la prueba realizada mediante la utilización del Token “nil”
Verificar que “let in end” funciona correctamente	Verificar el correcto funcionamiento del comando “let in end”	LetInEnd.tri	Que el comando ejecutado muestre un error al tener una declaración vacía.	Resultado exitoso ya que muestra el error de que el token “in” no puede empezar una declaración
Verificar que “if then else end” funciona correctamente	Verificar el correcto funcionamiento del comando “if then else end”	IfThenElseEnd.tri	Que se ejecute correctamente el comando “if then else end” en el compilador	La ejecución de la prueba fue exitosa ya que no presenta ningún error sintáctico en la compilación.
Verificar que“Repeat While do end”	Verificar el correcto funcionamiento	RepeatWhileDoEnd.tri	Que se ejecute correctamente el comando	La ejecución de la prueba realizada ha

funciona correctamente	del comando “Repeat while do end”		“Repeat while do end” en el compilador	sido exitosamente finalizada ya que no surgieron errores en la compilación.
Verificar que “Repeat until do end” funciona correctamente	Verificar el correcto funcionamiento del comando “Repeat until do end”	RepeatUntilDoEnd.tri	Que se ejecute correctamente el comando “Repeat until do end” en el compilador	La ejecución de la prueba realizada ha sido exitosamente finalizada ya que no surgieron errores en la compilación
Verificar que “Repeat do while end” funciona correctamente	Verificar el correcto funcionamiento del comando “Repeat do while end”	RepeatDoWhileErr.tri	Que aparezca un error por la falta del token “do” en el archivo .tri	Aparece el error como se esperaba en el resultado esperado ya que falta el token “do” en el comando.
Verificar que “Repeat do until end” funciona correctamente	Verificar el correcto funcionamiento del comando “Repeat do until end”	RepeatDoUntilEnd.tri	Que se ejecute correctamente el comando “Repeat do until end” en el compilador	La ejecución de la prueba realizada ha sido exitosamente finalizada ya que no surgieron de sintaxis errores en la compilación.
Verificar que “Repeat var in to do end” funciona correctamente	Verificar que el correcto funcionamiento “Repeat var in to do end”	Repeat var in to do end.tri	Que se ejecute correctamente el comando “Repeat var in to do end” en el compilador	Ejecución de la prueba realizada ha sido exitosA finalizada, al no haber errores de sintaxis en la compilación.

Pruebas de Declaraciones

Casos de pruebas	Objetivo	Diseño	Resultado esperado	Resultado obtenido
Probar que la compilación de let const declaration sea exitosa.	Verificar el funcionamiento correcto de la declaración	LetConstDeclaration.tri	Exitosa generación del AST y el xml	La ejecución de la prueba realizada ha sido exitosamente finalizada ya que no surgieron errores en la compilación
Probar que la compilación de let func declaration sea exitosa.	Verificar el funcionamiento correcto de la declaración	LetFuncDeclaration.tri	Exitosa generación del AST y el xml	La ejecución de la prueba realizada ha sido exitosamente finalizada ya que no surgieron errores en la compilación
Probar que la compilación de let proc declaration sea exitosa	Verificar el funcionamiento correcto de la declaración	LetProcDeclaration.tri	Exitosa generación del AST y el xml	La ejecución de la prueba realizada ha sido exitosamente finalizada ya que no surgieron errores en la compilación
Probar que la compilación de let var declaration sea exitosa.	Verificar el funcionamiento correcto de la declaración	letVarDeclaration.tri	Exitosa generación del AST y el xml	La ejecución de la prueba realizada ha sido exitosamente finalizada ya que no surgieron errores en la compilación
Probar que la compilación de rec several proc declaration sea	Verificar el funcionamiento correcto de la declaración	letRecProcDeclaration.tri	Exitosa generación del AST y el xml	La ejecución de la prueba realizada ha sido

exitosa.				exitosamente finalizada ya que no surgieron errores en la compilación

Análisis de resultados del plan de pruebas

Realizando un análisis en detalle, se puede determinar que el proyecto funciona en su totalidad , o por lo menos, se puede decir que los objetivos se han cumplido correctamente, esto se puede afirmar debido a que se hicieron las pruebas correspondientes, listadas en la tabla anterior, que logran exponer el correcto funcionamiento del compilador.

Se cumplió el objetivo general de extender el compilador ya creado por Watt y Brown, a un compilador que es capaz de procesar un lenguaje extendido, el cual fue descrito en la especificación de este proyecto.

Las pruebas fueron, una a una, dando el resultado esperado, y por lo tanto cumpliendo los objetivos. La sintaxis tuvo modificaciones en las reglas, las cuales fueron aplicadas correctamente. Se modificaron las reglas de los comandos y las reglas de las declaraciones, todas correctamente. Los cambios léxicos también fueron realizados correctamente, tales como agregar o eliminar palabras reservadas.

La creación del árbol de sintaxis abstracta fue desarrollada correctamente y fue, casi del mismo modo exitosa y la generación de documento html.

Discusión y análisis de los resultados obtenidos

Al estudiar, analizar y comprender la estructura y comportamiento del compilador de triángulo, las modificaciones pedidas para la extensión de este compilador se concretaron exitosamente, la modificación de los distintos tokens, métodos, comandos y declaraciones del analizador léxico y sintáctico se desarrollaron de forma completa. Las pruebas realizadas arrojaron los resultados esperados.

El compilador tiene una estructura que lo hace robusto e inclusive complejo. Después de estudiar a fondo los distintos casos donde este podía encontrar un error, y con esto que pudiera crear la estructura correcta de manejo de excepciones, la compilación de los archivos .tri funcionaron de manera correcta inclusive el comportamiento que tuvo con pruebas negativas fue positivo, ya que cada estructura está organizada en el código para que cumpla con su respectiva tarea.

Reflexión sobre la experiencia

Con este proyecto se puede llegar a entender en su totalidad el trabajo del analizador sintáctico en un compilador. Además de entender cómo es posible generar un compilador desde un lenguaje como Java, aprovechando todas sus características.

Al tener que modificar un código, en vez de realizarlo desde cero, se ponen a prueba otras características diferentes, tales como el análisis profundo del código existente, para poder comprender qué fue lo que los desarrolladores anteriores hicieron, para a partir de eso cambiarlo o extender el código de una manera diferente. Es una experiencia diferente pero que aporta mucho para poder trabajar con códigos que no son de uno.

Descripción de las tareas por miembro del equipo

Melanie Bermúdez:

- Modificación del parser.java. Agregando los nuevos comandos y modificando partes del código
- Modificación del token.java. Agregando nuevas palabras reservadas
- Creación de las estructuras para los AST
- Documentación

Alejandro Tapia:

- Modificación del parser.java. Agregando los nuevos comandos y modificando partes del código
- Modificación del token.java. Agregando nuevas palabras reservadas
- Creación de las estructuras para los AST
- Documentación

Carlos Vega:

- Modificación del parser.java. Agregando los nuevos comandos y modificando partes del código
- Creación de las estructuras para los AST
- Creación del HTML y del XML
- Documentación

Manual de Usuario

Para correr la aplicación solo se debe abrir el archivo jar que se encuentra en la carpeta llamado “IDE-TRIANGLE.JAR”.

Una vez ahí, se deberá abrir un archivo nuevo con extensión .tri o crear uno nuevo. Cuando ya esté el archivo, se deberá de compilar y este indicará en la sección de consola el resultado de la compilación.

Si la compilación es correcta en la sección de Abstract Syntax Tree se podrá observar el árbol de sintaxis abstracta, además que en la carpeta de “Casos de prueba” o donde esté guardado el archivo, se generan los archivos .html y .xml que contienen la estructura del árbol sintáctico del código.

En algunas computadoras el IDE no se logra observar bien, por lo que recomendamos abrir el proyecto y correrlo en NetBeans. El proyecto se encuentra en la carpeta “ide-triangle-v1.1.src”. De esa manera el IDE se verá de manera correcta.