Documentación del Proyecto 1a

Análizador Sintáctico y Léxico

Modificación del Compilador de Mini Triangle de Watt y Brown

Versión 1.0

Preparado por:

- **Joel Barrantes Garro**
  2013120962
- **Edisson López Díaz**
  2013103311
- **Katerine Molina Sanchez**
  2013109225

Instituto Tecnológico de Costa Rica

23 de octubre de 2017

Índice

1. Esquema para el manejo del texto fuente

No se hicieron cambios a la clase **Sourcefile.java** (Clase encargada de leer el archivo fuente).

2. Modificaciones al analizador léxico

La modificación al analizador léxico requirió de la adición de nuevas palabras reservadas a la tabla de variables reservadas de la clase **Token.java**, así como la modificación de las enumeraciones de la clase. El apartado "3. Modificaciones a los tokens y otras estructuras" contiene una lista con los cambios hechos a esta clase. No se requirió la implementación de nuevos algoritmos ni la creación de nuevos métodos/clases. El scanner tampoco requirió de cambios.

3. Modificaciones a los tokens y otras estructuras

Las modificaciones al analizador léxico se listan a continuación:

1. Cambios a la clase **Token.java**, en los atributos **public static final int** :

    1.1. Se elimina la enumeración:

    - BEGIN = 5

    1.2. Se cambió la numeración de las siguientes enumeraciones, con el fin de acomodar alfabéticamente los token a insertar:

    - ARRAY = 5
    - FUNC = 11
    - IF = 12
    - IN = 13
    - LET = 14
    - OF = 16
    - PROC = 18
    - RECORD = 19
    - THEN = 23
    - TYPE = 25
    - VAR = 27
    - WHILE = 28
    - DOT = 29
    - COLON = 30
    - SEMICOLON = 31
    - COMMA = 32

- BECOMES = 33
- IS = 34
- LPAREN = 35
- RPAREN = 36
- LBRACKET = 37
- RBRACKET = 38
- LCURLY = 39
- RCURLY = 40
- EOT = 41
- ERROR = 42

1.3. Se agregaron las enumeraciones:

- AND = 4
- FOR = 10
- LOCAL = 15
- PAR = 17
- RECURSIVE = 20
- REPEAT = 21
- SKIP = 22
- TO = 24
- UNTIL = 26

2. Cambios a la clase **Token.java**, en el atributo **String[] tokenTable** (Tabla de palabras reservadas):

2.1. Se eliminó el string:

- “begin”

2.2. Se agregaron en orden alfabético los siguientes strings:

- “and”
- “for”
- “local”
- “par”
- “recursive”
- “repeat”
- “skip”
- “to”
- “until”

3. Cambios a la clase **Token.java**, en el atributo **int firstReservedWord**:

    3.1. Se cambia el valor del atributo (antes de la asignacion: **Token.ARRAY**):

        - private final static int firstReservedWord = **Token.AND**

4. No se realiza ningún cambio a la clase **Scanner.java**, que es la encargada de leer y reconocer los Token.

4. Cambios a las reglas sintácticas de △ extendido

A continuación se listan los cambios realizados a las reglas sintácticas del lenguaje:

1. Se realiza el siguiente cambio a la regla **Declaration**:

```
1  Declaration
2      ::= compound-Declaration
3      | Declaration ";" compound-Declaration
4
5  ! Se cambió por:
6
7  Declaration
8      ::= compound-Declaration (";" compound-Declaration)*
9
```

En la línea 3, se puede observar que **Declaration** posee recursión por la izquierda, para indicar que se pueden realizar múltiples **compound-Declaration,** siempre y cuando estén separadas por el símbolo "**;**". Por esta razón, se puede reexpresar esta regla usando la notación "*****", como se muestra en la línea 8.

2. Se realiza el siguiente cambio a la regla single-Command:

```
1  single-Command
2      ::= ..
3      | ..
4      | "repeat" "while" Expression "do" Command "end"
5      | "repeat" "until" Expression "do" Command "end"
6      | "repeat" "do" Command "while" Expression "end"
7      | "repeat" "do" Command "until" Expression "end"
8      | "for" "var" Identifier ":=" Expression "to" Expression "do" Command "end"
9      | "for" "var" Identifier ":=" Expression "to" Expression
10           "while" Expression "do" Command "end"
11     | "for" "var" Identifier ":=" Expression "to" Expression
12           "until" Expression "do" Command "end"
13     | ..
14
15 !Se cambió por
16     single-Command
17     ::= ..
18     | ..
19     | "repeat" (("while"|"until") Expression "do" Command | "do" Command ("while"|"until") Expression) "end"
20     | "for" "var" Identifier ":=" Expression "to" Expression ("do" Command|("while"|"until") Expression "do" Command) "end"
21     | ..
```

En esta regla, se repiten los mismos inicializadores para distintas formas sintácticas. Se hace un cambio de notación para poder traducir la regla a código java más fácilmente

3. Se realiza el siguiente cambio a la regla **compound-Declaration**:

```
compound-Declaration
    ::= ..
    | ..
    | "par" single-Declaration ("and" single-Declaration)+ "end"

!Se cambió por

compound-Declaration
    ::= ..
    | ..
    | "par" single-Declaration-Sequence "end"

!Se crea la nueva regla

single-Declaration-Sequence
    ::= single-Declaration ("and" single-Declaration)+
```

Obtenemos una nueva regla para denotar una secuencia de declaraciones simples separadas por "**and**". Usamos esta regla para reexpresar la regla **compound-Declaration**.

4. Se realiza el siguiente cambio a la regla **single-Declaration**:

```
single-Declaration
    ::= "const" Identifier "~" Expression
    | "var" Identifier ":" TypeDenoter
    | "var" Identifier ":=" Expression
    | .
    ..

!Se cambió por

single-Declaration
    ::= "const" Identifier "~" Expression
    | "var" Identifier ( ":" TypeDenoter| ":=" Expression )
    | .
    ..
```

En esta regla, se puede observar como el inicializador "**var**" es usando en dos formas sintácticas. Se hace un cambio de notación para traducir la regla a código java más fácilmente.

5. Modificaciones y/o agregaciones al reconocimiento sintáctico

A continuación, se listan los cambios realizados al paquete de análisis sintáctico.

1. Cambios a la clase **Parser.java**:

    1.1. Cambios al método **parseSingleCommand( )**:

        1.1.1. Se elimina la sentencia **Case Token.BEGIN** y su cuerpo.

        1.1.2. Se elimina la sentencia **Case Token.WHILE** y su cuerpo.

        1.1.3. Se elimina la sentencia **Case Token.SEMICOLON** y su cuerpo.

        1.1.4. Se elimina la sentencia **Case Token.END** y su cuerpo.

        1.1.5. Se elimina la sentencia **Case Token.ELSE** y su cuerpo.

        1.1.6. Se elimina la sentencia **Case Token.IN** y su cuerpo.

        1.1.7. Se elimina la sentencia **Case Token.EOT** y su cuerpo.

        1.1.8. Se modifica el cuerpo de la sentencia **Case Token.LET**, de manera que reconozca un **Command** en lugar de **SingleCommand**. Se agrega al cuerpo la sentencia **accept(Token.END)**.

        1.1.9. Se modifica la sentencia **Case Token.IF**, de manera que reconozca dos **Command** en lugar de los **SingleCommand**. Se agrega al cuerpo la sentencia **accept(Token.END)**.

        1.1.10. Se agrega la sentencia **Case Token.SKIP** y su respectivo cuerpo. En la figura 1 se muestra el código implementado:

```
449       case Token.SKIP:
450       {
451         acceptIt();
452         commandAST = new EmptyCommand(commandPos);
453       }
```

Figura 1

        1.1.11. Se agrega la sentencia **Case Token.REPEAT** y su respectivo cuerpo. Los detalles de implementación se omiten por motivo de espacio. El código correspondiente a esta sentencia se encuentra disponible en el texto fuente del compilador, en el paquete **Triangle/SyntacticAnalyzer**, en la clase

Parser.java, en la línea 317. Dentro del cuerpo de esta sentencia, se encuentra una sentencia **Switch** que procesa por separado cada forma sintáctica inicializada por el terminal **repeat**: **WhileCommand**, **UntilCommand**, **DoWhileCommand** y **DoUntilCommand**.

1.1.12. Se agrega la sentencia **Case Token.FOR** y su respectivo cuerpo. Los detalles de implementación se omiten por motivo de espacio. El código correspondiente a esta sentencia se encuentra disponible en el texto fuente del compilador, en el paquete **Triangle/SyntacticAnalyzer**, en la clase **Parser.java**, en la línea 381. que procesa por separado cada forma sintáctica inicializada por el terminal **repeat**: **ForWhileCommand** y **ForUntilCommand** y **ForDoCommand**.

1.2. Cambios al método **parseDeclaration( )**:

1.2.1. Se modificó el código de manera que acepte **compoundDeclaration** en lugar de **singleDeclaration**. En la figura 2 se muestran los cambios realizados:

```
708  Declaration parseDeclaration() throws SyntaxError {
709    Declaration declarationAST = null; // in case there's a syntactic error
710
711    SourcePosition declarationPos = new SourcePosition();
712    start(declarationPos);
713    declarationAST = parseCompoundDeclaration();
714    while (currentToken.kind == Token.SEMICOLON) {
715      acceptIt();
716      Declaration d2AST = parseCompoundDeclaration();
717      finish(declarationPos);
718      declarationAST = new SequentialDeclaration(declarationAST, d2AST,
719        declarationPos);
720    }
721    return declarationAST;
722  }
```

Figura 2

1.3. Cambios al método **parseSingleDeclaration( )**:

1.3.1. Se modifica el cuerpo de la sentencia **Case Token.VAR**, de manera que distinga las dos formas posibles de declaración de una variable: **VarDeclaration** e **InitializedVarDeclaration**, correspondientes a las distintas formas sintácticas inicializadas por el Token **VAR**, de la regla **single-Declaration**. El código correspondiente a esta sentencia se encuentra disponible en el texto fuente del compilador, en el paquete **Triangle/SyntacticAnalyzer**, en la clase **Parser.java**, en la línea 745.

1.3.2. Se modifica el cuerpo de la sentencia **Case Token.PROC**, de manera que acepte un **Command** en lugar de un **SingleCommand**. Además, se agrega al cuerpo la sentencia **accept(Token.END)**. En la figura 3 se muestran los cambios realizados.

```
777     case Token.PROC:
778       {
779         acceptIt();
780         Identifier iAST = parseIdentifier();
781         accept(Token.LPAREN);
782         FormalParameterSequence fpsAST = parseFormalParameterSequence();
783         accept(Token.RPAREN);
784         accept(Token.IS);
785         Command cAST = parseCommand(); //Replaced By Command
786         accept(Token.END); // Added new token
787         finish(declarationPos);
788         declarationAST = new ProcDeclaration(iAST, fpsAST, cAST, declarationPos);
789       }
790       break;
791
```

Figura 3

1.4. Se agrega el nuevo método **parseCompoundDeclaration( )**, encargado de reconocer la regla sintáctica **compound-Declaration** y sus formas. El código correspondiente a este método se encuentra disponible en el texto fuente del compilador, en el paquete **Triangle/SyntacticAnalyzer**, en la clase **Parser.java**, en la línea 1115. Este método sigue la estructura general de un **SingleCommand**, modificando/reemplazando las sentencias **case** por otras nuevas. Estas nuevas sentencias **case** se explican a continuación:

1.4.1. Las sentencias case **Token.VAR**, case **Token.CONST,** case **Token.PROC**, case **Token.FUNC** y case **Token.TYPE** llaman al método de reconocimiento de **SingleDeclaration**, correspondiente a **parseSingleDeclaration( )**.

1.4.2. Se agrega la sentencia case **Token.RECURSIVE** y su respectivo cuerpo, encargado de reconocer los terminales y no-terminales apropiados. En la figura 4 se muestra el código implementado:

```
1130     case Token.RECURSIVE:
1131       {
1132         acceptIt();
1133         ProcFuncS pfsAST = parseProcFuncSequence();
1134         accept(Token.END);
1135         finish(declarationPos);
1136         declarationAST = new RecursiveDeclaration(pfsAST, declarationPos);
1137       }
1138       break;
```

Figura 4

1.4.3. Se agrega la sentencia case **Token.LOCAL** y su respectivo cuerpo, encargado de reconocer los terminales y no-terminales apropiados. En la figura 5 se muestra el código implementado:

```
1139     case Token.LOCAL:
1140       {
1141         acceptIt();
1142         Declaration localdAST1 = parseDeclaration();
1143         accept(Token.IN);
1144         Declaration localdAST2 = parseDeclaration();
1145         accept(Token.END);
1146         finish(declarationPos);
1147         declarationAST = new LocalDeclaration(localdAST1, localdAST2, declarationPos);
1148       }
1149       break;
```

Figura 5

1.4.4. Se agrega la sentencia case **Token.PAR** y su respectivo cuerpo, encargado de reconocer los terminales y no-terminales apropiados. En la figura 6 se muestra el código implementado:

```
1150     case Token.PAR:
1151       {
1152         acceptIt();
1153
1154         SingleDeclarationSequence sdsAST = parseSingleDeclarationSequence();
1155
1156         accept(Token.END);
1157         finish(declarationPos);
1158         declarationAST = new ParDeclaration( sdsAST, declarationPos);
1159
1160       }
1161       break;
```

Figura 6

1.5. Se agrega el nuevo método **parseProcFuncSequence( )**, encargado de reconocer la regla sintáctica **Proc-Funcs** y sus formas. El código correspondiente a este método se encuentra disponible en el texto fuente del compilador, en el paquete **Triangle/SyntacticAnalyzer**, en la clase **Parser.java**, en la línea 1172.

1.6. Se agrega el nuevo método **parseProperProcFuncSequence( )**, como método auxiliar para procesar correctamente el método **parseProcFuncSequence( )** y obligar al programador a declarar dos o más formas de **Proc-Func** . El código correspondiente a este método se encuentra disponible en el texto fuente del compilador, en el paquete **Triangle/SyntacticAnalyzer**, en la clase **Parser.java**, en la línea 1187.

1.7. Se agrega el nuevo método **parseProcFunc( )**, encargado de reconocer la regla sintáctica **Proc-Func**, que consiste de dos formas sintácticas. El código correspondiente a este método se encuentra disponible en el texto fuente del compilador, en el paquete **Triangle/SyntacticAnalyzer**, en la clase **Parser.java**, en la línea 1209.

1.8. Se agrega el nuevo método **parseSingleDeclarationSequence( )**, encargado de reconocer una nueva regla sintáctica que sigue la estructura de la regla **Proc-Funcs** El código correspondiente a este método se encuentra disponible en el texto fuente del compilador, en el paquete **Triangle/SyntacticAnalyzer**, en la clase **Parser.java**, en la línea 1257.

1.9. Se agrega el nuevo método **parseProperSingleDeclarationSequence( )**, como método auxiliar para procesar correctamente el método **parseSingleDeclarationSequence( )** y obligar al programador a declarar dos o más formas de **SingleDeclaration**. El código correspondiente a este método se encuentra disponible en el texto fuente del compilador, en el paquete **Triangle/SyntacticAnalyzer**, en la clase **Parser.java**, en la línea 1273.

6. Errores sintácticos detectados

El compilador detecta correctamente los siguientes errores sintácticos:

1. El compilador detecta correctamente los errores sintácticos, como la omisión de un token, para las nuevas alternativas de **single-Command**, incluyendo las formas inicializadas con for y repeat. Por ejemplo, en caso de omitir una expresión, el compilador muestra un error que informa al programador sobre la falta de un inicializador para **Expression**. Se utilizan los mensajes por defecto (como el error de lectura de un Token específico) que el compilador muestra.
2. El compilador detecta el comando skip, por lo que muestra un error al dar un comando vacío, pues se eliminó esta alternativa de **single-Command.**
3. El compilador detecta correctamente errores sintácticos en las variantes de **compound-Declaration**:

3.1. Errores sintácticos en la variante **Recursive**, incluyendo la falta de dos o más formas **ProcFunc**, la falta de tokens **AND** para dividir las declaraciones de **ProcFunc** y la falta del token **END.**

3.2. Errores sintácticos generales en la variante **Local**, como la incorrecta inicialización de una declaración, o la falta del token **END**.

3.3. Errores sintácticos en la variante **Par,** incluyendo la falta de dos o más declaraciones simples **single-Declaration**, la falta de tokens **AND** para dividir las declaraciones y la falta del token **END.**

3.4. Error sintáctico al declarar una *Variable Inicializada* incorrectamente, como la falta de una expresión en la declaración.

4. El compilador detecta correctamente errores sintácticos en las variantes de **Proc-Func**, como la falta del token **END** para la terminación de una declaración de **Proc**, o la omisión del tipo retornado por una declaración de **Func.**
5. El compilador detecta correctamente errores en la modificación de la variante **ProcDeclaration**, de la regla **single-Declaration**, como la falta del token **END.**
6. El compilador detecta correctamente errores léxicos en las variantes de **Command** eliminadas: **BEGIN** y **WHILE,** pues estos token ya no inicializan formas de **Command** (El token **BEGIN** fue eliminado completamente del lenguaje)

7. Modelaje de los AST

A continuación, se explica el modelaje realizado para los nuevos AST:

1. ASTs pertenecientes a la regla **Proc-Funcs** y **Proc-Func**:

1.1. Se crea la clase abstracta **ProcFunc**, que representa la regla **Proc-Func**. Las clase **ProcProcFunc** y **FuncProcFunc** heredan de **ProcFunc**. Estas clases representan las dos formas que puede tomar la regla **ProcFunc**. Ambas clases son implementadas de la misma manera en que fueron implementadas las clases **ProcDeclaration** y **FuncDeclaration**, respectivamente.

1.2. El árbol **ProcFuncs** es una clase abstracta creada con el fin de representar la cadena de dos o más **ProcFunc**, correspondiente con la regla **Proc-Funcs.** Este árbol hereda sus características a tres subclases: **EmptyProcFuncSequence, SingleProcFuncSequence** y **MultipleProcFuncSequence**. El árbol binario **SingleProcFuncSequence** está compuesta por un solo árbol del tipo **ProcFunc**. El árbol **EmptyProcFuncSequence** es un árbol sin hijos que representa un **ProcFuncs** vacío. Por su parte, el árbol **MultipleProcFuncSequence** está compuesto por un árbol del tipo **ProcFunc** y un árbol del tipo **ProcFuncs**, de

manera que se crea una estructura que tiende a crecer hacia la derecha, de manera recursiva.

2. ASTs pertenecientes a la regla **compound-Declaration**:

    2.1. Se crea la clase **LocalDeclaration**, que representa la estructura del AST de una *Declaración Local*. Este árbol está constituído por dos declaraciones del tipo **Declaration.**

    2.2. El árbol **SingleDeclarationSequence** es una clase abstracta creada con el fin de representar la cadena de dos o más **SingleDeclaration**, la cual compone la forma sintáctica **ParDeclaration.** Este árbol hereda sus características a tres subclases: **EmptySingleDeclarationSequence, SingleSingleDeclarationSequence** y **MultipleSingleDeclarationSequence**. El árbol unario **SingleSingleDeclarationSequence** está compuesta por un solo árbol del tipo **SingleDeclaration**. El árbol **EmptyDeclarationSequence** es un árbol sin hijos que representa un **SingleDeclarationSequence** vacío. Por su parte, el árbol binario **MultipleSingleDeclarationSequence** está compuesto por un árbol del tipo **SingleDeclaration** y un árbol del tipo **SingleDeclarationSequence**, de manera que se crea una estructura que tiende a crecer hacia la derecha, de manera recursiva.

    2.3. Se crea la clase **ParDeclaration**, que representa la estructura del AST de las *Declaraciones Paralelas.* Este árbol está constituído por un árbol sintáctico del tipo **SingleDeclarationSequence** (el analizador sintáctico se encarga de construir este árbol de manera que esté compuesto por dos o más **SingleDeclaration**).

    2.4. Se crea la clase **RecursiveDeclaration**, que representa el árbol de las *Declaraciones Recursivas*. Es un árbol unario que tiene como hijo un árbol del tipo **ProcFuncs** (el analizador sintáctico se encarga de construir este árbol de manera que esté compuesto por dos o más **ProcFunc**).

3. ASTs pertenecientes a la regla **Single-Declaration**:

    3.1. Se crea la clase **InitializedVarDeclaration**, que representa el árbol sintáctico de una *Variable Inicializada*. Este árbol sigue la estructura de la declaración de una *constante*, excepto por el uso del terminal **var** como inicializador.

4. ASTs pertenecientes a la regla **Command:**

4.1. Se crea la clase **UntilCommand**, que representa la estructura del AST de un *Ciclo Until-do*. Este árbol está constituído por un árbol del tipo **Expression** y un árbol del tipo **Command**.

4.2. Se crea la clase **DoWhileCommand**, que representa la estructura del AST de un *Ciclo do-While*. Este árbol está constituído por un árbol del tipo **Expression** y un árbol del tipo **Command**.

4.3. Se crea la clase **DoUntilCommand**, que representa la estructura del AST de un *Ciclo do-Until*. Este árbol está constituído por un árbol del tipo **Expression** y un árbol del tipo **Command**.

4.4. Se crea la clase **ForDoCommand**, que representa la estructura del AST de un *Ciclo For-do*. Este árbol está constituído por dos árboles del tipo **Expression** y un árbol del tipo **Command**.

4.5. Se crea la clase **ForUntilCommand**, que representa la estructura del AST de un *Ciclo For-Until-do*. Este árbol está constituído por tres árboles del tipo **Expression** y un árbol del tipo **Command**.

4.6. Se crea la clase **ForWhileCommand**, que representa la estructura del AST de un *Ciclo For-While-do*. Este árbol está constituído por tres árboles del tipo **Expression** y un árbol del tipo **Command**.

8. Visualización de los AST

Para desplegar correctamente los nuevos AST en el IDE de Luis Leopoldo Pérez, se realizaron las siguientes modificaciones:

1. Se modifica la interfaz **Visitor.java**, del paquete **Triangle.AbstractSintaxTrees** del compilador:
   1.1. Se importan los nuevos AST, para permitir la invocación de los métodos **visit** de cada AST.
   1.2. Se modifica el método visitEmptyCommand, de manera que se despliegue su AST con el nombre “Skip Command”.
   1.3. Se agregan los siguientes métodos abstractos:
      - visitWhileCommand(WhileCommand ast, Object o);
      - visitUntilCommand(UntilCommand ast,Object o);
      - visitDoWhileCommand(DoWhileCommand ast,Object o);
      - visitDoUntilCommand(DoUntilCommand ast,Object o);

- visitForWhileCommand(ForWhileCommand ast,Object o);
- visitForUntilCommand(ForUntilCommand ast,Object o);
- visitForDoCommand(ForDoCommand ast,Object o);
- visitRecursiveDeclaration(RecursiveDeclaration ast, Object o);
- visitLocalDeclaration(LocalDeclaration ast, Object o);
- visitParDeclaration(ParDeclaration ast, Object o);
- visitInitializedVarDeclaration(InitializedVarDeclaration ast, Object o);
- visitInitializedVarDeclarationFor(
  InitializedVarDeclarationFor ast,Object o);
- visitFuncProcFunc(FuncProcFunc ast, Object o);
- visitEmptyProcFuncSequence(EmptyProcFuncSequence ast, Object o);
- visitSingleProcFuncSequence(SingleProcFuncSequence ast, Object o);
- visitMultipleProcFuncSequence(
  MultipleProcFuncSequence ast, Object o);
- visitEmptySingleDeclarationSequence(
  EmptySingleDeclarationSequence ast, Object o);
- visitMultipleSingleDeclarationSequence(
  MultipleSingleDeclarationSequence ast, Object o);
- visitSingleSingleDeclarationSequence(
  SingleSingleDeclarationSequence ast, Object o);
- visitProcProcFunc(ProcProcFunc ast, Object o);

2. Se modifica la clase **TreeVisitor.java**, del paquete **Core.Visitors** del IDE:

   2.1. Se implementan los nuevos métodos abstractos de la interfaz Visitor. La implementación se puede encontrar a partir de la línea 462, de la clase **TreeVisitor.java**

3. Se modifica la clase **IDECompiler.java**, del paquete **Triangle** del IDE:

   3.1. Se comenta la línea 58, para que el compilador no ejecute el análisis contextual.

3.2. Se comenta la línea 62, para que el compilador no ejecute la fase de generación de código.

4. Se sigue la serie de pasos sugeridos por Christian Dávila Amador, disponibles en la sección “Integración con el IDE de Luis Leopoldo Pérez”, del enunciado del proyecto.

9. Análisis de la cobertura del plan de pruebas

Con el fin de dar una lectura más fluida de este documento se escribió el análisis de la cobertura plan de pruebas en el documento **IC-5701 - Plan de pruebas - 2013103311 + 2013109225 + 2013120962**, adjunto en la misma carpeta en la que está guardado este archivo.

10. Plan de pruebas

Con el fin de dar una lectura más fluida de este documento se escribió el plan de pruebas en el documento **IC-5701 - Plan de pruebas - 2013103311 + 2013109225 + 2013120962**, adjunto en la misma carpeta en la que está guardado este archivo.

11. Discusión y análisis de los resultados obtenidos

En esta primera fase del proyecto, logramos completar exitosamente los requerimientos establecidos. En la fase del plan de pruebas todas los resultados obtenidos fueron los que esperábamos, y por lo tanto, no existió la necesidad de realizar cambios al compilador modificado.

12. Reflexión

El proyecto en cuestión fue bastante educativo, pues se experimenta de primera mano cómo funciona un compilador y cada una de sus partes internas. El compilador escrito por Watt y Brown ofrece la oportunidad de modificar fácilmente sus partes, pues se encuentra dividido en múltiples paquetes, cada uno encargado de distintas tareas. Además, se usa el patrón de diseño *visitor*, lo que facilita en gran parte la comunicación entre las partes del compilador. Adicionalmente el IDE desarrollado por Luis Leopoldo Pérez se encuentra diseñado de manera que aprovecha el patrón *visitor***,** lo que hizo sumamente fácil la modificación del paquete encargado de dibujar el árbol sintáctico.

Asimismo, el compilador y el IDE ofrecen un código bastante limpio y estandarizado. Por ejemplo, el nombre de los métodos *visit* de cada árbol sintáctico sigue el estándar **visit<Nombre del AST>**, o el nombre de los métodos de reconocimiento **parse<AST>.**

13. Tareas realizadas por cada miembro del grupo

1. Isac Joel Barrantes Garro:

    1.1. Modificaciones en el analizador léxico (Adición de nuevos token y palabras reservadas)

    1.2. Cambios a la clase **Parser.java**:
        1.2.1. Modificación del método **parseSingleDeclaration**.
        1.2.2. Creación del método **parseCompoundDeclaration**.
        1.2.3. Creación del método **parseProcFuncSequence** y. **parseProperFuncSequence**.
        1.2.4. Creación del método **parseProcFunc**.

    1.3. Nuevos AST creados:
        1.3.1. RecursiveDeclaration
        1.3.2. ParDeclaration
        1.3.3. LocalDeclaration
        1.3.4. InitializedVarDeclaration
        1.3.5. ProcFuncS
            - EmptyProcFuncSequence
            - SingleProcFuncSequence
            - MultipleProcFuncSequence
        1.3.6. ProcFunc
            - FuncProcFunc
            - ProcProcFunc
        1.3.7. SingleDeclarationSequence
            - EmptySingleDeclarationSequence
            - SingleSingleDeclarationSequence
            - MultipleSingleDeclarationSequence

    1.4. Adaptación del IDE
        1.4.1. Cambios al **TreeVisitor.java**

    1.5. Redacción de parte de la documentación.

2. Edisson López Díaz

    2.1. Cambios a la clase **Parser.java**:

2.1.1. Creación del método **parseSingleDeclarationSequence** y **parseProperSingleDeclarationSequence**.

2.2. Revisión del código añadido y los cambios realizados al código fuente del compilador.

2.3. Códigos .tri para la ejecución del plan de pruebas para el analizador léxico y sintáctico:
- 2.3.1. Variantes de single-Command
- 2.3.2. Variantes de compound-Declaration
  - Recursive
  - Par
  - Local
- 2.3.3. Variantes de ProcFuncS
- 2.3.4. Variantes de single-Declaration
- 2.3.5. Modificaciones hechas a reglas existentes antes de la modificación

2.4. Redacción de parte de la documentación.

3. Katerine Molina Sánchez:

3.1. Cambios a la clase **Parser.java:**
- 3.1.1. Modificación del método **parseSingleCommand**. Incluye el reconocimiento de las formas **FOR** y **REPEAT**, así como el comando **skip** y múltiples modificaciones a las formas existentes.

3.2. Adaptación del IDE
- 3.2.1. Cambios al **TreeVisitor.java**

3.3. Nuevos AST creados:
- 3.3.1. DoUntilCommand
- 3.3.2. DoWhileCommand
- 3.3.3. ForDoCommand
- 3.3.4. ForUntilCommand
- 3.3.5. ForWhileCommand
- 3.3.6. UntilCommand
- 3.3.7. InitializedVarDeclarationFor

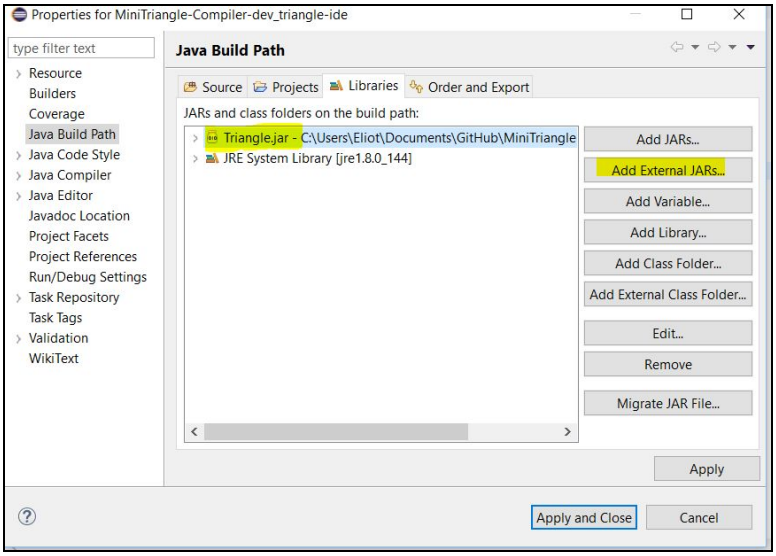
3.4. Depuración del código añadido.

3.5. Redacción de parte de la documentación.

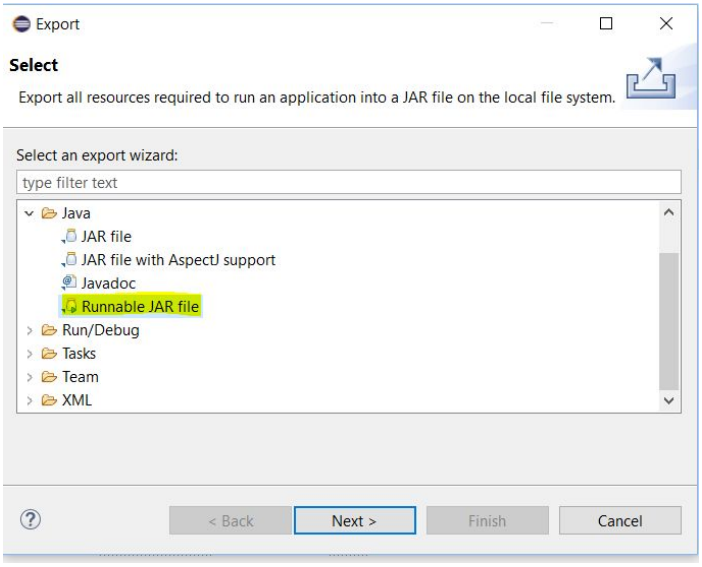
14. Indicar cómo debe compilarse su programa

Para compilar el texto fuente y el IDE en un solo ejecutable, se puede hacer uso de la herramienta Eclipse Oxygen, siguiendo las siguientes instrucciones:

1. Abrir el código fuente del compilador como un proyecto de Eclipse.
2. Exportar el código fuente del compilador como un archivo jar.
3. Abrir el código fuente del IDE como un proyecto de Eclipse
4. Importar el jar generado como una librería, usando las propiedades del proyecto del IDE.



5. Ya es posible ejecutar el IDE desde la herramienta eclipse, a continuación se encuentran los pasos para generar el código objeto para ejecutar desde la máquina virtual de Java.
6. Exportar el IDE como un *runnable jar file.*



15. Indicar cómo debe ejecutarse su programa

Para ejecutar el programa, basta con hacer doble click encima del **archivo jar** generado. Se adjunta el proyecto compilado en la raíz de la carpeta que contiene la solución del proyecto. Es necesario tener instalado el *Java Runtime Environment*.

16. Archivos con el código fuente del programa

Los archivos con el código fuente del compilador están organizados en la carpeta **Source Code** en la raíz del archivo **.zip**.

16. Archivos con el código objeto del programa

El archivo con el código objeto del compilador se encuentra en la carpeta raíz del archivo **.zip**.