

Proyecto 1

Las fases de análisis léxico y sintáctico de un compilador

Instituto Tecnológico de Costa Rica

Compiladores e Intérpretes

Profesor: Ignacio Trejos Zelaya

Estudiantes

Tania María Sánchez Irola	2018138723
Luis Diego Alemán Zúñiga	2018135386
Oswaldo Ramírez Fernández	2020044182

Tabla de Contenido

Introducción	4
Analizador sintáctico y léxico	4
1. Esquema para el manejo del texto fuente	4
2. Modificaciones hechas al analizador léxico.	4
3. Cambios hechos a los tokens y a cualquier otra estructura de datos.	5
4. Estrategia para generar el HTML	6
• Formato al código:	6
5. Cambio realizado a las reglas sintácticas de xt.	9
6. Nuevas rutinas de reconocimiento sintáctico y modificaciones a las existentes.	9
7. Lista de nuevos errores sintácticos detectados, con los nuevos mensajes de error.	10
8. Modelaje realizado para los árboles de sintaxis abstracta.	10
9. Extensión realizada a los métodos que permiten visualizar los árboles de sintaxis abstracta	11
10. Representación de los AST en texto en XML.	12
11. Plan de pruebas para validar el compilador:	15
• Analisis Lexico	15
• Analisis Sintactico	15
12. Discusión, análisis y conclusiones de los resultados obtenidos.	17
13. Experiencia de modificar fragmentos de un compilador / ambiente escrito por terceros.	18
14. Tareas realizadas por cada miembro del grupo de trabajo.	18
15. Indicar cómo debe compilarse el programa.	18
16. Indicar cómo debe ejecutarse el programa.	19
Apéndices.	20
1. Apéndice 1	20
2. Apéndice 2	20
3. Apéndice 3	20
4. Apéndice 4	20
• Apéndice 4.1	21
• Apéndice 4.2	21
5. Apéndice 5	21
6. Apéndice 6	21
• Apéndice 6.1	21
• Apéndice 6.2	22
7. Apéndice 7	22
8. Apéndice 8	22
9. Apéndice 9	22
10. Apéndice 10	23
11. Apéndice 11	23
12. Apéndice 12	23
13. Apéndice 13	23
14. Apéndice 14	23
15. Apéndice 15	23
16. Apéndice 16	24
17. Apéndice 17	24
18. Apéndice 18	24
19. Apéndice 19	24
20. Apéndice 20	24

21. Apéndice 21	25
22. Apéndice 22	25
23. Apéndice 23	25
24. Apéndice 24	25
25. Apéndice 25	26

Introducción

Este proyecto consiste en una extensión del compilador del lenguaje Triángulo implementado mediante los detalles relativos a las fases de análisis léxico y sintáctico de un compilador escrito "a mano" mediante las técnicas expuestas por Watt y Brown en su libro *Programming Language Processors in Java*. Por medio de esta extensión se será capaz de procesar una extensión del lenguaje Triángulo, pequeño lenguaje imperativo con estructura de bloques anidados, que descende de Algol 60 y de Pascal.

Analizador sintáctico y léxico

Esquema para el manejo del texto fuente

Se modificó la manera en que el IDE mantiene la posición de un token con el fin de poder mantener la posición de la columna en la que se encuentra. Para ello se modificaron los siguientes archivos: `SourceFile`, `SourcePosition`, y `Scanner`.

A continuación se detallarán las modificaciones:

- `SourceFile`: Se agregó un contador para cada char solicitado y las instrucciones necesarias para mantenerlo.
- `SourcePosition`: Se modificó de manera que `start` y `end` almacenan un arreglo de dos `int`, donde la posición 0 corresponde a la línea y la posición 1 corresponde a la columna.
- `Scanner`: se realizaron las modificaciones correspondientes para funcionar con la nueva estructura de datos.

Además, para la creación del html, se cuenta con un archivo fuente tipo plantilla, en base al cual se genera el archivo final.

Modificaciones hechas al analizador léxico.

Se agregó la alternativa para procesar el token `".."` aprovechándose de la estructura del switch ya existente, se modificó la alternativa correspondiente al token `'.'`. Además como se mencionó en el punto anterior, se realizó la modificación de la clase `Scanner` para cumplir con el nuevo manejo en las posiciones de los tokens.

Cambios hechos a los tokens y a cualquier otra estructura de datos.

Fueron agregadas las palabras reservadas `and`, `choose`, `elsif`, `for`, `from`, `nothing`, `private`, `recursive`, `repeat`, `until`, `when` y `leave`. Se agregaron en orden alfabético en la tabla de palabras reservadas y en las token clases.

Estrategia para generar el HTML

Para simplificar la edición del *HTML* se optó por utilizar una especie de plantilla, es decir, hacer todo el manejo de estilos mediante *CSS* y *JavaScript*, y solamente inyectar el código.

Es importante mencionar, que el estilo de visualización fue inspirado por la extensión *CodeSnap*, de *Visual Studio Code*.

Para manejar los tabuladores y espacios en blanco, se optó por usar dos tags nativos de *HTML*, los cuales son `<pre>` `</pre>` y `<code>` `</code>`, la estructura final es la siguiente:



```
1 <body>
2   <div class="code-style">
3
4     <div>
5       <div class="dot red"></div>
6       <div class="dot amber"></div>
7       <div class="dot green"></div>
8     </div>
9
10    <pre><code id="code"></code></pre>
11  </div>
12 </body>
13
```

Lo que hacemos ya para generar la visualización, es leer el archivo `.tri` en `java`, e insertar todo el texto en medio de los tags de `<code>` `</code>`.

Formato al código:

Respecto a la forma en la que realizamos el formato del código, es de la siguiente manera:

1. Primeramente definimos en *CSS* las distintas variables y estilos a utilizar (como es mucho se va a mostrar lo principal):



```
:root {
  --blue: #3737ff;
  --green: #2eaf77;
}

font-family: 'Fira Code', monospace;
font-size: 1em;
display: block;
}
```

2. En *JavaScript* hacemos el formato:
 - a. Primeramente colocamos los números de línea al lado izquierdo.
 - b. Definimos un array con las palabras reservadas (no se muestran todas aquí por cuestiones de formato):

```
const palabrasReservadas = ["and",  
    "array",  
    "chose",  
    "const",  
    "do",  
    "else",
```

c. Definimos un par de variable:

- i. `codigoFinal`: va a ser el texto ya con formato.
- ii. `isComment`: es una variable temporal para cuando una línea tiene un comentario.

```
var codigoFinal = ""  
var isComment = false
```

d. Dividimos el código en líneas y las vamos leyendo una por una:

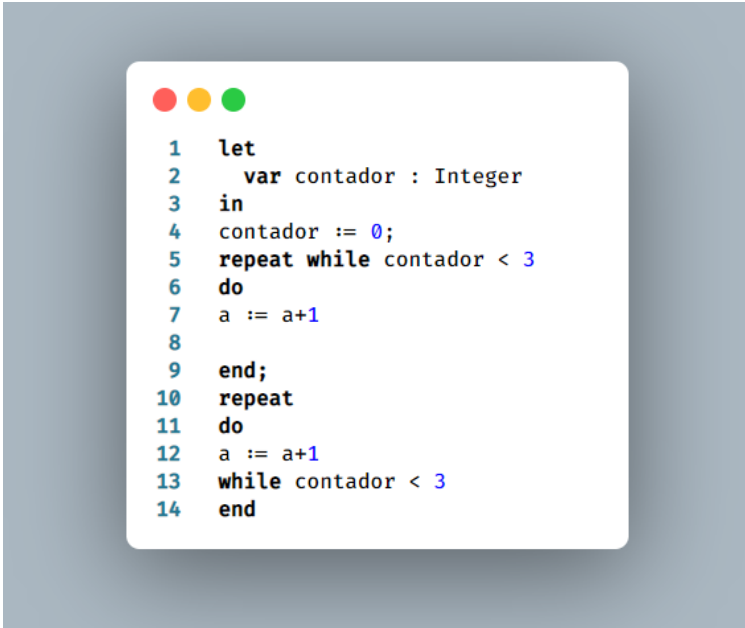
- i. Para optimizar lo primero que se hace es revisar si la línea comienza con `!` en caso de que sí, toda la línea se hace un comentario, se le hace append a `codigoFinal` y se pasa a la siguiente.
- ii. Si el caso anterior no se da, definimos una variable temporal llamada `tempLinea`, dividimos la línea en palabras (separadas por un espacio) e iteramos por estas:
 1. Revisamos si la palabra inicia con `!` en caso de que sí:
 - a. la variable `isComment` se hace **true**
 - b. le añadimos a `tempLinea` la palabra que estamos viendo, pero en color verde, además de un espacio en blanco
 - c. Pasamos a la siguiente palabra
 2. Revisamos si `isComment` es **true**, en caso que si:
 - a. le añadimos a `tempLinea` la palabra que estamos viendo, pero en color verde, además de un espacio en blanco
 - b. Pasamos a la siguiente palabra
 3. Si los casos contrarios no se dan, se hace lo siguiente:
 - a. Verificamos si la palabra actual se encuentra en la lista de reservadas, de ser cierto, la hacemos en negrita

- b. Usando la siguiente expresión regular `(['^']+)` hacemos azul cualquier conjunto que se encuentre en medio de comillas.
- c. Hacemos lo mismo con los caracteres numericos con:

```
for (let i = 0; i < 10; i++){  
    word = word.replace(new RegExp("(" + i + ")", "g"), makeblue("$1"))  
}
```

- d. Le añadimos a `tempLinea` la palabra que acabamos de modificar
- iii. Le añadimos a `codigoFinal` la variable `tempLinea` más un cambio de línea
 - iv. Reemplazamos el texto original que estaba en medio de los tags de `<code>` `</code>`, por el el código con formato en `codigoFinal`.

Un ejemplo de un HTML generado:



```
1  let  
2    var contador : Integer  
3  in  
4    contador := 0;  
5    repeat while contador < 3  
6    do  
7      a := a+1  
8  
9    end;  
10   repeat  
11   do  
12     a := a+1  
13   while contador < 3  
14   end
```


Cambio realizado a las reglas sintácticas de xt.

Se realizaron las correspondientes factorizaciones izquierdas, específicamente se realizó sobre repeat, for y var. En el caso de repeat habían 4 alternativas iniciadas en repeat por lo que se decidió factorizar, además se separó el cuerpo en una nueva regla, repeat-body, que contiene las alternativas do, while y until esto. Para for y var, se factorizaron alternativas con el mismo token de inicio.

Para ver el resultado concreto referirse al apéndice 24

Nuevas rutinas de reconocimiento sintáctico y modificaciones a las existentes.

Se modificaron las siguientes rutinas:

- `parseSigleCommand`: Se modificó para agregar las alternativas for, repeat y nothing, así como modificar las alternativas if para agregar el elsif, let para comprobar el token end al final. finalmente se eliminó la alternativa begin.
- `parseDeclaration`: Se modificó con el fin de que llame a `parseCompoundDeclaration` en vez de `parseSingleDeclaration`.
- `parseSingleDeclaration`: Se modificó la alternativa var para que pudiese se inicializada.
- `parseTypeDenoter`: Fue modificado para aceptar el array con intervalo.

Se agregaron las siguientes rutinas:

- `parseRestOfIf`: Función auxiliar encargada del parseo de los elsif de manera recursiva.
- `parseProcFunc`: Función encargada del parseo de un proc func.
- `parseProcFuncs`: Función encargada del parseo de Proc-Funcs, además comprueba que exista al menos un and.
- `parseRestOfPropFuncs`: Función auxiliar que parsea a los Proc-Funcs una de manera recursiva, es llamado por `parseProcFuncs` una vez comprobada que existe al menos un and.

Lista de nuevos errores sintácticos detectados, con los nuevos mensajes de error.

Estos son los nuevos mensajes de error añadidos al programa.

Error	Mensaje
Hay repeat con TOKEN <code>do</code> pero no hay un <code>until</code> o un <code>while</code>	<i>"Expected until or while"</i>
Hay <code>repeat</code> pero no hay un <code>until</code> o un <code>while</code>	<i>"Expected do until or while"</i>
Hay <code>for</code> mal formado	<i>"No se esperaba el token actual"</i>

Además se añadió la funcionalidad de errores léxicos, para así no generar el *HTML* en caso de errores de este tipo, la única vez que se llama esto es cuando no hay un *EOT* al final del documento.

Modelaje realizado para los árboles de sintaxis abstracta.

Array Type Denoter(ATD): Se agregó un Integer Literal para la nueva forma de ATD, en caso de ser la forma anterior, este integer literal es inicializado en null, dando lugar a si a ambos AST.

For Command (FC): El FC a nivel de código se compone de 6 elementos, ForCommandDefinition, Expression1, Expression2, do Command, leave Command y un boolean. Los componentes opcionales Expression2, y leave Command son nulos en caso de no estar presentes, el booleano es utilizado para que en caso de estar presente un while o until, poder diferenciarlos, si es true, se trata de un while y si es false se trata de un until. Se hizo así para utilizar la misma clase.

ForCommandDefinition(FCD): Contiene un Identifier y un Expression, se realizó de esta manera a recomendación del profesor.

IfCommand: Permaneció sin modificación a pesar de la existencia de el *elsif*.

InitializedVarDeclaration: Variable inicializada, compuesta de un identifier y un expression.

PrivateDeclaration: Compuesto con Declaration1 y Declaration2.

ProcFunc: Sirve como contenedor, tiene un ProcDeclaration y FuncDeclaration. Siempre habrá uno de ellos en null.

RecursiveDeclaration: Compuesto de un proc func y un RecursiveDeclaration, la comprobación de que exista al menos un and seguido de un proc func se realiza durante el parsing, por lo cual en este punto se asume como cumplida. Al finalizar el parsing de procFuncs se RecursiveDeclaration será null.

Repeat: AST para las formas repeat while y repeat until, diferenciando entre estas dos con un boolean. Compuesto de un expression, command y leave command. En caso de no existir leave, se inicializa en null.

RepeatDo: AST para las formas repeat do while y repeat do until, diferenciando entre estas dos con un boolean. Compuesto de un command, expression y leave command. En caso de no existir leave, se inicializa en null.

Extensión realizada a los métodos que permiten visualizar los árboles de sintaxis abstracta

Lo primero a destacar es la creación de la función createQuinary() en la clase TreeVisitor, la cual permite la creación de un árbol con 5 hijos, esto permite el despliegue de for A .. B while C do D leave E end siendo A,B,C,D,E sub arboles del for Command.

Para los árboles de sintaxis modificados, se agregaron las correspondientes condiciones, sobre todo en aquellos con elementos opcionales que se manejaron como elementos nulos. Por otro lado para los nuevos AST, se incluyó la función correspondiente en la interfaz Visitor lo que obligó implementar las funciones en múltiples clases, en aquellas clases que estaban fuera del alcance de este proyecto permanecieron con un UnsupportedOperationException, o con un return null según fuese conveniente.

En la clase `TreeVisito` se implementó el método de creación del AST en la interfaz según correspondiera.

Representación de los AST en texto en XML.

Se aprovechó la forma en que se genera el árbol de sintaxis en la interfaz gráfica.

Al observar cómo se genera el AST en GUI, vimos que se hace esta llamada:

```
((FileFrame)desktopPane.getSelectedFrame()).setTree((DefaultMutableTreeNode)treeVisitor.visitProgram(compiler.getAST(), null));
```

Al analizarlo se notó que lo que se le está enviando al método `setTree()`, es de tipo `DefaultMutableTreeNode`, al ver la documentación, se notó que esto corresponde a una estructura de datos tipo árbol, tomando esto en cuenta, y aprovechando lo aprendido en cursos anteriores como Estructuras de Datos, se decidió usar esto como guía para realizar el XML.

Se creó una clase dedicada a la generación del XML llamada `XMLGenerator`, esta clase se encarga de crear el archivo, construirlo, y finalmente exportarlo.

Los métodos encargados de añadir nodos son los siguientes:

```
//añadir un hijo a un nodo específico
public Element newChildSP(Element especial, String name) {
    Element element = doc.createElement(name);
    especial.appendChild(element);
    last = element;
    return element;
}

//añadir un hijo a un nodo específico, tiene value
public Element newChildSP(Element especial, String name, String value) {
    Element element = doc.createElement(name);
    especial.appendChild(element);
    element.setAttribute("value", value);
    last = element;
    return element;
}
```

```
}
```

Con estos métodos como base, se recorre el árbol de forma recursiva y se van añadiendo al *XML*.

Importante mencionar el caso de los nodos tipo hoja, viendo el formato general que se sigue en el *AST*, los nodos tipo hoja corresponden a el “*value*”, por lo que se tomó en cuenta si un nodo específico tiene un hijo que es hoja, para la creación del *XML*.

```
private void traverseTree(DefaultMutableTreeNode tree, Element father){
    if (tree.isLeaf()){
        return;
    }

    //para manejar las hojas del árbol(values)
    Boolean hasLeaf = false;
    String leafName = "";

    //iteramos por los hijos a ver si hay una hoja
    Enumeration<TreeNode> children = tree.children();
    while(children.hasMoreElements()){
        TreeNode child = children.nextElement();
        if(child.isLeaf()){
            hasLeaf = true;
            leafName = child.toString();
            break;
        }
    }

    String name = tree.toString();
    name = name.replace(" ", "");
    Element e;
    if (hasLeaf){
        e = newChildSP(father, name, leafName);
    }
    else{
```

```
        e = newChildSP(father,name);
    }

    children = tree.children();
    while(children.hasMoreElements()){
        traverseTree((DefaultMutableTreeNode) children.nextElement(), e);
    }
}
```

Plan de pruebas para validar el compilador:

✂ Se seteó la variable debug en true para las pruebas del Análisis .

✂ Las casillas de la columna “V” en color verde representan las pruebas que esperan un resultado positivo, mientras que las rojas, esperan un resultado negativo

Analisis Lexico

V	Caso de prueba (CP)	Objetivo del CP	Diseño del CP	Resultados esperados	Resultados observados
	Insertar una de las palabras reservadas añadidas	Identificar si el analizador lo logra detectar correctamente	Código del apéndice 1	Identifica el token	El token es reconocido correctamente
	Insertar un TOKEN no existente	Identificar si el analizador detecta error	Código del apéndice 2	Error Sintáctico	Error sintáctico al no asignar un identificador
	Insertar el Token “..”	Identificar si puede distinguir entre “.” y “..”	Código del apéndice 3	Reconoce el token “..”	El token se reconoce correctamente

Analisis Sintactico

V	Caso de prueba (CP)	Objetivo del CP	Diseño del CP	Resultados esperados	Resultados observados
	Utilizar el token nothing	Probar el correcto funcionamiento del <i>nothing</i>	Código del apéndice 4	Compilación correcta	Hubo un error inesperado con el visitor, se corrigió de forma correcta al modificar, en el apéndice 4.1 se muestra el error y en 4.2 la solución
	Utiliza el token nothing en una posición incorrecta	Probar el correcto funcionamiento	Código del apéndice 5	Error de sintaxis	ERROR: "nothing" cannot start a

		del <i>nothing</i>			declaration 2,1..2,8
	let ... end	Probar la modificación del <i>let</i>	Código del apéndice 6	Compilación correcta.	Hubo un error inesperado con el visitor, se corrigió de forma correcta al modificar, en el apéndice 6.1 se muestra el error y en 6.2 la solución
	let sin end	Probar que no se admita un <i>let</i> sin <i>end</i>	apéndice 7	Error de sintaxis.	ERROR: "end" expected here 5,1..5,1
	repeat while	Probar el reconocimiento del <i>while</i>	apéndice 8	Compilación correcta	Compilación correcta
	repeat sin nada	Probar que se genere un error al introducir un <i>repeat</i> de forma incorrecta.	apéndice 9	Error de sintaxis.	ERROR: Expected do until or while 1,9..1,10
	repeat until	Probar el reconocimiento del <i>until</i>	apéndice 10	Compilación correcta	Compilación correcta
	repeat do while	Probar el reconocimiento del <i>do while</i>	apéndice 11	Compilación correcta	Compilación correcta
	repeat do until	Probar el reconocimiento del <i>do until</i>	apéndice 12	Compilación correcta	Compilación correcta
	repeat sin end	Probar que genere un error al tener un <i>repeat</i> sin un <i>end</i>	apéndice 13	Error de sintaxis.	ERROR: "end" expected here 1,29..1,29
	for (sin leave)	Probar el reconocimiento del <i>for</i> .	apéndice 14	Compilación correcta	Compilación correcta
	for (con leave)	Probar el reconocimiento <i>leave</i>	apéndice 15	Compilación correcta	Compilación correcta

	for while (sin leave)	Probar el reconocimiento del <i>for while</i> .	apéndice 16	Compilación correcta	Compilación correcta
	for until(sin leave)	Probar el reconocimiento del <i>for until</i> .	apéndice 17	Compilación correcta	Compilación correcta
	for while (con leave)	Probar el funcionamiento del <i>leave</i> con otra variante del <i>for</i>	apéndice 18	Compilación correcta	Compilación correcta
	Variable sin inicializar	Probar el que aún funcione la declaración de una variable sin inicializar	apéndice 19	Compilación correcta	Compilación correcta
	Variable inicializada	Probar el funcionamiento de la variable inicializada	apéndice 20	Compilación correcta	Compilación correcta
	Declaración recursiva	Probar el funcionamiento de la declaración recursiva	apéndice 21	Compilación correcta	Compilación correcta
	Declaración privada	Probar el funcionamiento de la declaración privada	apéndice 22	Compilación correcta	Compilación correcta
	Array sin intervalo	Probar el funcionamiento de un array sin intervalo	apéndice 23	Compilación correcta	Compilación correcta
	Array con intervalo	Probar el funcionamiento de un array con intervalo	apéndice 24	Compilación correcta	Compilación correcta

Discusión, análisis y conclusiones de los resultados obtenidos.

Con los resultados obtenidos nos dimos cuenta de que el programa funcionaba de manera correcta. Se evidencia en los resultados que se obtuvieron en

las diferentes fases, es importante tomar en cuenta que lo que esperamos no siempre es lo que va a suceder realmente, por lo que es fundamental realizar pruebas a nuestros programas para verificar su correcto funcionamiento, ya que de no realizar las mismas es probable que se presenten problemas a futuro que no se tenían contemplados.

Experiencia de modificar fragmentos de un compilador / ambiente escrito por terceros.

La experiencia de modificar fragmentos de código de un compilador/ambiente escrito por terceras personas, puede ser frustrante. El hecho de que sea código ajeno y que para poder modificarlo sea completamente necesario el comprenderlo puede llegar a frustrar bastante en situaciones de proyecto. Si fuesen casos distintos donde uno lo hace por disfrute y con la intención de aprender por cuenta propia nos parece que podría llegar a ser muy interesante, retador y probablemente divertido.

Para poder realizar cambios en el código de otras personas es sumamente importante comprender cómo funciona el mismo (es por ello que se solicita bastante el uso de comentarios, pues a veces incluso uno mismo olvida cómo funciona), ya que si no se tiene este conocimiento es muy difícil saber exactamente dónde se deben realizar los cambios para que este funcione correctamente. Es por ello que muchas veces se invierte mucho del tiempo en lograr esa meta de comprensión.

Tareas realizadas por cada miembro del grupo de trabajo.

- Cambios sobre Triángulo: Tania y Luis
- Archivos XML y HTML: Oswaldo
- Documentación: Todos

Indicar cómo debe compilarse el programa.

1. Utilizando el IDE NetBeans, abrir el proyecto fuente.
2. En la pestaña Projects, hacer click derecho sobre el proyecto
3. clickear la opción "Clean and Build"

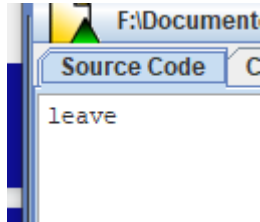
4. Se generará un .jar en la carpeta dist

Indicar cómo debe ejecutarse el programa.

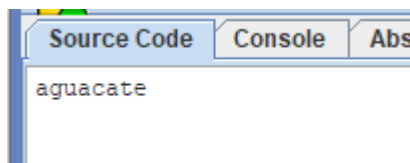
1. Tener el JRE instalado.
2. Doble click sobre el .jar del programa.
3. El programa se ejecutará.

Apéndice.

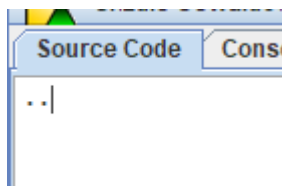
Apéndice 1



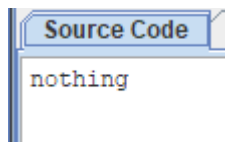
Apéndice 2



Apéndice 3



Apéndice 4



Apéndice 4.1

```
Exception in thread "AWT-EventQueue-0" java.lang.UnsupportedOperationException: Not supported yet.  
    at Core.Visitors.TableVisitor.visitNothing(TableVisitor.java:667)  
    at Triangle.AbstractSyntaxTrees.Nothing.visit(Nothing.java:23)  
    at Core.Visitors.TableVisitor.visitProgram(TableVisitor.java:623)  
    at Core.Visitors.TableVisitor.getTable(TableVisitor.java:654)  
    at GUI.Main.compileMenuItemActionPerformed(Main.java:626)  
    at GUI.Main$9.actionPerformed(Main.java:324)  
    at java.desktop/javax.swing.AbstractButton.fireActionPerformed(AbstractButton.java:1972)  
    at java.desktop/javax.swing.AbstractButton$Handler.actionPerformed(AbstractButton.java:2313)  
    at java.desktop/javax.swing.DefaultButtonModel.fireActionPerformed(DefaultButtonModel.java:405)  
    at java.desktop/javax.swing.DefaultButtonModel.setPressed(DefaultButtonModel.java:262)  
    at java.desktop/javax.swing.plaf.basic.BasicButtonListener.mouseReleased(BasicButtonListener.java:297)  
    at java.desktop/java.awt.AWTEventMulticaster.mouseReleased(AWTEventMulticaster.java:297)  
    at java.desktop/java.awt.Component.processMouseEvent(Component.java:6626)  
    at java.desktop/javax.swing.JComponent.processMouseEvent(JComponent.java:3389)  
    at java.desktop/java.awt.Component.processEvent(Component.java:6381)
```

Apéndice 4.2

```
public Object visitNothing(Nothing ast, Object o) {  
    throw new UnsupportedOperationException("Not supported yet."); // Generated from nbfs://nbhost/SystemFileSystem/Templates/Classes/Code/GeneratedMethodBody  
    return null;  
}
```

Apéndice 5

```
Source Code  Cons  
let  
nothing  
in  
nothing|
```

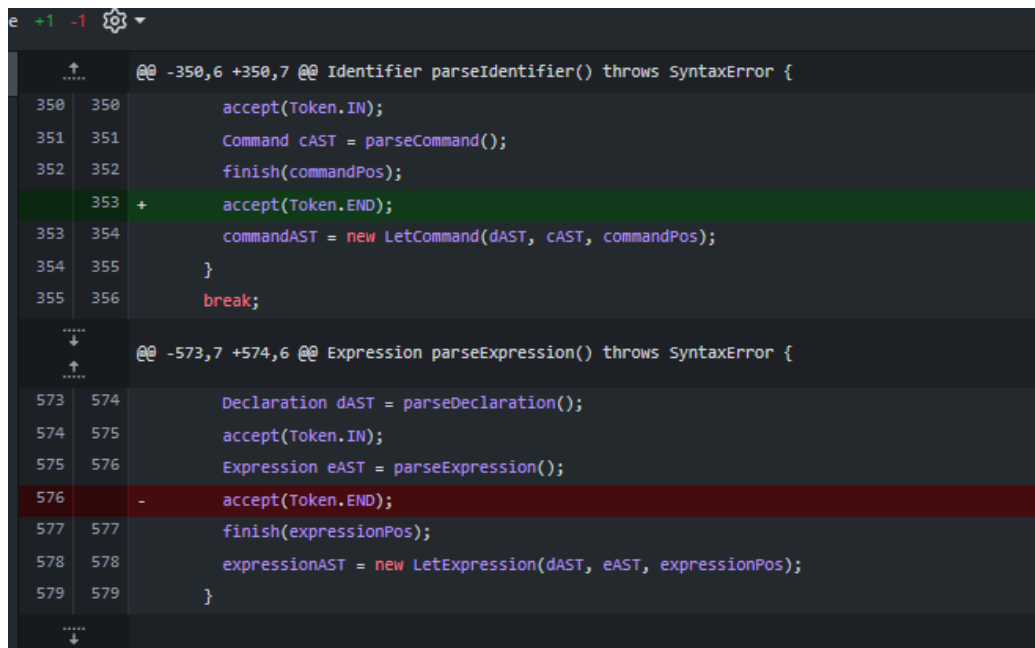
Apéndice 6

```
Source Code  Co  
let  
var a:Integer  
in  
putint(a)  
end|
```

Apéndice 6.1

```
Syntactic Analysis ...  
ERROR: "end" not expected after end of program 5,1..5,4  
Compilation was unsuccessful.
```

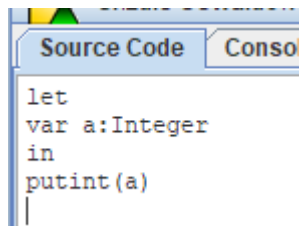
Apéndice 6.2



```
@@ -350,6 +350,7 @@ Identifier parseIdentifier() throws SyntaxError {
350 350     accept(Token.IN);
351 351     Command cAST = parseCommand();
352 352     finish(commandPos);
353 353     + accept(Token.END);
353 354     commandAST = new LetCommand(dAST, cAST, commandPos);
354 355 }
355 356 break;

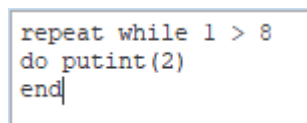
@@ -573,7 +574,6 @@ Expression parseExpression() throws SyntaxError {
573 574 Declaration dAST = parseDeclaration();
574 575 accept(Token.IN);
575 576 Expression eAST = parseExpression();
576 576 - accept(Token.END);
577 577 finish(expressionPos);
578 578 expressionAST = new LetExpression(dAST, eAST, expressionPos);
579 579 }
```

Apéndice 7



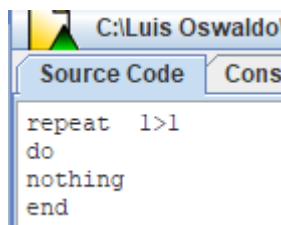
```
let
var a:Integer
in
putint(a)
|
```

Apéndice 8



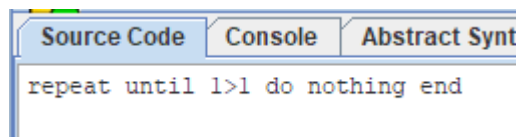
```
repeat while 1 > 8
do putint(2)
end|
```

Apéndice 9



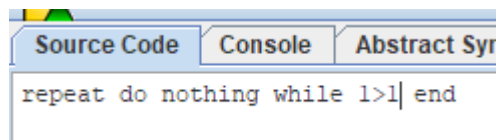
```
C:\Luis Oswaldo\
Source Code Cons
repeat 1>1
do
nothing
end
```

Apéndice 10



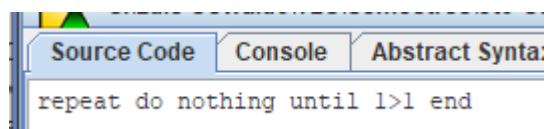
A screenshot of a code editor with three tabs: 'Source Code', 'Console', and 'Abstract Synt'. The 'Source Code' tab is active, showing the text: `repeat until 1>1 do nothing end`.

Apéndice 11



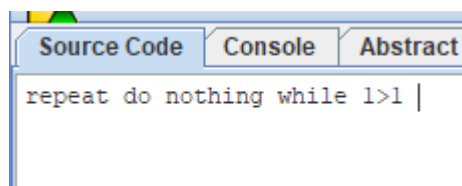
A screenshot of a code editor with three tabs: 'Source Code', 'Console', and 'Abstract Synt'. The 'Source Code' tab is active, showing the text: `repeat do nothing while 1>1 end`.

Apéndice 12



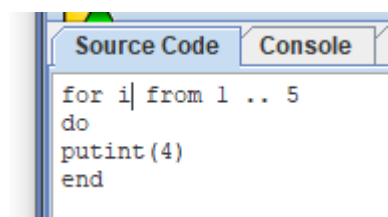
A screenshot of a code editor with three tabs: 'Source Code', 'Console', and 'Abstract Synta:'. The 'Source Code' tab is active, showing the text: `repeat do nothing until 1>1 end`.

Apéndice 13



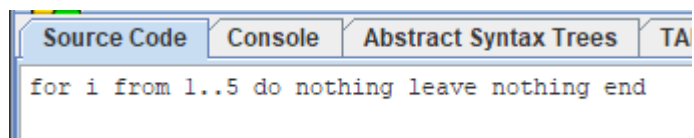
A screenshot of a code editor with three tabs: 'Source Code', 'Console', and 'Abstract'. The 'Source Code' tab is active, showing the text: `repeat do nothing while 1>1 |`.

Apéndice 14



A screenshot of a code editor with two tabs: 'Source Code' and 'Console'. The 'Source Code' tab is active, showing the text: `for i| from 1 .. 5
do
 putint(4)
end`.

Apéndice 15

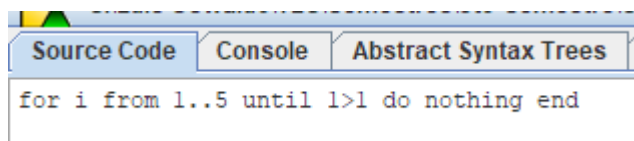


A screenshot of a code editor with four tabs: 'Source Code', 'Console', 'Abstract Syntax Trees', and 'TA'. The 'Source Code' tab is active, showing the text: `for i from 1..5 do nothing leave nothing end`.

Apéndice 16

```
for i from 1 .. 5
while
i>4
do
putint(4)
end
```

Apéndice 17



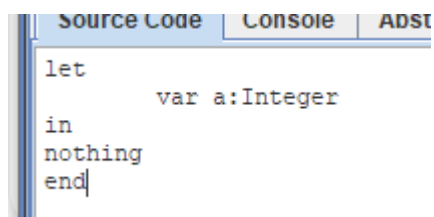
The screenshot shows a code editor with three tabs: 'Source Code', 'Console', and 'Abstract Syntax Trees'. The 'Source Code' tab is active, displaying the following code:

```
for i from 1..5 until 1>1 do nothing end
```

Apéndice 18

```
for i from 1 .. 5
while
i>4
do
putint(4)
leave
putint(5)
end
```

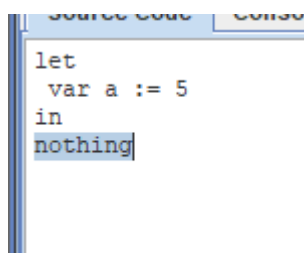
Apéndice 19



The screenshot shows a code editor with three tabs: 'Source Code', 'Console', and 'Abstract Syntax Trees'. The 'Source Code' tab is active, displaying the following code:

```
let
    var a:Integer
in
nothing
end
```

Apéndice 20



The screenshot shows a code editor with two tabs: 'Source Code' and 'Console'. The 'Source Code' tab is active, displaying the following code:

```
let
    var a := 5
in
nothing
```


Apéndice 21

```
let
recursive
  proc agua(cate:Integer) ~ nothing end
  and
  proc sal(chichon:Integer) ~ nothing end
in
nothing
end
```

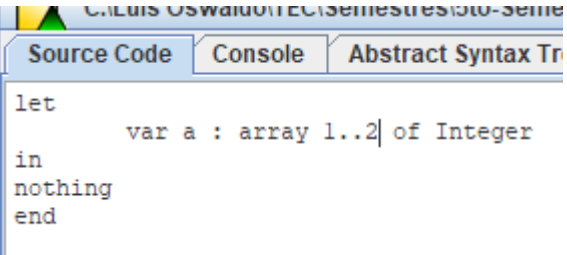
Apéndice 22

```
let
  private var j:Integer in
    proc useless() ~
      j := a
    end
  end
in
nothing
end
```

Apéndice 23

```
let
  var a : array 1| of Integer
in
nothing
end
```

Apéndice 24



```
let
  var a : array 1..2| of Integer
in
nothing
end
```

Apéndice 25

Programs

Program ::= Command

Commands

```
Command ::= single-Command
          | Command ; single-Command
single-Command ::= V-name := Expression
                | Identifier ( Actual-Parameter-Sequence )
                | nothing
                | let Declaration in Command end
                | if Expression then Command (elsif Expression then Command)*
else Command end
                | repeat repeat-body [leave Command] end
                | for Identifier from Expression .. Expression [while Expression |
until Expression] do Command [leave Command] end

repeat-body ::= while Expression do Command
              | until Expression do Command
              | do Command (while Expression | until Expression)
```

Expressions

```
Expression ::= secondary-Expression
            | let Declaration in Expression
            | if Expression then Expression else Expression
secondary-Expression ::= primary-Expression
                    | secondary-Expression Operator primary-Expression
primary-expression ::= Integral-Literal
                   | Character-Literal
                   | V-name
                   | Identifier (Actual-Parameter-Sequence)
                   | (Expressions)
                   | { Record-Aggregate }
                   | [ Array-Aggregate ]
Record-Aggregate ::= Identifier~Expression
                  | Identifier~Expression, Record-Aggregate
Array-Aggregate ::= Expression
                  | Expression, Array-Aggregate
```

Value-or-variable names

```
V-name ::= Identifier
        | V-name.Identifier
        | V-name [ Expression ]
```

Declarations

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

```
Single-Declaration ::= const identifier ~ Expression
                    | var identifier ( : Type-denoter | := Expression )
                    | proc identifier ( Formal-Parameter-Sequence ) ~
                        Command end
                    | func identifier ( Formal-Parameter-Sequence )
                        : Type-denoter ~ Expression
                    | Type identifier ~ Type-Denoter
```

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

```
Proc-Func ::= "proc" Identifier ( Formal-Parameter-Sequence ) ~ Command
            "end"
```

```
            | func Identifier ( Formal-Parameter-Sequence ) : Type-denoter ~
            Expression
```

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

Parameters

```
Formal-Parameter-Sequence ::= proper-Formal-Parameter-Sequence
```

```
Proper-Formal-Parameter-Sequence
    ::= Formal-Parameter
    | Formal-Parameter, proper-Formal-Parameter-Sequence
```

```
Formal-Parameter ::= Identifier : Type-denoter
                  | var Identifier : Type-denoter
                  | proc Identifier ( Formal-Parameter-Sequence )
                  | func Identifier ( Formal-Parameter-Sequence )
```

```
Actual-Parameter-Sequence ::= proper-Actual-Parameter-Sequence
```

```
Proper-Actual_Parameter-Sequence ::= Actual-Parameter
    | Actual-Parameter, proper-Actual-Parameter-Sequence
```

```
Actual-Parameter ::= Expression
                    | var V-name
                    | proc Identifier
                    | func Identifier
```

Type-denoters

```
Type-denoter ::= Identifier
               | array Integer-Literal(.. Integer-Literal) of Type-denoter
               | record Record-Type-denoter end
```

```
Record-Type-denoter ::= Identifier : Type-denoter
                     | identifier : Type-denoter , Record-Type-denoter
```

Lexicon

```
Program ::= (Token | Comment | Blank)*
```

```
Token ::= Integer-Literal | Character-Literal | Identifier | Operator |
array | begin | const | do | else | end | func | if | in | let | of | proc
| record | then | type | var | while | . | : | ; | := | ~ | ( | ) | [ | ] |
{ | }
```

```
Integer-Literal ::= Digit Digit*
```

```
Character-Literal ::= 'Graphic'
```

```
Identifier ::= letter(Letter|Digit)*
```

```
Operator ::= Op-Character Op-Character*
```

```
Comment ::= ! Graphic* end-of-line
```

```
Blank ::= Space|tab|end-of-Line
```

```
Graphic ::= Letter | Digit | Op-character | space | tab | . | : | ; | , |
~ | ( | ) | [ | ] | { | } | _ | | | ! | ' | ` | " | # | $
```

```
letter ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
```

```
Digit ::= 0|1|2|3|4|5|6|7|8|9
```

```
Op-Character ::= +|-|*|/|=|<|>|\|&|@|%|^|?
```