Instituto Tecnológico de Costa Rica
Escuela de Ingeniería en Computación
IC-5701 Compiladores e intérpretes

Proyecto #1, las fases de análisis léxico y sintáctico de un compilador

Historial de revisiones:
- 2023.03.28: v0.

Lea con cuidado este documento. Si encuentra errores en el planteamiento[1], por favor comuníqueselos inmediatamente al profesor.

Objetivo

Al concluir este proyecto, Ud. habrá comprendido los detalles relativos a las fases de análisis léxico y sintáctico de un compilador escrito "a mano" mediante los principios y las técnicas expuestos por Watt y Brown en su libro *Programming Language Processors in Java*. Ud. deberá extender el compilador del lenguaje Δ, desarrollado por Watt y Brown (en Java), de manera que sea capaz de procesar el lenguaje descrito en la sección *Lenguaje fuente* que aparece abajo. Su compilador será la modificación de uno existente, a fin de que sea capaz de procesar el lenguaje Δ extendido conforme se especifica en este documento. Además, su compilador deberá coexistir con un ambiente de edición, compilación y ejecución ('IDE'). Se le suministra un IDE construido en Java por el Dr. Luis Leopoldo Pérez, ajustado por estudiantes de Ingeniería en Computación del TEC.

Base

Para entender las técnicas expuestas en el libro de texto, Ud. deberá estudiar el compilador del lenguaje imperativo Δ y el intérprete de la máquina abstracta TAM, ambos desarrollados en Java por los profesores David Watt y Deryck Brown. El compilador y el intérprete han sido ubicados en la carpeta 'Recursos' del curso (@ OneDrive y @ tecDigital, bajo 'Documentos'), para que Ud. los descargue y los estudie; posteriormente, los modificará según lo especificado en los proyectos del curso. En ese repositorio también pueden encontrar un ambiente interactivo de edición, compilación y ejecución (IDE) desarrollado por el Dr. Luis Leopoldo Pérez (implementado en Java) y corregido por los ex-estudiantes Ing. Pablo Navarro y Ing. Jimmy Fallas. Si desea aprender acerca de cómo integrar el IDE y el compilador, siga las indicaciones preparadas por el Dr. Pérez y nuestro ex-asistente, Ing. Diego Ramírez Rodríguez, en cuanto a las partes del compilador que debe desactivar para poder trabajar, así como los ajustes necesarios para que el IDE y el compilador funcionen bien conjuntamente. *No se darán puntos extra a los estudiantes que desarrollen su propio IDE*; ***no*** *es objetivo de este curso desarrollar IDEs para lenguajes de programación*. Sin embargo, si uno o más grupos de estudiantes aportan un IDE interesante y confiable, podemos considerarlo para futuras ediciones de este curso.

Estudie el capítulo **4** y los apéndices **B** y **D** del libro de Watt y Brown, así como el código del compilador, para comprender los principios y las técnicas por aplicar, el lenguaje fuente original (Δ, que vamos a extender) y las interdependencias entre las partes del compilador.

Entradas

Los programas de entrada serán suministrados en archivos de texto. Los archivos fuente deben tener la extensión `.tri`. Si su equipo 'domestica' un IDE, como el suministrado por el profesor o algún IDE alterno, puede usarlo en este proyecto. En tal IDE, el usuario debe ser capaz de seleccionar el archivo que contiene el texto del programa fuente por procesar, editarlo, guardarlo de manera persistente, compilarlo y enviarlo a ejecución (una vez compilado).

Lenguaje fuente

El lenguaje fuente es una **extensión** del lenguaje Δ ('Triangle' o 'Triángulo'), un pequeño lenguaje imperativo con estructura de bloques anidados, que desciende de Algol 60 y de Pascal. Las adiciones a Δ se detallan abajo; **ponga**

[1] El profesor es un ser humano, falible como cualquiera.

mucho cuidado a los cambios que estamos aplicando sobre Δ. El lenguaje Δ original está descrito en el apéndice B del libro de Watt y Brown. En la carpeta 'Presentaciones' hay un documento con la sintaxis y el léxico de Δ [2].

La extensión de Δ que especificamos en este documento altera varias formas sintácticas de **Δ** y añade diversas formas de comando iterativo, un nuevo comando de selección, declaración de variables inicializadas, arreglos con cotas definidas, declaración de procedimientos o funciones mutuamente recursivos y declaraciones compuestas locales (privadas). Finalmente, hacemos una extensión que permite agrupar declaraciones dentro de *paquetes*, para proveer un mecanismo básico de modularización. La extensión será denominada **Δxt**.

Sintaxis

Convenciones sintácticas (metalenguaje sintáctico)

- **[*x*]** *equivale a* (***x*** | **ε**), es decir, ***x*** *aparece* cero *o* una *veces*.
- ***x**** *equivale a repetir* ***x*** *cero o más veces*, es decir se itera *opcionalmente* sobre ***x***.
- ***x+*** *equivale a repetir* ***x*** *una o más veces*, es decir se itera *obligatoriamente* sobre ***x***.

Cambios a la sintaxis

Esta es la sintaxis ***original*** para los *comandos* de Δ:

```
Command        ::=  single-Command
                  |  Command ; single-Command

single-Command ::=                    Note que está vacío. Es decir, es ε
                  |  V-name := Expression
                  |  Identifier ( Actual-Parameter-Sequence )
                  |  begin Command end
                  |  let Declaration in single-Command
                  |  if Expression then single-Command
                       else single-Command
                  |  while Expression do single-Command
```

Use esa sintaxis como referencia para las modificaciones que describimos a continuación.

~~**Eliminar** de `single-Command` la primera alternativa (ε, el comando vacío)~~[3]

~~**Eliminar** de `single-Command` estas otras alternativas:~~

```
| "begin" Command "end"
| "let" Declaration "in" single-Command
| "if" Expression "then" single-Command "else" single-Command
| "while" Expression "do" single-Command
```

~~**Conservar** esta alternativa en `single-Command`:~~

```
| V-name ":=" Expression
```

En `Single-Command` y en `primary-Expression` **modificar** (respectivamente) las líneas

```
|      Identifier "(" Actual-Parameter-Sequence ")"
```

para que se lean así:

```
|      Long-Identifier "(" Actual-Parameter-Sequence ")"
```

---

[2] Ver el documento 'Triangle_syntax_+_lexicon_(Cob_+_Tapia).docx' preparado por Susana Cob García y Alejandro Tapia Álvarez.

[3] Observe que en la regla original aparece blanco a la derecha de `::=`. Ahora tenemos una palabra reservada para designar el comando vacío (**`skip`**). Esto nos obliga a modificar `parseSingleCommand` en el compilador de base.

Añadir a `single-Command` lo siguiente [4]:

```
    "skip"
| "let" Declaration "in" Command "end"
| "if" Expression "then" Command ("|" Expression "then" Command)*
    "else" Command "end"
| "repeat" "while" Expression "do" Command "end"
| "repeat" "until" Expression "do" Command "end"
| "repeat" "do" Command "while" Expression "end"
| "repeat" "do" Command "until" Expression "end"
| "repeat" Expression "times" "do" Command "end"
| "for" Identifier ":=" Expression ".." Expression
      "do" Command "end"
| "for" Identifier ":=" Expression ".." Expression
      "while" Expression "do" Command "end"
| "for" Identifier ":=" Expression ".." Expression
      "until" Expression "do" Command "end"
| "for" Identifier "in" Expression "do" Command "end"
| "select" Expression "from" Cases ["else" Command] "end"
```

Observe que ahora *todos* los comandos compuestos terminan con **end**. Observe que en los comandos *compuestos* ya no se usa `single-Command`, sino `Command`.

Añadir las siguientes reglas:

```
Cases          ::= Case+
Case           ::= "when" Case-Literals "then" Command
Case-Literals  ::= Case-Range ("|" Case-Range)*
Case-Range     ::= Case-Literal [".." Case-Literal]
Case-Literal   ::= Integer-Literal | Character-Literal
```

Modificar `Declaration` para que se lea

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

Añadir esta nueva regla (declaración de funciones y procedimientos mutuamente recursivos, declaraciones locales):

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

Añadir estas reglas [5]:

```
Proc-Func
        ::= "proc" Identifier "(" Formal-Parameter-Sequence ")"
                "~" Command "end"
        | "func" Identifier "(" Formal-Parameter-Sequence ")"
                ":" Type-denoter "~" Expression

Proc-Funcs
        ::= Proc-Func ("|" Proc-Func)+
```

Considere la regla `single-Declaration`

[4] Recuerde que `single-Command` y `Command` son no-terminales (categorías sintácticas) distintos. ***No*** hemos factorizado las reglas para hacer evidentes las diferencias entre las nuevas formas de comando y las anteriores.

[5] Observe que al usar **recursive**, la sintaxis obliga a declarar al menos ***dos*** procedimientos y funciones como mutuamente recursivos.

```
single-Declaration  ::=  const Identifier ~ Expression
                       | var Identifier : Type-denoter
                       | proc Identifier ( Formal-Parameter-Sequence ) ~
                           single-Command
                       | func Identifier ( Formal-Parameter-Sequence )
                           : Type-denoter ~ Expression
                       | type Identifier ~ Type-denoter
```

En `single-Declaration`, **modificar** la opción referente a **`proc`** para que se lea:

```
        …
        |   "proc" Identifier "(" Formal-Parameter-Sequence ")"
                "~" Command "end"
        | …
```

Añadir a `single-Declaration` la declaración de variable inicializada[6]:

```
        | "var" Identifier ":=" Expression
```

Tengo la duda si también se debería poder hacer de otra manera

Adición de paquetes (`package`*s*)

Δxt proveerá una forma primitiva de *módulos* o 'paquetes', que permitirán agrupar declaraciones y darles un nombre colectivo. Por simplicidad, los paquetes no podrán ser anidados y todos estarán *antes* del comando que inicia propiamente el 'programa principal'.

Modificar `Program` para que se lea:

```
Program     ::=   Package-Declaration* Command
```

Reglas relativas a los **`package`***s*.

Añadir:

```
Package-Declaration ::= "package" Package-Identifier "~"
                                    Declaration "end"

Package-Identifier  ::= Identifier
Long-Identifier     ::= [ Package-Identifier "$" ] Identifier
```

Modificar `V-name` para que se lea:

```
V-name ::=    [ Package-Identifier "$" ] Var-name

Var-name      ::=   Identifier
              |     Var-name "." Identifier
              |     Var-name "[" Expression "]"
```

Modificar la definición de `Type-denoter` para que se lea así:

```
Type-denoter ::=   Long-Identifier
                   |     "array" Integer-Literal "of" Type-denoter
                   |     "record" Record-Type-denoter "end"
```

Recuerde que en `Single-Command` y en `primary-Expression` debe **modificar** (respectivamente) las líneas

```
|      Identifier "(" Actual-Parameter-Sequence ")"
```

para que se lean así:

```
|      Long-Identifier "(" Actual-Parameter-Sequence ")"
```

---

[6] Esta forma de declarar variables es ***adicional*** a la regla original de Δ donde aparece **`var`**. La declaración de variable original no desaparece.

Los anteriores cambios permitirán invocar funciones y procedimientos declarados en paquetes.

Cambios léxicos

- **Eliminar** la palabra reservada **`begin`**.
- **Añadir** las *palabras reservadas* **`for`**, **`from`**, **`package`**, **`private`**, **`rec`**, **`repeat`**, **`select`**, **`skip`**, **`times`**, **`until`**, **`when`**, **`while`** como *nuevas* alternativas en la especificación de `Token`.
- **Añadir** **`"|"`**, **`"$"`** y **`".."`** como *nuevos* símbolos léxicos (de puntuación) en la especificación de `Token`.
- Al igual que en Δ, en los identificadores las mayúsculas son significativas y distintas de las minúsculas.

Procesamiento y salidas

Ud. modificará el compilador+IDE de **Δ** escritos en Java para que sea capaz de procesar las extensiones especificadas arriba.

- Debe modificar el analizador de léxico para que trabaje con el lenguaje **Δxt** completo (reconocimiento de lexemas, categorización de lexemas en clases léxicas apropiadas, registro de coordenadas de cada lexema).
- Con base en el analizador de léxico (pero *no* como parte de él), su compilador debe *adicionalmente* generar un archivo HTML con el texto fuente del programa, con una apariencia como se indica adelante. Si el programa fuente es `<programa>.tri`, su procesador deberá generar un archivo llamado `<programa>.html`. En el archivo `.html` los siguientes elementos léxicos deberán estar en una tipografía ('fuente', 'font') *monoespaciada* (monospaced), de tamaño `1 em`[7]:
  - Palabras reservadas: en color negro y en negrita. Por ejemplo: **`while`**, **`then`**
  - Identificadores, operadores y separadores: en color negro y sin ningún resaltado. Por ejemplo: `contador`, `>=`, `[`
  - Literales (caracteres y numerales): en color azul oscuro. Por ejemplo: `101`, `'a'`
  - Comentarios: en color verde medio. Por ejemplo: `! Este es un comentario`

  Un reto para ustedes es decidir cómo manejar apropiadamente los tabuladores y los espacios en blanco en HTML. Los fines de línea y los retornos de carro no ofrecen mayor dificultad.
- El archivo HTML |*debe* ser generado si no hay errores léxicos, *aunque hubiera errores sintácticos*.
- Debe modificar el analizador sintáctico de manera que logre reconocer el lenguaje **Δxt** completo y construya los árboles de sintaxis abstracta que correspondan a las estructuras de las frases reconocidas[8, 9].
- El analizador sintáctico debe *detenerse* al encontrar el primer error, reportar precisamente la *posición* en la cual ocurre ese primer error y diagnosticar la naturaleza de dicho error[10].
- El analizador sintáctico debe llenar una estructura de datos en que el IDE presentará el árbol de sintaxis abstracta. Los árboles de sintaxis abstracta deben mostrarse en un panel (pestaña) del IDE, con estructuras de navegación semejantes a las que ya aparecen en la implementación de base que se les ha dado[11].

---

[7] Por ejemplo: Courier, Courier New, Andale Mono, FreeMono, DejaVu Sans, Lucida Console.

[8] Cada una de las variantes de los comandos **`repeat`** y **`for`** debe dar lugar a una *forma* ***distinta*** de árbol de sintaxis abstracta. Esto facilitará el análisis contextual y la generación de código en proyectos futuros.

[9] Para las variantes del comando **`for`** recomendamos usar un árbol *binario* (**`for in`**), *ternario* (**`for :=`**) o *cuaternario* (**`for :=`** con partículas condicionales **`while`** o **`until`**), agrupando las partes según los colores con que se resalta a continuación:

```
"for" Identifier ":=" Expression ".." Expression "do" Command "end"
"for" Identifier ":=" Expression ".." Expression "while" Expression "do" Command "end"
"for" Identifier ":=" Expression ".." Expression "until" Expression "do" Command "end"
"for" Identifier "in" Expression "do" Command "end"
```

El primer caso tiene *tres* subárboles. El primer subárbol tiene a `Identifier` y `Expression` juntos, como subárboles de un mismo árbol; el segundo subárbol corresponde a una `Expression`; el tercero a un `Command`. El árbol binario (`Identifier` , `Expression`) puede heredar de `Declaration`, a fin de simplificar las futuras fases del proceso de compilación.

[10] En el IDE suministrado, esta información se sincroniza de manera que el error sea visible en la ventana del código fuente. Lea bien ese código para comprender la manera en que interactúan las partes y se logra este efecto. Haga pruebas del analizador léxico dado como base, para asegurar que reporta bien la posición donde inicia cada lexema.

[11] Este componente, '`TreeViewer`', permite contraer o expandir los subárboles. Es conveniente usar el patrón 'visitante' ('visitor') para llenar la estructura de datos.

- A partir del AST creado por el analizador sintáctico, su compilador debe crear una representación en XML para el árbol de sintaxis abstracta correspondiente a cada programa analizado [12]. Si el programa fuente es <programa>`.tri`, su procesador deberá generar un archivo llamado <programa>`.xml`.Ver ejemplo al final de este documento [13].
- Si hay errores de sintaxis, el analizador sintáctico *no* debe mostrar el AST en el IDE *ni* generar la representación en XML. Sin embargo, *sí* debe generarse la representación en HTML.
- Las técnicas por utilizar en su trabajo son las expuestas en clase, en el libro de Watt y Brown, en el compilador escrito por Watt y Brown, y en el IDE creado por Luis Leopoldo Pérez.

Ustedes *deben* basarse en los programas dados por el profesor como punto de partida. Su programación debe respetar el estilo aplicado en el procesador usado como base (escrito en Java), eso favorecerá la legibilidad de su código. *En el código fuente debe estar claro dónde hay modificaciones introducidas por ustedes, mediante comentarios que indiquen el nombre de cada persona que hizo el cambio o la adición*.

Debe dar crédito por escrito a cualquier otra fuente de información o ayuda consultada.

Debe ser posible activar la ejecución del IDE de su compilador desde el Explorador de Windows haciendo clics sobre el ícono de su archivo `.jar`, o bien generar un `.exe` a partir de su `.jar`. *Por favor indique claramente cuál es el archivo ejecutable del IDE, para que el profesor o l@s asistentes puedan someter a pruebas su procesador sin dificultades. Si Ud. trabaja en Linux, Mac OS o alguna variante de Unix, por favor avise cuanto antes al profesor y a l@s asistentes.*

Documentación
La documentación es un entregable y va en un solo documento. Debe documentar clara y concisamente los siguientes puntos:

Analizador sintáctico y léxico
- Su esquema para el manejo del texto fuente (lectura de archivos de entrada o de estructura de datos del IDE). Si *no* modifica lo existente, *indique esto explícitamente*.
- Modificaciones hechas al analizador de léxico (*tokens*, tipos, métodos, etc.).
- Cambios hechos a los *tokens* y a cualquier otra estructura de datos (por ejemplo, tabla de palabras reservadas) para incorporar las extensiones al lenguaje.
- Explicación su estrategia para generar la versión HTML del texto del programa fuente.
- Justificar explícitamente *cualquier* cambio realizado a las reglas sintácticas de **Δxt**, para lograr que tenga una gramática LL(1) equivalente a la extensión descrita arriba.
- Generar un apéndice con la macro-sintaxis (sintaxis) y la micro-sintaxis (léxico) de **Δxt** tal que satisfaga las propiedades LL(1).
- Nuevas rutinas de reconocimiento sintáctico, así como cualquier modificación a las existentes.
- Lista de nuevos errores sintácticos detectados, con los nuevos mensajes de error.
- Modelaje realizado para los árboles de sintaxis abstracta (ponga atención al modelaje de categorías sintácticas donde hay ítems repetidos[14]).
- Extensión realizada a los métodos que permiten visualizar los árboles de sintaxis abstracta (desplegados en una pestaña del IDE).

---

[12] Es conveniente usar el patrón 'visitante' ('visitor') para crear las nuevas representaciones del AST. Es usual que las herramientas que despliegan archivos `.xml` permitan al usuario contraer y expandir las estructuras a voluntad (nuestros ASTs en este proyecto). Ver los componentes `Writer.java` y `WriterVisitor.java` como modelos; están en la carpeta 'Recursos'.
[13] En la carpeta 'Recursos', el profesor ha suministrado ejemplos de 'impresores' de XML para el lenguaje Δ original, creados por estudiantes del TEC. Estos pueden servirles de base.
[14] Es importante que decida si los ítems repetidos dan lugar a árboles (de sintaxis abstracta) que tienden a la izquierda o bien a la derecha. Sea consistente en esto, porque afecta los recorridos que deberá hacer sobre los árboles cuando realice el análisis contextual o la generación de código. Estudie el código del compilador de Δ original para inspirarse.

- Extensión realizada a los métodos que permiten representar los árboles de sintaxis abstracta como texto en XML.
- Plan de pruebas para validar el compilador. Debe separar las pruebas para cada fase de análisis: léxico o sintáctico. Debe incluir pruebas *positivas* (para confirmar funcionalidad con datos correctos) y pruebas *negativas* (para evidenciar la capacidad del compilador para detectar errores). Si el profesor distribuye casos de prueba, puede utilizarlos (y dar el crédito correspondiente). Debe especificar lo siguiente para cada caso de prueba [15]:
  - Objetivo del caso de prueba
  - Diseño del caso de prueba
  - Resultados esperados
  - Resultados observados
- Discusión y análisis de los resultados obtenidos. Conclusiones a partir de esto.
- Una reflexión sobre la experiencia de modificar fragmentos de un compilador/ambiente escrito por terceras personas.
- Descripción resumida de las tareas realizadas por cada miembro del grupo de trabajo.
- Indicar cómo compilar su programa.
- Indicar cómo ejecutar su programa.
- Archivos con el texto fuente de su compilador. El texto fuente debe incluir comentarios que indiquen con claridad los puntos en los cuales se han hecho modificaciones y cuál[es] persona[s] las realiz(ó | aron).
- Archivos con el código objeto del compilador. **El compilador debe estar en un formato ejecutable directamente desde el sistema operativo Windows** [16] **o – alternativamente – haber llegado previamente a un acuerdo con el profesor.**
- Debe guardar su trabajo en una carpeta comprimida (formato **`.zip`**) según se indica abajo [17]. Esto debe incluir:
  - Documentación indicada arriba, con una portada donde aparezcan los nombres y números de carnet de los miembros del grupo. Los documentos descriptivos deben estar en formato `.pdf`.
  - Código fuente, organizado en carpetas.
  - Código objeto. Recuerde que el código objeto de su compilador (programa principal) debe estar en un formato directamente ejecutable en Windows (o una alternativa, negociada antes con el profesor).
  - Programas (`.tri`) de entrada que han preparado para probar su analizador sintáctico+léxico.

Entrega

Fecha límite: **jueves 2023.04.27,** antes de las 23:45. No se recibirán trabajos después de la fecha y la hora indicadas.

Los grupos pueden ser de *hasta* **4** personas. Los grupos de *cuatro* miembros *obligatoriamente* deben procesar el comando **`select`** completo y la variante **`for in`**. Para los grupos de *hasta* tres miembros el procesamiento del comando **`select`** y la variante **`for in`** es opcional y permitirá obtener puntos extra.

Deben enviar por correo-e el ***enlace*** [18] a un archivo comprimido almacenado en alguna nube, con todos los elementos de su solución a: `itrejos@itcr.ac.cr`, `hilaryc.cr@estudiantec.cr` (Hilary Castro Cabezas, Asistente). El *archivo* comprimido debe llamarse **`Proyecto 1 carnet carnet carnet carnet`**

El asunto (*subject*) de su mensaje debe ser: **`IC-5701 Proyecto 1 carnet carnet carnet carnet`**

Si su mensaje no tiene el asunto en la forma correcta, su proyecto será castigado con −10 puntos; podría darse el caso de que su proyecto no sea revisado del todo (y sea calificado con **0**) sin responsabilidad alguna del profesor o

---

[15] Puede usar como base los casos de prueba que el profesor publique.

[16] En principio, se permitirá entregar el trabajo en otro ambiente, pero debe avisar de previo al profesor y a los asistentes.

[17] **No use** formato **`.rar`**, porque es rechazado por el sistema de correo-e del TEC. Si usa **`.rar`** obtendrá un 0 como calificación.

[18] Los sistemas de correo han estado rechazando el envío o la recepción de carpetas comprimidas con componentes ejecutables. Suban su carpeta comprimida (en formato **`.zip`**) a algún 'lugar' en la nube y envíen el hipervínculo al profesor y a nuestro asistente mediante un mensaje de correo con el formato indicado. Dar permiso de lectura a ambos. Deben mantener la carpeta viva hasta 31 de enero del 2023.

de l@s asistentes (caso de que su mensaje fuera obviado por no tener el asunto apropiado). Si su mensaje no es legible (por cualquier motivo), su carpeta está en formato **`.rar`**, o contiene un virus, la nota será **0**.

La documentación vale alrededor de un 25% de la nota de cada proyecto. La redacción y la ortografía deben ser correctas. El profesor tiene *altas expectativas* respecto de la calidad de los trabajos escritos y de la programación producida por estudiantes universitarios de tercer año de la carrera de Ingeniería en Computación del Tecnológico de Costa Rica. Los profesores esperamos que los estudiantes tomen en serio la comunicación profesional.

Integración con el IDE de Luis Leopoldo Pérez

Gracias a Christian Dávila Amador y a Diego Ramírez (ambos graduados del TEC), por su colaboración.

1. Seguir las instrucciones preparadas por Diego Ramírez Rodríguez, 'Ramírez_Manual_integración_IDE_v2_2018.08.pdf' (ver carpeta 'Documentos' > 'Recursos' en nuestro espacio en el tecDigital).
2. Leer las instrucciones ubicadas dentro de la documentación del IDE ('Pérez ide-triangle.pdf '/ 'ide-triangle.pdf', pág. 6).
3. Desactivar/cambiar las siguientes líneas dentro de **Main.java** dentro del código del IDE:
   - **Línea 617 (comentar):**
     ```
     disassembler.Disassemble(desktopPane.getSelectedFrame().getTitle().replace(".tri", ".tam"));
     ```
   - **Línea 618 (comentar):**
     ```
     ((FileFrame)desktopPane.getSelectedFrame()).setTable(tableVisitor.getTable(compiler.getAST()));
     ```
   - **Línea 620 (cambiar de true a false):**
     ```
     runMenuItem.setEnabled(true);
     ```
   - **Línea 621 (cambiar de true a false):**
     ```
     buttonRun.setEnabled(true);
     ```
4. **IDECompiler.java**: en realidad, el IDE nunca llama al Compiler.java del paquete de Triangle. El compilador crea uno propio llamado IDECompiler.java. Ahí llama al analizador contextual (Checker) y al generador de código (Encoder). Desactívelos allí.

Defecto conocido en el IDE de Luis Leopoldo Pérez

Gracias a Jorge Loría Solano y Luis Diego Ruiz Vega por reportar el defecto.

Este defecto fue corregido por Pablo Navarro y Jimmy Fallas, estudiantes de IC. Su solución fue colocada en el tecDigital: archivo `Triangle_Java_IDE_LL_Pérez_+_Navarro_&_Fallas.zip`

> *El IDE de Luis Leopoldo Pérez (en Java) tiene una pulga. El problema se da cuando se selecciona un carácter y se sobrescribe con otro. El IDE no detecta que el documento haya cambiado y al compilar, se compila sobre el documento anterior sin tomar en cuenta el nuevo cambio.*
>
> *El problema se da porque el IDE detecta cambios en el documento cuando este cambia de tamaño (se añade o se borra algún carácter), pero en el caso de sobrescribir un carácter esto no cambia el tamaño y por tanto al compilar no se almacenan los cambios.*

Ejemplo de salida en `.html`

Susana Cob García y Alejandro Tapia Álvarez crearon la salida en `.html` *para* Δ*, como parte de su trabajo en el curso 'Compiladores e intérpretes'.*

Programa fuente

Considere el siguiente programa fuente en **Δxt**:

```
! Test of if in command, it runs correctly.

if (1=1+3) then puteol() else put('x'); puteol() end
```

HTML en texto llano

Este es un ejemplo de una salida en `.html`, como texto llano:

```
<p style="font-family: 'DejaVu Sans', monospace;"><font color='#00b300'>! Test of if
in command, it runs correctly.□□</font><br><br><b>if</b><font style='padding-
left:1em'>(<font color='#0000cd'>1</font>=<font color='#0000cd'>1</font>+<font
```

```
color='#0000cd'>3</font>)<font style='padding-left:1em'><b>then</b><font
style='padding-left:1em'>puteol()<font style='padding-left:1em'><b>else</b><font
style='padding-left:1em'>put(<font color='#0000cd'>'x'</font>);<font style='padding-
left:1em'>puteol()<font style='padding-left:1em'><b>end</b></p>
```

HTML desplegado por Firefox (imagen)

```
! Test of if in command, it runs correctly.

if (1=1+3) then puteol() else put('x'); puteol() end
```

HTML desplegado por Chrome (imagen)

```
! Test of if in command, it runs correctly.

if (1=1+3) then puteol() else put('x'); puteol() end
```

Ejemplo de salida en `.xml`

José Antonio Alpízar, Pablo Brenes y Luis José Castillo crearon la salida en `.xml` *para Δ como parte de su Proyecto de Ingeniería del Software.*

Programa fuente

Considere el siguiente programa fuente en Δ:

```
! Test of if in command, it runs correctly.

if (1=1+3) then puteol() else puteol()
```

XML en texto llano

Este es un ejemplo de una salida en `.xml`, como texto llano:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<Program>
<IfCommand>
<BinaryExpression>
<BinaryExpression>
<IntegerExpression>
<IntegerLiteral value="1"/>
</IntegerExpression>
<Operator value="="/>
<IntegerExpression>
<IntegerLiteral value="1"/>
</IntegerExpression>
</BinaryExpression>
<Operator value="+"/>
<IntegerExpression>
<IntegerLiteral value="3"/>
</IntegerExpression>
</BinaryExpression>
<CallCommand>
<Identifier value="puteol"/>
<EmptyActualParameterSequence/>
</CallCommand>
<CallCommand>
<Identifier value="puteol"/>
<EmptyActualParameterSequence/>
</CallCommand>
</IfCommand>
</Program>
```

La salida luce clara, pero programas como Internet Explorer y Edge de Microsoft pueden desplegar archivos `.xml` que reflejen mejor la estructura (la hacen más comprensible y además las líneas con **–** son colapsables y con **+** son expandibles).

XML desplegado por Explorer (imagen)

XML desplegado por Edge (imagen)