






Instituto Tecnológico de Costa Rica  
Escuela de Ingeniería en Computación  
IC-5701 Compiladores e intérpretes  
Proyecto # 1, las fases de análisis de un compilador (léxico, sintáctico, contextual)

Historial de revisiones:


-  2017.09.26: Versión base.
-  2017.09.29: Completar v1.
-  2017.10.10: Cambio de fecha de entrega. v2.

**Lea con cuidado este documento.** Si encuentra errores en el planteamiento<sup>1</sup>, comuníquelos inmediatamente al profesor.

### Objetivo

Al concluir este proyecto Ud. habrá terminado de comprender los detalles relativos a las fases de análisis léxico, sintáctico y contextual de un compilador escrito "a mano" usando las técnicas expuestas por Watt en sus libros *Programming Language Processors* o *Programming Language Processors in Java* (con Deryck Brown). Ud. deberá extender el compilador del lenguaje  escrito en Java, desarrollado por Watt y Brown, 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, de manera que sea capaz de procesar el lenguaje  extendido. Además, su compilador deberá coexistir con un ambiente de edición, compilación y ejecución ("IDE"). Se le suministra un IDE implementado en Java.

### Base

Ud. usará como base el compilador del lenguaje  y el intérprete de la máquina abstracta TAM desarrollado en Java por los profesores David Watt y Deryck Brown. Los compiladores e intérpretes han sido ubicados en la carpeta 'Asignaciones' del curso, para que Ud. los descargue. En el repositorio también se les ha suministrado un ambiente interactivo de edición, compilación y ejecución (IDE) desarrollado por el Ing. Luis Leopoldo Pérez (implementado en Java). **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. Es probable que deba desactivar algunas partes del compilador para poder desarrollar este trabajo. Es probable que deba hacer ajustes a partes del compilador o del IDE por cambios en las versiones de Java ocurridas entre el 2012 y el 2017. ¡Lea el apéndice D del libro y el código del compilador para comprender las interdependencias entre las partes!





### Entradas


Los programas de entrada serán suministrados en archivos de texto. El usuario seleccionará cuál es el archivo que contiene el texto del programa fuente desde el ambiente de programación (IDE) base suministrado, o bien lo editará en la ventana que el IDE provee para el efecto (que permitirá guardarlo de manera persistente). Los archivos fuente deben tener terminación `.tri`.

### Lenguaje fuente

#### Sintaxis

#### Lenguaje fuente




El lenguaje fuente es una extensión del lenguaje  un pequeño lenguaje imperativo con estructura de bloques anidados, que descende de Algol 60 y de Pascal. Las adiciones a  se detallan abajo; **ponga mucho cuidado a los cambios que estamos aplicando sobre ** El lenguaje  original es descrito en el apéndice B del libro de Watt y Brown.

Esta extensión de  añade varias formas de comando iterativo, declaración de variables inicializadas, declaración de procedimientos o funciones mutuamente recursivos y otras declaraciones compuestas (colateral y local).

#### Convenciones sintácticas

---

<sup>1</sup> El profesor es un ser humano, falible como cualquiera.

-   $[x]$  equivale a  $(x \mid \text{skip})$  es decir,  $x$  aparece cero o una vez.
-   $x^*$  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

**Eliminar** de single-Command la primera alternativa (comando vacío)<sup>2</sup>.

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

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

**Añadir** a single-Command lo siguiente<sup>3</sup>:

```
"skip"
| "repeat" "while" Expression "do" Command "end"
| "repeat" "until" Expression "do" Command "end"
| "repeat" "do" Command "while" Expression "end"
| "repeat" "do" Command "until" Expression "end"
| "for" "var" Identifier ":@" Expression "to" Expression "do" Command "end"
| "for" "var" Identifier ":@" Expression "to" Expression
  "while" Expression "do" Command "end"
| "for" "var" Identifier ":@" Expression "to" Expression
  "until" Expression "do" Command "end"
| "let" Declaration "in" Command "end"
| "if" Expression "then" Command "else" Command "end"
```

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

Observe que se conservan estas alternativas en single-Command:

```
| V-name ":@" Expression
| Identifier "(" Actual-Parameter-Sequence ")"
```

**Modificar Declaration para que se lea**

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

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

```
compound-Declaration
  ::= single-Declaration
  | "recursive" Proc-Funcs "end"
  | "local" Declaration "in" Declaration "end"
  | "par" single-Declaration ("and" single-Declaration)+ "end"
```

**Añadir** estas reglas<sup>4</sup>:

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

<sup>2</sup> Observe que en la regla original aparece blanco a la derecha de  $::=$ . Ahora tenemos una palabra reservada para designar el comando vacío (skip). Esto le obliga a modificar parseSingleCommand en el compilador de base.

<sup>3</sup> Recuerde que single-Command y Command son no-terminales (categorías sintácticas) distintos.

<sup>4</sup> Observe que al usar recursive, la sintaxis obliga a declarar al menos *dos* procedimientos y funciones como mutuamente recursivos.

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

Añadir a single-Declaration lo siguiente (variable inicializada)<sup>5</sup>:

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

En la regla de single-Declaration

```
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
```


también modificamos la opción referente a proc para que se lea:










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

#### Cambios léxicos

**Añadir** las palabras reservadas and, for, local, par, recursive, repeat, skip, to, until, como nuevas alternativas en la especificación de Token. **Eliminar** la palabra reservada begin. En los identificadores, las mayúsculas son significativas y distintas de las minúsculas.



#### Contexto: identificación y tipos


Usaremos las reglas sintácticas del lenguaje  extendido para indicar las restricciones de contexto. Interprete las reglas sintácticas desde una perspectiva contextual, esto es, vea en ellas *árboles de sintaxis abstracta* y no secuencias de símbolos. En lo que sigue usaremos las siguientes *meta-variables* para hacer referencia a árboles sintácticos de las clases indicadas:

 $V$ : V-name	 $PF_1, PF_2, PF_i, PF_n$ : Proc-Func
 $Exp, Exp_1, Exp_2, Exp_3, Exp_i$ : Expression	 $PFs$ : Proc-Func*
 $Com, Com_1, Com_2$ : Command	 $FPS$ : Formal-Parameter-Sequence
 $Dec, Dec_1, Dec_2, Dec_3$ : Declaration	 $TD$ : Type-denoter
 $Id$ : Identifier	

El comando skip no requiere de revisión contextual, pues ya 'está bien'.

Considere el comando de asignación<sup>6</sup>,  $V := Exp$ . Las restricciones contextuales son:

-   $V$  debe haber sido declarada como una *variable*.
-   $Exp$  debe tener el mismo tipo que se declaró o se infirió para esa variable.

Considere estos comandos repetitivos añadidos a 

```
repeat while Exp do Com end
repeat until Exp do Com end
repeat do Com while Exp end
repeat do Com until Exp end
```

Las restricciones contextuales son:

<sup>5</sup> Estamos manteniendo la otra regla donde aparece var.

<sup>6</sup> Esta es una asignación. No la confunda con la *declaración* de variable inicializada.

- [E4] *Exp* debe ser de tipo Boolean.
- [E4] *Com* y sus partes deben satisfacer las restricciones contextuales<sup>7</sup>.

En el comando de repetición controlada por contador

```
for var Id := Exp1 to Exp2 do Com end
```

las restricciones son:

- [E4] *Exp*<sub>1</sub> y *Exp*<sub>2</sub> deben ser ambas de tipo *entero*. Los tipos de *Exp*<sub>1</sub> y *Exp*<sub>2</sub> deben ser determinado en el contexto en el que aparece *este* comando *for\_var\_:=\_to\_do\_end*.
- [E4] *Id* es conocida como la “variable de control”. *Id* es de tipo entero. *Id* es **declarada** en este comando y su alcance es *Com*; *esta* declaración de *Id* **no** es conocida por *Exp*<sub>1</sub> **ni** por *Exp*<sub>2</sub>.
- [E4] *Com* debe cumplir con las restricciones contextuales.
- [E4] La variable de control *Id* **no** puede aparecer a la izquierda de una asignación ni pasarse como parámetro *var*<sup>8</sup> en la invocación de un procedimiento o función dentro del cuerpo del comando repetitivo *for\_var\_:=\_to\_do\_end*<sup>9</sup>.
- [E4] Este comando repetitivo *funciona como un bloque* al declarar una variable local (la variable de control). Las reglas usuales de anidamiento aplican aquí (si se re-declara la variable en un comando o bloque anidado, esta es distinta y hace inaccesible la variable del *for\_var\_:=\_to\_do\_end*, que es más externa).

En el comando de repetición controlada por contador y condición (en sus dos variantes):

```
for var Id := Exp1 to Exp2 while Exp3 do Com end
```

```
for var Id := Exp1 to Exp2 until Exp3 do Com end
```

las restricciones son:

- [E4] *Exp*<sub>3</sub> debe ser de tipo Boolean.
- [E4] *Exp*<sub>3</sub> **sí** está en el alcance de *Id* (al igual que *Com*).
- [E4] Aplican también aquí todas las demás restricciones especificadas para *for var Id := Exp<sub>1</sub> to Exp<sub>2</sub> do Com end*

El condicional de [E4] fue modificado.

```
if Exp then Com1 else Com2 end
```

Las restricciones contextuales son:

- [E4] *Exp* debe ser de tipo Boolean.
- [E4] *Com*<sub>1</sub> y *Com*<sub>2</sub>, así como sus partes, deben satisfacer las restricciones contextuales.

El comando

```
let Declaration in Command end
```

se analiza contextualmente de manera idéntica al comando correspondiente en [E4] original (el original no tiene *end*, observe que ahora permitimos *Command* en lugar del *single-Command* original).

Los parámetros de una abstracción (función o procedimiento) forman parte de un nivel léxico (alcance) inmediatamente más profundo que aquel en el que aparece el identificador de la abstracción que los declara<sup>10</sup>. *Usted debe verificar que ningún nombre de parámetro se repita dentro de la declaración de una función o procedimiento (en el encabezado).*

En una declaración de variable inicializada:

```
var Id := Exp
```

<sup>7</sup> Observe que ésta, como muchas de las restricciones contextuales, se formulan de manera recursiva.

<sup>8</sup> No podrá pasarse por referencia.

<sup>9</sup> Es decir, en el cuerpo de este comando repetitivo (*Com*), la variable de control en realidad se comporta como una *constante*.

<sup>10</sup> Es decir, los parámetros se comportan como identificadores declarados localmente en el bloque de la función o procedimiento. Repase lo expuesto en los libros de Watt y Brown.

El tipo de la *variable* inicializada *Id* debe deducirse a partir del tipo de la expresión inicializadora *Exp*<sup>11</sup>. *Id* se exporta al resto del bloque donde aparece esta declaración.


En una declaración local *Dec<sub>1</sub> in Dec<sub>2</sub> end*, los identificadores declarados en *Dec<sub>1</sub>* son conocidos exclusivamente en *Dec<sub>2</sub>*. Únicamente se "exportan" los identificadores declarados en *Dec<sub>2</sub>*. Observe que tanto *Dec<sub>1</sub>* como *Dec<sub>2</sub>* son declaraciones generales (Declaration).


En una declaración par *Dec<sub>1</sub> and Dec<sub>2</sub> end*, el identificador declarado en *Dec<sub>1</sub>* no es conocido por *Dec<sub>2</sub>*, ni viceversa. Además, *Dec<sub>1</sub>* y *Dec<sub>2</sub>* deben declarar identificadores distintos<sup>12</sup>. Observe que todas las declaraciones paralelas son simples (single-Declaration). Se "exportan" todos los identificadores declarados en *Dec<sub>1</sub>* y *Dec<sub>2</sub>*. Es posible declarar paralelamente dos o más declaraciones simples; lo descrito antes corresponde al caso de **dos** declaraciones y *debe ser generalizado a más de dos declaraciones*: *par Dec<sub>1</sub> and ... and Dec<sub>n</sub> end (n ≥ 2)*.

En una declaración recursive *PFs end* se permite combinar las declaraciones de varios procedimientos o funciones, de manera que puedan invocarse unos a otros (para posibilitar la recursión mutua).

- Consideremos primero el caso de *dos* declaraciones: en *recursive PF<sub>1</sub> and PF<sub>2</sub> end* se declaran funciones y/o procedimientos mutuamente recursivos: el identificador<sup>13</sup> declarado en *PF<sub>1</sub>* es conocido por *PF<sub>2</sub>*, y viceversa.
- *PF<sub>1</sub>* y *PF<sub>2</sub>* deben declarar identificadores (de función o procedimiento) *distintos*.
- Se "exportan"<sup>14</sup> los identificadores de función o de procedimiento declarados en *PF<sub>1</sub>* y *PF<sub>2</sub>*. Los *parámetros* declarados en un encabezado de función o de procedimiento son *locales* y, por lo tanto, no se "exportan".
- En el caso general, es posible declarar dos o más procedimientos o funciones como mutuamente recursivos. Sintácticamente: *recursive PF<sub>1</sub> and PF<sub>2</sub> and ... and PF<sub>n</sub> end*. Todos los identificadores introducidos por las declaraciones *PF<sub>i</sub>* (1 ≤ *i* ≤ *n*) deben ser *distintos* y son conocidos al procesar los cuerpos de *todas* las funciones o procedimientos declarados en las *PF<sub>i</sub>*.
- Es un error declarar más de una vez el mismo identificador<sup>15</sup> en las declaraciones simples que componen una misma declaración compuesta recursive.
- Todos los identificadores de procedimiento o función declarados vía recursive son "exportados"; esto es, son agregados al contexto y son conocidos después de la declaración compuesta recursive<sup>16</sup>.
- Funciones o procedimientos *distintos* declarados vía recursive pueden declarar *parámetros* con nombres idénticos, eso *no* es problema.

## Proceso y salidas

Ud. modificará el procesador de  en Java para que sea capaz de procesar la extensión especificada arriba.

- Debe incluir el analizador de léxico completo (reconocimiento de lexemas, categorización de lexemas en clases léxicas apropiadas, registro de coordenadas del lexema).
- Debe modificar el analizador sintáctico de manera que logre reconocer el lenguaje  extendido completo y construya los árboles de sintaxis abstracta correspondientes a las estructuras de las frases reconocidas<sup>17</sup>.
- 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<sup>18</sup>.
- 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 del IDE, con estructuras de navegación semejantes a las que ya aparecen en la implementación de base que se les ha dado<sup>19</sup>.

<sup>11</sup> El procesamiento es semejante al que se hace para la declaración *const Id ~ Exp*. La diferencia es que, en el caso que nos ocupa, se está declarando a *Id* como una *variable*, no como una *constante*.

<sup>12</sup> Es un error declarar el mismo identificador en varias declaraciones que componen una declaración *par* (paralela o colateral).

<sup>13</sup> Que da nombre a la función o al procedimiento.

<sup>14</sup> Después de la declaración recursive los identificadores podrán ser utilizados.

<sup>15</sup> De función o procedimiento.

<sup>16</sup> Esto no sucede cuando recursive aparece dentro de una declaración private (antes del in).

<sup>17</sup> Cada una de las variantes del comando repeat y el for deben dar lugar a una *forma distinta* de árbol de sintaxis abstracta. Esto facilitará el análisis contextual y la generación de código en el futuro.

<sup>18</sup> En el IDE suministrado, se sincroniza esta información de manera que es visible en la ventana del código fuente. Lea bien el código para que comprenda la manera en que interactúan las partes y se logra este efecto.

<sup>19</sup> Este 'TreeView' permite contraer o expandir los subárboles. Es conveniente usar el patrón 'visitante' ('visitor') para este propósito.

- El analizador contextual debe realizar completamente el trabajo de identificación (manejo del alcance en la relación entre ocurrencias de definición y de aplicación de identificadores) y realizar la comprobación de tipos sobre el lenguaje extendido; debe reportar la posición de *cada uno* de los errores contextuales detectados (esto es, avisa de *todos* los errores contextuales encontrados en el proceso).
- Las técnicas por utilizar son las expuestas en clase y en los libros de Watt y Brown; en particular nos interesa que el analizador contextual deje el árbol sintáctico decorado apropiadamente para la fase de generación de código subsiguiente (Proyecto #2)<sup>20</sup>. Esto es particularmente delicado para el procesamiento de las variantes del comando `for` y las nuevas formas de declaración compuesta.

Como se indicó, ustedes deben basarse en los programas que se le dan como punto de partida. Su programación debe ser consistente con el estilo aplicado en el procesador en Java usado como base, y ser respetuosa de ese estilo. En el código fuente debe estar claro dónde ha introducido Ud. modificaciones.

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

**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 pueda someter a pruebas su programa sin dificultades.*

## Documentación

Debe documentar clara y concisamente los siguientes puntos<sup>21</sup>:

- Su esquema para el manejo del texto fuente (si *no* modifica lo existente, *indíquelo explícitamente*).
- Modificaciones hechas al analizador de léxico (tokens, tipos, métodos, etc.).
- Documentar los cambios hechos a los *tokens* y a algunas estructuras de datos (por ejemplo, tabla de palabras reservadas) para incorporar las extensiones al lenguaje.
- Documentar cualquier cambio realizado a las reglas sintácticas de extendido, para lograr que tenga una gramática LL(1) equivalente a la indicada para la extensión descrita arriba. Justifique cada cambio explícitamente.
- Nuevas rutinas de reconocimiento sintáctico, así como cualquier modificación a las existentes.
- Lista de errores sintácticos detectados.
- Modelaje realizado para los árboles sintácticos (ponga atención al modelaje de categorías sintácticas donde hay ítemes repetidos<sup>22</sup>).
- Extensión realizada a los procedimientos o métodos que permiten visualizar los árboles sintácticos abstractos (desplegarlos en una pestaña del IDE).
- Descripción de la comprobación de tipos para todas las variantes de `repeat ... end`, `for ... end`.
- Solución dada al manejo de alcance, del tipo y de la protección de la variable de control del `for ... end`, incluyendo la variante con `while` o `until`.
- Describir el procesamiento de la declaración de variable inicializada (`var Id := Exp`).
- Descripción de su validación de la unicidad<sup>23</sup> de los nombres de parámetros en las declaraciones de funciones o procedimientos.
- Descripción de la solución dada al procesamiento de la declaración compuesta `par`. Interesa la no-repetición de los identificadores declarados.
- Descripción de la solución dada al procesamiento de la declaración compuesta `local`. Interesa que la primera declaración introduzca identificadores que son conocidos *privadamente* por la segunda declaración; se exporta solamente lo introducido por la segunda declaración.
- Descripción de la solución dada al procesamiento de la declaración compuesta `recursive`. Interesa la no-repetición de los identificadores (de función o procedimiento) declarados y que estos identificadores sean













<sup>20</sup> Esto comprende introducir información de tipos en los árboles sintácticos correspondientes a expresiones, así como dejar “amarradas” las ocurrencias aplicadas de identificadores (poner referencias hacia el subárbol sintáctico donde aparece la ocurrencia de definición correspondiente), vía la tabla de identificación. También se determina si un identificador corresponde a una variable, etc.

<sup>21</sup> Nada en la documentación es opcional. La documentación tiene un peso importante en la calificación del proyecto.

<sup>22</sup> En particular, es importante que decida si los ítemes 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 original para inspirarse.

<sup>23</sup> I.e. no repetición.

conocidos en los cuerpos de las funciones o procedimientos declarados en una misma declaración compuesta *recursive*.

-  Nuevas rutinas de análisis contextual, así como cualquier modificación a las existentes.
-  Lista de errores contextuales detectados.
-  Plan de pruebas para validar el compilador. Debe separar las pruebas para cada fase de análisis (léxico, sintáctico, contextual). Debe incluir pruebas *positivas* (para confirmar funcionalidad con datos correctos) y pruebas *negativas* (para evidenciar la capacidad del compilador para detectar errores). Debe especificar lo siguiente para cada caso de prueba:
  - Objetivo del caso de prueba
  - Diseño del caso de prueba
  - Resultados esperados
  - Resultados observados
-  Análisis de la 'cobertura' del plan de pruebas (interesa que valide tanto el funcionamiento "normal" como la capacidad de detectar errores léxicos, sintácticos y contextuales).
-  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.
-  Indicar cómo debe compilarse su programa.
-  Indicar cómo debe ejecutarse su programa.
-  Archivos con el texto fuente del programa. El texto fuente debe incluir comentarios que indiquen con claridad los puntos en los cuales se han hecho modificaciones.
-  Archivos con el código objeto del programa. **El compilador debe estar en un formato ejecutable directamente desde el sistema operativo Windows.**
-  Debe enviar su trabajo en un archivo comprimido (formato **.zip**) según se indica abajo. 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.**
  - Programas de prueba que han preparado.

## Entrega

**Fecha límite:** lunes 23 de octubre antes de las 12 medianoche. No se recibirán trabajos después de la fecha indicada.

Debe enviar un archivo comprimido con todos los elementos de su solución a la dirección [itrejos@itcr.ac.cr](mailto:itrejos@itcr.ac.cr).

El asunto (subject) debe ser:

"IC-5701 - Proyecto 1 - " <carnet> " + " <carnet> " + " <carnet> " + " <carnet>".

Los carnets deben ir ordenados ascendentemente.





Los grupos pueden ser de *hasta 4* personas.

Si su correo 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 (caso de que obviara su correo por no tener el asunto apropiado). Si su correo no es legible (por cualquier motivo), o contiene un virus, la nota será 0.

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 producidos por estudiantes universitarios de tercer año de la carrera de Ingeniería en Computación del Instituto Tecnológico de Costa Rica.

## Integración con el IDE de Luis Leopoldo Pérez

*Gracias a Christian Dávila Amador, graduado del ITCR, por su colaboración.*

1. Seguir las instrucciones ubicadas dentro de la documentación del IDE (**ide-triangle.pdf, pág. 6**).
2. 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**);
3. **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 ahí.

### **Defecto conocido en el IDE de Luis Leopoldo Pérez**

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

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.