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:

- 2018.04.05: Versión base.
- 2018.04.10: Versión revisada.

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

Objetivo

Al concluir este proyecto Ud. habrá terminado de comprender los detalles relativos a las fases de análisis léxico y sintáctico de un compilador escrito "a mano" usando las técnicas expuestas por Watt y Brown en sus libro *Programming Language Processors in Java*. Ud. deberá extender el compilador del lenguaje Δ 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 Δ extendido. 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.

Base

Ud. usará como base el compilador del lenguaje Δ 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 Dr. Luis Leopoldo Pérez (implementado en Java) y corregido por los estudiantes Pablo Navarro y Jimmy Fallas. *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 2018. ¡Lea el apéndice D del libro de Watt y Brown 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

El lenguaje fuente es una extensión del lenguaje Δ, 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 mucho cuidado a los cambios que estamos aplicando sobre Δ**. El lenguaje Δ original es descrito en el apéndice B del libro de Watt y Brown.

Esta extensión de Δ añade varias formas de comando iterativo, un nuevo comando de selección, declaración de variables inicializadas, declaración de procedimientos o funciones mutuamente recursivos y declaraciones locales.

Sintaxis

Convenciones sintácticas

- **[x]** *equivale a* (**x | ε**), es decir, **x** *aparece cero o una vez.*

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

- **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  ::=
                  |   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
```

Note que está vacío. Es decir, es **ε**

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

Eliminar de `single-Command` la primera alternativa (comando vacío)[2].

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
```

Añadir a `single-Command` lo siguiente[3]:

```
          "nothing"
        | "loop" "while" Expression "do" Command "end"
        | "loop" "until" Expression "do" Command "end"
        | "loop" "do" Command "while" Expression "end"
        | "loop" "do" Command "until" Expression "end"
        | "loop" "for" Identifier ":=" Expression "to" Expression "do" Command "end"
        | "let" Declaration "in" Command "end"
        | "if" Expression "then" Command ("elsif" 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):

---

[2] Observe que en la regla original aparece blanco a la derecha de `::=`. Ahora tenemos una palabra reservada para designar el comando vacío (**nothing**). Esto le obliga a modificar `parseSingleCommand` en el compilador de base.
[3] Recuerde que `single-Command` y `Command` son no-terminales (categorías sintácticas) distintos. No factorizamos las reglas para hacer evidentes las diferencias entre las nuevas formas de comando.

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

Añadir estas reglas[4]:

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

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

Considere la regla `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
```

Allí, **modificar** la opción referente a **`proc`** para que se lea:

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

A `single-Declaration`, **añadir** lo siguiente (variable inicializada)[5]:

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

Añadir a Type-denoter esta regla[6]:

| "**array**" Integer-Literal ".." Integer-Literal "**of**" Type-denoter

Cambios léxicos

- **Añadir** las *palabras reservadas* **`and`**, **`for`**, **`loop`**, **`nothing`**, **`private`**, **`rec`**, **`to`**, **`until`**, como nuevas alternativas en la especificación de `Token`.
- **Eliminar** la palabra reservada **`begin`**.
- Al igual que en Δ, en los identificadores las mayúsculas son significativas y distintas de las minúsculas.

Proceso y salidas

Ud. modificará el procesador de Δ escrito 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 Δ extendido completo y construya los árboles de sintaxis abstracta correspondientes a las estructuras de las frases reconocidas[7].

---

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

[5] Estamos manteniendo la otra regla donde aparece **`var`**.

[6] Estamos manteniendo la otra regla donde aparece **`array`**.

[7] Cada una de las variantes del comando **`loop`** debe dar lugar a una *forma distinta* de árbol de sintaxis abstracta. Esto facilitará el análisis contextual y la generación de código en el futuro.

- 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[8].
- 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[9].
- Las técnicas por utilizar son las expuestas en clase y en los libros de Watt y Brown.

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 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 su asistente puedan someter a pruebas su programa sin dificultades.*

Documentación
Debe documentar clara y concisamente los siguientes puntos :

Analizador sintáctico y léxico
- 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 Δ 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[10]).
- Extensión realizada a los procedimientos o métodos que permiten visualizar los árboles sintácticos abstractos (desplegarlos en una pestaña del IDE).
- Plan de pruebas para validar el compilador. Debe separar las pruebas para cada fase de análisis (léxico, sintáctico). 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 y sintácticos).
- Discusión y análisis de los resultados obtenidos. Conclusiones a partir de esto.

---

[8] 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.
[9] Este 'TreeViewer' permite contraer o expandir los subárboles. Es conveniente usar el patrón 'visitante' ('visitor') para este propósito.
[10] 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.

- 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 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 (.tri) de prueba que han preparado.

Entrega
Fecha límite: miércoles 2018.04.25 antes de las 12 medianoche. No se recibirán trabajos después de la fecha y hora indicadas.

Debe enviar por correo-e un archivo comprimido con todos los elementos de su solución a la dirección 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 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 (caso de que obviara su mensaje por no tener el asunto apropiado). Si su mensaje 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 Tecnológico de Costa Rica.

Integración con el IDE de Luis Leopoldo Pérez
Gracias a Christian Dávila Amador, graduado del TEC, 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.

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.