Universidad Simón Bolívar
Departamento de Computación y Tecnología de la Información
CI3725 – Traductores e Interpretadores
Abril – Julio 2012

Etapa I
Análisis Lexicográfico

Esta primera etapa del proyecto corresponde al módulo de análisis lexicográfico del interpretador del lenguaje AsGArD que queremos construir. Específicamente, se desea que Ud. implemente el referido módulo utilizando una herramienta generadora de analizadores lexicográficos de las permitidas (mencionadas más adelante), y que además haga una breve revisión de los conceptos y métodos teórico-prácticos relevantes.

El analizador lexicográfico a ser construido en esta etapa deberá aceptar como entrada cualquier secuencia de caracteres y dar como salida los *tokens* relevantes reconocidos. Si se recibe caracteres que no corresponden a ningún *token* del lenguaje AsGArD, se debe dar un mensaje de error.

Los *tokens* relevantes serán:

- Cada una de las palabras claves utilizadas en la sintaxis de AsGArD, i.e. `using`, `repeat`, `boolean`, `if`, `of type`, etc. En este caso, los tokens deberán llamarse *"Tk⟨Palabra Clave⟩"*, donde ⟨*Palabra Clave*⟩ es la palabra clave a la que representa el token, con su primera letra en mayúscula. Por ejemplo, para la palabra clave `using`, su token sería `TkUsing`.

- Los identificadores de variables. A diferencia de las palabras claves, los identificadores corresponderán a un único *token* llamado `TkIdent`. Este *token* siempre tendrá asociado como atributo el identificador particular reconocido. Por ejemplo, al leer `contador`, se dará como salida `TkIdent("contador")`.

- Los literales numéricos, los cuales serán secuencias no-vacías de dígitos decimales. De manera análoga a los identificadores, éstos serán agrupados bajo el *token* `TkNum`. Este *token* tendrá como atributo al número reconocido, por ejemplo `TkNum(3000)`.

- Los literales booleanos, los cuales a diferencia de los literales numéricos, estarán representados por dos *tokens*. Uno de ellos para `true` (`TkTrue`) y otro para `false` (`TkFalse`).

- Los literales lienzos, los cuales serán uno de los siguientes: `<empty>`, `</>`, `<\>`, `<|>`, `<_>`, `<->` o `< >`. Todos estos serán representados por el token `TkLienzo`, parametrizado por el contenido envuelto entre los sìmbolos < y >. Por ejemplo, el literal de lienzo `</>` será representado por `TkLienzo("/")`.

- Cada uno de los símbolos que denotan separadores en AsGArD, los cuales se presentan a continuación:

  `","` – `TkComa`
  `";"` – `TkPuntoYComa`
  `"("` – `TkParAbre`
  `")"` – `TkParCierra`

- Cada uno de los símbolos que denotan a operadores aritméticos, booleanos, relacionales, o de lienzos en AsGArD, los cuales se presentan a continuación:

```
 "+"  —  TkSuma
 "-"  —  TkResta
 "*"  —  TkMult
 "/"  —  TkDiv
 "%"  —  TkMod
"/\"  —  TkConjuncion
"\/"  —  TkDisyuncion
 "^"  —  TkNegacion
 "<"  —  TkMenor
"<="  —  TkMenorIgual
 ">"  —  TkMayor
">="  —  TkMayorIgual
 "="  —  TkIgual
"/="  —  TkDesIgual
 ":"  —  TkHorConcat
 "|"  —  TkVerConcat
 "$"  —  TkRot
 "'"  —  TkTras
```

- La operación de asignación `":="`, la cual será representada por el *token* `TkAsignacion`.

Los espacios en blanco, tabuladores y saltos de línea deben ser ignorados. Así mismo, los comentarios no deben ser reconocidos como *tokens*, sino ser ignorados igualmente. Cualquier otro caracter será reportado como error.

A continuación algunos ejemplos, donde el texto ***inclinado*** corresponde a la secuencia de caracteres de entrada que recibe el analizador lexicográfico y el `resto` corresponde a la secuencia de *tokens* de salida:

```
using contador of type integer begin
  {- Asignar al contador
  el valor 35.  -}
  contador := 35
end
```

```
TkUsing TkIdent("contador") TkOfType TkInteger TkBegin
TkIdent("contador") TkAsignacion TkNum(35) TkEnd
```

Como puede verse, la región del programa que está encerrada entre los símbolos "`{-`" y "`-}`" y que por ende son comentarios en AsGArD, fueron ignorados completamente al presentar la salida.

Nótese que el analizador lexicográfico sólo puede ser capaz de reconocer secuencias <u>arbitrarias</u> de *tokens*, aunque tales secuencias sean sintácticamente incorrectas. Por ejemplo:

```
end
  contador using integer of type
  {- Asignar al 35
  el valor contador? -}
  35 := contador
begin
```

```
TKEnd TkIdent("contador") TkUsing TkInteger TkOfType
TkNum(35) TkAsignacion TkIdent("contador") TkBegin
```

Los <u>únicos</u> errores detectables a nivel lexicográfico corresponden a caracteres irrelevantes. Por ejemplo:

```
using contador! of type integer begin
  contador ?:= 35
end
Error:  Caracter inesperado "!" en la fila 1, columna 15
Error:  Caracter inesperado "?" en la fila 2, columna 12
```

Su analizador lexicográfico debe reportar *todos* los errores léxicos, en caso de haberlos. Cuando algún error es encontrado, los *tokens* se hacen irrelevantes (ya que no corresponde a un programa correcto), por lo que no deberán ser mostrados. Los errores deben mostrarse con el mismo formato en que se mostró en el ejemplo anterior.

Es importante notar que cada *token* producido por su analizador lexicográfico debe corresponder a un tipo de datos y no simplemente ser impreso a la salida estándar. Esto a modo de poder suministrarlos luego como información para el analizador sintáctico (a implementar en la segunda parte de este proyecto). De esta forma, usted deberá implementar un programa principal que invoque al analizador lexicográfico y sea luego el que imprima a la salida estándar la representación como texto de los *tokens* encontrados.

Tal como se indicó inicialmente, esta primera etapa consta tanto de la implementación del analizador lexicográfico utilizando alguno de los lenguajes y herramientas permitidas, como de una breve revisión teórico-práctica. A continuación se explica cuáles son los lenguajes y herramientas permitidas, los detalles de la entrega de la implementación y finalmente, en qué consiste la revisión teórico-práctica.

Lenguajes y Herramientas Permitidos Para la implementación de este proyecto se cuenta para escoger con cinco (5) lenguajes de programación. Estos son:

- *C++*: Lenguaje imperativo, orientado a objetos. Es recomendable utilizar versiones de *g++* compatibles con el estándar *C++03*, dependiendo también de los requisitos de las herramientas a utilizar. Es posible conseguir *C++* usando el manejador de paquetes, en sistemas Unix (por ejemplo, el programa nativo `apt-get`, en sistemas Debian). Para los usuarios de sistemas Windows, se aconseja utilizar el compilador *mingw*, incluido en el ambiente de desarrollo `Dev-Cpp`, disponible en la siguiente dirección Web: (http://www.bloodshed.net/devcpp.html).

  Para *C++*, la herramienta generadora de analizadores lexicográficos a utilizar se llama *Flex* y puede ser encontrada en la siguiente dirección Web: (http://flex.sourceforge.net/).

- *Java:* Lenguaje imperativo, orientado a objetos. Es recomendable utilizar versiones de *Java* iguales o posteriores a la 1.5, dependiendo también de los requisitos de las herramientas a utilizar. Es posible conseguir *Java* desde la siguiente dirección Web:
  (http://www.java.com/es/download/).

  Para *Java*, la herramienta generadora de analizadores lexicográficos a utilizar se llama *JFlex* y puede ser encontrada en la siguiente dirección Web: (http://jflex.de/download.html).

- *Ruby:* Lenguaje de `scripting`, orientado a objetos. Es posible conseguir *Ruby* desde la siguiente dirección Web: (http://www.ruby-lang.org/en/downloads/). En el caso particular de *Ruby* no hay una buena herramienta generadora de analizadores lexicográficos, por lo que el trabajo deberá hacerse manualmente a través de las expresiones regulares que provee el lenguaje.

- *Python:* Lenguaje de `scripting`, orientado a objetos. Es posible conseguir *Python* desde la siguiente dirección Web: (http://www.python.org/download/).

  Para *Python* la herramienta generadora de analizadores lexicográficos a utilizar es a la vez la misma que genera analizadores sintácticos. Por lo tanto, deberán manejarse algunas nociones de gramáticas antes de tiempo para poder trabajar con la misma. Esta herramienta se llama *PLY* y puede ser encontrada desde la siguiente dirección Web: (http://www.dabeaz.com/ply/).

- *Haskell:* Lenguaje funcional puro. Es posible conseguir *Haskell* desde la siguiente dirección Web: (http://www.haskell.org/ghc/download.html/). Si decide utilizar *Haskell*, su implementación deberá ser compatible con el compilador proporcionado (*GHC*).

  Para *Haskell* la herramienta generadora de analizadores lexicográficos a utilizar se llama *Alex* y puede ser encontrada en la siguiente dirección Web: (http://www.haskell.org/alex/).

Entrega de la Implementación Ud. debe entregar:

- Un correo electrónico con el código fuente en el lenguaje y la herramienta de su elección, entre los permitidos, de su analizador lexicográfico. Todo el código debe estar debidamente documentado. El analizador deberá ser ejecutado con el comando "`./LexAsgard`", por lo que es posible que tenga que incorporar un script a su entrega que permita que la llamada a su programa se realice de esta forma. Note que la entrada para su programa será a través de la entrada estándar del sistema de operación. Sin embargo, deben abstenerse de imprimir nada más que lo pedido y tener cuidado con que la salida del programa corresponda con fidelidad a los requisitos mencionados anteriormente.

- Un *breve* informe explicando la formulación/implementación de su analizador lexicográfico y justificando todo aquello que Ud. considere necesario. Además el informe debe contener sus respuestas a la revisión teórico–práctica, la cual será explicada más adelante.

Nota: Es importante que su código pueda ejecutarse en las máquinas del LDC, pues es ahí y únicamente ahí donde se realizará su corrección.

Revisión Teórico-Práctica Al informe de la implementación, Ud. deberá agregar el desarrollo de las preguntas que se listan en esta sección.

Para el desarrollo de estas preguntas, utilice el algoritmo de la Sección 7.4.2 (página 252) del libro [WM95]. Nos referiremos a este algoritmo con el nombre *A*.

1. Dé tres expresiones regulares *E1*, *E2* y *E3* que correspondan respectivamente al reconocimiento de la palabra clave `to`, de la palabra clave `true` y de identificadores.

2. Dé los diagramas de transición (i.e. representación gráfica) de tres autómatas finitos (posiblemente no-determinísticos) *M1*, *M2* y *M3* que reconozcan respectivamente a los lenguajes denotados por *E1*, *E2* y *E3*. Esto corresponde al paso (1) del algoritmo *A*, con *Ri'* y *Ri* refiriéndose a nuestras *Ei*. (*Nota:* La relación entre las expresiones *Ri'* y *Ri* está especificada en la Sección 7.3.2 (páginas 249-250) del mismo libro [WM95]; sin embargo, ésta no es relevante para nuestros propósitos.)

3. Construya un autómata finito no-determinístico *M* que reconozca a la unión de los lenguajes *L(M1)*, *L(M2)* y *L(M3)*, tal como se indica en el paso (2) de *A*.

4. Note que, a efectos de implementar un analizador lexicográfico, es importante que el autómata *M* sepa reportar a cuál de los tres lenguajes pertenece cada palabra que él reconozca. Esto significa que *M* debe poder identificar a cuál de los tres lenguajes corresponde cada estado final. Indique a qué lenguaje corresponde cada uno de los estados finales de su autómata *M*.

5. La asignación de estados finales a lenguajes de su respuesta a la pregunta 4 debe crear conflictos, pues hay elementos que pertenecen a más de uno de los tres lenguajes *L(M1)*, *L(M2)* y *L(M3)*. Cada conflicto corresponde a una palabra w que pertenece a más de un lenguaje, digamos Lx y Ly, la cual tendrá, por lo tanto, más de una "vía de reconocimiento" en M. Estas vías deben terminar en estados finales asociados a los distintos lenguajes correspondientes, i.e. Lx y Ly. Indique cuáles son los conflictos de su autómata M, especificando las palabras que los generan, y los lenguajes y estados finales involucrados.

6. Construya un autómata finito determinístico *Mdet* equivalente a M, i.e. que reconozca el mismo lenguaje que M (viz. $L(M1) \cup L(M2) \cup L(M3)$). Esto corresponde al paso (3) de A.

7. ¿Cómo se reflejan los conflictos de su respuesta a la pregunta 5 en su autómata *Mdet*?

8. Los conflictos deben ser resueltos mediante un orden lineal que priorice a los lenguajes involucrados. En nuestro caso, establecemos que el orden de prioridad viene dado por la secuencia $\langle L(E1), L(E2), L(E3) \rangle$ de manera decreciente, i.e. el primer lenguaje tiene más prioridad que los otros dos y el segundo más que el tercero. De acuerdo a esto, asocie un lenguaje (solo uno) a cada estado final de *Mdet* tal como lo hizo en la pregunta 4 para M.

9. Tal como se indica en el paso (4) de A, construya un autómata finito determinístico mínimo *MinM* equivalente a M (y, por tanto, también equivalente a *Mdet*). Explique por qué se debe usar la partición inicial de estados especificada en el paso (4) de A, e indique a cuál de los tres lenguajes corresponde cada estado final de *MinM*.

10. ¿Cómo relaciona Ud. el desarrollo de las preguntas 1–9 a la implementación de su analizador lexicográfico construido con la herramienta escogida?

Nota Importante: El algoritmo A (mencionado en las preguntas anteriores) difiere en algunos puntos con la forma en que se manejan los conceptos en otros libros importantes utilizado en el curso, por ejemplo [S97]. Como ejemplo de tales diferencias, las transiciones de los autómatas no-determinísticos son vistos como relaciones, mientras que en clase se verán como funciones que devuelven conjuntos. Sin embargo, teniendo en cuenta estos detalles, es posible transportar la información que se presenta en el algoritmo A para que sea consistente con lo acostumbrado en clase.

Referencia Bibliográfica

[WM95] R. Wilhelm & D. Maurer. *Compiler Design.* Addison-Wesley, 1995.

[S97] T. Sudkamp. *Languages and Machines.* Second Edition. Addison-Wesley, 1997.

Fecha de Entrega: Domingo, 13 de Mayo (Antes de semana 4), hasta las 11:00 pm.
Dirección de Entrega: Deberá entregar su proyecto tanto por Aula Virtual como por medio de correo electónico a los cuatro encargados del curso.
Valor: 6%.

C.A. Perez. R. Monascal, C. Gómez, M. Gómez, H. González y J. A. Goncalves / Abril 2012