Práctica 2: Introducción a los analizadores léxicos.

Entrega 1. Marzo 2018

1 Motivación

Las secuencias arbitrarias de caracteres en el código, no tienen sentido para un compilador, es por que ello que se necesita agrupar caracteres en átomos de categorías sintácticas definidas por los diseñadores del lenguaje. Este es justamente el trabajo de un analizador léxico.

2 Implementación de analizadores léxicos

Existes típicamente tres maneras de implementar un analizador léxico; *ad-hoc*, *dirigido por el código* y *dirigido por tablas*. Esta última será en la que se pondrá mayor énfasis ya que será usada posteriormente. Todas ellas parten de la idea de reconocer presencias de expresiones regulares en un texto dado.

2.1 Ad-hoc

Esta implementación está hecha especialmente para un conjunto de expresiones regulares y puede ser tan compleja o sencilla como el programador la decida hacer. Con mucha pericia esta implementación puede llegar a ser la más eficiente de todas y puede implementar otras funcionalidades extras durante su ejecución.

2.2 Dirigido por el código

El código refleja el comportamiento de un autómata finito determinsta que reconoce el conjunto de todas las expresiones regulares.

2.3 Dirigido por tablas

Para la implementación de un analizador dirigido por tablas, necesitamos:

1. Un autómata. Este autómata debe ser un modelo reconocedor de todo el conjunto de expresiones regulares que necesitemos identificar. Para hacer eficiente el análisis este autómata debe ser determinita y debe estar minimizado. El autómata es representado a través de su tabla de transiciones.

2. Un motor. Este motor es una pieza de código que dada cualquier secuencia de caracteres y cualquier tabla (de transiciones) pueda obtener caracteres de la entrada y determinar cuál será la siguiente transición.

Si ya hemos podido implementar un analizador léxico para un determinado conjunto de expresiones, construir uno para otro conjunto se resume en obtener el autómata mínimo determinista que sea reconocedor de las expresiones y dejar que el motor haga su trabajo. De lo anterior observamos que este tipo de implementación nos da gran flexibilidad.

3 Maxima Mordida

Pensemos en el siguiente escenario:

En el programa hay un identificador llamado `fortuna`, los posibles comportamientos del analizador podrían ser:

1. Reconocer el identificador `fortuna`.

2. Reconocer el identificador `fortuna` y además la palabra reservada `for`.

3. Reconocer la palabra reservada `for` y un identificador `tuna`.

Evidentemente el comportamiento deseado es el primero, independientemente de la implementación. Al reconocimiento de la expresión que abarca el mayor número de caracteres consecutivos se le denomina *máxima mordida*. Implementar este comportamiento de manera ingenua puede tener un costo cuadrático, algo muy elevado si tomamos en cuenta que hay programas con miles de líneas y que ésta es a penas la primera fase en la compilación. El artículo `‘‘Maximal-Munch’’ Tokenization in Linear Time` explica los casos en los que el reconocimiento tiene costo cuadrático y propone una modificación al algoritmo ingenuo de reconocimiento dirigido por tablas para que ante cualquier entrada la ejecución sea lineal.

4 Jflex [1]

Jflex no es un analizador léxico, es un generador de analizadores léxicos dirigidos por tablas y con comportamiento de máxima mordida. El código resultante está escrito en *Java*. Su funcionamiento a grandes rasgos es el siguiente:

1. Recibe un conjunto de expresiones regulares.
2. Construye de un autómata no determista que reconozca todas esas expresiones regulares.
3. Contruye del autómata determinista
4. Minimiza el autómata.
5. Regresa el analizador léxico para el conjunto de expresiones regulares dado.

4.1 Estructura del archivo

La estructura de la entrada para que *Jflex* pueda generar el analizador léxico consta de tres partes separadas por `%%` .

1. Primera parte. En ésta se incluyen todas la bibliotecas de *Java* de las que haremos uso.
2. *Jflex* provee varias opciones que modifican o extienden su funcionamiento, se mecionarán a continuación las más comunes:
   - `%debug` imprime en la salida estándar información de depuración del reconocimiento léxico.
   - `%standalone` incluye un método `main` en la clase del analizador léxico que permite usarlo como componente independiente.
   - `%class <nombre clase>` especifica el nombre de la clase. Por omisión el nombre es `Yylex`.
   - `%unicode` da soporte al ese conjunto de caracteres.
   - `%line` provee a la clase de una variable llamada `yyline` mediante la cuál se puede tener acceso a la línea del código que se está analizando.

[1] `http://jflex.de/`

En esta sección además de las opciones también se puede escribir:

- Código *Java* de funciones auxiliares escrito entre **%{** y **}%**
- Asignaciones de nombres a patrones comunes o complicados para que posteriormente pueda hacerse referencia a ellos mediante un nombre. Ejemplo:

  ```
  DIGITO  =   [0-9]
  ```

  Cuándo se requiere hacer uso la declaración, debe escribirse entre `{` y `}`.

3. En esta sección se escriben los patrones (reglas) y las acciones que se requiere llevar a cabo cuándo se reconozca esa regla. El formato es el siguiente:

   ```
   regla {acción}
   ```

   La sintáxis completa para escribir las reglas puede consultarse en la documentación oficial de *Jflex*. Las acciones son instrucciones en *Java* y pueden hacer uso de las variables o funciones provistas por *Jflex* como:

   - `yyline` ya se mencionó anteriormente.
   - `yytext(void)` regresa la cadena que empató con la regla correspondiente a esa acción.
   - `yypushback(int num)` regresa al flujo de entrada `num` caracteres de la cadena que empató con esa regla. `num` no puede ser mayor a la longitud de la cadena completa.

   Todo los caracteres que no empaten con ninguna reglan serán imprimidos en la salida estándar.

4.2 Ejemplo

A continuación se construirá un analizador léxico que reconoce números enteros y reales.

```
// AL.flex
%%
%public
%standalone
```

```
%unicode
%debug

PUNTO = \.
CERO               =         0+
ENTERO =  [1-9][0-9]* | 0+
%%
{ENTERO}        { System.out.print("ENTERO("+yytext() + ")"); }
{ENTERO}? {PUNTO} {ENTERO} | {ENTERO} {PUNTO} {ENTERO}?
                { System.out.print("REAL(" + yytext() +  ")" ); }
.               { }
```

Teniendo como entrada el siguiente archivo:

```
# numeros
123.122.21212
211.ccvvvv
.123
123
texto
```

Después *Jflex* se encargará de crear el analizador léxico.

```
$ jflex AL.flex
```

Después se compila la clase resultante y se prueba con el archivo `numeros`

```
$ javac Yylex.java
$ java Yylex numeros
```

5 Ejercicios

5.1 Maxima Mordida

Lee el artículo `‘‘Maximal-Munch’’ Tokenization in Linear Time` e implementa un analizador léxico dirigido por tablas que reconozca la cadenas formadas por las siguientes expresiones regulares: *{ab, (ab)* c}*. Debe tener desempeño lineal ante todas las entradas y su funcionamiento debe seguir el principio de *mordida máxima*. La implementación deberá estar hecha bajo un paradigma estructurado u orientado a objetos únicamente. Si la cadena no está formada con átomos de las expresiones regulares mencionadas, deberá indicar que está mal formada y detendrá su funcionamiento.

La entrada puede ser desde un archivo o de la entrada estándar. A continuación un ejemplo del funcionamiento del analizador:

```
//entrada.txt
abababababcab
abababababbba

$ ./atablas entrada.txt
cadena: abababababcab
Ok: [(AB)*C][AB]
cadena: abababababbba
Error: cadena mal formada
```

5.2 Uso de Flex (clase)

Extiende el ejemplo proporcionado para que se genere un analizador léxico para:

- enteros.
- números flotantes.
- identificadores. (No debe reconocer palabras reservadas como identificadores)
- Comentarios

Los lenguajes a escoger son:

- Python
- Java
- C
- Haskell
- Pascal

Un lenguaje no debe ser escogido por más de dos equios.

Asigna el nombre **Analizador[Lenguaje]** a la clase resultante. Es decir, si escogiste Python como lenguaje, tu clase resultante se llamará **AnalizadorPython**.

Crea un archivo **Readme** en el que se especifique el lenguaje que escogiste y las fuentes que documenten las expresiones regulares de tus categorías léxicas.