Práctica de lenguajes funcionales y compiladores
Lenguaje de programación FUN

Revisión 2~~Revisión 1~~ jfcmjfcm

Juan Francisco Cardona McCormick

7 de Octubre de 2015

1. Introducción

Los lenguajes funcionales son otro tipo de paradigma de programación que aunque fueron diseñados desde los años 50 del siglo pasado, muchas de sus características están siendo implementadas en muchos lenguajes de programación que son muy populares hoy en día: como el caso de Java, C#, Ruby, Python, C++.

Esta práctica tiene dos objetivos: el primero mostrar como luce un lenguaje funciona simple, esto se hará a través de la sintaxis; el segundo propósito es implementar partes de un compilador para este lenguaje a través de varias prácticas que llevaremos a cabo durante el curso.

2. Gramáticas del lenguaje fun

2.1. Gramática concreta de FUN

La siguiente es la definición de la gramática concreta del lenguaje de programación FUN:

$$
\begin{array}{rcl}
\textit{Expr} & \rightarrow & \text{'}\textbf{fn}\text{' } \texttt{ID } \prime{=}{>}\prime \textit{ Expr} \\
& | & \text{'}\textbf{fun}\text{' } \texttt{ID ID } \prime{=}{>}\prime \textit{ Expr} \\
& | & \textit{Expr Expr} \\
& | & \text{'}\textbf{if}\text{' } \textit{Expr } \text{'}\textbf{then}\text{' } \textit{Expr } \text{'}\textbf{else}\text{' } \textit{Expr} \\
& | & \text{'}\textbf{let}\text{' } \texttt{ID } \prime{=}\prime \textit{ Expr } \text{'}\textbf{in}\text{' } \textit{Expr} \\
& | & \textit{Expr Op Expr} \\
& | & \texttt{ID} \\
& | & \textit{Decimal} \\
& | & \textit{Booleano} \\
& | & \textit{Real} \\
& | & \text{'(' } \textit{Expr } \text{')'} \\
\textit{Decimal} & \rightarrow & \texttt{NÚMERO} \\
& | & \texttt{LITOCTAL} \\
& | & \texttt{LITHEXA} \\
\textit{Booleano} & \rightarrow & \text{'}\textbf{True}\text{'} \\
& | & \text{'}\textbf{False}\text{'} \\
\textit{Real} & \rightarrow & \texttt{FLOTANTE} \\
\textit{Op} & \rightarrow & \prime{<}\prime \mid \prime{>}\prime \mid \prime{=}{=}\prime \mid \prime{!}{=}\prime \mid \prime{<}{=}\prime \mid \prime{>}\underline{\leq}^{\text{jfcm}}{=}\prime \mid \prime\underline{=}^{\text{jfcm}}\prime \\
& | & \prime{+}\prime \mid \prime{-}\prime \mid \prime{*}\underline{\times}^{\text{jfcm}}\prime \mid \prime{/}\prime \mid \prime\%\prime \\
& | & \prime|\prime \mid \prime||\prime \mid \prime\&\prime \mid \prime\&\&\prime \mid \prime\wedge\prime \mid
\end{array}
$$

El lenguaje está compuesto de no-terminales: *Expr*; de terminales: '**fun**' o $\prime{+}\prime$; y de grupos de terminales: `ID`. Los grupos de terminales representa expresiones regulares.

2.2. Lexemas

Los lexemas son aquellos elementos que constituyen un programa concreto en el lenguaje de programación FUN.

2.2.1. Palabras reservadas

La siguiente es la lista de las palabras servadas que cuenta el lenguaje:

`fn`	`fun`	`if`	`then`	`then`	`else`
`let`	`in`	~~`True`~~$^{\text{jfcm}}$	~~`False`~~$^{\text{jfcm}}$		

2.2.2. Identificadores

Los identificadores sirven para reconocer el nombre de funciones, variables, etc. En FUN los nombres son reconocidos por `ID`: todo `ID` debe empezar con un letra y puede ser seguido de cualquier secuencia de letras, dígitos o guiones bajos (_).

2.2.3. Enteros

Los enteros son cualquier constante entera que no inicie con cero excepto el cero mismo y son aceptadas tres:

- Literales enteras,
- Literales octales: comienza con cero y seguido de un dígito entre uno y siete y secuencias de dígitos entre 0 y 7~~son seguidos de secuencias de dígitos entre 0 y 7~~[jfcm].
- Literales hexadecimales: comienza con `0x` o `0X` y son secuencia de dígitos entre 0 y 7 y `a` y `f` o `A` y `F`.

2.2.4. Flotantes

Representa números reales, son similares a las enteros pero tiene un punto decimal siempre en ellas, la siguiente es una lista de valores válidos:

- `1.0`
- `22423.01`
- `0.000000001`

No son válidos algunos similares a:

- `1.`
- `.0`

2.2.5. Booleanos

Los valores booleanos están representados por los siguientes literales:

- `True`
- `False`

2.2.6. Separadores

Los separadores son:

- (,)
- <eof>

3. Primera entrega

La primera entrega consiste en implementar el análizador léxico para el lenguaje de programación FUN. Este será implementado utilizando el generado de analizadores lexicográficos llamado flex y ANTLR[1]. Este analizará uno o más ficheros y generara una salida en la que describen los lexemas (*tokens*).

3.1. Formato de salida

La salida de cada analizador lexicográfico sigue el siguiente formato definido por la siguiente gramática:

Salida	→	ListaFicheros `EOF`
ListaFicheros	→	ElementoFichero ListaFicheros
	|	ElementoFichero
ElementoFichero	→	TipoFichero `EOL` ListaLexemas
TipoFichero	→	`fichero: NOMBREFICHERO`
	|	`fichero: entrada estandar`
ListaLexemas	→	ErrorLexema
	|	ϵ
	|	ElementoLexema
	|	ElementoLexema ListaLexemas
ErrorLexema	→	`error: CADENA fila: INT col: INT EOL`
ElementoLexema	→	`tipo: LEXEMA valor: CADENA fila: INT col: INT EOL`
Lexema	→	3operator^{jfcm} | 1keyword^{jfcm} | 0identifier^{jfcm} | 2integer^{jfcm}
	|	4separator^{jfcm} | 6^{jfcm} | 5^{jfcm}

Donde: `NOMBREFICHERO` es un nombre de fichero válido (esto permite que también incluya la ruta del fichero que puede ser relativa o absoluta); `EOF` es fin de archivo; `CADENA` es una cadena de caracteres imprimibles rodeada de comillas dobles o un número entero que representa un carácter no imprimible o $<eof>^{jfcm}$; `INT` es una secuencia de digitos no vacía; `EOL` es el fin de línea acorde a la plataforma[2]: en linux línea nueva, en windows retorno de carro y línea nueva; `LEXEMA` es uno de los siguiente tipos de lexemas:

- 0 Identificador.
- 1 Palabra reservada.
- 2 Entero.
- 3 Operador.
- 4 Separador.

[1] Versión 4
[2] Linux, UNIX y otros sistemas compatibles UNIX utilizan el mismo formato.

- 5 Booleano.
- 6 Real.

3.2. Errores

Al encontrar un error el programa termina inmediatamente.

4. Requerimientos

4.1. Técnicos

Lenguaje de programación: El lenguaje de programación Java [1].

Control de versiones: Se va utilizar el manejador de versiones *subversion* [3].

Repositorio de control de versiones: RiouxsvnAssemblajfcm (`https://www.riouxsvn.com`).

Estructura de directorios: La estructura de directorio es creada utilizando `maven`.

```
$ mvn archetype:generate -DgroupId=co.edu.eafit.dis.st0270.p2015<grpname>
      -DartifactId=fun -DarchetypeArtifactId=maven-archetype-quickstart
      -DinteractiveMode=false
```

Donde el `grpname` es el nombre del grupo de trabajo, los miembros del equipo escogen un nombre de grupo.

Y a partir de este comando se obtiene el siguiente directorio[3]:

```
+-- fun
    +-- miembros.xml
    +-- pom.xml
    +-- src
        +-- main
        |   +-- antlr4
        |       +-- <grpname>AntlrLexer.g4
        |   +-- jflex
        |       +-- <grpname>JFlexLexer.jflex
        |   +-- java
        |       +-- co
        |           +-- Main<grpname>Lexer.java
        |           +-- edu
        |               +-- eafit
        |                   +-- dis
```

[3]Realmente se obtiene una estructura diferente pero se deben hacer algunos cambios

```
|                          +-- st0270
|                              +-- p2015<grpname>
|                                  +-- lexer
|                                      +-- <grpname>AntlrLexer.java
|                                      +-- <grpname>JFlexLexer.java
|                                  +-- tokens
|                                  +-- helpers
```

`pom.xml` fichero de descripción del proyecto para maven.

`miembros.xml` fichero que contiene la definición de los miembros del grupo. El siguiente es el contenido de un posible grupo en java:

```
<grupo>
   <nombre>docente</nombre>
   <url>https://riouxsvn.com/svn/doc20151</url>
   <miembros>
      <miembro>
         <nombre>Juan Francisco Cardona McCormick</nombre>
         <codigo>1986100250010</codigo>
         <email>fcardona@eafit.edu.co</email>
      </miembros jfcm>
      <miembro>
         <nombre>Jonathan Eidelman Manevich</nombre>
         <codigo>201310029010</codigo>
         <email>jeidelma@eafit.edu.co</email>
      </miembro>
   </miembros>jfcm
   <proyecto>
     <java>
        <mainclases>
          <mainclass srcdir="src/main/java" package="co" clase="Main<nombregrupo>Lexer"/>
        </mainclases>
     </java>
   </proyecto>
</grupo>
```

El `url` identifica el repositorio donde está almacenada la práctica. El nombre del grupo es de una sola palabra, en minúsculas. Se debe añadir el correo electrónico en la segunda entrega. `mainclases` es una lista con las clases principales del proyecto tanto para ANTRL.

`<grpname>AntlrLexer.g4` fichero que contiene la descripción de analizador léxico para el generador de analizadores ANTLR4. (Esta descripción de la gramática es base tanto para la generación automática como la generación manual).

`<grpname>JFlexLexer.jflex` fichero que contiene la descripción de analizador léxico para el generador de analizadores JFlex. (Esta descripción de la gramática es base tanto para la generación automática como la generación manual).

`src` directorio donde residen los fuentes de los archivos de clases de java y los archivos generados por las descripciones de las gramáticas.

`Main<nombregrupo>Lexer.java` nombre de la clase que contiene el punto de entrada al analizador léxico para ANTRL y JFlex[jfcm]. Puede ser ejecutado directamente desde la terminal:

```
$ java co.Main<nombregrupo>Antlr[jfcm]Lexer {A|F|B} [{ficherofun} ... | - ]
```

Donde el primer argumento indica que si se utiliza el lexer generado por Antlr ó jFlex ó amBos; los restantes argumentos pueden ser un conjunto de ficheros de lenguajes fun[4] o que lean los datos de la entrada estándar.

4.2. Administrativos

Grupo de trabajo: Grupo máximo de dos estudiantes. *Cada* miembro del equipo debe registrarse en riousvn (`https://www.riouxsvn.com`) con un usuario propio. Uno de los miembros debe crear un repositorio donde los miembros del equipo son los dueños del repositorio. Deben enviar una invitación al monitor de la materia y al profesor de la misma.

Primera entrega: Lunes 26 de Octubre a las 9:00 am~~Sábado 24 de Octubre a las 23:59 pm~~[jfcm]. Esta se hará automática a través del sistema de control de versiones. Se *debe* cumplir todos los requerimientos señalados en este documento. Recuerde que la idea es entregar una gramática en el archivo de las gramáticas.

Pruebas: Se van a probar ambos analizadores con un conjunto de 20 pruebas cada una consta de una gramática que tiene diferentes carácterísticas. 12 de las pruebas van a ser públicas, las otras 8 se harán durante la sustentación, todas ellas son gramáticas que pueden ser válidas o inválidas.

Nota primera entrega: Se calcula con la siguiente formula.

$$Nota\ entrega = Sustentación \times (5 \times \frac{(\sum_{i=1}^{20} prueba_i)}{20})$$

Donde $Sustentación$ es valor obtenido en la sustentación: $0 \leq Sustentación \leq 1$. ~~*Gramática LL*(1) indica si la gramática cumple con la condición *LL*(1): $0 \leq$ *Gramática LL*(1) ≤ 1~~[jfcm]. $prueba_i$ indica si la prueba pasó o falló: $0 \leq prueba_i \leq 1$ si una prueba pasó o falló.

Control de versiones: El control de versiones no es solamente un herramienta que facilite la comunicación entre los miembros del grupo y del control de versiones, sino que también ayudará al profesor a llevar un control sobre el desarrollo de la práctica. Se espera que las diferentes registros dentro del control de versiones sean cambios graduales. En caso contrario, se procederá a realizar un escutrinio a fondo del manejo de control de versiones para evitar fraudes.

[4] Estos puede ser válidos o inválidos lexicograficamente o válidos o inválidos sintácticamente

Sustentación: Durante la semana de 26 de Octubre a 30 Octubre se hará la sustentación. El Domingo 25 de Octubre se enviará los horarios de sustentación durante la semana.

Bibliografía

[1] Arnold, Ken. Gosling, James. The Java Programming Language. 2005.

[2] Crespi Reghizzi, Stefano. Formal Languages and Compilation. (2008). 83–87.

[3] Pilato, Michael C. Collins-Sussman, Ben. Fitzpatrick, Brian W. Version Control with Subversion. 2008.

[4] Scott, Michael C. Programming Language Pragmatics. 2006.