Lenguaje de Dominio Específico CURSOR

Cristan Andrione
Programación Profesional
Analista Universitario en Sistemas
Instituo Politécnico "General San Martín"
Universidad Nacional de Rosario Profesor: Juan Manuel Rabasedas

26 de febrero de 2021

Resumen

Se presenta un `DSL`, para realizar procesamiento de texto, al que bautizamos con el nombre `Cursor`. Aprovecharemos, para este cometido, la potencia del editor de texto `Vim`, en particular de su lenguaje de srcipting `Vimscript`, un lenguaje de programación con todas las de la ley: sistema de tipos, variables, funciones, bucles, condicionales, listas, programación funcional, expresiones regulares, etc. Este lenguaje es muy potente y provee una granularidad muy fina a la hora de realizar operaciones sobre textos. El precio que pagamos por ello es una elevada curva de apredinzaje.

Considerando esto último como un obstáculo para su uso y aprendizaje por parte de quienes no son usuarios habituales de `Vim`, proponemos un `DSL` con menor potencia pero mas sencillo.

1. Dominio

El dominio de nuestro `DSL` es el procesamiento de texto. Dado un texto tendremos posibilidad de indicar una serie de operaciones sobre él mediante la sintaxis de nuestro `DSL` que será traducida a `Vimscript` para ejecutarse sobre el texto a tratar.

2. DSL

Se presenta un lenguaje imperativo simple con variables y comandos para asignación, composición secuencial, ejecución condicional (`if`) y ciclos (`while`)
Un ejemplo de la sintaxis:

```
// nos paramos al comienzo del archivo
origen;
```

```
// recorremos el archivo línea a línea desde la 1 hasta la última
// en cada línea preguntamos si contiene la substring "hola"
// si es true (distinto de -1 en este caso) se sustituye
// la string "hola" por la string "CHAU"
while
curLine < totLines
since
if
    (subStr "hola") > -1
then
    // imprime "si1" en linea de comandos de vim
    // mayormente para depuración
    echo "si1";
    sust hola HOLA;
    finalStr ___cambió_HOLA___
else
    // imprime "no1" en linea de comandos de vim
    echo "no1"
endif;
fLine
endwhile;

// volver cursor a la primera línea, primera columna
origen;

// recorremos el archivo línea a línea desde la 1 hasta la 10
// en cada línea preguntamos si contiene la substring "coma"
// si es true (distinto de -1 en este caso) se sustituye
// la string "coma" por la string "COMA"
a = 10;
while
curLine < 10
since
if
    (subStr "coma") > -1
then
    sust coma COMA;
    echo "si2";
    finalStr ___cambió_COMA___
else
    echo "no2";
    skip
endif;
fLine
endwhile;

// vamos a última línea
final;
```

```
// recorremos el archivo línea a línea desde la última hasta la 10
// sin incluir, desde abajo hacia arriba,
// en cada línea preguntamos si contiene la substring "auto"
// si es true (distinto de -1 en este caso) se sustituye
// la string "hola" por la string "AUTO"
b = 9;
c = 1 + b;
while
curLine > c
since
if
    (subStr "auto") > -1
then
    sust auto AUTO;
    echo "si3";
    finalStr ___cambió_AUTO___
else
    skip;
    echo "no3"
endif;
rLine
endwhile;

// nos paramos en linea 15
goToLine 15;

// recorremos el archivo línea a línea desde la 15 hasta la última
// en cada línea preguntamos si contiene la substring "lunes"
// si es true (distinto de -1 en este caso) se sustituye
// la string "lunes" por la string "LUNES"
while
curLine < totLines
since
if
    (subStr "lunes") > -1
then
    // imprime "si1" en linea de comandos de vim
    // mayormente para depuración
    echo "si4";
    sust lunes LUNES;
    finalStr ___cambió_LUNES___
else
    // imprime "no1" en linea de comandos de vim
    echo "no4"
endif;
fLine
endwhile;

// acedemos a cualquier commando de vim directamente
// excom "let x = 1005";
```

```
// borramos linea 25
delLine 25;

// asignamos algunas strings
x = "hola";
y = " que tal!";
u = x ++ y;
w = ("hola " ++ "que ") ++ "tal!";
q = "hola " ++ ("que " ++ "tal!");

// muestra en linea de commandos el valor de la variables definidas más arriba
echo u;
echo w;
echo q;

// vemos la strnig termina OK en línea de comandos
// si el programa llega al final llega al final
echo "termina OK"
```

El indentado es solo a los efectos de hacer mas legible el programa por parte de las personas. Los comentarios indican lo que va haciendo el programa. El compilador nos dará la siguiente salida en un archivo: `command.vim`

```
call cursor( 1, 1)
while line('.') < line('$')
if stridx(getline('.'), 'hola') >  -1
echom 'si1'
execute "s/hola/HOLA/g"
execute "normal! A___cambió_HOLA___\<ESC>"
else
echom 'no1'
endif
call cursor( line('.') + 1, 1)
endwhile
call cursor( 1, 1)
let a = 10
while line('.') < 10
if stridx(getline('.'), 'coma') >  -1
execute "s/coma/COMA/g"
echom 'si2'
execute "normal! A___cambió_COMA___\<ESC>"
else
echom 'no2'

endif
call cursor( line('.') + 1, 1)
endwhile
call cursor( line('$'), 1)
let b = 9
```

```
let c = 1 + b
while line('.') > c
if stridx(getline('.'), 'auto') >  -1
execute "s/auto/AUTO/g"
echom 'si3'
execute "normal! A___cambió_AUTO___\<ESC>"
else
echom 'no3'
endif
call cursor( line('.') - 1, 1)
endwhile
call cursor(15, 1)
while line('.') < line('$')
if stridx(getline('.'), 'lunes') >  -1
echom 'si4'
execute "s/lunes/LUNES/g"
execute "normal! A___cambió_LUNES___\<ESC>"
else
echom 'no4'
endif
call cursor( line('.') + 1, 1)
endwhile
call cursor(25, 1)
execute "normal! dd"
let x = 'hola'
let y = ' que tal!'
let u = x . y
let w = 'hola ' . 'que ' . 'tal!'
let q = 'hola ' . 'que ' . 'tal!'
echom u
echom w
echom q
echom 'termina OK'
```

Por último se ejecuta la secuencia de operaciones:

`vim -n -N -u NONE -S comand.vim achivo_a_procesar.txt`

La salida es el mismo archivo. El archivo no necesariamente debe estar escrito con `Vim`.

2.1. Variables, Tipos y Asiganciones

La variables podrán tener los tipos `INTEGER`, `STRING` o `BOOLEAN`. La asignación de un valor a una variables de realiza mediante el signo `=`, a la izquierda encontramos el identificador de la variable y a la derecha la expresión que generará el valor a asignar.

En el momento de la asignación se infiere el tipo de la variable.
`VARIABLE = EXPRESION`;

2.2. Expresiones, Operandos y Operadores

Las expresiones son una combinación de operandos y operadores.

2.2.1. Operandos

Pueden ser `LITERALES` o `VARIABLES`:

2.2.2. Operadores, Precedencia de las operaciones

Pueden ser `Aritméticos`, `Lógicos` o `de cadena de texto`:

Operadores Aritméticos: Ellos son suma, resta, multiplicación y división:

```
+ (SUM)
- (RES)
* (MUL)
/ (DIV)
```

Operadores Lógicos: Tenemos tres operadores lógicos:

```
& (Y)
| (O)
! (NEG)
```

Operadores de relación: Tenemos cuatro operadores de relación:

```
>(MAYOR)
<(MENOR)
== (IGUAL)
```

Operadores de cadena de texto: Solo hay un operador de cadenas y es para concatenación:

```
++ (CONCATENAR)
```

La precedencia de las operaciones es la siguiente (de mayor a menor):

```
                    asociatividad
 -(UNARIO) !
 * /            izquierda a derecha
 + -            izquierda a derecha
 < >            izquierda a derecha
 ==             izquierda a derecha
 & |            izquierda a derecha
```

```
=               derecha a izquierda
```

Los paréntesis pueden modificar estas reglas.

2.3. Comandos

Secuenciación: Los commandos deben separarse mediante punto y coma, pero el último de la lista no lleva signo de puntuación.

```
comm; comm
```

Condicional

```
if
    boolexpr
then
    comm
else
    comm
endif
```

Bucle

```
while
    boolexpr
since
    comm
until
```

sentencia vacía

```
skip
```

2.4. Funciones Built-in

Un concepto importante en nuestro `DSL` es el de `cursor`, que indica en qué lugar del texto nos encontramos ubicados. El `cursor` es un par de números enteros `(LINE, COLUMN)` que actúa como coordenada indicando número de línea (numeradas desde la línea 1 de arriba hacia abajo) y número de columna (numeradas desde 1 de izquierda a derecha) en que se halla el cursor; denominamos a esta: *línea actual*. El `DSL` provee algunas funciones (*built-in*), que hacen uso de estos conceptos, ellas son:

`curLine` devuelve un INTEGER que indica el número de línea actual archivo de entrada.

`curCol` devuelve un INTEGER que indica el número de columna de la línea actual.

`totLine` devuelve un INTEGER que indica el número de la última línea, es decir el número total de líneas del archivo de entrada.

`substring(src, line)` Si la string `src` se encuentra en la línea número `line` devuelve un INTEGER distinto de -1, de lo contrario devuelve el INTEGER -1.

`getCurrentLine` devuelve una `string` que se corresponde con la `línea actual`.

`getLastLine` devuelve una `sting` que se corresponde con la última línea del archivo.

`fLine` avanza una línea el cursor.

`rLine` retrocede una línea el cursor.

`sust (src, dst)` reemplaza todas las ocurrencias de la cadena `src` por la cadena `dst` en la linea actual, si S`src` no existe en la línea actual, no hace nada.

`origen` envía el cursor a la posición (1,1).

`final` envía el cursor a la posición (1, totLinea).

`gotoLine intExp` envía el cursor a la línea `intExp`.

`delLine intExp` borrar la línea `intExp`.

`delCurLine intExp` borrar la línea actual.

`addStr strtExp` agrega la string `strExp` la posición actual del cursor.

`beginStr strExp` agrega la string `strExp` al comienzo de la línea actual.

`finalStr strExp` agrega la string `strExp` al final de la línea actual.

3. Instalación (Linux)

Para la instalación descargamos los fuentes del repositorio *GitHub* `https://github.com/Cristian133/algoritmia` ya sea comprimido en formato zip desde la web o podemos clonarlo directamente en nuestro directorio de trabajo con el siguiente commando:

```
$ git clone https://github.com/Cristian133/algoritmia
```

Si no tenemos git instalado (en distribuciones derivadas de `Debian`):

```
$ sudo apt install git
```

Luego debemos ingresar al directorio `cursor` dentro de la carpeta `algoritmia`:

```
$ cd algoritmia/cursor
```

En esta carpeta nos encontramos los siguientes archivos:

- Los archivos principales escritos en haskell: `Main.hs`, `AST.hs`, `Parser.hs`, `Transp.hs`.
- El archivo de resaltado de sintaxis: `cursor.vim`.
- El archivo `input.txt` contiene el texto a procesar, a su vez será archivo de salida, ya que en él se graba el resultado global del proceso.
- El archivo `command.vim` es el script generado por nuestro DSL y que tomará vim para procesar el texto dado en `input.txt`. (Este archivo podría no estar ya que se genera durante la ejecución)
- El archivo `cursor` es un script bash que ejecutará el archivo resultado de la compilación con dos argumentos: el archivo extensión `.cur` con el programa y el archivo de entrada `input.txt`. Éste nos da el resultado final.
- Un archivo `makefile` que se encarga de la compilación y limpieza posterior.

Ejecutar en consola el comando:

```
$ make
```

Si todo está bien nos dará la siguiente salida:

```
> ghc -o Main Main.hs
> [1 of 4] Compiling AST              ( AST.hs, AST.o )
> [2 of 4] Compiling Parser           ( Parser.hs, Parser.o )
> [3 of 4] Compiling Transp           ( Transp.hs, Transp.o )
> [4 of 4] Compiling Main             ( Main.hs, Main.o )
> Linking Main ...
```

Indicando que la compilación fue exitosa!!!.
Opcional: Limpiar los archivos innecesarios (producto de la compilación `.o .hi`); para ello:

```
$ make clean
```

Ejecutar el DSL propiamente dicho:

```
$ . ./cursor prueba.cur input.txt
```

(Los símbolos `. ./` antes del nombre del archivo son necesarios porque nuestra carpeta de trabajo no está en el PATH). Esto nos abre el vim con el texto cambiado según las instrucciones del programa.

4. Desarrollo del trabajo y decisiones de diseño

Durante las clases de la materia trabajamos sobre un lenguaje imperativo simple llamado `LIS`. Mi idea para desarrollar el lenguage `cursor` fue basarme en el lenguaje anterior. No hubo mayor inconveniente para escribir el `AST`. Los problemas comenzaron con el `parser` ya que las librerías `haskell` que usamos, `parsec`, no son sencillas Aquí tuve que parar con la escritura de código y estudiar con detalle esta librería, para luego

retomar (2 semanas después) la escritura del `parser`. El parser funcionó bien, cada cosa estaba en su lugar en nuestro frondoso árbol. Creo que la decisión de diseño más importante la tomé en la intancia posterior: el temido `evaluador`!. Yo intentaba, como lo hacía el lenguaje `LIS`, mantener el `estado` de la variables del programa para ir resolviendo las expresiones a medida que dicho `estado` cambiaba, pero el problema eran las funciones `built-in`. Tomemos como ejemplo a la función `curLine`; la cual nos devuelve un INTEGER que representa el número de la línea actual. Por lo tanto esta función debe poder formar parte de una expresión entera. por ejemplo:

```
var = (1 + 1) * 5 + curLine
```

El problema es que el valor de `curLine` no puede ser conocido en tiempo de compilación, ya que durante la misma no se conoce el archivo de entrada. Por lo tanto después de probar varias opciones (entre ellas una resolución parcial de expresiones que me deparó varías decepciones) decidí hacer una traducción y no resolver nada, sino que resuelva `vim`. Entonces ya no se hizo necesario mantener un `estado`. El trabajo ahora se trataba de construir un AST, para luego a partir de él construir expresiones en la sintaxis de `vim`. Esta tarea la realiza el programa haskell `Transp.hs`.
Sabemos que haskell es un lenguaje interpretado y también puede compilarse. La idea fue que el usuario final no tenga que vérselas con el interprete `ghci` sino que use directamente los ejecutables compilados, de una forma amigable. Entonces se realizó un archivo `makefile` para automatizar la compilación de los fuentes y la limpieza de los archivo innecesarios y por último un archivo `bash` que ejecuta todo lo necesario.

5. Análisis del Lenguage

5.1. Sintaxis Concreta

La sintaxis concreta de un lenguaje incluye todas las caracter¿¿sticas que se observan en un progama fuente, como delimitadores y paréntesis.

```
var     ::= letter | letter var

intexp  ::= nat
        |   var
        |   intexp '+' intexp
        |   intexp '-' intexp
        |   intexp '*' intexp
        |   intexp '/' intexp
        |   'curLine'
        |   'curCol'
        |   'totLines'
        |   'subStr' strexp
        |   '(' intexp ')'

boolexp ::= 'true' | 'false'
            var
```

```
        |   intexp '==' intexp
        |   intexp '<' intexp
        |   intexp '>' intexp
        |   boolexp '&' boolexpr
        |   boolexp '|' boolexpr
        |   '!' boolexpr
        |   '(' boolexpr ')'

strexp  ::= '"' letter '"'
        |   var
        |   letter strexp
        |   strexp '.' strexp
        |   'getCurrentLine'
        |   'getLastLine'

comm    ::= 'skip'
        |   var '=' intexp | strexp | boolExp
        |   comm ';' comm
        |   'if' boolexp 'then' comm 'else' comm 'endif'
        |   'while' boolexpr 'since' comm 'endwhile'
        |   'origen'
        |   'goToLine
        |   'delLine'
        |   'delCurLine'
        |   'final'
        |   'addStr' strexp
        |   'beginStr' strexp
        |   'finalStr' strexp
        |   'sust' strexp strexp
```