# Sintaxis del DSL CURSOR

Cristian Andrione
Programación Profesional
Analista Universitario en Sistemas
Instituo Politécnico "General San Martín"
Universidad Nacional de Rosario

12 de enero de 2021

**Resumen**

Se propone 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 apredizaje 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

Un ejemplo de la sintaxis:

```
WHILE lineaActual() < totalLineas()
     SINCE
        IF
            SUBSTRING("hola", lineaActual())
        THEN
            REEMPLAZAR ("hola", "chau!");
```

```
            AGREGAR(";", finLinea());
        ELSE
            SKIP;
        ENDIF
        avanzarLinea(1);
     UNTIL
```

El indentado es solo a los efectos de hacer mas legible el programa por parte de las personas. Este programa recorre desde la primera hasta la última línea del texto, cuando encuentra la cadena `'hola'` la reemplaza por `'chau'` y agrega un punto y coma al final de la línea. El compilador nos dará la siguiente salida en un archivo: command.vim

```
while line('.') < line('$')
    call cursor( line('.')+1, 1)
    if stridx(getline('.'), 'hola') > -1
        execute "%s/'hola'/'chau'"
        execute "normal! A;\<ESC>"
    endif
endwhile
```

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

```
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:
`CLINE()` devuelve un INTEGER que indica el número de línea actual del archivo de entrada.
`LLINE()` devuelve un INTEGER que indica el número de la última línea del archivo de entrada.
`CURSOR(LINE, COL)` acepta dos parámetros INTEGER que indican número de línea y número de columna, su acción es colocar el cursor en la posición `(LINE, COL)`.
`SUBSTRING(SRC, LINE)` Si la string SRC se encuentra en la línea número LINE devuelve el INTEGER 1, de lo contrario devuelve el INTEGER -1.
`REPLACE (SRC, DST)` reemplaza todas las ocurrencias de la cadena SRC por la cadena DST en la linea actual, si SRC no existe en la línea actual, no hace nada y tampoco falla dando error.
`ADD(SRC)` agrega SRC en la posición del cursor hacia la derecha
`LEN(SRC)` devuelve un INTEGER representando la longitud de la cadena SRC

## 2.5. Sintaxis Concreta

```
var     ::= letter | letter var

intexp  ::= nat
        |   var
        |   intexp '+' intexp
        |   intexp '-' intexp
        |   intexp '*' intexp
        |   intexp '/' intexp
        |   '(' intexp ')'

boolexp ::= 'TRUE' | 'FALSE'
        |   intexp '==' intexp
        |   intexp '<' intexp
        |   intexp '>' intexp
        |   boolexp '&' boolexpr
```

```
        |   boolexp '|' boolexpr
        |   '!' boolexpr
        |   '(' boolexpr ')'

strexp  ::= '"' letter '"' | letter strexp
        |   strexp '.' strexp

comm    ::= 'SKIP'
        |   var '=' intexp | strexp
        |   comm ';' comm
        |   'IF' boolexp 'THEN' comm 'ELSE' comm 'END'
        |   'WHILE' boolexpr 'SINCE' comm 'UNTIL'
```

## 2.6. Semántica Operacional Big-Step para Expresiones

$$\frac{}{\langle nv, \sigma\rangle \Downarrow_{intexp} nv}\ \textsc{NVal} \tag{1}$$

$$\frac{}{\langle nv, \sigma\rangle \Downarrow_{intexp} nv}\ \textsc{Var} \tag{2}$$

$$\frac{\langle e, \sigma\rangle \Downarrow_{intexp} n}{\langle -_u e, \sigma\rangle \Downarrow_{intexp} -n}\ \textsc{UMinus} \tag{3}$$

$$\frac{\langle e_0, \sigma\rangle \Downarrow_{intexp} n_0 \quad \langle e_1, \sigma\rangle \Downarrow_{intexp} n_1}{\langle e_0 + e_1, \sigma\rangle \Downarrow_{intexp} n_0 + n_1}\ \textsc{Plus} \tag{4}$$

$$\frac{\langle e_0, \sigma\rangle \Downarrow_{intexp} n_0 \quad \langle e_1, \sigma\rangle \Downarrow_{intexp} n_1}{\langle e_0 -_b e_1, \sigma\rangle \Downarrow_{intexp} n_0 -_b n_1}\ \textsc{BMinus} \tag{5}$$

$$\frac{\langle e_0, \sigma\rangle \Downarrow_{intexp} n_0 \quad \langle e_1, \sigma\rangle \Downarrow_{intexp} n_1}{\langle e_0 \times e_1, \sigma\rangle \Downarrow_{intexp} n_0 \times n_1}\ \textsc{Times} \tag{6}$$

$$\frac{\langle e_0, \sigma\rangle \Downarrow_{intexp} n_0 \quad \langle e_1, \sigma\rangle \Downarrow_{intexp} n_1 \quad n_1 \neq 0}{\langle e_0 \div e_1, \sigma\rangle \Downarrow_{intexp} n_0 \div n_1}\ \textsc{Div} \tag{7}$$

$$\frac{\langle e_0, \sigma\rangle \Downarrow_{intexp} n_0 \quad \langle e_1, \sigma\rangle \Downarrow_{intexp} n_1}{\langle e_0 = e_1, \sigma\rangle \Downarrow_{intexp} n_0 = n_1}\ \textsc{Eq} \tag{8}$$

$$\frac{\langle e_0, \sigma\rangle \Downarrow_{intexp} n_0 \quad \langle e_1, \sigma\rangle \Downarrow_{intexp} n_1}{\langle e_0 < e_1, \sigma\rangle \Downarrow_{boolexp} n_0 < n_1}\ \textsc{Lg} \tag{9}$$

$$\frac{\langle e_0, \sigma\rangle \Downarrow_{intexp} n_0 \quad \langle e_1, \sigma\rangle \Downarrow_{intexp} n_1}{\langle e_0 > e_1, \sigma\rangle \Downarrow_{boolexp} n_0 > n_1}\ \textsc{Gt} \tag{10}$$

$$\frac{}{\langle nv, \sigma\rangle \Downarrow_{boolexp} nv}\ \textsc{BVal} \tag{11}$$

$$\frac{\langle p, \sigma\rangle \Downarrow_{boolexp} n}{\langle \neg e, \sigma\rangle \Downarrow_{boolexp} \neg n}\ \textsc{Not} \tag{12}$$

$$\frac{\langle p_0, \sigma\rangle \Downarrow_{boolexp} n_0 \quad \langle p_1, \sigma\rangle \Downarrow_{boolintexp} b_1}{\langle p_0 \vee p_1, \sigma\rangle \Downarrow_{boolintexp} b_0 \vee b_1}\ \textsc{Or} \tag{13}$$

$$\frac{\langle p_0, \sigma\rangle \Downarrow_{boolexp} n_0 \quad \langle p_1, \sigma\rangle \Downarrow_{boolintexp} b_1}{\langle b_0 \wedge b_1, \sigma\rangle \Downarrow_{boolintexp} b_0 \wedge b_1}\ \textsc{And} \tag{14}$$

## 2.7. Semántica Operacional Estructural para Comandos