

¡En pocos años los DSL estarán por todos sitios!

DOMINIOS

Los DSL están en boga ahora mismo, todo el mundo dice estar desarrollando un entorno para crearlos: Java, .Net, PHP y cómo no, Ruby.

POR JOSE MARÍA RUÍZ

¿Pero qué es un DSL? Son unas siglas que significan «Domain Specific Language», en castellano «Lenguaje Específico de Dominio». Suena sofisticado ¿verdad? Lo cierto es que es una de las técnicas más antiguas de la Informática. La idea no es otra sino crear un lenguaje de programación para resolver un problema que se repite a menudo. Pero ¿qué se gana al hacerlo?

La clave de esta técnica es una de las palabras que aparecen en su nombre: «Domain» (Dominio). Un Dominio es un entorno en el cual se repiten constantemente los mismos patrones de uso y que, hasta cierto punto, sabemos que es estable. Así, un Dominio puede ser casi cualquier cosa; podemos pensar que el software de reproducción multimedia es un dominio, o la creación de Blogs.

Los lenguajes de programación son diseñados para ser capaces de enfrentarse a cualquier problema, mientras que un DSL se diseñó para resolver sólo uno. Al hacerlo se simplifica enormemente el desarrollo de soluciones a ese problema, y como suelen ser «problemas resueltos» (problemas con los que nos hemos peleado muchas veces o que en general están bien definidos), es muchísimo más sencillo trabajar con un DSL que con código Python.

Cómo Llegar a un DSL

Todo lo anterior suena muy abstracto, por lo que vamos a centrarnos en algo en concreto. Digamos que constantemente estamos controlando los permisos de un sistema de ficheros, pues queremos comprobar que no han cambiado. Lo normal es hacer algún tipo de script en *Bash* o, si somos personas inteligentes, crear un programa en *Python* para comprobar los permisos. Podemos acabar con algo así:

```
import Permisos
Permisos.Comprueba("/root/","700")
Permisos.Comprueba("/usr/local/publico","777")
```

Se supone que tenemos una librería *Permisos* en la que definimos el método *Comprueba*, que admite como parámetros una expresión regular y el permiso que deberían tener los ficheros que se correspondan con ella.

Este código fuente es un programa *Python* válido, y es bastante simple, pero supérfluo. Sobra sintaxis y siempre estaremos tentados a añadir código. Hemos roto el nivel de abstracción, cuando queramos controlar qué ficheros han cambiado de permisos sólo deberíamos centrarnos en eso, y no en *Python*. Podemos reescribir el código como *XML* y que un programa nuestro lo analice y ejecute los comandos que indiquemos en él:

```
<permisos>
<comprueba regex="/root/"
permisos="700" />
<comprueba regex="/usr/local/publico"
permisos="777" />
</permisos>
```

Ahora el código parece un fichero de configuración, pero de nuevo sobran cosas. El *XML* es aún más engorroso que el código *Python*, no hemos ganado en claridad, pero por lo menos este código ya no habla de *Python*, sino de permisos y comprobaciones. Huyamos del *XML* y pongamos la notación de nuestros sueños:

```
ficheros "/root/" : permisos
"700"
ficheros "/usr/local/publico" :
permisos "777"
```

Esto ya es otra cosa. Es una sintaxis tan simple que cualquiera podría recordarla; al leerla y al escribirla no pensamos en *Python*, sino en el problema que estamos resolviendo. Además nos permite pensar en añadir acciones interesantes dentro de este Dominio, como por ejemplo:

```
ficheros "/root/":
permisos "700"
propietario "root"
grupo "root"

ficheros "/usr/local/publico":
permisos "777"
propietario "nobody"
grupo "usuarios"
```

Esto ya tiene mucha mejor pinta. Acabamos de crear un *DSL*, o al menos lo hemos creado conceptualmente. Es muy sencillo enseñar a otra persona a usar este nuevo lenguaje y es muy útil para cualquiera que tenga un ordenador de uso público y que quiera controlar los permisos de sus ficheros. Es ahora cuando surge la gran pregunta ¿y qué hacemos ahora con este código tan útil?

Interpretar Código

Para ejecutar código Python pasamos un fichero a un intérprete, generalmente el programa *python*, y éste ejecuta el código realizando acciones en consonancia con su significado (ver Figura [1]). Para interpretar un DSL debemos hacer exactamente lo mismo. El gran problema que tienen los DSL es que existe un enorme hueco entre interpretar un lenguaje y ejecutar sus acciones. Todos sabemos cómo crear objetos, métodos y librerías, esa es la parte fácil. El problema principal es que generalmente nadie sabe exactamente cómo interpretar código.

Hace un par de décadas los DSL eran muy normales, ya que los lenguajes de programa-

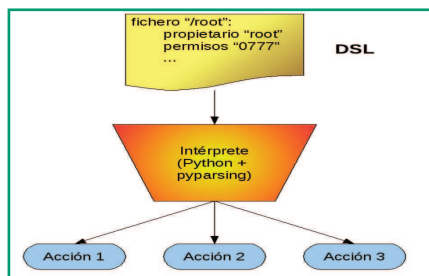


Figura 1: Esquema general de trabajo.

ción eran de bajo nivel (C,ASM,C+ + , Pascal...). Para sacar partido a una librería hecha con sudor y lágrimas, lo mejor era crear tu propio lenguaje de alto nivel. Incluso les dieron nombre: eran los llamados 4GL, lenguajes de cuarta generación. Pero a medida que los lenguajes de ámbito general fueron subiendo el listón, los programadores iban olvidando cómo crear un intérprete o un compilador.

La estructura de un intérprete no ha cambiado en toda la historia de la Informática. Se puede descomponer su trabajo en tres fases bien definidas: análisis sintáctico, análisis semántico y ejecución, ver Figura [2]. En la fase de análisis sintáctico se lee un fichero de texto y se van agrupando a los denominados «tokens», grupos de caracteres con algún significado. En esta fase se ignoran, por ejemplo, los espacios en blanco y cualquier elemento del fichero sin utilidad real (comentarios, saltos de líneas,...). A esta fase le sigue la fase de análisis semántico.

En el análisis semántico se comprueba si los tokens se corresponden con elementos de nuestro lenguaje. El analizador sintáctico puede haber encontrado el token «hola», pero es posible que no tenga significado en nuestro lenguaje a pesar de ser un token válido. El análisis semántico es el que enlaza los tokens con las acciones a realizar. Por ejemplo, si nuestro lenguaje posee funciones, es en esta fase cuando se comprueba que a la función se le está pasando el número correcto de parámetros o si falta un «return».

Una vez comprobado que todo está en regla, se ejecutan las acciones que se correspondan con el programa interpretado. Esta fase es la más sencilla, puesto que no difiere en nada respecto a un programa normal y corriente. Sobra decir que tanto el análisis sintáctico como el análisis semántico son muy complicados de realizar, y generalmente se emplean herramientas que generan el código fuente para los mismos a partir de, cómo no, un DSL espacial (es como la

pescadilla que se muerde la cola). Las más famosas son el dueto Lex/Yacc (o en su versión libre Flex, ver Recurso [1], y Bison, ver Recurso [2]) en C y para el mundo Java Antrl (ver Recurso [3]).

Para ejecutar «programas» escritos en nuestro DSL necesitamos un intérprete, y lo mejor es emplear alguna de las librerías que existen para Python. Usaremos *PyParsing* (ver Recurso [4]) creada por Paul McGuire con el objetivo de ayudarle a crear parsers de texto.

Creación de una Gramática.

A los conjuntos de reglas que reconocen los distintos tokens y les dan significado se les denomina «gramática». Veamos el código que aparece en el Listado [1]. Este es un ejemplo realmente sencillo de parsing: importamos las funciones necesarias de *pyparsing*, creamos una gramática compuesta por expresiones donde escogemos una que engloba a toda la gramática. En nuestro caso, llamamos a esta expresión *saludo* y la empleamos para parsear una cadena de texto. El resultado de ejecutar este código será:

```
> python hola-mundo.py
['Hola', ',', 'Mundo', '!']
```

El método *parseString()* analiza la cadena de texto que le pasamos y comprueba que se corresponde con la definición con la que hemos creado *saludo*. El resultado será una lista con los distintos tokens aislados, ignorando espacios y separadores. Da igual cuántos espacios, tabuladores o saltos de línea pongamos, si no forman parte del lenguaje *pyparsing* los ignorará.

Este pequeño ejemplo ya comienza a mostrarnos de lo que es capaz *pyparsing*, analicémoslo con cuidado. La expresión *saludo* es en realidad un objeto resultante de la ejecución de una serie de expresiones. La entidad más simple es *Literal()*, sólo reconoce una serie de caracteres precisos. Por comodidad, *pyparsing* también nos proporciona el objeto *CaselessLiteral*, que funciona igual que *Literal*, sólo que además reconoce el texto que contenga, ya sea en mayúscula o minúscula. Para *CaselessLiteral("hola")* reconoce como el mismo token las siguientes cadenas: "hola" "Hola", "HOLA", "hola"...

Word() nos permite definir qué caracteres pueden aparecer como primer carácter y los que le pueden seguir. Así, *Word(string.uppercase,string.lowercase)*

Listado 1: «Hola mundo»

```
01 # -*- coding: utf-8 -*-
02
03 from pyparsing import
    oneOf,Literal,Word
04 import string
05
06 saludo= Literal("Hola") +
    Literal(",") + \
07 Word( string.uppercase,
    string.lowercase ) + \
08 Literal("!")
09
10 print saludo.parseString("Hola
    , Mundo!")
```

implica que se admite cualquier palabra formada por caracteres siempre que el primero sea en mayúscula y el resto en minúscula, «Mundo» es aceptada, pero «mundo» no lo es. Cuando no se reconoce un token, *pyparsing* genera una excepción con el problema que ha localizado.

Los distintos elementos se unen mediante algún operador, como por ejemplo:

- +, que significa «seguido de».
- ^, que significa «o».
- |, que significa «el primero que coincida».

Así podemos escribir expresiones como:

```
saludo = Literal("buenos") |
+ (Literal("días") |
Literal("tardes"))
```

Como podemos ver en el ejemplo, es posible emplear los paréntesis para agrupar opciones, pero es posible hacer algo más interesante. El objeto *Group* nos permite definir un grupo de tokens con significado. Podemos sustituir lo anterior por:

```
hora = Group(Literal(
("días") | Literal("tardes")).
setName("HoraDelDia")
saludo = Group (Literal(
("buenos") + hora ).
setName("Saludo")
```

Cada grupo pasa a tener un significado especial para *pyparsing*, y es posible darle un nombre a cada uno, cosa muy útil para hacer debugging con el método *setDebug()* que nos permite ver los tokens que va encontrando *pyparsing*:

```
saludo = Group (Literal(
("buenos") + hora ).setName(
("Saludo").setDebug()
```

Listado 2: Código de nuestro DSL

```
01 ficheros "/root/":
02     permisos "0700"
03     propietario "root"
04     grupo "root"
05
06 ficheros "/usr/local/datos":
07     permisos "0777"
08     propietario "nobody"
09     grupo "usuarios"
10
11 ficheros "/home":
12     permisos "0600"
13     grupo "usuarios"
```

Por el momento hemos visto cómo reconocer una serie de tokens en una línea de texto. Para poder reconocer secuencias de líneas que se corresponden con un patrón, *pyparsing* nos ofrece los objetos *ZeroOrMore* («cero o más») y *OneOrMore* («uno o más»). Ambos parsean secuencias de grupos de tokens que se repiten:

```
saludos = OneOrMore(saludo)
```

En este ejemplo, nuestra gramática reconocería uno o más saludos. Si no hubiese ninguno, fallaría el reconocimiento, lanzando una excepción. Si queremos que la gramática trate de parsear hasta que no quede más texto, debemos finalizar nuestra expresión de mayor nivel con el objeto *StringEnd*:

```
texto_saludos =
OneOrMore(saludo) + StringEnd()
```

Si quedase texto por parsear que la gramática no reconociese antes de finalizar el fichero o cadena de caracteres, *pyparsing* lanzaría una excepción.

¡Semántica al Rescate!

Ya sabemos, de forma muy básica, cómo crear una gramática. Ahora veremos cómo dotarla de significado añadiendo semántica. Mientras la sintaxis se preocupa de qué caracteres forman los tokens, la semántica se preocupa de qué significado poseen. Podemos añadir semántica de dos formas.

La primera, más simple en un primer momento, consiste en procesar la lista de tokens que como resultado generan los métodos *parseFile()* y *parseString()*. Tras reconocer los tokens y los grupos, *parseString()* nos devolverá una lista, por ejemplo:

```
resultado =>
["buenas", "tardes"]
```

El problema surge cuando se dificulta un poco la gramática y comience a permitir estructuras más complejas. Además estaremos haciendo dos veces el mismo trabajo, puesto que *pyparsing* ya ha reconocido la cadena "buenas" y nosotros tendremos que volver a comprobarla contra una lista de acciones:

```
if resultado[0] ==>
"buenas":
...
```

La opción que propone *pyparsing* es la de añadir semántica directamente en los objetos con los que hemos compuesto nuestra gramática. Estos objetos poseen una serie de métodos que nos permiten actuar sobre los tokens que representan directamente. Por ejemplo, nuestra gramática puede obligar al usuario a escribir un punto o una coma, que debemos reconocer como tokens:

```
saludo = CaselessLiteral(
("hola") + Literal("."))
```

Pero este tipo de tokens no aportan nada a nuestro lenguaje, y podemos eliminarlos del resultado del parser mediante el método *suppress()*.

```
saludo = CaselessLiteral(
("hola") + Literal(".").
suppress())
```

Ahora podemos añadir una acción al token "hola" mediante el método *setParseAction()*. Este método acepta como parámetro una función que a su vez tiene tres parámetros:

- La cadena que está siendo parseada.
- La posición en la subcadena reconocida.
- Lista de tokens reconocidos.

Podemos pasar una función *lambda* que acepte tres caracteres:

```
>>> def func (a,b,toks):
>>>... print "He encontrado %s"%
%(toks)
>>> saludo = CaselessLiteral(
("hola").setParseAction(func)
+ Literal(".").suppress()
>>> saludo.parseString("hola.")
He encontrado ['hola']
(['hola'], {})
```

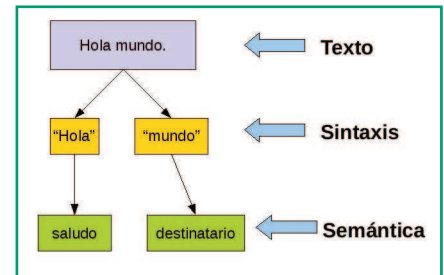


Figura 2: Fases.

Ya tenemos los ingredientes básicos para crear nuestro DSL de gestión de permisos.

DSL para Permisos

En el Listado [2] aparece el código de nuestro intérprete, mientras que en el Listado [3] podemos ver el fichero de nuestro lenguaje DSL. El método *Comprueba()* admite los tres parámetros que *setParseAction()* pasa a la función que recibe como parámetro. La clase *Group* agrupa su contenido dentro de una lista, por lo que el resultado de parsear la estructura *ficheros* será algo así:

```
[['/ruta', [['propietario',
'josemaria'], ['permisos',
'0700']]]]
```

De este modo, cuando accedemos a *toks[0][0]* estamos accediendo a la ruta del fichero. Respecto a los propiedades que debemos comprobar para esa ruta, empleamos un pequeño truco de Python. El método *dict()* convierte una lista de tuplas en un diccionario de forma automática, por lo que:

```
[['propietario',
'josemaria'], ['permisos',
'0700']]
```

acaba convirtiéndose en:

```
{'propietario':
'josemaria', 'permisos',
'0700'}
```

De esta forma tenemos un acceso rápido y sencillo a las distintas propiedades del fichero que queremos comprobar.

Para comprobar los metadatos de los ficheros nos apoyaremos en las librerías *pwd*, *grp*, *stat* y *os*. Usamos el método *os.path.exists()* para comprobar si existen los ficheros. Una vez lo confirmemos, podemos extraer los metadatos del fichero con la clase *os.stat()*. Esta clase contiene una estructura numérica de difícil comprensión, pero Python posee la librería *stat* que nos permite

Listado 3: El Intérprete de nuestro DSL

```

01 #!/usr/local/bin/python
02 # -*- coding: utf-8 -*-
03
04 from pyarsing import *
05 import string
06 import os
07 import os.path
08 import stat
09 import pwd
10 import grp
11
12 class ParserDSL:
13
14     def __init__(self):
15
16         letras_ruta =
17             alphanums+"/"+"-"+ "_"
18
19         # TOKENS
20         ficheros =
21             Literal("ficheros").suppress()
22         ruta = Word("/",letras_ruta)
23         permiso = Word(nums)
24         propietario =
25             Word(alphas,alphanums)
26         grupo =
27             Word(alphas,alphanums)
28         comillas =
29             Literal("'").suppress()
30         dospuntos =
31             Literal(':').suppress()
32         comprobar_grupo =
33             CaselessLiteral("grupo") +
34             comillas + grupo + comillas
35
36         comprobar_propietario =
37             CaselessLiteral("propietario")
38             + comillas + propietario +
39             comillas
40
41         comprobar_permisos =
42             CaselessLiteral("permisos") +
43             comillas + permiso + comillas
44
45         comprobar_ficheros =
46             ficheros + comillas + ruta +
47             comillas + dospuntos
48
49         accion = Group(
50             OneOrMore(Group(comprobar_grupo
51             | comprobar_propietario |
52             comprobar_permisos)) )
53
54         expr =
55             Group(comprobar_ficheros +
56             OneOrMore( accion
57             )).setParseAction(
58             self.Comprueba )
59
60         self.programa = OneOrMore(
61             expr ) + StringEnd()
62
63         def Comprueba(s,l,toks):
64
65             fichero = toks[0][0]
66             cmds=
67                 dict(toks[0][1].asList())
68
69             # Convertimos a Octal
70             cmds['permisos'] =
71                 oct(int(cmds['permisos'],8))
72
73             if os.path.exists(fichero):
74
75                 fichero_stats =
76                     os.stat(fichero)
77                 print "[Ruta %s]:" %
78                     (fichero)
79                 modo =
80                     fichero_stats[stat.ST_MODE]
81                 datos = { 'permisos':
82                     oct(modo & 0777),
83                     'propietario' :
84                     pwd.getpwuid(fichero_stats[stat.ST_UID])[0],
85                     'grupo' :
86                     grp.getgrgid(fichero_stats[stat.ST_GID])[0] }
87
88                 for comando in cmds.keys():
89                     if (datos[comando] !=
90                         cmds[comando]):
91                         print "\t%s fichero: <<%s>>
92                             => esperados <<%s>>" %
93                             (comando,datos[comando],cmds[comando])
94                 else:
95                     print "ERROR: el fichero %s
96                         no existe" % (fichero)
97
98                 print ""
99
100             def Ejecuta(self, fichero):
101
102                 self.programa.parseFile(fichero)
103
104                 p = ParserDSL()
105                 p.Ejecuta("permisos")

```

decodificarla. La clase *os.stat* se comporta como un diccionario, mientras que la librería *stat* nos permite usar nombres más o menos reconocibles para acceder a los valores de ese diccionario.

Los valores que nos interesan son: *stat.ST_MODE*, *stat.ST_UID* y *stat.ST_GID*. El primero devuelve un número que representa los permisos de un fichero. En Linux estamos acostumbrados a verlos en notación octal, por lo que antes de trabajar con ellos los convertimos. Para ello aplicamos el operador & (and binario) con la máscara 0777 (para más información ver Recurso [5]). Esta operación extrae únicamente los valores de los 3 últimos bytes del valor devuelto. Esos bytes contienen los permisos para el usuario, el grupo y el resto de usuarios del fichero. En nuestro DSL hemos permitido que el número se escriba en notación octal también, por lo que debemos convertir el número a octal con la función *oct()*.

Los valores que devuelve *os.stat* para el usuario y el grupo del fichero vienen dados por sus respectivos identificadores numéricos. En cambio, en nuestro DSL se escribe el nombre de ambos. Para realizar la conversión entre número y nombre empleamos las librerías *pwd* (que nos permite realizar operaciones con el fichero *passwd*) y *grp* (que hace lo propio con el fichero *groups*).

En el Listado [2] podemos ver todo esto en funcionamiento.

Conclusión

Crear un lenguaje de programación no es tan complicado. Este ejemplo es muy simple, no aparecen ni bucles ni estructuras recursivas, pero da una idea de lo sencillo que puede ser crear un DSL que cualquiera pueda usar. Muchas veces es más sencillo atacar un problema desde la perspectiva de crear un lenguaje para solucionarlo, que empleando estructuras de bajo nivel como listas o enteros.

Todas las grandes compañías están apoyando la aparición de frameworks para la creación de DSL, pero en realidad Python siempre ha podido crearlos sin problemas. Un vez más Python demuestra que la «alta tecnología» está disponible de forma sencilla. ■

RECURSOS

- [1] Página de inicio de Flex: <http://flex.sourceforge.net/>
- [2] El lenguaje Bison: <http://www.gnu.org/software/bison/>
- [3] Otra Herramienta para el Reconocimiento de Lenguajes: [http://www.antlr.org/](http://wwwantlr.org/)
- [4] Módulo de pyParsing: <http://pyparsing.wikispaces.com/>
- [5] Permiso de acceso a archivos: http://es.wikipedia.org/wiki/Permisos_de_acceso_a_archivos