



Trabajo de Fin de Grado

Grado en Ingeniería Informática

Un lenguaje para realizar DSLs

A language to make DSLs

Eleazar Díaz Delgado

La Laguna, 3 de Septiembre de 2018

D. Casiano Rodríguez León, con N.I.F. 42.020.072-S profesor Titular de Universidad adscrito al Departamento de TODO:Nombre del Departamento de la Universidad de La Laguna, como tutor

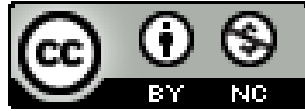
C E R T I F I C A (N)

Que la presente memoria titulada: "Un lenguaje para realizar DSLs" ha sido realizada bajo su dirección por D. Eleazar Díaz Delgado con N.I.F. 54.117.199-Q.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna, 3 de Septiembre de 2018.

Agradecimientos

Licencia



© Esta obra está bajo una licencia de Creative Commons Reconocimiento-NoComercial 4.0 Internacional

Resumen

ScriptFlow es un lenguaje de tipado dinámico para el desarrollo de scripts para la automatización de tareas, que requieran configuraciones. Se trata de un lenguaje basado en expresiones que da la opción a ser sensible a la indentación. Incluye una integración con Haskell por el cual puede ser ampliable. Este lenguaje propone diversas herramientas para ser usado como un DSL, usando la extensibilidad via Haskell.

Abstract

ScriptFlow is a dynamic typed language to develop scripts to automatize a sets of task, whose of these requires use of config files. It is a language based in expressions that allows you to use a syntax indentation-sensitive. It includes a integration with Haskell the language which one was built in. It can be extended using from scripts. This language provide several tools to be used as DSL.

Índice general

1	Introducción	7
2	Metodología	7
3	Lenguaje ScriptFlow	7
3.1	Sintaxis	7
3.1.1	Identación	7
3.1.2	Literales	8
3.1.3	Expresiones	9
3.1.4	Funciones	10
3.2	Orientado a objetos	11
3.2.1	Objetos	11
3.2.2	Clases	12
4	Configuración	13
4.1	Prompt	13
5	REPL	14
6	Arquitectura del proyecto	14
6.1	Introducción	14
6.2	Árbol abstracto sintáctico	17
6.3	Lenguaje intermedio	17
6.4	Interoperabilidad	17
7	Conclusiones	19

1 Introducción

Scriptflow es un lenguaje de programación interpretado pensado para la creación de lenguajes de dominios específicos. Los posibles DSL creados en esta plataforma. Son pensados para ser utilizados via terminal, o texto. Incluyendo TODO:

2 Metodología

TODO:

3 Lenguaje ScriptFlow

ScriptFlow es un lenguaje de tipado dinámico para el desarrollo de *scripts* para la automatización de tareas con servicios *webs*. La automatización de esta tareas se plantea mediante el uso de las propias API del servicio web, ya sea REST, o GraphQL.

3.1 Sintáxis

ScriptFlow se ha diseñado para ser sencillo de usar, para ello se optado por una sintaxis familiar a lenguajes como *python* o *c++*.

Los comentarios se comienzan con `#` y el retorno de línea delimita su fin. Los comentarios de líneas no están sujetos a reglas de indentación lo que permite cierta flexibilidad. Estos pueden también ser añadidos al final de una expresión.

```
# Variable ejemplo
  # Comentario
example = 1 + 1 # Comentario
```

```
if example == 2:
# Otro ejemplo
  print "something"
```

3.1.1 Identación

El lenguaje puede usar su sintáxis alternative sensible a la indentación como *python* o *haskell* pero se trata de algo opcional que puede ser omitido usando las llaves "{}":


```
fun saludar_a nombre:
  print("Saludos, " ++ nombre)
```

O la sintáxis equivalente al estilo de la familia de *C*:

```
fun saludar_a nombre {
  print("Saludos, " ++ nombre)
}
```

3.1.2 Literales

El lenguaje cuenta con diversas primitivas integradas por defecto, como booleanos, enteros, decimales, cadenas, expresiones regulares, comandos shell.

```
# none
none
```

```
# Booleanos
true
false
```

```
# Enteros
12 + 4 # ...
```

```
# Decimales (Se utiliza un double para su representación actualmente)
45.5
```

```
# Las cadenas pueden ser multilineas se crean con "
"Un\nejemplo
de cadena"
```

```
# Las regex se construyen con r" y terminan con ". Usan la sintaxis de PCRE.
r"a*" # Sin implementar aún
```

```
# Los comandos shell se crean con
$cd #HOME/repos; nvim .$
Ó mutltilinea
$
cd #HOME/repos
nvim .
```

```
$
```

```
0 la sintaxis alternativa  
$"pwd"
```

A su vez también existen tipos contenedores tales como los vectores y los diccionarios o tablas hash.

```
# Vectores  
[45, "tipos", []] # los vectores pueden contener diferentes tipos en el mismo vector  
  
# Diccionarios  
{ test -> [1,2,3,47,5]  
  , author ->  
    { name -> "Flynn"  
      , "vive en" -> "tal sitio"  
    }  
}
```

3.1.3 Expresiones

El lenguaje esta compuesto por expresiones, es decir, todas las estructuras devuelven algún valor. Estas expresiones, se encuentran delimitan de forma diferente según en que contexto se encuentren. Las expresiones en la base del archivo, tales como;

```
print "Hello World"
```

```
var = 67
```

```
func_call  
  first_param  
  second_param
```

Son delimitadas por el final de linea, o en el caso de exista cierto nivel de indentación mayor que el base '0' se agrupan con la primera sin indentación. Es decir, en el caso de `func_call` la expresion final sería `func_call(first_param, second_param)`. Se puede usar el carácter ';' para realizar esta separación (el cual es opcional al nivel base).

En el caso de expresiones más complejas que requieran un subconjunto de expresiones, hablamos de `if`, `for` Se contemplan dos casos para realizar

la terminación de las expresiones. Si se usa sintáxis sensible a la indentación, los niveles de indentación determinarán donde se halla la terminación de las expresiones. Pero, por si el contrario se usa sintáxis con llaves se necesitará añadir ';' para indicar la terminación de cada expresión. Y opcionalmente se puede quitar el ';' de la última expresión.

```
if always_true:
    make_test test1 test2
    other_func
    arg1
    arg2
end_test arg_end

if always_true {
    make_test test1 test2;
    other_func
    arg1
    arg2;
end_test arg_end
}
```

3.1.4 Funciones

La sintáxis permite definir dos tipos de funciones, aquellas que tienen un nombre y las lambda. Internamente solo hay lambdas debido a que la primeras son traducidas a una función lambda asignada a una variable.

La sintáxis de las funciones lambda es la siguiente:

```
# Con indentación
lam arg1 arg2:
    arg1

# O alternativamente
lam arg1 arg2 { arg1 }
```

Las funciones con nombre en el siguiente ejemplo;

```
fun func_name arg1 arg2 { arg1 }

fun func_name arg1 arg2:
    arg1
```

3.2 Orientado a objetos

3.2.1 Objetos

Un objeto en ScriptFlow es un diccionario con la clase a la que pertenece, si se diese el caso de ser un objeto instanciado.

En el siguiente ejemplo se enumeran las distintas formas de crear un objeto:

```
# A partir de un diccionario vacío
obj = {}

# A partir de none
obj2 = none
# Al asignar dentro de una variable establecida 'none' un "sub-item".
# Automáticamente se genera un objeto con ese item dentro
obj2.a = "ejemplo"
> { a -> "ejemplo" }

# A partir de una clase definida
class Test {}
# El constructor devolverá la instancia correspondiente
obj3 = Test()
```

Los objetos tienen diversas características incorporadas con el intérprete para mejorar su uso dentro de una DSL.

Las funciones **use** y **unuse** permiten modificar el ámbito actual de búsqueda de variables, y simplificar ciertos escenarios.

La resolución de nombres al usar **use** sobre un objeto. Tiene la menor precedencia dentro de la propia resolución de nombres, y la última llamada de **use** tiene mayor precedencia que las anteriores.

Y la función **unuse** deshace el último **use** usado. Se tiene planeado en futuras versiones realizar automáticamente un **unuse** al salir de un ámbito.

Un ejemplo ilustrativo de como trabaja esta funcionalidad dentro de un DSL.

```
class Github:
  fun repositories {} # return a list of repositories
  fun user_name {}
class Repository:
  fun name {}
```

```

    fun issues {}

gh = use Github()
filter_reg = Regex gh.user_name
for repo in repositories:
    use repo
    print name
    print issues.filter(filter_reg)
    unuse
unuse

```

3.2.2 Clases

El lenguaje tiene un básico soporte a la programación orientada a objetos. Permite la definición de clases sin la capacidad de herencia. El siguiente ejemplo sobrecarga el constructor de la clase usando el método especial `__init__`.

Los métodos asociados al objeto internamente se pasan a si mismo como argumento usando la palabra reservada `self`. El lenguaje no soporta métodos estáticos.

```

class Repository {
    fun __init__ new_name {
        self.url = none
        self.local_repo = none
        self.name = new_name
    }
}

```

La siguiente tabla muestra los métodos disponibles para sobrecargar.

Operador	Nivel de precedencia	Precedencia	Nombre método
**	8	Izquierda	<code>--pow--</code>
*	7	Izquierda	<code>--mul--</code>
/	7	Izquierda	<code>--div--</code>
%	7	Izquierda	<code>--mod--</code>
+	6	Izquierda	<code>--plus--</code>
-	6	Izquierda	<code>--minus--</code>
++	5	Derecha	<code>--append--</code>
==	4	Izquierda	<code>--eq--</code>
!=	4	Izquierda	<code>--neq--</code>
/=	4	Izquierda	<code>--neq--</code>
>	4	Izquierda	<code>--gt--</code>
<	4	Izquierda	<code>--lt--</code>
<=	4	Izquierda	<code>--le--</code>
>=	4	Izquierda	<code>--ge--</code>
&&	3	Derecha	<code>--and--</code>
	3	Derecha	<code>--or--</code>
!	1	Izquierda	<code>--not--</code>
@	1	Izquierda	<code>--at--</code>
print	-	-	<code>--print--</code>

El método especial `--print--` indica la forma de como debe mostrarse por pantalla el objeto al usar la función `print`.

4 Configuración

El fichero de configuración se localiza mediante el estándar XDG. Normalmente localizado en `/home/username/.config/scriptflow`. La configuración es un fichero tipo YAML. El cual permite especificar parámetros de configuración, tales como el prompt, shell. O parámetros específicos con la API Web; tales como la autenticación o posibles preferencias.

4.1 Prompt

En el modo interactivo del intérprete (repl) permite la personalización del **prompt**. Tales como la salida de la ejecución de comandos shell, y diversos comandos propios del intérprete. La configuración del prompt se puede realizar desde el fichero de configuración (véase: 4) en la sección **repl**.

Por defecto, la sección del *prompt* contiene la siguiente configuración:

```

repl:
# ...
prompt: |
    "$pwd".exec().strip() ++ " >>> "
# ...

```

La configuración del prompt debe ser una expresión de ScriptFlow.

5 REPL

El **REPL** puede ser accedido mediante comando de líneas **scriptflow** o con la finalización de ejecución de un **script** con la opción **-e**. Se pueden ver más opciones del ejecutable del intérprete mediante **scriptflow --help**. Una vez, iniciado el **REPL** se mostrará por defecto el **prompt** predeterminado (configuración véase: 4.1).

Desde el **REPL** se puede escribir cualquier tipo de expresión definida por el lenguaje. Y los comandos del intérprete los cuales comienzan por ":". Se puede ver una lista de los comandos con **:help**

- **:instr**

Permite visualizar, a que instrucciones se traduce el código. Estas instrucciones son parciales solo sirven de guía. (Véase: 6.3)

- **:mem**

Muestra parcialmente las variables disponibles en memoria.

- **:quit**

Sale del intérprete.

6 Arquitectura del proyecto

6.1 Introducción

El lenguaje se ha realizado usando un lenguaje puramente funcional lo que requiere diferentes enfoques al realizar el diseño del interprete. Ya que no posee una interfaz orientada a objetos. Dada esta diferencia voy a detallar en cierta medida peculiaridades del desarrollo, en las siguientes secciones. Antes de ello empezaremos con un pequeño análisis de como funciona el intérprete.

Dado un fichero de entrada con el código escrito en ScriptFlow.

```
fun say_hi name:
  "Hola, " ++ name

say_hi("Mundo")
```

Se procede al *parseo* del código, el cual, se realiza a dos fases. La primera el *tokenizador*, se encarga de transformar, el texto en de entrada, en una secuencia de *tokens*. Estos tokens representan los elementos importantes que se usarán para generar el AST (Abstract Syntax Tree). Cada *token* contiene la información necesaria para reconstruir la parte esencial del código.

```
[FunT, NameIdT "say_hi", NameIdT "name", OBraceT,
  LitTextT "Hola, ", OperatorT "++", NameIdT "name",
  CBraceT,
  NameIdT "say_hi", OParenT, LitTextT "Mundo", CParenT]
```

En esta fase de *tokenización*, se procede a identificar los niveles de indentación en el código en el caso necesario (Para más información ir: 3.1.1). El *tokenizador* procede a añadir las llaves necesarias en el caso de usar la gramática del language sensible al contexto. Estos *tokens* se identifican con OBraceT y CBraceT.

La segunda fase del *parseo* se encarga de generar el árbol sintáctico abstracto o sus siglas en inglés AST (Abstract Syntax Tree).

```
SeqExpr [
  VarDecl (Simple "say_hi")
    (FunDecl ["name"]
      (SeqExpr
        [Apply (Simple "++")
          [Factor (AStr "Hola, "),
            Identifier (Simple "name")]]
        )
      )
    ,
  Apply (Simple "say_hi")
    [SeqExpr [Factor (AStr "Mundo")]]
]
```

La salida del AST está simplificada en este ejemplo, se puede ver una salida más detallada, añadiendo una mayor verbosidad `scriptflow -v` (Ver `scriptflow --help` para más información).

Este proceso se realiza mediante un *parser combinador*, el cual se comporta de forma parecida a los PEGs. Un ejemplo simplificado es la definición de una función:

```
parseFunDecl :: TokenParser Expression
parseFunDecl = do
  funT
  funName <- nameIdT
  params  <- many nameIdT
  prog    <- parseBody
  return (VarDecl
    (Simple funName)
    (FunDecl params prog))
```

Una vez generado se realiza la comprobación del **scope** del AST. En esta fase comprueban si están usando variables que no existen, o si sobrescriben otra. Y se procede al renombrado de las variables.

```
SeqExpr [
  VarDecl var_0      -- say_hi
    (FunDecl [param_0] -- name
      (SeqExpr
        [Apply op_0  -- "++"
          [Factor (AStr "Hola, "),
            Identifier param_0]
        ]
      )
    )
  ,
  Apply var_0      -- say_hi
    [SeqExpr [Factor (AStr "Mundo")]]
]
```

Una de la últimas fases es la conversión del AST al conjunto de instrucciones simplificado. (Vease: 6.3)

```
Assign var_0
  OFunc [param_0]
    CallCommand op_0 ["Hola, ", GetVal param_0]

CallCommand var_0 ["Mundo"]
```

Y de esta forma es como se representa el código en memoria. Es decir, las funciones que se definan su contenido es guardado en este formato.

6.2 Árbol abstracto sintáctico

El AST (Abstract Syntax Tree) de ScriptFlow ha pasado por diversos cambios en el transcurso del proyecto. Inicialmente se considero usar el modelo conceptual que se aplica en el paquete "language-haskell-ext" el cual codifica el AST de forma genérica para que en cada nodo se encuentre el componente genérico. Este componente, se fija en el AST a lo largo de todos los nodos lo que conlleva a crear un componente complejo e innecesario en la mayoría de los nodos. Se crea un AST poco flexible.

La solución a este problema se encontró dentro de los *papers* que estan siendo implementados en el propio *GHC*. En el *paper*[Trees that Grow] se describe como se logra una estructura de datos más flexible que la convencional. Que por medio de los tipos de familia abiertos (Open Family types) se logra modificar el tipo de dato complementario en cada nodo del AST según que fase del compilador se encuentre.

6.3 Lenguaje intermedio

La última fase es la conversión del AST (Abstract Syntax Tree) en conjunto de instrucciones que se usarán, para describir las secuencia de acciones. Para llevar a cabo la ejecución de un script de $\{\{\{\text{lang_name}\}\}\}$ Este conjunto de instrucciones se encuentra expresado en ADT (Abstract Data Tree), de tal forma que encaje con la estructura de datos mónada libre (*Free Monad*). Esta estructura, secuencia las instrucciones y permiten usar la notación *do* de Haskell.

6.4 Interoperabilidad

La metaprogramación ha supuesto una simplificación en la comunicación entre lenguaje padre e hijo. Con el fin de reutilizar las funciones ya testeadas de Haskell, en ScriptFlow. Únicamente realizando cambios oportunos, como el orden de los argumentos.

El desarrollo de esta característica se basa en la definición de un isomorfismo entre los tipos de datos de haskell y los de ScriptFlow. este isomorfismo se encuentra en las clases de tipo `FromObject` y `ToObject`.

Apesar de este isomorfismo, existe una dificultad añadida debido a que las funciones en Haskell son currificadas. Por ejemplo dada la siguiente función `f` que recibe dos parametros y retorna un `Bool`.

```
f :: Int -> Int -> Bool
```

Se debe eliminar esta currificación, para que el tipo concuerde con algo más uniforme.

```
f :: [Int] -> Bool
```

La primera solución, que resuelve el problema, se hizo mediante clases de tipos.

```
class Normalize a
  normalize :: a -> [Object] -> Object

instance ToObject a => Normalize a where
  normalize = -- implementación omitida

instance (ToObject a, Normalize r) => Normalize (a -> r) where
  normalize = -- implementación omitida
```

Las cuales mediante el uso de la recursividad entre instancias de las clases de tipos se resolvía el problema. Sin embargo el método no es eficiente. Y requiere de una clase auxiliar para contar el número de argumentos que posee una función, la cual use sobrelapamiento entre instancias.

La opción actual reside crear las *wrappers* a medida para cada función convertida. Para ello se implementado una solución basada en el uso de la meta-programación conocida en Haskell por *Template Haskell*.

Ejemplo de código auto-generado, dada la función:

```
(>) :: Int -> Int -> Bool
(>) = -- implementación omitida
```

La salida obtenida es:

```
greaterThan :: [Object] -> StWorld Object
greaterThan objs =
  let expectedArgs = 2
      givenArgs     = length objs
  case compare givenArgs expectedArgs of
    LT -> throw $ NumArgsMismatch expectedArgs givenArgs
    GT -> throw $ NumArgsMismatch expectedArgs givenArgs
    EQ -> do
      let [arg1, arg2] = objs
```

```

val1 <- fromObject arg1
val2 <- fromObject arg2
toObject (>) val1 val2

```

Una de las desventajas de esta solución se encuentra en las propias limitaciones del *Template Haskell*. Debido a que no es posible inferir el tipo de una expresión, lo que requiere añadir el tipo de la expresión.

```

$(normalize [| (>) :: Int -> Int -> Bool |])
-- Se repite el tipo obligatoriamente

```

La implementación de la "meta-función" se encuentra en el módulo *Compiler.Prelude.Th*.

Otro factor de interoperabilidad a destacar, es la creación de un *QuasiQuoter*. Lo que permite incrustar fragmentos de ScriptFlow dentro de Haskell. Y dentro del propio *QuasiQuoter* realizar llamadas a funciones de Haskell usando el mecanismo anteriormente descrito para la conversión de funciones entre ambos lenguajes.

```

requestLogin :: String -> String -> IO ()
requestLogin = -- se omite implementación

githubClassSC :: Interpreter Object
githubClassSC = [scriptflow|
    # Github base class
    class Github:
        fun login:
            print "Logging to get authorization token to use in future connections"
            username = get_line "User Name: "
            password = ask_password "*" "Password: "
            __call__ ${requestLogin} username password
|]

```

7 Conclusiones

TODO: