DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Trabajo Práctico

Generador de JSON de ejemplo

Teoría de Lenguajes
Primer cuatrimestre 2022

Integrante	LU	Correo electrónico
Martinez, Franco	025/14	franco.martinez93@hotmail.com
Figarola, Lucas Adriel	953/13	lukas12_alfa56@hotmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2160 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
http://www.exactas.uba.ar

Índice

1. Introducción

El lenguaje Go permite definir tipos de datos como estructuras usando tipos básicos, arreglos u otras estructuras. En este trabajo construiremos un programa que permitirá leer definiciones de un subconjunto simplificado de tipos en Go y generar textos JSON válidos con datos aleatorios que respeten esas definiciones.

Ejemplo de entrada valida:

```
type tienda struct {
        nombre string
        producto int
        clientes []string
        ventas []float64
}

type producto struct {
        nombre string
        cantidad int
        precio int
}
```

Ejemplo de salida para esta estructura:

```
{
    "nombre":"fhdhfgh",
    "producto":{
        "nombre":"kjhkjdf",
        "cantidad": 5,
        "precio" : 200
    },
    "clientes":[
        "Juan Pablo", "Tomas Perez"
    ],
    "ventas":[
        32.5,
        223.3,
        999.0
    ],
}
```

2. Desarrollo

Para lograr aceptar este formato de tipos lo primero que se armó fue la gramática.

2.1. Gramática

$G = < \{start, atrib, atrib2, vartype\}, \{int, bool, float64, string, [], type\{,\}, nombre, struct\}, P, start >$

P:
start $->$ type nombre struct { atrib
atrib $->$ } | nombre vartype atrib | nombre nombre atrib type nombre struct { atrib | nombre struct { atrib2 } atrib
atrib2 $->$ nombre vartype atrib2 | lambda
vartype $->$ int | bool | float64 | string | [ ]vartype

Esta gramatica es LALR

Si bien no habia forma facil de definir atributos para este problema, pueden considerar los siguientes como atributos (o casi atributos)

Atributos;

atributo	tipo	sintetizado/heredado
start.dict	diccionario	sintetizado
atrib.dict	diccionario	sintetizado
vartype.tipo	string	sintetizado
vartype.dim	int	sintetizado

Basicamente en la implementacion de esta gramatica se va utilizar una estructura tipo diccionario de forma recursiva por los atributos start.dict y atrib.dict de modo que cada vez que se parsee un atributo se agrege a ese diccionario y ademas durante el parseo se verifique si ya existe el nombre del atributo que se quiere agregar. Para asignarle un valor a ese atributo nos asistimos de vartype.typo y vartype.dim, los cuales segun el tipo generan un valor que segun vartype.dim si es 0 agrega un solo valor del tipo indicado por vartype.tipo. Si es mayor a 0 (o sea un array) genera un array aleatorio segun la dimension de vartype.dim. Para esto se utliza una funcion que por el momento llamaremos random_value(tipo,dim) la cual genera este dato aleatorio y se encuentra en el codigo de implementacion. En el caso que se defina otro struct dentro del mismo parseado se creara un diccionario definido dentro de este mismo como otro atributo,y los atributos de este nuevo struct tambien se definiran recursivamente como lo explicado en lo anterior.

```
start -> type nombre struct { atrib
    { start.dict = dicc{nombre : atrib.dict} }

atrib -> }
    { atrib.dict = dicc_vacio }

atrib1 -> nombre vartype atrib2
        { atrib1.dict = atrib2.dict,
            IF nombre no definido en atrib1.dict
            THEN definir (nombre : random_value(vartype,tipo,vartype.dim))
                en atrib1.dict
             }

atrib1 -> nombre1 nombre2 atrib2 type nombre3 struct { atrib3
```

```
    { atrib1.dict = atrib2.dict,
      IF nombre1 != nombre2 AND nombre3 == nombre2
        THEN definir (nombre1 : atrib3.dict) en atrib1.dict
        }

atrib1 -> nombre struct {  atrib' } atrib2
    { atrib1.dict = atrib.dict,
        IF nombre no definido en atrib.dict
        THEN definir (nombre : random_value(vartype,tipo,vartype.dim))
            en atrib1. dict
             }
atrib'1 -> nombre vartype atrib'2
    { atrib'1.dict = atrib'2.dict,
            IF nombre no definido en atrib'1.dict
            THEN definir (nombre : random_value(vartype))
                    en atrib1.dict
             }

atrib' -> lambda
    { atrib2.diccionario = dicc_vacio }

vartype -> int
    {vartype.tipo = 'int' , vartype.dim = 0 }

vartype -> bool
    {vartype.tipo = 'bool' , vartype.dim = 0 }

vartype -> float64
    {vartype.tipo = 'float64' , vartype.dim = 0}

vartype -> string
    {vartype.tipo = 'string' , vartype.dim = 0}

vartype1 -> [ ]vartype2
    {vartype1.tipo = 'vartype2.tipo' , vartype1.dim = vartype2.dim + 1}
```

La idea de condicionar el definir un atributo en el diccionario, es para evitar alguna circularidad y tambien evitar atributos repetidos. Por ejemplo en 'atrib − > nombre vartype atrib' sabemos que no vamos a poder definir un atributo con ese "nombre"dentro de toda la estructura por que lo sobreescribe con un nuevo valor. En el caso de 'atrib1 − > nombre1 nombre2 atrib2 type nombre3 struct {' atrib3, nombre1 no puede ser igual a nombre2 dado que nombre2 es el nombre del nuevo type struct que se va a definir y tiene que ser ademas igual a nombre3 para que se respete la creacion de ese struct. En la implentacion estas condiciones rompen la ejecuccion si no se cumplen

3. Implementación

3.1. Componentes

La implementación fue realizada en Python 3.

La entrada es por consola y ahí se indica el archivo donde esta el struct. Ejemplo:

Figura 1: Entrada por consola

La salida es por consola y tambien por archivo (json_salida.txt). Ejemplo:

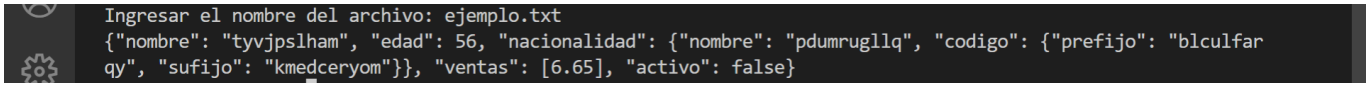

Figura 2: Salida por consola

3.2. Lexer

```
 # List of token names.   This is always required
tokens = (
        'TYPE',
        'STRUCT',
        'NOMBRE',
        'TYPESTRING',
        'TYPEINT',
        'TYPEFLOAT',
        'TYPEBOOL',
        'LBRACKET',
        'RBRACKET',
        'ARRLOCKS'
    )

# Regular expression rules for simple tokens
 #   t_STRUCT  = 'struct'
t_LBRACKET  = r'\{'
t_RBRACKET  = r'\}'
t_ARRLOCKS = r'\[]'




    # A regular expression rule with some action code
    # Note addition of self parameter since we're in a class

def t_TYPE(t):
    r'type'
    return t

def t_TYPESTRING(t):
    r'string'
    return t
```

```
33
34 def t_TYPEINT(t):
35     r'int'
36     return t
37
38 def t_TYPEFLOAT(t):
39     r'float64'
40     return t
41
42 def t_TYPEBOOL(t):
43     r'bool'
44     return t
45
46 def t_STRUCT(t):
47     r'struct'
48     return t
49
50 def t_NOMBRE(t):
51     r'[a-zA-Z0-9\.\,\-\_\~\+\´]+'
52     return t
53
54 # Define a rule so we can track line numbers
55 def t_newline(t):
56     r'\n+'
57     t.lexer.lineno += len(t.value)
58
59      # A string containing ignored characters (spaces and tabs)
60 t_ignore  = ' \t'
61
62      # Error handling rule
63 def t_error(t):
64     print("Illegal character '%s'" % t.value[0])
65     t.lexer.skip(1)
66     return 'ERROR'
67
68      # Build the lexer
69    #  def build(self,**kwargs):
70     #      self.lexer = lex.lex(module=self, **kwargs)
71
72      # Test it output
73 def test(data):
74     while True:
75         tok = lexer.token()
76         if not tok:
77             break
78         print(tok)
79
80
81 lexer = lex.lex()
```

3.3. Parser

Mediante el Parser determinamos si una entrada pertenece sintácticamente a nuestro lenguaje. Lo que devuelve el parseo, si este es correcto, es un diccionario con el nombre de nuestro JSON como clave el cual contiene al diccionario que representara el JSON que se pide.

```
letters = string.ascii_lowercase

def p_start(p):
    'start : TYPE NOMBRE STRUCT LBRACKET atrib'
    p[0] = {p[2]: p[5]}

def p_atrib(p):
    'atrib : NOMBRE vartype atrib'
    value = None
    if p[2]['dim'] == 0:
        value = rand_value(p[2]['tipo'])
    else:
        value = rand_list(p[2]['tipo'],p[2]['dim'])

    p[0] = {p[1]: value}

    if p[1] in p[3]:
        print('Alerta: el stributo "',p[1],'" se intenta definir de nuevo')
        raise Exception

    p[0].update(p[3])


def p_atrib_brack(p):
    'atrib : RBRACKET'
    p[0] = {}


def p_atrib_struct_type(p):
    'atrib : NOMBRE NOMBRE atrib TYPE NOMBRE STRUCT LBRACKET atrib'
    if p[1] == p[2]:
        print('Alerta: Se intenta definir el struct "',p[1],'" con el mismo nombre de
        ↪  tipo')
        raise Exception
    if(not(p[2] == p[5])):
        print('Alerta: Se intenta crear el nuevo type struct "', p[2], '" con otro nombre -
        ↪  "', p[5],'" ')
        raise Exception

    if p[1] in p[3]:
        print('Alerta: el nombre para el struct "',p[1],'" ya fue utilizado')
        raise Exception

    p[0] = {p[1]:p[8]}

    p[0].update(p[3])


def p_atrib_struct_var(p):
    'atrib : NOMBRE STRUCT LBRACKET atrib2 RBRACKET atrib'
    if p[1] in p[6]:
        print('Alerta: el nombre para struct "',p[1],'" ya fue definido')
        raise Exception
    if p[1] in p[4]:
        print('Alerta: el nombre para struct "',p[1],'" se repite dentro del struct')
        raise Exception
    p[0] = {p[1]:p[4]}
```

```
    p[0].update(p[6])


def p_atrib2(p):
    'atrib2 : NOMBRE vartype atrib2'
    value = None

    if p[2]['dim'] == 0:
        value = rand_value(p[2]['tipo'])
    else:
        value = rand_list(p[2]['tipo'],p[2]['dim'])

    p[0] = {p[1]: value}
    if p[1] in p[3]:
        print('Alerta: el atributo "',p[1],'" se intenta definir de nuevo')
        raise Exception

    p[0].update(p[3])


def p_atrib2_none(p):
    'atrib2 : '
    p[0] = {}

def p_vartype_int(p):
    'vartype : TYPEINT'
    p[0] = {'tipo': p[1], 'dim': 0}

def p_vartype_float(p):
    'vartype : TYPEFLOAT'
    p[0] = {'tipo': p[1], 'dim': 0}

def p_vartype_string(p):
    'vartype : TYPESTRING'
    p[0] = {'tipo': p[1], 'dim': 0}

def p_vartype_bool(p):
    'vartype : TYPEBOOL'
    p[0] = {'tipo': p[1], 'dim': 0}

def p_vartype_arr(p):
    'vartype : ARRLOCKS vartype'
    p[0] = {}
    p[0]['tipo'] = p[2]['tipo']
    p[0]['dim'] = 1 + p[2]['dim']


def p_error(p):
    print("error de sintaxis ", p)

def rand_value(t):
    if t == 'int':
        return rn.randint(0,100)
    elif t == 'string':
        return ''.join(rn.choice(letters) for i in range(10))
    elif t == 'bool':
        b = rn.randint(0, 1)
```

```
        if b == 0:
            return False
        else:
            return True
    elif t == 'float64':
        return round(rn.uniform(0,10),3)


def rand_list(t,d):
    if d == 0:
        return rand_value(t)
    r = rn.randint(0, 5)
    return [rand_list(t, d-1) for i in range(r)]


def checkJson(j_name,dict_j):
        for key in dict_j:
            if key == JSON_name:
                print('Error: un atributo fue definido con el mismo nombre que el JSON')
                raise Exception
            if type(dict_j[key]) == dict:
                checkJson(j_name, dict_j[key])


#  Aca empieza la el codigo principal

var = input("Ingresar el nombre del archivo: ")

file1 = open(var, 'r')
text = file1.read()
file1.close()

parser = yacc.yacc()

result = parser.parse(text)
#print(result)

JSON_out = {}
JSON_name = ''
for key in result:
    JSON_name = key
    JSON_out = result[key]

checkJson(JSON_name, JSON_out)


json_string = json.dumps(JSON_out)
print(json_string)

with open('json_salida.txt', 'w') as outfile:
    json.dump(JSON_out, outfile, indent=4)
```

Algo que no hace el parseo es verificar que el nombre del JSON no repite dentro de sus atributos sean variables simples u otros structs. eso se hace al final antes mandar por salida el JSON terminado. Si el nombre si se encuentra repetido corta la ejecucion

3.4. Casos de prueba

Se presentan a continuación algunos casos de prueba con expresiones válidas e inválidas y los resultados obtenidos.

3.4.1. Caso de expresión valida 1

Entrada:

```
type persona struct {
        nombre string
        edad int
        nacionalidad pais
        ventas []float64
        activo bool
}

type pais struct {
        nombre string
        codigo struct {
                prefijo string
                sufijo string
        }
}
```

Salida para esta estructura:

```
{
    "nombre": "zptlppasaq",
    "edad": 11,
    "nacionalidad":{
        "nombre": "avzhgveyaj", "codigo": {"prefijo": "hhktdgardk", "sufijo": "artrfacpbm"}
    },
    "ventas": [3.545], "activo": true
}
```

3.4.2. Caso de expresión valida 2

Entrada:

```
type pos struct {
        x int
        y int
}
```

Salida para esta estructura:

```
{
    "x": 82,
    "y": 64
}
```

3.4.3. Caso de expresión valida 3

En este ejemplo tenemos una estructura que esta añadida por nombre por otra estructura que a su vez esta última estructura esta añadida por otra.

Entrada:

```
type tienda struct {
        nombre string
        produc producto
        clientes []string
        ventas []float64
}

type producto struct {
        nombre string
        cantidad int
        precio int
        cate categoria
}

type categoria struct {
        nombre string
        codigo int
}
```

Salida para esta estructura:

```
{
    "nombre": "rdnoexleck",
    "produc": {
        "nombre": "wlmsroeniw",
        "cantidad": 82,
        "precio": 80,
        "cate": {
            "nombre": "fiaghubmob", "codigo": 35
        }
    },
    "clientes": [
        "pxkkvwrgmv",
        "vnkvkfiomn",
        "caosoytxoo",
        "qatdcxlbyq",
        "lpqeovadee"
    ],
    "ventas": [
        4.829,
        9.118
    ]
}
```

3.4.4. Caso de expresión valida 4

Similar al del caso 3 pero más compleja.
Entrada:

```
type tienda struct {
        nombre string
        produc producto
        clientes []string
        ventas []float64
        informe struct {
                id int
                compras []string
```

```
        }
}

type producto struct {
        nombre string
        cantidad int
        precio int
        cate categoria
}

type categoria struct {
        nombre string
        codigo int
        etiqueta struct {
                nombre string
        }
        mar marca
}

type marca struct {
        nombre string
}
```

Salida para esta estructura:

```
{
    "nombre": "kssggcneci",
    "produc": {
        "nombre": "uzuokmllmt",
        "cantidad": 79,
        "precio": 70,
        "cate": {
            "nombre": "zeefulzztc",
            "codigo": 39,
            "etiqueta": {
                "nombre": "bgctrwhsxs"
            },
            "mar": {
                "nombre": "ccxgtoczzc"
            }
        }
    },
    "clientes": [
        "uvziytiqxx",
        "jgkumuycys",
        "jmanmprgnm"
    ],
    "ventas": [
        0.731,
        9.009,
        8.669,
        1.478
    ],
    "informe": {
        "id": 26,
        "compras": [
            "ntsuuxwcea",
```

```
            "jfvrpfbrin",
            "tarhngcqxl"
        ]
    }
}
```

3.4.5. Caso de expresión invalida 1

Entrada:

```
type cliente struct {
        string
        apellido string
        dni int
}
```

Salida:

```
error de sintaxis  LexToken(TYPESTRING,'string',2,23)
```

3.4.6. Caso de expresión invalida 2

Entrada:

```
type cliente struct {
        nombre string
        apellido string
        dni int
        fact factura
}

type factura struct {
        codigo int
        pagada bool bool
}
```

Salida:

```
error de sintaxis  LexToken(TYPEBOOL,'bool',10,127)
```

3.4.7. Caso de expresión invalida 3

El siguiente caso invalido que probamos es con referencias circulares. Es decir, en este caso persona incluye a pais, y este ultimo a su vez incluye a persona.

Entrada:

```
type persona struct {
        nombre string
        edad int
        nacionalidad pais
        ventas []float64
        activo bool
}

type pais struct {
        nombre string
        cliente persona
```

```
        codigo struct {
                prefijo string
                sufijo string
        }
}
```

Salida:

```
error de sintaxis  None
```

4. Conclusión

Para definir los type struct deben estar en orden, es decir por ejemplo si defino los struct producto - categoría - marca entonces categoría puede incluir a marca pero producto no puede incluir a marca. No fue posible definir la gramatica de esa manera, a menos que involucrase pedir se extendiese la gramatica pero eso nos empezaba a generar conflictos de parseo.
No fue complicado trabajar con PLY e implementar una gramatica que se comporte como una gramatica de atributos sintetizados, posiblemente con atributos heredados se hubiesen chequeado cosas en el parseo en vez de al final, como lo del nombre del JSON dentro del mismo, pero con PLY no resultaba claro como implementar atributos heredados.