DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

TP: Generador de JSON de ejemplo

Teoría de Lenguajes
Primer Cuatrimestre de 2022

Integrante	LU	Correo electrónico
Ryan Itzcovitz	169/19	ryanitzcovitz@gmail.com
Guido Rodriguez Celma	374/19	guido.rc98@gmail.com
Miguel Rodriguez	57/19	mmiguerodriguez@gmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (+54 11) 4576-3300
https://exactas.uba.ar

Índice

1. Introducción

En el presente trabajo implementamos un programa que lee definiciones de tipos de estructuras (simplificadas) en el lenguaje Go, y genera una salida en formato JSON. Los datos de la salida se completan con valores aleatorios, y su formato se corresponde con la definición recibida como entrada.

En el siguiente ejemplo podemos ver, a partir de una entrada, un resultado esperado:

Código 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
    }
}
```

Código 2: Salida

```
{
  "nombre": "dgqxe",
  "edad": 224,
  "nacionalidad": {
    "nombre": "yrlkk",
    "codigo": {
      "prefijo": "nbirg",
      "sufijo": "mgqam",
    }
  },
  "ventas": [
    516.787,
    618.119,
    878.254,
    693.127,
    647.321
  ],
  "activo": true,
}
```

2. Desarrollo

2.1. Software utilizado

Para la implementación del trabajo utilizamos el lenguaje `Python`, junto con `PLY`, un paquete con las implementaciones de las herramientas `LEX` y `YACC` para poder tokenizar la entrada y definir los métodos de evaluación de las producciones. Para la generación de datos aleatorios utilizamos los paquetes `string` y `random`.

2.2. Gramática

Lo primero que notamos es que `PLY` requiere que la gramática a analizar sea LALR(1) por lo que como primer paso definimos, utilizando lo visto en la materia, una gramática de este tipo que pueda generar cadenas del lenguaje. La gramática utilizada es la siguiente:

Código 3: Gramática

```
S -> type DATA S'
S' -> type DATA S'
DATA -> id struct OBJ
OBJ -> { V }
V -> DATA
V -> id DATATYPE V
V -> λ
DATATYPE -> datatype
DATATYPE -> [] DATATYPE
```

Veamos mas en detalle que es lo que genera cada una de las producciones de nuestra gramática:

- `S`
  - Permite definir cual sera nuestro objeto principal (el que debemos imprimir), junto con la información de su estructura interior a través de la producción `DATA`.
  - Luego podemos seguir expandiendo el resto de objetos que se sigan definiendo con la producción `S'`.
- `S'`
  - Una copia de `S`, que nos permite definir otro comportamiento para producciones que no son la principal.
- `DATA`
  - Permite crear estructuras de objetos con el nombre del parámetro y luego expandir el struct con las producciones de `OBJ`.

- `OBJ`
  - Derivan en `V`, requiere que este entre corchetes.
- `V`
  - Es la producción encargada de crear parámetros con nombre y tipo de dato.
  - Además, podríamos tener estructuras dentro de estructuras, por lo que puede aceptar producciones de `DATA` también.
- `DATATYPE`
  - Va a tener el tipo de dato.
  - Puede ser un tipo de datos reservado `(int, string, float64, bool, [])` o un string no reservado el cual va a significar que hay un objeto que esta referenciando.
  - Permite anidamiento de listas `[]` con su segunda producción.

Es necesario poder separar `S` y `S'`, ya que vamos a querer devolver únicamente la primer estructura definida en nuestra entrada, mientras que el resto solo vamos a tener que insertarlas si se utilizan como hijos de la estructura principal o alguna de subestructuras.

Esta gramática es LALR(1), para verificar esto utilizamos la herramienta Grammophone[1].

2.3. Implementación

`YACC` nos permite definir el comportamiento a realizar dentro de una producción, por lo que nuestro enfoque fue definir distintas clases de Python que representaran a nuestros objetos de dominio (casi 1:1 con los nombres de las producciones). Estas clases `(MainStructure, OtherStructure, Struct, Variable, DataType, ListType)` tienen definidas dentro de cada una, el comportamiento a realizar para poder imprimir en la salida el resultado esperado y cuyos nombres son explícitos.

El trabajo requería que ante entradas inválidas el programa falle, y que su valor de retorno en ese caso incluya una descripción que permita determinar la posible ubicación del error. Esto implica generar un error cuando la sintaxis de la estructura de entrada no es correcta, como también si se intenta utilizar estructuras no definidas, o que definan dependencias circulares y sean utilizadas por la estructura principal.

Teniendo la entrada definida a partir de las estructuras de datos descriptas, podemos detectar referencias circulares utilizando un algoritmo basado en DFS, que ejecutamos sobre el grafo de dependencias, en dónde cada estructura es un nodo y dos nodos están conectados si una estructura incluye a la otra (el grafo es generado por la estructura principal y los elementos de tipo `Struct` saben calcular sus dependencias directas).

3. Conclusiones

Para la realización de este trabajo utilizamos todo el conocimiento adquirido a lo largo de la materia para diseñar una gramática a partir de la cual podamos tokenizar las cadenas de entrada, para luego definir el comportamiento de sus producciones.

Esto nos permitió partir de una cadena de caracteres con un formato especifico y poder generar una estructura de datos tipo JSON equivalente a lo que esta cadena definía.

Referencias

[1] Grammophone - `https://mdaines.github.io/grammophone/`