Trabajo Práctico de Autómatas, Teoría de Lenguajes y Compiladores: Especificaciones PenguinScript

Equipo:

Nombre	Apellido	Legajo	Correo
Alejo	Padula Morillo	63571	apadulamorillo@itba.edu.ar
Bernardo	Zapico	62318	bzapico@itba.edu.ar
Franco	Morroni	60417	fmorroni@itba.edu.ar
Tobias	Pugliano	62180	tpugliano@itba.edu.ar

Repositorio

El proyecto estará versionado en [nuestro repositorio de GitHub]

Dominio

Desarrollar un lenguaje que permita generar una representación gráfica de un autómata finito determinista utilizando la definición de la gramática regular equivalente. Además se espera que el lenguaje pueda recibir una palabra y dado un AFD obtener una representación visual (gif) de como se va consumiendo la palabra.

Para definir las gramáticas se procederá a definir los conjuntos que las forman (conjunto de símbolos no terminales, alfabeto (símbolos terminales), producciones y el símbolo inicial).

En cuanto al alfabeto, este estará acotado a caracteres y/o cadenas ascii, entendiendo que se puede realizar una simple codificación de los símbolos no ascii a cadenas ascii (Ej: si se quiere usar ξ como símbolo se lo puede mapear a la cadena ascii “xi”).

Este lenguaje permitirá obtener una rápida representación visual y observar el comportamiento del autómata.

Construcciones

Nuestro lenguaje les ofrecerá las siguientes construcciones:

- Tipos:
  - Primitivos: símbolo, producción
  - Compuestos: gramática, conjunto de símbolos, conjunto de producciones, lenguaje de una gramática.
- Definición de variables: **X** = **Y**
  - Donde **X** es el identificador de la variable e **Y** es un valor de alguno de los tipos compuestos especificados arriba, o una expresión que evalúe a uno de ellos.
  - Una gramática se define como: < **x**, **y**, **z**, **w** >
    - Donde **x** es un conjunto de símbolos (el alfabeto), **y** es un conjunto de símbolos (terminales), **z** es un conjunto de producciones y **w** es un símbolo (que se verificará que pertenezca a **y**).
  - Un conjunto de símbolos se define como: { $\mathbf{a_1}$, $\mathbf{a_2}$, ..., $\mathbf{a_n}$ }
    - Donde los $\mathbf{a_i}$ deben estar compuestos por caracteres ascii.
  - Un conjunto de producciones se define como: { $\mathbf{A_1 \rightarrow b_1}$, $\mathbf{A_2 \rightarrow b_2}$, ..., $\mathbf{A_n \rightarrow b_n}$ } (opcionalmente alguna o varias de las clausuras podría ser de la forma $\mathbf{A \rightarrow b_1 \mid b_2 \mid ... \mid b_n}$), donde:
    - Los $\mathbf{A_i}$ verifican que pertenecen a **y**

        - Los $\mathbf{b_i}$ son de la forma: **cD**, **c** o **lambda** (con **c** en **x** y **D** en **y**)

    Notar que se define de esta forma una gramática regular lineal por derecha.
- onExpresiones:
    - Operaciones de conjuntos del mismo tipo (símbolos o producciones):
        - Unión: **u**
        - Intersección: **n**
        - Resta: **-**
- Sea G una gramática definida, **L(G)** representa el lenguaje de G. Se definen las siguientes operaciones:
    - Unión: **u**
    - Intersección: **n**
    - Resta: **-**
    - Concatenación: **.**
    - Complemento: **¬**
    - Reverso: **LR(G)**
- Símbolos reservados: **{**, **}**, **=**, **<**, **>**, **->**, **lambda**, **u**, **n**, **-**, **|**, ‘ ‘ (espacio), **.** , **L**, **LR**, **¬**

Funcionalidades

Nuestro lenguaje les ofrecerá las siguientes funcionalidades:

- Obtener una representación visual (grafo) de los autómatas definidos.
- Dada una cadena y una gramática en Σ* obtener una representación visual del autómata consumiendo la misma.
- Se podrá obtener la representación de la minimización de una AFD dado.
- Se podrá obtener la expresión regular equivalente a la gramática.
- Se podrán definir varias gramáticas en un mismo archivo.
- Se podrá operar sobre los lenguajes de las gramáticas definidas (concatenación, unión, intersección, complemento y reverso) y conseguir el grafo del autómata resultante.

Casos de Prueba

Proponemos los siguientes casos de prueba de aceptación:

- Un programa que obtenga el grafo de un AFD dada su gramática regular.
- Un programa que obtenga el grafo del AFD minimal de un AFD dada su gramática regular..
- Un programa que dada una gramática, genere su AFD equivalente, reciba una cadena y se obtenga la representación (camino en un grafo) de cómo se consume la misma.
- Definir dos o más gramáticas y obtener sus AFD correspondientes.
- Un programa que genere la expresión regular equivalente de una gramática regular.
- Un programa que dada una gramática obtenga el AFD equivalente al reverso del lenguaje que describe.
- Un programa que dada una gramática obtenga el AFD equivalente al complemento del lenguaje de la gramática.
- Definir dos gramáticas y dados sus dos lenguajes generados, obtener el AFD equivalente a la concatenación de dichos lenguajes.
- Un programa que dadas tres gramáticas se obtenga el AFD equivalente al lenguaje dado por la concatenación de las tres gramáticas.
- Un programa que dados dos lenguajes se obtenga la expresión regular de la unión, intersección o resta de los mismos.

Proponemos los siguientes casos de prueba de rechazo:

- Un programa malformado
- Se ingresa una gramática que no es tipo 3.
- Se ingresa una palabra que no pertenece al lenguaje del AFD.
- Se define un conjunto de producciones lineal a la izquierda (Ej: P = { S -> Ab })
- El conjunto de símbolos terminales tiene intersección no vacía con el conjunto de símbolos no terminales.

Ejemplos

Definimos una gramática para obtener el AFD equivalente:

```
// gramatica.ps
G = < sigma, N, P, S >
// Definimos una gramática donde sigma = terminales, N = no terminales,
// P = productorias y S el símbolo inicial
sigma = { a, b, c }
N = { S, A }
P = { S -> bA | c, A -> a }
```

Con el siguiente comando se obtiene una imagen G.png con el grafo del AFD:

```
$ penguin gramatica.ps
```

Se obtiene una imagen mi_autómata.png con el grafo del AFD:

```
$ penguin gramatica.ps -o mi_autómata.png
```

Se obtiene una imagen G-min.png con la representación minimal:

```
$ penguin gramatica.ps --minimal
```

Se obtiene un gif G.gif de una cadena siendo consumida por el AFD:

```
$ penguin gramatica.ps --string=“abbcaaa”
```

Se obtiene la expresión regular equivalente a las gramáticas definidas en el archivo:

```
$ penguin gramatica.ps -e
```

Definimos dos gramáticas en el mismo archivo:

```
// gramatica.ps
G = < sigma  , N,   P, S  >
N   = { S, A }
P={S->bA|c, A -> a }
sigma = { a, b, c }

// Reutilizamos N y S
OtraGram = < sigma2, N, P2, S >
sigma2 = { c, p }
P2 = { S -> cA | c, A -> cp, A -> pA }
```

Se obtendrían dos imágenes G.png y OtraGram.png

```
$ penguin gramatica.ps
```

Operaciones de conjuntos:

```
// gramatica.ps
G = < sigma, N, P, S >
N = { S, A, B } - { B, C, D }
P = { S -> bA | c, A -> a }
sigma = { a, b, c }

// Reutilizamos N y S
OtraGram = < sigma2, N, P2, S >
sigma2 = { c, p } u sigma
P2 = P u { S -> cA | c, A -> cp, A -> pA }
```

Operaciones con lenguajes:

```
// gramatica.ps
G = < sigma, N, P, S >
N = { S, A, B }
P = { S -> bA | c, A -> a }
sigma = { a, b, c }

G2 = < sigma2, N, P2, S >
sigma2 = { c }
P2 =  { S -> c }

L1 = L(G) u L(G2)  //unión de los lenguajes de G1 y G2
L2 = L(G).L1       //concatenación
L3 = ¬ L2          //complemento
```

Ejemplos de rechazo

Falla por tener un elemento extra en la definición de la gramática

```
G = < sigma, N, P, S, Q >
N = { S, A }
P = { S -> bA | c, A -> a }
sigma = { a, b, c }
```

Falla por usar un símbolo no definido ni en el alfabeto ni entre los símbolos terminales

```
G = < sigma, N, P, S >
N = { S, A }
P = { S -> bX | c, A -> a }
sigma = { a, b, c }
```

Falla por reutilizar un símbolo terminal en el conjunto del alfabeto

```
G = < sigma, N, P, S >
N = { S, A, a }
sigma = { a, b, c }
P = { S -> bA | c, A -> a }
```

Falla por no ser una gramática de tipo 3

```
G = < sigma, N, P, S >
N = { S, A }
P = { S -> bA | c, A -> AbA }
sigma = { a, b, c }
```

Falla por definir una variable no asociada a ninguna gramática (trataría de parsear N antes de tener G)

```
N = { S, A }
G = < sigma, N, P, S >
P = { S -> bA | c, A -> AbA }
sigma = { a, b, c }
```

Falla por tratar de asignar a una variable de tipo **conjunto de producciones** un valor de tipo **conjunto de símbolos**

```
G = < sigma, N, P, S >
N = { S, A }
P = { a, b, c }
sigma = { a, b, c }
```

Falla por tratar de operar sobre conjuntos de distinto tipo

```
G = < sigma, N, P, S >
N = { S, A }
P = { S -> bA | c, A -> AbA } u { p, q }
sigma = { a, b, c }
```