
UNR Universidad Nacional de Rosario

Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Análisis de Lenguajes de Programación

# Trabajo Práctico Final

*Autor:*
Pablo Antuña

26 de febrero de 2023

# 1. Descripción del proyecto

## 1.1. Introducción

El trabajo elegido es una de las propuestas dadas por la cátedra: **RGrammar**, un interprete de gramáticas regulares. A continuación se desarrolla la idea del TP.

### 1.1.1. Lenguajes formales y gramáticas

Un lenguaje formal es un conjunto de cadenas finitas sobre un alfabeto finito dado. Para definir cuáles son las cadenas aceptadas por un determinado lenguaje y cómo construirlas, pueden utilizarse gramáticas formales. Una gramática formal es una tupla de 4 elementos $(NT, T, s, P)$, donde:

- $NT$ es un conjunto de símbolos no terminales
- $T$ es un conjunto de símbolos terminales
- $s \in NT$ es el no terminal inicial
- $P$ es un conjunto de reglas de producción

En 1956, Noam Chomsky definió una clasificación jerárquica para gramáticas formales que lleva a clasificar también los lenguajes que estas definen. De allí, existen 4 clases de lenguajes, cada una contenida en la anterior:

- Lenguajes sin restricciones o de tipo 0
- Lenguajes sensibles al contexto o de tipo 1
- Lenguajes libres de contexto o de tipo 2
- Lenguajes regulares o de tipo 3

La idea es que el TP se enfoque en los lenguajes y, más específicamente, las gramáticas regulares o de tipo 3.

### 1.1.2. Lenguajes y gramáticas regulares

Los lenguajes regulares son los más sencillos que se consideran y pueden ser procesados por un autómata finito. En sus gramáticas, las reglas de producción tienen únicamente un no terminal en el lado izquierdo y un solo terminal, posiblemente acompañado por un no terminal, en el lado derecho. También se permite en el lado derecho la cadena vacía ($\lambda$). Algunos ejemplos de estas reglas podrían ser:

- $A \to aB$
- $A \to c$
- $B \to Cb$
- $B \to \lambda$
- $\sigma \to aA$

Este conjunto de gramáticas se divide a su vez en dos subconjuntos: *izquierdas*, aquellas en las que en el lado derecho de las producciones el no terminal va a la izquierda; y *derechas*, en las cuales el

no terminal va a la derecha. Si bien lo más común es encontrar gramáticas derechas, puesto que son más intuitivas y simples de entender, también se utilizan mucho las izquierdas.

### 1.1.3. Propiedades de cierre

Otra característica importante de los lenguajes regulares es que son cerrados bajo ciertas operaciones. Esto significa que si uno o varios lenguajes son regulares, entonces determinados lenguajes relacionados con estos también lo son.
Esta particularidad hace que sea práctico y sencillo manipularlos para construir nuevos lenguajes a partir de lenguajes existentes.
Algunas de estas operaciones son:

- **Unión:** equivalente a la unión de conjuntos. Si $L_1$ y $L_2$ son regulares, entonces $L_1 \cup L_2$ también es regular.
- **Intersección:** equivalente a la intersección entre conjuntos. Si $L_1$ y $L_2$ son regulares, entonces $L_1 \cap L_2$ también es regular.
- **Complemento:** equivalente al complemento de un conjunto, donde el universo es el conjunto de todas las cadenas finitas que se pueden formar con los símbolos del alfabeto. Si $L$ es regular, entonces $\overline{L}$ también es regular.
- **Reversa:** la reversa de un lenguaje $L$ incluye las reversas de todas las cadenas pertenecientes a él, donde la reversa de una cadena $a_0a_1a_2...a_n$ es la cadena $a_na_{n-1}...a_1a_0$. Si $L$ es regular, entonces $L^R$ también es regular.
- **Concatenación:** dados dos lenguajes $L_1$ y $L_2$, su concatenación incluye todas las cadenas $c_1c_2$, donde $c_1 \in L_1$ y $c_2 \in L_2$. Si $L_1$ y $L_2$ son regulares, entonces $L_1L_2$ también es regular.

## 1.2. Objetivo del TP

Se dice que dos gramáticas son equivalentes si aceptan el mismo lenguaje. Pero esto no es algo fácil de determinar a simple vista, ya que dos gramáticas equivalentes pueden verse muy distintas a simple viste. Esto se debe a que la misma gramática puede expresarse con distinta cantidad de no terminales y reglas, e incluso las mismas reglas pueden escribirse de diferentes formas.
Resulta interesante entonces tener un método automático que pueda decidir su equivalencia. Esta es la principal inspiración de este proyecto. Para gramáticas de tipo 0, 1 y 2 esto no es decidible. Es por esto que se hace foco en las de tipo 3.
Entonces como objetivo del TP queremos tener un lenguaje para manipular las gramáticas regulares que sea capaz de realizar las operaciones descriptas en la sección anterior y principalmente decir si dos gramáticas son equivalentes.

# 2. Instalación

Para la instalación vamos a requerir tener previamente instalado en nuestro sistema:

- Haskell
- Stack
- Happy

Luego deberemos descargar el código fuente que se encuentra disponible en el repositorio de aquí

Una vez descargado nos situamos con la terminal en la carpeta raíz y deberemos ejecutar las siguientes instrucciones

- `stack setup`
- `stack build`
- `stack install`

Y finalmente podremos ejecutar `ALP-FINAL-exe`.

Como alternativa se puede simplemente utilizar `stack run` en lugar de todos los pasos anteriores.

# 3. Manual de uso

El uso de este proyecto consiste en dos partes. Primero deben cargarse gramáticas, para esto utilizamos archivos de extensión `.grm`. La segunda parte es utilizar las gramáticas cargadas desde los archivos para generar nuevas a través de las operaciones que permite el programa, para consultar la pertenecia de alguna palabra al lenguajes que representa alguna de las gramáticas o para consultar equivalencia de dos gramáticas para saber si los lenguajes que representan son equivalentes.
Indaguemos un poco en cada uno de sus 2 partes por separado

## 3.1. Archivos de gramáticas

Los archivos de las gramáticas como ya se mencionó antes, son archivos de extensión `.grm` que permiten cargar una gramática a nuestro programa.
¿Cómo creamos un archivo de gramática? Bueno la sintaxis es bastante sencilla. Escribir una gramática consiste en escribir sus reglas de producción con algunas cosas a tener en cuenta

- Los terminales deben escribirse entre comillas
- Los no terminales deben escribirse sin comillas
- El símbolo inicial debe ser &
- Cada regla de producción debe finalizar con un ;
- Se permite el uso de | para escribir en forma BNF
- Se utiliza la flecha $->$ para simbolizar la flecha que indica "este simbolo no terminal produce..."

Con esos detalles aclarados ya somos capaces de escribir nuestra gramatica como un conjunto de reglas de producción. La gramática a escribir puede ser tanto izquierda como derecha. A continuación un ejemplo de cada una

*Derecha*

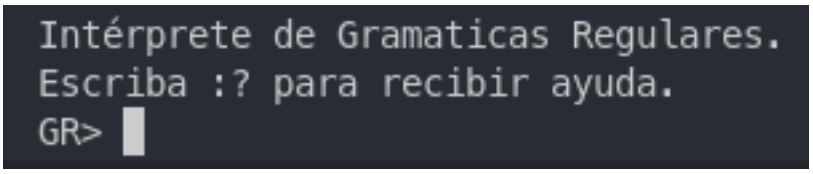
```
& -> "a" A | "b" B | \;
A -> "b" B | "a" &;
B -> "b";
```

*Izquierda*

```
& -> A "b" | B "a";
A -> B "b" | A "a";
B -> & "a" | \;
```

## 3.2. Intérprete

La segunda parte del programa es la parte interactiva, el interprete en el cual podemos realizar operaciones y consultas. Para ejecutarlo tenemos que estar con la terminal situados en la carpeta raíz del proyecto y utilizar el comando `stack run` (o realizar todo el proceso detallado en la sección de instalación). Una vez ejecutado el programa veremos el interprete como en la imagen




En este punto ya podremos empezar a cargar gramáticas para luego interactuar con ellas. Veamos un poco los comandos disponibles

### 3.2.1. Comandos

Disponemos de 7 comandos detallados a continuación:

El comando `:browse` que nos permite ver en pantalla la lista de los nombres de gramáticas guardadas hasta el momento.
El comando `:load <name> <file>` donde `<name>` y `<file>` son respectivamente el nombre que queremos asignarle a la gramática y la ruta del archivo que contiene la gramática que queremos cargar (por ejemplo `:load miGramatica ./ejemplos/ab.grm` cargará la gramática escrita en el archivo `ab.grm` que se encuentra en la carpeta `ejemplos` y la guardará con el nombre `miGramatica`).
El comando `:print <gram>` donde `<gram>` es el nombre de la gramática que queremos mostrar en pantalla. Si `<gram>` existe entonces el comando va a imprimirla en pantalla, se mostrará como gramática izquierda o derecha según si fue cargada como izquierda o derecha o en el caso de haber sido creada a través de operaciones con otras gramáticas será según si las otras gramáticas eran derechas o izquierda ya que se decide con un valor booleano que será `True` si es derecha o `False` si es izquierda, dicho booleano sera resultado de realizar la operación lógica `and` con los valores de las gramáticas utilizadas para su creación. Esta decisión y comando existen únicamente para darle un uso a la operación de "generar gramática izq (der) a partir de una gramática der (izq)". El comando `:lprint <gram>` funciona de forma similar al comando `:print` pero forzando a ser impresa en pantalla como gramática izquierda.
El comando `:rprint <gram>` funciona de forma similar al comando `:print` pero forzando a ser impresa en pantalla como gramática derecha.
El comando `:help` o `:?` mostrará en pantalla una lista con todos estos comandos y una breve descripción de su funcionalidad.
El comando `:quit` sale del interprete finalizando la ejecución del programa.

### 3.2.2. Operaciones entre gramáticas

Podemos separa las operaciones disponibles en 2: Las de consulta y las de construcción de nuevas gramáticas

Dentro de las de construcción de nuevas gramáticas se encuentran:

- La unión de dos gramáticas `A + B`
- La intersección de dos gramáticas `A . B`
- La resta de dos gramáticas `A - B`
- La concatenación de dos gramáticas `A ++ B`
- El complemento de una gramática `A~`
- La reversa de una gramática `A~~`
- El cambio de lado de una gramática `A!` que la pasa de izq (der) a der (izq)

Y la definición de una gramática a través de las operaciones anteriores es de la forma `<nombre> = <operacion>` donde `<nombre>` es el nombre de la nueva gramática y `<operacion>` es alguna de las operaciones anteriores, por ejemplo `C = A + B`. También se pueden utilizar paréntesis y realizar múltiples operaciones como `D = A - (B + C)`

Dentro de las consultas tenemos:

- Consultar si una cadena pertenece a un lenguaje `"cadena" ? A`
- Consultar si dos gramáticas son equivalentes `A == B`

# 4. Organización de los archivos

El proyecto consiste de 6 módulos y adicionalmente tenemos un parser (trabajado con Happy) y el moduloMain.hs que implementa el ejecutable final (el interprete). Este ultimo modulo se encuentra en la carpeta app mientras que el parser junto a los otros 6 módulos se encuentran en la carpeta src. Los módulos son:

- Common es el modulo que incluye las definiciones de los tipos de datos utilizados. Se encuentra en el archivo Common.hs
- Eval es el modulo que incluye las funciones que evalúan las operaciones de gramáticas (ya sean de consulta o de creación de nuevas gramáticas). Se encuentra en el archivo Eval.hs
- FiniteAutomata es el modulo que contiene todas las funciones relacionadas a los automatas de estado finito, ya sean deterministas o no deterministas. Se encuentra en el archivo FiniteAutomata.hs
- Grammar es el modulo que incluye todas las funciones para la manipulación de las gramáticas. Se encuentra en el archivo Grammar.hs
- Monads es el modulo en el cual se definen las clases de mónadas. Se encuentra en el archivo Monads.hs
- PPrint es el modulo correspondiente al Pretty Printer que se encarga de mostrar en pantalla las gramáticas solicitadas. Se encuentra en el archivo PPrint.hs

En nuestra organización encontramos también una carpeta de ejemplos con algunos archivos .grm para probar el programa

# 5. Decisiones de diseño

## 5.1. Estructuras utilizadas

La estructura utilizada para parsear la entrada de una gramática la llamada `Gram` que hace uso de otras estructuras, todas ellas a continuación:

```
newtype GramTerm = Gram [Rule]
 deriving (Eq, Ord, Show)

type Gram = Either GramTerm GramTerm

data Rule = Rule NT [RigthSide]
 deriving (Eq, Ord, Show)

data NT = NT String | Initial
 deriving (Eq, Ord, Show)

newtype T = T {runT :: String}
 deriving (Eq, Ord, Show)

data RigthSide = RT T | RTNT T NT | RL
 deriving (Eq, Ord, Show)
```

La elección de estas estructuras la tuve luego de un primer intento donde utilizaba una estructura un poco más complicada de trabajar similar a la estructura `LamTerm` del TP3 de la materia. Debido a las complicaciones que surgieron al intentar trabajarla busque algo que no fuese complicado de parsear y que fuese mucho más cómodo a la hora de trabajar con ella. Use provecho de la mónada `Either` para representar si una gramática iba a ser `Left` (izquierda) o `Right` (derecha) e hice uso de listas para tener más facilidad a la hora de requerir ver las reglas de producción de una gramática.

Luego tenemos las estructuras para las operaciones, las cuales también están inspiradas en los distintos trabajos realizados en la materia. Esta forma resultaba sencilla de parsear y facil de trabajar a la hora de evaluar cada una

```
data Op = OpDef Name OpGram
        | OpIn String OpGram
        | OpEqual OpGram OpGram

data OpGram = OpGram Name
            | OpUnion OpGram OpGram
            | OpIntersec OpGram OpGram
            | OpDiff OpGram OpGram
            | OpConcat OpGram OpGram
            | OpComplement OpGram
            | OpReverse OpGram
            | OpSide OpGram
```

Por otro lado tenemos las estructuras necesarias para los autómatas. Para esto cree una estructura para los símbolos pertenecientes a un autómata no determinista y otra para los símbolos de un autómata determinista, estructuralmente no difieren en mas que el constructor, pero conceptualmente van a diferir en que en los símbolos deterministas no puedo tener el símbolo vacío ya que no existen las transiciones vacías en los autómatas deterministas (las lambda/empty transitions). Luego tenía que pensar en como representaría los estados y para generalizar las cosas y sabiendo que probablemente resultara mas cómodo para trabajar con distintos tipos de datos (como por ejemplo a la hora de pasar un autómata no determinista a determinista que sabemos que los estados pasan a ser conjuntos de estados entonces de esta forma podría resultar mas intuitivo el pasaje), más allá de eso el estado es simplemente un constructor `St` con el tipo dado `a`. Las ultimas estructuras a pensar antes de componerlas para formar los autómatas eran las relaciones de transición y las funciones de transición, que terminaron siendo simplemente un constructor con una lista de 3-uplas donde difieren en el tipo de símbolos que utilizan y cada tupla representa una transición `(A, x, B)` representa la transición del estado `A` al estado `B` consumiendo el símbolo `x`. Finalmente construimos las estructuras para los automatas como la combinacion de las estructuras mencionadas y agregamos un valor booleano que nos indica si esa gramática por defecto debe ser a derecha (True) o a izquierda (False), esto es simplemente para darle una funcionalidad a la operación de cambiar de lado la gramática, de esta forma utilizando el comando `:print` utilizamos el valor de ese booleano para imprimir de una forma u otra sin necesidad de forzar la impresión en la forma deseada como realizariamos con los comandos `:rprint` y `:lprint`.

```
newtype SimbND = SimbND {runSimbND :: String}
  deriving (Eq, Ord, Show)

newtype SimbD = SimbD {runSimbD :: String}
  deriving (Eq, Ord, Show)

newtype St a = St {runSt :: a}
  deriving (Eq, Ord, Show)

newtype RelT a = RelT [(St a, SimbND, St a)]
  deriving Show

newtype FunT a = FunT [(St a, SimbD, St a)]
  deriving Show

data AEFD a = D [SimbD] [St a] (FunT a) [St a] (St a) Bool
  deriving Show

data AEFND a = ND [SimbND] [St a] (RelT a) [St a] (St a) Bool
  deriving Show
```

Ya sobre el final del trabajo de casualidad me encontré con unos casos donde la impresión de la gramática quedaba rara. En algunas gramáticas veía que no imprimía alfabeto alguno lo cual tenía sentido dado que obtenía los terminales de la gramática a través de las reglas de la misma pero

en estas gramáticas solo estaba una regla de producción desde el símbolo inicial a la cadena vacía. Este caso se dio en la intersección de dos gramáticas cuyos alfabetos eran distintos. Si bien el error era visual y no afectaba al funcionamiento ya que toda operación a realizarse se realizaba con los autómatas, seguía siendo un error, ya que conceptualmente sería correcto que si bien la intersección es vacía el alfabeto sea la unión de los alfabetos de las gramáticas a intersecar. Debido a esto decidí agregar una estructura nueva que solo se utiliza al momento de la impresión de la gramática en pantalla y la misma es muy similar a la estructura de la gramática mostrada anteriormente con la diferencia de llevar una lista de terminales, es decir, el alfabeto. De esta forma podía imprimir el alfabeto sin buscarlo en la lista de reglas

```
data GramShowTerm = GramShow [T] [Rule]
 deriving (Eq, Ord, Show)

type GramShow = Either GramShowTerm GramShowTerm
```

## 5.2. Guardado en el entorno

Una de las decisiones a hacer en el proyecto es como guardar las gramáticas cargadas o generadas en mi programa, es decir, como será mi entorno de variables. Decidí crear un nuevo tipo que sea un autómata determinista con estados de tipo `Int` para estandarizar a un mismo tipo de dato ya que debía almacenar todos los datos del mismo tipo. Podría haber tomada la decisión de almacenar la gramática, o el autómata no determinista, pero decidí almacenar el autómata determinista porque considere que la mayoría de las operaciones a realizarse podrían llegar a ser todas a través del autómata determinista y de esta forma ahorrar costo de pasar de una estructura a autómata determinista constantemente.

```
type Name = String

type AEFDG = AEFD Int

type Env = [(Name, AEFDG)]
```

## 5.3. Observaciones

Como algunas observaciones adicionales me gustaría mencionar un poco las limitaciones del programa. El programa no fue creado con el objetivo de ser eficiente por lo cual hay cosas a mejorar. En particular la minimización de los autómatas puede resultar bastante costosa ante varios estados. No todos los algoritmos utilizados son óptimos y algunos dependen del tipo de gramática que se manipule, como se mencionó anteriormente el de la minimización puede terminar siendo muy costoso en gramáticas más grandes. Debido a la complejidad de estos temas no se consideró como objetivo hacer el programa lo más eficiente posible pero sería una buena idea y un proyecto interesante rehacer o mejorar este interprete para generar un código más eficiente y así tener la posibilidad de manipular más gramáticas sin problemas. En particular considero que el mayor cuello de botella se encuentra en los algoritmos de los autómatas (tales como minimización, eliminación de determinados tipos de estados o incluso pasaje de un tipo de autómata a otro) no obstante seguramente se puede también mejorar otras

secciones del código o incluso representar los datos con mejores estructuras que nos ayuden a cumplir este objetivo

# Referencias

[1] Jay Bagga NFA to DFA Conversion Algorithm

[2] Department of Computer Science, Wellesley College. DFA Operations

[3] UNR, FCEIA, Mauro Jaskelioff, LCC 2021, Apuntes de clase de la materia ALP

[4] UNR, FCEIA, Pablo Verdes, LCC 2020, Apuntes de clase de la materia LFyC

[5] Equivalence of Regular Grammars - StackExchange

[6] Conversion From NFA to DFA - GeeksForGeeks

[7] Existences of NFA of a Reverse of a Language - StackExchange

[8] Reversal Process In DFA - Geeks For Geeks

[9] Left Regular Grammar To NFA - StackOverflow

[10] DFA Minimization - Wikipedia

[11] Minimization of DFA - GeeksForGeeks

[12] UNR, FCEIA, LCC 2021, TP3 ALP

[13] UNR, FCEIA, LCC 2021, TP4 ALP

[14] UNR, FCEIA, LCC 2022, TP de la materia Compiladores