Facultad de Ciencias Exactas, Ingeniería y Agrimensura - UNR
Departamento de Ciencias de la Computación

RGrammar: Intérprete de Gramáticas Regulares
Trabajo Final

ANÁLISIS DE LENGUAJES DE PROGRAMACIÓN

Realizado por

Bini, Valentina María

Legajo: B-5926/9

2020

Índice

1 Descripción del proyecto **1**
- 1.1 Introducción . . . 1
  - 1.1.1 Lenguajes formales y gramáticas . . . 1
  - 1.1.2 Lenguajes y gramáticas regulares . . . 1
  - 1.1.3 Propiedades de cierre . . . 1
- 1.2 Objetivo . . . 2
- 1.3 Funcionalidad . . . 2

2 Manual de uso **2**
- 2.1 Archivos *.grm* . . . 2
  - 2.1.1 Símbolos terminales . . . 3
  - 2.1.2 Símbolos no terminales . . . 3
  - 2.1.3 Reglas de producción . . . 3
  - 2.1.4 Gramáticas . . . 3
- 2.2 Intérprete . . . 3
  - 2.2.1 Carga de archivos e impresiones en pantalla . . . 4
  - 2.2.2 Operaciones entre gramáticas . . . 4
  - 2.2.3 Consultas y asignaciones . . . 4

3 Decisiones de diseño **5**
- 3.1 Almacenamiento de los datos . . . 5
- 3.2 Manejo de gramáticas con distintos alfabetos . . . 5
  - 3.2.1 Comportamiento del complemento . . . 5
- 3.3 Optimización . . . 6

4 Estructuración del código **6**

5 Descarga e instalación **6**

6 Bibliografía **7**

1 Descripción del proyecto

1.1 Introducción

Para comenzar, se hará una breve descripción del dominio en el cual está desarrollado el proyecto, para luego introducir el objetivo y la funcionalidad que este posee.

1.1.1 Lenguajes formales y gramáticas

Los lenguajes formales son uno de los fundamentos principales de las ciencias de la computación. Se utilizan para definir las sintaxis de lenguajes de programación, sistemas axiomáticos y formalismos matemáticos, y también para representar problemas de decisión y clasificarlos según su complejidad, entre otros fines.

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.

Este trabajo se enfocará 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 (λ). 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 otros.

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

Se dice que dos gramáticas son equivalentes si aceptan el mismo lenguaje. A la hora de trabajar con gramáticas, muchas veces resulta difícil saber si el resultado obtenido es el esperado. Puede suceder que se llegue a dos gramáticas que aparentemente son diferentes, pero si se analizan más en detalle se logra notar que son equivalentes. Esto se debe a que la misma gramática puede expresarse con distinta cantidad de no terminales y reglas. Según cómo haya pensado el problema la persona que lo resolvió, estas variables podrían cambiar.

Resulta interesante entonces tener un método automático que pueda decidir su equivalencia. Esta fue 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. Una vez acotado el dominio a esto, resultó evidente que podían realizarse más operaciones con estas gramáticas, aprovechando su clausura respecto de estas.

El objetivo del trabajo es, por ende, facilitar la tarea y la revisión de la misma a aquellas personas que estén trabajando con lenguajes y gramáticas regulares.

1.3 Funcionalidad

El proyecto consta de un intérprete para gramáticas regulares. Por un lado, tiene un *parser* que permite cargar gramáticas desde archivos de tipo *filename.grm*, las cuales pueden ser tanto izquierdas como derechas. Por otra parte, permite operar con las gramáticas cargadas creando nuevas, y también da la posibilidad de hacer consultas. Estas últimas pueden ser de dos tipos: de *equivalencia*, que permiten saber si dos gramáticas aceptan el mismo lenguaje; o de *pertenencia*, las cuales permiten saber si una cadena es aceptada por una gramática o no.

Las operaciones que admite son *unión*, *intersección*, *resta*, *reversa*, *complemento* y *concatenación*. Como se vio antes, todas estas producen gramáticas regulares si sus argumentos lo son.

Esta herramienta resulta útil a la hora de construir gramáticas y validar la corrección de las mismas. Si, por ejemplo, se quiere construir un lenguaje que consta de dos partes, se puede construir ambas por separado y luego dejar que el intérprete genere la unión. A su vez, si se prefiere realizar la unión por cuenta propia de la manera que resulte más intuitiva, luego se puede consultar la equivalencia de la gramática obtenida con la generada por el programa para verificar que sea correcta. Además, utilizar las consultas de pertenencia puede ser muy útil a la hora de checkear que la gramática que se creó acepte realmente las cadenas que se esperan.

Por último, el intérprete permite imprimir en pantalla las gramáticas guardadas eligiendo hacia qué lado se muestran. Esto puede servir si se necesita convertir una gramática derecha en izquierda o viceversa.

2 Manual de uso

Se dividirá el manual en varias secciones. Primero se verá la manera de crear gramáticas en archivos *.grm*, luego la forma de cargar estos archivos al intérprete y, por último, se describirá el modo de operar con las gramáticas cargadas.

2.1 Archivos *.grm*

Un archivo puede contener solo una gramática. Fuera de esta, únicamente puede haber comentarios. Los comentarios de una línea se realizan con `--`. Si se quieren comentar varias lineas juntas, se abre con `{-`

y se cierra con `-}`. En el archivo únicamente se explicitan las reglas de producción de la gramática. Los conjuntos T y NT se deducen de las reglas y el no terminal inicial siempre es `&`.

2.1.1 Símbolos terminales

Los símbolos terminales (T) pueden ser cadenas del largo que se desee (pero no vacías) y van encerrados entre comillas dobles. Se permite cualquier caracter dentro de un T, excepto las comillas que delimitan sus extremos y los espacios. Si hay espacios dentro de un T, estos serán ignorados.

Algunos ejemplos de T pueden ser:

- `"a"`
- `"ab"`
- `"++"`
- `"x,y"`
- `"x , y"`

Nótese que los dos últimos ejemplos serán considerados iguales dado que los espacios serán ignorados.

2.1.2 Símbolos no terminales

Todos los no terminales (NT), excepto el inicial, son cadenas del largo que se desee, solo con caracteres alfanuméricos. El único que no será alfanumérico es el inicial, que será siempre `&`. Vale aclarar que este caracter NO es permitido dentro de otros NT.

2.1.3 Reglas de producción

Una regla de producción consta de un lado izquierdo y un lado derecho. Ambos lados van separados por una flecha hacia la derecha: `->`. Cada regla de producción debe terminar con un punto y coma.

Del lado izquierdo va únicamente un NT. Del lado derecho, pueden ir varias "opciones" separadas por una barra vertical: `|`. Cada una de estas "opciones" puede tener uno de los siguientes formatos:

- Un T y un NT (el orden dependerá de si la gramática es derecha o izquierda).
- Un T solo.
- La cadena vacía, representada con una barra invertida: `\`.
  Aclaración: la cadena vacía NO es considerada un T, por lo que no puede ir junto con un NT.

Así, un ejemplo de una regla de producción derecha podría ser: `& -> "a" A | "b" | \;`.

2.1.4 Gramáticas

Una gramática será una secuencia de producciones. A continuación se muestra un ejemplo de una gramática izquierda y una gramática derecha completas:

Derecha

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

Izquierda

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

2.2 Intérprete

Al correr el intérprete, se muestra un mensaje de inicio y luego el *prompt*, donde podemos empezar a escribir comandos:

```
Regular Grammar interpreter.
Enter :?  to receive help.
RG>
```

El intérprete tiene 7 comandos básicos. `:?  o :help` mostrará una lista con los comandos principales y una breve descripción de lo que hacen. `:browse` muestra en pantalla la lista de todos los nombres de las

gramáticas guardadas hasta el momento. `:quit` se utiliza para salir y cerrar el intérprete. En el momento que se realiza esta acción, todas las gramáticas guardadas se pierden. Al reiniciar el intérprete, este estará vacío.

Por último, están `:rload`, `:lload`, `:rprint` y `:lprint`, los cuales se explicarán en la siguiente sección.

Todos los comandos pueden ser abreviados por `:c`, donde `c` es una parte del inicio del comando. Se requieren tantos caracteres como sean necesarios para que solo haya una opción que sea compatible. Por ejemplo, en el caso de `:quit` basta con `:q`. Pero si se ingresara `:r`, podría ser `:rload` o `:rprint`. En este caso se requiere ingresar un caracer más para eliminar la ambigüedad.

2.2.1 Carga de archivos e impresiones en pantalla

Hay dos comandos para cargar archivos: `:rload` y `:lload`, los cuales cargan gramáticas derechas e izquierdas respectivamente. Estos comandos tienen el siguiente formato: `RG> cmd name file`, donde `name` es el nombre con el que la gramática se guardará en el *scope* y `file` es el nombre del archivo (o la ruta si no se encuentra en la carpeta actual). Solo se podrán cargar archivos cuya extensión sea *.grm* y el nombre únicamente puede contener caracteres alfanuméricos. Caso contrario se mostrará un error de nombre de archivo inválido (`Invalid filename`) o nombre inválido (`Invalid name`), respectivamente. Si se carga un archivo con un nombre de una gramática que ya existe, esta será reemplazada por la nueva recién cargada.

Para imprimir una gramática, también se debe indicar si se muestra como izquierda o derecha. Por eso existen, también para esto, dos comandos: `:rprint` y `:lprint`, los cuales van seguidos del nombre de la gramática que se quiere imprimir. Se imprimirá primero el alfabeto de la gramática y luego las reglas de producción. En la sección 3.2 se explica el porqué de esta decisión.

2.2.2 Operaciones entre gramáticas

A continuación se describirá la notación de las operaciones entre gramáticas. Sean `A` y `B` dos gramáticas que están ya cargadas. Se define:

Unión de `A` y `B`:	`A + B`
Intersección entre `A` y `B`:	`A ! B`
Complemento de `A` (sobre el alfabeto actual):	`A'`
Diferencia entre `A` y `B`:	`A - B`
Reversa de `A`:	`A~`
Concatenación de `A` y `B`:	`A . B`

Estas operaciones pueden anidarse entre sí, utilizando paréntesis para encapsular operaciones dentro de otras. Las operaciones unarias son las de mayor precedencia. Dentro de las binarias, la primera es la concatenación, luego viene la intersección y, por último, la unión y la diferencia con en el mismo nivel. Las operaciones con igual precedencia se realizan en orden de izquierda a derecha. Todas las operaciones binarias asocian a izquierda.

El orden de precedencia es, en resumen, de mayor a menor: `(~,'), ., !, (+,-)`.

En la sección 3.2 se detalla un aspecto importante de las operaciones y cómo estas se realizan.

2.2.3 Consultas y asignaciones

Las operaciones anteriormente introducidas pueden ser utilizadas dentro de una asignación o una consulta.

Una *expresión* referencia una cierta gramática. Puede estar compuesta simplemente por el nombre de la misma o por una combinación de gramáticas guardadas utilizando las operaciones.

Las asignaciones son de la forma `RG> name = exp`, donde `name` es el nombre con el que se guardará la gramática obtenida y `exp` es una expresión. Si el nombre ya existe, la gramática existente será reemplazada por la nueva. Los nombres de gramáticas solo pueden contener caracteres alfanuméricos.

Las consultas de *equivalencia* son de la forma `RG> exp == exp`. El operador `==` tiene menos precedencia que todos los mencionados en la sección anterior, por lo que no hacen falta paréntesis en las expresiones. Una consulta de equivalencia podría ser: `RG> A == A~~`.

Las consultas de *pertenencia* se realizan con el operador `?`, el cual también tiene menos precedencia que los mencionados anteriormente. En el lado derecho va una expresión. En el lado izquierdo, por otra parte, va una cadena, la cual se consulta si es o no aceptada por la gramática obtenida del segundo argumento.

Las cadenas van encerradas entre comillas dobles y los diferentes símbolos se separan con espacios. Como se mencionó anteriormente, los T no tienen espacios, pero sí pueden tener más de un caracter, por lo que así

se distingue donde termina uno y empieza otro. La consulta `RG> "a bb c" ? exp` pregunta si la cadena de T `["a","bb","c"]` es aceptada por la gramática obtenida de evaluar `exp`.

Las consultas devuelven valores booleanos, `True` o `False`.

Es importante notar que las consultas y asignaciones no pueden realizarse en la misma línea de comando. Es decir que si ingresamos, por ejemplo, `RG> A = B' == C`, esto generará un error de parseo.

Se puede observar que para realizar consultas no es necesario guardar las combinaciones de gramáticas con un nuevo nombre, puede hacerse la consulta ingresando simplemente la expresión. Pero si se quisiera imprimir la gramática obtenida en pantalla sí se necesitaría obligatoriamente guardarla, puesto que los comandos `:rprint` y `:lprint` solo toman nombres de gramáticas y no expresiones.

3 Decisiones de diseño

3.1 Almacenamiento de los datos

Las gramáticas regulares pueden convertirse en autómatas finitos. Operar con autómatas es mucho más simple que con gramáticas. Es por esto que las gramáticas al ingresarse son transformadas en autómatas finitos no deterministas (NFA), y luego estos se convierten en autómatas finitos deterministas (DFA). Internamente se guardan los DFA asociándolos con sus respectivos nombres y se opera con estos.

Se consideraron dos opciones para almacenar las gramáticas, una era guardarlas como NFA y otra como DFA. La unión, el complemento, la diferencia y la intersección se realizan directamente sobre DFAs, devolviendo un DFA. Pero la reversa y la concatenación deben realizarse sobre NFAs y luego convertir el resultado en DFA, ya que requieren del uso de la cadena vacía. A su vez, para convertir el autómata en gramática, se requiere que sea DFA. Por ser más las veces que se necesita un DFA que un NFA, se tomó la decisión de guardar los datos de esta manera. Así, para hacer la reversa y concatenación se requiere una transformación a NFA (la cual solo es un renombramiento de símbolos) pero se evita convertir a DFA para hacer las demás operaciones y para imprimir en pantalla, lo cual sería mucho más costoso. Además, para las consultas, tanto de equivalencia como de pertenencia, también se utilizan directamente los DFAs.

El tipo de los DFAs es un tipo parametrizado, donde el parámetro es el tipo de los estados. Es por esto que se utiliza una funcion `dfaStandar` que renombra los estados a enteros para poder guardarlos, ya que tienen que tener todos el mismo tipo.

3.2 Manejo de gramáticas con distintos alfabetos

Al inferirse los alfabetos de los terminales que aparecen en las reglas de producción, una cuestión a considerar fue qué hacer cuando se realizan operaciones sobre dos gramáticas con diferentes alfabetos.

En la concatenación es claro que el nuevo alfabeto será la unión de los dos alfabetos originales. Pero, por otro lado, para los algoritmos de unión, resta e intersección, se necesita que los alfabetos sean iguales. Por esto, antes de invocarlos, se realiza una conversión en cada DFA, en la que se les agregan los símbolos que están en el otro operando y no en él. Para esto se agrega un nuevo estado "basura" al cual llegan todos los arcos etiquetados con los nuevos símbolos desde todos los estados preexistentes. Este estado será un estado *sin salida*, a partir de él con todos los símbolos se llegará a él mismo.

En la intersección y en la diferencia, agregar todos los símbolos puede generar que queden más estados "basura" en el resultado, y también, que haya símbolos que están en el alfabeto y no se utilizan en realidad. Pero quitar símbolos en lugar de agregarlos puede generar un autómata no determinista, que luego requeriría volver a convertirse en DFA y esto sería más costoso. Además, el alfabeto representa el "universo" donde se está trabajando, y no los símbolos que efectivamente se utilizan. Es por esto que se eligió la otra opción.

3.2.1 Comportamiento del complemento

El complemento de una gramática se realiza con respecto al alfabeto que posee en el momento. Es importante tener esto en cuenta a la hora de analizar el resultado obtenido al realizar algunas combinaciones de operaciones.

Si la gramática está recién cargada, el complemento se hará sobre el alfabeto inferido. Pero si se realizaron algunas operaciones, puede que el alfabeto sea más amplio que el que aparece en las reglas de producción que se muestran al imprimir, ya que solo se muestran aquellas que sean "útiles". Antes de convertir un DFA en gramática, se eliminan todos los estados *sin salida*.

Por ejemplo, si una gramática acepta cadenas que contengan `"a"` y `"b"`, y otra acepta cadenas que contengan `"b"` y `"c"`, su intersección solo aceptará cadenas que contengan `"b"`. Pero el alfabeto de la intersección será de todas formas `["a","b","c"]`. Al imprimir la intersección a derecha se mostrará solo la siguiente regla de producción:

```
& -> "b" & | \;
```

Al hacer el complemento, se debe recordar que el alfabeto incluye más símbolos que no se ven directamente en las reglas. Es por esto que se decidió incluir el alfabeto en la impresión en pantalla de las gramáticas.

3.3 Optimización

Los algoritmos elegidos para las operaciones no son necesariamente óptimos. Se seleccionaron algoritmos relativamente básicos y que no requieran una implementación muy compleja. El objetivo del trabajo no fue hacerlo eficiente, puesto que el tiempo no es un factor clave al trabajar con estos temas. No está pensado para cargar grandes gramáticas ya que pueden llevar mucho tiempo de procesamiento.

Si se quisiera, como trabajo futuro, podría pensarse la idea de optimizarlo. Hacerlo requeriría reimplementar únicamente algunos algoritmos de autómatas (del módulo FA.hs) y el resto quedaría igual.

4 Estructuración del código

El código está formado por un parser (hecho con Happy), 5 módulos de Haskell y un Main.

Los módulos son:

- Common.hs: incluye todas las definiciones de los tipos utilizados.
- FA.hs: incluye todas las funciones referidas a autómatas finitos.
- Grammar.hs: incluye todas las funciones que manipulan gramáticas.
- Eval.hs: incluye las funciones que evalúan los diferentes *statements*, utilizando las funciones de FA.hs.
- PrettyPrinter.hs: incluye las funciones para mostrar las gramáticas en pantalla con la misma notación que se parsean, incluyendo también el alfabeto.

Main.hs implementa el intérprete utilizando todos los módulos.

En la carpeta *examples* hay algunos achivos de ejemplo de gramáticas derechas e izquierdas.

5 Descarga e instalación

Para la instalación se requiere tener previamente instalado Haskell, Stack y Happy. La descarga se realiza del repositorio RGrammar. Una vez descargado, en consola dentro de la carpeta se ejecuta:

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

Una vez hecho esto se puede llamar al intérprete desde cualquier parte con `rgrammar`.

6 Bibliografía

1. Jielan Zhang, Zhongsheng Qian, The Equivalent Conversion between Regular Grammar and Finite Automata, Journal of Software Engineering and Applications, 2013, 6, 33-37.

2. Apuntes de clase, Pablo Verdes, “DFA”, Lenguajes Formales y Computabilidad, Departamento de Ciencias de la Computación, UNR.

3. DFA Minimization, Wikipedia.

4. Minimization of DFA, GeeksForGeeks.

5. Apuntes de clase, "DFA Operations", Department of Computer Science, Wellesley College.

6. Trabajo Práctico 3, Análisis de Lenguajes de Programación, Departamento de Ciencias de la Computación, UNR.