Teoría de Lenguajes: Grupo 1

Tomás Miguez, Martin Yoel Belgorodsky

December 1, 2023

Contents

1 Introducción

En este informe explicamos la resolución del ejercicio entregable 2, que incluye la creación de una gramática libre de contexto que interpreta expresiones regulares. Junto con la creación de un analizador léxico (lexer) y un analizador sintáctico (parser).

2 Gramatica

2.1 Definición

Se nos pide dar una gramática libre de contexto para interpretar cadenas que sean expresiones regulares. Creamos la siguiente gramática:

$$G =< N, T, R, P >$$

$$N = \{R, U, C, O, V, S, A\}$$

$$T = \{char, escaped, range, cls\_d, cls\_w, cls\_int, *, +, ?, |, (,), [,]\}$$

$P =$

Expresiones regulares:

R $\rightarrow$ U

R $\rightarrow \lambda$

Uniones:

U $\rightarrow$ C | U

U $\rightarrow$ C

Concatenaciones:

C $\rightarrow$ O C

C $\rightarrow$ O

Operadores unarios opcionales:

O $\rightarrow$ V $*$

O $\rightarrow$ V $+$

O $\rightarrow$ V ?

O $\rightarrow$ V *range*

O $\rightarrow$ V

Valores:

V $\rightarrow$ [S]

V $\rightarrow$ (R)

V $\rightarrow$ *cls_d*

V $\rightarrow$ *cls_w*

V $\rightarrow$ *cls_int*

Sets de clases:

S $\rightarrow$ A S

S $\rightarrow \lambda$

Átomos de clases:

A $\rightarrow$ *char*

A $\rightarrow$ *escaped*

A $\rightarrow$ *cls_int*

Cada símbolo terminal tiene la siguiente definición:

char : Carácter común no reservado.

escaped : Carácter cualquiera escapado (se escribe con \), excepto d y w.

range : Define un rango de repeticiones de una cadena. Ejemplos: $\{3\}$, $\{2, 4\}$.

cls_int : Define un intervalo de caracteres. Ejemplos: $0 - 9$, $a - z$, $- - \backslash @$.

cls_d : Define el conjunto de caracteres que representan números ($\backslash d$).

cls_w : Define el conjunto de caracteres que representan caracteres de palabras ($\backslash w$).

2.2 Tipo

Para construir la gramática primero debíamos saber a que tipo apuntar, para esto consultamos que tipos de parser permite implementar la herramienta propuesta por la cátedra, PLY. Esta implementa parsers $LALR(1)$, los cuales no estudiamos en la cursada, pero sí sabemos que $SLR(1) \subset LALR(1)$. Gracias a esta propiedad solo debímos asegurarnos que la gramática fuera $SLR(1)$ para asegurarnos que no hubieran conflictos en la tabla del parser final.

Para asegurarnos de que la gramática siempre fuera $SLR(1)$ nos apoyamos en esta herramienta, que resulto sumamente útil a lo largo del TP.

2.3 Decisiones de diseño

En el código fuente se pueden ver las diferentes iteraciones por las que paso la gramática en el archivo *gramatica.md*, que además es compatible con la entrada de la herramienta previamente mencionada (lo que permite ver la tabla de parsing y probar cadenas a mano).

A continuación se detallan algunas de los motivos que nos llevaron a la gramática final y como la afectaron.

- Empezamos con una gramática tentativa, que ni siquiera era SLR pero que tenia una forma similar a lo que nos imaginábamos. Esta fue iterada con nuevos no terminales y producciones hasta que fue SLR.

- Originalmente el token *range* no existía, simplemente en la misma producción se tenia la producción $V\{number\}$ con { y } literales, y *number* un token (que dejó de existir posteriormente). Para el rango con 2 valores era lo mismo con el literal adicional , en el medio. Esto no fue factible ya que las llaves y la coma no eran caracteres reservados y esperábamos que se parseen tanto como *char* como su literal correspondiente. Al ser ambos del mismo largo el lexer no puede desambiguarlos y esto nunca hubiera funcionado.

- Luego tuvimos dos tokens para representar rangos por separado, el de un solo parámetro y el de ambos. Pero al parser no le interesaba demasiado de donde venia el rango, solo en que rango de repeticiones debe aceptar la subexpresión anterior, por lo que los unificamos.

- También originalmente el átomo de las clases de símbolos incluían una producción de la forma $char - char$ con $-$ un literal. Pero esto presentaba el mismo problema que el punto anterior por lo que se introdujo el token *cls_int*.

- Introdujimos nuevos tokens para las cadenas $\backslash d$ y $\backslash w$, si bien esto se podría haber implementado con el token que ya teníamos (*escaped*), nos pareció que semánticamente tenia mas significado que fuera un token aparte.

3 Implementación

3.1 Estructura

Los archivos que se agregaron a la solución son los siguientes:

gramática.md: Producciones de la gramática utilizada.

lexer.py: Implementa el analizador léxico. Se basa en el módulo *ply.lex*. Se definen las reglas para dividir el input en tokens. Cada token representa una unidad de significado en el lenguaje de resolución de problemas, como un identificador, un operador, un número o una cadena de caracteres. Acá agregamos dos clases para usar más fácilmente los rangos e intervalos:

- *RegexRange* : tenemos la cantidad mínima y máxima de apariciones de la subcadena con los atributos *min* y *max*.
- *RegexClassInterval* : tenemos el primer y último carácter del intervalo representado con fst y lst, y además un método *all_symbols* que nos devuelve un conjunto con todos los símbolos posibles para ese intervalo.

parser.py Implementa el analizador sintáctico. Se basa en el módulo *ply.yacc*. Se definen las reglas para construir un árbol sintáctico abstracto a partir de los tokens generados por el analizador léxico. El árbol sintáctico representa la estructura semántica del código fuente. Acá creamos diferentes métodos para $parsear$ cada producción. El AST tiene en cada uno de sus nodos un valor asociado, la mayoría tiene como valor asociado una expresión regular (de la clase $RegEx$), y las producciones definen como deben combinarse. La excepción a esto son los nodos productos del no terminal S (que representan a las clases de símbolos). Estos nodos tienen como valor un *set* de símbolos.

test_lexer.py: Contiene una serie de tests que fuimos aplicando mientras desarrollábamos el analizador léxico.

3.2 Problemas encontrados

Si bien en normas generales la implementación fue bastante sencilla una vez definida la gramática, hay un par de puntos con los que nos encontramos que pensamos que vale la pena mencionar:

- Sistema de módulos de Python. El sistema de módulos mas esotérico y complicado con el que nos tocó trabajar. Tuvimos problemas tanto para definir nuevos módulos como para incluirlos en otros. En particular cuando creamos nuestros propios tests para el lexer, lograr que Pytest encuentre el archivo fue un suplicio. También tuvimos problemas con definiciones circulares de módulos.

- Nuestra primera implementación que pasaba los tests demoraba 50 (!!) minutos en ejecutarse. Claramente esto no era un nivel de performance aceptable, por lo que nos propusimos mejorarla. Primero debimos identificar los cuellos de botella en performance, lo que nos llevó al módulo pytest-profiling. El profiling que hicimos nos hizo darnos cuenta que la implementación provista de *normalize_states* era sumamente lenta, y nosotros la usábamos extensivamente. Debido a esto implementamos un nuevo método mas performante y restringido en su funcionalidad para usar en lugar de *normalize_states*. Como lo único para lo que lo usábamos era para evitar colisiones al mergear dos AFNDs, decidimos implementar add_prefix con el constraint de tener una complejidad de a lo sumo $\mathcal{O}(e+n)$.

  Esto mejoró pero siguió sin ser suficiente. A continuación nos enfocamos en el método *minimize*, bajo la hipótesis de que los problemas de performance eran producto de una mala implementación o algoritmo. Debido a esto, decidimos reimplementarlo usando el algoritmo de Hopcroft, teniendo especial cuidado en asegurarnos que fuera lo mas performante posible. Esto nuevamente produjo mejorías en performance, pero seguía sin ser satisfactoria.

  La ultima arista que atacamos fue producto de entender mas en detalle los algoritmos utilizados, y notar que todos dependían del producto entre la cantidad de nodos y/o transiciones y el tamaño del alfabeto. También observamos cuales eran los tests que mas tardaban, y vimos que eran todos los que tenían clases de símbolos. El motivo fue entonces claro, el problema estaba en que con la generación

de regex naive que estábamos usando, se generaba un nodo por cada símbolo de la clase, cuando en realidad 1 solo nodo terminal con tantas transiciones como símbolos de la clase hubiera sido suficiente. Para subsanar este problema implementamos la clase *RegexClass* que representaba una de estas clases de símbolos y que generaba el AFND de manera eficiente.

Se podría hacer algo similar para los rangos, pero la performance de los tests no se vio afectada de manera significativa por la implementación naive, así que decidimos dejarla por simplicidad.

Los resultados del profiling en distintas etapas están disponibles en el anexo 4; y en mejor calidad dentro de la carpeta $informe/profiling$, ordenados según se describió en este punto.

4 Anexo A: Profiling

Para las siguientes imágenes, se recortó hasta el nodo *match* (el principal) que es a partir de donde nos centramos para hacer las optimizaciones. Ya que ahí es donde estaba el 99.99% del tiempo.

Cada nodo representa un método y tiene un porcentaje de tiempo de procesamiento, uno es total a toda la ejecución, y el otro es relativo a sus nodos del mismo nodo padre padre. Cuanto más rojo, más tiempo demoró de procesamiento ese método.

Nota: las imágenes están en el orden que fuimos optimizando, podemos observar que a medida que fuimos optimizando, se visualizan más nodos, ya que la corrida duraba menos tiempo entonces se distribuye el tiempo en más nodos.

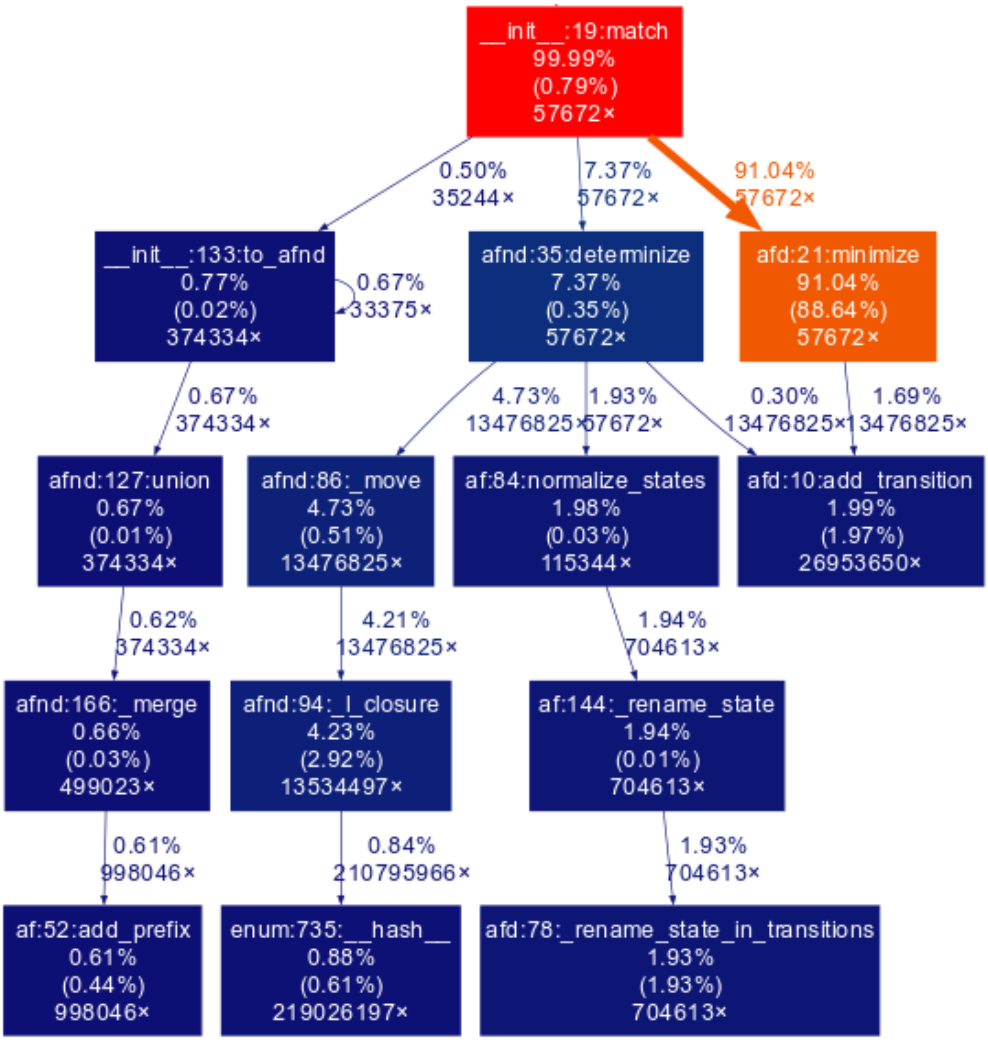

Figure 1: Ejecución completa luego de optimizar la normalización de estados, a pesar de que notamos una mejora, demoró aproximadamente 51 minutos.

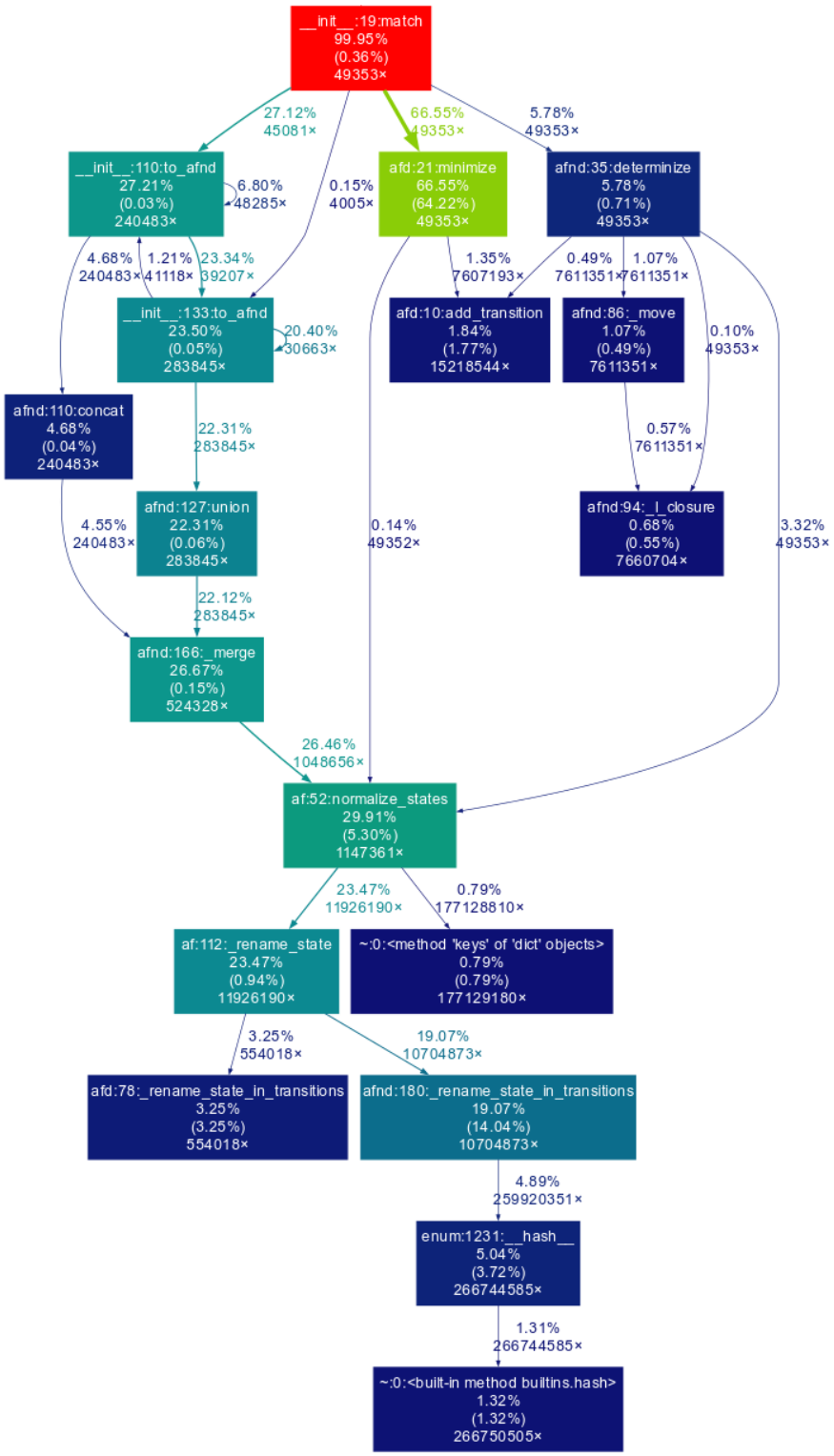

Figure 2: Ejecución completa luego de optimizar la minimización, demoró aproximadamente 8 minutos.

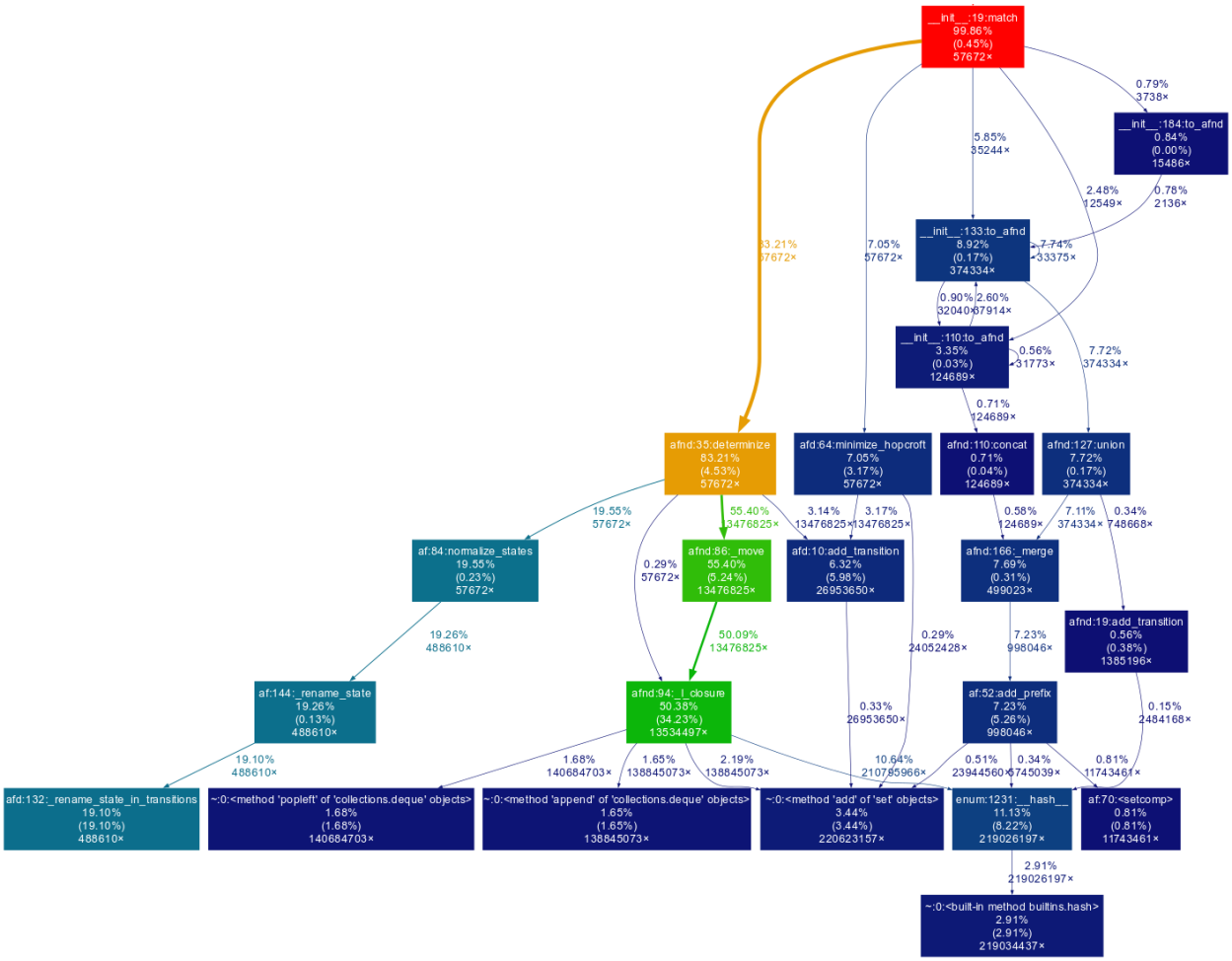

Figure 3: Ejecución completa luego de agregar la clase *RegexClass*, demoró aproximadamente 1 minuto.

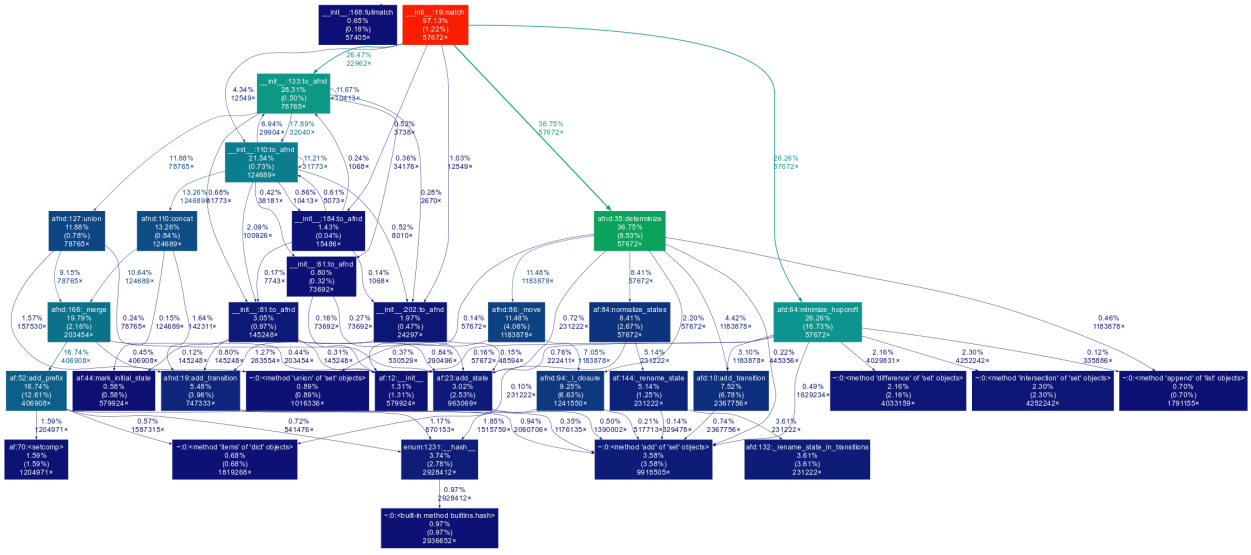

Figure 4: Ejecución completa final, demora aproximadamente 15 segundos.