Universidad Nacional de Rosario

Autómata Le Pila

Diseño e implementación de un DSL para autómatas de pila no deterministas

Autor:
Juan Ignacio Farizano

Trabajo Práctico Final
Análisis de Lenguajes de Programación
Departamento de Ciencias de la Computación
Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Rosario, Santa Fe, Argentina

10 de febrero de 2022

Índice

1. Motivación **2**

2. Decisiones de diseño **2**

3. Lenguaje **3**
3.1. Sintáxis concreta 3
3.2. Notación 3

4. Resolución **5**
4.1. Parseo 5
4.2. Uso de las mónadas 5
4.3. Evaluación 5
4.4. Otros archivos 5

5. Instalación y uso **6**
5.1. Requisitos 6
5.2. Instalación 6
5.3. Manual de uso 6

6. Bibliografía, material utilizado y software de terceros. **7**

1. Motivación

Este proyecto nace a partir de las clases de Lenguajes Formales y Computabilidad, donde tuvimos nuestro primer acercamiento a estas máquinas reconocedoras de lenguaje.

La idea principal es proveer a les alumnes que estén cursando segundo año una herramienta donde puedan testear los autómatas que construyan en la práctica, pudiendo ver si ciertas palabras del lenguaje pedido son reconocidas como esperado, ver el paso a paso de como la máquina elige la siguiente transición y como cambia su configuración en el momento (utilizando el modo *verbose*) además de luego poder graficar estos autómatas de forma sencilla.

2. Decisiones de diseño

Durante el tiempo transcurrido entre la primera presentación de la idea para el trabajo y su primera versión funcional el lenguaje pasó por muchos cambios.

Al principio se iba permitir cargar en memorias varios autómatas y modificarlos en tiempo real desde la consola, pero esto se mostró muy complejo además de innecesario ya que en casos de uso reales se estaría probando solo un autómata a la vez, por lo que se optó cargar uno solo a la vez y desde un archivo para mantener la consola interactiva lo maś sencilla y fácil de utilizar posible.

Otras ideas eran por ejemplo tener diferentes órdenes de evaluación, ya sea que en cada paso se elija una transición aleatoria entre todas las disponibles o recorrerlas a todas para ver por cuantos caminos se puedan reconocer una palabra, pero de nuevamente complejizaba el código, no eran características indispensables y podían traer problemas en la ejecución como caer en loops infinitos con mayor facilidad.

En la versión final del programa se carga un autómata desde un archivo y a través de una consola interactiva se permite cargar archivos nuevos, recargar el último archivo utilizado, imprimir en consola el autómata, graficarlo el mismo con salida a un archivo y reconocer palabras. También se cuenta con un modo verbose que hace que en cada comando pedido por consola se imprima información extra al respecto, por ejemplo en la evaluación se imprime cada paso y decisión tomada o en el proceso de graficar imprime los parámetros utilizados.

3. Lenguaje

Defino la sintaxis concreta del DSL a partir de la siguiente gramática, donde el autómata se lee a partir de la regla de producción PDA (PushDown Automaton), y a partir de ella se leen todas las partes que lo componen.

3.1. Sintáxis concreta

```
PDA ::= InputAlph '=' ALPH';'
        StackAlph '=' ALPH';'
        States '=' STATES';'
        AccStates '=' STATES';'
        Transitions '=' TRANSITIONS';'

ALPH ::= '{' SYMBOLS '}' | '{' '}'

SYMBOLS ::= SYMBOL | SYMBOL',' SYMBOLS

STATES ::= '{' STATES' '}' | '{' '}'

STATES' ::= STATE | STATE',' STS

TRANSITIONS ::= '{' TRANSITIONS' '}' | '{' '}'

TRANSITIONS' ::=  TRANSITION | TRANSITION',' TRANSITIONS'

TRANSITION ::= '('STATE',' SYMBOL',' SYMBOL',' SYMBOL',' STATE')'
```

Notas extras de la sintaxis

La regla de producción SYMBOL es análoga a leer chars individuales, solo que no se permiten ciertos caracteres reservados, como los espacios, los caracteres de control y los caracteres reservados '=', ',', ';', '(', ')', '{', '}'.

La regla de producción STATE es análoga a leer un string pero solo con los caracteres permitidos en SYMBOL.

3.2. Notación

Para definir un autómata, se lo debe escribir en un archivo con extensión **.pda** donde dentro se lo define siguiendo la gramática dada anteriormente, como se puede ver en el siguiente ejemplo:

```
InputAlph = {x, y};
StackAlph = {x, y, #};
States = {s0, s1, s2, s3};
AccStates = {s0, s3};
Transitions = {(s0, λ, λ, #, s1), (s1, x, λ, x, s1), (s1, y, x, λ, s2),
(s2, y, x, λ, s2), (s2, λ, #, λ, s3)}
```

En este autómata podemos ver que en cada línea se define:

- El alfabeto de entrada compuesto por x e y.
- El alfabeto de pila compuesto por x, y el símbolo #.

- Los estados s0, s1, s2 y s3.
- Los estados de aceptación s0 y s3.
- Las transiciones.

Para el estado inicial se utiliza el primer estado definido.
Las transiciones son tuplas compuestas por estado actual, símbolo a leer de la entrada, símbolo que se extrae de la pila, símbolo que se inserta a la pila y el estado siguiente.

Nota: El símbolo λ no es necesario incluirlo en los lenguajes de entrada y pila, ya que está incluido por defecto.

4. Resolución

4.1. Parseo

Para el parser se utilizó la herramienta Happy, un generador de parsers para Haskell, que definiendo una gramática en Yacc y un lexer en Haskell te genera un archivo Haskell con el parser. Este código se puede encontrar en el archivo **src/Parse.y**

4.2. Uso de las mónadas

En el archivo **src/Monad.hs** se encuentra definida la mónada *MonadPDA* que se utiliza en todo el programa, esta está basada en la mónada del compilador de la materia Compiladores y me permite realizar operaciones de entrada/salida, manejo de errores y llevar un estado global con todos los datos necesarios para la ejecución.

4.3. Evaluación

El evaluador de autómatas se encuentra en el archivo **src/Eval.hs**, allí se exporta la función evaluadora *evalPDA* que recibe un autómata y una palabra a reconocer, esta a su vez hace uso de la función *evalPDA'* a la que le pasa los argumentos anteriores más la configuración inicial de la máquina que es con el estado inicial y la pila vacía. La evaluación si se atasca en algún paso realiza backtracking, lo que permite evaluar autómatas de naturaleza no determinista.

4.4. Otros archivos

- En el archivo **src/Lang.hs** se encuentran las estructuras de datos utilizadas para representar a un autómata de pila, sus partes y una configuración del mismo.
- En el archivo **src/Global.hs** se encuentra la estructura de datos utilizada para llevar el estado global del sistema y todos sus datos.
- En el archivo **src/Lib.hs** se encuentra funciones utilizadas en el resto del programa, en este caso solo se exporta una función utilizada para verificar que un autómata dado sea válido y cumpla con su estructura y condiciones requeridas.
- En el archivo **src/PPrint.hs** se encuentran las funciones utilizadas para imprimir en consola de forma más clara y fácil de leer las diferentes estructuras utilizadas, mensajes de errores, mensajes que solo se imprimen con modo verbose, etc.
- En el archivo **src/Graphic.hs** se encuentran las funciones utilizadas para graficar un autómata y exportarlo a un archivo de salida.
- El archivo **app/Main.hs** es el archivo principal del programa, en él se realiza el loop interactivo con la consola con todo la interpretación y evaluación de comandos que se tienen que realizar en ella.

5. Instalación y uso

5.1. Requisitos

Para poder usar el programa es necesario tener instalada la plataforma de Haskell junto a Stack.

Para poder graficar autómatas es necesario tener instalado el paquete *graphviz*, en caso contrario este comando nunca se ejecutará.

5.2. Instalación

1. Clonar el repositorio con el comando:

```
git clone https://github.com/jfarizano/AutomataLePila.git
```

2. Moverse a la carpeta del programa creada por el comando anterior y compilar con el comando:

```
stack build
```

Con esto ya tenemos el programa instalado y está listo para su uso.

5.3. Manual de uso

Para ejecutar el programa utilizamos los comandos:

```
stack run
stack run -- [FLAGS] [FILE]
```

Donde el primero ejecutará el programa sin cargar ningún autómata ni recibirá flags, en este caso se podrá usar la consola pero solo se permitirá cargar archivos o usar la ayuda hasta que se use el comando de cargar archivos y se lea un autómata válido. A partir de este punto se puede utilizar los comandos restantes.

En la segunda opción se le puede dar un archivo de entrada directamente al programa para que este lo lea directamente sin tener que darlo en la consola interactiva, además de poder darle flags para cambiar parámetros de su ejecución y del comando de graficar.

Las acciones y opciones de estas flags se pueden ver con el comando

```
stack run -- -h
```

Una vez dentro del loop de la consola interactiva y con un autómata válido cargado podemos realizar diferentes comandos, entre ellos imprimir una pantalla de ayuda, activar o desactivar el modo verbose, cargar o recargar un archivo, imprimir en consola el autómata cargado o graficar y exportar el mismo. La ayuda de estos comandos se puede ver escribiendo el comando '**:h**' en consola.

6. Bibliografía, material utilizado y software de terceros.

- Diapositivas de clase de la materia Lenguajes Formales y Computabilidad por Pablo Verdes.
- Apunte de Lenguajes Formales y Computabilidad por Damián Ariel Marotte
- Mónada basada en la mónada del compilador de la materia Compiladores dada por Mauro Jaskelioff.
- Generador de parsers Happy.
- Librería CmdArgs para parseo de argumentos en consola del sistema.
- Librería Haskeline para entrada en la consola interactiva del programa.
- Librería PrettyPrinter para imprimir en consola de forma clara.
- Librería GraphViz utilizada para poder usar el programa de gráficos de mismo nombre dentro de Haskell.
- Librería FGL utilizada para una representación intermedia de los autómatas como grafos en la librería anterior.