Universidad de Antioquia
Departamento de Ingeniería de Sistemas
2508508 / Teoría de lenguajes y laboratorio
Semestre 2020 - I
Práctica #1
Juan David Zea Y Vanessa Tocasuche
Fecha de entrega: 01/11/2020

MANUAL TÉCNICO DEL MINI-COMPILADOR

Índice:

1. Introducción
2. Cómo instalar el programa
3. Cómo ejecutar el programa
4. Cómo hacer revisión
   4.1. Cómo leer archivos
   4.2. Cómo interpretar la respuesta
   4.3. Cómo hacer nuevas revisiones
5. Cómo cerrar el programa
6. Restricciones del programa
   6.1. Estructura permitida

1. Introducción:

En este documento se explicará a detalle cómo funciona el programa basado en autómatas finitos “mini-compilador”, su estructura, el recorrido principal, y todas las funcionalidades plasmadas en código.

2. Arquitectura del software:

El software se ha creado bajo el patrón de arquitectura Model-View-Controller (MVC) por lo que se abordarán primero estos aspectos del programa:

3. Modelos:

El software cuenta con 8 modelos y una interfaz, las cuales son:

3.1. Modelo NodoSimple:

Este modelo representa a un nodo simple de las listas simplemente ligadas, tiene unas ligeras modificaciones para adaptarse a los requerimientos de la práctica como tener un atributo de “clase” y uno de “valor” en lugar del convencional “dato”.

3.2. Modelo LSL:

Este modelo representa a las listas ligadas, carece de algunos métodos fundamentales porque no fueron necesarios en el programa (insertar, desconectar, etc...).

3.3. Modelos A1, A3, A4

Estos modelos representan a los autómatas más usados en el programa.

Estos autómatas comparten la misma estructura:

```
public class A1 implements Automaton {

    private String state;
    private int position;
    private ArrayList<Character> symbols;
    private ArrayList<Character> inputs = new ArrayList<>();

    public A1() { this.loadSymbols(); }

    /** Loads automaton symbols */
    @Override
    public void loadSymbols() { ... }

    @Override
    public String process(@Nullable String state, int position, ArrayList<Character> inputs) { ... }

    public String useAutomaton(String id, String state, int position) { ... }

    public String filter(char c) { ... }
}
```

Se componen de:

Un atributo **state** que guarda el estado en el que se encuentra el autómata.
Un atributo **position** que guarda la posición en la que se va recorriendo en el ArrayList inputs.
Un atributo **symbols** que guarda los símbolos de entrada del autómata.
Un atributo **inputs** que guarda los caracteres que leerá el autómata.

Un método **loadSymbols** que carga los símbolos de entrada del autómata en el atributo **symbols**.

Un método **process** que procesa según el estado del autómata, un conjunto de caracteres desde una posición establecida.

Un método **useAutomaton** que permite al autómata usar otros autómatas auxiliares.

Un método **filter** que permite determinar si un carácter pertenece a los símbolos de entrada del autómata.

De las anteriores características, la más importante es el método **process** que tiene la siguiente estructura:

```
public String process(@Nullable String state, int position, ArrayList<Character> inputs) {
    if (state == null) { ... } else { ... }
    this.position = position;
    this.inputs = inputs;

    String symbol;

    for (; this.position < this.inputs.size(); this.position++) {
        symbol = this.filter(this.inputs.get(this.position));
        switch (this.state) {
            case "S": { ... }
            case "D": { ... }
            case "NV": { ... }
            case "N1": { ... }
            case "N2": { ... }
            case "I": { ... }
            case "I+": { ... }
            case "I-": { ... }
            case "P": { ... }
            case "A": { ... }
        }
    }

    switch (this.state) { ... }
    return null;
}
```

Este método es quien le da a la clase automata su comportamiento, por medio de una sentencia **switch** responde de acuerdo al estado en el que se encuentre, cambiando el atributo **state** como lo hacen los autómatas finitos.

3.4. Modelo AG1

Este modelo también representa a un autómata, pero en este caso, a uno “generalizado”, esto significa que es un autómata que puede responder con estados recibidos por parámetros, esto permite poder usar el mismo autómata para dar respuestas personalizadas.

Las respuestas se establecen en su método constructor así:

```
AG1 automaton1_1 = new AG1( O: "NV", X1: "S", X2: "S", R1: "D", R2: "EP", R3: "EP");
String response = automaton1_1.process(state, position, this.inputs);
```

Este autómata simplifica el proceso de lectura de palabras reservadas en todas las

ocasiones que se requieren.

3.5. Modelo AP1

Este autómata representa al autómata de pila que analiza los paréntesis, tiene la siguiente estructura:

```
public class AP1 {
    private final Stack<Character> stack = new Stack<>();

    public String process(Character symbol) { ... }

    public String state() { ... }

    public String transition(int index) { ... }

    public void reset() { this.stack.clear(); }

    public Stack<Character> getStack() { return stack; }
}
```

Un atributo **stack** que guarda las entradas a la pila.

Un método **process** que procesa una entrada según el estado del autómata.

Un método **state** determina el estado del autómata según la pila.

Un método **transition** ejecuta la transición según el índice pasado como parámetro.

Un método **reset** devuelve la pila del autómata a su estado inicial.

Un **getter** para la pila del autómata.

3.6. Modelo Symbols:
Este modelo es útil para guardar la información de los símbolos que reciben los autómatas, optimiza el cargado de los mismos, y contiene todos los posibles aceptados.

3.7. Interfaz Automatón:
Esta interfaz reúne los métodos principales de un autómata finito, y es implementado por A1, A3, A4 y AG1. El autómata AP1 no hace uso de esta interfaz porque los autómatas de pila se abordaron con una estructura diferente.

4. Vistas:
El programa guarda como vistas las imágenes usadas por el mismo, y el archivo fxml de javaFx para la construcción de la ventana.

5. Controladores:
Solamente se usan 3 clases controladoras en el programa, las cuales son:

5.1. Program:
Es la clase principal, que inicia el programa y la ventana JavaFx:

```
public class Program extends Application{
    public static void main(String[] args) { launch(args); }

    @Override
    public void start(Stage primaryStage) throws IOException {
        Parent root = FXMLLoader.load(getClass().getResource( name: "/Views/Window.fxml"));
        primaryStage.setTitle("Mini-Compilador");
        primaryStage.getIcons().add(new Image( url: "Views/images/icon.png"));
        Scene scene = new Scene(root);
        primaryStage.setScene(scene);
        primaryStage.show();
    }
}
```

5.2. WindowController:
Es la clase enlazada con la ventana JavaFx, que permite darle funcionalidad a los botones, labels y textareas:

```java
public class WindowController implements ClipboardOwner{

    @FXML
    TextArea ta_file;
    @FXML
    TextArea ta_errors;
    @FXML
    TextArea ta_lsl;
    @FXML
    ImageView img_analyze;
    @FXML
    ImageView img_analyze1;
    @FXML
    Label lab_result;

    public void searchFile() { ... }

    public void analyze() { ... }

    private void results(ArrayList<String> responses, ArrayList<String> stringLists, int errors) { ... }

    private void updateImages(int image) { ... }

    public void link0() { ... }
    public void link1() { ... }
    public void link2() { ... }
    public void link3() { ... }

    @Override
    public void lostOwnership(Clipboard clipboard, Transferable contents) {}
}
```

Esta clase implementa la interfaz ClipboardOwner ya que es necesario guardar los links mostrados en la pestaña “Acerca De” en el portapapeles del sistema, los métodos link son quienes efectúan esta tarea.

El método **searchFile** permite leer un archivo .java.
El método **analyze** instancia el controlador **Compiler** que reproduce el análisis de los autómatas.
El método **results** modifica las pestañas del programa con las respuestas de los errores y las listas ligadas.
El método **updateImages** actualiza las imágenes del programa según el caso.
El método **lostOwnership** es requerido por la interfaz implementada.

5.3. Compiler:

Un controlador sobresaliente, ya que gestiona todo el texto a analizar y lo procesa con el autómata 1 (A1) y el autómata de pila 1 (AP1):

```
public class Compiler {

    public static LSL lsl = new LSL();
    public static int posI;
    public static int posF;
    public static AP1 stackAutomaton1;

    public ArrayList<ArrayList<String>> analyze(String text) { ... }

    public ArrayList<String> splitLines(String text){ ... }

    public ArrayList<Character> partition(String line) { ... }

    public static int hasError(ArrayList<String> responses) { ... }

    public static String translate(String pErrors) { ... }
}
```

Está compuesto por:

El método **analyze** que le entrega a un objeto de la clase A1, cada renglón del texto a analizar separado caracter a caracter, y de esta forma retorna 2 ArrayLists, **responses** que contiene las respuestas de los análisis realizados a las líneas, y **stringLists** que contiene las cadenas que representan a las listas ligadas de cada línea analizada.

El método **splitLines** que divide un bloque de texto en sus renglones.
El método **partition** que divide un renglón en sus caracteres individuales.
El método **hasError** retorna cuántos errores hay en un conjunto de respuestas.
El método **translate** retorna la cadena de definición de un conjunto de respuestas.

6. Ruta de ejecución:

Todo comienza cuando se pulsa el botón “Analizar”, el compilador divide el texto en líneas y le pasa las líneas particionadas en caracteres al autómata 1 (A1), este procesa la línea desde el primer carácter en su estado inicial, usará al autómata 3 (A3) para leer posibles nombres de variables y después del igual (=) usará al autómata 4 (A4) para leer una expresión válida.