Warsaw University of Technology
Faculty of Mathematics and Computer Science

ΦNITE

application for building finite automaton that is equivalent to a given regular expression and for simulating machine's evaluation of a given word

Author:
Mateusz Bysiek

Supervisor:
dr Lucjan Stapp

Warsaw, 14 Mar 2013

Document metric

Project	ΦNITE	**Company**	WUT
Document name	business analysis		
Document topics	initial analysis of the problem, requirements specification, guidelines		
Author	Mateusz Bysiek		
File	bysiekm-business-analysis.pdf		
Version no.	0.08 **Status** pre-alpha	**Opening date**	21 Feb 2013
Summary	Author of the document analyses the problem of designing an application for solving a well defined language-theory related problem, doing it from the business perspective, i.e. giving set of requirements that such application must fulfill and providing guidelines for developers.		
Authorized by	dr Lucjan Stapp	**Last modification date**	14 Mar 2013

History of changes

Version	Date	Author	Description
0.01	21 Feb 2013	Mateusz Bysiek	creation of template for the document
0.02	23 Feb 2013	Mateusz Bysiek	outline of the features section
0.03	25 Feb 2013	Mateusz Bysiek	added definitions section
0.04	25 Feb 2013	Mateusz Bysiek	added algorithm section
0.05	27 Feb 2013	Mateusz Bysiek	added GUI mockup, moved definitions
0.06	28 Feb 2013	Mateusz Bysiek	added use cases
0.07	8 Mar 2013	Mateusz Bysiek	removed GUI mockup
0.08	10 Mar 2013	Mateusz Bysiek	added word evaluation feature

Contents

Abstract

This document describes a business analysis (problem analysis and requirements specification) of a planned application called Φnite. This application is aimed at introducing the user to the world of regular expressions[1] and finite-state machines[2] (i.e. the theoretical constructs that let us evaluate regular expressions). The application will consist of GUI (graphical user interface) coupled with a regular expression evaluation engine. Together these two components will enable the user to enter an expression by hand (or, alternatively, load it from a list of prepared examples) and then semi-automatically construct a finite-state machine that is equivalent to this expression. In other words, given regular expression expands into the same set of words that a resulting finite-state machine accepts.

Furthermore, the user will be able to simulate how the built finite-state machine works for a given input word.

1 Feature overview

In this section, reader is presented with the features that must be implemented, unless a given feature is explicitly labeled as: optional, not mandatory, etc. The application will fulfill the requirements if it implements all of the given features, unless those features are impossible to implement while meeting the application environment constraints.

1.1 Input methods

Some definitions are vital in proper understanding of the calculation methods described below, and in proper understanding of the requirements.

Definition

- *input data* is a sequence of characters, which can be converted to a valid regular expression without ambiguity, and without user interaction.

Those characters must be writable with regular plain text editor. Moreover, the input data should, unless it is impossible in certain cases, be visually similar to a regular expression that is written without plain text editor. Using LaTeX math mode syntax is preferred, but not mandatory.

Ex. "`ab^*`" under a set of not ambiguous rules may represent ab^*, and on the other hand, there is no set of unambiguous rules to convert `(ab)*(` to a regular expression. The latter may represent $(ab) * ($ i.e. a word in which parentheses and other special characters denote letters (preventing the user from using parentheses - which is unacceptable), or maybe $(ab)^*($ where the last parenthesis serves as a letter, and the two first are parentheses. The opening parenthesis may have different meaning depending on context - it is also unacceptable.

Requirements

The user shall be able to enter the regular expression using at least two methods described below:

- *direct input* - user writes the expression into a single- or multi-line plain text field in the GUI.
- *example selection* - user selects expression from a list of predefined expressions. There should be at least 10 example expressions, and they (globally) have to show usage of:
  - *empty word*, ex. ϵ
  - *concatenation*, ex. ab is a concatenation of a and b
  - *union*, ex. $a + b$, which generates a and b, and is equivalent to $b + a$
  - *Kleene star*[3], ex. a^*, which generates a set of concatenations of a with itself $n \in \mathbb{N} \cup \{0\}$ times, and is equivalent to $\epsilon + a^+$
  - *Kleene plus*[3], ex. a^+, which generates a set of concatenations of a with itself $n \in \mathbb{N}\backslash\{0\}$ times, and is equivalent to aa^*
  - *precedence of operators* (i.e. the use of parentheses and/or their absence), ex. ab^+, is equivalent to abb^*, but $(ab)^+$ is equiv. to $ab(ab)^*$

  The selected example expression is loaded into the text field mentioned in the first input method, overwriting any previous input.

1.2 Input editing methods

The user will be able to edit the expression regardless of what the input method was, before he/she proceeds with calculations. Editing the input is to be done via a plain text field in the GUI.

1.3 Calculation methods

Definitions

- *one step of computation* is a smallest step that can be taken from the algorithm used to complete calculations (in this case it is to find finite-state machine equivalent to the given regular expression).
- *visualisation of current status of computation* consists of two tables that represent (respectively) the labeling and derivation process undertaken by the program (see section 2.3 for details regarding labeling and derivation process).

The first tables shows all currently labeled and not labeled symbols, therefore has to have at least two columns: regular expression, and label. When second field is empty then the expression is not labeled. If the algorithm is currently in labeling phase, the currently evaluated non-labeled expression is highlighted.

The second table has at list three columns: initial expression, subtracted symbol, resulting expression. If the algorithm is currently in derivation phase, the currently derived expression is highlighted.

Highlighting means that either 1) the background color of a cell in which a highlighted content resides is changed to the font color, and the font color is changed to the background color or 2) background color of a cell has color changed from white to gray (between `#c0c0c0` and `#808080`, inclusive).

Requirements

The user will be able to convert the entered expression to an equivalent finite-state machine using two methods:

- *immediate result method*: will proceed with calculations up to the moment when user interaction is necessary, and will show visualisation of current status of computation only in such situation. After such pause, when the user decides to end interaction, he/she will click a button that will cause the computation to continue.

  This method completes when there are no intermediate steps that need user interaction and the final result is presented in the GUI.

- *step-by-step method*: will perform one step of computation and present visualisation of current status of computation to the user. At that point the user can choose whether he/she would like to:

  - proceed with the next step of computation
  - abort computation, effectively reverting to the beginning
  - proceed with the computation using *immediate result method*, starting from the current position

  This method completes when there are no more steps needed to obtain the final result.

  In this method, user still might be obliged to perform some actions in between steps, as it is possible in case of *immediate result method*. In such cases, the 3 options for typical continuation shall be disabled, and only after the user indicates that the intermediate interaction is finished, they are enabled again and the user may choose further course of action.

1.4 Output analysis methods

Definition

- *final result* is a two-dimensional directed graph with labeled edges. It is a graphical representation of a finite state machine constructed.

Requirements

Final result has to conform with several constraints, most of which follow directly from the convention regarding drawing finite-state machines:

- all vertices, letters and edges on the graph have to be black and the background has to be white
- the vertices have to be black circles, and those representing accepting states have to have another black circle inside
- the initial state has to have index 0 ex. it can be q_0, s_0, r_0, ...
- the edges leading to the rejecting state should be omitted for better readability, and the rejecting state is not to be drawn at all
- the drawing has to be readable, i.e. font size used has to be at least 10 points.
- if the drawing is too big to fit in applicatin window or on user's screen, the output is not zoomed out but rather a method must be implemented so that the user can scroll around the drawing to be able to see everything piece by piece.

1.5 Simulation methods

Definition

- *finite-state machine simulation* is a process of evaluation of a sequence of letters given by the user using a finite-state machine.

Requirements

The finite-state machine simulation must conform with the following requirements:

- the simulation should be done step-by-step, much like in the case of machine construction process, but the difference is that the feature of immediate solution may be ommited (see requirements part in section 1.3 for details regarding solution methods)
- the evaluation is represented graphically, i.e. the requirements regarding graphical representation of the output apply to the finite-state machine simulation
- in each step the node that represents the current state of the machine is highlighted (see the end of definitions in section 1.3)
- if a rejecting state is entered than no node is highlighted anymore and the dialog is displayed that informs the user that the word was rejected by the machine
- if there are no more letters for evaluation and current state is not an accepting state, the dialog (analogous to the one in the previous point) is displayed
- if an accepting state is entered and there are no more letters remaining for evaluation, the dialog is displayed that informs the user that the word was accepted by the machine

2 Building finite-state machine: workflow with example

The computation process used by the application shall follow the following outline. Solutions to two sub-problems:

- determining if some regular expression is equivalent to some other expression
- determining whether some regular expression can generate an empty word

... do not have to be implemented, and can be left out for the user interaction. However, when algorithm requires user interaction, it has to stop and show to the user all the needed information so that he/she can really perform a task using only data provided by the application. The development team is free to choose at which steps the the algorithm will be interactive.

2.1 Input

Input is in every case a valid regular expression. Further steps cannot be performed until the given input satisfies validity criterion.

Example input:

$$a^+c^+ + ab^+c$$

This expression represents a following set of words:

$$L = \{ac, aac, aaac, aaaac, \ldots\} \cup \{acc, aacc, aaacc, aaaacc, \ldots\} \cup \{accc, aaccc, aaaccc, aaaaccc, \ldots\} \cup$$
$$\cup \{acccc, aacccc, aaacccc, aaaacccc, \ldots\} \cup \ldots \cup \{abc, abbc, abbbc, abbbbc, \ldots\}$$

2.2 Task

Task is the same for every input: "construct a finite-state machine equivalent to a given regular expression".

2.3 Solution process

In this part FSM is used as an acronym for finite-state machine.

2.3.1 Step 0

Labeling, step 0_L

At first we label our input as an initial state of the FSM:

$$a^+c^+ + ab^+c = q_0$$

Derivation, step 0_D

Then, we try to derive all states of FSM that can be reached from our initial state in one transition.

e/x means that we try to construct from expression e a new expression without initial symbol x, provided that such expression exists. If it does not, the result is the empty set:

$$q_0/a = a^*c^+ + b^+c$$
$$q_0/b = \emptyset$$
$$q_0/c = \emptyset$$

2.3.2 Steps 1 to N+1

Labeling, step 1_L In each of these steps we at first label all results from the last step as new states of the FSM, unless the equivalent expressions were previously encountered and are already labeled.

$$a^*c^+ + b^+c = q_1$$

Derivation, step 1_D Then we take the first unresolved state from the list of labeled states and we perform the same operation as in Step 0_D (described in point 2.3.1):

$$q_1/a = a^*c^+ \ , \ q_1/b = b^*c \ , \ q_1/c = c^*$$

Labeling, step 2_L $a^*c^+ = q_2$, $b^*c = q_3$, $c^* = q_4$

Derivation, step 2_D $q_2/a = a^*c^+$, $q_2/b = \emptyset$, $q_2/c = c^*$

Step 3_L a^*c^+ and c^* are already labeled

Step 3_D $q_3/a = \emptyset$, $q_3/b = b^*c$, $q_3/c = \epsilon$

Step 4_L b^*c is already labeled, $\epsilon = q_5$

Step 4_D $q_4/a = \emptyset$, $q_4/b = \emptyset$, $q_4/c = c^*$

Step 5_L c^* is already labeled

Step 5_D $q_5/a = \emptyset$, $q_5/b = \emptyset$, $q_5/c = \emptyset$

Step 6_L There are no states to label in this step. There may be some unresolved states, therefore we cannot end computation at this moment.

Step 6_D There are no more unresolved states, the algorithm moves to the next stage.

2.3.3 Step N+2

After derivation from all states (total number of states of the resulting FSM is $N+1$), each state is evaluated and it is determined if an expression equivalent to the state can generate an empty word (ϵ). If yes, such state is added to the set of accepting states. Initial state can also be an accepting state. There can be only one initial state, but multiple accepting states are possible.

In our example $c^* = q_4$ and $\epsilon = q_5$, therefore $Q_A = \{q_4, q_5\}$

There is a special case, an empty state, which is the rejecting state. There is only one rejecting state.

2.4 Final result

Final result is a transformation of a record of computation process into a graph that represents it.

$$\begin{cases} q_0/a = q_1 \\ q_0/b = \emptyset \\ q_0/c = \emptyset \end{cases} \quad \text{is transformed into} \quad \begin{cases} q_0 \xrightarrow{a} q_1 \\ q_0 \xrightarrow{b} \emptyset \\ q_0 \xrightarrow{c} \emptyset \end{cases}$$

Analogously, every other derivation that happened in the steps $1_D \dots N_D$ is transformed in the same way. After all derivations are transformed a graph is drawn in the GUI.

Rules regarding the graph are as follows:

- states $q_0, q_1, \dots, q_n$ become the set of vertices
- transition $q_i \xrightarrow{x} q_j$ becomes a labeled and directed edge from vertex q_i to vertex q_j, with label "x"

This concludes a complete computation process.

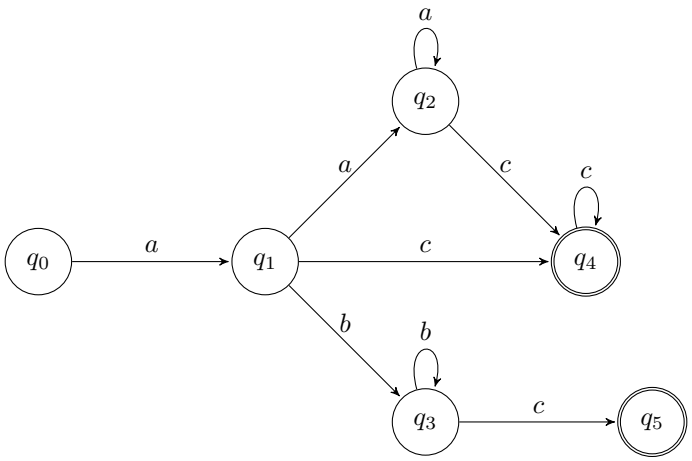

Figure 1: graphical representation of the final answer for the example problem

3 Use cases

Below is the diagram that puts the given requirements into a UML diagram. Division of the application into two parts as pictured is a recommendation. Input/output handler is meant to comprise the features related to getting the regular expression from the user, validating it, and also displaying the final result. Computing process is meant to handle conversion related calculations, without direct input from the user, but is able to receive suggestions regarding computation form him/her.

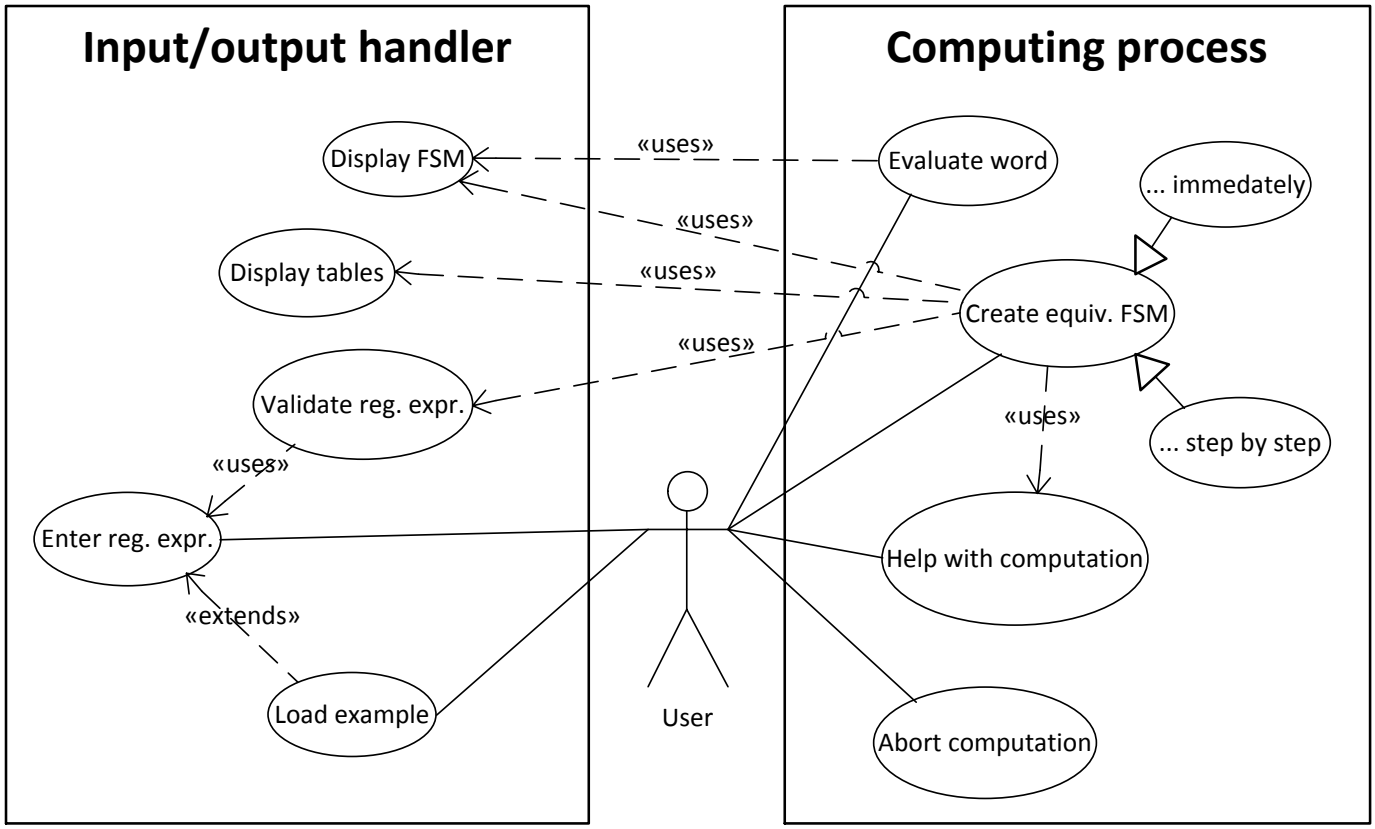

Figure 2: UML use case diagram for Φnite user

4 Exception handling

The theoretical background behind the concept of regular languages is very well established. This application will therefore implement no experimental features, and undefined behaviour is strictly forbidden.

The exception from this rule is a situation in which user input is needed during computation (there are two cases), and the amount of work the user has to do in order to submit his/her help is lowered due to some supporting experimental features, that may not always work, but sometimes help the user. Such solutions will be considered an advantage, but are not a must.

In order to ensure that the computation will proceed as expected, application will perform validation of input. Such validation does not have to be performed on per-character-written basis. It is required for the program to validate input when user attempts to commence computation. At this point, if the validation fails, either:

- a dialog is presented with information about this fact,
- or text box with the input data is highlighted by changing its border to red line that has width between 2 and 4 pixels; and text informing the user about the error is added to the status bar.

Program will prohibit user from changing input data after the computation has started. Editing capabilities will be restored after the computation has finished, or after the user aborts current computation before it has concluded and the final solution is presented.

References

[1] Wikipedia contributors, *Regular language*, 2013 https://en.wikipedia.org/wiki/Regular_language, accessed 25 Feb 2013.

[2] Wikipedia contributors, *Finite-state machine*, 2013 https://en.wikipedia.org/wiki/Finite-state_machine, accessed 23 Feb 2013.

[3] Wikipedia contributors, *Kleene star*, 2013 https://en.wikipedia.org/wiki/Kleene_star, accessed 25 Feb 2013.