Warsaw University of Technology
Faculty of Mathematics and Computer Science

ΦNITE

APPLICATION FOR BUILDING FINITE-STATE MACHINE 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, 21 Mar 2013

Document metric

Project	ΦNITE	**Company**	WUT
Document name	technical analysis		
Document topics	technical analysis of the problem, details of used algorihms, interface specification		
Author	Mateusz Bysiek		
File	bysiekm-technical-analysis.pdf		
Version no.	0.16 **Status** pre-alpha	**Opening date**	16 Mar 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 developer team leader perspective, i.e. providing definitions of expected application environment, developer environment, user interface requirements, and all relevant algorithms.		
Authorized by	dr Lucjan Stapp	**Last modification date**	21 Mar 2013

History of changes

Version	Date	Author	Description
0.10	16 Mar 2013	Mateusz Bysiek	creation of template for the document
0.11	15 Mar 2013	Mateusz Bysiek	added GUI mockup
0.12	15 Mar 2013	Mateusz Bysiek	added 1st algorithm
0.13	16 Mar 2013	Mateusz Bysiek	added 2nd algorithm
0.14	17 Mar 2013	Mateusz Bysiek	added class diagrams
0.15	19 Mar 2013	Mateusz Bysiek	added 3rd algorithm
0.16	20 Mar 2013	Mateusz Bysiek	added abstract

Contents

1 The runtime environment **4**
1.1 Hardware requirements . . . 4
1.2 Operating system . . . 4
1.3 Additional software . . . 4

2 Development process constraints **4**
2.1 Developing the application . . . 5
2.2 Source code documentation . . . 5

3 GUI mockup **6**
3.1 Main menu . . . 6
3.2 Main area . . . 6
3.3 Status bar . . . 6

4 Classes **7**
4.1 Regular expression . . . 7
4.2 Finite-state machine . . . 8

5 Algorithms **9**
5.1 Conversion of plain text into a RegularExpression object . . . 9
5.1.1 Tagging . . . 9
5.1.2 Tag counting . . . 9
5.1.3 Parsing . . . 9
5.1.4 Optimization . . . 10
5.2 Conversion of RegularExpression object into FiniteStateMachine object . . . 12
5.2.1 Graphics . . . 12
5.2.2 Step-by-step construction . . . 12
5.2.3 Ending the computation . . . 14
5.3 Evaluation of a given word using FiniteStateMachine object . . . 15
5.3.1 Graphics . . . 15
5.3.2 Procedure . . . 15

Abstract

This is a technical analysis of problem of creating an application for building finite-state machine that is equivalent to a given regular expression and for simulating machine's evaluation of a given word. Author of the document analyses the problem of designing an application for solving a well defined language-theory related problem, doing it from the developer team leader perspective, i.e. providing definitions of expected application environment, developer environment, user interface requirements, and all algorithms used to perform the regular expression and finite-state machine related tasks.

1 The runtime environment

This section provides a set of requiremets with regard to the environment, in which the developed program will be run.

Definition

- *the application* is a synonym for Φnite, the term includes the runtime of Φnite with the user documentation, but not the source code and class documentation.

1.1 Hardware requirements

The computer on which the application is run must conform with the minimum requirements that are defined for operating system and the components listed in further sections. The application itself must require at most 1GB of hard drive space and 1GB of RAM to work. The actual implementation does not have to use all the available space, but it must work within the boundaries.

1.2 Operating system

The application is expected to work on the following set of operating systems:

- Microsoft Windows 7 Professional x86
- Microsoft Windows 7 Professional x64
- Microsoft Windows 7 Ultimate x86
- Microsoft Windows 7 Ultimate x64

1.3 Additional software

Other than the operating system, the application cannot be expected to work corretly unless all of the following components are present in the operating system:

1. .NET Framework 4.0 Full Profile
2. PDF (Portable Document Format) file viewer
3. PDF-LaTeX toolkit that is able to create PDF files from LaTeX source code automatically, using command-line.

The application may seem to work correctly without some of these components being present, but it is undefined what will happen.

The development team shall provide portable (in the sense that their installation and/or configuration by the user will not be required in order for them to work) runtimes for the latter two components. The first component has to be installed by the user.

2 Development process constraints

This section provides a set of requiremets with regard to the environment, in which the application and its documentation will be developed.

Definitions

- *the project* is term used for all of the listed: the runtime of Φnite with all documentation, the source code and required external libraries.
- the development process will follow the waterfall model

2.1 Developing the application

- Visual Studio 2012 Ultimate will be used
- C# will be used to implement main algorithms of Φnite: for building finite-state machine that is equivalent to a given regular expression and for simulating machine's evaluation of a given word
- .NET Framework 4.0 features such as lambda expressions and Linq will be used to simplify the implementation proces and make the code more readable.
- WPF (Windows Presentation Framework) will be used to create the user interface, via use of C# and XAML

2.2 Source code documentation

- All namespaces introduced in the development process have to be documented.
- All classes, interfaces and enumerated types have to be at least briefly described.
- All public properites and UI event handlers also have to be documented.
- The source code documentation will follow the standard C# documentation rules, because the Visual Studio has built-in support for creation of such documentation.
- Where the rules of C# documentation are not enough, the doxygen documentation format can be used: ex. to group classes into modules, to document namespaces, etc.
- Doxygen will be used to process the source code to create a set of web pages (which use HTML and CSS) with the complete documentation.

3 GUI mockup

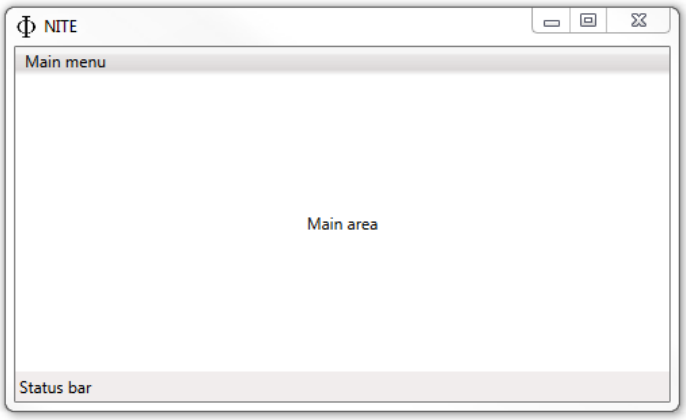

Figure 1: schema of the graphical user interface

3.1 Main menu

Main menu contains all general options that are required to be implemented, but are not connected with a specific step of calculation. Contents of the menu must not change during application operation, but its certain entries may be temporarily disabled during some phases of computation.

Main menu will contain, ex. exit option, example selection option.

3.2 Main area

Main area will contain frequently changing content, as it may have any of the following: a text field for entering a regular expression, button to proceed to the next step of computation, button to abort computation, button to display final result, a table with intermediate results, a final result, etc. Content of main area depends on context, i.e. previous actions of the user and current status of application.

3.3 Status bar

Status bar must contain one of three phrases at all times, unless there are equivalent indicators implemented (mentioned in each point):

- "busy" - if program computes the final or intermediate result.

  Equivalent of this is locking all elements of the GUI while the program is busy.

- "awaiting user interaction" - if program is not busy, but intends to be right after user gives some information that may help with further computation. The phrase may be changed to other with the same meaning. This status is intended only for computation phase, and only for those parts of computation phase that require user input.

  Equivalent of this status is displaying a new window. All information and input fields needed by the user to help the program must be in this new window. When it is closed, program resumes computation.

  Since the user feedback mechanism is optional (provided that it is of course substituted with working implementation), in certain implementation scenarios this status will never occur (or window will never be shown). If development team is able to prove that this feature would really never be used, this status indication does not have to be implemented.

- "ready" - in all other situations, for example: after start-up, after completing the computation.

  Equivalent of this status is a situation in which both previous mechanisms are implemented using second variant. If it is so, this status can be omitted.

Status bar may also temporarily contain some optional content that may help the user in completing the user interaction phase, provided that the main area is not a proper place for this content.

4 Classes

This section goes through the most important classes used in the project.

4.1 Regular expression

Numerous classes are required to store regular expression from its raw form up to completely parsed form.

1. First of all, `PartialExpressionRole`, enumeration of roles that a given part of the expression can assume.
2. `UnaryOperator`, enumeration of unary operators that a given part of the expression can be affected by.
3. `PartialExpression`, a single node of a parse tree that stores the actual expression as a generative grammar.
4. Enumeration of tags used to tag the input character sequence named `InputSymbolTag`.
5. Finally, `RegularExpression`, an object that stores all the data about the regular expression.

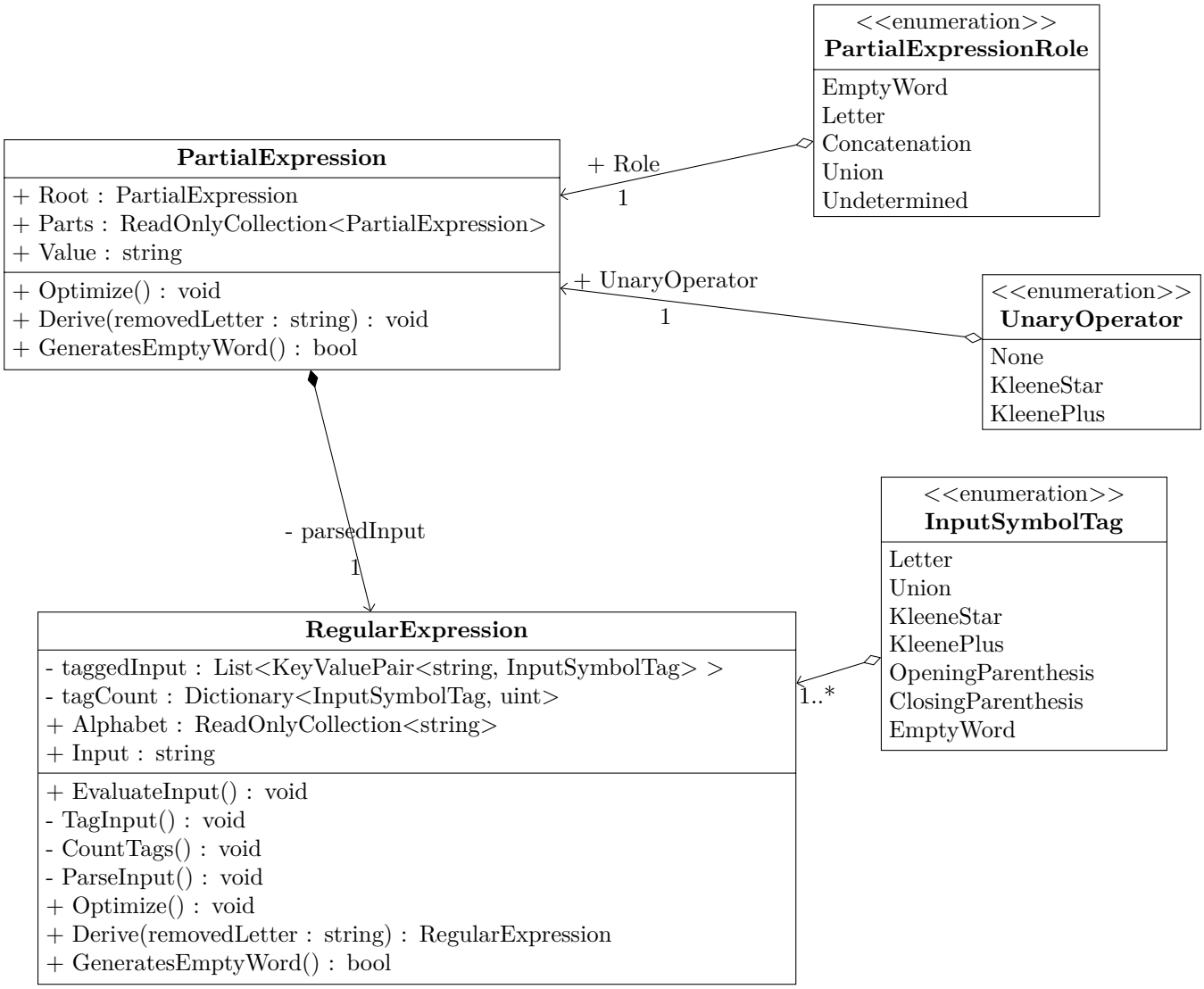

Figure 2: diagram of classes related to regular expression parsing

4.2 Finite-state machine

Few classes are required to store information about the constructed finite-state machine, because most of the computational complexity is on the side of `PartialExpression` class.

1. `MachineTransition`, that stores information about a single transition from a certain state to another.
2. Finally, `FiniteStateMachine`, a place to store all information about a single finite-state machine.

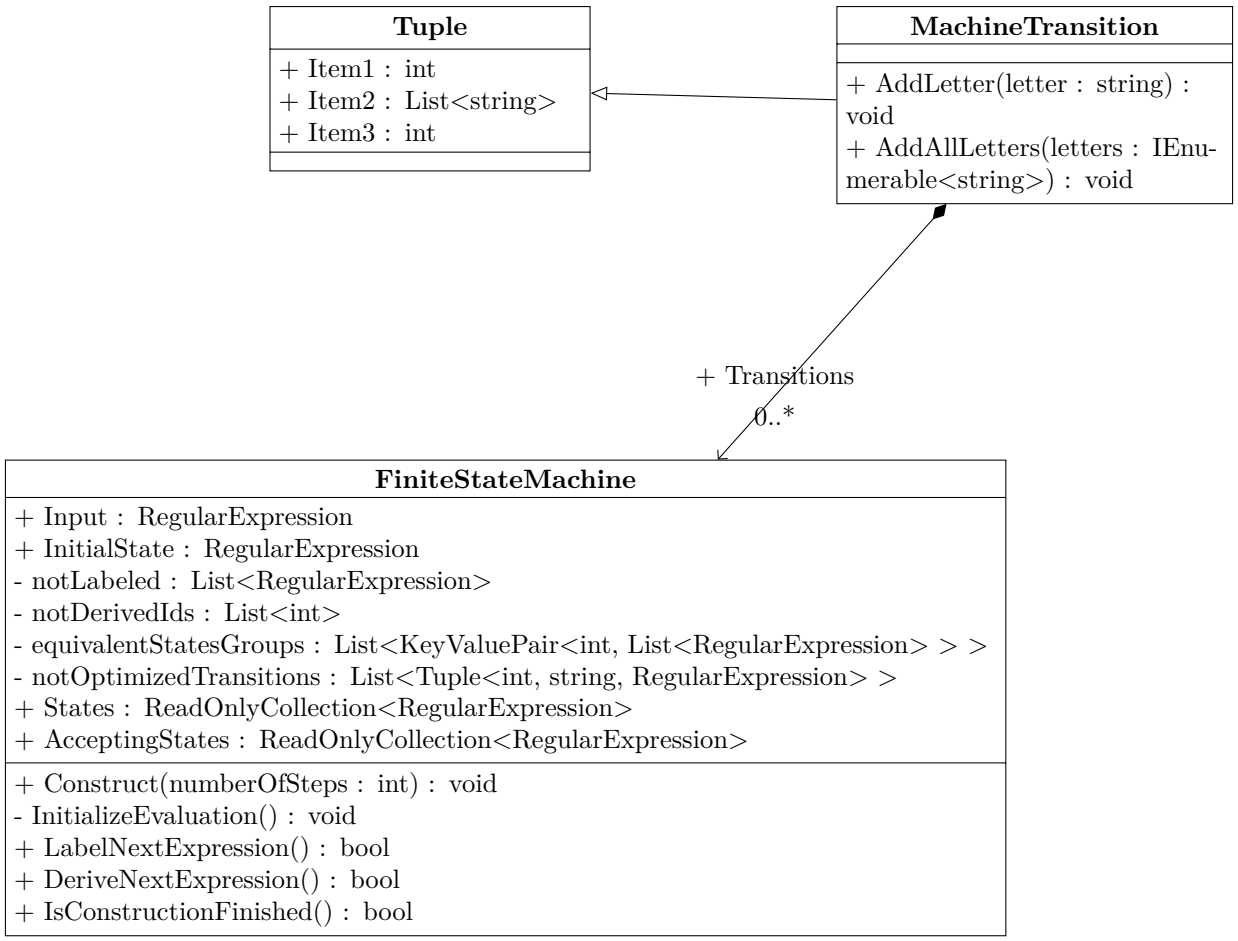

Figure 3: diagram of classes needed for finite-state machine related tasks

The development team must preserve the types and names of all public properties of all types mentioned in the two above sections. The names and return types of pulbic methods also have to be preserved. The names of all listed classes and enumerations also have to be exactly as specified. All public properties must be treated as read-only - some backing fields should be created and used for processing.

Other names and types can be changed, added or removed at will, provided that this is a properly justified decision, and all of the names still carry a meaning corresponding to their true role. The development team may also add some other classes that are used internally during the processing. One exception: the `parsedInput` field cannot be removed or changed.

5 Algorithms

5.1 Conversion of plain text into a RegularExpression object

The algorithm should follow the standard procedure applied in the are of natural language processing. Of course in case of this algorithm that process is simplified greatly because of the fact that we do not parse a human language but a machine language.

5.1.1 Tagging

First step is to tag the input data accordingly. This is done for several reasons:

1. As the tagging is done, algorithm eliminates ignored symbols by not adding them to the tagged input.
2. The tagging eliminates the difference between letters, all non-special symbols are simply regarded as "letters". From the point of view of further parsing, this greatly simplifies the process.
3. As the input is tagged, it becomes much easier to detect which parts of it belong together.
4. The validation and therefore stopping in case of some mistakes like open bracket without a corresponding closing one, or placing a unary operator after an opening bracket, etc. is simplified.

Program shall go through all the letters and assign a tag to each of them. The direction of the processing (left to right or right to left) does not matter. In fact, the solution can, but does not have to be, parallelized. In case of sequential solution the worst case running time must be $O(n)$, in case of parallel solution $O(n/k)$ assuming k is the number of cores used, $k \leq n$.

The developer should pay attention to the fact that some special symbols consist of several characters. In such cases a single tag is placed over all of the characters that constitute such symbol. Therefore the parallelization may add some complications.

Results of tagging are stored in `taggedInput` field. Relevant tag counts in the `tagCount` field are incremented on-the-go.

5.1.2 Tag counting

In this step the algorithm simply checks (using already prepared `tagCount` field) the counts of parentheses to ensure that there is the same number of opening and closing ones.

5.1.3 Parsing

Third step is to parse the tagged input into a tree. A tree created by parsing a plain text using a certain grammar is called a parse tree in research texts related to natural language processing.

The steps of the algorithm and definitions used later on:

1. a new, empty `PartialExpression` is created and it becomes a current part
2. current part is always labeled as "Part"
3. Part becomes a root part of the currently created parse tree
4. root of the current parse tree is always labeled as "Root"
5. the role of Part is set to `Undetermined`
6. role of the current part is always labeled as "Role"
7. the algorithm goes through all characters of the input, starting from the first
8. at each step, the currently evaluated symbol is labeled as "Symbol"
9. at each step, the actions of the algorithm depend on the Role, Symbol and Root; these actions are described in a separate table that follows
10. if there are no more symbols to parse, the part that is Root is set as a root of the parse tree of the whole regular expression (field `parsedInput` of `RegularExpression`)

The procedure is a recursive one. Parentheses are the triggers for recursion. The below table shows the behaviour of the algorithm. This table is a human-readable version of an underlying grammar that is used to create the parse tree.

Role	Symbol	Root	result
`Undetermined`	`EmptyWord`	$*$	1) Role is set to `Concatenation` 2) a new part with role `EmptyWord` is added to parts of Part
`Undetermined`	`Letter`	$*$	1) Role is set to `Concatenation` 2) and a new part with role `Letter` and value of the Symbol is added to parts of Part
`Concatenation`	`EmptyWord`	$*$	a new part with role `EmptyWord` is added
`Concatenation`	`Letter`	$*$	a new part with role `Letter` and value of the Symbol is added
`Concatenation`	`Union`	$\equiv$Part	1) a new part with role `Union` is created, and this new part is set as Root; 2) Part is added to the list of parts of Root (becoming the first part in this list) 3) a new part with `Undetermined` role is created, it is appended to the parts of the Root and it is set as Part
`Concatenation`	`Union`	$\not\equiv$Part	a new part with `Undetermined` role is created, it is appended to the parts of the Root and it is set as Part
`Concatenation`	`OpeningParenthesis`	$*$	1) the sub-procedure of parsing starts from step one (see algorithm), and the sub-procedure treats the first symbol after the parenthesis as the first symbol of input 2) when the sub-procedure ends, the returned partial expression is appended to the list of parts of Part 3) the procedure skips all symbols parsed by the sub-procedure and continues
`Concatenation`	`ClosingParenthesis`	$*$	the parsing sub-procedure ends, Root is returned
`Concatenation`	`KleeneStar`	$*$	the operator of the last of the parts of Part is set to `KleeneStar`
`Concatenation`	`KleenePlus`	$*$	the operator of the last of the parts of Part is set to `KleenePlus`

All other encountered pairs of Role-Symbol result in an undefined behaviour. Such pairs would however violate the rules of the underlying grammar, therefore they can, and shall be detected in the tagging step. Also, in the table there are some assumptions about data integrity which are ommited. Catching of invalid tag sequences in before starting parsing phase and correct implementation of the above algorithm ensures data integrity i.e. ensures that the correct parse tree will be built.

5.1.4 Optimization

The fourth and the last phase is to optimize the parse tree i.e. eliminate any redundant productions and useless symbols. This step is optional, but highly recommended.

Parse tree has redundant productions if its underlying grammar can generate the same word using more than one set of productions. Eliminating the redundant productions means that the grammar still generates the same language, but fewer (or no) productions are redundant.

Below is the list of patterns that are considered not optimal, together with methods of optimization:

1. A part that has role of `Concatenation` or `Union`, and has only one part - it shall be converted to the part itself.

   In this case special rules regarding operators apply:

Operator of parent	Operator of its single part	resulting operator
`None`	$*$	$*$
$*$	`None`	$*$
`KleeneStar`	$*$	`KleeneStar`
$*$	`KleeneStar`	`KleeneStar`
`KleenePlus`	`KleenePlus`	`KleenePlus`

   The algorithm shall analyze if the rules apply to the current situation from top to bottom, and apply the first matching rule.

2. A part that is a part of `Concatenation` and its role is equal to `EmptyWord` - such part can be deleted altogether unless it is the only part of that `Concatenation`.

3. If a `Union` has some parts that are equal, the duplicates can be safely deleted.

4. If we have two consequtive parts of a `Concatenation` that satisfy all of the following conditions:

   - both have unary operators
   - those unary operators may be different or the same
   - without taking operators into account, these parts are equal

   then, one of the parts can be safely removed provided that the operator of the other part is modified accordingly:

initial operator of the remaining part	operator of removed part	resulting operator of the remaining part
`KleeneStar`	`KleeneStar`	`KleeneStar`
`KleeneStar`	`KleenePlus`	`KleenePlus`
`KleenePlus`	`KleenePlus`	`KleenePlus`
`KleenePlus`	`KleeneStar`	`KleenePlus`

5. If we have a pair of parts of a `Union`, which are placed anywhere in the union, that satisfy all of the following conditions:

   - either both of them have any (but different) unary operator, or one of them has `KleenePlus` and the other has no operator
   - without taking operators into account, these parts are equal

   then, one of the parts can be safely removed provided that the operator of the other part is modified accordingly:

initial operator of the remaining part	operator of removed part	resulting operator of the remaining part
`KleenePlus`	`None`	`KleenePlus`
`None`	`KleenePlus`	`KleenePlus`
`KleeneStar`	`KleenePlus`	`KleeneStar`
`KleenePlus`	`KleeneStar`	`KleeneStar`

   and cases where operators are identical are covered by point 3. of this list.

6. If some part x is a is a `Concatenation`, it has no operator, and it is a part of another `Concatenation` y, which also has no operator, then all parts of x can be moved into y.

   For example: $y = ay$, $x = bc$, then, instead of having $y = a(bc)$, we can safely have $y = abc$

7. If some part x is a is a `Union`, it has no operator, and it is a part of another `Union` y, which also has no operator, then all parts of x can be moved into y.

   For example: $y = a + y$, $x = b + c$, then, instead of having $y = a + (b + c)$, we can safely have $y = a + b + c$

8. If some partial expression is a `Union` and has `KleeneStar`, then every its part that is a `EmptyWord` can be safely deleted, beceause the empty word will be generated anyway.

   For instance: $(a + \epsilon + b)^* \equiv (a + b)^*$

This step is needed to ensure that the implementation second algorithm can be greatly simplified. The implementation of the second algorithm becomes much more difficult without assumption that the parse tree is optimal.

It is still possible to implement the derivation and labeling algorithm without optimizations, that is why the implementation of optimization is not mandatory.

5.2 Conversion of RegularExpression object into FiniteStateMachine object

5.2.1 Graphics

At the begining, a screen is displayed where input word is shown, and two empty tables are displayed: one for list of labeled states and another for discovered transitions. In each step, entries in tables may be added if in a given step some new state is labeled or is derived.

Moreover, there is a place reserved (in a scroll box) to display a graphical representation of those two tables in their current state. This representation will follow coloring of the example in business analysis.

Moreover, the states derived in the last step have changed their border colors to gray. And, the state labeled in latest step is highlighted by having its background changed to gray.

All states that exist longer that since latest step follow the usual coloring scheme.

5.2.2 Step-by-step construction

The two following (labeling and derivation) steps are executed in the loop, until the computation is complete. Conditions for ending the computation are described in relevant section later on.

Splitting the computation into loop consisting of small, repetative steps, is necessary for implementation of step-by-step computation. When immediate result is needed, a loop can be simply set to run without interruption.

Labeling step

1. The first element from `notLabeled` list is taken, it becomes the current element: "Element".
2. If `equivalentStatesGroups` is empty:

   - the Element is added to that list with $key = 0$

   else:

   (a) The `equivalentStatesGroups` list is checked whether it contains Element. If it does, the algorithm goes to point 3.

   (b) if it does not, then the algorithm breaks and the user is prompted to provide his opinion regarding the Element, i.e. if in user's opinion some state from `States` is equivalent to the Element.

   (c) When the user answers false, the algorithm continues and adds new entry to `equivalentStatesGroups`, with new key =`equivalentStatesGroups.Count` and goes to point 3.

   (d) When the users indicates to what already labeled state the Element is equivalent, the algorithms adds a new state to the list of states with key equal to the user indicated state number - in the field `equivalentStatesGroups`.
3. Algorithm iterates through `notOptimizedTransitions` and for each its entry that has the resulting state equal to Element.

   (a) Adds a new `MachineTransition` to `Transitions` with the corresponding ids and letter(s).

   (b) Removes the entry from `notOptimizedTransitions`.
4. The Element is removed from `notLabeled` list.

Derivation step

1. The first element from `notDerivedIds` list is taken, then its corresponding element from `equivalentStatesGroups` is taken and it becomes the derived element: "Element".
2. The Element is derived for each letter that exist in its alphabet. The derivation procedure is as follows:

   (a) A copy of parse tree of Element is made, this copy becomes the currently processed tree.

   (b) Currently processed tree is alwayes labeled "Tree".

   (c) The Tree is processed using the rules contained in the tables that follow. In that tables: Role is simply the role of Tree, Operator is the Tree's unary operator.

(d) At first the transformations from the first table are applied, then those from the second.

(e) A new `RegularExpression` object is created from a transformed Tree, and it becomes the derived element.

3. for each derived element:

   - if derivation is not null (that is equivalent to entering the rejecting state), a new entry is added to a list of not optimized transactions: `notOptimizedTransitions`, and that entry has: a) the id from `notDerivedIds` corresponding to Element (exactly the same as in step 1.), b) the letter by which the derivation was made, and of course c) the result of derivation

4. the id of Element is removed from `notDerivedIds`

The first table with transformations:

Operator	actions
`None`	none
`KleeneStar`	1) new part with copy of Tree is created, but copy's operator is set to `None`; 2) new part with copy of Tree is created; 3) a `Concatenation` of parts from 1) and 2) is created; 4) new part that is an `EmptyWord` is created; 5) Role is set to `Union`; 6) Operator is set to `None`; 7) All Tree parts are removed; 8) part from 4) is added to Tree parts; 9) part from 3) is appended to Tree parts
`KleenePlus`	1) new part with copy of Tree is created, but copy's operator is set to `None`; 2) new part with copy of Tree is created, but copy's operator is set to `KleeneStar`; 3) Role is set to `Concatenation`; 4) Operator is set to `None`; 5) All Tree parts are removed; 6) part from 1) is added to Tree parts; 7) part from 2) is appended to Tree parts

The second table with transformations:

Role	actions
`EmptyWord`	Role is set to `Invalid`
`Letter`	if value equals to that of derived letter, Role is set to `EmptyWord`, otherwise Role is set to `Invalid`
`Concatenation`	**if** 1st part has a `KleeneStar` operator: 1) new partial expression is created as a copy of this 1st part, but operator of the copy is set to `None`; 2) new partial expression is created as a copy of Tree, and 1st part of this copy is removed; 3) yet another partial expression is created as a copy of Tree, and 1st part of this copy is set to expression created in 1); 4) Role is set to `Union`; 5) All Tree parts are removed; 6) part from 2) is added to Tree parts; 7) part from 3) is appended to Tree parts **else if** 1st part is a union that has empty word: 1) new partial expression is created as a copy of this 1st part; 2) all parts of this copy that are empty words are removed; 3) another partial expression is created as a copy of Tree, and 1st part of this copy is set to expression created in 1-2); 4) new partial expression is created as a copy of Tree, and 1st part of this copy is removed; 5) Role is set to `Union`; 6) All Tree parts are removed; 7) part from 3) is added to Tree parts; 8) part from 4) is appended to Tree parts **else**: for the first part of Tree, launch the derivation sub-procedure where only steps from 2.(b) to 2.(d) are executed
`Union`	for each part in Tree parts, launch a derivation sub-procedure where only steps from 2.(b) to 2.(d) are executed

5.2.3 Ending the computation

Computation is regarded as complete when after the derivation step, both `notLabeled` and `notDerived` lists are empty.

LaTeX output

There will be a feature that consists of constructing a latex soure and pdf file from that source. The latex code is created if user selects a relevant option. This option is made available only after the machine is fully constructed. The latex file will contain two tables: with labeled states and with transitions. It will also contain a code that, after processing by pdf-latex software, will generate a simple graphical representation of the answer. This representation must follow standard formatting and coloring used in mathematics to draw finite-state machine. The representation may differ from the one presented in the graphical user interface.

After the pdf file is generated, it should be automatically displayed to the user. If ther are any errors in the processing or creating the latex code, or displaying the made pdf file, the user must be notified via a message box.

5.3 Evaluation of a given word using FiniteStateMachine object

As the input, the algorithm takes fully constructed finite-state machine, and a word. Algorithm goes through the word from beginning to the end, letter by letter.

Remark:

User can use any characters to form the word, but if he/she uses some symbols that are used as special symbols in the process of constructing the machine, such as parentheses, the word will be always rejected, because no machine constructed by the application has transition over letter `(` or`)`.

5.3.1 Graphics

The word evaluation algorithm will use the same graphical representation of the machine as it was used for its construction. The difference is that the tables are not displayed, and the input expression is substituted with the input word that is evaluated.

Remark:

The machine does not change in current algorithm - only the highlighted state changes.

5.3.2 Procedure

1. All characters of input are marked as not processed.
2. Current state of machine is set to its `InitialState`. Current state is always highlighted by either interchanging background color with text color, or by changing background to gray. Only one highlight scheme must be chosen, and has to remain the same at all steps.
3. A loop starts.
4. If there are no not processed letters, go to one step before the last.
5. If there are, check if there is a transition from current state to some state, over the currently processed letter of input. If there is, change current state and mark the letter as processed. If there is no such transition, highlight current state in oragne and go to the last step.
6. Loop ends, go back to step 3.
7. If current state is on the list of `AcceptingStates`, the current state is highlighted in green. Otherwise, it is highlighted in orange.
8. Algorithm ends.