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-business-analysis.pdf		
Version no.	0.15 **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 used 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	16 Mar 2013	Mateusz Bysiek	added GUI mockup
0.12	16 Mar 2013	Mateusz Bysiek	added 1st algorithm
0.15	20 Mar 2013	Mateusz Bysiek	added abstract

Contents

Abstract

Write here!

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

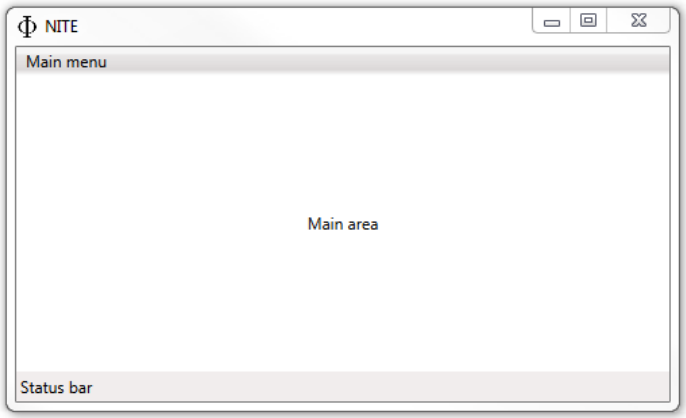

Figure 1: schema of the graphical user interface

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. First of all, enumeration of tags used to tag the input character sequence:

```
public enum InputSymbolTag
{
    Letter, Union, KleeneStar, KleenePlus, OpeningParenthesis, ClosingParenthesis, EmptyWord
}
```

An object that stores all the data about the regular expression:

```
public class RegularExpression
{
    public static readonly KeyValuePair<string, InputSymbolTag>[] ReservedSymbols;

    public static readonly Dictionary<InputSymbolTag, string> TagsStrings;

    public static readonly string[] IgnoredSymbols;

    public static readonly string[] ForbiddenSymbols;

    public string Input;

    public ReadOnlyCollection<string> Alphabet;

    private List<KeyValuePair<string, InputSymbolTag>> taggedInput;

    private Dictionary<InputSymbolTag, uint> tagCount;

    private PartialExpression parsedInput;

    private void TagInput();

    private void CountTags();

    private void ParseInput();

    public void Optimize();

    public RegularExpression Derive(string removedLetter);

    public bool GeneratesEmptyWord();
}
```

Enumeration of roles that a given part of the expression can assume.

```
public enum PartialExpressionRole
{
    EmptyWord, Letter, Concatenation, Union, Undetermined
}
```

Enumeration of unary operators that a given part of the expression can be affected by:

```
public enum UnaryOperator
{
    None, KleeneStar, KleenePlus
}
```

A single node of a tree that stores the actual expression in its parsed form:

```
public class PartialExpression
{
    public PartialExpressionRole Role;

    public PartialExpression Root;

    public ReadOnlyCollection<PartialExpression> Parts;

    public string Value;

    public UnaryOperator Operator;

    public void Optimize();

    public void Derive(string removedLetter);

    public bool GeneratesEmptyWord();
}
```

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.

A place to store all information about a single finite-state machine:

```
public class FiniteStateMachine
{
    public RegularExpression Input;

    public ReadOnlyCollection<RegularExpression> States;

    public ReadOnlyCollection<Tuple<RegularExpression, string, RegularExpression>> Transitions;

    public ReadOnlyCollection<RegularExpression> FinalStates;

    private void FindTransitions();

    private void FindFinalStates();
}
```

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 useless productions. This step is optional, but highly recommended.

Useless productions are understood here as expression parts that are not optimal, and are defined as those that are of one of the following kinds:

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`	`KleenePlus`	`KleenePlus`
`KleenePlus`	`KleeneStar`	`KleenePlus`

   and cases where operators are identical is covered by point 2. of this list.

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 is covered by point 3. of this list.

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

Write here!

5.2.1 Initialization

Write here!

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

Write here!

Derivation step

Write here!

5.2.3 Ending the computation

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

5.3 Evaluation of a given word using FiniteStateMachine object

Write here!