Computer Science. Vol. 3. 2001 519.713:81'32:004

Michał Korzycki[*]

FINITE-STATE METHODOLOGY IN NATURAL LANGUAGE PROCESSING

Since the Antiquity, the nature of the language has intrigued scientists and philosophers. They have tried to unveil the patterns and regularities, that seem to rule our main communication tool. Through centuries various grammars have attempted to systematize our pragmatic knowledge in this area.

Recent mathematical and algorithmic results in the field of finite-state technology, as well as the increase in computing power, have constructed the base for a new approach in natural language processing.

But the task of creating an appropriate model that would describe the phenomena of the natural language is still to be achieved. In this paper I'm presenting some notions related to the finite-state modelling of syntax and phonology.

1. Introduction

Grammar, or, as it has now been called, linguistic theory, is in contrast with other Natural Sciences in what concerns the approach toward the analyzed data. In Biology or Astronomy, a significant part of the accumulated data is usually treated as a noise insignificant to the global result of the measurement. The linguist must for example consider such fact, that the essential information of an average text is contained in its statistically unsignificant part. This turns the linguistic theory into a science concentrated not on the accumulation of data, but on the elaboration of general rules describing the properties of the analyzed material, without any sort of systematic observation. That approach leads easily to overgeneralization. To take a well-known example, using the grammatical categories of Classical Greek to describe exotic languages is often irrelevant [1].

The earliest grammatical models formalized sentence forms by replacing each word with its grammatical category. Owing to its conceptual simplicity, this model has been repeatedly used under different names. It might be proper to call this model Markovian – as

[*] Katedra Informatyki, Akademia Górniczo-Hutnicza, Kraków

Markov introduced it to study phonetic sequences. A much more refined model has been presented by N. Chomsky in 1957 in the form of, what he called, context-free grammars. This model is at the base of the contemporary theory of formal languages and grammars. But again, any of those types of grammar were restricted to the description of the linear order of categories with simplistic relations between them. Chomsky himself pointed the inadequacy of this model to describe the complex phenomena of the natural language.

It seems that the rules which govern the language are strongly context-restricted. It means that the behaviour of separate elements is strongly determined by their surroundings. What's more, this context must be taken from all the structural layers of the language – such as morphology (phonetical adjustment of the lexical stem to the suffixes), syntax (modifications of the lexical stem depending of the lexical form), and (which is especially difficult to formalize) semantics.

The definition of the local grammar, given by W. Woods (1970), was an important step in our attempt to obtain a full scale analysis of the language. He stipulated that the **global nature** of the language results from the **interaction** of a multiplicity of **local finite-state schemes** which we call finite-state local automata, or shortly, **local grammars**. This is a consequence of the language being a product of an evolutionary process, and the complex phenomena are a result of interaction between various grammars that date back to various historical periods.

2. The finite-state framework

The theory of finite-state automata is rich, and finite-state techniques are used in a wide range of domains. Finite-state systems have been present since the emergence of computer science, and are extensively used in such various areas as operating system analysis (ie. Petri-nets), program compilation, handwriting recognition or speech processing. So it's not suprising that finite-state formalisms have also been vastly employed in the recent industrial and academic researches aiming to create formal models of natural language processes.

One of the more elaborated contemporary models of the natural language is based on a formal description of **representations** and **relations** between them. Sometimes these restrictions refer to only one representation, and we prefer to call them **constraints**. A representation is a specific form of a language element (such as a word, sentence, morphem etc.). This can be its directly observable shape, such as its written form. Others are invented by the scholars to serve as building blocks of a theory describing specific phenomena. We will name the first form the **surface** representaion, the latter – the **lexical** one.

This model, known as two-level morpology, has been described by K. Koskenniemi in 1983 using a finite-state interpretation. The papers written by Kaplan and Kay in 1994 suggest that to describe this model, the use of a variant of the classical finite-state automaton (FSA) – the finite-state tranducer will be the most appropriate (a more exact description can be found in 2.2).

This tool proved itself to be even more useful, as the notion of context has been definied with the use of this formalism [2].

2.1. Finite-state automata and regular expressions

As a reminder, and to set the notation, I will now give the definition of the finite-state automaton.

A finite-state automaton is a quintuple

$$(\Sigma, Q, i, F, E),$$

where:

Σ –	a finite set called the alfabet,
Q –	a finite set of states,
$i \in Q$ –	the initial state,
$F \subset Q$ –	the set of final states,
$E \subset Q \times (\Sigma \cup \{ \varepsilon \}) \times Q$ –	the set of edges – (where ε – is the empty word).

One of the strongest properties of FSAs is the fact, that there is a correspondence between those automatons and a certain class of formal languages usually called **regular languages**. That makes the FSAs a very versatile tool, as the class of regular languages is closed under such operations as set union, set intersection, concatenation, Kleene Star etc. (it is a consequence of the Kleene theorem – 1956). This closure property can be applied to FSAs, what makes them equivalent to their syntactical description. As a result, most of the expressions used in computer programs are equivalent to an automaton. Such strong formal properties are uncommon to other formal tools used in natural and formal language processing. This makes the FSAs especially suitable for a vast range of theoretical applications.

To describe the FSA pictorially we adhere to the following convention: the FSA is treated as a graph, the states marked as circles are its nodes, and the FSA edges are the edges of this directed graph. The labels of the edges are alphabet letters that belong to the specific edge, the final states are depicted as two concentric circles, and the starting state is usually marked by an arrow signed "start".

Presented below is a range of symbols and operators used to describe regular expressions and corresponding regular languages:

`a`	– a single letter usually describes a single alphabet symbol,
`` `a' ``	– also a single alphabet symbol; the quotes are only used to avoid misinterpretations ex. where a symbol has the same graphical value as a regular expression operator,
`` `A123' ``	– once more a single alphabet symbol – one with a longer name,
`?`	– this symbol describes a language composed exclusively of all the singletons over the given alphabet,
`[]`	– an empty pair of bracket denotes the empty symbol i.e. the word of length 0 (also marked as ε),
`[ A ]`	– usually the brackets only describe the order in with the operations are executed. This expression has the same meaning as **A**,
`A B`	– this is an operation of concatenation between two regular languages – **A** and **B**,
`A~`	– Negation; denotes the language which is the set complement of A to the full language $\Sigma*$. i.e. the language containing all the strings (words) over Σ (Fig. 1),

A* – Kleene Star. Describes the language composed of 0 or more repetitions of the language **A** (Fig. 2),

A+ – Kleene Plus. Describes the language composed of 1 or more repetitions of the language **A** (Fig. 3),

A $ – Non contenance. Describes all strings containing no string from **A** (Fig. 4),

A | B – Set union. This language is composed of all the strings from **A** and all the strings from **B**,

A / B – Ignore. Describes the strings of **A** possibly interspersed with strings from **B** .

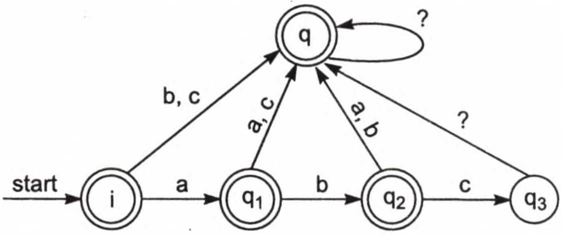

Fig. 1. The automaton corresponding to the **[abc~]** expression over the **{a,b,c}** alphabet

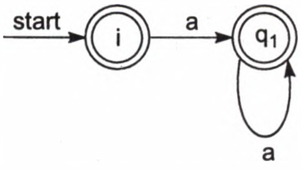

Fig. 2. The automaton accepting the **a*** language

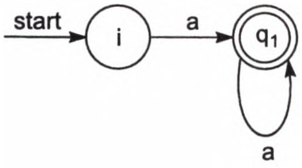

Fig. 3. The automaton accepting the **a+** language

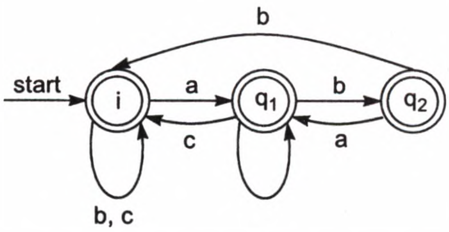

Fig. 4. The automaton accepting the **[abc]$** language over the **{a,b,c}** alphabet

Example

[a*]/b gives: ε, **a, ab, bab, baab,** etc. (Fig. 5).

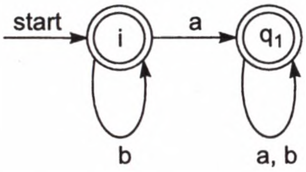

Fig. 5. The automaton accepting the language **[a*]/b**

2.2. Finite-state transducers and regular relations

We understand By the term **regular relation** a subset of a Cartesian product of two regular languages. This also often interpreted as a mapping from one regular language to another. Automata operating on such relations are called finite-state **transducers** (FST). The main difference between a transducer and a "classical" FSA (as defined in the last section) is the fact that in a transducer, the edges are labelled with **pairs** of symbols instead of singletons. Although transducers, on the whole, do not conserve all the strong algebraic properties of FSAs, they can still be regarded as equivalent to a certain set. Such set will always be a regular relation. In common applications, we treat the first symbol of a pair labelling an edge as an input symbol, and the second one as an output symbol. The graphical representation of a FST differs from the representation of a FSA by the fact that the graph is labelled with two symbols at each edge, both separated graphically with a slash ("/") sign. The graph searching algorithms that were used to decide the acceptability of an input in a FSA can also be applied to a FST. We consider only the left input symbol on the passed edges, and write on the output the right symbols that occured on our path. In his context, we can distinguish two distinct (but not exclusive) types of transducers. The ones that possess only left non-epsilon symbols, and those that have only right non-empty symbols. The first ones will be called acceptors, the latters, generators.

To describe regular relations we will use the calculus of extended regular expressions. Below we give the used extension to the classical regular expression calculus.

A .x. B – crossproduct (Cartesian product) of the regular language **A** and the the regular language **B**. It denotes all the string pairs whose first element belongs to **A** and the second to **B**. The implementing automaton is obtained by the concatenation of the acceptor of the language **A** with the generator of the language **B**.
Because we regard the identity relation on **A** as equivalent to **A**, we can, and usually do, simply write **a** instead of **a.x.a**.

R S – the operation of "coordinate-wise" concatenation of two regular relations (transducers). The obtained relation is a result of a coordinate-wise concatenation.

Example

`[a .x. b] [c .x. d]` gives **`[ac .x. bd]`**

`A -> B` – simple replacement operator. Replaces all the occurences of strings from one language (**A**) with strings of an other language (**B**). We define it as follows

`A->B := [ NOUPPER [A .x. B] ]* NOUPPER`

where **`NOUPPER`** designates **`[A-{ e }]$`**.

Example

For **`a -> b`** (Fig. 6)

the words:

```
aca                 aba                 aac
```

will be changed into:

```
bcb                 bbb                 bbc
```

`A -> B __ C` – marking operator. This operator precedes ("marks") all the occurences of the strings from **`A`** with strings from **`B`**, and follows the strings from **`A`** with the strings from **`C`**.

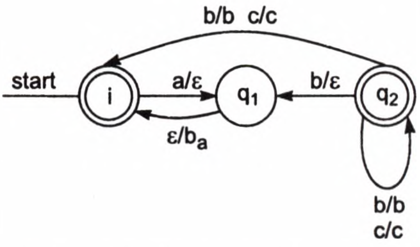

Fig. 6. The transducer implementing the **`a->b`** relation over the **`{a.b.c}`** alphabet

Example

For **`a -> b __ c`**

the words:

```
aca                 aba                 aac
```

will be converted to:

```
baccbac             bacbbac             bacbacc
```

`R .o. S` – composition of relations. It happens that, given two regular relations (transducers) **R** and **`S`**, a transducer can be defined that corresponds to their composition **`R .o. S`**. For additional details on this fact, the reader should refer to the reference marked as [4]. Technically speaking, the composition of transducers can be regarded as connecting the output of the first one to the input of the second.

Example

`[a.x.b] .o. [b.x.c]`

gives

`[a.x.c]`

`A -> B || L __ R` – conditional replacement operator. Replaces the occurences of strings from the language A to strings of the language B, but only if the string from the language **A** lies between the left context **L** and the right context **R**.

It can be defined as a composition of four relations:

```
                 [ L -> [ ] __ `<<' ],
                         .o.
                 [ R -> `<<' __ [ ] ],
                         .o.
          [ `<' A / [`<<' | `>>'] `>' -> B ],
                         .o.
              [ [ `<<' | `>>' } -> [ ] ].
```

The first relation marks all occurences of the left context.

Example

For **`a -> b || c __ d`**

the word	**c**		**a c**		**a d a**
will be changed into	**c**	**`<<'**	**a c**	**`<<'**	**a d a**

The second relation marks all occurences of the right context.

Example

For **`a -> b || c __ d`**

the word	**c**	**`<<'**	**a c**	**`<<'**	**a**		**da**
will be changed into	**c**	**`<<'**	**a c**	**`<<'**	**a**	**`>>'**	**da**

The third relation actually applies the replacement operator.

Example

For **`a -> b || c __ d`**

the word	**c**	**`<<'**	**a c**	**`<<'**	**a**	**`>>'**	**da**
will be changed into	**c**	**`<<'**	**a c**		**b**		**da**

The last relation removes all occurences of the auxiliary symbols **`<<'** and **`>>'**.

Example:

For **`a -> b || c __ d`**

the word	**c**	**`<<'**	**a c**	**`<<'**	**a**	**`>>'**	**da**
will be changed into	**c**	**`<<'**	**a c**		**b**		**da** (Fig. 7).

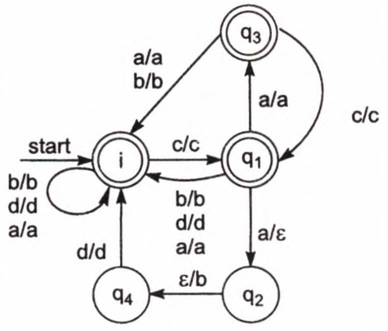

Fig. 7. The transducers implementing the **a -> b|| c __ d** expression over the **{a,b,c}** alphabet

This operator has many variants, as the context can be used in many ways to constrain the operator. More details can be found in [2].

%L% – in this the way I will distinguish a transducer symbol defined earlier from its graphical value.

Example

For **L:= [a.x.b]**

%L% .o. [b.x.c]

gives

[a.x.c].

3. Samples

The following sections present some simple examples of the application of this formalism to model the phenomena of the natural language. All the graphs and results presented in this text have been obtained by using of an implemented regular expression compiler.

3.1. Hyphenation

This simple example to ilustrate typical methods used in this approach toward the modelling of the phenomena of the natural language. The model desribes the principle of hyphenation. It rules the formation of new words by the addiction of the prefix "co-". The sign "–" (the hyphen) disappears when the modified word begins with a letter different from "o".

We begin by marking the beginning of the string by using a additional symbol "#". The first operator **? -> `#' __ [ ]** marks all the letters with the symbol "#", which gives us the following result for the word "operation"

#o#p#e#r#a#t#i#o#n.

The next operator **`#' -> [ ] || ? __ ?** removes the surplus of auxiliary signs

#operation.

After the addition of the prefix `` `#' -> c o `-' `` we obtain

`co-operation.`

Now we can remove the sign '-' if it appears before a sign different from "o":
`` `-' -> [] || o __ [o ~] `` gives (here without any changes for "operation")

`co-operation.`

But for the word

`education`

we would get

`coeducation.`

3.2. Simple conjugation modelling

This example illustrates the use of two-level morphology. The analyzed word will have two representations. The first one, the surface form, is the textual form of the word. The second one, the lexical one, is its symbolic representation dependent on its grammatical form.
Lets take, on the input, a typical Polish verb with the "ować" an ending of the infinitive.

`pracować` (to work).

We will start by marking the end of the input with an auxiliary symbol "&". We will use (as in the former example) two operators to obtain our goal.

The first one `` ? -> [ ] __ `&' `` marks all the letters with the symbol `` `&' ``, which gives us

`p&r&a&c&o&w&a&ć&`.

The next one `` `&' -> [ ] || ? __ ? `` removes the additional symbols.

`pracować&`.

We can now modify our word into its lexical form by means of the following operator `` ować`&' -> `+Inf' ``. That gives us now

`` prac`+Inf' ``

If we define the automaton **L** as
`L := uję | ujesz | uje | ujemy | ujecie | ują` which corresponds to the lexical suffixes of the present tense of the indicative mood, the operator
`` `+ Inf ' -> %L% `` gives now

pracuję	(I work),
pracujesz	(You work - sing.),
pracuje	(He/She works),
pracujemy	(We work),
pracujecie	(You work - pl.),
pracują	(They work).

This corresponds to the forms of this verb in the present tense of the indicative mood.

The compositions of all the operators gives for the input

gotować (to cook).

an output of:

gotuję	(I cook),
gotujesz	(You cook - sing.),
gotuje	(He/She cooks),
gotujemy	(We cook),
gotujecie	(You cook - pl.),
gotują	(They cook).

This type of modelling of linguistic phenomena has already been extensively used in the description of a vast group of languages – including English, Finnish, French, Russian, Swahili, Basque etc. The following example will hopefully show the vast range of problems that can be described with the use of this formalism.

3.3. Modelling a declination with a variable lexical stem

We discuss now a somewhat more complicated lexical example. Besides the addition of the lexical suffixes, we will be confronted with the variation of the lexical stem.

Let us suppose that we have on the input a typical Polish adjective

biały (white).

The first two operators of the last example will give us

biały&.

We obtain our lexical form by using of the following operator
y`&' -> `+Nom'. We now obtain

biał`+Nom'.

If we define the automaton **M**
as **M := y | i**

what corresponds to singular and plural suffixes of the Nominative case, the operator: **`+Nom' -> + %M%** gives:

biał+y,

biał+i.

We have obtained here an additional abstract form, in which the symbol **`+'** separates the lexical stem from the suffix. We can now adjust the stem phonetically:

The operator **ł -> l || ? __ `+' i** results in

biał+y,

bial+i.

Having removed of the auxiliary symbols `'+' -> [ ]` we obtain:

biały,

biali.

For lexical stems with no such endings, such adaptation will not occur. Taking as an example the word **czarny** `(black)` we obtain:

czarny,

czarni.

3.4. Modelling mobile lexical verb suffixes

In the Polish language there exists a phenomenon called mobile lexical verb suffixes. The lexical suffixes of a verb in the third person of the past tense of the indicative mood can be attached, instead of the verb, to almost any other part of the sentence. I present here a method to localize such suffixes and place them back in their place after the verb. It will show that finite-state rules can also be used to describe such "long-range" lexical rules.

Let us take the sentence

```
psa mi kupiłeś    (You) bought me a dog
```

Its version with a mobile verb suffix will have the form of

```
psaś mi kupił
```

Let us try to identify this phenomenon. We must first use an automaton that will serve us as a dictionary: - **DICTION**, and defined as:

```
DICTION := pies| psa | mi | kupiłeś | kupił
```

We now apply the operator `%DICTION% -> `<' __ `>'.`

Its role involves the recognition and marking of all known forms of words.

We obtain now

```
`<<'psa`>>'ś `<<'mi`>>' `<<'kupił`>>'
```

As we can see, "**psaś**" has not been recognized as a part of the dictionary.

If we use an additional lexicon containing only nouns

```
NOUN := psa | psu | kot | kotu
```

We can then search the text for nouns

```
%NOUN% -> [ ] __ `+' R
```

Which gives as an output

```
`<<'psa`+'R`>>'ś `<<'mi`>>' `<<'kupił`>>'
```

We look now for verbs in the third person of the past tense of the indicative mood:

```
VERB:= kupił | wziął
%VERB% -> [ ] __ `+' I3P
```

Which gives us

```
        `<‘psa`+’R`>’ś `<‘mi`>’ `<‘kupił+ I3P`>’.
```

We can now apply a constraint to work on only those texts where the phenomenon we are interested in has been identified. Formally speaking, this constraint is an identity relation on a certain set.

```
        ?* `+’ R `>>’ ś ` ‘ `<<‘ ?*
```

It gives us an empty output for an input with no identified phenomenon, and copies its input if such phenomenon occurs.
We place the suffix back in its rightful place **`+’ I3P -> eś** gives

```
        `<‘psa`+’R`>’ś `<‘mi`>’ `<‘kupiłeś`>’
```

We remove the auxiliary symbols:

```
                      `+’ R `>’ ? ` ‘`<‘ -> ` ‘ `<‘

                                 .o.

                            [`>’ | `<‘] -> ` ‘
```

Which gives us

```
      psa mi kupiłeś
```

As we can see, with a range of auxiliary symbols and constaints, we can apply this formalism to the description of phenomena that occur over a wide text span.

4. Conclusions

This formalism can be highly useful in the modelling of morphosyntactic phenomena of the language. Its simplicity and, what is more important, the ease with which it can be implemented in the form of a computer program make it an invaluable tool for a natural language scientist.

References

[1] Gross M.: *The Construction of Local Grammars.* In: Roche E., Schabes Y. (Eds), Computational Linguistics, Mitsubischi Electric Research Laboratory, 1996

[2] Kartunnen L.: *The Replace Operator*. Grenoble, Rank Xerox Research Centre, MLTT-017, 1995

[3] Koskenniemi K.: *Representations and Finite-State Components in Natural Language*. In: Roche E., Schabes Y. (Eds), Computational Linguistics, Mitsubischi Electric Research Laboratory, 1996

[4] Roche E. and Schabes Y.: *Introduction*. In: Roche E., Schabes Y. (Eds), Computational Linguistics, Mitsubischi Electric Research Laboratory, 1996