Searching for Morphological Structure with Regular Expressions

George Smith
Universität Potsdam

July 24, 2003

This paper explains how regular expressions can be used when searching for morphological structure in the TIGER treebank with the TIGERSearch query tool (Lezius, 2002). It is assumed that the reader has some basic familiarity with the syntax of the TIGER query language (König et al., 2003; König and Lezius, 2003) as well as with the encoding of grammatical structure in the TIGER treebank and can formulate simple queries. For introductory information on the encoding of linguistic information in the treebank, see Smith (2003)[1]. Regular expressions are a powerful tool and will prove to be extremely useful in the formulation of queries. The main advantage that we will see in them is that compact regular expressions can describe an infinite set of possible forms that would be impossible to enumerate. They will prove extremely useful when searching for Lexemes that share an affix or a set of affixes, or in searching for wordforms which share a common inflectional marker or set of markers.

Regular expressions are a form of notation for regular languages. They can be used to define sets of strings. Here, an introduction to the basic syntax of regular expressions will be given. This is followed by examples of the use of regular expressions which involve searching for particular classes of morphologically complex German word forms. The introduction is informal and is targeted at the linguist who does not yet have experience with using regular expressions. After reading the introduction, it should be possible to compose the types of regular expressions most commonly needed by language researchers. Furthermore, the reader should then be able to follow the technical documentation available elsewhere. A formal definition of the class of regular languages and a comparison

[1]Some of the queries in this paper assume a version of the treebank with word-level annotation of morphological information and lemmatisation such as the TIGER Sample Corpus. Version 1 of the treebank does not include this information, which we are currently annotating and which will be available with version 2.

of the properties of regular languages with other languages, such as, e.g., context free languages or context sensitive languages, can be found in Partee et al. (1987, p. 451–453, 464–473, 562–563). The syntax of regular expressions as used in the TIGER language is compatible with the syntax used in version 5 of the Perl programming language (Wall et al., 1996).

One possible use of regular expressions would be to describe an infinite set of word forms which share a common ending or a set of common endings. In this way, regular expressions can provide the functionality of rhyming dictionaries, which have long been a popular tool for linguists interested in suffixation. Examples of commonly used German rhyming dictionaries are Mater (1983) and Muthmann (1991). Regular expressions can also be used to describe a set of word forms which share a common beginning, making possible a search for prefixed words. The traditional tool for linguists interested in prefixes is a normal dictionary. Regular expressions can provide the functionality of a rhyming dictionary and a normal dictionary simultaneously. A regular expression describing all Lexemes which begin with the string `"Be"` and end with the string `"ung"` would, for example, be of interest to a linguist interested in the distribution of nominalizations of verbs containing the prefix *be*-. In contrast to the commonly used dictionaries, regular expressions are not restricted to word boundaries. It is, for example, a simple task to formulate a regular expression which can be used to retrieve all compounds in which a non-final element is a deverbal noun in which a verb with the suffix -*ieren* has been subject to nominalization with the suffix -*ung* and opened for conversion with the linking element -*s*-, such as *Antidiskriminierungspolitik* or *Zentralisierungstendenzen*. The linguist needs simply to understand the graphematic structure and the syntax of regular expressions.

In the TIGER language, regular expressions can be used as descriptions of features in feature-value pairs. Including a regular expression as a description for the feature `word` or the feature `lemma` in a feature constraint gives the linguist full control over the graphematical shape of the wordforms or the lemmas to be matched. Including a feature description of this type in a node description together with feature descriptions for word class and inflectional categories results in a powerful tool for researchers investigating areas of grammar where aspects of derivational or inflectional morphology play a role. Embedding such node descriptions in queries describing relations between nodes enables the formulation of sophisticated queries for investigating the interaction of syntax and morphology.

1 The Syntax of Regular Expressions

This section describes the syntax of regular expressions, introducing a set of notational conventions immediately useful to linguists. Standard notation for strings and sets will be used here. Strings will be enclosed in double quotes. The set consisting of the two strings `"a"` and `"aa"` will be written `{"a", "aa"}`.

Two different perspectives are commonly taken in discussions of regular expressions and strings. Both will be used here as appropriate. The first is a more formal perspective, in which it will be said that a regular expression is a notation for a grammar which defines a language consisting of a set of strings. This perspective will be familiar to linguists interested in phrase structure grammar formalisms. The second is a more pragmatic perspective common in contexts involving computer programming or database searching. According to this perspective it can be said that a string and a particular regular expression match. The regular expression is sometimes referred to as a pattern and reference is made to pattern matching. This second perspective will frequently prove more useful here: The most conceptually simple way to understand why a complex regular expression matches a particular string is often by breaking the expression down into simpler sub-expressions and understanding how these sub-expressions match particular substrings.

1.1 The Operator for Disjunction

This section describes basic pattern matching and introduces the operator for disjunction. The simplest possible regular expression consists of a single character and defines a language consisting of a single string which contains that character.

(1) a. `a`
  b. `b`
  c. `A`
  d. `B`

(2) a. {`"a"`}
  b. {`"b"`}
  c. {`"A"`}
  d. {`"B"`}

Each of the regular expressions in (1) defines a language consisting of a single string (2); upper and lower case characters are distinct. Regular expressions can be concatenated. When two regular expressions are concatenated (3a), the resulting expression defines a language consisting of all possible concatenations of a string in the first language and a string in the second language (4b). Longer regular expressions can be formed by concatenating shorter sub-expressions (3b-c).

(3) a. `ab`
b. `abc`
c. `Hund`

(4) a. {`"ab"`}
b. {`"abc"`}
c. {`"Hund"`}

From the perspective of pattern matching, we can think of a regular expression consisting simply of a sequence of characters as matching an equivalent string. Each of the regular expressions in (3) matches the equivalent string in (4).

(5) a. `a`
b. `b`
c. `a|b`

(6) a. `{"a"}`
b. `{"b"}`
c. `{"a", "b"}`

The operator for disjunction (`|`) is used to define a language consisting of all the strings in the language defined by the regular expression to the left of the operator and all the strings in the language defined by the regular expression to the right of the operator. While the regular expression in (5a) defines the language in (6a), and the regular expression in (5b) defines the language in (6b), the regular expression in (5c) defines the language (6c), the union of the languages (6a) and (6b). From the perspective of pattern matching, we can think of the expression (5c) matching a string consisting of either the character `a` or the character `b`.

(7) a. `(D|d)er`
b. `dies(e|er|es)`
c. `(V|v)o(n|m)`

(8) a. {`"Der", "der"`}
b. {`"diese", "dieser", "dieses"`}
c. {`"Von", "von", "Vom", "vom"`}

Some simple examples of regular expressions that could be used to search for wordforms of German are given in (7) together with the wordforms matched by them in (8). The expression in (7a) demonstrates the use of disjunction to search for wordforms irrespective of the capitalization of the initial letter. It can be broken down into the sub-expressions `(D|d)`, which matches the substrings `"D"` and `"d"`, and the sub-expression `er` which matches the substring `"er"`. Similarly, the expression in (7b) demonstrates the potential for searching for particular inflectional endings. It can be broken down into the sub-expression `dies` which matches the substring `"dies"` and the sub-expression`(e|er|es)` which matches any of the substrings `"e"`, `"er"` or `"es"`. The regular expression in (7c) can be used to search for the preposition *von* as well as it's fused form *vom*, irrespective of capitalization: the sub-expression `(V|v)` matches either the substring `"V"` or the substring `"v"`, the sub-expression `o` matches the substring `"o"`, and the sub-expression `(n|m)` matches either the substring `"n"` or the substring `"m"`.

In cases such as those in (7), it is of course also possible to list the forms to be searched for. Someone well-versed in regular expressions will be able to formulate the expression in (7c) more quickly than to type the list in (8c), someone just starting out with regular expressions may be able to type the list more quickly. In the next section, we will see how regular expressions can be used to define sets of strings which cannot be listed, simply because they are infinite.

1.2 Operators for Concatenation

This section introduces three operators for concatenation. From the point of view of regular expressions defining languages, these operators define languages consisting of strings which are concatenations of strings of the language to which the operator applies. From the point of view of pattern matching, these operators are often referred to as operators for repetition. The regular expression, or pattern, that they are applied to is seen as potentially matching repeatedly. The following example will illustrate what is meant here.

(9) a. `a*`
b. `a+`
c. `a?`

(10) a. {`""`, `"a"`, `"aa"`, `"aaa"`, `"aaaa"`, `...`}
b. {`"a"`, `"aa"`, `"aaa"`, `"aaaa"`, `...`}
c. {`""`, `"a"`}

The regular expression in (9a) uses the operator for unrestricted concatenation (`*`). The expression (9a) defines the language (10a), the set of all strings consisting of zero or more concatenations of the string `"a"`. The regular expression (9b) uses another operator for concatenation (`+`) which indicates one or more concatenations and defines the language (10b). The language in (10a) differs from the language in (10b) in that the former contains the empty string. The regular expression (9c) uses an operator for zero or one concatenations (`?`). The language defined contains two elements (10c).

From the perspective of pattern matching, the expression in (9a) can be thought of as matching zero or more instances of the substring `"a"`, the expression in (9b) can be thought of as matching one or more instances of the substring `"a"`, and the expression in (9c) can be thought of as matching zero or one instances of the substring `"a"`. We now turn to some more complex examples involving combinations of operators and the use of parentheses.

(11) a. `ab+`
     b. `(ab)+`

(12) a. {`"ab", "abb", "abbb", "abbbb", "abbbbb", ...`}
     b. {`"ab", "abab", "ababab", "abababab", ...`}

In the regular expression in (11a) the operator for one or more concatenations applies to the previous expression. The expression `ab` is composed of two sub-expressions `a` and `b`. The expression immediately preceding the operator is then `b`; the language defined is (12a). In order to apply the operator to the entire expression `ab`, parentheses must be used as in (11b), defining the language (12b).

From the perspective of pattern matching, the expression in (11a) matches one instance of the substring `"a"` followed by one or more instances of the substring `"b"`, and the expression in (11b) matches one or more instances of the substring `"ab"`.

(13) a. `a|b+`
     b. `(a|b)+`

(14) a. {`"a", "b", "bb", "bbb", "bbbb", "bbbbb" ...`}
     b. {`"a", "b", "aa", "bb", "ab", "aaa", "aab" ...`}

The expression in (13a) uses the operator for one or more concatenations and the operator for disjunction and defines the language in (14a), which contains all strings defined by the language `a` or the language `b+`. In the regular expression in (13b), the parentheses denote that operator for concatenation is applied to the entire regular expression `a|b`. The language defined by the expression is the set of all possible concatenations of strings defined by the language `a|b`, given in (14b).

From the perspective of pattern matching, the expression in (13a) matches either one instance of the substring `"a"`, or one or more instances of the substring `"b"`, and the expression in (13b) matches zero or more instances of either the substring `"a"` or `"b"`, in any order.

(15) a. `Löwen?`
b. `Mensch(en)?`

(16) a. {`"Löwe", "Löwen"`}
b. {`"Mensch", "Menschen"`}

In the expression in (15a), the operator for zero or one concatenations is applied to the sub-expression `n`, yielding an expression which matches the forms of the lexeme *Löwe* with and without the inflectional ending *-n* (16a). In (15b) we see a similar expression yielding the forms of the lexeme *Mensch* with and without the inflexional ending *-en*. In this case it is necessary to use parentheses so that the operator for concatenation applies to the subexpression `en`.

1.3 The Wildcard Operator

This section introduces the wildcard operator (`.`), which matches any character. In the context of searching for lexemes or wordforms, it is highly useful together with an operator for unrestricted concatenation for matching unspecified substrings.

(17) a. `.`
b. `.*`
c. `.+`

(18) a. {`"a", "b", "c", ..."A", "B", "C", ...`}
b. {`"", "a", "aa", ..., "ab", ..., "az", ...`}
c. {`"a", "aa", ..., "ab", ..., "az", ...`}

The expression in (17a) matches all strings consisting of one single character. The expression in (17b) matches all strings which are zero or more concatenations of any string in (18a). The expression in (17b) is highly useful as a sub-expression which matches any substring, including the empty string. The expression in (17c) is similarly useful and differs from (17b) only in that it does not match the empty string.

(19) a. `.m`
b. `(V|v)ver.*`
c. `.*ung`

(20) a. {`"am", "Am", "im", "Im", "um", "Um", ...`}
b. {`"Vera", "verabredet", "Verabredungen", ...`}
c. {`"Abbildung", "Abfallbeseitigung", ...`}

The expression in (19a) matches all strings consisting of any character followed by the character `m` (20a). The expression in (19b) defines the list of all strings beginning with `"ver"` or `"Ver"` (20b). The expression in (19c) defines the list of all strings ending with `"ung"` (20c).

The strings listed in (20) are wordforms of German which actually occur in the TIGER Treebank. The regular expressions do of course match other strings as well, inlcuding strings which in principal cannot be wordforms of German, such as `"verbtdg"` or `"sdtlung"`. The fact that these expressions overgenerate is not usually of concern to a linguist searching for data in a corpus. To the contrary, if the only concern is matching wordforms which actually occur in the corpus, the task of formulating queries is much simpler than it would be were the task to be using regular expressions to generate possible wordforms.

1.4 Lists of Characters and Character Ranges

This section describes notational conventions for specifing lists of characters and character ranges. These conventions have numerous uses. They provide more control than the wildcard operator discussed in section 1.3. They are useful in expressing graphotactic constraints, making it easier to formulate certain classes, such as graphemes for vowels, consonants, or subclasses thereof.

(21) a. `[ptk][ieaou][ptk]`
b. `[ˆAÄEIOÖUÜaäeioöuü][aäeioööuü][ˆaäeioöuü]`

(22) a. {`"tat", "tot", "tut", ...`}
b. {`"Bad", "bad", "Bar", "Bay", "Baß", ...`}

The regular expressions in (21) use a notational device for a list of characters to be matched along with a negational variant. The expression in (21a) can be broken down into three sub-expressions, each of which consists of characters in square brackets. The first (and last) sub-expression, `[ptk]`, matches a substring consisting of any of the single characters `p`, `t` or `k`. The second sub-expression, `[ieaou]`, matches a substring consisting of any of the single characters `i`, `e`, `a`, `o` or `u`. The expression in (21a) matches the strings in (22a). The expression in (21b) uses a notational convention for negated character lists, in which an accent circonflex is preposed to the list. The sub-expression `[ˆaäeioöuü]` matches substrings consisting of any single character except the characters `a`, `ä`, `e`, `i`, `o`, `ö`, `u` or `ü`. The regular expression (21b) matches the wordforms in (22b), which all conform to the graphematic pattern consonant, vowel, consonant.

(23) a. `[0-9]*`
b. `[A-Z]*`
c. `[a-z]*`

The queries in (23) illustrate a notational convention for ranges of characters. In general regular expressions of this type are interpreted in terms of the coding system used on a particular computer platform. In order to use character ranges, it is thus important to be familiar with the coding system used for a particular corpus, e.g., ANSI or UNICODE. The queries in (23) will be interpreted as follows in common encoding systems, including that used for the TIGER Treebank. The expression in (23a) matches all strings consisting of a sequence of digits. The expression (23b) will match strings consisting solely of upper case letters of the Latin alphabet without diacritics. The expression (23c) will match strings consisting solely of lower case letters of the Latin alphabet without diacritics.

(24) a. `[a-zA-Z]*`
b. `[äöüßa-z]`
c. `[ÄÖÜA-Z][ßäöüa-z]+`

It is possible to use more than one range of characters, as in the expression (24a) will match strings consisting of upper and lower case letters of the Latin alphabet without diacritics. It is possible to use both a list of characters and a range of characters within square brackets, in which case the list must come first, e.g., the expression in (24b), which will match all lower case characters native to German orthography. The expression (24c) will match strings consisting solely of characters native to German orthography which have an uppercase character in first position.

1.5 The Escape Character

In the previous sections we have seen the use of a number of operators. It is of course possible that one might want to search for wordforms containing a character equal to one of those operators. In such cases it is necessary to prepose a backslash \ to the character to be searched for. The backslash is said to be an escape character. The character to which it is preposed is said to be escaped.

(25) a. `[word=/.+\./]`
b. `[word=/.+\-.+/]`

For example, one might want to search for abbreviations ending with a period (25a). Or one might want to search for hyphenated wordforms (25b). In the context of using regular expressions in TIGERSearch, operators of the TIGER language must also be escaped.

2 Using Regular Expressions in Queries

This section will explore various possible uses to which regular expressions can be put when formulating queries in TIGERSearch. The regular expressions are used to restrict queries to wordforms or lexemes which potentially contain particular morphological or graphematical structures. When used in a node description in a TIGERSearch query, regular expressions must be enclosed in slashes, e.g., `[lemma=/.*ung/]`.

The first example shows how a concatenation of an initial string, a wildcard, and a final string can be used to search for strings which have a graphematic shape which is compatible with a class of words which share a common prefix and suffix.

(26) a. `Be.+ung`
b. `[pos = "NN" & lemma = /Be.+ung/]`
c. `"Berechnungen", "Befragung", "Begeisterung"`

The regular expression (26a) matches lemmas with a graphematic shape compatible with nominalized verbs containing the prefix *be-* and the suffix *ung*. This expression could be embedded in a node description such as (26b). The strings in (26c) are among those retrieved from the corpus by the query (26a).

(27) a. `[pos="NN" & word=/Berge?/]`
b. `[pos="NN" & word=/Berge?s?/]`
c. `[pos="NN" & word=/Berg(e|en)?/]`
d. `[pos="NN" & word=/Berg(e?s?|en)/]`
e. `[pos="NN" & word=/Berg(e?s?|en)?/]`

(28) a. `"Berg", "Berge"`
b. `"Berg", "Berge", "Bergs", "Berges"`
c. `"Berg", "Berge", "Bergen"`
d. `"Berge", "Bergs", "Berges", "Bergen"`
e. `"Berg", "Berge", "Bergs", "Berges", "Bergen"`

In (27), the operator for zero or one concatenations (?) is used in regular expressions in TIGERSearch queries which could be used to access nodes containing forms of the noun *Berg*. The query in (27a) would access nodes containing the strings in (28a), the operator for zero or one concatenation applying to `e`. The query in (27b) contains a regular expression with two operators for concatenation, one applying to `e` and the other to `s`, yielding the strings in (28b). In (27c), the operator applies to the sub-expression in parentheses: `e|en`. The query in (27d) yields only inflected forms. The query in (27e) yields the full paradigm (28e).

(29) a. `[pos = "ADJA" & word = /[.+en/]`
b. `[pos = "ADJA" & word = /[.+em/]`
c. `[pos = "ADJA" & word = /[.+(en|em)/]`

The queries in (29) find forms of adjectives containing a nasal inflectional suffix: those ending with the substring `"en"` (29a), those ending with the substring `"em"` (29b), and those ending with either the substring `"en"` or `"em"` (29c).

(30) a. `[pos="NN" & morph=/.*Dat.Pl/ & word=/.*[ˆn]/]`
b.
```
[pos="NN" & morph=/(Masc|Neut).Gen.Sg
          & word=/.*[ˆs]/]
```

The regular expression in (30a) can be used to access common nouns (`NN`) in dative plural that do not end in the substring `"n"`. It uses the notational convention for negated character lists in the subquery `[ˆn]`. In a similar way, the query (30b) can be used to access masculine or neutral common nouns in genitive singular that do not end in the substring `s`.

(31) a. `[lemma=/Ver.+ung/]`
b. `[pos="NN" & lemma=/Ver.+ung/]`

Certain types of regular expressions, such as those containing a combination of a wildcard (`.`) and an operator for unlimited concatenation (`+` or `*`), are computationally expensive. Often it is possible to construct a query in TIGERSearch that limits the number of words that need to be checked for a match. If, for example, one is interested in deverbal nouns containing the prefix *ver-* and the suffix *-ung*, then restricting the node description to nouns will result in significant time savings, as only nouns need to be checked for matches. The query (31b) is approximately five times faster than the query (31a) in the version of TIGERSearch available at the time of writing. In queries making heavy use of regular expressions this degree of time saving can be substantial.

The use of regular expressions in feature descriptions adds significant power to TIGERSearch queries. It is important to remember that regular expressions in queries cannot access morphological structure of words, but rather can be used to impose conditions on graphematical shape. Together with word form class and morphological specifications, they can significantly narrow a search, making it easier to find desired structures.

References

König, Esther, and Wolfgang Lezius. 2003. The TIGER language - a description language for syntax graphs, Formal definition. Tech. rep., IMS, University of Stuttgart.

König, Esther, Wolfgang Lezius, and Holger Voormann. 2003. *TIGERSearch User's Manual*. IMS, University of Stuttgart, Stuttgart.

Lezius, Wolfgang. 2002. Ein suchwerkzeug für syntaktisch annotierte textkorpora. Ph.D. thesis, IMS, University of Stuttgart. Arbeitspapiere des Instituts ür Maschinelle Sprachverarbeitung (AIMS), Volume 8, Number 4.

Mater, E. 1983. *Rückläufiges Wörterbuch der deutschen Gegenwartssprache*. Oberursel, 4th ed.

Muthmann, Gustav. 1991. *Rückläufiges deutsches Wörterbuch*. Reihe Germanistische Linguistik 78. Tübingen: Niemeyer, 2nd ed.

Partee, Barbara H., Alice ter Meulen, and Robert E. Wall. 1987. *Mathematical Methods in Linguistics*. Studies in Linguistics and Philosophy 39. Dordrecht: Kluwer Academic Publishers.

Smith, George. 2003. A brief introduction to the TIGER treebank, version 1. Tiger Projektbericht, Univ. Potsdam.

Wall, Larry, Tom Christiansen, Randal L. Schwartz, and Stephen Potter. 1996. *Programming Perl*. Cambridge: O'Reilly.