TGrep2 User Manual

version 1.15

Douglas L. T. Rohde

May 10, 2005

1 Introduction

Statistical analysis of large, syntactically tagged corpora is playing an increasingly important role in language research. In particular, the Penn Treebank (Marcus, Santorini, & Marcinkiewicz, 1993; Marcus et al., 1994) version of the Wall Street Journal and Brown corpora of English text is a frequently studied resource. Until now, the software tool of choice for analyzing such corpora has been the TGrep program, developed by Richard Pito.

The primary function of TGrep is to extract parse trees whose structures match a specified pattern. It is, essentially, *grep* for trees. However, working with TGrep and talking to others who use it in their research resulted in a wish list of improvements. Eventually, I undertook to rewrite TGrep in its entirety. The result is TGrep2.[1] TGrep2 is almost completely backward compatible with TGrep, but introduces a number of new features, including the following major enhancements:

- Rather than simply having a set of required relationships and a set of prohibited relationships, nodes can have full boolean expressions of relationships to other nodes.
- Nodes can be given unique labels and may then be referred to by those labels in the pattern specification or in selecting trees for printing.
- Patterns are no longer restricted to simple tree architectures. The use of node labels and segmented patterns allows links in a pattern to form back-edges as well, permitting cycles of links.
- Customizable output formats allow a variety of information to be reported in a flexible manner.
- Multiple search patterns may be specified and one can retrieve the first subtree matching any pattern, the first subtree matching each pattern, or all subtrees matching any patterns, or all matches between subtrees and patterns.
- Subtrees can be reported using a code rather than by printing the whole structure. The trees themselves can later be retrieved using the codes.
- A variety of new links have been added and the immediately-precedes link now has a more conventional meaning.
- Macros can be defined and used to simplify pattern specification.
- TGrep2 corpus files are substantially smaller than TGrep corpora.

2 Preparing Corpora

Before using TGrep2, a special corpus file must be created in a format that is a bit different from that of the corpus files used by TGrep. Corpus files are generated using the **-p** option to TGrep2. Following the **-p** are two arguments, one giving the name of a text input file containing the corpus trees and one giving the name

[1]The design and implementation of TGrep2 is Copyright (C) 2001-2005 Douglas Rohde. Comments, questions, or bug reports should be sent to dr+tg@tedlab.mit.edu. TGrep2 is distributed under the conditions of the GNU General Public License, version 2.

of the binary output file to which the corpus will be written. TGrep2 corpus files normally end in the ".t2c" extension, but that is not required.

The input file contains sentences in parenthesized tree format.[2] Each tree must begin with an open parenthesis. Here is an example of an input file with three very short sentences:

```
(TOP (NP (NP (NN Research)) (CC and) (NP (NN development))))
(TOP (NP (NP (NN Budget)) (VP (VBD increased))))
# This is a comment.
(TOP (NP (JJ Stretch) (NN yarn) (NNS machines)))
```

There is currently a limit of 255 children per node and 65,535 nodes per sentence due to the format in which corpus files are stored. The limit on children per node can be raised to 65,535 by using the **-K** flag when building the corpus.

Sentences parsed in Combinatory Categorial Grammar (CCG) style will have a somewhat different format. When building corpora in this style, the **-g** flag should precede **-p**. See Section 6.1 for more details.

Comments can appear in the input file if they are on a line starting with a `#`. By default, comments in the input file are not stored in the corpus file. But if the **-C** option is specified prior to **-p**, the comment immediately preceding each sentence will be associated with the sentence and recorded in the corpus. The comments can later be printed when a match is made during a search.

If either the input or output files have the .gz, .bz, .bz2, or .Z extension, they will automatically be decompressed or compressed, respectively. If the input file is compressed, these extensions need not be specified. To write to standard output, "**–**" can be given in place of the output file. However, the input file cannot be read from a pipe or standard input. The input must come from a static file.

Here is one procedure for converting a TGrep corpus file to a TGrep2 corpus file:

```
% export TGREP_CORPUS=wsj_mrg.crp
% tgrep -n __ | grep . | gzip > wsj_mrg.txt.gz
% tgrep2 -C -p wsj_mrg.txt wsj_mrg.t2c.gz
```

To combine all of the Brown corpus .mrg files in the Treebank 3 distribution into a TGrep2 corpus file, you can do the following. Notice that the label TOP will be inserted at the head of each main sentence tree:

```
% cd parsed/mrg/brown
% cat */*.mrg | grep -v '^\*' | sed 's/^( /(TOP /' > brown.txt
% tgrep2 -p brown.txt brown.t2c
```

On most machines, it is not much slower (and is sometimes faster) to use a gzipped compressed corpus file rather than an uncompressed one because of the tradeoff between the decompression time and the reduced disk access. When uncompressed, TGrep2 corpus files are about 20% the size of TGrep corpus files. And when compressed, they are under 5% the size of an uncompressed TGrep corpus. Bunzip2 (.bz2) will achieve about twice as good compression as gunzip on corpus files, but the decompression may take over twice as long during querying.

3 Command-line Arguments

TGrep2 is used as follows:

[2]Although the ability to search for tree structures may be helpful in many other domains, TGrep2 is most often used for processing corpora of parsed sentences. Therefore, the terms *tree* and *sentence* are used interchangeably throughout this manual, since each sentence is associated with a single parse tree.

```
% tgrep2 [options] <pattern>...
```

Following the options, which are listed below, should be one or more pattern specifications. A pattern specification is either a pattern itself or the name of a file containing patterns. If there is a readable file matching the argument, the patterns will be taken from that file. Otherwise, the argument will be treated as a pattern. If more than one pattern or pattern file is given, they are effectively concatenated.

There are two exceptions to the usual use of command-line arguments. If the **-p** option is used to construct a corpus, the final pattern is optional. If the **-e** option is used, the last argument must be the name of a file containing subtree codes. In this case, only one file may be specified.

See Section 4 for more about specifying search patterns.

3.1 Options

-c <corpus-file> This sets the corpus file used by TGrep2. If no corpus is specified on the command line, it must be given by the **TGREP2_CORPUS** environment variable.

-a By default, only the first subtree matching each pattern is reported. By itself, this option reports all matches between subtrees and patterns. In this case, a subtree will be reported once for each pattern it matches, so there may be duplication. In conjunction with the **-f** option, all subtrees matching one or more patterns will be reported, but each subtree will only be reported once. Note that a given pattern might match a given subtree in more than one way, especially if the pattern is vague. When this happens, only one alignment between pattern and subtree is reported, even with the **-a** option.

-f By default, the first subtree matching *each* pattern is reported. The **-f** option by itself reports only the first subtree matching any pattern. Thus, there will be at most one match per sentence, regardless of the number of patterns. In conjunction with the **-a** option, all subtrees matching one or more patterns will be reported, but each subtree will only be reported once. This is the most common setting for counting structures in corpora because it reports each subtree that matches one or more patterns, but without duplication.

-v Normally, TGrep2 reports subtrees that do match the pattern(s). This option invokes "filter mode," which causes trees to be reported which *do not* match any patterns. If the output format is specified with **-m**, it will behave as if a match occurred on the top node in the tree.

-i By default, TGrep2 is case-sensitive when matching node names. The **-i** flag causes it to ignore case. This can be especially useful when working with English sentences, where sentence-initial capitalization could cause problems.

-b <format> This specifies the output format that determines what is printed at the start of every sentence in the corpus, whether or not there is a match. See Section 5.2 for details on formatted output.

-s <format> This specifies the output format that determines what is printed before the first match on sentences for which there is at least one match. If the **-v** option is used, this defines the format used in printing sentences with no matches. If the **-f** option is used, this is essentially redundant with **-m**. See Section 5.2 for details on formatted output.

-m <**format**> This specifies the output format that determines what is printed every time there is a match between a pattern and a sentence subtree. If this is not defined, then matching subtrees are printed one-per-line in short (parenthesized) form, unless either **-l**, **-t**, **-u**, or **-x** is given. See Section 5.2 for details on formatted output.

-l If **-m** is not specified, this causes matching subtrees to be printed in long format (indented with one symbol per line).

-t If **-m** is not specified, this causes only the terminals of matching subtrees to be printed. Thus, the trees are printed like a sentence, with no structural information.

-u If **-m** is not specified, this causes only the name of the top symbol of each matching subtree to be printed. This will print the name of a non-terminal without printing its children.

-x If **-m** is not specified, this prints only the subtree code for matching subtrees. This is a unique identifier for the subtree and is in the form s:n, where s is an integer specifying the sentence number in the corpus (starting with 1), and n is an integer giving the order in which the node is encountered in a depth-first search starting with 1 at top node in the sentence tree.

-w If **-m** is not specified, the whole tree is printed on each match, rather than the matching subtree. The format in which the tree is printed is affected by the use of the **-l** and **-x** flags.

-d Ordinarily, TGrep2 reorders the links in the search pattern for greater efficiency. This can alter the order in which marked nodes are printed and the order in which links in a disjunction are searched. If that is a problem, the **-d** option will suppress the reordering of links.

-r <**seconds**> If a positive integer is specified, it determines how often, in seconds, a progress report is printed. Progress reports consist of the completion percentage and are written to the standard error output. By default, there are no progress reports.

-e This option causes TGrep2 to operate in a special "extraction mode". The final command-line argument should be the name of a file containing subtree codes, rather than a pattern. A subtree code is in the form s:n, where s is an integer specifying the sentence number in the corpus (starting with 1), and n is an integer giving the order in which the node is encountered in a depth-first search starting with 1 at top node in the sentence tree.

Each subtree whose code is given will be reported. For the purposes of formatted output, it behaves as if the head node in the pattern matched the indicated subtree. The subtree codes must be ordered so that the sentence numbers are non-decreasing.

-p <**text-file**> <**corpus-file**> This is used to generate a TGrep2 corpus file. See Section 2 for details on corpus files. When using this option, if no search pattern is given, TGrep2 will simply exit. But one can also specify a pattern to be searched once the corpus is built. This makes it possible to generate a corpus and search the corpus in a single step. To facilitate this, the active corpus is set to the one newly built. This can be overridden with a subsequent use of **-c**.

-z	This causes TGrep2 to parse the patterns, possibly reorder the links for improved speed, and pretty-print the patterns back to standard output. This is helpful for debugging complex patterns.
-C	If this precedes **-p**, comments in the input file will be stored in the corpus. Multiple lines starting with # will be considered one comment if they are consecutive, but separate comments if there is a blank line between them. Only the comment immediately preceding a sentence is stored. This option can also be used to print the comments while searching a corpus. If you are not using formatted output (not using the **-m** flag), any comment associated with a sentence will be printed just before the first match made to that sentence.
-K	Ordinarily there is a limit of 255 children per parse tree node. That is because the number of children is stored using an 8-bit char in the corpus file. If you have nodes with more than 255 children, you can use the **-K** flag prior to the **-p** flag to increase the limit to 65,535. This causes the number of children to be stored using 16-bit words, so the corpus file will be a bit larger.
-h	Prints a summary of the command-line arguments, output formatting fields, and link codes.
-g	Causes the program to operate in Combinatory Categorial Grammar (CCG) mode. This is normally only used when preparing corpora (prior to **-p**). See Section 6.1 for more details.

4 Specifying Patterns

Unless the **-e** or **-p** options are used, the last argument to TGrep2 must be either a pattern or the name of a pattern file. If the argument happens to be the name of a readable file, the file will be used. Otherwise, it will be treated as the pattern itself.

It is generally a good idea to store all patterns in files, rather than always using the command line. Complex patterns can be too long and unwieldy for the command line. Also, a pattern used now will eventually be needed again and it is good to have a record of it.

The format for patterns is the same whether they are in a file or on the command line. Line breaks in the file are simply treated as white space. However, any line starting with a hash mark (`#`) is considered a comment and ignored. If multiple patterns are specified, it is not sufficient to put them on different lines. They must be separated with semicolons (`;`), as discussed in Section 4.8.

4.1 Basic Pattern Syntax

TGrep patterns ought to work just fine with TGrep2, but TGrep2 patterns can be significantly more expressive.

TGrep2 patterns mainly consist of node names and relationships, which define *links* to other nodes. A simple node consists of just a node name, which is a string, regular expression, or OR'd combination of the two that is matched against the names of nodes in the sentence tree. A complex node consists of a node name followed by a set of relationships, all surrounded by parentheses.

The first node in the pattern is called the *head* node. It is not necessary to enclose the head node and its relationships in parentheses.

As in TGrep, a set of relationships can simply be a list of links or negated links to other nodes, all of which must be satisfied for the node to match a subtree. However, TGrep2 also allows Boolean expressions of relationships, as explained in Section 4.5.

Nodes can also be assigned labels and may be referred to elsewhere in the pattern by those labels (see Section 4.6).

Finally patterns can be broken into multiple segments (see Section 4.7) and multiple patterns may be specified (see Section 4.8).

Whenever a new pattern is written, it is a good idea to check the pattern by running TGrep2 with the **-z** option. This parses the pattern and reprints it in standard format before exiting and is helpful in diagnosing problems that do not result in syntax errors. Note that when a pattern is parsed, segments are removed, crossing-links are replaced by copies of the nodes to which they point, and the links are reordered for improved efficiency. The **-d** option can be used to prevent link reordering.

By and large, spaces are optional in patterns. But for readability, it is usually best to place spaces between link codes, node names, and other operators.

Example Patterns

This matches any `NP` node that immediately dominates a `PP`:

```
NP < PP
```

This is a regular expression that matches any node whose name starts with `NP`, including `NP-SBJ`:

```
/^NP/
```

This matches an `NP` that dominates a `PP` *and* is immediately followed by a `VP`:

```
NP << PP . VP
```

This matches an `NP` that dominates a `PP` *or* is immediately followed by a `VP`:

```
NP << PP | . VP
```

This matches an `NP` that does not dominate a `PP`. Also, the `NP` must either have a parent that is an `NP` or be dominated by a `VP`:

```
NP !<< PP [> NP | >> VP]
```

This matches an `NP` that dominates a `PP` which itself is immediately followed by a `VP`. Note the use of parentheses to group "`. VP`" with the `PP` rather than with the `NP`:

```
NP << (PP . VP)
```

This matches an `NP` whose last child is a `PP` that begins with the preposition `on`:

```
NP <‘ (PP <, (IN < on))
```

The following pattern matches an `S` which has a child `A` and another child that is a `C` and that the `A` has a child `B`:

```
S < (A < B) < C
```

However, this pattern means that `S` has child `A` and that `A` has children `B` and `C`:

```
S < ((A < B) < C)
```

It is equivalent to this:

```
S < (A < B < C)
```

The reason it might be useful to allow the extra set of parentheses in the first version is that the sub-pattern `(A < B)` may have been defined as a macro. Macros are discussed further in Section 4.9.

Labels allow the creation of patterns with cycles in the link structure, which is often quite useful. For each `NP` that is followed by a `VP`, the following pattern prints the lowest node dominating both the `NP` and the `VP` (when used with the **-a** option). Note that the `NP` is given the label `n` and is later referred to by that label. `*` matches any node and is thus equivalent to `__`. The `'` marks that node for printing:

```
NP=n .. (VP=v >> ('* << =n))
```

The following pattern is a bit different. It matches any node that dominates an `NP` and a `VP`, such that the `NP` is followed by a `VP` and the `NP` and `VP` are not dominated by a node lower than the original one. This differs from the previous pattern in that the prior one may print the same subtree more that once.

```
*=p << (NP=n .. (VP=v >> =p !>> (* << =n >> =p)))
```

As one can see, patterns can become rather complex when using labels. However, judicious use of labels and Boolean expressions can greatly simplify searches. This is discussed more in Sections 4.5 and 4.6.

4.2 Node Names

A node name is an expression that matches the name of a single node in the tree. Node names are similar, but not identical, in TGrep and TGrep2. A node name can be a simple constant string (e.g., `NP-SBJ`), a regular expression (e.g., `/^NP/`), or any number of these things separated by pipes (`|`) with no spaces (e.g., `S|/^SBAR/`).

A constant string that is not enclosed in double quotes cannot contain white space or any of the following special characters: `;:.,&|<>()[]$!@%'^=`

If a constant string is enclosed in double quotes, it can contain any symbols (e.g., `"*$#&%!"`). If it must contain a double quote, the double quote should be preceded by a backslash.

Regular expressions must be enclosed in forward slashes. See the *grep* manual page for an explanation of regular expression syntax. As with TGrep, the pattern `__` matches any node. In TGrep2, the more conventional `*` is preferred.

If a pattern contains several components separated by `|`s, the pattern will match a node name if any of its components match. Thus, the `|` should be read as "or".

The following pattern matches names like Robert, Bob, bob, bobby, and Bobafett:

```
Robert|/^[Bb]ob/
```

If TGrep2 is given the **-i** flag, case will be ignored when matching either constant or regular expression patterns. This is especially useful when working with English sentences, where sentence-initial capitalization could cause problems.

If the node name expression begins with a `!`, the matching process will be complemented. That is, matches will turn into non-matches, and vice-versa. For example, `!DT` will match all symbols whose names are *not* DT, and `!/^a/|/a$/` will match any symbols that do not start with "a" and do not end with "a". The `!` must be the first thing in the node name, prior to any `/` or `"` (but after `'`, which is used to mark the node for printing, as discussed in Section 5.1).

4.3 Basic Links

Relationships define connections between the node being defined and other nodes. A relationship consists of a link followed by a node. The links used in TGrep have been adopted and some new links have been added so there is a complete pairing of forward and backward links.

`A < B`	A is the parent of (immediately dominates) B.
`A > B`	A is the child of B.
`A <N B`	B is the *N*th child of A (the first child is <1).
`A >N B`	A is the *N*th child of B (the first child is >1).
`A <, B`	Synonymous with `A <1 B`.
`A >, B`	Synonymous with `A >1 B`.
`A <-N B`	B is the *N*th-to-last child of A (the last child is <-1).
`A >-N B`	A is the *N*th-to-last child of B (the last child is >-1).
`A <- B`	B is the last child of A (synonymous with `A <-1 B`).
`A >- B`	A is the last child of B (synonymous with `A >-1 B`).
`A <‘ B`	B is the last child of A (also synonymous with `A <-1 B`).
`A >‘ B`	A is the last child of B (also synonymous with `A >-1 B`).
`A <: B`	B is the only child of A
`A >: B`	A is the only child of B
`A << B`	A dominates B (A is an ancestor of B).
`A >> B`	A is dominated by B (A is a descendant of B).
`A <<, B`	B is a left-most descendant of A.
`A >>, B`	A is a left-most descendant of B.
`A <<‘ B`	B is a right-most descendant of A.
`A >>‘ B`	A is a right-most descendant of B.
`A <<: B`	There is a single path of descent from A and B is on it.
`A >>: B`	There is a single path of descent from B and A is on it.
`A . B`	A immediately precedes B.
`A , B`	A immediately follows B.
`A .. B`	A precedes B.
`A ,, B`	A follows B.
`A $ B`	A is a sister of B (and A $\neq$ B).
`A $. B`	A is a sister of and immediately precedes B.
`A $, B`	A is a sister of and immediately follows B.
`A $.. B`	A is a sister of and precedes B.
`A $,, B`	A is a sister of and follows B.
`A = B`	A is the same node as B (see also Section 4.4, Link Modifiers).

`A ~ B`	`A` has the same name as `B`.

Note that the immediate precedence relationship in TGrep2 works as one might expect, rather than the odd way it was defined in TGrep. Tree node `A` immediately precedes node `B` if the last terminal symbol (word) produced by `A` immediately precedes the first terminal symbol produced by `B`. In TGrep, `A` only immediately preceded `B` if `B` was the next node in depth-first search order after `A` that was not a descendant of `A`, which is more restrictive.

The order in which the search is conducted for a link always starts close to the node on the left, `A`, and works away. For example, `A >> B` will try to find the lowest `B` that is an ancestor of `A`, starting with `A`'s parent and working up towards the root of the tree. Similarly, `A .. B` will find the next `B`, in depth-first search order, following `A`.

Note that = matches identical nodes (a node and itself), while ~ matches nodes that share the same name. The = link can be used to force a node's name to match multiple regular expressions or to compose patterns from several pieces. It is also commonly used as a back link (see Section 4.6) to ensure that two nodes are identical. The ~ operator can be used as a forward link to match any two nodes (possibly identical) that have the same name, although it is rather inefficient in this case. It is more often used as a back link to ensure that two nodes have the same name.

4.4 Link Modifiers

An exclamation mark (`!`) can be placed immediately before any link to negate it. Thus, `A !.. B` means that `A` is not followed by `B`. This use of negation is actually a special case of the more general Boolean expressions described in the next section.

Any link can be immediately followed by an equal sign to allow for the possibility of a match occurring on the first node itself. For example, `A <<= B` means that `B` is either dominated by `A` or `B` is equal to `A`. This is, of course, only useful if the patterns `A` and `B` could both apply to the same node. If = is used, the first potential match checked during the search will be if `A` itself matches `B`.

If a link is preceded by a question mark (`?`) it is considered optional. If the link fails to match, it will not cause the pattern to fail. However, the node on the right side of the link will be matched to a subtree if possible. This is only useful when that node or another node that it links to is labeled or marked for printing. The following pattern prints all singular nouns and also prints the immediately preceding adjective, if it exists. If no adjective precedes the noun, `<none>` will be printed:

```
'NN ?, 'JJ
```

Although it is not always required, it is a good idea to put a space before and after a link when it is given in a pattern. This will prevent parsing ambiguities, such as those created by `'` and =, which may appear at the end of the link or the start of the node name following the link, with different meaning.

In order to be compatible with TGrep, several alternative characters are permitted as replacements in specifying links, but their use is discouraged:

- `!` can be replaced by `@` (not recommended)
- < can be replaced by { or by ˆ
- > can be replaced by }
- `$` can be replaced by `%`

4.5 Boolean Expressions

One of the major additions in TGrep2 is the ability to specify Boolean expressions of relationships. Formerly, for each node in the pattern, one could only specify a set of relationships that must exist and a set of relationships that could not exist (by using the `!` operator). What was lacking, essentially, was disjunction.

In TGrep2, as in TGrep, if no operator is placed between two relationships, they are assumed to be connected by an *and*. In TGrep2 an ampersand (`&`) can optionally be placed between them for clarity. The following patterns both mean that `A` has child `B` *and* is immediately followed by `C`:

```
A < B . C
A < B & . C
```

If a pipe (`|`) is placed between two relationships, only one of them must be satisfied. The possible relationships are tested in order. If one matches, the remaining ones are ignored. This means that node `A` has child `B` *or* is immediately followed by `C`:

```
A < B | . C
```

The *and* binds tighter than the *or*. So this expression evaluates to (`A` has child `B` *and* is immediately followed by `C`) *or* (`A` has child `D` *and* is immediately followed by `E`):

```
A < B . C | < D . E
```

In order to create more complex expressions, square brackets (`[]`) can be used to group terms. Thus, (`A` has child `B` *or* is immediately followed by `C`) *and* (`A` has child `D` *or* is immediately followed by `E`), can be written:

```
A [< B | . C] [< D | . E]
```

The bracketed expression itself acts, syntactically, as a relationship. Note that an open bracket never follows a link but always precedes a link or another open bracket.

Any relationship, including a bracketed one, can be negated by immediately preceding it with an exclamation mark (`!`). Thus, this expression means (`A` has child `B` *or* it does *not* (immediately precede `C` and *not* immediately follow `F`)) *or* (`A` does *not* (have child `D` *and* is *not* followed by `E`)):

```
A [< B | ![. C !, F]] | ![< D !.. E]
```

4.6 Labeled Nodes

Another major addition of TGrep2 is that a node can be given a unique label and then referred to elsewhere by that label. To explain why that is useful, we will need a bit of notation.

It is helpful to think of a TGrep2 pattern as a graph. Nodes are vertices and there is an edge from node `A` to node `B` if there is any link from `A` to `B` in the pattern. Thus, the pattern:

```
A .. (B !< C . D) | ![<< (E , F) $ G]
```

has the graph structure shown in Figure 1.

Without the ability to label nodes and refer to them by label, it is not possible to express patterns whose links do not have a regular tree structure. Using labels permits the construction of patterns with more complex structures, which can make patterns more concise and allow one to perform some queries that are not possible otherwise.

For example, imagine that we wanted to find a sentence node, `S`, that dominates both a `VP` and a `PP` such that the `VP` precedes the `PP`. This cannot be expressed in a tree of links because there is a relationship between the `S` and the `VP`, between the `S` and the `PP`, and between the `VP` and the `PP`. However, it can be written in TGrep2 as follows:

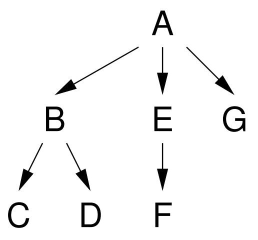

Figure 1: The link structure of pattern "`A .. (B !< C . D) | ![<< (E , F) $ G]`"

```
S=foo << (VP .. (PP >> =foo))
```

`S=foo` matches any tree node whose name is `S`. Furthermore, when a matching tree node is found, it is given the label `foo`. Later, "`PP >> =foo`" indicates that the `PP` must be dominated by that very same node, not just any `S`. The relational structure of this pattern is shown in Figure 2.

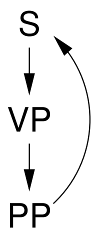

Figure 2: The link structure of pattern "`S=foo << (VP .. (PP >> =foo))`"

If a node name is followed by an equal sign (=) and a label, the label will be assigned to the node. If the node name only consists of an equal sign and a label, it refers to the node that has been assigned to that name elsewhere in the pattern. Every label must be assigned to exactly one node. Labels may not contain any of the special characters that are prohibited in node names unprotected by double quotes. You cannot use the expression `!=foo` to match any node but the one labelled `foo`.

As an example of the use of ∼ in a back link, this pattern will match any noun that is repeated within a Treebank sentence:

```
*=a >> /^NN/ .. (*=b >> /^NN/ ~ =a)
```

The way that a node label is used is actually pretty complex and can be confusing. If node `A` has a link that refers to the node labeled `foo` and the label `foo` is assigned to node `B`, the meaning of that link depends on the relationship between `A` and `B`.

If `B` is an ancestor of `A` in the pattern tree (as in the previous example), the link forms a *back link*. In this case, the label refers to the same tree node matched by `B`. Thus, the same node is the ancestor of both the `VP` and the `PP`.

However, it is possible to express patterns in which the labeled node `B` may not be an ancestor of `A`. Consider this example:

```
NP < (PP=pp < (IN < on)) | < (NP < =pp)
```

In this case, the `NP` has a link to the node labeled `pp`, but the node labeled `pp` is not an ancestor of the `NP`. If we were to draw the link structure of this pattern, it would look like Figure 3.

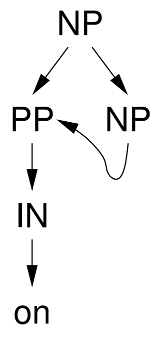

Figure 3: The initial structure of pattern "`NP < (PP=pp < (IN < on)) | < (NP < =pp)`"

The strange link in Figure 3 is a *crossing link*. A crossing link extends from a node to a subtree on its left. The problem with crossing links is that they can make matching a pattern to a tree an extremely slow and complicated computational process. Therefore, TGrep2 does not permit crossing links. If a link to a labeled node forms a crossing link, then a copy is made of the whole subtree in the pattern pointed to by the link. The result is shown in Figure 4. Subtrees to which a crossing link is extended may not contain back links that extend above the node at the top of the subtree.

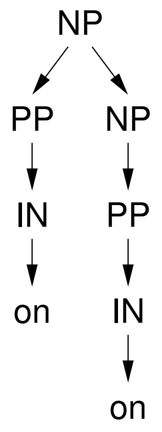

Figure 4: The final structure of pattern "`NP < (PP=pp < (IN < on)) | < (NP < =pp)`"

In this case, the use of a node label didn't enable us to do something we couldn't have done without a label. It simply made the pattern more concise. Note that if a node is copied and has been assigned a label, the new node will have the same label with one or more pluses (+) appended to the end to create a unique label. Use the **-z** option to print the pattern as it was interpreted by TGrep2 in order to check if any crossing links were resolved by copying.

A final convenience of node labels is that they can be referred to when producing formatted output. A subtree can be printed by specifying the label of the node matching that subtree. See Section 5.2 for more about formatted output.

4.7 Segmented Patterns

For clarity's sake, it is not always convenient to write a complex pattern in fully parenthesized form. It is easy to get lost in a sea of nested parentheses. Therefore, TGrep2 provides a way to split patterns into multiple segments, which are separated by colons (`:`). Each segment following the first must begin with a reference to a labeled node that is defined in a previous segment.

In fact, if all complex nodes are labeled, segments can entirely replace parentheses. Consider this example, which expresses the same pattern with and without segments:

```
S < (NP=n1 .. (VP=v < PP)) < (NP=n2 !.. VP)
S < NP=n1 < NP=n2 : =n1 .. VP=v : =v < PP : =n2 !.. VP
```

The segments do not necessarily make the pattern shorter, but they can make it easier to read and debug. If patterns are stored in files, it may be convenient to place each segment on its own line.

If links are specified from a labeled node in more than one segment, they are assumed to be and'd together, meaning they all must apply. For example, these two patterns are equivalent:

```
S << (VP=v < NP) : =v < /^PP/
S << (VP=v < NP < /^PP/)
```

It is frequently the case that one compares several patterns that are mostly the same but differ in only one region. If the region that changes is placed in its own segment, then it is easy to compose the full patterns by appending a unique segment to a common base. If a change is made in the base, it will propagate to all of the patterns built from that base.

4.8 Multiple Patterns

It is sometimes necessary to compare or combine the results of multiple search patterns. With TGrep, one must run the program separately for each pattern. However, TGrep2 provides a mechanism for specifying an arbitrary number of search patterns in a single invocation.

Multiple patterns are separated by semicolons (`;`). The patterns are independent of one another in that node labels used in one pattern can only refer to nodes in that pattern and are disjoint from labels used in other patterns. In this example, the first pattern selects `NP`s that immediately dominate a `JJ` and the second pattern selects `NP`s that immediately dominate a `DT`:

```
NP < JJ; NP < DT
```

Consider the following two-sentence corpus:

```
(S (NP (DT the) (JJ big) (NN dog)) (VP bit) (NP (DT a) (NN cat)))
(S (NP (DT the) (NNS dog)) (VP has) (NP (DT a) (JJ new) (NNS trick)))
```

TGrep2 searches the sentence tree and tries to match each pattern to each subtree. When using formatted output, one has the option of printing the number of the pattern resulting in the match (**%p**) and the number of times that pattern has resulted in a match on the current sentence (**%j**), as well as the total number of matches on the sentence (**%i**). This makes it possible to distinguish subtrees matched by the various patterns in the output.

The default behavior is to report the first subtree match for each pattern. If there are N patterns, there can be up to N matches. In the first sentence of the example, "the big dog" matches both patterns and would therefore be reported twice, once for each. In the second sentence, "the dog" would be reported because it is the first match for the second pattern, and then "a new trick" would be reported because it is the first match for the first pattern. Note that it is not reported twice even though it also matches the second pattern because that pattern already matched on the sentence. This mode is most useful when the patterns match different kinds of subtrees. For example, you might want to report the first noun and the first verb in each sentence.

If the **-f** option is used (without **-a**), then only the first match of any kind on the sentence is reported. In this case, that would be "the big dog" and "the dog". This is useful if you are interested in the sentence as a whole (**-w**) and you don't want duplicates.

If the **-a** option is used (without **-f**), all matches between subtrees and patterns are reported. Thus, "the big dog" and "a new trick" will be reported twice, and "a cat" and "the dog" will be reported once. This is useful for corpus structural frequency counts where the patterns are to be counted independently.

If the **-f** option is combined with **-a**, then each subtree that matches one or more patterns will be reported, without duplication. In this example, each of the four `NP`s would be reported once. This is useful for corpus structural frequency counts where the different patterns match variants of a single structure type. The same structure will not be counted more than once even if it matches several patterns.

4.9 Macros

TGrep2 patterns can become quite long and complex and therefore difficult to read and manage. It is frequently the case that a subpattern will be repeated and shared across many related patterns. Therefore, TGrep2 supports the use of macros to simplify pattern specification. In addition to simplifying the patterns, the use of macros that are shared across multiple patterns makes revision easier. A change to the macro will automatically affect each of the patterns.

A macro can be defined on the command line or in a pattern file in place of a pattern. The macro definition must begin with an `@` sign *followed by white space* and then the name of the macro. Following this is the value of the macro, ending with a semicolon (`;`). The name of the macro cannot contain any of the special characters: `; : . , & | < > ( ) [ ] $ ! @ % ' ^ =`, but underscores and dashes are permitted. The value of the macro can contain spaces, although leading white space will be ignored.

To use a macro, simply write `@` immediately followed by the name of the macro. This will be replaced by the macro's value. Macro substitution is done as a string, prior to any syntactic parsing of the pattern. Macro substitution will occur even within a quoted node name. To specify a literal `@` within a node name and prevent macro substitution, it can be preceded by a backslash (`\`).

When defining the macro, its name and value can make use of previously defined macros. In this case, substitution occurs immediately, before the new macro is defined, rather than when it is used. Therefore, macros can be built up from other macros. A set of macro definitions might look like the following:

```
@ NP   /^NP/;
@ NN   /^NN/;
@ CNP  @NP=cnp [!< @NP | < @NN] !$.. @NN;
```

In this case, the macro `@NP` now matches any symbol starting with `NP`, which, in the Treebank, will match any noun phrase. Likewise, `@NN` refers to any noun: singular, plural, or proper. A `@CNP`, in this case, stands for "core NP," which is defined here as any NP that is not followed by a sister noun and either dominates a noun or does not dominate any other NP.

It is generally a good idea to define any meaningful component of a pattern using a macro, even if that component, like `/^NP/`, is quite simple. Later on, you may want to revise that component, which will be easy if it is defined as a macro, but difficult if it is repeated throughout many pattern files.

Macro definitions can be given on the command line in place of a pattern or can be included within a pattern file, interspersed with the patterns. However, it may be a good idea to maintain a single macro file for each project, which will be listed on the TGrep2 command line prior to any patterns or pattern files that may use the macros.

5 Controlling the Output

By specifying output formats, it is possible to have TGrep2 report a variety of information in a customizable format. This is detailed in Section 5.2. But for most users, the simpler command-line arguments **-w**, **-l**, **-t**, **-u**, and **-x** will suffice.

By default, when a pattern matches a tree in the corpus, TGrep2 prints the subtree matched by the head node in the pattern. The head node is the first node specified. If the **-w** option is used, the whole sentence is printed, regardless of where within it the match occurred.

The default format for printing trees is the *short form*. This renders each tree on a single line with non-terminals enclosed in parentheses:

```
(NP (NP (NNS sales)) (PP (IN of) (NP (NNP U.S.) (NNS savings) (NNS bonds))))
```

The **-l** option causes the *long form* to be used, which renders trees across multiple lines. This was the default in TGrep:

```
(NP (NP (NNS sales))
    (PP (IN of)
        (NP (NNP U.S.)
            (NNS savings)
            (NNS bonds))))
```

The **-t** option prints only the terminals (words) of the sentence, eliminating all of the structural information:

```
sales of U.S. savings bonds
```

The **-u** option prints only the name of the top symbol in each matching subtree. It will not print the children of that symbol. This would normally be used when the symbol is matched by a wildcard or regular expression, and thus could have several possible names.

Finally, the **-x** option prints the *subtree code* for each match. The subtree code consists of the sentence number followed by a colon (`:`) and the number of the top node in the subtree in the order in which it is encountered on a depth-first search. See Section 5.3 for more information on using subtree codes to later retrieve those subtrees from the corpus.

If you ever request that a node be printed, but the node is not matched to a subtree in the current tree, "`<none>`" is printed in place of the tree. However, if the **-x** option is in use, the sentence number will be printed followed by a colon and 0 for the subtree number.

5.1 Marking Nodes for Printing

As in TGrep, it is possible to select nodes other than the head node for printing. If the name of any node is preceded by a single back-quote (`‘`), the tree matched by that node will be printed, rather than the head node. If multiple nodes are marked, their trees are printed on separate lines.

Without any marks, the pattern "`NP << JJ < /^PP/`" might print this on a match:

```
(NP (NP (DT a) (JJ high) (NN point)) (PP (IN for) (NP (DT the) (NN year))))
```

But "`NP << ‘JJ < ‘/^PP/`" would print:

```
(PP (IN for) (NP (DT the) (NN year)))
(JJ high)
```

Note that the subtrees in this example are not printed in the order in which their nodes were specified because the pattern was reordered for greater speed. If it is important that subtrees be printed in a specific order, one can label the nodes and print the labeled nodes using the **-m** option. It is also possible to suppress the reordering of links with the **-d** option.

If there is any negated link between the head and a node, that node cannot be marked for printing. The reason is that, following a successful match, that node will not be aligned with any particular subtree.

If there is a disjunction of links leading to a node, the node may be marked for printing. However, it is possible that the node will not be matched to any subtree. In that case, "`<none>`" is printed.

5.2 Formatted Output

Although the default output styles should be sufficient for many users, it is also possible to customize the information printed by TGrep2 and the format in which it is printed. The options **-b**, **-s**, and **-m** can be used to set the output formats used at various times.

The **-b** format defines the information printed for each sentence before any matches are attempted. This might be used to print a separator, to print the sentence number, or to print the terminals in the sentence.

The **-s** format defines the information printed for each sentence for which at least one match was successful. Note that this differs from **-b**, which applies to all sentences, regardless of whether there was a match.

The **-m** format is the most useful. It is used every time there is a match between a pattern and a sentence. If the **-f** option was used, there can be at most one match per sentence. Otherwise, there may be more than one match per sentence.

Each of these three formats is a string somewhat like those used by *printf*. The string can contain text, special characters like `\n` or `\t`, and codes representing particular types of fields. Each field code begins with the `%` character. The following fields produce strings or integers:

Non-trees

%f	The name of the corpus file.
%s	The number of the sentence, starting with 1.
%p	The number of the matching pattern, starting with 1.
%i	The number of the current match on the current sentence. The first match to a sentence is 1, the next is 2, and so on.
%j	The number of the current match on the current sentence using the current pattern. This is similar to **%i** but resets every time a new pattern is used.
%c	This prints the comment associated with the current sentence. If there is no comment, nothing will be printed.

If a positive integer N is placed immediately after the `%`, the above fields are right-justified in a column of width N. If the integer is negative, the field is left-justified.

Trees

%h	The head node of the pattern.
%m	The nodes marked for printing. If there are none, the head node is printed. If more than one node is marked for printing, a line break appears between their trees.

%w	The top of the sentence tree.
%=foo=	The node labeled `foo`. Note the = both before and after `foo`.
%*N*b	Prints the sentence tree that is *N* before the current sentence. *N* should be a positive integer.
%*N*a	Prints the sentence tree that is *N* after the current sentence. *N* should be a positive integer.

Tree Printing Styles

By default, trees are printed in short form. One of the following characters can be placed immediately after the `%` to print the tree in a different format.

l	Prints the tree in long, multi-line, form.
t	Prints just the terminal symbols.
u	Prints just the top (*u*ppermost?) symbol in the tree.
n	Prints the node number of the top of the tree in depth-first search order.
x	Prints the subtree code, which consists of the sentence number followed by a colon and the node number.
k	Prints the length of the tree in terminal symbols.
d	Prints the maximum depth of the tree, where a terminal has depth 1.
y	Prints the index of the first terminal in the tree in the sequential ordering of terminals. The first terminal (word) of the sentence has index 1.
z	Prints the index of the last terminal in the tree in the sequential ordering of terminals.

Other than the above fields, any other text in the format is reproduced in the output. Special character codes like `\n` and `\t` can be used to produce newlines or tabs. See the *printf* man page for a complete list of these codes. To produce a single `%` character, `%%` must be specified. No spaces or newlines are printed after fields or trees automatically. Thus, any spaces or line breaks must be specified by the user.

The following example prints a short line, the file name, and sentence number for each sentence for which there is at least one match, followed by the terminals of the previous sentence. For each match, it prints the subtree code for the node labeled `np` followed by a space and the head of the pattern in short form.

```
% tgrep2 -s '----%f %s\n%t1b\n' -m '%x=np= %h\n' 'VP < NP=np'
```

5.3 Extracting Subtrees Using Codes

It is often the case that one needs to compare the subtrees matched by a number of minimally different patterns to detect overlap or subtrees matched by one pattern and not by another. For example, one set of patterns might classify relative clauses based on the position of the modified word. Another set of patterns might classify the clauses based on the internal structure of the clause. By intersecting the set of subtree codes for subject-modifying clauses with the set of codes for object-extracted relative clauses, one could obtain the subject-modifying object-extracted clauses.

Rather than storing and comparing the entire subtrees, which may not be unique, it is more efficient to store and compare the subtree codes. Subtree codes are unique identifiers of subtrees in the corpus and are produced when the **-x** option is used or a tree is printed using a formatting field with the **x** prefix.

If the intersection of two sets of subtree codes has been generated, it might be necessary to retrieve the actual clauses corresponding to those codes. That can be done with the **-e** option. When this is used, the name of a file containing subtree codes is given in place of a search pattern. A match occurs on each of those subtrees and the tree can be printed out as if the head of the search pattern matched the tree. Note that the file of subtree codes must be sorted in non-decreasing sentence order. If a "**–**" is given in place of the file name, the codes will be read from the standard input.

6 Special Topics

6.1 Combinatory Categorial Grammar (CCG) mode

Combinatory Categorial Grammar (Steedman, 2000) is a particular grammar formalism is which a node can be labeled with functor categories, as in `(S[dcl]\NP)/NP`. These node labels can use a number of special characters, including: `()[]\/`. In order to avoid ambiguity, parentheses therefore can't be used to specify the structure of parse trees or patterns, and square brackets can't be used for logical expressions of pattern links. Thus, a special "CCG mode" must be used when working with CCG corpora.

CCG parse tree text files should use curly braces, `{}`, instead of parentheses to surround sub-trees. When converting a text file to a TGrep2 corpus file, the **-g** option should precede **-p**, so that CCG mode will be used. The fact that the corpus is in CCG format will be encoded in the corpus file itself, so users of the file will not need to specify the **-g** flag when using the corpus (although it doesn't hurt).

In specifying CCG search patterns, use curly braces in place of parentheses. Instead of surrounding boolean expressions with `[` and `]`, use `+` and `-`. Also note that within a regular expression, you may need to protect these special characters with a preceding backslash: `()[]\/`. The following is a normal search pattern:

```
S < (NP [< NP | < /^VP/])
```

And this is how it might be written in CCG mode:

```
S[dcl] < {NP + < NP | < /^S\[dcl\]\\NP/ -}
```

7 Differences from TGrep

This section reviews the large and small differences between TGrep and TGrep2.

Differences from and Unsupported Features of TGrep

- There is no distinction between vocab files and corpus files in TGrep2. There are just corpus files.
- TGrep2 does not read commands from a script file, only from the command line.
- The immediate precedence operator does not have the same meaning in TGrep2 as it did in TGrep. This is explained in Section 4.3.
- If no nodes are marked for printing, TGrep2 will print only the subtree matched by the head (first) node in the pattern. TGrep attempted to print all nodes whose subtrees were not descendants of the head node's tree, but did not do it correctly and was thus rather confusing.
- TGrep2 does not have an option to allow a node to be its own sister.
- There is no separate *tprep2* program. The **-p** option is used to generate TGrep2 corpora.
- The **-f**, **-l**, **-c**, **-T**, **-p**, **-P**, **-n**, **-N**, **-s**, and **-e** options to TGrep are subsumed by the use of formatted output in TGrep2.
- The **-v**, **-u**, **-g**, **-O**, **-i**, **-$**, and **-o** options are unsupported in TGrep2.
- TGrep2 uses different flags to represent some of the options also provided by TGrep.

Additional TGrep2 Features

- The `,` `,,` `$,` `$,,` `>>,` `>>'` `>N` `>-N` `>-` `<:` `>:` `<<:` `>>:` and = links have been added.
- There is now direct support for the `<,` `<'` `>,` and `>'` links.
- An = can be appended to any link to allow for a direct match between the node on the left and the right.
- `*` can be used instead of __ to match any node.
- Optional links allow certain features of a tree to be reported if they occur, without preventing a match when they don't.
- Boolean expressions of relationships to other nodes are allowed.
- Node-name matching patterns can be negated.
- Nodes can be labeled, allowing patterns whose link structure is not in tree form and the selection of nodes for printing by label.
- Patterns can be read from files rather than listed on the command line.
- Patterns can be segmented for greater readability and to aid the piecewise composition of patterns.
- Multiple patterns can be searched in a single invocation of TGrep2.
- Macros can be defined to ease the construction of patterns.
- TGrep2 corpus files are up to 20 times smaller than TGrep files.
- Greater control is provided over the printing of context sentences.
- Formatted output allows more information to be printed in a customizable way.
- For most searches, TGrep2 seems to be somewhat faster than TGrep. That is not true, however, for searches involving words that do not occur in the corpus, for which TGrep is specially optimized.
- Supports CCG mode.

References

Marcus, M., Kim, G., Marcinkiewicz, M. A., MacIntyre, R., Bies, A., Ferguson, M., Katz, K., & Schasberger, B. (1994). The Penn Treebank: annotating predicate argument structure. In *Proceedings of the human language technology workshop.* Morgan Kaufmann Publishers Inc.

Marcus, M., Santorini, B., & Marcinkiewicz, M. A. (1993). Building a large annotated corpus of English: the Penn Treebank. *Computational Linguistics*, *19*.

Steedman, M. (2000). *The syntactic process.* Cambridge, MA: The MIT Press.