RC22976 (W0311-046) November 6, 2003
Computer Science

IBM Research Report

The Talent FST System

Branimir K. Boguraev, Mary S. Neff
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

The Talent FST System

Branimir K. Boguraev and Mary S. Neff[1]
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights
NY 10598, USA
`bran@us.ibm.com`, `maryneff@us.ibm.com`

(1) with substantial design and implementation contributions by
Albert Eskenazi and Son Bao Pham

October 31, 2003

Abstract

This document describes the TALENT 5.1 Finite State subsystem (TFST). It positions the process of finite state matching over annotations within the larger context of TALENT's infrastructure. It then describes how to specify rules for pattern matching (over sequences of annotations), and how to specify the features or properties of annotations that a match would be focusing on. The process of grammar writing and development is broadly outlined, and some indication is given concerning the different kinds of tasks and applications which can effectively utilise finite state technology. Some general guidelines on grammar development and matching strategies are offered, and sample grammars are included as examples. Finally, a brief sketch is offered of an experimental environment for finite state grammar development.

The TFST subsystem, as decsribed here, is hosted by the Talent 5.1 document processing infrastructure. Design considerations addressing the encapsulation of FS matching functionality as a Talent (meta-)plugin are discussed in (Neff, Byrd, and Boguraev, 2003). Work is under progress on making the full functionality available, via a new and revised formalism (which targets a typed feature structures-based representation model), within an emerging framework for unstructured information management (UIM; see (Ferrucci and Lally, 2003) for details of UIM architecture).

Evolution of the TFST capability is motivated, beyond well-articulated arguments promoting the deployment of finite state processing techninques for NLP application development, by considerations of enabling such processing within industrial strength NLP frameworks which exploit emerging notions like pipelined architectures, open-ended intercomponent communication, and in particular the adoption of *linguistic annotations* as fundamental descriptive/analytic device.

Contents

1 Introduction

This document assumes some knowledge of the basic design and architectural charactertistics of the TEXTRACT system[2]; see (Neff, Byrd, and Boguraev, 2003) (Neff et al., 2003). We briefly recap here the features of particular relevance to understanding, and using, its finite state (FS) subsystem.

TEXTRACT is a robust document analysis framework, whose design has been motivated by the requirements of an operational system capable of efficient processing of thousands of document/gigabytes of data. It has been engineered for flexible configuration in implementing a broad range of document analysis and linguistic processing tasks. As an architecture, it is modelled upon IBM's SWS Text Analysis Framework:

- interchangeable document parsers allow the 'ingestion' of source documents in more than one format (specifically, XML, HTML, ASCII, as well as a range of proprietary ones);
- a document model provides an abstraction layer between the character-based document stream and annotation-based document components, both structurally derived (such as paragraphs and sections) and linguistically discovered (such as named entities or term phrases);
- linguistic analysis functionalities are provided via individual plugin components[3]; these share an annotation repository (AR) and communicate with each other by posting results to, and reading prior analyses from, it;
- plugins share common interface, and are dispatched by a plugin manager; at a higher level of abstraction, an engine controls shared resources, and maintains the document processing cycle;
- the system is softly configurable, completely from the outside, by means of `.Ini` file;

TEXTRACT's finite state executor (TFST) is implemented as a plugin, which can be configured to interpret one or more grammars in sequence. (The process of applying more than one grammar, with later invocations using results of earlier grammar applications, is called *cascading*.) For any configuration of TEXTRACT, TFST can be positioned anywhere in a plugin sequence. Depending on what plugins (annotators) have run before it, a grammar can specify patterns querying the range of annotation types posted by the upstream annotators.

[2]This document uses, interchangeably, the designations TALENT, TALENT 5.1, and TEXTRACT.
[3]Throughout this document, "annotator" is also used interchangeably with "plugin".

At the time of writing, TEXTRACT has a finite, fixed number of annotation types; as far as a grammar writer is concerned, these fall into three broad categories (families). *Lexical annotations* cover tokens, lexicalised expressions, and vocabulary items. *Syntactic annotations* mark grammatical units (phrases, clauses, grammatical relations). *Document structure* annotations span structural units in a document (sentences, paragraphs, titles, and so forth). TFST grammars can appeal to any, or all, of these annotation types.

1.1 TFst design overview

Numerous NLP applications today deploy finite state processing techniques-for, among other things, efficiency of processing, perspicuity of representation, rapid prototyping, and grammar reusability (see, for instance, (Karttunen et al., 1996), (Kornai, 1999) Karttunen et al., 1996; Kornai, 1999). TEXTRACT's TFST transducer plugin encapsulates FS matching and transduction capabilities and makes these available for independent development of grammar-based linguistic filters and processors.

In a pipelined architecture, and in an environment designed to facilitate and promote reusability, there are some questions about the underlying data stream over which the FS machinery operates, as well as about the mechanisms for making the infrastructure components—in particular the annotation repository (henceforth AR) and shared resources (TEXTRACTs lexical cache and vocabulary)—available to the grammar writer. Given that the document character buffer logically 'disappears' from a plugin's point of view, FS operations now have to be defined over annotations and their properties. This leads to a notation in which grammars can be written with reference to TEXTRACT's underlying data model, and which still have access to the full complement of methods for manipulating annotations.

We make use of an abstraction layer[4] between an annotation representation (as

[4]In the extreme, what is required is an environment for FS calculus over typed feature structures (see (Becker et al., 2002)), with pattern-action rules where patterns would be specified over type configurations, and actions would manipulate annotation types in the AR. Manipulation of annotations from FS specifications is also done in other annotation-based text processing architectures (see, for instance, (Cunningham, Maynard, and Tablan, 2000)). However, this is typically achieved by allowing for code fragments on the right-hand side of the rules.

Both assumptions—that a grammar writer would be familiar with the complete type system employed by all upstream (and possibly third party) plugins, and that a grammar writer would be knowledgeable enough to deploy raw API's to the annotation repository and resource manager—go against the grain of TEXTRACT's design philosophy.

However, in response to moving away from a text processing infrastructure with a pre-defined and it is implemented) and a set of annotation property specifications which define individual plugin capabilities and granularity of analysis. The notation developed for specifying of FS operations, and described in this document, appeals to the system-wide set of annotation families, with their property attributes. The notation also encapsulates operations over annotations—such as create new ones, remove existing ones, modify and/or add properties, and so forth—as primitive operations. The abstraction thus hides from the grammar writer system-wide design decisions, which separate the annotation repository, the lexicon, and the vocabulary: thus, for instance, access to lexical resources with morpho-syntactic information, or, indeed, to external repositories like gazetteers or lexical databases, appears to the grammar writer as querying an annotation with morpho-syntactic properties and attribute values; similarly, a rule can post a new vocabulary item using notational devices identical to those for posting annotations.

For grammars which examine uniform annotation types, it is relatively straightforward to infer, and construct (for the runtime FS interpreter), an iterator over such a type (in this example, sentences). However, expressive and powerful FS grammars may be written which inspect, at different—or even the same—point of the analysis annotations of different types. In this case it is essential that the appropriate iterators get constructed, and composed, so that a felicitous annotation stream gets submitted to the run-time for inspection; TEXTRACT deploys a special dual-level iterator designed expressly for this purpose.

Additional features of TFST allow for seamless integration of character-based regular expres-sion matching, morpho-syntactic abstraction from the underlying lexicon representation and part-of-speech tagset, composition of complex attribute specification from simple feature tests, and the ability to constrain rule application within the boundaries of specified anno-tation types only. This allows for the easy specification, via the grammar rules, of a variety of matching regimes which can transparently query upstream annotators of which only the externally published capabilities are known.

Applications utilising TFST include a shallow parser (Boguraev, 2000), a front end to a glossary identification tool (Park, Byrd, and Boguraev, 2002), a parser for temporal expressions, a named entity recognition device, and a tool for extracting hypernym relations.

fixed data model to an environment where NLP applications will define and manipulate arbitrarily open sets of annotations (see (Ferrucci and Lally, 2003)), the next major release of the TFST system will incorporate a more flexible notation which can mediate between a data model defined specifically for a particular application and a grammar writer operating at an abstraction level where an annotation is expressed as a typed feature structure.

Work is under progress on making the full functionality available, via a new and revised formalism (which targets a typed feature structures-based representation model), within an emerging framework for unstructured information management (UIM; see (Ferrucci and Lally, 2003) for details of UIM architecture).

Evolution of the TFST capability is motivated, beyond well-articulated arguments promoting the deployment of finite state processing techninques for NLP application development, by considerations of enabling such processing within industrial strength NLP frameworks which exploit emerging notions like pipelined architectures, open-ended intercomponent communication, and in particular the adoption of *linguistic annotations* as fundamental descriptive/analytic device. For such frameworks, certain issues arise concerning the underlying data stream over which the FS machinery operates. A review of recent work on finite-state processing of annotations and an elaboration of some essential features required from a 'congenial' architecture for NLP aiming to be broadly applicable to, and configurable for, an open-ended set of tasks, was presented in Boguraev (2003).

2 Overview of the process

Typically, the TFST component of TEXTRACT gets invoked by configuring and running the FST executor plugin. The executor loads one or more FST files, looks for (repeated) patterns of annotations—in the current annotation repository, as populated by prior plugins—which match any given FS automaton, and applies the automata in sequence, cascading them if necessary (when there is more than one specified in the sequence). There are provisions in the TEXTRACT system to output formatted stream of matches, which displays the state of the annotation repository after the FST executor has completed.

The grammar writer specifies patterns over annotations by means of a high-level notation. The process is not unlike specifying regular expressions over character sequences; however, given that at each transition of the equivalent automaton queries/evaluates arbitrarily complex set of constraints, the notation incorporates complex (and arguably less than intuitive[5]) syntax for specifying what annotation to match, and the conditions under which the match is deemed to be succesful.

Pattern files are thus text files, written to a set of syntactic and orthographic rules. These are compiled, outside of the TEXTRACT environment, into FST files which encapsulate machine-readable representations of the equivalent automata. TEXTRACT's FST executor is configured to use these, for a particular run, by means of a stanza in the configuration (.Ini) file. The results can be viewed by means of further configuring the output (dump) plugin; this is a non-processing, read-only, plugin which selectively (and under .Ini control) extracts certain annotations from the annotation repository and prints them in a human-readable form.

2.1 Matching specifics

The TFST executor applies an FS automaton to an annotation stream, with transitions between states in the automaton conditioned upon matching one or more features of the current annotation. A successful match (conceptually) effects a transition; it also advances an annotation iterator, to yield the next annotation against which new matches will be attempted, according to the outgoing arcs of the new state of the automaton. If no match can be found, the automaton is applied again at the next annotation returned by a (suitable) iterator.

Currently, the matching regime returns only one—the longest—match. After a

[5]The next major release of TFST will appeal, more perspicuously, to TALENT's underlying data model.

successful match, the annotation stream gets reset to the first annotation following the matching span, and the automaton is applied again; this process is repeated over the entire annotation repository, thus resulting in all matches against a given FST over the document.

As an example, consider a simple grammar specifying a noun sequence[6], applied to the following text; the matching text fragments are underlined:

> *American aircraft frequently bomb missile and radar equipment sites that allegedly target U.S. planes. In recent weeks, the attacks have focused on Iraqi weapons that might be used in a ground war.*

Note that internally, the complete scan of the input, with repeated attempts to apply an automaton at every point of a string, needs to be mediated in a way which takes into account natural separation boundaries. In particular, an inner loop over sentences ensures that no patterns—especially syntactic ones—match across sentence boundary.

An annotation-based regime of FS matching needs a mechanism for picking a particular path through the input annotation lattice, over which a rule should be applied: thus, for instance, some grammars would inspect raw tokens, others would abstract over vocabulary items (some of which would cover multiple tokens), yet others might traffic in constituent phrasal units (with an additional constrain over phrase type) or/and document structure elements (such as section titles, sentences, and so forth). Typically, at any given point in the text, there would be more than one annotation types either covering that point, or starting and/or ending at that position. For instance, considering the second sentence in the example above, the position at the beginning of *"have"* might be'covered' both by a token annotation and a verb group annotation[7]. Similarly, the position before *"Iraqi"* is a starting position for a token annotation (*"Iraqi"*), for a semantic modifier (*"Iraqi"*, as in *"of Iraq"*), a syntactic noun phrase (*"Iraqi weapons"*), a more complex noun phrase, composed of a simple NP and a relative clause (*"Iraqi weapons that might be used in a ground war"*), and even a grammatical function annotation, specifying that the complex noun phrase also functions as grammatical object.

Thus, when in this position of the annotation stream the TFST executor attempts a match over the *next* annotation, it is important that the grammar writer appreciates

[6]Such a grammar would be written as: `nn = {NN}.{NN}* ;` see 3 below, and its application would assume a part-of-speech tagger running over the text as a prior process

[7]This assumes a parser—such as TEXTRACT's shallow parser—has run

that there are multiple options as to the choice of annotation, and that the grammar needs to communicate precise instructions about which of these annotations is to be examined upon the current transition. Note that this is a crucial difference between an FS matching regime over annotations, compared to the more familiar, and more common, application of FS automata over purely character sequences. In the former case, the input to a matcher is a lattice, with ambiguous choices of tracing a particular route through it; this contrasts with unambiguously advancing that 'match position' character after character, with no ambiguity as to what is the next character.

The information about what annotation to inspect at any given point of applying an automaton over the annotation repository is encoded in the symbol on the current transition. Symbols thus encapsulate instructions for which iterator needs to be (re-)activated to deliver an annotation of a particular type, and what constraints/ features are to be examined while attempting this transition.

A later section (4) describes the syntax, orthography, and semantics of TFST symbols. Before that, section 3 describes the notation for specifying annotation patterns, and for composing these in .cfg files.

3 TFst grammars

Specifying patterns of annotations will be familiar to everyone who has written a regular expression. In much the same way in which a regular expression specifies an automaton encoding a set of allowed sequences of characters, a TFST grammar specifies an automaton whose traversal—from beginning to accepting state—would indicate that a sequence of annotations has been found which matches the pattern specified by the grammar.

To illustrate, consider a (very simple) grammar for noun phrases, defined over the part-of-speech tags of token annotations.[8]

```
np      = {DT}|<E> . {JJ}* . {NN}|{NNS}  ;
```

Without going into much detail (but see 3.1 below), we note that {DT}, {JJ}, and {NN} denote, respectively, a token tagged as a determiner, an adjective, or a noun; the symbol `<E>` marks an empty transition, and the operators `.` and `*` specify, respectively, sequential composition and zero or more repetitions. In effect, the grammar looks for a sequence of tokens, which starts with an optional determiner, includes zero or more adjectives, and terminates with a noun (singular or plural).

3.1 Grammar notation

More formally, a pattern for matching a sequence of annotations is defined by composing *symbols* (see 4 for definition and syntax of TFST symbols) according to a few rules.

- The symbol `<E>` denotes the empty symbol. Matching against it always succeeds.
- A symbol defines a pattern by itself; thus {NN} defines a very simple pattern which will be matched by tokens whose syntactic category is `"NN"` (noun).
- If P is a pattern, so is $(\,P\,)$. It matches the same text span as P.
- If P_1 and P_2 are patterns, so is $P_1 \,.\, P_2$. Assuming T_1 and T_2 are text spans which, under any given analysis by prior plugins, are matched by P_1 and P_2

[8]Part-of-speech disambiguation is carried out by a prior process to TFST execution, by one of TEXTRACT's *tagger* plugins; see 5.2.

respectively, the combined pattern matches the concatenation of T_1 and T_2.

- If P_1 and P_2 are patterns, so is $P_1 \mid P_2$. It matches any text span which will be matched by either P_1 or P_2 .
- If P is a pattern, so is $P*$. It matches a contiguous (non-broken) sequence of zero or more text spans, each of which matches P.
- If P_1 and P_2 are patterns, so is P_1`&`P_2. Both P_1 and P_2 must 'match' over the same text span, i.e. they are applied to the same annotation. Thus, the `&` operator functions as a conjunction.
- Each pattern specification in a grammar is named. There is no 'semantics' to the name assigned to a pattern; it is purely mnemonic. The name of a pattern, however, acts as a 'macro', and can be used by other rules (naming of a pattern acts as a combined declaration/definition; thus named patterns can only be used in patterns specified *after* the definition). In a grammar which contains the following pattern specification: `namedP =` P`;`, subsequent rules can refer to pattern P by writing `:namedP`.

The core grammar writing notation does not provide other familiar regular expression-like operators, in particular wild-card (cf. `"."`), zero or one occurrences of a pattern (`"?"`), and one or more repetitions (`"+"`). Note, however, that these are easily specified, or handled elsewhere in the formalism:

- If P is a pattern, $P \mid$ `<E>` matches zero or one occurrences of that pattern in the text.
- If P is a pattern, P `.` P`*` matches at least one, and possibly more, occurrences of that pattern.
- Given that wild-carding over an annotation at a given point of the annotation stream is ambiguous—recall that more than one annotations may be available as beginning at any token position—different symbols denote the annotation type of the wild card; in particular, `<SWORD>` matches any token, `<WORD>` matches any lexical family annotation (thus it will match vocabulary items[9] in text), and `[SYN]` matches any syntactic family annotation (see 4.7).

3.2 Grammar organisation

A grammar consists of one or more pattern specifications. Patterns are named, and delimited by semicolon (`;`).

A grammar defines a single automaton; thus all the patterns are ultimately—or

[9]Define...

should be, or they will be lost—used in the final, 'root' rule of the grammar. By convention, a grammar file is a text file, whose basename *must* be identical[10] to the root rule name. Returning to our simple one-rule grammar of earlier, this could be reorganised as follows:

```
det      = {DT}|<E>    ;      # optional determiner
premod   = {JJ}*       ;      # zero or more pre-mod adjectives
noun     = {NN}|{NNS}  ;      # the head nominal

np       = :det . :premod . :noun  ;
```

This grammar will be stored in a file, `np.cfg`, named after the root pattern rule. Comments are allowed, and introduced by a comment character (`#`); comments run to the end of line.

3.3 Grammar to Talent interface

The grammar in the example above defines an automaton which will match certain simple NP's. The success or failure of applying this automaton to any given text[11] is ephemeral; unless something is done at the point of a successful match, there will be no memory of it in the system.

The TFST executor applies an automaton to the annotation repository. According to grammar instructions, matches are recorded, as new annotations (and/or features on annotations) in the same repository. Conceptually, in much the same way in which a FS transduction modifies a character string, a TFST 'transduction' modifies an annotation repository.

Operations on annotations are typically signalled by the meta-character `/`. The general notion is that if P is a pattern, and T is an operation over an annotation repository, then P/T is a composite symbol, whose interpretation is to execute the operation T if and only if the pattern P matches.

Currently there are three categories of transactions: create, and post, new annotations, and optionally post vocabulary items; post properties on annotations[12]; and

[10] Failure to do so results in error during compilation of the grammar into an FST: see 5.1

[11] Re-write, consistently, to address that matching of an FST is against an AR, really, not a text...

[12] Check: can we modify existing properties, on existing annotations?

delete annotations.

As an example, consider again our NP grammar. To ‘remember’, and record, the match, it needs addition to the root rule:

```
    np       = <E>/[simpleNP:phrasal .      # post a new anno from here...
                  :det . :premod . :noun . # covering the np match,
               <E>/]simpleNP               # to here; label it "simpleNP"
               ;
```

Ignoring for the time being the `:phrasal` specification (but see section 4.9.1), the pair of transduction operations jointly achieve the effect of posting an annotation over the span of text matched by the `np` rule, with a label `"simpleNP"`. More specifically, the meta-character `[` simply prepares for posting, by ‘dropping’ a marker right before a match; later, `]` actually carries out the operation of adding a labelled annotation to the annotation repository. Note that the label strings on the `[` and `]` operators must be identical.

Annotations can carry features, and these can be set from the grammar rules. In the above example, something about the matching span (namely, that it is a simple—as opposed to, say, complex—noun phrase) is registered in the label. Similar effect could be achieved by means of setting an appropriate feature on the new annotation:

```
    np       = <E>/[NP:phrasal .            # post a new anno from here...
                  :det . :premod . :noun . # covering the np match,
               <E>/]NP:+simple             # to here; label it "NP",
                                           # set a binary feature simple=true
               ;
```

Features can be binary, or they can hold numeric values. Accordingly, the transduction should specify one of the following (for more on features, see section 4.9.1).

- `/]` *Label:+Feature*
- `/]` *Label:-Feature*
- `/]` *Label:Feature=IntValue*

Finally, annotations can be deleted directly from the grammar. Deletion—signalled by a `-` operation on the transduction—removes the annotation just matched. Deleting annotations typically corrects earlier process decisions (either made by an upstream annotator, or a lower grammar in a multi-level FS cascade of grammars; see 3.4), or removes 'scratch' annotations, used to hold intermediate results.

Consider a grammar which runs after identification of simplex noun phrases has already happened (see section 3.4, on cascading multiple grammars). The following rule implements the notion that a possessive construction over a simplex noun phrase can function, syntactically, as a complex determiner.

```
np  = <E>/[PossNP:phrasal .
         <E>/[PossDet:lexical . [NP]/- . <POSS> . <E>/]PossDet .
         [NP] .
      <E>/]PossNP
      ;
```

The grammar assumes that syntactic annotations with an `"NP"` label have already been posted, and it looks for patterns of larger possessive noun phrases with a contour of *NP's NP* (a possessive marker, *'s*, will match the `<POSS>` symbol). When it finds such a pattern, the `"NP"` annotation over the first noun phrase is deleted, a new annotation with label `"PossDet"` gets posted over the sequence of noun phrase and possessive marker, and a covering (also new) annotation is created to span the entire possessive phrase, with a label `"PossNP"`.

In effect, the annotations over the text have undergone the following transformation:

```
from                [NP NP NP] <POSS> [NP NP NP]

to     [PossNP [PossDet NP <POSS> PossDet] [NP NP NP] PossNP]
```

3.4 FS grammar cascading

The notion of running more than one grammar, in sequence, with latter ones using matches from earlier scans, is commonly referred as *cascading*. This is a common, and effective, strategy for a variety of different tasks.

One of the primary resons for organising an analysis task as a sequence of cascaded grammars, as opposed to designing a single automaton, is that frequently more complex patterns can mush easier, and more naturally, be described in terms of simpler ones. Examples from natural language syntax abound. Consider a grammar for sinding noun groups in isolation 5.3. Such a grammar might decide to focus on the basic contour of a noun group: determiner unit, pre-modifier(s), and head nominal. It can be arbitrarily sophisticated in defining different types of nominal which can function as heads, and in specifying a broad variety of pre-modifying constructs. Once defined, and with annotations marking the span of such text fragments, the definition of more complex nominal expressions—such as those that include post-head modifiers, or recursively embedded noun phrases acting as pre-determiners for other noun phrases (as exemplified in the preceding section)—becomes both more natural and convenient.

This is because it is now straightforward to define higher level rules in terms of single annotations, abstracting from the complexity of earlier patterns which resulted in those annotations being created in the first place. Also, there may be multiple places in higher level analysis where a lower level abstraction would be required: consider how many syntactic and grammatical and semantic and so forth constructs are describable in terms of some kind of a noun phrase.

Cascading also offers a convenient way of being sensitive to larger context. Frequently, specifying all the interdependencies among constituents, and making sure that these are triggered on in some, but not other, contexts may turn out to be arbitrarily complex, confusing, and even impossible. Splitting the analysis into separate *identify-and-group*, or *over-generate-and-filter* makes the task tractable.

Consider, for example, finding appositive noun phrases. These are expressions where in a single noun phrase construct there is both a mention, and a description, of an object: *“Scott Ritter, a former inspector”*, *“the toughest Soviet commander in Afghanistan, Lt. Gen. Boris Gromov”*. Assuming a separate noun group identification grammar, which can tag the spans it marks with features like `+properNoun` (to denote that the head of the noun group is a proper noun name), and which can be run as a component to a larger noun phrase grammar (which folds post-modifying prepositional phrases within the NP span), a simple rule can express the observation that an appositive noun phrase conjoins two NPs, one of which has a proper noun head, in arbitary order, separated by a comma, and delimited by a comma or end of sentence. (The exact semantics of the different kinds of symbol used in the example are described in the following section, 4.)

```
  appNP  = <E>/[appositiveNP:phrasal .
              ( [NP:-properNP] . <COMMA> . [NP:+properNP] ) |
              ( [NP:+properNP] . <COMMA> . [NP:-properNP] ) .
              ( <COMMA> | <PERIOD> )
           <E>/[appositiveNP
           ;
```

The simplicity and perspicuity of such an expression ows largely to the fact that much of the complexity in identifying and marking `[NP]`'s is relocated to lower levels of the cascade.

By its nature, cascading grammars is a mechanism which promotes multi-level structural descriptions of complex constituent phrases and strings. An interpretation of a cascade by the TFST interpreter mimics, in effect, bottom-up recognition; if each level of the cascade assigns—by means of a suitable annotation—a label to its analysis, the process of passing these analysis to be consumed, and used, at higher levels will result in a structured representation of the string ultimately recognised by the cascade.[13]

Another strategy for which cascading is useful, *over-generate-and-filter*, is based on the observation that sometimes it is much easier to recognise—and recognise only—strings which conform to a higher level, uniform, description, without trying to carry out, simultaneously with the recognition, more elaborate analysis of these, such as e.g. sub-typing. For instance, it is relatively straightforward to write (for English) a grammar for a verb group; this would recognise expressions like *"sleeps"*, *"is not walking"*, *"would have most likely been seen"*, *"has also been known as"*, and so forth. Operationally, it is both easier and cheaper to defer till later—i.e. to a subsequent grammar—computation of certain syntactic features, such as modality and tense, on the verb group. To that end, a special notation is provided, as a device for examining the *inner contour* of an annotation already

[13]Maybe an example here of the structure of an appositive analysis: `[appNP [headNP [... headN]] [descriptiveNP ...[]...]]`.

posted by an earlier component in a cascade (see section 4.10); an example of the use of this notation is available in section 5.3.

Finally, TFST cascades are a natural means for stratification of analysis. In typical information extraction tasks, an earlier component identifies mentions in the text of semantic categories of particular relevance to the domain of application. Examples here would include, for instance, diseases, symptoms and drugs, in a medical domain, or geopolitical entities, political figures, and geographical locations, in a political domain. Grammars for such categories are typically idiosyncratic, both in terms of vocabulary coverage and identification strategy. Nonetheless, higher levels of abstraction in either domain are likely to appeal to more traditional syntactic notions like verb groups and its arguments, or a complex nominal structure and its pre-modifiers. Being able to separate semantic analysis from syntactic behaviour not only facilitates cross-domain adaptability and reuse of grammars, but also offers a natural way of incorporating semantic categories into syntactically-mediated analysis of relationships among such categories. In essence, the ability to cascade FS grammars is the cornerstone of rich and flexible domain semantics.

4 TFst symbols

A symbol in a TFST grammar is a template for an annotation which is to be matched againjst the grammar at a given position of the matching process. Much in the same way as in a conventional regular expression a character is matched against the character at the current position of the input buffer, a TFST symbol specifies the test(s) to be carried out, which would determine whether the match of the symbol against the current annotation succeeds or fails.

Recall that the notion of "current" annotation is an ambiguous one in TEXTRACT. A symbol, then, not only encapsulates one or more constraint specifications; it also communicates to the Tfst executor engine how to iterate over the annotation repository in order to identify which annotation the constraints should be tested again.

In line with the major annotation families in TEXTRACT (see 1), the overall shape of a symbol is broadly indicative of the annotation type it targets. Symbols can be enclosed in different type of delimiters, or not delimited (explicitly) at all. Thus there are five general categories of symbol.

- Not delimited by any kind of brackets: `however`, `\Atomic`. This kind of symbol typically targets the string property of a token.
- Delimited by angle brackets (`<...>`): `<Sentence::IsTitle>`, `<SWORD>`, `<PUNC>`. Tests for orthographic features on lexical annotations, configurational relationships between lexical annotations and document structure characteristics, or property lookup with respect to external authotity.
- Delimited by curly brackets (`{...}`): `{NN}`, `{VB+AUX}`, `{PERSON}`. Tests on lexical properties of lexical family annotations, vocabulary properties, morphosyntactic categories of word tokens,
- Delimited by square brackets (`[...]`): `[NP]`, `[SYN]`. Tests for syntactic categories with given labels.
- Delimited by double angle brackets (`<<...>>`): `<<^[12][0-9]{3}$>>`. Specifies a regular expression match over the underlying string image of a lexical annotation.

As a general rule, the meta-character `!` is used to negate the match. Negation only applies to symbols, and its interpretation is that only after the symbol (without negation) has been fully interpreted and matched against the current annotation, the polarity of the result is reversed, and returned as the final value from testing against the composite symbol.

The negation character has to be ‘inserted’ into the rule, right after the opening bracket(s): `<!LOWER>` will match a lexical annotation whose string image contains at least one upper-case character; `[!NP]` will match a syntactic annotation with any label but `"NP"`. Note that, counter-intuitively, this convention makes the `!` look like a part of a regular expression; still `<<!Jan\.?$>>` will succeed on any token but *Jan* or *Jan.*. Also, note that for symbols of the first category above—without brackets—the `!` is prepended to the symbol expression: `!\OK`.

Below are the semantics of the different symbol categories.

Symbols not enclosed in any kind of brackets specify matches over the strings of lexical family annotations.

4.1 Tests over lexical annotation strings

- Symbol strings entirely in lower case specify a match which succeeds upon case-insensitive string equality to the annotation text string. This the symbol `abracadabra` will match all of *abracadabra*, *Abracadabra* , and *AbraCaDabra*.
- Symbol strings which are not all lower case succeed only upon exact string match: `Ok` will match *Ok*, but fail on *OK*.
- Symbols preceded by escape character (\) succeed only upon exact string match.

4.2 Tests for orthographic properties

Symbols in this category test one or more characteristics of the shape of a lexical family annotation.

`<WORD>`	: a lexical family annotation
`<PUNC>`	: special character(s)
`<POSS>`	: a possessive type annotation (*’s*, *’*)
`<SWORD>`	: a single word (no blank spaces)
`<MWORD>`	: a multi-word (has blank space(s))
`<ALPHA>`	: a single or multi-word having only alphabetic characters
`<LOWER>`	: has no uppercase letters
`<UPPER>`	: has no lowercase letters
`<LUPPER>`	: has leading upper case
`<LEAD>`	: only the first character is upper case
`<MIXED>`	: has lower case letters and at least one upper case one after a non-blank
`<NUMERIC>`	: has only numbers
`<COMMA>`	: comma

`<PERIOD>`	: period
`<ODQ>`	: open double quotes
`<CDQ>`	: closed double quotes
`<DDQ>`	: double quote
`<OSQ>`	: open single quote
`<CSQ>`	: closed single quote
`<E>`	: empty transition. This is a no-op.

If the annotation being matched refers to a word, the symbol may incorporate an additional test for its morpho-syntactic (in particular, part-of-speech; see 4.6) properties; thus, `<ALPHA:NN>` will match a single word which has been tagged as a noun {`NN`}.

4.3 Tests for document structure relationships

Always with respect to the current annotation, the tests below succed or fail upon establishing whether a specified relationship holds between the current anotation and some enclosing (larger) document structure annotation. The semantics of the symbols are, hopefully, directly inferrable. They denote a particular relationship (from an exhaustive inventory of pre-defined relations) between (larger) document structure annotations covering the annotation at hand; another denotation tests a particular property, especially on the enclosing `Sentence` annotation. (Properties on sentence annotations—the sentence is marked as a heading, or belonging to a document abstract, or is 'vanilla', i.e. a regular sentence—are set by a document structure analysis plugin which is part of TEXTRACT's base services; the detail of analysis, however, varies depending on the quality and extent of markup in the original document source.)

```
<Document::IsSectionFirst>          <Paragraph::IsSentenceFirst>
<Document::IsSectionLast>           <Paragraph::IsSentenceLast>
<Document::IsParagraphFirst>        <Paragraph::IsWordFirst>
<Document::IsParagraphLast>         <Paragraph::IsWordLast>
<Document::IsSentenceFirst>
<Document::IsSentenceLast>          <Sentence::IsWordFirst>
<Document::IsWordFirst>             <Sentence::IsWordLast>
<Document::IsWordLast>              <Sentence::IsVanilla>
                                    <Sentence::IsHeading>
<Segment::IsSentenceFirst>          <Sentence::IsNonText>
<Segment::IsSentenceLast>           <Sentence::IsAbstract>
<Segment::IsWordFirst>              <Sentence::IsCaption>
<Segment::IsWordLast>               <Sentence::IsMetaData>
```

```
<Section::IsParagraphFirst>          <Sentence::IsAppendix>
<Section::IsParagraphLast>           <Sentence::IsTitle>
<Section::IsSentenceFirst>           <Sentence::IsJunk>
<Section::IsSentenceLast>            <Sentence::IsTable>
<Section::IsWordFirst>
<Section:IsWordLast>
```

Thus `<Document::IsSentenceFirst>` succeeds only if the sentence enclosing the current annotation is, in fact, the first sentence in the document; and `<Sentence::IsTitle>` tests whether the *Title* property is set for the sentence annotation covering the current 'point' in the annotation stream.

It is clear that there is no room in this category of symbol for additional tests on the annotation itself. This is generally true of a range of complex tests: it is not always the case that a ready-made symbol exists which encapsulates just the right combination of constraints over the current annotation. These are prime examples of a situation which requires the use of the `&` grammar operator (see 3.1). Thus, a match of a pattern within a particular document structure configuration might take the form:

```
np        = <!Sentence::IsTitle> & :det . :premod . :noun  ;
```

This rule would find all noun phrases (see the `"NP"` grammar in 3.2) in the document, except those which are in the title sentence.

It is also possible to test properties of document structure annotations. Typically, these would have been set by a transduction operation (see 4.9 below). Given that properties can be either binary, or of numeric (integer) values, the test might take the form of `<Paragraph::+boolProp>`, or `<Document::numProp=5>`.

4.4 Tests against an authority file

TEXTRACT provides a general mechanism for consulting an external authority file. Generally speaking, an authority file assigns a lexical item to one or more semantic categories, and (optionally) tags it with a number of property tags. The most common use of an authority file, to date, has been to inform a named entity recogniser (Ravin and Wacholder, 1997); to that end, the following illustrates the general type of entry that could be found in an authority file.

```
@ ORG   ORG_OFTAG   @ 0   0   0   0   1 |Council|
@ PERSON   PERSON_OFTAG   @ 0   0   0   0   1 |Inspector|
@ PERSON   PERSON_OFTAG PERSON_PLURAL   @ 0   0   0   0   1 |Inspectors|
@ PLACE   PLACE_CITY   @ 0   0   0   0   1 |Peshawar|
@ PLACE   PLACE_NAME @ 0   0   0   0   1 |Ivanovskaya|
@ PERSON   PERSON_FIRSTNAME @ 0   0   0   0   4 |William| |Willy| |Bill| |Billy|
```

It is beyond the scope of this document to present details of TEXTRACT's external authority mechanism. Concerning the format and tags of the authority source, associated tests and API's, extensions and/or modifications to incorporate larger and/different sets of categories and tags, and so forth, the reader should contact one of the authors; also see (Ravin and Wacholder, 1997) for some background information.

Broadly speaking, the API to an external authority is organised in such a way that for each tag in the authority vocabulary (such as the `ORG`, `PERSON`, `PLACE_CITY`, and so forth exemplified above, there is a test. The authority file gets compiled into a hash structure, available to TEXTRACT's system as a `VDict` resource. Such a resource gets registered with the system by means of a declaration in TFST's executor configuration setup (see 5.2):

Authority_Identifier = `VdictAuthority(`*filename*`)`

For example, in order to register TEXTRACT's default authority file (the one used by the `Nominator` plugin) under the symbolic name `Nominator` with TFST, the following declaration should be placed in the `[nfstxeq]` section of the `.Ini` file[14].

```
Nominator = VdictAuthority(C:/TalentRT/TalentData/nom.authority.vdict)
```

After registration, the name of the *Authority_Identifier* becomes available to the grammar via one of the following constructs:

[14]Allowing, of course, for local directory structure of the TALENT run-time environment.

```
< Authority_Identifier? test>
< Authority_Identifier? test, value>
< Authority_Identifier? test, op, value>
```

The *op* parameter can be either `EQ` (equals), `GT` (greater than), `GTE` (greater than or equal to), `LT` (less than), or `LTE` (less than or equal to). The first syntax option is equivalent to <*Authority_Identifier*?*test*`EQ`*true*, and the second syntax option is equivalent to <*Authority_Identifier*?*test*`EQ`*value*.

The complete sets of tests against TEXTRACT's standard Nominator `VDict` authority is given below.

`Person`	: tests if person
`PersonOFTag`	
`PersonEndTag`	
`PersonBegTag`	
`PersonWeakBeg`	
`PersonWeakEnd`	
`Male`	: tests if word indicates a male
`Female`	: tests if word indicates a female
`LowerCasePrefix`	: tests if is a lower case prefix of a person's name
`UpperCasePrefix`	: tests if is a prefix of a person's name containing upper case letters
`Royal`	: tests if word is the name of royalty
`Surname`	: tests if word is a surname
`FName`	
`FirstName`	: tests if word is a first name
`LowerCaseName`	: tests if word is a lowercase person name
`PersonWeakOFTag`	
`PersonPlural`	: tests if word is a plural name
`Org`	: tests if word is an organization
`OrgOFTag`	
`OrgTag`	
`OrgWeakEnd`	
`OrgEndTag`	
`OrgSingleWord`	: tests if a word is a one word organization
`OrgName`	: tests if word is an organization name
`Government`	: tests if word is part of a governmental organization name
`Company`	: tests if word is part of a company name
`OrgLongTag`	
`OrgWeakTag`	
`Place`	: tests if word is a place
`PlaceOFTag`	
`PlaceBegTag`	
`PlaceEndTag`	

`City`	: tests if word is a city
`Country`	: tests if word is a country
`Continent`	: tests if word is a continent
`USPlace`	: tests if word is a place in United States
`PlaceName`	: tests if word is a place name
`PlaceWeakTag`	
`Road`	: tests if word is part of a road name
`Capital`	: tests if word is a capital
`Region`	: tests if word is a region
`CanadaPlace`	: tests if word is a place in Canada
`Word`	: tests if word
`LowerCaseWord`	: tests if word contains all lower case letters
`Func`	
`PNAdjective`	: tests if word is a proper noun adjective
`Singular`	: tests if word is singular
`NoName`	
`NotAlone`	
`Date`	: tests if word is part of a date
`RomanNumeral`	: tests if word is a roman numeral
`Modifier`	: tests if word is a modifier
`NatModifier`	: test if word is a national modifier
`Day`	: tests if words is a day
`NoEOS`	: tests if word cannot be at the end of a sentence
`LowerCaseName`	: tests if word is a name with all lower case letters
`TechNoName`	
`AbbreviatedWord`	: tests if word is an abbreviation
`Lemma`	: tests if word is a lemma form
`Separator`	
`Other`	: tests if word is categorized as other
`Entity`	: tests if word is an entity
`Area`	: tests if word is an area
`Product`	: tests if word is a product
`Fund`	: tests if word is a fund
`OtherSingleWord`	
`WordNumber`	
`Uncategorized`	: tests if word is uncategorized
`MakeUncat`	
`Term`	
`Abbreviation`	
`Place?`	
`Person?`	
`Special`	
`SpecialMake`	
`SpecialIn`	
`SpecialModal`	
`SpecialLocation`	
`SpecialNPrep`	
`SpecialCompany`	
`SpecialEmpty`	
`SpecialFinance`	
`SpecailMakeIdiom`	
`NP`	

For an `ImDict` authority file, such as TEXTRACT's statistics authority file (used by e.g. the Summarizer application), the following tests are supported.

`exists`	: tests if word is in the authority file
`IQ`	: tests the IQ value
`freq`	: tests the frequency of the word in the collection
`total_freq`	: tests the collection size

4.5 Tests of vocabulary properties

Symbols enclosed in curly brackets specify tests on the properties of lexical family annotations. These are currently limited to lemma forms of words, canonical forms of vocabulary, the morpho-syntactic properties of words, and the category of vocabulary.

- {*text string*}, where *text string* is a lemma form or a canonical form of a word or vocabulary item. If the lemma form or canonical form begins with an upper case letter, it must be escaped with the TFST escape character, by default \ (cf. 4.1: `\United States`). The text string may optionally be followed by a syntactic category or vocabulary category specification (see below), e.g. \{`Chicago:PLACE`}.
- {`VCAT`} where `VCAT` is a vocabulary category: `UWORD`, `UABBR`, `UNAME`, `OTHER`, `PLACE`, `PERSON`, `ORG`, `PLACE?`, `PERSON?`, `ORG?`, `UTERM`. These strings are listed in `talent_cats.cpp`. Currently these must match case exactly.[15] Special ones are: {`VOCAB`}, matching any vocabulary type annotation, and {`EXPR`}, matching any expression type annotation.
- {*PropertyTest*}, where *PropertyTest* is a specific test on a certain property of the lexical annotation. The possible tests are listed overleaf.

[15]Later, we may decide that they must begin with an upper case letter, but otherwise match would be case-insensitive.

`IsStop`	: matches if word is a stop word
`IsEmpty`	: matches if word is an empty word
`IsEmptyInterior`	: matches if word is an empty interior word
`IsEmptyInitial`	: matches if word is an empty initial word
`IsEmptyNonFinal`	: matches if word is an empty non-final word
`IsEmptyFinal`	: matches if word is an empty final word
`IsVariant`	: matches if word is a variant of some lemma form
`IsInLex`	: matches if word occurs in the lexicon
`IsCompound`	: matches if word is a compound
`IsLemma`	: matches if word is its own lemma form

As an example, a small grammar which picks up all vocabulary item annotations created by TEXTRACT's vocabulary plugins (such as Abbreviator, Expressions, Nominator, Terminator; see (Neff et al., 2003)) can be written as follows:

```
vocabulary = <E>/[vocab:lexical . {VOCAB} . <E>/]vocab ;
expression = <E>/[xprsn:lexical . {EXPR}  . <E>/]xprsn ;

category   = :vocabulary |
             :expression
             ;
```

4.6 Tests over morphosyntactic features

A broad range of grammars can be developed by writing patterns which exploit morphosyntactic properties of words. Typically, and minimally, such properties include part of speech and inflectional information. In a pipelined system, such as TEXTRACT, it is the responsibility of a lexical lookup module, possibly followed by a part-of-speech tagger, to derive such information and place it on lexical items (typically, words).

TEXTRACT currently uses a tagger trained over a (slightly expanded) Penn tag set (Santorini, 1995); also see (Neff et al., 2003, appendix). This maps directly onto the set of TFST symbols for matching against part of speech and inflectional morphology. We have already seen some some examples of patterns which appeal to this information, in the sample garmmars earlier for simple noun phrases; their general form is {`Morpho_Syn_Test`}.

In TALENT 5.1, the set of *Morpho_Syn_Test*s is defined externally as a list in `TalentData/FstLexSymbols`. The file maps (typically) atomic tag symbols—from the Penn tag set in this instance, but in principle from any tag set[16]—to *UniLex* style of morphosyntactic feature cluster.

The current set of definitions is given below. Evert tag, or tag feature cluster, can be specified as a *Morpho_Syn_Test*}. At present, all tags/ feature clusters are treated as atomic; eventually, the notation will be modified so feature clusters can be specified over an open vocabulary of tag and feature names.

```
AUX             _auxiliary _infinitive
AUXD            _auxiliary _past
AUXG            _auxiliary _present_participle
AUXN            _auxiliary _past_participle
AUXP            _auxiliary _present _vbp
AUXZ            _auxiliary _present _3p _singular

VB+AUX          _auxiliary
VB+AUX:D        _auxiliary _past
VB+AUX:G        _auxiliary _present_participle
VB+AUX:N        _auxiliary _past_participle
VB+AUX:P        _auxiliary _present _vbp
VB+AUX:Z        _auxiliary _present _3p _singular
VB+AUX:I        _auxiliary _infinitive

VB-AUX          _nonmodal_nonaux
VB-AUX:D        _nonmodal_nonaux _past
VB-AUX:G        _nonmodal_nonaux _present_participle
VB-AUX:N        _nonmodal_nonaux _past_participle
VB-AUX:P        _nonmodal_nonaux _present _vbp
VB-AUX:Z        _nonmodal_nonaux _present _3p _singular
VB-AUX:I        _nonmodal_nonaux _infinitive

VB              _nonmodal
VBD             _nonmodal _past
VBG             _nonmodal _present_participle
VBN             _nonmodal _past_participle
VBP             _nonmodal _present _vbp
VBZ             _nonmodal _present _3p _singular
VBI             _nonmodal _infinitive

VB              _nonmodal
VB:D            _nonmodal _past
```

[16]Other, or different, symbols may be defined by a grammar writer, as long as they use the same set of bit definitions defined in `TalentData/Emorph.txt`.

```
VB:G            _nonmodal _present_participle
VB:N            _nonmodal _past_participle
VB:P            _nonmodal _present _vbp
VB:Z            _nonmodal _present _3p _singular
VB:I            _nonmodal _infinitive

CC              _coordinating_conjunction
CS              _subordinating_conjunction
CD              _cardinal_number
DT              _ordinary_determiner
EX              _existential
FW              _unique _foreign
IN              _preposition
JJ              _ordinary_adjective
JJR             _ordinary_adjective _comparative
JJS             _ordinary_adjective _superlative
JJ-C-S          _ordinary_adjective !_comparative !_superlative

LS              _unique _list
MD              _modal
NN              _common_noun
NNS             _common_noun _plural
NP              _proper_noun _singular
NPS             _proper_noun _plural
PDT             _predeterminer
POS             _unique _possessive
PP              _personal_pronoun _other_cases
PP$             _personal_pronoun _possessive
RB              _ordinary_adverb
RBR             _ordinary_adverb _comparative
RBS             _ordinary_adverb _superlative
RP              _particle
SYM             _unique _symbol
TO              _to
UH              _interjection
WDT             _wh_determiner
WP              _relative_pronoun _other_cases
WP$             _relative_pronoun _possessive
WRB             _interrogative_adverb
```

4.7 Tests over syntactic annotations

Syntactic family annotations can be examined with symbols which use the `[`*`Label`*`]` notation. In the current TEXTRACT system, the only way to post syntactic annotations is through a TFST grammar. Thus this category of tests is typically used in multi-level grammar cascades, and any syntactic annotation posted by an earlier level can be tested for by checking for equality between the labels: if our simple grammmar (reproduced below) posts an `"NP"` annotation, any grammar applied after it can reasonably specify, in a rule, `[NP]`—assuming, of course, that no deletions of `[NP]`s have happened in the mean time.

```
det      = {DT}|<E>    ;                # optional determiner
premod   = {JJ}*       ;                # zero or more pre-mod adjectives
noun     = {NN}|{NNS}  ;                # the head nominal

np       = <E>/[NP:phrasal .            # post a new anno from here...
              :det . :premod . :noun .  # covering the np match,
           <E>/]NP:+simple              # to here; label it "NP"
                                        # set a binary feature simple=true
           ;
```

In addition to checking the label of a syntactic annotation, it is also possible to test a feature: for instance, the following pattern will find all, and only, `"NP"` annotations posted by the earlier grammar[17].

```
simpleNP = <E>/[simpleNP:phrasal . [NP:+simple]/- . <E>/]simpleNP ;
```

’Out of the box’, there are the following syntactic annotation labels:[18]

```
AdjP,
NP, NPP, PNP, NPS, CNP, NPList,
VG, PVG,
PP,
MC, TC, WHP, SC,
SUB, PSUB, OBJ
```

A rule can also test for any label that has been posted (and thus defined) by the current, or lower level (i.e. preceding this grammar in a cascade) FST’s.

[17]Check...
[18]This needs a more elaborate write-up

4.8 Regular expressions over lexical annotation strings

A symbol enclosed in double angle brackets `<<`*`RegEx`*`>>` specifies a character-based regular expression match over the string image of a lexical annotation. TFST's regular expressions follow closely GNU's regular expression specification, found, for instance, in the *man* entry for the GNU *grep* command.[19]

A regular expression defines a set of strings, according to certain composition and interpretation rules. Fundamentally, the simplest regular expression matches a single character. All alpha-numeric characters, and most other (graphical) characters too, denote regular expressions that match themselves. The exceptions are a few meta-characters, which signal special matching behaviour to the regular expression interpreter. All meta-characters, of course, can be escaped, with a *RegEx* quote character (a backslash). Note that the regular expression quote character is not the same as the escape character used in certain TFst symbols (4.1, 4.5). Also, note that while the same character is used for both, the symbol-level escape character can be redefined (via the TFst executor `.Ini` file; see 5.2. Finally, note that another symbol meta-character, `/`, is escaped inside of a regular expression by writing `//`.

The period `(.)` matches any single character. Common composition operators, for building larger expression out of smaller ones, are *concatenation, alternation*, and *repetition*. Concatenation is implied by joining two regular expressions: for instance, given the expression `a`, `a.` will match any two-character sequence beginning with `a`. Alternation is signalled by `|`: e.g. `before|after` will match if the target string contains either of the substrings *before* or `after`. Repetition comes in different flavours, and is specified using several meta-characters. If `re` is a regular expression, then the different ways of specifying repeated occurrences of `re` in the match are described below.

```
re*        =>  re  will be matched zero or more times;
re?        =>  re  will be matched zero or one time;
re+        =>  re  will be matched one or more times;
re{N}      =>  re  will be matched exactly N times;
re{N,}     =>  re  will be matched N or more times;
re{M,N}    =>  re  will be matched at least M times, but not more than N times.
```

[19]The next major release of]sc TFst will be based on ICU's native *RegEx* package; see `http://oss.software.ibm.com/icu/userguide/regexp.html`. Grammar writers are encouraged to consider portability of the regular expressions they write.

Repetition takes precedence over concatenation, which in turn takes precedence over alternation. A whole sub-expression may be enclosed in parentheses to override these precedence rules.

The `.` meta-character implicitly defines a set (of all characters) which the regular expression `.` will match. It is possible to define narrower sets and thus constrain the match. A *bracket expression* lists a number of characters, enclosed in `[` and `]`; it matches any single character in the list. If the list starts with a caret (`^`), a bracket expresison matches any single character not in the list. Metacharacters typically lose their special meaning inside bracket expressions. To include a literal `]` it should be placed first in the list; to include a literal `^` it should be placed anywhere but first; to include a literal `-` it should be placed last.

Additionally, two meta-characters act as *anchors*: the caret (`^`) and the dollar sign (`$`) respectively match the empty string at the beginning and the end of a the string being matched. Anchoring at the beginning or end of a word is specified with `\<` and `\>`: `\<`*re* succeeds only if *re* is matched at the beginning of a word, and *re*`\>` succeeds only if *re* is matched at the end of a word. As a generalisation, `\b` matches the empty string at a word boundary, and `\B` matches the empty string when it is not at a word boundary. For a regular expression like `^`*re*`$` to succeed, it must match the entire string.

In addition to explicitly specifying precedence, parentheses are also used to group parts of regular expressions, so that an operator can be made to apply to the whole group. Another function supported by enclosing parts of regular expressions in parentheses is that of *back-referencing*. Specifying `\`n, where n ranges from 0 to 9, requests for a match against the substring previously matched by the n-th parenthesised sub-expression of the regular expression: thus `(abra)[abcd]{3}\0` will match *abracadabra*.

4.9 Symbols for transduction operations

As discussed earlier (section 3.3), transductions upon successful matches ofer a way for the grammar writer to interact directly with the annotation repository or the vocabulary. The different categories of transduction—create new annotations, manipulate features on annotations, and delete existing annotations—are described below.

4.9.1 Creating annotations

Matching labels on a pair of symbols which mark the span of a match are the notational device for posting new annotations. If the label is one of the pre-registered syntactic types defined and used by the shallow parser (see 4.7), or if the label is that of an existing type, specifying it, and it alone, is sufficient for a well-defined transduction: `/[`*`Label`* `....` `/]`*`Label`*.

The first time that a label is referenced, what needs to happen is its dynamic definition, and registration, by the system. Annotations do not exist in isolation, but always in relation to each other. At the very least, the annotations within the same family instantiate a hierarchical relationship which allows them to possibly share (inherit0 some properties, and defines a 'priority' which controls the order in which ambiguous iterators present annotations with co-terminus spans to a client (be it a program, or a grammar).

For this purpose, a slightly more explicit construct is used: `/[`*`Label:Type`* `....` `/]`*`Label`*. This works much as the earlier pair, but *Label* is a new, dynamically created type. *Type* is the type that is to be the parent of the new label. If a label with that name already is defined, then the new type is ignored. The new label is registered with the system regardless of whether or not any new annotations with that label are actually created; the definition does not persist across TEXTRACT invocations.

We can thus return to our example grammar. Since the label posted by the final underlying pattern is not *a priori* registered with the system, we offer a clue as to what its position is within the syntactic annotations type hierarchy.

```
det      = {DT}|<E>    ;               # optional determiner
premod   = {JJ}*       ;               # zero or more pre-mod adjectives
noun     = {NN}|{NNS}  ;               # the head nominal

np       = <E>/[simpleNP:phrasal .     # post a new anno from here;
                                       # declare its position in the
                                       # type hierarchy;
              :det . :premod . :noun . # covering the np match,
           <E>/]simpleNP               # to here; label it "simpleNP"
           ;
```

4.9.2 Setting properties on annotations

Through earlier examples, we have seen that properties (features) on annotations can be either binary flags (with values of *true* or *false*), or they can hold integer values. The general syntax for setting a property is +*feature*, or -*feature*, or *feature*=*value*; these set *feature* to be *true*, *false*, or equal to *value*.

Properties can be set on syntactic family annotations, or on document structure family annotations. In the former case, the specification is placed on the transduction which posts the annotation. In the later case, as document structure annotations cannot be posted from a grammar, the property setting specification is on a transduction associated with a separately matching symbol. The following shows the full complement of property setting expressions.

- `/[`*`Label`*` ... /]`*`Label`*`:+`*`feature`*, or `/[`*`Label`*` ... /]`*`Label`*`:-`*`feature`*.
- `/[`*`Label`*` ... /]`*`Label`*`:`*`feature`*`=`*`value`*.
- `/`*`DocStructAnno`*`::+`*`feature`*, or `/`*`DocStructAnno`*`::-`*`feature`*, where *DocStructAnno* can be one of `Document`, `Section`, `Paragraph`, or `Sentence`.[20]
- `/`*`DocStructAnno`*`::`*`feature`*`=`*`value`*.

4.9.3 Adding to the vocabulary

A grammar can directly add items to TEXTRACT's vocabulary (see (Neff, Byrd, and Boguraev, 2003) for discussion on vocabulary and vocabulary items). Procedurally, an annotation needs to be posted over a span of text; adding the vocabulary item is on the associated transduction, and amounts to adding the text span to the vocabulary, as a given type. This category of transduction comes in three flavours.

- `/[vocabulary ... /]vocabulary`: creates an annotation over the span covered by the pair, as type VOCAB. The text span is also added to the vocabulary as type VOCAB.
- `/[vocabulary:`*`Label`*` ... /]vocabulary:`*`Label`*: same as above, but the new annotation is of type *Label*, and the text span added to the vocabulary is also of type *Label*. The type is known (registered with) the system.
- `/[vocabulary:`*`Label`*`:`*`Type`*` ... /]vocabulary:`*`Label`*: same as the last construct above, except *Label* is a new type label to be created. *Type* is the type that is to be the parent of the new label. If a label with that name already is defined, then the new type is ignored. The new label is registered with

[20]What about `Segment`?

the system regardless of whether or not any new annotations with that label are actually created.

4.9.4 Deleting annotations

Deleting annotations, by means of a transduction operator `/-`, has already been discussed in 3.3.

4.10 Dual scanning regimes

One of the advantages of TFST is its ability to treat an annotation—any annotation—as an atomic object against which a match can be specified. Usually, one or more tests on a cluster of features associated with the annotation itself (family, type, label, underlying string, morpho-syntactic properties, and so forth) is sufficient to specify a match; the notational devices described so far are designed to specify such constraints to the TFST executor.

Occasionally, however, it would be convenient to test for a specific annotation, and then examine its 'inner contour'—to see whether a sequence of annotations that the covering one spans conforms to certain configurational constraints. In effect, what we would need to communicate to the executor is a complex command: test for an annotation, specified by any of the means described so far in this section; upon a successful match, *descend* into this annotation; test whether a given pattern matches *exactly* the sequence of lower annotations covered by the higher match (making sure that the right route through the annotations sub-lattice gets chosen; cf. section 2.1); if the sub-match succeeds, pop back at a point suitable for registering overall success for the higher level match; succeed, and then proceed.

The notational device used for such an operation employs a pair of *push* and *pop* operators, available as meta-characters on symbols.[21] Conceptually, if S is a symbol matching an annotation which could be covering other annotations, S_ would signal the 'descend into' operation. At that point, the full complement of TFST symbols would be available to specify a pattern we would want to match at the lower level; the match needs to completely consume the sequence of sub-annotations. At the end of the lower level scan, the meta-character ^ requests a pop to the prior (higher) level of analysis, restoring all context as appropriate.

[21]Why aren't we using a `subFSt` device, after all?

Typically, this machinery is used with analysis of the inner contour of previously created syntactic family annotations. The example below illustrates a grammar fragment which determines whether a verb group (analysed as `VG` by an earlier grammar in a cascade; see section 3.4) contains a negative marker. Note that the *push*/slash*pop* metacharacters are folded inside of the symbol brackets. The expression specifies that to qualify as a `negativeVG`, the verb group has to contain a `not` string (with an adverbial reading, for safe measure) anywhere inside it: all of *"does not know"*, *"would not have been known"*, *"is not"*, *"not reporting"*, for instance, will match the pattern.

```
negativeVG  = [VG_] . <SWORD>* . ( not & {RB} ) . <SWORD>* . [VG^] ;
```

The grammar compiler (see section 5.1) treats `[VG^]` as synonymous with `<E>/^`; `[VG^]` is more indicative of the pairing, and dual effect, of the *push*/*pop* operations. Strictly speaking, `^` in this context is a transduction operator, with a special meaning to the TFST executor.[22] Also note that it is possible to place these operators on consuming symbols; thus `<<!^[0-9]+$^>>` is a valid symbol, specifying that upon a successful match (for any lexical annotation that is not a sequence of digits) a *pop* to a higher level should reset the scanning regme to that higher level of analysis.

[22]This is somewhat messy: syntactically, the *caret* meta-character does look like a transduction; it does not, however, affect the annotation repository. Also, if the *caret* looks like a transduction, so should the *underscore* meta-character; maybe it will, for the next release...

5 Putting it all together

Grammars are written and maintained as text files; by convention, these are defined with a `.cfg` extension. Before TEXTRACT's TFST executor can load and apply a grammar against an annotation repository, it needs to be compiled to FST format. Furthermore, the executor needs to be configured for interpreting the FST.

5.1 Grammar compiler

A `.fst` file encodes the topology of a single finite state automaton, which encapsulates, in a somewhat optimised form, all the rule patterns defined by the grammar.

The FST compiler is a stand-alone executable, provided as a command-line tool, `fstcmp`. It is, at present, a 'bare bones' utility, which has a simple invocation, and offers very little in terms of diagnostics if there is a problem with the source grammar.

Assuming that our example grammar (33 is in a file `np.cfg`—remember that the base name of the file must be the same as the root rule of the grammar—the compiler is invoked as follows.

```
bkb @ .../doc > fstcmp np.cfg
 > "np.fst" : 7 symbols, 11 transitions, 8 states (down from 33)
bkb @ .../doc >
```

The FST compiler reports the name of the file with the compiled automaton, and some rudimentary statistics concerning the compilation. In essence, if the output from a compiler run looks like the fragment above, the compilation has been successful.

If the grammar is deficient (syntactically) in some way, compilation fails. Error recovery and self-diagnostics currently leave a lot to be desired. Most likely the author of a faulty grammar will see one of the following two messages.

```
bkb @ .../doc > fstcmp np.cfg

 > Unrecognized character: < [ ’<’ / x3c ]

bkb @ .../doc >
```

This is indicative of failure of the lexical scanner which tokenises the grammar source into TFst symbols. The offending character (straying from an acceptable symbol syntax) is shown in graphical and hex formats.

```
bkb @ .../doc > fstcmp np.cfg

 parse error:

 > priorCompilationFor: no definition for ’np’ found

bkb @ .../doc > fstcmp np.cfg
```

This is indicative of compiler failure in parsing a pattern rule. The faulty rule is not necessrily the one reported. Check for missing concatenation operators (`.`), or terminating semicolons)`;`) , mismatched parentheses, or misspelt references to names of earlier defined rules. Also, check that the root rule is named identical to the base name of the `.cfg` file.

5.2 Configuring the TFst executor

In line with TEXTRACT’s mechanism for externally defined (re-)configuration via a `.Ini` file, the TFST executor’s operation is controlled by means of setting its run-time parameters in an `[nfstxeq]` stanza. The full set of parameters configurable in this way is shown below; this is an extract from a valid `.Ini` file; note that some paths will need to be modified to reflect local directory structure and file organisation..

```
[nfstxeq]
fst_pathname      = C:\Emma\Projects\TFst
fst_filename      = markYears.fst pickYears.fst
fst_max_level     = 2
Nominator         = VdictAuthority(C:\Frasier\nomRef\nomAuth.vdict)
fst_escape_char   = \
definitions_file  = C:\Frasier\cvsTextract\TalentData\FstLexSymbols.txt
```

Information about the `.fst` file to be loaded and applied by the TFST executor is separately encoded in two parameters: `fst_pathname` and `fst_filename`. This makes it easy to specify a regime of cascading FST grammars (see section 3.4): the names and sequence—from first to last—of `.fst` files comprising the cascade are listed in the `fst_filename` declaration. It is assumed that the files for all the automata in a cascade would share a directory path, and this is what `fst_pathname` specifies.

The extent of the cascade for any particular run is declared by `fst_max_level`. If this is less than the number of files specified in `fst_filename`, `fst_max_level` is honoured, and not all levels of the cascade are activated on this partoicular run. If `fst_max_level` is higher than the number of files specified, then all of them get applied.

The `definitions_file` would not normally be modified by a grammar writer, and would be configured to refer to an external resource which is part of normal TALENT 5.1 distribution. It defines the mapping between a particular part-of-speech tag set and UniLex feature clusters; for more details on this file, see section 4.6.

The `VdictAuthority` declaration registers a specified authority file (pre-compiled in a *VDict* format; see 4.4) with the system, and makes it accessible to grammar rules via the symbolic name `Nominator`. This kind of declaration is only necessary if a grammar in a given TEXTRACT configuration checks properties of lexical annotations against external authority.

Finally, `fst_escape_char` defines the meta-character used for TFST symbol level escape (see section 4.1, 4.5). This is not to be confused with the escape (quote) character for overriding the special meaning of meta-characters in regular expression specifications.

5.2.1 Seeing the results

Since the FS automaton is applied to an annotation repository, and not a character string, and more importantly, since the transduction operators manipulate an annotation repository, it is necessary to invoke a specially configured `dump` application[23]

There are two components to seeing the results of a match. The sc TFst executor has to be not only configured, as described in the previous section, but it also needs to be enabled. This is done by setting the `nfstxeq` parameter in the `[run]` section of the `.Ini` file:

```
[run]
doc_structure        = 1
lex                  = 1
expressions          = 0
abbreviator2         = 0
nominator            = 1
tagger               = 1
yatagger             = 0
nfstxeq              = 1
dump                 = 1
```

Note that for this particular example, a number of plugins, in addition to `nfstxeq`, have been enabled. For any particular configuration of the TFST executor, at the very least `lex` and `doc_structure` are necessary, as part of TEXTRACT's basic services. Beyond that, the grammar(s) may, or may not, require e.g. `nominator` and/abbreviator (for instance, they may need access to vocabulary items) or `tagger` (for patterns with morpho-syntactic match symbols in them.

Note that `dump` is also enabled. The `dump` plugin itself offers a variety of output options and formats, for selective display of all or parts of the anotation repository after any given combination of plugins have been instantiated. For a complete specification of these, see (Neff et al., 2003). For seeing the results of TFST matching, the following is sufficient.

[23]A TALENT 5.1 application is not dissimilar to a plugin, at least in that it subscribes to the same API as a plugin does, and it has the same access to the annotation repository, lexical cache and vocabulary as plugins do. Applications usually run after all the process plugins have finished; the `dump` application certainly runs last, for obvious reasons.

```
[dump]
text                    = 1
format                  = ascii_parse
```

In essence, this invokes a special print option suitable for display of shallow parse structures. TEXTRACT's shallow parser is configured as a multi-grammar cascade, and implemented entirely as a TFST application. Since the only annotation type a TFST grammar can post is within the syntactic family (see 4.9), the output capability of a shallow parser `[dump]` option is fully adequate for displaying the annotations created after successful matches.

As a reminder, the reader is cautioned that a successful match, by itself, would be invisible outside of TFST, and that it is imperative to post an annotation over the matching span, which would be the tangible result from the match, remaining in the AR for processes behind TFST to exploit.

Having compiled one or more `.fst` files, and having prepared a configuration `.Ini` file, the system gets invoked as a command-line executable.

```
bkb @ .../talent > Talent.exe -config tfstRun.ini para.txt | less
```

Assuming that para.txt is a file containing some text to be analysed by our grammar of the preceding sections (`np.cfg`), the following is indicative of the kind of output produced.

```
Throughout
[simpleNP the 1980s simpleNP]
,
the
Soviet
Union
threw
almost
[simpleNP every weapon simpleNP]
```

```
it
had
,
short
of
[simpleNP nuclear bombs simpleNP]
,
at
[simpleNP the Afghan camps simpleNP]
attacked
by
the
United
States
[simpleNP last week simpleNP]
.
```

The text fragments paired with `[simpleNP ... simpleNP]` labels indicate the spans of successful matches (with respect to this grammar; see p. 4.9.1) over which a `"simpleNP"` annotation has been posted.[24]

One possible variation in the sample configuration above is to specify, instead of `ascii_parse` value for the `format` parameter, `tag_parse`. This produces similar output to the one above, but the tagger analyses are displayed inline; this makes it easier to analyse the input-output behaviour of a grammar which exploits, in particular, morpho-syntactic information in its rule patterns.

[24]For those who wonder why *"the Soviet Union"* and *"the United States"* are not marked as noun phrases, the answer is because the grammar has not picked them up as such: for this particular run, a named entity extractor has been enabled prior to TFST, and since the two phrases in question are found to be named entities, they are covered by lexical annotations of `vocabulary` type. This is not something thatour simple grammar is sensitive to.

5.3 Sample grammars

```
# this grammar assumes that Nominator has run.
# simply match on the annotation types Nominator has already posted.
#
# note:
# Abbreviator(2) annotations can similarly be picked up by "{UABBR}"
#

  person     = <E>/[Person:lexical .
                 ( {PERSON} | {PERSON?} ) .
               <E>/]Person ;

  place      = <E>/[Place:lexical  .
                 ( {PLACE}  | {PLACE?}  ) .
               <E>/]Place  ;

  org        = <E>/[Org:lexical    . {ORG}    . <E>/]Org    ;

  uname      = <E>/[UName:lexical  . {UNAME}  . <E>/]UName  ;

# _________________________________________________________________

  nominator  = :person |
               :place |
               :org |
               :uname
               ;
```

```
# this grammar assumes that any combination of plugins that detect
# and post vocabulary item annnotations have been run
#

  vocabulary = <E>/[vocab:lexical . {VOCAB} . <E>/]vocab ;
  expression = <E>/[xprsn:lexical . {EXPR}  . <E>/]xprsn ;

  category   = :vocabulary |
               :expression
               ;
```

```
# an example of regular expression match, over date/time tokens like
#
# 3-13-2002 or 06/15/1998 or May-13-2003,
# and times like 12:00am or 12:00p.m.
#

  mDate      =
   ( <E>/[mDate:lexical .

       ( <<^[01]?[0-9][-//][1-3]?[0-9][-//][12][0-9]{3}$>> ) |
       ( <<(Jan\.?)|(Feb\.)|(May)[-][1-3]?[0-9][-][12][0-9]{3}>> ) ) .

     <E>/]mDate ) |

   ( <E>/[mTime:lexical .

       ( <<[01]?[0-9]:[0-5][0-9][apm.]+>> ) .

     <E>/]mTime )
   ;
```

```
# example of a two-level cascade; implements 'over-generate and filter'
# strategy: level 1 identifies 'year' strings, such as "1999", "2003',
# and "late 1990s", and sets features to remember the shape of the
# phrase; level 2 picks only those expressions satisfying some
# criterion (feature) of the original specification.


# Grammar 1 : simple grammar to mark year-based denotations in text
#
# illustrates
#
#   = regular expressions match over lexical tokens
#   = lexical string match
#   = posting novel types
#   = use of variables to mark properties of annotations
#   = Fst cascading
#

 concreteYear   = <<^[12][0-9]{3}$>> ;

 tempPtr        = early | middle | late ;

 vagueYear      = the . <E>|:tempPtr . <<^[12][0-9][0-9][0-9]s$>> ;

 markYears      = <E>/[Year:unknownsyn .
                  ( ( :concreteYear . <E>/]Year:+concrete ) |
                    ( :vagueYear    . <E>/]Year:-concrete )
                  )
                  ;


#_____________________________________________________________________

# Grammar 2 : simple grammar to pick some annos, posted by Grammar 1,
# on the basis of a property value.
#
# illustrates
#
#   = matching over annotations,
#   = testing properties,
#   = deleting an annotation,
#   = Fst cascading
#

 pickYears      = [Year:-concrete]/- |

                  ( <E>/[YearPoint:phrasal .
                       [Year:+concrete]/- .
                    <E>/]YearPoint )
                  ;
```

```
#  a slightly more comprehensive (albeit still far from complete)
#  noun phrase grammar
#

#  __________________________________________________________________

   detP         = ( {PDT}|{DT}|{CD} ) . ( {PDT}|{DT}|{CD} ) * ;

   adjP         = ( {RB}|<E> ) . {JJ} . {JJ}* ;

#  __________________________________________________________________

   preMod       = ( {NN}|{NP}|{NPS} ) |
                  {CD} |
                  ( :adjP . ( <COMMA>|{CC}|<E> ) ) ;

   possMod      = ( <E>/[NPS . {PP$} . <E>/]NPS )  ;

#  __________________________________________________________________

   simpleNP     =   ( :detP | <E> ) .
                      :preMod* .
                      {NN} .
                    ( {CD} | <E> ) ;

   properNP     =   ( :detP | <E> ) .
                      :preMod* .
                    ( {NP} | {NPS} ) .
                    ( {CD} | <E> ) ;

   elidedNP     =   :detP . ( :adjP | {CD} ) ;

   possNP       =   <E>/[PNP .
                      :possMod .
                      <E>/[NP . :simpleNP . <E>/]NP .
                    <E>/]PNP ;

#  __________________________________________________________________

   np           =   ( <E>/[NP .
                        ( :simpleNP | {PP} | :elidedNP ) .
                      <E>/]NP )
                    |
                    ( <E>/[NP .
                        ( :properNP ) .
                      <E>/]NP:+propNHead )
                    ;
```

```
#  Grammar 1 : mark the boundaries of a wide range of verb groups
#

   InfV         =  {TO} . {RB}* . {VB+AUX}|<E> . {RB}* . {VB} . {VB}* ;


   VbKernel     =  ( {VB}|{VB+AUX} ) . ( {VB}|{VB+AUX} )* ;

   GenericV     =  {MD}* . {RB}* . {VB+AUX}* . {RB}* . :VbKernel ;


   VrbGroup     =  <E>/[VG .
                     ( :InfV | :GenericV ) . {RB}* .
                   <E>/]VG
                   ;


#_____________________________________________________________________

#  Grammar 2 : 'descends' into previously marked verb groups, in order
#  to identify certain (configurational) features

   AuxTensed    = {VB+AUX:P} | {VB+AUX:Z} | {VB+AUX:D} ;

   VrbTensed    = {VB-AUX:P} | {VB-AUX:Z} | {VB-AUX:D} ;
   VrbUnTensed  = {VB-AUX:I} ;

   VrbGrpModal  = (  [VG_] .
                     {MD} .
                     {RB}* .
                     ( ( {VB-AUX:I} ) | ( {VB+AUX:I} . {VB-AUX:G} ) ) .
                     {RB}* .
                     <E>/^ )
                  ;


   VrbGrpTensed = (  [VG_] .
                       {RB}* .
                       ( ( :AuxTensed . {RB}* . ( {VB:G} | {VB:N} ) ) |
                         ( {VB+AUX} . {RB}* .
                           :VrbUnTensed | :VrbTensed ) |
                         ( :VrbTensed . <E>|{VB} )
                       ) .
                       {RB}* .
                     [VG^]
                  ) | :VrbGrpModal
                  ;
```

```
#  a very simple (and simple-minded) grammar identifying noun phrases
#  with an internal named entity component; sample output below...

  Noun             = {NN} | {NNS} ;
  Adj              = {JJ}         ;

  NamedComponent   = <UPPER> | <LUPPER> ;

  namedNPs         = <!Sentence::IsWordFirst> &   # left context...

                     <E>/[namedNP:phrasal .

                       ( {DT}|<E> ) .
                          :NamedComponent . :NamedComponent* .
                          ( ( :Noun | :Adj ) . Noun* ) * .

                     <E>/]namedNP ;


# ____________________________________________________________________
#
# [namedNP the Soviet Union namedNP]
# [namedNP the Afghan camps namedNP]
# [namedNP the United States namedNP]
# [namedNP Afghanistan namedNP]
# [namedNP Khost namedNP]
# [namedNP Scud missiles namedNP]
# [namedNP Soviet commander namedNP]
# [namedNP Afghanistan namedNP]
# [namedNP Lt. Gen. Boris Gromov namedNP]
# [namedNP the Afghan holy warriors namedNP]
# [namedNP the Soviets namedNP]
# [namedNP Thursday namedNP]
# [namedNP the Soviets namedNP]
# [namedNP Paktia province namedNP]
# [namedNP Afghan resistance leaders namedNP]
# [namedNP Soviet troops namedNP]
# [namedNP December namedNP]
# [namedNP American intelligence veterans namedNP]
# [namedNP Afghan resistance namedNP]
# [namedNP the United States namedNP]
# [namedNP Saudi Arabia namedNP]
# [namedNP the Saudi exile namedNP]
# [namedNP Osama bin Laden namedNP]
# [namedNP U.S. intelligence official namedNP]
# [namedNP the CIA namedNP]
# [namedNP the Afghan rebels namedNP]
# [namedNP the Soviet-supported garrison town namedNP]
# [namedNP Khost namedNP]
# [namedNP CIA namedNP]
```

6 Interactive grammar development

TEXTRACT is available as a command-line executable. In general, TEXTRACT offers little beyond an architecture and API's, by means of which specific applications can be configured.

The TFSTsystem as one of TEXTRACT's numerous plugins. This makes it a 'first-class' citizen, as far as deployment is concerned of a grammar, or a cascade of grammars, within a larger process pipeline. However, this same architecture is not very well suited for the inherently incremental, experimental, trial-and-error process of developing and tuning a grammar. While the system is streamlined and optimised to initialise a number of plugins and apply them, in sequence, over a number of document, there is no built-in support for iterative re-initialisation and re-invocation of just one plugin (in particular, `nfstxeq`), with very local changes to the run-time parameters settings (possibly just a modified and recompiled grammar).

Furthermore, while a suitable format exists in the `dump` application for displaying the results of a grammar application, it may be hard to relate a strictly sequential dump of annotations, mixed with begin-end match markers, to any diagnostics which might correlate grammar behaviour with pattern specification.

In a larger community of grammar developers, it is unrealistic to expect that all of them would be familiar with programming issues to the extent that they can configure a grammar development scaffold out of raw TEXTRACT API's. Such users need not only insulation from the vagaries and idiosyncracies of the underlying system, but also require a set of tools facilitating grammar development, diagnostics and debugging—activities which, by definition, have no relationship to production-level deployment of 'release' quality grammars.

An experimental tool, under development at present, and thus offered on strictly 'as-is' basis, is an interactive graphical environment for developing TFSTgrammars. This environment, hereafter WTEXTRACT[25], addresses two out of the three fundamental expectations of interactive programming environments.

- rapid *edit-compile-run* cycle,
- multiple perspectives over the output, facilitating diagnostics,
- incremental process stepping and tracing, with source grammar level debugging; this is currently not available, as process control with this kind of granularity

[25]Or occasionally referred to HMTEXTRACT, for reasons too complex to go into here.

is hard[26] to expose to a plugin 'client' through the convenional (and official) document-based process API.

6.1 Overview of wTextract

wTEXTRACT is, in general, a set of graphical widgets which implement custom viewers for an annotation repository which has been populated by one or another of TEXTRACT's plugins. Process pipelines (i.e. plugin chains) can be constructed interactively, and on demand; such chains can be configured (and re-configured) dynamically, without having to wind down completely and restart from the command line; individual plugins can be re-invoked repeatedly, with or without modifying their run-time parameters.

In a mode where a plugin developer wishes to inspect the annotation repository at a any given stage of a plugin pipeline, wTEXTRACT offers the capability of a side-by-side inspection of the systems' internal data objects and repositories.

The TFST capability[27] of wTEXTRACT, in particular, allows an end user—typically a grammar writer—to re-invoke the TFST executor repeatedly over the same document. During such a development cycle, the process settings may be changed at will; at the very minimum, even if the settings remain the same, the system is capable of absorbing changes in the grammar, by reloading the FST's in its cascade after grammar modification and recompilation.

This streamlines, and largely by-passes, the cycle of modifying a `.Ini` file, and restarting a whole TEXTRACT process. In contrast, whenever there is a change pertinent to the run-time environment for the TFST executor, suitable—and minimal—reconfiguration happens behind the scenes. The grammar writer is largely unaware of that; furthermore, they remain within their operational environment, without losing context.

Additionally, the interactive TFST environment offers a variety of different ways to see the results of a grammar application. Fundamentally, wTEXTRACT regards the set of matches against a grammar (or a cascade of grammars) as a concordance, and variations of how the concordance is organised and presented are under grammar writer's control. The concordance correlates the successful matches amongst

[26]But not impossible; future releases may address this issue.

[27]The original prototype of a wTEXTRACT system was developed and built by B. Boguraev, in Borland C++ Builder; a rationalised and enhanced version of the TFST capability was implemented in Microsoft Visual Studio .NET by Son Bao Pham.

themselves; this makes it possible to notice general patterns in what has, and has not, been picked up by the grammar(s). At the same time, by maintaining synchronicity between the concordance view of a match, and displaying this match within it original document context, it is possible to examine the contexts for successful and failed matches and form hypotheses for the grammar behaviour—be it over-, or under-generation.

Any modifications to a grammar, in response to, or exploration of, such hypotheses will be immediately absorbed by the environment. And, while native grammar editing and compilation is not currently supported[28], a more useful capability is available, for 'one-off', throwaway, pattern specification and testing.

The broad WTEXTRACT behaviour allows for multiple document files to be opened, and at various stages of plugin pipeline execution and inspection; it is not, however, allowed to have multiple analyses (by the same plugin) of the same document. The exception to that rule is the TFST subsystem within WTEXTRACT, where multiple invocations of the same plugin (`nfstxeq`), differing only in the FS automaton applied over the document, can co-exists side by side: thus it is posisble to compare different sets of outputs, derived from different grammar sources.

6.2 Elements of the TFst subsystem interface

This section offers a quick tour of the basic visual elements of the TFST analysis portion of the WTEXTRACT system. The emphasis is on what can be done within a task of developing TFST gramars; other ares of the WTEXTRACT interface are either disabled in the pre-release, or not fully functional (at best). *Caveat Emptor!*[29]

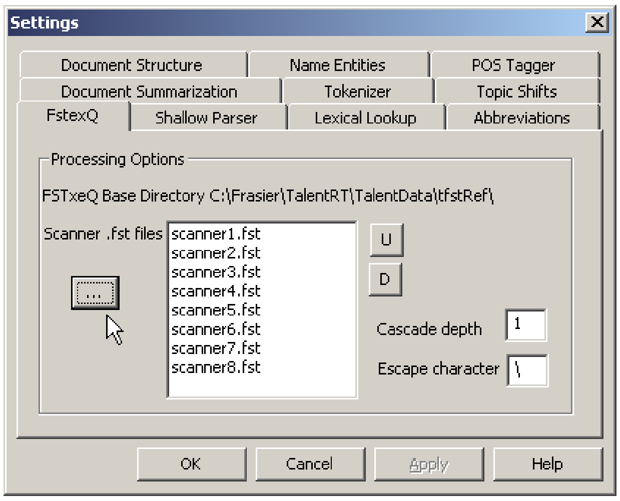
Primary entry point is via the PLUGIN menu, invoking FSTEXEQ submenu. Prior to that,we need a document file, and a working configuration of the TFST executor.

[28] Strictly speaking, it is supported, but not enabled.

[29] Important footnote: Best results are obtained with pure text (`.txt`) files, preferably saved as `"raw text"`. The FILE⇒OPEN menu brings a document file into the workspace. The document window which opens and display the file is an instance of Microsoft's web browser[30]; so if the fonts look odd, or large, this means that the VIEW⇒TEXT SIZE settings for your Internet Explorer settings is too large. Note that most of the display functionality described below is achieved by dynamically—i.e. in response to the user selecting menu items and clicking on buttons—generating highly adorned DHTML, behind the covers, and submitting that to an encapsulated web browser for (native) rendering. One consequence of this approach is that long documents result in slow re-generation of the DHTML source for each pane (think of all those strings...). *Thus, for development purposes, grammar writers are advised to keep their documents not too long, and in* `"raw text"` *encoding.*

6.2.1 Configuration

The EDIT⇒SETTINGS⇒PLUGINS menu brings up a multu-tab panel for setting the operational parameters of individual plugins. By default, the `FstexQ` tab is active; the parameters it allows manipulating map, naturally, to the parameters in an `.Ini` file configuring `[nfstxeq]`. By default, the settings are those for the shallow parser. The directory button `(...)` selects the FST cascade directory; a standard file picking interface identifies the `.fst` files. Multiple file selection is allowed (with `Ctrl` key); the `Up` and `Down` buttons order the cascade files in sequence, as desired. The cascade depth should be set to emulate `fst_max_level` (see section 5.2).



FSTxeQ Base Directory C:\Frasier\TalentRT\TalentData\tfstRef\

Scanner .fst files: scanner1.fst, scanner2.fst, scanner3.fst, scanner4.fst, scanner5.fst, scanner6.fst, scanner7.fst, scanner8.fst

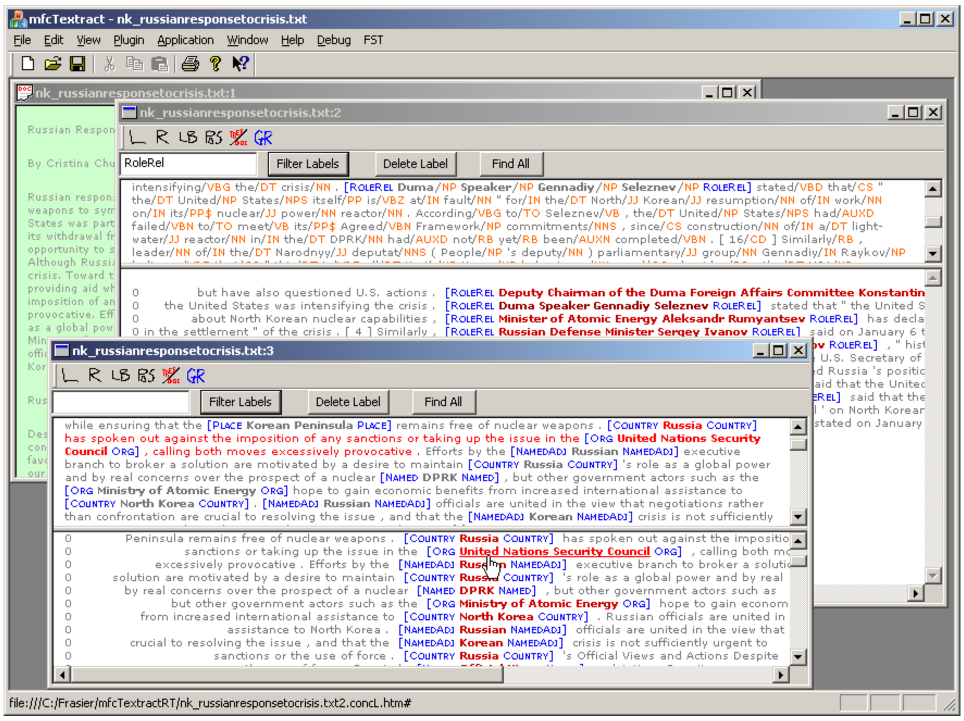
Cascade depth: 1

Escape character: \

With a document file open, and the TFST executor configured, the grammar cascade can be invoked by PLUGIN⇒FSTEXEQ menu. The results are displayed in a new window, with two panes: at the bottom is a concordance of all the matches found by the grammar(s), at the top is an inline display of the same (all) matches overlayed onto the original document source. Strictly speaking, the display shows the annotations that have been added to the annotation repository, following successful matches, during the cascade invocation (see 3.3).

The two panes are synchronous: rolling the mouse over a matching phrase in the concordance pane, and holding it there, brings into view into the top pane, and highlights, the sentence in which this particular match has been found. Thus, while the concordance offers an overview of what matched, and what did not, on a grammar-per-document basis, for each matched item it is possible to examine the larger local context of its enclosing sentence.

In the screen shot below, the window at the forefront (....txt:3) is indicative of the view described above.


mfcTextract - nk_russianresponsetocrisis.txt
File Edit View Plugin Application Window Help Debug FST
nk_russianresponsetocrisis.txt:1
nk_russianresponsetocrisis.txt:2
RoleRel
Filter Labels
Delete Label
Find All
nk_russianresponsetocrisis.txt:3
Filter Labels
Delete Label
Find All

```
intensifying/VBG the/DT crisis/NN . [ROLEREL Duma/NP Speaker/NP Gennadiy/NP Seleznev/NP ROLEREL] stated/VBD that/CS "
the/DT United/NP States/NPS itself/PP is/VBZ at/IN fault/NN " for/IN the/DT North/JJ Korean/JJ resumption/NN of/IN work/NN
on/IN its/PP$ nuclear/JJ power/NN reactor/NN . According/VBG to/TO Seleznev/VB , the/DT United/NP States/NPS had/AUXD
failed/VBN to/TO meet/VB its/PP$ Agreed/VBN Framework/NP commitments/NNS , since/CS construction/NN of/IN a/DT light-
water/JJ reactor/NN in/IN the/DT DPRK/NN had/AUXD not/RB yet/RB been/AUXN completed/VBN . [ 16/CD ] Similarly/RB ,
leader/NN of/IN the/DT Narodnyy/JJ deputat/NNS ( People/NP 's deputy/NN ) parliamentary/JJ group/NN Gennadiy/IN Raykov/NP
```

```
0          but have also questioned U.S. actions .  [ROLEREL Deputy Chairman of the Duma Foreign Affairs Committee Konstantin
0     the United States was intensifying the crisis .  [ROLEREL Duma Speaker Gennadiy Seleznev ROLEREL]  stated that " the United S
0          about North Korean nuclear capabilities ,  [ROLEREL Minister of Atomic Energy Aleksandr Rumyantsev ROLEREL]  has decla
0 in the settlement " of the crisis . [ 4 ] Similarly ,  [ROLEREL Russian Defense Minister Sergey Ivanov ROLEREL]  said on January 6 t
```

```
while ensuring that the [PLACE Korean Peninsula PLACE] remains free of nuclear weapons . [COUNTRY Russia COUNTRY]
has spoken out against the imposition of any sanctions or taking up the issue in the [ORG United Nations Security
Council ORG] , calling both moves excessively provocative . Efforts by the [NAMEDADJ Russian NAMEDADJ] executive
branch to broker a solution are motivated by a desire to maintain [COUNTRY Russia COUNTRY] 's role as a global power
and by real concerns over the prospect of a nuclear [NAMED DPRK NAMED] , but other government actors such as the
[ORG Ministry of Atomic Energy ORG] hope to gain economic benefits from increased international assistance to
[COUNTRY North Korea COUNTRY] . [NAMEDADJ Russian NAMEDADJ] officials are united in the view that negotiations rather
than confrontation are crucial to resolving the issue , and that the [NAMEDADJ Korean NAMEDADJ] crisis is not sufficiently
```

```
0      Peninsula remains free of nuclear weapons .  [COUNTRY Russia COUNTRY]  has spoken out against the impositio
0              sanctions or taking up the issue in the  [ORG United Nations Security Council ORG]  , calling both mc
0           excessively provocative . Efforts by the  [NAMEDADJ Russian NAMEDADJ]  executive branch to broker a soluti
0      solution are motivated by a desire to maintain  [COUNTRY Russia COUNTRY]  's role as a global power and by real
0       by real concerns over the prospect of a nuclear  [NAMED DPRK NAMED]  , but other government actors such as
0             but other government actors such as the  [ORG Ministry of Atomic Energy ORG]  hope to gain econom
0          from increased international assistance to  [COUNTRY North Korea COUNTRY]  . Russian officials are united in
0                       assistance to North Korea .  [NAMEDADJ Russian NAMEDADJ]  officials are united in the view that
0         crucial to resolving the issue , and that the  [NAMEDADJ Korean NAMEDADJ]  crisis is not sufficiently urgent to
0                 sanctions or the use of force .  [COUNTRY Russia COUNTRY]  's Official Views and Actions Despite
```

```
file:///C:/Frasier/mfcTextractRT/nk_russianresponsetocrisis.txt2.concL.htm#
```

When the TFST executor has been invoked, and the focus is on a system window populated by its output results, an additional menu item is added to the main applicaiton menu bar: FST. It offers six commands; these are described below. Each of the commands is also directly available via a button, on a toolbar attached to the TFST executor output window.

6.2.2 FST⇒Concord Left

This acts as a toggle between the default ordering of the concordance list (sequential, as per appearance in the document), and sorted by left-most token in the matching items. It allows examination of left context of similar match items. Shortcut is available via button L.

The `...txt:2` window, just in the background in the figure above, shows a (filtered selection; see 6.2.4, 6.2.10 below) sorted to the left.

6.2.3 FST⇒Concord Right

A toggle between the default ordering of the concordance list (sequential), and sorted by right-most token in the matching items. This allows examination of right context of similar match items. Shortcut is available via button R.

6.2.4 FST⇒Show Label

As discussed earlier (see, in particular, section 4.9.1), TFST grammars currently post only syntactic family annotations. All such annotations have labels, either predefined for TEXTRACT, or dynamically defined by a grammar. This menu item (also available via button LB) is a toggle between hiding (by default) and displaying the labels on the annotations posted for each matching item. The toggle affects the diaply in both panes of the TFST executor output window.

The `...txt:3` window in the foreground in the figure above, shows a concordance list, unsorted, with labels on matches displayed.

6.2.5 FST⇒Show Pos

A majority of TFST grammars will query morphosyntactic features on the lexical elements (usually word tokens) in the document. Some partial support for examining the 'bottom-level' token stream, the lexical lookup results, and the output of the part-of-speech tagging analysis is available via corresponding menu items on the PLUGIN menu: PLUGIN⇒TOKENISE, PLUGIN⇒LEXALYSE, PLUGIN⇒TAG.[31] At best, this is only of partial utility.

The FST⇒SHOW POS command, also available via the POS button, toggles between inline expansion of all tokens in the document pane of the TFST executor output

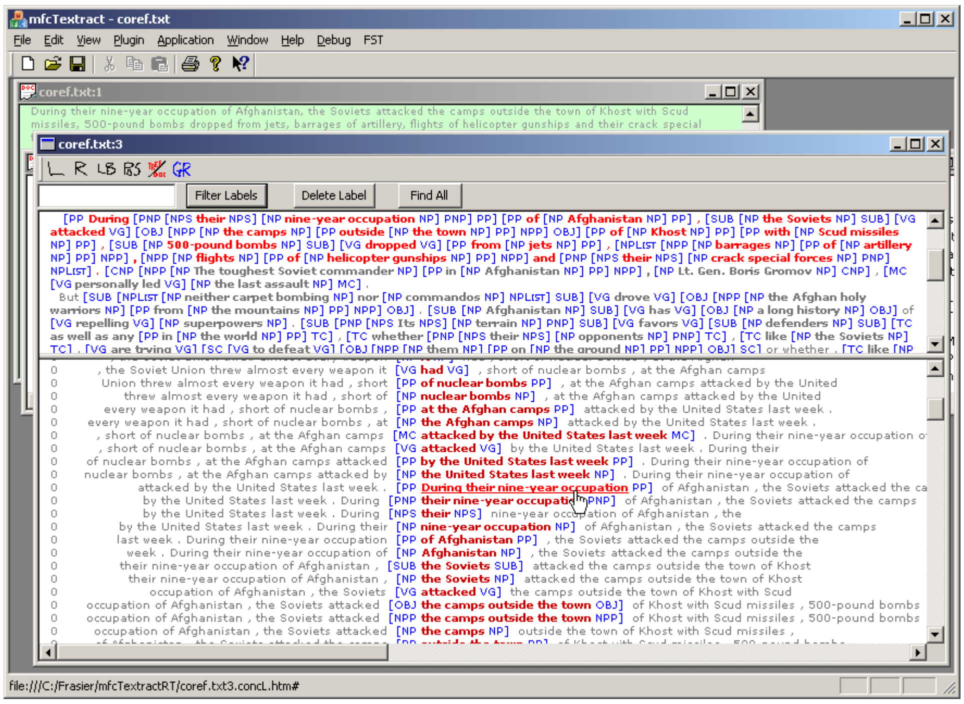
[31]The intent is to show the results of TEXTRACT's base services, insofar as they generate features used extensively by TFST grammar symbols; cf. section 4. However, not all such features are currently visible through the custom displays. Also, note that the PLUGIN menu offers access to the document structure analysis service too, as at least the sentence level annotations are used to constrain the operation of the TFST executor; section 2.1

window to a token/tag format. This makes explicit the underlying POS stream that drives the matching. What makes this feature particularly useful is that given the stochastic nature of TEXTRACT's part-of-speech tagging algorithms, it is impossible to always correctly predict (or, worse, intuit) what part of speech might have been assigned for a token in any particular context.

The top pane of the `...txt:2` window, just in the background in the figure above, shows a display of part-of-speech information for the document stream.

6.2.6 FST⇒Switch View

Some match items reflect composite application of rules, in that complex grammars, and/or multi-level grammar cascades typically result in posting annotations over annotations, in a hierarhic fashion. This is, indeed, the conventional way of constructing a tree-like representation over a matched string. However, an in-line view of a multiply-embedded set of labels and substrings tends to be hard to parse by a naked eye. Consider, for example, the view of the shallow parser output, which produces syntactic analysis of phrasal and clausal fragments of text.



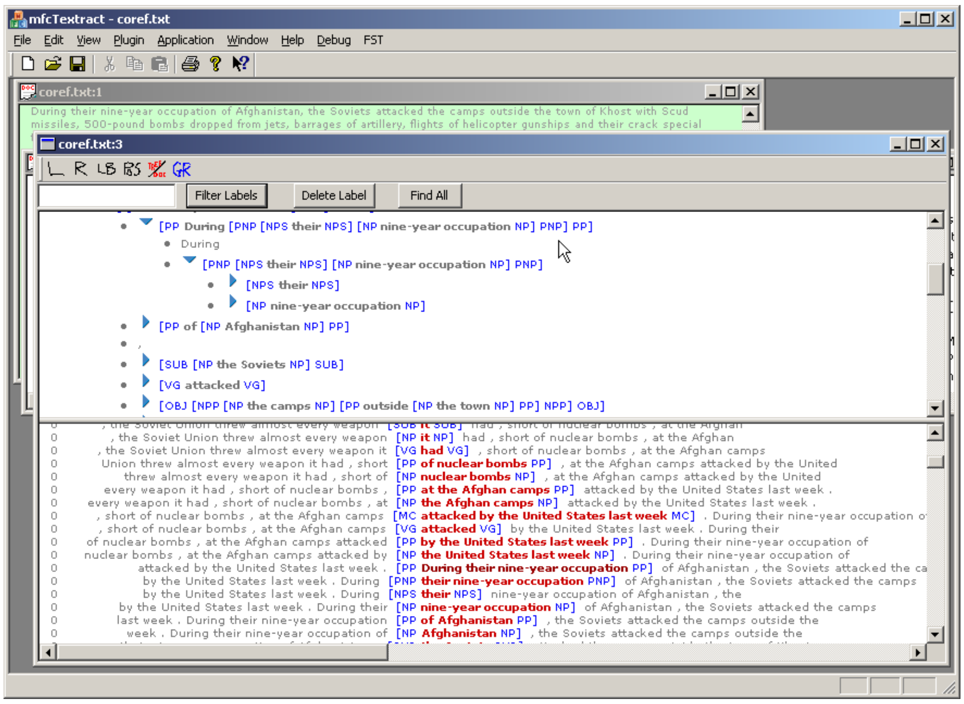
During their nine-year occupation of Afghanistan, the Soviets attacked the camps outside the town of Khost with Scud missiles, 500-pound bombs dropped from jets, barrages of artillery, flights of helicopter gunships and their crack special

```
[PP During [PNP [NPS their NPS] [NP nine-year occupation NP] PNP] PP] [PP of [NP Afghanistan NP] PP] , [SUB [NP the Soviets NP] SUB] [VG
attacked VG] [OBJ [NPP [NP the camps NP] [PP outside [NP the town NP] PP] NPP] OBJ] [PP of [NP Khost NP] PP] [PP with [NP Scud missiles
NP] PP] , [SUB [NP 500-pound bombs NP] SUB] [VG dropped VG] [PP from [NP jets NP] PP] , [NPLIST [NPP [NP barrages NP] [PP of [NP artillery
NP] PP] NPP] , [NPP [NP flights NP] [PP of [NP helicopter gunships NP] PP] NPP] and [PNP [NPS their NPS] [NP crack special forces NP] PNP]
NPLIST] . [CNP [NPP [NP The toughest Soviet commander NP] [PP in [NP Afghanistan NP] PP] NPP] , [NP Lt. Gen. Boris Gromov NP] CNP] , [MC
[VG personally led VG] [NP the last assault NP] MC] .
But [SUB [NPLIST [NP neither carpet bombing NP] nor [NP commandos NP] NPLIST] SUB] [VG drove VG] [OBJ [NPP [NP the Afghan holy
warriors NP] [PP from [NP the mountains NP] PP] NPP] OBJ] . [SUB [NP Afghanistan NP] SUB] [VG has VG] [OBJ [NP a long history NP] OBJ] of
[VG repelling VG] [NP superpowers NP] . [SUB [PNP [NPS Its NPS] [NP terrain NP] PNP] SUB] [VG favors VG] [SUB [NP defenders NP] SUB] [TC
as well as any [PP in [NP the world NP] PP] TC] , [TC whether [PNP [NPS their NPS] [NP opponents NP] PNP] TC] , [TC like [NP the Soviets NP]
TC] . [VG are trying VG] [SC [VG to defeat VG] [OBJ [NPP [NP them NP] [PP on [NP the ground NP] PP] NPP] OBJ] SC] or whether . [TC like [NP
```

```
0        , the Soviet Union threw almost every weapon it  [VG had VG]  , short of nuclear bombs , at the Afghan camps
0         Union threw almost every weapon it had , short  [PP of nuclear bombs PP]  , at the Afghan camps attacked by the United
0            threw almost every weapon it had , short of  [NP nuclear bombs NP]  , at the Afghan camps attacked by the United
0          every weapon it had , short of nuclear bombs ,  [PP at the Afghan camps PP]  attacked by the United States last week .
0       every weapon it had , short of nuclear bombs , at  [NP the Afghan camps NP]  attacked by the United States last week .
0        , short of nuclear bombs , at the Afghan camps  [MC attacked by the United States last week MC]  . During their nine-year occupation o
0        , short of nuclear bombs , at the Afghan camps  [VG attacked VG]  by the United States last week . During their
0       of nuclear bombs , at the Afghan camps attacked  [PP by the United States last week PP]  . During their nine-year occupation of
0       nuclear bombs , at the Afghan camps attacked by  [NP the United States last week NP]  . During their nine-year occupation of
0                attacked by the United States last week .  [PP During their nine-year occupation PP]  of Afghanistan , the Soviets attacked the ca
0                 by the United States last week . During  [PNP their nine-year occupation PNP]  of Afghanistan , the Soviets attacked the camps
0                 by the United States last week . During  [NPS their NPS]  nine-year occupation of Afghanistan , the
0             by the United States last week . During their  [NP nine-year occupation NP]  of Afghanistan , the Soviets attacked the camps
0             last week . During their nine-year occupation  [PP of Afghanistan PP]  , the Soviets attacked the camps outside the
0              week . During their nine-year occupation of  [NP Afghanistan NP]  , the Soviets attacked the camps outside the
0             their nine-year occupation of Afghanistan ,  [SUB the Soviets SUB]  attacked the camps outside the town of Khost
0              their nine-year occupation of Afghanistan ,  [NP the Soviets NP]  attacked the camps outside the town of Khost
0                 occupation of Afghanistan , the Soviets  [VG attacked VG]  the camps outside the town of Khost with Scud
0       occupation of Afghanistan , the Soviets attacked  [OBJ the camps outside the town OBJ]  of Khost with Scud missiles , 500-pound bombs
0       occupation of Afghanistan , the Soviets attacked  [NPP the camps outside the town NPP]  of Khost with Scud missiles , 500-pound bombs
0        occupation of Afghanistan , the Soviets attacked  [NP the camps NP]  outside the town of Khost with Scud missiles ,
```

file:///C:/Frasier/mfcTextractRT/coref.txt3.concL.htm#

The FST⇒SWITCH VIEW command, also available via the TREE/DOC button, alternates between a document (with inline labels) view and a tree view[32].



```
During their nine-year occupation of Afghanistan, the Soviets attacked the camps outside the town of Khost with Scud
missiles, 500-pound bombs dropped from jets, barrages of artillery, flights of helicopter gunships and their crack special
```

```
• [PP During [PNP [NPS their NPS] [NP nine-year occupation NP] PNP] PP]
    • During
    • [PNP [NPS their NPS] [NP nine-year occupation NP] PNP]
        • [NPS their NPS]
        • [NP nine-year occupation NP]
• [PP of [NP Afghanistan NP] PP]
• ,
• [SUB [NP the Soviets NP] SUB]
• [VG attacked VG]
• [OBJ [NPP [NP the camps NP] [PP outside [NP the town NP] PP] NPP] OBJ]
```

```
0         , the Soviet Union threw almost every weapon  [SUB it SUB]  had , short of nuclear bombs , at the Afghan
0          , the Soviet Union threw almost every weapon  [NP it NP]  had , short of nuclear bombs , at the Afghan
0        , the Soviet Union threw almost every weapon it  [VG had VG]  , short of nuclear bombs , at the Afghan camps
0          Union threw almost every weapon it had , short  [PP of nuclear bombs PP]  , at the Afghan camps attacked by the United
0              threw almost every weapon it had , short of  [NP nuclear bombs NP]  , at the Afghan camps attacked by the United
0            every weapon it had , short of nuclear bombs ,  [PP at the Afghan camps PP]  attacked by the United States last week .
0         every weapon it had , short of nuclear bombs , at  [NP the Afghan camps NP]  attacked by the United States last week .
0          , short of nuclear bombs , at the Afghan camps  [MC attacked by the United States last week MC]  . During their nine-year occupation o
0           , short of nuclear bombs , at the Afghan camps  [VG attacked VG]  by the United States last week . During their
0         of nuclear bombs , at the Afghan camps attacked  [PP by the United States last week PP]  . During their nine-year occupation of
0        nuclear bombs , at the Afghan camps attacked by  [NP the United States last week NP]  . During their nine-year occupation of
0                attacked by the United States last week .  [PP During their nine-year occupation PP]  of Afghanistan , the Soviets attacked the ca
0                 by the United States last week . During  [PNP their nine-year occupation PNP]  of Afghanistan , the Soviets attacked the camps
0                 by the United States last week . During  [NPS their NPS]  nine-year occupation of Afghanistan , the
0            by the United States last week . During their  [NP nine-year occupation NP]  of Afghanistan , the Soviets attacked the camps
0           last week . During their nine-year occupation  [PP of Afghanistan PP]  , the Soviets attacked the camps outside the
0             week . During their nine-year occupation of  [NP Afghanistan NP]  , the Soviets attacked the camps outside the
```

6.2.7 FST⇒Edit Grammar

As already mentioned, the current pre-release does not support native grammar editing and compilation. The *edit-run-debug* cycle requires, for the time being, a context switch between WTEXTRACT, your favourite editor, and a command line shell (for re-compiling the grammar). Clearly, at any point the runtime settings for the TFST executor can be explicitly changed via the EDIT⇒SETTINGS⇒PLUGINS dialog. If the only change between two runs, however, is the grammar source, simply rerunning the executor from within WTEXTRACT is going to pick up, and load, the new `.fst`[33], and reapply the cascade, displaying the results in a new TFST executor window. (The old window is kept, so that new and old output can

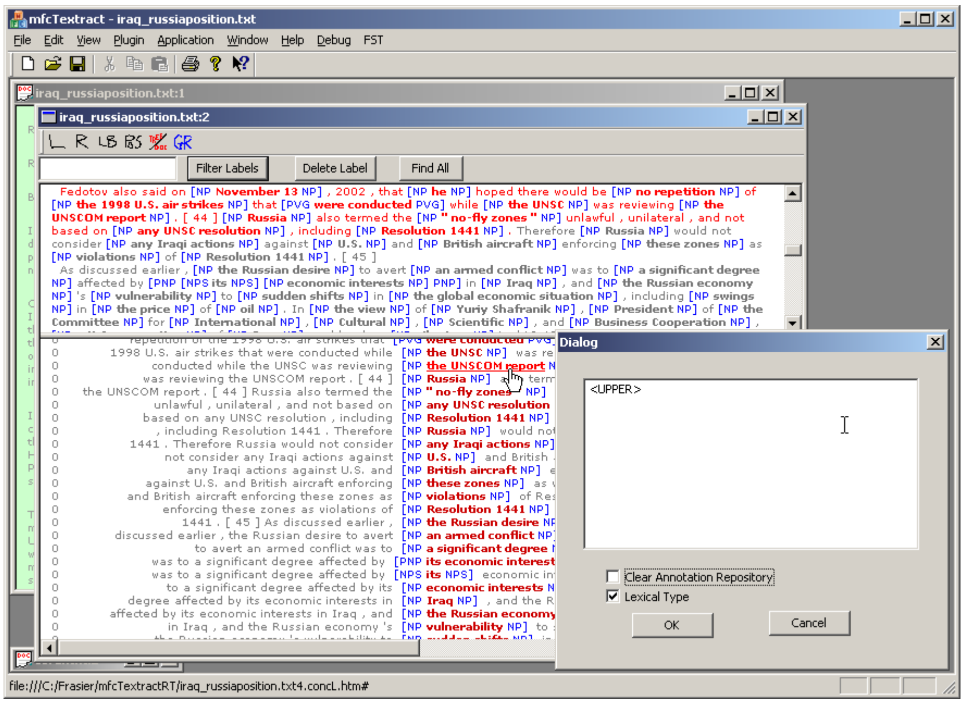
[32] Arguably, a marginally better rendering of a tree view is available by running APPLICATION⇒PARSER, with the same settings as PLUGIN⇒FSTEXEQ. The intent here is to bring this kind of display within the TFST executor output subsystem.

[33] Assuming, of course, that the modified `.cfg` has been recompiled.

be compared, thus assessing the effect(s) of modifying the grammar. Old windows can be closed, from the CLOSE (X) generic window manager button, at any time.)

Often in the course of grammar development, the grammar writer may focus on a particular rule, and even a specific symbol, with a question: what are the effects of applying this to the current document? To determine this, it would be necesary to create a new grammar file; input a simple rule which exercises the symbol, or pattern, in question; remember to post an annotation (so the matches can be viewed); name the file appropriately, and save it; compile it; reset the TFST executor to pick that `.fst` and load it; run the plugin; and view the results.

The FST⇒EDIT GRAMMAR command, also available via the GRM button, encapsulates this procedure into a single click:



```
 Fedotov also said on [NP November 13 NP] , 2002 , that [NP he NP] hoped there would be [NP no repetition NP] of
[NP the 1998 U.S. air strikes NP] that [PVG were conducted PVG] while [NP the UNSC NP] was reviewing [NP the
UNSCOM report NP] . [ 44 ] [NP Russia NP] also termed the [NP " no-fly zones " NP] unlawful , unilateral , and not
based on [NP any UNSC resolution NP] , including [NP Resolution 1441 NP] . Therefore [NP Russia NP] would not
consider [NP any Iraqi actions NP] against [NP U.S. NP] and [NP British aircraft NP] enforcing [NP these zones NP] as
[NP violations NP] of [NP Resolution 1441 NP] . [ 45 ]
 As discussed earlier , [NP the Russian desire NP] to avert [NP an armed conflict NP] was to [NP a significant degree
NP] affected by [PNP [NPS its NPS] [NP economic interests NP] PNP] in [NP Iraq NP] , and [NP the Russian economy
NP] 's [NP vulnerability NP] to [NP sudden shifts NP] in [NP the global economic situation NP] , including [NP swings
NP] in [NP the price NP] of [NP oil NP] . In [NP the view NP] of [NP Yuriy Shafranik NP] , [NP President NP] of [NP the
Committee NP] for [NP International NP] , [NP Cultural NP] , [NP Scientific NP] , and [NP Business Cooperation NP] ,
```

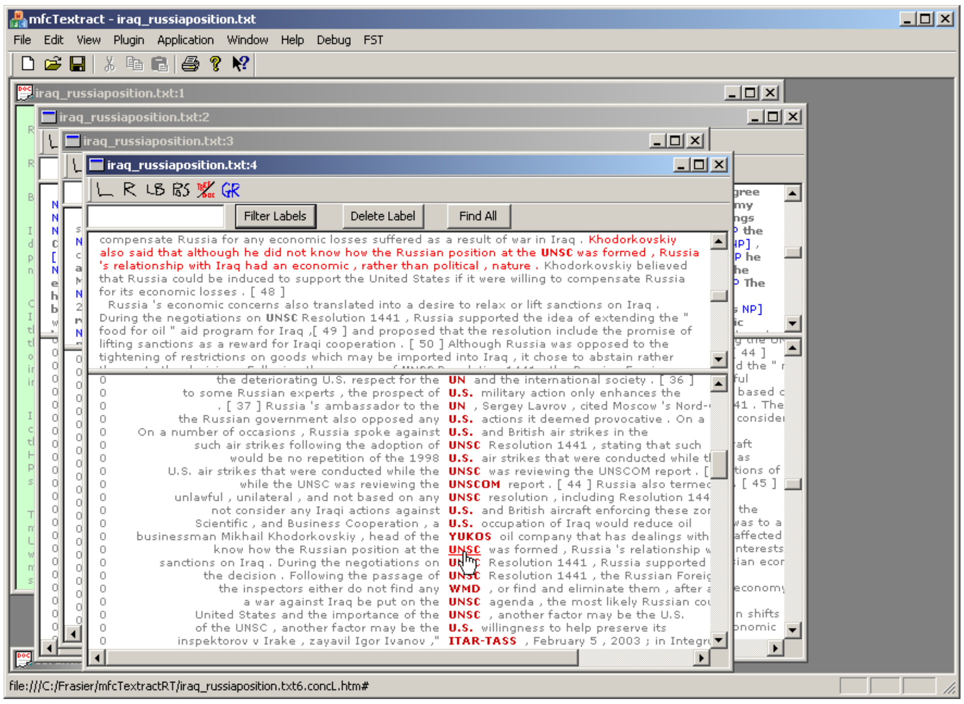
```
0        1998 U.S. air strikes that were conducted while  [NP the UNSC NP]  was re
0             conducted while the UNSC was reviewing  [NP the UNSCOM report N
0            was reviewing the UNSCOM report . [ 44 ]  [NP Russia NP]  a  term
0    the UNSCOM report . [ 44 ] Russia also termed the  [NP " no-fly zones  NP]
0              unlawful , unilateral , and not based on  [NP any UNSC resolution
0            based on any UNSC resolution , including  [NP Resolution 1441 NP]
0              , including Resolution 1441 . Therefore  [NP Russia NP]  would not
0          1441 . Therefore Russia would not consider  [NP any Iraqi actions NP]
0              not consider any Iraqi actions against  [NP U.S. NP]  and British
0                   any Iraqi actions against U.S. and  [NP British aircraft NP]  e
0           against U.S. and British aircraft enforcing  [NP these zones NP]  as
0        and British aircraft enforcing these zones as  [NP violations NP]  of Res
0               enforcing these zones as violations of  [NP Resolution 1441 NP]
0                  1441 . [ 45 ] As discussed earlier ,  [NP the Russian desire NP
0        discussed earlier , the Russian desire to avert  [NP an armed conflict NP]
0                     to avert an armed conflict was to  [NP a significant degree N
0            was to a significant degree affected by  [PNP its economic interest
0            was to a significant degree affected by  [NPS its NPS]  economic in
0              to a significant degree affected by its  [NP economic interests N
0       degree affected by its economic interests in  [NP Iraq NP]  , and the R
0      affected by its economic interests in Iraq , and  [NP the Russian economy
0             in Iraq , and the Russian economy 's  [NP vulnerability NP]  to
```

```
<UPPER>
```

[ ] Clear Annotation Repository
[✓] Lexical Type

file:///C:/Frasier/mfcTextractRT/iraq_russiaposition.txt4.concL.htm#

A modal dialog offers a text edit pane into which a simple rule (unnamed, no need to explicitly post a covering annotation) can be typed. The system does the rest; it also offers a choice between running the test over an annotation repository which has been cleared of the most recent set of TFST-created annotations, or which retains those analysis. Clearing the repository will allow for base-level analysis through a simple pattern: for instance, the output of the interaction from the screen shot above would be all occurrences of a <MIXED> tokens:


mfcTextract - iraq_russiaposition.txt
File Edit View Plugin Application Window Help Debug FST
iraq_russiaposition.txt:1
iraq_russiaposition.txt:2
iraq_russiaposition.txt:3
iraq_russiaposition.txt:4
Filter Labels
Delete Label
Find All

```
compensate Russia for any economic losses suffered as a result of war in Iraq . Khodorkovskiy
also said that although he did not know how the Russian position at the UNSC was formed , Russia
's relationship with Iraq had an economic , rather than political , nature . Khodorkovskiy believed
that Russia could be induced to support the United States if it were willing to compensate Russia
for its economic losses . [ 48 ]
 Russia 's economic concerns also translated into a desire to relax or lift sanctions on Iraq .
During the negotiations on UNSC Resolution 1441 , Russia supported the idea of extending the "
food for oil " aid program for Iraq ,[ 49 ] and proposed that the resolution include the promise of
lifting sanctions as a reward for Iraqi cooperation . [ 50 ] Although Russia was opposed to the
tightening of restrictions on goods which may be imported into Iraq , it chose to abstain rather
```

```
0          the deteriorating U.S. respect for the  UN   and the international society . [ 36 ]
0               to some Russian experts , the prospect of  U.S.  military action only enhances the
0                 . [ 37 ] Russia 's ambassador to the  UN   , Sergey Lavrov , cited Moscow 's Nord-
0            the Russian government also opposed any  U.S.  actions it deemed provocative . On a
0       On a number of occasions , Russia spoke against  U.S.  and British air strikes in the
0            such air strikes following the adoption of  UNSC  Resolution 1441 , stating that such
0                would be no repetition of the 1998  U.S.  air strikes that were conducted while t
0       U.S. air strikes that were conducted while the  UNSC  was reviewing the UNSCOM report . [
0                 while the UNSC was reviewing the  UNSCOM  report . [ 44 ] Russia also termec
0            unlawful , unilateral , and not based on any  UNSC  resolution , including Resolution 144
0              not consider any Iraqi actions against  U.S.  and British aircraft enforcing these zor
0             Scientific , and Business Cooperation , a  U.S.  occupation of Iraq would reduce oil
0     businessman Mikhail Khodorkovskiy , head of the  YUKOS  oil company that has dealings with
0              know how the Russian position at the  UNSC  was formed , Russia 's relationship w
0       sanctions on Iraq . During the negotiations on  UNSC  Resolution 1441 , Russia supported
0            the decision . Following the passage of  UNSC  Resolution 1441 , the Russian Foreic
0             the inspectors either do not find any  WMD  , or find and eliminate them , after a
0                a war against Iraq be put on the  UNSC  agenda , the most likely Russian cou
0         United States and the importance of the  UNSC  , another factor may be the U.S.
0             of the UNSC , another factor may be the  U.S.  willingness to help preserve its
0      inspektorov v Irake , zayavil Igor Ivanov ,"  ITAR-TASS  , February 5 , 2003 ; in Integr
```

file:///C:/Frasier/mfcTextractRT/iraq_russiaposition.txt6.concL.htm#

On the other hand, not resetting the annotation reporitory might be useful in investigating a question like: have all the <MIXED> tokens been subsumed, one way or another, inside of a generic noun phrase analysis?

This feature acts like a generalised *grep*: thus the annotations it posts are labelled TGREP.

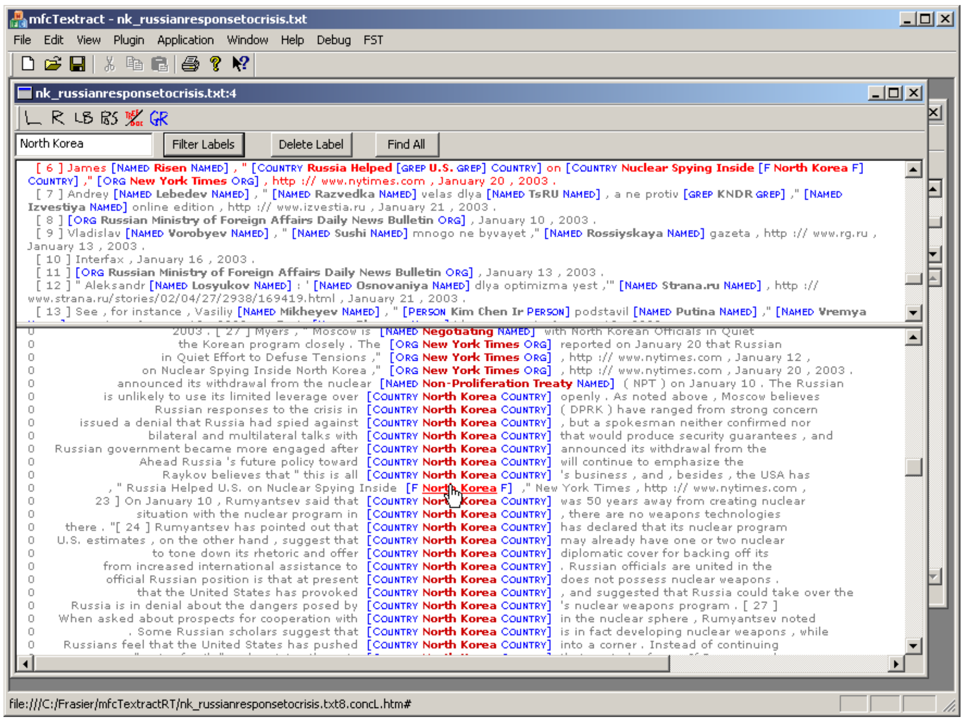
The following three operations are only available from a TFST executor window toolbar. They take as argument a string typed into the EDIT BOX, and interpret it to their own semantics.

6.2.8 Find All

A generic FIND command is available for any TFST executor window, from the EDIT⇒FIND menu. This uses the familiar FIND interface, and operates over the currently selected pane of the currently active TFST executor window. This offers some limited navigation through a document source and/or a concordance list. One possible use of FINDing is to scan all the occurrences of a text string in the top (document) pane, and observe whether they have been picked up by a grammar or not.

The FIND ALL button is an attempt to encapsulate such an operation; it is useful in observing the global behaviour of a grammar, acorss the entire document.

If a token string is typed into the EDIT BOX, the system will generate automatically a (TGREP) grammar (see 6.2.7 above), and run that against a (populated) annotation repository. If it finds any occurrences of the string which are not matched by the current grammar(s), they will be marked with an F (for FAILED?) label. Using the concordance navigation facilities we have at our disposal, it is now possible to get a display like the one below, which shows that on occurrence of a string, "*North Korea*", has not been picked up as a COUNTRY; looking at the concordance view alone, it is posisble to conjecture why; looking further into the document context view, it is also possible to both confirm the conjecture, and to discover that the grammar is wrong in a particular way.


mfcTextract - nk_russianresponsetocrisis.txt
File Edit View Plugin Application Window Help Debug FST
nk_russianresponsetocrisis.txt:4
North Korea
Filter Labels
Delete Label
Find All

```
[ 6 ] James [NAMED Risen NAMED] , " [COUNTRY Russia Helped [GREP U.S. GREP] COUNTRY] on [COUNTRY Nuclear Spying Inside [F North Korea F]
COUNTRY] ," [ORG New York Times ORG] , http :// www.nytimes.com , January 20 , 2003 .
 [ 7 ] Andrey [NAMED Lebedev NAMED] , " [NAMED Razvedka NAMED] velas dlya [NAMED TsRU NAMED] , a ne protiv [GREP KNDR GREP] ," [NAMED
Izvestiya NAMED] online edition , http :// www.izvestia.ru , January 21 , 2003 .
 [ 8 ] [ORG Russian Ministry of Foreign Affairs Daily News Bulletin ORG] , January 10 , 2003 .
 [ 9 ] Vladislav [NAMED Vorobyev NAMED] , " [NAMED Sushi NAMED] mnogo ne byvayet ," [NAMED Rossiyskaya NAMED] gazeta , http :// www.rg.ru ,
January 13 , 2003 .
 [ 10 ] Interfax , January 16 , 2003 .
 [ 11 ] [ORG Russian Ministry of Foreign Affairs Daily News Bulletin ORG] , January 13 , 2003 .
 [ 12 ] " Aleksandr [NAMED Losyukov NAMED] : ' [NAMED Osnovaniya NAMED] dlya optimizma yest ,'" [NAMED Strana.ru NAMED] , http ://
www.strana.ru/stories/02/04/27/2938/169419.html , January 21 , 2003 .
 [ 13 ] See , for instance , Vasiliy [NAMED Mikheyev NAMED] , " [PERSON Kim Chen Ir PERSON] podstavil [NAMED Putina NAMED] ," [NAMED Vremya
```

```
0            2003 . [ 27 ] Myers , " Moscow is  [NAMED Negotiating NAMED]  with North Korean Officials in Quiet
0                 the Korean program closely . The  [ORG New York Times ORG]  reported on January 20 that Russian
0                in Quiet Effort to Defuse Tensions ,"  [ORG New York Times ORG]  , http :// www.nytimes.com , January 12 ,
0             on Nuclear Spying Inside North Korea ,"  [ORG New York Times ORG]  , http :// www.nytimes.com , January 20 , 2003 .
0         announced its withdrawal from the nuclear  [NAMED Non-Proliferation Treaty NAMED]  ( NPT ) on January 10 . The Russian
0           is unlikely to use its limited leverage over  [COUNTRY North Korea COUNTRY]  openly . As noted above , Moscow believes
0                    Russian responses to the crisis in  [COUNTRY North Korea COUNTRY]  ( DPRK ) have ranged from strong concern
0         issued a denial that Russia had spied against  [COUNTRY North Korea COUNTRY]  , but a spokesman neither confirmed nor
0                   bilateral and multilateral talks with  [COUNTRY North Korea COUNTRY]  that would produce security guarantees , and
0      Russian government became more engaged after  [COUNTRY North Korea COUNTRY]  announced its withdrawal from the
0                  Ahead Russia 's future policy toward  [COUNTRY North Korea COUNTRY]  will continue to emphasize the
0                     Raykov believes that " this is all  [COUNTRY North Korea COUNTRY]  's business , and , besides , the USA has
0          , " Russia Helped U.S. on Nuclear Spying Inside  [F North Korea F]  ," New York Times , http :// www.nytimes.com ,
0              23 ] On January 10 , Rumyantsev said that  [COUNTRY North Korea COUNTRY]  was 50 years away from creating nuclear
0                  situation with the nuclear program in  [COUNTRY North Korea COUNTRY]  , there are no weapons technologies
0        there . "[ 24 ] Rumyantsev has pointed out that  [COUNTRY North Korea COUNTRY]  has declared that its nuclear program
0       U.S. estimates , on the other hand , suggest that  [COUNTRY North Korea COUNTRY]  may already have one or two nuclear
0                   to tone down its rhetoric and offer  [COUNTRY North Korea COUNTRY]  diplomatic cover for backing off its
0           from increased international assistance to  [COUNTRY North Korea COUNTRY]  . Russian officials are united in the
0           official Russian position is that at present  [COUNTRY North Korea COUNTRY]  does not possess nuclear weapons .
0                 that the United States has provoked  [COUNTRY North Korea COUNTRY]  , and suggested that Russia could take over the
0         Russia is in denial about the dangers posed by  [COUNTRY North Korea COUNTRY]  's nuclear weapons program . [ 27 ]
0      When asked about prospects for cooperation with  [COUNTRY North Korea COUNTRY]  in the nuclear sphere , Rumyantsev noted
0                . Some Russian scholars suggest that  [COUNTRY North Korea COUNTRY]  is in fact developing nuclear weapons , while
0        Russians feel that the United States has pushed  [COUNTRY North Korea COUNTRY]  into a corner . Instead of continuing
```

file:///C:/Frasier/mfcTextractRT/nk_russianresponsetocrisis.txt8.concL.htm#

6.2.9 Delete Label

Some of the annotations posted to the AR through the ‘snooping’ operations described above are, by their nature, only relevant to a very particular moment in time, and they should not survive further interaction cycles. However, it may be necessary for the annotation repository—as it has been populated by the recent grammar application—to remain intact for a while.

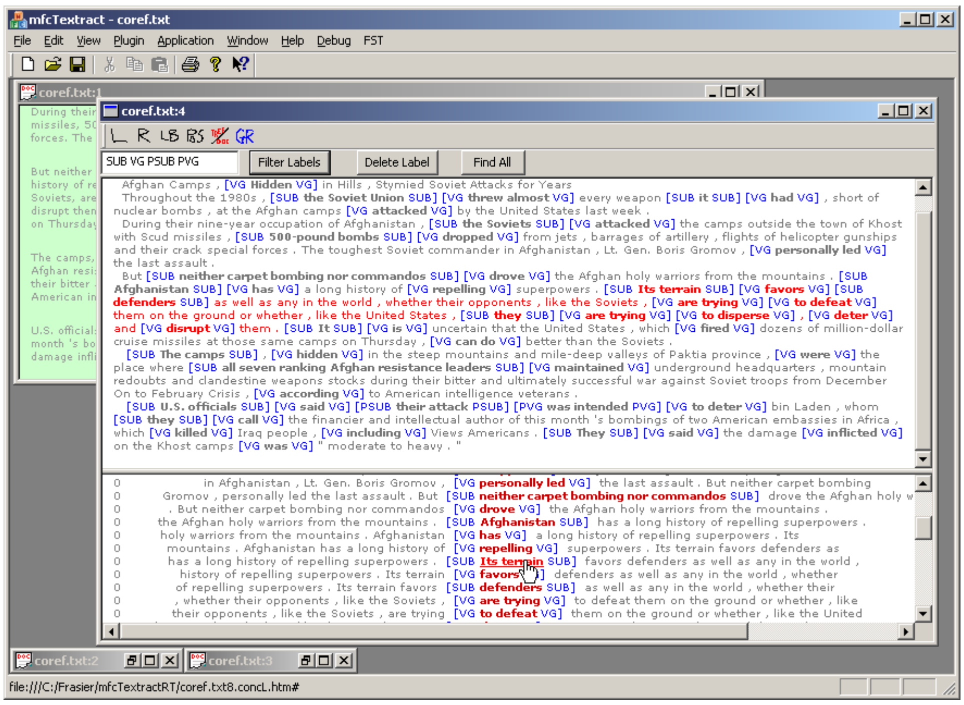
The DELETE LABEL interface allows for selective deletion of specified annotations. It is primarily a book-keeping device.

6.2.10 Filter labels

Any matching regime, especially one implementing complex grammars and multi-level cascades, may deposit a range of new annotation types into the annotation repository. This command offers a way of temporarily suppressing from display some of these annotations; it is a way of viewing, selectively, only a subset of new annotations, identified by their labels.

A sequence of label strings, separated by blanks, needs to be input into the EDIT BOX. The sequence is parsed out into one or more labels; these will be used as a filter by the display functions described earlier. A label so parsed needs to match (string, and case, equality) with the label of an AR annotation, for that annotation to be displayed.

In addition to reducing potential information overload (see the ROLEREL pattern filtered out in the figure on p.54), this is useful as a rudimentary facility for seeking patterns underlying relationships among items identified by patterns so far. For instance, the screen shot below illustrates the distribution of subjects and verb groups (both active and passive) across a document—an exercise which would facilitate the development of a grammar for relation finding.



```
 Afghan Camps , [VG Hidden VG] in Hills , Stymied Soviet Attacks for Years
 Throughout the 1980s , [SUB the Soviet Union SUB] [VG threw almost VG] every weapon [SUB it SUB] [VG had VG] , short of
nuclear bombs , at the Afghan camps [VG attacked VG] by the United States last week .
 During their nine-year occupation of Afghanistan , [SUB the Soviets SUB] [VG attacked VG] the camps outside the town of Khost
with Scud missiles , [SUB 500-pound bombs SUB] [VG dropped VG] from jets , barrages of artillery , flights of helicopter gunships
and their crack special forces . The toughest Soviet commander in Afghanistan , Lt. Gen. Boris Gromov , [VG personally led VG]
the last assault .
 But [SUB neither carpet bombing nor commandos SUB] [VG drove VG] the Afghan holy warriors from the mountains . [SUB
Afghanistan SUB] [VG has VG] a long history of [VG repelling VG] superpowers . [SUB Its terrain SUB] [VG favors VG] [SUB
defenders SUB] as well as any in the world , whether their opponents , like the Soviets , [VG are trying VG] [VG to defeat VG]
them on the ground or whether , like the United States , [SUB they SUB] [VG are trying VG] [VG to disperse VG] , [VG deter VG]
and [VG disrupt VG] them . [SUB It SUB] [VG is VG] uncertain that the United States , which [VG fired VG] dozens of million-dollar
cruise missiles at those same camps on Thursday , [VG can do VG] better than the Soviets .
  [SUB The camps SUB] , [VG hidden VG] in the steep mountains and mile-deep valleys of Paktia province , [VG were VG] the
place where [SUB all seven ranking Afghan resistance leaders SUB] [VG maintained VG] underground headquarters , mountain
redoubts and clandestine weapons stocks during their bitter and ultimately successful war against Soviet troops from December
On to February Crisis , [VG according VG] to American intelligence veterans .
  [SUB U.S. officials SUB] [VG said VG] [PSUB their attack PSUB] [PVG was intended PVG] [VG to deter VG] bin Laden , whom
[SUB they SUB] [VG call VG] the financier and intellectual author of this month 's bombings of two American embassies in Africa ,
which [VG killed VG] Iraq people , [VG including VG] Views Americans . [SUB They SUB] [VG said VG] the damage [VG inflicted VG]
on the Khost camps [VG was VG] " moderate to heavy . "
```

```
0          in Afghanistan , Lt. Gen. Boris Gromov ,  [VG personally led VG]  the last assault . But neither carpet bombing
0       Gromov , personally led the last assault . But  [SUB neither carpet bombing nor commandos SUB]  drove the Afghan holy w
0        . But neither carpet bombing nor commandos  [VG drove VG]  the Afghan holy warriors from the mountains .
0      the Afghan holy warriors from the mountains .  [SUB Afghanistan SUB]  has a long history of repelling superpowers .
0      holy warriors from the mountains . Afghanistan  [VG has VG]  a long history of repelling superpowers . Its
0       mountains . Afghanistan has a long history of  [VG repelling VG]  superpowers . Its terrain favors defenders as
0       has a long history of repelling superpowers .  [SUB Its terrain SUB]  favors defenders as well as any in the world ,
0        history of repelling superpowers . Its terrain  [VG favors VG]  defenders as well as any in the world , whether
0        of repelling superpowers . Its terrain favors  [SUB defenders SUB]  as well as any in the world , whether their
0        , whether their opponents , like the Soviets ,  [VG are trying VG]  to defeat them on the ground or whether , like
0        their opponents , like the Soviets , are trying  [VG to defeat VG]  them on the ground or whether , like the United
```

file:///C:/Frasier/mfcTextractRT/coref.txt8.concL.htm#

References

Becker, Marcus, Witold Drozdzynski, Hans-Ulrich Krieger, Jakub Poskorski, Ulrich Schfer, and Feiyu Xu. 2002. SProUT-shallow processing with unification and typed feature structures. In *Proceedings of the International Conference on Natural Language Processing (ICON 2002)*, Mumbai, India.

Boguraev, Branimir. 2000. Towards finite-state analysis of lexical cohesion. In *Proceedings of the 3rd International Conference on Finite-State Methods for NLP*, INTEX-3, Liege, Belgium, June.

Cunningham, Hamish, Diana Maynard, and Valentin Tablan, 2000. JAPE*: A Java Annotation Patterns Engine*. Institute for Language, Speech and Hearing (ILASH), and Department of Computer Science, University of Sheffield, UK. Research memo CS-00-10.

Ferrucci, David and Adam Lally. 2003. Accelerating corporate research in the development, application and deployment of human language technologies. In *Proceedings of HLT-NAACL Workshop on Software Engineering and Architectures of Language Technology Systems*, Edmonton, ALberta, Canada.

Karttunen, Lauri, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schiller. 1996. Regular expressions for language engineering. *Natural Language Engineering*, 4(1):305–328.

Kornai, Andras, editor. 1999. *Extended finite state models of language*. Cambridge University Press.

Neff, Mary, Branimir Boguraev, Herb Chong, Albert Eskenazi, Youngja Park, and Max Silberztein, 2003. *The Talent System: Design Document and Usage Notes*. IBM T.J. Watson Research Center, Yorktown Heights, NY, v. 2 edition.

Neff, Mary, Roy Byrd, and Branimir Boguraev. 2003. The Talent system: TEXTRACT architecture and data model. In *Proceedings of HLT-NAACL Workshop on Software Engineering and Architectures of Language Technology Systems*, Edmonton, Alberta, Canada.

Park, Youngja, Roy Byrd, and Branimir Boguraev. 2002. Automatic glossary extraction: beyond terminology identification. In *Proceedings of the 19th International Conference on Computational Linguistics (COLING)*, pages 772–778, Taiwan.

Ravin, Yael and Nina Wacholder. 1997. Extracting names from natural-language text. Technical Report RC-20338, IBM T.J. Watson Research Center, Yorktown Heights, NY.

Santorini, Beatrice, 1995. *Part-of-Speech Tagging Guidelines for the Penn Treebank Project*. University of Pennsylvania, (3rd revision, 2nd printing) edition.