The Complete *TextPro* Reference Manual

June 15, 1999

Douglas E. Appelt

Table of Contents

1. Introduction...3
2. Using TextPro...4
3. Lexicons and Lexicon Compilation...7
4. The Gazetteer and Gazetteer Compilation...9
5. Using POS Tag Frequency Data....10
6. The User Interface...11
The Display Menu...11
Contextual Menus...12
7. The Common Pattern Specification Language...13
The Declarations Section...13
The Rules Section...13
Macro Declarations...16
8. Calling External Functions...17
Argument Passing Conventions...18
Compiling a Shared Library...19
9. Debugging and Tracing...19
10. Interapplication Communication...20
11. Writing Input and Output Plugins...22
Input Plugins...22
Output Plugins...22
Useful Methods in TextPro Lib...22
Useful global variables exported by TextPro Lib...26
11. A Formal Description of CPSL...27

1. Introduction

This document describes the *TextPro* information extraction system, designed by Douglas E. Appelt. *TextPro* was a project undertaken for the pure fun of making interesting technology take shape. The original development of *TextPro* was not supported by any company or funding agency, and is mainained by its author on a strictly voluntary basis. However, TextPro just seems like it must be useful for something, although nobody is sure quite what. Therefore, SRI International is exercising some nebulous intellectual property rights. I will continue to give away the object code for free, but source code is being more strictly controlled. If you have questions, please contact Doug Appelt (appelt@ai.sri.com).

TextPro is an *Information Extraction System*. Its architecture is similar to that of many existing research information extraction systems, however, the *TextPro* system is distinguished by the incorporation of a number of features derived from the DARPA TIPSTER Program. *TextPro* reads natural-language texts, and places annotations on those texts as directed by a series of finite-state transducers. The structure of these annotations corresponds to the annotations described in the TIPSTER Architecture [1], and the finite-state transducers that generate these annotations are specified in the TIPSTER Common Pattern Specification Language.

The *TextPro* system consists essentially of a graphical user interface shell around a grammar interpreter engine. The grammar interpreter engine incorporates, in addition to the series of transducers mentioned above, a comprehensive lexicon of the grammar being processed, a gazetteer of place names that gives the proper names of geographical locations around the world, and some basic containment information, such as what state, province, or country a particular city is located in. Also, the system incorporates a table of unigram frequency data extracted from Wall Street Journal data in the Penn Tree Bank. This table gives, for each of about 26,000 English words, the frequencies associated with possible part-of-speech tags for that word.

TextPro is not really intended for the casual user. Out of the box, it comes with a sample grammar that does phrase parsing and name tagging of person, organization, and location names in newspaper text. However, it is possible to write arbitrary grammars for *TextPro*, and also to write plugins that allow *TextPro* to accept input in different formats, and produce output in whatever form is desired. This means that to extend its minimal out-of-the-box functionality to something really useful, you have to be prepared to write some (probably a lot) of code, and to do grammar development.

Although *TextPro* employs the TIPSTER architecture for the conceptual (and to a large extent, the actual) design of its internal data structures, it does not actually rely on a TIPSTER document manager. In a sense, *TextPro* is a document manager for the single text it is processing at the moment. *TextPro* incorporates an API based on AppleScript to share the annotations it places on a document with other applications. The elements of the Apple Event Object Model correspond straightforwardly to the TIPSTER Architecture elements "annotation," "attribute," and "span." This means that it would be a trivial matter to write an AppleScript that would interface with a real TIPSTER document manager, since the data structures used by the two modules are essentially isomorphic.

The *TextPro* grammar interpreter interprets finite-state grammars written in the *Common Pattern Specification Language (CPSL)* . *TextPro* initially annotates a text with "Token" annotations for each non-whitespace element in the text. The transducers defined by the grammar take TIPSTER annotations as input, and produce more annotations as output. Each grammar rule is a pattern-action rule. The pattern part of the rule tests the presence of annotations of the specified type with specified attributes. If the desired sequence of annotations is found, the rule match is recorded. When all rules have been tried at a given point in the document, the interpreter selects the rule that matches the longest sequence of input annotations, and among those tied for longest, choses the rule of highest priority. The action part of that rule is then interpreted. The action describes how to create

new annotations, and what attributes to associate with those annotations.

Because *TextPro* is an evolving organism, this documentation may or may not be consistent with the actual state of the currently released system. If it is not, please be patient, because the documentation should catch up with the program eventually!

2. Using *TextPro*

TextPro can be started one of two ways: by double-clicking its icon, or by sending it an "activate" or "run" Apple Event from a script. *TextPro* does not produce documents in the usual sense that applications do on the Macintosh. It does produce a compiled lexicon and grammar file, and double-clicking an icon of one of these files will launch *TextPro*, the file will not be loaded, because the 'open' Apple Event is intended for opening text files for processing. *TextPro* does enable the user to export its annotations as text files in certain pre-defined formats. However, these text files are given the type of the BBEdit program, rather than *TextPro*, because in most situations I envisioned, any subsequent processing of these files would be likely to involve a text editor, and the size of the files would be likely to exceed the limits imposed by SimpleText. This section describes running *TextPro* from the desktop as a normal application. Scripting *TextPro* will be discussed in the chapter on Apple Events.

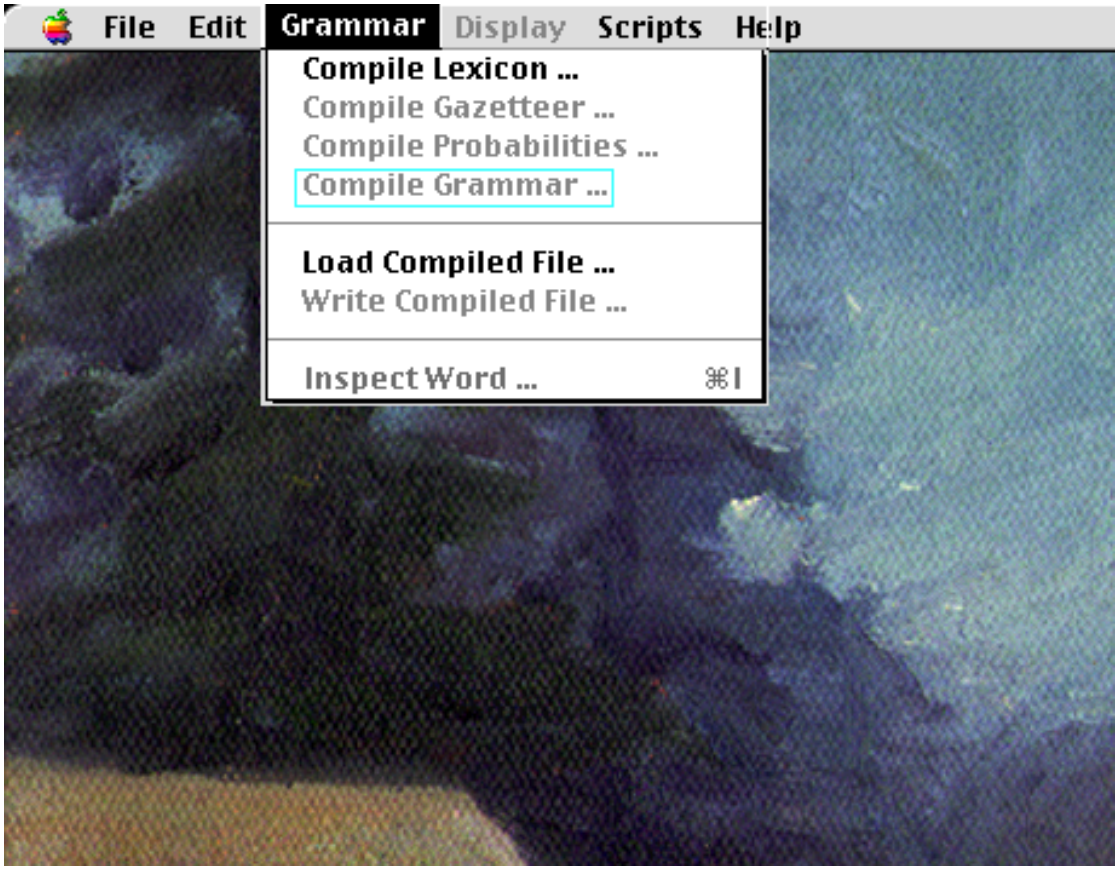


Figure 1

TextPro assumes that the user more or less knows what he's doing, so when it starts up, it just sits there, waiting for the user to tell it what to do. This section will describe what needs to be done to get it working.

TextPro is pretty useless without a grammar to interpret. However, a grammar is pretty useless unless there is a lexicon that defines the features on the words it will be dealing with, so before we do anything else, we need to load a lexicon.

To load a lexicon, simply select the "Load Compiled File..." option from the Grammar menu, and then select a compiled lexicon file to be loaded. A compiled file contains at a minimum a lexicon. It may also contain a gazetteer of place names, a grammar, and unigram frequency data for part-of-speech tags. A compiled lexicon file may be rather large (at least on the order of 10Mb for a real English grammar) so don't be surprised if it takes a few seconds to load. Once a lexicon has been loaded, it is possible to inspect an entry in the lexicon by selecting the "Inspect Word..." option. The system will display the lexical features of your chosen word for the various case possibilities listed in the lexicon, as well as the tag frequency data, if any is present.

A compiled lexicon can consist of lexical entries, and optionally a gazetteer, grammar, and tag frequency data. If the compiled lexicon file you loaded did not contain a grammar, a set of grammar

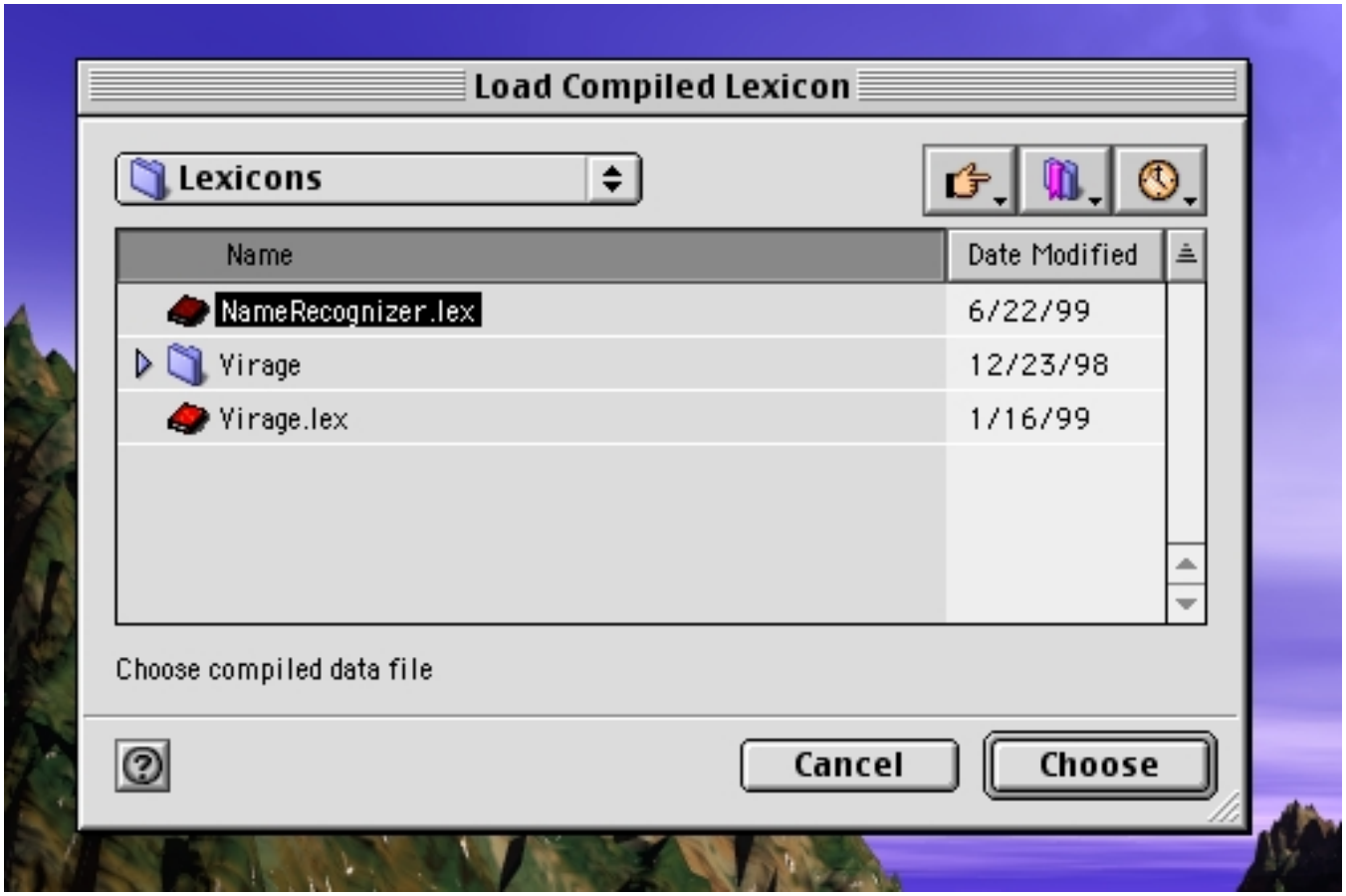


Figure 2

source files must be compiled. To compile a grammar, select "Compile Grammar..." from the Grammar menu, after which you will be presented with the multi-file selection dialog illustrated in Figure 2. The system expects that each phase (i.e. transducer) of the grammar is defined in its own file. The phases will be executed in the order that they are specified in this get-file dialog. The files are currently sorted into alphabetic order, so a simple expedient is to name your grammar phases in alphabetical order, although this is not great user-interface design.

It is possible to replace a grammar by compiling a new one on top of an existing lexicon. However, compiling a new lexicon invalidates any existing grammar, and if you do so, you must compile the grammar again.

After you have compiled the grammar (hopefully without errors — you cannot open a text until a grammar has been loaded, or successfully compiled) you are ready to get a document to process. To do this, choose "Open..." from the File menu. You can process any file of type 'TEXT', but it is wise to choose a file that really has text in it, since the system is not yet idiot proof. A Wall Street Journal text would be a good choice, and the standard distribution includes a few for test and demonstration purposes.

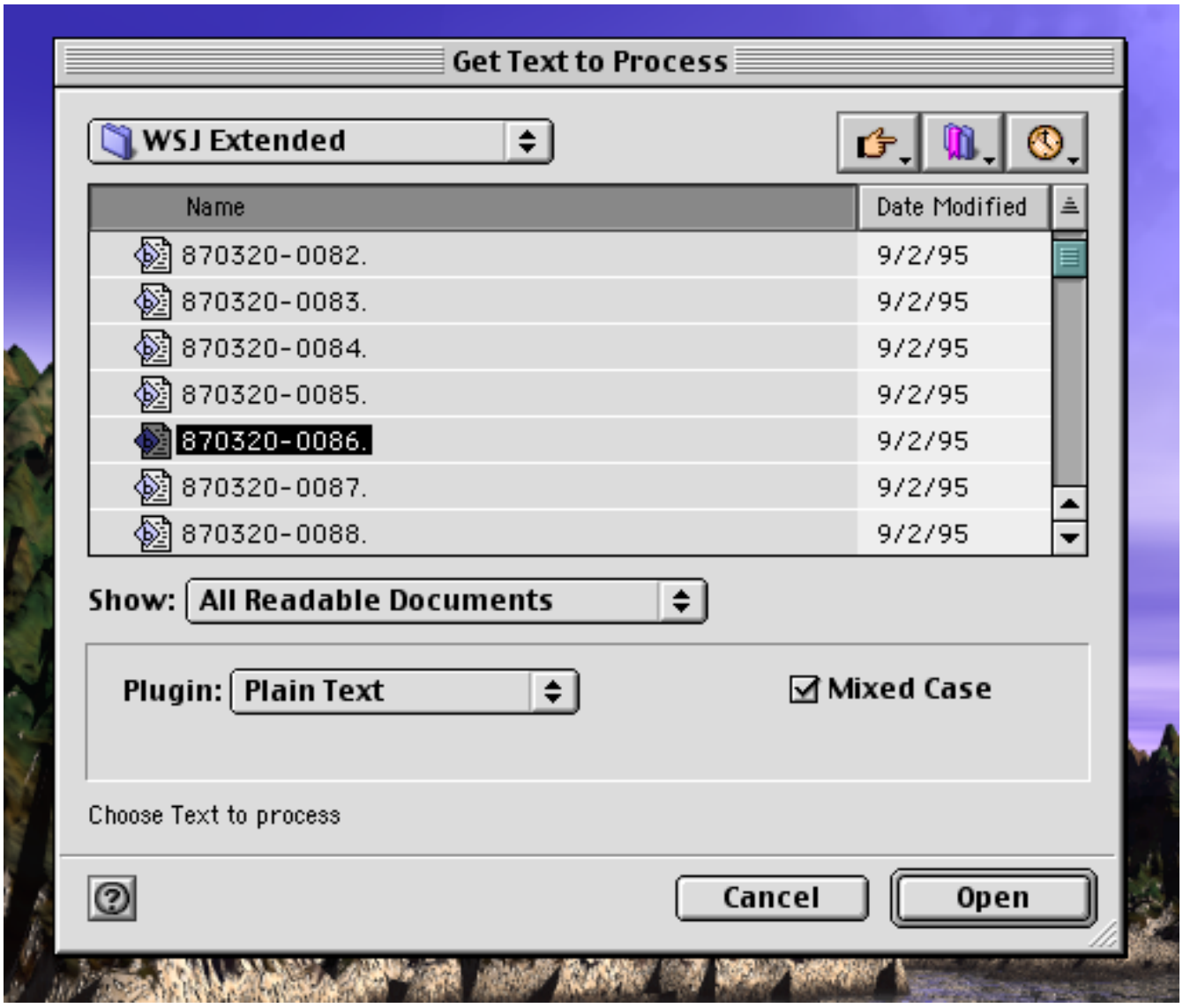


Figure 3.

The figure above illustrates the open dialog. Please direct your attention to the two custom controls in this dialog box. The checkbox marked "mixed case" should be checked when you want to eable mixed-case processing. This box should be checked if the selected grammar is designed to handle mixed case text (the included NameRecognizer.lex grammar is), and the text to be read contains upper and lower case characters. Typically, checking the box is manditory for grammars that

are optimized for mixed case text (as well as reading actual mixed case text), while grammars that process upper case only text will also typically work with mixed case texts, albeit at a lower level of accuracy than would be possible with a grammar optimized to upper and lower case text. The input plugin selector popup contains a list of all the input plugins found in the plugins folder when *TextPro* starts up. You should select the plugin that is appropriate for the text you are processing. The "Plain Text" plugin will work with just about anything.

After the document file has been selected, a window will appear with the grayed-out text of the document displayed within. (This window is actually a text editor, but the buffer is read-only, so you can select and copy text, but you are not allowed to alter the contents of the buffer.) When you see the text displayed, *TextPro* has already done a considerable amount of work. It has annotated each token in the text with annotations of type "Token," identifying all the non-whitespace things, including SGML tags, and tokens outside the area of interest in the document. Currently, the region is defined as anything between pairs of SGML tags <TXT> and </TXT> or <TEXT> and </TEXT>, or the entire buffer if these tags are not found.

In addition, *TextPro* examines the tokens in the region of interest, and annotates them with "Word" annotations. Each "Word" annotation has a pointer to its definition in the lexicon (assigned as an integer to the Lexentry attribute), or it has the boolean attribute "unknown" set to "true." If a sequence of words comprise a multiword, (like "because of") the span covering the entire multiword sequence is annotated as a "Word."

As the next operation in the text initialization, *TextPro* marks sentences in the text with annotations of type "Sentence." It is possible to select in the preferences panel which algorithm is used to segment the text into sentences. Because some texts (including the Wall Street Journal examples included with the distribution) contain <s> and </s> SGML tags to delimit sentences, there is an SGML-based segmenter that annotates based on these tags. There is also a heuristic sentence-breaking algorithm that segments sentences on the basis of punctuation, capitalization, and other factors, which is useful for texts that do not contain such SGML tags.

Some of the coreference resolution algorithms pay attention to paragraph structure, limiting their search for antecedents to a certain number of paragraphs as specified in the coreference preferences dialog. If coreference resolution is enabled, the system annotates paragraphs as well as sentences. The standard set of input plugins supports SGML-based paragraph breaking (and optionally, tags) or heuristic paragraph breaking based on line spacing and indentation.

To run the CPSL grammars over the currently selected text, simply choose "Run" from the File menu. A progress bar will appear that enumerates sentences as they are processed. The sample grammar included with the distribution produces a substantial variety of annotation types, including currency quantities, named entities, and basic English phrases. To examine the results, go to the "Display" menu and select which annotation type you wish to display, as described in Section 6. All annotations of the selected type will be displayed in a characteristic color in the display. You will see that *TextPro* is not always correct in its annotations, but it is right about 90% of the time in identifying named entities in Wall Street Journal texts.

3. Lexicons and Lexicon Compilation

Because *TextPro* is a descendent of the SRI FASTUS system, it incorporates various elements of the FASTUS system into its design. One of these elements is the lexicon. The lexicons are exactly the same as the SRI FASTUS lexicons, incorporating the same part-of-speech tags and syntactic format.

TextPro, unlike the original FASTUS, is capable of compiling lexicons, and writing out a binary file that can be loaded very quickly later. Loading a compiled lexicon is very fast — only a second or two

is required to load a 50,000 word compiled lexicon.

The *TextPro* lexicons inherit a very rigid and unforgiving syntax from FASTUS, but its rigidity makes errors easy to detect. Each lexicon source entry is contained on a single line and conforms to this general specification:

```
LexEntry: “base form” ; “var1” CAT1 , ... , “varn” CATn ; features .
```

The lexicon reader is quite fussy that all delimiters be surrounded by spaces. The “base form” of a word is its root morphological form, and can be used to group together words of a family that share different inflectional endings. Each morphological variant is stored as a separate entity in the internal lexicon. Each variant is paired with a syntactic category (part-of-speech tag). A set of boolean features is also provided, and the boolean features apply to each morphological variant in the lexicon entry. One of the features can be a number, which becomes a numeric value associated with that lexical entry. For example the word “five” would be associated with the numerical value 5, as well as the ordinal, “fifth.” When the system annotates words in the initial phases of processing, the numeric value becomes the value of the “Numval” attribute associated with the “Word” annotation.

If a lexical item contains a space, it is considered a *multiword*. Multiwords are grouped together in the process of annotating words. Dashes are allowed to freely alternate with spaces in separating the elements of `a multiword.` If the dash is mandatory, use the dash in the definition.

Lexicon entries also have a “case” attribute, which is derived from their printed representation. The case attribute is an integer that can take on the value 0 for words consisting entirely of lower case letters like “dog,” 1 for words that have all upper case letters like “AIDS,” 2 for words for which the initial letter is capitalized (or the initial letter of each part of a multiword) `like` “San Francisco”, and 3 for words that have irregular capitalization, like “NeXT.” If a word consists of a single uppercase letter, its case is stipulated to be upper-initial rather than upper only. *TextPro* treats words that are capitalized differently as entirely different lexical items. For example, “NeXT” is assumed to be an entirely different word, with a different set of features from “next”. If *TextPro* looks a word up in the lexicon and doesn’t find it, it annotates the word with an attribute “unknown” with value “true.” However, if there is a word in the lexicon with the same sequence of letters, except that it is capitalized differently, rather than annotating the word as unknown, it is annotated the union of the features for the capitalization possibilities that are defined. Also, if the word appears in a “capitalization context,” such as the first word of a sentence, it’s features are considered to be the union of its upper-initial and lower case possibilities.

In *TextPro,* all lexical features for a word are stored in a packed bit vector. An index to this vector is the value of the “Lexentry” attribute of the annotation spanning the text associated with the group of lexical features. The grammar specification does not distinguish between lexical boolean features represented in the bit vector and those that are represented explicitly with boolean valued attributes. The compiler just generates the right code to access them by knowing their names. This is why the compiled code depends on the lexicon that is loaded at the time of compilation. If you alter the lexicon, you have to recompile the grammar. By assigning the value of the “Lexentry” attribute to the “Lexentry” attribute of another annotation in the action part of a grammar rule, you can transfer all the lexical features to the new annotation in one operation.

Typical applications will require multiple lexicon files. TextPro allows you to load multiple lexicon source files and compile them together into a single compiled lexicon. If a single word has definitions in multiple files, (or in the same file for that matter) the final set of features on that word will be the union of the features defined for each individual entry.

The dialog box for lexicon compilation is the same as the one for grammar compilation. You select the "Compile Lexicon" option from the Grammar menu, and you are presented with a dialog box like the one shown in Figure 3. The lexicon files will be compiled, and a progress box shows how many lexical items have been processed so far.

4. The Gazetteer and Gazetteer Compilation

Intelligent processing of texts often requires the ability to recognize the names of locations. Although location names sometimes have predictable structure, usually they are frustratingly ad-hoc, and the only way to recognize them reliably is to have large lists of locations from all over the world.

Such large lists can be a mixed blessing, because locations names often overlap with the names of people, corporations, and ordinary English words. Foreign location names can also overlap with English words that have nothing whatsoever to do with what you would normally think of as locations, and the resulting situation can be quite confusing indeed.

For this reason, it is a necessary, but not sufficient strategy to simply enumerate the names of locations in a lexicon with a feature like LNAME. To handle the resulting frequent ambiguity, it is also necessary to include a little knowledge about the locations, so it is possible to have a prayer of disambiguating the name when it occurs in context. Providing this additional information is the duty of a *gazetteer*.

In addition to simply telling you that a word is a location name, the gazetteer provides some additional information about the location that can be useful in disambiguation. This information

takes the form of some basic containment data. Each location is identified by its basic *location type*, together with a pointer to a location of which it is a physical or political part.

So, for example, it would be reasonable to conclude on the basis of its internal structure that "Joe Montana" is the name of a person. However, Joe could be (and in fact is) the name of a location. If a program finds the words "Joe Montana," it could be more confident in the hypothesis that "Joe" is a location if it knew that "Montana" was also a location, and that "Joe" was in fact a town in "Montana." This kind of information is provided by a gazetteer.

For a number of years MUC and TIPSTER evaluations have employed a gazetteer that is freely available from NMSU to specify such hierarchical containment relations among locations all over the world. In the gazetteer, each location is associated with a location type. In addition, each location is also associated with up to two other locations that are its immediate superiors in a political containment hierarchy. The basic location types are CITY, PROVINCE, and COUNTRY. A typical gazetteer entry lists on a single line a CITY, the PROVINCE in which the city is located, and the COUNTRY in which the province is located. There are also types of cities ranging from CITY-1 (the capital of a country) to CITY-2 (the capital of a province) to CITY-3 (a large metropolitan city like New York) to CITY-4 (a smaller, though still important major city). Provinces are divided into PROVINCE-1 entities

5. Using POS Tag Frequency Data.

Many information extraction systems use part-of-speech tagging to improve parsing accuracy. This seems to be necessary once one includes a large enough lexicon to include very rare word senses. For example, the word "fell" can be an adjective in some archaic senses, and in idioms like "one fell swoop." However, in normal English text, the word "fell" is almost certainly a verb, and one doesn't want to risk parsing "fell 10%" as a noun group in a sentence like "The stock fell 10% in heavy trading." One could use a statistical part-of-speech tagger to try to compute the most likely part of speech for each word, given the words to its immediate left and right. The only problem with this approach is that it costs time to do the tagging, it costs storage space to store, in addition to the frequency data for the many thousands of words, the probabilities of all possible tag trigrams, and it might not even work. Even the best part of speech taggers are right only 95% of the time (think about one error per sentence, on the average) and the situations in which you would most like the disambiguating information are those situations in which the tagger is most likely to make mistakes.

As a cheap alternative to real part-of-speech tagging, TextPro will make use of a table of frequency data. We (actually Andy Kehler at SRI) has compiled a table of frequency data for about 26,000 English words, taken from the Penn Tree Bank Wall Street Journal data. For 26,000 words, the table gives the percent of time each word appears as a given part of speech, and how often in the corpus (about a million words of data) each word appears as each sense. For example, in the Wall Street Journal corpus, the word "dog" appears a total of 25 times, once as a verb, and 24 times as a noun. From this data we might estimate that "dog" is a verb less than 5% of the time. If we are analyzing a phrase in which the word "dog" could be either a noun or a verb, we would be right much more often than wrong to assume that it is a noun.

TextPro uses this frequency data to do "cheap" part-of-speech tagging. If it is possible to parse several phrases using a single word in both common and rare senses, the rare sense parses are discarded in favor of the more common ones, if the frequency data indicates that the number of corpus instances of the word is over a count threshold, and the frequency for the tag is less than a probability threshold. These thresholds can be set in the preferences panel. After a lexicon is loaded, the frequency data can be loaded by selecting "Compile Probabilities ..." in the Grammar menu, and specifying a file. When the compiled file is written out, probability data is included if it exists.

6. The User Interface

TextPro provides the user with a simple, but useful user interface for examining the results of the analysis of an input text. The user interface is based on the well known WASTE text editor (much praise is deserved by Marco Piovanelli for this solid freeware product), but the editing functions are disabled, and it serves simply as a display.

The Display Menu

When the system starts up, the Display menu is unavailable, because it makes no sense when there is no processed text to display. Inspection of the menu will reveal a set of display options at the top of the menu, and in the bottom of the menu below the line are several annotation types. Because TextPro contains standard input plugins for making certain annotations in English, these annotations (including Token, Word, Sentence and Paragraph) are already in the menu at startup time. When you load a grammar, each annotation type that can potentially be created by the grammar is reflected in a dynamically-added choice in the Display menu.

When you open a text for processing, the input plugins are assumed to create Token, Word, Sentence and Paragraph annoations, and these annotations can be examined before any further processing is done. Other annotation types cannot be displayed before the interpreter has been run.

After running the interpreter over a text, it is possible to show all the annotations of a given type in the document by selecting the type from the Display menu. Each annotation is displayed in a particular color, which is determined by the contents of the CSPC resource, described in Section ??.

Further information about the displayed annotations is available through the Display menu, as well as contextual menus. To examine the attributes of an annotation, display annotations of the desired type by making a choice in the Display menu. Then, place the cursor within the scope of the annotation of interest, and select the appropriate display operation from the top part of the display menu. Selecting "Display Annotation" will bring up a new window in which all the attributes of the annotation will be displayed in a hierarchical view. Attributes that refer to other annotations are displayed recursively, except that circular references are obviously not displayed, nor are coreference chains, which would inflate the size of the display too much. (There are some bugs in the hierarchical display code that prevent the "twisty triangles" from working properly. This code needs a lot of flogging, which I have not had time to do. The fully expanded display, which you get by default, is correct.)

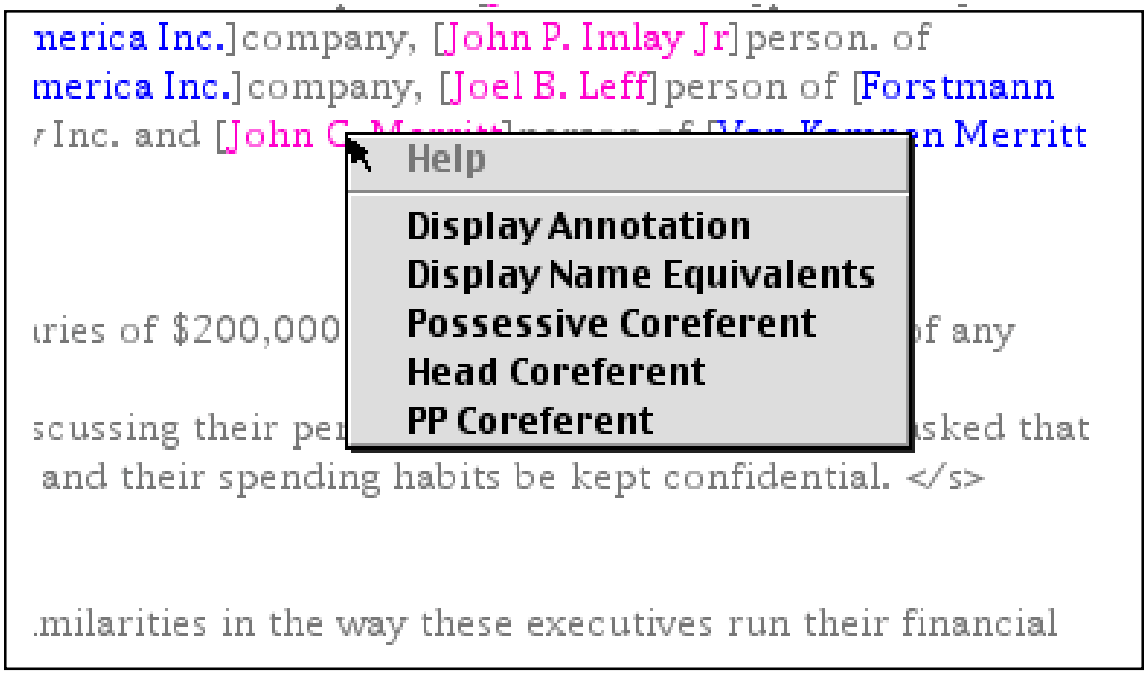
The display of coreference data is possible using other choices in the upper part of the Display menu. If annotations of type "NamedEntity" are selected for display, then clicking on "Display Name Equivalents" displays every name in the text that has been deemed by the name coreference module to refer to the same entity. For example, clicking "Display Name Equivalents" on "IBM" may also highlight an annotation of "International Business Machines, Inc." as well as other occurances of "IBM" in the same article.

Selecting annotations of type "Coreferent" displays noun groups that are either the antecedents of coreferring noun groups, or corefer with other antecedents. Selecting a Coreference annotation, and clicking "Head Coreferent" displays all previous phrases in the text that are coreferent with the selected phrase. The system seeks coreferents for pronouns (including posessive pronouns), and noun groups with definite determiners whose heads fall into categories for which a coreference plugin is available and willing to do the resolution. Writing coreference plugins is described in Section ??

Clicking "Possessive Coreferent" with a Coreferent annotation corresponding to a noun group with a posessive pronoun will reveal in the interface display the noun group that was deemed to be coreferential with the possessive pronoun.

Contextual Menus

Each of the alternatives in the top portion of the Display menu is available through the MacOS contextual menu interface. To display a contextual menu, hold down the control key while positioning the mouse over the annotation of interest. While doing so, the cursor should change from the normal arrow to a special contextual menu cursor. Actually clicking the mouse will bring up the contextual menu, from which you can select the same options that are available in the top portion of the Display menu.



7. The Common Pattern Specification Language

The Common Pattern Specification Language (CPSL) is a language defined by a TIPSTER working group for specifying finite-state patterns in information extraction systems. The idea was to devise a language for specifying patterns that would be general enough to be usable by many different systems as a "lingua franca" for finite-state patterns. TextPro uses a very close variant of CPSL as its pattern specification language. This document describes the language that is actually used by TextPro, and indicates where there are minor differences between the current implementation and the official specification.

A CPSL grammar is divided into three sections. The first section, which is optional, is a Macro section, where any macros used by the grammar are defined. The next section is a Declarations section, where various declarations about the grammar and how it is used are made. Finally, there is a Rules section, which contains the actual grammar rules.

The Declarations Section

The declarations should begin with a "Phase" declaration to declare the name of the phase. This is important for printing error messages, and for specifying tracing information. Then there is an "Input" declaration, where you list the annotation types of the annotations that will be used by this phase. If an annotation is not declared on this list, it is treated by the phase as though it did not exist. Finally there can be an Options declaration for telling the system about any compilation options that might be desirable. Currently, there is only one option, namely "Probfilter," which enables filtering on probability data, as described in the previous section. An example of declarations for a typical grammar phase is

Phase: company_names
Input: Word, Location
Options: Probfilter

The Rules Section

Following the Declarations section is the Rules section. The Rules section consists simply of a sequence of rules. Each rule has a name declaration, optionally a priority declaration, followed by a rule body as follows:

Rule: MyRule
Priority: 5
<rule body>

The rule priority can be any positive integer. If multiple rules match at a single point in the input stream, the longest match is selected. If several matches are of the same length, the match of the highest priority is selected. If several matches of the same length have the same priority, then the first match (i.e. the match corresponding to the earliest rule in the file) is chosen.

sequence of rules. Each rule has a name declaration, optionally a priority declaration, followed by a rule body as follows:

```
Rule: MyRule
Priority: 5
<rule body>
```

The rule priority can be any positive integer. If multiple rules match at a single point in the input stream, the longest match is selected. If several matches are of the same length, the match of the highest priority is selected. If several matches of the same length have the same priority, then the first match (i.e. the match corresponding to the earliest rule in the file) is chosen.

A rule body consists of a pattern part and an action part according to the following syntax:

```
Rule: MyRule
< rule pattern part>
   -->
<rule action part>
```

Each rule must have both a pattern and an action.

The rule pattern part consists of a sequence of labeled groups followed by an optional postfix pattern. The labeled groups specify what annotations are matched by the rule. The postfix pattern matches annotations following the labeled groups, but doesn't actually "consume" those annotations. (Note: the official CPSL language allows arbitrary prefix patterns as well as postfix patterns, but prefix patterns are not currently implemented in *TextPro*.) The syntax of the pattern part looks like this:

```
labeled group  < postfix pattern >
```

Labeled groups are simply groups of pattern elements with a label optionally attached. When a pattern matches annotations in the text, the annotations matched by the pattern are accessible through the attached label. Also, the span of the matched text is associated with the label, so when the action part of a rule creates a new annotation from a label, the new annotation receives the span associated with the label.

Groups are enclosed in parentheses. There are two types of label expressions: ":" labels and "+:" labels. Each is treated differently in the action part of a rule. When used to access an existing annotation, the label following the ":" operator refers to the rightmost annotation matched by the annotations within its scope. When used to create a new annotation, the label refers to the span encompassing the spans of all the annotations matched within the scope. The "+:" label, on the other hand, matches the sequence of annotations within the label scope when referring to existing annotations, and when the +: label is used to create a new annotation, it creates a multiple-span annotations, with spans corresponding to each individual annotation matched within its scope. Some examples are

```
(  < match specs > ):MyLbl
(  < match specs> )+:MyLbl
```

The pattern parts referred to as match specs consist of one or more annotation specifications, or a group of match specs containing or followed by a regular expression operator. The possible regular expression operators in CPSL are

```
Alternation: ( alt1 | alt2 | ... | altn)
Kleene Star:  (annot1 ... annotn)* or (annot1 ... annotn)+
Bounded Op:  (annot1 ... annotn)*N or (annot1 ... annotn)+N
Optionality: ( annot1 ... annotn)?
```

The bounded operators are similar to their more common unbounded kin, except that they will match at most N instances of a pattern. This is often very useful in defining patterns that will not become lost in a combinatorial explosion if the interpreter runs into a section of text that is not "real" natural language, such as a table that hasn't been marked as excludable text.

The annoation patterns consist of lists of attribute-value expressions. There are several common relational operators that can be used to compare attributes with values: "==" (equality) "!=" (inequality) ">" (greater than) ">=> (greater than or equal) "<" (less than) "<=" (less than or equal). An annotation pattern can also be a call to a boolean external function as described in the next section.

The list of attribute-value constraints is contained within "{}" pairs, and the elements are separated by commas. A special attribute expression is simply a quoted string. An expression like "MyString" is syntactic sugar for

{ Annot.string == "MyString }

In the above example "Annot" is actually a meta-variable. The value substituted for this meta-variable is the first element of the declared list of input annotations that appears in the Declarations section.

For example, if "Word" is the first Input annotation type, then

`"MyString"`

is exactly equivalent to

{Word.lemma == "MyString"}

assuming that "Word" is the first annotation on the Input declaration list. This is probably also a good time to mention that, in general, quoted strings and symbols are interchangeable in CPSL. The only time they are not is when the quoted string contains a space or a delimiter character. In that case, your only option is to quote the string. Within a quoted string, it is possible to quote the quote character by using a backslash as in \".

The action part of a rule consists primarily of assignment statements. As you might guess, the general form of an assignment statement is

```
Annotation/attribute spec  <assignment operator> value
```

An annotation/attribute spec consists of a label followed by the annotation type followed optionally by an attribute name. Actually in practice, it will almost always be followed by an attribute. A value can be a constant or a reference to a value. Possible constants are booleans ("true" and "false"), integers, floating point numbers, strings, or symbols (same as strings). A reference to a value can be a reference to an annotation, or to an attribute. An annotation reference consists of a label followed by an annotation type. This references the annotation of the given type starting at the first character of the matched annotation. (Note that it is possible to match an annotation of one type, and construct a reference to an annotation of a different type, as long as it starts at the same point in the input stream). An attribute reference consists of a label followed by an annotation type followed by an attribute name. It refers to the value of the named attribute on the referenced annotation. If the attribute does not exist, the value is stipulated to be boolean "false". The attribute name can be the name of a lexical feature. If this is the case, the value will be the value of the attribute in the packed bit-vector accessed through the pointer in the "Lexentry" attribute. If there is no such attribute, then the value is boolean "false."

A value expression can be the character "@". This is used when the annotation/attribute spec refers to an annotation. This tells the interpreter to construct an annotation with no attributes, which is useful for annotations like the "Sentence" annotations, which are used only to mark spans.

To make this all a little clearer, here is an actual rule taken from the name recognition grammar to illustrate the syntax of rules:

```
Rule: compound_loc1
( {Word.LNAME == true, Word.CITY == true} ","
     ({Word.LNAME == true, Word.PROVINCE == true} |
       {Word.LNAME == true, Word.COUNTRY == true})
  TestGazContainment[]):rhs
     -->
   :rhs.NamedEntity.Type = location,
   :rhs.NamedEntity.Subtype = city
```

The above rule matches sequences like "Palo Alto, California", and if confirmed by the gazetteer, creates a NamedEntity annotation spanning the entire match identifying the matched text as the

name of a city.

The most common assignment operator is "=" but there is also a "+=" operator, which assumes that the left hand side specifies an attribute with a sequence value, and the operator pushes the right-hand-side value onto the sequence.

The action parts of rules are also allowed to have conditional expressions of the form

```
(if conditional THEN actions1 ELSE actions2)
```

This conditional expression has the obvious interpretation. The conditional can refer to any attributes accessible through any label in the matched annotations. Also, the conditional grammar does not parse arbitrary expressions or implement operator precedence, so all conditionals are evaluated in the order they come. It's best to keep these expressions simple.

Macro Declarations

The Macro declaration section comes at the beginning of a grammar file, and here one can declare a number of text substitution macros to make the rules easier to right and more perspicuous to read.

The general form of a macro definition in CPSL is

```
MacroName[args] ==>  LHS --> RHS
```

In this definition, args is simply a list of one or more symbols. As in most macro processors, the parameters are bound to arguments by the macro invocation, and any occurrence of the parameters in the body of the macro are replaced with the strings that the parameters are bound to. What makes the CPSL macro language quite different from other macro preprocessors is the handling of the body. The macro is always invoked from the pattern part of the rule. When it is invoked, the macro invocation is replaced by the LHS string (after parameter substitution). Then the RHS string is added to the action part of the rule, preceding the action part of the invoking rule. The RHS part of the rule must always be present. If the rule doesn't need to augment the action part, you still need to insert a dummy placeholder.

The macro invocation consists of the name of the macro followed by the parameter list enclosed in double angle brackets. The arguments are separated by semicolons, a character that occurs nowhere else in CPSL, so the arguments to the macro can be arbitrarily complex strings that do not even have to be syntactically correct CPSL fragments.

```
MyMacro<< foo; bar >>
```

The only requirement is that after all macro substitutions are done, the result is a syntactically correct rule.

Here are a couple of important hints:

Since the RHS part of a macro is prepended to the action part of the rule it is invoked from, it must always end with a comma.

If a macro introduces nontrivial actions to a rule, you should always conditionalize the actions somehow in case the macro is invoked within the scope of an optional element (Kleene * or ?), otherwise the actions will be executed even if the conditions in the LHS didn't match anything. This probably needs a more elegant solution, but I usually use a test whether a matched annotation has a lemma attribute that is boolean false. Since all Word annotations have a lemma that is a string, the test succeeds if and only if the macro matched nothing.

If your macro introduces a label to the rule, you should use a label that depends on the name of the macro, so the introduced label won't conflict with other labels in the invoking rule. If the macro

will be invoked multiple times in a single rule, you had better parameterize the label.

Here is an example of a macro with a conditional expression It is intended to match ordinals that could possibly refer to dates:

```
#| Examples: 21st,  3rd, but not 43rd. |#
ORDINAL_DAY_OF_MONTH[ ] ==>
   ({OrdinalNum.Numval < 32}):dom
      -->
   ( IF :dom.OrdinalNum.Numval != 0
      THEN :rhs.DateTime.Day = :dom.OrdinalNum.Numval
   ), ;;
```

A detailed description of the syntax of CPSL is given in the last section. You should study the name recognition grammar provided with the *TextPro* distribution to gain a better understanding of exactly how the CPSL language is used to construct useful patterns.

8. Calling External Functions

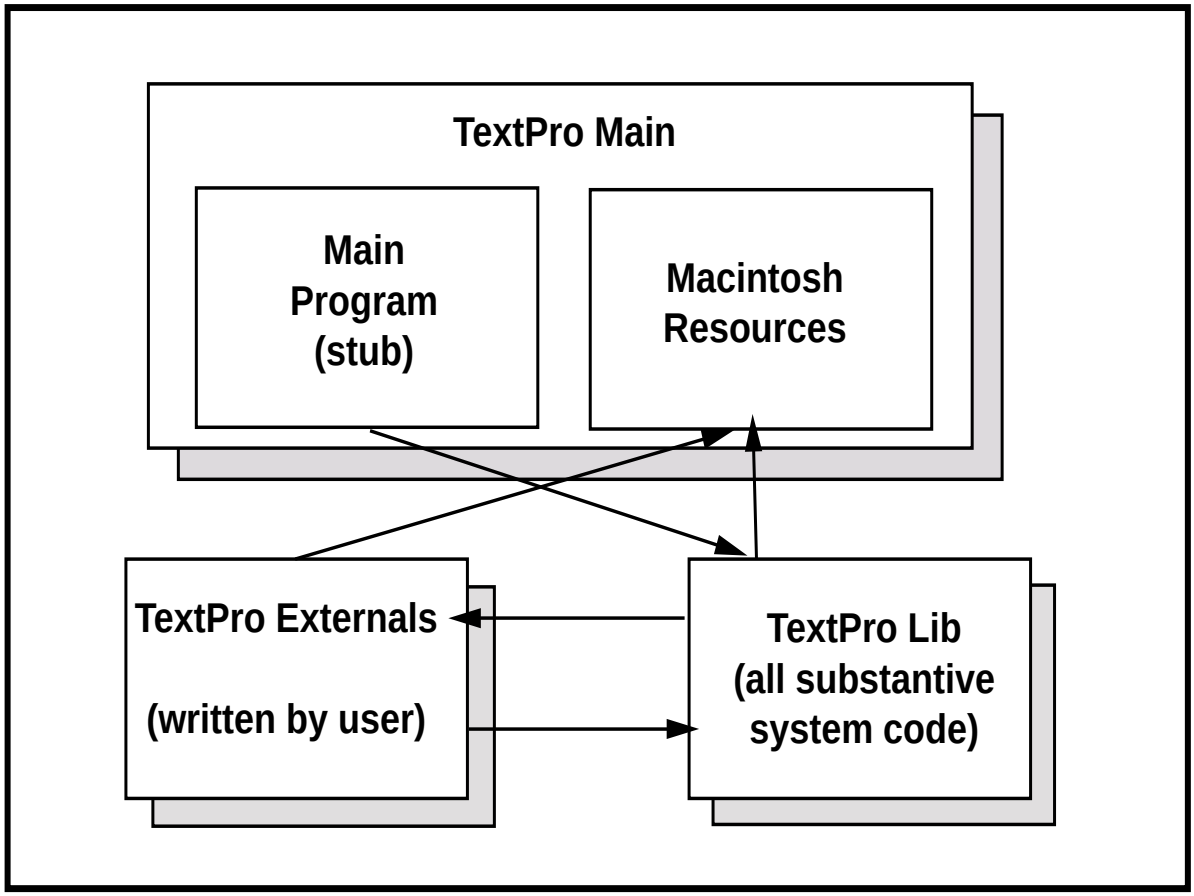
TextPro gives users the ability to write their own code and call this code from both the pattern and action parts of grammar rules during execution of the interpreter. This user supplied code is referred to as "external functions."

If you want to write external functions for *TextPro,* it is preferable to use Metrowerks Code Warrior, because Code Warrior takes advantage of a shared library of Metrowerks C++ runtime routines, and if you use Code Warrior, your external functions can share these runtime routines as well.

The diagram below illustrates the modularization strategy followed by *TextPro:*

The main program, which is what you launch when you launch *TextPro*, actually consists only of a stub and a resource fork containing all the Macintosh resources needed by the application. The substantive code of the *TextPro* system is contained within the TextPro Lib shared library, which is immediately called by the main program stub. The external functions are placed in a shared library called TextPro Externals, and they will then be able to be linked to all of the methods and global data structures of TextPro Lib. When the grammar interpreter encounters a reference to an external function, it looks up the symbol in the symbols exported from TextPro Externals, and calls the function. If the external function needs access to some functionality provided by methods in TextPro Lib, the method can simply be called. Header files for all the exported code in TextPro Lib are included in the "software development kit."

Not all globals are exported from TextPro Lib. Experimentation revealed that exporting all globals was a bad idea because some irregularities in library linkage would result in a system that was extremely unstable. The more conservative approach was adopted of exporting only those symbols from TextPro Lib that would be likely to be useful to external routines. This includes classes associated with the grammar, the lexicon, and the document manager, but excludes those classes associated with the interface, and grammar and lexicon compilers



Argument Passing Conventions

There are really two types of external functions in *TextPro:* those called from the pattern parts of rules (Boolean externals) and those called from the action part of rules.They use different argument passing conventions.

If a function is invoked in the action part of a rule, it is not assumed to return a value directly (i.e. it is declared void), but rather the value is passed via one of the parameters. Each action-callable function has two parameters: a list of input objects, and an output object. Each of these objects is a structure that the interpreter uses to represent data objects in a way that explicitly encapsulates the object's type. Definitions of those structures are found in the header file *InterpRegister.h*. When you write an external function, you need to check the arguments to see if the type you are getting is the type you expect. If not, your external function is responsible for coercing the types to something reasonable.

The current pattern-callable external functions are a kludge to enable things to more or less work until I get around to writing the real thing. Pattern-callable externals always return a Boolean value, which is true if the continuation is to be called by the interpreter, and false otherwise. If the pattern function returns false, the interpreter will fail and backtrack. Pattern-callable externals have a single argument which is a pointer to a linked data structure that has one node for each recursive call to the interpreter.Thus, by tracing up this stack, the external function can examine the entire state of the match up to the call.This means that writing these functions requires understanding of lots of undocumented code.That's why this is just a stopgap measure until I implement something better.

Compiling a Shared Library

To compile an externals library for *TextPro,* it is probably best to use Metrowerks Code Warrior C++.You must write the external functions in C++ (not in C) because the interpreter knows how to set up C++ linkages. If you compile in C, the interpreter will not be able to find and link to your functions. Future versions may support C in addition to C++.You should construct a project with all the externals files you want, and include the project stationery for a basic Macintosh Toolbox C++ program.

The externals library must be named TextPro Externals. You must include in your project the basic C++ runtime package, which is another shared library, MSLRuntimePPC++.DLL. The standard __Initialize and __Terminate functions should suffice for library initialization and termination, unless your routines require opening their own resource fork. You should include #pragma export directives in your header files to ensure that the entry points to your external functions are exported.

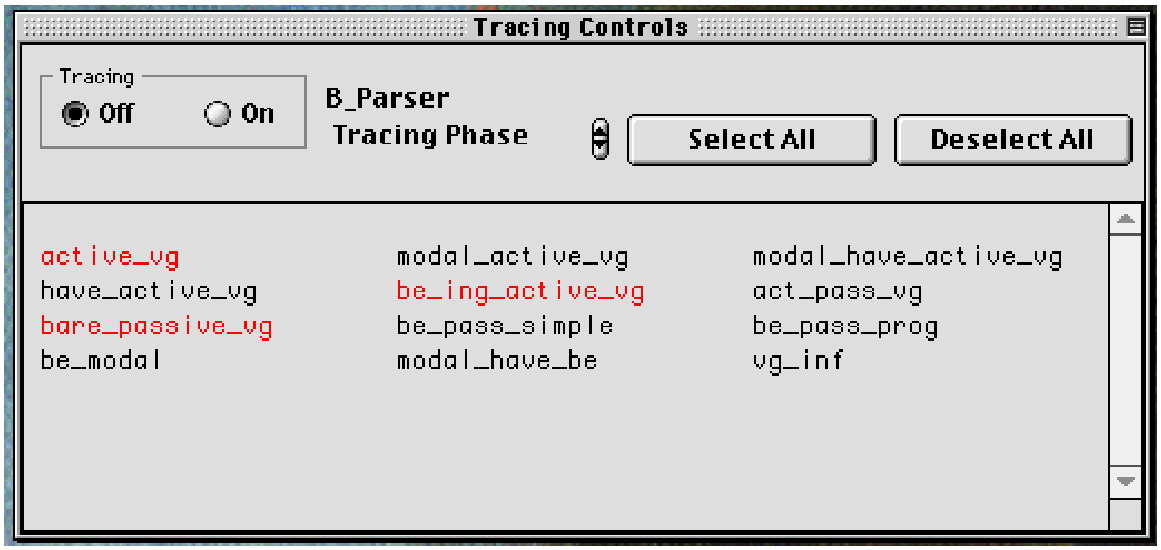
If you wish to call functions in TextPro Lib, you should include the relevant header files with your code, and also include TextPro Lib in the project libraries for TextPro Externals.

The software development kit contains the source code for all the external functions used in the name recognizer, as well as general arithmetic operations.This will give you an idea of how things work. Pay special attention to getting the declarations in the project settings correct.

9. Debugging and Tracing

Some people claim that they have a hard time writing programs that run the first time they try them. I can certainly testify to that fact, for if it were not true, Mad Doug would be as placid as the little old lady in Sunday school class. This is why Mad Doug Software comes with built-in tracing and debugging tools, as described in this section.

To debug patterns in the Common Pattern Specification Language, it is necessary to understand how the *TextPro* interpreter works. If you are familiar with the PROLOG programming language, you will quickly see that the *TextPro* Interpreter shares a lot in common with a PROLOG interpreter. *TextPro* does not employ unification or an associative store like PROLOG, but each rule is interpreted as though it were a PROLOG rule trying to prove that the pattern part of the rule matches the input annotations at the current position. If the annotation at the current input position matches the first pattern element, then the cursor is moved, and the remaining pattern is interpreted starting at the new input position. If the pattern element fails to match any annotation at the current input position,



then the interpreter fails, and backtracking causes any alternative choices to be tried. If the interpreter works its way through the entire pattern, it *succeeds.*, at which time a record of the match is place din temporarily allocated storage. The interpreter then forces a failure to ensure that backtracking exhaustively tries any alternatives to the current match. If failure propagates all the way back to the beginning of the match, the next rule is tried, until all the rules in a phase have had a chance to match the input at the current position. When the interpreter is done trying all the rules, it sees if any matches have succeeded along the way. Typically, several matches will have succeeded, and a "best match" must be chosen. The chosen match will be the rule that matches the longest sequence of input tokens. If several maximal length matches exist, then the one with the highest priority is chosen. If several matches have the same rule priority, then the first rule to match is selected as the chosen match.

When you want to observe a trace of *TextPro,* you should select the command "Show Tracing Controls" under the File menu. This will display the floating palette shown in the illustration below. You can select on e of the currently loaded phases from the popup menu, and then the names of the rules defined in that phase are shown in the scrolling pane in the bottom half of the control palette. You can select and deselect these rules by clicking the mouse on their names. The radio buttons in the upper left offer a way to turn tracing on or off globally.

After selecting the rules you want to trace, and enabling tracing, you should open a text for processing by *TextPro.* When the text appears in the display window, you select a range of text over which you want tracing to be done. Then select "Run" from the File menu. When the interpreter enters the suggested region, and if tracing is enabled on the control panel, the interpreter is traced for each rule mentioned in the control palette. You should use caution to avoid selecting either too much text to trace, or too many rules for tracing. If one were to trace every rule in a long and potentially complex text using this heuristic, it is likely that one could completely fill all of available memory with output from the Trace application, bringing *TextPro* to a grinding halt. Usually one is curious about why a rule that superficially should have applied did not in fact apply. Tracing that rule usually gives one a good idea of what is going on.

The trace produced by *TextPro* should remind one of the trace produced by a PROLOG interpreter. When the interpreter fully matches a pattern, the word "Succeed!" is printed in the trace. However, if multiple rules are matched at a given input position that falls within the range of the text to be traced, the tracing program prints the names of all the rules that matched (whether they were traced or not), and which of the matched rules was ultimately selected as the best. Often you find out that the rule you thought should match actually did match, only some other competing rule was actually selected as the best match. The fix will probably involve adjusting the priorities of the rule involved.

10. Interapplication Communication

TextPro would be little more than a curiosity if there were no way for other applications to use the services that *TextPro* provides. Therefore, there has to be a way to tell *TextPro* to process texts, and give back the resulting annotations, all under program control without the intervention of a human user.

If you have the luxury of writing the program that will use *TextPro's* services yourself, you can take advantage of the fact that all of the important functionality in *TextPro* is implemented in the TextPro Lib shared library. All the important methods and data structures are exported, so using *TextPro* in another application means nothing more than having your program call the appropriate methods.

However, the real benefit of *TextPro* would be realized by having *TextPro* work together with other legacy applications — applications for which you might not have access to the source code,

and for which the designers never conceived of the existence of a module with the functionality that *TextPro* provides. One might imagine an Email reader that would allow the user to use *TextPro* to analyze a message about a meeting announcement, and then annotate the time, date, place and purpose of a meeting, and then automatically enter this information into an appointment calendar.

Fortunately, such a program is not hard to implement, and one does not have to completely rewrite the Email reader and appointment scheduler to do so. The reason is because Apple has encouraged programmers to develop an API to all the applications they write that allows them to be contoured by Apple Events, and hence by scripts written in Apple Script. Fortunately for the *TextPro* user, many of the programs, such as web browsers, Email readers, databases, and personal productivity tools available on the Macintosh have pretty good Apple Script support. This, coupled with Apple Event support in *TextPro,* enables the user/developer to easily tie applications together with scripts.

You should use the Script Editor to open the Apple Event Terminology entries for the *TextPro* application. You will notice that under the heading "TextPro Suite" most of the basic menu commands from the top-level user interface are implemented. This means you can get *TextPro* to compile, load, and save grammars as well as to open and process documents under program control. There is also a command to get the contents of the clipboard as a document, which is useful for processing text selected by the user from some other application. In addition to all this, there is an "Accept String" command that has no corresponding function in the user interface. It allows a script to just hand a string of text to *TextPro* to be used as a document for analysis. This supports scripts that obtain text from another application under script control, and give it to TextPro for processing.

In addition to placing the top-level functionality of the user interface under script control, *TextPro* implements an *Apple Event Object Model* that gives scripts access to the annotation data structures that *TextPro* creates.

Not surprisingly, the elements of this object model correspond to the basic classes of the TIPSTER architecture, namely annotations, attributes, and spans. Since *TextPro* only processes one document at a time, it doesn't have anything corresponding to the TIPSTER collection and document classes. The annotation elements are associated directly with the application, and these are the "annotations of the document being processed at the moment."

It is possible to use the Apple Script "get" command to get annotations by numeric index. This sounds somewhat useless, except when you realize that attributes whose values are annotations, are treated by the Object Model as though they were integer value attributes. The integers that are the values of annotation-reference attributes are simply the numeric index of the referenced annotation. Thus it is easy to get the actual annotation referenced by another attribute. You can also retrieve annotations by annotation type by using a "whose" clause in Apple Script, for example:

```
get every annotation whose Type is "NamedEntity"
```

Just like the TIPSTER architecture, annotation Objects have two Apple Event sub-models: attributes and spans. You can get attributes from annotations by numeric index, or more usefully, by name. The Type property of the attribute is most useful for handling annotation references as described above, since Apple Script is a loosely typed language, and ordinarily you wouldn't have to know the type of the data explicitly. Spans are available by numeric index, and 99.9% of the time you'll just want the first span. As a matter of convenience, the Object Model treats the document text covered by the span as a property of the span, so you can use "get" with a span to get the exact text associated with the span.

As a matter of convenience for the user interface, there is a "Scripts" menu in the *TextPro* menu bar. This provides convenient access to scripts through the application menu instead of having to futz with the Script Editor. When you have a working script, you can place the script in the "Script

Menu Items" folder in the same folder as the application. All these scripts will appear in the menu when the application is launched. You can add additional scripts with the "Attach Script" command, if you wish.

11. Writing Input and Output Plugins

TextPro has a flexible, modular architecture for adapting the system to different input and output formats required by specific applications. This is done by writing plugins. When opening a file for processing, the user is given a choice among all available input plugins. Input plugins are files in the plugins folder that is in the same folder as the application, and which have a creator of 'TPro' and a file type of 'TPip'. Output plugins have a creator of 'TPro' and a file type of 'TPop'. These plugins are actually shared libraries that are required by the API to export a single symbol, which is a C-callable function "RunPlugin". The function is C-callable to avoid C++ name mangling, but it really has to be written in C++ because it takes pointers to C++ objects as input, and must call routines in TextPro Lib to place certain required annotations on the text.

Input Plugins

Input plugins have the following declaration:

```
extern "C" void RunPlugin(CLexicon* inLex, CDocument* inDoc) ;
```

When the input plugin is called, the document object contains the raw text of the document, but no annotations have yet been created. The job of the input plugin is to create annotations of type Token, Word, Sentence, and optionally Paragraph. Paragraph annotations are only required if coreference resolution is to be performed.

TextPro Lib already contains basic tokenization and lexical lookup functions suitable for English and similar European languages, as well as basic functions for sentence and paragraph breaking. For English texts, the primary duty of an input plugin will be to locate the relevant portion of the text that needs to be processed, and perhaps annotate some global information, such as document date, newswire source, etc. Langauges like Japanese and Arabic are left as an exercise for the reader. (Lots of extra credit.)

Output Plugins

Output plugins are pretty simple. They have the declaration:

```
extern "C" void RunPlugin(CDocument* inDoc, LFileStream* inStream) ;
```

The submenu of the File>Export command lists all the available output plugins for exporting annotations. The job of an output plugin is pretty loosely defined. The plugin is supposed to take some stuff from the document and write it onto the stream. Just what is written and in what format it is written is entirely dictated by the requirements of the application. The calling function will open and close the stream. The job of the plugin is to just write away. It is instructive to take a look at the ENAMEX output plugin included in the distribution SDK. It writes out a copy of the original input text, with ENAMEX, TIMEX and NUMEX annotations marked by special SGML tags as defined by various DARPA-sponsored evaluations.

Useful Methods in TextPro Lib

There is not much documentation written (yet) on the internal workings of *TextPro*. *TextPro* exports a lot of symbols from TextPro Lib that can be very useful for writing external functions, and input, output, and coreference plugins. For writing input and output plugins, you will certainly need

to call methods in TextPro Lib. Some of the most important ones are documented here.

`CPhase()`

This is the base class for all the tokenizer, multiword, sentence, and paragraph marking phases. It defines a virtual Run method.

`CTokenizer()`

Constructor for an object that implements a tokenizer for normal text in European languages.

`CSNORTokenizer()`

This is a tokenizer that is designed to run on UTF SGML-marked up documents. These documents contain so much SGML that it would be a waste of space to tokenize all of it, since for name tagging purposes, it all gets ignored anyway. Therefore, only the non-SGML portions of the text are tokenized.

```
void CTokenizer::Run(CDocument* inDoc, char* inType)
```

This method runs the tokenizer over the raw text of the document, and produces tokens with an annotation type indicated by inType. Usually, inType should be "Token", but some very unusual input formats like the output of speech recongizers with timing data, and perhaps N-best hypotheses, may best be handled with a two pass tokenizer, where CTokenizer marks up the input in a preliminary pass, and the actual Token annotations are constructed in a second pass.

`CMultiword()`

Constructor for an object that implements lexical lookup and multiword combination for normal text. It assumes that the text has already been marked up with Token annotations.

```
void CMultiword::Run(CDocument* inDoc, CLexicon* inLex) ;
```

This is the actual Run method for the lexical lookup phase. After this method runs, the document will contain Word annotations.

`CSGMLSentenceBreaker()`

Constructor for an object to do sentence breaking in a text that has <s> and </s> tags delimiting sentences. Its Run method places "Sentence" annotations whose span marks the extent of each sentence. It assumes Word annotations.

`CHeuristicSentenceBreaker()`

Constructor for an object to do sentence breaking based on punctuation and lexical heuristics. It assumes Word annotations.

`COneBigSentence()`

Constructor for an object that marks the entire span of relevant text as a single sentence. This is useful for texts without punctuation, like speech recognizer output.

`CSGMLParagraphBreaker()`

Constructor for an object that marks paragraphs with "Paragraph" annotations whose span marks the extent of each paragraph. It looks for and possibly tags to find the paragraphs. Assumes

Sentence annotations.

`CHeuristicParagraphBreaker()`

Constructor for an object that marks paragraphs based on heuristics of line spacing and indentation. Assumes Sentence annotations.

`void CDocument::LockText()`

Locks the document's raw text handle, so it doesn't move while processing.

`void CDocument::UnlockText()`

Unlocks the document's raw text handle, so the memory manager can muck with it.

`char* CDocument::GetRawText()`

Returns a pointer to the document's raw text.

`Int32 CDocument::GetDocumentLength()`

Returns the length of the document raw text, in bytes.

`void CDocument::PutAnnotation(Annotation* inAnnot)`

Adds an annotation to the document. The annotation will be deleted when the document is deleted.

`Sequence* CDocument::AnnotationsAt(Int32 inPosition)`

Returns the set of annotations starting at inPosition in document.

```
Annotation* CDocument::NextAnnotationOfType(const char* inType, Int32
inPos, Boolean allowHidden)
```

Searches for an anotation of type inType in the document starting at position inPos, or later. If allowHidden is true, then the annotation is returned even if it is marked as hidden. Otherwise, only visible annotations are returned.

```
Annotation* CDocument::NextAnnotationInTypeSet(Int16 inTypeCount,
char[][32] inTypes, Int32 inPosition)
```

Takes an array of annotation type names consisting of inTypeCount entries. It then returns the next annotation in the text, starting at inPosition or later, whose type is one of the members in the inTypes set.

```
Annotation* CDocument::PreviousAnnotationOfType(const char* inType, Int32
inPos, Boolean allowHidden)
```

Returns the annotation in the text the occurs at position inPos or earlier, of type inType.

`void CDocument::ExtractString(Int32 inStart, Int32 inEnd, char* outStr)`

Copies the raw text from the document into outStr, from inStart to inEnd, inclusive.

`DMSpan::(Int32 inStart, Int32 inEnd)`

Creates a TIPSTER Document Manager span object representing a span of text in the current document from characters inStart to inEnd, inclusive.

`Int32 DMSpan::GetStart()`

Returns the start offset of the span

`Int32 DMSpan::GetEnd()`

Returns the end offset of the span

`Sequence(AttributeType inType)`

Creates a TIPSTER Document Manger Sequence of objects of type inType.

`AttributeType Sequence::TypeOf()`

Returns the type of elements in a sequence.

`void* Sequence::Nth(Int16 inN)`

Returns the nth element of a sequence (the first element is at index 0).

`Int16 Sequence::Length()`

Returns the length of the sequence.

`void Sequence::Push(const void* inObj)`

Pushes an object onto the end of sequence. There is one polymorphic variant for double_t arguments, which are 8 bytes long instead of 4.

`void* Sequence::Pop()`

Pops an object off of the sequence.

`double_t Sequence::PopFloat()`

Pops a floating point double_t number off a sequence (an 8 byte value)

`Annotation(const char* inType, Sequence* inSpans, Sequence* inAttrs)`

Constructor that constructs an annotation of type inType, with spans inSpans and attributes inAttrs.

`Sequence* Annotation::GetSpans()`

Returns the span sequence of an annotation

`Sequence* Annotation::GetAttributes()`

Returns the attribute sequence of an annotation.

`void* Annotation::GetAttribute(const char* inName)`

Returns the value of the named attribute on this annotation. There is no provision for double_t here. For double_t attributes, get the attribute sequence, then search through it to find the desired attribute, then use GetFloatValue.

`Int32 Annotation::GetStartOfAnnotation()`

Returns the start offset of the first span.

`Int32 Annotation::GetEndOfAnnotation()`

Returns the end offset of the first span.

`Int32 Annotation::Length()`

Returns the number of characters spanned by the first span of the annotation.

```
char* Annotation::GetString()
```

Returns the type of the annotation.

```
Attribute(const char* inName, xxx inObj)
```

Polymorphic set of constructors that construct attribute-value objects.The name of the attribute will be inName, the value of the attribute inObj, and its type is determined by the type of inObj.

```
AttributeType Attribute::TypeOf()
```

Returns the type of the value of attribute.

```
xxx Attribute::GetXXXValue()
```

Polymorphic set of functions that return the value of an attribute, coercing or defaulting on type mismatches as reasonable. xxx ranges over Int32, char*, double_t, Annotation, Attribute, Sequence, and XXX ranges over Int, String, Float, Annotation, Attribute, and Sequence.

```
void Attribute::SetXXXValue(xxx inObj)
```

Ploymorphic set of functions that assign a value to an attribute object.

Useful global variables exported by TextPro Lib

```
CDocument* gDocument
```

Pointer to the document object

```
CLexicon* gLexicon
```

Pointer to the Lexicon object

```
CPreferences* gPreferences
```

Pointer to the object that manages the application preferences.This object can be queried for the current parameter settings selected by the user in the various options dialogs.

```
CGrammar* gGrammar
```

The grammar that is currently being processed by the interpreter.This object does not have a defined value while the plugins are executing, so it's only useful for grammar-callable external functions.

11. A Formal Description of CPSL

This section includes a formal description of CPSL, derived directly from the YACC grammar used by *TextPro* to parse it.

```
    phase:
  macros declarations rules

macros:
     | macros macro

macro:
     macro_header  DOUBLE_ARROW  arbitrary  RIGHT_ARROW   arbitrary
DOUBLE_SEMICOLON

macro_header:
   SYMBOL DOUBLE_LEFT_BRA param_list DOUBLE_RIGHT_BRA
arbitrary:
     | arbitrary something_or_other

something_or_other:
       SYMBOL
     | LEFT_ANGLE_BRA
     | RIGHT_ANGLE_BRA
     | DOUBLE_LEFT_BRA
     | DOUBLE_RIGHT_BRA
     | LEFT_SET_BRA
     | RIGHT_SET_BRA
     | LEFT_BRA
     | RIGHT_BRA
     | LEFT_SQUARE_BRA
     | RIGHT_SQUARE_BRA
     | AMPERSAND
     | BAR
     | NUMBER
     | MINUS
     | QUOTED_STRING
     | ANY
     | TEMP
     | NORM
     | IF
     | THEN
     | ELSE
     | STAR
```

```
      | PLUS
     | QUESTION
     | COLON
     | SEMICOLON
     | PLUS_COLON
     | ASSIGN
     | ADD_ASSIGN
     | COMMA
     | STOP
     | EQUAL
     | NOT_EQUAL
     | LESS_OR_EQUAL
     | GT_OR_EQUAL
     | CARAT
     | TK_TRUE
     | TK_FALSE
     | ATSIGN

declarations:
     | declarations decl

decl:
     PHASE COLON SYMBOL
     | input_token  COLON symbol_list

input_token:
    TK_INPUT

     SYMBOL
     | symbol_list COMMA SYMBOL
param_list:
     SYMBOL
     | param_list SEMICOLON SYMBOL

rules:
     | rules rule
     | rules error rule
rule:
     namedecl constraints RIGHT_ARROW actions
  | namedecl prioritydecl constraints RIGHT_ARROW actions

namedecl:
    RULE COLON SYMBOL

prioritydecl:
    PRIORITY COLON NUMBER
constraints:
```

```
     pre_condition constraint_group post_condition
pre_condition:
    | LEFT_ANGLE_BRA constraint_group RIGHT_ANGLE_BRA

post_condition:
     | LEFT_ANGLE_BRA constraint_group RIGHT_ANGLE_BRA

constraint_group:
    pattern_elements BAR constraint_group
    | pattern_elements

pattern_elements:
     pattern_element
    | pattern_element pattern_elements

pattern_element:
    basic_pattern_element
    | LEFT_BRA constraint_group RIGHT_BRA kleene_op binding
    | LEFT_BRA constraint_group RIGHT_BRA
    | LEFT_BRA constraint_group RIGHT_BRA kleene_op
    | LEFT_BRA constraint_group RIGHT_BRA binding
kleene_op:
     STAR
     | STAR NUMBER
     | PLUS
     | PLUS NUMBER
     | QUESTION
binding:
     index_op index
basic_pattern_element:
     LEFT_SET_BRA c_expression RIGHT_SET_BRA
     | QUOTED_STRING
     | SYMBOL
     | function_name LEFT_SQUARE_BRA RIGHT_SQUARE_BRA
c_expression:
     constraint
     | constraint COMMA c_expression
index_op:
     COLON
   | PLUS_COLON

index:
     NUMBER
     | SYMBOL
constraint:
    anno test_op value
    | anno_type
```

```
    anno:
         anno_type STOP attr_name

    anno_type:
         SYMBOL
        | ANY
    attr_name:
         SYMBOL
    test_op:
         EQUAL
         | NOT_EQUAL
         | LEFT_ANGLE_BRA
         | RIGHT_ANGLE_BRA
         | LESS_OR_EQUAL
         | GT_OR_EQUAL

    value:
         NUMBER
         | QUOTED_STRING
         | SYMBOL
         | TK_TRUE
         | TK_FALSE
    actions:
         action_exp more_actions
         | action_exp
    more_actions:
         COMMA action_exp more_actions
         | COMMA action_exp
    action_exp:
         LEFT_BRA IF a_c_expression THEN action_exp RIGHT_BRA
           | LEFT_BRA IF a_c_expression THEN action_exp ELSE action_exp
RIGHT_BRA
    a_c_expression:
         a_constraint
         | a_c_expression boolean_op a_constraint
    boolean_op:
         AMPERSAND
         | BAR
    a_constraint:
         index_expression test_op value
    action:
         assignment
         | function_call
    assignment:
         index_expression ASSIGN ATSIGN
         | index_expression ASSIGN value
          | index_expression ASSIGN index_expression
```

```
      | index_expression ASSIGN function_call
      | index_expression ADD_ASSIGN value
      | index_expression ADD_ASSIGN index_expression
     | index_expression ADD_ASSIGN function_call
index_expression:
     COLON index field
field:
      STOP anno_type STOP attr_name
    | STOP anno_type

function_call:
    function_name LEFT_SQUARE_BRA arglist RIGHT_SQUARE_BRA
function_name:
     SYMBOL

arglist:
     index_expression
    |  CARAT index_expression
     | value COMMA arglist
     | index_expression COMMA arglist
```