Department of Computer Science
CSC 247/447 and BCS 235
Fall 2006

Lisp Assignment 3

Due: 3:15pm, Thursday, Nov. 30 (hand in to TA)

Some Applications of Tree-Matching

Goal

The goal is to write high-level subroutines SUBST-CHUNKS and TRY-RULES for deriving a new tree or trees from a given tree (in Treebank form) that has been matched using MATCH-TREE, and to apply these programs to sentence simplification (using a SIMPLIFY-S program), and perhaps to some simple examples of generating responses (using a RESPOND-TO program), or some simple examples of making inferences (using a MAKE-INFERENCES program). Any effort devoted to the latter two applications will be counted as eligible for bonus marks – the simplification task alone is sufficiently challenging in the time available. All 3 applications require formulation of appropriate patterns to be matched, and templates for assembling the pieces derived from the input tree. You will also need a WORDS-OF-TREE program to obtain the list of words at the bottom (leaves) of a Treebank-style tree. Use this to show the actual input and output *sentences*, instead of just showing Treebank trees (which are hard to "read"!)[1]

Call formats and general approach

Function calls for SUBST-CHUNKS should be of form

`(SUBST-CHUNKS <template> <assoc-list>)`,

where the arguments are a tree-like *template* containing some numerical indices in place of subtrees or sequences of subtrees, and an *association* list of the type obtained in Lisp assignment 3 (i.e., a list of sublists, with successive sublists headed by indices 0, 1, 2, ..., and each sublist containing, in addition to the initial index, 0 or more successive phrase type specifications or subtrees; phrase type specifications are ones like `NN, NP` or `(NP CAT)` (the last specifying the head word along with the phrase type, `NP`), and subtrees are ones like `(ADVP (ADV YESTERDAY))` or `((NP CATS) (NNS CATS))` (the latter recording the head word of the `NP` on the initial element)). The result, as you may guess, should essentially be the template with occurrences of the numerical indices replaced by the corresponding chunks in the association list. Some care is required to get proper Treebank-style output trees, to make sure that initial elements of subtrees (indicating the syntactic type of the subtree) are not "decorated" with head words. Note also that "substitution" here really means splicing-in a list of 0 or more elements. For example, if we have `(3 <expr1> <expr2>)` on the association list, and the template contains a sublist `(<expr3> 3 <expr4>)`, then upon substitution that sublist should become `(<expr3> <expr1> <expr2> <expr4>)`; here the `cdr` of the association list item, `(<expr1> <expr2>)`, has been spliced into the position occupied in the template by '3'.

Function calls for TRY-TRANSFORMS should be of form

[1] As before, the data to be used are the excerpts from the Brown corpus at http://www.cs.rochester.edu/u/schubert/247-447/assignments/brown-extract.

`(TRY-RULES <tree> <rules>)`,
where the arguments are a (Treebank-style) *tree* and a *rules* list. Each rule in the rules list is a 2-element list whose first element is a pattern and whose second element is a template (of the kind described above). The output should be a list of all the non-nil results obtained when each rule is tried (i.e., we can have as many results as there are rules). It makes sense to implement TRY-RULES using a program TRY-RULE that just tries a single rule on a tree, i.e., it tries to match the pattern (given as first element of the rule) to the tree, and if successful, uses SUBST-CHUNKS on the template (given as second element of the rule) to obtain a result.

The main routines SIMPLIFY-S, RESPOND-TO, and MAKE-INFERENCES should just take a (Treebank-style) tree as input. In the body, they should decide what rules to try.

An example of the desired behavior of SIMPLIFY-S is that for input "*Nick Skorich, the line coach for the football champion Philadelphia Eagles, was elevated today to head coach*" we should obtain output "*Nick Skorich was elevated to head coach*" ("*today*" could perhaps be retained, as temporal adverbials are not very easy to identify). The simplest approach to the design of SIMPLIFY-S seems to be a "level-by-level recursive match" approach. In such an approach we use a pattern or patterns that just match the top-level constituents of a sentence, in particular "looking for" the main `NP` and `VP`, binding any other constituents that are present to indices whose associated "chunks" will be ignored in the template for forming the simplified tree. (For example, in a sentence of form `ADVP+NP+VP+PUNC`, we might match all parts but use only the `NP` and `VP` parts in the output template.) But note that if we use the top-level association list to fill in the output template, we're not done, because the subject `NP` and the `VP` will still be in unsimplified form. So at that point we should take the result of the top-level simplification and recursively replace each of its top-level parts by a simplified version. That means we need subroutines SIMPLIFY-NP and SIMPLIFY-VP – which will again work in the same sort of way. Similarly, we'll need SIMPLIFY-X subroutines for all types of constituents `X` that may occur in a simplified Treebank tree. (Of course, your program needn't be "universal", handling all cases; it will be sufficient if it handles some commonly occurring sentence types some of the time!)

The RESPOND-TO program (if you choose to do this), should make use of various rules (pattern+template pairs) aimed at particular kinds of inputs, and providing "appropriate" outputs (as far as this can be judged from the kind of input!) For example, a response to "*What can we do with the general?*" might be "*I'm not sure what we should do with the general*". Since we're imagining here that we're designing an ELIZA-like conversational system (but implementing only a capability for responding to individual Treebank sentences), you should probably experiment primarily with sentences from the Brown extract that contain `(PRP I)`, `(PRP me)`, `(PRP$ my)`, `(PRP you)`, `(PRP$ your)`, `(PRP we)`, or `(PRP$ our)`, and ones containing "?" or "!", as these tend to correlate with more interpersonal utterances. Note – it'll be important to handle some questions, not just declarative sentences.

Since responses need to be "finely attuned" to details of the input, it would be very inefficient (for any substantial rule base) to try all rules on all inputs. Rather, the method devised by Weizenbaum in building ELIZA was to use some of the words in the input as hash keys, and then to associate sets of rules with particular key words. However, we are only trying to demonstrate "proof of concept" here, so it will be OK to just design a smallish set of rules that are tried on every input; but probably you'll

want to make use of the head-word matching capability here, e.g., looking for a sentence whose subject has head word "*we*" and/or whose verb phrase has head word "*can*". **Note**: although you need only give a few illustrative examples, each of your rules should be demonstrated on more than one input sentence – otherwise you might be tempted to just design rules whose output looks good for the one input sentence you apply it to! Possibly, lacking enough suitable examples from the Brown extract, you might also apply the response program to *simplified* versions of some of the Brown sentences (using your simplification routine), and to some hand-constructed examples (taking care to conform with the Treebank syntax). Another productive method of obtaining more inputs to work with is to extract embedded sentences from the larger ones in the Brown extract – there are many such embedded sentences, though you have to watch out for gaps in those sentences (e.g., in relative clauses).

The MAKE-INFERENCES program (if you choose to do this one) would be quite similar to the RESPOND-TO program. Again, only a proof-of-concept demonstration is sought here. Examples of inferences might be that from "*I came into the lobby*" we infer "*I was in the lobby*"; from "*I admire the music of Prokofieff*" we infer "*I like the music of Prokofieff*", or perhaps more riskily, "*I like music*". You can see that as in the case of response generation, the patterns will need to be quite specific to particular word occurrences. Also, as in the case of response generation, it'll often make sense to generate inferences from simplified sentences or from embedded sentences. E.g., the Brown excerpt contains the sentence "*After all, a guy's gotta have a little ego*", from which we might simply infer the embedded sentence, "*A guy's gotta have a little ego*". Also, not only sentences but other constructions can be used to draw conclusions. For example, from "*Nick Skorich, the line coach for the football champion Philadelphia Eagles ...*" (an example of *apposition*) we can infer "*Nick Skorich was the line coach for the Philadelphia Eagles*".

Simplification rules – examples

The following examples and discussion should clarify what the simplification programs should do.

Here is a pattern for matching the top level of some common types of sentences (ignoring the top-level pairing of each sentence with a punctuation constituent in the Brown corpus):

```
(S (NP) (* (~ (VP))) (VP) (*T))
```

The subpattern `(* (~ (VP)))` could logically be replaced by just `(* T)`, but the slightly more complicated version "helps" the pattern matcher by not allowing the `VP` of the sentence to be bound to index 2, if for instance the match as a whole fails and the matcher backtracks and considers that binding. The corresponding template for a simplified version is

`(S 1 3)`,

which simply assembles the `NP` and `VP` from the original sentence, ignoring other constituents (if any). As pointed out, we still need to simplify the resulting top-level constituents in the `cdr` of the resulting tree, i.e., the `NP` and the `VP`. In simplifying an `NP`, we might use the following pattern and template, among others:

```
(NP (* (~ (DET) (N))) (- (DET)) (* (~ (DET) (N)))
    (* (N)) (N) (* (~ (N) ((PP of)))) (- ((PP of)))
    (* (~ (N) ((PP of)))))

(NP 2 4 5 7)
```

The pattern checks for an NP that may have some initial material that is neither a determiner nor a nominal (where we take DET to be a generalization of CD, DT, PDT, PRP$, WDT, and WP$, and N to be a generalization of EX, NN, NNS, NNP, NNPS, PRP, and WP); then it checks for a possible determiner, then some possible constituents that are neither determiners nor nominals (e.g., adjective phrases), then some possible premodifying nominals, then the head nominal, then some possible constituents that are neither nominals nor PP[of] phrases, then a possible PP[of] phrase, and then some possible constituents that are neither nominals nor PP[of] phrases. The template then forms a simplified NP containing only the determiner (if any), the premodifying nominals (if any), the head nominal, and a postmodifying PP[of] (if any). The reason for picking out postmodifying PP[of]'s is that they are quite common, and almost always represent an essential complement, not a modifier; (a rare exception: "*a person of great integrity*").

One point you might wonder about is why use such complex negative patterns like (* ((N) ((PP of)))), rather than just (* T). The answer is that the latter could in principle "absorb" the head noun (if there are premodifying nouns) and a PP[of]-complement – and we would then discard these constituents, which we are trying to treat as essential.

We again have to recursively simplify the remaining constituents of an NP, after the above simplification step (or similar ones). So this requires simplification rules for PP's – and since these contain NP's, we will be applying the NP-simplification rules recursively.

In much the same way, we need to supply rules for simplifying VP's. This is probably the trickiest part, because making good guesses about what parts of a VP are essential subcategorized complements, and which ones are inessential adjuncts (modifiers) is not easy to guess. One obvious point is that we should probably treat NP's in the VP as complements. But be alert to 2 complications at least: one is that *temporal* NP's such as "*this week*" are almost always modifiers; the other is that the Brown trees can contain "flattened" coordinate phrases – for instance treating the VP in "*I saw John and Mary*" as being of form (VP V NP CC NP).

Remark on response generation and entailment generation: Though this has been relegated, because of time constraints, to a bonus part, it's worth noting that these applications are rather more interesting than simplification, because they would probably involve rules that use "deeper" patterns than the single-level patterns we used for simplification. In other words, the simplification task doesn't really show the full power of tree-matching.

Write-up, etc.

The usual principles of good program design and documentation apply:

- The documentation should describe and explain the structure of the programs, and should point out any ways in which your approach differs from or augments the suggested methods; point out potential problems or subtleties you noticed and whether (and how) you dealt with them, and any noteworthy features of your approach or program.
- The programs should contain headers briefly explaining what they do (including what the input parameters mean and what sorts of output are produced), and comments in the code where this is helpful;
- The programs should be as "elegant" as possible; i.e., they should be concise, clearly structured, and make good use of the constructs of LISP such as recursion and MAP functions (and other "fell swoop" functions) where appropriate.
- The programs should not have obvious, easily remedied inefficiencies;
- Find good examples in the Brown corpus excerpt to illustrate what your programs can (and can't) do.
- State your conclusions about the effectiveness and coverage of your simplification routines, and any ideas you have about how they could be improved; similarly for any work done on response generation or inference generation.