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

Lisp Assignment 3

Due: 3:15pm, Thursday, Nov. 2 (hand in to TA)

Pattern Matching on Trees

Goal

The goal is to write a program MATCH-TREE that matches a tree-pattern against a parse tree (in Treebank form), and returns matched chunks of the parse tree as an "association list" (whose elements are of form `(<index> <subtree1> <subtree2> ...)`, with indices increasing as we go through the list). This will take us another step closer to being able to generate new sentences from given sentences (both in Treebank form), for purposes such as simplification, patterned response generation, or (superficial) inference.

Call format

Function calls for MATCH-TREE should be of form

`(MATCH-TREE <pattern> <tree> <index>)`,

where the arguments are a tree-pattern, a parse tree (in Treebank form), and a starting index (normally 0 when matching a sentential tree). A basic option we'll allow for is that for any tree or subtree that we're matching, we can either retrieve the entire (subtree) and associate it with the given index, or we can match and retrieve parts of the (sub)tree and associate them with indices that start at the given index and increase as we move left-to-right through the pattern and the subpatterns that are being matched against the tree. This option of matching-as-a-whole versus matching-parts is important because when we are matching parts of a tree, we already get back an analysis of the entire tree (in the pieces comprising the association list), and don't want to be forced to return the unanalyzed tree as well.

The choice between these options will be implicit in the form of the pattern, with a pattern that is a singleton list corresponding to the whole-tree option. Since we also provide a choice between "decorating" the initial element of a matched tree with head words (possibly prespecifying one or more of the head words that must be found), or leaving the initial element as-is, we have 4 cases to consider for a successful match, assuming that the given index is i:

1. If the pattern is of form (X) where X is an atom (e.g., `S`), a singleton "association list" of form

   `((i <tree>))`

   should be returned, where `<tree>` is the parse tree given as input (which will have initial element X). (It's an association list in that `<tree>` becomes associated with i.)

2. If the pattern is of form ((X w1 w2 ...)) where X is an atom and w1, w2, ... are 0 or more words (atoms such as `cat`), then an association list

`((i <augmented-tree>))`

should be returned, where `<augmented-tree>` is the parse tree given as input, except that its initial atom X has been expanded into a list (X hw1 hw2 ...) where hw1, hw2, ... are the head-words of the X-tree, which (for a successful match) include the words w1, w2, ... specified in the pattern.

3. If the pattern is of form (X ...) where X is an atom and the dots stand for a *nonempty* sequence of additional pattern elements, then an association list

   `((i X) (i+1 <trees1>) (i+2 <trees2> ...)`

   should be returned, where `<trees1>`, `<trees2>`, ... are sequences of 0 or more subtrees of the parse tree (typically exactly one) corresponding to various parts of the given pattern, possibly with the initial atoms of some of these subtrees expanded into lists that include the head words of the subtrees. The indices *i, i+1, i+2, ...* correspond to parts of the pattern in a way that will become clear. (As will be seen, the pattern/parse-tree correspondence is not in general between top elements of the pattern and top elements of the parse tree, because tree-matching potentially involves descent in the pattern and in the tree.)

4. If the pattern is of form ((X w1 w2 ...) ...) where X is an atom and the second set of dots stands for a *nonempty* sequence of additional pattern elements, then an association list

   `((i (X hw1 hw2 ...)) (i+1 <trees1>) (i+2 <trees2> ...)`

   should be returned, where hw1, hw2, ... are the head-words of the X-tree, which (for a successful match) include the words w1, w2, ... specified in the pattern, and `<trees1>`, `<trees2>`, ... are as in the previous case.

This covers the non-compound patterns, but we will also allow certain compound patterns, still aimed at individual trees, of form `(? <patt1> ... <pattk>)` ($k \geq 1$) indicating alternatives, and of form `(~ <patt1> ... <pattk>)` ($k \geq 1$) indicating patterns that should *not* match the given tree. (For the latter type of pattern, we need a special convention about what to return: we simply return the entire tree that was matched – i.e., which did *not* match any of the specified patterns – associating this tree with the given index.) Additional compound patterns (using `-` for optionality and `*` for iteration) aimed at lists of trees rather than individual trees will be introduced below.

One final kind of pattern is the atom `T`, which is taken to be true for any tree, so that `((<index> <tree>))` should be returned. In case of match failure, the result should be `nil`.

Examples

The following examples and discussion are intended to provide you with an intuitive understanding of the pattern language and its meaning. If you prefer, look ahead to the more formal/technical information given subsequently, and then come back to these examples.

```
(match-tree '(S (NP) ((VP died)) (* (ADVP)))
            '(S (NP (DT the) (ADJP (JJ old)) (NN cat)) (VP (VBD died)))
            0 )
===>

 ((0 S)
  (1 (NP (DT THE) (ADJP (JJ OLD)) (NN CAT)))
  (2 ((VP DIED) (VBD DIED)))
  (3))
```

The pattern (first argument) can be read as looking for a match to a tree (list structure) whose initial element is `S`, whose first subtree matches the pattern `(NP)`, whose second subtree matches the pattern `((VP died))` and whose sequence of remaining subtrees (if any) matches the pattern `(* PP)`. The correspondence between the given pattern and the indices for returning the output is as follows:

```
(S (NP) ((VP died)) (* (ADVP)))
(0  1       2           3  )
```

In general, all elements of the pattern, and their subelements to any depth, are assigned indices in left-to-right ascending order. However, we do not descend recursively into the initial element of any element or subelement in assigning these indices. For example, the third element `((VP died))` of the above pattern has as its initial (and only) element the list `(VP died)`, but we do not assign separate indices to its parts. Also, we do not assign indices to special initial list elements "`*`", "`-`", "`?`", or "`~`".

Note that only the type symbol S itself, but no subtree, is returned corresponding to the initial atom `S` of the pattern (i.e., corresponding to index 0). If instead of `S` we had used `(S)` or `(S died)` as the initial element of the pattern, then `(S died)` would have been returned corresponding to index 0; i.e., a nonatomic initial pattern element indicates that head words are to be returned. If the initial element of the pattern had *not* been followed by any further elements (i.e., if the pattern had been `(S)` or `((S))` or `((S died))`), the entire tree would have been returned. As already noted, without this distinction between non-singleton and singleton pattern specifications, we would always be forced to echo the entire input tree in the output (even when supplying an exhaustive set of parts on the association list).

Similarly for every subpattern: if the subpattern is a list of 2 or more elements, then no tree, only the atomic type symbol (plus head word(s) if "requested" via a non-atomic initial element), is returned for the index corresponding to the initial element of the subpattern. But if the subpattern is a singleton list, then the entire corresponding subtree is returned corresponding to its index (with the initial atom of the subtree expanded into a list including the head word(s) of that subtree, if "requested" via a non-atomic

initial subpattern element). In the example, the subpatterns `(NP)` and `((VP died))` are singleton lists, therefore complete NP- and VP-subtrees are returned corresponding to indices 1 and 2, with the initial type symbol expanded in the case of the VP into the list `((VP died))`, since inclusion of the head word(s) was "requested" via the initial VP-subpattern element `(VP died)` (where the inclusion of the specific word `died` also specifies that this must be the head word, or one of the head words, for a successful match).

The final subpattern, `(* (ADVP))`, is headed by the special atom "`*`", indicating "any number of". Since in the example there are no ADVPs in the given tree, the final element of the association list is just `(3)` – so, its `cdr`, which gives the corresponding retrieved sequence of trees, is `nil`. Note that it is important to treat special initial atoms in subpatterns, such as "`*`", distinctly from initial atoms such as `S`, `NP`, or `VP`, which indicate a phrase type.

As a variant of the above example, consider

```
(match-tree '(S ((NP)) ((VP died) (* (~ (ADVP))) (* (ADVP))))
            '(S (NP (DT the) (ADJP (JJ old)) (NN cat))
                (VP (VBD collapsed) (CC and) (VBD died)
                    (ADVP (ADV suddenly)) (ADVP (ADV yesterday)) ))
            0 )
===>

 ((0 S)
  (1 ((NP CAT) (DT THE) (ADJP (JJ OLD)) (NN CAT)))
  (2 (VP COLLAPSED DIED))
  (3 (VBD collapsed) (CC and) (VBD died))
  (4 (ADVP (ADV SUDDENLY)) (ADVP (ADV YESTERDAY))))
```

In this case the correspondence between the given pattern and the indices for returning the output is as follows:

```
(S ((NP)) ((VP died) (* (~ (ADVP))) (* (ADVP))))
(0   1        2             3           4  )
```

The second element of the pattern, `((NP))`, has a non-atomic initial subelement `(NP)`, hence this "requests" retrieval of the head word(s) for the NP, without requiring the head word(s) to be anything in particular. Since there are no further subelements, the entire NP-tree (with the augmented initial element) is returned corresponding to index 1.

The third element of the pattern, `((VP died) (* (~ ADVP)) (* ADVP))`, has subelements beyond the initial subelement `(VP died)`, so this means we should *not* return the VP as a whole corresponding to the initial subelement, only the type along with the head word(s) (which should include `died`). Thus the two head words `collapsed` and `died` of the VP are returned corresponding to index 2 as the `cdr` of the list `(VP collapsed died)`. For the subpattern `(* (~ ADVP))` the three successive subtrees that are *not* of type `ADVP` are returned, and for the subpattern `(* (ADVP))` the subtrees of type `ADVP` are returned.

Pattern syntax and semantics

Here is the pattern syntax in BNF (apart from the use of continuation dots "..."; curly brackets contain comments):

```
<pattern> ::= <tree-pattern> | <treelist-pattern>
<tree-pattern> ::= (<initial-el> <pattern> ... <pattern>) {0 or more patterns} |
                   (? <tree-pattern> ... <tree-pattern>) {1 or more patterns) |
                   (~ <tree-pattern> ... <tree-pattern>) {1 or more patterns} |
                   T {matches any tree}
<treelist-pattern> ::= (* <tree-pattern>) | (- <tree-pattern>)
<initial-el> ::= <phrase-type> | (<phrase-type> word ... word) {0 or more words}
<phrase-type> ::= <basic-phrase-type> | <generalized-phrase-type>
<basic-phrase-type> ::= S | NP | VP | NN | NNS | VBD | ...
<generalized-phrase-type> ::= N | V | ...
<word> ::= the | old | cat | collapsed | ...
```

The difference between tree-patterns and treelist-patterns is that the former apply to individual phrase structure trees, while the latter apply to lists of phrase-structure trees. Note that when sequentially matching successive subtrees of a given tree, we are working our way through a list of trees, and this perspective needs to be kept in mind when matching treelist patterns (starting with the Kleene star "*", or with the optionality symbol "-") against subtrees (further explanation to follow).

As explained in the examples, `<initial-el>` can be either atomic (a phrase type) or a list consisting of a phrase type followed by 0 or more words (specifying that head words are to be found, including any that are listed). In the first of the 4 `<tree-pattern>` types shown, the interpretation depends on whether there is at least one subpattern after the `<initial-el>`. If there is, then the given index will be assigned to the `<initial-el>`, and additional, higher indices to (appropriate chunks of) each of the subpatterns. If there are no subpatterns, then the entire tree (if successfully matched) is returned, associated with the given index.

The second of the 4 `<tree-pattern>` types gives a successful match if any of the tree-patterns following the "?" match the given tree, and the association list for the first successful match (starting from the left) is returned.

The third of the 4 `<tree-pattern>` types gives a successful match if *none* of the tree-patterns following the "~" match the given tree; in that case the given tree is returned on the output association list, associated with the given index. The fourth type of tree-pattern, `T`, has already been explained.

Treelist-pattern matching will be done in a manner that yields a leftmost pattern match: when trying to match a pattern of form `(- <tree-pattern>)`, we first try to continue on *without* matching `<tree-pattern>`, creating a partial association list `((i))`, where `i` is the index for the match corresponding to subpattern `(- <tree-pattern>)`. (So this associates the null list with `i`, as the `cdr` of `(i)`.) If the rest of the pattern match against the rest of the (sub)trees subsequently fails, we try to match `<tree-pattern>` against the first of the remaining (sub)trees, and then continue the pattern match against the remaining (sub)trees. Of course we must then use the result of he successful match against the subtree instead of `((i))` as the initial part of the association list.

Matching a pattern of type `(* <tree-pattern>)` is similar: we first try an empty match, and failing to complete the rest of the pattern match, we try to match `<tree-pattern>`

against the first of the remaining (sub)trees, and if that succeeds, we try to complete the match. However, in this case we need to retain the `*`-pattern on the list of patterns (so that potentially additional instances of the `*`-pattern can be matched).

Two potentially useful patterns worth noting that can be formed with the above syntax are `(- T)`, meaning "0 or 1 (sub)trees of any type", and `(* T)`, meaning '0 or more (sub)trees of any type".

We should also note that the given syntax allows for some rather "awkward" patterns, such as a `?`-pattern nested inside another `?`-pattern (which could be collapsed into a single `?`-pattern). However, allowing this keeps the syntax simpler than it would be if we tried to rule out such constructs, without sacrificing nested patterns such as `~`-patterns within `?`-patterns or within `*`-patterns.

Some algorithm details

The preceding syntactic definition and comments on semantics should already provide considerable guidance in how to design an algorithm for MATCH-TREE. Here is a possible pseudocode outline for the algorithm (without any syntax error checking). RETURN always means return from MATCH-TREE. When constructing lists we use Lisp-like notation, e.g., the list of elements p, q is written $(p\ q)$. (Do not read such lists as function application.) We also use the Lisp functions `cons` and `listp`, but with standard function application notation (not as in Lisp). Note that among the compound patterns, i.e., those starting with one of the 4 special symbols, the ones starting with `?` or `~` apply to individual trees rather than treelists, so it's probably best to handle them directly in MATCH-TREE. Those starting with `-` or `*` will be handled by a subroutine MATCH-TREES applicable to lists of trees (and co-recursive with MATCH-TREE).

```
MATCH-TREE(pattern,tree,index);
 {pattern is any tree-pattern, tree is a Treebank tree, and index >= 0}}
1.  IF pattern = T THEN RETURN ((index tree));
2.  init := first(pattern); {first list element; examples: S; (VP died)}
3.  subpatterns := rest(pattern); {all but the first list element}
    {First take care of compound tree-patterns starting with ? or ~}
4.  IF init = ? THEN RETURN match-alternatives(subpatterns,tree,index);
5.  IF init = ~
6.     THEN IF match-alternatives(subpatterns,tree,index) = nil
7.             THEN RETURN ((index tree)) ELSE RETURN nil;
    {Now handle non-compound tree-patterns}
8.  IF atom(init) THEN type := init ELSE type := first(init);
9.  IF not(subtype(first(tree),type) THEN RETURN nil; {fail}
10. IF listp(init) THEN {headwords required}
11.    BEGIN given-heads := rest(init);
12.          actual-heads := head-words-of(tree);
13           IF not(subset(given-heads,actual-heads))
14.             THEN RETURN nil; {fail}
15.          output-type := cons(type,actual-heads)
16.    END
17. ELSE output-type := type;
```

```
18. IF subpatterns = nil THEN RETURN ((index (output-type rest(tree))));
    {Now we need to deal with the subpatterns, matching them to subtrees}
19. subtrees := rest(tree);
20. assoc-list := match-trees(subpatterns,subtrees,index+1);
21. IF assoc-list = nil THEN RETURN nil ELSE
22.    RETURN cons((index output-type),assoc-list).
```

Note that we're assuming a subroutine MATCH-ALTERNATIVES that sequentially tries matching the tree-patterns in a given list of such patterns against a given tree, returning the association list for the first match it finds (if any). Naturally, this subroutine should recursively call MATCH-TREE.

The reason for a subtype check rather than equality check in line 9 is that we want to allow generalizations of atomic types in patterns, e.g., the generalization `V` for verb forms `VB, VBD, VBG, VBN, VBP,` and `VBZ` (see pattern syntax). Of course, we also want to allow equality, so we assume that every type is a subtype of itself.

The suggested algorithm for MATCH-TREE relies for much of its work on another algorithm, MATCH-TREES, for matching a *list* of patterns to a *list* of trees. This is the algorithm that needs to take care of treelist-patterns with "`-`" and "`*`" headers; of course, it should make recursive use of MATCH-TREE.

So as not to take all the fun out of this assignment, we leave some key details of MATCH-TREES to you `:-)` But see the earlier remarks on how to interpret and process the various pattern types, especially the treelist patterns. Begin by filling in details of the pseudocode for MATCH-TREES. This will count as part of the marks for this assignment! One small caveat is this: one might say that if we're matching an empty list of patterns against an empty list of trees then we should immediately succeed, returning association list `nil`. But `nil` is taken elsewhere to mean failure, so either we have to avoid inputs to MATCH-TREES where the first two arguments are `nil`, or else we have to use a special result such as `T` for this case, in which case the calling program should be able to "make sense" of this result (and not try to append `T` to another association list!). The route taken in the suggested pseudocode is the first one, i.e., we avoid dual `nil` inputs.

Another caveat is that the two pieces of pseudocode provided here have been written by a fallible human being (and not yet elaborated into full Lisp code and tested), and may have flaws; so you need to think it through, and make any necessary repairs as you elaborate, code up, and test the algorithms. Also, it would be good to include some syntax error checking, so that faulty input generates helpful error messages. In fact, you need not adhere to the pseudocode here at all for MATCH-TREE, if you prefer another approach. But in that case devise and show your own pseudocode, not just your Lisp code.

For the "head-hunting" function, you can use either your own code from the previous assignment or any code provided by the TA (or another student's head-hunting function, if OK'd by the TA).

```
MATCH-TREES(patterns,trees,index); {patterns, trees are not BOTH nil}

1.  IF (patterns = nil AND not(trees = nil)) THEN RETURN nil; {fail}
2.  pattern := first(patterns);
3.  is-tree-pattern := (pattern = T OR not(member(first(pattern),(- *))));
4.  IF is-tree-pattern
5.     THEN BEGIN IF trees = nil THEN RETURN nil {fail};
6.                tree := first(trees);
7.                assoc-list1 := match-tree(pattern,tree,index);
8.                IF assoc-list1 = nil THEN RETURN nil; {fail}
9.                IF rest(patterns) = rest(trees) = nil THEN {we're done}
10.                  RETURN assoc-list1;
11.               i := largest-index-in(assoc-list1) + 1;
12.               assoc-list2 := match-trees(rest(patterns),rest(trees),i);
13.               IF assoc-list2 = nil THEN RETURN NIL {fail} ELSE
14.                  RETURN append(assoc-list1,assoc-list2)
15.         END;
16. IF first(pattern) = -
       THEN BEGIN
            {etc... here we should succeed immediately with result ((i))
            if this is the last pattern and trees = nil; otherwise we
            should first try an empty match with result ((i)) and append
            the result of recursion (if successful), and if that fails,
            try a nonempty match and append the result of recursion
            (if successful)}
            END;
xx. {first(pattern) = *}
            {etc... again start by trying the empty-match result ((i))
            and finishing recursively, and if necessary trying a nonempty
            match and then finishing recursively (but with the *-pattern
            still in place}.
```

Demonstrating the capabilities of your program

This aspect is particularly important here. Demonstrate your understanding of the matching algorithms and the various types of patterns, and your understanding of the structure of Treebank trees, by providing a variety of examples of pattern matches against various sample trees, taken as before from

http://www.cs.rochester.edu/~schubert/247-447/assignments/brown-extract.

Indicate what features of the matcher you are illustrating with the various examples.

Write-up, etc.

The usual principles of good program design and documentation apply:

- The documentation should describe and explain the structure of the programs, providing detailed pseudocode and indicating any ways in which your approach differs from or augments the suggested method; 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;
- Be sure that your examples adequately demonstrate the capabilities of your program, as indicated above.
- State your conclusions about the effectiveness and coverage of your pattern matcher, and any ideas you have about how it could be improved.