TTT: A tree transduction language for syntactic and semantic processing

Abstract

In this paper we present the tree to tree transduction language, TTT. We motivate the overall ”template-to-template” approach to the design of the language, and outline its constructs, also providing some examples. We then show that TTT allows transparent formalization of rules for parse tree refinement, parse correction, predicate disambiguation, and refinement and verbalization of logical forms.

1 Introduction

Pattern matching and pattern-driven transformations are fundamental tools in AI. Many symbol manipulation tasks including operations on parse trees and logical forms, and even inference and aspects of dialogue and translation can be couched in the framework of pattern-directed transduction applied to list-structured symbolic expressions or trees.

The TTT system is directly applicable to concise and transparent specification of rules for such tasks, in particular (as we will show), parse tree refinement, parse tree correction, predicate disambiguation, logical form refinement, and verbalization of logical forms into English.

Parse tree refinement (for our purposes) has encompassed such tasks as distinguishing passive participles from past participles, temporal nominals from non-temporal ones, assimilation of verb particles into single constituents, deleting empty constituents, and particularizing prepositions. For example, standard treebank parses tag both past participles (as in “has written”) and passive participles (as in “was written”) as VBN. This is undesirable for subsequent compositional interpretation, as the meanings of past and passive participles are distinct. We can easily relabel the past participles as VBEN by looking for parse tree subexpressions where a VBN is preceded by a form of “have”, either immediately or with an intervening adverb or adverbial, and replacing VBN by VBEN in such subexpressions. Of course this can be accomplished in a standard symbol manipulation language like Lisp, but the requisite multiple lines of code obscure the simple nature of the transduction.

We have successfully used the new tree transduction system sketched below to repair malformed parses of image captions which were derived from the Charniak-Johnson parser (Charniak and Johnson, 2005). In particular, we have been able to repair systematic PP misattachments, at least in the limited domain of image captions. For example, a common error is attachment of a PP to the last conjunct of a conjunction, where instead the entire conjunction should be modified by the PP. Thus a statistical parse of the sentence “ Tanya and Grandma Lillian at her highschool graduation party” brackets as “Tanya and (Grandma Lillian (at her highschool graduation party.))”. We want to lift the PP so that “at her highschool graduation party” modifies “Tanya and Grandma Lillian”.

Another systematic error is faulty classification of relative pronouns/determiners as wh-question pronouns/determiners, e.g., “the student *whose* mother contacted you” vs. “I know *whose* mother contacted you” – an important distinction in compositional semantics. (Note that only the first occurrence, i.e., the relative determiner, can be paraphrased as *with the property that his*, and only the second occurrence, in which *whose* forms a wh-nominal, can be paraphrased as *the person with the property that his*.) An important point here is that detecting the relative-determiner status of a wh-word like *whose* may require taking account of an arbitrarily deep context. For example, in the phrase “the student in front of whose parents you are standing”, *whose* lies two levels of phrasal structure below the nominal it is semantically bound to. Such phenomena motivate the devices in TTT for detecting “vertical patterns” of arbitrary depth. Furthermore, we need to be able to make local changes ”on the fly” in matching vertical patterns, because the full set of tree fragments flanking a vertical match cannot in general be saved using match variables. In the case of a wh-word that is to be re-tagged as a relative word, we need to rewrite it *at the point where the vertical pattern matches it*, rather than in a separate tree-(re)construction phase following the tree-matching phase. .

We have been also able to perform Skolemization, conjunct separation, simple inference, and logical form verbalization with TTT and suspect its utility to logic tasks will increase as development continues.

A beta version of our system will be made available; however the URL is not included in this paper for anonymity.

2 Related Work

There are several pattern matching facilities available; however, none proved sufficiently general and perspicuous to serve our various purposes.

The three related tools Tgrep, Tregex, and Tsurgeon provide powerful tree matching and restructuring capabilities (Levy and Andrew, 2006). However, Tgrep and Tregex provide no transduction mechanism, and Tsurgeon's modifications are limited to local transformations on trees. Also, it presupposes list structures that begin with an atom (as in Treebank trees, but not in parse trees with explicit phrasal features), and its patterns are fundamentally tree traversal patterns rather than tree templates, and can be quite hard to read.

Peter Norvig's pattern matching language from *Paradigms of AI Programming* provides a nice pattern matching facility within the Lisp environment, allowing for explicit templates with variables (that can bind subexpressions or sequences of them), and including ways to apply arbitrary tests to expressions and to match boolean combinations of patterns. However, there is no provision for ”vertical” pattern matching or subexpression replacement ”on the fly”. TTT supports both horizontal and vertical pattern matching, and both global (output template) and local (on the fly) tree transduction. Also the notation for alternatives, along with exclusions, is more concise than in Norvig's matcher, for instance not requiring explicit ORs.

Mathematica also allows for sophisticated pattern matching, including matching of sequences and trees. It also includes a sophisticated expression rewriting system, which is capable of rewriting sequences of expressions. It includes functions to apply patterns to arbitrary subtrees of a tree until all matches have been found or some threshold count is reached; as well, it can return all possible ways of applying a set of rules to an expression. However, as in the case of Norvig's matcher there is no provision for vertical patterns or on-the-fly transduction. (Wolfram Research, Inc, 2010)

Snobol, originally developed in the 1960's, is a language focused on string patterns and string transformations (Griswold, 1971). It has a notably different flavor to the other transformation systems. Its concepts of cursor and needle support pattern based transformations which rely on the current position in a string at pattern matching time, as well as the strings which preceding patterns matched up to the current point. Snobol also supports named and thereby recursive patterns. While it includes recognition of balanced parenthesis, the expected data type for Snobol is the string – leaving it a less than direct tool for intricate manipulation of trees. An Python extension SnoPy adds Snobol's pattern matching capabilities to Python. (Rozenberg, 2002)

Haskell also includes a pattern matching system, but it is weaker than the other systems mentioned. The patterns are restricted to function arguments, and are not nearly as expressive as Mathematica's for trees nor Peter Norvig's system or Snobol for strings. (Hudak et. al, 2000)

3 TTT

Pattern Matching

Patterns in TTT are hierarchically composed of sub-patterns. Literal elements (such as atomic symbols or tree fragments) are patterns which match only themselves. More complicated patterns are constructed through the use of pattern operators. Most pattern operators require arguments, but some may appear free-standing. Each pattern operator has a unique method of directing a match according to its supplied arguments. The pattern syntax has been chosen to directly reflect the structure of the trees to be matched. Transductions are specified by a special pattern operator and will be described in the next section.

The ten basic pattern operators are:

- `!` - match exactly one sub-pattern argument
- `+` - match a sequence of one or more arguments
- `?` - match the empty sequence or one argument
- `*` - match the empty sequence or one or more arguments
- `{}` - match any permutation of the arguments
- `<>` - match the freestanding sequence of the arguments
- `^` - match a tree which has a child matching one of the arguments
- `^*` - match a tree which has a descendant matching one of the arguments
- `^@` - match a vertical path
- `/` - transduction operator (explained later)

Negation: The operators `!`, `+`, `?`, `*`, and `^` support negated patterns. Matching a negated pattern causes the overall match to fail.

Iteration: The operators `!`, `+`, `?`, `*`, and `^` also support iterative constraints. This enables one to write patterns which match exactly n, at least n, at most n, or from n to m times, where n and m are integers. Eg. `(![3] A)` would match the sequence `A A A`.

Unconstrained Patterns: The first four pattern operators may also be invoked without arguments, as: `_!`, `_+`, `_?`, `_*`. These match any single tree, any non-empty sequence of trees, the empty sequence or a sequence of one tree, and any (empty or non-empty) sequence of trees.

Vertical Paths: The vertical path operator `(^@ X1 X2 ... Xm)` matches a tree which has root matching `X1`, a child matching `X2`, which in turn has a child that matches `X3`, and so on.

Any of the arguments to a pattern operator may be composed of arbitrary sub-patterns.

Bindings: Operators may be bound, in that the tree sequence which was matched by an operator is retained in a variable. The variable names may be specified by appending additional information to the operator names (i.e. `_!1`, `_!a`, `_!2a`, ...).

Constraints on Bindings Arbitrary computable constraints on bindings are supported. [Example? –NOT REALLY IMPLEMENTED– satisfies-constraints always returns bindings for now]

Sticky Variables: Variables may be specified as sticky or non-sticky, where sticky variables which appear more than once in a pattern are constrained to match structurally identical tree sequences to the first occurrence of the variable at each location.

Predicates Arbitrary predicates can be used during the pattern matching process (and consequently the transduction process). Symbols with names ending in the question mark, and with associated function definitions, are interpreted as predicates. When a predicate is encountered during pattern matching, it is called with the current subtree as input. The result is only inspected as nil/non-nil, and when nil is returned the current match fails. [Predicates are not bound to right now! Predicates with pattern-level arguments are not supported!]

Pattern Details and Examples Pattern Examples:

- `(! (+ A) (+ B))` - matches a non-empty sequence of `A`'s or a non-empty sequence of `B`'s, but not a sequence containing both
- `(* (<> A A))` - matches a sequence of an even number of `A`'s
- `(B (* (<> B B)))` - matches a sequence of an odd number of `B`'s
- `(({} A B C))` - matches `(A B C)` `(A C B)` `(B A C)` `(B C A)` `(C A B)` and `(C B A)` and nothing else
- `((<> A B C))` - matches `(A B C)` and nothing else

- `(^* X)` - matches any tree with descendant `X`
- `(^@ (+ (@ _*)) X)` - matches a tree with leftmost leaf `X`

Binding Examples:

Pattern	Tree	Bindings
_!	(A B C)	(_! (A B C))
(A _! C)	(A B C)	(_! B)
(_* F)	(A B (C D E) F)	(_* A B (C D E))
(A B _? F)	(A B (C D E) F)	(_? (C D E))
(A B _? (C D E) F)	(A B (C D E) F)	(_?)
(^@ _! (C _*) E)	(A B (C D E) F)	(^@ (A B (C D E) F)) (_* D E)
(A B (<> (C D E)) F)	(A B (C D E) F)	(<> (C D E))
(A B (<> C D E) F)	(A B (C D E) F)	fail

Transductions

Transductions are specified with the transduction operator, `/`, which takes two arguments. The left argument may be any tree pattern and the right argument may be constructed of literals, variables from the lhs pattern, and function calls [NEED EXAMPLE; SHOULD I MENTION APPLY!?]. [MENTION LOCAL TRANSDUCTIONS FOR PARALLELISM]

Transductions may be applied to the roots of trees, subtrees, at most once, or until convergence. When applying transductions to arbitrary subtrees, they are searched top-down, left to right. When a match to the transduction lhs pattern occurs, the resulting bindings and transduction rhs are used to create a new tree, which then replaces the tree which matched.

Here are a few examples of simple template to template transductions:

- `(/ X Y)` - replaces the symbol `X` with the symbol `Y`
- `(/ (! X Y Z) (A))` - replaces any X, Y, or Z with A
- `(/ (! X ) (! !))` - duplicates an X
- `(/ (X _* Y) (X Y))` - remove all subtrees between X and Y
- `(/ (_! _* _!1) (_!1 _* _!))` - swaps the subtrees on the boundaries

A transduction operator may appear nested within a composite pattern. The enclosing pattern effectively restricts the context in which the transduction will be applied, because only a match to the entire pattern will trigger a transduction. In this case, the transduction is applied at the location in the tree where it matches. The rhs of such a transduction is allowed to reference the bindings of variables which appear in the enclosing pattern. We call these local transductions, to distinguish from whole-tree replacement. Local transductions are especially advantageous when performing vertical path operations, and have a very concise syntax. For example, the transduction `(^@ (* ((! S SBAR) _+)) (/ (WH _!) (REL-WH (WH _!)))))` applied to the tree `(SBAR (S (S (WH X) B) A))` yields the new tree `(SBAR (S (S (REL-WH (WH X)) B) A))`. Additional examples appear later (especially in the parse tree refinement section). [SHOULD I MENTION NON-ITERATED ENCLOSING PATTERNS?] [SIMPELR EXAMPLE?] [WHAT ABOUT LONG-DISTANCE BINDINGS SUCH AS (_! (/ _!1 _!))?]

TTT also supports functions, with bound variables as arguments, in the rhs templates, such as `join-with-dash!`, which concatenates all the bound symbols with intervening dashes. E.g. the transduction
“`(/ (PP (IN _!)) ((join-with-dash! PP _!) (IN _!)))`” applied to the subtree `(PP (IN FROM))` yields `(PP-FROM (IN FROM))` and applied to the subtree `(PP (IN TO))` yields `(PP-TO (IN TO))`. [DO I NEED A SIMPLER EXAMPLE?] One can imagine additional functions, such as `reverse!`, `l-shift!`, `r-shift!`, or any other function of a list of nodes which is useful to the application at hand. Symbols with names ending in the exclamation point, which are associated with function definitions, and appear as the first element of a list are executed during template construction.

Relations between TTT and formal models

[RENAME AS THEORETICAL PROPERTIES?] A good overview of the dimensions of variability among formal tree transducers is given in Capturing practical natural language transformations, Keven Knight. The main properties are restrictions on the height of the tree fragments allowed in rules, linearity, and whether the rules can delete arbitrary subtrees. Among the more popular and recent ones, synchronous tree substitution grammars (STSG), synchronous tree sequence substitution grammars (STSSG), and multi bottom-up tree transducers (MBOT) constrain their rules to be linear and non-deleting, which is important for efficient rule learning and transduction execution (Chiang, 2004; Galley et. al, 2004; Yamada and Knight, 2001; Zhang et. al, 2008; Maletti, 2010). The language TTT does not have any such restrictions. While this flexibility does increase worst-case computational complexity of some operations, TTT is intended to be a full programming language, enabling the user to direct powerful and dramatic transformations on trees. In fact, TTT is Turing complete, as we will soon show.

Additionally, pattern predicates, binding constraints, and function application in the right hand sides of rules are features present in TTT which are not included in the above formal models.

Turing Completeness: An informal argument

A Turing machine is a 5-tuple $(\Sigma, Q, F, q_0, \delta)$, where Σ is a finite alphabet, Q is a finite state set, $F \subseteq Q$ is a set of accepting states, q_0 is the start state, and $\delta : Q \times \Sigma \Rightarrow Q \times \Sigma \times \{L, R\}$ is the transition function. A Turing machine is equipped with a double-ended infinite tape (only finitely many cells of which may be non-blank at any particular time) and a movable head, which slides from cell to cell on the tape according to the current state, current symbol on the cell under the head, and the corresponding $\{L, R\}$ entry in the transition table. The Turing machine begins in the start state, and halts once it reaches a halting state (one with no transitions out). If the halting state is also an accepting state, then the original string contents of the tape is said to be accepted, otherwise it is said to be rejected. Not every Turing machine halts on all inputs.

In order to show Turing equivalence of TTT, we must show how to simulate an arbitrary Turing machine with transduction rules. Each element of the finite state sets and alphabet can easily be represented by symbols in Lisp. The tape can be represented as a sequence of height 1 trees. The state and head position can be encoded into the symbol sequence by wrapping the symbol under the head in a list, with the first element being the state.

Let the current state be q, the symbol under the head s, the symbol to output r, the next state p for an arbitrary transition. The transition table can be encoded into TTT rules as follows, according to the head direction specified by the rule: (q, s, r, p, L) becomes `(/ (_* _!l (q s) _!r _*r) (_* (p _!l) r _!r _*r))` (q, s, r, p, R) becomes `(/ (_* _!l (q s) _!r _*r) (_* _!l r (p _!r) _*r))`

Anotating the first element of the input sequence via the rule ((/ _? (q_0 _?)) _*) ensures that TTT's simulation begins in the start state.

The stipulation that no moves originate from the halting state, and that only one state exists in the sequence at a time, forces TTT to halt when appropriate.

Nondeterminism and noncommutativity: In general, given a set of transductions (or even a single transduction) and an input tree there may be several ways to apply the transductions, resulting in different trees. This phenomenon comes from two sources:

- rule application order - transductions are not in general commutative
- bindings - a pattern may have many sets of consistent bindings to a tree (E.g. pattern `(_* _*1)` can be bound to the tree `(X Y Z)` in four distinct ways).
- subtree search order - a single transduction may be applicable to a tree in multiple locations [Ex: `(/ _! X)` could replace any node of a tree, including the root, with a single symbol].

Therefore some trees may have many reduced forms with respect to a set of transductions (where by reduced we mean a tree to which no transductions are applicable) and even more reachable forms.

[I DON'T LIKE THIS SECTION, I FEEL LIKE PROBABILISTIC SEARCH IS NOT TREATED IN SUFFICIENT DETAIL AND WHAT WE ACTUALLY DO SOUNDS LIKE A COP OUT NEXT TO IT]

One can imagine a few ways to tackle this:

- Exhaustive exploration - Given a tree and a set of transductions, compute all reduced forms. [note: it is possible for this to be an infinite set, so a lazy computation may be necessary.] Mathematica provides this style of feature with its expression rewriting system.

- Probabilistic search - Assign weights to transductions, where the resulting trees are weighted according to the product of the weights of the rules applied, starting with a fixed weighed source tree.

- What we actually do - Given a tree and a list of transductions, for each transduction (in order), apply the transduction in top-down fashion in each feasible location (matching lhs), always using the first binding which results from a "left most" search.

The first method has the unfortunate effect of transducing one tree into many (bad for parse repair, probably bad for other applications as well). The latter method is particularly reasonable when your set of transductions is not prone to interaction or multiple overlapping bindings. We intend to implement an "all-reductions" method which would parallel the ReplaceAll function of Mathematica. [SHOULD I CUT THIS?]

4 Some illustrative examples

Working with Parse Trees

Refinement: To distinguish between past an passive participles, we want to search for the verb has, and change the participle token correspondingly. These two transductions are equivalent, the first is global and the second is an example of a local or on-the-fly transduction. Observe the more concise form, and simpler variable specifications of the second transduction.

```
  (/ (VB _* (VBZ HAS) _*1 (VBN _!)
_*2) (VB _* (VBZ HAS) _*1 (VBEN
_!) _*2))
  (VB _* (VBZ HAS) _* ((/ VBN
VBEN) _!) _*)
```

Here is a simple transduction to delete empty constituents, which sometimes occur in the Brown corpus [CITE THIS?].

```
  (/ (_* () _*1) (_* _*1))
```

pTo distinguish temporal and non-temporal nominals, we use a predicate function to detect temporal nouns, and then annotate the NP tag accordingly. [WHICH EXAMPLE SHOULD I KEEP, GLOBAL OR LOCAL?]

```
  (/ (NP _* nn-temporal?)
(NP-TIME _* nn-temporal?))
  ((/ NP NP-TIME) _*
nn-temporal?)
```

Assimilation of verb particles into single constituents is useful to semantic interpretation, and is accomplished with the transductions: `(/ (VP (VB _!1) ( (PRT (RP _!2)) (NP _*1))) (VP (VB _!1 _!2) (NP _*1)))`

We often particularize PPs to show the preposition involved, e.g., PP-OF, PP-FROM, etc. Note that this transduction uses the `join-with-dash!` function, which enables us to avoid writing a separate transduction for each preposition. `(/ (PP (IN _!) _*1) '((join-with-dash! PP _!) (IN _!) _*1))`

We also change (PP (TO TO) ...) to (PP-TO (IN TO) ...) (since the WSJ annotations don't distinguish preposition TO and verb TO!) [CITE PTB? TAKE THIS OUT? ARE THESE TRANSDUCTION OK? (NO GAPS BETWEEN PP AND (IN _!))] `(/ (PP (TO TO) _*) (PP-TO (IN TO) _*))`

Statistical Parse Repairs: - parse tree correction e.g., correcting certain systematic PP misattachments, at least for certain applications (ours was caption processing); e.g., misattachment of certain types of PPs to the last conjunct of a conjunction, where instead the entire conjunction should be modified by the PP adjunct; (give specific example desired kind of transduction) " Tanya and Grandma Lillian at her highschool graduation party"

Working with Logical Forms

Skolemization: [SHOULD I MENTION EPILOG? – it would compromise anonymity but if I'm using it here...]

We wrote the function `subst-new!` to replace all occurrences of a symbol in an expression with a new one, consistently labeled as such. It uses a TTT transduction to accomplish this. Skolemization is then performed via the transduction `(/ (EXISTS _! _!1 _!2) (subst-new! _! (_!1 and.cc _!2))))`

Inference: We use the following rule to accomplish inferences such as if most things with property X have property Y, and most things with property Y have property Z, then many things with property X also have property Z.

```
  (/ ( _* (most _!.x (_!.x (!.p
pred?)) (_!.x (!.q pred?))) _*
(most _!.x (_!.x !.q) (_!.x (!.r
pred?))) _*) (many _!.x (_!.x
!.p) (_!.x !.r)))
```

[mention reversed order?]

Predicate Disambiguation: The following rules disambiguate between various senses of have (e.g. as-part, as-possession, as-eating-food, as-experience, as-feature):

```
(/ ((det animal?) have.v (det
animal-part?)) (all-or-most
x (x animal?) (some e ((pair
x e) enduring) (some y (y
animal-part?) ((x have-as-part.v
y) ** e)))))
  (/ ((det agent?) have.v (det
possession?)) (many x (x agent?)
(some e (some y (y possession?)
(x possess.v y) ** e))))
  (/ ((det animal?) have.v (det
food?)) (many x (x animal?)
(occasional e (some y (y food?)
(x eat.v y) ** e))))
  (/ ((det person?) have.v (det
event?)) (many x (x person?)
(occasional e (some y (y event?)
((x experience.v y) ** e)))))
  (/ ((det agent?) have.v (det
property?)) (many x (x agent?)
(some e ((pair x e) enduring)
(some y ((y (apply! append-of
property?)) x) ** e))))
```

Logical Interpretation:

The following transductions directly map from parse trees to an intermediate logical form:

```
(/ (S (NP (DT the) _!) (VP _+))
(some x (x _!) _+)) (/ (some X
(X _!.x) (AUX IS) _!.y) (some x
(x _!.x) (x _!.y))) (/ (_*.a (NNP
_!.x) (NNP _!.y) (NNP _!.z) (NNP
_!.f) _*.b) (_*.a (NNP _!.x _!.y
_!.z _!.f) _*.b)) (/ (_*.a (NNP
_!.x) (NNP _!.y) (NNP _!.z) _*.b)
(_*.a (NNP _!.x _!.y _!.z) _*.b))
(/ (_*.a (NNP _!.x) (NNP _!.y)
_*.b) (_*.a (NNP _!.x _!.y) _*.b))
(/ (NNP _+) (make-name! (_+))) (/
(NN _!) (make-noun! _!)) (/ (JJ
_!) (make-adj! _!)) (/ (ADJP _!)
_!) (/ (NP _!) _!) (/ (S _!.x (vp
(aux _!.f) (np (dt _!.y) _!.z)))
(_!.x _!.z))
```

5 Conclusion

The TTT language is well-suited to the applications it was aimed at, and is already proving useful in current syntactic/semantic processing applications. It provides a very concise, transparent way of specifying transformations that previously required extensive symbolic processing. Remaining issues (e.g., efficient access to rules that are locally relevant to a transduction; ...).

The language also holds promise for rule-learning, thanks to its simple template-to-template basic syntax. The kinds of learning envisioned are learning parse-tree repair rules, and perhaps also LF repair rules and LF-to-English rules (which is made plausible by the very English-like syntax of LF in Episodic Logic).

References

Eugene Charniak and Mark Johnson. 2005. Coarse-to-Fine n-Best Parsing and MaxEnt Discriminative Reranking. *Proceedings of the 43rd Annual Meeting of the Association for Computational Linguistics* (ACL'05), 173–180. Association for Computational Linguistics, Ann Arbor, MI, USA.

David Chiang. 2004. Evaluation of Grammar Formalisms for Applications to Natural Language Processing and Biological Sequence Analysis. Phd. Thesis. University of Pennsylvania.

Michel Galley and Mark Hopkins and Kevin Knight and Daniel Marcu 2004. What's in a Translation Rule?. *Proceedings of the 2004 Meeting of the North American chapter of the Association for Computational Linguistics* (NAACL '04), 273–280. Boston, MA, USA.

Ralph Griswold 1971. The SNOBOL4 programming languge. Prentice-Hall, Inc. Upper Saddle River, NJ, USA.

Paul Hudak, John Peterson, and Joseph Fasel. 2000. A Gentle Introduction To Haskell: Version 98. Los Alamos National Laboratory. http://www.haskell.org/tutorial/patterns.html.

Roger Levy and Galen Andrew. 2006. Tregex and Tsurgeon: tools for querying and manipulating tree data structures. Language Resources Evaluation Conference (LREC '06).

Andreas Maletti. 2010. Why synchronous tree substitution grammars?. Human Language Technologies: The 2010 Annual Conference of the North American Chapter of the Association for Computational

Linguistics (HLT '10). Association for Computational Linguistics, Stroudsburg, PA, USA.

Don Rozenberg 2002. SnoPy - Snobol Pattern Matching Extension for Python. http://snopy.sourceforge.net/user-guide.html.

Wolfram Research, Inc. 2010. Wolfram Mathematica 8 Documentation. Champagne, IL, USA. http://reference.wolfram.com/mathematica/guide/RulesAndPatterns.html.

Kenji Yamada and Kevin Knight 2001. A Syntax-Based Statistical Translation Model. *Proceedings of the 39th Annual Meeting on Association for Computational Linguistics* (ACL '01), 523–530. Stroudsburg, PA, USA.

Min Zhang and Hongfei Jiang and Aiti Aw and Haizhou Li and Chew Lim Tan and Sheng Li 2008. A tree sequence alignment-based tree-to-tree translation model. *Proceedings of the 46th Annual Meeting of the Association for Computational Linguistics* (ACL '08).