A System for Testing Synchronous Tree-Adjoining Grammar Analyses of Linguistic Phenomena

Jennifer Hu

February 2, 2017

Contents

1 Getting Started

This system automates the derivations specified by a synchronous multicomponent tree-adjoining grammar, allowing the linguist to rigorously test his/her analyses of linguistic phenomena. The user begins by specifying a high-level description of the grammar following a notation for trees, links, etc. described in the Metagrammar section of this document. The system then parses this textual input, performs the user-specified derivations, and outputs the derived trees in an intuitive bracketed notation. See the Usage section for more details.

1.1 Acknowledgments

This system was developed in summer 2016 with Professor Stuart M. Shieber at Harvard University.

1.2 Prerequisites & Background

- OCaml 4.02 (or later) and Ocaml Menhir are required. Once OCaml is installed, Menhir can be installed through a simple `opam install menhir` command.
- For more background information on tree-adjoining grammars, the following papers are a good place to start:
  - Joshi, A. K., & Schabes, Y. (1997). Tree-adjoining grammars. In *Handbook of formal languages* (pp. 69-123). Springer Berlin Heidelberg.
  - Shieber, S. M., & Schabes, Y. (1990, August). Synchronous tree-adjoining grammars. In *Proceedings of the 13th conference on Computational linguistics-Volume 3* (pp. 253-258). Association for Computational Linguistics.

2 Usage

2.1 Building the Project

To build all the files for the first time, run `ocamlbuild -use-menhir mctag.byte`. A sample grammar, specified in `in.txt`, has already been compiled and is ready to run with the command `./out.byte`.

2.2 Writing Your Own Grammars

- To learn the metagrammar rules, refer to the Metagrammar section of this document. This document also contains a toy grammar to help the user get started.
- The files `in.txt`, `test.txt`, and `paper.txt` contain full-sized grammars dealing with a variety of linguistic phenomena such as reflexives, topicalization, features unification, and quantifier scope. They can be found in the `grammars` folder. The corresponding output files `out.ml`, `test.ml`, and `paper.ml` are all ready to be compiled and run.

2.3 Using the Project

- The user must input the grammar and derivation trees in an input file following the convention described in the section "The Metagrammar."
- The user can then choose to either directly perform the derivations and print the resulting derived treesets to standard output (`interpret` mode) or compile the forest and derivations into an `.ml` file and separately compile/run the output file (`compile` mode).
- To interpret, run `./mctag.byte -i <input file>`.

- To compile, run `./mctag.byte -c <input file> <output file>`, then `ocamlbuild <output file>.byte` to build the output file. To run the output file, use `./<output file>.byte`.
- For a full list of options at any time, run `./mctag.byte -help`.

3 The Metagrammar

The metagrammar provides the convention for specifying a multicomponent tree-adjoining grammar and its derivation trees. This section is broken into three subsections, which contain general rules as well as concrete examples highlighted in blue.

3.1 General

- Spaces, tabs, and newlines are ignored.
- Supports nested comments delimited by curly braces {}.
- Can use any alphanumeric character, hyphens, and underscores for identifiers and lexical items. The user should note that characters with special OCaml semantics will cause issues when building the compiled output file, e.g. - and /. This shouldn't matter for interpreting.

3.2 Forests

3.2.1 Trees

- Trees are recursively contained within brackets `[]`.
- Require a category and can include an operation marker, links, features, kids, or a lexical item.

```
[s@1@2 [np!@1 (case:nom)] [vp@3 [v 'likes'] [np!@2 (case:acc)]]]
```

3.2.2 Words

- Lexical items are delimited by single quotation marks.
- Note that multi-word lexical items, such as "each other" or "john and mary", should not contain spaces, as they will not be properly parsed.

```
[np 'john_and_mary']
```

3.2.3 Quantifiers and Variables

- Quantifiers themselves are treated similarly to categories, but a quantifier tree takes in a variable and two propositions (trees).
- Bound variables are preceded by a $ and have scope in the treeset.

```
[exists $x [t [et 'person'] [e $x]] [t*]]]
```

3.2.4 Substitution Nodes

- Substitution nodes are marked with a `!` after the category.

```
[np!]
```

3.2.5 Foot Nodes

- Foot nodes are marked with a * after the category.

```
[vp [adv 'apparently'] [vp*]]
```

3.2.6 Links

- Links signify relationships between operable sites across the treeset.
- Specified by `@<link index>`, where the index is represented as an int.
- Note that a node can have multiple links, a single link, or no link at all.

```
[s@1 [np!@1] [vp [v 'sleeps']]]
```

3.2.7 Features

- A feature structure is represented as a semicolon-separated list of (feature,value) pairs, where each pair is represented by `<feature> : <value>`. The whole list is then delimited by parentheses `()`.
- To specify top features for a node, simply use this notation, and to specify bottom features for a node, precede the list with a period.
- If no features are specified, the parser will create empty feature structures.

```
[np@1 .(num : sg; per : 3) 'bill'], [e!@2 (case: acc)]
```

3.2.8 Treesets

- A treeset is a list of trees, preceded with `SET <id>:`, separated by semicolons, and ended with a period.

```
SET himself:
    [s*@1] ; [s*@2] ;
    [np@2 .(case : acc; num : sg ; per : 3) 'himself'] ; [np!@1] ;
    [t@1 [et [eetet 'refl/recp'] [eet [lambda $a] [et [lambda $b] [t*]]]] [e!@1]] ;
    [t*@2] ; [e@2 $a] ; [e $b].
```

3.2.9 Forests

- A forest is a list of treesets, preceded with `FOREST <id> =`.

```
FOREST example =
    SET jenn:
        [np 'jenn'] ; [e 'j'].
    SET cristina:
        [np 'cristina']; [e 'c'].
```

3.3 Derivations

3.3.1 Basic Derivation Trees

- Derivation trees are preceded with `DERIV <id>:` and ended with a period.
- The actual recursive structure contains a lexical item, which identifies a treeset in the forest, and a list of links matched with the sets meant to operate at those links.

- The link, represented as `@<link index>`, is followed by a colon `:` and the name of the set in brackets `[]`.
- Every link in the host treeset must be specified, even if it is empty (e.g. `@3:[]` in the example below). If you forget to do this, the system will raise the exception `Not_found`.

```
{ Derives the topicalized sentence "Everyone someone likes (t)" }
DERIV everyone_top:
    likes
        @1:[someone]
        @2:[everyone_top]
        @3:[].
```

3.3.2 Multiple Sets per Link

- To specify that multiple sets operate at a single link, concatenate the sets with a plus +.

```
{ Derives the sentence "John quite apparently likes Mary" }
DERIV johnlikesmary:
    likes { "quite" and "apparently" both use link 3 in the "likes" treeset }
        @1:[john @1:[]]
        @2:[mary @1:[]]
        @3:[quite]+[apparently].
```

3.3.3 Scope

- To specify different scope orderings when two trees could adjoin at the same node, simply order the set with higher scope before the set with lower scope.

```
DERIV forall_exists: { forall > exists }
    likes
        @2:[everyone]
        @1:[someone]
        @3:[].
DERIV exists_forall: { exists > forall }
    likes
        @1:[someone]
        @2:[everyone]
        @3:[].
```

3.3.4 Multiple Links per Set

- To specify that a set operates at multiple links, concatenate the links with a plus +.
- The system will test different partitions of the set across the links to find one that matches the categories and operations specified by the target treeset.

```
{ Derives the sentence "John saw himself" }
DERIV refl_john:
    saw { The reflexive set operates at links 1 and 2 in the verb set }
    @1+@2:[himself
            @1:[john @1:[]]
            @2:[]]
            @3:[].
```

3.3.5 Priority Orderings

- Ambiguities can arise when there are multiple ways to partition a set that is meant to be used across multiple links. We address this issue by (1) allowing the user to provide a priority ordering of trees in the treeset, and (2) generating permutations of the trees in the set in lexicographic order, i.e. in increasing number of inversions. This ensures that in the case of ambiguous orderings of trees, the system chooses the one closest to the ordering preferred by the user.
- The ordering is specified by a semicolon-separated list of tree indices, represented as ints, and delimited by parentheses `()`. It should come after the name of the set in the derivation.
- If no order is specified, we simply use the order of the trees specified in the grammar.

```
{ Both derive the sentence "John saw himself", but avoids ambiguities }
DERIV refl_john1:
    saw
        @1+@2:[himself (1;2;3;4;5;6;7;8) { Optional, as it matches the grammar }
            @1:[john @1:[]]
            @2:[]]
        @3:[].
DERIV refl_john2:
    saw
        @1+@2:[himself (2;1;3;4;5;6;7;8)
            @1:[john @1:[]]
            @2:[]]
        @3:[].}
```

4 Toy Grammar

```
{ Derives "John saw Mary" with features }
FOREST toy =
    SET john:
        [s*] ; [np@1 .(num:sg ; per:3) ‘john’] ;
        [t*] ; [e@1 .(num:sg ; per:3) ‘j’].
    SET mary:
        [s*] ; [np@1 .(num:sg ; per:3) ‘mary’] ;
        [t*] ; [e@1 .(num:sg ; per:3) ‘m’].
    SET saw:
        [s@1@2 [np!@1 (case:nom) ] [vp@3 [v ‘saw’] [np!@2 (case:acc)]]] ;
        [t@1@2@3 [et [eet ‘saw’] [e!@2 (case:acc)]] [e!@1 (case:nom)]].
DERIV johnsawmary:
    saw
        @1:[john @1:[]]
        @2:[mary @1:[]]
        @3:[].
```

5 Code Structure

The code is organized into the following files, grouped together by the purpose they serve in the project. For more information on any of the files, please refer to the comments in the code.

5.1 Sample Input & Output Files

- `in.txt` & `out.ml`

Contains analyses of reflexives, quantifier scope ambiguity, topicalization, adjunction at substitution nodes, and more.

- `paper.txt` & `paper.ml`

  Contains analyses from Cristina's paper.

- `test.txt` & `test.ml`

  Contains analyses of basic feature structures. A full version of Cristina's trees with features is commented out in the file.

5.2 Parsing & Lexing

- `myLexer.ml` & `myLexer.mll`

  Lexer. Contains rules for separating the stream of characters from the input file into tokens.

- `myParser.ml` & `myParser.mly`

  Parser. Contains rules for assigning semantic content to these tokens.

5.3 Classes, Objects, & Derivations

- `unify.ml`

  The `fs` class, which corresponds to feature structures.

- `tree.ml`

  The `tree` class.

- `links.ml`

  The `links` class, which creates a reference for the locations and categories associated with every instance of every link in a treeset.

- `treeset.ml`

  The `treeset` class.

- `forest.ml`

  The `forest` class, which contains a list of `treesets`.

- `argument.ml`

  The `argument` class, which specifies the relationship between `treesets` and the sets of links where they are expected to operate.

- `derive.ml`

  Contains the functions needed to actually perform MCTAG derivations, including the substitution and adjunction operations.

- `derivation.ml`

  The `derivation` class, which corresponds to a derivation tree.

- `parsed.ml`

  The `parsed` class, which contains a `forest` and a list of `derivations`.

5.4 Miscellaneous

- `basics.ml`

  Contains helper functions for basic data types like lists and strings, including functions for generating combinations/permutations and printing.

- `mctag.ml`

  Contains the main function and parses the command-line arguments.

6 Known Issues

As this project is still in process, there are a few known issues that we are actively trying to resolve. Below is a list of these issues with their temporary workarounds. Any comments, suggestions, or questions are welcome at jenniferhu@college.harvard.edu.

- Feature structure unification does not work for variables across a treeset yet, but this should be resolved soon.
- As features are unified and destructively modified during every derivation, multiple derivations using the same treeset cannot properly be performed. For the time being, I only perform derivations one at a time, but I am working on creating copies of features structures to allow for multiple derivations.