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

Jennifer Hu

November 17, 2018

Contents

1 Getting Started

This system automates the derivations specified by a synchronous multicomponent tree-adjoining grammar (STAG). The user begins by specifying a high-level description of the grammar, using the notation described in Section 2. The system then parses this textual input, performs the user-specified derivations, and prints the derived trees in an intuitive bracketed form.

For an introduction to TAGs, Joshi & Schabes (1997) is a good place to start.

1.1 Acknowledgments

This system was developed in collaboration with Stuart Shieber and Cristina Aggazzotti during the summer of 2016. I was supported by the Harvard College Program for Research in Science and Engineering.

1.2 Dependencies

You will need OCaml 4.02 (or later) and Menhir. Refer to the OCaml docs for installation instructions. If you install the package manager OPAM with OCaml, Menhir can be installed with `opam install menhir`.

You may compile the files with your favorite OCaml compiler, but I suggest Ocamlbuild, which can be installed through OPAM with `opam install ocamlbuild`.

1.3 Building

To build all the files for the first time, run the command `ocamlbuild -use-menhir mctag.byte`.

This command creates a `_build` directory within your current working directory. The executables will be stored there, but a symlink should be created in your current directory. If you can't find the target `mctag.byte` file, look under `_build`.

1.4 Usage

To use the system, you must specify the grammar and derivation trees in an input file following the convention described in Section 2. Run the command `./mctag.byte -i <input file>` to perform the derivations and print the resulting derived treesets to standard output. For a full list of options at any time, run `./mctag.byte -help`.

As a test, try running `./mctag.byte -i grammars/toy.txt`. If you see the derived syntax and semantics trees for "John saw a girl", then the system is working!

1.5 Sample Grammars

The quickest way to learn the metagrammar may be to tweak existing example grammars. The project comes with the following sample grammars, which illustrate a wide variety of linguistic phenomena:

Path	Description
`grammars/toy.txt`	Toy grammar to derive 'John saw a girl'
`grammars/example.txt`	Full grammar with analyses of reflexives, topicalization, quantifier scope ambiguity, adjunction at substitution nodes, etc.
`grammars/paper.txt`	Grammar to derive trees from Aggazzotti & Shieber (2017)
`grammars/features.txt`	Grammar with analyses of features. Contains Cristina's trees with features (commented out). Note that the current implementation of feature unification is destructive.

2 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.

2.1 General

- Spaces, tabs, and newlines are ignored.
- Nested comments are delimited by curly braces {}.
- You can use alphanumeric characters, hyphens, and underscores for identifiers and lexical items.

2.2 Forests

2.2.1 Trees

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

```
{ syntax tree for ‘likes’ }
[s@1@2 [np!@1 (case:nom)] [vp@3 [v ‘likes’] [np!@2 (case:acc)]]]
```

2.2.2 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’]
```

2.2.3 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.

```
{ semantics tree for ‘someone’ }
[exists $x [t [et ‘person’] [e $x]] [t*]]]
```

2.2.4 Substitution Nodes

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

```
[np!]
```

2.2.5 Foot Nodes

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

```
[vp [adv ‘apparently’] [vp*]]
```

2.2.6 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']]]
```

2.2.7 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)]
```

2.2.8 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].
```

2.2.9 Forests

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

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

2.3 Derivations

2.3.1 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:[].
```

2.3.2 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].
```

2.3.3 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:[].
```

2.3.4 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:[].
```

2.3.5 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:[].}
```

3 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:[].
```

4 Code Structure

4.1 Parsing & Lexing

- `tagLexer.mll`

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

- `tagParser.mly`

  Parser containing rules for assigning semantic content to the lexed tokens.

4.2 STAG Operations

- `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`.

4.3 Miscellaneous

- `utils.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.

5 Known Issues

Current known issues:

- Feature structure unification does not work for variables across a treeset.
- As features are unified and destructively modified during every derivation, multiple derivations using the same treeset cannot properly be performed. Currently, the user can only perform one feature-based derivation at a time.
- Brackets are not properly indented during tree printing.

Feel free to raise an issue or create a pull request at https://github.com/jennhu/tag.