# User Guide

Dan Bikel

## Contents

# 1 Preliminaries

## 1.1 The right java

In theory, the parsing engine software works with any Java2 JVM as of v1.3.x and higher. In practice, I have only tested it with JVM's from IBM and Sun (I typically use Sun's). To use the provided shell scripts to train and run the parser, there is no need to set any environment variables; just make sure a `java` executable for an appropriate version is in your path. To determine this, when you execute the command

```
java -version
```

you should see a long string that indicates a version of 1.3.x, or preferably 1.4.x.

## 1.2 Settings files

The parser comes with several, crucial settings files, all located in the

```
<parser home>/settings
```

directory. A given training or parsing run needs only a single settings file, which determines, among other things, which language the parser will work in.

**N.B.**: Virtually all settings that can appear in a settings file are contained as documented constants in the `danbikel.parser.Settings` class. Please see that API documentation for this class for information about all possible settings.

## 1.3 Scripts

The distribution comes with several shell scripts, all of which are hard-coded to use `/bin/tcsh`. You may need to modify this if `tcsh` lives in a different place in your environment. As we do not yet use a true installation scheme (such as `autoconf`/`configure`), you must make any modifications by hand. This may change in the near future.

Most of the provided scripts spit out their usage if they are run with no arguments.

## 1.4 File formats

Most I/O of the parsing engine is performed by a Lisp-style S-expression reader/writer written entirely in Java. As such, newlines are generally irrelevant, being treated as just another form of whitespace. The one exception to this is a comment, which consists of a semicolon and anything after that semicolon character to the end of the line (just as in Lisp).

# 2 Quick Start

This section assumes you have already read §1 and just want to get going parsing English.

If you want to parse English sentences using the engine in its "Collins emulation mode", do the following,

- where `<dbparser home>` is the install directory created by the install script, and
- where, if you have a copy of the Penn Treebank, `<Penn Treebank home>` is the root directory of the Penn Treebank CD-ROM (or the directory to which the CD-ROM's hierarchy has been copied):

1. Train the parser on Sections 02–21 of the WSJ Penn Treebank in one of the following two ways. One way assumes you have access to the Penn Treebank, while the other does not; both ways assume that `<dbparser home>/bin` is in your path. The result of either way will be a "derived data file" called `wsj-02-21.obj.gz`.

   (a) If you have access to the Penn Treebank, execute the following commands to create the file `/tmp/wsj-02-21.obj.gz`:

      i. `cd <Penn Treebank home>/combined/wsj`

      ii. `cat 0[2-9]/*.mrg 1[0-9]/*.mrg 2[01]/*.mrg > /tmp/wsj-02-21.mrg`

      iii.
      ```
      train 800 <dbparser home>/settings/collins.properties \
      /tmp/wsj-02-21.mrg
      ```

   (b) If you do not have access to the Penn Treebank, do the following to create the file `wsj-02-21.obj.gz`:

      i. Download the observed events file `wsj-02-21.observed.gz` from Dan Bikel's homepage.

      ii. Execute the command
      ```
      train-from-observed 400 <dbparser home>/settings/collins.properties \
      wsj-02-21.observed.gz
      ```

2. Copy the file `wsj-02-21.obj.gz` created by Step 1 to a safe place. You will use this file in the next step.

3. To parse a file in the format specified in §4.1, execute the following command (use `<dbparser home>/bin/parse` if `<dbparser home>/bin` is not in your path):
   ```
   parse 400 <dbparser home>/settings/collins.properties wsj-02-21.obj.gz \
   <input file>
   ```

## 3 Training

An input file for the trainer must be in the de facto standard format of Penn Treebank `.mrg` files, which contain trees with part-of-speech tag preterminals and words as leaves. The original Penn Treebank enclosed sentences in an extra set of parentheses; the trainer disregards these parens, if they are present.

The easiest way to use the trainer is via the script `<parser home>/bin/train`, which has the following usage:

```
train <max. heap size in megabytes> <settings file> \
        <parse tree input file>
```

Note that all training trees are expected to be in one file.

Training is, for the most part, performed in-memory. This means the heap size for training needs to be rather large; a value in the range of 500-800 is generally required, but this value depends greatly on the number of sentence being fed to the trainer. (The working set is much smaller than the maximum heap size, however.)

However, in order to reduce the required memory, the user may use the `danbikel.parser.Trainer` class with the new `-it` or `--incremental-training` options, which allow the trainer to read an observations file (output from a previous training run) in 500,000-event chunks, deriving counts after each reading each chunk (chunk size is controlled via the setting `parser.trainer.maxEventChunkSize`). This prevents the trainer from reading the entire observations file into memory before deriving counts. Type

```
java danbikel.parser.Trainer -help
```

for complete usage information.

### Example

To train on a file `wsj-02-21.mrg` containing Section 02–21 of the WSJ Penn Treebank data, one would issue the command

```
train 800 <parser home>/settings/collins.properties \
        wsj-02-21.mrg
```

The train script spits out the actual `java` command that is doing the training.

The trainer outputs two files: an `.observed.gz` file, containing (a compressed stream of) human-readable, top-level event counts that were derived rather directly from the training trees, and an `.obj.gz` file, which is a series of serialized Java objects containing the actual, derived counts used by the parser. The `.observed.gz` file is called the “observations file” and the `.obj.gz` file is called “derived data file”, and is the main output file of the trainer.

# 4 Parsing

## 4.1 Uniprocessor or non–distributed computing environment

The easiest way to parse in a non-distributed fashion is to use the `<parser home>/bin/parse` script, the usage of which is:

```
parse <max. heap> <settings> <derived data file> \
      <input file>
```

The parser does not need as much memory as the trainer; for English, a value less than or equal to `500` for `<max. heap>` should suffice.

The input file should have one of two Lisp-style formats:

1. `((word1 (pos1)) (word2 (pos2)) ...  (wordN (posN)))`

2. `(word1 word2 ... wordN)`

Format 1 is typically used, where each part of speech was that produced by some (possibly automatic) tagging program.

Format 2 is used when it is desirable to have the parser do all its own part of speech tagging as part of the parsing process, but all the provided settings files assume that tagging will be performed as a pre-processing step.

A part of speech that is supplied for a word is only used when that word was never observed in training; nevertheless, *every* word must have a non-empty part of speech list; *i.e.*, the format `((word1 ()) (word2 ()) ... (wordN ()))` is *not* valid.

Here is the first sentence of Section 00 of the WSJ Penn Treebank in Format 1:

```
((Pierre (NNP)) (Vinken (NNP)) (, (,)) (61 (CD))
 (years (NNS)) (old (JJ)) (, (,)) (will (MD)) (join (VB))
 (the (DT)) (board (NN)) (as (IN)) (a (DT))
 (nonexecutive (JJ)) (director (NN)) (Nov. (NNP))
 (29 (CD)) (. (.)))
```

## 4.2 Distributed computing environment

### 4.2.1 Script usage

The easiest way to parse in a distributed-computing environment is to use the `<parser home>/bin/internal-server-run` script. The usage is:

```
internal-server-run <settings file> <derived data file> \
        <input file>+
```

Note that you can specify multiple input files. In fact, if any of the input files specified is a directory, the engine will non-recursively get the names of all files contained the directory. For example, suppose you have the following four files to parse:

- `~/data/inputFile1`
- `~/data/inputFile2`
- `~/data/new/newFile1`
- `~/data/new/newFile2`

The following three input file lists are all treated the same by the `internal-server-run` script (it's really the Java class `danbikel.parser.StartSwitchboard` that is performing this magic):

- `~/data/inputFile1 ~/data/inputFile2 ~/data/new/newFile1 ~/data/new/newFile2`
- `~/data/inputFile1 ~/data/inputFile2 ~/data/new`

- ~/data ~/data/new

While this script makes using multiple hosts easy, it may not work out-of-the-box, requiring a few caveats:

1. The script uses the environment variable JAVA_HOME to locate the J2SDK you wish to use, so you must make sure this variable is set in your environment. The two J2SDK executables that are used are $JAVA_HOME/bin/java (by the internal-server-run script) and $JAVA_HOME/bin/rmiregistry (by the no-cp-rmiregistry script, which is called by the start-rmiregistry script, which is in turn called by the internal-server-run script).
2. Most batch queues copy a batch script before executing it, which breaks any reliance on the $0 variable. Accordingly, you may have to modify the script to hard-code paths for the scriptDir variable (near the beginning of the script). Alternatively, you can use the simple wrapper script (called, conveniently enough, wrapper) to avoid this problem.
3. The script uses ssh for logging into nodes, and assumes that it will not need to enter a password for doing so. The remote shell mechanism of the internal-server-run script, as well as that of a dependent script called start-rmiregistry, may be changed by altering the definition of the RSH variable.
4. The script assumes the nodes will be provided via whitespace-separated list in a NODES environment variable, and preprends "node" to every element in this list (appropriate for Clubmask/Beowulf environment at Penn). Please comment out the shell script code that does the prepending if this behavior is not appropriate for your environment.
5. The script assumes that the unique identifier for the batch job will be contained in a JOBID environment variable. This job identifier is used only to guarantee that different jobs have distinctly-named experiment directories (see §4.2.2 for more information on experiment directories).
6. The script relies on other scripts in its bin directory, and expects the file dbparser.jar to be located in its parent directory. The script finds these relatively-located resources by using the $0 variable (see point No. 2, above).
7. When the reap variable is defined, there is a section of code at the end of the script that logs onto all hosts and uses the killall command to kill all rmiregistry and java processes; this behavior may literally be overkill for your environment, if, for example, you have other, non-parsing java processes that you do not wish to kill. The solution is to comment-out the line in the script reading set reap, or to modify the reaping code to be more discriminating in the processes that it kills.

In the future, we may customize the internal-server-run script via an installation procedure, instead of forcing you, the user, to perform the customizations directly.

### 4.2.2 Experiment directory

The script creates an experiment directory whose name is ~/experiments/<date>/<time>

The experiment directory will contain the following items:

1. Log files for all the parsing clients, where each log file has the name `<host>-<uid>.log`, where `<host>` is the host name on which the client is running and where `<uid>` is a unique integer, so that two clients running on the same host will have differently-named log files.

2. A log file containing the incremental work of the distributed-computing run; this file has the same name as the input file plus a `.log` extension.

3. A file called switchboard.messages, which can be monitored to track the progress of the distributed-computing run, via the command `tail -f switchboard.messages`

4. Finally, the `internal-server-run` script copies a specialized version of *itself* to the experiment directory, in order to facilitate re-running the experiment, or continuing an experiment that had to be killed before all input sentences were parsed. This latter feature—being able to recover from a previous, incomplete run—can be very useful. Crucially, the ability to re-start an experiment from where it left off makes use of the log file that contains incremental work (list item No. 2, above).

## 5 Advanced usage

It is, of course, possible to call the `java` process directly for training and parsing. The normal mechanism to specify a settings file is to provide a command-line definition of the `parser.settingsFile` system property, as follows:

```
java -Dparser.settingsFile=collins.properties ...
```

To avoid specifying the settings file on the command line, you can provide a default settings file: the parsing engine will always check to see if the file `~/.db-parser/settings` exists (where ~ is your home directory), and if so, use it if there is no definition of the `parser.settingsFile` system property. For example, if by default you will be parsing English with the parser in its Collins-emulation mode, you can execute the following commands:

```
mkdir ~/.db-parser
cp <parser home>/settings/collins.properties ~/.db-parser/settings
```

### 5.1 Training

The Java class used for training is `danbikel.parser.Trainer`. A typical usage is as follows (assumes you have `dbparser.jar` in your class path):

```
java -Xms800m -Xmx800m -Dparser.settingsFile=<settings> \
        danbikel.parser.Trainer -i <training file> \
        -o <observed file> -od <derived data file>
```

You can see its full usage by executing

```
java danbikel.parser.Trainer -help
```

If either <observed file> or <derived data file> ends with the extension .gz it is automatically compressed before being written to disk.

### A note on the <observed file>

The <observed file> is a human-readable file consisting of the top-level events and counts from which all other events and counts may be derived. As such, training may be performed in two steps, outputting an <observed file> and then reading that file in to produce a <derived data file>:

```
java -Xms400m -Xmx400m -Dparser.settingsFile=<settings> \
        danbikel.parser.Trainer -i <training file> \
        -o <observed file>

java -Xms400m -Xmx400m -Dparser.settingsFile=<settings> \
        danbikel.parser.Trainer -it -l <observed file> \
        -od <derived data file>
```

Note the use of the -it option in the second of the two commands. This indicates to derive counts incrementally from an <observed file>, and reduces the RAM footprint considerably (all the heap sizes in this section are applicable when training on the standard Penn Treebank WSJ training set, Sections 02–21).

For the curious, incremental training is performed by iteratively reading top-level events and counts from the <observed file> one “chunk” at a time, additively deriving events and counts after each such “chunk” read. The number of top-level events read in at each increment is determined via the

```
parser.trainer.maxEventChunkSize
```

setting, the default value for which is 500000.

## 5.2 Parsing

The Java class used for parsing is danbikel.parser.Parser. A typical usage is as follows (assumes you have dbparser.jar in your class path):

```
java -Xms400m -Xmx400m -Dparser.settingsFile=<settings> \
        danbikel.parser.Parser -is <derived data file> \
        -sa <sentence input file>
```

You can see its full usage by executing

```
java danbikel.parser.Parser -help
```

If <derived data file> ends with the extenson .gz it is automatically decompressed as it is read from disk.

### 5.3 Switchboard

The Java class used for starting the switchboard, which is the central component in a distributed-computing run, is `danbikel.parser.StartSwitchboard`. You can see its full usage by executing

```
java danbikel.parser.StartSwitchboard -help
```

Developer-level documentation is available in the `<parser home>/doc` directory. Currently, not all methods and classes are documented.

### 5.4 *k*-best Parsing

As of version 0.9.4, there is a hack to do *k*-best parsing, where basically dynamic programming is eliminated by having no two chart items be equivalent (thanks to Mike Collins for suggesting this hack). There are three settings one should adjust in order to do *k*-best parsing:

- `parser.chart.itemClass=danbikel.parser.CKYItem$KBestHack`
- `parser.decoder.kBest=`*k* (where *k* is the desired maximum number of best parses to output)
- `parser.decoder.pruneFactor=`<*something smaller than 4*> (you want to use a small beam to avoid pursuing too many theories—remember, we have simply turned off dynamic programming)

The output of the parser is slightly different when $k > 1$: in this case, instead of an S-expression representing a tree, it is a list of S-expressions representing trees (a list of lists), in decreasing order of likelihood.

### 5.5 Knesser-Ney Smoothing

The type of smoothing used is determined by the type of `Model` object that a `ProbabilityStructure` instance wraps itself in, as determined by the method `ProbabilityStructure.newModel`. The default behavior is to create an instance of `Model`, which uses a variant of Witten-Bell smoothing.

However, this default behavior can be changed in a concrete subclass of `ProbabilityStructure` by overriding the `newModel` method, or by simply using a run-time setting,

`parser.probabilityStructure.defaultModelClass`.

The default value for this setting is `danbikel.parser.Model`, but Knesser-Ney smoothing can be used by default by changing this setting to be

`danbikel.parser.InterpolatedKnesserNeyModel`.

For more information on this setting, see the API documentation for the `danbikel.parser.Settings` class.