Lab for week 5: Recursive Descent Parser

Author: Henry Thompson
Author: Bharat Ram Ambati
Author: Sharon Goldwater
Author: Ida Szubert

Date: 2015-10-14, updated 2016-10-14, 2017-10-09, 2018-10-11

Use the html version[2] of this lab if you want to easily access the links or copy and paste commands.

Or use the pdf version[3] if you want nicer formatting or a printable sheet.

Goals and motivation of this lab

The *syntactic parsing* of a sentence consists of finding the correct syntactic structure of that sentence in a given formalism. Formalisms are called grammars, and contain the structural constraints of the language. Parsing is one of the major tasks which helps in processing natural language.

In this lab you will:

- Explore *recursive descent parsing* using some simple Phrase Structure Grammars (CFGs). We aim to give you a sense of how much computation is *potentially* involved in parsing sentences, and thus why cleverer parsing algorithms are needed.
- Practice writing grammars, to see how the choices you make can interact with the parsing algorithm, sometimes in undesirable ways. This exercise should also give you a better sense of some weaknesses of simple CFGs.
- Explore the grammar rules and parsed sentences in a more realistic broad-coverage grammar, extracted from a subset of the Penn Treebank. This will give you an idea of the size and complexity of treebank grammars.

NLTK

In this lab, we use the NLTK[4] library. NLTK (Natural Language Toolkit) is a popular library for language processing tasks which is developed in Python. It has several useful packages for NLP tasks such as tokenisation, tagging, parsing etc. In this lab, we use very basic functions for loading the data, accessing it as sentences and running a built-in parser. We also use a graphical demo app that runs inside NLTK and shows you a recursive descent parser in action.

[1] http://creativecommons.org/licenses/by-nc/4.0/.
[2] lab5.html
[3] lab5.pdf
[4] http://www.nltk.org/
[5] http://www.inf.ed.ac.uk/teaching/courses/anlp/labs/lab5.py
[6] http://www.inf.ed.ac.uk/teaching/courses/anlp/labs/lab5_fix.py
[7] http://www.inf.ed.ac.uk/teaching/courses/anlp/labs/lab4.html

Preliminaries

As usual, create a directory for this lab inside your `labs` directory:

```
cd ~/anlp/labs
mkdir lab5
cd lab5
```

Download the files lab5.py[5] and lab5_fix.py[6] into your `lab5` directory: Right-click on each link below and select *Save link as...*, then navigate to your lab5 directory to save.

Start up Spyder and open `lab5.py` in the editor.

Running the code

Run the code by clicking the green triangle or typing `%run lab5.py` in the interpreter. You should see a parse of the sentence *he ate salad* and a list of the production rules used in this parse. The parse is based on the rules in `grammar1`, defined in our code.

The output you see was generated by these two lines:

```
parse_tree = recursive_descent_parser(grammar1, sentence1, trace=0)
print_parse_info(parse_tree)
```

The next section of the lab uses a demo app to visualize the actions of the parser. You can also see a printed record of the actions (Expand, Match, or + to indicate a completed parse) by setting the `trace` argument to 2:

```
parse_tree = recursive_descent_parser(grammar1, sentence1, trace=2)
```

If you want, call `parse_tree.draw()` to see a prettier version of the parse. In Spyder, you'll also get a pretty version by just typing `parse_tree` into the interpreter. This won't work in all Python interpreters, which is why the `draw()` method is useful.

Recursive Descent Parser (RDP) Demo App

To better visualize the parse tree and the sequence of parser actions, we'll use NLTK's `RecursiveDescentApp`.

In this section we'll use the toy grammar defined as `grammar1`:

```
# Grammatical productions.
S -> NP VP
NP -> Pro | Det N | N
Det -> Art
VP -> V | V NP | V NP PP
PP -> Prep NP
# Lexical productions.
Pro -> "i" | "we" | "you" | "he" | "she" | "him" | "her"
Art -> "a" | "an" | "the"
Prep -> "with" | "in"
N -> "salad" | "fork" | "mushrooms"
V -> "eat" | "eats" | "ate" | "see" | "saw" | "prefer" | "sneezed"
Vi -> "sneezed" | "ran"
Vt -> "eat" | "eats" | "ate" | "see" | "saw" | "prefer"
Vp -> "eat" | "eats" | "ate" | "see" | "saw" | "prefer" | "gave"
```

Run the app using the command:

```
app(grammar1,sentence1)
```

You should see an app window with five coloured buttons at the bottom to control the parser. Try 'Step' a few times. Note how the display corresponds to the description of the recursive descent algorithm in the lecture, in particular the way the current subgoal and possible expansions are highlighted.

(If the app window is tiny or has tiny fonts, try closing it and repeating the above command).

Now 'Autostep'.

Watch progress in the tree window, and individual actions in the 'Last Operation' display. Hit 'Autostep' again to stop the parser. Try 'Step' again a few times, and see how the app indicates which options at a choice point have already been tried, and which remain.

Try some other sentences (using `Edit -> Edit Text` in the menu bar) and watch how the parser runs. You can speed things up by selecting `Animate -> Fast Animation`. The parser stops when it gets the first valid parse of a sentence. You can start it again (with 'Step' or 'Autostep') to see if there are more valid parses.

You can reset the parser to the starting point using `File -> Reset parser`.

Try parsing:

```
i ate the salad with a fork
```

Slow the animation back down, and start with Autostep, but stop that when it gets to work on the Det N version of NP as applied to "the salad". Step through the next bit one at a time, to see just how much backtracking (and wasted effort) is involved in getting back to the *correct* expansion of VP, followed by a complete (and lengthy) recapitulation of the parsing of "the salad" as an NP.

Adding productions to the grammar

Run the parser on the sentence `He ate salad` (note the capital 'H'). Can you parse this sentence using the toy grammar provided? What production would you have to add to the grammar to handle this sentence? Pro->'He'

Use `Edit -> Edit Grammar` to bring up an edit window for the grammar, and make the necessary change. Note that terminals must be entered using quotes.

Look at the parse trees for the following sentences:

```
he ate salad with a fork ✓
he ate salad with mushrooms
```

should be but attached with ate

Is there a problem with either of the parse trees ? (*Hint*: Observe the attachments of prepositional phrases ('with a fork', 'with mushrooms') in both the parse trees).

Edit the grammar again and add a production `NP -> NP PP`, *after* the other `NP` rules. Re-run the parser on one of the above sentences and take note of the parse tree. Then, change the order of the rules `NP -> N` and `NP -> NP PP`, so that the `NP PP` expansion is used by the parser first. Run the parser on the one of the sentences again.

What is the problem with this new ordering and how does the parser's behaviour differ from the other ordering? left recursion

How do you think this behaviour depends on the particular way this recursive descent parser chooses which rule to expand when there are multiple options?

Remove the offending rule before going on.

Ungrammatical sentences

Run the parser on following sentences:

```
he sneezed
he sneezed the fork
```

不及物动词

"sneeze" is an intransitive verb which doesn't allow an object. Though the second sentence is ungrammatical, it is parsed by our grammar.

Modify the grammar to handle such cases. What rules did you add/change? (*Hint*: Required *lexical productions* are already provided in the grammar)

Vt|Vi|Vp 区分后面是否可以接其他成分

Number agreement (optional)

If you're more interested in looking at real treebank rules, feel free to skip this and go to the next section.

Our toy grammar ignores the concept of number agreement. Change the grammar to handle number agreement by creating subcategories of the appropriate categories and use your new grammar to parse/rule out the following sentences, or others of your choosing:

```
he eats salad
he eat salad
he ate a mushrooms
```

绵羊，胆小鬼

Try an ambiguous sentence, such as `the sheep ran`. You can force the parser to look for additional parses after it finds a complete parse by hitting Autostep again. Now try `the sheep sneeze` and compare.

Exploring a treebank grammar

The previous parts of the lab dealt with a toy grammar. The remaining parts give you some idea of what a broad-coverage treebank and treebank-derived grammar looks like. We consider a small part of the Penn Phrase Structure Treebank. This data consists of around 3900 sentences, where each sentence is annotated with its phrase structure tree. Our code uses NLTK libraries to load this data and extract parsed sentences.

Extracting and viewing parsed sentences

We obtained the treebank with the import statement `from nltk.corpus import treebank`. Uncomment the following two lines in the lab code:

```
psents = treebank.parsed_sents()
print_parse_info(psents[0])
```

Then save and re-run the file.

It should print the first parsed sentence (at index 0) of the Penn Treebank, and the grammatical and lexical productions used in it.

The first line above gets the list of parsed sentences from the treebank, and so `psents[0]` gives the first parsed sentence. When you `print` it (this happens inside `print_parse_info`), it shows up as a sort of left-to-right sideways tree, with parentheses around each node label and its children. Words are at the leaves, with each word dominated by a single pre-terminal label (POS category). These POS-word pairs are then grouped together into constituents labelled with nonterminal labels (phrasal categories).

You can also look at the parses and productions for other sentences by changing which sentence is passed in to `print_parse_info`. Verify this by looking at the parse and productions for the second sentence in the corpus.

What type of object is the parsed sentence object? (*Hint*: use the `type` function) nltk.tree.Tree

Check the methods available for this object using the command `help(psents[0])`. You can ignore the ones beginning with '__', they are Python-internal ones.

Try using the `draw()` method on some of the sentences. Remember, `draw()` is a *method*, not a *function*, so the syntax is, e.g., `psents[0].draw()`.

Extract the list of words and the list of (word, pos-tag) tuples from `psents[0]` using some of the other available methods.

psents[0].leaves()
psents[0].pos()

Distribution of Productions

As we've seen before, a Python dictionary can be used to store a frequency distribution, where each key is an element of the population being analysed and the value for each key is its count.

Construct frequency distributions of the productions in the corpus by completing the code in the `production_distribution` function, which returns two dictionaries, one for lexical productions and the other for grammatical productions.

Start by using `help(defaultdict)` and `help(int)` to understand the first two lines of the `production_distribution` function.

When calling `production_distribution`, remember that Python supports destructuring assignment, so you can do e.g.:

```
(lexicalFD,phrasalFD) = production_distribution(psents)
```

Do you expect there to be more lexical or non-lexical (grammatical) productions in the grammar? Roughly how many non-lexical productions do you think there might be? Now check how many productions of each type there actually are. Do the numbers surprise you?

What are the 10 most frequent and least frequent lexical and grammatical productions? (*Give up?* See the hint below if you get stuck.)

Going further

1. What is the percentage of the 10 most frequent productions (lexical *or* grammatical) from the corpus taken together, with respect to the total number of productions?

2. Does the treebank grammar enforce subcategorization constraints such as number agreement or verb argument structure (e.g., transitive/intransitive/ditransitive)? If so, find some examples of rules that enforce these constraints. If not, find some examples of parses where you could replace one of the words in the parse with another word that is grammatical under the treebank grammar, but not in real English.

3. By reusing code from 2014's lab 4[7], plot histograms of the top 50-100 grammatical and lexical productions in the treebank grammar. Do these distributions look familiar?

Promised hint

Try this, where `xxxFD` is one of the two dictionaries returned by `production_distribution`:

```
lexCounts = sorted(xxxFD.items(), key=lambda x: x[1])
```