Lab 5: Recursive Descent Parser

Author: Henry Thompson
Author: Bharat Ram Ambati
Author: Sharon Goldwater
Date: 2015-10-14

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

Parsing a sentence consists of finding the correct syntactic structure for that sentence using a given grammar, where the grammar contains the structural constraints of the language. Parsing is one of the major tasks which helps in processing and analysing natural language.

In this lab we 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. We also aim to give you some practice writing grammars, to see how the choices you make can interact with the parsing algorithm, sometimes in undesirable ways.

NLTK

In this lab, we use 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 like tokenisation, tagging, parsing etc. In this lab, we use very basic functions like loading the data, reading sentences. You will be asked to write few lines of code in a function to answer some questions in this lab. Some of these functions are already implemented in NLTK. You can explore those options for questions in 'Going Further' section as Home work.

The first part of this lab is based on a graphical demo app that runs inside NLTK and shows you a recursive descent parser in action. The second part looks at the grammar rules and parsed sentences in a more realistic broad-coverage grammar (from a subset of the Penn Treebank).

Preliminaries

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

[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

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

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

Open the file with your preferred editor and start up ipython:

```
gedit lab5.py &
ipython
```

Check that NLTK is working by downloading the corpus for the second part of this lab:

```
import nltk
nltk.download('treebank')
```

Recursive Descent Parser (RDP)

We'll start with the following toy grammar, implemented as `grammar1` in the lab code:

```
# 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"
```

Our code reads in this grammar and turns it into an `nltk.grammar.ContextFreeGrammar` object using `parse_grammar` (defined in `lab5_fix.py`, which you need not examine).

Launch the NLTK recursive descent parser app from ipython like this:

```
%run lab5.py
app(grammar1,sentence1)
```

This shows an app. The five coloured buttons at the bottom of the resulting app window control the parser. Try 'Step' a few time. Note how the display corresponds to the description of the recursive descent algorithm in Monday's lecture, in particular the way the current subgoal and possible expansions are highlighted.

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 to see if there are more valid parses, or you can reset it under the `File` menu.

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?

Look at the parse trees for the following sentences:

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

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).

Using the demo app, add a production `NP -> NP PP`, *after* the other `NP` rules. (Click on `Edit->Edit Grammar` to do this). 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?

How do you think this behaviour depends on the particular way this RD 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
```

"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)

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
```

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

Make sure you've downloaded the corpus (see Preliminaries) and 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), and the grammatical and lexical productions used in the first sentence.

The first line of these two extracts the list of parsed sentences from the treebank (which is imported at the top of the file), and `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)

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.

Distribution of Productions

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

(Don't forget to do:

```
    %run lab5.py
```

after you've done some edits, in order to get your new function definitions.)

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 next page for code which implements one to do this 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. Run the parser on the sentence "John ate salad". Add required grammatical and lexical productions to handle this sentence.

3. Does the rule-ordering trick work if you add a rule of the form:

```
Det -> NP 's'
```

to `grammar1` ? Try it on the sentence `the salad s mushrooms sneezed` (sorry about this app's inability to handle the apostrophe). Try forcing it to look for another parse -- what happens ?

4. The grammar which handles number agreement still parses ungrammatical sentences like "i sneezes". Adding more agreement rules is a messy process. What is the better way to deal with this problem ?

5. By reusing code from last year'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 `xxxdict` is one of the dictionaries returned by `production_distribution`:

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