Natural Language Processing CS224N/Ling237
Programming Project 1

A simple tabular parser

Handout #5

Due Monday, 21 Apr 2001, 5 p.m.

Read this assignment soon, and especially if you are less familiar with programming, start working on it, discover stumbling blocks, and ask questions early. If anything is unclear or you need some kind of assistance, feel free to send a question to cs224n-spr0203-staff@lists.stanford.edu. However, before you do that, you're encouraged to look for among the FAQs on the class website http://www.stanford.edu/class/cs224n/

The aim of this project is to get experience with writing some simple parsers, and then to be able to develop grammars and lexicons to allow parsing limited, but non-trivial, amounts of English text. In the course of doing that, you should get to see how ambiguous English is in terms of syntactic categories.

You may write the program in any language you wish. You'll have an easier time if you write it in a language like Java or Python which gives you a bunch of standard datatypes (lists, hashtables, ...) for free. We also provide a convenient code framework in Java, using which you will need to implement only the core parsing algorithm (more on this later). Feel free to change and use that code in any way you like.

If you have other hashtable, etc. modules from other classes that you are familiar with, feel free to use those instead. However, we will not be able to provide support in that case. While you are welcome, in this way, to reuse code for standard data structures, you naturally shouldn't be scouring the web to find a complete parser to reuse. That's not the aim of the project. In any case, *if you are reusing code of others from places other than standard system libraries, you should be sure to acknowledge it in your write-up.*

What to do

Part 1

Write a simple top-down parser, which reads a grammar and a lexicon, and will then parse sentences. You should implement this via control structures for a general search strategy that explores rules top-down and left-to-right. You can assume that the grammar does not contain any left recursive rules.

The grammar and lexicon are specified in separate files in the following format:

```
S --> NP VP
NP --> DT NN
NP --> NNP

DT --> the
NNP --> Chris
NNP --> Adil
```

The `S` symbol is always assumed to be the distinguished start symbol of the grammar. You should assume all other symbol names to be arbitrary.

Similarly, the input sentences will be provided one per line in a file:

```
Chris saw Adil
Adil saw Chris
```

Your program should accept 4 arguments - the first one is always `T` for this part (to differentiate from part c) and the other 3 are the respective filenames to use. For example, with our code, the parser should be run as:

```
java Parser T grammarfile lexiconfile sentencefile
```

Your parser should take in the three file names as arguments, and write out the parses for each sentence in the sentences file in some reasonable (e.g., bracketed [p.98 of the book] or indented list) format. At the end of each sentence, it should print out a line like:

```
This sentence had 5 parses
```

Important: It *must* print out exactly that. We're going to check the correctness of your parser by searching the output files for regular expressions of the form "This sentence had [0-9]+ parses". Make sure your output doesn't cause us a headache. Note that your parser should work for any grammar and lexicon without left-recursive rules: we should be able to try it out with a little grammar of, say, Inuit, and expect it to work.

If you use the code framework provided in our Java files, you will only need to implement the core parsing function in `TDParser.java`. The grammar, lexicon and the sentences have been read in, and are passed to your parsing function. Have a look at the comments in that file, and feel free to ask if some part of the code is unclear.

If you are using Java but not our code, please put your main routine in a class labeled `Parser`, as in our framework. For other languages, please provide an executable or shell-script named `Parser` that will run with arguments as shown above.

Part 2: Grammar development

Write a grammar and lexicon (only as much as you need!) for your parser. (Make sure it has no left-recursive rules, so that the parser doesn't get into an infinite loop!) Ensure that your grammar and lexicon can parse the correct sentences, and returns an interesting selection of the ambiguities present in them:

> cats with claws scratch people
> cats that bite people scratch people in the face
> Chris saw the man in the corner of the room through a telescope
> the post office will hold out discounts as incentives to the franchisees

Try to make sure that your grammar also doesn't parse ungrammatical sentences:

> *Chris saw

Keep a copy of your grammar, lexicon and sentence files in your code directory under the names `mygrammar`, `mylexicon` and `mysentences`. Submit these along with the rest of the code (as described at the end). Try to choose your sentences to demonstrate different aspects of the grammar, and also include ungrammatical sentences that are rejected by your grammar. You can also write a brief note on your observations with these in your write-up.

Part 3

If you try your parser on larger sentences, you should see that the runtime of the above parser gets quite bad – it's exponential in sentence length (assuming a suitably ambiguous grammar). One way to fix this problem would be to throw this parser away, and to start on writing a CKY parser or an active chart parser, as in class. But an alternative (but related) idea is to get the same speed-up by memoization – we record the results of computations that we've done before, and reuse them.

Here's the idea. At each stage in the top-down search, we'll be exploring a goal such as "Can I build a VP starting at position 5 in the sentence?", and the answer(s) might be "Yes, one can build a VP spanning words 5 through 8 in two ways, like this, and a VP spanning words 5 through 10 like this". Every time that we've found all the ways of building a nonterminal starting at a certain position, we record the answers. That is we do *memoization*. And then, when we're about to start into a parsing problem, such as "Can I build an NP starting at position 6?", rather than immediately starting to solve that problem with exhaustive search using the grammar, we instead first look at our results table to see if we have already memoized that computation. If so, we just re-use that work. We can do this efficiently (in terms of time – perhaps not in terms of space) if we keep possible subgoals in a hashtable, and can just retrieve the answers when needed.

Write a version of a top-down parser that memoizes results in this way. Make sure that your parser gives the same answers as the simple top-down parser without memoization.

Your program should accept 4 arguments - the first one is always `M` for this part (to differentiate from part a) and the other 3 are the respective filenames to use. For example, with our code, the parser can be run as:

```
java Parser M grammarfile lexiconfile sentencefile
```

If you use the code framework provided in our Java files, you will only need to implement the core parsing function in `MemoParser.java`. The structure is like in part a, and much of the code may be similar. You should also feel free to use inheritance if it helps to cleanly reuse the top down parser from part a, and provide memoization within it.

Questions: [Answer this in your write-up!] Does this technique solve all the problems of efficient parsing? What is its time complexity? Does it still have limitations compared to an active chart parser?

Compare the run time of the memoized and the unmemoized parser on sentences of different lengths with a sample grammar (eg, with attachment ambiguities). The run time is already calculated in our Java implementation, and is output with the number of parses. Provide a graph of run time of the two parsers vs sentence length or number of different parses.

Extra credit

Extra credit of up to 10% will be given for significant improvements over the basic requirements, such as effectively using lexical lookup rather than parsing lexical items top-down, superior grammars, etc. Another alternative is to make the parser work for empty rules: `NP --> e`, by means of recognizing `e` as a special empty rule token. (This would make it hard to recognize some languages where `e` is a word, but never mind.)

Files

Some materials that you may use or need for this assignment are in the following directory in AFS/on the Sweet Hall machines: /afs/ir/class/cs224n/pp1

We'll put there a sample grammar, lexicon and test sentences. To test your code, make sure that:

```
java Parser T (or M) gfile lfile sfile
```

results in the 2 correct parses for the sentence in sfile.

The Java code is in the /afs/ir/class/cs224n/pp1/java directory. You will need to fill in the `findAllParses(...)` method in `TDParser.java` (part a) and `MemoParser.java` (part c). Feel free to change the code in any way you like. The grammar and lexicon are represented in the same way for simplicity (as `Grammar` objects), and some hints on using them are included in the comments in `TDParser.java`.

Evaluation criteria

Submission of the program code will be via a script (available nearer the submission date). To submit your program, put all the needed files in one directory on a Sweet Hall machine (or, possibly, another machine with AFS on it and with you being correctly authenticated) and type:

```
/afs/ir/class/cs224n/bin/submit-pp1
```

If you need to resubmit it type

```
/afs/ir/class/cs224n/bin/submit-pp1 -replace
```

You should make sure that you include the source code for your programs, a Makefile that will build them, and data files for your grammar and lexicon. We will run programs on the Sweet Hall systems, so they should run without problems there. If there is something we should know to correctly run your parser, include that in a `README` file along with your code.

You may complete the assignment using whatever programming language(s) you wish, but we expect clear readable code, and clear textual description of the algorithms employed in your write-up. The report need not be long but should accurately describe the architecture of your program, the testing you did, any extensions you implemented, and any relevant discussion of the structure of your grammar, or problems you had implementing the grammar. It should also contain answers to the explicit questions in part c, and graphs if any. **Your write-up must be submitted as a hard copy.**

This assignment may be done **individually or in groups of two**.

What we're looking for in the grading is:

- A parser that works correctly when we run it over test gramars. [This is the most important thing!]
- A linguistically sensible grammar for the range of sentences indicated.
- A clear discussion of the algorithms/method used.
- Clean, intelligible code.
- A discussion of the testing you did.
- Your answers to the questions from part c, and any graphs you might have.
- Your results on parsing speed [it isn't the goal to optimize the code as much as possible, but you should have sensible comparative results]
- A discussion of alternatives or extensions you implemented.