600.465 — Intro to NLP
Assignment 4: Parsing

Prof. J. Eisner — Spring 2016
Due date: Wednesday 23 March, 2 pm (subject to change)

Now's your chance to try out some parsing algorithms! In this assignment, you will build a working Earley parser—not just a recognizer, but an actual probabilistic parser.

Collaboration: ***You may work in pairs on this assignment,*** as it is programming-intensive and requires some real problem-solving. That is, if you choose, you may collaborate with one partner from the class, handing in a single homework with both your names on it. However:

1. You should do all the work *together*, for example by pair programming. Don't divide it up into "my part" and "your part."
2. Your README file should describe at the top what each of you contributed, so that we know you shared the work fairly.

In any case, observe academic integrity and never claim any work by third parties as your own.

All the files you need can be found in http://cs.jhu.edu/~jason/465/hw-parse, or in /usr/local/data/cs465/hw-parse on the ugrad machines. You can download the files individually as you need them, or download a zip archive that contains all of them. Read Table 1 for a guide to the files.

You should actually look inside each file as you prepare to use it!

- For the scripts, you don't have to understand the code, but do read the introductory comments at the beginning of each script.

- For the sample grammars, look at the .gra files, because they include comments. However, you can ignore the syntactic and semantic attributes that appear between square brackets []. Those are for the *next* assignment. For now, your parser will consider only the .gr files, which are produced automatically from the .gra files (see Table 1).

Programming language: You may write your parser in any programming language you choose (except Dyna), so long as the graders can run it on the ugrad machines (ugrad1–ugrad24). If you need a new language installed there, let us know ASAP and we can try to arrange that with the sysadmins. I happened to use LISP, where it was 130–150 lines or about 3 pages of code (plus a 1-line parse script to invoke LISP from the command line).

As always, it will take far more code if your language doesn't have good support for debugging, string processing, file I/O, lists, arrays, hash tables, etc. Choose a language in which you can get the job done quickly and well.

If you use a slow language, you may regret it. Leave plenty of time to run the program. My compiled LISP program—*with* the PREDICT and left-corner speedups in problem 3—took about 70

`*.gra`	full grammar with rule *frequencies*, attributes, comments
`*.gr`	simple grammar with rule *probabilities*, no attributes, no comments
`delattrs`	script to convert `.gra` ⟶ `.gr`
`*.sen`	collection of sample sentences (one per line)
`*.par`	collection of sample parses
`checkvocab`	script to detect words in `.sen` that are missing from the grammar `.gr`
`parse`	program **that you will write** to convert `.sen` $\xrightarrow{\texttt{.gr}}$ `.par`
`prettyprint`	script to reformat `.par` more readably by adding whitespace

Table 1: Files available to you for this project.

seconds[1] to get through the nine sentences in `wallstreet.sen`, spending most of its time on the two long sentences. Interpreted languages like Perl will run many times *slower*. So if you want to use Perl, think twice and start extra early—your program may take hours, especially before you add the speedups.

> *Python hint:* Try running PyPy instead of Python. It's an alternative implementation with a just-in-time compiler that makes it run much faster than interpreted Python.
>
> *Java hint:* Java usually runs a bit faster than LISP. By default, a Java program can only use 64 megabytes of memory by default. To let your program claim more memory, for example 128 megabytes, run it as `java -Xmx128m parse ....` But don't let the program take more memory than the machine has free, or it will spill over onto disk and be *very* slow.
>
> *C++ hint:* For many programs, C++ runs several times faster than LISP or Java. But don't try this assignment in C++ without taking advantage of the data structures in the Standard Template Library: `http://www.sgi.com/tech/stl/`.
>
> `http://shootout.alioth.debian.org/` compares the speed of compute-intensive benchmarks coded in different languages. The k-nucleotide benchmark (`http://snurl.com/si7oj`) may be roughly comparable to parsing.

On getting programming help: Same policy as on assignment 3. (Roughly, feel free to ask anyone for help on how to use the attributes of the programming language and its libraries. However, for issues directly related to NLP or this assignment, you should only ask the course staff or your partner for help.)

How to hand in your work: As usual. As for the previous assignments, put everything in a single submission directory. Besides the comments you embed in your source files, put all answers, notes, etc. in a `README` file. Depending on the programming language you choose, your submission directory should also include your commented source files, which you may name and organize as you wish. If you use a compiled language, provide either a Makefile or a `HOW-TO` file in which you give precise instructions for building the executables from source. The graders must then be able to run your parser by typing `./parse arith.gr arith.sen` and `./parse2 arith.gr arith.sen` in your submission directory on the `ugrad` machines.

[1] That is, 70 seconds using the CLISP compiler. It was a few times slower with the CMUCL compiler. These times are on my ThinkPad T530i laptop.

1. To familiarize yourself with parsing, try out a state-of-the art parser. You don't have to spend a long time on this question, but experiment by having it parse at least a few sentences. **In your README, discuss what you learned, for example:**

   (a) Was there anything interesting or surprising about the style of trees produced by the parser?

   (b) What were some things that the parser got wrong? What were some hard things that it managed to get right?

   (c) Can you stump it with a grammatical sentence that you make up? With a grammatical sentence that you find in an online document?

   Hints: The following parsers have online demos, so they are easy to try:

   - Berkeley Parser: http://tomato.banatao.berkeley.edu:8080/parser/parser.html
   - Stanford Parser: http://nlp.stanford.edu:8080/parser/ (for English, Chinese, and Arabic)

   The Penn Treebank is a collection of manually built parses covering about a million words (40,000 sentences) of English *Wall Street Journal* text. The English parsers above were trained on a subset of the Penn Treebank. They were formally evaluated by comparing their output to the manual parses on a different subset of the Penn Treebank sentences (i.e., the test set).

   But when you give the parser a sentence from *today's* news, how should you tell whether its answer is correct—i.e., whether it matches what the humans would have done manually?

   In principle, you could find out the "right answer" by carefully reading the guidelines of the Penn Treebank project, at http://cs.jhu.edu/~jason/465/hw-parse/treebank-manual.pdf. This 300-page manual lists many phenomena in English and describes how the Penn Treebank linguists agreed to handle them in the parse trees.

   But reading the whole manual would be way too much work for this question. You should just try to see where the parse looks "reasonable" and where it doesn't. You can find a simple list of nonterminals at http://cs.jhu.edu/~jason/465/hw-parse/treebank-notation.pdf. You can experiment to figure out the other conventions. For example, if you are not sure whether the parser correctly interpreted a certain phrase as a relative clause, then try parsing a very simple sentence that contains a relative clause. This shows you what structures are "supposed" to be used for relative clauses (since the parser will probably get the simple sentence right). You can also discuss with other students on Piazza.

   Extra credit: Not all parsers produce parse trees of the sort we've been studying in class! They may produce other kinds of sentence diagrams. For your interest, here are some other good parsers with online demos. Play with them as well and report on what you find.

   - TurboParser: http://demo.ark.cs.cmu.edu/parse (dependency grammar)
   - Link Grammar Parser: http://www.link.cs.cmu.edu/link/submit-sentence-4.html (link grammar)
   - C&C parser: http://svn.ask.it.usyd.edu.au/trac/candc/wiki/Demo#Tryityourself (combinatory categorial grammar)

2. Write an Earley parser that can be run as

```
parse foo.gr foo.sen
```

where

- each line of `foo.sen` is either blank (and should be skipped) or contains an input sentence whose words are separated by whitespace
- `foo.gr` is a grammar file in homework 1's format, except that
  - you can assume that the file format is simple and rigid; predictable whitespace and no comments. (See the sample `.gr` files for examples.) The assumption is safe because the `.gr` file will be produced automatically by `delattrs`.
  - you can assume that every rule has at least one element on the right-hand side. So $X \to Y$ is a possible rule, but $X \to$ or $X \to \epsilon$ is not. This restriction will make your parsing job easier.

    Remark: Remember, it is legal to have a rule like `NP` → `a majority of N`. The right-hand side of this rule has three terminal symbols and a nonterminal, giving rise to 5 dotted rules such as `NP` → `a majority . of N`. Another example is `EXPR` → `EXPR + TERM` in `arith.gr`, with terminal `+`.
  - the number preceding rule $X \to Y Z$ is the rule's estimated *probability*, $\Pr(X \to Y Z \mid X)$, which is proportional to the number of times it was observed in training data. The probabilities for X rules already sum to 1—whereas in homework 1 you had to divide them by a constant to ensure this.
- These files are case-sensitive; for example, `DT` → `The` and `DT` → `the` have different probabilities in `wallstreet.gr`.

To prevent underflow, your parser should use weights internally instead of probabilities. You can convert the probabilities to weights as you read them, by taking their $-\log_2$.

As in homework 1, the grammar's start node is called `ROOT`. For each input sentence, your parser should print the single *lowest-weight* parse tree followed by its weight, or the word `NONE` if the grammar allows no parse. When you print a parse, use the same format as in your `randsent -t` program from homework 1.

(The required output format is illustrated by `arith.par`. As in homework 1, you will probably want to pipe your output through `prettyprint` to make the spacing look good. If you wish your parser to print useful information besides the required output, you can make it print comment lines starting with `#`, which `prettyprint` will preserve as comment lines.)

The weight of any tree is the total weight of all its rules. Since each rule's weight is $-\log_2 p(\mathit{rule} \mid \mathtt{X})$, where `X` is the rule's left-hand-side nonterminal, it follows that the total weight of a tree with root `R` is $-\log_2 p(\mathit{tree} \mid \mathtt{R})$.[2] (Think about why.) Thus, the highest-probability parse tree will be the lowest-weight tree with root `ROOT`, which is exactly what you are supposed to print.

[2] Where $p(\mathit{tree} \mid \mathtt{R})$ denotes the probability that if `randsent` started from nonterminal `R` as its root, it would happen to generate *tree*.

Not everything you need to write this parser was covered in detail in class! You will have to work out some of the details. **Please explain briefly (in your `README` file) how you solved the following problems:**

(a) Make sure not to do anything that will make your algorithm take more than $O(n^2)$ space or $O(n^3)$ time. For example, before adding an entry to the parse table (the main data structure shown on the slides, sometimes called the "chart" as in CKY), you must check in $O(1)$ time whether another copy is already there.

(b) Similarly, you only have $O(1)$ time to add the entry to the appropriate column if it is new, so you must be able to append to the column quickly. (This may be trivial, depending on your programming language.)

(c) For each entry in the parse chart, you must keep track of that entry's current best parse and the total weight of that best parse. Note that these values may have to be updated if you find a better parse for that entry.

You need not handle rules of the form $A \to \epsilon$. (Such rules are a little trickier because a complete entry from 5 to 5 could be used to extend other entries in column 5, some of which have not even been added to column 5 yet! For example, consider the case $A \to XY$, $X \to \epsilon$, $Y \to X$.)

What to hand in: **Submit your `parse` program (as well as answers to the questions above).** It might be fun to try it on the grammars that you wrote for assignment 1.

Hints on data structures:

- If you want to make your parser efficient (which you'll have to do for the next question anyway), here's the key design principle. Just think about every time you will need to look something up during the algorithm. Make sure that anything you need to look up is already stored in some data structure that will let you find it *fast*.
- Represent the rule $A \to WXYZ$ as a list (A, WX, Y, Z) or maybe (W, X, Y, Z, A).
- Represent the dotted rule $A \to WX.YZ$ as a pair $(2, R)$, where 2 represents the position of the dot and R is the rule or maybe just a pointer to it.

  (Another reasonable representation is just (A, Y, Z) or (Y, Z, A), which lists only the elements that have not yet been matched; you can throw W and X away after matching them. As discussed in class, this keeps your parse chart a little smaller so it is more efficient.)
- Represent each column in the parse chart as some kind of extensible vector, or a linked list with a tail pointer.
- The duplicate check discussed in (a) above could be handled by various means—dividing each column up into rows by start position (like CKY), using a hash table, etc. I strongly recommend a hash table because you will want something fast for problem 3.
- Use a few big hash tables, not lots of little hash tables. In particular, try to avoid arrays of hash tables, or hash tables of hash tables. Why? Each hash table has considerable memory overhead, e.g., lots of empty cells for future entries.

  In general, think about memory efficiency a bit. You'll need that in problem 3, when you'll deal with big grammars and parse charts.

- It can be wasteful to store multiple separate copies of a rule or entry. It is more economical to store multiple pointers to a single shared copy. In object-oriented terms, you want to avoid having several *equal* instances of an object—it's enough to have one instance and store it in several places.

- You might start out by building a weighted recognizer, which only finds the weight of the best parse, without finding the parse itself. Each entry in the parse chart must store a weight.

  If the entry is a dotted rule R, should the weight of its best parse include the weight of R itself? Doesn't matter, as long as the weight of R gets counted by the time you complete the rule. (All that really matters in the end is the weight of the whole-sentence parse tree . . .)

- To figure out how to print the best parse as well, as discussed in (c) above. you might want to review the slides from the "Probabilistic Parsing" lecture. The Earley technique is quite similar to the CKY technique. If you are clever, each entry only has to store *two backpointers* along with a weight. The backpointers must suffice for you to extract the parse at the end.

  Remember the idea of parsing: anything in the parse chart got there for a reason. It has an ancestry that explains how it got there, and the parse tree is just a way of printing out that ancestry. So each entry in the parse chart can point to its "progenitors" (i.e., the entries that combined to produce it), which in turn point to their progenitors, and so on.

  Hint: It turns out that entries added by a PREDICT step (such as $(3, A \to .BCD)$) don't actually need to point to anything. They don't have any substructure to remember, because they don't cover any words yet.[3]

Hints for avoiding some common pitfalls:

- Think in advance about the data structures you will need. Don't implement them until you're pretty sure they will work! Otherwise, you can waste a lot of time going down the garden path. :-)

  So draw your data structures and variables on paper first. Hand-simulate examples to make sure you've got all your bases covered. Try the example from the lecture slide. For example, you will need pointers or indices to locate the current (blue) rule; to move down the column to the next rule; to jump to column i to look for (purple) customers; etc. All of these basic operations should be fast (constant time).

  You are welcome to run your design by the course staff at office hours.

[3] If you still find that surprising, let's do a thought experiment to understand the role of these entries. Suppose you built a version of the Earley parser where every column was initialized to contain *every* rule with a dot at the start. For example, column i would contain the pair $(i, X \to .YZ)$ for every rule $X \to YZ$, on the theory that there is defintely an empty string from i to i that matches the part before the dot. This would be a perfectly accurate parser! It would just be slower than the real Earley's algorithm, because (like CKY) it would build whatever it could at position i without paying attention to the left context.

In this version, clearly entries with a dot at the start wouldn't need backpointers: they are spun out of thin air. And in the real Earley's algorithm, we can also regard such entries as spun out of thin air. It's just that to save time, we don't let them into the parse chart unless they have a "customer" looking for them. Nothing will point back to the customer until we have actually completed the constituent and ATTACHed it to the customer.

- Make sure you check for duplicates *whenever* you add an entry to a column, no matter how that entry got created.
- What does "duplicate" mean in practice? Duplicates are entries that are totally interchangeable except for having different weights. Then if you kill off the heavier one, the lighter one can play its role exactly the same, but more cheaply. So why not kill the heavier one? (It's like the plot of a bad political conspiracy thriller.)

  If two entries have different starting positions, or different ending positions (column), or different dot positions, then they're *not* duplicates. Both have to be kept alive because they can combine with different things. If you killed one off, the other one might not be enough to build a parse of the whole sentence.
- Suppose you are processing the entry

  $$(i, \text{NP} \rightarrow \text{Det N .})$$

  in column j. This newly completed NP spans the input substring from i to j. You should look only in column i for "customers" to ATTACH this new NP to. (Remember, column i contains entries whose dot has advanced up to position i in the input.) The parse chart is organized into columns specifically to facilitate this search.
- Remember that a SCAN action may have to apply to a dotted rule like

  $$\text{NP} \rightarrow \text{NP . and NP}$$

  where the thing after the dot is the terminal symbol "and." Make sure that your backpointers are general enough to handle this case. SCAN is actually very much like ATTACH—you are advancing the dot in a dotted rule; so, like ATTACH, it should result in a dotted rule with two backpointers.
- Use a recursive `print_entry` function to print the parse. When you write a recursive function and tell it to call itself, you should assume that that recursive call will "do the right thing." Concentrate on making the function itself do the right thing assuming that it can trust the recursive call.

  You should be able to call `print_entry` function on *any* entry in the parse chart. You know what is the "right thing" for `print_entry` to do on a complete entry, such as PP → P NP .: print a parse tree for that PP. But this should be accomplished, in part, by recursively calling `print_entry` on the two things that the entry points to. One of these will be a dotted entry. From this, you should be able to deduce what is the "right thing" for `print_entry` to do on a dotted entry.
- If you're using C++, the STL will work well. One thing to watch out for: you may want to iterate over the columns, but you can't use an STL iterator over a vector that changes during the iteration. (Just iterate with your own index.)

Allowed bug / extra credit: There is one subtle bug that you are *allowed* to have. Sometimes, *after* ATTACHing a completed constituent Z to its customer(s) Y to get X, you might end up building a lower-weight duplicate of Z. But oops—you already processed the higher-weight version of Z! Correctness demands that you re-process Z, which will ATTACH it again to Y to get a lower-weight duplicate of X. Unfortunately, if you have to process entries lots of times, your runtime can be worse than $O(n^3)$. So you have 3 options:

(a) Ignore the bug – don't re-process Z. This gives you an $O(n^3)$ algorithm that might occasionally find something other than the *lowest*-weight parse. You'll get full credit for this; the assignment is plenty hard already.

(b) Detect this case and re-process Z. This gives you a correct algorithm that no longer runs in $O(n^3)$.

(c) Find a way to fix the bug and still be $O(n^3)$ or close to it. This gets extra credit! I can think of two $O(n^3)$ solutions and one $O(n^3 \log n)$ solution ...

To help check your program: For grading, your program will be tested on new grammars and sentences that you haven't seen. You should therefore make sure it behaves correctly in all circumstances. To help you check, some simple `.gr` and `.sen` files are provided for you:

- Under `permissive.*`, every column of the parse chart should contain all (start position, dotted rule) entries that are possible for that column. Column n will contain $O(n)$ entries.
- Under `papa.*`, your program should exactly *mimic* the Earley animation slides from the "Context-Free Parsing" lecture. Compare and contrast!
- We give you a file `arith.par` that you can check your output against. Under `arith.*`, your output (if piped through `prettyprint`) should exactly match `arith.par`.
- You might also try `english.*`.[4]
- You might try writing some very small nonsense grammars, where you think you know what the right behavior is, and running the parser on those.

You can also compare your results to those of the `parse` program given with assignment 1:

```
/usr/local/data/cs465/hw-grammar/parse -tP -g arith.gr -i arith.sen
    # change -tP flag to -b for less detailed output (twice as fast)
```

3. It's always good to work with real data. A great deal of parsing research since 1995 has been based on the Penn Treebank (see question 1). And the parser you just wrote will actually get rather decent results on real English text by exploiting it, albeit with a few goofs here and there.

   The rules in `wallstreet.gr`, and their probabilities, have been derived from about half of the Treebank[5] by reading off the rules that were actually used by the human annotators. To keep the size more manageable, a rule was included in `wallstreet.gr` only if it showed up at least 5 times in the Treebank (this sadly kills off many useful vocabulary rules, among others). This is nonetheless a large grammar and you are going to feel its wrath.

   Some carelessly chosen sample sentences are in `wallstreet.sen`. I made up the first three; the rest are taken from a recent *Wall Street Journal*, with minor edits in order to change vocabulary that does not appear in the grammar.

   If you try running

[4] To produce `english.gr` from `english.gra`, use the `delattrs` script.

[5] Specifically, the sentences not containing conjunction, for reasons not worth going into here.

```
parse wallstreet.gr wallstreet.sen
```

you will get results, but they will take a long time even for the first sentence ("`John is happy .`") and a looooong time for the longer sentences. The problem is that there are a great many rules, especially vocabulary rules. You want to keep the parser from even thinking about most of those rules!

So you will have to implement a speedup method from the "parsing tricks" lecture. Using the first method listed below plus one other is probably enough to meet the requirement of the assignment: just that you make it through `wallstreet.sen` in a reasonable amount of time. But you can improve performance by combining more methods. Extra credit will be assigned for particularly fast or interesting parsers.

> Don't speed up your `parse` program. Make a copy of it, called `parse2`, and speed up the copy. You will hand in both programs, which will receive separate grades.
>
> ***What to hand in for this question:*** In addition to the source code of `parse2`, you must also hand in the output of `parse2` (i.e., the lowest-weight parse—if any—and its weight) for each sentence in `wallstreet.sen`. Submit this as a file `wallstreet.par`.
> In your `README` file, comment on any problems you see in the parses (as in question 1). Also describe in your `README` file what speedup method you used, and estimate how much speedup you got on short sentences (try `time parse ...` in Unix).

`parse2` should behave just like `parse`, only faster.[6] *Note:* The reason you are submitting both programs is only so that you can get full credit on `parse` even if `parse2` has a problem. If you don't want to bother with this, just submit `parse2` and let us know in your `README`.

You might enjoy typing in your own newspaper sentences and seeing what comes out. Just use the `checkvocab` script first to check that you're not using out-of-vocabulary words.

Some possibilities for speedups:

- (Strongly recommended.) Keep track of which categories have already been PREDICTed for the current column. If you're about to PREDICT a batch of several hundred NP rules (all rules of the form NP → . BLAH BLAH), then it should be a quick check to discover whether you've already added that batch to the current column.[7]
- Figure out which words are the terminals, and temporarily delete rules for terminals that aren't in the sentence.
- A pruning strategy (or better, an agenda-based, "best-first" strategy) lets you ignore low-probability rules or low-probability entries unless you turn out to really need them.

[6] With one exception. `parse` should always find the lowest-weight parse. `parse2` occasionally might not, if you chose to use an unsafe pruning method. But try to set the parameters of your pruning method so that `parse2` does seem to find the lowest-weight parse.

[7] Without this speedup, you would try to add all the rules in the batch, checking each *individually* (see 2a) to discover whether it was already there. This takes constant time but it's a big constant.

This approach is indispensable in the real world, where one wants to parse hundreds of sentences per minute. If you try an unsafe form of pruning, try to examine the effect on parse accuracy.

- Merge related entries in the parse chart, using one of the following safe strategies discussed in class:
  - Represent everything of the form $(3, X \to \cdots\ .\ CDE)$ as a single entry in the chart. Among other things, this reduces the number of entries that need to be ATTACHed, since now we just have the single entry $(3, X \to \cdots\ .\ )$ to say that there is a complete constituent X starting at 3, rather than many entries for complete X's with different internal structures.
  - Or, there is a simplified version that deals *only* with this important last case. Just say that $(3, X \to ABCDE\ .\ )$ is processed using a new COMPLETE operation that produces $(3, X)$ (duplicate copies of this are suppressed as usual). Then $(3, X)$ in turn is processed with ATTACH.
  - Or, represent everything of the form $(3, X \to AB\ .\ C \cdots)$ as a single entry in the parse chart.[8]

    Once you advance the dot past the C, you will have to find all D such that the grammar allows the dotted rule $X \to ABC.D\cdots$. You might additionally require D to be a left ancestor of the next word (that is, w_j in the terminology below).

    Hint: Representing the grammar rules in a trie gives you a natural implementation of partially matched rules that also allows you to move very fast from $X \to AB.C\cdots$ to $X \to ABC.D\cdots$. Other reasonable solutions also exist.
  - Or, as we saw in class, you can do even better by merging all the NP rules (for example) into a single finite-state automaton, and representing a dotted NP rule as a state in this automaton.

    This subsumes the speedups above, and is some sense the "right" way to do it.
- You will often have to search column i for all entries with X after the dot (for a given i and X). If you store column i as a single indiscriminate list, this requires examining *every* entry in column i. Can you design a better way of storing or indexing column i, so that you can quickly find *just* the entries with X after the dot?
- Some kind of left-corner method. *I can confirm* from direct experience that the following version[9] suffices to make parsing time tolerable (though still slow) for this problem:

  Represent the grammar in memory as a pair of hash tables, which your parser can construct as it reads the `.gr` file:
  - The **prefix table** R: $R(A, B)$ stores the set of all grammar rules of the form $A \to B\cdots$.
  - The **left parent table** P: $P(B)$ stores the set of all A such that there is at least one grammar rule of the form $A \to B\cdots$. (B is said to be the "left child" of A, so we may as well call A a "left parent"of B.)

  When you read a grammar rule of the form $A \to B\cdots$, simply add A to $P(B)$ iff $R(A, B) = \emptyset$ (this test avoids duplicates in $P(B)$) and then add the rule itself to $R(A, B)$.

[8] Then you'll no longer need the first speedup (tracking which categories have already been PREDICTed). Why not?

[9] Which would not work in quite this form if $A \to \epsilon$ rules were allowed; but fortunately we're not allowing them for this problem.

Let w_j be the word that starts at position j. Before you begin to process entries on column j, construct a third hash table that will only be used during processing of that column:

- The **left ancestor pair table** S_j: $S_j(A)$ stores the set of all B such that A is a left parent of B and B is a left ancestor of w_j. (That is, $A \in P(B)$, and either $B = w_j$ or $B \in P(w_j)$ or $B \in P(P(w_j))$ or $\ldots$)

It is reasonably straightforward and very fast to compute S_j by depth-first search. The basic step is to "process" some Y (initially w_j itself) by adding Y to $S_j(X)$ for each $X \in P(Y)$. Where this was the first addition to $S_j(X)$, recursively process X.[10]

Now, when you are processing column j, you will use S_j to constrain the PREDICT operation that starts new rules. When you need to add $A \to \cdots$ rules to the parse chart, you should add exactly the rules in $R(A, B)$ for each $B \in S_j(A)$. (A further trick is that once you have added these rules, you can set $S_j(A) = \emptyset$. Do you see why this is okay and how it helps?)

Notice that w_j itself was the only terminal you considered during this whole process—you were not bogged down by the rest of the vocabulary.[11]

Some of you may not have previously been in classes where your programs take hours to run. Some comments about how to deal with this:

- Why will your program be slow? `wallstreet.gr` is a large, permissive grammar with many long rules (e.g., have a look at the set of NP rules). So the Earley parse chart will be quite large. And the undergrad machines are not especially fast.
- Leave time to compute, and recognize that you will be competing for the same processors. The machines `ugrad1` through `ugrad24` have 4 processors each. So basically only 4 of you at once can share a single machine. If 8 jobs are running at once, then they all run *less* than half as fast: there is added overhead as the OS juggles the jobs. Use the `top` command to see what jobs are running and what resources they are consuming. All of the `ugrad` machines share a filesystem and should behave alike.

[10] Why only on the first addition? Because you mustn't process any symbol more than once. If you did, you might end up adding duplicates to $S_j(X)$, or even looping forever, e.g. if X is its own left grandparent.

[11] Here's an example of the left-corner method. Suppose w_j is the word `lead`, which could be either a noun or a verb. Then $P(w_j) =$ {`N`,`V`}. Moreover, suppose the grammar is such that

P(`N`)	=	{`NP`}	
P(`V`)	=	{`VP`}	
P(`NP`)	=	{`NP`,`S`}	so `NP` can be the first child of either `NP` or `S`
P(`VP`)	=	{`VP`}	so `VP` can be the first child only of `VP`
P(`S`)	=	{}	so `S` can't be the first child of anything

Then

S_j(`N`)	=	{`lead`}	so Predict(`N`) adds all `N` → . `lead` ... rules via R(`N`,`lead`)
S_j(`V`)	=	{`lead`}	so Predict(`V`) adds all `V` → . `lead` ... rules via R(`V`,`lead`)
S_j(`NP`)	=	{`N`,`NP`}	so Predict(`NP`) adds all `NP` → . `N` ... rules via R(`NP`,`N`) and all `NP` → . `NP` ... rules via R(`NP`,`NP`) *but* does not add any `NP` → . `Det` ...rules, since `lead` can't be the first word of a `Det`
S_j(`VP`)	=	{`V`,`VP`}	so Predict(`VP`) adds all `VP` → . `V` ... rules via R(`VP`,`V`) and all `VP` → . `VP` ... rules via R(`VP`,`VP`)
S_j(`S`)	=	{`NP`}	so Predict(`S`) adds all `S` → . `NP`... rules via R(`S`,`NP`) *but* does not add any `S` → . `PP` ... rules, since *lead* can't be the first word of a `PP`

You had to recurse during the construction of S_j to find all the nonterminals that `lead` could be the first word of.

- If you have access to other machines (CS research network, your own computer, etc.), you are free to use them so long as the final program you submit will run on the `ugrad` machines.
- For most debugging, you'll want to use smaller grammars or shorter sentences where things run fast.
- Don't fill up all the available memory. If you do, the OS will start using the disk as auxiliary storage, making things extremely slow. You can check the size and CPU usage of running processes by typing `top`.
- If you are using too much memory, it may mean that you are not eliminating duplicates correctly. Or it may mean that you designed your program to have many little hash tables (see discussion at problem 2).
- Again, for comparison, my compiled LISP program took about 70 seconds (and 30 MB of memory) to get through the 9 sentences in `wallstreet.sen`. The first sentence took only 1 second because it is short, but the algorithm is $O(n^3)$, so longer sentences take *much* longer.
- The `parse` program from HW1 (written in the old version of Dyna) is 10 or 20 times faster than mine. In past years, some students have managed to write parsers that run (I think) about 50 times faster than mine.
- As you probably noticed in question 1, "real" parsers run at hundreds of sentences per minute despite having more complicated probability models. How?
  - probabilistic pruning—very important![12]
  - careful code optimization
  - merging the grammar rules into finite-state automata, as we discussed in class; this avoids dealing separately with all of the similar long rules
  - "dependency parsing" formulations that only find a tree structure without any nonterminals

  You are certainly welcome to use any of these techniques (other than dependency parsing), but you are not required to. It is up to you how you want to balance programming time and runtime, so long as you implement some non-trivial speedup.

To help check your program:

- You can run many of the same checks that were suggested in problem 2.
- Your new parser is just a fast version of your old one. So try them on some of the same examples and make sure that they get the right answer.
- Tracing is wise. An Earley parser can still get the right answer even if it adds way too many dotted rules to the parse chart (unnecessary rules, duplicate rules, rules that

[12] Recently, "coarse to fine" techniques have become popular. Here, a simpler parser is run first, and its output is represented as a packed forest of reasonable parses. The simpler parser is fast because it considers fewer nonterminals (e.g., with fewer attributes) or has a simpler probability model. If the simpler parser doesn't think that a certain substring can reasonably be analyzed in context as an NP, then it doesn't make sense for the complex parser to spend time figuring out whether it is an NP[plural,case=accusative,gender=fem,head=senator]. So the complicated parser focuses only on constituents that are compatible with the parses produced by the simpler parser.

are inconsistent with the left context, etc.). It will just be slower than necessary. So use some kind of tracing to examine what your parser is actually doing . . . Just print comment lines starting with `#`; such lines will be passed through by `prettyprint` and ignored by the graders.

- For the first two sentences in `wallstreet.sen`, the lowest-weighted parses have weights of 34.2234 and 104.9087 respectively. If you have the allowed bug discussed on page 7, you may get a higher-weighted parse for the second sentence, usually of weight 113 or so.