600.465 — Intro to NLP
Assignment 3: Parsing and Semantics

Prof. J. Eisner — Fall 2006
Due date: Monday 13 November, 2 pm

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.

In the second half of the assignment, you will run your parses through a post-processing script that computes their features, including semantic features. You will be asked to understand and tweak the grammar that assigns the features.

Dividing the assignment into these two halves is largely a matter of convenience. It would obviously improve accuracy for your parser to compute constituents' features *during* parsing, as humans probably do. Then the parser could rule out some constituents, or give them lower probabilities, on the basis of feature mismatch (e.g., you can't combine a singular subject with a plural verb). In general the parser could use the features to help compute less biased probabilities. However, it is easier for you to write a faster parser that doesn't have to worry about features at all.[1]

All the files you need can be found in `http://cs.jhu.edu/~jason/465/hw3`. You can download the files individually as you need them, or download a zip archive that contains all of them. Read Figure 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.

Programming language: You may write your parser in any programming language you choose (except Dyna), so long as the graders can run it on `barley`. 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).

[1] And it may even be good engineering. You might be interested to know that most modern probabilistic English parsers compute little more than the **head** feature while parsing. Conditioning the probabilities on the head feature makes them substantially more accurate, and this accuracy is useful. But syntactic features such as **agreement** rarely help for choosing among probable parses. The more detailed **semantics** we will consider here could help in principle, but only in a system that can reason about the semantics and relate it to a database of knowledge about the world in order to decide whether a constituent is plausible.

`*.grf`	full grammar with rule *frequencies*, features, comments
`*.gr`	simple grammar with rule *weights*, no features, no comments
`delfeats`	script to convert `.grf` $\longrightarrow$ `.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
`buildfeats`	script to convert `.par` $\xrightarrow{\texttt{.grf}}$ a feature assignment
`parsefeats`	simple script to convert `.sen` $\xrightarrow{\texttt{.grf}}$ a feature assignment (calls `checkvocab`, `parse`, and `buildfeats`)
`simplify`	script that lets you experiment with lambda terms

Figure 1: Files available to you for this project.

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. For example, my compiled LISP program—*with* the PREDICT and left-corner speedups in problem 2—took 75 minutes on `barley` to get through the nine sentences in `wallstreet.sen`. For many programs, C/C++ will run a few times faster than compiled LISP. Java will run slightly faster than LISP. But interpreted languages like Perl will run several times *slower*. So if you want to use Perl, think twice and start extra early.

> *Java hint:* 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:* 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/.

On getting programming help: Same policy as on assignment 2. (Roughly, feel free to ask anyone for help on how to use the features of the programming language and its libraries. However, for issues directly related to NLP or this assignment, you should only ask the course staff 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 and your modified `.grf` 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 `barley`.

1. 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
     - the number preceding rule $X \rightarrow Y Z$ is the rule's *weight*, $-\log_2 \Pr(X \rightarrow Y Z \mid X)$.
       (By contrast, in homework 1 it was the rule's *frequency*, i.e., a number that is proportional to $\Pr(X \rightarrow Y Z \mid X)$ and is typically the number of times the rule was observed in training data.)
     - 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 `delfeats`.
     - you can assume that every rule has at least one element on the right-hand side. So $X \rightarrow Y$ is a possible rule, but $X \rightarrow$ or $X \rightarrow \epsilon$ is not. This restriction will make your parsing job easier.
   - These files are case-sensitive; for example, `DT` → `The` and `DT` → `the` have different probabilities in `wallstreet.gr`.

   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 delete.)

The weight of any tree is the total weight of all its rules. Since each rule's weight is $-\log_2 p(rule \mid \texttt{X})$, where `X` is the rule's left-hand-side nonterminal, it follows that the total weight of a tree with root `R` is $-\log_2 p(tree \mid \texttt{R})$.[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.

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, 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 if it is new, so you must be able to find the bottom of the appropriate column quickly. (This may be trivial, depending on your programming language.)

(c) For each entry in the parse table, 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 $A \rightarrow \epsilon$. (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 $A \rightarrow XY$, $X \rightarrow \epsilon$, $Y \rightarrow X$.)

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 $A \rightarrow WXYZ$ as a list (A, WX, Y, Z) or maybe (W, X, Y, Z, A).
- Represent the dotted rule $A \rightarrow WX.YZ$ 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 table a little smaller so it is more efficient.)

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

- Represent each column in the parse table 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 2.
- 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 2, when you'll deal with big grammars and parse tables.
- 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 table 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 table 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 table 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 $(3, A \rightarrow .BCD)$) 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]

[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

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.

- 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 table is organized into columns specifically to facilitate this search.

---

with a dot at the start. For example, column i would contain the pair $(i, X \rightarrow .YZ)$ for every rule $X \rightarrow YZ$, 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 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.

- 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 table. 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 table 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.

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.

2. It's always good to work with real data. In class we discussed the Penn Treebank, a collection of manually built parses covering about a million words (40,000 sentences) of *Wall Street Journal* text. A great deal of parsing research since 1995 has been based on this corpus. 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.

   You must hand in your parser's output (i.e., the lowest-weight parse—if any—and its weight) for each sentence in `wallstreet.sen`. Submit this as a file `wallstreet.par`.

[4] To produce `english.gr` from `english.grf`, use the `delfeats` script.

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

S	Sentence or clause.
SBAR	Clause introduced by a (possibly empty) subordinating conjunction.
SBARQ	Direct question introduced by a *wh*-word or *wh*-phrase.
SINV	Inverted declarative sentence.
SQ	Inverted yes/no question, or main clause of a *wh*-question.
ADJP	Adjective Phrase.
ADVP	Adverb Phrase.
CONJP	Conjunction Phrase.
FRAG	Fragment.
INTJ	Interjection.
LST	List marker. Includes surrounding punctuation.
NAC	Not A Constituent; used within an NP.
NP	Noun Phrase.
NX	Used within certain complex NPs to mark the head.
PP	Prepositional Phrase.
PRN	Parenthetical.
PRT	Particle.
QP	Quantity Phrase (i.e., complex measure/amount) within NP.
RRC	Reduced Relative Clause.
UCP	Unlike Coordinated Phrase.
VP	Verb Phrase.
WHADJP	*Wh*-adjective Phrase, as in *how hot*.
WHADVP	*Wh*-adverb Phrase.
WHNP	*Wh*-noun Phrase, e.g. *who*, *which book*, *whose daughter*, *none of which*, or *how many leopards*.
WHPP	*Wh*-prepositional Phrase, e.g., *of which* or *by whose authority*.
X	Unknown, uncertain, or unbracketable.

-ADV	Constituent is used adverbially
-LOC	Constituent indicates event location
-PRD	Constituent serves as a sentence's predicate but is not a VP
-NOM	Constituent is used as a noun, e.g., *what I really like* is chocolate
-TMP	Constituent indicates when, how often, how long

CC	Coordinating conjunction
CD	Cardinal number
DT	Determiner
EX	Existential *there*
FW	Foreign word
IN	Preposition or subordinating conjunction
JJ	Adjective
JJR	Adjective, comparative
JJS	Adjective, superlative
LS	List item marker
MD	Modal
NN	Noun, singular or mass
NNS	Noun, plural
NNP	Proper noun, singular
NNPS	Proper noun, plural
PDT	Predeterminer
POS	Possessive ending
PRP	Personal pronoun
PP$	Possessive pronoun
RB	Adverb
RBR	Adverb, comparative
RBS	Adverb, superlative
RP	Particle
SYM	Symbol (mathematical)
TO	The word *to*
UH	Interjection
VB	Verb, stem
VBD	Verb, past tense
VBG	Verb, present participle
VBN	Verb, past participle
VBP	Verb, present but not VBZ
VBP	Verb, present, 3rd-person sing.
WDT	*wh*-determiner
WP	*wh*-pronoun
WP$	Possessive *wh*-pronoun
WRB	*wh*-adverb

Figure 2: Nonterminals in `wallstreet.gr` (from Marcus, Santorini and Marcinkiewicz 1993). Preterminals are shown separately, omitting punctuation-mark preterminals, which are trivial.

In your `README` file, comment on any problems you see in the parses; you may find Figure 2 and the `prettyprint` script helpful.

If you try running

```
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 make it through `wallstreet.sen`. But you can improve performance (and maybe get extra credit) by combining more methods.

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.[6]
- 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. 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.
- Build a trie that allows you to represent everything of the form $(3, A \rightarrow B.C \cdots)$ as a single entry in the parse table.

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

  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.

[6] Without this speedup, you would try to add all the rules in the batch, checking each *individually* (see 1a) to discover whether it was already there. This takes constant time but it's a big constant.

- You will often have to search column i for all entries with X after the dot (for some 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[7] 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 $R(A, B)$.

  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 $\dots$)

  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.[8]

  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 table, 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?)

[7] 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.

[8] 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.

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.[9]

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 table 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. `barley` has 4 processors, so basically only 4 of you can run intensive jobs on it at once. 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.
- Fortunately, you can also use the machines `ugrad1` through `ugrad18`, which have one processor each. These machines share a file system with `barley` and should behave identically, except that they will run out of memory sooner. As far as I can tell, they also run at about the same speed.
- 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 `barley`.
- For most debugging, you'll want to use smaller grammars or shorter sentences where things run fast.

---

[9] 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.

- 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 1).
- Again, for comparison, my compiled LISP program took about 75 minutes and 42M of memory to get through `wallstreet.sen`. The first sentence took only 1 minute because it is short, but the algorithm is $O(n^3)$, so longer sentences take *much* longer.
- If I recall, the class record was set in 2004 by Johnny Graettinger, whose program took about 1 minute total on `wallstreet.sen`.
- For the record, "real" parsers run at hundreds of sentences per minute despite having more complicated probability models. How?
  - probabilistic pruning—very important!
  - 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

  You are certainly welcome to use any of these techniques, 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 1.
- 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 table (unnecessary rules, duplicate rules, rules that 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 `#`, which will be deleted by `prettyprint` and ignored by the graders.
- For the first two sentences in `wallstreet.sen`, the lowest-weighted parses have weights of 34.2301 and 104.9127 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 113.1897.

Your new parser should be called `parse2` and should behave just like `parse`, only faster.[10] Submit the code as well as its output `wallstreet.par` and your discussion of the output. 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).

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.

3. We now turn to semantics. Parsing speed will not be a big issue for this part of the assignment, so you can use `parse` again rather than `parse2`.

   Your first job is to understand the notation for adding features to a grammar. A grammar with features is a `.grf` file; the corresponding `.gr` file can be produced by using `delfeats` to strip the features and comments. You have been given some simple `.grf` files that demonstrate the different features of the notation.

   (a) Read the file `arith.grf` carefully and examine the output of the following commands, especially the part of the output that is *not* indented:

   ```
   parse arith.gr arith.sen > arith.par
   buildfeats arith.grf arith.par
   ```

   The output for each parse is an indented trace, showing how the features for each constituent are built bottom-up. The traces for different parses are separated by `---`.

   At the end of a trace (not indented) is the final result: the features for the parse as a whole. This is what you should usually study, but if something is mysterious you can look earlier in the trace to see how the parse's features arose from those of smaller constituents.

   (b) Now study `arith-infix.grf` and try

   ```
   buildfeats arith-infix.grf arith.par
   ```

   (c) Finally, study `arith-typed.grf` and try

   ```
   parse arith.gr arith-typed.sen > arith-typed.par
   buildfeats arith-typed.grf arith-typed.par
   ```

[10] 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.

Note that you can abbreviate this process using the `parsefeats` script, which also does some other nice things for you (look at the script to see what!):

```
parsefeats arith-typed.grf arith-typed.sen
```

There is nothing to hand in for this question—just make sure you understand what's going on before it gets more confusing!

4. `times(x,y)` is all very well, but to build interesting natural-language semantics we are going to have to use lambda terms. So here are some simple exercises—you don't have to hand the answers in.

   You can check your answers using the `simplify` script, which will simplify any lambda-expression you type in. (Look at the top of the script for documentation. Start it by typing `./simplify` with *no arguments*.) But try to come up with each answer on your own first ...

   (a) Simplify $(\lambda x\ x * x)3$.

   (b) Simplify $(\lambda x\ x * x)(y + y)$.

   (c) Simplify $(\lambda x\ x * x)y + y$.

   (d) Simplify $(\lambda a\ a)(\lambda b\ f(b))$.

   (e) Simplify $(\lambda a\ 3)(\lambda b\ f(b))$.

   (f) Simplify $(\lambda x\ green(x))(y)$. Since the result holds for any y, what do you conclude about the relation between $\lambda x\ green(x)$ and $green$?

   (g) Simplify $(\lambda x\ \lambda y\ ate(x, y))(lemur, leopard)$.

   (h) Simplify $(\lambda x\ \lambda y\ ate(x, y))(lemur)$.

   (i) Apply the previous answer to "leopard": simplify $(\lambda x\ \lambda y\ ate(x, y))(lemur)(leopard)$.

   (j) Simplify $(\lambda x\ f(x, y))(a)(b)(c(z))$.

   (k) Simplify $(\lambda x\ f(x, y))(a, b, c(z))$. This is just an abbreviation for the previous case.

   (l) Simplify $(\lambda f\ f(x))g$.

   (m) Simplify $(\lambda f\ f(f(f(x))))g$.

   (n) Simplify $(\lambda f\ f(f(f(x))))(\lambda t\ a(c(t)))$.

   (o) Simplify $(\lambda f\ f(f(f(x))))(\lambda t\ t * t)$.

   (p) Simplify $(\lambda f\ f(f(f(x))))(\lambda t\ a(b, c[t], d))$.

Feel free to play around more with `simplify`. You can actually do some outrageous things with it, including using lambda terms to represent integers, pairs, stacks, conditionals, recursion, loops, and in fact any Turing machine. (Can you write an expression whose simplification doesn't terminate?) More information is at the top of the file `LambdaTerm.pm`.

5. For these, you should hand your answers in. (Put them in your `README`, using the same notation used by `simplify`. You can use `simplify` to check your answers.)

   Several of these are basically division problems (analogous to "If $x \cdot 3 = 21$, what is x?"). For example, if $f(6) = 6 \cdot 6$, then what is f? Answer: $f = \lambda x\ x \cdot x$. That's all there is to it.[11]

   (These "division" problems are related to the end of the semantics lecture. We wanted a particular meaning for "Every nation wants George to love Laura," and worked backwards to figure out what functions f should be associated with the words. Those slides will make more sense once you've done this question.)

   (a) Suppose $f(John) = loves(Mary, John)$. What is f,
      i. written in the form $\lambda x\ \cdots$?
      ii. written without any λ ?
      (For example, $(\lambda x\ x)(3)$ can be written as 3. $\lambda x\ s(x)$ can be written as s.)

   (b) In our semantics, $loves(Mary, John)$ will be the interpretation of "John loves Mary," not vice-versa. This is just more convenient because then the VP in that sentence has a nice, compact semantics. Namely, what?

   (c) Suppose $f(John) = (\forall x\ woman(x) \Rightarrow loves(x, John))$.
      i. What is f?
      ii. Translate f and $f(John)$ into English.

      Note: To type an expression such as $\forall x\ woman(x) \Rightarrow loves(x, John)$ into the `simplify` script, write something like `A%x woman(x) => loves(x,John)`.[12] Notating $\forall$ as `A%` (or anything ending in `%`) tells `simplify` that the following x is a dummy variable, not a constant.

   (d) Suppose $f(\lambda x\ loves(Mary, x)) = (\lambda x\ Obviously(loves(Mary, x)))$. What is f and how would you use it in constructing the semantics of "Sue obviously loves Mary?"

   (e) Suppose $f(\lambda x\ loves(Mary, x)) = (\forall y\ woman(y) \Rightarrow loves(Mary, y))$.

[11] Other possible answers are $f = \lambda x\ 6 \cdot x$, $f = \lambda x\ x + 30$, and $f = \lambda x\ 36$. These are technically correct, since $f(6) = 36$ in all these cases. But $f = \lambda x\ x \cdot x$ is the answer we'd be looking for.

[12] This particular expression cannot be simplified further by `simplify`, so don't be alarmed if you type it in and it comes right back at you.

i. What is f?

ii. Translate $f(\lambda x\ loves(Mary, x))$, $(\lambda x\ loves(Mary, x))$, and f into English.

(f) Let f be your answer from question 5(e)i. Suppose $g(woman) = f$.

i. What is g as a lambda term?

ii. What English word does it represent?

Hint: Substituting $g(woman)$ for f in question 5e yields $g(woman)(\lambda x\ loves(Mary, x)) = (\forall y\ woman(y) \Rightarrow loves(Mary, y))$. If you replaced every other term in this equation with the English phrase of which it is the semantics, then what would you have to replace g with?

(g) Suppose $f(\lambda x\ loves(Mary, x)) = loves(Mary, Papa)$.

i. What is f as a lambda term?

ii. Why would one want to give Papa these funny semantics (rather than just `sem=Papa`, as in the original `english.grf`)? (*Hint:* Look back at question 5e, translate both expressions into English, and think "consistency.")

6. Now you're ready to look at a (small) English semantic grammar: study `english.grf`. The syntactic coverage is nowhere near that of the Penn Treebank's grammar, but it does have semantics.

   Try running (as in question 3c)

   ```
   parsefeats english.grf english.sen
   ```

   This will convert the grammar to a `.gr` file, parse some English sentences using *your* Earley parser, and then assign features with `buildfeats`.

   Each of the 22 sentences in `english.sen` should have yielded a parse with your parser. For each sentence, inspect its features and decide whether they are appropriate:

   - For a grammatical sentence, did the system find the most plausible semantics?
   - For an ungrammatical sentence, did the system print the message "there is no consistent way to assign features"?

   List the sentences where you think the feature assignment may be inappropriate, and explain why. In each case, say whether it would have helped if the parser had chosen a different valid parse of the same sentence. (Remember, the parser uses probabilities but without considering features, and then `buildfeats` is stuck computing features for whatever the parser chose.) If so, what parse would have worked better?

   For example, if the sentence is

   ```
   Meilin saw a bird with the telescope
   ```

then you should notice a problem if the representation is

```
Past(see(a(%x bird(x) ^ with(the(telescope),x)),Meilin))
```

since that says that the bird has the telescope. Probably this is *not* the semantics that the author of the sentence intended. A different parse would have gotten the correct semantics.

Important: Don't kill yourself. Once you are sure your parser is working, you don't have to pore over the output for hours with a monocle and tweezers. Just try to find the major problems and briefly say why they are problems. We won't penalize you for missing a few. The point of this problem is not to torture you, only to make you stare at the output long enough to Understand™ what's going on and make some intelligent comments.

Certainly you do not have to second-guess the *style* of the representations. That is,

```
Past(with(the(telescope),see(a(bird),Meilin)))
```

may not be the ideal semantic representation, since the handling of prepositions, determiners, and tense is pretty primitive. But it is reasonable enough that you needn't take issue with it.

7. In `english.grf` and `english.sen`, Papa is eating bonbons rather than caviar. This is because I couldn't figure out whether *caviar* was singular or plural. You say "All caviar is delicious," but *all* only combines with plural nouns and *is* only combines with singular nouns . . . so which is *caviar*?

   In fact, caviar (like chocolate and dirt and camera film) is what is called a "mass noun." Modify `english.grf` to admit *three* values for the `num` feature: `sing`, `pl`, and `mass`. Add *caviar* as a mass noun. Make sure that mass nouns work correctly both with verbs (which always treat them as singular) and with determiners (which don't).

   To do this, you'll need to work out the facts about which determiners can go with which nouns. You may want to make a grid of determiners versus nouns and see which ones can combine, using the vocabulary in `english.grf`. You'll notice that mass determiners are always plural determiners as well (*all caviar* → *all bonbons*) but not vice-versa (*two bonbons* ↛ **two caviar*).[13]

[13] It would be nice to capture this asymmetric generalization with a rule like `N[num=pl]` → `N[num=mass]`, which says that mass determiners can always be used where plural determiners are called for. One could similarly write `NP[num=sing]` → `NP[num=mass]`, which says that mass NPs can be used to agree with singular verbs or singular pronouns. Unfortunately, these elegant rules introduce an extra `NP` node into the tree when mass nouns are involved. That would require the shape of the tree to be affected by the `num` features. So they won't work with our system, which parses *before* it looks at the features.

Try to handle these inelegant facts elegantly, using as small and simple a system of rules as you can under the circumstances. Submit your modified `english.grf`. Run it on a few sentences about caviar—both grammatical and ungrammatical ones—and report what happened.

8. `english.grf` doesn't attempt any real semantics for determiners. In particular, quantifiers like "every" are left as atomic elements with no internal semantics.

   `english-fullquant.grf` fixes this, reorganizing the grammar along the lines you explored in questions 5e–5g.[14] At the end of the semantics lectures, we also handled "every nation" in this style—you could review those slides.

   Try `parsefeats english-fullquant.grf english.sen` to see the new form of the output. Study `english-fullquant.grf` to see how it's done; just look at the changes, which are marked with `***`.

   (a) The new grammar gives pretty complicated semantic features to *two* and to singular and plural *the*. Justify the features it uses (i.e., explain what those lambda-terms mean). The `!` symbol means "not."

   (b) The semantics of one rule in the new grammar has been left as `???`. It affects the sentence *Papa want -ed George to eat a pickle*. What should replace the `???`? (Try your answer out, but see if you can get it without trial and error! It's hard to wrap your brain around, I know.)

[14] This approach (due to Montague) is called the "Proper Theory of Quantification" because it says that proper nouns have the same semantic type as NPs containing quantifiers.