601.465/665 — Natural Language Processing
Homework 4: Parsing

Prof. Jason Eisner — Fall 2025
Due date: Monday 20 October, 11 pm

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

Homework goals: This is the primary algorithms homework for this class. After completing it, you should be comfortable designing and implementing dynamic programming algorithms. You should have a strong understanding of

- how to maintain a record of the items (partial constituents) that have been built so far
- how to maintain each item's optimal derivation and its weight
- how to index these data structures for quick lookup when building new items
- how to organize the computation so that all possible items will eventually get built
- how to speed up the algorithm by skipping unnecessary work (pruning/prioritization/filtering), avoiding duplicate work (merging items), and improving the data structures (indexing)
- why the algorithm runs in $O(n^3)$, and what changes would put that at risk

Collaboration: ***You may work in pairs on this homework,*** 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 PDF submission should describe at the top what each of you contributed, so that we know you shared the work fairly.
3. Your partner can't be the same as your partner from homework 3.

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

Reading: Read the handout attached to the end of this homework!

Materials: 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! Examine the grammars, and of course read the starter code carefully if you choose to use it.

Programming language and starter code: We are providing some starter code in Python. For this assignment, however, Python is not required. We're giving you the liberty to write the

`*.gr`	simple grammar with rule *probabilities*
`*.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
`recognize.py`	starter code: an unweighted recognizer

Table 1: Files available to you for this project.

program in any language you choose (except Dyna); see reading section C for discussion. Even if you pick another language, you may still find the starter code useful to read.

The starter code is a complete implementation of an Earley's *recognizer*. This builds a chart of partial constituents from left to right using SCAN, PREDICT, and ATTACH operations. However, it doesn't save back-pointers, doesn't construct the parse tree, and doesn't handle weights. It just tells you whether each sentence is accepted or rejected by the grammar.

You can find the starter code `recognize.py` and some brief `INSTRUCTIONS` at `http://cs.jhu.edu/~jason/465/hw-parse`.

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

How to hand in your written work: Via Gradescope as before. Besides the comments you embed in your source files, put all other notes, documentation, and answers to questions in a PDF file.

How to test and hand in your programs: Similar to the previous homework. An autograder will test your code on some new sentences, reporting both runtime and errors. We will post more detailed instructions on Piazza.

1. A "phrase-structure parser" (or "constituency parser") constructs trees like those we've seen in `randsent`, CKY, and Earley's algorithm.

   To familiarize yourself with parsing, find a state-of-the art phrase-structure parser of English, and try it out.

   You don't have to spend a long time on this question, but experiment by having it parse at least a few sentences. You could make up some interesting sentences, or copy some from a book or website. **In your PDF submission, discuss what you learned, for example:**

   1 (a) Was there anything interesting or surprising about the style of trees produced by the parser? (Not the visual style, but the choices about what kind of structures to assign to the sentences.)

   2 (b) What were some things that the parser got wrong? What were some hard things that it managed to get right? (This question is asking about "ordinary" sentences of the kind that it probably saw in training data.)

3 (c) Can you design a grammatical sentence that confuses the parser in a way that you intended, even though it is familiar with the words in your sentence? (This question is asking about "adversarial" sentences that are intended to stump the parser, perhaps because they do not resemble the ones in training data.)

Suggestions: Some modern parsers are listed at http://nlpprogress.com/. The following parsers have online demos, so they are easy to try:

- Berkeley Neural Parser: https://parser.kitaev.io/
- AllenNLP Constituency Parser: https://demo.allennlp.org/constituency-parsing/
- Stanford Parser: https://corenlp.run/ (make sure to select "constituency parsing") (for English, Chinese, Arabic, . . .)

Understanding the output: 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.

How does the parser work? If you're curious about the parser you tried, you can easily find the paper that describes it. Use Google, or search at http://www.aclweb.org/anthology/, where most NLP papers are permanently available for free. Ask on Piazza if you need help finding or understanding it.

Most recent parsers use neural networks ("deep learning") to choose good parsing actions. Often the neural networks are used to evaluate each action in the context of the entire sentence or the entire parse, which makes the parser non-context free. We'll discuss these ideas later in the class.

2. *Extra credit:* Experiment also with some other kinds of parsers. You could try any or all of the questions below.

(a) Not all parsers produce phrase-structure trees of the sort we've been studying in class. They may produce other kinds of sentence diagrams. Here are some other good parsers with online demos. Play with them as well and report on what you find. ✌4

Dependency grammar spaCy — https://explosion.ai/demos/displacy (you should uncheck "Merge Phrases" to see the full parse)

Link grammar Link Grammar Parser: http://www.link.cs.cmu.edu/link/submit-sentence-4.html (uses a hand-written grammar)

Head-driven phrase-structure grammar PET system parser: http://erg.delph-in.net/ (uses the hand-written English Resource Grammar)

Combinatory categorial grammar C&C parser: http://svn.ask.it.usyd.edu.au/trac/candc/wiki/Demo#Tryityourself (alas, this online demo is now down, but you can download the C++ parser from https://github.com/chrzyki/candc, or a Java port from https://github.com/darrenfoong/candc)

(b) If you speak a language other than English, find a parser that can run on that language and experiment with it. What did you find out? Are the parse trees in that language in the same style as in English? Do the parses appear to be as accurate? ✌5

3. Write a probabilistic Earley parser that can be run as

```
./parse.py 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, but easier:
  - you can assume that the file format is simple and rigid; predictable whitespace and no comments. (See the sample `.gr` files for examples.)
  - 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.
  - you may 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 (see reading section B.3).
- these files are case-sensitive; for example, `DT` $\to$ `The` and `DT` $\to$ `the` have different probabilities in `wallstreet.gr`.

The *weight* of a rule or a parse will be its negative $\log_2$-probability, measured in bits. Thus, the minimum-weight parse is the maximum-probability parse. Your parser should convert probabilities to weights as it reads the grammar file. It can work with weights thereafter.[1]

[1] Alternatively, if you prefer, you can do the conversion at output time instead of at input time. That is, following Appendix G of the previous homework, your parser can work with probabilities as long as you represent them internally by their logs to prevent underflow. You would then convert log-probabilities to weights only as you print out your results. But this is really the same as the previous strategy, up to a factor of $-\log_2$.

As in homework 1, the grammar's start nonterminal is called `ROOT`. For each input sentence, your parser should print the single *minimum-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 use the `logging` module as illustrated in the starter code.)

We have provided starter code for you, which implements an unweighted recognizer. See the `INSTRUCTIONS` file to get started.

Not everything you need to write this parser was covered in detail in class! You will have to work out some of the details. See reading sections A to C for hints. **Please explain briefly (in your PDF submission) how your submitted code solved the following problems.** Some of your submitted code may have been copied or adapted from the starter code, but you still have to explain why the whole solution works, including the parts we gave you!

(a) **Correctness:** How did you keep track of each item's current best derivation and the total weight of that best derivation? How did you react if you found another derivation for that item?

(b) **Efficiency:** How did you ensure that (for any given grammar) the algorithm runs in $O(n^2)$ space and $O(n^3)$ time? For example, you'll have to explain how pushing each item to the agenda is made to run in $O(1)$ time (including the duplicate check), and why that is necessary.

Note: For purposes of this question, though not for question 3a, you may assume that you get lucky and never need to reprocess an item (see reading section B.2). But adding a precise discussion of reprocessing—how to analyze its runtime or how to avoid it or speed it up—can earn extra credit.

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

The autograder will test your program on new grammars and sentences that you haven't seen. You should therefore make sure it behaves correctly in all circumstances. See reading section D for advice on how to check your work.

4. 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[2] 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

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

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 actual sentences from the *Wall Street Journal*, with minor edits in order to change vocabulary that does not appear in the grammar.

If you try running

```
./parse.py 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 one or more speedup methods from the “parsing tricks” lecture. Some ideas are given in reading section E. Using the method of reading section E.1 plus one other is probably enough to meet the requirement of the homework: 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.

7 ***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 PDF submission, comment on any problems you see in the parses (as in question 1). Also describe 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.[3] *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 PDF submission.

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]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 in practice.

601.465/665 — Natural Language Processing
Reading for Homework 4: Parsing

Prof. Jason Eisner — Fall 2025

We don't have a required textbook for this course. Instead, handouts like this one are the main readings. This handout accompanies homework 4, which refers to it.

A Hints on Data Structures for Implementing Earley's Algorithm

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 slides. 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).

If you want to make your parser efficient, 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*. Remember that you can point to the same object from two different data structures.

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

A.1 Storing rules

Represent the rule $A \to W\ X\ Y\ Z$ as a list or tuple. Something like (A, W, X, Y, Z) or $(A, [W, X, Y, Z])$ or (W, X, Y, Z, A).

Represent the dotted rule $A \to W\ X\ .\ Y\ Z$ as a pair $(2, R)$, where 2 represents the position of the dot and R is the rule or a pointer to the rule.

(Another reasonable representation of this dotted rule is just something like (A, Y, Z), which drops the right-hand-side symbols before the dot. That is, you can throw W and X away after matching them; it's Y and Z that still have to be matched. As discussed in class, this keeps your parse chart a little smaller so it is more efficient.)

It's easiest to represent the terminals and nonterminals as strings. However, once your code is working, you could choose to make it more efficient by integerizing or interning those strings, as discussed in previous homeworks.

A.2 Storing the parse chart

Suppose you are processing the item

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

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 items whose dot has advanced up to position i in the input.) The parse chart is organized into columns specifically to facilitate this search.

Represent each column in the parse chart as some kind of extensible list or vector.

The column of the chart changes as you iterate over it. So you may not be able to use a standard iterator (depending on what language and library you're working with). You can just iterate with a while-loop and your own index.

A.3 Duplicate handling

Make sure you check for duplicates *whenever* you add an item to a column, no matter how that item got created.

The duplicate check 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 that your solution include a hash table for speed.

What does "duplicate" mean in practice? Duplicates are items 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 items 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.

A.4 Weights

The weight of any tree is the total weight of all its rules.

Since each rule's weight is $-\log_2 p(\mathit{rule} \mid \texttt{X})$, where `X` is the rule's left-hand-side nonterminal, it follows that the total weight of a tree with root `R` is $-\log_2 p(\mathit{tree} \mid \texttt{R})$.[1] (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.

All of this is just like the weighted CKY case, and you should be able to proceed analogously. The only difference is that in Earley's algorithm, the items in the parse chart are dotted rules rather than nonterminals. Think carefully about how to define weights for these items in the context of a dynamic programming algorithm. At the end of the day, all you care about is finding the weight of the *best* complete parse, but you have to build that up from the weights of other items—so define those other weights to help you in your task.

A.5 Backpointers: From recognition to parsing

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

If the item 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 . . .)

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

To figure out how to print the best parse as well, 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 item only has to store one *backpointer pair* along with a weight. The backpointer pair 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 item in the parse chart can point to its "progenitors" (i.e., the items that combined to produce it), which in turn point to their progenitors, and so on.

Some hints on storing and following backpointers for different kinds of items:

- Think first about the items that get added by an ATTACH step. Such an item should be equipped with a backpointer pair.
- When you think about items added by a SCAN step, remember that the SCAN may have applied to a dotted rule like

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

  since the thing after the dot was a 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 new dotted rule with a backpointer pair.
- It turns out that items added by a PREDICT step (such as $(3, A \rightarrow .\ B\ C\ D)$) don't actually need to point to anything. They don't have any substructure to remember, because they don't cover any words yet.[2]

Once you have completed Earley's algorithm, use a recursive `print_item` function to follow the backpointers and 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_item` on *any* item in the parse chart. You know what is the "right thing" for `print_item` to do on a complete item, such as PP → P NP .: print a parse tree for that PP. But this should be accomplished, in part, by recursively calling `print_item` on the two things that the item back-points to. One of these will be a dotted item. From this, you should be able to deduce what is the "right thing" for `print_item` to do on a dotted item.

[2]If you still find that surprising, let's do a thought experiment to understand the role of these items. 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 $(i, X \rightarrow .\ Y\ Z)$ for every rule $X \rightarrow Y\ Z$, 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 items 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 items 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.

B Avoiding Common Pitfalls

B.1 Common inefficiencies

In general, think about memory efficiency a bit. You'll need that to deal with big grammars and parse charts. Some of you may not have previously been in classes where your programs take hours to run. But `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.

It can be wasteful to store multiple separate copies of a rule or item. 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.

Use a few big hash tables (dictionaries), not lots of little ones. In particular, try to avoid arrays of hash tables, or hash tables of hash tables. Why? Each hash table has additional memory overhead, e.g., lots of empty cells for future items.

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.

B.2 Reprocessing

In class, we only covered the standard unweighted version of Earley's algorithm. When you improve the algorithm to use weights, there's a subtle bug you should avoid regarding duplicates.

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 this lower-weight version of Z, which will ATTACH it *again* to Y to get a lower-weight duplicate of X.

The easiest solution: When you find a lower-weight duplicate of Z, don't merely overwrite the old version of the item with the new, lower-weight version of the item. You also have to essentially move that revised item down to the unprocessed part of the column if it has already been processed, so that it can be re-processed. Make sure that this move takes only $O(1)$ time—there are a few ways to achieve that. I say "move" because the column shouldn't end up with two copies of Z—if it did, then you'd waste time later by ATTACHing stuff to both copies, whereas you really only need the lower-weight one.)

While this approach is pretty easy, the reprocessing means that the runtime is no longer guaranteed to be $O(n^3)$.[3] You're allowed to use this approach anyway, since the homework is already plenty hard.

Awesome extra credit challenge: Can you find a way to avoid reprocessing without introducing bugs? Discuss in your PDF submission. The goal is to avoid computing the weight of an item until you know the best way to build it and hence can get that weight right. I can think of an $O(n^3 \log n)$ solution, as well as a few $O(n^3)$ solutions. Some of the $O(n^3)$ solutions use more space, or fail to run in $O(n)$ time on grammars that should allow that—but there's at least one great $O(n^3)$ design.

[3] There is a lot of reprocessing in practice. If I turn reprocessing off, my parser will run 3–4 times faster on `wallstreet.sen` ... but it will only get the best parse for 3 of the 9 sentences, and will report incorrect probabilities for 2 of those 3!

B.3 Weird grammar rules

Remember, it is legal to have a rule like `NP` $\rightarrow$ `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` $\rightarrow$ `a majority . of N`. Another example is `EXPR` $\rightarrow$ `EXPR + TERM` in `arith.gr`, with terminal `+`.

For this homework, you can assume that the grammar contains no rules of the form $A \rightarrow \epsilon$. However, if you opt to handle such rules, be aware that they are a little trickier. The reason is that a complete item from 5 to 5 could be used to extend other items in column 5, some of which have not even been added to column 5 yet! For example, consider the case $A \rightarrow X\ Y$, $X \rightarrow \epsilon$, $Y \rightarrow X$.

Note that there might be multiple ROOT rules.

C Programming Language Advice

Python is the default choice and we've given you starter code in Python. Python has 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.

I happened to use LISP, where the full homework took about 150 lines or about 3 pages of code (plus a 1-line `parse` script to invoke LISP from the command line). This included the PREDICT and left-corner speedups from question 4.

If you use an interpreted language like Python, leave extra time. Leave plenty of time to run the program. My LISP program was reasonably fast (see reading section E.7) when compiled with the CLISP compiler, and a few times slower with the CMUCL compiler. Interpreted languages like Python will run slower yet.

Python hint: You could try PyPy, an alternative implementation of Python that does just-in-time bytcode compilation of its interpreter. They claim a 4× speedup on average.

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 homework in C++ without taking advantage of the data structures in the Standard Template Library.

`https://benchmarksgame-team.pages.debian.net/benchmarksgame/` compares the speed of compute-intensive benchmarks coded in different languages. The *k*-nucleotide benchmark may be roughly comparable to parsing.

D Checking your Parser

To help you check your parser, the homework will provide some simple grammars and sentences for you (as `.gr` and `.sen` files).

- Under `permissive.*`, every column of the parse chart should contain all (start position, dotted rule) items that are possible for that column. Column n will contain $O(n)$ items. A bigger version is `permissive2.*`.

- 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`.[4]
- For the first two sentences in `wallstreet.sen`, the lowest-weighted parses have weights of 34.22401 and 104.90923 respectively.
- You might also try `english.*`.[5]
- For most debugging, you'll want to use smaller grammars or shorter sentences where things run fast and you can trace the parser's behavior. 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 homework 1, except that that program prints probabilities instead of weights:

```
parse -P -g arith.gr < arith.sen
```

That is an interpreted Perl script using a modified version of CKY. You can also try the much faster `dynaparse` program given with homework 1 (available only on the `ugrad` or `grad` network). That one is Earley's algorithm, written as a very short Dyna program with no special speedups, and compiled into C++.

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 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 ... you can log debug output, or use an actual debugger.

E Speedups

Once you have a correct parser, here are some possibilities for speedups. You are not limited to these.

E.1 Batch duplicate check

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 $\rightarrow$. BLAH BLAH), then it should be a quick check to discover whether you've already added that batch to the current column.

Without this speedup, you would try to add all the rules in the batch, checking each *individually* (see question 3b) to discover whether it was already there. This takes constant time but it's a big constant.

[4] Assuming you use 64-bit double-precision floating-point arithmetic and print the same number of digits.

[5] We'll see `english.gr` again in the semantics homework. It was produced from that homework's `english.gra` using the `delattrs` script.

E.2 Vocabulary specialization

You could figure out which words are the terminals, and temporarily delete rules that use terminals that aren't in the sentence.

E.3 Pruning

A pruning strategy (or even better, a prioritization strategy) lets you ignore low-probability rules or low-probability items unless you turn out to really need them. We discussed several such strategies in class. These approaches are indispensable in the real world, where one wants to parse hundreds of sentences per minute. If you try an unsafe form of pruning or prioritization, try to examine the effect on parse accuracy.

E.4 More aggressive dynamic programming

Merge related items in the parse chart, using one of the following safe strategies discussed in class:

- Represent everything of the form $(3, X \to \cdots \ . \ C \ D \ E)$ as a single item in the chart.

  Among other things, this reduces the number of items that need to be ATTACHed, since now we just have the single item $(3, X \to \cdots \ . \ )$ to say that there is a complete constituent X starting at 3, rather than many items 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 $(3, X \to A \ B \ C \ D \ E \ . \ )$ 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 $(3, X \to A \ B \ . \ C \cdots)$ as a single item in the parse chart.[6]

  Once you advance the dot past the C, you will have to find all D such that the grammar allows the dotted rule $X \to A \ B \ C \ . \ D \cdots$. 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 $X \to A \ B \ . \ C \cdots$ to $X \to A \ B \ C \ . \ D \cdots$. 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. Handling weights is a bit tricky, especially if you want to minimize the automaton, but there are libraries for that. If you are using Python, we specifically recommend the Pynini weighted finite-state library—it's an interface to the OpenFST library that we'll be using later in the course. If you are using C++, you could call OpenFST directly.

[6] Then you'll no longer need the speedup of reading section E.1. Why not?

E.5 Indexing customers

You will often have to search column i for all items 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* item in column i. Can you design a better way of storing or indexing column i, so that you can quickly find *just* the items with X after the dot?

E.6 Left-corner filtering

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 items 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 ...)

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 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.

Example. 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

[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 homework (see reading section B.3).

[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.

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.

E.7 Discussion: How fast can you go?

To get through the 9 sentences in `wallstreet.sen`, my compiled LISP program with a couple of speedups (see reading section C) took a little over 2 minutes, either on the `ugrad` machines (in 2020) or on a fast laptop (purchased in 2020). The first sentence took a fraction of a second because it is short, but the algorithm is $O(n^3)$, so longer sentences take *much* longer. The two long sentences take most of the time.

However, the `parse` program from homework 1 (written in the oldest version of Dyna) is 10 or 20 times faster than my LISP parser, despite having none of the speedups (other than a faster solution to reprocessing as hinted in reading section B.2). In past years, some students have managed to write parsers that run (I think) about 50 times faster than my LISP parser, by using more optimizations.

And faster yet is possible. 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![9]
- 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
- doing most of the work in $O(n)$ time by "supertagging" each word with a syntactic tag
- "dependency parsing" formulations that only have to tag each word with a parent word that it modifies (this yields a tree structure without any nonterminals)
- "transition-based" strategies for greedily building up a good parse tree without dynamic programming

You are certainly welcome to use any of the first three techniques in this list, 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.

[9] For example, "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.