Massachusetts Institute of Technology
6.863J/9.611J, Natural Language Processing, Spring, 2002
Department of Electrical Engineering and Computer Science
Department of Brain and Cognitive Sciences

Laboratory 3: Phrase Structure Parsing

Handed out: March 10, 2003 **Due: March 19, 2003**

1 Introduction: goals of the laboratory

The major aim of this laboratory is to give you an (admittedly brief) experience in building "real" grammars, along with an understanding of how syntactic phenomena interact with context-free parsing algorithms. The laboratory is designed so that you will become familiar with the Earley algorithm/AKA chart context-free parsing. By the end of the lab, if you want to build a large grammar on your own, for any language, you'll know the basics about how to proceed (see the first footnote). In addition, you will learn how to use *features* to simplify grammar construction. Here are the main tasks for this laboratory:

1. We will have you do a very simple warm-up exercise, and answer two basic questions about the parser's operation.
2. We will give you a simple context-free "starter" grammar in the course locker. We ask you ask you to extend this grammar to handle a slightly wider variety of syntactic phenomena with different kinds of verb arguments and embedded sentences.
3. Next, you will extend your augmented grammar to handle simple questions, such as "Which detectives solved the case?"
4. Finally, using *features*, you'll write an augmented context-free grammar to cover the same examples, and compare its size to the grammar you have built that does not use features.

Similar to previous labs, your completed laboratory report should be written as a web page, sent to `ktkohl@ai.mit.edu` and contain at a minimum these items:

1. A description of how your system operates. How does your system handle each of the syntactic phenomena described below? Include a description of the rules that were used to encode the relevant changes to the grammars and how these were implemented (e.g. "I added the rule `VP...` In order to implement this rule, we had to ..."). If appropriate, mention problems that arose in the design of your grammar.
2. Your modifications to any initial grammar rule files that we supply to you.
3. A record of a parser run on the relevant sentences. (We supply them as part of the "starter" grammar, as described below — you won't have to type these in yourself.)
4. Your feature-based context-free grammar for the same sentences, as well as a comparison to your feature-free context-free grammar.
5. Bonus: if you are (very) ambitious, you might want to tackle a larger set of "challenge" sentences that are very much harder to successfully parse using a context-free grammar. You are welcome

to try your hand at these. This makes a very wonderful final project. It shows how truly hard it is to write a complete context-free grammar for English.[1]

Before proceeding to the laboratory questions themselves, we first describe the Earley/chart parser implementation in the course locker, and how to use it.

General Introduction to the Earley Parser Implementation

The Earley parser we have implemented runs in Common Lisp. It resides in the `earley` directory of the course locker `/mit/6.863/` (and linked on the course web page). The following is a brief description of how to load and start the system, load in grammar rules, list, and modify them, etc. For complete documentation of the system, you can, and should, go to the links on the course web page, which has `doc.pdf` and `doc.ps` files. These pdf and ps files are also in the `earley` directory in the course locker.

The 6.863 Earley parser will work with any well-defined context-free grammar (CFG), including recursive ones and those containing empty categories (rules that rewrite as the empty symbol). It can use either no lookahead or a one word lookahead, *i.e.* $k = 0$ or $k = 1$. One-word lookahead is typically a big 15% time win, so it is the default setting.

A feature system has been overlaid on the parser, allowing powerful grammars to be created more easily than with a simple CFG. We will use this feature system in the **final** part of this laboratory — NOT in the first portions of this lab.

Following this convention, when we use the term **CFG** we mean an ordinary context-free grammar without superimposed features. When we use the term **GPSG** we mean a *Generalized* Phrase structure grammar, namely, a context-free grammar with features imposed on top of it. We discuss this extension in the lab's final section.

In addition to the parser, the parser system has Lisp facilities for creating and editing CFG and GPSG grammars, dictionaries, and sentence sets.

2 Loading the System

All of the parser and utility code is in Common Lisp, and were all Common Lisps equal, the parser would be machine independent. The files are set to load into the package `USER`. The file `earley.lisp` contains all the necessary code. As it stands, the current system runs on Athena under the implementation of Common Lisp called `clisp`. As for other Common Lisps, your mileage may indeed vary. (For example, we have not tested this code on Allegro Common Lisp for PCs or for Macs, but in all likelihood with some tweaking on your part the code would work; but we cannot supply system assistance for this.)

2.1 Athena business to load the system

The Earley system stores various types of *rule sets*, such as CFG grammars, GPSG grammars, or sentence sets. These can be edited and used as input for the parser. The parser requires special grammar tables as input, which are compiled from featureless context-free grammars (GPSGs are automatically translated into featureless CFGs).

[1] If you want to know more, you can ask me for a copy of one of the "original" papers on this, by G. Gazdar (1981) "Unbounded dependencies and coordinate structure," *Linguistic Inquiry* **12**, 155-184. Reprinted in Walter J. Savitch, Emmon Bach, William Marsh and Gila Safran-Naveh, eds. **The Formal Complexity of Natural Language** Dordrecht: Reidel, 183-226 (1987). In past years, the challenge was to build grammar rules to parse *all* the 100-plus example sentences in this paper. This sounds easy, but it is very difficult — it includes such sentences as, "Mary shot, and John killed, a rabid dog"; and "I know John has been, and Mary was previously, a rabid Red Sox fan". The point about this is simply that the Gazdar paper claimed efficient parsability on the grounds that the formalism described in the paper was context-free; but it is apparent when one tries to implement the rules that a real grammar for the example sentences probably was never even attempted.

In order to access Common Lisp and run the system to parse sentences, you need to follow these four steps:

1. Attach 6.863 to get the course locker available and change directories to where the earley parser is located in our course locker;
2. Fire up Clisp and then load the parser and grammar rule files;
3. "Compile" the test context-free grammar(s)
4. Select a grammar and dictionary to use to parse your sentences

You will then be ready to parse sentences — your first objective will be to parse and trace them, so as to see how the chart parser operates.

Let's now go through these steps in more detail.

Step 1. You attach the course locker direcctory & cd to the `earley` directory:

```
athena> add 6.863
athena> cd /mit/6.863/earley
athena>
```

Step 2. We have provided a simple `clisp` shell startup script in the course locker `earley` directory, that will fire up clisp. If you execute this as follows, it will bring up the lisp interface; then you can load the parser code, as follows:

```
athena>./clisp

 i i i i i i i       ooooo    o        ooooooo   ooooo   ooooo
  I I I I I I I      8     8   8           8     8     o  8    8
  I I I I I I I      8         8           8     8        8    8
  I I I I I I I      8         8           8      ooooo   8oooo
  I  \ `+' /  I      8         8           8           8  8
   \  `-+-'  /       8     o   8           8     o     8  8
    `-__|__-'         ooooo    8ooooooo  ooo8ooo   ooooo   8
        |
  ------+------     Copyright (c) Bruno Haible, Michael Stoll 1992, 1993


> (load "earley")
;; Loading file /afs/athena.mit.edu/course/6/6.863/earley/earley.lisp ...
;; Loading of file /afs/athena.mit.edu/course/6/6.863/earley/earley.lisp is finished.
T
>
```

Step 3. You are now ready to load the file `simple-grammars.lisp`, which contains your starter grammar, with rules and dictionary to try out the parser. In the trace below, you can see the system building a compact production table for the final parser, at the very end:

```
> (load "simple-grammars")
;; Loading file afs/athena.mit.edu/course/6/6.863/earley/simple-grammars.lisp ...
Rule set SENT1 added.
Adding rule (THE MAN FROM TIMBUCTOU SHOT THE GIRL FROM ISTANBUL) to rule set SENT1.
```

```
...
Round 1. 7 added.
Round 2. 1 added.
Adding rule (#:|production-table450| (ROOT S VP PP NP *D0*) #((ROOT $)
(ADJ) (DET N) (NP PP) (NP*) (P NP) (P PP) (V) (V NP) (VP PP) (NP VP) (S PP) (S))
 #(*D0* NP NP NP NP PP PP VP VP VP S S ROOT) #((NP* DET ADJ) (NP* DET ADJ)
  (NP* DET ADJ) (NP* DET ADJ) (NP* DET ADJ) (P) (P) (V) (V) (V) (NP* DET ADJ)
  (NP* DET ADJ) (NP* DET ADJ))) to rule set LAB3G.
Done CFG Parse.
;; Loading of file /afs/athena.mit.edu/course/6/6.863/earley/simple-grammars.lisp is finished.
T
```

To look at the rule sets stored via the loaded file, evaluate `(list-all)`. This prints out the name, type, and number of rules in each rule set. There are 6 different types of rule sets: CFG, GPSG, CFG-TABLE, GPSG-TABLE, SENTENCES, and DICTIONARY. The -TABLE types are used as input for the parser. They cannot be edited, but all other types of rule sets can.

```
athena> (list-all)


1.   LAB3G          CFG-TABLE          1
2.   SENT2-GPSG     SENTENCES          5
3.   DICT2          DICTIONARY         14
4.   GPSG1          GPSG               5
5.   DICT1          DICTIONARY         66
6.   LAB3GR         CFG                13
7.   SENT1          SENTENCES          18
```

At this point, you will now have 3 separate sets of grammars, sentences, and dictionaries to try. For this laboratory, we will only be interested in grammar `LAB3GR`, the simple starter grammar for this laboratory, as well as `DICT1`, which contains enough lexical entries to do most of the work for the lab (but you will have to make some modifications here too). Item 4 corresponds to a feature-based context-free grammar, that we shall turn to in the last part of the lab.

The grammar rule format is straightforward, and follows what you have seen in class or in textbooks. Here is the actual file as extracted from the `simple-grammars` file (the only item requiring comment below is the `NP*` element, which indicates that `NP*` can be a *terminal* that appears directly as a category in the lexicon (as a proper name, for example, ISTANBUL or POIROT). Note the use of the *type* keyword for the grammar, which declares it to be an ordinary context-free grammar, without features: CFG.

```
;;; GRAMMAR DATA FILE
; Name: LAB3G  Type: CFG
(add-rule-set 'LAB3G 'CFG)
(add-rule-list 'LAB3G
       '((ROOT ==> S)
 (S ==> NP VP)
 (S ==> NP AUXV VP)
 (S ==> AUXV NP VP)
 (VP ==> VP PP)
 (VP ==> V NP)
```

```
 (VP ==> V)
 (PP ==> P PP)
 (PP ==> P NP)
 (NP ==> NP*)
 (NP ==> NP PP)
 (NP ==> DET N)
 (NP ==> ADJ)
 ))
```

Step 3. Next, in order to *use* a particular grammar you must "compile" it into a more efficient table format. In the `simple-grammars` file we have actually done this for you and put in a line to compile the starter grammar you want to use for lab 3 — you can see this in the last few lines of the loading transcript above. However, we give the instructions below because you will need to know how to do this for the last part of this lab, or if you want to experiment with other grammars.

You have to recompile this grammar table after every addition/emendation to the rules that you make, but compilation is fast, don't worry. (It is often easier just to reload the grammar file all over again.) To create the parsing table, you call the function `create-cfg-table`, and give your table any (nonconflicting) name you want. Note that `*DO*` is automatically added as the "Start" symbol for you. In this function call, we have specified 1 word of lookahead and supplied our own "Root" symbol, for parse tree printing later on:

```
> (create-cfg-table 'LAB3G 'LAB3GR 'root 1)
Round 1. 7 added.
Round 2. 1 added.

Adding rule (#:|production-table450| (ROOT S VP PP NP *DO*) #((ROOT $)
(ADJ) (DET N) (NP PP) (NP*) (P NP) (P PP) (V) (V NP) (VP PP) (NP VP)
(S PP) (S)) #(*DO* NP NP NP NP PP PP VP VP VP S S ROOT) #((NP* DET
ADJ) (NP* DET ADJ) (NP* DET ADJ) (NP* DET ADJ) (NP* DET ADJ) (P) (P)
(V) (V) (V) (NP* DET ADJ) (NP* DET ADJ) (NP* DET ADJ))) to rule set
LAB3G.
NIL
```

Step 4 Finally we are ready to parse! The first time you parse a sentence, via the function `p`, you must supply arguments for the grammar table and the dictionary you want to use. (This lets you switch grammars and dictionaries.) These values are 'sticky' so you don't have to supply them again unless you want to switch grammars and/or dictionaries. Once again we've already done this for you in the `simple-grammars` file, where the grammar name and dictionary name have already been set to `LAB3GR` and `DICT1` (Note that you supply the name of the *grammar table*, NOT the grammar name. Note also that upper and lower case is ignored in Clisp. So — if you want to use another grammar and dictionary than this "default" one, you'll have to specify it as an argument to `p` the first time around.)

```
> (p '(Poirot solved the case))
Done CFG Parse.
Done Parse Retrieval
((ROOT (S (NP ((NP*) POIROT)) (VP ((V) SOLVED) (NP ((DET) THE) ((N) CASE)))))
```

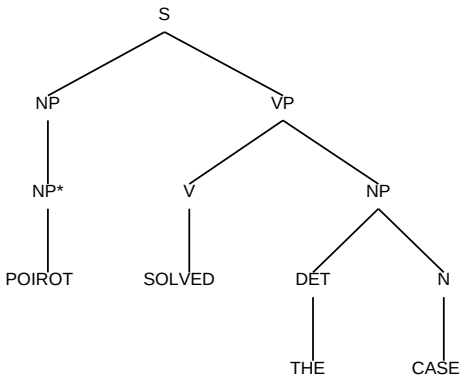
Parses are returned as lists of possible tree structures, where each tree structure has the form (1(Category SubTree1 SubTree2 ...). Not very pretty. You can do even better by saving the output and running it rhough the script `runvt` in the course locker on this output. This script will take any set of parses output, one per line, and then run them into an X-windows tree-viewer:

```
athena> add tcl
athena> /mit/6.863/runvt <file>   (where <file> is a file of the
                                   parser output)

alternatively:

athena>add tcl
athena> wish viewt <filename>
```

This will pop up an X window with a display of each parse tree in the file, one at a time. You can then scroll through the tree displays, going forward via a click on the right mouse button, and back through your set of trees with the left mouse button. If you click click `p`, you can save a postscript version of the tree you are currently viewing (`q` quits the program and kills the window). So, for the example above we can get this much nicer picture:



Poirot solved the case

(*Poirot* is the name of a famous literary detective. You'll see his name again.)

3 Problem 1: Warmup

OK, so much for preamble. As a warmup, you will run the parser and see if you understand how the algorithm handles ambiguity. To do this, please use the supplied `simple-grammar` and parse the following sentence, using the parse command as shown. This will print out the list of states the system enters as it constructs each State Set, as each word is processed. This will let you see the action in detail and is very helpful if something goes awry with your new grammars. We'll explain the tracing output just below.

```
Poirot sent the solution to the police

athena> (p '(Poirot sent the solution to the police) :print-states t)
```

The beginning of the tracing output looks like the following. At the very right is the "dotted rule" that you should be familiar with from the lectures and notes. The first two numbers in each row give the progress point so far for the edge labeled by that dotted rule, followed by the the number of the State Set where the parser first started constructing that rule (i.e., the start point of the phrase). The first number is also the State Set number. For instance, in State Set 0, at the very start, the parser adds the dotted rule `ROOT ==> .S`, starting at 0, and so far ending at 0. (Note the first few dotted rules have to do with special start-up rules ROOT and DO* — you don't need to pay attention to these.) The second State Set begins with a "1" in the very left-hand side, the dotted rule `NP ==> NP* .` , showing that "Poirot" has been recognized as a complete noun phrase (a name). So, the parser jumps over the NP, and this completes the NP, letting the parser advance the dot in the S expansion rule too. And so on.

```
_ 0  0   *DO* ==> . ROOT $
_ 0  0   ROOT ==> . S
_ 0  0   S ==> . NP VP
_ 0  0   S ==> . NP AUXV VP
_ 0  0   S ==> . AUXV NP VP
_ 0  0   NP ==> . NP*
_ 0  0   NP ==> . NP PP
_ 0  0   NP ==> . DET N
_ 0  0   NP ==> . ADJ
_ 1  0   NP ==> NP* .
_ 1  0   NP ==> NP . PP
_ 1  0   S ==> NP . AUXV VP
_ 1  0   S ==> NP . VP
_ 1  1   VP ==> . VP PP
_ 1  1   VP ==> . V NP
_ 1  1   VP ==> . V
_ 2  1   VP ==> V . NP
_ 2  1   VP ==> V .
_ 2  2   NP ==> . NP*
_ 2  2   NP ==> . NP PP
_ 2  2   NP ==> . DET N
_ 2  2   NP ==> . ADJ
...
etc.
```

Now to the questions about this parse.

Question 1: Simple to start – How many distinct parses are produced? Please supply pictures for the corresponding parse tree(s) and describe what the differences in meaning are between them, if any.

Question 2: Using the output from `:print-states`, please explain how the dotted rules in the state sets are used to handle the possible ambiguity of this sentence, by referring to the specific dotted rules in the specific states in the trace sequence that are actually involved in the parse(s).

Hand in: Your answers to the two questions, as described above.

4 Problem 2: Writing grammar rules

In the next 2 subsections, we would like you to augment the starter grammar you are given, producing new grammar/dictionary/sentence sets that address new kinds of sentences. Each new grammar you construct should build on the previous one: that is, each new grammar should be able to handle all the

example sentences of the previous ones, plus the new phenomena that are introduced at each step. The sentences you must be able to parse (or reject, if ill-formed) are already collected for you at the very beginning of the `simple-grammar` file.

For additional useful information on how to manipulate sentence, dictionary, and grammar sets with this system, including how to individually edit grammar rules, dictionary entries, and select parses, please see the Appendix to this document, or look at the more complete file `doc.pdf` on the course website and in the course locker under `earley`.

If you are unfamiliar with the grammatical terms used, you should skim through the English grammar slides also provided online. For additional help, please ask!

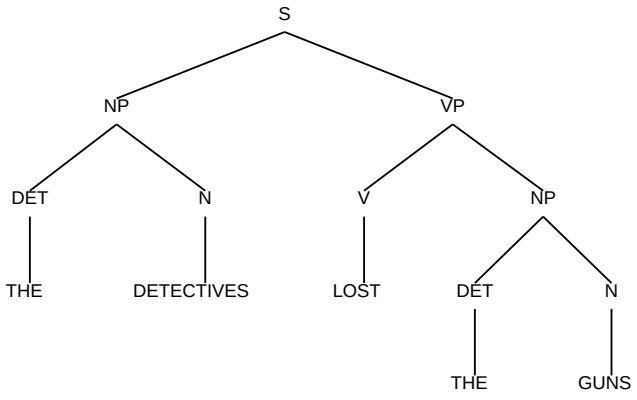
4.1 Grammar 1: Rules for declarative sentences

The grammar that you already have can parse these sentences (and more), which we call "simple declarative sentences."

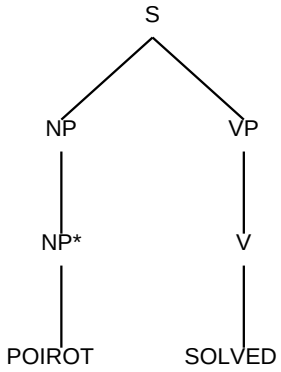
1. Poirot solved the case
2. The detectives lost the guns
3. Poirot thought
4. Poirot sent the solution to the police

You've already seen the stucture the parser produces for the first sentence in the list above. For the second and third sentences, here are pictures of the resulting structures. For the fourth sentence, we also display an output parse structure. When we ask you below to extend the parser to produce *correct* parses, we mean that it should produce structures like these corresponding to the structures we give for the additional sentences below. The parser should not produce 'extraneous' parses.

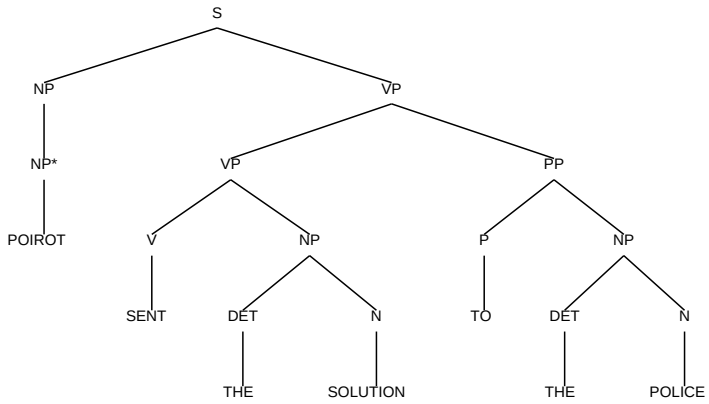
Here are the parse structure pictures mentioned above.



The detectives lost the guns



Poirot thought



Poirot sent the solution to the police

It is also **very important** that your grammar does not *overgenerate* in the sense of being able to parse "ungrammatical" sentences or assign parses (structures) that are incorrect to otherwise correct sentences. After all, if overgeneration were allowed, then one could simply write very very general rules like `S==> w*`, that is, any string of words at all could form a sentence. This would "parse" any string of words, but the results would not be very informative.

For this reason, your grammar should not allow examples such as the following, where we have marked these (artificially) with an asterisk in front. The asterisk is *not* part of the input but simply indicates that the sentence is to be rejected, yielding no parses with respect to your grammar. These are all violations of the subcategorization (argument requirements) for their respective verbs.

1. * Poirot solved (invalid because *solve* requires an object as an argument)
2. * Poirot thought the gun (invalid because *think* requires a certain kind of object, namely, a full proposition or sentence)

3. * Poirot sent to the police (invalid because *send* requires two objects)[2]

On the other hand, your revised grammar must still be able to parse sentences like *Poirot sent the solution* as before (though *something* will have to be different about the final parse trees — they cannot be exactly the same as before.)

More generally, the initial starter grammar does not correctly distinguish between verbs that take zero, one, or two arguments (some perhaps optional). You first job, then, is to augment the grammar by adding *verb subcategories* so that sentences such as *Poirot solved* will be rejected by the parser (no output), while *Poirot thought* will still be OK. You will also have to add/modify the dictionary entries. (This will entail reloading the modified grammar file. If you include a 'compile' procedure call at the end, as in the simple-grammar file, then you'll be all set to test the new grammar and dictionary.)
Hand in: Your modified grammar rules and dictionary, and output from the parser illustrating that it rejects the three sentences above while still correctly parsing the sentences "Poirot solved the case"; "Poirot sent the solutions to the police"; "The detectives lost the guns"; and "Poirot thought." (Pictures would be nice, though not necessary.)

4.2 Grammar 2: Verbs that take full sentences (propositions) as arguments

Still other verbs demand full propositions — sentences — as arguments:

1. Poirot believed the detectives were incompetent

2. Poirot believed that the detectives were incompetent

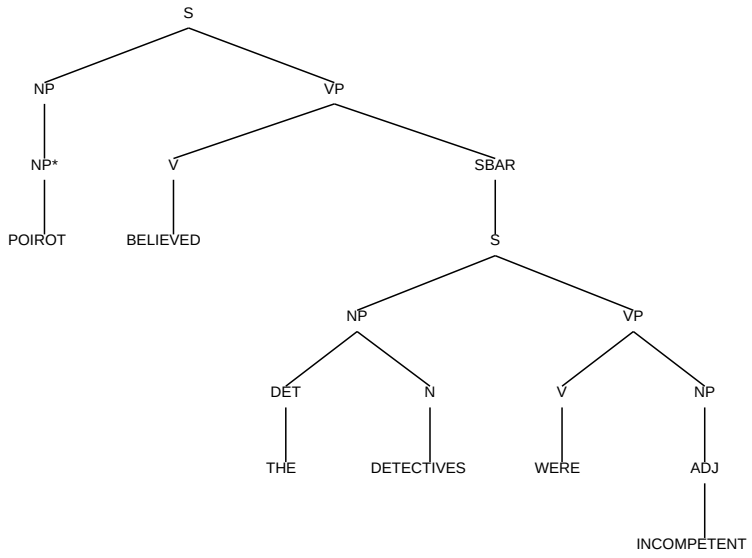
Here, the verb *believe* takes as an argument a full Sentence: *the detectives were incompetent*. As the second example immediately above shows, a *that* is optional in English (though not, for instance, in Spanish). Handling these new sentences will require further elaboration of the subcategories of verbs, as you may appreciate.

To do this, we shall specify a special structure for such sentences that we want you to follow, s shown in the figure that comes on the next page. In order to accomodate the possibility of *that* and for other linguistic reasons, the Sentence phrase itself should is elaborated so that the actual combination of *that* and S forms a phrase called *S-bar*, labeled in linguistics texts usually as $\overline{S}$. The component before the S is called its *complementizer*, abbreviated *COMP*, and is filled by words such as *that*, *who which*, etc. in English. (This position is also present but usually empty in a simple declarative sentence.)

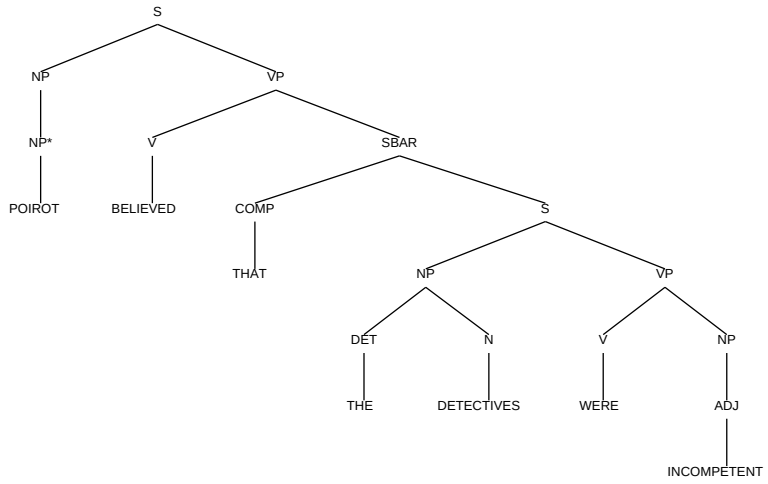
Here are the parse structures corresponding to the two *believe* sentences. In fact, one can immediately see just by looking at these tree structures what the corresponding new context-free rules should be. That's your next job: add to your grammar the grammar rules and dictionary entries so that the parser will parse and output the tree structures like the ones we show (they will not have exactly the same lexical categories for verbs, however):[3]

[2]Yes: there *is* a reading of this sentence which is somewhat acceptable — similar to "Poirot sent for the police — but please put that one aside.

[3]Yes: this suggests that one might be able to *automatically* compile context-free rules simply by looking at lots of example parses — this is how the parsed "gold standard" data is actually used.



Poirot believed the detectives were incompetent



Poirot believed that the detectives were incompetent

Hand in: Your augmented grammar that can correctly parse these two sentences, as we have illustrated, along with supporting output and figures. Note: you will gain bonus points if you can figure out how to produce a strictly accurate structure for the first of the *believe* sentences, where there is in fact no *that* present, but the COMP node in the corresponding parse tree should really still be present — i.e., the lexical material or word below the COMP node in the tree is the empty string.

4.3 Grammar 3: Unbounded Dependencies and Questions

As a final example of grammar augmentation, you will now add rules to handle certain kinds of *question* sentences.

This will involve implementing a so-called *slashed category* notation for nonterminals. Here, we augment a context-free grammar with new "slash rule" nonterminals and "slash rule" expansions, as we explain in more detail below. Again, you are to use just the simple CFG option of the Earley parser system.

Listed below are examples from each of the construction types your new grammar should handle, as well as examples that it should reject. The sentences that should be rejected are marked with asterisks in front. Of course, as before these asterisks are *not* part of the actual input sentence!

Question Formation

1. Which case did Poirot solve
2. Who solved the case
3. Which solution did Poirot send to the police
4. Which solution did Poirot believe that the detectives sent to the police
5. * Which solution did Poirot send the gun to the police
6. * Who solved

In these examples, we will again explain the structures we want you to output with your parser, which will assist you in adding context-free grammar rules to be able to parse them.

In sentences like these, we find *wh*-Noun Phrases, such as *which case*, *who*, *which solution*, etc., that are Noun Phrases "displaced" from their canonical syntactic position (usually the object of the main verb) — but with the added "feature" of being marked $+wh$. So for example, in *Which case did Poirot solve?* the object of *solve* does not appear directly after the verb as it would normally, but is in fact the phrase *which case*. Note that there is no pronounced lexical material following solve — just an empty spot in the string, which is "invisible" — not pronounced. Hence, your grammar must contain the following context-free rule, where X represents a nonterminal name that you will have to provide. By placing *nothing* on the right-hand side of the rule, it is expanded as the empty string.

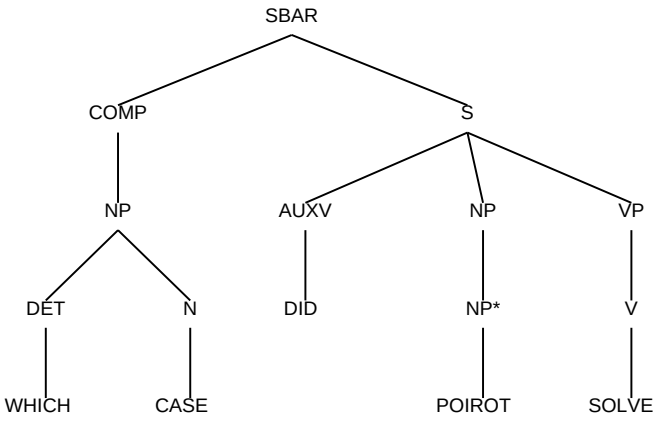
```
X ==>
```

We stress that this is NOT the exact rule that will work — it must be a rule that somehow "links up" with the *wh*-Noun Phrase at the front. Further, the displaced *wh*-Noun Phrase itself can appear only in certain positions: it must be in the *Complementizer* (COMP) position of the top-most SBAR phrase for the sentence. So, one must also alter the basic form for sentences: as mentioned in the previous section. We really want *all* sentences to begin uniformly, by expanding the ROOT node as an SBAR, and SBAR as a COMP followed by S. So, the following rules (approximately) will also have to be in your revised grammar.

```
Root ==> Sbar
Sbar==> Comp S
```

We say "approximately" because, obviously, we do not want to admit a sentence where COMP is expanded as *that* at the very beginning of a top-level sentence — then it would be a sentence fragment, as in "That Poirot solved the case." We only want Noun Phrases that dominate a *wh* word like *which* or *what* to be allowed here. One solution is to call these words like —it which or *what wh-determiners* and combine these with nouns as before to get Noun Phrases that are wh-Noun Phrases. To do this, we can introduce a new nonterminal name, call it `NP+wh`. Beyond this, to assemble the wh-Noun Phrase correctly is a design problem that you will have to solve.

You will also have to make sure the grammar you have been given can already can handle examples such as *Did Poirot solve the case* before proceeding to add rules to deal with such wh-questions — check that it does. Again: Please do NOTE that for pedagogical reasons we have *not* given all the details of the correct parse tree here, in particular, the details of the nonterminal names. Otherwise, the solution could be simply "read off" the parse tree.



Which case did Poirot solve

For the next wh-type question sentence, you will have to make a design decision as to the parse tree for *Who solved the case* — in such sentences, *Who* is the Subject of *solve*. The question is whether it is in its ordinary Subject position (as in, "Poirot solved the case"), or whether it too has been displaced from its canonical position and moved into a position under a COMP node headed by SBAR. You ought to reflect on why there is a choice here at all. Please include in your lab report a few sentences on this issue.

Finally, you will have to add rules to handle the two sentences beginning "Which solution...".

Finally, as before, you must make sure your grammar does not overgenerate, and accept, incorrectly, the sentences:

1. * Which solution did Poirot send the gun to the police
2. * Who solved

As before, the asterisk in front of each sentence is not in the actual input stream — it is there for reference purposes, to denote an unacceptable sentence.

Hand in: The grammar rules that can properly parse the correct wh-question forms above, as well as reject the incorrect forms above. Please describe the rationale for your design decisions, in particular the choices you have made to deal with *who*.

5 Problem 3: Using feature grammars

For the last part of this lab, we turn to an extension of context-free grammars: feature-based context-free grammars. In these grammars, rather than atomic nonterminal (phrase) names like NP or NP+wh, the phrase names may be combinations of (feature value) pairs. What we want you to do is to re-implement your grammar rules from the previous section, but use (feature value) pairs to describe the features on basic phrase names like NP. Then, considering the full range of wh-type questions and other sentence of this type (which we shall describe below), we would like you to discuss the difference in grammar size between an unaugmented context-free grammar that can parse such sentences, vs. a feature-based context-free grammar.

To use feature-based context-free grammars in the implemented parser, do this, you **must** name your grammar type "GPSG" rather than "CFG", and modify the lexicon and grammar rules to follow the following semantics.

In a GPSG rule, each category can either be a simple category symbol or a list of a category symbol and feature-value pairs. A feature-value pair is a combination of a symbol that names a feature and a value for that feature. Possible GPSG categories are `NP`, `(NP)`, `(NP PLURAL +)` and `(NP PLURAL ?X / VP)`. In these categories the symbols `plural` and `/` are feature names and `+`, `?X` and `VP` are feature values. Features found in the left hand side of a GPSG rule are given to the phrase produced by the rule. Features on the right hand side refine which daughters will match the categories- daughter phrases must contain the features specified. So the rule

```
((NP PLURAL +) ==> (DET SING -) ADJ (N PLURAL +))
```

will fire if its first daughter has the value `-` for its `SING` feature and the third daughter has the value `+` for its `PLURAL` feature. Naturally the categories `DET`, `ADJ` and `N` must also match. Notice that the rule does not imply that the second daughter can not have features; it does not matter what extra features the daughters have so long as the specified ones match. The rule will produce a `NP` phrase that has the value `+` for its `PLURAL` feature, and no other features.

5.1 GPSG Rules and Feature Passing

GPSG rules are preprocessed before parsing. Their features are stripped to created a featureless CFG that is used by the parser in the first stage of parsing. Parse trees that are recovered by the initial parse are augmented with features in a second pass that utilizes the original featured rules.

Features are used to refine rules. If a rule has in its right hand side a category with a certain feature specification, then whatever subphrase matches that category must have the same feature specification. This can be used to ensure, for instance, that a rule matches only plural NPs and not just NPs in general without having to create a seperate category for plural NPs. Feature values can be passed upwards, so that a phrase inherits feature values from its daughters.

With the feature system it is possible to create ridiculously baroque devices. As much care as possible should be taken to avoid unchecked feature generation; unfortunately the simplicity of the context-free-grammar formalism sometimes necessitates this.

In a GPSG rule, each category can either be a simple category symbol or a list of a category symbol and feature-value pairs. A feature-value pair is a combination of a symbol that names a feature and a value for that feature. Possible GPSG categories are `NP`, `(NP)`, `(NP PLURAL +)` . In these categories the symbol `plural` is a feature name and `+` is a feature values Features found in the left hand side of a GPSG rule are given to the phrase produced by the rule. Features on the right hand side refine which daughters will match the categories- daughter phrases must contain the features specified. So the rule

`((NP PLURAL +) ==> (DET SING -) ADJ (N PLURAL +))`

will fire if its first daughter has the value `-` for its `SING` feature and the third daughter has the value `+` for its `PLURAL` feature. Naturally the categories `DET`, `ADJ` and `N` must also match. Notice that the rule

does not imply that the second daughter can not have features; it does not matter what extra features the daughters have so long as the specified ones match. The rule will produce a `NP` phrase that has the value `+` for its `PLURAL` feature, and no other features.

5.2 Feature Matching Semantics

Two values are considered equal for the purposes of matching if they are `eq` or if one of them is the `car` of the other, or if one of them is `*`, or if they are both lists and each subelement matches.

If a category specification specifies a value for a certain feature and the phrase it is matching does not contain a value for that feature (or the value is `nil`), then the specification matches so long as the feature name does not start with `+`. So the specification `(NP PLURAL -)` will match `(NP)` and `(NP PLURAL -)`, but `(NP +PLURAL -)` will only match `(NP +PLURAL -)`, not `(NP)`.

5.3 Unification

If a specification contains a symbol starting with `?` in a feature value slot, then that symbol will match any value in that slot. Furthermore, if the same symbol is found in another part of the same or a different specification in the same rule, it must match the same value it matched before. The symbol can be used in the left hand side of a rule to define a feature value. The rule

```
((NP CLR ?X) ==> (DET PLURAL ?A) (ADJ COLOR ?X) (N PLURAL ?A))
```

specifies that the value for `PLURAL` must be the same in the first and third daughters, and that the value for the `COLOR` feature in the second daughter will be given to the resulting phrase in the `CLR` feature.

If a feature is not present in a daughter phrase, then unification does not take place on its value. So if the `N` daughter does not contain a value for the `PLURAL` feature (or it is `nil`), then `?A` is left free to vary in the `DET` phraseq. And if `COLOR` is not specified for `ADJ`, then the resulting `NP` will have no feature specification for `CLR`.

To see what this can do, you should experiment by loading the grammar `GPSG1`. This feature grammar implements simple checking of NUMBER between subjects and Verbs, as well as between DETerminers and Nouns (a plural determiner like "these" must have a plural noun like "books"). It also checks whether Noun Phrases have the proper CASE. For example, the object of a preposition for "he" must marked in fact as "him": we say "talked to him," not "talked to he." Similarly, the subject of a sentence must have what is called NOMinative case: we say "I talk" not "me talk".

Please look at the section for `GPSG1` and `DICT2` in the `simple-grammars` file so that you can understand the format for writing such rules. You will note that the lexical entries for words like `man` and `men` are now in feature-value format. For instance, "woman" has the feature of being a N(oun), and also has a − feature for PL(ural) — i.e., it is not plural. In contrast, "woman" is marked +PL(ural). The format is `(WOMAN (N PL -))` vs. `WOMAN (N PL +))`.

To parse with this grammar and dictionary, simply compile the GSPG1 grammar as a parsing table, giving it a different name than previous tables you've made:

```
(create-gpsg-table 'gnew 'gpsg1 's 1)

Rule set GNEW added
Original rule set length: 5
With new slash rules added: 9
After slash instantiation: 5
Rules filtered down to: 5
Round 1. 4 added.
Round 2. 1 added.
Round 3. 1 added.
```

```
Adding rule.....

NIL

(p '(the men talk to me) :grammar 'gnew :dictionary 'dict2)
Done CFG parse.
Done parse Retrieval.
(((S) ((NP PL +) ((DET THE) ((N PL +) MEN)) ((VP PL +) ((V PL +) TALK)
((PP)) ((P) TO) ((NP CASE ACC) ((N CASE ACC) ME))))))
```

Try parsing the sentences *The man talks to sheep* vs. *the man talk to sheep* or *the man talk to me* using the dictionary `DICT2`.

Debugging rules with features is more complex than with vanilla context-free grammars. The `print-states` option often doesn't provide enough information for debugging GPSG parses. After a parse has been completed the function `states` can be used to get more detailed information about what occurred.

```
(states start end \&key matching print-parses print-rules)
```

`states` takes two arguments which specify the start point and end point of states to be printed. In other words, it only prints information about completed states that build phrases between the starting and ending points. If `:print-rules` is specified to be true, then along with each state a corresponding GPSG rule is printed. If `:print-parses` is true, then all of the phrases that were created are printed for each state. The `:matching` argument is a filter to apply to these phrases. Each of the states is printed with an asterisk preceding it if no phrases were produced during the GPSG unification process.

`states` prints an asterisk in front of a state if that state failed to produce any phrases during the GPSG feature-passing portion of a GPSG parse. Unfortunately `states` can not provide any information on why a rule might have failed. Using the `:print-parses` option lets you see the phrases that result from successful rule applications, but not failed ones. This information is available during the parse by parsing with `verbose` on. For instance, we can find out why a starred state failed by turning this switch on and then watching the unification process in action — we can see a mis-match in features that should not have failed.

```
>  (verbose :set t)
T
>  (p '(who saw mary))
Done CFG Parse.
Rule Failure:  (SBAR) ==> (S FIN +  NUM +)
Sub Trees:  (S FIN +  NUM -  )
```

In this deliberately artificial example, the SBAR rule requires an S that has a NUM feature that is marked +, but the feature value passed up from below is minus, as the subtree shows.

OK, enough about the GSPG parsing system itself. Your job in this section is to use features of your own choice to accomplish what you did in the previous section via nonterminal names. Take the wh-questions from that section and re-implement them using a GSPG.

Next, consider that in fact there are many more displaced phrases than just simple wh-Noun Phrases. We can move objects to the front of a sentence; we can move Prepositional Phrases, and much more:

> Beans, I never liked ("Beans" is moved from object position: "I never liked beans")

To what he said, I gave no response

The guy that I like went to the store (the object of "that I like" is "the guy")

John cooked, and Mary ate, all kinds of leftovers (the object of "cooked" is "leftovers")

And so on — this is a very very widespread phenomenon in language. In addition, we would have to have agreement between subjects and verbs, and determiners and nouns (in English); case agreement (in many languages), and so on.

Pondering all this, try to estimate the number of context-free rules one would need to handle a system that checks 5 such independent features (Number, Person, Case, Displaced phrases of type 1,...) vs. a feature-augmented context-free grammar (i.e., a GPSG). We are not looking for an exact number, but rather a rough ratio.

Hand in: A listing and description of your new grammar rules, using features and, most importantly, a comparison of the relative grammar sizes of your *original* grammar that did not use features and your grammar that does. Finally, a generalization of your result to the broader situation, as described in the previous paragraph, and the design implications for writing parsers for English (or any other natural language

(This is the end of the laboratory)

6 Appendix A: A more detailed description of the parser system options

The exact incantation for parsing is:

```
(p sentence &key (template '*) grammar dictionary (use-lookahead t)
                 print-states file sentence-set)
```

where `sentence` can either be a list or a string; `:template` specifies which parses are to be accepted; `:grammar` and `:dictionary` specify the grammar and dictionary to use; `:use-lookahead` specifies whether or not to use lookahead if it is available; `:print-states` specifies whether or not to print out all the Earley states at the end of the parse; `:file`, if given, is the name of a file to send all output to; and `:sentence-set` specifies a set of sentences to parse. After a parse has been completed the function `retrieve-parses` with the format

```
(retrieve-parses &optional (template '*))
```

can be used to retrieve the results again. More parsing functions, including ones to parse sets of sentences at one time, are given in the function listing in the file `doc`.

You will probably find that the `:print-states` function is quite useful in debugging where your rules have gone wrong.

6.1 Editing Rule Sets

For this laboratory, of course, you will find that you have to edit rule sets and dictionaries to handle new phenomena — that is the whole job of building a natural language grammar. The Earley parser system has some tools for this.

We can list and edit and rules in a rule set, whether a grammar or a dictionary. If there is a rule set `SD`, a DICTIONARY, it can be listed by evaluating `(list-rules 'SD)`. This will print out all of the rules (dictionary entries) in the set. A template may be specified, so all verbs in `SD` could be listed

with `(list-rules 'SD '(* V))`. Likewise, all rules with verb phrases as heads in the CFG `GRAMMAR1` could be listed with `(list-rules 'GRAMMAR1 '(VP **))`.

List-rules does more than just print out rules: it also sets up rule sets for quick editing. Let's say we evaluated `(list-rules 'SD '(* A))` to list out all the adjectives in the dictionary `SD`. Perhaps the result was

```
>  (list-rules 'SD '(* A))
  1.    HARD A
  2.    QUICK A
  3.    RED A
  4.    SLIMY A
  5.    SLOW A
```

We can now use the three functions `ADD`, `DEL`, and `RPL` to edit the dictionary `SD`. To delete "hard" and "slimy" we evaluate `(del 1 4)`. Now, we decide to add "lazy" in their place, so we do `(add '(LAZY A))`, and finally we decide to replace "quick" with "fast" and "slow" with an adverb, "slowly" – `(rpl 2 '(FAST A) 5 '(SLOWLY ADV))`. These three functions work on the last rule set listed with `list-rules`. Thus they permit quick and easy editing of grammars, dictionaries, and test sentences.

If you want to do more extensive modifications offline, you should use a text editor, like emacs.

6.2 Creating and Removing Rule Sets

To simply remove or add a rule set, we can use the functions `(remove-rule-set name)` and `(add-rule-set name type)`. Then rules can be added or removed from them either by using the previously mentioned editing functions, or through

```
(add-rule name-of-rule-set rule)
;; (add-rule 'GRAMMAR1 '(S ==> NP AUXP VP))

(del-rule name-of-rule-set rule)

(add-rule-list name-of-rule-set rule-list)
;; (add-rule-list 'DICT '((THE D) (AND CONJ)))

(del-rule-list name-of-rule-set rule-list)
```

These will not help us generate the tables the parsers need for running, but only base rule sets. Specialized functions are provided to compile rule sets into -TABLEs. Translating a CFG `GRAMMAR1` to a CFG-TABLE `CT` usable by the parser takes one step, which requires specifying the root node of the CFG and also whether $k = 0$ or $k = 1$. The lookahead table takes little time to compute, so presumably k will be 1. The translation function is `create-cfg-table`, which takes the form

```
(create-cfg-table table-name cfg-name root k)
```

or in our case, with root `ROOT`

```
(create-cfg-table 'CT 'GRAMMAR1 'ROOT 1)
```

The corresponding function for GPSG grammars is `create-gpsg-table`, which takes the same arguments, but as mentioned, we shall not need these sorts of grammars in this laboratory.

If you need more details on all the functions in the parsing system, please refer to the documentation files `doc.pdf` or `doc.ps`.