Three files:

- `vocab.txt`, the dictionary
- `grammar.txt`, the syntax definition, the rules of the language
- `atn.py`, the program that does the work

To change the names of the two `.txt` files, simply change the definitions of the variables `dictionary_file_name` and `grammar_file_name` at the very beginning of the program.

vocab.txt

Very few requirements. Each line must begin with a word and its part of speech. Many words have more than one classification (e.g. "ache" can be both a very and a noun), in which case they have more than one line in the file. No two lines may have the same word and the same part of speech. Keeping the file in alphabetical order is not required, but makes it easier to manage.

The part of speech can be anything you want. You don't have to do anything to create a new one, just start using it. The parts of speech in use in my grammar are as follows; some have been stretched in meaning from their traditional uses.

`adj`	adjective: big, fat, hairy
`adv`	adverb: quickly, hungrily, badly
`art`	article: a, an, the
`cnj`	conjunction: and, but, or
`dem`	demonstrative: this, that, those
`madj`	meta-adjective (works on other adjs or advs): very, quite, possibly
`nn`	noun: cat, hat, house
`num`	number: five, nine, some,
`pos`	possessive: my, her, our
`prn`	pronoun: she, him, it
`prp`	preposition: near, on, at
`que`	question starter: who, what, why
`vb`	verb: eat, chase, sit

After the word and part of speech, the rest of the line contains any number of pieces of extra information about this sense of the word. Each consists of a name (called a tag), optionally followed by a colon and a collection of values. If there is more than one value they are comma separated. Each piece of extra information is separated from the others by spaces, so they may not contain spaces themselves. If no value is given (meaning no colon) it defaults to true.

As with parts of speech, you can make up new ones at any time. Just use a new name or value and it will be accepted. Those in use in my grammar rules are:

`gen` gender, `f` for feminine, `m` for masculine, `n` for neuter; used in
- `nn`: man/woman/ache
- `pos`: his/her/its
- `prn`: she/he/it

`mod` general purpose modifier connected with `root` (below), used in
- `adj`:
  - `neg`: negative - "nasty" has `mod:neg` and `root:nice`
  - `more`: comparative, "nicer" has `mod:more` and `root:nice`
  - `most`: superlative, "nastiest" has `mod:most,neg` and `root:nice`
  - `nbn`: not before noun "the cat is yours" is OK, "yours cat is black" is not.
- `adv`: the only value used is `neg`.
- `cnj`: "eat a fish" is OK, "eat a fish and a fox" is OK, "eat a fish but a fox" is bad. "but" can only join larger elements, so "but" has `mod:big`
- `que`: `npa`: noun-phrase allowed - "which cat sat" is OK, "who cat sat" is bad, so "which" has `mod:npa`

`num` number, `s` for singular, `p` for plural; used in

- `art`: a is singular so `num:s`, the can be either so `num:s,p`
- `dem`: that is singular, those is plural
- `nn`: cat is singular, cats is plural
- `num`: one is singular, some is plural
- `vb`: eats in singular, eat can be either (based on subject - I eat, they eat)
- `pos`: my is singular, our is plural
- `prn`: she is singular, we is plural

`typ` type, getting more specific than part of speech; used in

- `art`: "an" has `typ:ind`, indefinite; "the" has `typ:def`, definite
- `vb`:
  - `t`: transitive - needs an object - "cats like mice", so like has `typ:t`
  - `i`: intransitive - object not wanted - "cats exist", so exist has `type:i`
  - `g`: gerund - an ...ing word, often used as an adjective, "existing mice"
  - `e`: existence - am, is, are, etc.
  - `b`: belief - think, believe, assume

`case` grammatical case, only used with `prn`:

- `n` nominative - she, we, they
- `a` accusative - her, us, them

`tns` tense, only used with `vb`: `pr` for present, `ps` for past.

only used with `nn`, no particular value associated, just true:

- `ani`: this is an animal
- `food`: this is generally considered to be food
- `name`: this is a proper noun, a name

`root` intended to make connections between closely related words. "slept" and "sleeping" both have `root:sleep`. The idea is that the final result of parsing would only use root words, so slept would be replaced by sleep but with an indication that the action happened in the past. This is most of the reason for `adj` and `adv` having `mod` entries, they serve no grammatical purpose.

The dictionary is read and processed when the `read_dictionary()` function call happens very near the end of the program. It is very easy to make hard-to-find mistakes when editing the dictionary file, so after any large edit add a `summarise_dictionary()` call next. That lists all the parts of speech that have been used, and for each it also lists all the extra information types that were used, and for each of those, every value that was given. Mistakes stand out clearly.

grammar.txt

The individual parsable items such as sentence, noun-phrase, and so on are referred to as graphs. The nodes of those graphs are usually referred to as states when talking about this sort of thing, so that is the term I have used in class. Unfortunately I slipped and went back to calling them nodes in the software. Changing that now would almost certainly introduce too many errors. I use the word "state" for the objects that get pushed on the stack to allow backtracking.

The indentation I use in the file is not required (meaning the software doesn't care about it) but it is important. Without it the whole thing becomes unreadable very quickly.

Each graph begins with a line that just says `graph` followed by its name. No two graphs may have the same name. The only things that appear inside a graph description are the descriptions of its states. No two states of the same graph may have the same name. A state description starts with a line that just says `state` followed by the state's name.

The system needs to know where to start. A line that just says `start` followed by the name of a graph indicates where the whole parsing process begins. Every graph must have a node named `start` and that is where the parsing process for that state starts.

The arcs are all represented by lists of length three: `[condition, progression, action]`. The condition is something that must be true for the arc to be followed. Progression is what happens to make progress through the input (consume a word, call a sub-graph, etc). Action is a thing (more usually some things) that will be done once the arc has been successfully followed.

Consider a simplified version of the graph for a noun-phrase, the sequence of words that specifies a *thing* of some kind. The first word is usually an article ("a", "the", etc) but that is optional. Then there can be some adjectives ("big", "ugly", etc) but they are optional too, and then there must be a noun ("cat", "house"). Alternatively, instead of all that, the whole thing could just be a pronoun ("she", "we", etc). This is how it begins:

```
graph np
  node start
    [("cat", "art"), "consume", ["build", ("to", "after-art")]]
    [("cat", "adj"), "consume", ["build", ("to", "after-art")]]
    [("cat", "prn"), "consume", ["build", "accept"]]
    [("cat", "nn"), "consume", ["build", "accept"]]
```

In the action part, `("cat",` *POS*`)` means that the next word must have a dictionary entry saying that its part of speech ("category") can be *POS*.

In the progression part, `"consume"` means that the word will no longer be there to be seen. We have read it, and will not see it again unless there is backtracking.

Whenever the action part is a list, all of its items are action parts themselves and will be carried out in left-to-right order. `"build"` means that the word just consumed will be added to the result object that we are creating, labelled with whatever part of speech was mentioned in the `"cat"`, and any other information the dictionary had for it. Obviously there can't be a `"build"`-like action unless there was a `"cat"`-like condition.

After an article or an adjective, it is permissible for another adjective to appear, but another article is not allowed, and a pronoun is no longer possible either. So we need to transition to another node to handle that different situation. This new node will have to accept a noun too.

The action `("to",` *NODE*`)` means that we will move to the named node in this same graph. The action `"accept"` means this we have successfully traversed the whole noun-phrase graph, and will return to wherever we came from. In most cases that will be to complete the traversal of an arc (of a kind we haven't seen yet) in another graph (or possibly in this same graph: they can be recursive). The result object build by all of our `"build"`-like actions is made available to that arc's actions.

After an article or adjective has been consumed:

```
  node after-art
    [("cat", "nn"), "consume", ["build", "accept"]]
    [("cat", "adj"), "consume", ["build", "stay"]]
```

The `"stay"` action is just a shorthand notation for `("to",` whichever-node-we-are-already-at`)`. That is not the same as not moving at all, it just represents a tight loop in the graph. We will start processing at the after-article node again at its first arc. The after-article node will be on the stack twice.

In all, this two node definition has covered all the given cases for the simplified version of a noun phrase: "you", "a cat", "the cat", "a big cat", "a big fat cat", "cats", "big fat cats", etc., but not the ungrammatical "you cat", "fat you", "a you", or just "the big". Sadly the ungrammatical "a cats" is also accepted, but that is fixable.

With that behind us, perhaps we can handle the complete noun-phrase definition (complete as in all that is in my version, not necessarily all that you will want). This does not match the rules of English precisely, but it gets close enough to be usable.

- a noun phrase can start with
  - an article,
  - a demonstrative ("this", "those", etc) instead
  - a pronoun ("i", "me", "mine", etc), but that would be the end of the NP
  - a possessive ("my", "your")
  - an adjective
  - or a noun all by itself
- the NP may continue with any number of adjectives
- after the optional adjectives there is another chance for a pronoun ("lucky you")
- or the noun itself, but not both
- numbers behave a bit like adjectives but not quite:
  - both numbers and adjectives are allowed unless there was a pronoun
  - there can only be one number and if there is one it must not come after an adjective
- words like "very" and "slightly" are often categorised as adverbs, but don't follow the rules for adverbs (you can run quickly and you can run very quickly but you can't run very). I call them meta-adjectives, and a group of them followed by one adjective behaves like a single adjective.

To save duplication the `("jump", "after-art")` action is used. It means just go on to the after-art node as though it were just a continuation of this node. The input word is not consumed, so we only need to deal with numbers and adjectives at one node. The `"always"` condition essentially just means "true", there are no conditions to following this arc. It does not mean that the arc must be followed, just that it can.

```
graph np
  node start
    [("cat", "art"), "consume", ["build", ("to", "after-art")]]
    [("cat", "dem"), "consume", ["build", ("to", "after-art")]]
    [("cat", "prn"), "consume", ["build", "accept"]]
    [("cat", "pos"), "consume", ["build", ("to", "after-art")]]
    ["always", (), ("jump", "after-art")]

  node after-art
    [("cat", "num"), "consume", ["build", ("to", "after-art-num")]]
    ["always", (), ("jump", "after-art-num")]
```

After-art only deals with numbers, but its own jump action moves us on to the one and only node that has to deal with adjectives. Seeing an adjective leaves us at the same node, we can take more of them, but as soon as a noun appears that's the end. Almost.

```
  node after-art-num
    [("cat", "madj"), ("sub", "adj-grp"), ["receive", "stay"]]
    [("cat", "adj"), "consume", ["build", "stay"]]
    [[("cat", "nn"), ("not", ("cat", "adj"))],       \
         ("sub", "nn-grp"),                          \
         ["receive", ("to", "after-nn")]]
    [("cat", "nn"), "consume", ["build", ("to", "after-nn")]]
```

(parenthetically this also illustrates splitting long lines. If the last non-space character on a line is `\`, the next input line will be taken as a continuation. This only happens in the grammar file because the the arcs from a node are processed by the Python `exec()` function, which only accepts entire Python expressions, so something is needed to tell the grammar reader to extend the line)

A meta-adjective ("very", etc) with the words it modifies is handled as a separate parsable thing with a graph of its own. The progression `("sub", "adj-grp")` calls another graph in the way that one function can call another. The parser starts anew at the adj-grp graph's start node, with an empty result object to build on. When and if that graph performs an accept action, control returns to this graph and the arc with the `"sub"` progression is at last followed and its cations carried out. `sub` actions do not consume the input word, it is left for the subgraph to see.

The action `"receive"` is a lot like `"build"`, but instead of taking the word that was just read and adding it to the result object, it takes the whole result object that the `"sub"` subgraph created and adds that to the current result object. Just as `"build"` may only be used after `"cat"`, `"receive"` may only be used after `"sub"`.

Compound nouns ("dog park", "bird house") add a small complication. Grammatically any two nouns may join together in this way, and because a lot of nouns are also adjectives, that leads to multiple parses that do not reflect a serious ambiguity. To avoid this, I decided that when a noun that can also be an adjective appears before another noun, it is always taken as an adjective, never as the beginning of a compound noun. The combination of conditions `[("cat", "nn"), ("not", ("cat", "adj"))]` requires that the current word can be classified as a noun but not as an adjective.

After the noun or compound noun, the noun-phrase could have more to come. A prepositional phrase tells us more about the thing in question ("the cat on the big chair"), so the after-nn node will look for a prepositional phrase if a preposition appears next, otherwise it will just accept what we have seen so far as the complete noun phrase.

```
  node after-nn
    [("cat", "prp"), ("sub", "prpp"), ["receive", "accept"]]
    ["always", (), "accept"]
```

Here are the three graphs that were used in the processing of a noun phrase:

```
graph adj-grp
  node start
    [("cat", "madj"), "consume", ["build", "stay"]]
    [("cat", "adj"), "consume", ["build", "accept"]]

graph nn-grp
  node start
    [("cat", "nn"), "consume", ["build", ("to", "after-nn")]]
  node after-nn
    [("cat", "nn"), "consume", ["build", "accept"]]

graph prpp
  node start
    [("cat", "prp"), "consume", ["build", ("to", "after-prp")]]
  node after-prp
    ["always", ("sub", "np"), ["receive", "accept"]]
```

Conjunctions bring two noun phrases together to act as one: "that cat ate two fish and a chicken", so the real top of the syntax tree for noun-like things is the noun phrase conjunction `"np-cnj"` graph:

```
graph np-cnj
  node start
    [("group", "npstart"),                 \
        ("sub", "np"),                     \
        ["receive", ("to", "had-np")]]
  node had-np
```

```
    [[("cat", "cnj"), ("not", ("has", "mod", "big"))],    \
         "consume",                                        \
         ["build", ("to", "had-cnj")]]
    ["always", (), ("accept", "nosolo")]
  node had-cnj
    ["always", ("sub", "np"), ["receive", ("to", "had-np")]]
```

The `("group", "npstart")` condition is simply a shorthand notation for a large `"or"` expression. At the beginning of the grammar there are two group definitions:

```
   group npstart art dem num pos prn madj adj nn
   group vpstart madj adv vb
```

This effectively makes a new part of speech called `npstart` that includes all articles, demonstratives, numbers, possessives, pronouns, meta-adjectives, adjectives and nouns, in other words any word that can be the first word of a noun phrase. It is not really a new part of speech, so we can't use it in `("cat", "npstart")` only with `"group"`.

The next big thing (nothing like as big as noun phrases) is the verb phrase, which groups together the words that describe something like an action "ate", "hungrily ate", "very hungrily ate", "quickly and hungrily ate", "ate with a spoon".

```
graph vp
  node start
    [("cat", "madj", "adv"), ("sub", "advp-grp"), ["receive", ("to", "need-vb")]]
    [("cat", "vb"),      \
        "consume",       \
        [("note", "vtype", ("tag", "typ")), "build", ("to", "had-vb")]]
  node need-vb
    [("cat", "vb"),      \
        "consume",       \
        [("note", "vtype", ("tag", "typ")), "build", ("to", "had-vb")]]
  node had-vb
    [("cat", "madj", "adv"), ("sub", "advp-grp"), ["receive", "accept"]]
    ["always", (), "accept"]
```

The action `("cat", "madj", "adv")` from the start node is a smaller version of the large condition that `("group", "npstart")` relieves. The current word is checked against all the given parts of speech in order. As soon as one matches it is exactly as though just a single `cat` with that part of speech only had been used.

The action `("note",` *NAME*, *VALUE*`)` evaluates the expression *VALUE* and makes a note of it. Just as the following of a node's arc adds information the the current result object, it may also add to a collection of other relevant information, the list of notes (which is actually a Python dictionary, *NAME* is the key) These notes may be referred to later and are made available to the actions of the `"sub"` arc that led to this graph.

The expression used here, `("tag", "typ")` looks at the dictionary entry for the current word (so it needs a previous `"cat"` condition) to find the value associated with the tag `"typ"`. As we are dealing with a verb here, the result could be either `"t"` for transitive or `"i"` for intransitive. That is information that will be important later to see if verb has (or can take) an object. The second arc from node `start` could just as well have been written as `["always", (), ("to", "need-vb")]`.

The `had-vb` node corresponds with the `had-np` node for a noun phrase, it allows another adverbial phrase to be added after the verb. The subgraph `advp-grp` is so named because it allows conjunctions like "quickly and hungrily".

```
graph advp-grp
  node start
    [("cat", "madj", "adv"), ("sub", "advp"), ["receive", ("to", "after-advp")]]
  node after-advp
    ["always", (), ("accept", "nosolo")]
    [[("cat", "cnj"), ("not", ("has", "mod", "big"))],     \
         "consume",                                        \
         ["build", ("to", "start")]]
    [("cat", "madj", "adv"), ("sub", "advp"), ["receive", "stay"]]

graph advp
  node start
    [("cat", "madj"), "consume", ["build", "stay"]]
    [("cat", "adv"), "consume", ["build", "accept"]]
```

The adverbial phrase graph advp, accepts an optional sequence of "very"-like words followed by an adverb, and advp-grp must start with one of those. That is usually all there is, hence the "always" arc. The ("accept", "nosolo") action is slightly different from the usual "accept" action, and removes unnecessary levels in the parse tree. If there was indeed a conjunction with two or more noun phrases and one or more "and"-like word, then we do need a new tree node to hold them all together. But if there was no conjunction, just a noun phrase, we do not need an extra tree node and we certainly don't want it labelled as exactly what it isn't: "advp-grp". The "nosolo" addition means that if the current result object has only one thing in it, that one thing is returned on its own, so in this case the node we get is labelled only as and "advp".

The second arc accepts what I earlier called small conjunctions. This is a mistake. It is correct in noun phrases (you can "eat a sausage and a potato" but you can't "eat a sausage but a potato") but adverbs are different, you can "eat hungrily but sincerely"). I am also not convinced of the correctness of the third arc, it allows the conjunction to be left out so "eat hungrily quickly" is taken as correct.

Now these things can be put together to make various kinds of sentences. I separate them into:

Statement - has subject and maybe an object: "the cat eats", "the cat ate a big dog".
Command - no subject: "eat", "eat that sandwich".
Question - with or without obvious subject: "who ate that", "which cat ate that"

```
graph sentence
  node start
    [("group", "npstart"),   \
        ("sub", "np-cnj"),   \
        [("note", "kind", "statement"), "receive", ("to", "need-vb")]]
    [("group", "vpstart"),                 \
        ("sub", "vp"),                     \
        [  ("lift", "vtype"),              \
           ("note", "kind", "command"),    \
           "receive",                      \
           ("to", "had-vb")]]
    [[("cat", "que"), ("has", "mod", "npa")],     \
         "consume",                               \
         [("note", "kind", "question"), "build", ("to", "need-subj")]]
    [("cat", "que"),      \
        "consume",        \
        [("note", "kind", "question"), "build", ("to", "need-vb")]]
```

First arc: if the first word can be at the beginning of a noun phrase, parse a noun phrase or conjunction of noun phrases, and add the fact that "kind" is "statement" to the list of notes, and continue knowing that we need a verb phrase.

Second arc: if the first word can't begin a noun phrase, there is no subject so this can't be a statement, but if it can begin a verb phrase, this must be a command. So parse a verb phrase and move on to the "had verb" node. There is a new action on this arc: `("lift", "vtype")`, `"lift"` can only be used after a `"sub"` and it simply takes the value of the note called `"vtype"` from the just-completed sub-graph, and add it to our own list of notes. The verb phrase had a note of whether the verb was transitive or intransitive or both, now the sentence also has that information.

Third arc: a question word that also allows a noun phrase to follow as in "which cat ate the dog" but not "who cat ate the dog". Simply note that the sentence type is `"question"` and continue as for a normal sentence.

Fourth arc: and kind of question word, including those that allow a noun phrase to follow. The `npa` question words allow a noun phrase to follow, they don't require it: "which ate the dog" is elliptical but not wrong.

The "need subject": node seems to be totally wrong at first sight, it is forbidding all the words that usually begin a noun phrase ...

```
  node need-subj
    [("cat", "art"), (), "fail"]
    [("cat", "dem"), (), "fail"]
    [("cat", "prn"), (), "fail"]
    [("cat", "pos"), (), "fail"]
    [("group", "npstart"),      \
        ("sub", "np-cnj"),      \
        [("note", "kind", "question"), "receive", ("to", "need-vb")]]
```

... but it isn't. `need-subj` is only reached after a question word, and you can't say "which the cat ..." or "which me ...". The `"fail"` actions all do what their name suggests. They act as though a condition failed and cause backtracking. The first four arcs act as filters, only words that are not articles or demonstratives or pronouns or possessives will ever reach the fifth arc which allows anything else that can begin a noun phrase.

"Need verb" is straightforward, it tries to parse a verb phrase and if successful moves on to the same "had verb" as we reached after the verb in a command.

```
  node need-vb
    [("group", "vpstart"),      \
        ("sub", "vp"),          \
        [("lift", "vtype"), "receive", ("to", "had-vb")]]
```

"Had verb" now makes use of our knowledge of what kind of verb we saw.

```
  node had-vb
    [("=", ("value", "vtype"), "t"),      \
         ("sub", "np-cnj"),               \
         ["receive", ("note", "had-obj"), ("to", "had-obj")]]
    [("=", ("value", "vtype"), "i"), (), ("to", "had-obj")]
    [("=", ("value", "vtype"), "e"), (), ("to", "need-comp")]
```

The first arc's condition is an equality test. The expression `("value", "vtype")` evaluates to the value of the `"vtype"` note in the current note list. Single strings as expressions evaluate to themselves, so this arc is taken if the verb was (or could be) transitive. We need another noun phrase to be the object of the sentence.

The second arc, for intransitive verbs, has the progression `()` which means do nothing, no input is consumed. So it is the same as the first arc, but it doesn't parse a verb phrase before moving to the same node.

The third arc is for verbs that express existence ("is", "are", etc) they can take a description (or complement) instead of an object as in "the cat is orange", "they are here". Note that the existence verbs have three types attached to them, `"i"`, `"t"`, and `"e"`, so they can be used in other ways: "the dog is an animal", "be a pal".

```
node had-obj
  [[("cat", "prp"), ("noted", "had-obj")],    \
       ("sub", "prpp"),                       \
       ["receive", ("to", "had-prpp")]]
  [("cat", "madj", "adv"),                    \
      ("sub", "advp"),                        \
      ["receive", ("to", "had-advp")]]
  ["always", (), ("accept", ("value", "kind"))]
```

The node name "had-obj" is slightly misleading. It is the node we would reach if we had an object, but we would also eventually reach it if we hadn't. It is really the node at which we will not accept any more objects.

The first arc's strangeness is a mistake for you to set straight. It is supposed to deal with situations like "the cat frightened the dog near the door". Is it saying that the dog that was frightened by a cat is (now) near the door, or that the frightening of the dog by the cat took place near the door? In one case the preposition attaches to the noun phrase "the dog", in the other it attaches to the verb phrase "frightened". They are different but both potentially correct. It is ambiguous so two parse trees should eventually be produced.

The third arc's `("accept", ("value", "kind"))` is also new. If `"accept"` is combined with any expression other than `"nosolo"`, it changes the label of the tree node that is about to be created. Normally `"accept"` labels the node with the name of the current graph. Here it uses the value of the expression instead. Recall that the note called `"kind"` was previously set to one of `"statement"`, `"command"`, or `"question"`.

After that, the next two nodes are nothing new:

```
node had-advp
  [("cat", "prp"), ("sub", "prpp"), ["receive", ("to", "had-advp-prpp")]]
  ["always", (), ("accept", ("value", "kind"))]
node had-prpp
  [("cat", "madj", "adv"), ("sub", "advp"), ["receive", ("to", "had-advp-prpp")]]
  ["always", (), ("accept", ("value", "kind"))]
```

The final two allow conjunctions of prepositions and adverbs as in "near the door and above the floor". "near the door and very quickly". The node `"had-advp-prp"` is only reached after we have had an adverb or a preposition. If it sees a "small" conjunction it will required another adverb or preposition and repeat if necessary. The "small"-ness requirement is probably wrong, you can "eat near the window and above the floor" and "eat near the window but above the floor" equally well.

```
node had-advp-prpp
  [[("cat", "cnj"), ("not", ("has", "mod", "big"))],    \
       "consume",                                       \
       ["build", ("to", "need-more")]]
  ["always", (), ("accept", ("value", "kind"))]
node need-more
  [("cat", "prp"), ("sub", "prpp"), ["receive", ("to", "had-prpp")]]
  [("cat", "madj", "adv"), ("sub", "advp"), ["receive", ("to", "had-advp")]]
```

The complete list of possible expressions in my version (I expect you will add more) are:

`always`	always true
`end`	true if there are no more input words
`more`	true if there is at least one more input word
`(cat,` *P1*`,` *P2*`, ...)`	true if the current input word has any of those parts of speech
`(word,` *W1*`,` *W2*`, ...)`	true if the current input word is exactly equal to any of those listed
`(group,` *G*`)`	true if the current input word has a part of speech included in group *G*
`(noted,` *N*`)`	true if there is a current note named *N* and its value is not False
`(noted,` *N*`,` *V1*`,` *V2*`, ...)`	true if there is a current note named *N* and its value is equal to the value of one of the expressions *Vi*
`(value,` *N*`)`	equal to the value of the current note named *N* or False is there isn't one
`(=,` *A*`,` *B*`)`	expressions A and B are evaluated. If they both have simple values this is an equality test, if one has a simple value and the other has a set of values this is a membership test. If are sets of values this is true if there are any elements at all in their intersection
`(has,` *T*`)`	true if the current input word's entry for the part of speech recognised by the prior `(cat, ...)` has a tag named *T* and its value is not False
`(has,` *T*`,` *V1*`,` *V2*`, ...)`	true if the current input word's entry for the part of speech recognised by the prior `(cat, ...)` has a tag named *T* and its value is equal to the value of any of the expressions *Vi*
`(tag,` *T*`)`	true if the current input word's entry for the part of speech recognised by the prior `(cat, ...)` has a tag named *T* this returns its value (or set of values) and an empty set otherwise
`(not,` *E*`)`	if the value of expression *E* is a bool return its negation, error otherwise
`(or,` *E1*`,` *E2*`, ...)`	true if any *Ei* expression evaluates to true, false if all evaluate to false, error otherwise

The complete list of possible actions in my version (I expect you will add more) are:

`accept`	equivalent to `(accept, asis)`
`build`	add the current (`cat` accepted) input word to the current result object, labelled with the part of speech that `cat` accepted
`receive`	add the current (`sub` accepted) subgraph parse tree to the result object
`stay`	equivalent to `(to,` the-current-node's-name`)`
`fail`	force backtracking
`[`*A1*`,` *A2*`,` *A3*`, ...]`	perform all the actions *Ai* in turn
`(accept,` *X*`)`	exactly as seen in the earlier descriptions; `"accept"` is converted to `("accept", "asis")`
`(to,` *X*`)`	transition the arc to node *X*, adding a new state to the stack.
`(jump,` *X*`)`	transition the arc to node *X*, the top state in the stack is modified, no new state is added.
`(note,` *N*`)`	add a note that *N*'s value is true to the current note list
`(note,` *N*`,` *E*`)`	add a note that *N*'s value is the value of expression *E* to the current note list
`(lift,` *N*`)`	add note *N* from the recently `sub`-accepted subgraph to the current note list

atn.py

Many of these Python functions can be left untouched and unexamined, you can use them as they are. But you are invited to examine and touch (modify) them because you might want to add a new feature to support parts of your design. If you keep a note of what everything's type is, you shouldn't have much trouble understanding them.

`read_dictionary` does what its name says.

`show_dictionary` does too. It is helpful when something goes wrong and the dictionary was not read as expected but you can't see why.

`summarise_dictionary` should be used any time you make additions or changes to the dictionary `vocab.txt`, it is very easy to mistype something in that file and never see it. This function's output is very small but shows just once everything that appeared anywhere. Mistakes stand out.

`read_grammar` and `show_grammar` are also both exactly what they claim to be. `show_grammar` produces a lot of output, but all in a uniform way, so mistakes are easier to see.

`clean` is something you will probably not need. I need it when I am running Python interactively under unix, where backspaces are not processed correctly for some reason.

`make_start_state` prompts the user and reads the whole line that they type, it splits the line into a list of words based on space separation, and creates the first state to go on the stack.

It has a useful debugging feature: if part of your grammar isn't working properly, let's say the verb phrase graph, it is very annoying to have to type whole sentences when you only want to test a couple of words. If the first word of input begins with a *, it is removed from the input and taken as the main graph to be parsed.

Recall that a grammar file must include a start line, something like `"start sentence"` that tells it that the graph for `sentence` is where to start. If the first word is `"*vp"` then the graph for `vp` is where it will start and end. `make_start_state` creates a fake graph called `**` so that at can insist that no stray words follow your test `vp`.

`backtrack` trivially does what its name says, discards the newest state from the stack. It only exists so that backtracking actions stand out to the eye.

`simple`, `minimal`, `almost_nothing`, `print_stack_item`, and `print_stack` are used to print states from the stack or the entire stack in a way that shows detail where it is usually needed but leaves out details that are usually unimportant. Stack states get big and unreadable very quickly otherwise.

The dictionary is a Python `dict` whose keys are the individual unique words. The dictionary entry for a word is another dictionary, its keys are the parts of speech that the word can have, and the associated values are also dictionaries whose keys are the tags from the dictionary entries and the associated values are always sets containing all the values (strings) associated with that tag, or "true" if none were given. For example, the `vocab.txt` entry for "sat" is

```
      sat vb num:s,p tns:ps typ:i root:sit
```

and `dictionary["sat"]["vb"]` is

```
      { "num": {"s", "p"},
        "tns": {"ps"},
        "typ": {"i"},
        "root": {"sit"} }
```

The grammar is also a Python `dict`, its keys are the graph names and the associated values are the graphs themselves, also represented as dictionaries whose keys are the node names and associated values are lists containing that node's arcs. The arcs are three-item lists `[condition, progression, action]`. For example, the `grammar.txt` entry for the graph `np-cnj` is

```
      graph np-cnj
        state start
           [  ("group", "npstart"),
              ("sub", "np"),
              ["receive", ("to", "had-np")]  ]
        state had-np
           [  [ ("cat", "cnj"), ("not", ("has", "mod", "big")) ],
              "consume",
              [ "build", ("to", "had-cnj") ]  ]
           [  "always",
              (),
              ("accept", "nosolo")  ]
        state had-cnj
           [  "always",
              ("sub", "np"),
              [ "receive", ("to", "had-np") ]  ]
```

and `grammar["np-cnj"]` is

```
   { 'start': [ [ ('group', 'npstart'),
                  ('sub', 'np'),
                  ['receive', ('to', 'had-np')] ] ],
     'had-np': [ [ [ ('cat', 'cnj'),
                     ('not', ('has', 'mod', 'big')) ],
                   'consume',
                   [ 'build', ('to', 'had-cnj') ] ],
                 [ 'always',
                   (),
                   ('accept', 'nosolo') ] ],
     'had-cnj': [ [ 'always',
                    ('sub', 'np'),
                    [ 'receive', ('to', 'had-np') ] ] ] }
```

`parse` is the big function, it handles everything from reading input to printing the results. Some sort-of clumsy global variables are used to communicated between various parts:

`dictionary`	is the dictionary `dict`, described above.
`grammar`	is the grammar `dict`, described above.
`catword`	is a two-tuple set to `(`part of speech`,` actual word`)` by a successful `cat` condition
`used`	is also a two-tuple with a similar but more general purpose. A successful `cat` condition sets it to `(`part of speech`,` actual word`)`, A successful `word` condition sets it to `("word",` actual word`)`, and an `accept` action sets it to `(`name of graph`,` parse tree node result object`)`.
`upnotes`	is set by an `accept` action to the successful graph's list (really `dict`) of notes.
`typed`	is the list of input words
`stack`	the stack of states: the stack is a tuple, either `()` (when empty) or `(`current state`,` rest of stack`)` otherwise, so if there are four states on the stack, the top one being S_0, the stack is (S_0 `, (`S_1 `, (`S_2 `, (`S_3 `, ( )))))`.

States (on the stack) are lists of length 7, and the positions have mnemonic names:

0, `s_graph`:	the current graph's name
1, `s_node`:	the current graph node's name
2, `s_inpos`:	how many input words have been consumed, so `typed[state[s_inpos]]` is the current input word
3, `s_arcpos`:	which arc from this node is to be tried next
4, `s_build`:	the current parse tree node or result object
5, `s_notes`:	the current list of notes
6, `s_lastcall`:	is the representation of the "meta-stack", and is just a reference to the stack as it was at the time of the most recent `sub` call to a subgraph. It is the actual two-tuple that represented the whole stack at that moment.

print_parse prints the parse trees (result objects) in a readable way.

evaluate is used to evaluate conditions like ("cat", "nn") and expressions like ("value", "vtype"). First it prepares frequently needed information, setting word to the next input word or "<end>" if there is no input left, and looking up the word in the dictionary. The variable entries is set to a dictionary whose keys are the parts of speech that the word may have, and the associated values are everything from that particular dictionary entry.

The evaluate function is split into sections based on the expression's type: simple strings usually represent themselves, but "always" returns True, "end" is True if there is no input left, and "more" is True is there is some input left. Then there is a big section for tuples because they are the form most non-trivial expressions have, and finally a section for lists. Lists are just an implicit and of all of their items, treated as individual expressions.

Given the descriptions of the essential variables and what the various conditions do, the details of the code should be comprehensible.

perform is evaluate's counterpart for carrying out actions like "accept" and ("note", "a", "x"). Given the earlier descriptions of what all the actions do and the meanings of the important variables and the structure of a state, everything except accept should be easy to understand. It is unlikely that you will want to modify action, but here is a very short summary of it: first it checks for the various accept options like nosolo, Then it deals with complete successes (an accept in the root graph named ** that was created by make_start_state) by printing the results and deliberately causing backtracking to find alternative parses. The final part deals with accepts from sub-graphs by extracting the important information from the state, finding the sub arc that called this graph, and completing its actions.