Lecture 24: Parsing via searching

This lecture describes an approach to parsing context free grammars via searching a state space. This is *not* the most efficient parsing method, but it illuminates properties of both state space search and context free grammars.

Another example grammar

We consider another grammar, for a small and idiosyncratic fragment of English sentences. The rules are as follows.

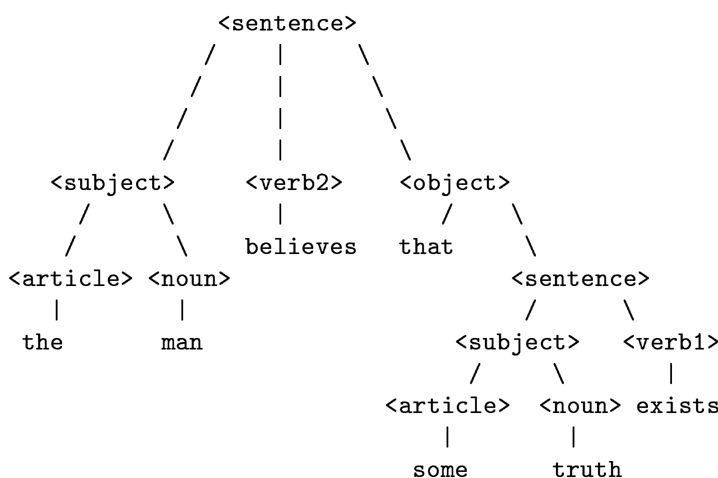
```
<sentence> -> <subject> <verb1>
<sentence> -> <subject> <verb2> <object>
<subject> -> <article> <noun>
<subject> -> <pronoun>
<object> -> that <sentence>
<verb1> -> swims | pauses | exists
<verb2> -> believes | hopes | imagines
<article> -> a | some | the
<noun> -> lizard | truth | man
<pronoun> -> he | she | it
```

Here we have used the notation -> to separate the lefthand and righthand sides of each rule, and the notation | to separate alternative righthand sides for some of the rules. The terminal symbols of this grammar are the words swims, pauses, exists, and so on. The nonterminal symbols of this grammar are the grammatical categories <sentence>, <subject>, <object>, and so on. We specify that the start symbol is <sentence>.

One of the strings of words that can be generated from <sentence> in this grammar is the following.

```
the man believes that some truth exists
```

As "proof" that this string can be so generated, we may provide a parse tree for the string as follows.



Leftmost derivations

Another way of demonstrating that the string can be generated from the nonterminal `<sentence>` is to give a leftmost derivation of the string from the nonterminal. A leftmost derivation starts with the single nonterminal symbol and repeatedly replaces the leftmost nonterminal symbol in the string with the righthand side of a grammar rule that has that symbol as its lefthand side, until the target string has been derived. Here is a leftmost derivation for this example.

```
<sentence>
<subject> <verb2> <object>
<article> <noun> <verb2> <object>
the <noun> <verb2> <object>
the man <verb2> <object>
the man believes <object>
the man believes that <sentence>
the man believes that <subject> <verb1>
the man believes that <article> <noun> <verb1>
the man believes that some <noun> <verb1>
the man believes that some truth <verb1>
the man believes that some truth exists
```

Note that one rule is used at each step. For example, the rule `<subject> -> <article> <noun>` is used to obtain the third line from the second.

Parsing context free languages

We consider writing a program for the following task. Given a nonterminal symbol, a string, and a context free grammar, decide whether the string can be derived from the nonterminal symbol using the rules of the given grammar. Note that this is a more general task than devising a particular program to solve this problem for a particular context free grammar, for example, a program to recognize the sentences generated by the grammar above.

In fact, at this point, we have no particular assurance that there is such a program – the task might specify an algorithmically unsolvable problem.

One possible approach is "brute force search." We seek a way to generate every possible string of terminal and nonterminal symbols that can be derived from the given nonterminal symbol using the rules of the given grammar. If we discover the target string among these possibilities, we can stop and output `#t`. However, if the target string cannot be generated, we need something that will tell us we can stop and output `#f`. From the nonterminal `<sentence>` we can derive an infinite number of sentences. (Stop and convince yourself of this if necessary.) Thus we cannot simply examine all the possibilities.

We make a simplifying assumption at this point. We assume that no nonterminal symbol in the grammar derives the empty string of symbols. Thus, starting from a string of 10 terminal and nonterminal symbols, no string of fewer than 10 symbols can be derived. Thus, if the target string has 10 symbols in it, we do not need to consider any strings of terminal and nonterminal symbols of length greater than 10.

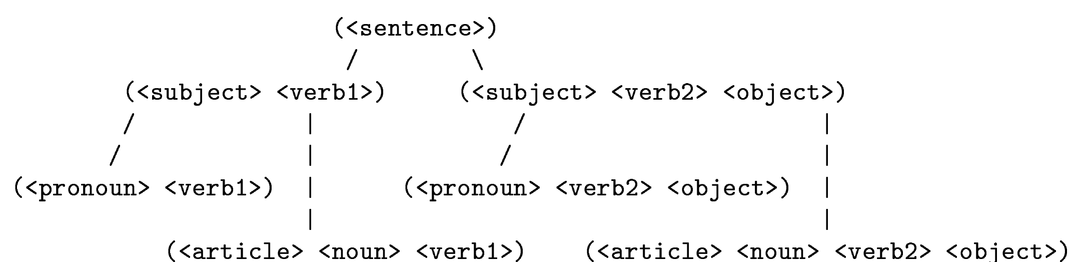
This assumption allows us to give a finite bound for the size of the set of possibilities we have to consider. The finite bound is large, exponential in the length of the target string, with a base depending on the number of terminal and nonterminal symbols in the grammar. For example, for the grammar above, there are 16 terminal symbols and 8 nonterminal symbols. So, considering all strings of 10 terminal or nonterminal

symbols, we get 24^{10} possible strings. Very large, yes, but finite.

Searching a state space

We can model the parsing problem as searching in a state space. For example, suppose we want to decide whether we can derive the string (the truth pauses) from the nonterminal <sentence> in the above grammar.

Each state is a finite string of terminal and nonterminal symbols of the grammar. The start state is the string containing just the nonterminal symbol of interest, in the example, <sentence> The goal state is the string containing the specified target string, in the example, the truth pauses. We put in an arrow from state1 to state2 if there is a rule in the grammar whose lefthand side is the leftmost nonterminal in state1, and when we replace that nonterminal by the righthand side of the rule, we get state2. (This corresponds to one step in a leftmost derivation.) A picture of the states near the start state follows.



The reader should imagine arrows directed downward in this diagram. Also, there may in general be arrows from several different states coming into one state; an example is given in the last section. Our task is then to establish whether there is a path of edges from the start state to the goal state.

People have spent a lot of effort on ideas for search methods in such state spaces. For example, we could search forward from the start state, or backward from the goal state (following arrows in the reverse direction), or both at once. We could search in a breadth-first manner, exploring all the states that are one step from the start state, then all the states that are two steps from the start state, then all the states that are three steps from the start state, and so on. We could search in a depth-first manner, choosing one of the states reachable in one step from the start state and recursively searching from there, and only backtracking to try other states at one step from the start state if the recursive call fails to find the goal state.

Do we have to worry about wandering in circles during our search? That is, could it happen that we follow a path of arrows that brings us back to the same state again? We could in general avoid this by keeping a growing list of all the states we have visited in our search, to avoid repeating our earlier searches. However, for this problem, we will make another assumption about the grammar. We assume that there is no leftmost derivation of one or more steps that leads from a single nonterminal back to that nonterminal. As an example of the behavior we are ruling out, consider the following small grammar.

```
A -> B | a
B -> A | b
```

Here the terminal symbols are a and b and the nonterminal symbols are A and B. There is a leftmost derivation of more than one step of A from A as follows.

```
A
```

```
B
A
```

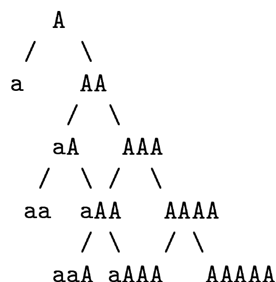
It is clear how this can lead to loops in the state space. What is not so clear, but true, is that ruling this phenomenon out is sufficient to avoid loops in the state space.

State space not always a tree

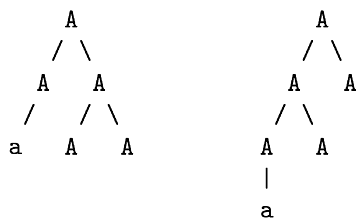
We consider another small example to show that the state space reachable from the start state is not always a tree. The grammar has one terminal, `a`, and one nonterminal, `A`, and two productions, as follows.

```
A -> a | AA
```

The state-space near the state `A` looks as follows.



Note that the state `aAA` is reachable via two different paths from `A`, corresponding to two different parse trees as follows.



If some string has two different parse trees with respect to a grammar, the grammar is said to be ambiguous. Ambiguity is generally an undesirable property in grammars for programming languages because the meaning of a piece of program text is based (in part) on its parse tree. Unfortunately, testing a context free grammar for ambiguity is an undecidable problem.