The LR theorem

As mentioned in class, the actual construction that ocamlyacc does – the “LALR(1)” construction – is rather involved. If you’re curious to see it, you can find it in any compiler textbook, or on the web. You will actually want to look for the “SLR(1)” construction, which is much simpler but gives the idea; the “LR(1)” construction is much more involved than the SLR(1) construction, and the LALR(1) construction is a refinement of the LR(1).

However, if you don’t want to see the construction but are curious about what the verbose ocamlyacc output means – and, in particular, what “states” are – I can kind of explain that. The entire LR(1) approach to parsing is based on this theorem, which I’ll attempt to explain:

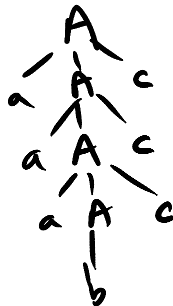
Theorem: For any grammar G, consider all parse trees of G. Each tree corresponds to a shift/reduce parse (as we discussed in class). Consider all the stack configurations that occur during any of these parses. Together they form a language over the symbols (terminals and non-terminals) of G. That language is finite-state.

First, what does it mean to say “each tree corresponds to a shift/reduce parse”? Recall that when we were practicing shift/reduce parses in class, we always referred to the tree to decide whether to shift or reduce. It is important to understand that the shift/reduce parsing method can, in principle, produce any parse tree; it has no inherent limitations. The only question is whether it is possible to *decide on the correct action* based only on the stack and a single lookahead symbol.

Now, the idea is to look at the stack configurations – that is to say, what is on the stack – as a language. We’ll do some examples. Consider this simple grammar:

A -> aAc | b

This produces the language $a^n bc^n$, i.e. a b surrounded by matching a’s and c’s. Note that this is *not* a finite-state language, because it involves nesting. Consider the parse tree for sentence aaabccc:



Now consider the shift-reduce parse for this tree: It starts by shifting all the a’s, then the b; so the stack configurations up to now can be described as some a’s, possibly followed by a b. At this point, we do a reduce action, leaving the stack aaaA. We now shift the c, giving aaaAc, and then reduce, giving aaA.

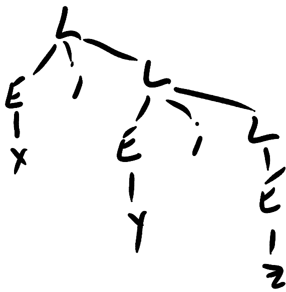
Shift the next c and reduce, giving aA, and finally the last c, and reduce, giving A. Here, then, are *all* the stack configurations encountered during this parse: a, aa, aaa, aaab, aaaA, aaaAc, aaA, aaAc, aA, aAc, A.

It's pretty clear, I suppose, that it doesn't matter how many a's and c's we start with, the process will be essentially the same, and the stack configurations will be: a, aa, …, a^n, a^nb, a^nA, a^nAc, a^{n-1}A, a^{n-1}Ac, …, aAc, A. This is a regular set; it can be defined by the regular expression a*(b|A|Ac).

As another example, consider the grammar

L -> E; L | E
E -> id

which describes sequences of id's separated by semicolons. A typical sentence is x;y;z, with parse tree:

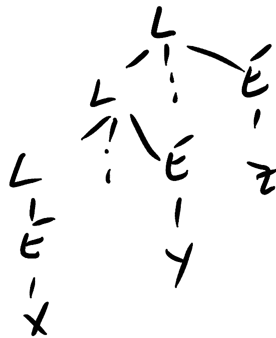


Again consider the shift/reduce parse that produces this tree: It shifts x and reduces (so the stack can be x or E), then shifts ; and y (giving stacks E; and E;y), then reduces (E;E) and so on. In the end, all the stack configurations – and again, we could extend this to a list of any length – have the form of some number of E's separated by semicolons, possibly followed by a semicolon and an id. This is again a regular set, described by the regular expression: id | E (; E)* (; (id)?)?.

It is instructive to look at the left-recursive version of this last grammar:

L -> L ; E | E
E -> id

x;y;z has parse tree:



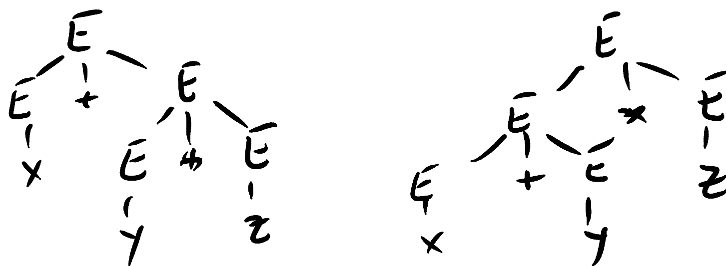
If we look at the shift/reduce parse, we first put x on the stack, then reduce to E, then L (so, so far, the stack can just be either id, E, or L), then shift the ; and the y (so we can have L; or L;id), then reduce

(giving L;E) and reduce again (L), and we're back where we started. In fact, in this case, no matter how long the list is, the only stack configurations possible are: id, E, L, L;, L;id, and L;E. This language is not only finite-state, it's actually finite!

As a last example, consider the expression grammar

E -> E + E | E * E | id

This grammar is, of course, ambiguous, so it can have many parse trees for the same input. Consider input x + y * z. It has two parse trees:



which lead to different shift/reduce parses. For the left tree, the stack configurations are any sequence of E's separated by + or * (with the last E possibly being an id instead); this is given by the regular expression (E (+|*))* (id|E)?. For the right tree, it is a finite language: it's either id, E, E+, E+id, E+E, E*, E*id, or E*E – that's it, just those eight possibilities. We can see that this finite language is a subset of the first language. It is not perfectly obvious that this language accounts for every parse tree, because some parse trees can contain within them both kinds of trees (left-leaning and right-leaning). But in fact, that regular expression does account for all stack configurations for this language.

Obviously, this is not a proof of the theorem – the proof is essentially the LR(1) construction – but hopefully makes the theorem seem plausible. But so what?

Here's why this matters: If the stack configurations for a grammar constitute a finite-state language, then we can find a DFA that recognizes them. Now we can do the following: given any stack configuration – starting with empty, which is always a legal stack configuration (even though we didn't include it in our languages) – look at the next symbol (the lookahead symbol) and run the DFA to answer this question: if we shift, will we still have a legal stack configuration? If so, then shift; if not, then reduce.

That is the basic idea behind LR(1) parsing. It's not complete, for several reasons; for starters, it doesn't say which production to reduce by. But it is the starting point. The states of the verbose ocamlyacc output are just the states of this DFA, called the "characteristic automaton" of the grammar.