LR parsing w action/goto tables

- Stack-based LR parsing algorithm, with language-specific details contained within an action table and a goto table
- Maintain stack of states during derivation (each state reflecting current derivation frontier)
- For state/input combinations: action table identifies a shift/reduce to use
- For state/nonterminal combinations, goto table specifies an updated state
- Ideally, we want to automatically generate the action and goto tables from the grammar rules for a language

Action/goto-based algorithm

- read first input word into a
- while not finished (not accepted/rejected yet)
  - assuming s is state on top of stack
  - if (action[s,a]==shift t) then push t, read next input into a
  - else if (action[s,a]==reduce A->X) then
    - pop length(X) symbols off the stack, let t = resulting top of stack state
    - push goto[t,A] onto stack, i.e. our new state gets pushed
    - store or output the A-->X production
  - else if (action[s,a]==accept) then break and accept
  - else break and invoke error handling

Action/goto tables

- Of course the algorithm relies on having the tables correct
- For each possible current state + input, the action table was used to specify either
  - (a) what to shift (what state to push on stack), or
  - (b) what to reduce (which grammar rule to apply)
- For each possible state + nonterminal, the goto table was used to specify a replacement state (to push on stack)

Computing the tables

- We'll look at an algorithm to build the tables for us, based on the language grammar rules
- Our algorithm will implicitly model a DFA that reflects all the states the top of stack could be in, and the appropriate transitions between them based on the next input character
- We can identify the possible conditions for the top of the stack based on the available production rules and the portion of each rule's RHS that is on the top of the stack

Example

- Suppose we have rules
  - S --> XY
  - X --> aXb
  - X --> c
  - Y --> de

The possible top of stacks represent where we could be in each of these rules RHS so far, here indicated with a .

S-->.XY   S--> X.Y   S-->XY.
X--> .aXb   X-->a.Xb X--> aX.b   X-->aXb.
Y--> .de   Y-->d.e   Y-->de.

Pairing condition and input

- We pair the top of stack condition with tokens that could potentially follow the production
- [A --> BC.D, x] indicates we've seen the B and C portions that match this A production, and we can apply this rule if we match D followed by an x (i.e. the x will be the first token after the completion of the A)
- [A --> .BCD, x] means this production rule is a possibility
- [A --> BCD., x] means that if an x is next in our input then this production can be completed
- Intermediate steps mean the rule is partially complete

Building the DFA (implicitly)

- We'll start with the grammar start state, and incrementally build reachable states
- To do so, we'll use two functions: goto and closure
- goto identifies transitions and computes the new state that would result from current state s and input symbol x
- closure takes an existing state and adds all conditions implied by that state
  - e.g. given [S --> X.YZ, x], a Y followed by Zx would complete this reduction to S, but so should a string that reduces to Y (then is followed by Zx)

The goto function

- goto(s,a)
  - s represents a current state as a set of production rules with derivation state noted, e.g. { [X-->A.B, x], [Y-->C.E, x] }
  - a is a grammar symbol (token or nonterminal)
  - goto computes and returns the state that would be the result of a transition from s on a, using algorithm:
    - results = { }
    - for each element, e, of s
    -     if e has form [X-->Y.aZ, x] add [X-->Ya.Z,x] to results
    - return closure(results)

The closure function

- closure(s)
- s again represents a state as a set of production rules with the derivation state noted, e.g. { [X-->A.B, x], [Y-->C.E, x] }
- closure computes all corresponding derivation states using the following algorithm:

```
repeat until s does not change:
  for each element, e, of s [ A-->B.CD, x]
    for each production C --> E in the grammar
      for each character f in FIRST(Dx)
        add [C-->.E,f] to s
```

The construction algorithm

```
given S is root nonterminal for the grammar, add: S'-->S
start state, c0 = closure({ [S'-->.S, eof]}
set of all states, All = { c0 }
repeat until All does not change:
    for each state, ci, in All that has not yet been processed
        mark ci as processed
        for each symbol x following the . in any item in ci
            t = goto(ci,x), record that transition(ci,x) is t
            add t to All (if not already present)
```

Intuitively...

- a ci state represents the collection of all the derivations that are equivalent in terms of what we can do with the top of the stack (how we got to the current top-of-stack can affect what we want to do with it next, hence the need)
- goto identifies out where we can transition to next, based on the state of the top of the stack and the coming input
- closure identifies out all the derivations that are equivalent to the ones for our current state

Brutal to do by hand

- even for simple grammars, applying goto and closure in the algorithm is very detailed/error prone/long to do by hand
- however, it is ideally suited for automation, and enables fast LR(1) parsing from the generated tables (as long as the language is actually in LR(1) of course)