PARSER ALGORITHMS
=================
Robert Heckendorn
Computer Science
University of Idaho

TOP DOWN PARSER ALGORITHMS
-----------------------------

Ad Hoc Algorithm

Build a recognizer *procedure* for every production.
Avoid using left recursive grammar or infinite recursion in parser will happen.

LL(1) Algorithm

Use a table M index by nonterminal on top of the stack and terminal on input: M(T, N)

Start with the goal nonterminal.

if top of stack is nonterminal then
  if M(input, top of parse stack) is a production then
    replace the token on the top of the stack with the rhs of the production
  else
    error
else
  if input==top of stack then
    pop stack
  else
    error

BOTTOM UP PARSER ALGORITHMS
-----------------------------

LR(0) Algorithm

// shift (shifts from input and so can only shift a terminal)
if state contains an item: A ::= B.XC where X is a terminal then
  if top(input) is X then
    shift X -> stack
    push state containing A ::= BX.C and set state to this state
  else
    ERROR
end if
// reduce (reduces the stack. No input is consumed.)
if state contains a complete item A ::= X, then
    pop the list of symbols X and their states
    state is now the new top(stack) // we set the state twice in a reduce
// state must have a state B:=C.AD
    push A on the stack // the case of pushing a nonterminal happens here
    push the state B:=CA.D // this state is the "go to" in the LR parse table
    set state to state containing B:=CA.D // this is the second time we set the state
end if

Note:
1. Reduce and Shift cannot occur in the same state
2. Only one of the two cases must actionable for any properly constructed DFA
3. Reduce only affects the parse stack

Shift/Reduce Conflict:
if both a reduce and a shift pattern exist in the same state

Reduce/Reduce Conflict:
multiple reductions because a state has more than one complete item

-----------------------------------------------------------------------------

SLR(1) algorithm

// shift
if state contains an item: A ::= B.XC where X is a terminal then
    if top(input) is X then
        shift X
        push state containing A ::= BX.C and set state to this state
    else
        try the reduce part // this is different from LR(0)!!!
    end if
end if

// reduce
if state contains a complete items A ::= X., B::=Y., C::=Z. then
    // this next decision is different from LR(0)!!!
    chose A if input \isin Follow(A), B if input \isin Follow(B), etc
    pop the list of symbols X and their states
    set state to top(stack)
// state must have a state B:=C.AD
    push A on the stack
    push the state B:=CA.D
    set state to state containing B:=CA.D
end if

Note:
1. uses LR(0) items (dot-thingys)
2. this algorithm gives us the option to not pop off the element from the stack but if it doesn't work and instead try a reduce.
3. this algorithm lets us have both shift and reduce in the same state.
4. This algorithm lets us have multiple reduces in the same state as long as they have different follow sets.

Shift/Reduce Conflict: if both a reduce and a shift pattern exist in the same state *plus* terminal X for the shift (A::=B.XC) is also in the follow set for B::=Z.

Reduce/Reduce Conflict: multiple reductions because a state has more than one complete item *plus* the follow sets for two reduces are not disjoint. (they share a terminal)

---

LR(1) Algorithm

// shift
if state contains an item: \([A ::= B.XC, z]\) where X is a terminal then
  if top(input) is X then
    shift X
    push state containing \([A ::= BX.C, z]\) and set state to this state
  else
    try the reduce part
  end if
// reduce
if state contains a complete items \([A ::= X., u]\), \([A::=X., v]\), \([B::=Y., w]\) then
  chose A if top(input) \(\in\) u etc. // choose between reduces by lookahead symbol
  pop the list of symbols X and their states
  set state to top(stack)
// state must have a state \([B:=C.AD, z]\)
  push A on the stack
  set state to state containing \([B:=CA.D, z]\)
end if

Note:
1. uses LR(0) items augmented with look ahead symbols called LR(1) items

Shift/Reduce Conflict: if both a reduce and a shift pattern exist such that for terminal X for the shift \([A::=B.XC, a]\) is also in the follow set for \([B::=Z,. X]\)

Reduce/Reduce Conflict: multiple reductions because a state has more than one complete item such that \([A::=X,. a]\) and \([B::=Y,. a]\)
LALR(1) Algorithm

1. Is almost the same as the LR(1) algorithm but with a reduced parse table size and the possibility of a reduce/reduce conflict.

2. It has the same size state machine as the SLR(1) parser.