Informal Bottom-Up Parsing Algorithm with Example

* in bottom-up parsing, we start at the input string (which is a sequence of terminals in the grammar) and make our way towards the start symbol (opposite of top-down parsing)
  + a derivation is a sequence α0α1 · · · αn such that α0 = S (the start symbol), αn = x (the input string), and αi ⇒ αi+1 for 0 ≤ i < n
  + find the derivation in reverse so we begin at x, find αn−1, then αn−2, etc.
  + at each step, we will replace some β in αi with A, where A → β is a rule in the grammar, to obtain αi−1
* algorithm is:
  + begin reading input symbols one character at a time, left to right
  + if we recognize the RHS of a rule, replace it with its LHS
* algorithm accepts input when there’s no more unread input left and the stack contains just the start symbol
* use stack to track each αi
  + shift consumes next input symbol and pushes it onto the stack
  + reduce pops the RHS rule off the stack and pushes its LHS
    - includes rules where RHS is empty
* e.g.

Text

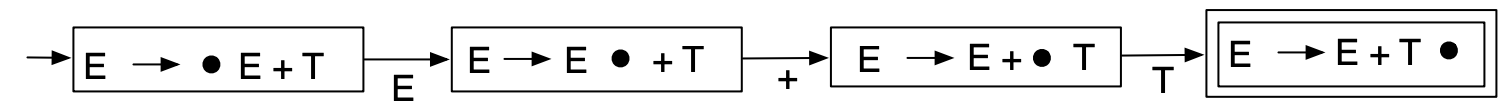
Description automatically generated

Table

Description automatically generated

A Less Informal Bottom-Up Parsing Algorithm

* an item is a production with a bookmark (represented as a •) somewhere on the RHS of a rule
  + e.g. an item E → • E + T is called a fresh item, indicating that none of the RHS is on the stack
    - if an E was pushed on the stack, update the fresh item to get the new item, E → E • + T
    - if a + and then a T were pushed on the stack the updated item, E → E + T •, is called reducible since the bookmark indicates the entire RHS of the rule is on the stack
  + treat positions of bookmark as states of a DFA and transition between states on the symbol that gets pushed on the stack



* to create DFA that updates items for all rules in a grammar:
  + create a start state with a fresh item for the single rule for the start symbol
  + select a state qi that has at least one non-reducible item and for each non-reducible item in qi, create a transition to a new state qj on the symbol X that follows the bookmark; take all items from qi where the bookmark is followed by an X, update the bookmark to be right after X and add them as items in qj
    - for each item in the newly created states, if the symbol following the updated bookmark is a non-terminal, say A, add fresh items for all rules for the non-terminal A to the new state
    - if this creates fresh items where a bookmark is followed by another non-terminal, say B, add fresh items for all rules of B
    - repeat if necessary
  + repeat previous step until no new states are discovered
  + mark states containing reducible items as accept states
* above DFA is called the LR(0) Parsing DFA
  + L stands for left to right scan of input
  + R is for rightmost derivations
  + 0 is number of symbols we look ahead to make a decision
* run the contents of the stack through the LR(0) DFA and see which state we end up in
  + if we end up in a state with a reducible item (an accepting state), we will reduce using the rule for the reducible item
  + if not, we will shift the next input symbol
* e.g.

A picture containing table

Description automatically generated

* it is inefficient; at each stage, we run the contents of the entire stack through the LR(0) DFA to determine what the next action should be
* previous stack will be called symbol stack and we will maintain a new stack called the state stack, where we push/pop state numbers
  + updates to the symbol stack are made exactly as before but each time we update the symbol stack, we make corresponding updates to the state stack

Graphical user interface, text, application

Description automatically generated

* + algorithm begins with the state stack containing the start state
  + queries the top of the state stack to determine if it is a reduce state
    - if it is not, we shift/push the next symbol on to the symbol stack and at the same time push on to the state stack the new state we end up in
    - if it is, we pop n times from both the symbol and state stack (n is the number of symbols on the RHS of the rule we are reducing with) n
      * note that n could be zero if the right-hand side is the empty string
      * push the LHS of the rule onto the symbol stack; to push the corresponding state on the state stack, we query the new top of the state stack and use that along with the symbol we just pushed to determine the new state to push

Action Conflicts

* shift-reduce conflict occurs when a state in the parsing DFA has two items of the form A → α • a β and B → γ •
  + e.g.

Diagram

Description automatically generated

* + - in state 5, there’s one reducible item and one non-reducible item
    - the algorithm doesn’t look at the next input symbol so it has no way of knowing whether to shift or reduce
* reduce-reduce conflict occurs when a state in the parsing DFA has two items of the form A → α • and B → β •
* having 2 shifts (even on diff symbols) is not a conflict because the action of “shifting” doesn’t care about what input symbol it’s shifting in
* a grammar is LR(0) if and only if the LR(0) automaton does not have any shift-reduce or reduce-reduce conflicts

Using Lookaheads

* SLR(1) (simplified LR(1)) uses one symbol of lookahead and makes a decision to reduce based on whether the lookahead symbol is in the Follow set for the LHS non-terminal
* in SLR(1), every reducible item will also include the Follow set for the LHS non-terminal to the item as the lookahead
  + algorithm will reduce using the reducible item only if the next input symbol is in the Follow set of this reducible item
* e.g. for the above grammar, updated right-associative SLR(1) automaton:

Diagram

Description automatically generated

* SLR(1) fails in the following grammar:

Diagram

Description automatically generated

* + state 5 has two reducible items where their follow sets intersect (both contain -|)
  + if we have input string |- ID -|:
    - algorithm begins by shifting |- and then ID on the symbol stack, which gets us to state 5
    - at this point, the look ahead is -| and the algorithm would be unable to decide which of the two reducible rules in state 5 to use
* LR(1) DFA is created by only adding the lookahead that should be present for a particular rule to be used in the reduce step (i.e. only adding a subset of the Follow set to each reducible item)
  + for the example above, we should only reduce using the rule S → ID if the next input symbol is -| and the rule E → ID if the next input symbol is =
  + although powerful, it’s hardly used in practice because number of states in the resulting LR(1) DFA can increase exponentially depending on the grammar
* Lookahead LR(1) or LALR(1) uses the LR(0) DFA but uses locally computed Follow sets to overcome the limitations of Follow sets (they’re global)
* below diagram rep power of diff parsing algorithms:

Diagram

Description automatically generated

* only change in algorithm from LR(0) to LR(1) is on line 3, where the decision on when to reduce is based on the current state at the top of the state stack and next input symbol a

Graphical user interface, text, application

Description automatically generated

Parse Trees

* to parse an input string:
  + if the algorithm decides to shift a terminal, it pushes onto the tree stack a node that rep that terminal
  + for a reduce, the algorithm pops the subtree nodes that represent the RHS of the rule and a new tree is created whose root is the LHS non-terminal and whose children are the nodes that were just popped
  + new tree is then pushed back on to the tree stack
* e.g. for below grammar and SLR(1) algorithm, parse is constructed as such:

Diagram

Description automatically generated

Table

Description automatically generated