Bottom-up parsing

We ended the last module having concluded that top-down parsing was ill-suited for left-associative grammars. Unfortunately, left-associative grammars include virtually all programming languages, so this restriction is quite severe. In this module, we discuss an alternate approach to top-down parsing: bottom-up parsing. Briefly, 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 for the grammar. Recall that a derivation is a sequence $\alpha_0\alpha_1 \cdots \alpha_n$ such that $\alpha_0 = S$ (the start symbol), $\alpha_n = x$ (the input string) and $\alpha_i \Rightarrow \alpha_{i+1}$ for $0 \leq i < n$. We will find the derivation in reverse, i.e., we begin at x, find α_{n-1}, then α_{n-2} and so on. At each step, we will replace some β in α_i with A, where A $\rightarrow \beta$ is a rule in the grammar, to obtain α_{i-1}.

1 Informal Bottom-Up Parsing Algorithm with Example

Since we start at the input string, the algorithm goes as follows: begin reading input symbols one character at a time, left to right. If we recognize the right-hand side of a rule (β in the description above), replace it with its left-hand side (A from above). Of course, we will need to track what each α_i is; we will use a stack for it. We define two actions for the stack:

shift: a shift consumes the next input symbol and pushes it on the stack

reduce: a reduce pops the right-hand side of a rule off the stack and pushes its left-hand side. (This includes rules where the right-hand side is empty, e.g. $A \rightarrow \epsilon$, in which case the end result is that the left-hand side non-terminal is pushed).

Let's begin with the same example we used for top-down parsing and parse the same input string:

(1) S' $\rightarrow$ $\vdash$ S $\dashv$
(2) S $\rightarrow$ AyB
(3) A $\rightarrow$ ab
(4) A $\rightarrow$ cd
(5) B $\rightarrow$ z
(6) B $\rightarrow$ wx

Suppose our input string s is:
$\vdash$ a b y w x $\dashv$.

An interesting thing to notice is that at any time we can obtain our current α_i by concatenating the stack and the unread input. Also, the algorithm accepts the input when there is no more unread input left and the stack must necessarily contain just the start symbol.

As before, the end goal was to obtain the derivation. The rules used during the reduction steps give the derivation. But remember, we are going in reverse! So, while the rules were Rule 3, 6, 2 and 1, the derivation is obtained by applying Rule 1, then 2, then 6 and 3, S' $\Rightarrow \vdash$ S $\dashv \Rightarrow \vdash$ AyB $\dashv \Rightarrow \vdash$ Aywx $\dashv \Rightarrow \vdash$ abywx $\dashv$. Another interesting observation is that our algorithm just produced a rightmost derivation; notice how each step of the algorithm expands the rightmost non-terminal. This is a natural consequence of finding the derivation steps in reverse while reading left to right.

The Bottom Up Parsing Algorithm video builds an intuition behind Bottom Up Parsing.

Read	Unread	Stack	Action
ϵ	$\vdash$ a b y w x $\dashv$		shift $\vdash$
$\vdash$	a b y w x $\dashv$	$\vdash$	shift a
$\vdash$ a	b y w x $\dashv$	$\vdash$ a	shift b
$\vdash$ a b	y w x $\dashv$	$\vdash$ a b	Reduce Rule 3; Pop b, a Push A
$\vdash$ a b	y w x $\dashv$	$\vdash$ A	shift y
$\vdash$ a b y	w x $\dashv$	$\vdash$ A y	shift w
$\vdash$ a b y w	x $\dashv$	$\vdash$ A y w	shift x
$\vdash$ a b y w x	$\dashv$	$\vdash$ A y w x	Reduce Rule 6; Pop x, w Push B
$\vdash$ a b y w x	$\dashv$	$\vdash$ A y B	Reduce Rule 2; Pop B, y, A Push S
$\vdash$ a b y w x	$\dashv$	$\vdash$ S	shift $\dashv$
$\vdash$ a b y w x $\dashv$	ϵ	$\vdash$ S $\dashv$	Reduce Rule 1; Pop $\dashv$, S, $\vdash$ Push S'
$\vdash$ a b y w x $\dashv$	ϵ	S'	Accept

2 A Less Informal Bottom-Up Parsing Algorithm

At each step of the algorithm, we need to either shift the next symbol from the input to the stack or reduce if the top of the stack is the right-hand side of a rule. The vagueness of our algorithm comes from the fact that we have not formalized how to determine when the top n symbols of the stack represent the right-hand side of a rule. There is a beautiful theorem dictating when to reduce. We begin with developing an intuition before discussing the theory. Consider the following grammar:

(1) $S' \rightarrow \vdash$ E $\dashv$
(2) E $\rightarrow$ E + T
(3) E $\rightarrow$ T
(4) T $\rightarrow$ ID

As an aside, notice this is the left-recursive grammar from the top-down parsing module which led us to claim that LL(1) parsing cannot handle left-recursive grammars. We use this grammar intentionally to highlight the power of bottom-up parsing.

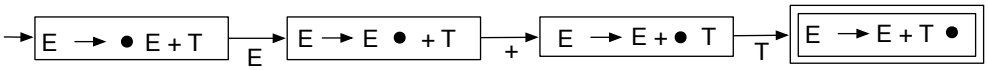
Let's pick rule 2 to discuss. Ask yourself the question, what needs to be on the stack for us to reduce using rule 2? The answer is that the top of the stack must be T, below it + and below it E. Let's create notation to track how much of the right-hand side of a rule is on the stack.

Definition 1 *An **item** is a production with a bookmark (represented as a $\bullet$) somewhere on the right-hand side of a rule.*

The item E $\rightarrow \bullet$ E + T is called a *fresh item*, indicating that none of the right-hand side is on the stack. If the algorithm was to push an E on the stack, we would update the fresh item to get the new item, E $\rightarrow$ E $\bullet$ + T. This tells us that E is on the stack. If we then push +, we get the item E $\rightarrow$ E + $\bullet$ T and finally, once T is pushed on the stack, we get the updated item E $\rightarrow$ E + T $\bullet$. This last item is *reducible*, since the bookmark indicates that the entire right-hand side of the rule is on the stack.

If the above was hard to follow, let's reword the idea; at the start state, none of the right-hand side for this rule is on the stack. Once we push E, we transition to a state where the bookmark had

been moved past E. Similarly, after pushing + and T, we transition the bookmark forward until we reach a state where the bookmark is at the end of the right-hand side. At this point, we are willing to accept that the right-hand side is all on the stack. Sound familiar? We are treating the positions of the bookmark as states of a DFA and are transitioning between states on the symbol that gets pushed on the stack. The diagram below captures this:



This DFA can be used to determine if the right-hand side of `this` rule is on the stack [1]. To track all rules at the same time, we could create such automaton for each rule and combine (glue) them using ϵ transitions to get an ϵ-NFA. While the ϵ-NFA could be used to recognize when a reducible item is on the stack, if a DFA is preferred, we could leverage the ϵ-NFA to NFA conversion algorithm and subsequently the NFA to DFA conversion algorithm to produce a DFA.

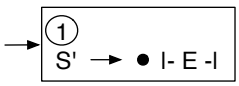
The following steps can be used as a shortcut to produce the DFA directly:

1. Create a start state with a fresh item for the single rule for the start symbol
2. Select a state q_i that has at least one non-reducible item. For each non-reducible item in q_i, create a transition to a new state q_j on the symbol X that follows the bookmark. Take all items from q_i where the bookmark is followed by an X, update the bookmark to be right after X and add them as items in q_j.
    - 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 non-terminal A to the new state. If this creates fresh items where a bookmark is followed by a non-terminal, say B, add fresh items for all rules of B. Repeat if necessary.
3. Repeat step 2 until no new states are discovered.
4. Mark states containing reducible items as accept states.

We show this DFA construction algorithm through a step-by-step construction below. We number each state so that the description can refer to the numbered states. Let's use the grammar discussed above (grammar reproduced for convenience):

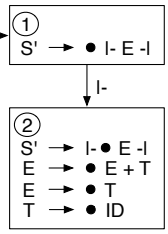
(1) $S' \rightarrow \vdash$ E $\dashv$
(2) E $\rightarrow$ E + T
(3) E $\rightarrow$ T
(4) T $\rightarrow$ ID

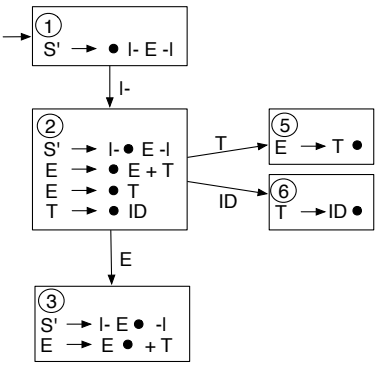
We begin by creating a fresh item for rule 1. This creates state 1 shown on the right.



[1] We will start to use rectangles for the states as opposed to circles like we previously did, primarily for aesthetics (circles labelled with items make for very big circles)

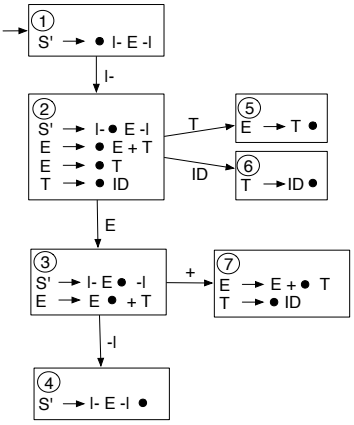
In step 2, we select state 1 since it has a non-reducible item. We create state 2. We have updated the item from State 1 to $S' \rightarrow \vdash \bullet \text{ E } \dashv$. Notice how, since the bookmark now precedes a non-terminal E in state 2, we added two fresh items, $\text{E} \rightarrow \bullet \text{ E} + \text{T}$ and $\text{E} \rightarrow \bullet \text{ T}$. Since this introduced a new fresh item where the bookmark is followed by a non-terminal (T), we added the fresh item for the rule for T, i.e., $\text{T} \rightarrow \bullet \text{ ID}$.

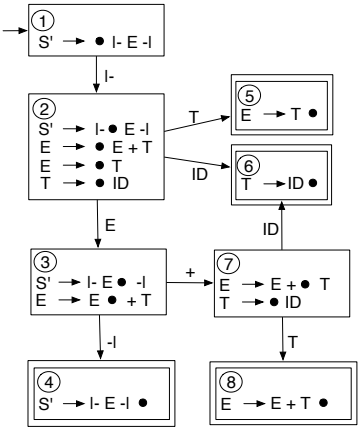




Still in step 2, we select state 2 since it has multiple non-reducible items. There are two items that have the bookmark right before the non-terminal E. We create a transition to state 3 on symbol E. In state 3, these two items have been updated by transitioning the bookmark past E. Also, since state 2 has a non-reducible item where the bookmark is followed by T, we create state 5 with the reducible item $\text{E} \rightarrow \text{T} \bullet$. Similarly, state 2 also had the non-reducible item, $\text{T} \rightarrow \bullet \text{ ID}$, this leads us to create state 6, with the reducible item $\text{T} \rightarrow \text{ID} \bullet$.

Still in step 2, the only state still to be processed is state 3 (since state 5 and 6 do not have non-reducible items). In processing state 3, we create state 4 that transitions the item $S' \rightarrow \vdash \text{E} \bullet \dashv$ to the reducible item $S' \rightarrow \vdash \text{E} \dashv \bullet$. Also, from state 3 we transition to state 7 which updates the item $\text{E} \rightarrow \text{E} \bullet + \text{T}$ to $\text{E} \rightarrow \text{E} + \bullet \text{ T}$. Since this has created an item where the bookmark is followed by a non-terminal, we add fresh items for the rules for that non-terminal. In this case, the fresh item $\text{T} \rightarrow \bullet \text{ ID}$ was added.





Still in step 2, this time we only have state 7 to process. We create state 8 by transitioning the T symbol to produce the reducible item $\text{E} \rightarrow \text{E} + \text{T} \bullet$ in state 8. For the other non-reducible item, $\text{T} \rightarrow \bullet$ ID, in state 7, we need a state with the item, $\text{T} \rightarrow \text{ID} \bullet$. We choose to reuse the existing state 6. Alternately, we could have created a new state 9 with this reducible item. There are no more new states that contain non-reducible items so step 2 and 3 are complete. Step 4 requires adding accept states to those states that have a reducible item. For our example, these are states 4, 5, 6 and 8.

The Tracking items in an automaton video discusses the construction of the automaton.

Let's summarize what we have so far: we are building a bottom-up parsing algorithm where, at each step of the algorithm, we have a choice between shifting the next input symbol to the stack or reducing the right-hand side of a rule from the stack and replacing it with its left-hand side. To determine when to reduce, we have introduced the terminology of *items*, which are a way of bookmarking how much of the right-hand side of a rule is on the stack. The algorithm above showed

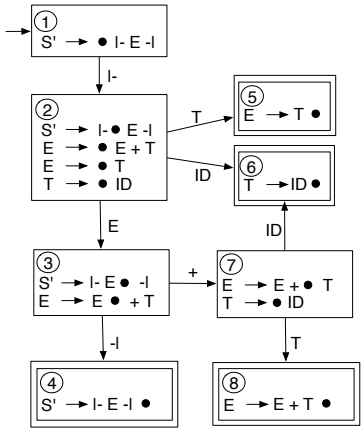
how to create a DFA that updates items for all rules in a grammar. The DFA we have constructed is called the LR(0) Parsing DFA, and the algorithm we are discussing is called the LR(0) parsing algorithm. The L stands for left to right scan of the input, the R for Rightmost derivations, and the 0 is the number of symbols we look ahead in the input to make a decision (note we aren't looking ahead at any input symbols to decide, for now).

The idea is that we would 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. Note that we've used the convention of marking reduce states as accepting states, but since both reducing and non-reducing states have an associated action (reducing or shifting, respectively), both are technically accepting states, and some formalisms will describe them this way. In this module, we will continue to describe only reduce states as accepting states.

2.1 A trace through the algorithm

Let's look at the first few steps of the algorithm in action. We will use the same grammar and LR(0) DFA for this grammar that we just constructed and parse the input string $\vdash$ ID + ID $\dashv$. The action column is determined by running the current contents of the stack through the LR(0) DFA and choosing the action based on the state we end up in:

(1) $S' \rightarrow \vdash$ E $\dashv$
(2) E $\rightarrow$ E + T
(3) E $\rightarrow$ T
(4) T $\rightarrow$ ID



Read	Unread	Stack	Action
ϵ	$\vdash$ ID + ID $\dashv$		Running stack through DFA results in state 1. Shift $\vdash$
$\vdash$	ID + ID $\dashv$	$\vdash$	Running stack through DFA results in state 2. Shift ID
$\vdash$ ID	+ ID $\dashv$	$\vdash$ ID	Running stack through DFA results in state 6. Reduce rule 4.
$\vdash$ ID	+ ID $\dashv$	$\vdash$ T	Running stack through DFA results in state 5. Reduce rule 3.
$\vdash$ ID	+ ID $\dashv$	$\vdash$ E	Running stack through DFA results in state 3. Shift +.
$\vdash$ ID +	ID $\dashv$	$\vdash$ E +	.... and so on.

We describe how to read the table above. In row 1, we have not read any input, and the stack has no symbols on it. Running the empty stack contents through the DFA means no transitions are taken, so we stay in state 1, the start state. Since there is no reducible item in this state, the decision is to shift the next symbol from the input ($\vdash$). In row 2, when we run the stack contents through the DFA, we start at state 1, and then transition on $\vdash$ to end up in state 2. Since state 2 does not have a reducible item, the decision is to shift the next input symbol (ID). In row 3, we first

transition on $\vdash$ to get to state 2 and then transition on ID to get to state 6. This has a reducible item, so we reduce using the rule T $\rightarrow$ ID; ID gets popped and T gets pushed. In row 4, we first transition on $\vdash$ to state 2 and then transition on T to state 5. And so on.

We did not complete the parse above as 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. It is worth making the observation that actions (shift or reduce) only affect the top few elements on the stack. In the case of a shift, a single new symbol is pushed on the stack. In the case of a reduce, some number of symbols n are popped (corresponding to the number of symbols on the right-hand side of the rule we are reducing by) and then a single symbol gets pushed. A more efficient approach would be to keep track of which DFA state we would end up at with the current contents of the stack. Of course, as we pop and push symbols from the stack, the state would need to be updated, which would require knowing what the previous state was. For example, if the stack contains $\vdash$ ID, not only do we need to keep track that this stack configuration gets us to state 6, we also need to know that we got to state 6 via state 2 to which we got to through state 1. In other words, we need to maintain the sequence of states that are visited as we run the stack contents through the DFA.

This is easy enough to implement; let's maintain *another* stack, where we push/pop state numbers [2]. Since we are now dealing with two stacks, we will call our previous stack the symbol stack. The new one, where we push states, we will call the state stack. The two stacks are kept in sync; 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. The algorithm begins with the state stack containing the start state. The algorithm simply queries the top of the state stack to determine if it is a reduce state, i.e., whether it contains a reducible item. If it does 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, using the current top of the state stack and the symbol we just pushed on to the symbol stack. If at any time, the top of the state stack tells us that we are at a reduce state, we pop n times from both the symbol and state stack, where n is the number of symbols on the right-hand side of the rule we are reducing with. Note that n could be zero if the right-hand side is the empty string. We push the left-hand side 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. This is more efficient than our previous approach since we no longer have to run the contents of the symbol stack through the DFA every time we want to decide which action to take; the top of the state stack tells us what to do. The above description of the algorithm is formalized below. In the pseudocode, `Reduce[state]` is a function that, given a state, tells us whether this is a reduce state and, if yes, it gives the rule that is applicable.

[2] From an implementation perspective it makes sense to have separate stacks for symbols and states. From a theoretical standpoint, you could imagine that the same stack can be used to push and pop both symbols and states.

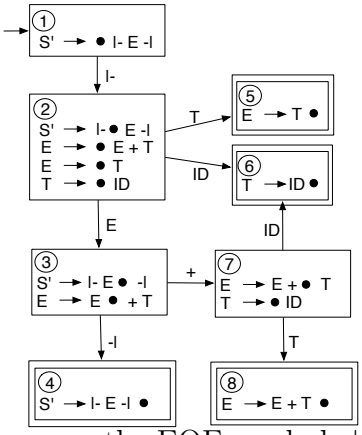
Algorithm 1 LR(0) algorithm, input LR(0) DFA(Σ,Q,q_0,δ,A)

```
1:  stateStack.push q_0
2:  for each symbol a in ⊢ x ⊣ from left to right do
3:    while Reduce[stateStack.top] is some production B → γ do
4:      symStack.pop symbols in γ
5:      stateStack.pop |γ| states
6:      symStack.push B
7:      stateStack.push δ[stateStack.top, B]
8:    end while
9:    symStack.push a
10:   reject if δ[stateStack.top, a] is undefined
11:   stateStack.push δ[stateStack.top, a]
12: end for
13: accept
```

Notice line 10. If the top of the state stack tells us that we cannot reduce, we should try to shift, but this can cause a parse error if the LR(0) DFA does not define a transition from the current state on the next input symbol (a in the algorithm).

We show the complete derivation for the earlier example below:

(1) $S' \rightarrow \vdash$ E $\dashv$
(2) E $\rightarrow$ E + T
(3) E $\rightarrow$ T
(4) T $\rightarrow$ ID



An interesting thing to note is that technically as soon as the EOF symbol $\dashv$ is pushed, it indicates a successful derivation. The algorithm could choose to stop then, i.e., as soon as the right-hand side of the rule for the grammar's start symbol is on the stack.

The Parsing using the LR(0) algorithm video traces the LR(0) algorithm for an input string.

For the grammar and LR(0) DFA discussed above, show that $\vdash$ + $\dashv$ will cause a parser error.
[3]

2.2 Optional: LR(0) Formalism

Above, we discussed how we can use the LR(0) DFA to guide when to reduce and which rule to reduce with. This stems from a theorem by Knuth (*On the Translation of Languages from Left*

[3] The answer is found in Section 7.1 at the end of this module.

Read	Unread	Symbol Stack	State Stack	Action
ϵ	$\vdash$ ID + ID $\dashv$		1	State 1 is a shift state. Shift $\vdash$
$\vdash$	ID + ID $\dashv$	$\vdash$	1 2	State 2 is a shift state. Shift ID
$\vdash$ ID	+ ID $\dashv$	$\vdash$ ID	1 2 6	State 6 is a reduce state. Reduce rule 4
$\vdash$ ID	+ ID $\dashv$	$\vdash$ T	1 2 5	State 5 is a reduce state. Reduce rule 3
$\vdash$ ID	+ ID $\dashv$	$\vdash$ E	1 2 3	State 3 is a shift state. Shift +
$\vdash$ ID +	ID $\dashv$	$\vdash$ E +	1 2 3 7	State 7 is a shift state. Shift ID
$\vdash$ ID + ID	$\dashv$	$\vdash$ E + ID	1 2 3 7 6	State 6 is a reduce state. Reduce rule 4
$\vdash$ ID + ID	$\dashv$	$\vdash$ E + T	1 2 3 7 8	State 8 is a reduce state. Reduce rule 2
$\vdash$ ID + ID	$\dashv$	$\vdash$ E	1 2 3	State 3 is a shift state. Shift $\dashv$
$\vdash$ ID + ID $\dashv$	ϵ	$\vdash$ E $\dashv$	1 2 3 4	State 4 is a reduce state. Reduce rule 1
$\vdash$ ID + ID $\dashv$	ϵ	S'		Accept as soon as we push S'

to Right, 1965) which we discuss very briefly for the interested reader. Those who feel lost in the formalism can skip the text. Those who wish for a little more are recommended to take CS 444. Those wishing for a lot more can look at CS462.

Definition 2 *Given a grammar G = (N,T,P,S), γ is a sentential form if $S \Rightarrow^* \gamma$.*

Note that γ is not necessarily a string of terminals; it is a form that can be derived by repeated application of rules starting at the start symbol. It may include both terminals and non-terminals.

Definition 3 *We say α is a viable prefix if it is the prefix of a sentential form and the remainder of the sentential form after α is terminals, i.e., $\exists$ some string of terminals y s.t. $S \Rightarrow^* \alpha y$.*

In words, a viable prefix when concatenated with some string of terminals forms a sentential form, i.e., it can be derived from the start symbol. The crux of Knuth's theorem is that the set of `viable prefixes`, configurations that have the viability to lead to a successful derivation, is itself a regular language. The DFA we just constructed is a representation of this regular language, i.e., the DFA contains states that represent viable prefixes. By tracing the contents of the stack through the DFA, and only performing actions based on the DFA, our stack contents will continue to be viable prefixes.

The crux of LR parsing is in making the decision on when the stack should be reduced. Consider the situation where the stack contains α. Suppose the top symbols of α form γ, i.e., $\alpha = \beta\gamma$ and that A $\rightarrow \gamma$ is a rule in the grammar. The question is, should we reduce using this rule? If we do reduce, the stack will become βA (since γ will be popped and A would be pushed). We should do the reduction if βA is a viable prefix. This is exactly what the LR(0) DFA tells us. If we were to pass the contents $\beta\gamma$ through the DFA, we would end up in a state which contains the reducible item A $\rightarrow \gamma \bullet$. The rules that can be used to reduce the stack for a given stack configuration α can be written as a set:

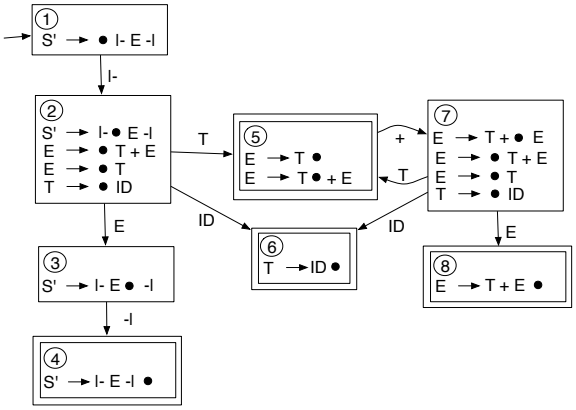
Reduce(α) = { A $\rightarrow \gamma : \alpha = \beta\gamma$ and βA is a viable prefix }

In other words, this set tells us that if α is on the stack, we should reduce using one of the rules in this set, since the resulting stack configuration will be a viable prefix. Note that the occurrence of more than one rule in this set can be problematic for a parsing algorithm, and we address this in the coming sections.

3 Action Conflicts

We have learned how to construct the LR(0) parsing DFA and the LR(0) parsing algorithm that uses this DFA to perform bottom-up parsing. While many grammars can be parsed using this algorithm, there exist grammars that are not LR(0) (meaning they cannot be parsed by the LR(0) algorithm). We discuss one such example and then define *parsing conflicts*. Consider the following grammar and the associated LR(0) DFA. Notice that the grammar is similar to the grammar above except that it is right-recursive in rule 2.

(1) $S' \rightarrow \vdash$ E $\dashv$
(2) E $\rightarrow$ T + E
(3) E $\rightarrow$ T
(4) T $\rightarrow$ ID



On a first look, the LR(0) DFA for the grammar does not seem to have any problems. But if you look closer, there is a major issue. Consider state 5. State 5 has two items, one of which, E $\rightarrow$ T $\bullet$, is reducible. According to our algorithm, if we have a reducible item at a state, we should reduce using the rule for that item. But state 5 also has a non-reducible item, E $\rightarrow$ T $\bullet$ + E, which suggests that shifting the next symbol is also a feasible choice. The algorithm we have discussed will not be able to decide which of the two actions is the right decision, since it does not look at the next input symbol. If the next input symbol is a +, then of course shifting the + makes sense. However, if the next input symbol is $\dashv$, then shifting $\dashv$ does not make sense, as we would have no transition on $\dashv$ in the DFA and this would generate a parse error. The conflict in this DFA is called a *shift-reduce conflict*.

A **shift-reduce** conflict occurs when a state in the parsing DFA has two items of the form A $\rightarrow \alpha \bullet$ a β and B $\rightarrow \gamma \bullet$ where, as always, $a \in T'$ is a terminal and $\alpha, \beta, \gamma \in V^*$ are words over the vocabulary.

The issue is that, given this state, the algorithm cannot decide whether to shift as suggested by the non-reducible item, A $\rightarrow \alpha \bullet$ a β, or reduce using the reducible item, B $\rightarrow \gamma \bullet$.

Another possible conflict is called a *reduce-reduce conflict*, where two reducible items exist within the same state:

A **reduce-reduce** conflict occurs when a state in the parsing DFA has two items of the form A $\rightarrow \alpha \bullet$ and B $\rightarrow \beta \bullet$ where $\alpha, \beta \in V^*$.

Note that having two shifts, even on different symbols, is not a conflict, as the action is simply "shift". Having two reduces is a conflict because the algorithm needs to make a *specific* reduction.

Definition 4 *We say that a grammar is LR(0) if and only if the LR(0) automaton does not have any shift-reduce or reduce-reduce conflicts.*

Based on this definition, the right recursive grammar discussed above is not LR(0). In the next section, we discuss how we can increase the power of bottom-up parsing algorithms by looking ahead at the next input symbol.

We used lookahead in top-down parsing as well, but in top-down parsing, the algorithm was useless without it: without lookahead, *all* non-terminals with at least two productions needed a lookahead to distinguish them. In bottom-up parsing, this decision is being made much later (*after* reading all the symbols on the right-hand side of a production, in the case of reduce), and as a consequence, LR(0) is still useful, and the cases where lookahead is needed are more specific. Nonetheless, the grammars of most programming languages require lookahead even in bottom-up parsing.

4 Using Lookaheads

In top-down parsing, we began with a single lookahead, in LL(1). We can do the same for LR parsing. However, while LR(1) is very powerful, it has some disadvantages that we discuss later that make it impractical to use. Before going for this nuclear option, there are a number of compromises that can be made.

Simplified LR(1), or SLR(1) for short, uses one symbol of lookahead and makes a decision on whether to reduce based on whether the lookahead symbol is in the Follow set for the left-hand side non-terminal. Before going any further, let's recall our definition of Follow(A) (which we covered in the top-down parsing module):

Follow$(A) = \{b \in T' : S' \Rightarrow^* \alpha A b \beta$ for some $\alpha, \beta \in V^*\}$, i.e., Follow(A) is the set of terminals that can come immediately after A in a derivation starting at the start symbol S'.

For the grammar under discussion, Follow(T) = { +, ⊣ } and Follow(E) = { ⊣ }.

In SLR(1), we extend the LR(0) DFA by extending our definition of *items*: for each reducible item, add the Follow set for the left-hand side non-terminal to the item as the *lookahead*. In our example, this creates the items E → T + E • : {⊣}, E → T • : {⊣} and T → ID • : {+,⊣}. This isn't needed for non-reducible items (shift items), as the lookahead is evident from the item: it's the terminal that comes after the bookmark.

In Figure 1, we show the SLR(1) automaton (where each reducible item other than that for the start symbol has the Follow set for the left-hand side non-terminal added to the item).

Take a look at State 5. Previously, in the LR(0) DFA, we had a shift-reduce conflict. With SLR(1), we have added the Follow set for E to the reducible item. The SLR(1) algorithm will reduce using the reducible item only if the next input symbol is in the Follow set of this reducible item. Notice how this eliminates the conflict. If the next input symbol is ⊣, i.e., we are done with our input string, we will reduce using the rule E → T. However, if the next input symbol is not ⊣, we will shift the next input symbol. Of course, if the next input symbol happens to be something other than +, this would generate a parse error.

A video showing a grammar that is not LR(0) but is SLR(1) is available.

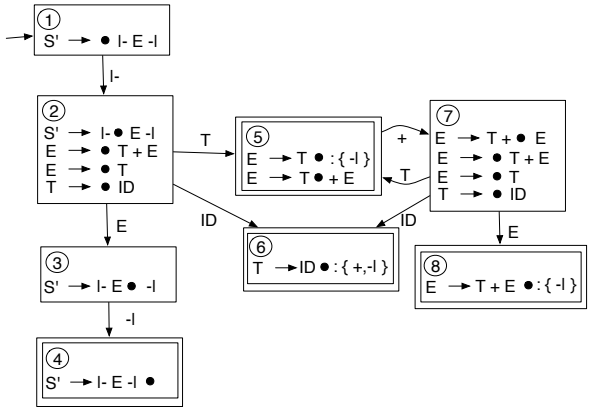


Figure 1: Right-associative SLR(1) automaton

Create the SLR(1) DFA for this grammar. Are there any Reduce-Reduce conflicts? Are there any Shift-Reduce conflicts? Is the grammar SLR(1)?

(1) $S' \rightarrow \vdash$ S $\dashv$
(2) S $\rightarrow$ Aa
(3) S $\rightarrow$ bAc
(4) S $\rightarrow$ dc
(5) S $\rightarrow$ bda
(6) A $\rightarrow$ d

[4]

4.1 Optional: Formally defining LR(1) and SLR(1)

Once again, we briefly discuss the formal theory behind these parsing algorithms for the interested readers. This section can be skipped without loss of continuity. In our previous optional section, we formalized LR(0) parsing and defined the `Reduce` set as, Reduce(α) = { A $\rightarrow \gamma : \alpha = \beta\gamma$ and βA is a viable prefix }. This set of rules are the rules that can be used to reduce α, the current contents of the stack. Since then, we have talked about Reduce-Reduce conflicts. A Reduce-Reduce conflict occurs if the `Reduce` set has more than one rule for any given α. That is, a grammar is not LR(0) if there exists an α such that $|\texttt{Reduce}(\alpha)| > 1$.

How does LR(1) work? In LR(1) we strengthen our definition of `Reduce` to be:

Reduce(α, a) = { A $\rightarrow \gamma : \alpha = \beta\gamma$ and βAa is a viable prefix }.

The a is the look ahead symbol. In other words, now the Reduce set contains the rules we should use to reduce given that we know that a is the next input symbol and with the condition that after the reduction βAa continues to be a viable prefix. Notice that this is a stronger condition than that for LR(0) parsing, which did not take into account the next input symbol. A strong condition implies that fewer rules will satisfy the condition and fewer applicable rules can result in fewer

[4] The answer is found in Section 7.2 at the end of this module.

chances of conflicts.

What does SLR(1) do? SLR(1) is a compromise between LR(0) and LR(1). Instead of checking that βAa is a viable prefix, we define the SLR(1) `Reduce` set to be:

Reduce(α, a) = { A $\rightarrow \gamma : \alpha = \beta\gamma$ and βA is a viable prefix and a $\in$ Follow(A) }.

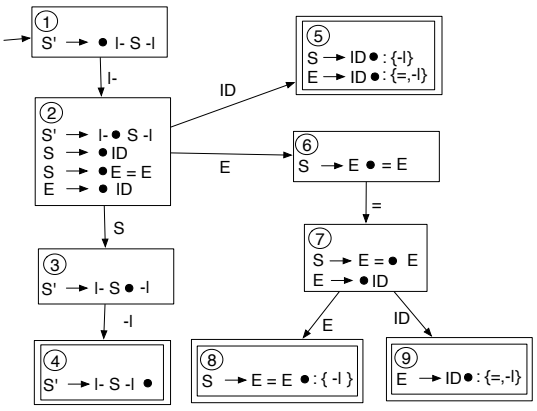
In other words, we now add the condition that we will only consider reducing using a rule if the condition for LR(0) holds as well as that the next input symbol is in the Follow set of the non-terminal whose rule we are considering. Note that this is a heuristic. If we know that βAa is a viable prefix (LR(1) requirement), this implies that βA is a viable prefix and a $\in$ Follow(A). However, the reverse is not necessarily true, i.e., knowing that βA is a viable prefix and a $\in$ Follow(A) does not guarantee that βAa is a viable prefix. Practically speaking, this means that, compared to the LR(1) Reduce set, the SLR(1) Reduce set is more likely to incorrectly tell you to reduce when shifting is the right thing to do.

4.2 SLR(1), LALR(1) and LR(1)

We have discussed how the SLR(1) parsing DFA can be created: by adding the follow sets of the left-hand side non-terminal for each reducible item in the LR(0) DFA. However, using follow sets to determine whether we should reduce is a heuristic; a good heuristic, but nevertheless a heuristic. As an example to see where SLR(1) fails, consider the following grammar and the corresponding SLR(1) DFA:

$S' \rightarrow \vdash$ S $\dashv$
S $\rightarrow$ ID
S $\rightarrow$ E = E
E $\rightarrow$ ID

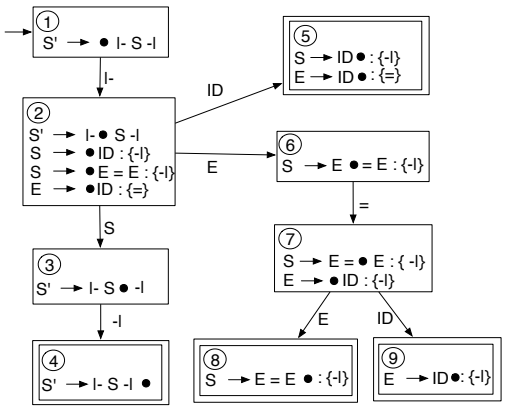
Follow(S) = $\{\dashv\}$
Follow(E) = $\{=, \dashv\}$



Consider state 5, which has two reducible items. Since this is the SLR(1) DFA, each of these reducible items contains the follow sets of the left-hand side non-terminal. However, notice that the intersection of these two follow sets is not empty, i.e., follow(S) and follow(E) both contain the terminal $\dashv$. This poses a problem. Consider the input string $\vdash$ ID $\dashv$. The algorithm begins by shifting $\vdash$ and then ID on the symbol stack. This gets us to state 5. At this point, the look ahead is $\dashv$. The algorithm would be unable to decide which of the two reducible rules in state 5 to use, since the lookahead is in the follow sets of both rules. Of course, we can see that reducing using E $\rightarrow$ ID is a bad idea, since this would take us to state 6 and the next input symbol being $\dashv$ would result in a parse error. This example highlights the limitation of SLR(1): just because a symbol is in the follow set of a non-terminal does not mean that there is a viable derivation if we rely on just that information. This example shows a reduce-reduce conflict, but note that shift-reduce conflicts in SLR(1) are also possible.

The 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 $\rightarrow$ ID if the next input symbol is $\dashv$. Similarly, and perhaps more importantly, we should only reduce using the rule E $\rightarrow$ ID, if the next input symbol is =. While we do not discuss the details of the construction of the LR(1) DFA on purpose, it uses First sets to determine what lookahead to use and tags this information to all items not just reducible items. Below we show the LR(1) DFA for the above grammar. We do not expect the reader to know how to construct such machines in CS241.

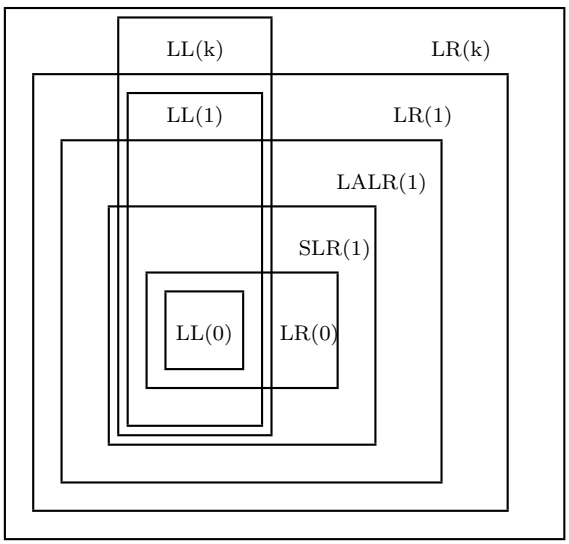
Recall that in the SLR(1) DFA for this grammar, state 5 continued to have a Reduce-Reduce conflict. Consider the same state in the LR(1) DFA to the right. There is no longer a conflict. The state indicates that the rule S $\rightarrow$ ID will be used to reduce if the look ahead is $\dashv$ and the rule E $\rightarrow$ ID will be used if the lookahead is =.



While LR(1) is very powerful, it is hardly ever used in practice. The primary reason is that the number of states in the resulting LR(1) DFA can increase exponentially depending on the grammar.

While many prefer to use SLR(1) instead of LR(1), an even better compromise is to use a construction for the DFA called LALR(1), Lookahead LR(1). Very briefly (and vaguely), LALR(1) attempts to overcome the limitations of Follow sets (in that they are global) by using the LR(0) DFA but associating with it locally computed Follow sets. The grammar above is LALR(1).

The following diagram concisely represents the power of the different parsing algorithms discussed in this and the previous module. Each "box" represents the set of grammars that can be parsed using the parsing algorithm that box is annotated with. For example, the LR(0) box represents all grammars that can be parsed using an LR(0) parsing algorithm.



Apart from depicting the obvious increasing strengths of the different LR parsing algorithms, some interesting observations include:

- Grammars that can be parsed by an LL(1) algorithm can also be parsed by an LR(1) algorithm.
- There exist grammars that can be parsed by an LL(1) algorithm that cannot be parsed by an SLR(1) or LALR(1) algorithm.

We have on purpose covered the differences at a superficial level, since the details are beyond the scope of this course. In particular, while we have discussed in detail the construction of the LR(0) and SLR(1) parsing DFA, we have not discussed how the LALR(1) and LR(1) parsing DFAs are constructed. In practice, SLR(1) and LALR(1) are "good enough" for most programming languages. For example, GNU Bison is a parser generator that, given a context free language, will generate a parser for the context free language. It defaults to LALR(1) although that can be changed.

One interesting thing to note is that irrespective of how the parsing DFA is constructed, the same algorithm can be used to achieve the parse. We give a modified version of our LR(0) algorithm that can be used with SLR(1), LALR(1) or LR(1) DFAs. The modification only requires a slight update from our previous algorithm that did not use lookaheads to make decisions.

Algorithm 2 LR(1) algorithm, input SLR(1) or LALR(1) or LR(1) DFA(Σ,Q,q_0,δ,A)

```
1:  stateStack.push q_0
2:  for each symbol a in ⊢ x ⊣ from left to right do
3:    while Reduce[stateStack.top, a] is some production B → γ do
4:      symStack.pop symbols in γ
5:      stateStack.pop |γ| states
6:      symStack.push B
7:      stateStack.push δ[stateStack.top, B]
8:    end while
9:    symStack.push a
10:   reject if δ[stateStack.top, a] is undefined
11:   stateStack.push δ[stateStack.top, a]
12: end for
13: accept
```

Note the **one** change in the algorithm, on line 3: the decision on when to reduce is based on the current state at the top of the state stack **and** the next input symbol a.

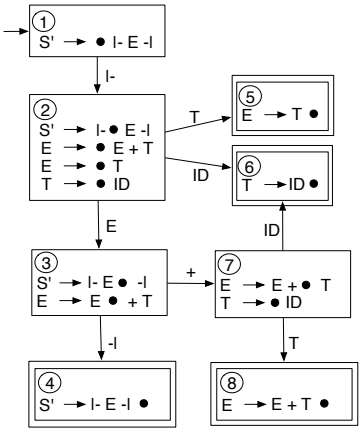
5 But Where is my Parse Tree

We first talked about parsing in the module on context-free grammars and discussed how the output of a parser is a parse tree. Further, we had discussed how a derivation uniquely defines a parse tree. Since then, our algorithms have mostly focused on obtaining the derivation. This is fine since given a derivation, a parse tree is easy to obtain. However, it is worth noting that practical parser implementations do not treat the creation of the parse tree as a two-step process of first obtaining the derivation and then using the derivation to create the parse tree. Instead, parsers create the

parse tree during the execution of the parsing algorithm. Here is how it is done.

Recall, that the symbol stack we have used in our parsing algorithms contained symbols (terminals, non-terminals). We can change this stack to be a stack of tree nodes! In other words, instead of pushing and popping symbols, we will be pushing and popping leaf nodes, subtrees and eventually the full parse tree. We illustrate the implementation through the same example we used earlier:

(1) $S' \rightarrow \vdash$ E $\dashv$
(2) E $\rightarrow$ E + T
(3) E $\rightarrow$ T
(4) T $\rightarrow$ ID



The table shown on the next page should be self explanatory. If the algorithm decides to shift a terminal, it actually pushes onto the tree stack a node that contains/represents that terminal. For a reduce, the algorithm pops the subtree nodes that represent the right-hand side of the rule. A new tree is created whose root is the left-hand side non-terminal and whose children are the nodes that were just popped. This new tree is then pushed back on to the Tree Stack.

A video Constructing the Parse Tree during LR(1) parsing is available.

(1) $S' \rightarrow \vdash$ S $\dashv$
(2) S $\rightarrow$ AyB
(3) A $\rightarrow$ ab
(4) A $\rightarrow$ cd
(5) B $\rightarrow$ z
(6) B $\rightarrow$ wx

Create the LR(0) DFA.

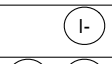
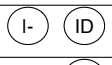
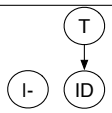
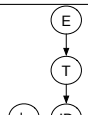
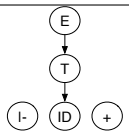
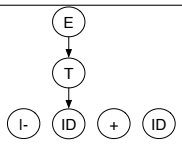
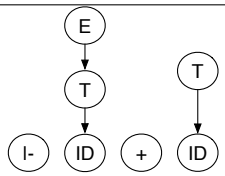
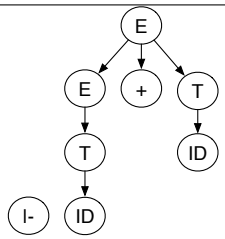
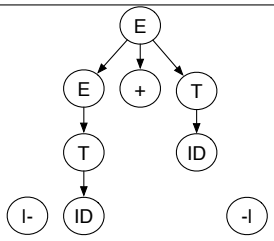
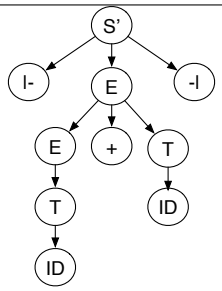
Given the input string: $\vdash$ a b y w x $\dashv$, give a step by step construction of the parse tree showing `Read`, `Unread`, `Tree Stack`, `State Stack` and `Action` columns like the examples in this module.

[5]

6 Looking Ahead

In this module, we have discussed a powerful parsing technique called bottom-up parsing that takes a sequence of tokens as input and checks that the input matches the syntax of the language. If the syntax of the input is indeed valid, we now have the ability to produce a parse tree for the input that gives us its structure. However, not all rules in a programming language can be enforced through a context-free grammar. Things like checking that a variable used has been previously defined, and a value assigned to a variable has the right type, requires context-**sensitive** analysis, where context *does* matter. In the next module, we will talk about such analyses.

[5] The answer is found in Section 7.3 at the end of this module.

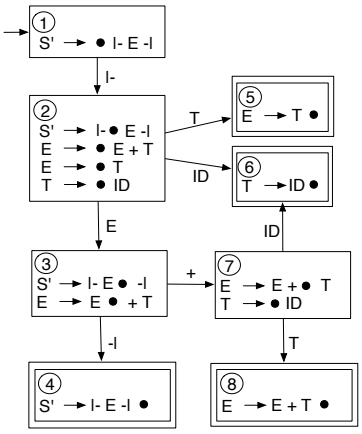
Read	Unread	Tree Stack	State Stack	Action
ϵ	$\vdash$ ID + ID $\dashv$		1	State 1 is a shift state. Shift $\vdash$
$\vdash$	ID + ID $\dashv$		1 2	State 2 is a shift state. Shift ID
$\vdash$ ID	+ ID $\dashv$		1 2 6	State 6 is a reduce state. Reduce rule 4
$\vdash$ ID	+ ID $\dashv$		1 2 5	State 5 is a reduce state. Reduce rule 3
$\vdash$ ID	+ ID $\dashv$		1 2 3	State 3 is a shift state. Shift +
$\vdash$ ID +	ID $\dashv$		1 2 3 7	State 7 is a shift state. Shift ID
$\vdash$ ID + ID	$\dashv$		1 2 3 7 6	State 6 is a reduce state. Reduce rule 4
$\vdash$ ID + ID	$\dashv$		1 2 3 7 8	State 8 is a reduce state. Reduce rule 2
$\vdash$ ID + ID	$\dashv$		1 2 3	State 3 is a shift state. Shift $\dashv$
$\vdash$ ID + ID $\dashv$	ϵ		1 2 3 4	State 4 is a reduce state. Reduce rule 1
$\vdash$ ID + ID $\dashv$	ϵ			Accept as soon as we push S'

7 Answers to longer self-practice questions

7.1 Self-practice answer 1

For the grammar and LR(0) DFA shown below, show that $\vdash + \dashv$ will cause a parser error.

(1) $S' \rightarrow \vdash$ E $\dashv$
(2) E $\rightarrow$ E + T
(3) E $\rightarrow$ T
(4) T $\rightarrow$ ID



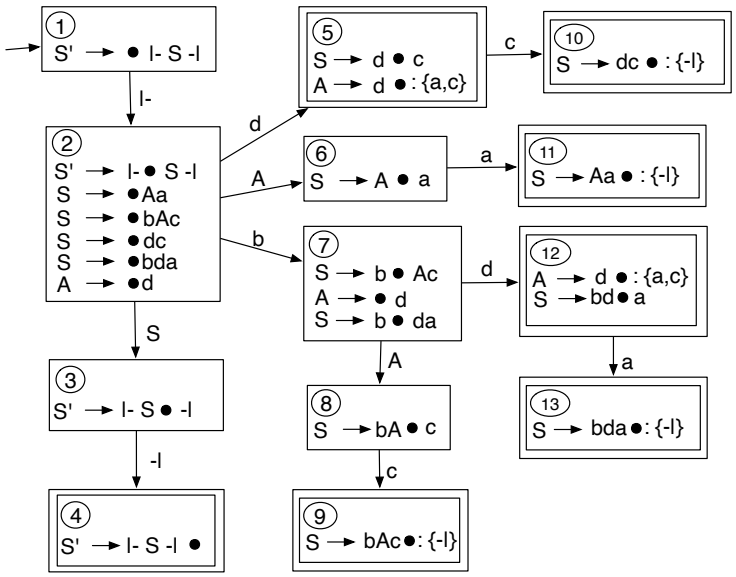
Read	Unread	Symbol Stack	State Stack	Action
ϵ	$\vdash + \dashv$		1	State 1 is a shift state. Shift $\vdash$
$\vdash$	$+ \dashv$	$\vdash$	1 2	State 2 is a shift state. Shift +

While we can push + on to the Symbol stack there is no transition defined from the current state 2 on the symbol we just pushed, +. The parser reports a parse error.

7.2 Self-practice answer 2

Create the SLR(1) DFA for this grammar. Are there any Reduce-Reduce conflicts? Are there any Shift-Reduce conflicts? Is the grammar SLR(1)?

(1) $S' \rightarrow \vdash$ S $\dashv$
(2) S $\rightarrow$ Aa
(3) S $\rightarrow$ bAc
(4) S $\rightarrow$ dc
(5) S $\rightarrow$ bda
(6) A $\rightarrow$ d



Shift-Reduce conflict: There are in fact two shift-reduce conflicts. In state 5 and in state 12. In state 5, we have the option to reduce to A if the next input symbol is a or c since follow(A) is {a,c}. However, we could also shift c. A similar conflict arises in state 12.

Reduce-Reduce conflict: There are no reduce-reduce conflicts.

Is the grammar SLR(1)? No. A grammar that has a Shift-Reduce or Reduce-Reduce conflict in the SLR(1) DFA is by definition not SLR(1).

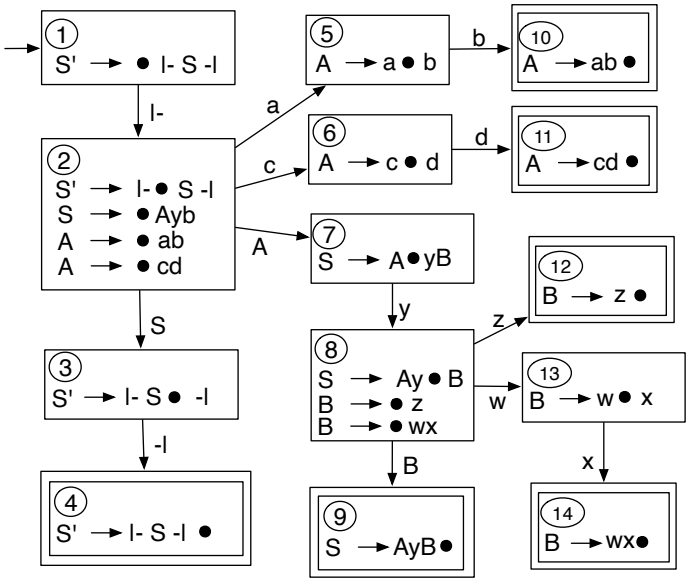
7.3 Self-practice answer 3

(1) $S' \rightarrow \vdash$ S $\dashv$
(2) S $\rightarrow$ AyB
(3) A $\rightarrow$ ab
(4) A $\rightarrow$ cd
(5) B $\rightarrow$ z
(6) B $\rightarrow$ wx

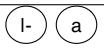
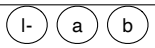
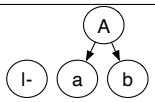
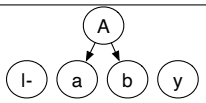
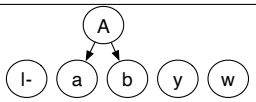
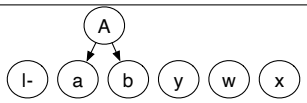
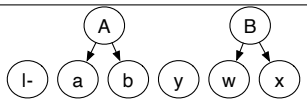
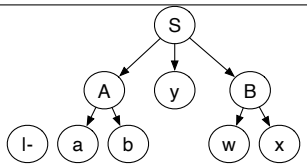
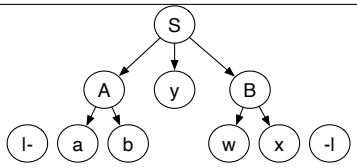
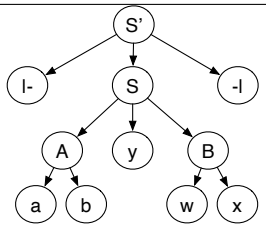
Create the LR(0) DFA.

Given the input string: $\vdash$ a b y w x $\dashv$, give a step by step construction of the parse tree showing `Read`, `Unread`, `Tree Stack`, `State Stack` and `Action` columns like the examples in this module.

First, we create the LR(0) DFA:



Since there are no Shift-Reduce or Reduce-Reduce conflicts, the grammar is LR(0). Let's now parse the input string $\vdash$ a b y w x $\dashv$ using the grammar and DFA we just created:

Read	Unread	Tree Stack	State Stack	Action Action
ϵ	$\vdash$ a b y w x $\dashv$		1	State 1 is a shift state. Shift $\vdash$
$\vdash$	a b y w x $\dashv$		1 2	State 2 is a shift state. Shift a
$\vdash$ a	b y w x $\dashv$		1 2 5	State 5 is a shift state. Shift b
$\vdash$ a b	y w x $\dashv$		1 2 5 10	State 10 is a reduce state. Reduce Rule 3
$\vdash$ a b	y w x $\dashv$		1 2 7	State 7 is a shift state. Shift y
$\vdash$ a b y	w x $\dashv$		1 2 7 8	State 8 is a shift state. Shift w
$\vdash$ a b y w	x $\dashv$		1 2 7 8 13	State 13 is a shift state. Shift x
$\vdash$ a b y w x	$\dashv$		1 2 7 8 13 14	State 14 is a reduce state. Reduce Rule 6
$\vdash$ a b y w x	$\dashv$		1 2 7 8 9	State 9 is a reduce state. Reduce Rule 2
$\vdash$ a b y w x	$\dashv$		1 2 3	State 3 is a shift state. Shift $\dashv$
$\vdash$ a b y w x $\dashv$	ϵ		1 2 3 4	State 4 is a reduce state. Reduce Rule 1
$\vdash$ a b y w x $\dashv$	ϵ			Accept as soon as we push S'