LR parsing

1 Introduction

Recursive descent parsing or parsing using an LL table are 'top-down'; the parse tree is built from the top downwards. There are also 'bottom-up' parsing methods where the parse tree is built in the other direction starting with leaves and then composing them to bigger trees.

The top-down methods are easy to understand and implement, but may require modification of the grammar to eliminate left recursion and common prefixes. Sometimes you have to use a stack to build a tree that respects conventional operator association rules.

The bottom-up methods are harder to understand and unfeasible for implementation by hand. They are generated by programs. The advantages of the generated parser include better error diagnoses, higher efficiency, and better capability to handle some grammars that cannot be used with top-down methods.

The most basic methods use LR parsing, and we will only construct such parsers without lookahead. LR-methods may work with no lookahead since the production rule to be used is not chosen until the full right hand part of the rule has been matched with the input string.

LR parsing was invented by Knuth [2]. Every text book on compiler construction has a chapter on LR-parsing going further than this introduction. The presentation here is inspired by a text book by Grune and Jacobs [1].

1.1 LR-parsing

The letters 'LR' stands for 'scanning from the Left constructing a Rightmost derivation'. We assume that we have grammar on canonical form and that there is just a single occurrence of the start symbol in the grammar. Let us consider a very simple grammar

$$\begin{aligned} p_0&: \mathsf{S} \rightarrow \mathsf{E}\ \$ \\ p_1&: \mathsf{E} \rightarrow \mathsf{E}\ \text{-}\ \mathsf{x} \\ p_2&: \mathsf{E} \rightarrow \mathsf{x} \end{aligned}$$

and the input string x - x - x $. The parser will discover the following derivation in reverse.

$$\mathsf{S} \Rightarrow \mathsf{E}\ \$ \Rightarrow \mathsf{E}\ \text{-}\ \mathsf{x}\ \$ \Rightarrow \mathsf{E}\ \text{-}\ \mathsf{x}\ \text{-}\ \mathsf{x}\ \$ \Rightarrow \mathsf{x}\ \text{-}\ \mathsf{x}\ \text{-}\ \mathsf{x}\ \$$$

For each derivation step the parser has to decide which rule to use to get the previous string in the derivation. Using a production backwards is called a *reduction*.

Starting with the final string, x - x - x $, the last rule can be used to reduce one of the three x's. Obviously the parser needs help to select the right one, i.e. to use the first one. In the next step with E - x - x $, there are also three possible reductions. Again it is correct to reduce at the beginning of the string. One might guess that you should always make this choice. However, other examples prove this to be wrong.

The parser uses a stack to build the parse tree. The stack will always contain a prefix of a sentential form and the concatenation of the stack with the remaining input will constitute a sentential form. In each step either a symbol is *shifted* from the input to the stack or a number

of symbols at the topmost part of the stack are *reduced* to a single nonterminal symbol using one of the grammar rules backwards. The string of symbols that are reduced is called a *handle*.

Before constructing an LR parser we shall look at the excecution. The table below shows the contents of the stack, the remainder of the input, and the next action of the parser. There are three kind of actions: *shift*, *reduce*, and *accept*. The accept action takes place when the stack just contains the start symbol of the grammar.

stack	input	action
	x - x - x $	shift
x	- x - x $	reduce by E → x
E	- x - x $	shift
E -	x - x $	shift
E - x	- x - x $	reduce by E → E - x
E	- x - x $	shift
E -	x - x $	shift
E - x	- x $	reduce by E → E - x
E	- x $	shift
E -	x $	shift
E - x	$	reduce by E → E - x
E	$	shift
E $		reduce by S → E $
S		accept

It remains to describe how to decide when to reduce and when to shift. Reducing is not possible at every step while shifting is always an option if there are further input tokens. We will use the grammar to check that the contents of the stack is a prefix of some sentential form and to detect when there is a handle on the topmost part of the stack.

The grammar will be represented by a finite automaton so that we can keep track of which production we are using and exactly where in the right hand part we are. A finite automaton is not powerful enough to accept most languages described by a context free grammar, but analyzing the stack is much simpler.

Each state of the automaton is labeled with an *LR item*. Lets call the left hand part of a production the *head* and the right hand part the *body*. An LR item is a grammar rule with a dot inserted somewhere in the body. The rule E → E - x gives rise to four different items:

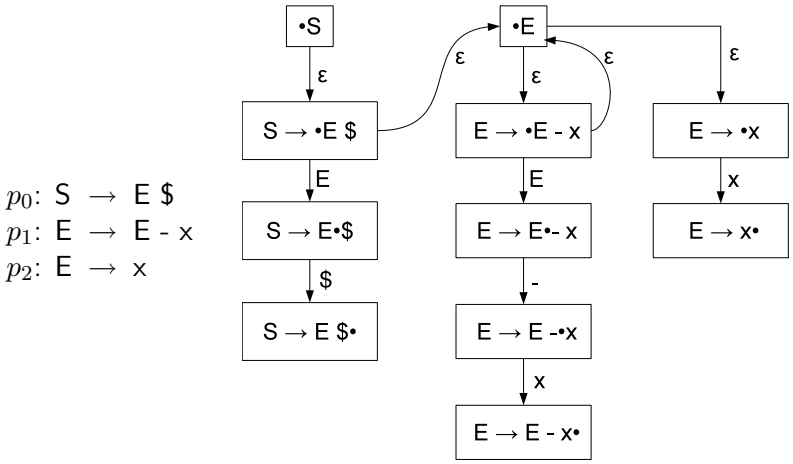
$$\mathsf{E} \to \bullet\mathsf{E}\text{ - x} \quad \mathsf{E} \to \mathsf{E}\bullet\text{- x} \quad \mathsf{E} \to \mathsf{E}\text{ -}\bullet\mathsf{x} \quad \mathsf{E} \to \mathsf{E}\text{ - x}\bullet$$

The dot indicates how far we have used the production. The symbols before the dot in the body will appear on the topmost part of the stack and symbols after the dot are expected to be matched by a prefix of the remainder of the input string.

A rule with an empty body $\mathsf{E} \to \epsilon$ generates just one item, $\mathsf{E} \to \bullet$.

We will use two extra states corresponding to the nonterminals of the grammar labeled by $\bullet S$ and $\bullet E$. They will help in understanding the automaton and may reduce the number of arrows when the grammar is more complicated, but they may be ignored when using the automaton. There will be ϵ transitions from these states to all items with same symbol as the left hand part hand a right hand part starting with a dot. There will also be ϵ transitions back from all items having the label as a substring. These back arrows will correspond to the recursive use of a

grammar rule inside a rule body. Notice that the automaton is not deterministic due to the ϵ transitions.



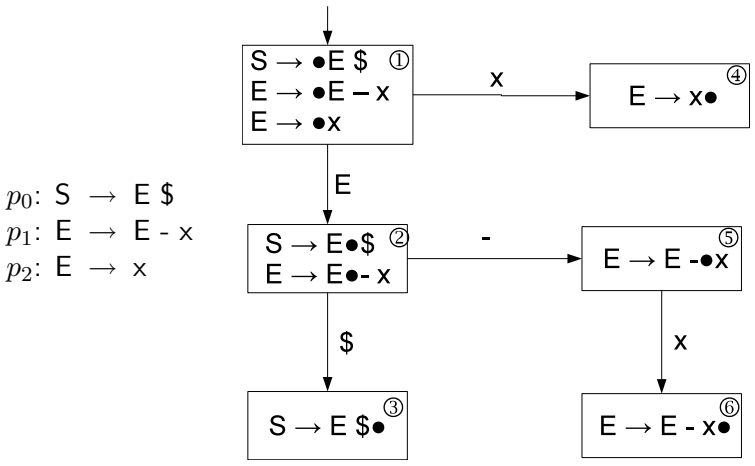
The other transitions are obtained by moving the dot one step while accepting the symbol that was exchanged with the dot. When the parser is in a state labeled by, e.g. $\mathsf{E} \rightarrow \mathsf{E}\text{ -}\bullet\mathsf{x}$, it means that we are under way using the rule $\mathsf{E} \rightarrow \mathsf{E}\text{ - x}$ and have reached the position indicated by the dot. It is obvious that the grammar can be recovered from the automaton. It is just another representation.

If we make all the states of the automaton accepting then it will accept the language of all prefixes of the sentential forms that can be generated by the grammar. The stack should always contain such a prefix, and we are going to use the automaton to check this condition.

If the automaton accepts the contents of the stack in a state where the item terminates with a dot then we know that there is a handle on the topmost part of the stack that can be used for reduction by the rule indicated by the item label. We will call it a *reducing item*.

If we apply the automaton to a given stack we may get stuck in a state where the is no transition on the next symbol. The reason is either that the string is not generated by the grammar or that we have taken the wrong path through the automaton.

We can eliminate the problem of choosing an incorrect path by constructing an equivalent deterministic automaton using the subset construction algorithm. The states of this automation will be labeled by sets of items. During this construction we will ignore the states having labels that are not items. The result is as follows.



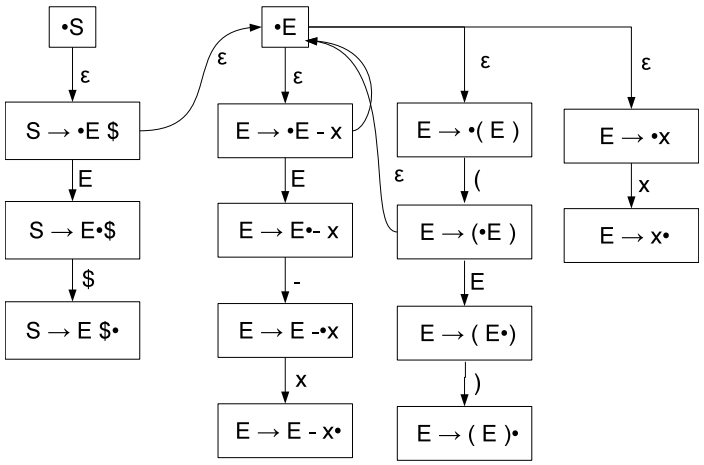
The contents of stack determines in what state we are. For each row in the table below we can compute the current state by giving the contents of the stack as input to the automaton.

stack	input	state	action
	x - x - x $	1	shift
x	- x - x $	4	reduce by E → x
E	- x - x $	2	shift
E -	x - x $	5	shift
E - x	- x - x $	6	reduce by E → E - x
E	- x - x $	2	shift
E -	x - x $	5	shift
E - x	- x $	6	reduce by E → E - x
E	- x $	2	shift
E -	x $	5	shift
E - x	$	6	reduce by E → E - x
E	$	2	shift
E $		3	reduce by S → E $
S			accept

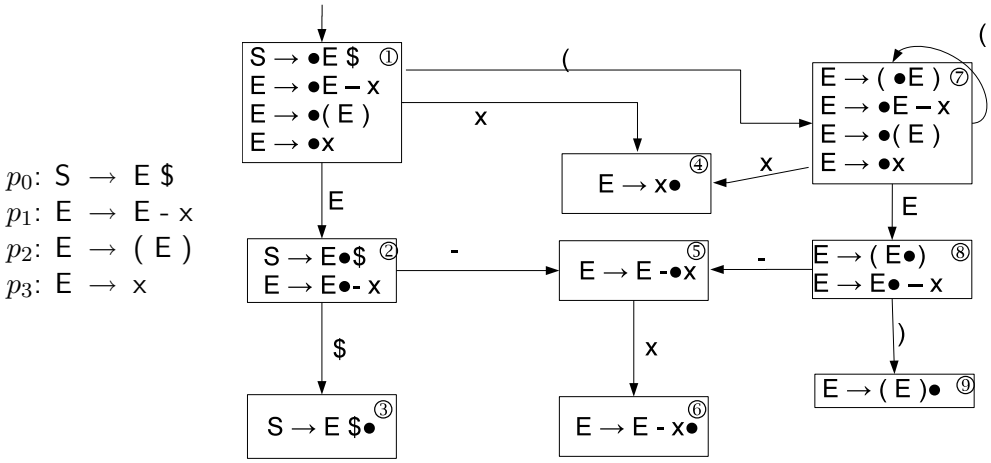
When we arrive in a state with an item ending with a • we have found a handle and can use it to reduce the stack. We notice that the last row in the table is unnecessary; when we are able to reduce to the start symbol we already know that the parser will accept the input string.

Let us consider another example. We now may have parentheses around an expression. The corresponding NFA is shown to right.

p_0: S → E $
p_1: E → E - x
p_2: E → (E)
p_3: E → x



Using the subset construction we obtain the following DFA.



Parsing the string ((x - x)) $ proceeds as follows.

stack	input	state	action
	((x - x)) $	1	shift
(	(x - x)) $	7	shift
((	x - x)) $	7	shift
((x	- x)) $	4	reduce by E → x
((E	- x)) $	8	shift
((E -	x)) $	5	shift
((E - x	)) $	6	reduce by E → E - x
((E	)) $	8	shift
((E)	) $	9	reduce by E → (E)
(E	) $	8	shift
(E)	$	9	reduce by E → (E)
E	$	2	shift
E $		3	reduce by S → E $
S			accept

Now we are ready for some optimizations. A reduction always take place at the top of stack while the bottom remains intact. At the first and second reduction above the two left parentheses at the bottom of the stack are unaffected. It shouldn't be necessary to inspect the contents of the stack from the bottom to find the current state, but to do so we need to remember which state is associated with each symbol on the stack. We will push the current state on stack and pop those states along with the symbols when we reduce.

stack	input	state	action
1	((x - x)) $	1	shift
1 (7	(x - x)) $	7	shift
1 (7 (7	x - x)) $	7	shift
1 (7 (7 x 4	- x)) $	4	reduce by E → x
1 (7 (7 E 8	- x)) $	8	shift
1 (7 (7 E 8 - 5	x)) $	5	shift
1 (7 (7 E 8 - 5 x 6	)) $	6	reduce by E → E - x
1 (7 (7 E 8	)) $	8	shift
1 (7 (7 E 8) 9	) $	9	reduce by E → (E)
1 (7 E 8	) $	8	shift
1 (7 E 8) 9	$	9	reduce by E → (E)
1 E 2	$	2	shift
1 E 2 $ 3		3	reduce by S → E $
1 S			accept

The terminal and nonterminal symbols on the stack can easily be computed from states and the DFA. In , e.g. 1 (7 E 8) 9 the symbols are just the transition labels on the path from state 1 via 7 and 8 to 9, so there is no reason to put symbols on the stack. We will remove them shortly.

From the DFA we can construct a table that will guide the parser. The action column specifies, for each state, if the parser should shift one symbol from input to the stack or reduce by the indicated rule. The remainder of the table is just the transition table for the DFA.

p_0: S → E $
p_1: E → E - x
p_2: E → (E)
p_3: E → x

state	action	x	-	(	)	$	E
1	shift	4		7			2
2	shift		5			3	
3	reduce by p_0						
4	reduce by p_3						
5	shift	6					
6	reduce by p_1						
7	shift	4		7			8
8	shift		5		9		
9	reduce by p_2						

After a shift action the state and the newly shifted symbol determine the next state. So, after shifting a left parenthesis in state 1 the parser moves to state 7.

After a reduce action the top of stack will be a nonterminal symbol and below that a state. This state will be where we end up if use the DFA to accept the symbols below. From this state we just follow the edge labeled with the top of stack symbol. This will become the next state of the parser and subsequently we push it on the stack.

As an example consider the following reduction from the previous table

stack	input	state	action
1 (7 (7 E 8 - 5 x 6	)) $	6	reduce by E → E - x
1 (7 (7 E 8	)) $	8	shift

Since the production has three symbols in the body we pop the same three symbols from the stack together with the three states. Then we push the head of the rule on stack and consult transition table for state 3 and symbol E. The entry 8 is pushed as the new current state.

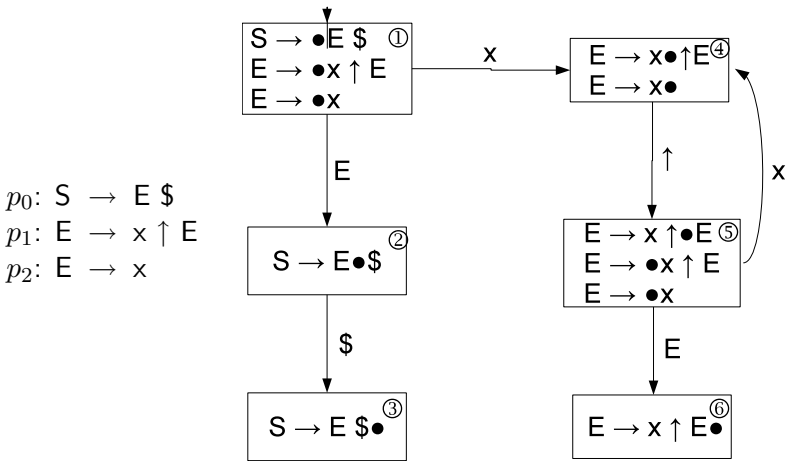
Since we actually never use the symbols on the stack, except the top one, it suffices to push and pop states. We just need to record the topmost symbol after a shift or reduce action. A simple variable will do.

When doing a reduction the parser should pop the same number of states as there are symbols in the right hand part of the production and push the state indexed by the topmost state and the left hand part of the production.

Will this procedure work for any grammar? The answer must be no since if the grammar is ambiguous there must be a conflict. There are two kinds of conflicts, *reduce/reduce* conflicts and *shift/reduce* conflicts. They can occur even if the grammar is unambiguous. A *reduce/reduce* conflict occurs when there are two ore more reducing items in a state. The other kind occurs when a state has both a reducing item and an outgoing transition. There cannot be *shift/shift* conflicts since the subset construction will join multiple transitions on the same symbol to one transition.

There are no such conflicts in the DFAs for our previous examples. When this is the case we say that the grammar is LR(0). The number 0 indicates that we have used no lookahead; we don't inspect the next input symbol to decide which action to take.

If there is a conflict it might help to use some lookahead, i.e. inspect symbols on the input that have not been shifted yet. Next we consider such an example. It is about exponentiation which, by convention, associates to the right. This means that x ↑ x ↑ x should be interpreted as x ↑ (x ↑ x) Thus the exponentiation production should be right recursive.



Now, in state 4 there is a shift/reduce conflict. We could either reduce by E → x or shift. If the next input symbol is ↑ then shifting is an option. Otherwise it is not and we should certainly reduce.

What will happen if we decide to reduce when the next input symbol is ↑ ? Consider the input string x ↑ x $:

stack	input	state	action
	x ↑ x $	1	shift
x	↑ x $	4	reduce
E	↑ x $	1	shift
E ↑	x $	?	error

Since E ↑ x $ is not a sentential form for this grammar an error should be reported.

We can avoid making the wrong choice by analyzing the grammar. We can compute the follow set for symbol E. It turns out be {$}. This means that we should only reduce to an E if the next input symbol is $.

When all conflicts in the DFA can be resolved by using one symbol lookahead the grammar is said to be LR(1). In such a case there is a similar procedure to construct an automaton, but then the items will include a lookahead symbol. This means that there will be many more items and the corresponding DFA will have too many states for nontrivial grammars to be useful in practice.

There are ways to add lookahead information into the LR(0) automaton without making it much bigger. The algorithms to do that are clever. Consult almost any compiler construction text book for the details. These parsers are somewhat less powerful than LR(1) parsers, but excellent for most programming languages.

References

[1] Dick Grune and Ceriel J.H. Jacobs: *Parsing Techniques — A Practical Guide*, Ellis Horwood, 1990. Also retrieved from `http://dickgrune.com/Books/PTAPG_1st_Edition/`, 2012.

[2] Donald E. Knuth: *On the Translation of Languages from Left to Right*, Information and Control 8, 607-639, 1965. Also retrieved from `www.cs.dartmouth.edu/~mckeeman/cs48/mxcom/doc/knuth65.pdf`, 2012.