Programming Languages and Computation

Week 4: Obtaining LL(1) Grammars

* 1. Consider the following grammar for binary list literals:

$$\begin{array}{rcl} L & ::= & [\,] \mid [B] \mid [M] \\ M & ::= & M,\ M \mid B \\ B & ::= & 0 \mid 1 \end{array}$$

Give an equivalent grammar that avoids left recursion.

Solution

$$\begin{array}{rcl} L & ::= & [\,] \mid [B] \mid [M] \\ M & ::= & B,\ M \mid B \\ B & ::= & 0 \mid 1 \end{array}$$

* 2. Left factor the following grammar (that is, give an equivalent grammar in which no two productions for the same non-terminal have the same prefix):

$$S ::= \text{if bexp then } S \mid \text{if bexp then } S \text{ else } S \mid \text{other}$$

Solution

$$\begin{array}{rcl} S & ::= & \text{if bexp then } S\ T \mid \text{other} \\ T & ::= & \text{else } S \mid \epsilon \end{array}$$

** 3. For each of the following grammars, give an equivalent LL(1) grammar. Typically, you will need to look at examples of strings in the language of the grammars to get an idea of the shape of the sequences generated by left recursive rules in order to give insight into the problem.

(a)

$$S ::= 0\ S\ 1 \mid 0\ 1$$

(b)

$$\begin{array}{l} S ::= (\ L\ ) \mid a \\ L ::= L,\ S \mid S \end{array}$$

(c)

$$S ::= S\ (\ S\ )\ S \mid \epsilon$$

Solution

(a) This example has no left recursion, but it does have a factoring problem. Suppose we are constructing a derivation from S and the next letter of the input is 0, which rule do we use? It is not clear without more information, because both the rules for S have a 0 in the leftmost position on the right-hand side. Factoring out the common prefix '0' gives the following:

$$\begin{array}{lll} S & ::= & 0\ T \\ T & ::= & 1 \mid S\ 1 \end{array}$$

To check that this grammar is LL(1), we need to check whether, for each combination of nonterminal and terminal, the choice of rule is uniquely determined. For S combined with any terminal, this is clear because there is only one rule anyway. For T, when the next letter of the input is 1, there is only one rule we can choose, $T ::= 1$ because we cannot derive a string starting 1 from the other choice $T ::= S\ 1$ (S only derives strings starting 0). Similarly, for T when the next letter of the input is 0, the only choice is $T ::= S\ 1$, for similar reasons.

Strictly speaking, to check that a grammar is LL(1), I should construct the parse table and check that there are no cells containing more than one rule. You may notice that the reasoning above is essentially the same as the reasoning I would go through when I construct the parse table: for each cell of the parse table I have to check that I only put at most one rule in the cell, and each cell corresponds to a particular pair of a nonterminal (the row) and terminal (the column).

Since none of the nonterminals is nullable, I only have to check the first of the two conditions in the definition of parse table, which concerns the first set of the RHS of the rule, to see whether a given rule should be inserted in a given cell. If this was not the case, and I also had to check the second condition, I think it would no longer be manageable to do it in my head, and I would want to construct the table carefully (or at least the follow sets).

(b) Inspection of the given grammar shows that it contains left recursion in the rules for L. So, this should be removed as a step towards an equivalent LL(1) grammar. We ask ourselves what kind of sequences do the problematic L-rules derive? One way to try to answer this question is to look at the sentential forms that can be derived from L only using the L rules (never replacing an S).

$$\begin{array}{l} L \rightarrow S \\ L \rightarrow L, S \rightarrow S, S \\ L \rightarrow L, S \rightarrow L, S, S \rightarrow S, S, S \\ \ldots \text{and so on} \end{array}$$

So, one can be convinced that the purpose of L is to generate all nonempty sequences of S separated by commas. So, our task is to replace the current L rules by new ones that also generate all nonempty sequences of S separated by commas, but without using left recursion. We can do this using a "cons" formulation of sequences (lists):

$$\begin{array}{lll} S & ::= & (\ L\ ) \mid a \\ L & ::= & S\ L \mid S \end{array}$$

Here, rule $L ::= S\ L$ can be read as producing a new list $S\ L$ by adding ("consing") a single S onto the front of an existing list (L).

Unfortunately, the grammar is not yet LL(1) because there is now a left factoring problem. There is no way to choose between the two productions for L based only on the next letter of the input, because the RHS of both rules start with S. Therefore, we must left factor, by which we obtain:

$$\begin{array}{lcl} S & ::= & (\ L\ ) \mid a \\ L & ::= & S\ R \\ R & ::= & ,\ S\ R \mid \epsilon \end{array}$$

(c) In this part, there is left recursion, but it's a bit difficult to see immediately what kind of sequences the problematic rules generate. It may be clearer if we simplify the S rule a little. We could rephrase the grammar as:

$$\begin{array}{lcl} S & ::= & S\ T\ S \mid \epsilon \\ T & ::= & (\ S\ ) \end{array}$$

Hopefully you can see that this does not change the strings that are derivable compared to the original grammar. Now, one can see that using, *only* the S rules (never replacing any T), simply generates finite sequences of T of all lengths, e.g.

$S \to \epsilon$ zero Ts
$S \to S\ T\ S \to T\ S \to T$ one T
$S \to S\ T\ S \to S\ T\ S\ T\ S \to T\ S\ T\ S \to T\ T\ S \to T\ T$ two Ts
...and so on.

So, the task is to replace these S-rules by some different one(s) that similarly produce any finite sequence of T, but which are not left-recursive. This can be done by using a "cons" approach to describing sequences, as follows:

$$\begin{array}{lcl} S & ::= & T\ S \mid \epsilon \\ T & ::= & (\ S\ ) \end{array}$$

** 4. The following grammar describes the language of Regular Expressions (regex), which are a standard way of describing pattern matching on strings in programming languages.

$$S ::= S + S \mid S\ S \mid (\ S\ ) \mid S* \mid a$$

Give an LL(1) grammar for the same language.

Solution

$$\begin{array}{lcl} S & ::= & T\ R \\ R & ::= & +\ T\ R \mid T\ R \mid *\ R \mid \epsilon \\ T & ::= & (\ S\ ) \mid a \end{array}$$

*** 5. Consider again the grammar given in answer to Question 2.

(a) Explain why your left-factored grammar is not LL(1).

(b) Rather than giving an equivalent LL(1) grammar, which is impossible in this case, describe a simple modification to the parsing function for the nonterminal which exhibits the rule

conflict which will allow for parsing the left-factored grammar.

Solution

(a) It is not LL(1) because when deriving for T with else the next letter of the input, both of the rules for T are possible. This is called the "dangling else" problem. Consider the string if bexp then if bexp then other else other. Intuitively, it is not clear whether the else is counterpart to the first or second then.

(b) In the left-factored version of this grammar there is a conflict between the two rules for T on seeing else. A key observation is that, although there is a conflict, the first rule will always work (will lead to a correct derivation of a given string in the language). Therefore, we can just modify the parsing function for T to always choose the first, which corresponds to matching every else to the closest preceding then.