

# CS385-Assignment 6

Due Thursday, October 25

## 1 Problems from the Text

A couple of problems on proving that languages are not context-free. For Problems 2.31 and 2.32, you can use the result of Problem 2.18 — take the intersection of the given language with an appropriately chosen regular language. If the given language were context-free, Problem 2.18 says that the intersection would be too, and in many cases the intersection is easier to work with.

2.31, 2.32

And another problem: 2.18 itself. A solution is actually given, but I want you to try another approach: Given a grammar  $G$  for  $C$  and an automaton for  $R$ , construct a grammar  $G'$  for  $C \cap R$ . Here is a huge hint: The set of variables of  $G'$  can be taken as

$$S \cup (Q \times V \times Q) \cup (Q \times \Sigma \times Q),$$

where  $Q$  is the set of states of the automaton, and  $V$  the set of variables of  $G$ . What you have to describe is how to turn rules

$$X \rightarrow v$$

of  $G$  into rules of the form

$$(q_i, X, q_j) \rightarrow v'.$$

In general, each rule of  $G$  will give rise to many rules of  $G'$ . The idea is to ensure that any string over  $\Sigma$  derived from the start symbol is also the label of a path from the initial state to an accepting state of the automaton. Illustrate your construction with the grammar of Exercise 2.1, where  $R$  is the language consisting of all strings with an even number of occurrences of  $a$ . You probably won't be able to write out every rule of  $G'$ , but you can give examples of these

## 2 Problems with yacc

I've posted a yacc script, containing a grammar for postfix expressions, along with the parsing table it generates and a sample parse using the table.

Observe how the parsing algorithm works: We determine what move to make by the state number at the top of the stack, and the next symbol on the input. If the move says "shift  $k$ ", where  $k$  is a state number, we push the next input symbol, and then  $k$ , onto the stack.

If the move says “reduce  $m$ ”, then we reduce by rule number  $m$ , which has the form  $V \rightarrow w$ . The right-hand side  $w$  of this production should be on the stack (along with state numbers between the symbols). We pop  $w$  from the stack and note the state number  $k$  that is uncovered. We then go to the table entry for state  $k$  and the variable  $V$ , where we will see an entry “goto  $q$ ”. We push  $V$  and then  $q$  on the stack. The parsing ends with a move that says “accept” or “error”.

Devise yacc scripts for (a) prefix expressions—you just have to turn around the right-hand sides of the productions in the postfix rules—and (b) the grammar of Exercise 2.1. Here you will need new token symbols for the parentheses—use LPAR and RPAR—and new variable names—use “term” and “factor” along with “expression” and “fullexpression”.

You compile your script by naming the file something.y, and then typing  
yacc -v something.y

The parsing table is written on a file called y.output, which you should rename.

Use the resulting tables to show the parse of

(a)  $* + \text{op op op}$  and  $* \text{op op op}$  for the prefix grammar.

(b)  $\text{op} + \text{op} * \text{op}$  and  $(\text{op} + \text{op}) * \text{op}$  for the grammar of 2.1