

Final Exam, CS301, Spring '09, McConnell

1. Thinking of proof by induction in terms of proof by contradiction.

Consider the equality $\sum_{i=1}^n i^2 = n(n+1)(2n+1)/6$. Show that this holds for all $n \geq 1$.

- (a) Show that the equality holds in the case of $n = 1$.

Solution: $1 \times 2 \times 3/6 = 1 = \sum_{i=1}^1 i^2$.

- (b) Just because it holds for some small examples doesn't mean that it holds for all $n > 1$. Let's prove this by contradiction.

Suppose that there is some $n \geq 1$ that it fails on. We might as well let n be the *smallest* $n \geq 1$ that it fails on. Why is it that we can deduce that $n > 1$?

Solution: *Because we've shown that it doesn't fail for $n = 1$.*

- (c) Why is it that we can deduce that $\sum_{i=1}^{n-1} i^2 = (n-1)n(2(n-1)+1)/6$? Your argument should make use of the fact that $n > 1$, because this is not true for $n = 1$.

Solution: *Because it's the smallest value $n \geq 1$ for which the equality doesn't hold and $n > 1$, equality holds for the next smaller value, $n - 1$.*

- (d) Use this and some algebra to derive a contradiction. If you succeed, you've shown that the equality holds for all $n \geq 1$: Since your only questionable step was assuming that there was an $n \geq 1$ that it fails on, it must be the case that it doesn't fail on *any* $n \geq 1$.

Solution: $\sum_{i=1}^n i^2 = \sum_{i=1}^{n-1} i^2 + n^2 = (n-1)n(2(n-1)+1)/6 + n^2 = (2n^3 - 3n^2 + n)/6 + n^2 = (2n^3 + 3n^2 + n)/6 = n(2n^2 + 3n + 1)/6 = n(n+1)(2n+1)/6$.

(Notice that $\sum_{i=1}^{n-1} i^2 = (n-1)n(2(n-1)+1)/6$ is the induction hypothesis. In some presentations of induction, it seems like a "leap of faith," but thinking about induction in terms of proof by contradiction allows us to **deduce** the induction hypothesis.)

2. Prove that the language of palindromes on $\{a, b\}$ is not regular.

Solution: *Suppose the language is regular. The pumping lemma applies. There exists an n large enough such that for all strings of length at least n in the language there is a nonempty substring in the first n characters that can be pumped. The string $a^n b a^n$ is in the language, but pumping anything out of the first n characters results in a string that isn't in the language, a contradiction.*

3. For each of the following regular languages, find the shortest strings on $\{a, b\}^*$ not in the language. If there is more than one shortest string, list all of them.

- (a) $(aa)^*(bba)^*(bb)^*$

Solution: a and b

- (b) $a^*(bba)^*b^*$

Solution: ba

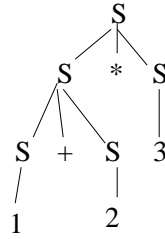
- (c) $a^*(ab \cup ba)^*b^*$

Solution: baa and bba

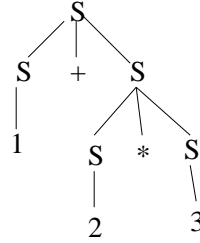
4. Consider the following grammar:

$$S \rightarrow -S \mid S + S \mid S - S \mid S * S \mid S / S \mid (S) \mid 0 \mid 1 \mid 2 \mid 3 \mid x \mid y \mid z.$$

Show that the grammar is ambiguous by giving two parse trees for $1 + 2 * 3$.



The addition is
evaluated first



The multiplication
is evaluated first

5. Ambiguity is bad, because if a compiler uses one of the parse trees for the above expression, its value is 9, and if it uses the other, the value is 7. That's because the parse tree tells which operator should be applied first. (The lower an operator is in the tree, the earlier it is applied, since evaluation works by induction from the leaves to the root.)

To avoid ambiguity, the designer of a language must assign higher priority, or *precedence*, to some operators than to others. In most programming languages, multiplication and division have higher precedence than addition or subtraction. Multiplication and division are tied, so the designer may decide to break the tie by evaluating them in left-to-right order. Similarly, addition and subtraction are tied, and these can be evaluated in left-to-right order.

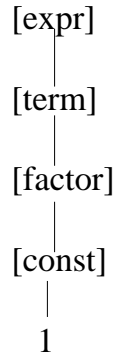
We can think of *terms* as operands of addition and subtraction operators, and *factors* as operands of multiplication and division operators. We can design nonterminals for the language of terms and for the language of factors. By controlling which can generate which, we can enforce precedence on the operators.

$$\begin{aligned} [expr] &\rightarrow [term] \mid [sign][term] \mid [expr][addop][term]; \\ [sign] &\rightarrow + \mid -; \\ [addop] &\rightarrow + \mid -; \\ [term] &\rightarrow [factor] \mid [term][multop][factor]; \\ [multop] &\rightarrow * \mid / \mid div \mid mod; \\ [factor] &\rightarrow [var] \mid [const] \mid ([expr]); \\ [var] &\rightarrow x \mid y \mid z; [const] \rightarrow 0 \mid 1 \mid 2 \mid 3 \end{aligned}$$

Let's call this *Grammar A*. It is unambiguous. Let's verify that this is true for a few examples.

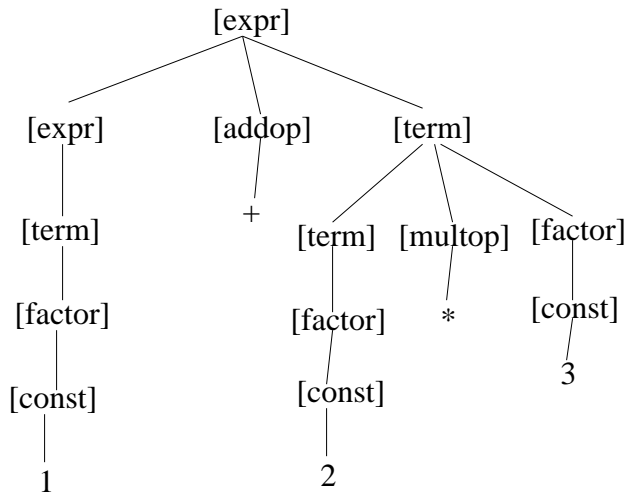
- (a) Give the Grammar-A parse tree of the following expression; 1.

Solution:



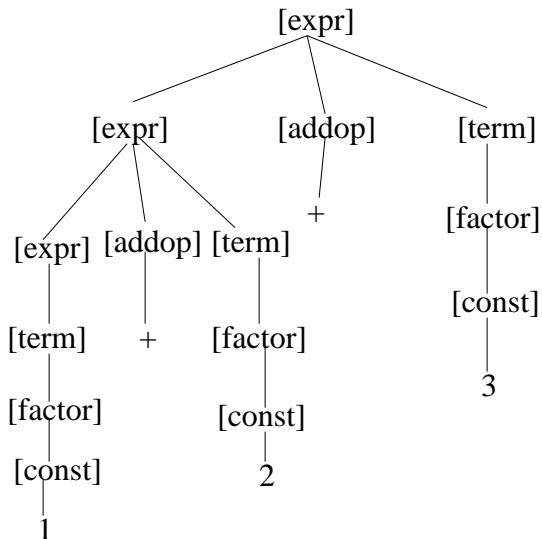
- (b) Give the Grammar-A parse tree of $1 + 2 * 3$. The result should be a parse tree where the multiplication is lower in the tree than the addition, and therefore evaluated first, as you would expect.

Solution:



- (c) Give the Grammar-A parse tree of $1 + 2 + 3$. The result should be a parse tree where the operators are evaluated in left-to-right order.

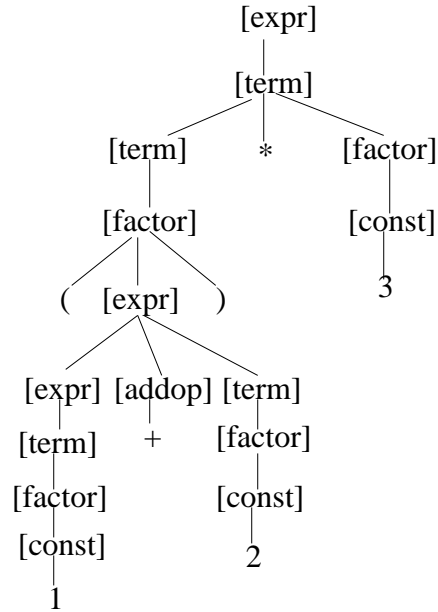
Solution:



- (d) Give the Grammar-A parse tree of $(1 + 2) * 3$. The result should be a parse

tree where the addition is evaluated first, as the operator precedence has been overridden by the parentheses.

Solution:



6. Change the rules involving $[var]$ in Grammar A so that it generates any string that begins with one of $\{a, b, c\}$ and is followed by any sequence of length 0 or greater on alphabet $\{a, b, c, 0, 1, 2, 3\}$. You can make use of an additional nonterminal, B .

Solution: $[var] \rightarrow aB \mid bB \mid cB;$
 $B \rightarrow aB \mid bB \mid cB \mid 0B \mid 1B \mid 2B \mid 3B \mid \epsilon$

7. Change the rules involving $[const]$ so that it generates any sequence on alphabet $\{0, 1, 2, \dots, 9\}$ that doesn't begin with a 0. You can make use of an additional symbol, C .

Solution: $[const] \rightarrow 1C \mid 2C \mid 3C \mid \dots \mid 9C;$
 $C \rightarrow 0C \mid 1C \mid 2C \mid \dots \mid 9C \mid \epsilon;$

Note: when you design a language, it's okay to forbid leading zeros in constants, except for one case: the representation of the number 0 has a leading zero. Adding the following rule fixes this:

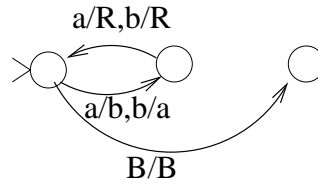
$[const] \rightarrow 0$

8. Consider a model of a Turing machine where each transition has a *single* action, which can *either* be writing a character, *or* moving left or right.

Draw a Turing machine that conforms to this model, and transforms its input string by replacing every a with a b and every b with an a , then halts with the tape head over the first blank.

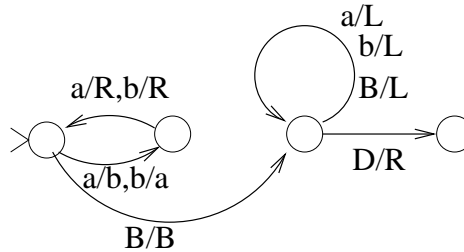
Assume that the initial configuration consists of the start-of-tape symbol, D , followed by the input string, followed by blanks. The read head initially sits over the first character of the input string.

Solution:



9. Modify this Turing machine so that it still conforms to the model, performs the swapping of *a*'s and *b*'s as above, and halts with the read head over the first character following the start-of-tape symbol.

Solution:

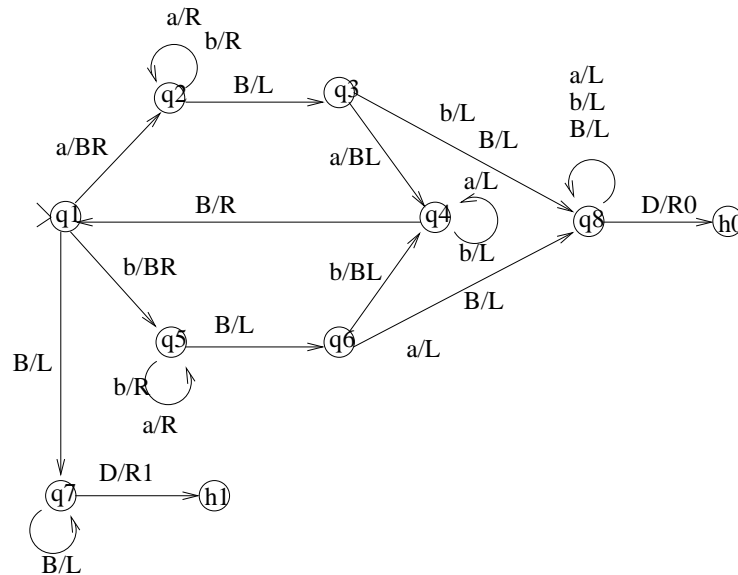


10. Let us now allow a transition to perform up to two actions, one of which is writing a character, and one of which is moving left or right.

Consider the following Turing machine, which recognizes even-length palindromes.

Turing machine for recognizing even-length palindromes

D: marks left end of tape
 B: blank
 L: move left
 R: move right



The initial configuration consists of the start-of-tape symbol, *D*, followed by the input string and the read head over the first character of the input string. It halts with a 0 in the first position following the *D* if the string is not an even-length palindrome, and a 1 in the first position following the *D* if the string *is* an even-length palindrome.

- (a) Show the computation of this machine when the input string is $baaa$ by giving the sequence of configurations that occur until the halting point.

Solution: $Dbaaa, q_1, DBaaa, q_5, DBaaa, q_5, DBaaa, q_5, DBaaa\underline{B}, q_5, DBaaa, q_6, DBaaa, q_8, DBaaa, q_8, D\underline{B}aaa, q_8, \underline{D}Baaa, q_8, D\underline{0}aaa, h_0.$

11. Modify the above Turing machine so that it recognizes even-length palindromes on alphabet $\{a, b, c\}$.

Solution:

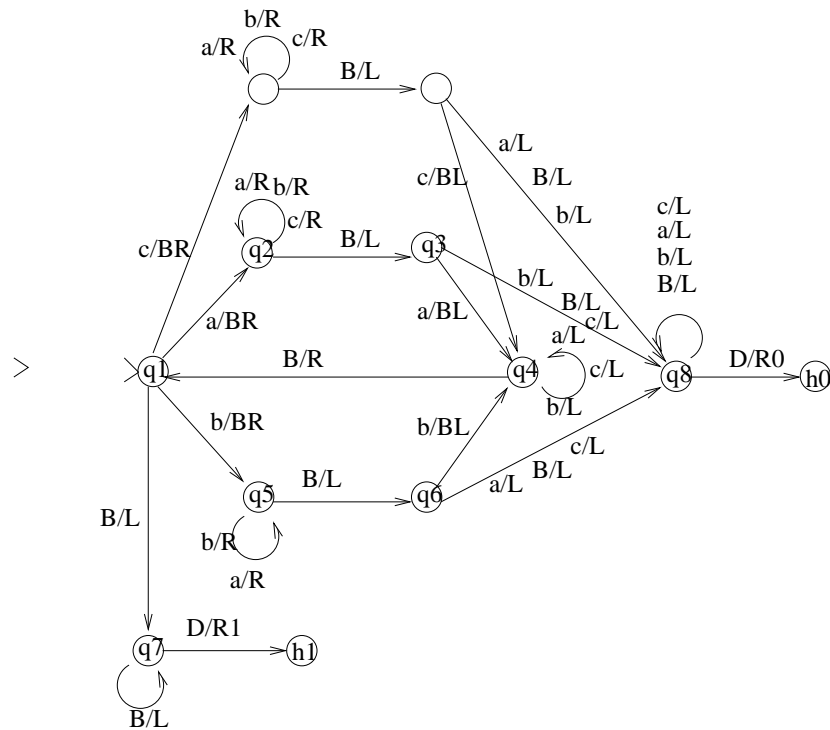
Turing machine for recognizing even-length palindromes

D: marks left end of tape

B: blank

L: move left

R: move right



Extra Credit

12. Reduction of Hamiltonian Cycle to Traveling Salesman.

- (a) State in plain English what the Traveling Salesman Problem (TSP) consists of.

Solution: You're given a graph where the edges have costs, and a trip cost c . You must determine whether there is a Hamiltonian cycle whose total trip cost is at most c .

- (b) The handout describes how to take any instance of Hamiltonian Cycle and create an instance of TSP such that the answer to the TSP problem is yes if and only if the answer to the Hamiltonian Cycle problem is YES. The instance of TSP is a complete graph (a clique on all the vertices). What is the rule for assigning costs to the edges and what is the cost limit for the traveling salesman tour?

Solution: Let G be the input graph to the Hamiltonian cycle problem. Create a graph G' on the same set of vertices. Put an edge between every pair of vertices in G' . Make the cost of an edge in G' be 0 if it's an edge of G and 1 if it's not. Ask whether there is a traveling-salesman tour of length at

- (c) Does the reduction prove that if there is no polynomial-time algorithm for TSP, then there is no polynomial-time algorithm for HAMILTON-CYCLE? Why or why not?

Solution: No, it leaves open the possibility that there is a polynomial-time algorithm for HAMILTONIAN-CYCLE, none for TSP, and that the reduction just gives a way to solve a polynomial problem in exponential time.

- (d) Does the reduction show that if there is no polynomial-time algorithm for HAMILTONIAN-CYCLE, then there is no polynomial-time algorithm for TSP? Why or why not?

Solution: Yes. We can prove this by contradiction. Suppose there is no polynomial-time algorithm for HAMILTONIAN-CYCLE but there is one for TSP. Generating an instance of TSP from an instance of HAMILTONIAN-CYCLE takes polynomial time. Since the answer to the instance of TSP is yes if and only if the answer to the instance of HAMILTONIAN CYCLE is yes, we can find out the answer to our instance of HAMILTONIAN-CYCLE in polynomial time by applying the polynomial-time algorithm for TSP to the generated instance.

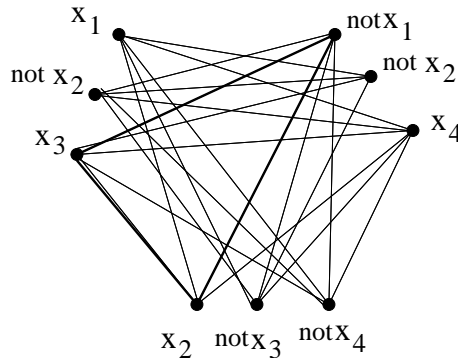
13. Reduction of 3-CNF-SAT to CLIQUE

- (a) The handout shows how to take any instance of 3-CNF-SAT and construct an instance of CLIQUE such that the answer is *yes* if and only if the given instance of 3-CNF-SAT is satisfiable.

Show that you understand the reduction by drawing the graph and the size of clique you're looking for when the instance of 3-CNF-SAT is $(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_4) \wedge (x_2 \vee \bar{x}_3 \vee \bar{x}_4)$.

- (b) A satisfying assignment of the instance of 3-CNF-SAT is $x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 0$. What are the vertices of the clique of the desired size that corresponds to this satisfying assignment?

Solution:



14. Reduction of 3-CNF-SAT to SUBSET-SUM.

- (a) The handout shows how to take any instance of 3-CNF-SAT and construct an instance of SUBSET-SUM, such that the answer to the instance of SUBSET-SUM is *yes* if and only if the instance of 3-CNF-SAT is satisfiable.

Show that you understand the reduction by finding the set of integers to choose from and the target for the constructed instance of SUBSET-SUM when the instance of 3-CNF-SAT is $(x_1 \vee \bar{x}_2 \vee x_3) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_4) \wedge (x_2 \vee \bar{x}_3 \vee \bar{x}_4)$.

- (b) A satisfying assignment of the instance of 3-CNF-SAT is $x_1 = 0, x_2 = 1, x_3 = 1, x_4 = 0$. Give the set of integers summing to the target that this satisfying assignment corresponds to in your constructed instance of SUBSET-SUM.

x1	x2	x3	x4	C1	C2	C3	
1	0	0	0	1	0	0	Asterisks indicate the subset
1	0	0	0	0	1	0	corresponding to the satisfying
0	1	0	0	0	0	1	* assignment. Notice that they
0	1	0	0	1	1	0	* add up to the target, t = 1111444.
0	0	1	0	1	0	0	*
0	0	1	0	0	0	1	
0	0	0	1	0	1	0	
0	0	0	1	0	0	1	*
0	0	0	0	1	0	0	*
0	0	0	0	2	0	0	*
0	0	0	0	0	1	0	*
0	0	0	0	0	2	0	*
0	0	0	0	0	0	1	
0	0	0	0	0	0	2	*
<div style="display: flex; justify-content: space-between; align-items: flex-end;"> 1 1 1 1 4 4 4 = t </div>							