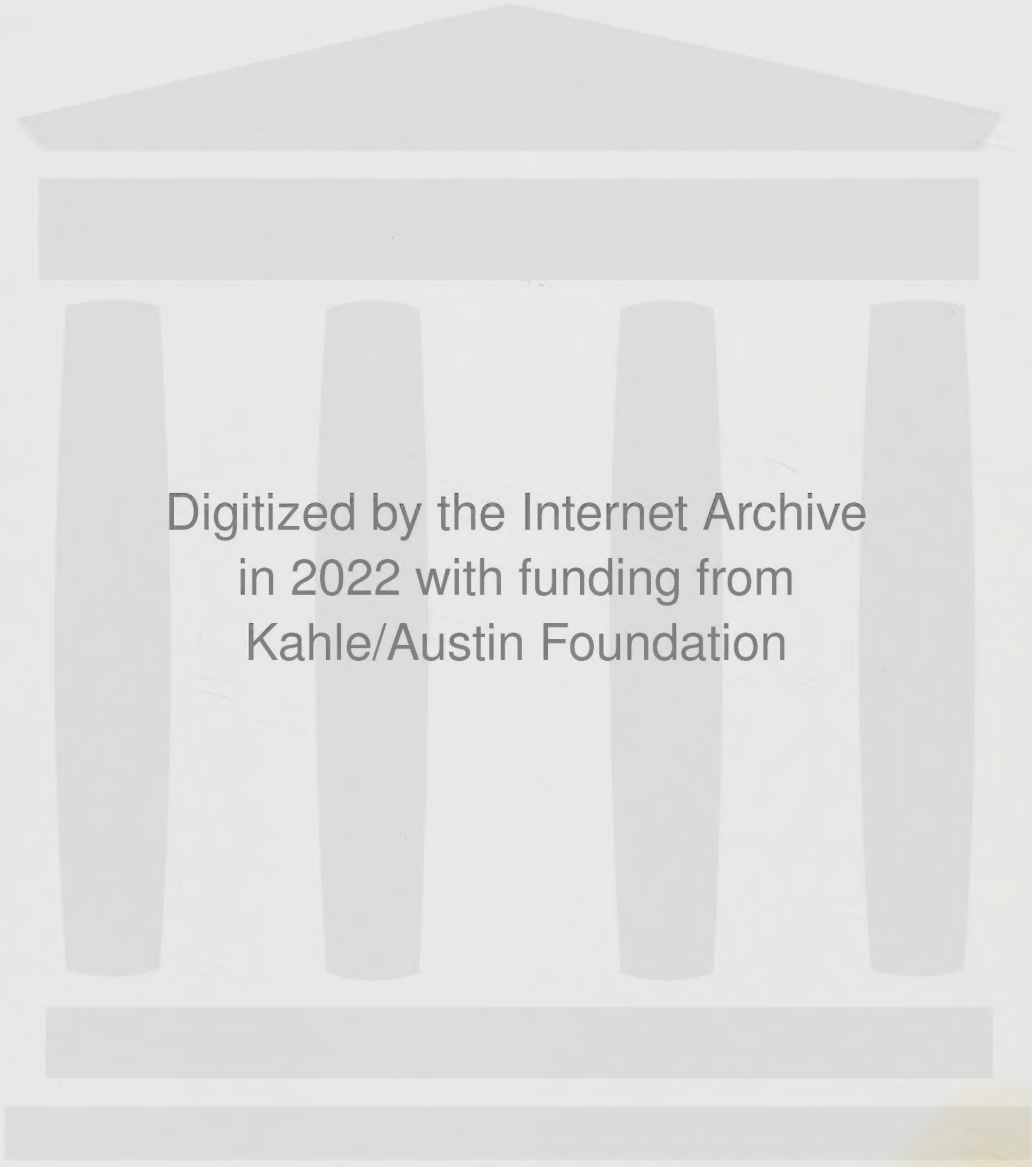
Solutions Manual

Introduction to Computer Theory

Second Edition

Daniel I. A. Cohen

Prepared By
Chanah Brenenson



SOLUTIONS MANUAL

TO ACCOMPANY

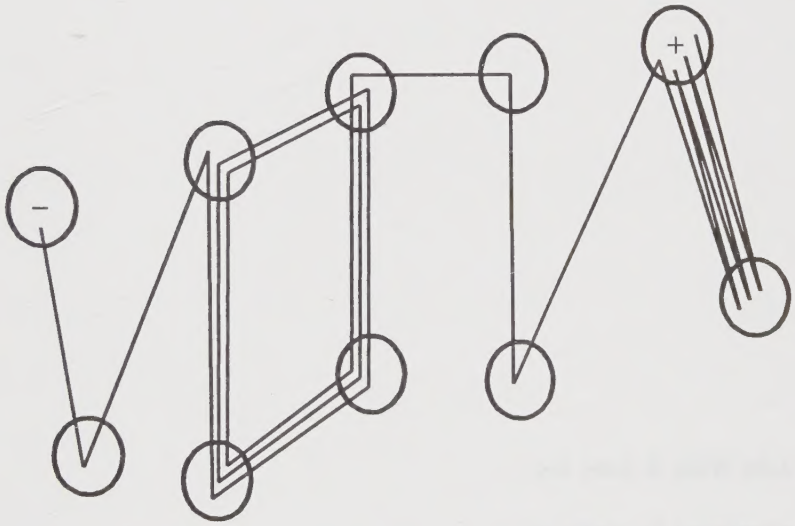
INTRODUCTION TO COMPUTER THEORY

SECOND EDITION

DANIEL I. A. COHEN

Hunter College
City University of New York

Prepared by
CHANAH BRENENSON



John Wiley & Sons, Inc.
New York • Chichester • Brisbane • Toronto • Singapore • Weinheim

CONTENTS

CHAPTER 1

The main goal of this chapter is to give at least a vague idea of what Computer Theory might be and why it might be studied. This course is often required of all majors and so they enrol without even the usual pretense of acting volitionally. Unlike courses offering programming skills this one is not primarily vocational -- it is, of course indirectly of benefit to the art of programming, but it's main thrust is intellectual enrichment. Plumbers may not know exactly who invented plumbing but they do know why, and perhaps how, it came about. For all their mental achievements programmers generally have no basis for answering this same question about their profession.

Computer Science majors are very suspicious about being slipped extra theoretical math courses under the guise of "this will be good for you somehow." To allay this fear we emphasize that this is not a course in mathematics as such but a course in mathematical models, a friendly technique that arises in social science and linguistics and other unthreatening subjects. Of course, this is a course in mathematics. Ha ha. But the joke is more on the mathematical community who have no idea what beautiful results lie in this disenfranchised corner of their discipline -- this despite the fact that its grandfathers, Hilbert, Gödel, Church, von Neuman et al., are held in reverence for their other, more "pure" contributions.

In general the format of this text is in the classical theorem-proof-theorem-proof model, but unfortunately most students who need to know the material in this course are (functionally) mathematically illiterate. Even if they have been forced to take a course in Calculus or Linear Algebra they have almost no conception of what it means to *prove* anything. Worse yet, they are afraid of ever being responsible for constructing proofs themselves. This book is realistically sensitive to this deplorable situation and it takes great pains to assuage such anxiety. This is a gentle introduction to the art of providing convincing explanations (i.e. proofs) for proposed propositions that may even usefully serve as a primer for those math majors who suffer from proof-anxiety.

For the student who is more sophisticated mathematically there are other texts which cover approximately this same material in many fewer pages. Some of these books are excellent. We can recommend, without reservation, the following list from our own experience: Hopcroft and Ullman, Lewis and Papadimetriou, Mana, Harrison, Kain, Minsky, etc. No claim is made that this list is exhaustive and there may be many more texts of recent vintage of which we are unaware.

Students should be informed about these books as possible other sources of reference, though they should be cautioned about the vast disparity that may exist in notation and be alerted to possible subtle differences in definitions. If any of these books are approached they should be read from page one.

Before our current book was released there was some confusion as to whether the books listed above were texts for graduate courses or undergraduate courses. There is a strong

feeling that Computer Theory should be taught to undergraduates, especially context-free grammars a topic and notation that appears importantly in other undergraduate courses such as compilers, AI etc. However, a graduate level text does not become an undergraduate level text simply by reading it slowly.

Even though it is certainly true that the books listed above are more advanced this does not mean that they contain everything of value that is in our text. There are original contributions to be found among these pages, not original theorems but some new ideas in the proofs, some original problems, some useful examples and many instances where an idea that is only vaguely sketched in other sources is made explicit here. (Sometimes making an idea explicit can seem painful or frightening, but there is always what seems to us a good justification for giving a student more than just the impression that she knows what is going on.)

If a student finds this book easy, let her read through it quickly. There are 600 pages here and no one is going to read it over night, but we believe strongly that a bright student should be encouraged to set her own pace. For the strongest foundation in this subject one might start by finishing this book and then read one (or all) of the texts mentioned before. A student who approaches the more advanced books already knowing most of the material in them (our material) will not only have an easier time reading but will develop an appreciation for the power of the advanced mathematical notation for its efficiency and clarity of abstraction.

We generally avoid advanced mathematical notation for two reasons: psychological and pedagogical. The fear and alienation engendered by a page of symbols more than neutralizes the efficacy of introducing the notation as it defeats the purpose of education. It is also true that far too often a student (or a professional mathematician) gets a false sense of confidence in the material by merely having followed the manipulation of the symbols on the paper. We have too frequently heard, "I followed everything you did in class but I couldn't do the problems." It is our belief that the hasty adoption of abstract notation is largely responsible for this predicament. Notation should be introduced only when a student feels that her thoughts are flying faster than her ability to express them. At that point students should be encouraged to invent their own personalized shorthand bearing their own private meanings. Only as a final stage, when the need for a standardized medium of communications becomes clear, should symbolism be introduced.

The important mistake not to make is to feel that a subject becomes more rigorous, more precise or more mathematical because it is riddled with symbols. Greek letters are no more exact than the English words that defined them. Many shoddy proofs go undetected because they are disguised in opaque notation. If an argument is rigorous in symbols it remains rigorous when expressed in English sentences. What it takes us 600 pages to do here could have been condensed to fifty -- with complete loss of educational value.

CHAPTER 2

It is important to distinguish for the student what material from this chapter will be required in the future and what will not. Obviously the silly languages L_1, L_2, L_3 and L_4 are not important for the rest of the book, though they do reappear briefly in Chapter 3. On the other hand the language **PALINDROME** must become a member of our immediate family. A big fuss should be made about the null word and the Kleene star. We have found that Theorem 1 is a good introduction to the concept of proof. We challenge the students to provide "an explanation of why this is true that would convince someone who didn't already believe it." There is usually some obliging student in the class who doesn't understand the result and when the other students try to convince him that it is not only true but painfully obvious he is sometimes able to shake their confidence in their ability to articulate their thoughts. This is the beginning of the understanding of what proof is all about.

I have assumed throughout the book that the readers are acquainted with standard elementary computer programming concepts such as the notion of formal strings of characters, concatenation, prime numbers, factorials, symbolic logic, etc. Other than that the text is so self-contained that it defines many terms the student is already familiar with. What could be the harm?

It is our habit to give exactly 20 questions per chapter many of which have multiple parts. Problem # 20 is usually more theoretical in nature and often quite challenging. Sometimes #20 is so difficult that it contains a hint that gives virtually the whole problem away. This #20 is not too hard.

Chapter Two

1. S* has 4 2-letter words, 8 3-letter words and 2^n n-letter words.

2. 5 words of length 4; 8 words of length 13; 13 words of length 6. In general, the number of words of length n is the sum of the number of words of length (n-1) and the number of words of length (n-2), a Fibonacci sequence.

3. length 2: *ab, ba*; length 4: *abab, abba, baab, baba*;
length 6: *ababab, ababba, abbaab, abbaba, baabab, baabba, babaab, bababa*;
S* contains no words of odd length, and no word containing a triple letter substring. The smallest word not in this language is *a*.

4. The string *abbba* is not in the language.
length 1: *a*; length 2: *aa, ab, ba*; length 3: *aaa, aab, aba, baa*;
length 4: *aaaa, aaab, aaba, abaa, abab, abba, baaa, baab, baba*;
length 5: *aaaaa, aaaab, aaaba, aabaa, aabab, aabba, abaaa, abaab, ababa, abbaa, baaaa, baaab, baaba, babaa*;
length 6: *aaaaaa, aaaaab, aaaaba, aaabaa, aaabab, aaabba, aabaaa, aabaab, aababa, aabbaa, abaaaa, abaaab, abaaba, ababaa, ababab, ababba, abbaaa, abbaab, abbaba, baaaaa, baaaab, baaaba, baabaa, baabab, baabba, babaaa, babaab, bababa*;
This language is all the strings of *a*'s and *b*'s where each *b* has its own *a* on its left or right. No two *b*'s may share the same *a*.

5. Factoring gives: *aa|baa, baa|aba|aa, baa|aa|aba|baa|aa*
No word can be factored in more than one way (look for the first odd length substring of *a*'s and/or the position of the first *b*). No word can have an odd number of *a*'s because each factor contains two.

6. 1 (xxx) and 8 (xx) - 9 arrangements (nine choose one)
3 (xxx) and 5 (xx) - 56 arrangements (eight choose three)
5 (xxx) and 2 (xx) - 21 arrangements (seven choose five)
total 86 arrangements

7. (i) To determine if a string is palindrome:
   1) If the length (string) < 2 then the string is palindrome, otherwise continue.
   2) Compare the first letter(s) with the reverse of the last letter(s). If they match then delete them both and repeat step 1.
   Since x is palindrome, x = reverse(x). Following the algorithm to test x^n, two copies of x are repeatedly deleted (one from each end, because they match) until the string is reduced either to Λ (when n is even) or to x (when n is odd). Both of which are palindrome, therefore x^n is palindrome.

   (ii) If a string is palindrome, then deleting an equal number of letters from the both ends leaves a palindrome word. Hence, removing the front and rear copy of y from y^3 which is palindrome, leaves the palindrome string y.

(iii) Continuing the proofs above applied now to the palindrome string z^n, repeatedly remove two copies of z at a time, one from either end, until if n is odd only a palindrome string z remains. If n is even, then stop shrinking the string when zz remains. Note that any palindrome can be viewed as a string concatenated with its own reverse. zz is palindrome and zz = z reverse(z), implies that z = reverse(z) and z is palindrome.

(iv) The are four palindromes of length 3; *aaa, aba, bab, bbb* and of length 4; *aaaa, abba, baab, bbbb*. For each odd length palindrome insert another copy of the middle letter adjacent to it to make an even length palindrome.

(v) By using the algorithm in part one, we can reduce any palindrome to a central core of one or two letters. On {a, b}, there are as many palindromes of length 2 (aa, bb) as there are of length 1 (a, b). To make palindromes of length 2n, choose a core of length 2, and then make n-1 choices for the letters to the left which determine the letters to the right. To make palindromes of length 2n-1, choose a core of length 1 and then make n-1 choices for the other letters. In each case n choices determine the word. Since there are two choices for letters, there are 2^n palindrome words of length 2n or 2n-1.

8. Here is an algorithm for finding z.
1) If length(x) = length(y). Then we have a palindrome of even length, so it must be of the form z reverse(z), and x=y=z.
2) If the strings are of different lengths then the longer one either begins or ends with the shorter one. That is if length(x) < length(y) then xy = xsx, where y = sx, and symmetrically if length(x) > length(y) then xy = ysy where x = ys. In both cases the substrings s are palindrome (because they are each the center of a palindrome). In addition, the longer string L is palindrome that is the concatenation of two palindromes, so repeat algorithm on L.

9. (i) $S \subset T$, so $S^* \subset T^*$. *bbbb* is the only word in T but not in S. However, *bb* $\in$ S so *bbbb* $\in S^*$ and $T^* \subset S^*$, therefore $S^* = T^*$.
(ii) $S \subset T$, so $S^* \subset T^*$. However there is no way to generate *bbb* with the elements of S, so $S^* \neq T^*$.
(iii) $S^* = T^*$ even though $S \neq T$ only when all the words in the symmetric difference can be generated by words in the intersection.

10. No changes in the equalities and inclusions.

11. (i) $(S^+)^*$ includes Λ even is S does not, so $(S^+)^* = S^*$. $S^* = S^{**}$ by Theorem 1.
(ii) There can be no factor in $(S^+)^+$ that is not in S^+, $(S^+)^+ \subset S^+$. In general, any set is contained in its positive closure, $S^+ \subset (S^+)^+$. Therefore $(S^+)^+ = S^+$.
(iii) Yes. If $\Lambda \in S$, then $S^* = S^+$. $(S^+)^* = (S^*)^* = (S^*)^+$. If $\Lambda \notin S$ then $\Lambda \in S^*$ anyway, and $(S^*)^+ = (S^*)^* = S^* = S^+ \cup \Lambda = (S^+)^*$.

12. No words in S^* contain an odd number of b's (each factor contributes two if any), so none of the examples is in the set.

Chapter Two

13. This is the same as saying that the language L would allow all concatenations that did not produce squares. First observe that $\Lambda = \Lambda\Lambda$, so Λ cannot be in the language. Consider $w_1 \neq w_2$ and $w_1w_2 \in L$. Let $w_3 = w_1w_2$, since $\Lambda \notin L$, $w_3 \neq w_2$, so $w_4 = w_2w_3 \in L$ where $w_4 \neq w_1$, finally let $w_5 = w_1w_4 \in L$. However, $w_5 = w_1w_2w_1w_2 = w_3w_3$ which is a square so $w_5 \notin L$.

14. $(S^{**})^* = (S^*)^* = S^*$ by Theorem 1. It is often bigger than S.

15. (i) no.
(ii) yes. $T = S + \{w\} \rightarrow w \in T \rightarrow w \in T^*$ and $T^* = S^* \rightarrow w \in S^*$

16. Let $S = \{aaa\}$, S^* has one six-letter word and no seven-letter words nor eight-letter words. However it is impossible for S^* for any S to contain more six-letter words than twelve-letter words because for every six-letter word w there is a twelve-letter word ww in S^*.

17. (i) All words over $\Sigma = \{a, b\}$ of even length.
(ii) $S = \{ aaa, aab, aba, abb, baa, bab, bba, bbb \}$
(iii) All strings of *a*'s and *b*'s except Λ

18. (iv) S^* and T^* both represent the set of all strings of a's and b's. Therefore T must include at least the words *a* and *b*, which is the set S.
(v) $S = \{ a, bb \}$, $T = \{a, aa, bb \}$.

19. The word *abaaba* disproves the algorithm.

20. Since T is closed and $S \subset T$, any factors in S concatenated together two at a time will be a word in T. Likewise concatenating factors in S any number of times produces a word in T. That is any word in S^* is also in T. However we are given that $T \neq S^*$ so T contains some words that are not in S^*. We can conclude that S^* is a proper subset of T, in other words S^* is smaller than T, and in symbols $S^* \subset T$.

CHAPTER 3

Computer Science majors are familiar with the term "recursive programming" and this will be very helpful with the concept of recursive definitions. The same idea of self-reference is not carried over directly to "recursively enumerable" since that is a back formation from "recursive functions." Students should be told not to worry that the mathematical examples of polynomial and derivative set a precedent of a strong connection between this course and mathematical analysis. These examples are just too perfect not to have been included. if Calculus were properly taught the concept of recursive definition would have been introduced there.

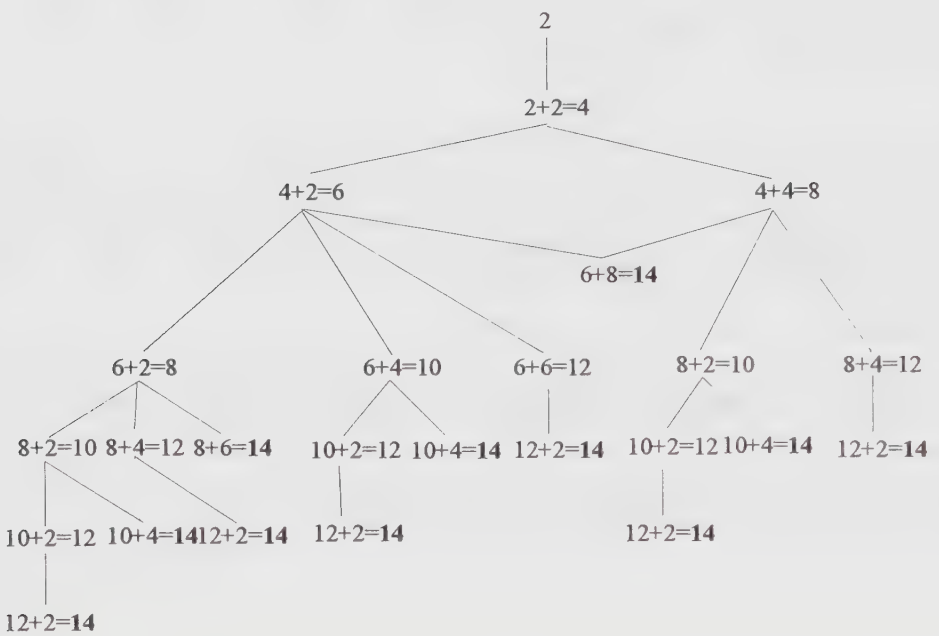
Theorems 2, 3 and 4 are simply illustrations of the theorem-proving power of recursive definitions. They also serve as a pre-introduction to parsing. The discussion of well-formed formulas is also tangential to the material in the text but computer students should have mastered this useful tool by this time in their education.

Why have we defined recursion but avoided all reference to proof by mathematical induction. The answer is simply that we hate proofs by mathematical induction. They verify the truth of a result without shedding any understanding of why the claim is true or where it came from. It is a fine tool for research but a poor one for education. It is employed by lazy authors who wish to compress as many results into as short a space as possible. The benefit of this in a textbook is a mystery to me.

Chapter Three

1. Rule 1: x is in L_1
   Rule 2: If w is in L_1 then so is wx.

2. There are eleven ways to prove that 14 is even.



3. $2 \Rightarrow 4 \Rightarrow 8 \Rightarrow 16 \Rightarrow 32 \Rightarrow 64 \Rightarrow 96 \Rightarrow 100$: 8 steps.
   To show that 2n is in EVEN: Keep adding to itself the largest number in the set until the first result that is greater than or equal to n. If the result equals n then adding it to itself gives 2n. If the result is greater then n, add to it the largest value in the set that will not bring the total above 2n. Continue this procedure until adding that value gives the result 2n.

4. We must show 1) that all positive even numbers can be produced from this definition and 2) that the definition produces nothing else.
   1) We have 2 and 4 and 2+4=6, 4+4=8, 6+4=10, 8+4=12, ..., so if there is an even number that cannot be produced from this definition it is large. Let us suppose that there are some such numbers and let us call the smallest of them n, (that is n is the smallest even number that cannot be produced from this definition). But that means that all smaller even numbers can be produced by this definition, and in particular, that n-4 can be produced. So to produce n, we apply rule Rule 2 to n-4. Since there is no smallest even number that cannot be produced from this definition, all even numbers can.
   2) How can we produce an odd number from this definition? If we add an even number to some arbitrary integer n the resulting sum has the same parity as n. The increment in the definition, 4, is even. Both of the elements known to be in the set (2 and 4) are even.

Therefore application of Rule 2 will never alter the parity, and all numbers in the set will be even.

5. We can make up any rules as long as they do not change the parity and providing that they cover all cases. If Rule 1 lists the smallest n even numbers and the increment in Rule 2 is 2n (the next highest in the set), both conditions are met. There are infinitely many such lists of rules.

6. Use the following recursive definition of EVEN:
   Rule 1: 2, 4, 6, 8 and 10 are in EVEN.
   Rule 2: If x is in EVEN then so is x + 10.
   Since adding 10 never changes the last digit of the number, all numbers in the set end in 0, 2, 4, 6, or 8. This defintion satisfies the conditions in the answer to Problem 5, so it will not allow change of parity and covers all cases.

7. Rule 1: Any number is in POLYNOMIAL.
   Rule 2: The variables x and y are in POLYNOMIAL.
   Rule 3: If a and b are in POLYNOMIAL, then so are a+b, a-b, (a) and ab.

8. (a) 3 is in ALEX (Rule 1). x is in ALEX (Rule 2). 3x is in ALEX (Rule 3). 2 is in ALEX (Rule 1). x+2 is in ALEX (Rule 3). Therefore by ALEX Rule 1, x+2 and 3x are both in ALEX. Since x+2 is in ALEX, by Rule 2a, so is (x+2) and by Rule 2g, so is $(x+2)^{3x}$.
   (b) Elementary calculus contains rules for differentiating sums, differences, products, quotients, and exponetiations of differntiable functions. By the rules given here, if these functions are polynomials, they are composed only of elements that are differentiable functions and are therefore differentiable.
   (c) No, when it is a matter of polynomials, the permissible functions are all defined in the other rules.

9. (((((3)x)+7)x)-9) contains only two products: the product of 3 and x and the product of (((3)x)+7) and x.

10.

$x^2 = x \cdot x$	(one step)	$x^{12} = x^4 \cdot x^8$	(four steps)
$x^3 = x \cdot x^2$	(two steps)	$x^{13} = x \cdot x^4 \cdot x^8$	(five steps)
$x^4 = x^2 \cdot x^2$		$x^{14} = x^2 \cdot x^4 \cdot x^8$	
$x^5 = x \cdot x^4$	(three steps)	$x^{15} = x \cdot x^2 \cdot x^4 \cdot x^8$	(six steps)
$x^6 = x^2 \cdot x^4$		$x^{16} = x^8 \cdot x^8$	(four steps)
$x^7 = x \cdot x^2 \cdot x^4$	(four steps)	$x^{17} = x \cdot x^{16}$	(seven steps)
$x^8 = x^4 \cdot x^4$	(three steps)	$x^{18} = x^2 \cdot x^{16}$	
$x^9 = x \cdot x^8$	(four steps)	$x^{19} = x \cdot x^2 \cdot x^{16}$	(six steps)
$x^{10} = x^2 \cdot x^8$		$x^{20} = x^4 \cdot x^{16}$	(five steps)
$x^{11} = x \cdot x^2 \cdot x^8$	(five steps)	$x^{21} = x \cdot x^4 \cdot x^{16}$	(six steps)

Chapter Three

$x^{22} = x^2 \cdot x^4 \cdot x^{16}$
$x^{23} = x \cdot x^2 \cdot x^4 \cdot x^{16}$ (seven steps)
$x^{24} = x^8 \cdot x^{16}$ (five steps)
$x^{25} = x \cdot x^8 \cdot x^{16}$ (six steps)
$x^{26} = x^2 \cdot x^8 \cdot x^{16}$
$x^{27} = x \cdot x^2 \cdot x^8 \cdot x^{16}$ (seven steps)
$x^{28} = x^4 \cdot x^8 \cdot x^{16}$ (six steps)
$x^{29} = x \cdot x^4 \cdot x^8 \cdot x^{16}$ (seven steps)
$x^{30} = x^2 \cdot x^4 \cdot x^8 \cdot x^{16}$

11. Forbidden substrings of length 2:

++	-+	*+	/+	(+	)(
+*	-*		/*	(*	
+/	-/	*/	//	(/	
+)	-)	*)	/)	()	

12. Forbidden substrings of length 3 that do not contain shorter frobidden substrings:

+--	---	*--	/--	(--	***

13. Not without very careful stipulation of many contingencies. It is much more practical to allow them to accumulate and eliminate redundancy in (or reduce) the configuration later.

14. (i) Rule 1: Any letter is in Prep-Calculus
Rule 2: If x and y are in Prep-Calculus so are (x), ~x, x∧ y, x∨ y, x→ y.

(ii)

)(	()	)~	
	(∨	(∧	(→
~)	~∨	~∧	~→
∨)	∨∨	∨∧	∨→
∧)	∧∨	∧∧	∧→
→)	→∨	→∧	→→

15. (i) Rule 1: a, b and Λ are in PALINDROME.
Rule 2: If x is in PALINDROME, then so are xx, axa, and bxb.
(ii) Rule 1: aa and bb are in EVENPALINDROME.
Rule 2: If x is in EVENPALINDROME, then so are xx, axa, and bxb.

16. (i) Rule 1: 1 is ODD.
Rule 2: If x is in ODD, so is x+2.
(ii) Rule 1: 1, 2, 3, 4, 5, 6, 7, 8 and 9 are in DIGITS.
Rule 2: If x and y are in DIGITS then so are x0 and xy.

17. This is the set of positive rational numbers. Starting with Rule 1 and applying addition, we get the positive natural numbers. Then applying x/y, we get all the fractions.

18. Def 1. Rule 1: 1 is in POWERS-OF-TWO.
Rule 2: If x is in POWERS-OF-TWO, so is 2 × x.
Def 2. Rule 1: 1 and 2 are in POWERS-OF-TWO.

Chapter Three

Rule 2: If x and y are in POWERS-OF-TWO, so is $x \times y$.
The proof is rule 2 of definition 2.

19. (i) Rule 1: Λ is in EVENSTRING.
Rule 2: If w is in EVENSTRING, so are waa, wab, wba and wbb.
(ii) Rule 1: a and b are in ODDSTRING.
Rule 2: If w is in ODDSTRING, so are waa, wab, wba and wbb.
(iii) Rule 1: aa is in AA.
Rule 2: If w is in AA, so are aw, wa, bw and wb.
(iv) Rule 1: Λ, a and b are in NOTAA.
Rule 2: If w is in NOTAA, so are wb, and wba.

20. (i) xyz can be any word in the set. Rule 1 gives us an initial element. So far 3-PERMUTATION = {123}. Let xyz = 123 and applying rule 2 gives zyx = 321 and yzx = 231. So far our set is {123, 321, 231}. Now let xyz be a different element in the set; xyz = 321. Applying rule 2 on xyz = 321 gives zyx = 123 and yzx = 213. Only the second string is a new element, when added makes {123, 321, 231, 213}. Apply rule 2 on xyz = 231 gives zyx = 132 and yzx = 312. Both of these can be added to our set, hence {123, 321, 231, 213, 132, 312 }. Applying the rule to the remaining three elements , 213, 132 and 312 does not generate any new elments so we have verified the six 3-PERMUTATION's.
(ii) Of the 24 possible arrangements of the digits 1, 2, 3, 4, only these 8 can be obtained from this definition by the same method detailed in part (i):
1234 4321 2341 3214 1432 3412 4123 2143

CHAPTER 4

Throughout the text we stick to the alphabet {a,b} instead of the more logical alphabet {0,1}. We are not the only text to do this. We find that 0 and 1 are hopelessly overworked in Computer Science already: boolean, binary, bit string etc. We already use the plus sign for disjunction and for union; if it sat between bit strings the temptation to mistake it for addition would be too great.

In the text we emphasize the distinction between a language defining expression and the language it defines -- perhaps we over emphasize this. In class we are certainly much more careless. The fact that **r*** really means (**r**)* is another example of the confusion of word and object that should be made clear. In this chapter we begin the theme of finite versus infinite. The only infinite that we can really deal with is that which can be made finite. In future chapters this motif will echo furiously.

Although it is not so mentioned Theorem 5 is proved by constructive algorithm and this could be pointed out. The end-of-proof mark does not appear on Theorem 5 until after the algorithm is illustrated once. Students might note that the inclusion of this illustration "inside" the proof is for the purpose of being sure that the description given of the algorithm is clearly understood. In writing a proof it is so important that the explanation be understood that any clarifying discussion may be included.

The distinction we belabor between having a well-defined language (by regular expression say) and actually *knowing* what the language is, is crucial to understanding that the concept of *meaning* is not (yet) properly part of mathematics. Hence there are no algorithms for it and no theorems about it. This is what it is that makes Artificial Intelligence artificial.

The complicated regular expression for the language **EVEN-EVEN** is worth covering since this language will appear many times throughout the text. It will be revisited every time we introduce a new characteristic of languages.

Chapter Four

1. The language associated with $(r_1 + r_2)r_3$ is the language of all words made up of an initial factor from the language associated with either r_1 or r_2 and a second factor from the language associated with r_3. The language associated with $r_1r_3 + r_2r_3$ is the language of all words of the following two types: a factor from the language associated with r_1 followed by a factor from the language associated with r_3 or else a factor from the language associated with r_2 followed by a factor from the language associated with r_3. But these are merely two ways of saying the same thing: all words in either language must end with a factor from the language associated with r_3, and the first factor must be from languages associated with either r_1 or r_2.

2. **(aaa + b)*** is the most obvious, or **((aaa)* b*)***

3. $(a + b)^* (s_1 + s_2 + s_3 + s_4) (a + b)^*$

4. **a* b a* b a* (b + Λ) a***

5. (i) **(a + b)* (aa + bb)**
   (ii) **(a + b)* (ab + ba) + a + b + Λ**

6. **(b + Λ)(ab)* aa (ba)*(a + Λ) + (a + Λ)(ba)* bb (ab)*(a + Λ)**

7. **a* ((b + bb) aa*)* (b + bb + Λ)**

8. Building on previous results, we have the language *aaa* but no triple *b*, to which we add its a-b complement: the language with bbb but no triple a.
   a*((b + bb)aa*)*(b + bb + Λ) aaa a*((b + bb)aa*)*(b + bb + Λ)
   + b*((a + aa) bb*)*(a + aa + Λ) bbb b*((a + aa)bb*)*(a + aa + Λ)

9. (i) **b*a***
   (ii) **(a + ba)* b* + b* a(a + ba)* (Λ + b)**

10. **b* (ab*ab*a)* b***

11. (i) **a* (b(bb)*aa*)* b(bb)* a* + a***
    (ii) EVEN-EVEN **(b + ab(bb)*a)** EVEN-EVEN, where EVEN-EVEN stands for the regular expression **(aa + bb + (ab + ba)(aa + bb)*(ab + ba))***
    (iii) EVEN-EVEN **(ab + ba)** EVEN-EVEN

12. (i) **(a+b)* ab (a+b)*** is a regular expression for the language of all words containing the substring ab: *arbitrary ab arbitrary*. **(a+b)* a (a+b)* b (a+b)*** is a similar regular expression of the form *arbitrary a arbitrary b arbitrary*. The difference is the additional requirement of some string between the mandatory a and b. So to show equivalence, we need to show that for all choices for the middle arbitrary string, *a*

arbitrary b contains an *ab*. In all cases, once an *ab* is located then the substrings to the left and right each become the new choices for the front and back *arbitrary*'s. If the middle *arbitrary* = Λ, then the expressions are clearly identical. If the middle *arbitrary* contains the mandatory *ab* then just redefine the origins of the substrings on either side. Otherwise the middle *arbitrary* does not contains the mandatory *ab*. In this case the substring must either start with a *b* or end with an *a*, either way an *ab* is formed by using the mandatory a or b that surround the middle arbitrary string. Therefore the two expressions are equivalent.

(ii) **(a+b)* ab (a+b)*** represents all the strings containing the substring *ab*. Compared with all possible strings of a's and b's, there are three types of strings missing: strings of all a's, strings of all b's, and strings in which all b's precedes all a's. Since **b*a*** contributes all these missing strings, the union is **(a+b)***.

(iii) Substituting **(a+b)*** for the expression in the brackets gives an equivalent expression as proved in part (ii). Now the resulting expression is exactly the same as above.

(iv) We can keep making substitutions of this kind, so there are infinitely more beasts in the cave.

13. (i) Every word in S* is the concatenate of some finite number of words from S. Let us say w is the concatenation of *k* words from S, then w is also an element of S^k. Therefore if we take the union of S^k for all *k*, we get S*.
(ii) Obvious.

14. No. Let $L = (a+b)^+$. L and L* differ by only Λ, but L^2 and L* differ by Λ, *a*, and *b*.

15. (i) **(ab)*a** alternates the a's and b's and tacks on an a at the end. **a(ba)*** starts with an a and then alternates the b's and a's. Both define odd length strings the no double letters, where every word begins and ends in a.

(ii) **(a*+b)*** can not generate more words than **(a+b)*** which already generates everything. Moreover **a*** can just as easily be generated by **(a+b)***.

(iii) Again **(a*+b*)*** can not generate more words than **(a+b)*** and either factor can just as easily be generated by **(a+b)***.

16. (i) Λ* is all the words produced by concatenating factors of length zero. There is only one such word, Λ.

(ii) **a*b**is the language of all words that have exactly one *b* which is the last letter. Its Kleene closure, **(a*b)***, is the language of all words that do not end in *a*. Concatenating the **a*** permits words to end in *a*. So **(a*b)*a*** gives all words over {a, b}. Symmetrically, **ba*** is all the words that have one *b*, the first letter. Its closure is all words that begin with *b*. Concatenating the initial **a*** gives all words over {a, b}, so the regular expressions define the same language.

(iii) Similar to the previous problem only this time both expressions define the language of all words over {a, b} where the *b*'s occur in clumps divisible by three.

Chapter Four

17. (i) **(a+bb)*aa** describes the language of string where *b*'s occur in even clumps and every word ends in double *a*. Apply closure only adds the null word, because we can already use the factor inside the parentheses to make any strings of *aa*.

(ii) Both expressions define all strings of only *a*'s. **(aa)*** gives the even length strings and then **(Λ+a)** gives the option of having no *a*'s at all or an odd number.

(iii) **a(aa)*** gives all strings with an odd number of *a*'s. **(Λ+a)** gives the option of even length strings of *a*'s. Every word must have one final *b*. **+b** allows for the string with no *a*'s. This is the same as any number of *a*'s if any followed by a single *b*; **a*b**.

(iv) **a(ba+a)*b** describes words that begin with *a* and end with *b*. In the body of any word *b*'s are always followed by an *a* (which prevents clumps of *b*'s), however the clumps of *a*'s are unrestricted. The associated language contains all strings where each b is surrounded by at least one *a* on either side and that ends in *b*. **aa*b(aa*b)*** describes the same language where each *b* is preceded by at least one *a*.

(v) The three cases are clear in the first regular expression: Λ or all words that begin with *a* or all words that contain the substring *aa*. Since the entire second expression is starred, it produces Λ. The second factor of the second expression **(ab*)*** generates all words that begins with *a*. If the first factor, **(b*a)*** is taken at least once then the last one juxtaposed with **ab*** produces the desired double *a*; ...b*aab*....

18. (i) all words that end in either *a* or *bbbb*.

(ii) all words that do not begin with *b* and in which *b*'s appear in clumps of even lengths.

(iii) Λ and all words in which both *a*'s and *b*'s occur in odd clumps and that start with *a* and end with *b*.

(iv) Λ and all words in which both *a*'s and *b*'s occur in odd clumps and that end with *b*.

(v) Λ and all words in which both *a*'s and *b*'s occur in odd clumps.

(vi) all words of even length such that (except for Λ) a's occupy all even positions.

19. (i) R = SR+T defines R as containing all the words in T and also all words composed of an initial part from S concatenated with another words from R. This permits us to continue concatenating factors from S as many times as we like, but we are still required to finish the word with a factor from R. The only type of factor from R that does not entail this kind of recursion is a factor from the language T. So we have any number of factors from S followed by a factor from T, or we have just a factor from T, equivalently S*T.

(ii) R=S*T defines R as words with any number (if any) factors from S followed by a single factor from T. If the star operator is taken for zero, we have for R any word in T. Suppose the star operator is taken once, then we have exactly one word from S followed by one word from T. But the word from T is itself a factor from R, so we can see that this word is in the set SR. Consider what happens if we take the star more than once; the star operates on S, so we have SR, SSR, SSSR,... all in the set S*T. In fact, we established S^+T in this way. The difference between S*T and S^+T is that only S*T contains T. But that means that S*T must equal exactly SR+T.

Chapter Four

20. (i) It is easiest to understand this in terms of the recursive definition of regular expressions. Any letter of the alphabet is itself a regular expression. The rule for forming regular expressions apply to these and to all more complicated regular expressions. Each regular expression describes a set. Equivalent expressions describe equivalent sets. The equivalences are not disturbed by taking unions, products, or Kleene closures. This means that any set must be considered a building block in the construction of other sets.

(ii) With $R = (ba^*)^*$ and $S = (b+\Lambda)$, we have

$$(R^*S)^* R^* = R^* (SR^*)^*$$

$$(((ba^*)^*)^*(\Lambda +b))^* ((ba^*)^*)^* = ((ba^*)^*)^* ((\Lambda +b)((ba^*)^*)^*)^*$$

Both sides reduce to $(ba^*)^*$, (which is Λ + all words that begin with b).

CHAPTER 5

It is the policy of this text to define all machines in terms of their pictorial representation and base all proofs about the machines on the ability to argue from the pictures. It is therefore important that this ability be developed as early as possible. This chapter contains no theorems, only a definition and illustrations, yet it is very important to begin the ability to reason from pictures at this stage. Students should be reassured that although there is also a non-pictorial tabular representation of these machines the pictorial representations are every bit as sound a mathematical object.

All the pictures in this chapter of FA's are planar graphs whose edges are drawn without crossing. Students sometimes need to be informed that if the pictorial representation requires crossed edges this is perfectly fine so long as they are drawn so that no confusion arises.

Problem #20 is actually an interesting theorem and if it is not assigned it should be covered in class. It is another instance of the struggle of finite versus infinite.

Chapter Five

1.

p.56	a	b
x -	y	z
y	x	z
z +	z	z

p.58	a	b
L -	R	R
R +	R	R

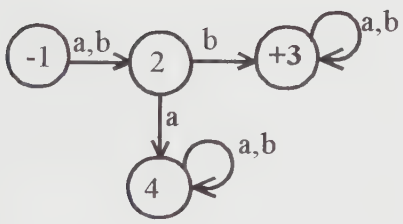
p.58	a	b
S ±	S	S

p.63	a	b
1 -	2	3
2	4	3
3	2	4
4 +	4	4

p.64	a	b
1-	2	2
2	3	3
3	4	5
4	4	4
5 +	5	5

p.69	a	b
1 ±	3	2
2	4	1
3	1	4
4	2	3

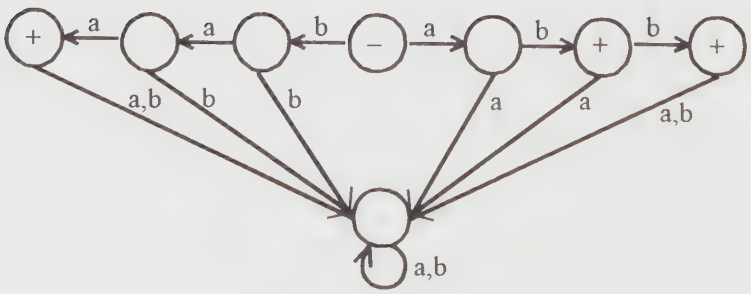
2.



	a	b
1 -	2	2
2	4	3
3 +	3	3
4	4	4

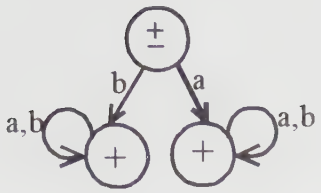
(a+b)b(a+b)*

3.



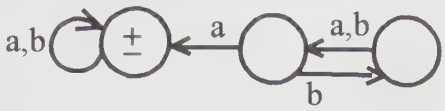
Chapter Five

4. (i) (ii)

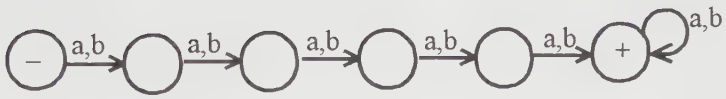


If all the states in an FA are final then automatically all strings are accepted because no matter where the string ends its accepted.

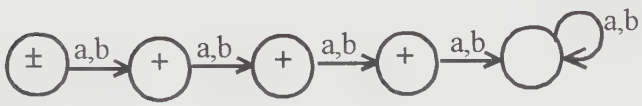
(iii) Not necessarily, for the non-plus states may be unreachable as in the following FA:



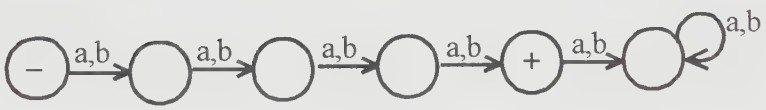
5. (i)



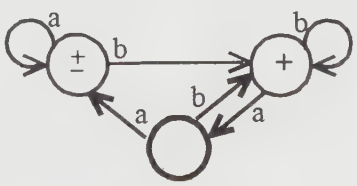
(ii)



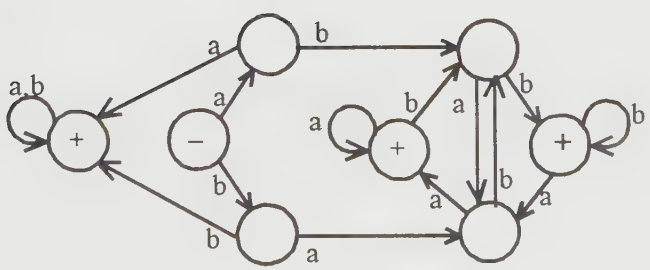
(iii)



6.

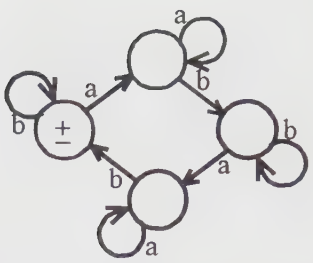


7.

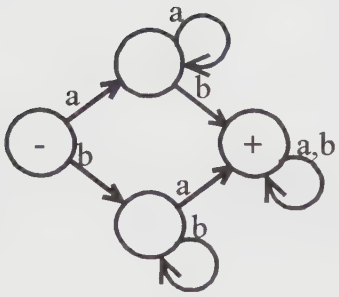


Chapter Five

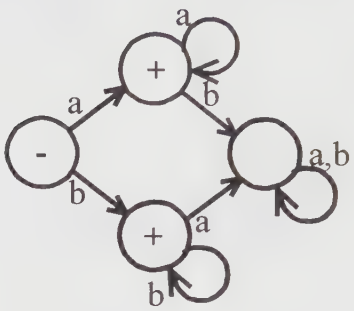
8.



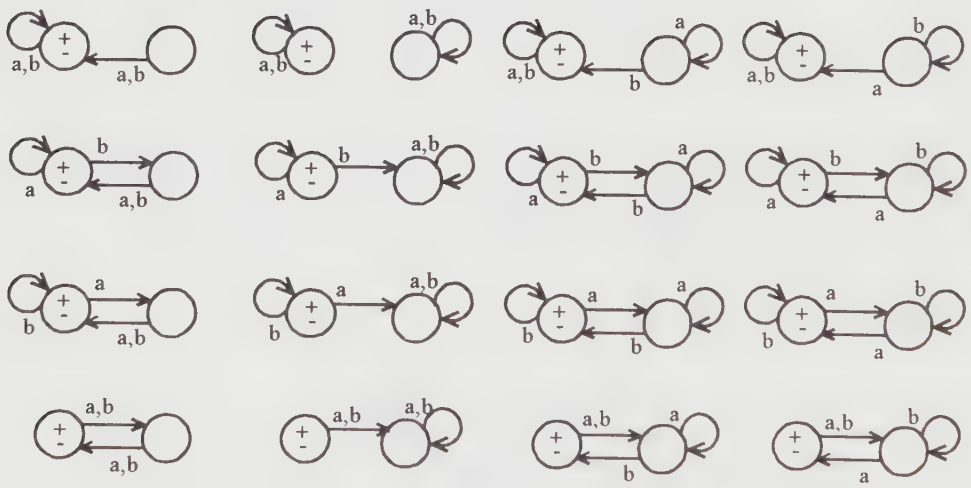
9. (i)



(ii) **aa* + bb***

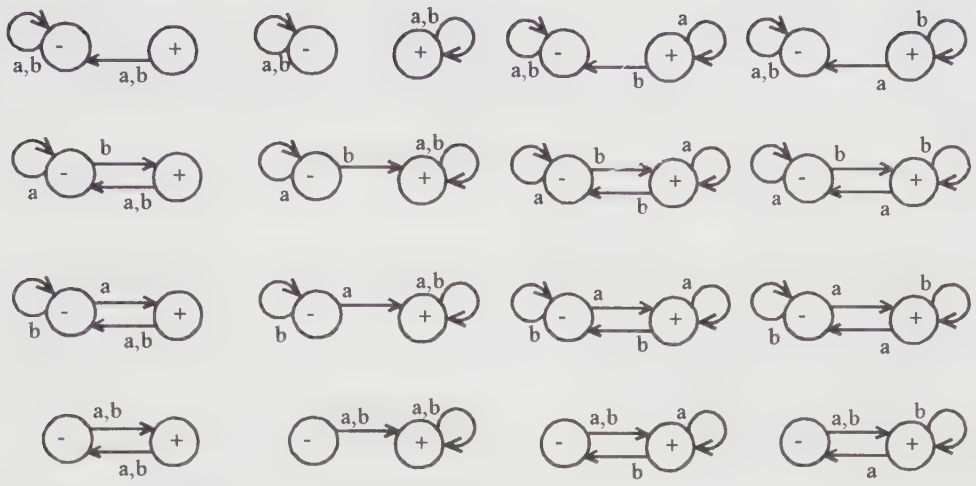


10. (i)



Chapter Five

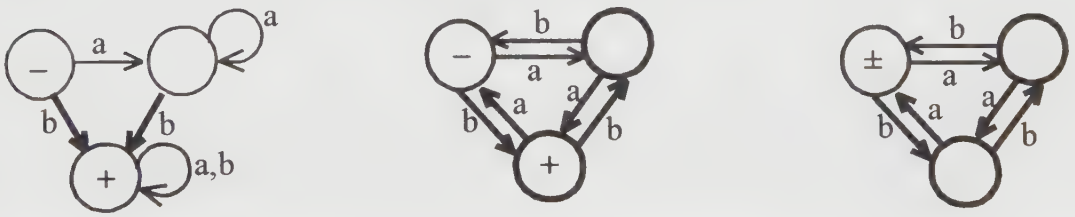
(ii)



(iii) Transition tables for two-state machine has four cells, (two rows and two columns). Each may be filled with 1 or 2. So there are 2^4 possibilities. There are still the same four ways to designate final states. Hence the total number of different transition tables is suprise 64, ($2^4$4).

11. An FA with a designated start state and two other states can have up to three final states if any at all. There is only one way it can have no final states, there are three ways it can have one final state, there are three ways that it can have two final states, and only one way that the FA can have three final states. For each of these 8 possibilities, there are 6 transitions each of which can have one of three destinations. In total, there are $3^6$8 = 729*8 = 5832 different finite automata of three states.

12. (i) (ii) (iv)

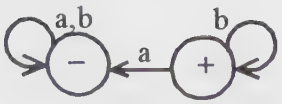


(iii) The language accepted by NIF is the reverse of every word in the language accepted by FIN. If there was a path from one state to another on FIN by a sequence of letters, then there must be a path on NIF using the sequence backwards.

(iv) The FIN pictured above accepts the non-palindrome words *ab* and FIN = NIF.

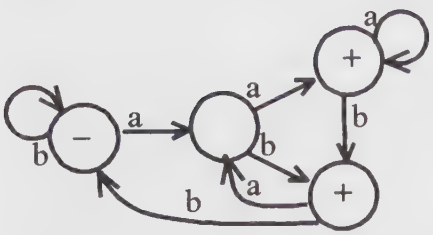
Chapter Five

13. (i)

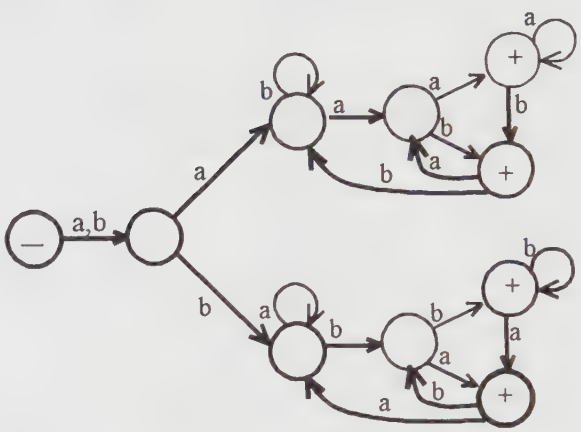


(ii) An FA only retains its properties if the state being removed could not actually be reached, that is a state is removable only if there are no edges coming into it. Clearly if the state is never used it has no bearing on the language that the FA accepts.

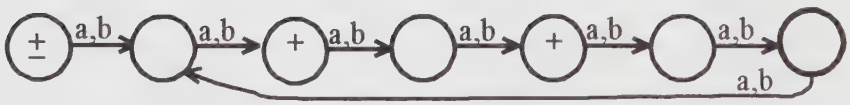
14. (i)



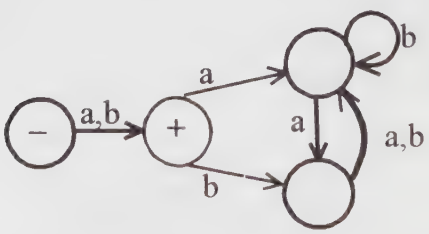
(ii)



15.



16.



17. (i) The language of odd length words over {a,b} that end with *a*.
(ii) The language of all words over {a,b} of even length that end with *a*.
(iii) The language of all words over {a,b} of even length that have *a*'s in all even numbered positions.
(iv) **(aa+ab+ba+bb)*a**
(a+b)(aa+ab+ba+bb)*a
(aa+ba)*(aa+ba)

18. We start in state 1 and remain there until we encounter an *a*. State 2 = we have just read an *a*. Scan any *a*'s and return to state 1 on reading *c*. State 3 = we have read a *b* following an

a. Reading an *a* puts us back to state 2 and reading a *b* sets us back to state 1. However state 4 = we have just found a substring *abc*, and if the whole sequence was read the string is accepted.
Now states 4, 5 and 6 exactly mirror states 1, 2 and 3. Returning to state 1 indicates that we just found another occurrence of the substring *abc*. Being in one of the first three states means that we have read an even number of *abc* substrings (if any) and are in the midst of finding another one. Ending in an accepting state, 4, 5 or 6, means that we have read an odd number of *abc*'s.

19. (i) This FA starts out like a binary tree. For the first three letters of the input, each time we read a letter we reach some node on the next level. However reading any more letters traps us in the dead end state at the bottom. So no word of more than three letters can be accepted.
(ii) There are exactly three final states in the machine. Concatenation of the labels on the transitions to these states shows that the machine accepts the words *a*, *aab* and *bab*.
(iii) Eliminate the three existing pluses. Place new pluses in each of the following states, the right-most node on level two, the third node from the left on level three and the second node from the right also on level three.
(iv) Each of the 15 possible strings with less than four letters is represented by one of the nodes (excluding the dead end state). (Note the lexicographic order.) Simply mark those nodes that represent the words in the language with plus signs.
(v) Since L is finite, we can determine the longest word and then build a binary tree large enough to accommodate it. Next we add a dead end state with transitions from all the leaves and itself. For every word in L, we place a plus in the appropriate, unique state.

20. Even though the binary tree is infinite, the machine retains its uniqueness property (of the machines described in problem 19) that the path for every string ends in a different state. Surely just by placing the plus signs in the nodes that represent words in L would constitute an Infinite Automata. However, the machine obviously fails to define the language because no one can write an down the infinitely large picture.

CHAPTER 6

Transition Graphs are nondeterministic, and non-determinism is a horse of a different color. Students who are forced to jump into nondeterminism at the level of complexity theory (e.g. Cook's Theorem relating hard problems to nondeterministic TM's) but who have not seen nondeterminism at the FA level are confronted with a sea of gobble-de-gook such as oracles and other demons.

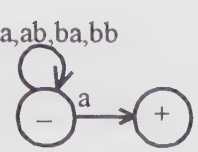
It is stated in the text that TG's were invented in 1957 to simplify the proof of Theorem 6 which (as we see later) was proved in 1956. Clearly the 1956 version of Kleene's Theorem did not explicitly mention TG's. We are committed to develop this subject in logical order, not chronological order. Paradoxically, the oldest mathematics in the book is the material at the end.

The transpose operator of problem #17 will be brought up again later and it may be useful to cover it now.

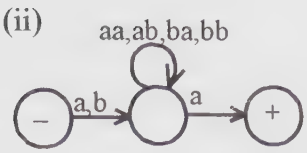
Chapter Six

1.

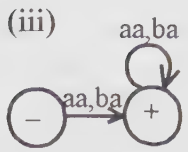
5-16 (i)



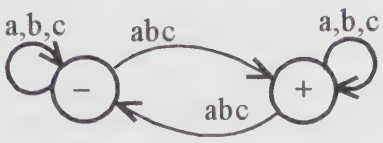
(ii)



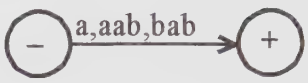
(iii)



5-18



5-19



2.

(i) 2,5
(ii) 4
(iii) 1,2,4
(iv) 1,3,5,6
(v) 1,2,4,6
(vi) 1,5,6
(vii) 1,5,6
(viii) 5
(ix) 5
(x) 3,4,6

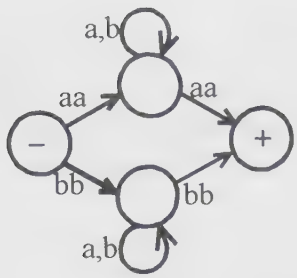
3. If the TG has an odd number of states, simply add another state. If your definition insists that the graph be connected, then add an edge from that state to any other state with an arbitrary label. With no transitions to enter the new state, it can not possibly interfere with the language accepted.

4. There are infinitely many TG's with two states over any non-empty alphabet.

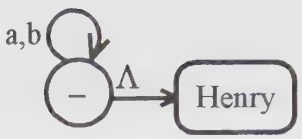
5. Proof by constructive algorithm: If the TG has no plus state, then we must add a plus state that cannot be reached from the start state. If the TG has only one plus state, no conversion is necessary. Otherwise, add a new plus state. Then for every original final state, delete the plus and add a Λ-transition from that state to the new plus state. Any string that was accepted by the old machine will be accepted by the new one by taking the Λ-move, and no string will be able to reach the new + state that could not reach one of the final states on the old machine.

Chapter Six

6.



7.



8. (i) There are many machines that accept this (and other any finite) language. One way to build this FA is to designate a start state. For every word in L, add a plus state and a transition from the start state with the word as the label.
   (ii) Two states are required; a start state and a final state. The one edge should be labeled with each word in the language, i.e. $w_1, w_2, w_3 \ldots w_k$ The reason why one state is not enough is that the word x which is a concatenation of two words in the language would then be accepted even though x is not in the language and there is no label x.

9. Make a new designated start state. Add two Λ-transition edges, one to the start state of TG1 and one to the start state of TG2. The final states remain the same as on the individual machines. The resulting machine TG3 now accepts all the words that used to be accepted by TG1 by taking the Λ-edge to the old start state on TG1 and continuing to a final state. Similarly TG3 accepts all the words that were accepted by TG2 and finally accepts no words that were not accepted by either TG1 or TG2.

10. This time no new states will be necessary. Simply add Λ-transitions from every plus state in TG1 to every start state of TG2. Next erase the minus from the TG2's start state. Finally delete all the pluses from any state that came from TG1, leaving the pluses in TG2's states unchanged. The resulting machine TG4 is a connected graph that accepts words that are the concatenation of a word from L1 followed by a word from L2.

11. L^+. L* must contain Λ, and unless Λ was a word in L originally, it will not be accepted by the modified machine.

12. (i) If there are no transitions that lead into the start state then we can simply make this a final state as well. Otherwise we can add a new state marked ± with a Λ-edge to the original start state and then remove the old minus sign.
   (ii) Add Λ-transitions from all final states back to the start state. If L1 includes Λ then stop otherwise now apply one of the methods described in part one.

13. For every sub-expression that consists entirely of letters of the alphabet we can make a separate TG that accepts it by the method described in problem 8 for finite languages. For any sub-expressions that are joined with a + in the regular expression, their respective (mini)

Chapter Six

TG's can be combined as described in problem 9. For each sub-expression that is starred in the regular expression, the appropriate TG can be transformed as outlined in problem 12(ii).

14. Let us number the states clockwise from - (1) to + (3) and around to 6.
From start, there are two choices for the first *a*: 1-2 or 1-6.

1-6 If we take 1-6 edge then we have no more choices. We take the only outgoing *b*-edge and arrive at 4, then *bb*-edge to 5, *a*-loop at 5, *bbb*-edge to 3, *ab*-edge back to 4. The remaining string is *ba* and there is no edge we can take so the input crashes.

1-2 In this case there are five choices for *bbb*: 2-2-2-2, 2-2-2-3, 2-2-3-3, 2-3-3-3, 2-6.
2-6 gives us two choices for *a*:

6-1 in which the word crashes, or

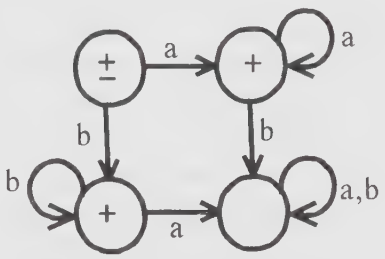
6-3 in which case we loop at 3 for *bbb*, take the *ab*-edge to 4 and crash.

2-2-2-2 means that we use *a* to get to 3, then loop at 3 for *bbb*, take *ab*-edge to 4 and crash with *ba* left unread.

(2-2-2-3, 2-2-3-3, 2-3-3-3) For the other three choices have read *abbb* when they reach state 3. Follow the *ab*-edge to 4, *bb*-edge to 5, *a*-loop at 5, *bb*-edge to 6, and finally *a*-edge to 3, where the word is accepted.

To summarize the paths are: 1-2-2-2-3-4-5-5-6-3, 1-2-2-3-3-4-5-5-6-3, 1-2-3-3-3-4-5-5-6-3. And there are no other possible paths.

15.



16. (i) If there are edges out of the final state by which a string can reach the same or another plus state, the new machine will accept not only the old language L and *ba* but also the product of *ba* and all strings that can be formed by concatenating the labels on the path from the final state to itself or another final state.

(ii) If there is a circuit that goes back to the start state, we face a problem similar to the one above: the machine will accept L and *ba* and also the product of any strings that can finish their processing in the start state concatenated with the string *ba*.

(iii) Add a new start state and a new final state to the machine and connect them with a transition labeled *ba*. Connect the minus state to another minus state by a Λ-edge.

17. (i) Change all final states of the FA into start states of the TG for the transpose. Change the start state of the FA into a final state in the TG. Reverse the direction of every transition in the FA. This machine accepts exactly the reverse of every word in the original language.

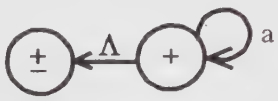
(ii) First apply all the steps outlined in part one. Then if the original machine was a TG not an FA, then we must also change the labels. For each edge, replace each string that

appears in the label with the reverse of that string.

(iii) $L_1L_2 = \{w_1w_2 \mid w_1 \in L_1, w_2 \in L_2\}$ When w_1w_2 is transposed, the last letter of w_2 becomes the first letter of the new word followed by the rest of w_2 spelled backwards, then the letters of w_1 spelled backwards. This is exactly the word transpose (w_2) · transpose (w_1). Therefore the transpose of the set L_1L_2 is transpose (L_2) · transpose (L_1).

18. We can think of a word in the language of a TG as the concatenate of edge labels on a path from - to +. Suppose no edge of T has a label of odd length. Then no concatenate of edge labels from T can possibly produce a string of odd length. So if L is the language of T and if L contains an odd length word the T must have a edge with a label of odd length.

19. For example:



20. (i) In a machine without Λ-edges, every transition consumes at least one letter of input. Therefore, we know that Step 2 is true. The list in Step 3 is exhaustive; it includes all possible paths through the machine (where a path is represented by the sequence of integers that number its consecutive edges) and also sequences of integers that cannot represent paths because the edges are discontinuous in the graph. There is an upper bound on the magnitude of the integers used as edge numbers and on the length of the sequence. Since the list is finite , we can actually check it for the two required characteristics: 1) Does the sequence represent a path from a start state to a final state, and 2) when we concatenate the edge labels, does the result match the word we are testing? Since all possible paths are listed, if our test word has a path we must find it. Since we test paths for validity at Step 4, we are not deceived by matching edge labels on an invalid path. The list and the procedures we apply to it are finite and so this algorithm must terminate.

(ii) In a machine with Λ-edges, it is not true that every transition consumes at least one letter of input. We can use infinitely many Λ-transitions without consuming any letters at all, and therefore we cannot set an upper bound on the length of the path that gives us the test word.

CHAPTER 7

It is possible to teach Computer Science without covering Computer Theory but it is not possible to teach Computer Theory without covering Kleene's Theorem. We believe that the excruciating detail that we go through in this proof is completely justified and several grounds:

1. the theorem is so important it merits a complete proof
2. the proof is a beautiful example of the strength of recursive definitions
3. it is an important illustration of the technique of proof by constructive algorithm

There is one pitfall that we must be careful to avoid here. We do not want to over emphasize the importance of developing facility with the method of conversion. It is only important to see that conversion is possible; it is not important to "get good" at doing it. In some courses it is necessary to drill on a new technique until the students become expert in applying it. That is not the case here. The only purpose the examples and problems serve in this chapter are to insure that the student really understands what is going on. It would be misleading to give the impression that the ability to convert regular expressions into FA's and vice versa has practical or commercial value.

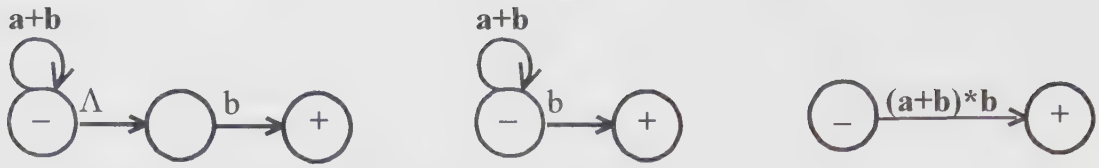
This chapter usually takes a week to cover satisfactorily.

We have heard of zealous students (or intense teachers) who have written (assigned) computer programs to perform the algorithms on FAs in this chapter. As long as it does not displace some of the more important material this seems harmless.

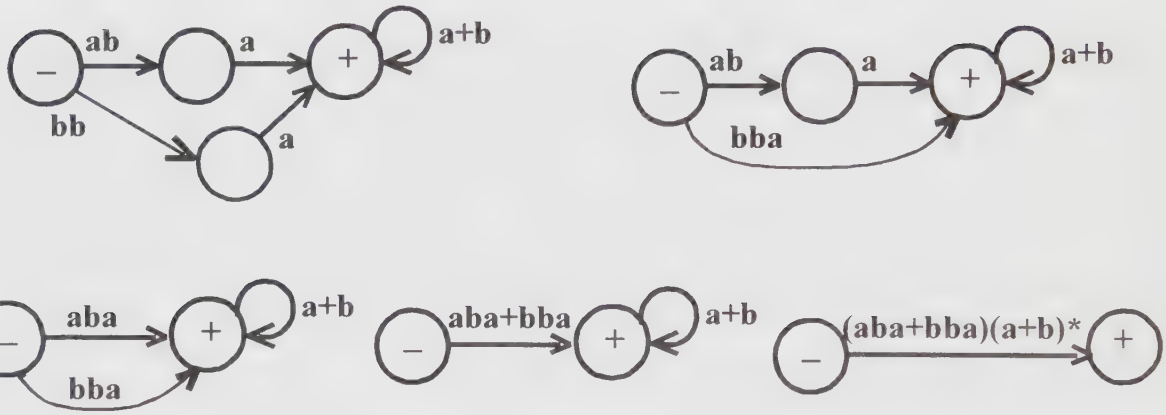
Many authors prove Kleene's Theorem by first proving that nondeterminism can be removed from FA's and then the result follows easily. We include this approach in this edition without replacing the old method for two reasons. The old method is a perfect illustration of the nature of how to design a constructive algorithm -- an art which can only be taught by example. And the fact that there are two proofs for the same result is in itself a great example of the power of mathematical techniques as opposed to mathematical results alone. Each proof is in essence its own separate result.

Chapter Seven

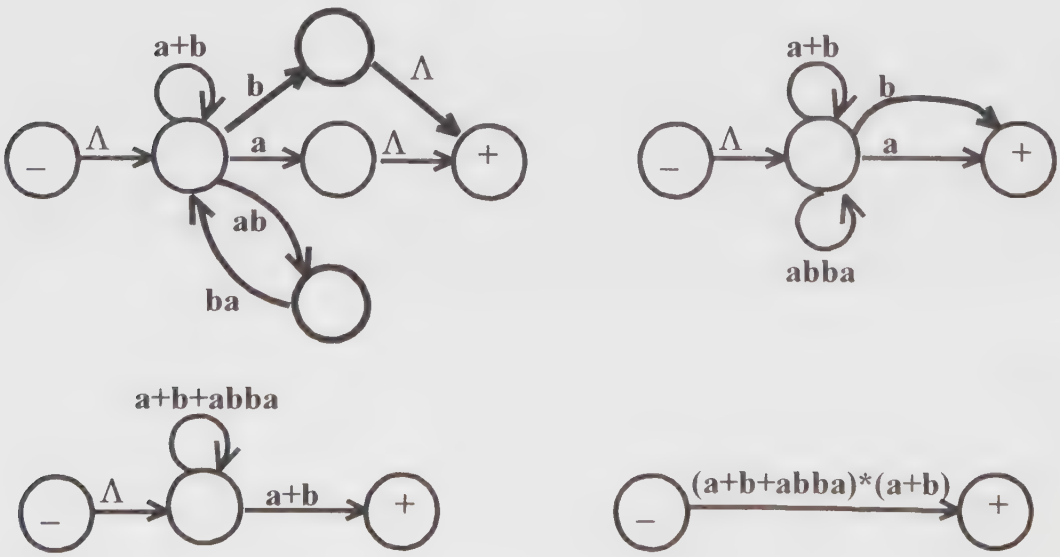
1. (i) (a+b)*b



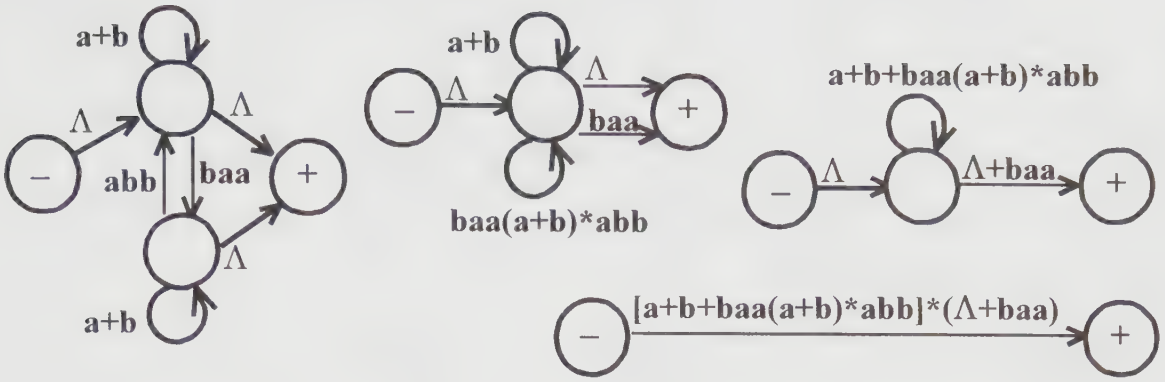
(ii) (a+b)ba(a+b)*



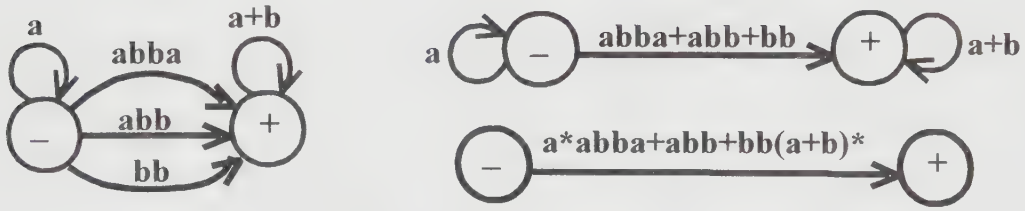
(iii) (a+b)*(a+b)



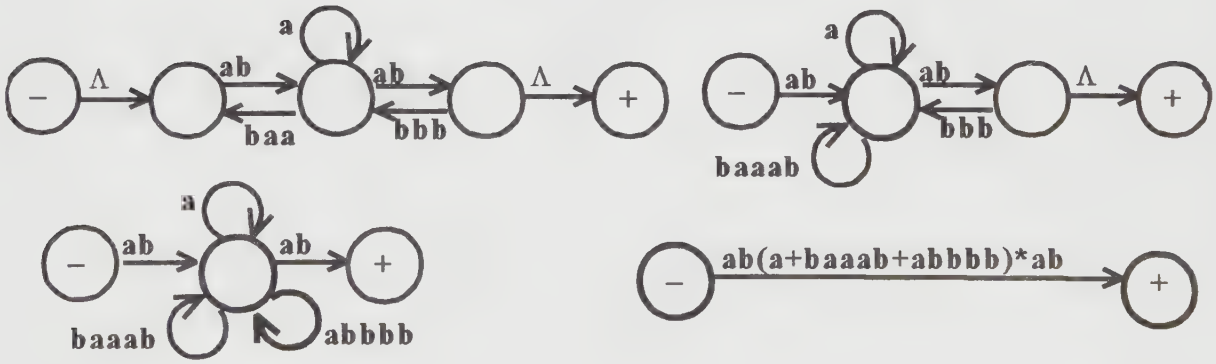
(iv) (a+b)*



(v) a*bb(a+b)*



(vi)



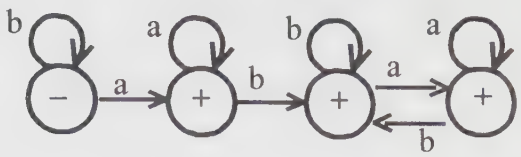
2.

(a+b)*	(a+b)*	(a+b)*	(a+b)*
(a+ba+bb)*	a*	(a+ba*b)*	(a+bb*a)*
(b+aa+ab)*	b*	(b+aa*b)*	(b+ab*a)*
[(a+b)(a+b)]*	Λ	[(a+b)a*b]*	[(a+b)b*a]*
∅	∅	∅	∅

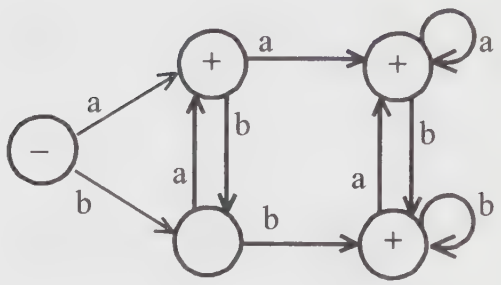
Chapter Seven

a*b[(a+b)a*b]*	**a*b(a+b)***	**a*b(a+ba*b)***	**a*b(b+aa*b)***
b*a[(a+b)b*a]*	**b*a(a+b)***	**b*a(a+bb*a)***	**b*a(b+ab*a)***
(a+b)[(a+b)(a+b)]*	**(a+b)(a+b)***	**(a+b)(a+ba+bb)***	**(a+b)(b+aa+ab)***

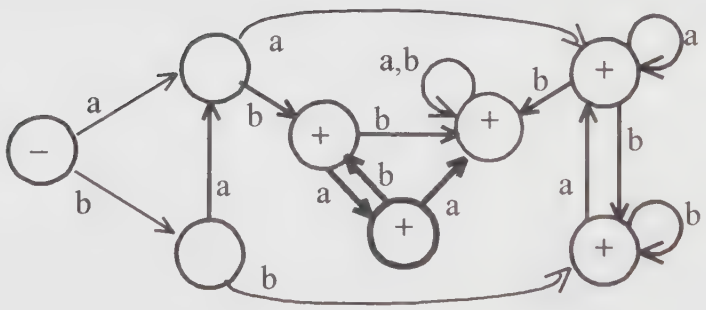
3. (i)



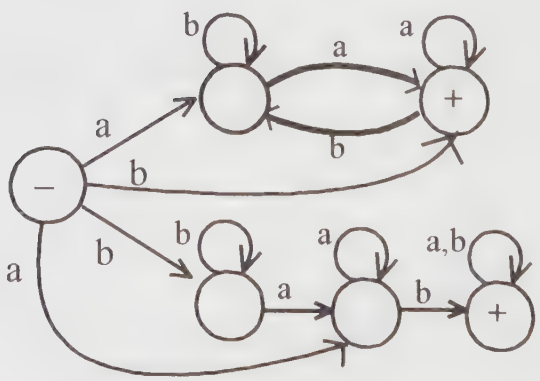
(ii)



(iii)

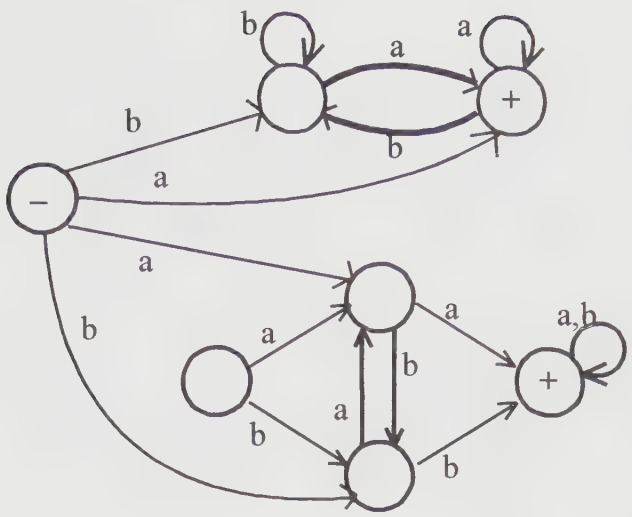


4. (i)

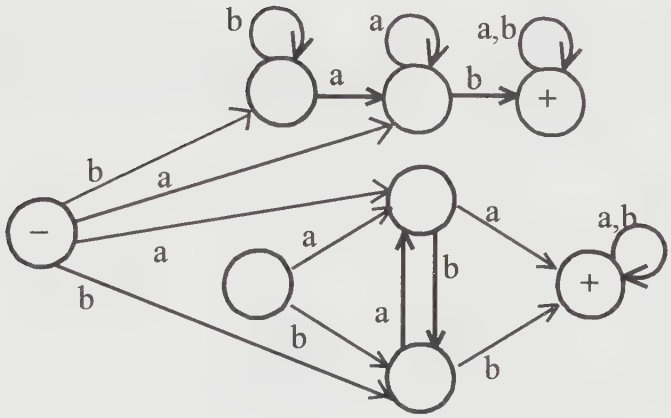


Chapter Seven

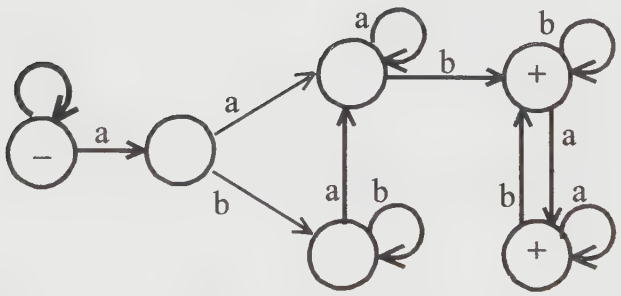
(ii)



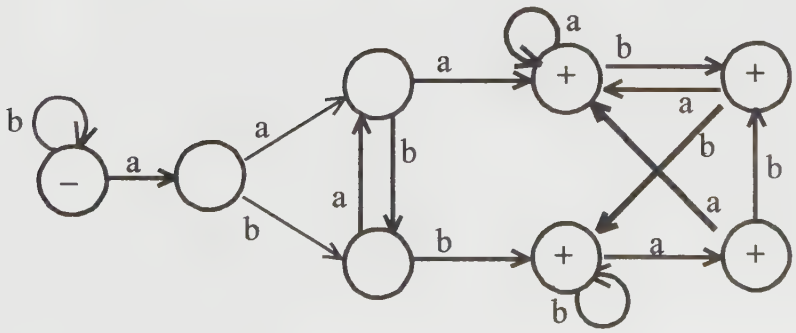
(iii)



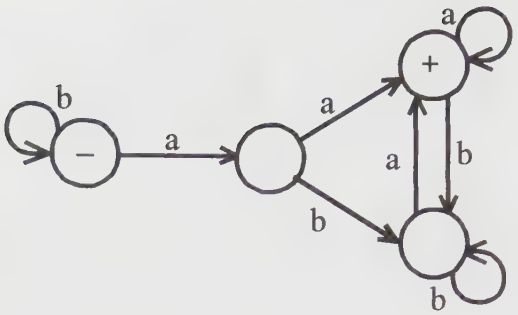
5. (i)



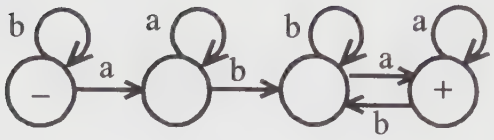
(ii)



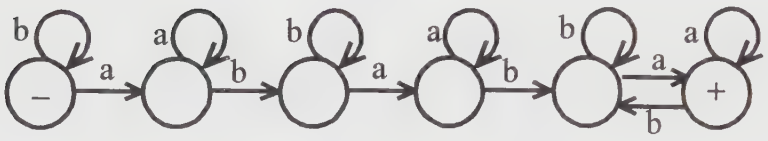
(iii)



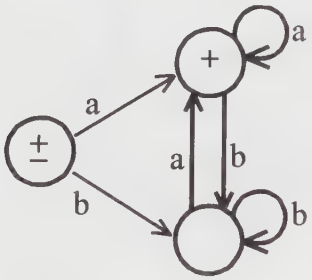
(iv)



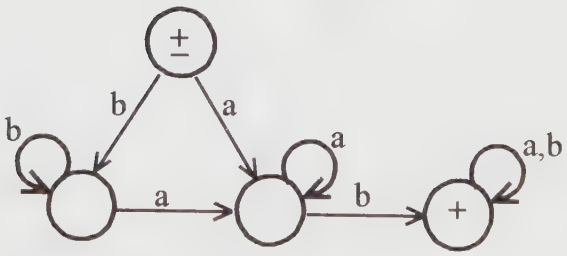
(v)



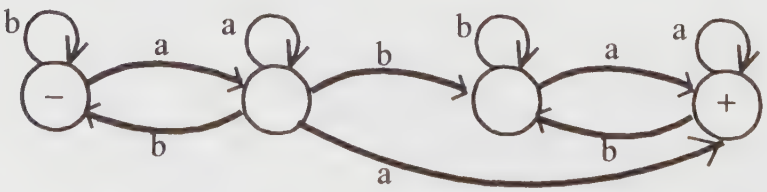
6. (i)



(ii)

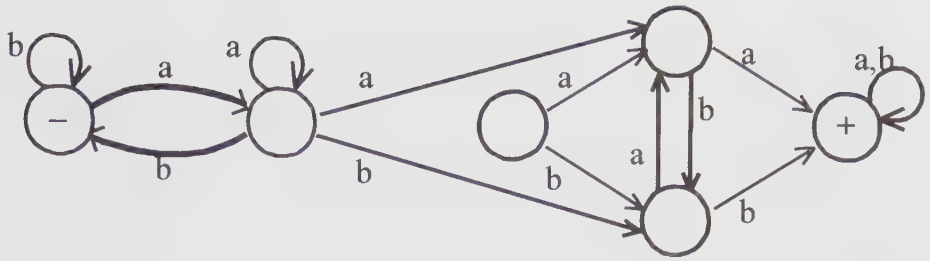


7. (i)

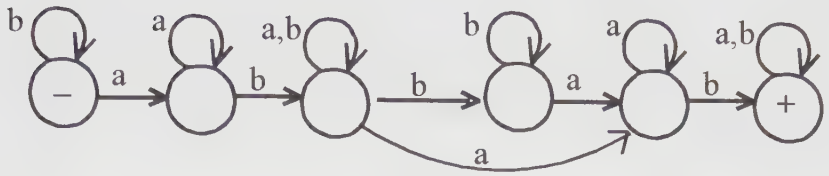


Chapter Seven

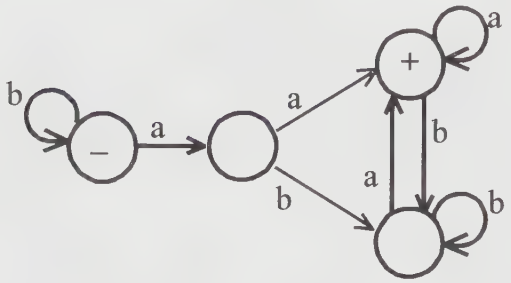
(ii)



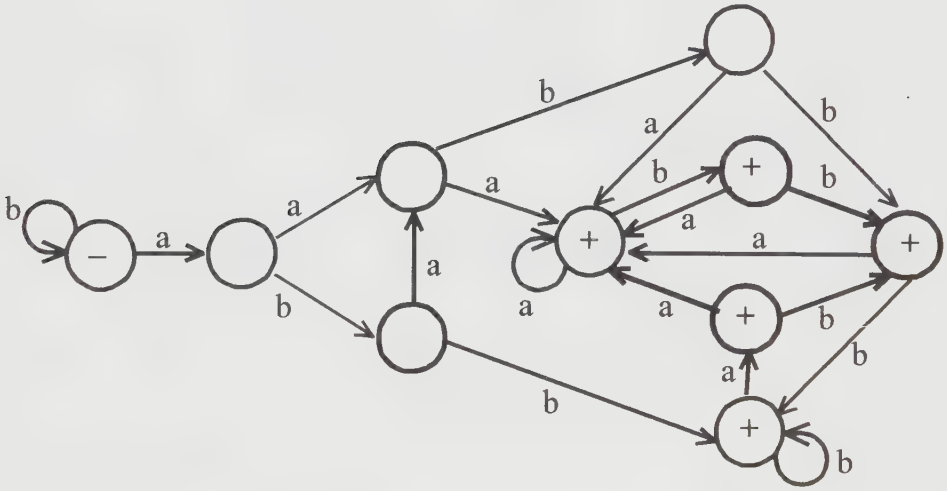
(iii)



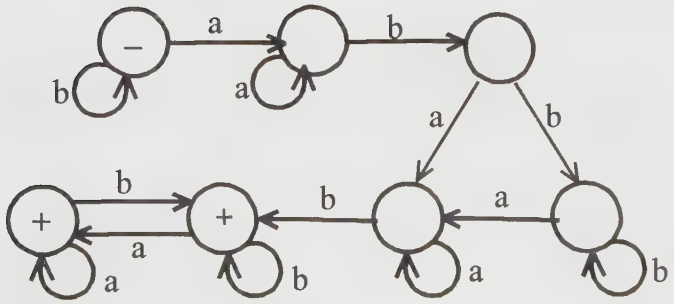
8. (i)



(ii)



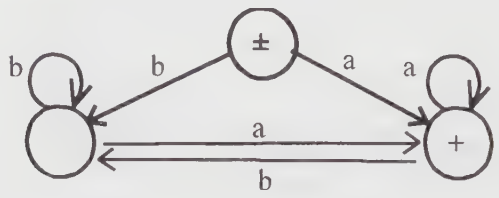
(iii)



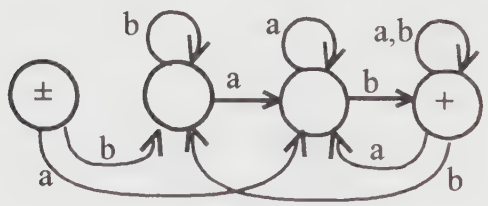
Chapter Seven

9. (i) The point is to ensure the option of getting from any final state to the start state. If there is only one final state which is also the start state no changes are necessary. In general, for every transition from the start state to some state x (possibly the start state) with label l, add an edge from each final state to x with able l. (If the original start had an l-edge to a final state then the resultant machine will have an l-loop at that final state.) If there are no incoming edges (or loops) at the start state then simply insert a + sign in order to accept Λ. However if there is even one incoming edge at start then make a clone start state. The new start state is marked $\pm$ and has edges going to the same places as the original start state. (If the original start had a loop on c, then there is a c-edge from the new start to the old start in addition to the c-loop which is still at the old start.) After the edges are drawn then erase the - sign from the old start state.

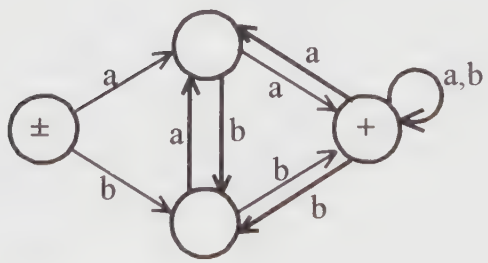
(ii)



(iii)

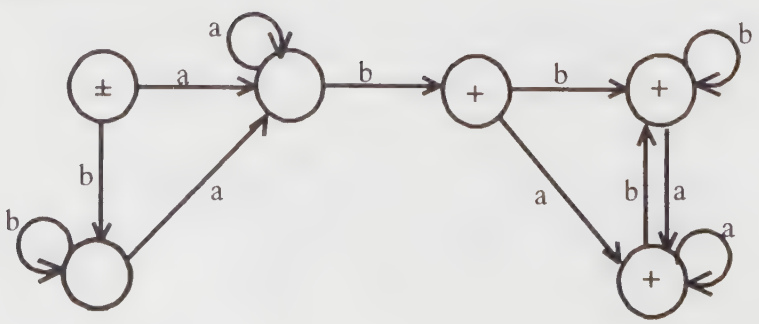


(iv)



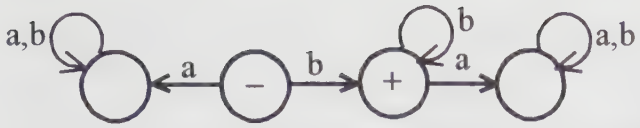
10. 9(ii) Already is an FA.

9(iii)

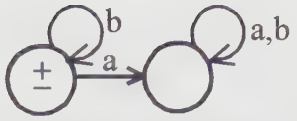


Chapter Seven

11. (i)



(ii)

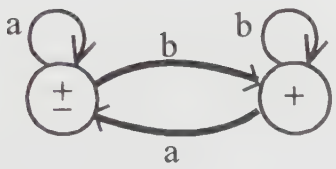


12. (i) Both the machines and the languages are different. The product language does not include Λ, while the closure does.
(ii) The product and the closure define the same language but the machines are different.

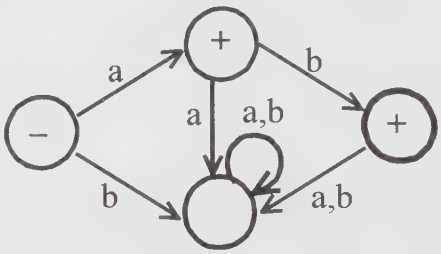
13. (i) The direct method usually produces less states.
(ii)

Union	a) $n2^m$	b) 2^{n+m}
Product	a) m^n	b) 2^{n+m+1}
Closure	a) 2^n	b) 2^n

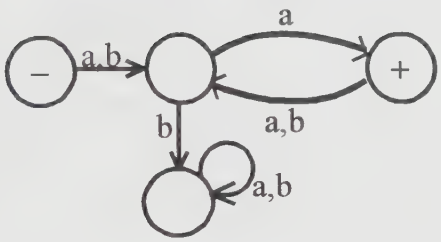
14. (i)



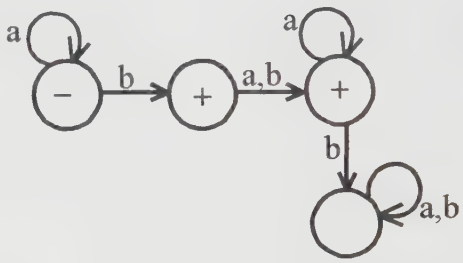
(ii)



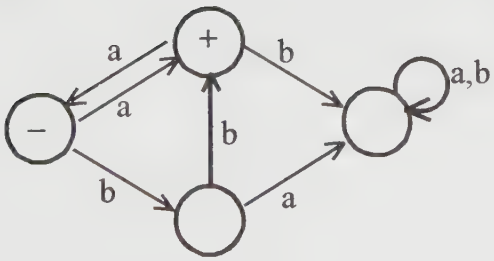
(iii)



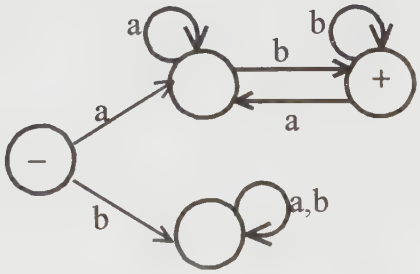
(iv)



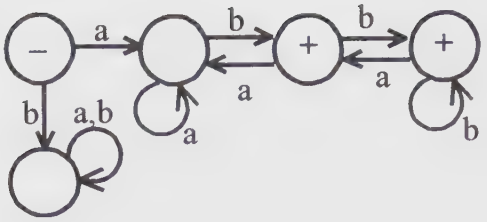
(v)



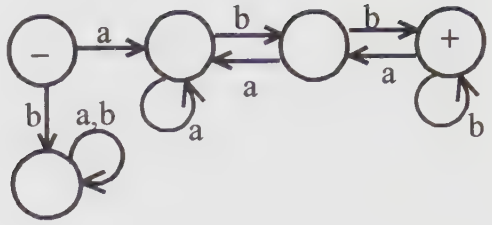
(vi)



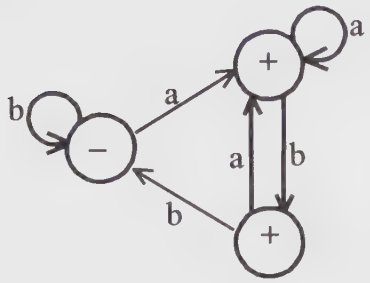
(vii)



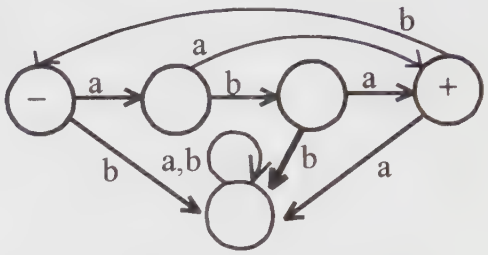
(viii)



(ix)



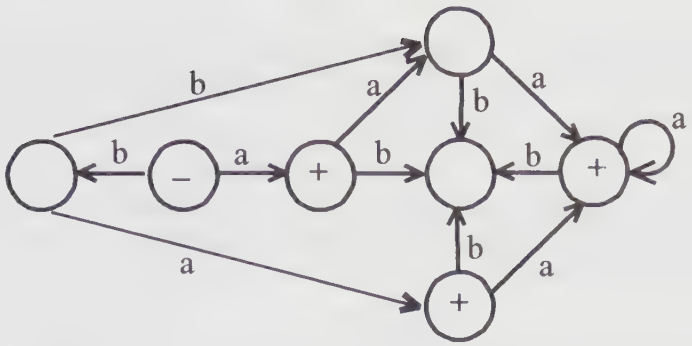
(x)



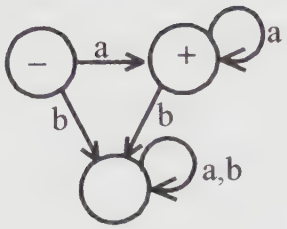
15. An NFA-Λ can be converted into an FA by treating states on the FA as representing a subset of states on the NFA-Λ. Begin with making a new start state whose set includes the old start state and any other state that can be reached from any state in the set by a Λ-edge. The *a*-edge goes to a state whose set includes all destination states that could be reached by an *a*-edge from any state on the NFA-Λ that is in the set of the source state on the FA. Likewise, draw the *b*-edge to a state which represents the subset of states reachable from any state in the source-state-set. For each new state generated in the FA, before drawing the appropriate outgoing edges, add to its associated set any state that is reachable by a Λ-edge from any state already in the set. When all the states in a set have no edge for a given able on the NFA-Λ, then draw that edge to the dead-end state on the FA. However, if at least one member of the set associated with a source state has an edge with that able then (even though some other member state would have gone to the dead-end) the edge is draw to a destination state whose set includes the optimistic choices. Continue drawing edges and generating new states until every state on the FA has both an *a* and *b* edge. Any state on the FA that represents a state which was final on the NFA-Λ must be marked with a +.

Chapter Seven

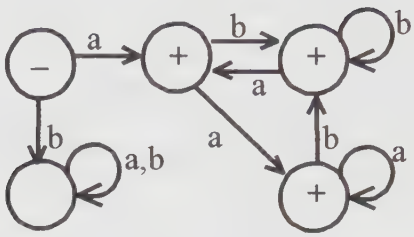
16. (i)



(ii)

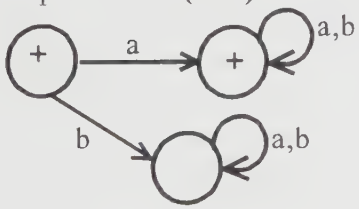


(iii)

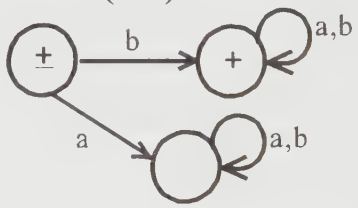


17. (i) Union can be accomplished by adding a new start state with two Λ edges, each one leading to an original start state. Then remove the extra - signs.
(ii) Product is made by adding Λ edges from each final state in FA_1 to the start state of FA_2.
(iii) Similarly the closure machine is produced by adding Λ edges from each final state to the start state and if the start state does not already have a + sign as well then add it.

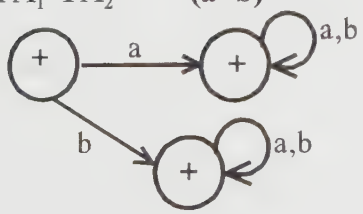
18. (i) FA_1 **Λ + a(a+b)***



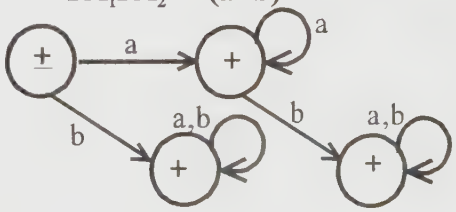
FA_2 **Λ + b(a+b)***



FA_1+FA_2 **(a+b)***

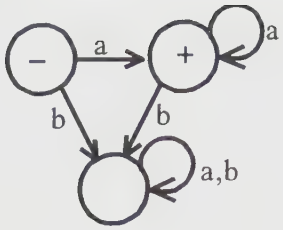


FA_1FA_2 **(a+b)***

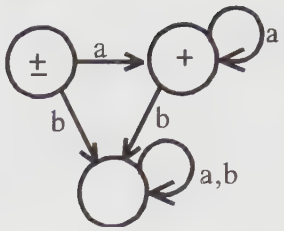


Chapter Seven

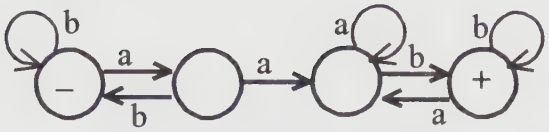
(ii) **aa***



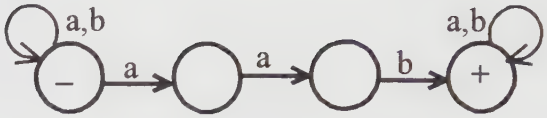
a*



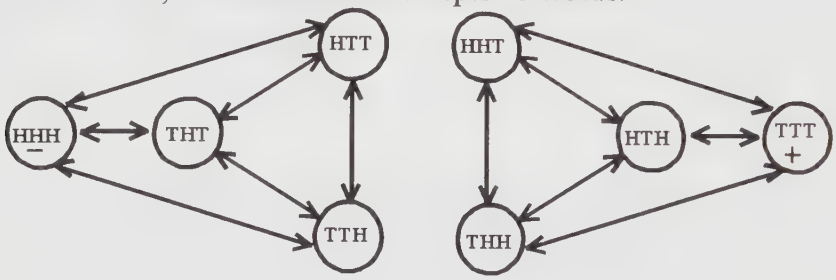
19. FA



NFA



20. No paths from - to +, and this machine accepts no words.



CHAPTER 8

This is the first chapter with is truly independent of the rest of the book. Despite the fact that it is mentioned later when output become a hallmark of Turing machines, the material here is not required there.

However there are good reasons for not skipping this chapter unless absolutely necessary. For one thing Mealy and Moore machines model sequential circuits and these are an important topic in courses in computer logic, switching theory and architecture.

The topic of nondeterministic finite automata with output would very interesting to discuss at this point. Students should be able to appreciate the fact that a single input may take several different paths but not necessarily give different outputs. This question is interesting but too much of a digression for the main text.

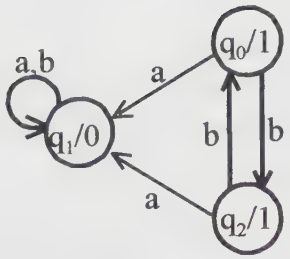
The AND and OR gates in the feedback circuits are not drawn using the usual symbols. The purpose for this was to not disadvantage students who had not seen those symbols before.

Some of the problems emphasize the nature of the transducer as a function from strings to strings. The ultimate importance of this is beyond the scope of undergraduate education.

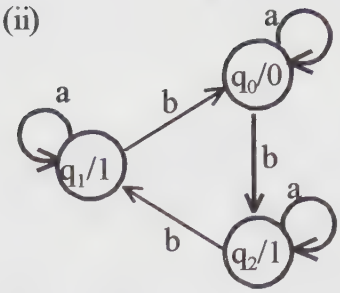
Chapter Eight

1.

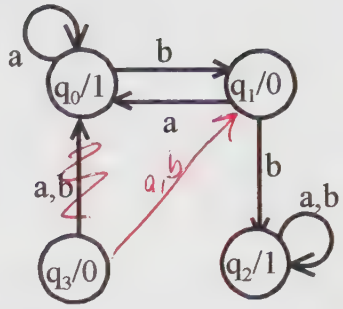
(i)



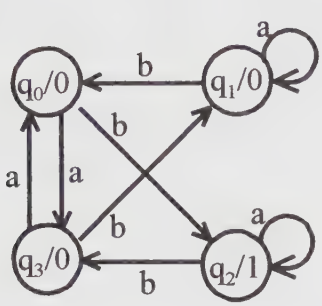
(ii)



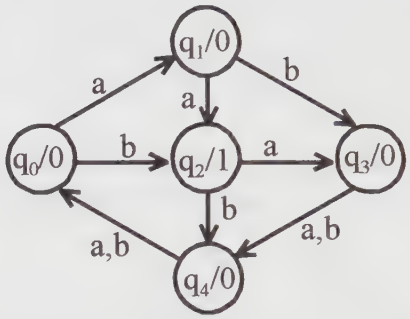
(iii)



(iv)



(v)



2. (i) For each state there are two choices of output 0/1, hence there are 2^4 ways of labeling the states. Then any of the eight edges can lead to any of the four states, that's 4^8 ways. So there are $2^4 4^8$ four state Moore machines.

   (ii) There are $2^n n^{2n}$ Moore machines for n states where the input and the output alphabet each have exactly two characters.

3.

(i)

	a	b	Output
q_0	q_0	q_1	0
q_1	q_1	q_1	1

(ii)

	a	b	Output
q_0	q_1	q_1	0
q_1	q_0	q_0	1

(iii)

	a	b	Output
q_0	q_1	q_2	0
q_1	q_1	q_1	1
q_2	q_1	q_0	0

(iv)

	a	b	Output
q_0	q_1	q_1	1
q_1	q_2	q_2	0
q_2	q_2	q_2	1

(v)

	a	b	Output
q_0	q_2	q_3	1
q_1	q_3	q_2	0
q_2	q_1	q_2	1
q_3	q_2	q_1	1

Chapter Eight

4. 1. (i) 100000 (ii) 000111 (iii) 111010 (iv) 000110 (v) 001001
   3. (i) 000111 (ii) 010101 (iii) 011111 (iv) 101111 (v) 110101

5. We can convert a Less machine into a Mealy machine by the same algorithm that we used to convert a Moore machine into a Mealy machine. (The start state of the Less machine will have no print instruction unless there is at least one edge into it, but this is not a problem.) To convert a Mealy machine into the equivalent Less machine, we follow the algorithm for converting Mealy to Moore. We do not have to worry about a print instruction for a start state with no in-edges since the start state of a Less machine does not print unless it is re-entered. It is possible that there are other states in the machine that have no in-edges. Output instructions for such states can be chosen arbitrarily without affecting the results obtained by running the machines, because such states can never be entered. (This point applies equally to Moore machines.)

6. (i)

	q_0	q_1
q_0	a/0	b/1
q_1	a/1	b/0

(ii)

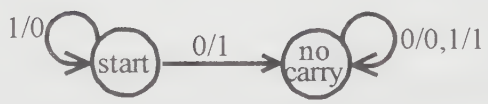
	q_0	q_1	q_2
q_0		b/1	a/0
q_1	a/0		b/0
q_2	a/1		b/1

(iii)

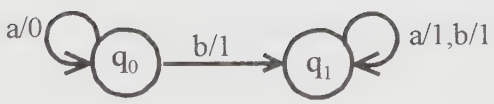
	q_0	q_1
q_0		a/0 b/0
q_1		a/0 b/1

(iv)

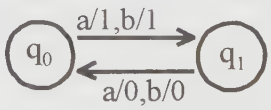
	q_0	q_1	q_2
q_0	a/0	b/1	
q_1		a/1	b/0
q_2	b/1		a/0

7. 

8. (i)

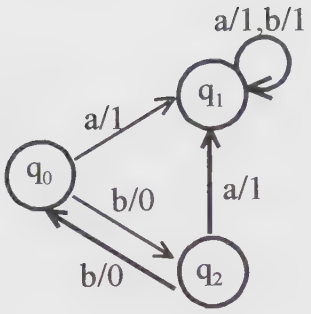


(ii)

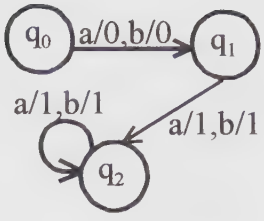


Chapter Eight

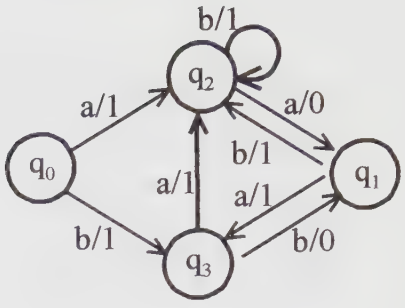
(iii)



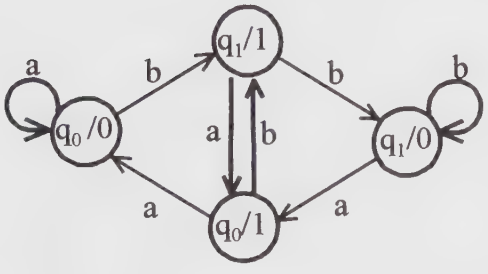
(iv)



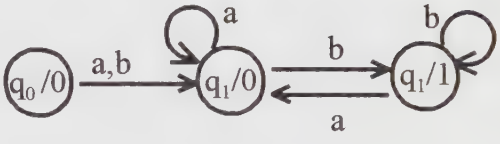
(v)



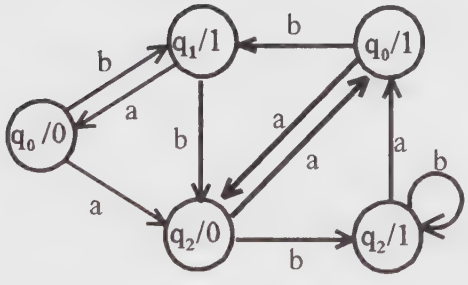
9. (i)



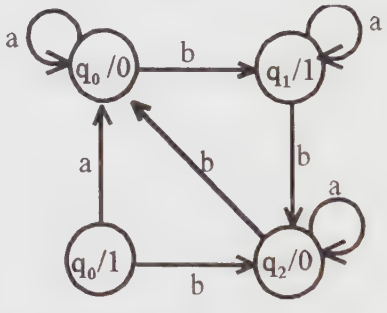
(ii)



(iii)



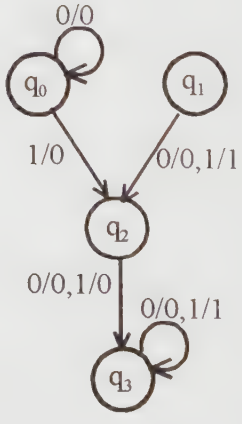
(iv)



10. New A = Input OR old A OR old B
New B = old A
Output = Input AND old B

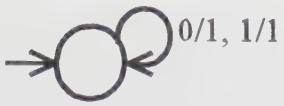
A=0 and B=0 is q_0
A=0 and B=1 is q_1
A=1 and B=0 is q_2
A=1 and B=1 is q_3

A	B	IN	A	B	OUT
0	0	0	0	0	0
0	0	1	1	0	0
0	1	0	1	0	0
0	1	1	1	0	1
1	0	0	1	1	0
1	0	1	1	1	0
1	1	0	1	1	0
1	1	1	1	1	1

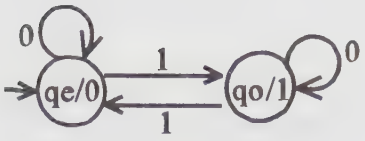


Chapter Eight

11.

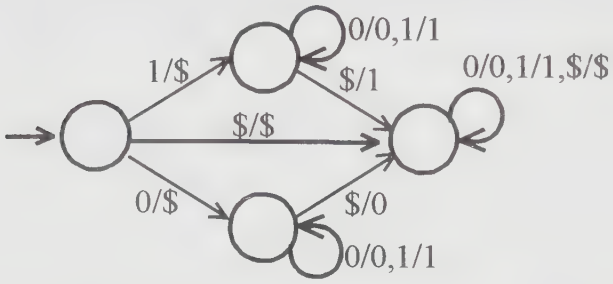


12. (i)



(ii) The Moore machine is a natural language acceptor.

13. (i)

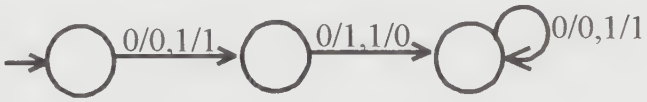


(ii) There is no way of knowing when the last letter is read and its time to print the letter that was read first.

14. Me_1 does not change the string at all so reprocessing a string through it any number of times does not change it. Me_2 swaps 1's for 0's and 0' for 1's but reprocessing will return them to their original values.

15. This machine flips the first bit and preserves the rest. So to undo the effect on the first bit, simply run the resulting string through the machine again.

16.



17. If Me_1 took two different input strings, x and y, and outputted the same string z then when z is fed into Me_2, it could at best produce one of the original strings x or y and Me_1Me_2 would not be an identity for all strings. So Me_1 must produce a unique string for each input so that Me_2 can distinguish it.

18. It is easy to have a machine that does not change the outputs from Me_1. Let us say that the output of Me_1 is zero for all inputs. Then Me_2 could be any machine at all and Me_2Me_1 would acts like an identity because $(Me_1)(Me_2Me_1) = Me_1$. But this is not a true identity because an identity must act on all inputs not just the one string that constitutes the only possible output

for a particular machine. So we note the fact that we confirmed in problem 17 that is that all Me_1's that can possibly participate in an M_1e M_2e identity must have the property that its function is bijective. Hence for each input x Me_1 produces a unique y which when fed into Me_2 reproduces x, likewise for each x fed into M_2e, the resultant y reproduces x on M_1e. This proves that if Me_1Me_2 is an identity then Me_2Me_1 is also an identity.

19. M_1 flips just the first bit leaving the rest of the string unchanged if the first bit was 0, otherwise it takes the complement of the entire string. M_2 recovers the changes that M_1 makes on a string. It flips the first bit alone if the first bit was 1, otherwise it flips all the bits in the string. $M_1M_2 = M_2M_1$ = Identity. The two machines are inverses of each other.

20. After you read the first bit Mealy machines must produce the first output bit but there is no way of knowing what the last bit will be at the onset of processing. So transpose cannot be implemented by Mealy machines.

CHAPTER 9

This is a workhorse chapter. It is straightforward and exhausting. Students are often tempted to try to take the shortcut of saying, "But I understand what the languages of these two regular expressions are, and I can understand what the intersection is, and I can write a regular expression for it." It should be explained to them why this is irrelevant to the topic of the chapter.

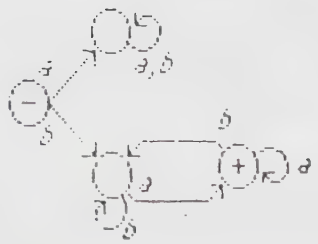
Any student who has never seen a Venn diagram should still be able to work out De Morgan's law for themselves.

This is another chapter in which it is possible to over emphasize the facility with the techniques of conversion, while missing the basic point that it is only the *possibility* of conversion that we are truly interested in.

This chapter is another good place to discuss the problems of finite and infinite. A student should be able to notice that despite the fact that the union of two regular languages is regular the union of infinitely many regular languages is not necessarily regular (assuming for the moment that there are non-regular languages). This is because any language can be considered the union of its separate words each as a regular language. On the other hand students may want to test their skill at the conjecture that the intersection of infinitely many regular languages is still regular, even if this intersection is still an infinite language.

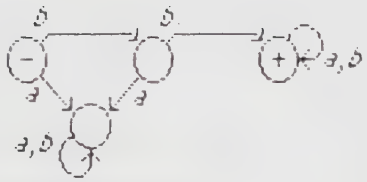
Chapter Nine

1. $b(a+b)^*a$ (begins with *b*, ends with *a*)

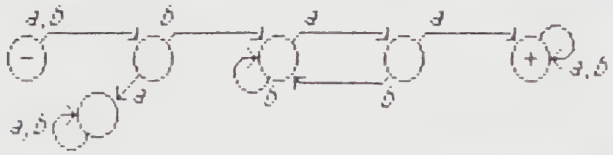


2. $(a+b)^*aa(a+b)^*a$ (contains *aa*, ends with *a*)

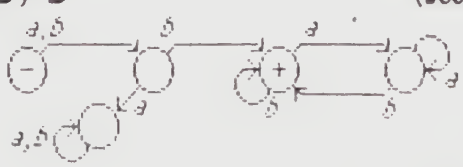
3. ϕ (ends with *a*, ends with *b*)
4. $bb(a+b)^*$ (first letter is *b*, second letter is *b*)



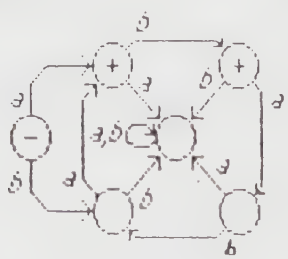
5. $(a+b)b(a+b)^*aa(a+b)^*$ (second letter is *b*, contains *aa*)



6. $(a+b)b(a+b)^*b$ (second letter is *b*, ends with *b*)

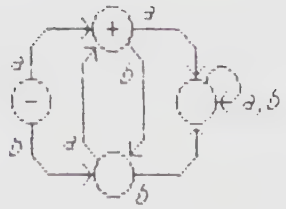


7. ϕ (*a* never doubled, contains *aa*)
8. $b^*a(bb^*abb^*a)^*b^*$ (*a* never doubled, odd *a*'s)



Chapter Nine

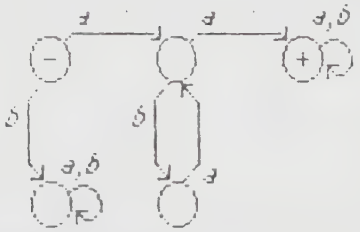
9. $(b+A)(ab)^*a$ (*a* not doubled, *b* not doubled, ends in *a*)



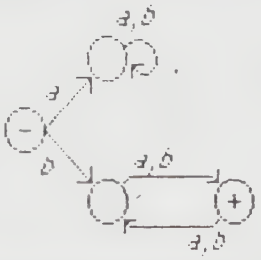
10. ϕ (starts with *a*, starts with *b*)

11. $a(a+b)^*$ (starts with *a*)

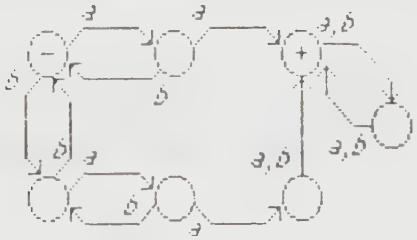
12. $a(a+b)^*aa(a+b)^*$ (starts with *a*, contains *aa*)



13. $b(a+b)(aa+ab+ba+bb)^*$ (even length starting with *b*)

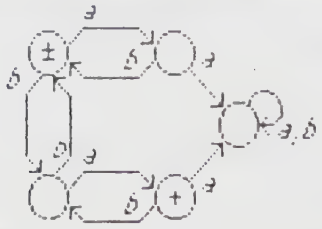


14. $(aa+ab+ba+bb)^*aa(aa+ab+ba+bb)^*$ (even length, contains *aa*)

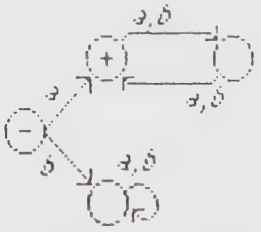


Chapter Nine

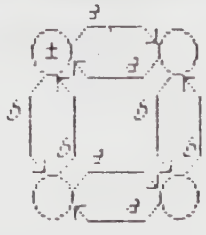
15. $(ab+bb)^*(\Lambda + b|ab+bb+a(ba)^*b|^*|b(ab+bb)^*+a(ba)^*|)$ (even length, no double a)



16. $a(aa+ab+ba+bb)^*$ (odd length, begins with a)

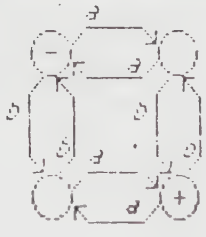


17. $(aa+bb+(ab+ba)(aa+bb)^*(ab+ba))^*$ (EVEN-EVEN)



18. (i) same as 17 (EVEN-EVEN)

(ii) $(aa+bb+(ab+ba)(aa+bb)^*(ab+ba))^*(ab+ba)(aa+bb+(ab+ba)(aa+bb)^*(ab+ba))^*$



19. (i) same as 18(ii) (ODD-ODD)

(ii) ϕ (even length, odd length)

20. There are many such: the set of finite languages, the set of languages of words of even length, the set of languages that begin with an a, etc.

CHAPTER 10

Both the pumping lemma and the pumping lemma with length are worth covering in detail. If the example **PRIME** is skipped as being "too mathematical" then one of the examples from the problem section should be substituted.

Students should also be encouraged to worry about whether all languages that are nonregular can be proven so using the pumping lemma. This raises both the question of whether the pumping lemma is a necessary and sufficient condition and the possibility of true statements (e.g. that some L is non-regular) having no known method of proof. Perhaps some more non-regular languages should be shown to be "pumpable" to drive home the point.

An appreciation of what kind of conditions define a regular language should be developing. Problem #20 should be very helpful in this respect. It is important to notice that there is an asymmetry between the beginning of the string and the end of a string as far as language-defining conditions goes despite the fact that the transpose of any reglar language is also regular. Paradox? not so.

Myhill-Nerode is new in this edition. I didn't include it in the first edition because its customary application is to find the "minimal" FA that accepts a given language. This is not a question that interests us enough to suffer through the distinction between the two types of equivalence classes that the theorem requires. However, as an alternate method of proving that certain languages are non-regular, it is simple, direct and useful. I like having multiple approaches to mathematical chores. It adds depth of understanding. Especially since the Pumping Lemma is difficult to conceptualize as a mere *tool* without this parallel approach.

The quotient languages section is skippable. The only future reference made to this result is in Chapter 22 in the proof that a Read-Only TM is just an FA. Both of these results can be sacrificed if the whole book is attempted in one term.

Chapter Ten

1. (i) Let *xyz* be a word in $\{a^n b^{n+1}\}$. Consider *y*, it could be one of three possible forms: a^+, a^+b^+, or b^+. If $y = a^+$ then *xyyz* would have too many a's to be a word in the language. If $y = b^+$ then *xyyz* would have too many b's. If $y = a^+b^+$ then *xyyz* would have some a's among the b's instead of separated.

   (ii) Let *xyz* be a word in $\{a^n b^n a^n\}$. Consider *y*, it could be one of five possible forms: a^+, a^+b^+, $a^+b^+a^+$, b^+a^+ or b^+. If $y = a^+$ then *xyyz* would have too many a's on one side of the b's. If $y = b^+$ then *xyyz* would have too many b's. If $y = a^+b^+$ or $y = a^+b^+a^+$ or $y = b^+a^+$ then *xyyz* would have some b's among the a's instead of all clumped in the center.

   (iii) Let *xyz* be a word in $\{a^n b^{2n}\}$. Consider *y*, it could be one of three possible forms: a^+, a^+b^+, or b^+. Just like part (i) doubling *y* would spoil the proportion or the order of the word.

   (iv) Let *xyz* be a word in $\{a^n b a^n\}$. Consider *y*, it could be one of five possible forms similar to part (ii): a^+, a^+b, a^+ba^+, ba^+ or b. If $y = a^+$ then *xyyz* would have too many a's on one side of the b. If $y = b$ or $y = a^+b$ or $y = a^+ba^+$ or $y = ba^+$ then *xyyz* would have too many b's.

   (v) Using the stronger Pumping Lemma, $|xy|<N$ (number of states), consider $xyz = a^N b^N a$. Hence *y* must be contained in the first set of a's and when pumped the a's would not balance the b's. So $xy^n z$ is not in the language.

2. Consider the infinite set of strings **aa*b**. For each string *x* in this set, a different string *z* is needed to make *xz* a word in any of the five languages. So in each case there are infinitely many classes.

3. For this we need the Pumping Lemma with length condition. Let the FA have N states. Consider $xyz = a^N b a^N b$ a word in DOUBLEWORD. $|xy|<N$ so $xy = a^N$ and when pumped, it is no longer in DOUBLEWORD.

4. (i) Using the length condition in the Pumping Lemma, consider $xyz = a^N b a^{N+1}$. *y* is a string of a's preceding the b. When pumped the number of trailing a's will be too few to indicate the length.

   (ii) Consider the infinite set b*. For each string a different z is necessary to constitute a word in TRAILING-COUNT. Therefore there are infinitely many classes.

5. (i) Consider $xyz = a^m b b a^m$, where m is an even number greater than N, a word in EVENPALINDROME. *y* is a string of a's preceding the bb. When pumped the number of succeeding a's will be too few. Likewise for ODDPALINDROME simply consider $xyz = a^m b a^m$.

   (ii) Again consider a*bb or a*b, and note that any such string will be in a different class.

6. (i) The squares have an interesting property. Consecutive squares differ by consecutive odd numbers. $1^2 = 1$ $2^2 = 1+3$ $3^2 = 1+3+5$ $4^2 = 1+3+5+7$ ...

Chapter Ten

This is because $(n+1)^2 - (n)^2 = 2n+1$ (an odd number.)
So the gaps between the squares grows larger and larger. For any number M eventually no two squares will differ by M. Certainly if $x > M$ and $y > M$ then $x^2-y^2 > M$ (unless x=y) since the closest they would be is $(M+1)^2-M^2 = 2M+1 > M$.
So when we pump, let $s = n^2$, $a^s = xyz = a^p a^q a^r$, the Pumping Lemma says that *xyz, xyyz, xyyyz*, ... are all in $\mathbf{a}^s$, which are a^{p+r+q}, a^{p+r+2q}, a^{p+r+3q}, ... However, in this sequence consecutive terms differ by the constant *q*, while squares get further and further apart. Therefore these terms can not all be squares.

(ii) For each *p*, xa^p is a square for infinitely many *x*'s. ($x=a^{s-p}$, $s = n^2$) However xa^p and xa^q are both squares only when $n^2-p = m^2-q$ for some *n* and *m*. But then $n^2-m^2 = p-q$ which cannot happen for large *n* and *m*. This is because, as we have seen above) the gap between the squares keep increasing and get above *p-q* and stop there. Therefore xa^p and xa^q are in SQUARE only for finitely many *x*'s. The class of *x*'s which become square by concatenation with a^p is different from the class of *x*'s which become square by concatenating with a^q. So a^p and a^q would require different states in an FA for any different *p* and *q*. Hence the machine would need infinitely many states and SQUARE is not regular.

7. (i) Consider $w = xyz = a^n b^n \in$ DOUBLESQUARE. If *y* contains the substring *ab* then *xyyz* contains two *ab* substrings and is not in DOUBLESQUARE. Therefore *y* is all *a*'s or all *b*'s. In either case *xyyz* increases only one letter not both and so breaks the form $a^n b^n$. Therefore *w* cannot be pumped and the language is not regular.

(ii) For each *n*, $a^n bx$ is only a word in the language when $x = b^{n-1}$. So there are infinitely many different classes.

8,9. (i) The argument in 7(i) never used the fact that *n* was restricted to squares, so the proof works equally well for DOUBLEPRIME and DOUBLEFACTORIAL.

(ii) The same as 7(ii). Notice that all three problems are special cases of the following general theorem: Let $n_1, n_2, n_3, \ldots$ be an infinite sequence of integers. The $L = a^{n1}b^{n1}$, $a^{n2}b^{n2}$, $a^{n3}b^{n3}$,... is nonregular by either the Pumping Lemma or Myhil-Nerode.

10. (iii) Let *xyz* be a word in $\{a^n b^n c^n\}$. Consider *y*, it could be one of five possible forms: a^+, a^+b^+, b^+, $a^+b^+c^+$, b^+c^+ or c^+. If *y* is a string of all the same letter than *xyyz* would have too many occurrences of that letter. If $y = a^+b^+$ or $y = a^+b^+c^+$ or $y = b^+c^+$ then the order of the letters in *xyyz* would be mixed up.

(iv) Consider again **a*b**, for each string a different z is needed to make a word. So there are infinitely many classes.

11. For each string in the infinite set **a***, consider z to be of the form **ba*b**. No two strings from

the former set will both be accepted by the same string from the latter set. So there infinitely many classes.

12. For each string in the infinite set (*x, a different number of close parentheses are needed to bring the word to acceptance, so there are an infinite number of classes.

13. (i) If MOREA were regular then a) MOREB would be regular by symmetry b) MOREA' and MOREB' would both be regular by theorem 11 c) EQUAL would be regular by theorem 12. But we have shown in the chapter (albeit also by indirect proof) that EQUAL is nonregular. So we conclude that MOREA is not regular.

(ii) Because there is a way to divide words so that they can be pumped. Namely let y be some substring of only a's. If there were more a's than b's originally then adding more a's preserves (and enhances) this property.

(iii) Consider **b***; b^p and b^q where p<q are in different classes since $a^{p+1}b^p$ is in MOREA but $a^{p+1}b^q$ in not in MoreA.

14. (i) Infinite union is not regular. Let L be any nonregular language. Each single word of L is a regular language itself, but the union of these infinitely many languages is not regular.
(ii) Infinite intersection is not regular. Let L be any nonregular language. The complement of each single word in L is regular. The intersection of all these languages is L', which is also nonregular.

15. (i) R = **(a+b)***, N = any nonregular language.
(ii) R = any finite language, N = any nonregular language.

16. (i) Suppose PRIME' was regular, then its complement PRIME would be regular but PRIME was already shown to be nonregular. So PRIME' is also nonregular.

(ii) The length of every word in PRIME' has some divisor, k, that is PRIME' = $\{a^{km}\}$. To satisfy the Pumping Lemma, let $x = \Lambda$, $y = a^k$, and $z = a^{k(m-1)}$. Then $xy^nz = a^{kn}a^{k(m-1)}$ which clearly still has a divisor k and is in PRIME'.

(iii) The Pumping Lemma sets the condition on regular languages. We use the contra positive of the conditional as our proof method. If a language does satisfy the lemma then no conclusion about regularity can be drawn.

17. (i) By Theorem 5, any finite set of words is a regular language. By Theorem 10, the set of regular languages is closed under union, so the addition of a finite ser of words to a regular language yields a regular language.

Chapter Ten

(ii) The difference between two sets is the intersection of the first with the complement of the second. Since any finite set of words is a regular language its complement is regular. By Theorem 12, the intersection of two regular languages is regular.

(iii), (iv) if *nonregular + finite = regular* then *regular - finite = nonregular* (in short hand) and we know the latter to be false, hence the antecedent is false. So if a language is nonregular, adding or eliminating a few words does not effect its "regularity".

18. Given an FA that accepts P, take a word in Q, say e.g. *bba*. Then some (maybe none) of the states of the FA have the property that from there *bba* will take them to a + state. Label these states with a tiny "*bba*". Do the same for all words in Q (in theory only since this is infinitely many steps.) Any states in the FA that have a complete set of labels *bba*, ..., for all words in Q, are the final states of the FA for P/Q. Since these states all define Myhil-Nerode classes and there are at most finitely many off them P/Q is regular.

19. (i) The strings (* already define an infinite number of classes.

(ii) Every string contains the substring () at least once. Let y = (), then pumping the word preserves the necessary property.

(iii) Since in PARENTHESES every) must be preceded somewhere by an (so to every *b* will be preceded by an *a*.

20. (i) It should be clear from the discussion that S_1 is the finite set of words accepted by the FA before entering the circuit and S_2 the finite set of words accepted by the FA on the first trip around the circuit. Concatenating $(a^n)^*$ after S_2 simply permits any number of trips around the circuit. [For a more complete discussion of ultimate periodicity see Harrison, pp. 47-50.]

(ii) This problem clarifies the relationship pf the Pumping Lemma to the concept of ultimate periodicity. No matter how many words we take to be in the finite set S_1, and no matter how large we make the circuit (i.e. the size of the finite set S_2), the circuit will bring the number of a's in an accepted word to a magnitude expressed in base 10 by an odd number of digits.

CHAPTER 11

Students have often complained that this chapter belabors some obvious points. Decidability seems trivial as long as the question in question is decidable. As trivial as decidability seems to them now, that is just how impossible un-decidability seems to them later. When a class takes this position they can be given harder questions to puzzle over, such as: given two regular expressions, without converting them into FA's, determine through algebraic means whether they are equivalent. Once we have shown that the question is algorithmicly decidable are there necessarily also algebraic algorithms? If that question seems too easy (of course it isn't, but who knows whether someday a student will answer it), ask them to decide whether the language of one regular expression is a subset of the language of the other. Or find an algebraic way of writing down the regular expression that defines the intersection of the languages of two other regular expressions.

If a class has had some graph theory they may want to investigate methods of deciding whether the language accepted by an FA is infinite simply by looking at the graph itself (disregarding labels). They will have to decide whether there is a path from a start state to a final state that contains an circuit. There are such algorithms that involve pruning the graph of redundant edges and moving the start and final states.

Chapter Eleven

1.

FA_1	a	b
$\pm q_1$	q_2	q_3
$+q_2$	q_2	q_4
$+q_3$	q_5	q_3
$+q_4$	q_5	q_4
q_5	q_5	q_5

FA_2	a	b
$\pm r_1$	r_1	r_2
$+r_2$	r_3	r_2
r_3	r_3	r_3

FA_1 / FA_2		a	b
$-s_1$	$q_1 r_1$	s_2	s_3
s_2	$q_2 r_1$	s_2	s_4
s_3	$q_3 r_2$	s_5	s_3
s_4	$q_4 r_2$	s_5	s_4
s_5	$q_5 r_3$	s_5	s_5

$FA_1 \cap FA_2'$ accepts at $q_1 r_3$, $q_2 r_3$, $q_3 r_3$, $q_4 r_3$

$FA_1' \cap FA_2$ accepts at $q_5 r_1$, $q_5 r_2$

2.

FA_1	a	b
$-q_1$	q_2	q_1
$+q_2$	q_2	q_1

FA_2	a	b
$-r_1$	r_2	r_3
$+r_2$	r_2	r_3
r_3	r_2	r_3

FA_1 / FA_2		a	b
$-s_1$	$q_1 r_1$	s_2	s_3
s_2	$q_2 r_2$	s_2	s_3
s_3	$q_1 r_3$	s_2	s_3

$FA_1 \cap FA_2'$ accepts at $q_2 r_1$, $q_2 r_3$

$FA_1' \cap FA_2$ accepts at $q_1 r_2$

3.

FA_1	a	b
$-q_1$	q_2	q_3
$+q_2$	q_1	q_3
$+q_3$	q_1	q_4
$+q_4$	q_4	q_4

FA_2	a	b
$-r_1$	r_1	r_2
$+r_2$	r_1	r_3
r_3	r_3	r_3

FA_1 / FA_2		a	b
$-s_1$	$q_1 r_1$	s_2	s_3
s_2	$q_2 r_1$	s_1	s_3
s_3	$q_3 r_2$	s_1	s_4
s_4	$q_4 r_2$	s_5	s_4

$FA_1 \cap FA_2'$ accepts at $q_2 r_1$, $q_2 r_3$, $q_3 r_1$, $q_3 r_3$, $q_4 r_1$, $q_4 r_3$, $q_5 r_1$, $q_5 r_3$

$FA_1' \cap FA_2$ accepts at $q_1 r_2$

4. Neither machine has any nonfinal state. Therefore neither machine's complement can have any final states at all. Therefore neither intersection machine can accept any words.

5. These machines do not accept the same language. FA_1 accepts Λ and FA_2 does not. The machine for $FA_1 \cap FA_2'$ will have a final state at the start state.

6.

FA_1/FA_2		a	b
$\rightarrow s_1$	x_1y_1	s_2	s_3
s_2	x_2y_2	s_3	s_4
s_3	x_3y_3	s_3	s_1
s_4	x_2y_1	s_5	s_6
s_5	x_3y_2	s_3	s_1
s_6	x_2y_3	s_3	s_4

$FA_1 \cap FA_2'$ has final state $x_3y_2 = s_6$

7.

FA_1/FA_2		a	b
$\rightarrow s_1$	x_1y_1	s_2	s_3
s_2	x_2y_2	s_4	s_5
s_3	x_3y_3	s_6	s_7
s_4	x_1y_3	s_6	s_8
s_5	x_3y_1	s_2	s_4
s_6	x_2y_1	s_7	s_3
s_7	x_1y_2	s_9	s_5
s_8	x_3y_2	s_9	s_1
s_9	x_2y_3	s_1	s_8

$FA_1 \cap FA_2'$ has final state $x_2y_2 = s_2$ and $x_3y_2 = s_8$.

$FA_1 \cap FA_2'$ has final states $x_1y_1 = s_1$ and $x_1y_3 = s_4$.

8. *aaa, aab, aba, abb, baa, bab, bba, bbb, aaaa, aaab, aaba, aabb, abaa, abab, abba, abbb, baaa, baab, baba, babb, bbaa, bbab, bbba, bbbb, aaaaa, aaaab, aaaba, aaabb, aabaa, aabab, aabba, aabbb, abaaa, abaab, ababa, ababb, abbaa, abbab, abbba, abbbb, baaaa, baaab, baaba, baabb, babaa, babab, babba, babbb, bbaaa, bbaab, bbaba, bbabb, bbbaa, bbbab, bbbba, bbbbb*

9. yes 10. no 11. no 12. no
13. (i) finite (ii) finite (iii) infinite (iv) infinite
14. The blue paint algorithm for FA's works equally well for TG's.
15. The process of testing all strings in the range $N \leq$ length of string $\leq 2N$ works equally well for NFA's.
16. The process of problem 15 works as well for NFA-λ's.
17. By Theorem 15, an FA with N states that accepts a non-empty language accepts a word of length $\leq N$. Since the original machine was an FA, we know that the final state must have had out-edges. By changing all of these into loops, we change any string that would have passed through a final state into a string that dead-ends into a final state. No matter how many fewer than N letters it takes to get to the final state, the rest of the word will stay at that state. Therefore, since a string of length N is long enough to reach any final state in the machine, it must be accepted if any of its prefixes is accepted.
This is an effective procedure. Because the original machine was an FA, we know that there are exactly as many edges leaving each state as there are letters in the original alphabet. (When change the alphabet to $\Sigma = \{x\}$ we may get some duplication of edges, but this does not change the upper bound.) Suppose there were m letters in the original alphabet. Then, at each of the N letters in the string to be tested we have a choice of exactly m edges to follow, so the upper bound is m^N.
18. Follow the a-edge out of the original start state and paint its destination state blue. From this point (instead of from the original start state) follow the blue paint procedure of the chapter.
19. (i) Test all strings of length $\leq N$ that contain a b. If the machine accepts none of them, it accepts no words that contain a b. Alternatively, copy the machine. Follow the b-edge out of start and (as in Problem 18) proceed from there to see whether

the machine accepts any words beginning with *b*. If so, stop; the answer is yes. Otherwise, proceed. Follow the *b*-edges out of each of the states that can be reached on one letter. Apply the blue-paint algorithm from there, etc.
(ii) Test all strings of even length $\leq N$.

20. Construct FA_1 for the language of $\mathbf{r}_1$ and FA_2 for the language of $\mathbf{r}_2$. Intersect FA_2 with FA_1'. If the resulting machine accepts no words, then the language of $\mathbf{r}_1$ is contained in the language of $\mathbf{r}_2$. (If it is properly contained in the language of $\mathbf{r}_2$, the intersection of FA_1 with FA_2' accepts a nonempty set.)

Chapter 12

On the surface we present one primary reason for introducing CFG's: regular expressions were inadequate to define all interesting languages and therefore we turned to the method of defining human languages to find a more powerful tool. If this were really the case, and our whole goal was to find the most powerful language defining structure, why did we waste time on Part I at all since Part II is better (dominating), and Part III will be better still. There must be a reason why Parts I and II exist. Students should be asked to figure this out for themselves.

Loosely speaking, Chomsky's goal in inventing generative grammars was that he had a fundamental theory of why humans can learn to speak and why all human languages are so similar (assuming that they are). He reasoned that all human brains come hard-wired (like nest-building in birds) with a language grammar of everything but the terminals. Each human language invented need only supply the terminals "chair", "table", "cat", etc. Infants *know* the grammar already and are eager to learn the terminals so that they can form sentences. I like to have the students speculate on this possibility and to contrast it to computers, programs, and AI. If this were really literally the case why would so few human-uttered sentences be grammatical?

I think the term Polish Notation for Łukasiewicz Notation is politically incorrect and demeaning. The man deserves personal credit, not synecdoche. This is also a topic which, if not already presented in some other course, all CS students should learn.

To make the distinction between ambiguous grammars and ambiguous languages meaningful takes some classroom discussion. Just to say that there is a distinction will not impress the students as much as forcing them to explain it themselves.

Chapter Twelve

1. There is only one nonterminal S. There are only two productions, S → aS and S → bb. We must apply the second production exactly once to eliminate the nonterminal S from the working string and form a word. (Therefore all words in the language of the grammar end in *bb*.) As soon as we apply it we are finished, since there can never be more than one occurrence of S in the working string. If we apply the first no times at all we make the word *bb*. If we apply it exactly n times, we make $a^n bb$. Thus, we can make all of a^*bb. We can make nothing else because we have only the two productions offered, and the substring *bb* must terminate the word.

2. The grammar has three nonterminals. S → XYX gives us a word of three factors. The outside factors are each of type X which we recognize as having the form as grammars for the language of the expression **(a+b)***. The center factor is type Y, which allows only the string *bbb*. Since all strings in the language start from S all must have the Y factor in the middle, with an X factor concatenated before it and one after it, just as in the regular expression **(a+b)*bbb(a+b)***.

3. (i) **a(a+b)*** The X factor in this grammar is exactly like the X factor in the grammar of Problem 2, and generates the same language, namely **(a+b)***. Since S gives the single production S → aX, the grammar produces all words that begin with *a*.

   (ii) **(a+b)*a(a+b)*a(a+b)*** Again, what the X factor produces is entirely arbitrary. The S production requires exactly two a's (in addition to any that might be introduced from X). So this grammar gives the language of all strings that have at least two *a*'s.

4. (i) By taking X → Λ, we get the null string from X. We must apply this production once to eliminate the nonterminal X. To make the substring b^n for any *n*, we apply the other X production exactly *n* times before eliminating X.

   (ii) We have shown that X gives us all of b*. We can derive any of the strings in the language from wherever the nonterminal X occurs in a workins string, by applylng productions as shown above. The requisite *a*'s are included in this S production.

   (iii) Consider the two productions S → SS and S → Λ. By applying the former production n+1 times and then applying the latter to any one of the resulting S's, we derive a working string of exactly n S's. Since we have shown that S can produce any number of S's (including 0), we see that we can use it to derive something we might call S*. Each S can generate XaXaX, which is a word from **b*ab*ab***. Therefore the language generated is **(b*ab*ab*)*** = [all strings with an even number of a's]-[b*]+[Λ].

Chapter Twelve

(iv) This language includes no words with an odd number of *a*'s, since each repetition of the factor(b*ab*ab*) introduces exactly two of them. The number of *b*'s introduced with any factor is entirely arbitrary. However, there is no way to generate any b's at all without also introducing two *a*'s. Therefore, of all the words with no b's, only Λ can be generated.

(v) We have already shown that we can make all words over (a+b)* that have some positive even number of *a*'s, and Λ. We must show that adding the new production S → XS to the grammar gives us the possibility of producing the words in b* without adding undesirable strings to the language. By using the sequence S ⇒ XS ⇒ XΛ ⇒ X we can easily see that we can now derive the strings in b* from S. Since we have not introduced an unpaired *a* into the grammar, we may safely assert that we can now produce all strings with an even number of *a*'s.

5. This grammar generates the language of all words that either begin with *a* or contain the substring *baa* (or both). The word *abaa* can be derived in two ways:

S ⇒ aX ⇒ abX ⇒ abaX ⇒ abaaX ⇒ abaaΛ = abaa

S ⇒ XbaaX ⇒ abaaX ⇒ abaaΛ = abaa

6. (i) Sentence ⇒ subject predicate
⇒ noun-phrase predicate
⇒ adjective noun-phrase predicate
⇒ adjective article noun-phrase predicate
⇒ adjective article noun predicate
⇒ adjective article noun verb noun-phrase
⇒ adjective article noun verb adjective noun-phrase
⇒ adjective article noun verb adjective article noun-phrase
⇒ adjective article noun verb adjective article noun
⇒ itchy the bear hugs jumpy the dog

(ii) Change rules 3, 4 and 5:

3. A *noun-phrase* can be an *article* followed by an *a-phrase*.
4. An *a-phrase* can be an *adjective* followed by an *a-phrase*.
5. An *a-phrase* can be a *noun*.

(iii) Sentence ⇒ subject predicate
⇒ noun-phrase predicate
⇒ article noun-phrase predicate
⇒ article article noun-phrase predicate
⇒ article article article noun-phrase predicate
⇒ article article article noun verb noun-phrase
⇒ article article article noun verb noun
⇒ the the the cat follows cat

(iv) The alternate rules listed in part (ii) prevent this.

Chapter Twelve

7. (i) $S \to Sb \mid a$

(ii)
$S \to XY$
$X \to aX \mid \Lambda$
$Y \to bY \mid \Lambda$

(iii) $S \to SS \mid baa \mid abb \mid \Lambda$

8. (i)
$S \to aS \mid bX \mid \Lambda$
$X \to aS \mid bY \mid \Lambda$
$Y \to aS \mid \Lambda$

(ii)
$S \to aS \mid bY$
$X \to aX \mid bY$
$Y \to aY \mid bZ \mid \Lambda$
$Z \to aZ \mid \Lambda$

(iii)
$S \to bS \mid aX \mid \Lambda$
$X \to aX \mid \Lambda$

(iv)
$S \to aS \mid bX \mid \Lambda$
$X \to bX \mid aY \mid \Lambda$
$Y \to bX \mid \Lambda$

(v)
$S \to aX \mid bY$
$X \to aX \mid bW$
$Y \to bY \mid aZ$
$W \to bW \mid aX \mid \Lambda$
$Z \to aZ \mid bY \mid \Lambda$

9. Consider the choices of productions from nonterminal A. Eventually, if the grammar produces a word, each A in the working string must be replaced by exactly one *a*. Furthermore, the choices $A \to bA \mid Ab$ mean that when we are adding the terminal *b* to the working string we are not changing the number of A's in that working string. The production $A \to AAA$ means that we can at any time add exactly two A's to a working string. This will not change the parity of the number of A's in the working string. The first production in the grammar gives us two A's, so the parity must be even. Eventually, each of those A's must terminate as *a*'s, so there cannot be a word without any *a*'s in this language. This language is the langauge defined in Problem 4 without Λ.

10. This grammar generates the language of all words containing a (positive) number of *b*'s divisible by 3.

Chapter Twelve

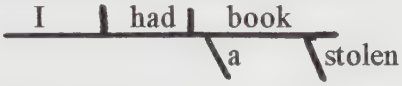
11. S → SS | EXE
X → aX | a
E → EQUAL

12. We can convert the CFG for any CFL into its transpose by replacing the right sides of each production with its reverse: S → XY becomes S → YX.

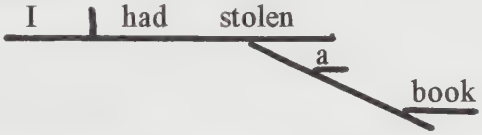
13. In the productions, the nonterminals on the left side of S become the letters of the word and the a's on the right side of S keep count of the number of letters in the word.

14. (i) At least two;
An observation: I conclude that the watch is broken.
A command: It must be done - break my watch.

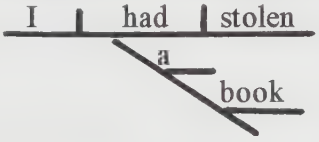
(ii) I had a book and someone stole it.



I arranged the theft of a book.

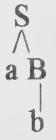


I was at the point of stealing the book when...



15. (i) CFG 1

CFG 5



(ii) CFG 2

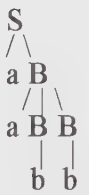
CFG 3

CFG 4

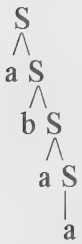
Chapter Twelve

(iii) CFG 1

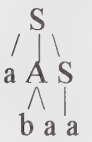
CFG 5



(iv) CFG 2

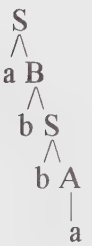


CFG 4

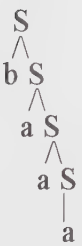


(v) CFG 2

CFG 5

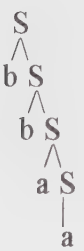


(vi) CFG 2

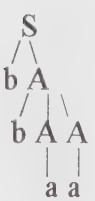


(vii) CFG 5

(viii) CFG 2

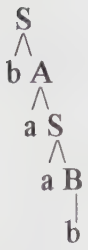


CFG 5



Chapter Twelve

(ix) CFG 5



16. (i) bababababab
(ii) aab
(iii) aaaaaaaa
(iv) aaa
(v) aaaa

17. (i) $S \Rightarrow \underline{X}aX \Rightarrow a\underline{X}aX \Rightarrow a\Lambda a\underline{X} \Rightarrow a\Lambda a = aa$
$S \Rightarrow Xa\underline{X} \Rightarrow a\underline{X}\Lambda \Rightarrow aa\underline{X} \Rightarrow aa\Lambda = aa$

(ii) $S \Rightarrow a\underline{S}X \Rightarrow aa\underline{S}XX \Rightarrow aa\Lambda X\underline{X} \Rightarrow aa\underline{X}a \Rightarrow aaaa = aaaa$
$S \Rightarrow a\underline{S}X \Rightarrow a\Lambda\underline{X} \Rightarrow aa\underline{X} \Rightarrow aaa\underline{X} \Rightarrow aaaa = aaaa$

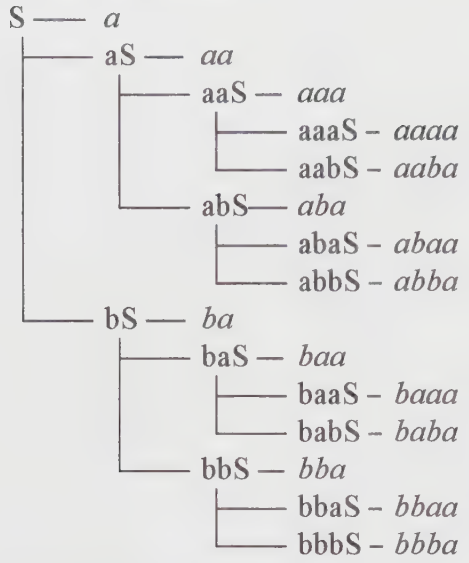
(iii) $S \Rightarrow a\underline{S} \Rightarrow aa\underline{S} \Rightarrow aa\Lambda = aa$
$S \Rightarrow aa\underline{S} \Rightarrow aa\Lambda = aa$

(iv) (i) $S \rightarrow bS \mid aX$
$X \rightarrow aX \mid bX \mid \Lambda$
(ii) $S \rightarrow aX$
$X \rightarrow aX \mid a$
(iii) $S \rightarrow aS \mid bS \mid \Lambda$

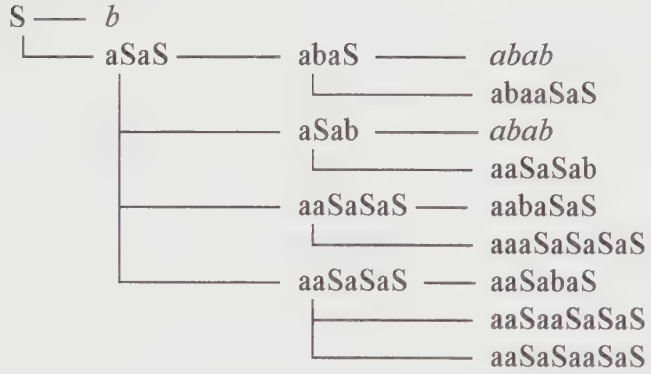
(v) (i) $S \rightarrow bS \mid aX \mid a$
$X \rightarrow aX \mid bX \mid a \mid b$
(ii) same as above
(iii) $S \rightarrow aS \mid bS \mid a \mid b$

Chapter Twelve

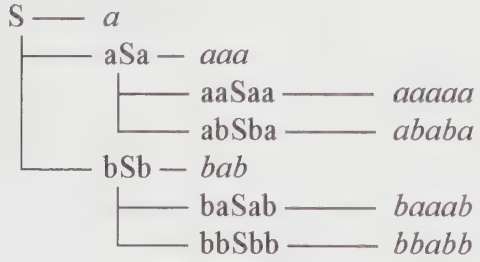
18. (i) not ambiguous



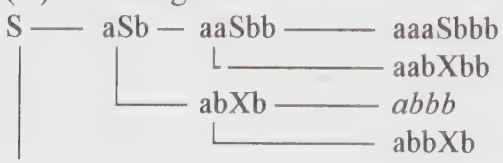
(ii) ambiguous



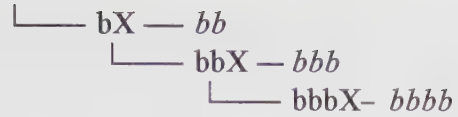
(iii) not ambiguous



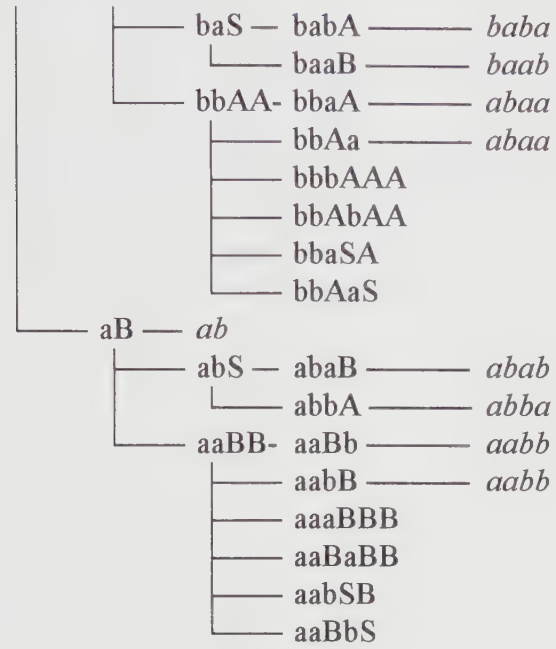
(iv) not ambiguous



Chapter Twelve



(v) ambiguous



19. (i) **123
(ii) +*123
(iii) *1+23
(iv) **1+234
(v) +*+1234
(vi) +1*2+34
(vii) ++1*234

20. The binary operators are $\vee$, $\wedge$ and $\supset$. The unary operator is $\sim$. Using prefix notation, we can distinguish between infix $\sim p \vee q = (\sim p) \vee q$ and $\sim p \vee q = \sim(p \vee q)$ by writing $\vee \sim pq$ and $\sim \vee pq$ respectively. Construct a grammar analogous to the one used to generate the prefix trees and proceed in the same fashion.

Chapter 13

We make an enormous big deal out of Chomsky Normal Form, in part for the parallel to regular grammars, in part for proving later that there is a PDA for every CFG, in part for parsing by CYK, and in part to prepare for Kuroda Normal Form for CSLs. However the true reason for all the fanfare is that this conversion in its naive form provides two wonderful examples of when what seems like a completely well defined algorithm is actually not.

If the teacher would rather cover the whole mater in one class, fine.

Notice also that there is mathematical induction disguised in the proof of the grammar for **EQUAL**.

Chapter Thirteen

1. (i) $S \to SS \mid aaa \mid b$

   (ii) $S \to XYX$
   $X \to aX \mid bX \mid \Lambda$
   $Y \to aaa \mid bbb$

   (iii) $S \to aX \mid bS \mid \Lambda$
   $X \to aY \mid bS \mid \Lambda$
   $Y \to bS \mid \Lambda$

   (iv) $S \to aS \mid bX$
   $X \to aX \mid bY$
   $Y \to aS \mid bX \mid \Lambda$

   (v) $S \to aaS \mid abS \mid baS \mid bbS \mid a \mid b$

   (vi) $S \to X \mid Y$
   $X \to bX \mid Xb \mid a$
   $Y \to aY \mid Ya \mid b$

   (vii) $S \to X \mid Y$
   $X \to bX \mid aW$
   $W \to bW \mid aX \mid \Lambda$
   $Y \to aY \mid bZ \mid \Lambda$
   $Z \to aZ \mid bY$

2. (i) S → aaaS | bS | aaa | b

   (ii) S → aS | bS | X
   X → aaaY | bbbY
   Y→ aY | bY | Λ

   (vi) S → bX | aY | a | b
   X → bX | aB B → bB | Λ
   Y → aY | bA A → aA | Λ

   The other four grammars are already in regular format.

3. (i) $\mathbf{b^*(a+b)a^*}$ nonnull words in which all *a*'s follow all *b*'s

   (ii) $\mathbf{b^*(ab^*a)b^*(aa+b)}$ nonnull words with an even number of *a*'s

4. (i) $\mathbf{(aa+ab+ba+bb)^*}$ words of even length

   (ii) $\mathbf{(ab+ba)^*}$ words of even length where no letter is tripled and initial and final letters are not doubled

5. (i) $\mathbf{a(ba)^*(b+ba) + b(ab)^*(a+ab)}$ words of length ≥2 where *a*'s and *b*'s alternate

   (ii) $\mathbf{a^*(b+A)a^*(b+A)a^*a}$ words ending in a having at most 2 *b*'s

6. (i) $\mathbf{a^*(a+b[a^*(a+ba^*a+ba^*ba^*a)])}$ words ending in *a* having at most 3 *b*'s

   (ii) $\mathbf{(a+b)^*aa(a+b)^*(a+b)}$ words including the substring *aa* followed by at least one more letter

7. (i) $S \rightarrow SS \mid (S) \mid S+S \mid S^* \mid \mathbf{a} \mid \mathbf{b}$

   (ii) The language is nonregular. (Parentheses may be arbitrarily deeply nested.)

8. (i) $S \rightarrow aS \mid bS \mid \Lambda$

   (ii) $S \rightarrow aaaX$
   $X \rightarrow aX \mid \Lambda$

   (iii) $S \rightarrow aaX$
   $X \rightarrow aX \mid \Lambda$

9. Every edge of the TG is labeled with a word. Give the states names (as in the FA algorithm) and then, for each transition, create a production of the form

   STATE OF ORIGIN → (*edge label*) *DESTINATION STATE*

10. We can turn the bad grammar into a machine of the type formed in the proof of Kleene's theorem that is like a TG but with edges that are labeled with regular expressions. Consider each nonterminal to be a state, from which there will be one transition to correspond to each production. The Λ-productions all go to a final state. This machine can be reduced to a regular expression by the algorithm of Kleene's theorem.

Chapter Thirteen

11. (i) $S \to aX \mid bX \mid a \mid b$
$X \to a \mid b$

(ii) $S \to aX \mid bS \mid a \mid b$
$X \to aX \mid a$

(iii) $S \to aS \mid bX \mid b$
$X \to aX \mid a$

(iv) $S \to XaX \mid bX \mid a \mid Xa \mid aX \mid b$
$X \to XaX \mid Xa \mid aX \mid a \mid XbX \mid Xb \mid bX \mid b$

12. (i) Choose a symbol for a new nonterminal, $*$, and add the production $* \to \Lambda$. This will not change the grammar because $*$ is not derivable from S.

(ii) Add a new unit production $* \to$ N, where N is some nonterminal already in the grammar. Again this will not adversly effect the grammar for the same reason as above.

13. (i) $S \to aS \mid Yb$
$Y \to bY \mid b$

(ii) $S \to AA \mid BB$
$A \to BB$
$B \to abB \mid b \mid bb$

(iii) $S \to AB \mid aB \mid a \mid Bb \mid b$
$A \to aB \mid a \mid Bb \mid b$
$B \to aB \mid a \mid Bb \mid b$

14. (i) $S \to SS \mid a$ (already in CNF)

(ii) $S \to aR \mid SR \mid a$
$R \to SA$
$A \to a$

(iii) $S \to AR$
$R \to XX$
$X \to AS \mid BS \mid a$
$A \to a$
$B \to B$

(iv)
$E \to EP$ $\quad$ $E \to 7$
$E \to ET$ $\quad$ $X \to +$
$E \to LR$ $\quad$ $Y \to *$

$P \to XE$ $\quad L \to ($

$T \to YE$ $\quad C \to)$

$R \to EC$

(v) |1- and 2-letters| $S \to a \mid b \mid AA \mid AB \mid BA \mid BB$

|3-letters| $S \to AR_1 \mid AR_2 \mid AR_3 \mid AR_4 \mid BR_1 \mid BR_2 \mid BR_3 \mid BR_4$

|4-letters| $S \to R_1R_2 \mid R_1R_3 \mid R_1R_4 \mid R_2R_1 \mid R_2R_2 \mid R_2R_3 \mid R_2R_4 \mid R_3R_2 \mid R_3R_3 \mid R_3R_4 \mid R_4R_2$

|5-letters| $S \to BR_5 \mid AR_5 \mid R_7R_2 \mid R_6R_2 \mid R_5B \mid R_5A$

|6-letters| $S \to R_2R_5$

$R_1 \to AA$ $\quad R_4 \to BB$ $\quad R_7 \to R_2B$

$R_2 \to AB$ $\quad R_5 \to R_2R_2$ $\quad A \to a$

$R_3 \to BA$ $\quad R_6 \to R_2A$ $\quad B \to b$

(vi) $S \to SR_1 \mid SA \mid AS \mid a$

$S \to R_5R_3 \mid R_4R_2 \mid R_4R_3 \mid R_1R_3 \mid R_1B \mid R_4B \mid AR_3 \mid AB$

$S \to R_2R_5 \mid R_2R_4 \mid R_2R_1 \mid R_3R_1 \mid R_2A \mid R_3A \mid BR_1 \mid BA$

$R_1 \to AS$ $\quad R_4 \to SA$ $\quad A \to a$

$R_2 \to SB$ $\quad R_5 \to SR_1$ $\quad B \to b$

$R_3 \to BS$

(vii) No change. This grammar generates no words, but it is in CNF.

15 (i) $S \to AA$ $\quad Z \to AA$

$X \to AA$ $\quad A \to a$

$Y \to AA$

(ii) $S \to SS \mid AS \mid a$

$A \to SS \mid AS \mid a$

16. Not necessarily. For example if the first rule in a CFG was S → XS | XY then adding the rule S → Λ would change the language. Instead, we need a new start symbol S' and the productions (not in Chomsky normal form) S' → S | Λ.

17. (i) S⇒ AA ⇒ aBA ⇒ abBA ⇒ abbBA ⇒ abbΛA ⇒ abbaB ⇒ abbaΛ

18. Yes. For example the word *aababb* can be derived in at least two ways from the grammar: S → XY | XX | YY, X → XY | XX | a, Y → YX | YY | b.

19. Draw the derivation tree for the word in question. Start with the original production from the Start symbol S. Thereafter, find on the tree the node corresponding to the rightmost nonterminal in the working string and apply to the nonterminal whatever production shows in the derivation tree. Since, at any point in the derivation of a word, if there is at least one nonterminal in the working string, there is a rightmost nonterminal, this works for all grammars. Since every word derivable from a CFG has at least one derivation tree and since that tree (by definition) terminates when the word is formed, the process is finite.

20. Since the conditions specify that Λ is not a word in L we know (by Theorem 21) that there is a CFG in CNF for L. We derive our grammar from this one. We know that in the process of deriving a word in L we start with a string of all nonterminals (the single nonterminal S) and end up with a string of all terminals (the word). We observe that whatever nonterminals were used in the derivation process have all been turned into terminals eventually, since none can have been eliminated directly by applying a Λ-production. We do not have to worry about both nonterminals on the right of a production arrow; if we know that they will both eventually turn into terminals then, in particular, the first one will. Start by separating the productions.

STEP 1. For all "dead" productions, that is, of the form $N \rightarrow \alpha$, $\alpha \in \Sigma$, paint α blue and paint the nonterminal on the left of the production arrow red (at that place only).

STEP 2. Wherever a nonterminal that is red somewhere appears to the right of a production arrow, paint it red.

STEP 3. For all productions having a right side that begins with a red nonterminal, replace each of them by the set of all blue terminals that can be generated by that red nonterminal. Now paint the nonterminal to the left of the production arrow in

each of these productions red (again, only at that place).

Repeat STEPS 2 and 3 until nothing new is colored. All useful productions (see Chapter 18) now begin with a terminal. All useful nonterminals will be red. Any productions that do not now begin with a terminal cannot ever be used in the derivation of a word.

Chapter 14

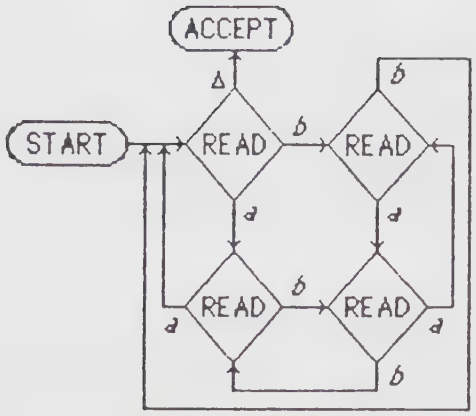
We define PDAs *as* pictures, not *illustrated* by pictures. Manna does this too, but slightly differently. We correspond runs of inputs to trace tables, but we never replace the pictures by tables (that is until the proof that every PDA corresponds to a CFG). To define PDAs (or FAs or TMs for that mater) by transition functions is a simple exercise in the elementary theory of using function notation but it adds nothing of pedagogical merit to the discussion. We could eliminate the pictures from all of graph theory and be left with graph-less theory with no intuitive combinatorial value. Let us not do this.

PDA languages can also be defined by acceptance by empty STACK, completely read TAPE, etc. If we had world enough and time we would prove the equivalence of these other definitions to the one given here, but this task seems less exciting than progressing, as quickly as possible, to TMs where the true surprises lie.

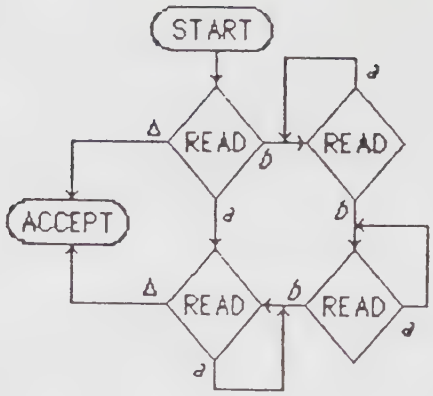
It would be nice to prove by an elementary argument at the early stages of this chapter, that **PALINDROME** cannot be accepted by a deterministic PDA. When I figure out one of these slick proofs I will print a new edition. Till then the necessity of the non-deterministic nature of PDAs will have to rest on arguments like the one given here. That generation by CFG allows (and requires) choice and that that is why PDAs, in order to correspond to them, must also have choice in the nature of non-determinism, is bogus. Remember that generation of words from regular expressions also allows choice but FAs can be defined as deterministic (this is because for them NFA=FA). So this feeling is imprecise to the point of being misleading. Try to do something about this in class. Personally, it is still a heuristic mystery why CFLs are not closed under complementation.

Chapter Fourteen

1.



2.



3. (i)

STATE	STACK	TAPE
START	Δ	*abb*
$READ_1$	Δ	*bb*
PUSH *a*	*a*	*bb*
$READ_2$	*a*	*b*
$READ_3$	*a*	Δ
POP	Δ	Δ
$READ_4$	Δ	Δ
POP	Δ	Δ
ACCEPT		

(ii)

STATE	STACK	TAPE
START	Δ	*abab*
$READ_1$	Δ	*bab*
PUSH *a*	*a*	*bab*
$READ_2$	*a*	*ab*
$READ_3$	*a*	*b*
REJECT		

(iii)

STATE	STACK	TAPE
START	Δ	$aabb$
$READ_1$	Δ	abb
PUSH a	a	abb
$READ_2$	a	bb
PUSH a	aa	bb
$READ_2$	aa	b
$READ_3$	aa	Δ
POP	a	Δ
READ4	a	Δ
POP	Δ	Δ
REJECT		

(iv)

STATE	STACK	TAPE
START	Δ	$aabbbb$
$READ_1$	Δ	$abbbb$
PUSH a	a	$abbbb$
$READ_2$	a	$bbbb$
PUSH a	aa	$bbbb$
$READ_2$	aa	bbb
$READ_3$	aa	bb
POP	a	bb
$READ_4$	a	b
$READ_3$	a	Δ
POP	Δ	Δ
$READ_4$	Δ	Δ
POP	Δ	Δ
ACCEPT		

4. (i) $\{a^n b^{2n}\}$

(ii) $S \rightarrow aSbb \mid abb$

(iii) No. (It can be proven nonregular with the weak form of the pumping lemma.)

5. (i)

STATE	STACK	TAPE
START	Δ	*aaabbb*
$READ_1$	Δ	*aabbb*
PUSH *a*	*a*	*aabbb*
$READ_2$	*a*	*abbb*
PUSH *a*	*aa*	*abbb*
$READ_2$	*aa*	*bbb*
PUSH *a*	*aaa*	*bbb*
$READ_2$	*aaa*	*bb*
POP	*aa*	*bb*
$READ_3$	*aa*	*b*
POP	*a*	*b*
$READ_3$	*a*	Δ
POP	Δ	Δ
$READ_3$	Δ	Δ
POP	Δ	Δ
ACCEPT		

(ii)

STATE	STACK	TAPE
START	Δ	*aaabab*
$READ_1$	Δ	*aabab*
PUSH *a*	*a*	*aabab*
$READ_2$	*a*	*abab*
PUSH *a*	*aa*	*abab*
$READ_2$	*aa*	*bab*
PUSH *a*	*aaa*	*bab*
$READ_2$	*aaa*	*ab*
POP	*aa*	*ab*
$READ_3$	*aa*	*b*
POP	*a*	*b*
$READ_3$	*a*	Δ
POP	Δ	Δ
$READ_3$	Δ	Δ
POP	Δ	Δ
ACCEPT		

(iii)

STATE	STACK	TAPE
START	Δ	*aaabaa*
$READ_1$	Δ	*aabaa*
PUSH *a*	*a*	*aabaa*
$READ_2$	*a*	*abaa*
PUSH *a*	*aa*	*abaa*
$READ_2$	*aa*	*baa*
PUSH *a*	*aaa*	*baa*
$READ_2$	*aaa*	*aa*
POP	*aa*	*aa*
$READ_3$	*aa*	*a*
POP	*a*	*a*
$READ_3$	*a*	Δ
POP	Δ	Δ
$READ_3$	Δ	Δ
POP	Δ	Δ
ACCEPT		

(iv)

STATE	STACK	TAPE
START	Δ	*aaaabb*
$READ_1$	Δ	*aaabb*
PUSH *a*	*a*	*aaabb*
$READ_2$	*a*	*aabb*
PUSH *a*	*aa*	*aabb*
$READ_2$	*aa*	*abb*
PUSH *a*	*aaa*	*abb*
$READ_2$	*aaa*	*bb*
PUSH *a*	*aaaa*	*bb*
$READ_2$	*aaaa*	*b*
POP	*aaa*	*b*
$READ_3$	*aaa*	Δ
POP	*aa*	Δ
$READ_3$	*aa*	Δ
POP	*a*	Δ
CRASH		

6. (i) Consider the machine in sections.

1. The machine accepts $\Lambda = a^0 s$ where length(s) = 0.
2. The first letter of any other string must be a. That a is stored. As long as the machine continues to read a's they continue to be stored. No word consisting only of a's can be accepted.
3. At the first b, control passes to the second loop. For each letter read (including that b) the STACK is popped once. If there are fewer a's in the STACK than letters from the first b to the end of the word the string crashes. When there are no more letters on the TAPE (read Λ) the string crashes if there are a's left on the STACK. Only those words are accepted that have the same number of letters from the first b to the end of the word as initial a's.

(ii) $S \rightarrow aSb \mid aSa \mid ab$

(iii) Use the strong version of the Pumping Lemma. Assume a machine that has n states, and take the counterexample of $(a^{2n}ba^{2n-1})$. There must be a circuit within the first clump of a's, and this can displace the b that initiates the second half.

7. (i)

State	Tape	Stack
Start	aababb	Λ
Read (a)	ababb	
Push a		a
Read (a)	babb	
Push a		aa
Read (b)	abb	
Push b		baa
Read (a)	bb	
Pop (b)		aa
Read (b)	b	
Pop (a)		a
Read (b)	Λ	
Pop (a)		Λ
Read (Λ)		
Pop (Λ)		
Accept		

State	Tape	Stack
Start	abbbaaab	Λ
Read (a)	bbbaaab	
Push a		a
Read (b)	bbaaab	
Push b		ba
Read (b)	baaab	
Push b		bba
Read (b)	aaab	
Push b		bbba
Read (a)	aab	
Pop (b)		bba
Read (a)	ab	
Pop (b)		ba
Read (a)	b	
Pop (b)		a
Read (b)	Λ	
Pop (a)		Λ
Read (Λ)		
Pop (Λ)		
Accept		

(ii) Focus on the four loops dividing the machine into four steps. The first loop reads n a's pushing then on the stack, the next pushes m b's on the stack. The third loop matches the second clump of a's with the b's on the stack to ensure that there are m a's. Finally the last loop matched the last clump of b's with the a's on the stack hoping to find exactly n of them.

8. (i) $S \rightarrow aSb \mid aXb$
$X \rightarrow bXa \mid ba$

(ii) The strings a^nba are all in different Myhil-Nerode classes since b^n turns only one of them into a word.

9. (i)

STATE	STACK	TAPE
START	Δ	*ab*
$READ_1$	Δ	*b*
PUSH *X*	*X*	*b*
$READ_1$	*X*	Δ
POP_1	Δ	Δ
$READ_2$	Δ	Δ
ACCEPT		

(ii)

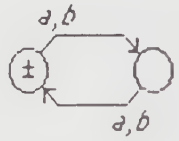
STATE	STACK	TAPE
START	Δ	*bbba*
$READ_1$	Δ	*bba*
PUSH *X*	*X*	*bba*
$READ_1$	*X*	*ba*
PUSH *X*	*XX*	*ba*
$READ_1$	*XX*	*a*
POP_1	*X*	*a*
$READ_2$	X	Δ
POP_1	Δ	Δ
$READ_2$	Δ	Δ
POP_2	Δ	Δ
ACCEPT		

10. The algorithm for accepting any string of length $2n$ ($n > 0$) is to read n letters while looping at $READ_1$ and then cross to POP_1 on the $n+1^{th}$ letter and continue the rest of the word from

there. The loop at $READ_1$ puts some number of x's into the STACK. After the transition is made to POP_1 nothing more can be pushed. Starting with the transition letter, for every letter we read we must be able to pop a counter. When the TAPE reads Δ, the STACK must also read Δ. In other words, the number of letters read in the first loop (counters in the STACK) must equal the number of letters read in the second loop, so the total number of letters in the string must be a double, that is, an even number.

No word of odd length can be accepted because it is impossible to break it into two equal parts. If $\lfloor n/2 \rfloor$ is the transition letter, there will be one fewer counter in the STACK than remaining letters on the TAPE and the string will crash popping Δ at POP_1. If $\lceil n/2 \rceil$ is the transition letter, the TAPE will empty first, and the string will crash popping x at POP_2.

11. $S \rightarrow SS \mid aa \mid ab \mid ba \mid bb$



12. (i)

STATE	STACK	TAPE
START	Δ	aa
$READ_1$	Δ	a
PUSH x	x	a
$READ_1$	x	Δ
POP_1	Δ	Δ
$READ_1$	Δ	Δ
POP_1	Δ	Δ
ACCEPT		

(ii)

STATE	STACK	TAPE
START	Δ	$babaaa$
$READ_1$	Δ	$abaaa$
PUSH x	x	$abaaa$
$READ_1$	x	$baaa$
PUSH x	xx	$baaa$

	$READ_1$	XX	aaa
*******	PUSH X	XXX	aaa
	$READ_1$	XXX	aa
	POP_1	XX	aa
	$READ_2$	XX	a
	POP_1	X	a
	$READ_2$	X	Δ
	POP_1	Δ	Δ
	$READ_2$	Δ	Δ
	POP_2	Δ	Δ
	ACCEPT		

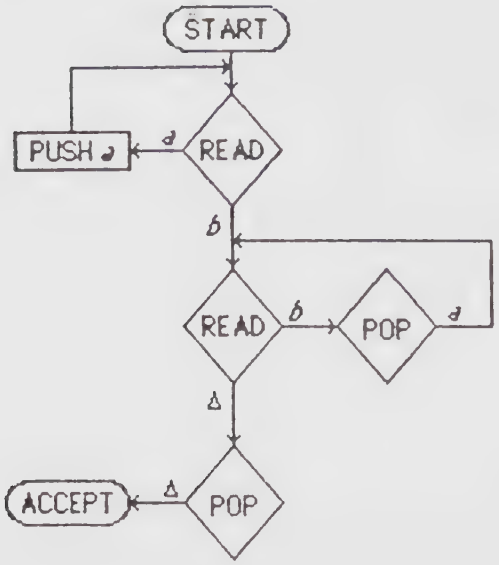
*******We have to start the second part of the process with the first of the trailing a's; that is, when the TAPE is reduced to exactly as many a's as there are X's in the STACK; reading the first of those a's should initiate phase 2.

(iii) For a word to be accepted by this machine it must contain at least one a that is not followed by any b's. The string $baba$ must crash either by reading Δ in $READ_1$, by popping Δ in POP_1 or by reading b in $READ_2$.

(iv) For a word to be accepted by this machine it must have even length. (The structure of the machine is exactly like that of the machine for Problem 9.) If the last letter counted is the second b in the string, the string will crash reading Δ in POP_1. If the last letter counted is the a following that b, the string will crash popping Δ in POP_2.

13. If length(s) = n, then every string in TRAILINGCOUNT has length $2n$, in other words, all strings are of even length. Further, while the first n letters are arbitrary, the last n letters must all be a's. We have established (by the pairing of X's popped to letters read) that this machine accepts only words of even length. The transition from the READ - PUSH loop is a. Thereafter, reading any other letter will cause the string to crash. Therefore, this machine accepts exactly those strings of even length of which the second half is all a's.

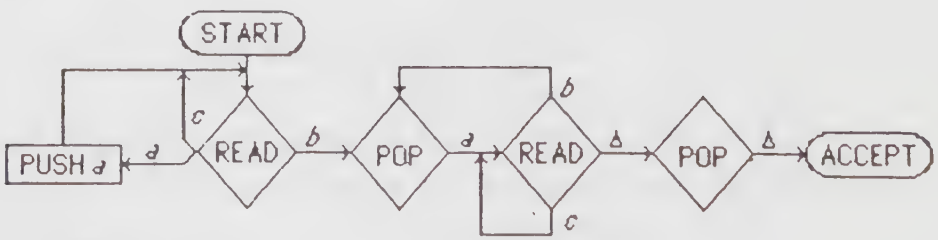
14.



15. (i) *c, ab, cc, ccc, cccc, ccccc, cccccc, abc, acb, cab, aabb, abcc, acbc, cabc, accb, cacb, ccab, aabbc, aabcb, aacbb, acabb, caabb, abccc, acbcc, cabcc, acccb, caccb, ccacb, cccab, aaabbb, aabbcc, aabcbc, aacbbc, acabbc, caabbc, aabccb, aacaca, acabcb, caabcb, aaccbb, acacbb, caacbb, accabb, cacabb, ccaabb, abcccc, acbccc, cabccc, accbcc, cacbcc, ccabcc, acccbc, caccbc, ccacbc, cccabc, accccb, caccccb, ccacb, cccacb, ccccab*

(ii) Use a counterexample of the form ($a^n b^n$), which is a subset of this language.

(iii)

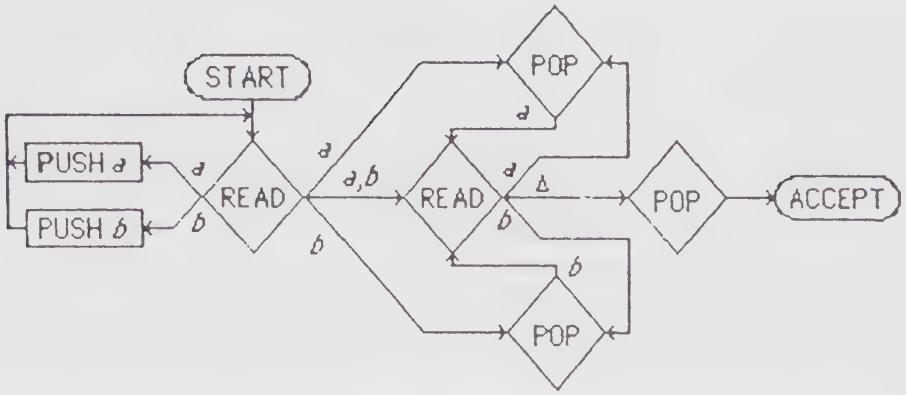


(iv) $S \rightarrow YaXbY \mid Y$

$X \rightarrow aXb \mid XY \mid YX$

$Y \rightarrow cY \mid L$

16.



17. Using the transition function $\delta(q, \alpha)$ to represent the destination reached from q on input α, where q_0 is the start state, the PDA starts by pushing q_0 onto the STACK. Then control passes to a central READ state which has one branch out for every letter in the alphabet and one for Δ. On each of the paths the next state is a POP. On the alphabetic paths, for whatever state q is popped, the machine pushes the state $\delta(q, \alpha)$ onto the STACK and loops back into the central READ. On the Δ path, if the state popped is a final state there is a path to ACCEPT, otherwise not.

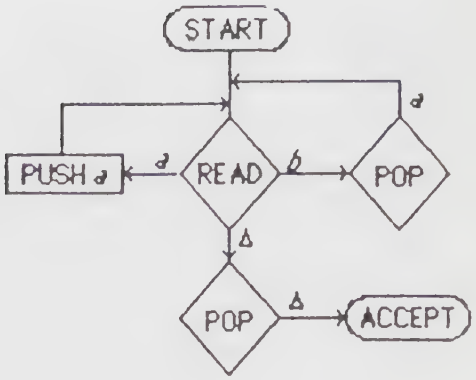
18. Use a machine similar to the one of Problem 17, except that the letters must be read in pairs. (An alphabet of m letters has m^2 possible pairs to account for.) The state $\delta(q, \alpha)$ pushed onto the STACK is a state two FA transitions away from the state that was popped.

19. (i) The PDA for $|L|$ reads an odd-numbered letter and ignores it, then reads an even-numbered letter and processes it just as the machine for L would have done.

    (ii) Yes. The FA for L can be converted into a TG for $|L|$ be relabeling the transitions with pairs of letters. A transition on a becomes a transition on aa or ba, etc.

20. (i) ***aaaabbbb, aaababbb, aabaabbb, abaaabbb, aaabbabb, aabababb, abaababb, ababaabb, aaabbbab, aababbab, abaababb, aabbaabb, ababaabb, aaabbbab, aababbab, abaabbab, aabbabab, ababbaab***

(ii)



(iii) Use a counterexample of the form ($a^n b^n$) and the strong form of the Pumping Lemma.

(iv) $S \rightarrow SS \mid aSb \mid ab$

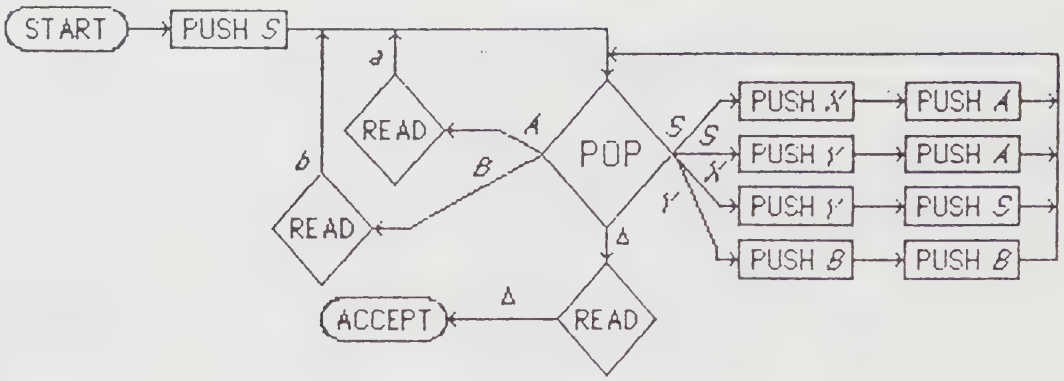
(v) They are matched pairs (equal numbers of a's and b's), where the nesting requires that each close parenthesis (b) be prepared by an open parenthesis (a) that has preceded it.

Chapter 15

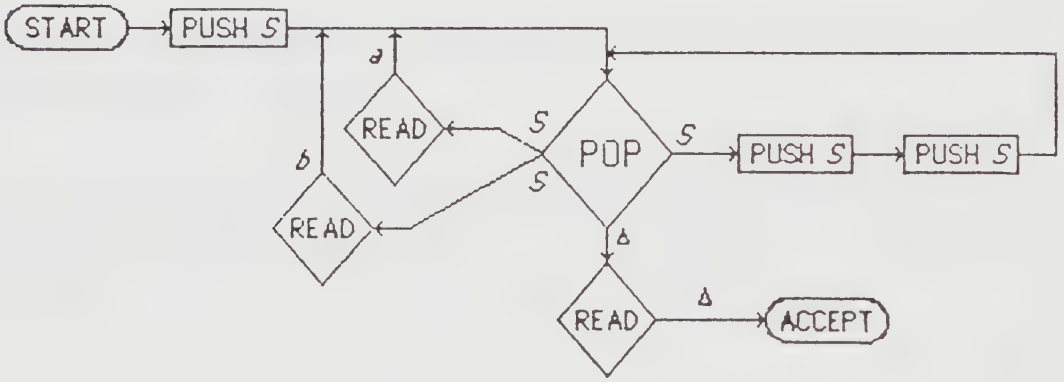
This theorem has two halves. I always cover the first and skip the second. This is *ipso facto* inexcusable but I plead the necessity of time constraints. I do think the proof in the text is understandable and self-contained and can be assigned to students to read as long as they are promised it will not appear on the final exam. I have not made the usual big deal that the STACK must always return to a previous length condition and that that is why the Net nonterminals will always be sufficient. This is partially because it is as obvious as many other points that are slurred over expressly and partly because our approach seems to include this as a necessary consequence -- also the STACK-content graph-pictures appearing in other texts always seem to imply that the STACK contains a continuous rather than discrete number of elements, which negates the mathematical precision of the whole exercise.

Chapter Fifteen

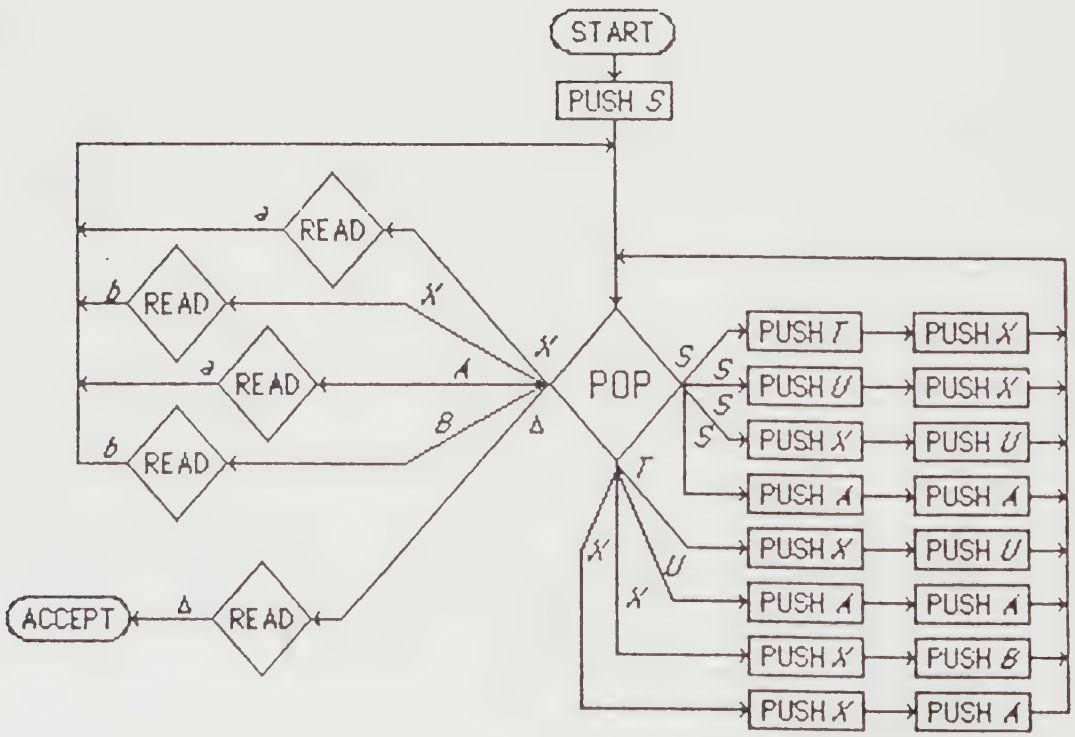
1. (i)



(ii)

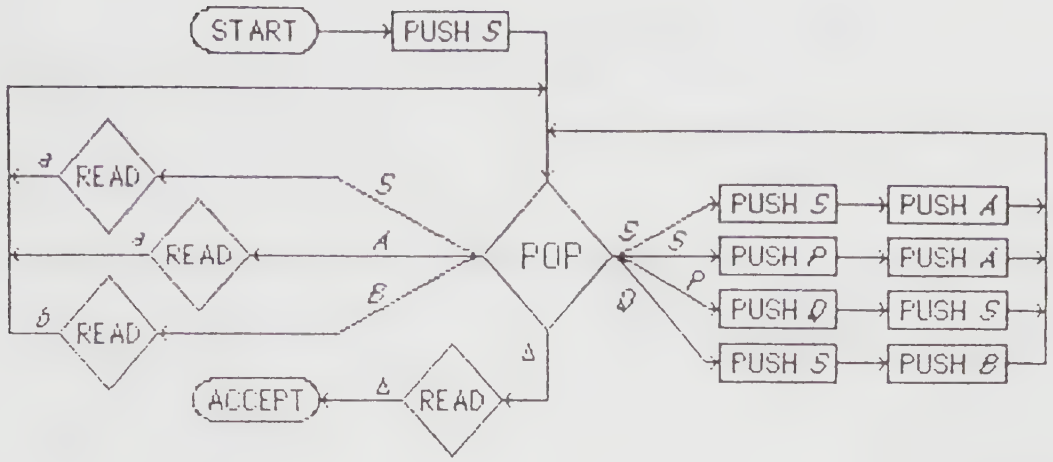


2.

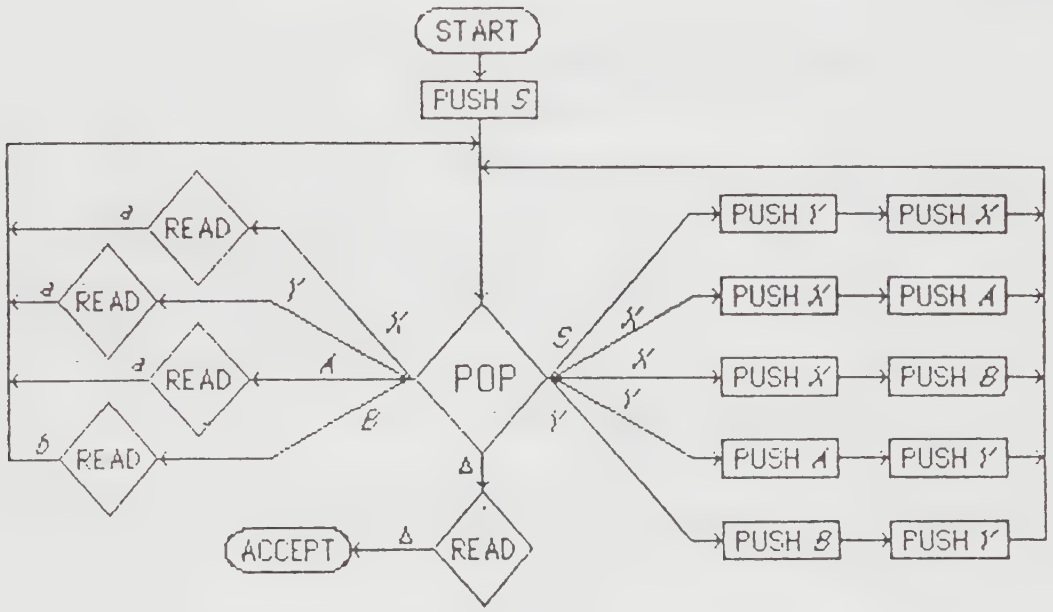


Chapter Fifteen

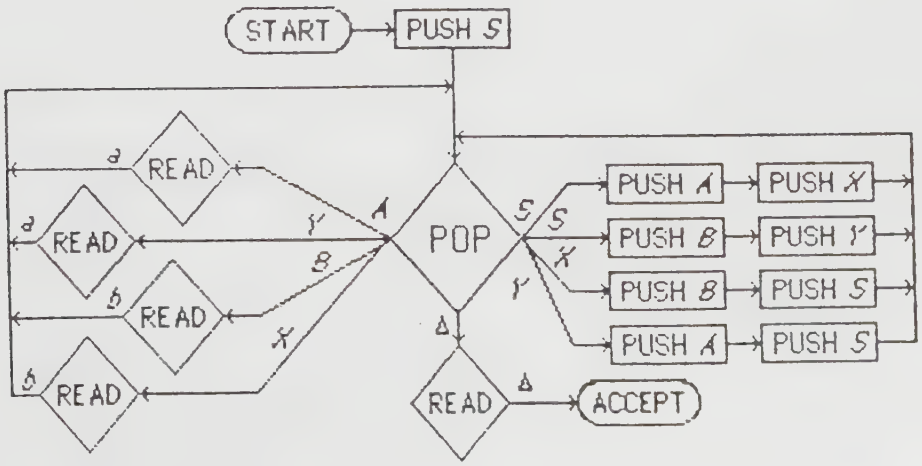
3.



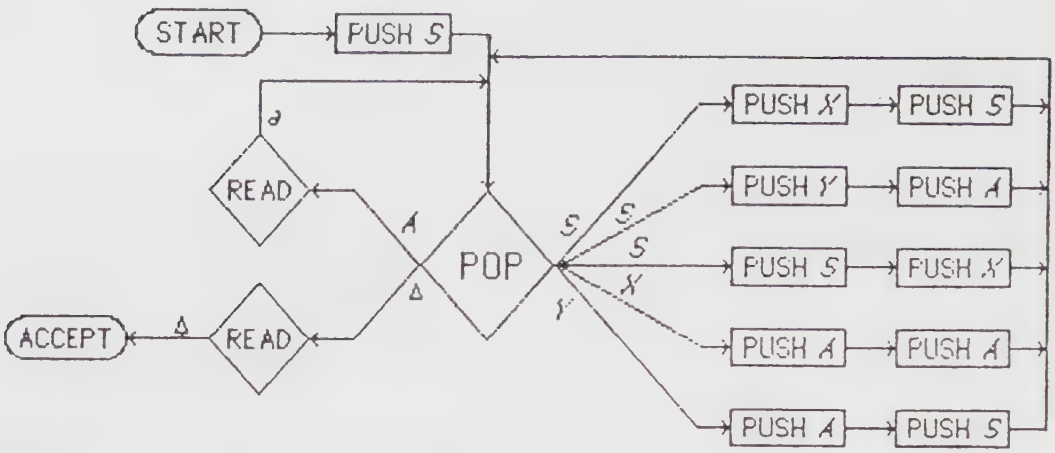
4.



5.

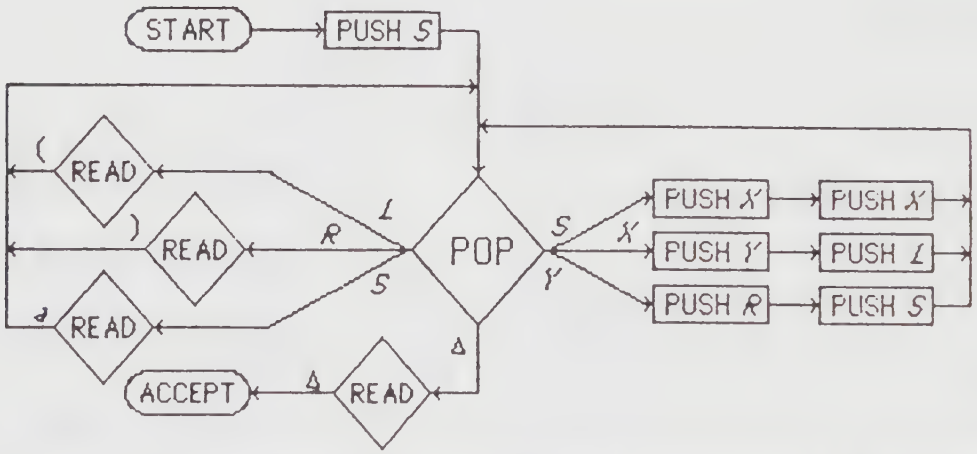


6. (i)



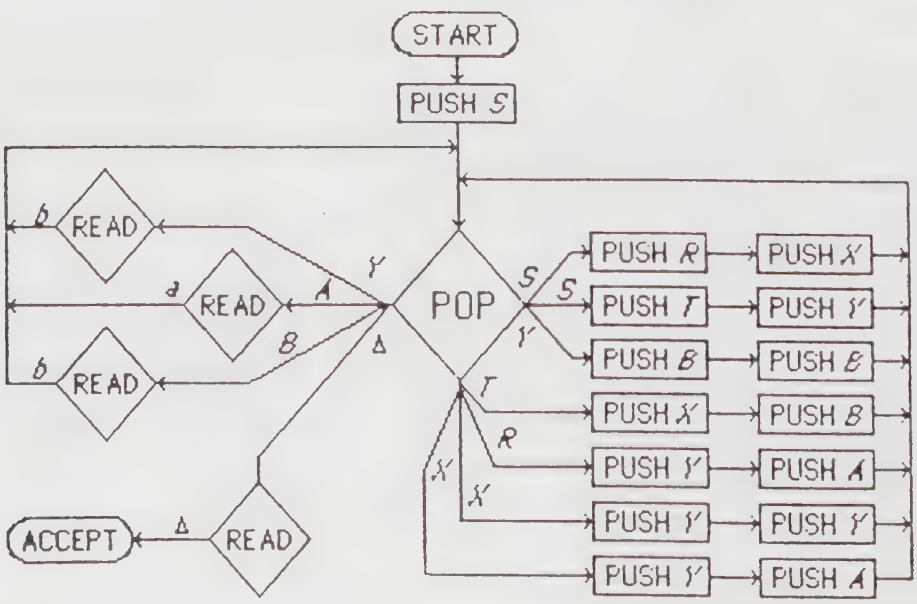
(ii) None

7. (i)

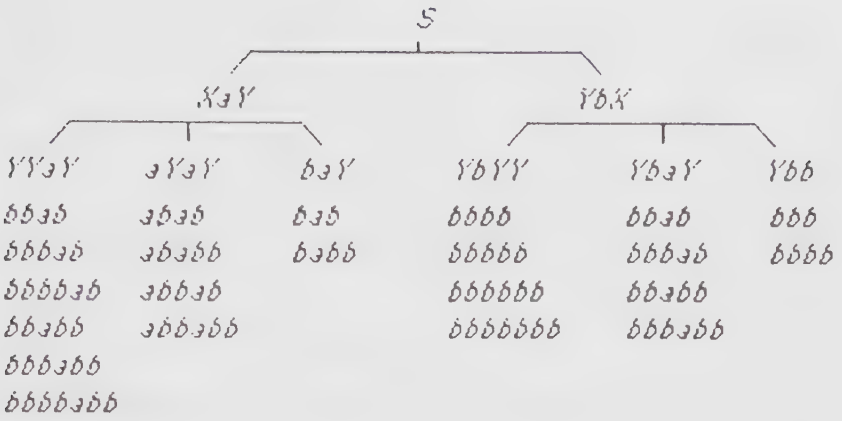


(ii) 5: ((a)(a))((a)(a)), (a)((a)(a))(a)), (a) ((a)((a)(a))), ((a)((a)(a)))(a), (((a)(a))(a))(a)

8. (i)



(ii)



9. The same process can be used to create PDA's for any CFG's. Everything is stacked; nonterminals, when popped, follow procedures as given, terminals are read.

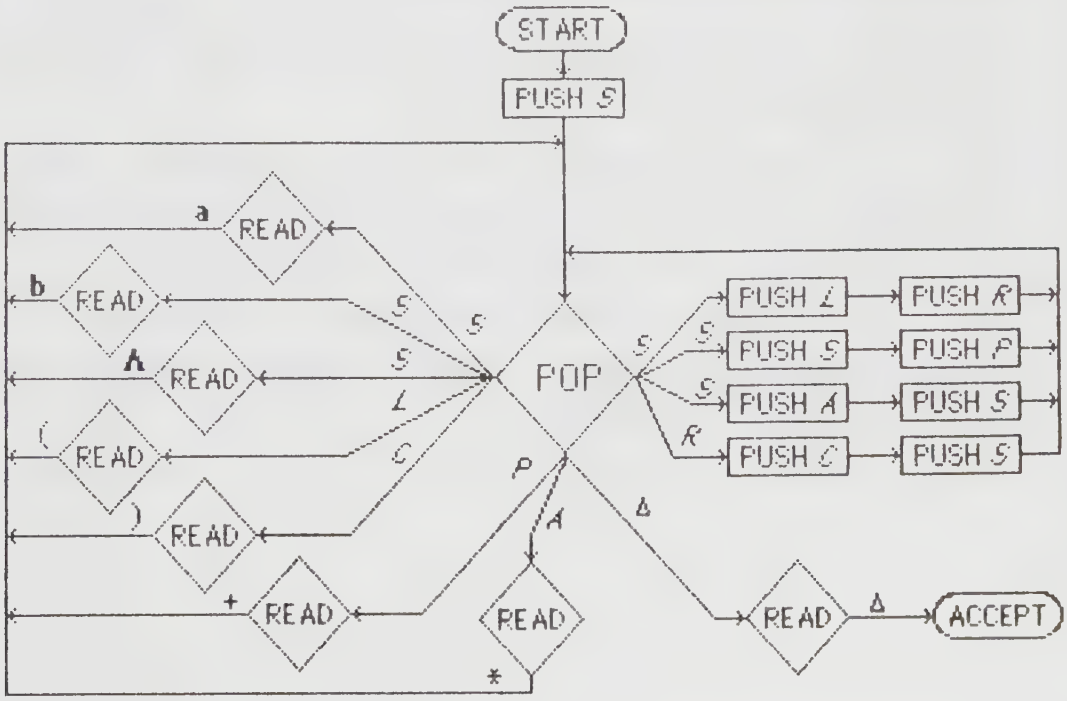
10. (i) Use the strong form of the Pumping Lemma. Consider a machine with n states and the counterexample of $((^{2n}\ \mathbf{a}\ )^{2n})$.

(ii) $S \rightarrow SS \mid (S) \mid S + S \mid S^* \mid \mathbf{a} \mid \mathbf{b} \mid \mathbf{A}$

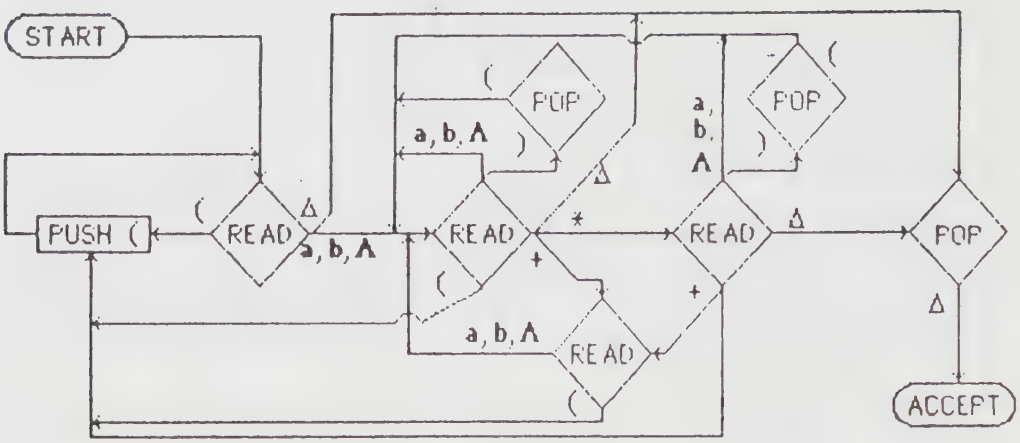
(iii) CNF: $S \rightarrow LR \mid SP \mid SA \mid \mathbf{a} \mid \mathbf{b} \mid \mathbf{A}$

$R \rightarrow SC$

$L \rightarrow ($ $\quad C \rightarrow)$ $\quad P \rightarrow +$ $\quad A \rightarrow *$



(iv)



11. (i) In conversion form every READ must be followed immediately by a POP, and since no branching can occur at POP states the same POP cannot be used to follow two READ's. Therefore it is impossible to have a PDA in conversion form with twice as many READ's as POP's.

(ii)

12. (i), (ii) Yes to both. Note that the question refers to rows in the table, not to paths through the machine. Rows just show alternative paths available between consecutive joint states, without restrictions of consistency.

(iii) No. Now we are talking about paths to ACCEPT. To get there we must POP everything, including the $, from the STACK. We cannot have pushed more letters than we POP.

13 (i) $a(a+b)^*$ the language of all words beginning with a

(ii-iii)

FROM	TO	READ	POP	PUSH	ROW
START	READ		$	$	1
$READ_1$	HERE	a	$	$aab\$$	2

HERE	HERE		a		3
HERE	$READ_2$		b		4
$READ_2$	$READ_2$	a	\$	\$	5
$READ_2$	$READ_2$	b	\$	\$	6
$READ_2$	ACCEPT	Δ	\$		7

14. (i) W, X, Y and Z represent the sets of possible joints. Not all combinations are useful in the production of words.

$PROD_1$ $S \rightarrow$ NET(START, ACCEPT, \$)

$PROD_2$ NET(START, X, \$) $\rightarrow$ Row_1NET($READ_1$, X, \$)

$PROD_3$ NET($READ_1$, X, \$) $\rightarrow$ Row_2NET($READ_1$, Y, a)NET(Y, Z, a)NET(Z, W, b)NET(W, X, \$)

$PROD_4$ NET(HERE, HERE, a) $\rightarrow$ Row_3

$PROD_5$ NET(HERE, $READ_2$, b) $\rightarrow$ Row_4

$PROD_6$ NET($READ_2$, X, \$) $\rightarrow$ Row_5NET($READ_2$, X, \$)

$PROD_7$ NET($READ_2$, X, \$) $\rightarrow$ Row_6NET($READ_2$, X, \$)

$PROD_8$ NET($READ_2$, ACCEPT, \$) $\rightarrow$ Row_7

All derivations begin

$PROD_1$ $S \rightarrow$ NET(START, ACCEPT, \$) $\rightarrow$ **$PROD_2$** NET(START, ACCEPT, \$)

$\rightarrow$ Row_1NET($READ_1$, ACCEPT, \$)

$\rightarrow$ **$PROD_{3,\ 4,\ 5}$** $Row_1Row_2Row_3Row_3Row_4$ NET($READ_2$, ACCEPT, \$)

There is no choice so far. This gives us

$\rightarrow$ **$PROD_8$** $Row_1Row_2Row_3Row_3Row_4\ X\ Row_7$

where

$X \rightarrow Row_5\ X \mid Row_6\ X \mid \Lambda$.

The CFG for the row-language is

$S \rightarrow Row_1Row_2Row_3Row_3Row_4\ X\ Row_7$

$X \rightarrow Row_5\ X \mid Row_6\ X \mid \Lambda$.

(ii) To covert it to the language over $\Sigma = \{a, b\}$ add the productions

$Row_1 \rightarrow \Lambda$ $\quad$ $Row_5 \rightarrow a$

$Row_2 \rightarrow a$ $\quad$ $Row_6 \rightarrow b$

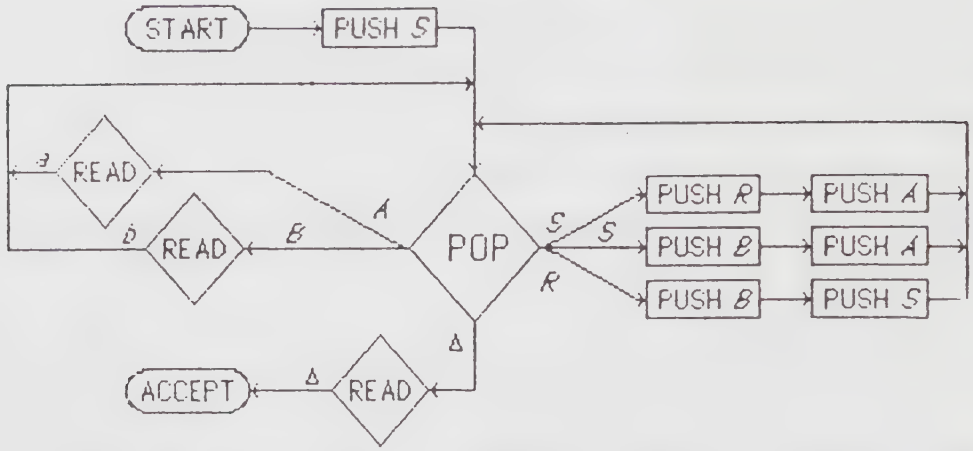
$Row_3 \rightarrow \Lambda$ $\quad$ $Row_7 \rightarrow \Lambda$

$Row_4 \rightarrow \Lambda$

Chapter Fifteen

15. (i) $S \rightarrow AR \mid AB$
$R \rightarrow SB$
$A \rightarrow a$
$B \rightarrow b$

(ii)



16.

FROM	TO	READ	POP	PUSH	ROW
START	HERE		$	S	1
HERE	HERE		S	AR	2
HERE	HERE		S	AB	3
HERE	HERE		R	SB	4
HERE	$READ_1$		A		5
HERE	$READ_2$		B		6
HERE	$READ_3$		$	$	7
$READ_1$	HERE	a	B	B	8
$READ_1$	HERE	a	R	R	9
$READ_2$	HERE	b	B	B	10
$READ_2$	HERE	b	$	$	11
$READ_3$	ACCEPT	Δ	$		12

17. (i) The easy productions are

$PROD_1$ $S \rightarrow$ NET(START, ACCEPT, $)

$PROD_2$ NET(HERE, $READ_1$, a) $\rightarrow$ ROW_5

$PROD_3$ NET(HERE, $READ_2$, b) $\rightarrow$ ROW_6

$PROD_4$ NET($READ_3$, ACCEPT, $) $\rightarrow$ ROW_{12}

To simplify the other productions, we note that in NET(X, Y, Z)

1. If X is READ_3 then Y must be ACCEPT and Z must be \$, since READ_3 goes nowhere else.
2. Y cannot be START.
3. X cannot be ACCEPT.

PROD_5 NET(START, X, \$) → ROW_1NET(HERE, Y, S)NET(Y, X, \$)

PROD_6 NET(HERE, X, S) → ROW_2NET(HERE, Y, A)NET(Y, X, R)

PROD_7 NET(HERE, X, S) → ROW_3NET(HERE, Y, A)NET(Y, X, B)

PROD_8 NET(HERE, X, R) → ROW_4NET(HERE, Y, S)NET(Y, X, B)

PROD_9 NET(HERE, X, \$) → ROW_7NET(READ_3, X, \$)

PROD_{10} NET(READ_1, X, B) → ROW_8NET(HERE, X, B)

PROD_{11} NET(READ_1, X, R) → ROW_9NET(HERE, X, R)

PROD_{12} NET(READ_2, X, B) → ROW_{10}NET(HERE, X, B)

PROD_{13} NET(READ_2, X, \$) → ROW_{11}NET(HERE, X, \$).

This gives us the following:

PROD_1 S → NET(START,ACCEPT,\$)→**PROD_5**ROW_1NET(HERE,READ_2,S)NET(READ_1,ACCEPT, \$)

→ **PROD_{13}** ROW_1NET(HERE,READ_2,S)ROW_{11}NET(HERE, ACCEPT, \$)

→ **PROD_9** ROW_1NET(HERE,READ_2,S)$\text{ROW}_{11}\text{ROW}_7$NET($\text{READ}_3$,ACCEPT, \$)

→ **PROD_4** ROW_1NET(HERE,READ_2,S)$\text{ROW}_{11}\text{ROW}_7\text{ROW}_{12}$

So all the words in the row-language are of this form.

Examining the key portion NET(HERE,READ_2,S) we find that it must use either PROD_6 or PROD_7. If it uses PROD_7 Y must be READ_1 and X must be HERE. since the STACK will have A and then B, and only the edge to READ_1 pops the A and from there only ROW_8 pops the B. This sequence consumes the stack down to whatever is under the S.

NET(HERE,READ_2,S) → **PROD_7** ROW_3NET(HERE,READ_1,A)NET(READ_1,HERE,B)

→ **PROD_2** ROW_3ROW_5NET(READ_1,HERE,B)

→ **PROD_{10}** $\text{ROW}_3\text{ROW}_5\text{ROW}_8$NET(HERE,HERE,$B$).

If we use PROD_6 from here instead we have this sequence:

NET(HERE,READ_2,S) → **PROD_6** ROW_2NET(HERE,READ_1,A)NET(READ_1,HERE,R)

$\rightarrow$ **PROD**$_2$ $\text{ROW}_2\text{ROW}_5\text{NET}(\text{READ}_1, \text{HERE}, R)$

$\rightarrow$ **PROD**$_{11}$ $\text{ROW}_2\text{ROW}_5\text{ROW}_9\text{NET}(\text{HERE}, \text{HERE}, R)$.

$\rightarrow$ **PROD**$_8$ $\text{ROW}_2\text{ROW}_5\text{ROW}_9\text{ROW}_4\text{NET}(\text{HERE}, \text{READ}_2, S)\text{NET}(\text{READ}_2, \text{HERE}, B)$

$\rightarrow$ **PROD**$_{12}$ $\text{ROW}_2\text{ROW}_5\text{ROW}_9\text{ROW}_4\text{NET}(\text{HERE}, \text{READ}_2, S)\text{ROW}_6\text{ROW}_{10}$

We replace $\text{NET}(\text{HERE}, \text{READ}_2, S)$ by the simpler symbol M. The CFG becomes

$S \rightarrow \text{ROW}_1\ M\ \text{ROW}_6\text{ROW}_{11}\text{ROW}_7\text{ROW}_{12}$

$M \rightarrow \text{ROW}_2\text{ROW}_5\text{ROW}_9\text{ROW}_4\ M\ \text{ROW}_6\text{ROW}_{10} \mid \text{ROW}_3\text{ROW}_5\text{ROW}_8$.

(ii) The row-language to input-language conversion is

$\text{ROW}_1 \rightarrow \Lambda$	$\text{ROW}_5 \rightarrow \Lambda$	$\text{ROW}_9 \rightarrow a$
$\text{ROW}_2 \rightarrow \Lambda$	$\text{ROW}_6 \rightarrow \Lambda$	$\text{ROW}_{10} \rightarrow b$
$\text{ROW}_3 \rightarrow \Lambda$	$\text{ROW}_7 \rightarrow \Lambda$	$\text{ROW}_{11} \rightarrow b$
$\text{ROW}_4 \rightarrow \Lambda$	$\text{ROW}_8 \rightarrow a$	$\text{ROW}_{12} \rightarrow \Lambda$

The resulting CFG is $S \rightarrow Mb$

$M \rightarrow aMb \mid a$

which is clearly a grammar for the language $(a^n b^n)$.

18. Consider the "Grand Central POP" machine of Theorem 28. This machine uses exactly one READ state for each "dead" production (Nonterminal $\rightarrow$ terminal). Each of these READ states has only one transition, on the terminal required at that point in the derivation, and returning to the central POP state. Because the transitions out are the same for each READ that looks for a given terminal, we can combine them, so that every production of the form $N \rightarrow \alpha$ $(\alpha \in \Sigma)$ is represented by a transition from POP to a single READ state, and that state has a transition back to POP on α. For $\Sigma = \{a, b\}$, that is two READ's; the third is for Δ.

19. The algorithm presented to construct a PDA from a grammar in CNF requires only on POP state. So any CFL can be implemented by a machine with one POP.

20. In these PDA's each path from the POP goes to either a sequence of two PUSH 's or a READ state before looping back to POP. A deterministic PDA has one such loop per stack letter, in this case, nonterminal. So starting with the single S edge emanating form the POP, at most one path will lead to accept, recognizing only one word. The S edge

either goes directly to a READ in which case the word is only one letter, or replaces S with two nonterminals (which may be the same). For each of these nonterminals there is only one path to follow from POP which have the same possibilities as S just described.

Chapter 16

This chapter is more important for its convincing discussion of why certain languages are inherently not context free than it is for the Pumping Lemma itself. The parallel between self-embeddedness and FA-loops / Kleene closure, is a profound insight and the beginning of mathematical sophistication about this whole subject. Perhaps we give too many pictures illustrating the same point, but the point itself is reiteration and so it is better to iterate too many time than too few. It is actually not all that important in practice to develop facility with demonstrating that languages are not CFLs but it is an easy advanced topic to formulate test questions about and so students should attend carefully to it to become prepared for Master's and PhD qualifying exams.

There are two sloppinesses in this chapter. One is whether it is necessary for a self-embedded nonterminal to reproduce using the very same production X→NN or whether it is sufficient that the second occurrence use a different production X→MM. The other carelessness is counting the rows of the derivation tree -- is the S symbol the zero-th row, is the bottom row of all terminals counted in the tree? Some times $p+1$ should be just p or $p+2$. The point is clear enough and the issue is that there is *some* predictable finiteness about how far one must go to use the Pumping Lemma -- $O(p)$, as complexity analysis would say. Long enough and just so long. If you wish to be more precise about this go to it.

Remember to emphasize that, like the Pumping Lemma for regular languages, this one is a necessary but not sufficient condition for CFLs; and not all non-CFLs can be shown to be so using the Pumping Lemma, even with length.

Chapter Sixteen

1. Only $S \Rightarrow \Lambda$

2. (i) To make aa: Apply $S \rightarrow AA$ and then $A \rightarrow a$ twice.
To make bb: Apply $S \rightarrow BB$ and then $B \rightarrow b$ twice.
To make any longer word in NONNULLEVENPALINDROME:
If the length of the word is $2n$, then for n-1 times,
If the next letter is a apply the productions $S \rightarrow AX$ and $X \rightarrow SA$; if the next letter is b apply the productions $S \rightarrow BY$ and $Y \rightarrow SB$. Then, if the n^{th} letter is a use $S \rightarrow AA$; if b use $S \rightarrow BB$. Then apply the dead productions $A \rightarrow a$ and $B \rightarrow b$.
This shows we can make every word in the language. We can make nothing else.
The only choices from S are $S \rightarrow AA$ and $S \rightarrow BB$, which lead only to dead productions, and $S \rightarrow AX$ and $S \rightarrow BY$. Since the only choice from X is $X \rightarrow SA$, the choice of $S \rightarrow AX$ gives the structure $S \Rightarrow ASA$. Identically, $S \rightarrow BY$ must give the symmetrical derivation $S \Rightarrow BSB$. So the grammar produces only palindromes of even length, and does not produce Λ.

(ii)

(iii) Putting the grammar into the binary form of CNF did not change the result. The real difference comes from introducing the productions $S \rightarrow AA$ and $S \rightarrow BB$, which do not reintroduce the nonterminal S. It is still true that only the shortest words in the language can be derived without using a self-embedded nonterminal.

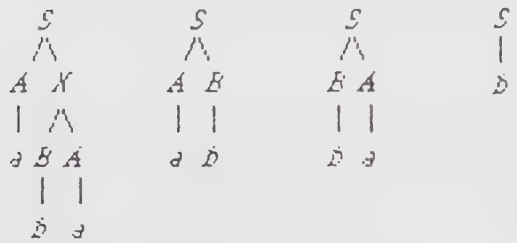
3. The theorem does not give the "wrong" number, it just gives an upper bound that is well above the actual result in this case, 4. If a word of length 4 guarantees self-embedding, a word of 64 must also.

Chapter Sixteen

4. (i) $S \rightarrow AbA \mid Ab \mid bA \mid b$
$A \rightarrow Aa \mid a$

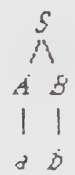
(ii) $S \rightarrow AR \mid AB \mid BA \mid b$
$A \rightarrow AA \mid a$
$R \rightarrow BA$
$B \rightarrow b$

(iii)

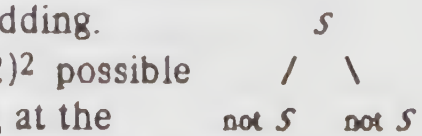


5. (i) $S \rightarrow AX \mid AB$
$X \rightarrow SB$
$A \rightarrow a$
$B \rightarrow b$

(ii)



6. If a grammar in CNF has generated a word of length $2m+1$ there must be $m+1$ levels of live productions in its derivation tree. If the grammar has only m live nonterminals some of these will have to have been repeated by the time the $m+1^{th}$ level is reached. However, repetition does not necessarily guarantee self-embedding. To establish that there is necessarily self-embedding: If there are m different live non-terminals, than at the first level of the tree there are $(m-1)^2$ possible productions that avoid self embedding. Similarly, at the next level there are $(m-2)^2$ possible non-self-embedding productions. Clearly, at the m^{th} level there are $(m-m)^2$ possible non-self-embedding productions; in other words, we necessarily incur a self-



embedded nonterminal if we extend the tree to this level, as we must to generate a word of length 2^{m+1}.

7. The live nonterminals are S, X and Y, so $m = 3$ and $2^{m+1} = 16$. Therefore any word of length 16 or longer must have self-embedding. This is obviously true (see 2(ii)) and provides a closer bound than Theorem 34 provides.

8. The live nonterminals are S, X and Y, so $m = 3$ and $2^{m+1} = 16$. Note that any tree with an X or a Y in it will have a self-embedded S.

9., 10. Prove both by the weak form of the Pumping Lemma. If v and y contain some a's and some b's, uv^2xy^2z will contain too many occurrences of key substrings. If they consist of solid blocks of a's or b's then one or two clumps will be expanded without changing the others, giving a result with nonmatching exponents.

11. (i) 8 words of length 105:

a^{105} $a^{35}b^{35}a^{35}$ $a^{21}b^{21}a^{21}a^{21}b^{21}$ $[a^{15}b^{15}]^3a^{15}$,

$[a^7b^7]^7a^7$ $[a^5b^5]^{10}a^5$ $17[a^3b^3]a^3$ $[ab]^{52}a$

(ii) Use the strong form of the Pumping Lemma.

12. $\{a^nb^{2n}a^n\}$ is not context free. Use the weak form of the Pumping Lemma and show unequal clumps and/or forbidden substrings.

13. $\{a^nb^nc^m\}$ is context free:

$S \rightarrow XY$

$X \rightarrow aXb \mid ab$

$Y \rightarrow cY \mid c$

14. $\{a^nb^nc^nc^m\}$ is not context free, by the weak form of the Pumping Lemma.

15. Since PALINDROME is context-free the Pumping Lemma must apply to it. Take v and y symmetrically around the central letter (or two letters).

16. Use the strong form of the Pumping Lemma and a long word of the form $a^n b^n c^n$ where $n > 2^p$.

17. Use the weak form of the Pumping Lemma. If v and y contain parts of s only, $uvvxyyz$ changes the s part but not the reverse(s) part, so they no longer correspond, and similarly if they contain substrings from the middle (reversed) section only. If they contain substrings from both parts, since UPDOWNUP does not have the bilateral symmetry of PALINDROME either the inflated left s will not match the inlated right s or the inflated s will not correspond to the central reversed section.

18. 19. 20. The Pumping Lemma for CFL over a one letter alphabet is the same as the Pumping Lemma for regular languages over one letter.

$w = uvxyz = a^p a^q a^r a^s a^t$

$uv^n xy^n z = a^p a^{nq} a^r a^{ns} a^t = a^{p+r+t}\ a^{n(q+s)}$

While $w = xyz = a^f a^g a^h$

$xy^n z = a^f a^{ng} a^h = a^{f+h}\ a^{n(g)}$

Both are of the form $a^{\text{constant}}\ a^{n(\text{constant})}$.

If a language fails the Pumping Lemma for regular languages then it fails the Pumping Lemma for CFL

Chapter 17

One of the high points of this chapter is definitely that the machine proof for the closure of CFLs under product is bogus. This is yet another mathematical caveat. More is learned from wrong answers than from right answers, especially if they are someone else's. The cautionary tales of these "almost" proofs are as important in learning what PROOF is and what PROOF isn't as are numerous examples of correct proofs alone.

In the proof involving the intersection of a PDA and an FA we state that the PUSH states need no FA labels because no branching occurs at such states. Profs. A. Barnard, L. Leenen and T.A. Meyer have pointed out to me (and their students) that a better explanation is that FAs only change state when reading input. I feel that this is what I meant, but they state it better.

Chapter Seventeen

1. (i) $S \rightarrow aX$
$X \rightarrow aX \mid bX \mid \Lambda$

(ii) $S \rightarrow aB \mid bA$
$A \rightarrow aS \mid bAA \mid a$
$B \rightarrow bS \mid aBB \mid b$

(iii) $S \rightarrow SS \mid BS \mid USU \mid \Lambda$
$B \rightarrow aa \mid bb$
$U \rightarrow ab \mid ba$

(iv) $S \rightarrow XY$
$X \rightarrow aXb \mid ab$
$Y \rightarrow aYb \mid ab$

2. (i) $S \rightarrow XY$
$X \rightarrow aXb \mid ab$
$Y \rightarrow bYa \mid ba$

(ii) $S \rightarrow XY$
$X \rightarrow aXbb \mid abb$
$Y \rightarrow bbYa \mid bba$

(iii) $S \rightarrow XY$
$X \rightarrow aXbb \mid abb$
$Y \rightarrow bYa \mid ba$

(iv) $S \rightarrow XYZ$
$X \rightarrow aXb \mid ab$
$Y \rightarrow bYa \mid ba$
$Z \rightarrow aZb \mid ab$
We get all words of the form $a^m b^m a^m b^n$

3. (i) $S \rightarrow S_1 \mid S_2$
$S_1 \rightarrow aS_1b \mid ab$
$S_2 \rightarrow bS_2a \mid ba$

(ii) No. Every word in the closure of this language must be factored into substrings that are words in EQUAL. The string $aabaabb$ is in EQUAL but not in the closure of this language.

(iii) $S \rightarrow SS \mid S_1 \mid S_2 \mid \Lambda$
$S_1 \rightarrow aS_1b \mid ab$
$S_2 \rightarrow bS_2a \mid ba$

(iv) See (ii)

(v) L *ab ba aabb abab abba baab baba bbaa*
aaabbb aabbab aabbba ababab ababba abbaab abbaba abbbaa baaabb baabab baabba babaab bababa babbaa bbaaab bbaaba bbbaaa aaaabbbb aaabbbab aaabbbba aabbaabb aabbabab aabbabba aabbbaab aabbbaba aabbbbaa abaaabbb abaabbab abaabbba ababaabb ababbaab ababbbaa abbaaabb abbaabab abbaabba abbabaab abbababa abbabbaa abbbaaab abbbaaba abbbbaaa baaaabbb baaabbab baaabbba baabaabb baababab baababba baabbaab baabbaba baabbbaa babaaabb babaabab babaabba bababaab babababa babbbaaa bbaaaabb bbaaabab bbaaabba bbaabaab bbaababa bbaabbaa bbbaaaab bbbaaaba bbbbaaaa

4. All finite languages are regular. (This of course includes the one-word languages (Λ) and, over $\Sigma = \{a, b\}$, $\{a\}$ and $\{b\}$, out of which we can construct all regular expressions.) There is a CFG for any one-letter language: Just write each letter in Σ that is a word as the right side of a production from S.
Now we can build up grammars as we do regular expressions: The union, product and Kleene closure of any two regular languages are regular, and, since we have grammars to start with and theorems to account for union, product and closure, by grammars, we can construct a grammar for any language that can be represented by a regular expression.

5. (i) $S \rightarrow aX$
$X \rightarrow bbX \mid \Lambda$

(ii) $S \rightarrow SS \mid aX \mid \Lambda$
$X \rightarrow bbX \mid \Lambda$

(iii) $S \rightarrow Xa$
$X \rightarrow bbX \mid \Lambda$

(iv) $S \to S_1 S \mid \Lambda$
$S_1 \to Xa$
$X \to bbX \mid \Lambda$

(v) $S \to S_1 S_2 S_1 \mid S_1$
$S_1 \to bb$
$S_2 \to aS_2 \mid \Lambda$

6. (See also Chapter 4, Problem 20.)

(i) Any language is regular that can be defined by a regular expression. Replace each a in the regular expression for L by s_a and each b by s_b. Since the strings can be represented by regular expressions, this can be done. The resulting regular expression defines the substitution language, which is therefore regular.

(ii) In the CFG for the language, change each terminal a to the string s_a and each b to the string s_b. Eventually, in the derivation of any word in the original language, the terminal must have come from some production where it occurred on the right side. This occurrence has been changed to the substitute string.

7. (i) $S \to X \mid Y$
$X \to AXb \mid aab$
$A \to aA \mid a$
$Y \to aYB \mid abb$
$B \to bB \mid b$

(ii) $S \to XY$
$X \to aXb \mid ab$
$Y \to bYa \mid ba$

(iii) $S \to XY$
$X \to ZaaZ$
$Z \to aZ \mid bZ \mid \Lambda$
$Y \to aY \mid bY \mid a$

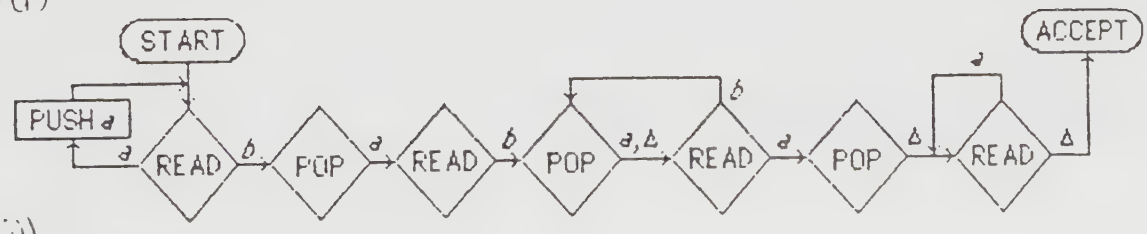
8. (i) In general, there are languages for which the decision procedure can be made before the entire input string has been examined, as in $a(a+b)^*$, and languages about which no decision can be made until the whole input string has been examined, as is the case with $(a+b)^*a$. We do not want to permit nondeterministic branching to the next machine unless the language in question is one of the first type. (Note also that a language may be the union of a language of each of these types.)

(ii) We know that every CFL has a PDA. We can make the product language grammar by the algorithm of Theorem 31 and convert this into a machine by the algorithm of Theorem 28.

(iii) We can use the method of part (ii), making the CFG for L^* and constructing the corresponding PDA à la Theorem 28. However, we can also permit nondeterministic branching as in 18(iv), because any string that can reach ACCEPT for the machine of L is necessarily accepted by the machine for L^*.

9. (i) context-free: ODDPALINDROME's beginning with a
(ii) context-free: either Λ or ϕ
(iii) context-free: $(a^n b^n)$
(iv) context-free
(v) context-free: (NONNULL)PALINDROME
(vi) non-context-free: $(a^n b^{2n} a^n)$
(vii) non-context-free

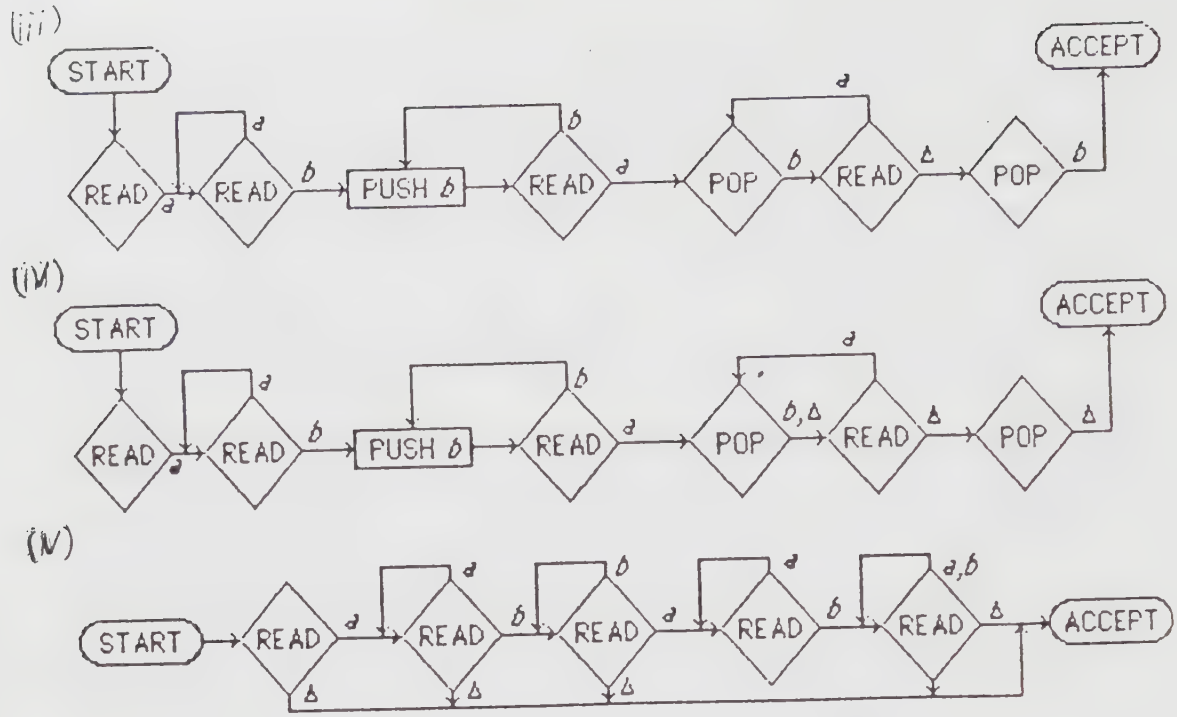
10. (i)



(ii) CFG for $(a^n b^+ a^n)$

$S \rightarrow aSa \mid aXa$

$X \rightarrow bX \mid b$



11. (i) $S \rightarrow aSb \mid aXb$
$X \rightarrow YZ$
$Y \rightarrow bY \mid b$
$Z \rightarrow aZ \mid a$

(ii) $S \rightarrow XY$
$X \rightarrow aXa \mid aYa$
$Y \rightarrow bY \mid b$

(iii) $S \rightarrow XY$
$X \rightarrow aXb \mid ab$
$Y \rightarrow aY \mid Yb \mid a \mid b$

(iv) $L_1 \cap L_2 \cap L_3 = \{a^n b^n a^n b^n\}$, which is not context-free.

12. Suppose VERYEQUAL is context free. Then, by Theorem 40, its intersection with a regular language must be context-free. The intersection of VERYEQUAL with the regular language $\mathbf{a^*b^*c^*}$ is $\{a^n b^n c^n\}$, which we have shown is not context-free. Therefore VERYEQUAL cannot be context-free.

13. (i) $L' = (\mathbf{a}^*\mathbf{b}^*)' + \{a^n b^n\}$. This is the union of two context-free languages and is therefore context-free. If either L or L' is regular the other must be also. In that case, if we intersect either with any regular language the result will be a regular language. Intersecting L' with the regular language $\mathbf{a}^*\mathbf{b}^*$ gives $\{a^n b^n\}$ which is not regular.

(ii) Use the grammars $S \to ASb$ $\quad A \to Aa \mid a$ $\qquad S \to aSB$ $\quad B \to Bb \mid b$

to show that they are context-free. Since each is the transpose of the other, if either were regular the other would have to be (see p. 98). Their intersection is $\{a^n b^n\}$, so they are not regular.

(iii) Their intersecton is $\{a^n b^n\}$.

(iv) Their union is $\mathbf{aa^*bb^*}$

14. (i) $S \to aSa \mid aXa$
$X \to bXa \mid ba$

(ii) $S \to XY$
$X \to aXb \mid ab$
$Y \to aY \mid a$

(iii) Their intersection is $\{a^n b^n a^{2n}\}$, which can be shown not to be context-free by the weak form of the Pumping Lemma.

15. $S \to V \mid W \mid X \mid Y \mid Z$ $\qquad W \to ARA$
$V \to V_1 \mid V_2 \mid V_3 \mid V_4$ $\qquad X \to RBA$
$V_1 \to V_1 a \mid V_1 b \mid b$ $\qquad Y \to ABT$
$V_2 \to aV_2 \mid \Lambda$ $\qquad Z \to ATA$
$V_3 \to A \mid B \mid AB$ $\qquad R \to aRb \mid ab$
$V_4 \to ABAV_1$ $\qquad T \to bTa \mid ba$
$A \to aA \mid a$ $\qquad B \to bB \mid b$

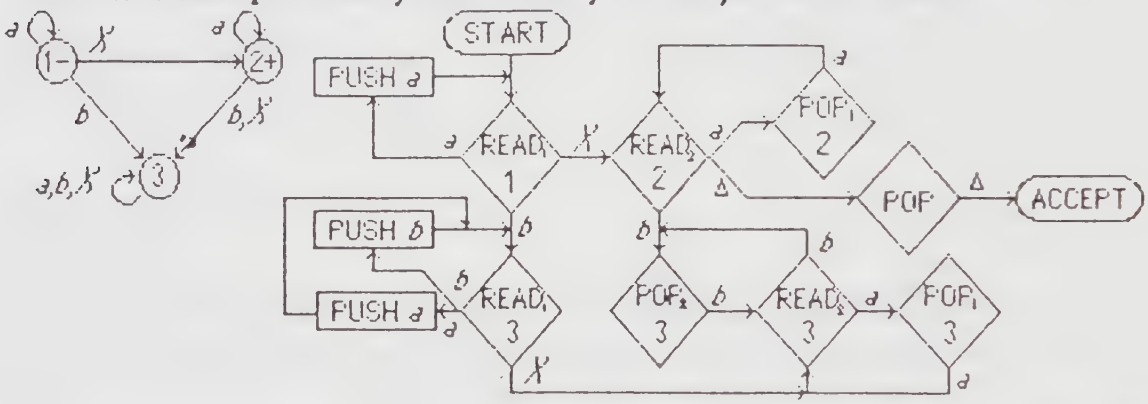
16. Assume L as defined in (i). Again, $L - R = L \cap R'$. Since R is regular, so is R', and the intersection of a context-free language with a regular language is context-free.

Chapter Seventeen

17. (i) The algorithm works fundamentally by de Morgan's law (see Chapter 10, p. 183), that is, we complement the union of the complements. If a PDA is nondeterministic then some word that is accepted by one path will be rejected on another path. When the ACCEPT/REJECT status of the states is changed, such a word will still be accepted, by one of the paths that did not work on the original machine. This shows that the procedure is bad, since a word cannot be in a language and its complement both.

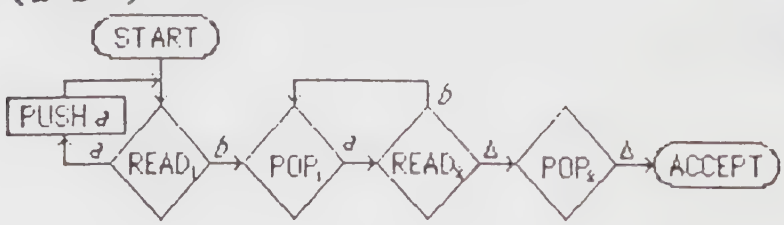
(ii) Yes. It may result in a nondeterministic PDA, however.

18. (i) The PDA is on page 350. The FA and the intersection machine are below. Note that the alphabet is $\Sigma = \{a, b, X\}$. The FA accounts for all transitions. The PDA accounts for strings that are not accepted only when they are rejected on the FA.



(ii) First note that any b causes the string to be rejected by taking it to a state that includes FA state 3. These states have no Δ- edges. The a's read initially are stacked. The X causes a transition to the next state of both machines. Thereafter, any X read (or popped) causes rejection. For every a read, the stack is popped. If TAPE and STACK are both empty at the same time, the string is accepted. As many a's must be read after the X as were stacked before.

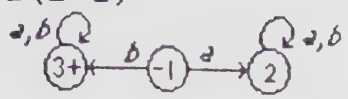
19. $(a^n b^n)$



a(a+b)*

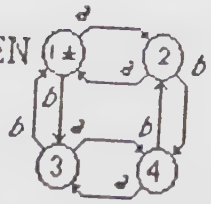
Chapter Seventeen

b(a+b)*

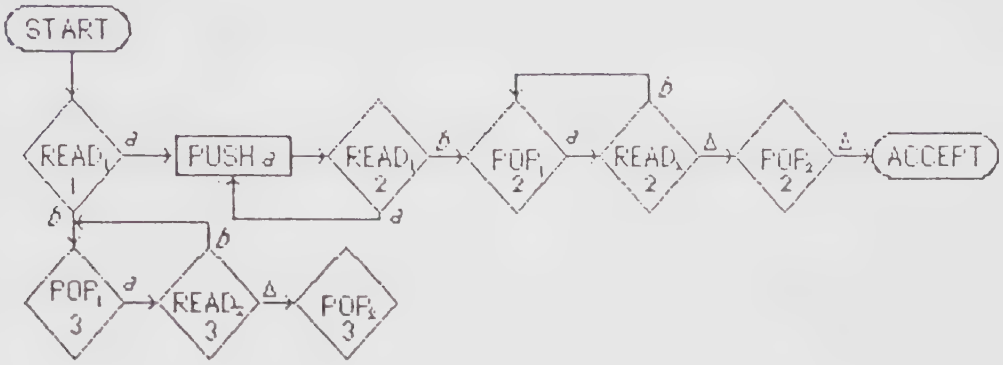


(a+b)*aa(a+b)*

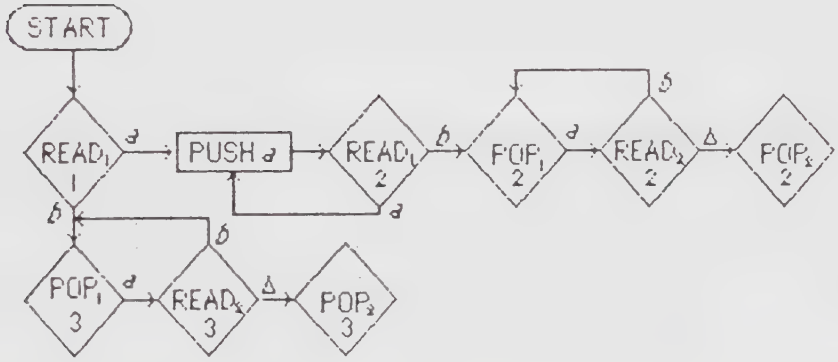
EVEN-EVEN



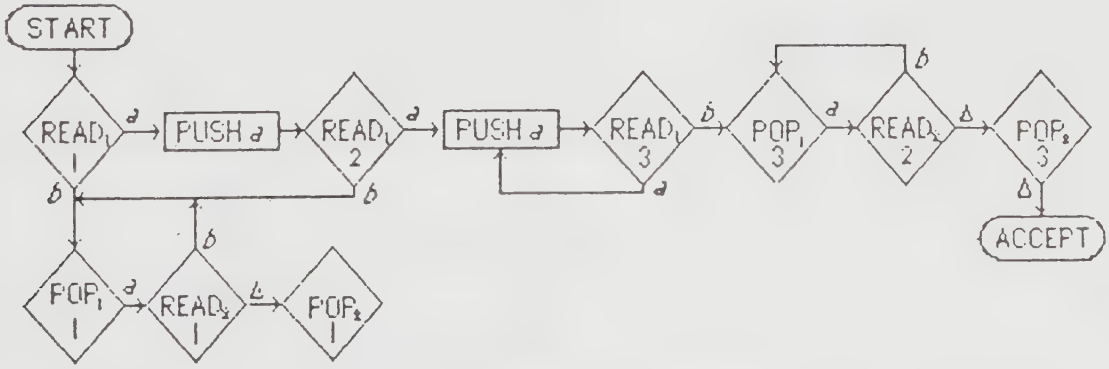
(i) $\{a^n b^n\}$



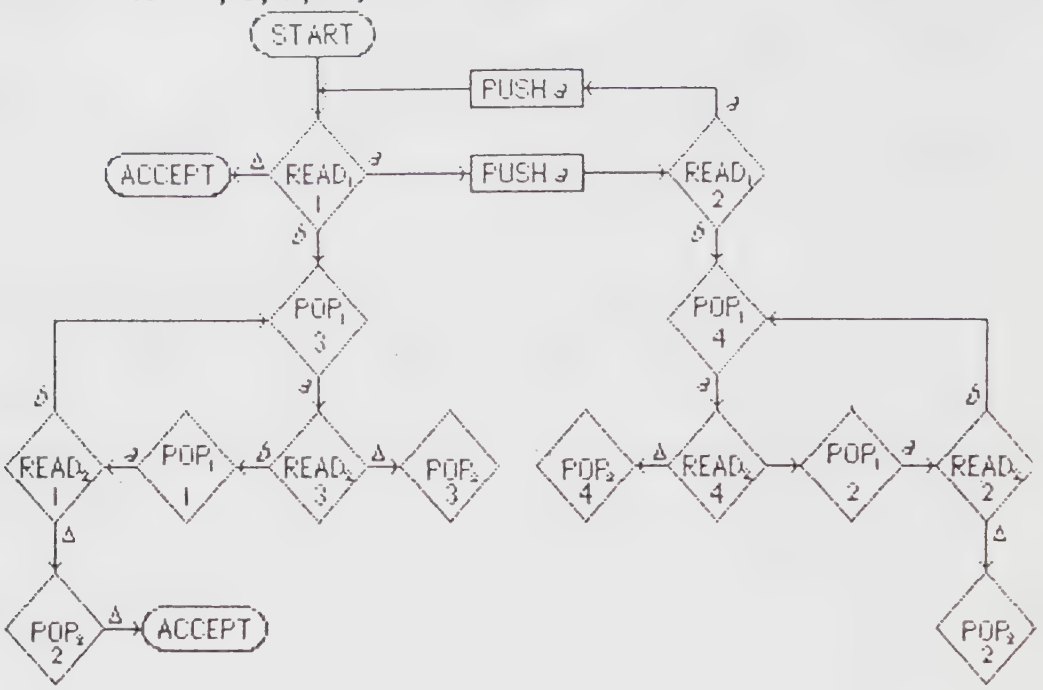
(ii) ϕ



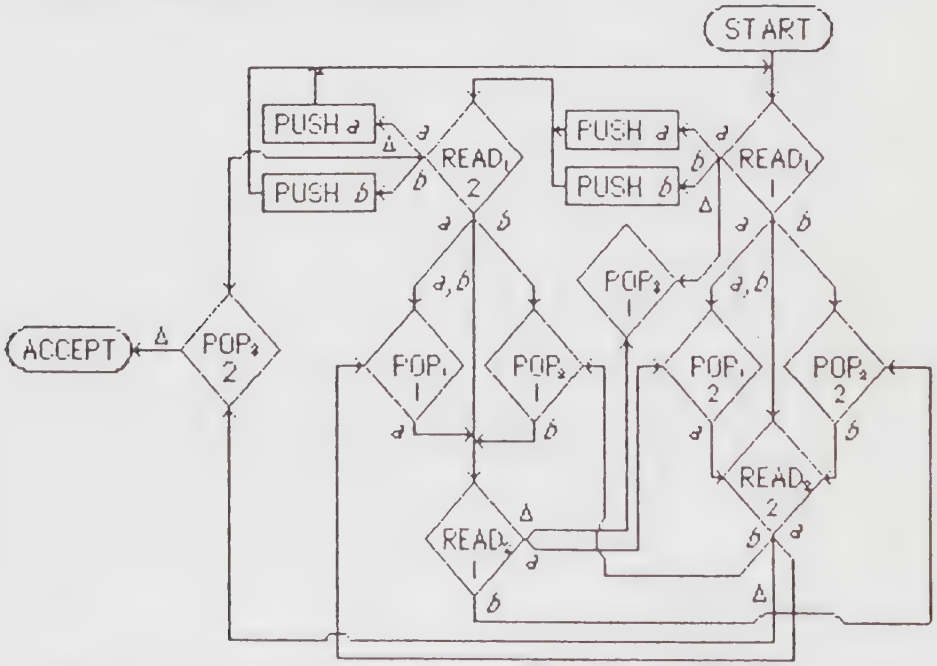
(iii) $\{a^n b^n \quad n \geq 2\}$



(iv) $\{a^{2n}b^{2n} \quad n = 0, 1, 2, \ldots\}$



20. Here is the intersection machine.



In the READ and 1 states reading Δ causes rejection. In the READ and 2 states, where we have read an odd number of letters, reading Δ takes us to ACCEPT.

Chapter 18

The numerous references to what is undecidable can only be verified by something like Rice's Theorem which cannot be given until after TMs are studied, but by PART III we have about lost interest in CFLs and their petty questions. What is undecidable about TMs becomes so much more interesting. Perhaps in another edition Rice's Theorem and the proofs of the undecidability of the questions in this section will be formally presented. At the moment students must be satisfied with a feeble "we'll see soon" type of excuse.

The CYK algorithm as presented in the first edition of this text turns out to be not the CYK algorithm at all, but my own invention. Oh my. It is a perfectly effective algorithm for deciding whether w can be generated from G but it is not the creation of John Cocke. I made his acquaintance in the summer of 1979 at IBM Research at Yorktown Heights, NY and over one lunch I jumped too quickly to an interpretation of what he said. The algorithm made perfect sense to me and I published it as his. This is a disservice to C and Y and K since their version is better. They have my apologies, as do the students who have read the first edition. But since the theorem was decidability the proof was not wrong, merely the attribution. And as is stated repeatedly throughout the text, it is not how best to do it that we are concerned with but whether it can be done at all. Still it behooved me to correct this mistake, and we do here present the CYK in all its glory.

The PDA-transducer for parsing simple arithmetic is a valuable link to courses on compiler construction, and for those students not taking such a course a satisfactory convincing demonstration of how compiling can be done. It is quick reading and needn't take up much class time. On the other hand, it is interesting to discuss what it means for a nondeterministic machine to be a transducer. What if the same input string can produce many different outputs, is this a problem or an opportunity?

Chapter Eighteen

1. (i) no words
   (ii) yes; *ab*
   (iii) yes; *baba*
   (iv) no words
   (v) yes; *abbbb*

2. The algorithm actually does not really use the fact that the grammar is in CNF at all. Allowing t to be a terminal or string of terminals covers grammars in any form.

3. (i) X and Z are useless; finite
   (ii) Y and Z are useless; infinite
   (iii) X and Y are useless; finite
   (iv) infinite
   (v) A, B and C are useless; finite
   (vi) A, B and C are useless; infinite
   (vii) X is useless; infinite
   (viii) infinite

4. Nothing in the algorithm requires the grammar to be in CNF.

5. If the only nonterminal is S then a live production means that S is self-embedded and can generate strings with any number of occurances of S. Moreover a dead production means that all those S can be turned into substrings of terminals. Hence the infinitely many strings generated of any length are all words.

6. Yes

a	b	b	a	ab	bb	ba	abb	bba	abba
S	-	-	S	-	S	-	S	S	S

7. No

b	a	a	b	ba	aa	ab	baa	aab	baab
S	X	X	S	-	X	S	-	S	-

8. No

a	b	b	a	a	ab	bb	ba	aa	abb	bba	baa	abba	bbaa	abbaa
									XX	XX	-	-	XY	XS
X	-	-	X	X	-	X	-	Y	-	-	-	-	S	-

Chapter Eighteen

9. Yes

b	b	a	a	b	bb	ba	aa	ab	bba	baa	aab	bbaa	baab	bbaab
					BB	BA	AA	AB	AA	-	AB	-	BB,BS	BA ,AB
B	B	A	A	B	A	-	-	S,B	-	-	S,B	-	A	S,B

10. Yes

a	b	ba	ab	bab	aba	baba	abab	babab	ababa	bababa	ababab	bababab
		SS, SA, DS, DA	SS, SD, AS, AD	BS, BD, CS, CD	SB, SC, AB, AC	SS,SA, DS,BB, BC,CB CC	SS,AS SD	BS, BD, CS, CD	SB, SC, AB, AC	SS,DS SA,BB BC,CC	SS,AS SD	CS,CD, BS,BD
S,A	S,D	B,C	-	S	S	B,C	-	S	S	C,B	-	S

11. The algorithm processes substring ranging in length between 1 and n. There is no difference when comparing the substrings of length 1 to the right side of the productions. However for length 2, not only must all possible NN combinations be tried but also the possibilities of the forms tt, tN and Nt. Likewise for substrings of greater length, all possible combinations, considering terminals and appropriate nonterminals must be tested. The advantage of CNF is great in this case because regardless of the length of the substring, only the concatenation of two nonterminals must be considered.

12. (i) No

S	X	Y	A
	a b	*a*	*a*
aa bb	*aa ba*	*aa*	
baa	*baa*		
aaaa baaa	*baaa*		
baaaa			

(ii) Yes

S	X	Y	A	B
a b			*a*	*b*
	aa bb	*ab bb*		
aaa aba bab bbb				
	aaaa abaa baba bbba	*aaab abab babb bbbb*		
aaaaa aabaa ababa abbba baaab babab bbabb bbbbb				

(iii) No

S	X	Y
	a	
	bb	*aa*
aaa		
bbaa		

13 (i) $S \Rightarrow E \Rightarrow T+E \Rightarrow F+E \Rightarrow i+E \Rightarrow i+T+E \Rightarrow i+F+E \Rightarrow i+i+E$
$\Rightarrow i+i+T \Rightarrow i+i+F \Rightarrow i+i+i$

(ii) $S \Rightarrow E \Rightarrow E+E \Rightarrow T+E \Rightarrow T*F+E \Rightarrow F*F+E \Rightarrow i*F+E$
$\Rightarrow i*i+E \Rightarrow i*i+T \Rightarrow i*i+T*F \Rightarrow i*i+F*F \Rightarrow i*i+i*F$
$\Rightarrow i*i+i*i$

(iii) $S \Rightarrow E \Rightarrow T \Rightarrow T*F \Rightarrow T*F*F \Rightarrow F*F*F \Rightarrow i*F*F$
$\Rightarrow i*(E)*F \Rightarrow i*(E+E)*F \Rightarrow i*(T+E)*F$
$\Rightarrow i*(F+E)*F \Rightarrow i*(i+E)*F \Rightarrow i*(i+T)\ t\ F$
$\Rightarrow i*(i+F)*F \Rightarrow i*(i+i)*F \Rightarrow i*(i+i)*i$

(iv) $S \Rightarrow E \Rightarrow E+T \Rightarrow T+T \Rightarrow F+T \Rightarrow (E)+T \Rightarrow (T)+T \Rightarrow (T*F)+T$
$\Rightarrow (F*F) \Rightarrow ((E)*F)+T \Rightarrow ((E)*F)+T \Rightarrow ((T)*F)+T$
$\Rightarrow ((F)*F)+T \Rightarrow ((i)*F)+T \Rightarrow ((i)*(E))+T \Rightarrow ((i)*(T+E))+T$

$\Rightarrow ((i)*(F+E))+T \Rightarrow ((i)*(i+E))+T \Rightarrow ((i)*(i+T))+T$
$\Rightarrow ((i)*(i+F))+T \Rightarrow ((i)*(i+i))+T \Rightarrow ((i)*(i+i))+F$
$\Rightarrow ((i)*(i+i))+i$

(v) $S \Rightarrow E \Rightarrow T \Rightarrow F \Rightarrow (E) \Rightarrow (T+E) \Rightarrow (F+E) \Rightarrow ((E)+E)$
$\Rightarrow ((T)+E) \Rightarrow ((F)+E) \Rightarrow (((E))+E) \Rightarrow (((T))+E)$
$\Rightarrow (((F))+E) \Rightarrow (((i))+E) \Rightarrow (((i))+T) \Rightarrow (((i))+F)$
$\Rightarrow (((i))+(E)) \Rightarrow (((i))+(T)) \Rightarrow (((i))+(F)) \Rightarrow (((i))+((E)))$
$\Rightarrow (((i))+((F))) \Rightarrow (((i))+((i)))$

14 (i) $i*(i)_{from}\, i*(F)_{from}\, i*(T)_{from}\, i*(E)_{from}\, i*F_{from}\, i*T_{from}\, F*T_{from}\, T_{from}\, E_{from}\, S$

(ii) $((i)+((i)))_{from}((F)+((i)))_{from}((T)+((i)))_{from}((E)+((i)))_{from}(F+((i)))$
$_{from}(T+((i)))_{from}(T+((F)))_{from}(T+((T)))_{from}(T+((E)))_{from}(T+(F))$
$_{from}(T+(T))_{from}(T+(E))_{from}(T+F)_{from}(T+T)_{from}(T+E)_{from}(E)_{from}F_{from}$
$T_{from}E_{from}S$

(iii) $(i*i+i)_{from}(F*i+i)_{from}(F*F+i)_{from}(F*T+i)_{from}(T+i)_{from}(T+F)$
$_{from}(T+T)_{from}(T+E)_{from}(E)_{from}F_{from}T_{from}E_{from}S$

(iv) $i*(i+i)_{from}\, F*(i+i)_{from}\, F*(F+i)_{from}\, F*(T+i)_{from}\, F*(T+F)$
$_{from}\, F*(T+T)_{from}\, F*(T+E)_{from}\, F*(E)_{from}\, F*F_{from}\, F*T_{from}\, T_{from}\, E_{from}\, S$

(v) $(i*i)*i_{from}(F*i)*i_{from}(F*F)*i_{from}(F*T)*i_{from}(F*T)*i$
$_{from}(T)*i_{from}(E)*i_{from}F*i_{from}F*F_{from}F*T_{from}T_{from}E_{from}S$

15 (i) $S \Rightarrow E \Rightarrow E-T \Rightarrow T-T \Rightarrow T/F-T \Rightarrow F/F-T \Rightarrow (E)/F-T$
$\Rightarrow (E-T)/F-T \Rightarrow (T-T)/F-T \Rightarrow (F-T)/F-T$
$\Rightarrow ((E)-T)/F-T \Rightarrow ((E+T)-T)/F-T \Rightarrow ((T+T)-T)/F-T$
$\Rightarrow ((F+T)-T)/F-T \Rightarrow ((i+T)-T)/F-T$
$\Rightarrow ((i+i)-T)/F-T \Rightarrow ((i+i)-T*F)/F-T$
$\Rightarrow ((i+i)-i*F)/F-T \Rightarrow ((i+i)-F*F)/F-T$
$\Rightarrow ((i+i)-i*F)/F-T \Rightarrow ((i+i)-i*i)/F-T$
$\Rightarrow ((i+i)-i*i)/i-T \Rightarrow ((i+i)-i*i)/i-F$
$\Rightarrow ((i+i)-i*i)/i-i$

(ii) $S \Rightarrow E \Rightarrow E+T \Rightarrow T+T \Rightarrow T/F+T \Rightarrow F/F+T \Rightarrow i/i+T \Rightarrow i/i+F \Rightarrow i/i+i$

(iii) $S \Rightarrow E \Rightarrow E-T \Rightarrow T-T \Rightarrow T/F-T \Rightarrow T*F/F-T \Rightarrow F*F/F-T$
$\Rightarrow i*F/F-T \Rightarrow i*i/F-T \Rightarrow i*i/i-T \Rightarrow i*i/i-F \Rightarrow i*i/i-i$

(iv) $S \Rightarrow E \Rightarrow T/F \Rightarrow T/F/F \Rightarrow F/F/F \Rightarrow i/F/F \Rightarrow i/i/F \Rightarrow i/i/i$

Left to right: divide the quotient of the two left parts by the

one on the right.

(v) i-i-i from i-i-F from i-i-T from i-F-T from i-T-T from F-T-T from T-T-T from E-T-T from E-T from E from S

16. (i)

STATE	STACK	TAPE	OUTPUT
START	▲	2*(7+2)	
READ	▲	*(7+2)	
PRINT	▲	*(7+2)	2
READ	▲	(7+2)	
POP	▲	(7+2)	
PUSH *	*	(7+2)	
READ	*	7+2)	
PUSH (	(*	7+2)	
READ	(*	+2)	
PRINT	(*	+2)	7
READ	+	2)	
POP	*	2)	
PUSH (	(*	2)	
PUSH +	+(*	2)	
READ	+(*	)	
PRINT	+(*	)	2
READ	+(*	▲	
POP	(*	▲	
PRINT	(*	▲	+
POP	*	▲	
READ	*	▲	
POP	▲	▲	
PRINT	▲	▲	*
ACCEPT			272+*
START	▲	272+*	
READ	▲	72+*	
PUSH 2	2	72+*	
READ	2	2+*	
PUSH 7	7 2	2+*	
READ	7 2	+*	
PUSH 2	2 7 2	+*	
READ	2 7 2	*	
ADD	9 2	*	
READ	9 2	▲	
MPY	18	▲	
READ	18	▲	
PRINT	▲	▲	18

(ii)

STATE	STACK	TAPE	OUTPUT
START	▲	3*4+7	
READ	▲	*4+7	
PRINT	▲	*4+7	3
READ	▲	4+7	
POP	▲	4+7	
PUSH *	*	4+7	
READ	*	+7	
PRINT	*	+7	4
READ	*	7	
POP	▲	7	
PRINT	▲	7	*
POP	▲	7	
PUSH +	+	7	
READ	+	▲	
PRINT	+	▲	7
READ	+	▲	
POP	▲	▲	
PRINT	▲	▲	+
POP	▲	▲	
ACCEPT			34*7+
START	▲	34*7+	
READ	▲	4*7+	
PUSH 3	3	4*7+	
READ	3	*7+	
PUSH 4	4 3	*7+	
READ	4 3	7+	
MPY	12	7+	
READ	12	+	
PUSH 7	7 12	+	
READ	7 12	▲	
ADD	19	▲	
READ	19	▲	
PRINT			19

Chapter Eighteen

(iii)

STATE	STACK	TAPE	OUTPUT
START	▲	(3+5)+7*3	
READ	▲	3+5)+7*3	
PUSH (	(	(3+5)+7*3	
READ	(	+5)+7*3	
PRINT	(	+5)+7*3	3
READ	(	5)+7*3	
POP	▲	5)+7*3	
PUSH (	(	5)+7*3	
PUSH +	+(	5)+7*3	
READ	+(	)+7*3	
PRINT	+(	)+7*3	5
READ	+(	+7*3	
POP	(	+7*3	
PRINT	(	+7*3	+
POP	▲	+7*3	
READ	▲	7*3	
POP	▲	7*3	
PUSH +	+	7*3	
READ	+	*3	
PRINT	+	*3	7
READ	+	3	
POP	▲	3	
PUSH +	+	3	
PUSH *	*+	3	
READ	*+	▲	
PRINT	*+	▲	3
READ	*+	▲	
POP	+	▲	
PRINT	+	▲	*
POP	▲	▲	
PRINT	▲	▲	+
POP	▲	▲	
ACCEPT			35+73*+
START	▲	35+73*+	
READ	▲	5+73*+	
PUSH 3	3	5+73*+	
READ	3	+73*+	
PUSH 5	5 3	+73*+	
READ	5 3	73*+	
ADD	8	73*+	
READ	8	3*+	

(iv)

STATE	STACK	TAPE	OUTPUT
START	▲	(3*4+5)*(2+3*4)	
READ	▲	3*4+5)*(2+3*4)	
PUSH (	(	3*4+5)*(2+3*4)	
READ	(	*4+5)*(2+3*4)	
PRINT	(	*4+5)*(2+3*4)	3
READ	(	4+5)*(2+3*4)	
POP	▲	4+5)*(2+3*4)	
PUSH (	(	4+5)*(2+3*4)	
PUSH *	*(	4+5)*(2+3*4)	
READ	*(	+5)*(2+3*4)	
PRINT	*(	+5)*(2+3*4)	4
READ	*(	5)*(2+3*4)	
POP	(	5)*(2+3*4)	
PRINT	(	5)*(2+3*4)	*
POP	▲	5)*(2+3*4)	
PUSH (	(	5)*(2+3*4)	
PUSH +	+(	5)*(2+3*4)	
READ	+(	)*(2+3*4)	
PRINT	+(	)*(2+3*4)	5
READ	+(	*(2+3*4)	
POP	(	*(2+3*4)	
PRINT	(	*(2+3*4)	+
POP	▲	*(2+3*4)	
READ	▲	(2+3*4)	
POP	▲	(2+3*4)	
PUSH *	*	(2+3*4)	
READ	*	2+3*4)	
PUSH (	(*	2+3*4)	
READ	(*	+3*4)	
PRINT	(*	+3*4)	2
READ	(*	3*4)	
POP	*	3*4)	
PUSH (	(*	3*4)	
PUSH +	+(*	3*4)	
READ	+(*	*4)	
PRINT	+(*	*4)	3
READ	+(*	4)	
POP	(*	4)	
PUSH +	+(*	4)	
PUSH *	*+(*	4)	
READ	*+(*	)	

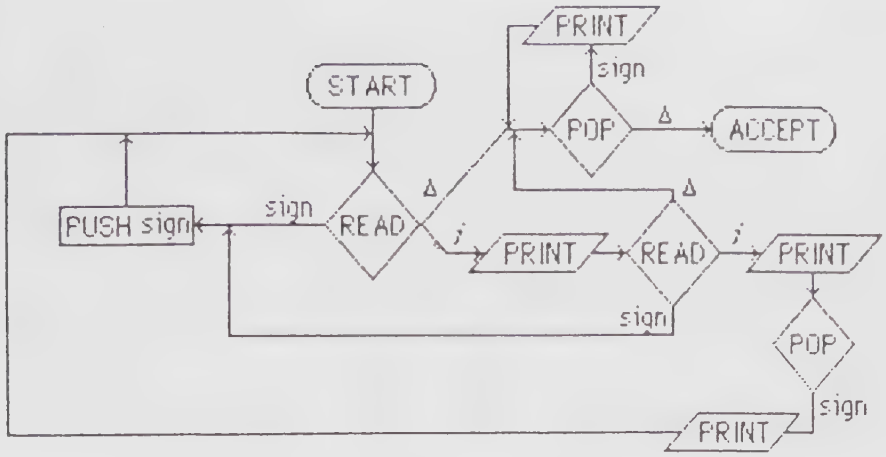
PUSH 7	7 8	3*+	
READ	7 8	*+	
PUSH 3	3 7 8	*+	
READ	3 7 8	+	
MPY	21 8	+	
READ	21 8	▲	
ADD	29	▲	
READ	29	▲	
PRINT	▲	▲	29

(IV cont'd)

START	▲	34*5+234*+*	
READ	▲	4*5+234*+*	
PUSH 3	3	4*5+234*+*	
READ	3	*5+234*+*	
PUSH 4	4 3	*5+234*+*	
READ	4 3	5+234*+*	
MPY	12	5+234*+*	
READ	12	+234*+*	
PUSH 5	5 12	+234*+*	
READ	5 12	234*+*	
ADD	17	234*+*	
READ	17	34*+*	
PUSH 2	2 17	34*+*	
READ	2 17	4*+*	
PRINT	*+(*)		4
READ	*+(* ▲		
POP	+(* ▲		
PRINT	+(* ▲		*
POP	(* ▲		
PRINT	(* ▲		+
POP	* ▲		
READ	* ▲		
POP	▲ ▲		
PRINT	▲ ▲		*
ACCEPT			34*5+234*+*

PUSH 3	3 2 17 4*+*	
READ	3 2 17 *+*	
PUSH 4	4 3 2 17 *+*	
READ	4 3 2 17 +*	
MPY	12 2 17 +*	
READ	12 2 17 *	
ADD	14 17 *	
READ	14 17 ▲	
MPY	238 ▲	
READ	238 ▲	
PRINT		238

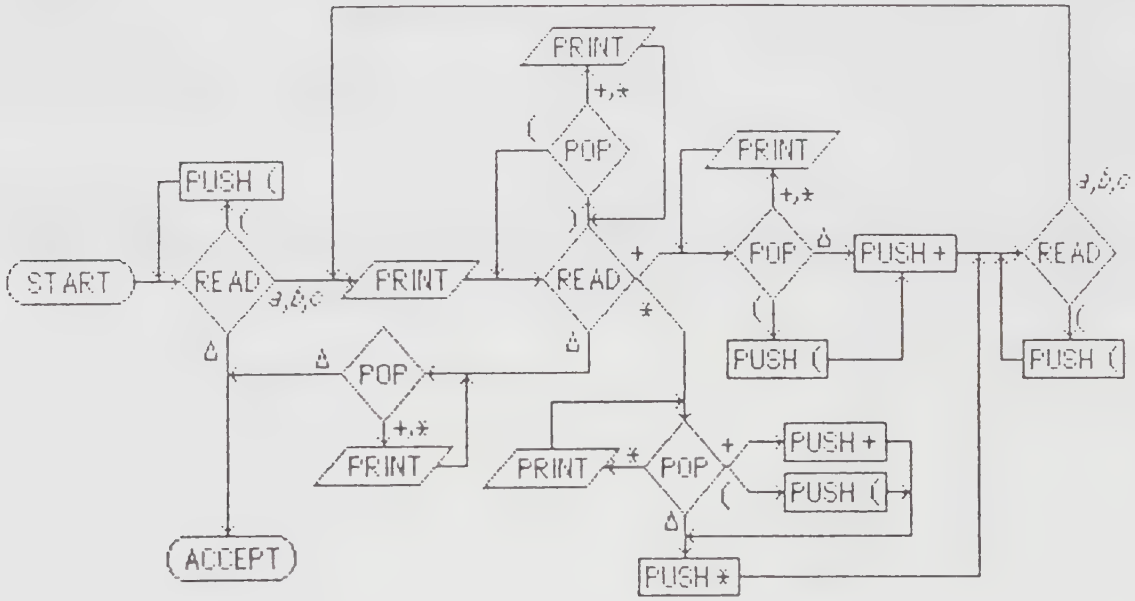
17. The machine on p. 522 can evaluate prefix with one slight alteration and one major intuitive leap: The alteration is the exchange of open and close parentheses in the processing; the intuition is that the infix string must be fed in backwards. The output of this transducer is the transpose of the prefix equivalent of the reverse of the original string.
18. Again, the trick is to feed in the reverse of the prefix string. If this is done, we process just as we did postfix. It should be noted that the reverse of a prefix expression is not its postfix equivalent.
19. While an algorithm is not necessarily restricted to a single STACK, it is worth demonstrating that only one is needed to do this.



20. (i) The evaluator has no ACCEPT state. It evaluates 345+ as 9 leaving 3 on the STACK.

The translator will accept the string +*+ and give output *++.

(ii)



Chapter 19

Turing machine definitions and their associated pictorial representations are fairly standard with the exception that we have chosen the unauthentic version of the one-way infinite TAPE. In general I would have said that this is a bad choice and the two-way TAPE has obvious advantages: (1) we do not have the artificial threat of inadvertently crashing by a left TAPE HEAD move, (2) the association to Post machines, 2PDAs and multi-track TAPE TMs is easier to make, and (3) it is Turing's own choice. The reasons for not using this scheme are: (1) in order to hook onto the students previous intuition developed on the PDA, whose TAPE is one-way infinite because it is a manifestation of the FA invisible TAPE, (2) it does tend to anchor the input as opposed to letting it float in indeterminate space where we are constantly in fear of losing its location, (3) it helps us begin to draw the inference that computers are examples of limited TMs, and (4) it justifies numbering the cells from (i). 4>3 QED.

Yes, there is some unjustified asymmetry between illustrating INSERT over a two letter alphabet while we illustrate DELETE over a three letter alphabet. The purpose is clearly that the earlier example is drawn in a simpler fashion and INSERT is more natural to be the first discussed, especially since we have the perennial "rewind the TAPE HEAD to cell i" problem. The instructor should probably do some blackboard version of the alphabet of n letters if he feels the need for pure mathematical full generality. But I have a feling that any student who has the mental abstraction capability to understand the design for the alphabet $\sum = \{ x_1, x_2, \ldots x_n \}$ can imagine it for herself from what is presented in the text.

There is no such thing as giving too many examples of TMs. Being able to work within the constraints of the TM instruction set is the essence of understanding how a computer works. Unlike FAs or PDAs, programming TMs is real programming.

Chapter Nineteen

1. (i) START $\underline{a}aa \to 3\ \#\ \underline{a}a \to 4\ \#\ a\underline{a} \to 4\ \#\ aa\underline{\Delta} \to 5\ \#\ a\underline{a} \to 6\ \#\ \underline{a}\# \to 7\ \#\ \underline{a}\# \to 1\ \#\ \underline{a}\# \to 3\ \#\#\#\underline{\ } \to$ HALT

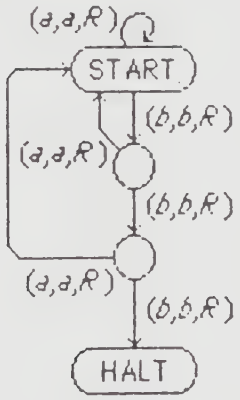
(ii) START $\underline{a}ba \to 3\ \#\ \underline{b}a \to 4\ \#\ b\underline{a} \to 4\ \#\ ba\underline{\Delta} \to 5\ \#\ b\underline{a} \to 6\ \#\ \underline{b}\# \to 7\ \#\ \underline{b}\# \to 1\ \#\ \underline{b}\# \to 2\ \#\#\#\underline{\ } \to$ CRASH

(iii) START $\underline{b}aaba \to 3\ \#\ \underline{a}aba \to 4\ \#\ a\underline{a}ba \to 4\ \#\ aa\underline{b}a \to 4\ \#\ aab\underline{a} \to 4\ \#\ aaba\underline{\Delta} \to 5\ \#\ aab\underline{a} \to 6\ \#\ aa\underline{b}\# \to 7\ \#\ a\underline{a}b\# \to 7\ \#\ \underline{a}ab\# \to 7\ \#\ \underline{\ }aab\# \to 1\ \#\ \underline{a}ab\# \to 3\ \#\#\ \underline{a}b\# \to 4\ \#\#\ a\underline{b}\# \to 4\ \#\#\ ab\underline{\ } \to 5\ \#\#\ a\underline{b}\# \to 6\ \#\#\ \underline{a}\#\# \to 7\ \#\#\ \underline{\ }a\#\# \to 1\ \#\#\ \underline{a}\#\# \to 3\ \#\#\#\#\#\underline{\ } \to$ HALT

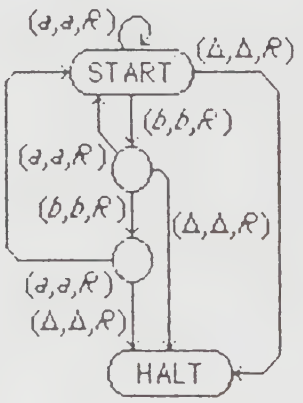
(iv) START $\underline{a}babb \to 3\ \#\ \underline{b}abb \to 4\ \#\ b\underline{a}bb \to 4\ \#\ ba\underline{b}b \to 4\ \#\ bab\underline{b} \to 4\ \#\ babb\underline{\Delta} \to 5\ \#\ bab\underline{b} \to 6\ \#\ ba\underline{b}\# \to 7\ \#\ b\underline{a}b\# \to 7\ \#\ \underline{b}ab\# \to 7\ \#\ \underline{\ }bab\# \to 1\ \#\ \underline{b}ab\# \to 2\ \#\#\ \underline{a}b\# \to 4\ \#\#\ a\underline{b}\# \to 4\ \#\#\ ab\underline{\ } \to 5\ \#\#\ a\underline{b}\# \to 6\ \#\#\ \underline{a}\#\# \to 7\ \#\#\ \underline{\ }a\#\# \to 1\ \#\#\ \underline{a}\#\# \to 3\ \#\#\#\#\#\underline{\ } \to$ HALT

2. The machine changes the two outer letters to # until there are no two outer letters left to change. Then, if the original string had even length (and therefore no middle letter) the string crashes by reading # in 1. If the string had odd length but the middle letter was b it crashes reading # in 2 (or Δ in 2 if the string was just *b*). If the string had an a in the middle that a is turned to # in 1, and the # that starts the second half of the string is read in 3 and goes to HALT.

3. (i)

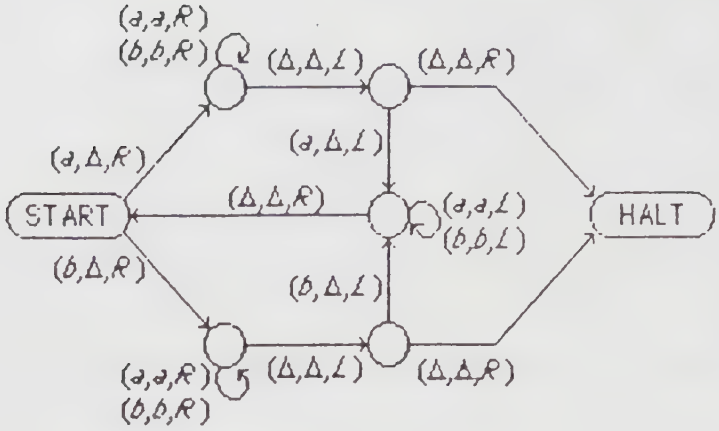


(ii)

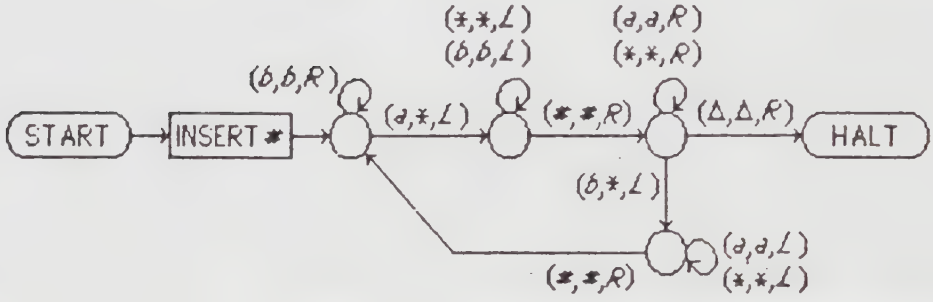


Chapter Nineteen

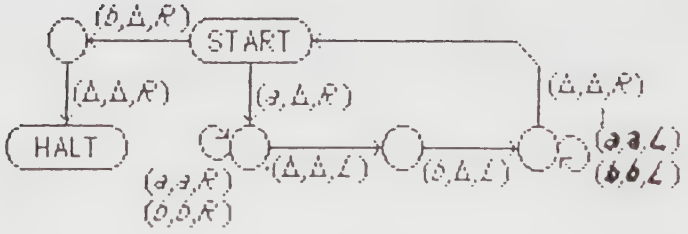
4.



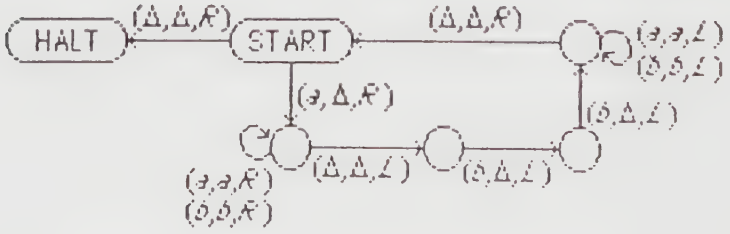
5.



6. (i)



(ii)



7. (i) In states 4 and 7, the last letter has been compared to the first and changed to a Δ if it matches. The transitions at both states are a loop on *a* or *b* and a transition to 1 on Δ. Since the transitions match perfectly the states can be coalesced.

(ii) By using the algorithm of deleting an initial *a* and then deleting a terminal *b*, the machine can be constructed with 5 states including START and HALT. (See 6(ii), above.)

8.(i) $1\ \#\underline{a}aa \to 2\ \#A\underline{a}a \to 3\ \#Aa\underline{a}\Delta \to 4\ \#A\underline{a}a \to 5\ \#\underline{A}Xa \to 7\ \#AXa \to 8\ \#\underline{A}Xa$
$\to 10\ \#\#\underline{X}a \to 12\ \#\#\#a \to 8\ \#\#\#a \to 13\ \#\#\#\Delta \to$ HALT

(ii) $1\ \#\underline{a}aaa \to 2\ \#A\underline{a}aa\ 3\ \#Aa\underline{a}a \to 3\ \#Aaa\underline{a}\Delta \to 4\ \#Aa\underline{a}a \to 5\ \#A\underline{a}Xa \to$
$6\ \#\underline{A}aXa \to 1\ A\underline{a}Xa \to 2\ \#AA\underline{X}a$ CRASH

(iii) $1\ \#\underline{a}aaaa \to 2\ \#A\underline{a}aaa \to 3\ \#Aa\underline{a}aa \to 3\ \#Aaa\underline{a}a \to 3\ \#Aaaa\underline{a}\Delta$
$\to 4\ \#Aaa\underline{a}a \to 5\ \#Aa\underline{a}Xa \to 6\ \#A\underline{a}aXa \to 6\ \#\underline{A}aaXa \to 1\ \#A\underline{a}aXa$
$\to 2\ \#AA\underline{a}Xa \to 3\ \#AAa\underline{X}a \to 4\ \#AA\underline{a}Xa \to 5\ \#A\underline{A}XXa \to 7\ \#\underline{A}AXXa$
$\to 7\ \#AAXXa \to 8\ \#\underline{A}AXXa \to 10\ \#\#\underline{A}XXa \to 10\ \#\#A\underline{X}Xa \to 12\ \#\#\underline{A}XXa$
$\to 12\ \#\#AXXa \to 8\ \#\#\underline{A}XXa \to 10\ \#\#\#\underline{X}Xa \to 10\ \#\#\#X\underline{X}a \to 12\ \#\#\#Xa$
$\to 12\ \#\#\#Xa \to 8\ \#\#\#Xa \to 13\ \#\#\#Xa \to 13\ \#\#\#Xa\Delta \to$ HALT

(iv) $1\ \#\underline{a}abaab \to 2\ \#A\underline{a}baab \to 3\ \#Aa\underline{b}aab \to 3\ \#Aab\underline{a}ab$
$\to 3\ \#Aaba\underline{a}b \to 3\ \#Aabaa\underline{b} \to 3\ \#Aabaab\Delta \to 4\ \#Aabaab$
$\to 5\ \#Aaba\underline{a}Y \to 6\ \#Aab\underline{a}aY \to 6\ \#Aa\underline{b}aaY \to 6\ \#A\underline{a}baaY$
$\to 6\ \#\underline{A}abaaY \to 1\ \#A\underline{a}baaY \to 2\ \#AA\underline{b}aaY$
$\to 3\ \#AAb\underline{a}aY \to 3\ \#AAba\underline{a}Y \to 3\ \#AAbaa\underline{Y} \to 4\ \#AAba\underline{a}Y$
$\to 5\ \#AAb\underline{a}XY \to 6\ \#AA\underline{b}aXY \to 6\ \#A\underline{A}baXY \to 1\ \#AA\underline{b}aXY$
$\to 1\ \#AAB\underline{a}XY \to 2\ \#AAB\underline{a}XY \to 3\ \#AABaX \to 4\ \#AAB\underline{a}XY$
$\to 5\ \#AA\underline{B}XXY \to 7\ \#A\underline{A}BXXY \to 7\ \#\underline{A}ABXX \to 7\ \#AABXXY$
$\to 8\ \#\underline{A}ABXXY \to 8\ \#\#\underline{A}BXXY \to 10\ \#\#A\underline{B}XXY \to 10\ \#\#AB\underline{X}XY$
$\to 12\ \#\#A\underline{B}XXY \to 12\ \#\#\underline{A}BXXY \to 12\ \#\#AB\underline{X}XY \to 8\ \#\#\underline{A}BXXY$
$\to 10\ \#\#\#\underline{B}XXY \to 10\ \#\#\#B\underline{X}XY \to 10\ \#\#\#BX\underline{X}Y \to 12\ \#\#\#B\underline{X}XY$
$\to 12\ \#\#\#\underline{B}XXY \to 12\ \#\#\#BXXY \to 8\ \#\#\#\underline{B}XXY \to 9\ \#\#\#\#XXY$
$\to 9\ \#\#\#\#XXY \to 9\ \#\#\#\#X\underline{X}Y \to 9\ \#\#\#\#XXX \to 11\ \#\#\#XXX$
$\to 11\ \#\#\#XXX \to 8\ \#\#\#XXX \to 13\ \#\#\#XXX \to 13\ \#\#\#XXX$
$\to 13\ \#\#\#XXX\Delta \to 13\ \#\#\#XXX\Delta\Delta \to$ HALT

(v) 1 # $\underline{a}bab$ → 2 # $A\underline{b}ab$ → 3 # $Ab\underline{a}b$ → 3 # $Abab\underline{\Delta}$
→ 4 # $Ab\underline{a}b$ → 5 # $Ab\underline{a}Y$ → 6 # $A\underline{b}aY$ → 6 # $\underline{A}baY$ → 1 # $A\underline{b}aY$
→ 2 # $AB\underline{a}X$ → 3 # $AA\underline{a}Y$ → 4 # $AB\underline{a}Y$ → 5 # $A\underline{B}XY$ → 7 # $\underline{A}BXY$
→ 7 # $ABXY$ → 8 # $\underline{A}BXB$ → 10 # # $\underline{B}XY$ → 10 # # $B\underline{X}Y$ → 12 # # $\underline{B}*Y$
→ 12 # # $\underline{B}*Y$ → 8 # # $\underline{B}*Y$ → 9 # # # $*\underline{Y}$ → 9 # # # $*\underline{Y}$ → 11 # # # $**$
→ 11 # # # $**$ → 8 # # # $**$ → 13 # # # $**$ → 13 # # # $**\underline{\Delta}$ → HALT

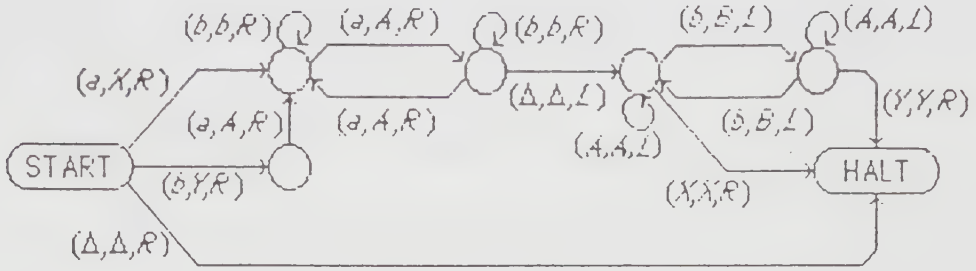
9. (i) The circuit 1-2-3-4-5-6-1 divides the string in half. First-half *a*'s become *A*; *b*'s *B*; second-half uses *X* and *Y* respectively. State 1: Treat the next untreated letter and account for it by putting it into upper case. State 2: Start moving down the TAPE. Crash here if the string is of odd length. State 3: Find the end of the processed string. State 4: This is the last unprocessed letter (found by bouncing off the next cell). Process it . State 5: If there are unprocessed letters, continue the loop. If not, check that the string repeats. State 6: Move leftward until a processed letter is found and bounce off it to the first unprocessed letter remaining. The second half of the machine (states 8-HALT) compares the two halves of the string. State 7 is a rewind state that returns the TAPE HEAD to the beginning of the input. State 8 is the state from which the comparison is made: an *a* sends us to 10 and 12, a *b* to 9 and 11. State 9: The letter in the first half was *b*. Look for the next letter in the second half. Crash if it does not match. State 10: The letter in the first half was *a*. Look for the next letter in the second half. Crash if it does not match. State 11: Get here if the *b* was matched. State 12: Get here if the *a* was matched. Return to process another letter. In State 8, if the string has not crashed, the first half will be #'s and the second half will be all *'s. State 13: Verify that there is nothing left to process before the first blank.

(ii) Words of odd length crash in State 2. Words of even length such that the first half and the second half do not match crash in State 9 if the first-half letter was *b* and in State 10 if it was *a*. These are all the kinds of strings not in DOUBLEWORD.

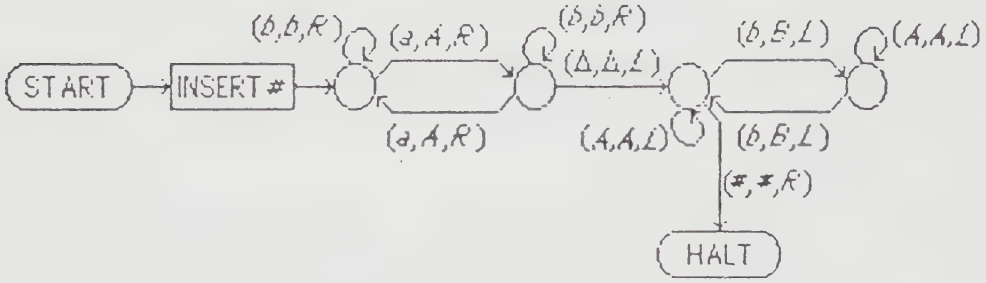
10. (i) Since all out-edges are identical, these states can be combined.

(ii) If State 8 reads * the two halves of the string have been matched. Therefore State 8 can go directly to HALT on (*,*,*R*).

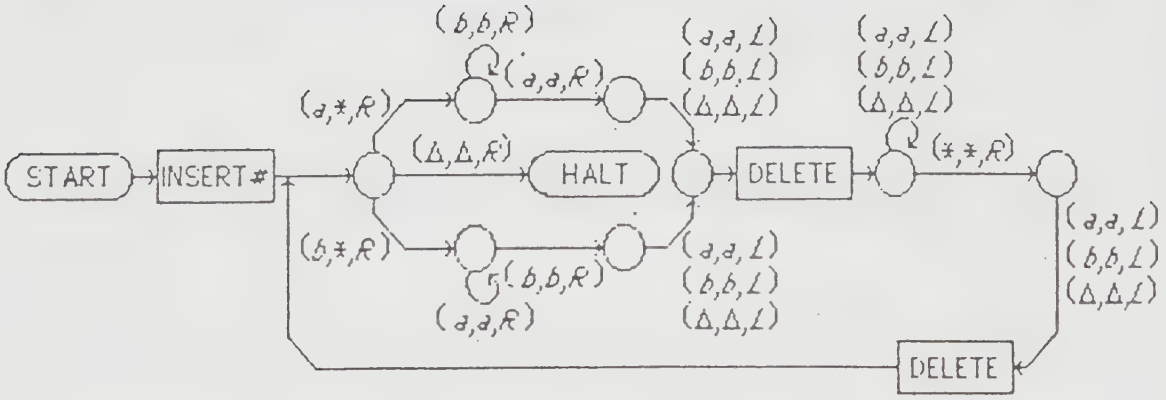
11.



12.



13.



14. The loop at 5 can be (*a*,*a*,*L*); (*b*,*b*,*L*), (Δ, Δ,*L*). We know that there is a # to prevent a crash by moving left from cell i, so there is no reason not to loop on Δ as well as on *a* and *b*.

15.(i) 1 $\underline{a}abbaa \rightarrow 2\ ^{*}\underline{a}bbaa \rightarrow 2\ ^{*}a\underline{b}baa \rightarrow 3\ ^{*}ab\underline{b}aa \rightarrow 3\ ^{*}abb\underline{a}a$
$\rightarrow 4\ ^{*}ab\underline{b}aa \rightarrow 5\ ^{*}aba\underline{a}a \rightarrow 5\ ^{*}abaa\underline{a} \rightarrow 5\ ^{*}abaa\underline{\Delta}$
$\rightarrow 6\ ^{*}aba\underline{a}\Delta \rightarrow 7\ ^{*}ab\underline{a}\Delta\Delta \rightarrow 8\ ^{*}a\underline{b}\Delta\Delta \rightarrow 8\ ^{*}\underline{a}b\Delta \rightarrow 8\ ^{*}\underline{\Delta}ba$
$\rightarrow 8\ ^{*}\underline{\Delta}ba \rightarrow 1\ ^{*}\underline{a}ba \rightarrow 2\ ^{**}\underline{b}a \rightarrow 3\ ^{**}b\underline{a} \rightarrow 4\ ^{**}\underline{b}a \rightarrow 5\ ^{**}a\underline{a}$
$\rightarrow 5\ ^{**}a\underline{\Delta} \rightarrow 6\ ^{**}\underline{a}\Delta \rightarrow 7\ ^{**}\underline{\Delta}\Delta \rightarrow 8\ ^{**}\underline{\Delta}\Delta \rightarrow$ HALT

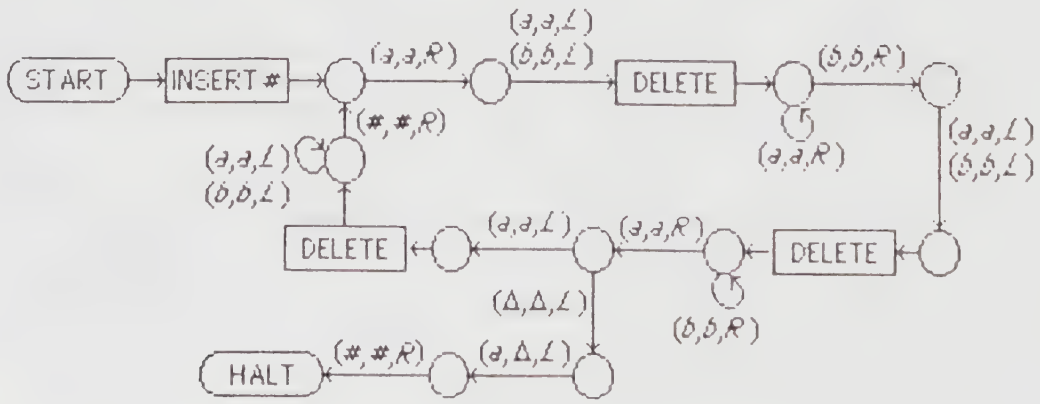
(ii) 1 $\underline{a}abbaaa \rightarrow 2\ ^{*}\underline{a}bbaaa \rightarrow 2\ ^{*}a\underline{b}baaa \rightarrow 3\ ^{*}ab\underline{b}aaa$
$\rightarrow 3\ ^{*}abb\underline{a}aa \rightarrow 4\ ^{*}ab\underline{b}aaa \rightarrow 5\ ^{*}aba\underline{a}aa \rightarrow 5\ ^{*}abaa\underline{a}a$
$\rightarrow 5\ ^{*}abaaa\underline{a} \rightarrow 5\ ^{*}abaaa\underline{\Delta} \rightarrow 6\ ^{*}abaa\underline{a} \rightarrow 7\ ^{*}aba\underline{a}\Delta$
$\rightarrow 8\ ^{*}ab\underline{a}\Delta\Delta \rightarrow 8\ ^{*}a\underline{b}aa \rightarrow 8\ ^{*}\underline{a}baa \rightarrow 8\ ^{*}\underline{\Delta}abaa \rightarrow 8\ ^{*}\underline{\Delta}abaa$
$\rightarrow 1\ ^{*}\underline{a}baa \rightarrow 2\ ^{**}\underline{b}aa \rightarrow 3\ ^{**}b\underline{a}a \rightarrow 4\ ^{**}\underline{b}aa \rightarrow 5\ ^{**}a\underline{a}a \rightarrow 5\ aa\underline{a}$
$\rightarrow 5\ ^{**}aa\underline{\Delta} \rightarrow 6\ ^{**}aa\underline{a} \rightarrow 7\ ^{**}a\underline{\Delta}\Delta \rightarrow 8\ ^{**}\underline{a}aa \rightarrow 8\ ^{**}\underline{\Delta}\Delta\Delta \rightarrow 1\ ^{**}\underline{a}$
$\rightarrow 2\ ^{***}\underline{\Delta}$ CRASH

(iii) 1 $\underline{a}abaa \rightarrow 2\ ^{*}\underline{a}baa \rightarrow 2\ ^{*}a\underline{b}aa \rightarrow 3\ ^{*}ab\underline{a}a \rightarrow 4\ ^{*}ab\underline{a}a$
$\rightarrow 5\ ^{*}aa\underline{a}a \rightarrow 5\ ^{*}aaa\underline{a} \rightarrow 5\ ^{*}aaa\underline{\Delta} \rightarrow 6\ ^{*}aaa\underline{a} \rightarrow 7\ ^{*}aa\underline{\Delta}\Delta$
$\rightarrow 8\ ^{*}a\underline{\Delta}\Delta\Delta \rightarrow 8\ ^{*}\underline{a}a \rightarrow 8\ ^{*}\underline{\Delta}aa \rightarrow 1\ ^{*}\underline{a}a \rightarrow 2\ ^{**}\underline{a} \rightarrow 2\ ^{**}\underline{\Delta}$ CRASH

(iv) 1 $\underline{a}abbaabb \rightarrow 2\ ^{*}\underline{a}bbaabb \rightarrow 2\ ^{*}a\underline{b}baabb \rightarrow 3\ ^{*}ab\underline{b}aabb$
$\rightarrow 3\ ^{*}abb\underline{a}abb \rightarrow 4\ ^{*}ab\underline{b}aabb \rightarrow 5\ ^{*}aba\underline{a}abb \rightarrow 5\ ^{*}abaa\underline{a}bb$
$\rightarrow 5\ ^{*}abaaa\underline{b}b$ CRASH

(v) Strings that do not start with *a* or that have fewer leading *a*'s than *b*'s crash in 1 reading *b*. Strings with fewer *b*'s than trailing *a*'s crash in 2 reading Δ. Strings with too few trailing *a*'s crash in 3 reading Δ. Strings that have a *b* after the trailing *a*'s crash in 5 on *b*.

16.



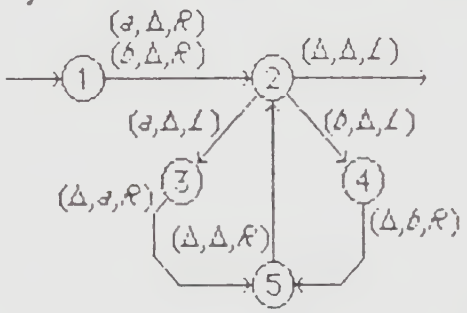
Chapter Nineteen

17.(i) 1 $a\underline{b}ab \rightarrow$ 2 $a\Delta\underline{a}b \rightarrow$ 2 $a\Delta a\underline{b} \rightarrow$ 2 $a\Delta ab\underline{\Delta} \rightarrow$ 3 $a\Delta a\underline{b} \rightarrow$ 5 $a\Delta\underline{a}\Delta \rightarrow$ 4 $a\underline{\Delta}b\Delta \rightarrow$ 7 $aa\underline{b}\Delta \rightarrow$? $a\underline{a}b$

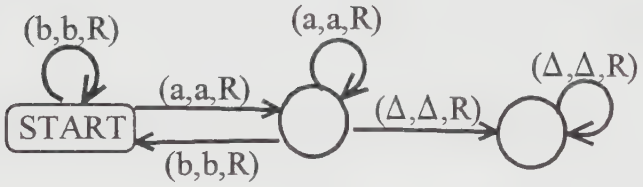
(ii) 1 $ab\underline{a}a \rightarrow$ 2 $ab\Delta\underline{\Delta}\Delta \rightarrow$ 3 $ab\Delta\underline{\Delta} \rightarrow$ CRASH This routine will not work on final letters. By putting a (Δ,Δ,R) transition from state 3 to state 7, and including (Δ,Δ,L) in the possible moves out of 7, we allow deletion of a final letter. However, the routine then leaves the TAPE HEAD pointing to the first blank after the string, which might be inadvisable.

(iii) 1 $ab\underline{a}b\Delta \rightarrow$ CRASH The routine does not permit deleting Δ.

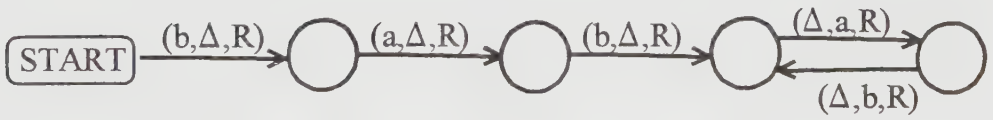
18. (For a and b only)



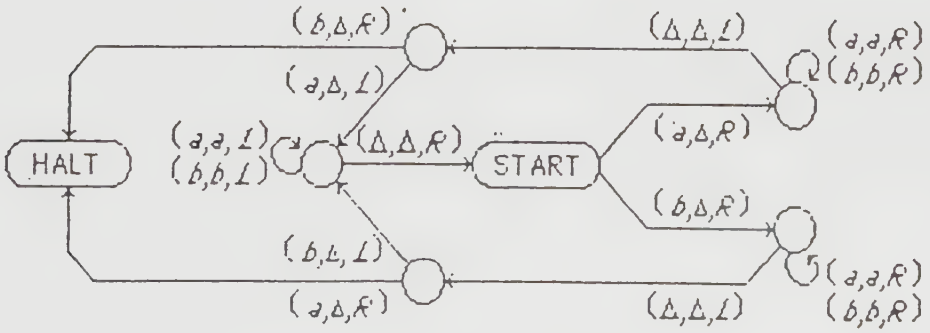
19. (i)



(ii)



20.



Chapter 20

Many instructors skip this chapter but then find themselves embarrassed by the references to it in the later sections of the text (in particular the next chapter and the discussion of r.e. and recursive languages). I personally believe that the work of Emile Post is as brilliant as Turing's and that but for the Allied decoding project Colossus, Post would get an equal share in the credit for founding computer theory. The fact that changing a STACK into a QUEUE escalates the ability of the device orders of magnitude is an insight of the greatest importance in this subject. Most people mistakenly believe that the Post model is limited as a practical model of a computer since it does not provide random access to the data field, only to the two ends of the STORE. However they then fail to realize that Turing's model also provides us with only two data cells of access at any instruction, the cell to the right of the TAPE HEAD and the cell to the left. The true advantage of Turing's description is that in its two-dimensional form, it allows us to imagine the whole blackboard as a TAPE to be operated on and so heuristically explains why the TM can perform all known and knowable "algorithms."

The fact that given READ-FRONT and ADD-BACK we can write subprograms for ADD-FRONT and READ-BACK whereas given ADD-FRONT and READ-FRONT we are stuck, is the purest example of the importance of data structures as a deep field of investigation.

Another selling point for the importance of this chapter is that it is the first and best example for simulating one entire machine on another (except for Mealy = Moore, which is essentially trivial). This is a skill that crops up in the actual practice of computer science all the time, and the student should have the theoretical basis of how to do it: equate data bases and instruction sets.

The whole chapter can be treated in one class and it richly deserves at least that.

Chapter Twenty

1.(i) START-$abab$ → $READ_2$-bab → $READ_1$-ab → $READ_2$-b $READ_1$ - Λ
→ $READ_2$ - Λ - ACCEPT

(ii) START-$baabba$ →$READ_2$- $aabba$ → $READ_3$- $abba$ → $READ_2$- bba → $READ_1$- ba
→ $READ_2$ - a → $READ_3$ - Λ → $READ_2$ - Λ - ACCEPT

(iii) START - $aaabbb$ → $READ_2$ - $aabbb$ → $READ_1$ - $abbb$ → ADD a - $abbba$
→ $READ_1$ - $bbba$ → $READ_2$ - baa →$READ_3$ - aa → ADD b - aab →$READ_3$ - ab
→ $READ_2$ - b →$READ_1$ - Λ → $READ_2$ - Λ → ACCEPT

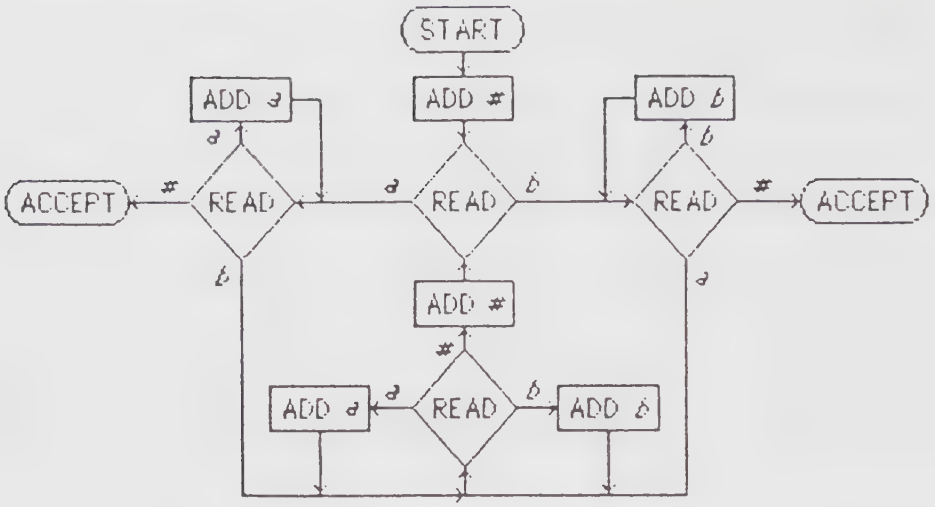
(iv) START - $aabbbb$ → $READ_2$ - $abbbb$ → $READ_1$ - $bbbb$ → ADD a - $bbbba$
→ $READ_1$ - $bbba$ → $READ_2$ - bba →$READ_3$ - ba → ADD b - bab → $READ_3$ - ab
→ ADD b - abb → $READ_3$ - bb → $READ_2$ - b → $READ_3$ - Λ
→ [ADD b - b → $READ_3$ - Λ → ADD b ...] INFINITE LOOP

(v) START - $bbabaaa$ → $READ_2$ - $babaaa$ → $READ_3$ - $abaaa$ → ADD b - $abaaab$
→ $READ_3$ - $baaab$ → $READ_2$ - $aaab$ → $READ_3$ - aab → $READ_2$ - ab → $READ_1$ -
b → ADD a - ba → $READ_1$ - a → $READ_2$ - Λ → $READ_1$ - Λ → CRASH

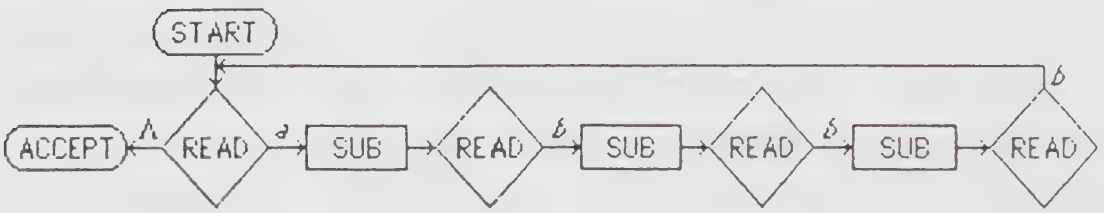
2 and 3. The machine works as follows: $READ_2$ is the state of having consumed equal numbers of a's and b's. If the string becomes null in this state we accept it. $READ_1$ is the state of just having read the first excess a, $READ_3$ of just having read the first extra b. In these states, reading the minority letter just cancels out the previous excess letter. However, if we read the majority letter in one of these states we add it to the STORE. A string that contains a single surplus a consumes all the b's and the non-excess a's, perhaps on several passes through the input. Finally, the extra a is read in $READ_2$, control transfers to $READ_1$ where the string crashes reading Λ. A symmetric argument applies for b's. More than one letter in excess results in the cancellation of equal numbers of both, as before, but once the minority letter has been exhausted there is no way to pair the excess letters. Every excess letter after the first is added to the STORE, which is never shortened to Λ.

Chapter Twenty

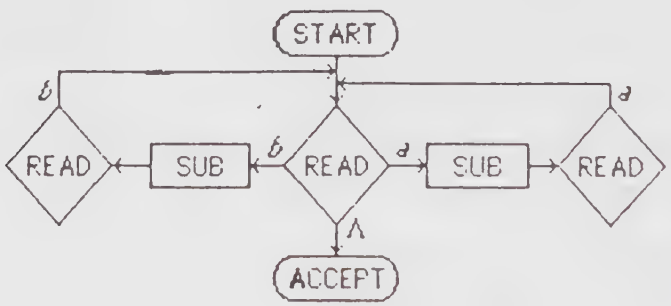
4.



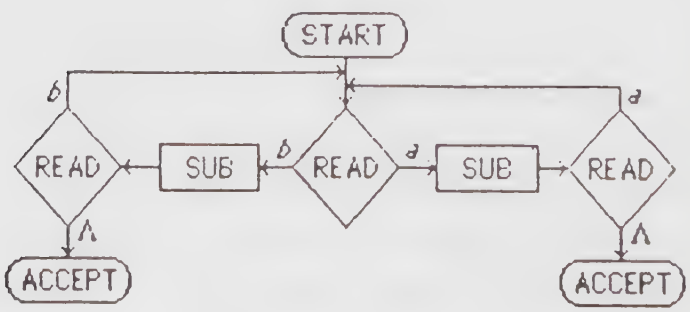
5.



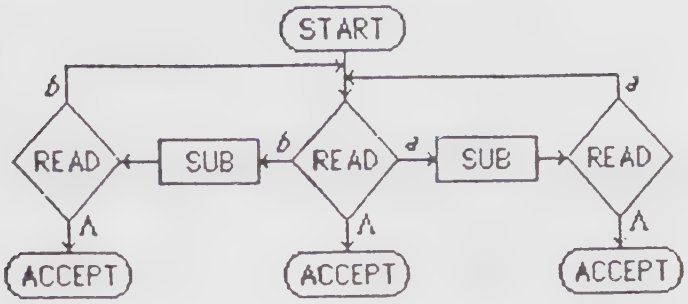
6.



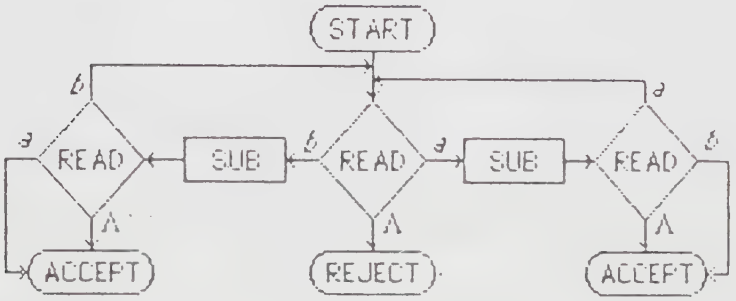
7. (i)



(ii)



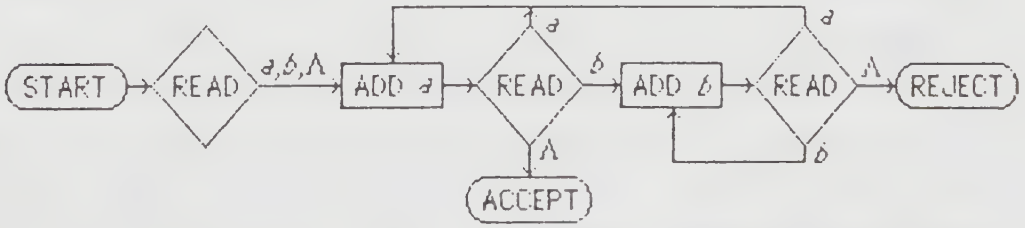
8.



9. (i) There might not be a way to accept the words in the LOOP set of the PM.

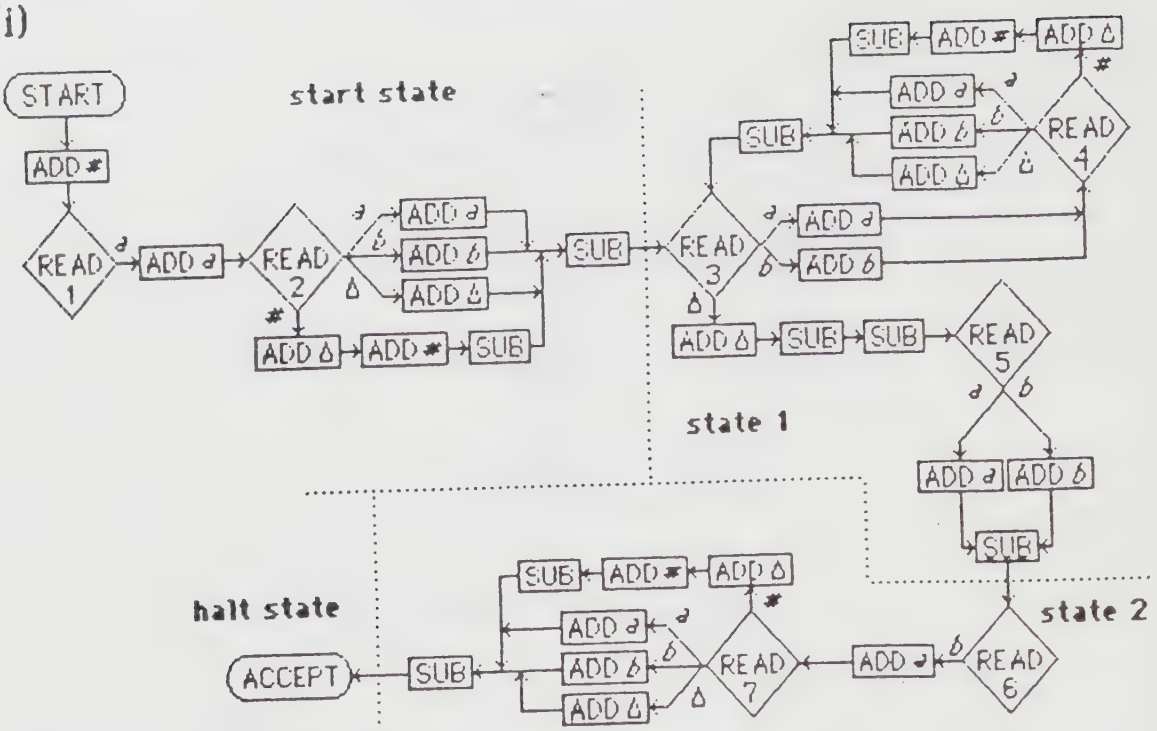
(ii) The PM for EQUAL on p. 613 cannot be complementing by reversing ACCEPT and REJECT states, even if ACCEPT states are drawn in for the condition of READ Λ in $READ_1$ and $READ_3$ of the complement machine.

(iii)



10. By Theorem 45, every regular language can be accepted by some Turing machine. By Theorem 47 every Turing machine can be converted into a Post machine. Therefore all regular languages can be accepted by some PM.

11.(i)



(ii) (Turing) START $\underline{a} \rightarrow 1\ a\underline{\Delta} \rightarrow 2\ a$ CRASH

(Post) START- $a \rightarrow$ ADD $*$- $a^* \rightarrow$ READ$_1$-$^* \rightarrow$ READ$_2$-$\Delta \rightarrow$ ADD Δ- $\Delta \rightarrow$ ADD $*$ - $\Delta^* \rightarrow$ SUB - $^*\Delta \rightarrow$ SUB - $\Delta^* \rightarrow$ READ$_3$ - $^* \rightarrow$ ADD Δ - $^*\Delta \rightarrow$ SUB - $\Delta^* \rightarrow$ SUB $^*\Delta \rightarrow$ READ$_5$ - Δ CRASH

(iii)(Turing) START $\underline{ab} \rightarrow 1\ a\underline{b} \rightarrow 1\ ab\underline{\Delta} \rightarrow 2\ a\underline{b} \rightarrow$ ACCEPT $a\underline{a\Delta}$

(Post) START $ab \rightarrow$ ADD * - $ab^* \rightarrow$ READ$_1$ - $b^* \rightarrow$ ADD a - $b^*a \rightarrow$ READ$_2$ - $^*a$ $\rightarrow$ ADD b - $^*ab \rightarrow$ SUB $b^*a \rightarrow$ READ$_3$ - $^*a \rightarrow$ ADD b - $^*ab \rightarrow$ READ$_4$ - $ab \rightarrow$ ADD Δ - $ab\Delta \rightarrow$ ADD * - $ab\Delta^* \rightarrow$ SUB - $^*ab\Delta \rightarrow$ SUB - $\Delta^*ab \rightarrow$ READ$_3$ - $^*ab \rightarrow$ ADD Δ- $^*ab\Delta \rightarrow$SUB - $\Delta^*ab \rightarrow$ SUB - $b\Delta^*a \rightarrow$ READ$_5$ - $\Delta^*a \rightarrow$ ADD b - $\Delta^*ab \rightarrow$ SUB - $b\Delta^*a \rightarrow$ READ$_6$ - $\Delta^*a \rightarrow$ ADD a - $\Delta^*aa \rightarrow$ READ$_7$ - $^*aa \rightarrow$ ADD Δ - $^*aa\Delta \rightarrow$ SUB - $\Delta^*aa \rightarrow$ ACCEPT

(iv) (Turing) START $\underline{a}bb \rightarrow 1\ a\underline{b}b \rightarrow 1\ ab\underline{b} \rightarrow 1\ abb\underline{\Delta} \rightarrow 2\ ab\underline{b} \rightarrow$ ACCEPT $ab\underline{a\Delta}$

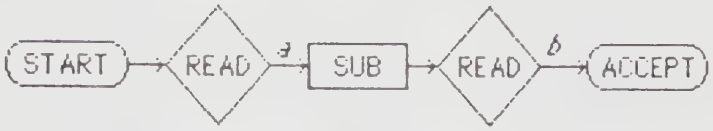
(Post) START- $abb \rightarrow$ ADD *- $abb^* \rightarrow$ READ$_1$- $bb^* \rightarrow$ ADD a- $bb^*a \rightarrow$ READ$_2$- b^*a $\rightarrow$ ADD b- $b^*ab \rightarrow$ SUB - $bb^*a \rightarrow$ READ$_3$- $b^*a \rightarrow$ ADD b - $b^*ab \rightarrow$ READ$_4$ - $^*ab$ $\rightarrow$ ADD b - $^*abb \rightarrow$ SUB - $b^*ab \rightarrow$ READ$_3$ - $^*ab \rightarrow$ ADD b - $^*abb \rightarrow$ READ$_4$ - abb $\rightarrow$ ADD Δ- $abb\Delta \rightarrow$ ADD *- $abb\Delta^* \rightarrow$ SUB - $^*abb\Delta \rightarrow$ SUB - $\Delta^*abb \rightarrow$ READ$_3$- $^*abb$

→ ADD Δ- $*abb\Delta$→ SUB- $\Delta*abb$→SUB- $b\Delta*ab$→READ$_5$ - $\Delta*ab$ → ADD b- $\Delta*abb$
→ SUB- $b\Delta*ab$→READ$_6$- $\Delta*ab$ → ADD a- $*aba$ → READ$_7$ - aba → ADD Δ - $ab\Delta$
→ ADD * - $ab\Delta*$ → SUB - $*ab\Delta$ → SUB - $\Delta*aba$ ACCEPT

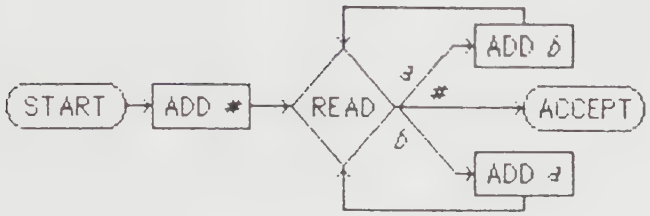
Note in (iii) and (iv) that the initial $\Delta^{\#}$ at ACCEPT shows that the (simulated) TAPE HEAD is reading the first blank after the input.

(v) **a(a+b)*b**

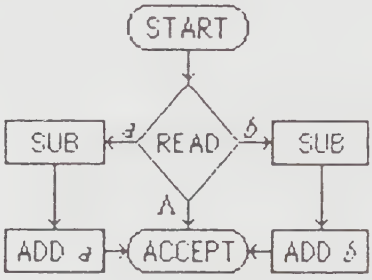
(vi)



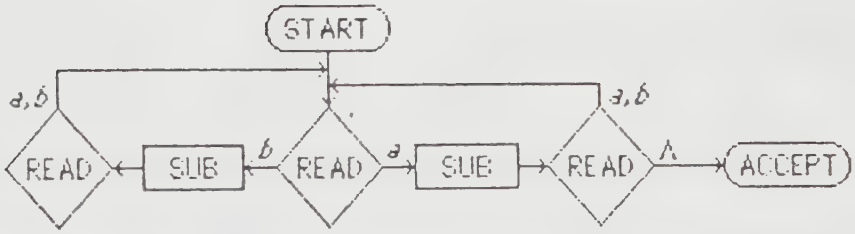
12.(i)



(ii)



13.(i)

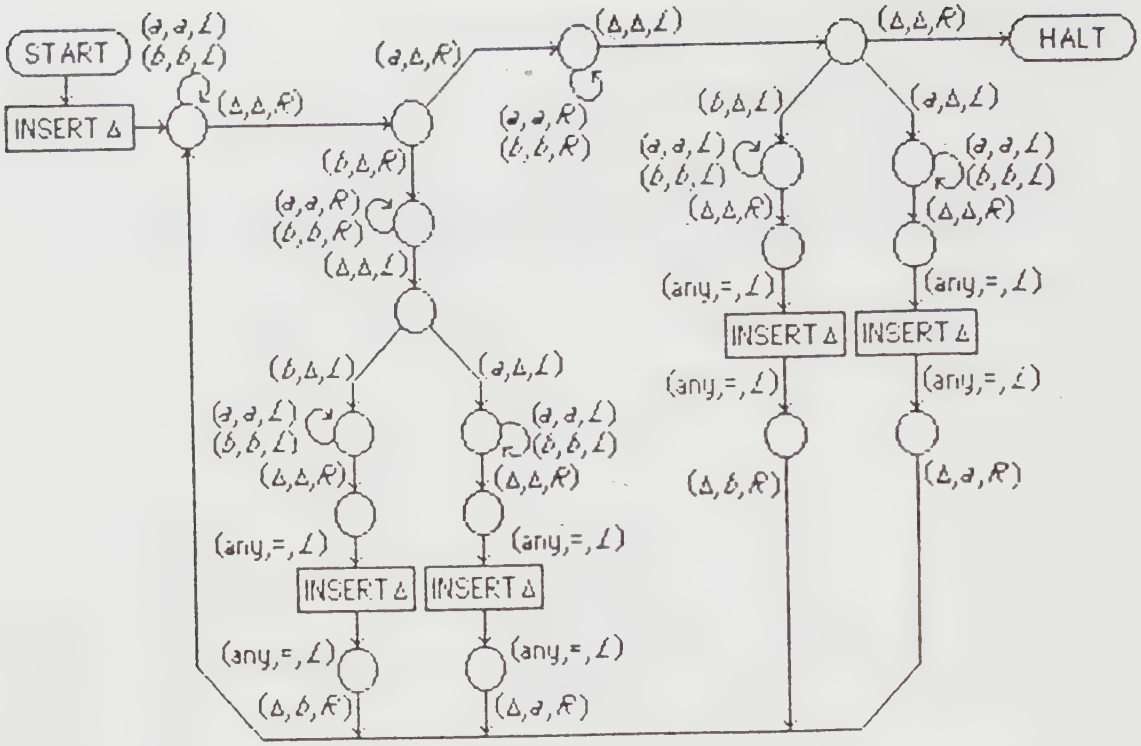


(ii) Take a counterexample of the form ($b^n a b^n$) and use the strong form of the Pumping Lemma for regular languages.

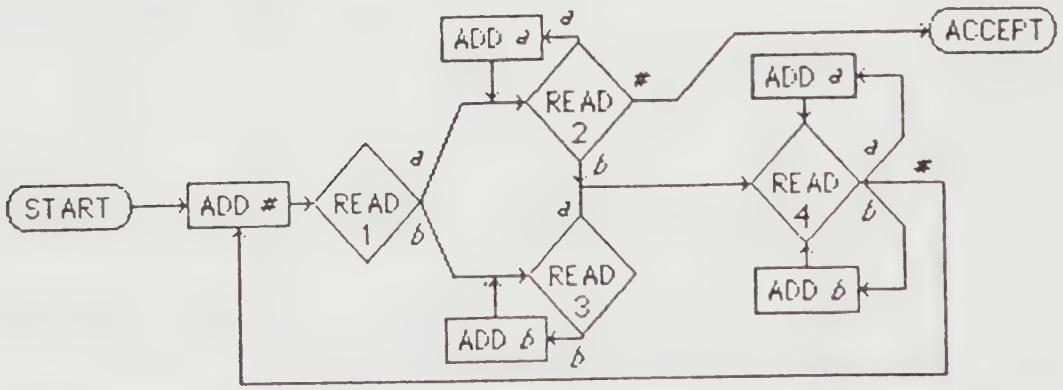
(iii) It can be generated by the CFG $S \rightarrow aSa \mid aSb \mid bSa \mid bSb \mid a$

14. The algorithm in the chapter does not account for SUB. 1. Read and delete the rightmost letter; 2. Move all the way left; 3. Insert Δ before the string; 4. Write the deleted letter onto the Δ; 5. Return the head to the reading position:

Chapter Twenty



15.



(i) START-$aabb$ → ADD #-$aabb\#$ → $READ_1$-$abb\#$ → $READ_2$-$bb\#$ → ADD a-$bb\#a$
→ $READ_2$-$b\#a$ → $READ_4$-$\#a$ → ADD b-$\#ab$ → $READ_4$-ab → ADD #-$ab\#$
→ $READ_1$-$b\#$ → $READ_2$-$\#$ → $READ_4$-Λ → ADD #-# → $READ_1$-Λ CRASH

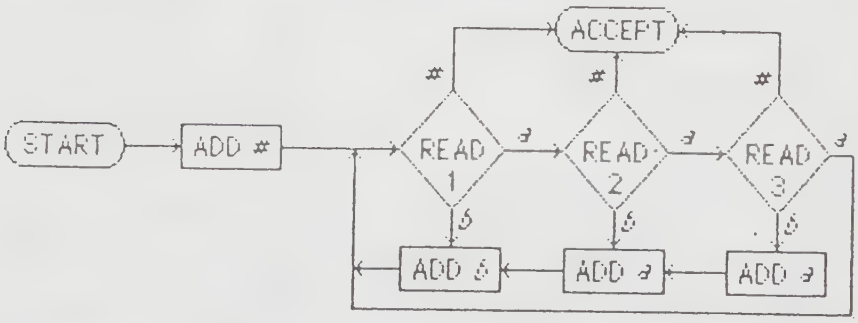
(ii) START-$aaabb$→ADD #-$aaabb\#$→$READ_1$-$aabb\#$→$READ_2$-$abb\#$→ADD a-$abb\#a$
→$READ_2$-$bb\#a$→ADD a-$bb\#aa$→ $READ_2$-$b\#aa$→ $READ_4$-$\#aa$→ ADD b-$\#aab$
→ $READ_4$-aab→ ADD #-$aab\#$ → $READ_1$-$ab\#$ → $READ_2$-$b\#$ → ADD a-$b\#a$
→ $READ_2$-$\#a$→ $READ_4$-a→ ADD #-$a\#$ → $READ_1$-$\#$ → $READ_2$-Λ → ACCEPT

Chapter Twenty

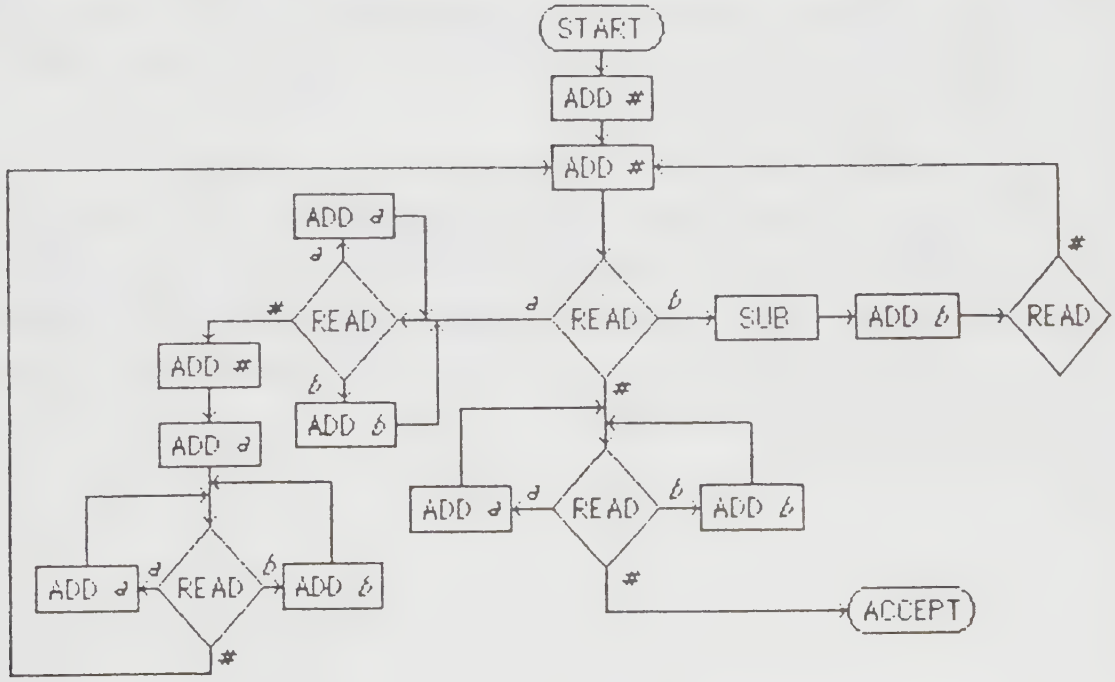
(iii) START- $ababa \rightarrow$ ADD $\#$ - $ababa^{\#} \rightarrow$ READ$_1$ - $babaa^{\#} \rightarrow$ READ$_2$ - $abaa^{\#} \rightarrow$ READ$_4$ - $baa^{\#}$
$\rightarrow$ ADD a - $baa^{\#}a \rightarrow$ READ$_4$ - $aa^{\#}a \rightarrow$ ADD b - $a^{\#}ab \rightarrow$ READ$_4$ - $^{\#}ab \rightarrow$ ADD a - $^{\#}aba$
$\rightarrow$ READ$_4$ - $aba \rightarrow$ ADD $\#$ - $aba^{\#} \rightarrow$ READ$_1$ - $baa^{\#} \rightarrow$ READ$_2$ - $a^{\#} \rightarrow$ READ$_4$ - $^{\#}$
$\rightarrow$ ADD a - $^{\#}a \rightarrow$ READ$_4$ - $a \rightarrow$ ADD $\#$ - $a^{\#} \rightarrow$ READ$_1$ - $^{\#} \rightarrow$ READ$_2$ - $\Lambda \rightarrow$ ACCEPT

(iv) START- $ababab \rightarrow$ ADD $\#$ - $ababab^{\#} \rightarrow$ READ$_1$ - $babab^{\#} \rightarrow$ READ$_2$ - $abab^{\#}$
$\rightarrow$ READ$_4$ - $bab^{\#} \rightarrow$ ADD $a \rightarrow bab^{\#}a \rightarrow$ READ$_4$ - $ab^{\#}a \rightarrow$ ADD b - $ab^{\#}ab$
$\rightarrow$ READ$_4$ - $b^{\#}ab \rightarrow$ ADD a - $b^{\#}aba \rightarrow$ READ$_4$ - $^{\#}aba \rightarrow$ ADD b - $^{\#}abab$
$\rightarrow$ READ$_4$ - $abab \rightarrow$ ADD $\#$ - $abab^{\#} \rightarrow$ READ- $bab^{\#} \rightarrow$ READ$_2$ - $ab^{\#} \rightarrow$ READ$_4$ - $b^{\#}$
$\rightarrow$ ADD a - $b^{\#}a \rightarrow$ READ$_4$ - $^{\#}a \rightarrow$ ADD b - $^{\#}ab \rightarrow$ READ$_4$ - $ab \rightarrow$ ADD $\#$ - $ab^{\#}$
$\rightarrow$ READ$_1$ - $b^{\#} \rightarrow$ READ$_2$ - $^{\#} \rightarrow$ READ$_4$ - $\Lambda \rightarrow$ ADD $\#$ - $^{\#} \rightarrow$ READ$_1$ - Λ CRASH

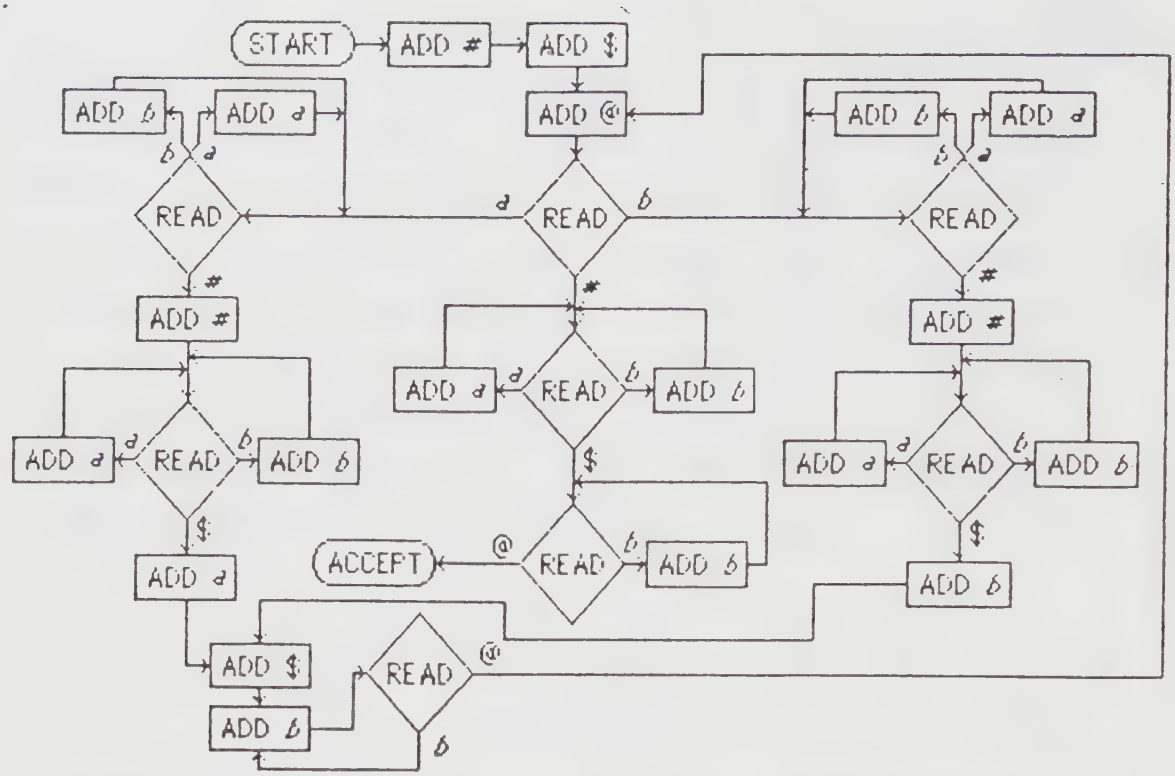
16.



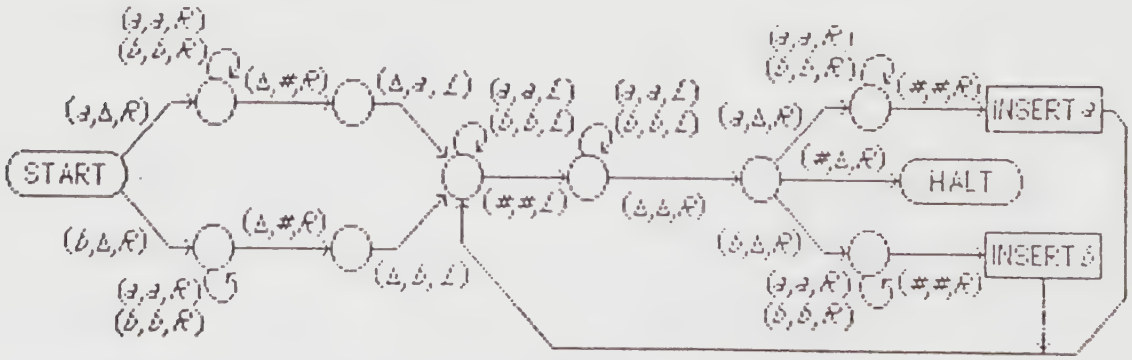
17.



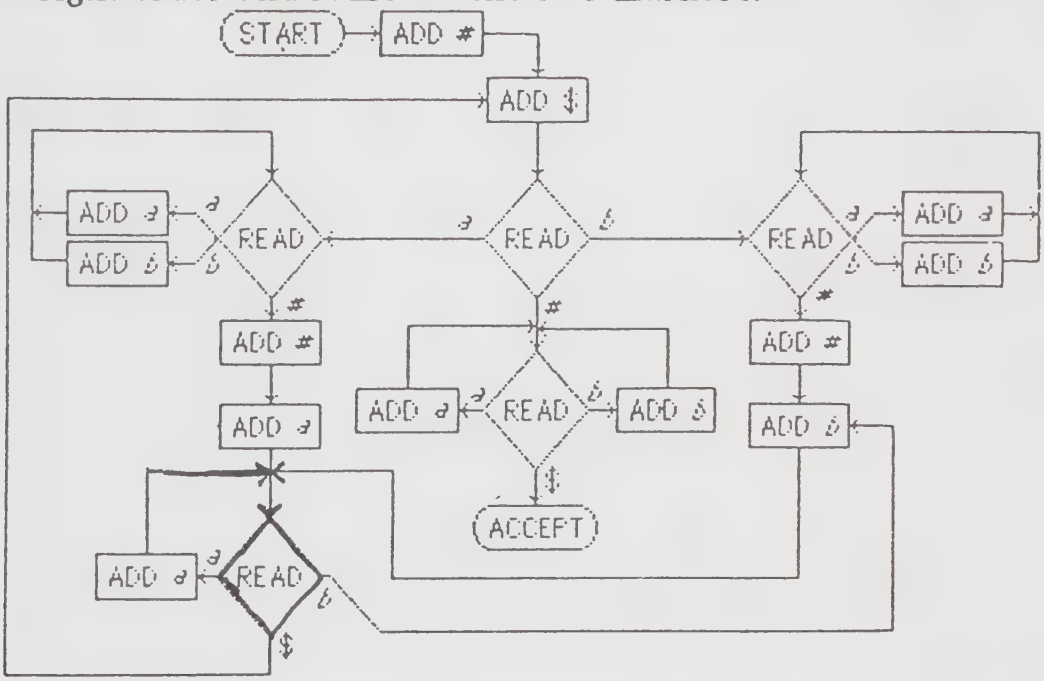
18.



19. (i) Delete the first letter and remember it by path choice; mark the end of the string with #; use INSERT to write the remembered first letter right after the #. When # is read, all letters have been processed. The INSERT routine will have to be a variant of the one presented that permits replacing Δ by the letter to be inserted.



(ii) Mark the end of the string with # and use $ to separate the reversed string from the unreversed. Read the first letter and remember it by path choice. Cycle through the string reading and adding each letter as it is encountered. When # is read it is added back and the remembered letter is added immediately after it. The rest of the string is ignored (i.e., read a letter, add it), until the $ that marks the end of one trip through the input. When the initial READ state reads # the entire string has been reversed. Cycling through to $ puts it in the right order and removes the two markers.



20. We can construct the PM, G algorithmically. Use the reversing machine above as a preprocessor and send its output, the reversed string, into P as input. If the reversed string is accepted by P the string is in L^T.

Chapter 21

This is another chapter that should not be skipped. Again it takes only one class and it pays off in two ways -- it organizes the whole structure of the course 0PDA. 1PDA, 2PDA, and at the same time it begins to convince students that TMs are in some sense "ultimate" expressions of algorithm design: TM = 2PDA = 3PDA = 4PDA = ...

Several of the variations covered in the next chapter could be proven equivalent to kPDAs as easily as to TMs directly. We did not do this for fear of disadvantaging the students who were told to skip this chapter. If this chapter is covered in your course, keep it in mind as a variant method of covering much of the material in the rest of the course.

Students should also be advised to read the excellent books by Minsky, not only a founder of the subject but a great expositor.

Chapter Twenty One

1.(i)

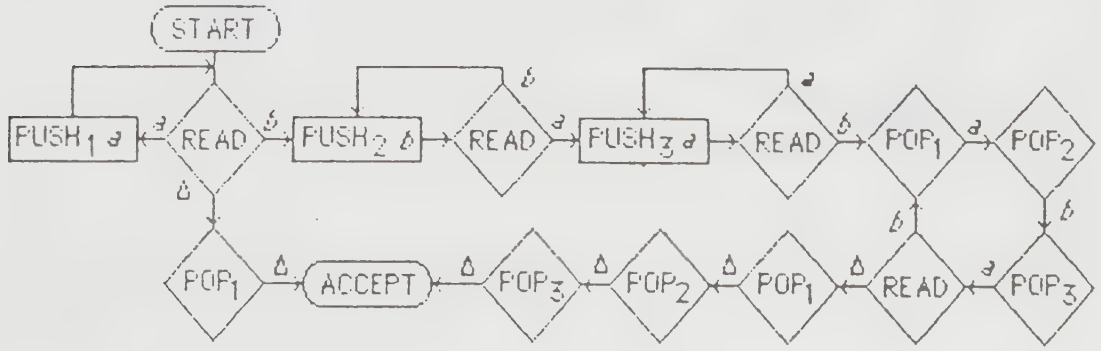
STATE	TAPE	$STACK_1$	$STACK_2$	
START	*aabb*	Δ	Δ	
READ	*abb*	Δ	Δ	
$PUSH_1$ *a*	*abb*	*a*	Δ	
READ	*bb*	*a*	Δ	
$PUSH_1$ *a*	*bb*	*aa*	Δ	
READ	*b*	*aa*	Δ	
$PUSH_2$ *b*	*b*	*aa*	*b*	
READ	Δ	*aa*	*b*	
$PUSH_2$ *b*	Δ	*aa*	*bb*	
READ	Δ	*aa*	*bb*	
POP_1	Δ	*a*	*bb*	
POP_2	Δ	*a*	*b*	
POP_1	Δ	Δ	*b*	
POP_2	Δ	Δ	Δ	
POP_1	Δ	Δ	Δ	
POP_2	Δ	Δ	Δ	ACCEPT

(ii)

STATE	TAPE	$STACK_1$	$STACK_2$	
START	*babab*	Δ	Δ	
READ	*abab*	Δ	Δ	
$PUSH_2$ *b*	*abab*	Δ	*b*	
READ	*bab*	Δ	*b*	
$PUSH_1$ *a*	*bab*	*a*	*b*	
READ	*ab*	*a*	*b*	
$PUSH_2$ *b*	*ab*	*a*	*bb*	
READ	*b*	*a*	*bb*	
$PUSH_1$ *a*	*b*	*aa*	*bb*	
READ	Δ	*aa*	*bb*	
$PUSH_2$ *b*	Δ	*aa*	*bbb*	
READ	Δ	*aa*	*bbb*	
POP_1	Δ	*a*	*bbb*	
POP_2	Δ	*a*	*bb*	
POP_1	Δ	*d*	*bb*	
POP_2	Δ	*d*	*b*	
POP_1	Δ	*d*	*b*	
POP_2	Δ	Δ	Δ	CRASH

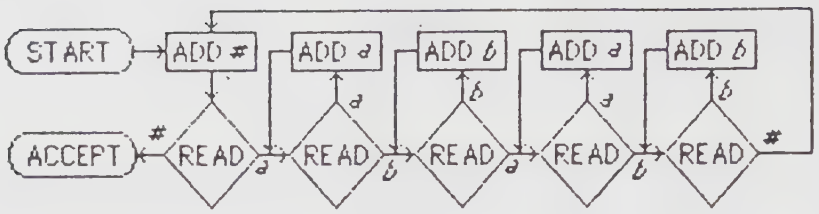
2. As the entire string is read, *a*'s are pushed onto $STACK_1$, *b*'s onto $STACK_2$. When the input is exhausted, the stacks are compared. To get to accept, they must both run out at the same time. Therefore, the strings accepted by this machine must have the same number of *a*'s as *b*'s.

3.

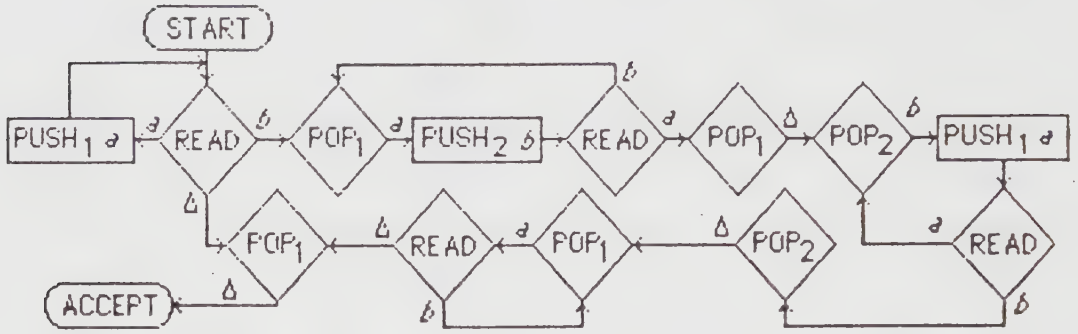


Chapter Twenty One

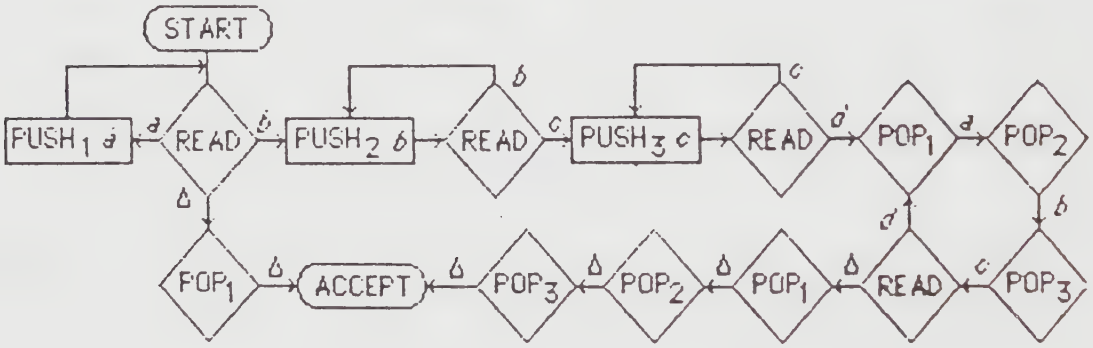
4.



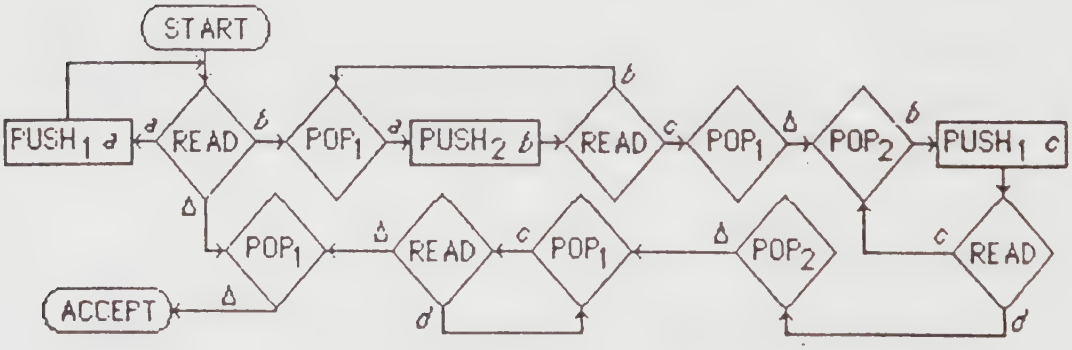
5.



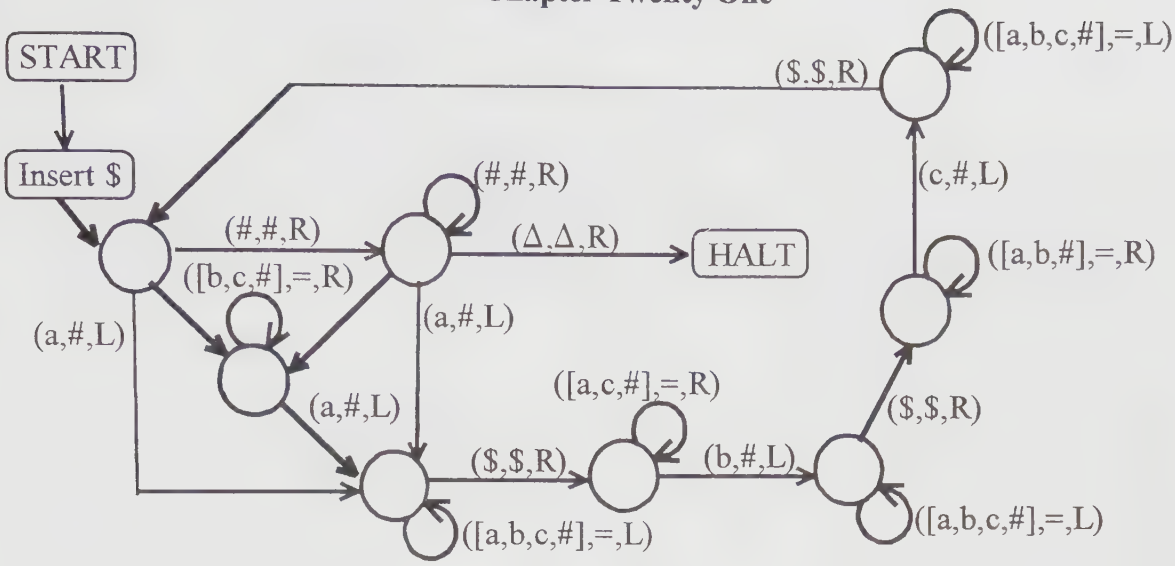
6.



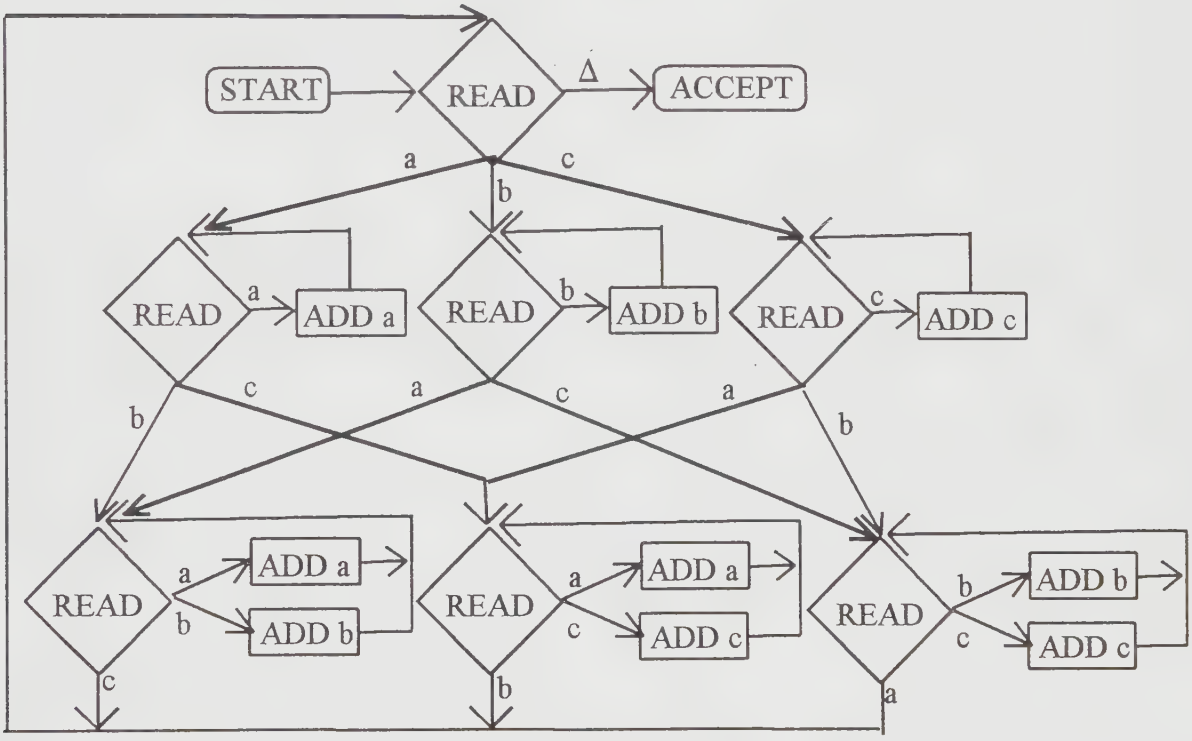
7.



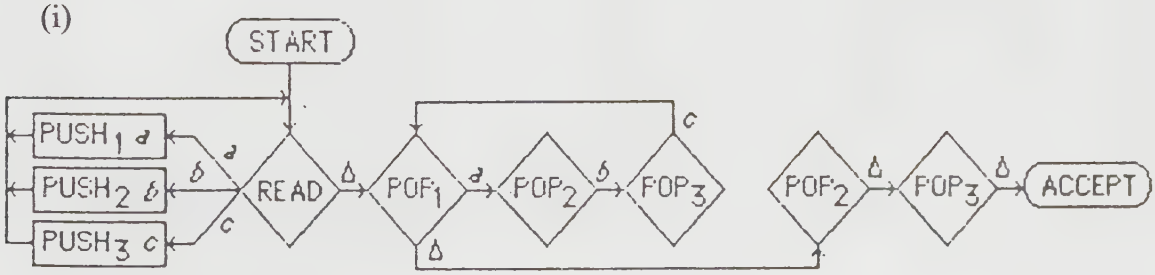
8.


START
Insert $
([a,b,c,#],=,L)
($,$,R)
(c,#,L)
([a,b,#],=,R)
(#,#,R)
(#,#,R)
(Δ,Δ,R)
HALT
([b,c,#],=,R)
(a,#,L)
(a,#,L)
(a,#,L)
([a,c,#],=,R)
($,$,R)
($,$,R)
(b,#,L)
([a,b,c,#],=,L)
([a,b,c,#],=,L)

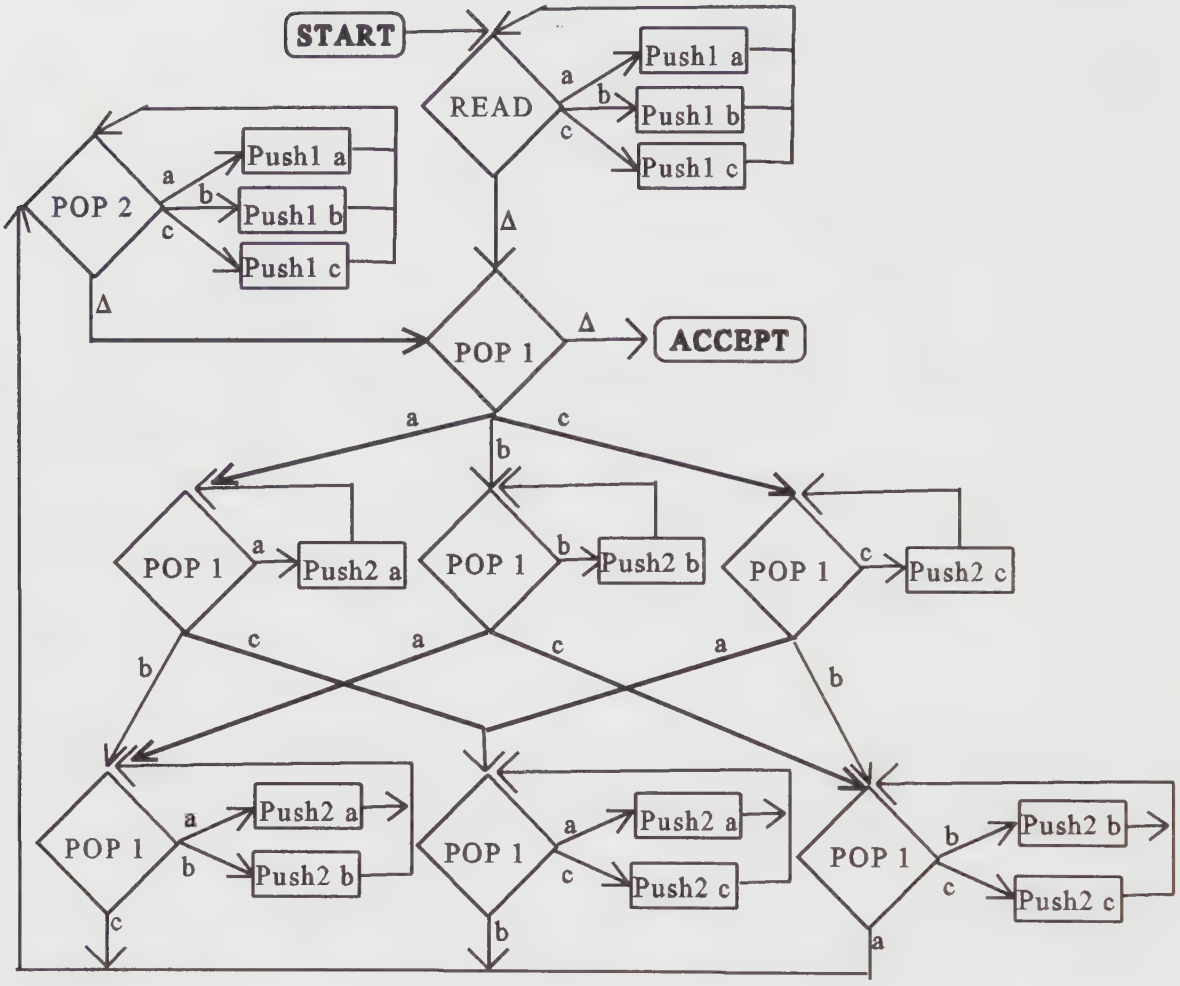
9.


START
READ
Δ
ACCEPT
a
b
c
READ
a
ADD a
READ
b
ADD b
READ
c
ADD c
READ
a
ADD a
b
ADD b
c
READ
a
ADD a
c
ADD c
b
READ
b
ADD b
c
ADD c
a

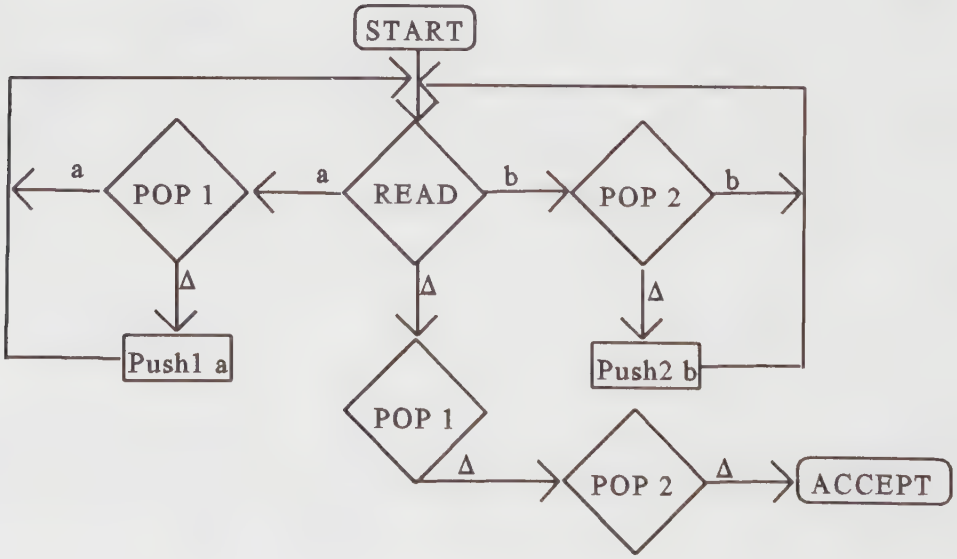
10. (i)


START
PUSH 1 a
PUSH 2 b
PUSH 3 c
READ
POP 1
POP 2
POP 3
POP 2
POP 3
ACCEPT
Δ

(ii)

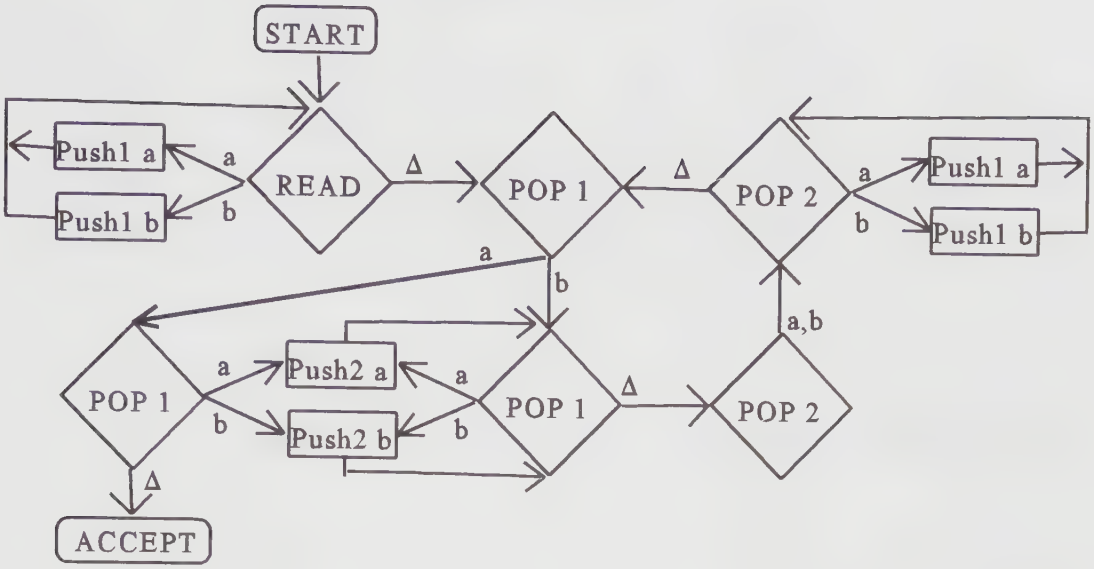

START
READ
Push1 a
Push1 b
Push1 c
POP 2
Push1 a
Push1 b
Push1 c
POP 1
ACCEPT
POP 1
Push2 a
POP 1
Push2 b
POP 1
Push2 c
POP 1
Push2 a
Push2 b
POP 1
Push2 a
Push2 c
POP 1
Push2 b
Push2 c

11.

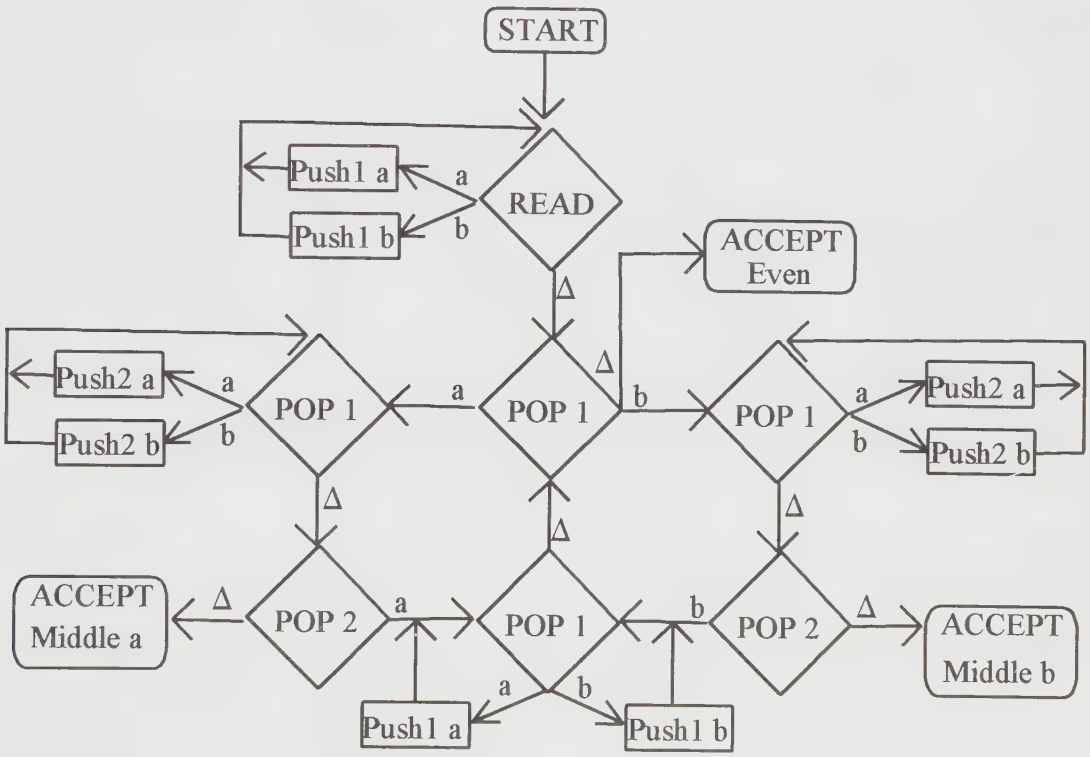

START
READ
POP 1
POP 2
Push1 a
Push2 b
POP 1
POP 2
ACCEPT

Chapter Twenty One

12.

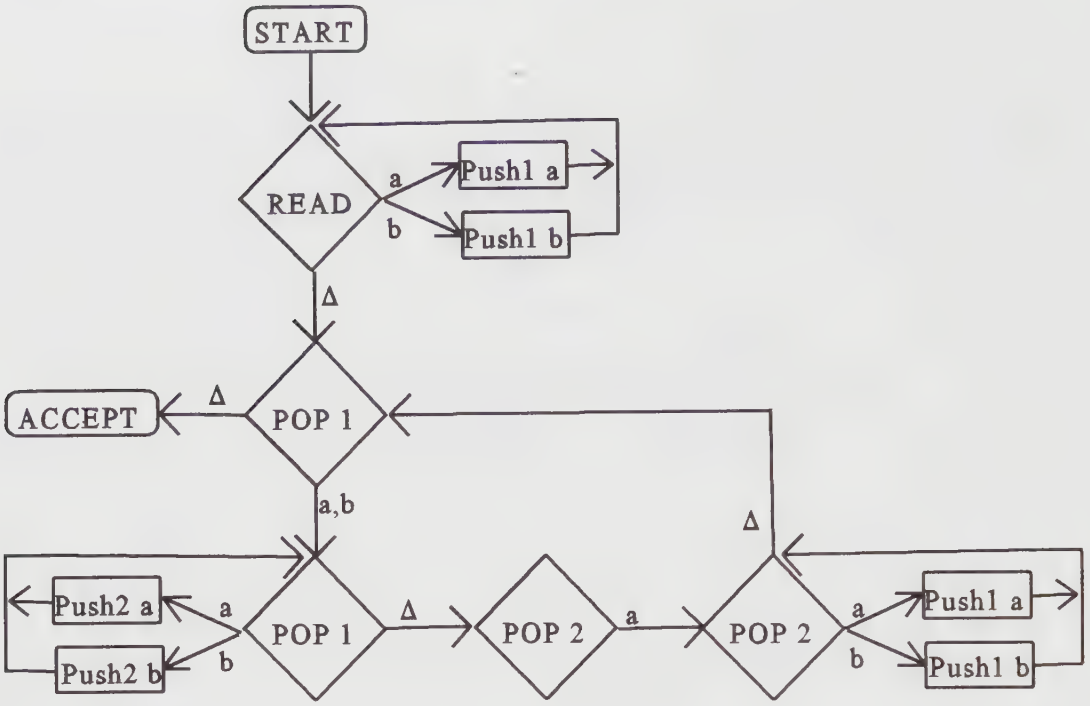


13.

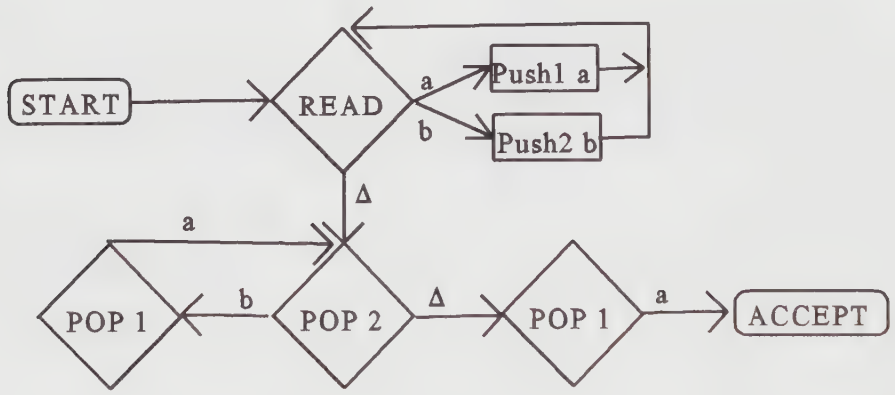


Chapter Twenty One

14.



15.



Chapter Twenty One

16.(i) START $\underline{a}bb \rightarrow 2\ \Delta\underline{b}b \rightarrow 3\ \Delta a\underline{b} \rightarrow 3\ \Delta a\underline{b}\Delta \rightarrow 4\ \Delta abb\Delta \rightarrow 5\ \Delta abb^{\#} \rightarrow 6\ \Delta abb^{\#}\_$
$\rightarrow 6\ \Delta a\underline{b}b^{\#} \rightarrow 6\ \Delta a\underline{b}b^{\#} \rightarrow 6\ \Delta\underline{a}bb^{\#} \rightarrow 6 - \Delta\underline{a}bb^{\#} \rightarrow 7\ \Delta\underline{a}bb^{\#} \rightarrow 8\ \Delta\Delta bb^{\#}$
$\rightarrow 8\ \Delta\Delta\underline{b}b^{\#} \rightarrow 8\ \Delta\Delta\underline{b}b^{\#} \rightarrow 8\ \Delta\Delta b\underline{b}^{\#} \rightarrow 8\ \Delta\Delta bb^{\#}\_ \rightarrow 9\ \Delta\Delta b\underline{b}^{\#} \rightarrow$ HALT $\Delta\Delta bb^{\#}\_$

(ii) START $\underline{b}aa \rightarrow 3\ \Delta\underline{a}a \rightarrow 2\ \Delta b\underline{a} \rightarrow 2\ \Delta b\underline{a}\Delta \rightarrow 4\ \Delta ba\Delta \rightarrow 5\ \Delta ba\underline{a}^{\#} \rightarrow 6\ \Delta baa^{\#}\_$
$\rightarrow 6\ \Delta ba\underline{a}^{\#} \rightarrow 6\ \Delta b\underline{a}a^{\#} \rightarrow 6\ \Delta\underline{b}aa^{\#} \rightarrow 6 - \Delta\underline{b}aa^{\#} \rightarrow 7 - \Delta\underline{b}aa^{\#} \rightarrow 11 - \Delta\Delta aa^{\#} \rightarrow$
$11 - \Delta\Delta\underline{a}a^{\#} \rightarrow 11 - \Delta\Delta\underline{a}a^{\#} \rightarrow 11 - \Delta\Delta a\underline{a}^{\#} \rightarrow 11 - \Delta\Delta aa^{\#}\_ \rightarrow 12 - \Delta\Delta a\underline{a}^{\#} \rightarrow 13 -$
$\Delta\Delta aa^{\#}\_ \rightarrow 14 - \Delta\Delta aa^{\#}\Delta \rightarrow 19 - \Delta\Delta aa^{\#}\_b \rightarrow 20 - \Delta\Delta aa^{\#}\underline{b} \rightarrow 21 - \Delta\Delta aa^{\#}b\Delta \rightarrow 24 -$
$\Delta\Delta aa^{\#}\underline{b}\Delta \rightarrow 25 - \Delta\Delta aa^{\#}b\Delta$ - CRASH

17.(i)

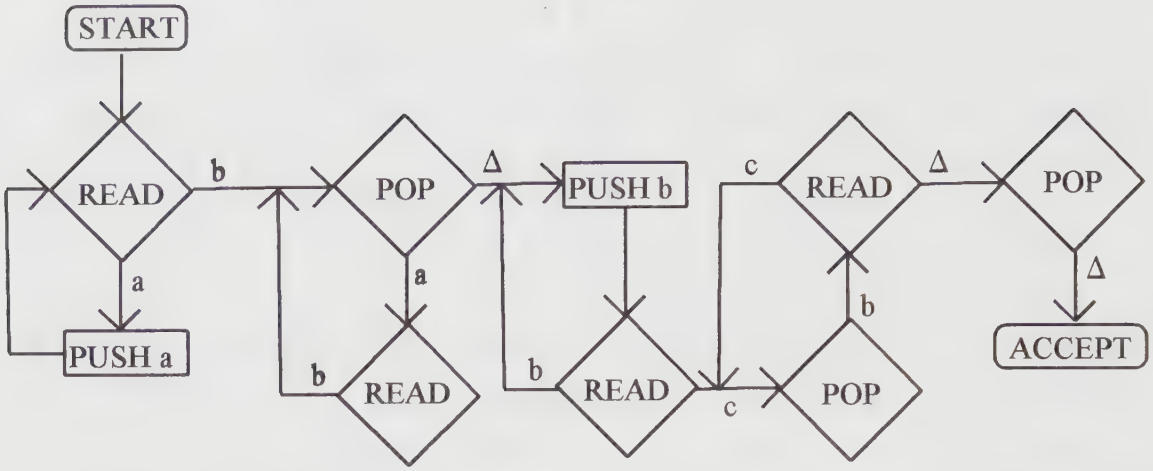
STATE	TAPE	$STACK_1$	$STACK_2$
START	$abba$	Δ	Δ
READ	bba	Δ	Δ
$PUSH_2$ a	bba	Δ	a
READ	ba	Δ	a
$PUSH_2$ b	ba	Δ	ba
READ	a	Δ	ba
$PUSH_2$ b	a	Δ	bba
READ	Δ	Δ	bba
$PUSH_2$ a	Δ	Δ	$abba$
READ	Δ	Δ	$abba$
POP_2	Δ	Δ	bba
$PUSH_1$ a	Δ	a	bba
POP_2	Δ	a	ba
$PUSH_1$ b	Δ	ba	ba
POP_2	Δ	ba	a
$PUSH_1$ b	Δ	bba	a
POP_2	Δ	bba	Δ
$PUSH_1$ a	Δ	$abba$	Δ
POP_2	Δ	$abba$	Δ
POP_1	Δ	bba	Δ
POP_1	Δ	ba	Δ
POP_1	Δ	a	Δ

(ii)

STATE	TAPE	$STACK_1$	$STACK_2$
START	$babab$	Δ	Δ
READ	$abab$	Δ	Δ
$PUSH_2$ b	$abab$	Δ	b
READ	bab	Δ	b
$PUSH_2$ a	bab	Δ	ab
READ	ab	Δ	ab
$PUSH_2$ b	ab	Δ	bab
READ	b	Δ	bab
$PUSH_2$ a	b	Δ	$abab$
READ	Δ	Δ	$abab$
$PUSH_2$ b	Δ	Δ	$babab$
READ	Δ	Δ	$babab$
POP_2	Δ	Δ	$abab$
$PUSH_1$ b	Δ	b	$abab$
POP_2	Δ	b	bab
$PUSH_1$ a	Δ	ab	bab
POP_2	Δ	ab	ab
$PUSH_1$ b	Δ	bab	ab
POP_2	Δ	bab	b
$PUSH_1$ a	Δ	$abab$	b
POP_2	Δ	$abab$	Δ
$PUSH_1$ b	Δ	$babab$	Δ

Chapter Twenty One

18. (iii)



19. L is accepted by some 2PDA; therefore, by Minsky's Theorem, there is a PM that accepts L. In chapter 20, problem 20 there is a PM that accepts L^T. (It is not possible to use the reversing procedure of problem 19 as a preprocessor unless the algorithm either uses only one stack for processing or simulates a PM, in which case a marker can be uses to separate the reversed input in one stack from the contents of the other stack.)

20. (i) Use the folded tape model. If all three stacks were being maintained on a single track tape, and the tape was folded at the point of the tape head then two stacks could easily simulate the tape head movement (move right means pop 2 x, push 1 x; move left means pop 1 x, push 2 x.) So initially place four #'s one stack 1 subsequently pop1-push2 the items over to stack 2 to adjust to the top of the desired stack before changing that stack. Finally before the simulator resumes processing, restore the stacked stacks to stack 1 with a series of pop2-push1 operations. For the most part stack 1 will look like:

#
stack 1
#
stack 2
#
stack 3
#

(ii) Obvious.

Chapter 22

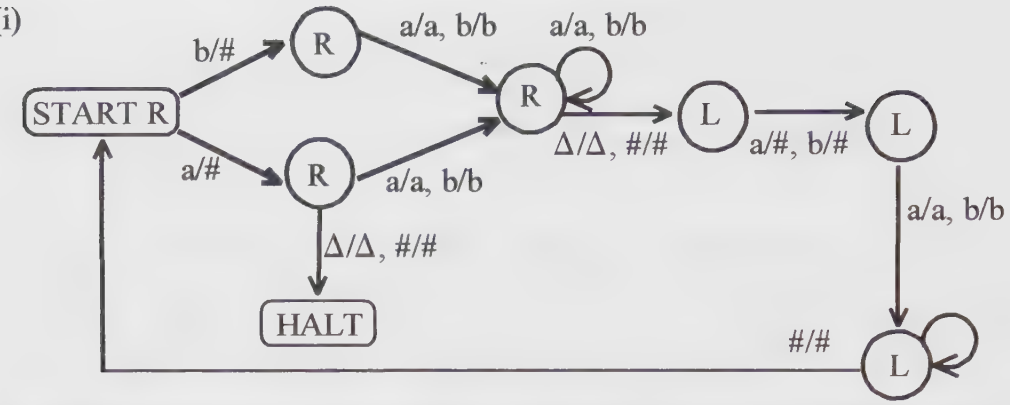
Despite the fact that all of these results are very important, the student who has suffered through all that has come before should be able to create most of these proofs for herself, with the possible exception of the equivalence of nondeterminism and determinism. Once the trick of how to do this is seen it should be obvious that it is not important to use a multi-track TAPE but that a few markers on the one track TAPE should suffice.

I toyed with the idea of proving that NPM=PM and that N2PDA=2PDA as separate results independent of TMs. The trouble was that since the best place to organize the idea of "try every alternative in sequence until we find one that works" is with TMs, and the best time is after the introduction of PMs and 2PDAs and their equivalence to TMs, by the point that the proof would be easy it is already moot.

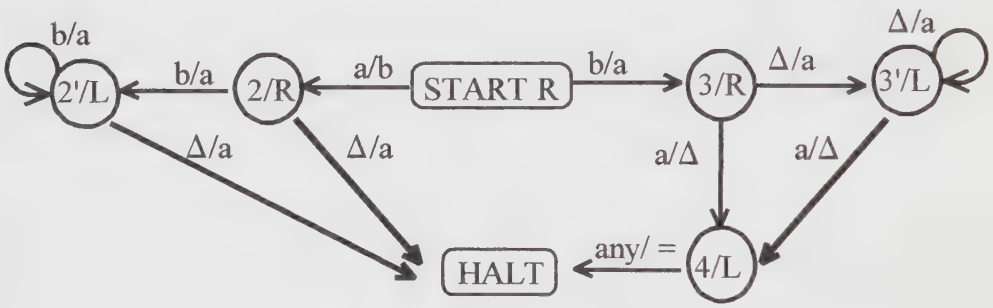
The discussion of Church's Thesis relys on the two-dimensional TM TAPE. Therefore it is necessary to cover kTMs, at least conversationally.

Chapter Twenty Two

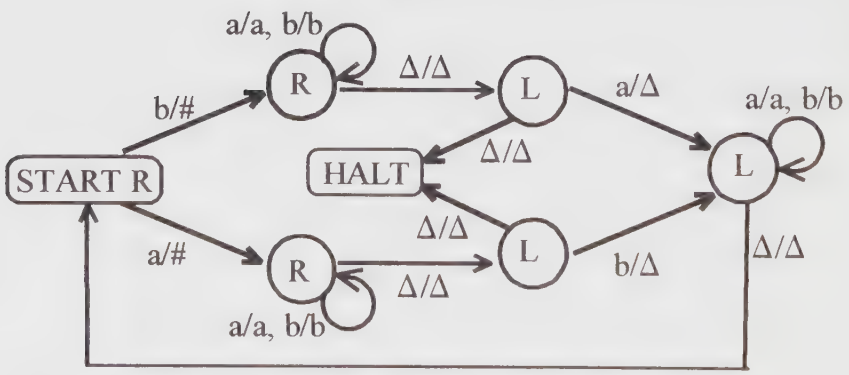
1. (i)



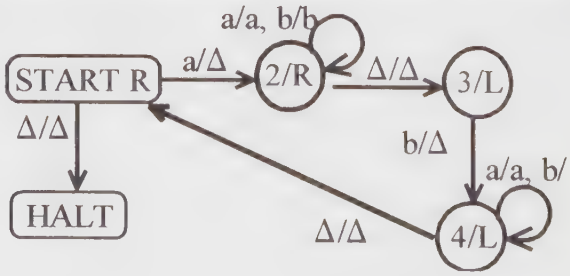
(ii) Each of the two plain states needs a twin state to handle the different move directions. Also notice in the machine below that we distinguish between a move left and move right to HALT. The tape head can always move right but moving left from the cell 1 causes a crash. Hence we needed to add yet one more new state.



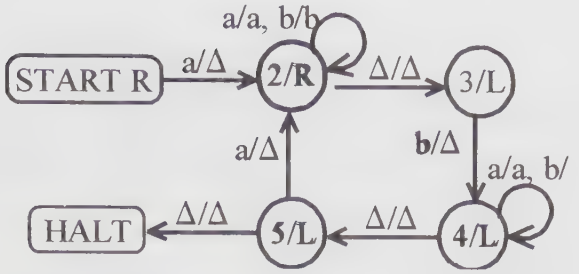
2. (i)



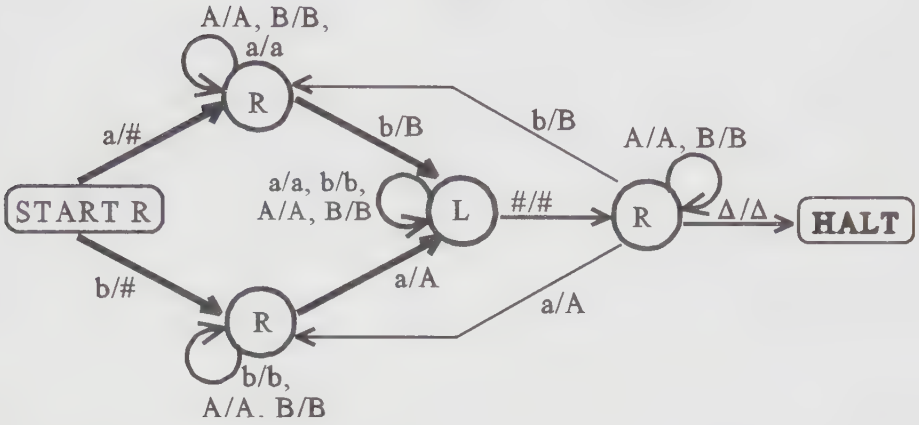
(ii) $\{a^n b^n \mid n \geq 0\}$



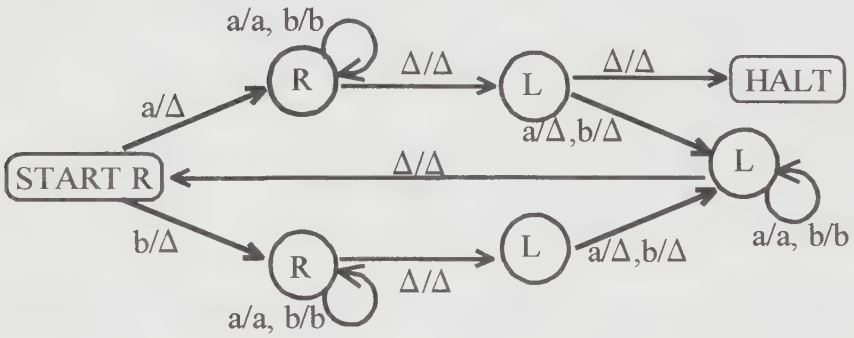
$\{a^n b^n \mid n \geq 1\}$



(iii)



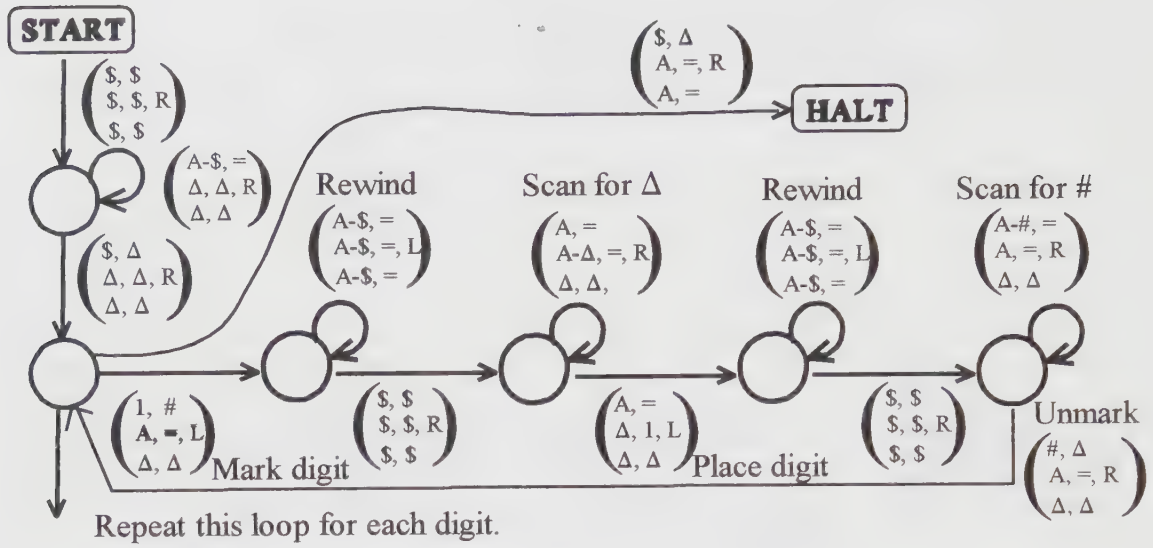
(iv)



3. Convert the multi-cell-move machine to a TM by adding states: If the instruction on the transition from q_i to state q_j is to move *n* cells in direction *d* then between states q_i and q_j insert *n*-1 new states each with the transition (any, =, *d*). These additional instructions move the TAPE HEAD as required by the multi-cell-instruction. This algorithm demonstrates the equivalence of the machines.

Chapter Twenty Two

4.



5. Clearly k+1 markers are needed to separate the data section of the k tracks. In addition, in order to keep track of the position of the tape head, we need k more markers of a different symbol. Notice that this method provides extra versatility, because each of the k abstract tape heads can move independently of the others. In total that makes $2k$+1 markers - not all of which must be different. Now we already have INSERT and DELETE subroutines, so we can easily maintain that the markers surround real data without extra working space. Any time we encounter an end of data section marker for some track we can always insert the necessary blanks and keep working.

6. (i) 1) Scan the tape inserting zeros on the track 1 for as long as there are digits in a cell on tracks 2, 3, or 4.

2a) For each combination of values for carry1, digit2, digit3, and digit4 such that their sum is a single digit, place the result on track 5 and move left.

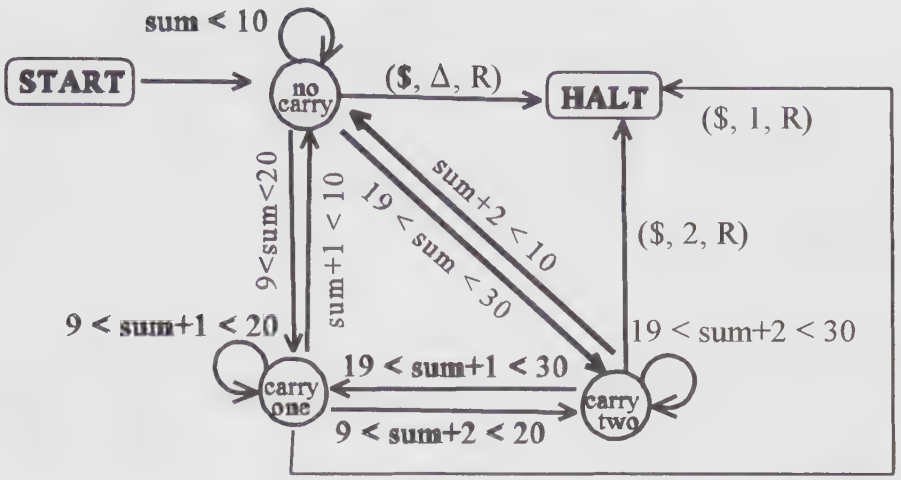
2b) For each combination of values for carry1, digit2, digit3, and digit4 such that their sum is 10 + a single digit, place the single digit result on track 5, move left, place a 1 on track 1 (carry) and do not move the tape head.

2c) For each combination of values for carry1, digit2, digit3, and digit4 such that their sum is 20 + a single digit, place the digit result on track 5, move left and place a 2 on track 1 (carry) and do not move the tape head.

3) Finally when the tape head encounters the $ marking the end of the tape, bring down any carry that might remain, e.g. copy track 1 cell 1 to track 5 cell 1.

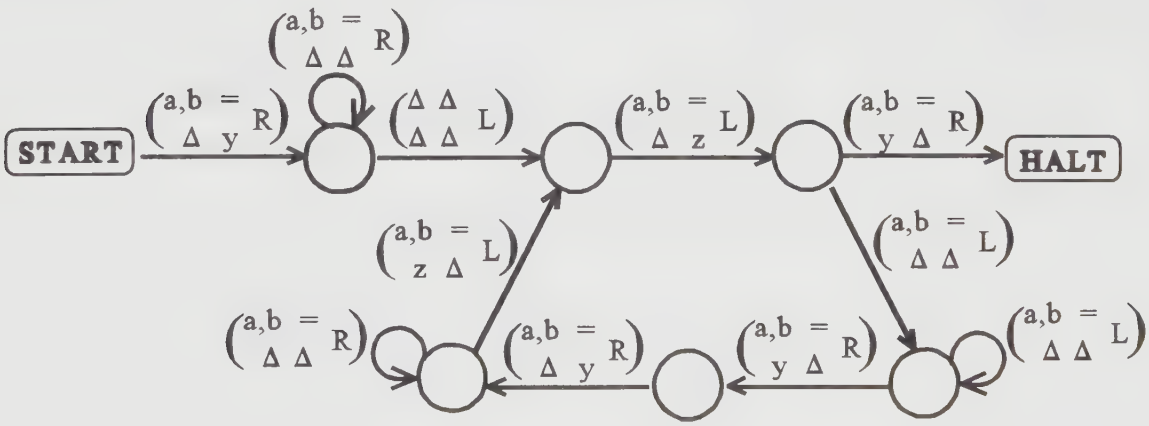
Chapter Twenty Two

(ii)



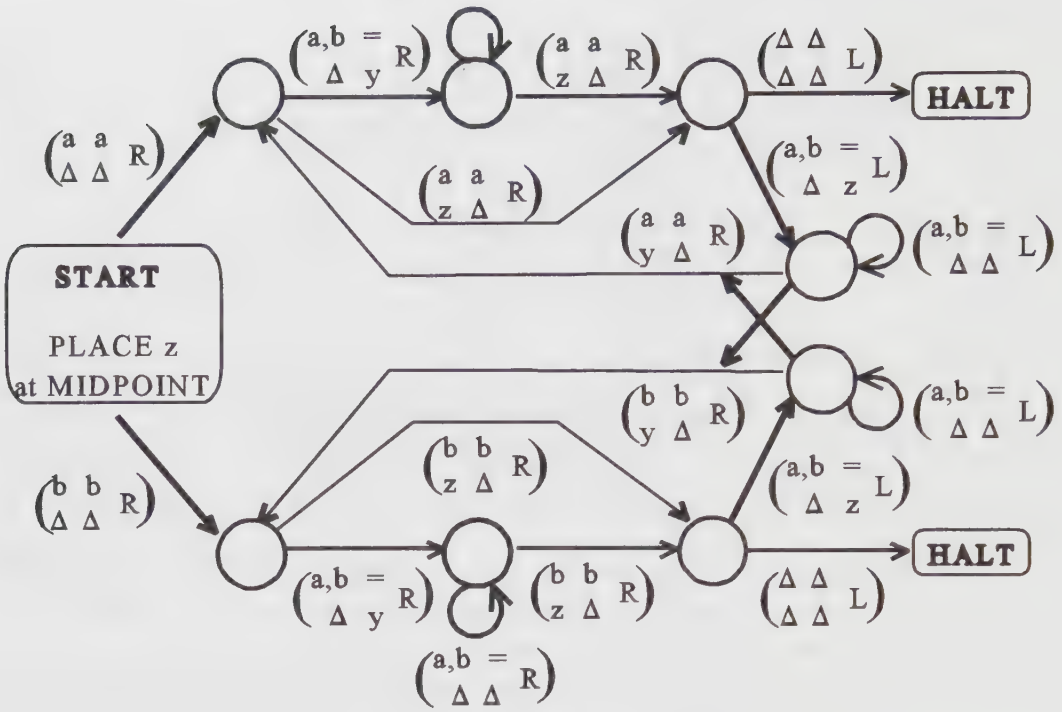
7. (Assume tracks 1 and 2 contain "$bit-string*" and tracks 3, 4, and 5 contain "$*"
   1) Copy bit-string from track 1 unto track 4 in between the $ and * markers.
   2) Read rightmost bit on track 2. (Scan right to left bitwise)
   3) If the bit = 1 then align data on tracks 3 and 4, add the two bit-strings placing sum on track 5. Copy track 5 onto track 3 (between the markers).
   4) Insert 0 to the left of * on track 4.
   5) If there is more data on track 2, GOTO step 2.

8. (i)



(ii)

Chapter Twenty Two



9. (i) For inserting or deleting from track 2, each instruction takes the form $\begin{pmatrix} \text{any}, = \\ \text{---}, \_, \_ \\ \text{any}, = \end{pmatrix}$

Using the notation (state, read, write, move, state) the following is a method for

INSERT d:

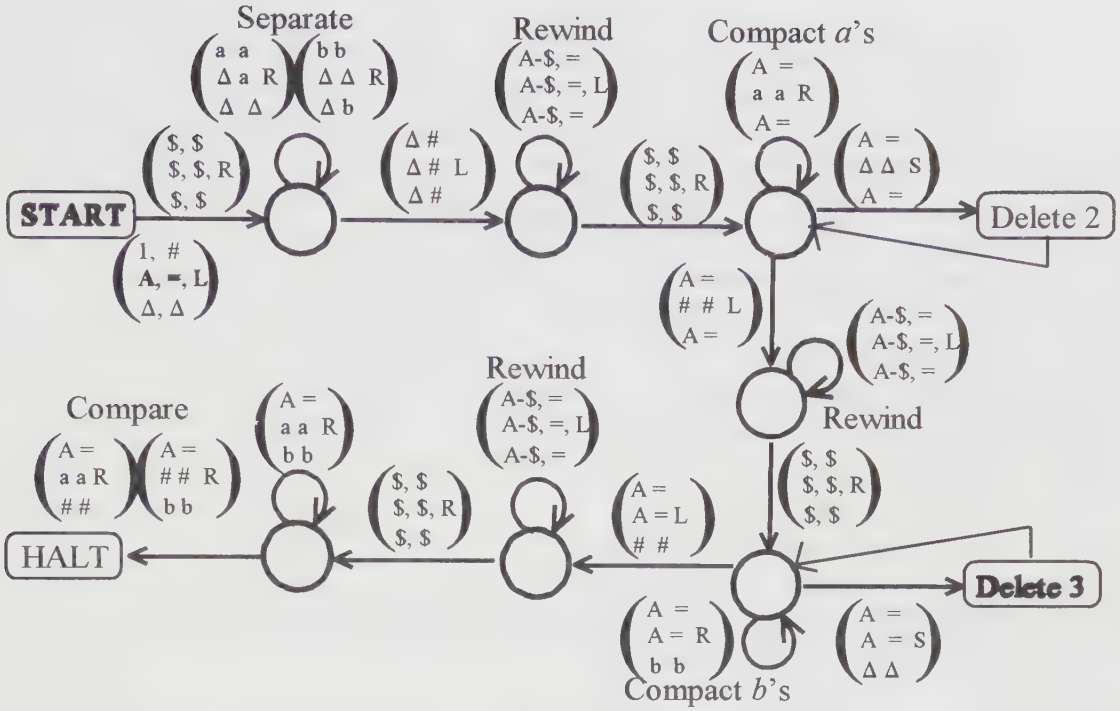
(1, a, d, R, 2) (1, b, d, R, 3) (1, Δ, d, R, HALT)
(2, a, a, R, 2) (2, b, a, R, 3) (2, Δ, a, R, HALT)
(3, a, b, R, 2) (3, b, b, R, 3) (3, Δ, b, R, HALT)

DELETE:

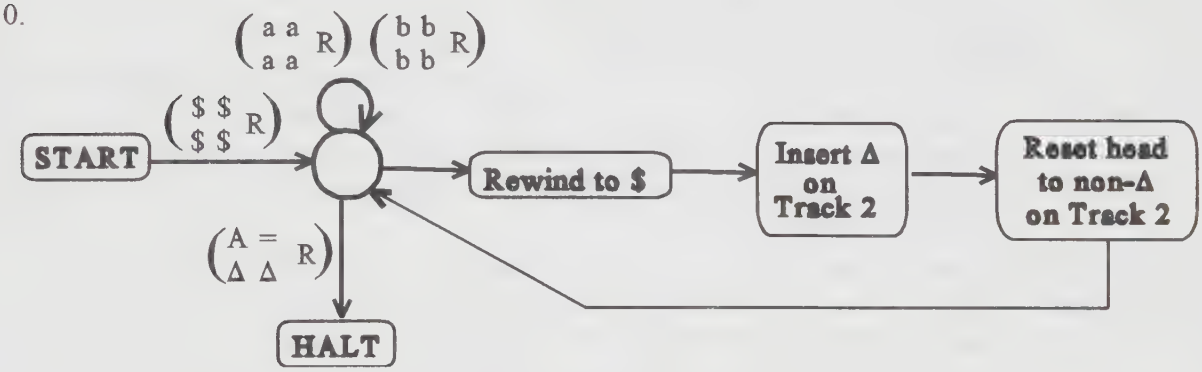
(1, a, #, R, 2) (1, b, #, R, 2) (1, Δ, Δ, R, HALT)
(2, a, a, R, 2) (2, b, b, R, 2) (2, Δ, Δ, L, 3)
(3, a, Δ, L, 4) (3, b, Δ, L, 5) (3, #, Δ, R, HALT)
(4, a, a, L, 4) (4, b, a, L, 5) (4, #, a, R, HALT)
(5, a, b, L, 4) (5, b, b, L, 5) (5, #, b, R, HALT)

(ii)

Chapter Twenty Two



10.



11. (i) Assume that cells 1 contain $'s. Read2 means on track 2, etc.

1) scan $, write2 #

2) read1 *a*, write1 A, GOTO 3
read1 *b*, write1 B, GOTO 4
read1Δ, goto5

3) reset track 2 to # - read2 #, write2 *a* - write2 # - reset track 1 to A - read1 A, write1 *a*, move right, GOTO 2.

4) reset track 2 to # - read2 #, write2 Δ - reset track 2 to $ - move right - read2 *a*, write2 # - reset track 1 to B - read1 B, write1 *b* - move right GOTO 2

5) reset track 2 to # - read2 #, write2 *a* - HALT

(ii) 5) reset track 2 to # - read2 #, write2 *a*
6) copy *a*'s from track 2 to track 3
7) insert *b* end of track 3
8) use modifies matching procedure from problem 10 to find number on track 2 among numbers on track 1
9) delete the number from the sequence including one *b*.
10) clear track 2
11) GOTO step 2

12. 1) place $0# on track 2
2) read1 *a*, write1 *b*, move right - GOTO 3
read1 *b*, move right - GOTO 2
read1 Δ - HALT
3) read2 #, move left - GOTO 4
read2 not #, move left - GOTO 3
4) read2 0, write 1, move right - GOTO 2
read2 1, write 0, move left - GOTO 4
read2 $, INSERT 1, move right - GOTO 2

13. (i) 1) initialize track 2 to "1" binary one
2) copy number from track 2 to track 3
3) compute square on track 4 using the multiplier from problem 7 and tracks 5 and 6 as working space
3) if number on track 1 is the same as number on track 4 then HALT (square)
4) if length of number on track 4 is greater than length of number on track 1 HALT (not a square)
5) increment number on track 2 GOTO step 2

(ii) 1) use procedure from problem 12 to convert number to binary.
2) use procedure from part one to test number.

14. Use a two track TM
1) Insert #s in the first cells.
2) If Read input = *a* then erase it and place an *$* on track 2; move right.
3) If Read input = *b* then overwrite it with B, find and erase one *$* on track 2, and find and erase B on track 1; move right.
3) If Read input = Δ then moving left, scan track 2 for *$* and HALT; if no *$* found crash.

Chapter Twenty Two

15. The tape looks like this:

$	0	1	-1	2	-2	3	-3	...
(i)	(ii)	(iii)	(iv)	(v)	(vi)	(vii)	(viii)	

A move RIGHT instruction (c,c′,R) on 2-way TM is equivalent to the following instructions on this deterministic TM:

\+ If in an odd numbered cell then move right twice i.e. (c, c′, R) (any, =, R)

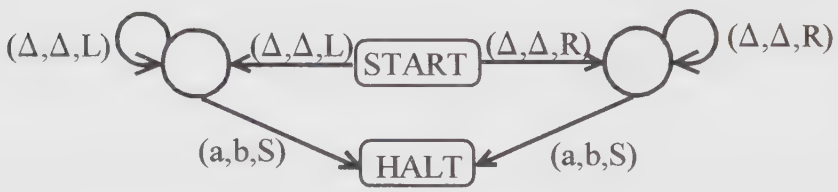
\- If in an even numbered cell then move left twice i.e. (c, c′, L) (any-$, =, L)
Be careful around the 'bend', instead use (c, c′, L) ($,$,R) (any,=,R)

Similarly the move LEFT instruction is simulated as follows:

\+ If in an odd numbered cell then move left twice i.e. (c, c′, L) (any, =, L)
Be careful around the 'bend', instead use (c, c′, L) (any, = L) ($,$,R)

\- If in an even numbered cell then move right twice i.e. (c, c′, R) (any, =, R)

16. (i)



(ii) As in problem 6, place two markers, *y* and *z*, on track 2 of the tape, in consecutive cells. Alternately, move *y* one cell to the left and *z* cell to the right until one of the markers in situated under a non-blank character.

(iii) The machine is as described above, except that the markers are on the same track so the checking for an input letter must be done before moving the marker each time.

17. (i) NPM = NTM = TM = PM
(ii) N2PDA = NTM = TM = 2PDA

18. (i) PM = TM no
2PDA = TM marginally easier because each stack on separate track.
(ii) They were incomplete without TM = NTM.

19. Using the Myhill Nerode theorem, since **r** is regular there is a finite number of equivalence classes for this language. Chop (**r**) certainly defines fewer equivalence classes. Hence the number of classes for Chop (**r**) is finite and Chop (**r**) is regular.

20. The difference between instructions (right)(left) and (left)(right) is that the net result is the suffix of one expression in the other as opposed to the prefix. We do not have a suffix language that is known to be regular. However, if use the transpose if the strings then take the correct prefix and transpose again then we have essentially obtained the suffix. We

already know that the Transpose language and Prefix language of regular languages are also regular. So the agglomerated instruction is

$$(T\,[\,\mathrm{Pref}(\,T\,[\,\mathbf{r_2(a+b)}\,]\,\text{in}\,T\,[\,\mathbf{(a+b)r_1}\,]\,)\,],\,L),$$
$$(T\,[\,\mathrm{Pref}(\,\mathrm{Chop}(\,T\,[\,\mathbf{(a+b)r_1}\,]\,)\,\text{in}\,T\,[\,\mathbf{r_2}\,]\,)\,],\,R),$$
$$(\Lambda,\,S)$$

Chapter 23

In the previous edition this was two different chapters. The amalgamation has focused attention on the r.e. languages as a class with questions similar to those answered or considered for regular languages and CFLs. I think I previously made too much of the details of the decoding process. Good programmers and readers of this text up to this point should be able to write TM code to do most of the tasks required without my help.

The encoding of TMs for the purpose of constructing the languages **ALAN** and **MATHISON** is another example of where we have started with a basically pictorial description of a machine and then later had to convert it into and algebraic equivalent in order to establish some results. This again does not mean that it would have been superior to have started with the algebraic formulation in place of the pictorial one originally.

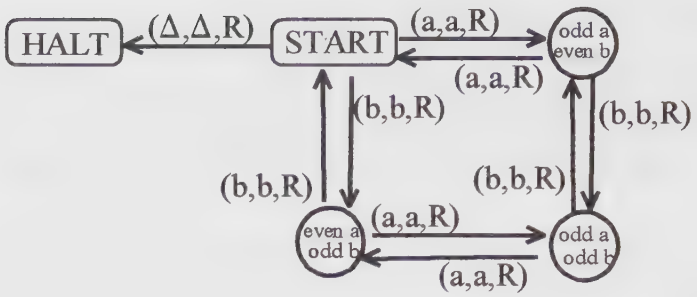
The existence of the loop set, and in fact its inevitability for certain languages not just certain machines, is a point to dwell on since it is the foreshadowing of the Halting Problem. The sooner the ineluctability of indecisiveness is appreciated as an essential of the language and not a fault of the programmer, the better.

We have avoided the standard black-boxing of TMs and tying them together with mystery strings such as "feed the output of this into that", or "run it on both machines" and instead give explicit illustrations of how this can be done. I find that more satisfying. After the details are covered an overall view of what is going on should emerge (as in Kafka's Penal Colony).

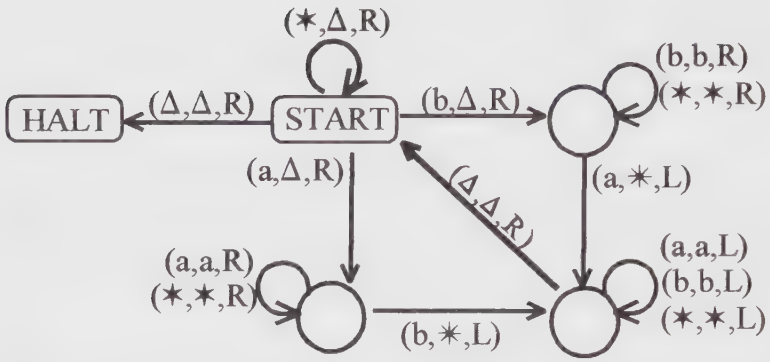
The significance of the non-r.e. language should not be minimized since it paradoxically undercuts the promised universality of the TM. The Halting Problem should be presented as a problem in two senses. I hope the machine LAMBDA is clear. One could, of course, write it out as an example for some particular word if one thought that would increase any student's understanding.

Chapter Twenty Three

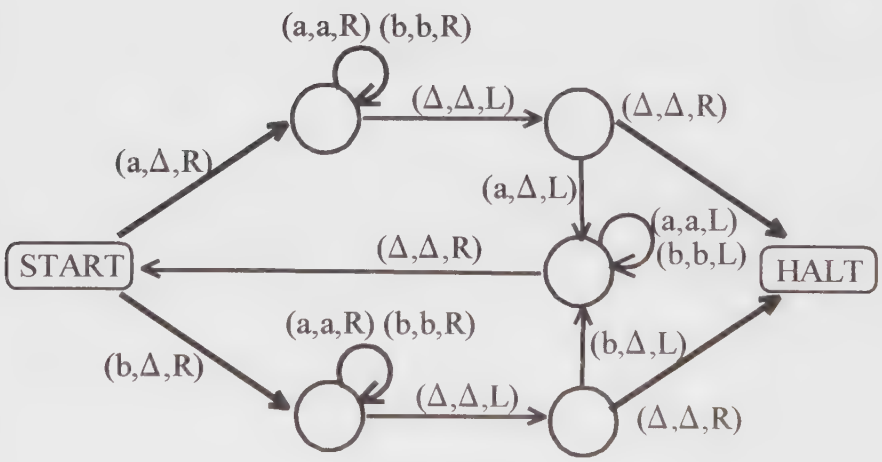
1. (i)



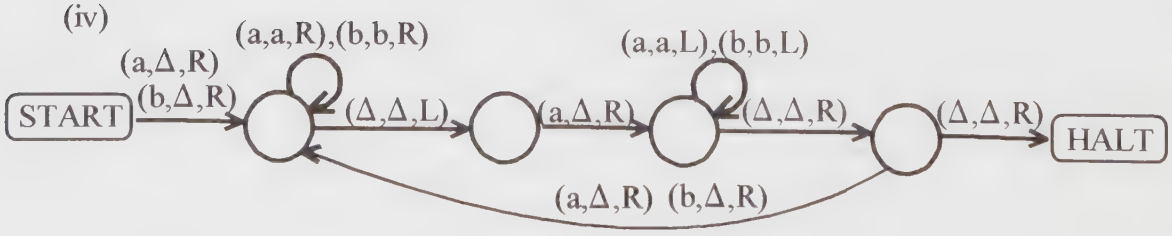
(ii)



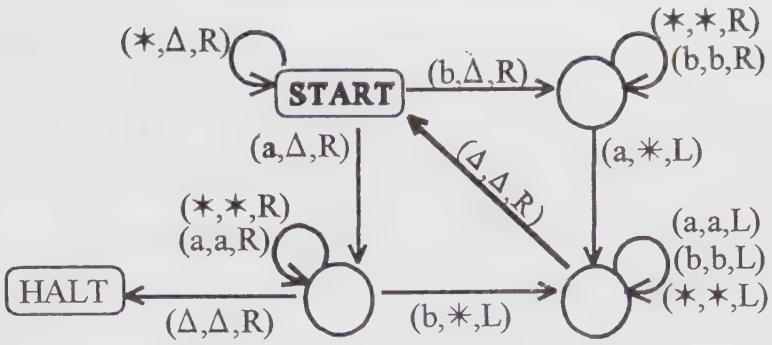
(iii)



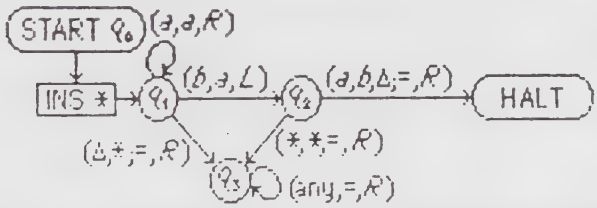
(iv)



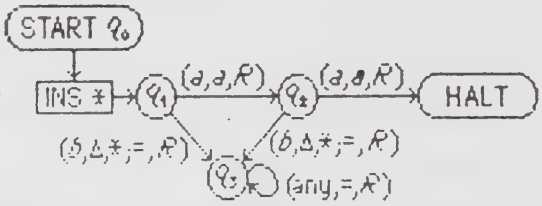
(v)



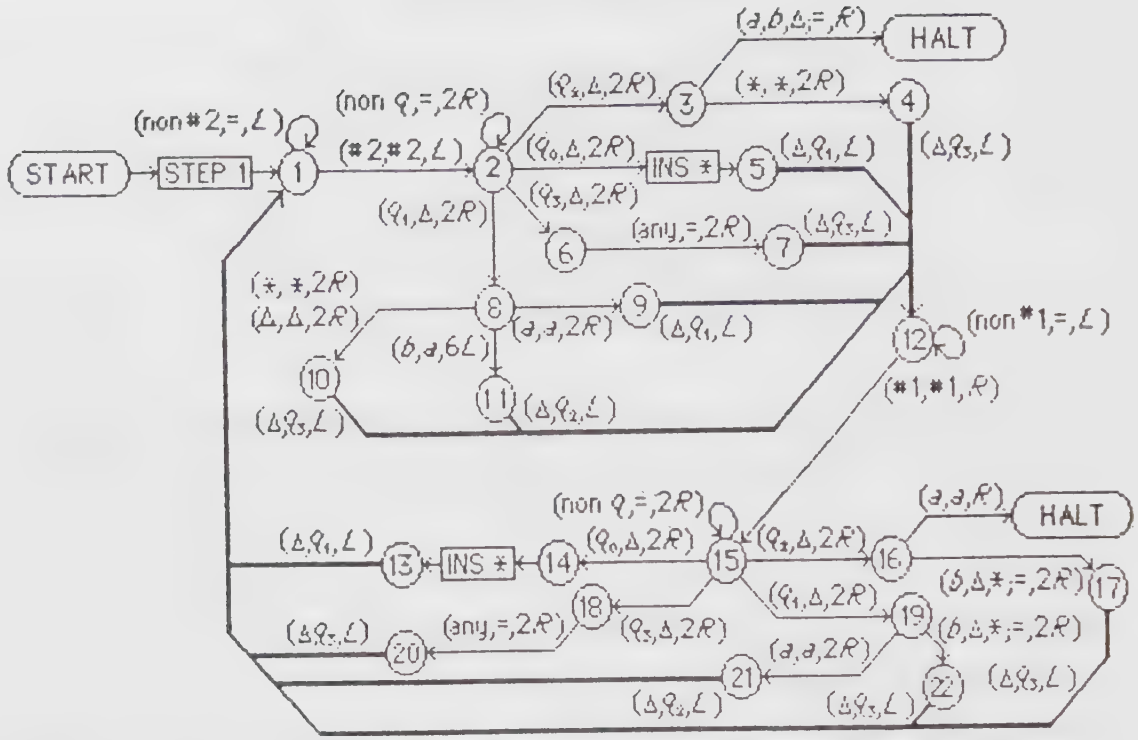
2. accept(T_1) is the language of the regular expression $\mathbf{aa^*b(a+b)^*}$
   reject(T_1) is the language of the regular expression $\mathbf{a^* + b(a+b)^*}$
   loop(T_1) is empty
3. accept(T_2) is the language of the regular expression $\mathbf{aa(a+b)^*}$
   reject(T_2) is the language of the regular expression
   $\mathbf{(ab+ba+bb)(a+b)^*}$
   loop(T_2) is empty
4. T_1, modified



T_2, modified



T_3: remember that the INSERT subroutine must also be run in alternating steps on the two machines as described in 12, above.



5.(i) We use the modifications of Problems 12 and 14. The TAPE after Step 1 is $*1*2\,q_0\,q_0\alpha\alpha\Delta\Delta\ldots$. The INSERT routines (after states 2 and 14) leave both T_1 and T_2 in q_1 and cell i symbols in the first data cells. These traces assume this preprocessing has been done.

(i) 1 $*1*2\Delta\Delta^{**}\,q_1\,q_1\underline{\Delta}\Delta\Delta\ldots$ 1 $*1*\underline{2}\Delta\Delta^{**}\,q_1\,q_1\Delta\Delta\Delta$ 2 $*\underline{1}*2\Delta\Delta^{**}\,q_1\,q_1\Delta\Delta\Delta$
2 $*1*2\Delta\Delta^{**}\,\underline{q_1}\,q_1\Delta\Delta\Delta$ 8 $*1*2\Delta\Delta^{**}\Delta\,q_1\underline{\Delta}\Delta\Delta$ 10 $*1*2\,\Delta\Delta^{**}\Delta\,q_1\Delta\underline{\Delta}\Delta$
12 $*1*2\Delta\Delta^{**}\Delta\,q_1\Delta\underline{\Delta}\,q_3\Delta\ldots$ 12 $*\underline{1}*2\Delta\Delta^{**}\Delta\,q_1\Delta\Delta\,q_3\Delta$ 15 $*1*\underline{2}\Delta\Delta^{**}\Delta\,q_1\Delta\Delta\,q_3\Delta$
15 $*1*2\Delta\Delta^{**}\Delta\,\underline{q_1}\Delta\Delta\,q_3\Delta$ 19 $*1*2\Delta\Delta^{**}\Delta\Delta\Delta\underline{\Delta}\,q_3\Delta$ 22 $*1*2\Delta\Delta^{**}\Delta\Delta\Delta\,q_3\underline{\Delta}\Delta\Delta$
1 $*1*2\Delta\Delta^{**}\Delta\Delta\Delta\,\underline{q_3}\,q_3\Delta\Delta\Delta$ ... (set for infinite loop)

(ii) 1 $*1*2\Delta\Delta^{**}\,q_1\,q_1\,b\underline{b}\Delta\Delta\ldots$ 1 $*1*\underline{2}\Delta\Delta^{**}\,q_1\,q_1\,b\,b\Delta\Delta$ 2 $*\underline{1}*2\Delta\Delta^{**}\,q_1\,q_1\,b\,b\Delta\Delta$
2 $*1*2\Delta\Delta^{**}\,\underline{q_1}\,q_1\,b\,b\Delta\Delta$ 7 $*1*2\Delta\Delta^{**}\Delta\,q_1\,\underline{b}\,b\Delta\Delta$ 11 $*1*2\underline{\Delta}\Delta^{**}\Delta\,q_1\,a\,b\Delta\Delta$
12 $*1*\underline{2}\,q_2\Delta^{**}\Delta\,q_1\,a\,b\Delta\Delta$ 12 $*\underline{1}*2\,q_2\Delta^{**}\Delta\,q_1\,a\,b\Delta\Delta$ 15 $*1*\underline{2}\,q_2\Delta^{**}\Delta\,q_1\,a\,b\Delta\Delta$
...15 $*1*2\,q_2\Delta^{**}\Delta\,\underline{q_1}\,a\,b\Delta\Delta$ 19 $*1*2\,q_2\Delta^{**}\Delta\Delta\,a\,\underline{b}\Delta\Delta$ 1 $*1*2\,q_2\Delta^{**}\Delta\Delta\,a\,b\underline{\Delta}\,q_3$
1 $*1*\underline{2}\,q_2\Delta^{**}\Delta\Delta\,a\,b\Delta\,q_3$ 2 $*\underline{1}*2\,q_2\Delta^{**}\Delta\Delta\,a\,b\Delta\,q_3$ 2 $*1*2\,\underline{q_2}\Delta^{**}\Delta\Delta\,a\,b\Delta\,q_3$
3 $*1*2\Delta\Delta^{**}\Delta\Delta\,a\,b\Delta\,q_3$ 4 $*1*2\Delta\Delta^{**}\,q_3\Delta\,a\,b\Delta\,q_3$ ... (set for infinite loop)

Chapter Twenty Three

(iii)
1 •1•2ΔΔ** q_1 q_1 *a a* Δ *a a* Δ *b b* ΔΔ ...
2 •1•2ΔΔ** q_1 q_1 *a a* Δ *a a* Δ *b b* ΔΔ ...
8 •1•2ΔΔ**Δ q_1 *a a* ΔΔ *a a* Δ *b b* ΔΔ
12 •1•2ΔΔ**Δ q_1 *a a* q_1Δ *a a* Δ *b b* ΔΔ ...
15 •1•2ΔΔ**Δ q_1 *a a* q_1Δ *a a* Δ *b b* ΔΔ ...
19 •1•2ΔΔ**ΔΔ *a a* q_1Δ *a a* Δ *b b* ΔΔ
1 •1•2ΔΔ**ΔΔ *a a* q_1 q_2 *a a* Δ *b b* ΔΔ ...
2 •1•2ΔΔ**ΔΔ *a a* q_1 q_2 *a a* Δ *b b* ΔΔ ...
8 •1•2ΔΔ**ΔΔ *a a* Δ q_2 *a a* Δ *b b* ΔΔ
12 •1•2ΔΔ**ΔΔ *a a* Δ q_2 *a a* q_1Δ *b b* ΔΔ ...
15 •1•2ΔΔ**ΔΔ *a a* Δ q_2 *a a* q_1Δ *b b* ΔΔ ...
16 •1•2ΔΔ**ΔΔ *a a* ΔΔ *a a* q_1Δ *b b* ΔΔ

1 •1•2ΔΔ** q_1 q_1 *a a* Δ *b b* ΔΔ
2 •1•2ΔΔ** q_1 q_1 *a a* Δ *b b* ΔΔ
9 •1•2ΔΔ**Δ q_1 *a a* ΔΔ *a a* Δ *b b* ΔΔ
12 •1•2ΔΔ**Δ q_1 *a a* q_1Δ *a a* Δ *b b* ΔΔ
15 •1•2ΔΔ**Δ q_1 *a a* q_1Δ *a a* Δ *b b* ΔΔ
21 •1•2ΔΔ**ΔΔ *a a* q_1Δ *a a* Δ *b b* ΔΔ
1 •1•2ΔΔ**ΔΔ *a a* q_1 q_2 *a a* Δ *b b* ΔΔ
2 •1•2ΔΔ**ΔΔ *a a* q_1 q_2 *a a* Δ *b b* ΔΔ
9 •1•2ΔΔ**ΔΔ *a a* Δ q_2 *a a* ΔΔ *b b* ΔΔ
12 •1•2ΔΔ**ΔΔ *a a* Δ q_2 *a a* q_1Δ *b b* ΔΔ
15 •1•2ΔΔ**ΔΔ *a a* Δ q_2 *a a* q_1Δ *b b* ΔΔ
HALT

(iv)
1 •1•2ΔΔ** q_1 q_1 *a a* Δ *b b* ΔΔ ...
2 •1•2ΔΔ** q_1 q_1 *a a* Δ *b b* ΔΔ ...
8 •1•2ΔΔ**Δ q_1 *a a* ΔΔ *b b* ΔΔ
12 •1•2ΔΔ**Δ q_1 *a a* q_1Δ *b b* ΔΔ ...
15 •1•2ΔΔ**Δ q_1 *a a* q_1Δ *b b* ΔΔ ...
19 •1•2ΔΔ**ΔΔ *a a* q_1Δ *b b* ΔΔ
1 •1•2ΔΔ**ΔΔ *a a* q_1 q_2 *b b* ΔΔ ...
2 •1•2ΔΔ**ΔΔ *a a* q_1 q_2 *b b* ΔΔ ...
8 •1•2ΔΔ**ΔΔ *a a* Δ q_2 *b b* ΔΔ
12 •1•2ΔΔ** q_2Δ *a a* Δ q_2 *b b* ΔΔ ...
15 •1•2ΔΔ** q_2Δ *a a* Δ q_2 *b b* ΔΔ ...
16 •1•2ΔΔ** q_2Δ *a a* ΔΔ *b b* ΔΔ

1 •1•2ΔΔ** q_1 q_1 *a a* Δ *b b* ΔΔ
2 •1•2ΔΔ** q_1 q_1 *a a* Δ *b b* ΔΔ
9 •1•2ΔΔ**Δ q_1 *a a* ΔΔ *b b* ΔΔ
12 •1•2ΔΔ**Δ q_1 *a a* q_1Δ *b b* ΔΔ
15 •1•2ΔΔ**Δ q_1 *a a* q_1Δ *b b* ΔΔ
21 •1•2ΔΔ**ΔΔ *a a* q_1Δ *b b* ΔΔ
1 •1•2ΔΔ**ΔΔ *a a* q_1 q_2 *b b* ΔΔ
2 •1•2ΔΔ**ΔΔ *a a* q_1 q_2 *b b* ΔΔ
11 •1•2ΔΔ**ΔΔ *a a* Δ q_2 *b b* ΔΔ
12 •1•2ΔΔ** q_2Δ *a a* Δ q_2 *b b* ΔΔ
15 •1•2ΔΔ** q_2Δ *a a* Δ q_2 *b b* ΔΔ
17 •1•2ΔΔ** q_2Δ *a a* ΔΔ *b b* ΔΔ

Chapter Twenty Three

6. Use the algorithm in Chapter 19 to convert an FA into a TM. The - state becomes START; a transition labeled *a* becomes an instruction (a,Δ,R); a transition labeled *b* becomes an instruction (b,Δ,R); from every + state add the instruction (Δ,Δ,R) on an edge to ACCEPT; from every non+ state add the instruction (Δ,Δ,R) on an edge to REJECT. Since every instruction cause a move right, eventually the input string will be exhausted and when the first blank cell is encountered, the string is either accepted or rejected.

7. Yes. Don't worry about nondeterminism: Make a TM that performs the CYK algorithm, taking a CFG and a target string as input and accepting the string if it can be generated by the grammar and rejecting it otherwise. The CYK algorithm is a decision procedure for membership for CFL's; no input causes it to loop indefinitely.

8. Recall Theorem 60 which states that if a language and its complement are both recursively enumerable then they are both recursive. L is the complement of M∪N and M∪N is r.e. (Theorem 62). Hence L is recursive. By symmetry M and N are also recursive.

9. Notation: R = recursive RE = r.e. but not recursive NRE = not r.e.
Consider the following table of apparent possibilities:

	1	2	3	2	4	5	3	5	6
L	R	RE	NRE	R	RE	NRE	R	RE	NRE
L′	R	R	R	RE	RE	RE	NRE	NRE	NRE

Equivalent cases have been given the same number. Case 1 is (i), case 6 is (ii) and case 5 is (iii) all as given. Cases 2 and 3 are impossible by Theorem 60. Case 4 is impossible by Theorem 61.

10. (i) Let R_1 be the machine that accepts L_1 and R_2 the machine for L_2 then we construct R_3 by the algorithm of Theorem 61. That is simulate running the two TM's alternately on the one input string. If the string was accepted by either machine then it will be accepted by R_3. Accept $(R_3) = L_1 + L_2$. $L(R_3)$ is recursive because loop (R_1) and loop (R_2) were each ϕ therefore loop$(R_3) = \phi$. Reject (R_3) = reject $(R_1) \cap$ reject (R_2) + reject $(R_1) \cap$ loop (R_2) + loop $(R_1) \cap$ reject $(R_2) = L_1' \cap L_2'$.

(ii) Using DeMorgan's Law, we can conclude that the intersection language is recursive using the facts already established that complements (in Theorem 60) and union of two recursive languages (as shown above) are recursive. If one wanted to construct the machine, it would be similar to the union machine; simulate testing a string on both machines simultaneously in alternation. Only difference is that once one machine accepts the string then the simulator must continue to test the string on the other machine without forgetting that if and when this second machine (with longer processing) accepts the simulator accepts and HALTS.

Chapter Twenty Three

11. If loop(T) was finite then the machine could be modified to check for those particular set of strings and reject them. The result would be an empty loop set and the language would be recursive. Since the language is specifically not recursive the loop language must be infinite and moreover it cannot be regular.

12. Construct a TM that accepts the product of two r.e. languages. Instead of testing one input string on two machines, we need to try all possible ways of dividing the input into two strings testing the first substring on machine for L_1 and the latter substring on the machine for L_2. (There are $n+1$ ways of dividing the input, where n is the length of the input.) Again we must use sufficient interlacing to avoid getting stuck in loop before exhausting all choices. Alternatively we can simplify the solution by using a nondeterministic machine to magically divide the word into two in the most favorable way.

    For product the division was simply into two parts, but for Kleene closure there can be any number of parts. The complexity using the first method would increase dramatically because of the multiple nesting necessary. However using a similar nondeterministic machine, Kleene closure can be implemented easily, by magically dividing the string into favorable parts.

13. Spaces inserted for legibility.

    (i) *abababab ababbabab abaabaaaba* **$b^+a(a+b)^*$** ALAN

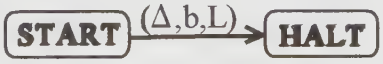
    (ii) *abaaabaabab aaababab bba aaabaabbaaab* **a** ALAN

    (iii) *abaaabaabbb abaaaababbbb aaabaaabaaaab aaabaabbaaab aaabaaaababaab aaaabaaaababab aaaabaabbaabb aaaabaaabaaabb* **$(a+b)^+$**

    (iv) *abaaabaaaab abaaababab aaababaaaab aaababababb aaabaabbabab*
    (Strings of odd length) ALAN

    (v) *ababaaaba abaaababaab aaabaaabbabab aaababababaa* ϕ ALAN

    (vi) *abaabbabab abaaabaabab abaaababbab aaabaaabaabab aaabaaababbab aaababbaaaa* **$(a+b)^*$**

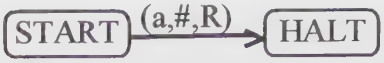
14. Yes, consider the palindrome *abaabbaaba* for the TM 

15. (i) ALAN 

    (ii) ALAN 

Chapter Twenty Three

(iii) MATHISON

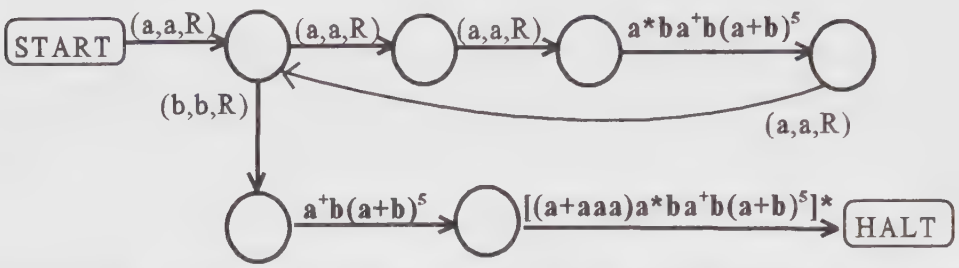


(iv) MATHISON

(v) ALAN

(vi) ALAN, this string is not in CWL.

16.



We use regular expressions as edge lables as a shorthand for inserting their FAs in TM form. There is a simple regular expression for CWL. However we must be certain that the word has the two properties that make it a TM: There must be a START state and there may not be any edges emanating from HALT (state 2). The above machine accounts for these things. Also note that this machine accepts encoded nondeterministic TM's.

17. Blue paint would follow the simple path from START to HALT and claim that the machine accepts some word(s); in this case **(a+b)(a+b)a**. However because of the abilities to move left and to write on the tape, the existence of a path is not sufficient proof of acceptance of a word. As we see, since the first edge writes a *b*, it can not possibly back up and read an *a* from that same cell.

18. Examine T_2; it accepts all strings if and only if T_1 accepts the particular string *w*. T_2 accepts all strings except *w* if and only if T_1 does not accept *w*. Imagine that there is a machine ACCEPTALL which takes an encoded TM as input and outputs yes or no. Then use T_2 to be the input of ACCEPTALL. The effect is a machine that will decide if T_1 accepts *w*. This is the Halting Problem which is unsolvable. Hence there is no such decision procedure for ACCEPTALL.

19. Let T_2 be a machine such that accept (T_2) = **(a+b)***, then this EQUIVALENCE machine reduces to ACCEPTALL of the previous problem.

Chapter Twenty Three

20. If the word "heterothetic" is homothetic - it really does describe itself. Of course, because it is homothetic it also doesn't describe itself; it describes its opposite. So "heterothetic" is also heterothetic. But because it is heterothetic, it becomes homothetic. And so on.

Chapter 24

The proof by Chomsky and Schutzenberger (of whose recent passing I have just sadly learned) that type O = TM, I consider to be a triumph of mankind equal to any work of art or science. Any who skip this material are risking my personal wrath.

It is always a delight to find a phrase-structure language that is not context-free. Unfortunately almost none of the examples is easy. Also the limited relevance for Computer Science (as opposed to mathematical logic or philosophy) disallows us to indulge in this protracted esoteric adventure.

New in this edition, but important for gratifying comprehensiveness as well as their important Computer Science application, is the discussion of context-sensitive languages. The next fifty years will see natural language processing as a dominant issue. Teachers who know some AI teaching students who want to know some AI will easily bring in the relevent applications.

How explicit one wants to be about the usefulness of the linear bound in lbas is a matter of choice. I considered it important for two reasons: it distinguishes a different type of machine for a different type of grammar, and because in answers the question of decidability of membership, which distinguishes CSLs from r.e. languages.

The product and Kleene closure of r.e. languages could have been done in the previous chapter but that would have deprived us of the pleasure of cooking the "obvious" grammatical proofs. And since this is book about learning what a proof is and ISN'T (as well as about pleasure) we delayed this presentation till here.

Chapter Twenty Four

1.(i) $S \Rightarrow AB\underline{S} \Rightarrow ABAB\underline{S} \Rightarrow AB\underline{AB} = ABBA \Rightarrow \ldots \Rightarrow abba$

(ii) $S \Rightarrow AB\underline{S} \Rightarrow ABAB\underline{S} \Rightarrow ABABAB\underline{S} \Rightarrow ABABABAB\underline{S}$
$\Rightarrow ABABABABAB\underline{S} \Rightarrow ABABABABAB = \underline{AB}ABABABAB$
$\Rightarrow BA\underline{AB}ABABAB \Rightarrow BABAAB\underline{AB}AB \Rightarrow BABAABBA\underline{AB}$
$\Rightarrow BABAABB\underline{AB}A \Rightarrow BABAABBBAA \Rightarrow \ldots \Rightarrow babaabbbaa$

2. PROD 1 (a) expands the working string by one *A* and one *B*. PROD 1 (b) eliminates the Start symbol *S*. PROD's 2 and 3 work on pairs of *A*'s and *B*s, and do not change the number of either. Therefore, we must always have the same number of *A*'s and *B*s in any working string. The terminating productions are simple replacements of *A* with *a* and *B* with *b*; since we had the same number of *A*'s and *B*s we must have the same number of *a*'s and *b*'s in any strings that we generate.

3. We offer an algorithm to generate any string having the same number of *a*'s as *b*'s. Apply PROD 1 (a) half as many times as there are letters in the desired string, then apply PROD 1 (b). (This gives us a working string of the right number *A*'s and *B*s in alternation.) If the first letter of the desired string is *a*, use PROD 4 on the first *A* in the string, if it is *b*, use PROD 2 and then apply PROD 5 to the leftmost *B*. Thereafter, if the leftmost capital letter generates the next desired terminal in the string apply PROD 4 or 5, otherwise, apply PROD 2 or 3 (only one will be possible) to the leftmost dissimilar nonterminal pair until the leftmost capital does generate the desired letter, and then apply PROD 4 or 5 to it. We know that we began this step having the right number each of *A*'s and *B*s. Any leftmost position wrongly occupied by, say, an *A* means that there must be a *B* somewhere to the right that we can use to fill the spot. If it is the very next symbol, we do a simple swap. However, wherever it falls, there must be an *A* immediately to its left, so we can apply PROD 2 as often as we must until we have applied it to that leftmost out-of-position *A*. An analogous argument applies if the leftmost out-of-position symbol is *B*.
Every time we place a new symbol in its proper position we

reduce the number of symbols remaining. Since we must have had a finite number of symbols at the end of the first step, the procedure is finite.

4.(i) $S \rightarrow ABS \mid A$
$AB \rightarrow BA$
$BA \rightarrow AB$
$A \rightarrow aA \mid a$
$B \rightarrow b$
The grammar of Chapter 13, Problem 19 will also serve.

(ii) $S \rightarrow S_1 \mid S_2$

$S_1 \rightarrow A_1A_1B_1S_1 \mid a$ $\quad$ $S_2 \rightarrow B_2B_2A_2S_2 \mid b$

$A_1B_1 \rightarrow B_1A_1$ $\quad$ $A_2B_2 \rightarrow B_2A_2$

$B_1A_1 \rightarrow A_1B_1$ $\quad$ $B_2A_2 \rightarrow A_2B_2$

$A_1 \rightarrow aA_1 \mid a$ $\quad$ $A_2 \rightarrow a$

$B_1 \rightarrow b$ $\quad$ $B_2 \rightarrow bB_2 \mid b$

(iii) EQUAL is recursive. The TM built in Chapter 24 (pp. 569-573) accepts or rejects all strings. Or, by Chapter 28, Problem 11, all CFL's are recursive, and EQUAL was shown to be context free in Chapter 13.

5.(i) $S \Rightarrow ABC\underline{S} \Rightarrow ABCABC\underline{S} \Rightarrow AB\underline{C}ABC\Lambda \Rightarrow ABA\underline{CB}C \Rightarrow ABABCC$
$\Rightarrow \ldots \Rightarrow abacc$

(ii) $S \Rightarrow ABC\underline{S} \Rightarrow ABCABC\underline{S} \Rightarrow ABCABCABC\underline{S} \Rightarrow ABCABCABC\Lambda$
$= A\underline{BC}ABCABC \rightarrow \underline{AC}BABCABC \rightarrow C\underline{AB}ABCABC$
$\rightarrow CBAABCA\underline{BC} \rightarrow CBAABC\underline{AC}B \rightarrow CBAABCC\underline{AB}$
$\Rightarrow CBAABCCBA \Rightarrow \ldots \Rightarrow cbaabccba$

6. The argument is similar to that of Problem 2: We generate the terminals a, b and c from the simple nonterminals A, B and C respectively; we always add one of each or none at all, so we can never have a working string with unequal numbers of (A or a), (B or b) and (C or c), and therefore never derive a strings in which the number of each does not equal the numbers of each of the others.

7. Again, the argument is an extension of that of Problem 3. PRODS 2 and 4 say that if an A is out of place to the left, it can be moved right; PRODS 3 and 5 say the same for B, PRODS 6 and 7 for C. PRODS 5 and 7 say that if an A is out of place to the right it can be moved left; PRODS 2 and 6 say the same for B, PRODS 3 and 4 for C. Therefore, any of these letters (and they are the only ones we have to work with) can be moved either left or right past either of the others, so all permutations can be derived.

8. (i) $S \Rightarrow \underline{UV}X \Rightarrow \Lambda\underline{X} \Rightarrow \Lambda$

(ii) $S \Rightarrow \underline{UV}X \Rightarrow aU\underline{YX} \Rightarrow a\underline{UV}aX \Rightarrow a\Lambda\, a\underline{X} \Rightarrow a\Lambda\, a\Lambda = aa$

(iii) $S \Rightarrow \underline{UV}X \Rightarrow bU\underline{ZX} \Rightarrow b\underline{UV}bX \Rightarrow b\Lambda\, b\underline{X} \Rightarrow b\Lambda\, b\Lambda = bb$

(iv) $S \Rightarrow \underline{UV}X \Rightarrow aU\underline{YX} \Rightarrow a\underline{UV}aX \Rightarrow abU\underline{Z}aX \Rightarrow abUa\underline{ZX}$
$\Rightarrow abU\underline{a}VbX \Rightarrow ab\underline{UV}abX \Rightarrow ab\Lambda\, ab\underline{X} \Rightarrow ab\Lambda\, ab\Lambda = abab$

9. We can make the left w of our string by applying PROD 2 if we want the next letter to be a and PROD 3 if we want it to be b. (If the string is already long enough we use PROD 10.) Until we reach this point we always have a UV in the working string when we are ready to choose the next letter, at the beginning from PROD 1, thereafter because PRODS 2 and 3 leave a U in the working string PRODS 4 and 5 return a V which is moved immediately to the right of the U by PRODS 12 and 13.

A working string with a Y or a Z in it has added the next letter to its first half but not yet added that letter to its second half. PRODS 2 and 3 introduce the new first half letter and the Y or Z for the second half. PRODS 4 and 5 add to the second half of the string the same letter that was added to the first half. Prods 6-9 move the Y or Z to the right where it can be expanded in combination with the final X. That expansion restores the V to the working string and PRODS 12 and 13 move the V left until it meets the U. This leaves the working string in the form $wUVwX$, which can either be expanded by repeating this process or terminated with PRODS 10 and 11, making it ww.

10. (i) $w\underline{UV}wX \overset{(10)}{\Rightarrow} w\Lambda\, wX \overset{(11)}{\Rightarrow} w\Lambda\, w\Lambda$. We have no other choice.

(ii) $wUVwX \overset{(11)}{\Rightarrow} wUVw\Lambda = wUVw \overset{(10)}{\Rightarrow} w\Lambda w = ww$

$\overset{(2)}{\Rightarrow} waUYw \Rightarrow \ldots \Rightarrow waUwY$

$\overset{(3)}{\Rightarrow} wbUZw \Rightarrow \ldots \Rightarrow wbUwZ$

We cannot use either the Y or the Z without a final X, nor can we use the U without a V to its right. These sequences do not produce words.

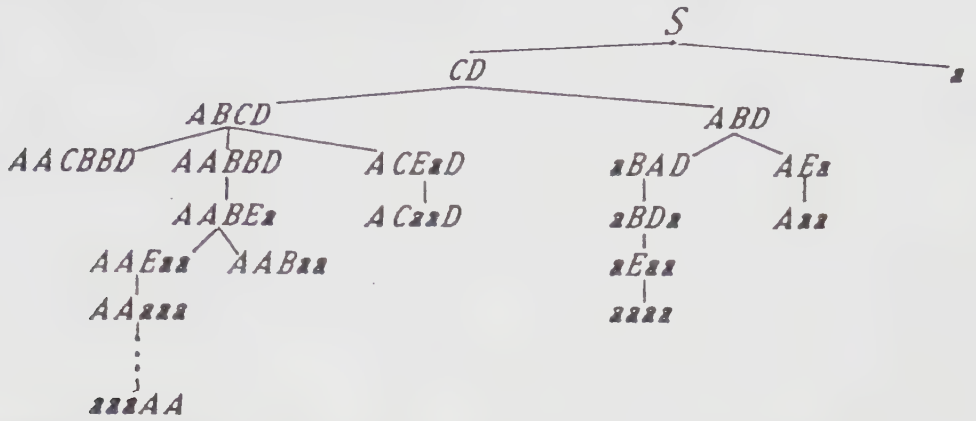
(iii) $wUVwX \overset{(2)}{\Rightarrow} waUYwX$ (That is, we have lengthened the first w by adding an a to its right.)

Consider the possibilities: PROD 1 needs an S, and we have none; PRODS 2, 3, 10, 12 and 13 need a V and we have none; PRODS 5, 8 and 9 need a Z and we have none. PROD 4 requires the Y adjacent to the X, and if our current w is not null, it is not. This leaves us with PRODS 6 and 7, and one of them must apply if w is not null. In fact, we find that we apply them until Y is the next to last symbol in the working string (adjacent to X) and then we apply PROD 4. This gives us a working string of the form $waUwVaX$. The only possible productions we can use are 11 , 12 and 13. If we use 11 we eliminate our final X and must either stop (by eventually eliminating the UV pair) or land in a nonterminating sequence. If we use 12 or 13 (depending on the last letter of w) we begin the process of moving the V to the left until it meets the U, resulting in a string of the form $waUVwaX$.

(iv) By the same token, if we apply PROD 3 the starting point is $wbUZwX$, which results in a nonterminating sequence (if we apply PROD 11 prematurely), the word $wbwb$ (if we move the Z to the right and then apply PROD 5, followed by PROD 11 and, eventually, PROD 10) or a working string of the form $wbUVwbX$, to be expanded further or terminated.

(v) Problem 11 establishes that any string of the form ww can be generated from this grammar. Problem 12 establishes that no other strings can be generated from it. Therefore the language generated by this grammar is exactly the set of all strings of the form ww, where w is any string of a's and b's; that is, DOUBLEWORD.

11(i) The nonterminals C and AB always expand the working string. We do not test them once the length of the string is 5.



(ii) $\underline{S} \overset{(2)}{\Rightarrow} \underline{C}D \overset{(3)}{\Rightarrow} A\underline{C}BD \overset{(4)}{\Rightarrow} A\underline{AB}BD \overset{(5)}{\Rightarrow} \underline{Aa}BABD$

$\overset{(6)}{\Rightarrow} a\underline{AB}ABD \overset{(5)}{\Rightarrow} aaBA\underline{AB}D \overset{(5)}{\Rightarrow} aaBA\underline{Aa}BAD$

$\overset{(6)}{\Rightarrow} aa\underline{Ba}ABAD \overset{(7)}{\Rightarrow} aaaB\underline{AB}AD \overset{(5)}{\Rightarrow} aaa\underline{Ba}BAAD$

$\overset{(7)}{\Rightarrow} aaaaBBA\underline{AD} \overset{(8)}{\Rightarrow} aaaaBB\underline{AD}a \overset{(8)}{\Rightarrow} aaaaB\underline{BD}aa$

$\overset{(9)}{\Rightarrow} aaaa\underline{BE}aaa \overset{(10)}{\Rightarrow} aaaa\underline{E}aaaa \overset{(11)}{\Rightarrow} aaaaaaaaa$

(iii) Consider PRODS 2, 3 and 4. By PROD 2, S gives a C part and a D part. PROD 3 allows the addition of an A to the left of the C and a B to its right and PROD 4 eliminates the C from the working string, replacing it by AB. In other words, for any n, to derive the string A^nB^nD, apply PROD 2 once, then apply PROD 3 n-1 times and then PROD 4.

(iv) PROD 5 is applied every time there is an A to the left of a B. It adds an a to the string and moves the A to the right of the B. (PRODS 6 and 7 move the newly introduced a leftward, so all upper case symbols are together at the right.) PROD 5 is applied exactly as many times as there are A's left of Bs. Since we started with exactly n A's to the left of exactly n Bs, it gives us n^2 a's in the string when every A is to the right of every B. The D that was introduced in PROD 2 has not been affected by any of these productions, so our working string is of the form

$$a^{n^2}B^nA^nD$$

(v) The working string can contain at most one D. For every A preceding it, PROD 8 gives us one a without removing the D from the string. Then PROD 9 introduces the new nonterminal E (at the first BD occurrence) and an a, and PROD 10 continues the process of giving one a for each B. We started with exactly n A's and n Bs. PROD 8 gives us n more a's, PROD 9 gives us one a and PROD 10 gives us $(n-1)$ a's, for a total of $2n$ a's. We also have the nonterminal E, which gives us a by PROD 11. This is $2n+1$ a's in addition to the n^2 already accounted for in (iii), for a total of (n^2+2n+1) a's, that is, $(n+1)^2$ of them.

(vi) PROD 1 gives us the word a, or a^{1^2}.

We have shown that application of PRODS 2-7 gives us a working string of the form $a^{n^2}B^nA^nD$, and that this string eventually gives us a word of the form $a^{(n+1)^2}$. Therefore, this grammar can produce any a^{n^2}.

As we have seen from the tree (Part (i)), if we try to apply PROD 9 before moving the A's to the right of the B's we cannot complete a word. So the only sequences that produce any words are those which give us a^{n^2} for some n.

12. This grammar generates the language $\{ab^na^n, n>0\}$. We get a single initial a from PROD 1. Then, by applying PROD 2(b) immediately, we get the word aba. If instead we apply PROD 2(a) we add no more a's to the left of the string, we keep a single XY part, add one b and a Z part. PROD 3 moves the b's left through the Z's. When the Z's are all to the right of the b's (just left of the final a) they can become a's, otherwise they do not terminate. Therefore the only words that can be formed are those with a solid clump of a's on the right preceded by a solid clump of b's, after a single initial a. Any such string can be formed: Apply PROD 1, then PROD 2(a) $n-1$ times, then apply PROD 2(b) and finally PRODS 3 and 4 to terminate. To change this to a grammar for $a^nb^na^n$ make PROD 2(a) read $XY \rightarrow aXYbA$.

13. (i) ab, $aabb$, $aaabbb$, $aaaabbbb$ (ii) (a^nb^n)

Chapter Twenty Four

14. Combine the context-sensitive grammars the same way we combined CFG. Ensure that the nonterminal symbols for each language are different. Let the start symbols be S_1 and S_2 respectively. Add the new productions $S \rightarrow S_1 \mid S_2$. Since there is a CSG that generates the union language, it is context-sensitive.

15. Again ensure that the symbols used for nonterminals in each grammar are different and add the new production $S \rightarrow S_1S_2$. The new grammar produces exactly the words of the product language.

16. We do not have a simple technique for combining the grammars however we can describe the TM that would accept the intersection of two languages. Using Theorem 79 that shows that membership is Turing decidable for type 1 languages, let T_1 be a TM that decides if a string is in one CSL and let T_2 be a membership TM for another CSL. Then construct a third TM which takes an input string and runs it on T_1 if the result is no then reject the string but if the result of T_1 is yes then run the string on T_2. Again if the result is negative then reject and if positive then accept; the string is a word in the intersection language.

17. Even more simple than problems 14 and 15, we can form the grammar for the Kleene closure language by adding the new productions $S \rightarrow SS \mid \Lambda$.

18. To construct a grammar for the reverse language, transpose each side of each production of the CSG. For example XYZ → BbaaaT becomes ZYX → TaaabB. The length condition is maintained and so the grammar is still context-sensitive.

19. (i) Convert the grammar in a way similar to the way we converted CFG to CNF. Except for productions which are already in good form, change all terminals to appropriate nonterminals. When necessary, add the corresponding $N \rightarrow t$. Now shorten strings on either side of the rules by introducing unique nonterminals to represent the remaining part. For example, XYZ → BaCD first becomes XYZ → BACD (adding $A \rightarrow a$ if it is not already a rule.) Then XYZ becomes XR_1 where $R_1 \rightarrow YZ$ and BACD becomes BR_2 where $R_2 \rightarrow AR_3$ and $R_3 \rightarrow CD$. The replacement productions of XYZ → BACD are:

$XR_1 \rightarrow BR_2$ $\quad$ $R_1 \rightarrow YZ$ $\quad$ $R_2 \rightarrow AR_3$ $\quad$ $R_3 \rightarrow CD$

(ii) Yes. Comparable to the CYK algorithm for CNF, use a variant for KNF where the various possibilities for each length are tested.

20. (i) Remember that the productions of the grammar are restricted in length such that the working string of any derivation of any word grows increasingly longer. Since there is no way of shortening the working string, at every point in the derivation the length of string must be less than or equal to the input if the word is to be derived at all.

(ii) Recall from the text that the format of the tape is $input$ _____.

Chapter 25

There is no doubt but that this material on recursive functions is anticlimactic, obvious, and tedious. Still, it is necessary to include it here in order to prepare the student for possible future investigations into more advanced topics. I wouldn't waste too much time on it (especially since the term is probably over already anyway) but skip straight to a quick trip through Church's Thesis (a must) and on to TMs as language generators.

The theorems about language generation in lexicographic order are not only important but tie together several key concepts form the course: what the problem in looping is all about, why algorithms have to have predictable stopping times, why nondeterminism can be simulated by determinism but at an important cost, and why doing many things at once is a necessity but not a complete cure.

Chapter Twenty Five

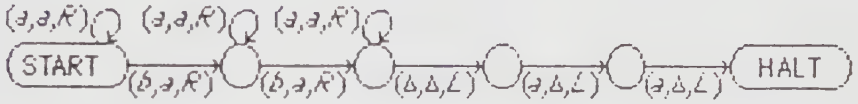
1. (i) START $\underline{a}aba \rightarrow$ START $a\underline{a}ba \rightarrow$ START $aa\underline{b}a \rightarrow$ 1 $aaaa \rightarrow$ 1 $aaaa\Delta$ $\rightarrow$ 2 $aaa\underline{a}\Delta \rightarrow$ 2 $aaaa\Delta\Delta \rightarrow$ HALT $aaa\underline{a}\Delta\Delta$

   (ii) START $\underline{a}ab \rightarrow$ START $a\underline{a}b \rightarrow$ START $aa\underline{b} \rightarrow$ 1 $aa\underline{a}\Delta \rightarrow$ 2 $aa\underline{a}\Delta$ $\rightarrow$ HALT $aa\underline{a}\Delta\Delta$

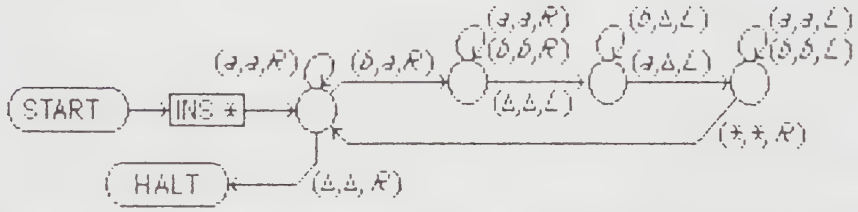
   (iii) START $\underline{b}aaa \rightarrow$ 1 $a\underline{a}aa \rightarrow$ 1 $aa\underline{a}a \rightarrow$ 1 $aaa\underline{a} \rightarrow$ 1 $aaa\underline{a}\Delta \rightarrow$ 2 $aaa\underline{a}\Delta$ $\rightarrow$ HALT $aa\underline{a}\Delta\Delta$

   (iv) START $\underline{b} \rightarrow$ 1 $\underline{a}\Delta \rightarrow$ 2 $\underline{a}\Delta \rightarrow$ 2 $\Delta\underline{\Delta} \rightarrow$ HALT $\Delta\Delta\underline{\Delta}$

2.(i)



(ii)



3. Use the approach described on pages 769-771 to add the second and third addends. Delete one $ and the zeros between where it was and the other $, and add the result of the first addition to the first addend by the same means.
4. STEP 1. Change the first blank after the second $ to *.

   STEP 2. Read the character immediately to the left of the second $. If it is $ go to STEP 5. If it is 0 delete it and go to STEP 4. If it is 1 delete it and go to STEP 3.

   STEP 3. In the (blank) cells immediately following the occupied cells, write as many *a*s as there are *'s after the second $. (That is, mark the leftmost unmarked *, move right to the first blank cell, write an *a*, move left to the first unmarked *, mark it, etc. Unmark the *'s.) Go to STEP 4.

   STEP 4. Double the number of *'s. (This can be done by the same method that was used to count *a*'s in STEP 3.) Go to STEP 2.

   STEP 5. Delete the two $'s and all *'s. The number of *a*'s equals the binary number represented by the original zeros and ones.

Chapter Twenty Five

5.(i) START $\underline{a}aabaa \to 1\ A\underline{a}abaa \to 1\ Aa\underline{a}baa \to 1\ Aaa\underline{b}aa$
$\to 2\ Aaab\underline{a}a \to 2\ Aaaba\underline{a} \to 2\ Aaabaa\underline{\Delta} \to 3\ Aaaba\underline{a}$
$\to 5\ Aaab\underline{a}\Delta \to 5\ Aaa\underline{b}aa\Delta \to 6\ Aa\underline{a}ba \to 7\ Aa\Delta\underline{b}a$
$\to 8\ Aa\Delta b\underline{a} \to 8\ Aa\Delta b\underline{\Delta} \to 3\ Aa\Delta b\underline{a} \to 5\ Aa\Delta\underline{b}\Delta$
$\to 6\ Aa\underline{\Delta}b\Delta \to 6\ A\underline{a}\Delta b\Delta \to 7\ A\Delta\Delta b\Delta \to 7\ A\Delta\Delta\underline{b}\Delta$
$\to 8\ A\Delta\Delta b\underline{\Delta} \to 3\ A\Delta\Delta\underline{b}\Delta \to 4\ A\Delta\underline{\Delta}\Delta\Delta \to 4\ A\underline{\Delta}\Delta\Delta\Delta$
$\to 4\ \underline{A}\Delta\Delta\Delta\Delta \to$ HALT $\underline{\Delta}\Delta\Delta\Delta\Delta$

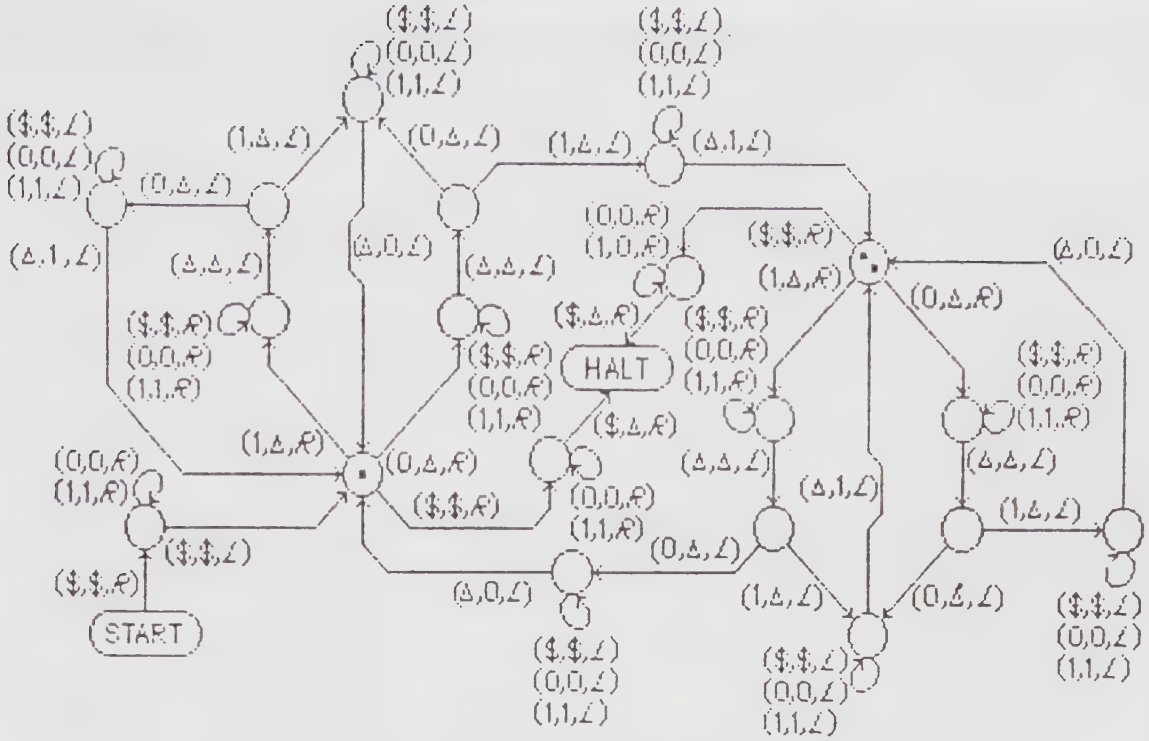
(ii) START $\underline{a}baaa \to 1\ A\underline{b}aaa \to 2\ Ab\underline{a}aa \to 2\ Aba\underline{a}a \to 2\ Abaa\underline{a}$
$\to 2\ Abaaa\underline{\Delta} \to 3\ Abaa\underline{a} \to 5\ Aba\underline{a}\Delta \to 5\ Ab\underline{a}aa \to 5\ A\underline{b}aa$
$\to 6\ \underline{A}baa \to 9\ \Delta\underline{b}aa \to 10\ \Delta\Delta\underline{a}a \to 10\ \Delta\Delta\Delta\underline{a} \to 10\ \Delta\Delta\Delta\Delta\underline{\Delta}$
$\to$ HALT $\Delta\Delta\Delta\Delta\Delta\underline{\Delta}$

(iii) START $\underline{b}aa$ CRASH

(iv) START $\underline{a}aab \to 1\ A\underline{a}ab \to 1\ Aa\underline{a}b \to 1\ Aaa\underline{b} \to 2\ Aaab\underline{\Delta}$
$\to 3\ Aaa\underline{b} \to 4\ Aa\underline{a}\Delta \to 4\ A\underline{a}a \to 4\ \underline{A}aa \to$ HALT $\underline{a}\underline{a}a$

6. (This change simplifies the machine on p. 775.) The label from START reads (b,b,R). State 4 is eliminated entirely so that (b,Δ,L) from 3 goes to HALT. The edge from 6 to 9 is changed to (b,b,R).

7. This machine assumes that the two inputs inputs have the same number of bits and that the data is in the proper form. (It is easy to write a preprocessing routine to check form and pad the shorter string with high-order zeros.)
The single dot state is a no-borrow state; the double dot marks the borrow state. In these states we erase (and remember) the last bit of the minuend. We cycle right to the last bit of the subtrahend and erase it. Now we go left to the blank in the minuend and write in the difference, moving into borrow or to no-borrow as necessary. If we reach the left of the minuend at no-borrow, we just erase the second $; if we reach it in borrow we change 1's to 0's and erase the $.



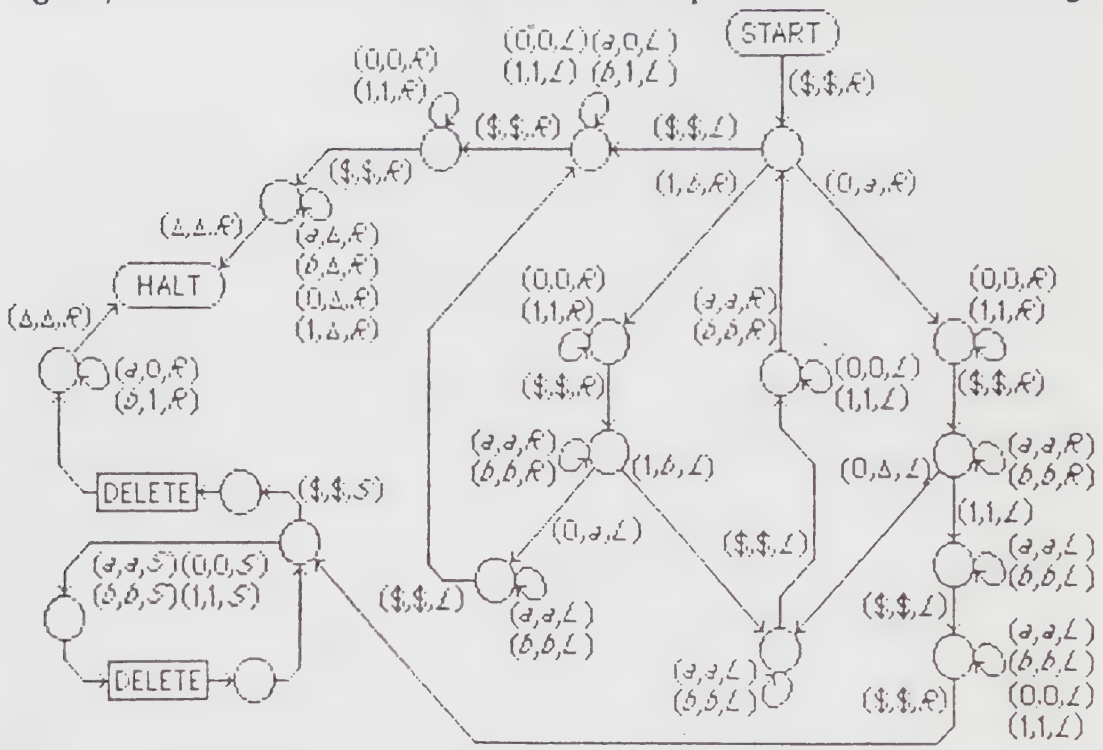
8. (i) START *aaaba* →1 *Aaaba* →1 *Aaaba* →1 *Aaaba* →2 *Aaaba*
→2 *Aaaba*△ →3 *Aaaba* →5 *Aaaby* →6 *Aaaby* →7 *Aaxby*
→8 *Aaxby* →3 *Aaxby* →4 *Aaxby* →4 *Aaxby*△ →10 *Aaxby*
→10 *Aaxb*△ →10 *Aax*△△ →10 *Aa*△△ →10 *A*△△ → HALT *aaa*

(ii) START *baaa* CRASH [Suggested interpretation: It is always preferable to design a more robust machine.]

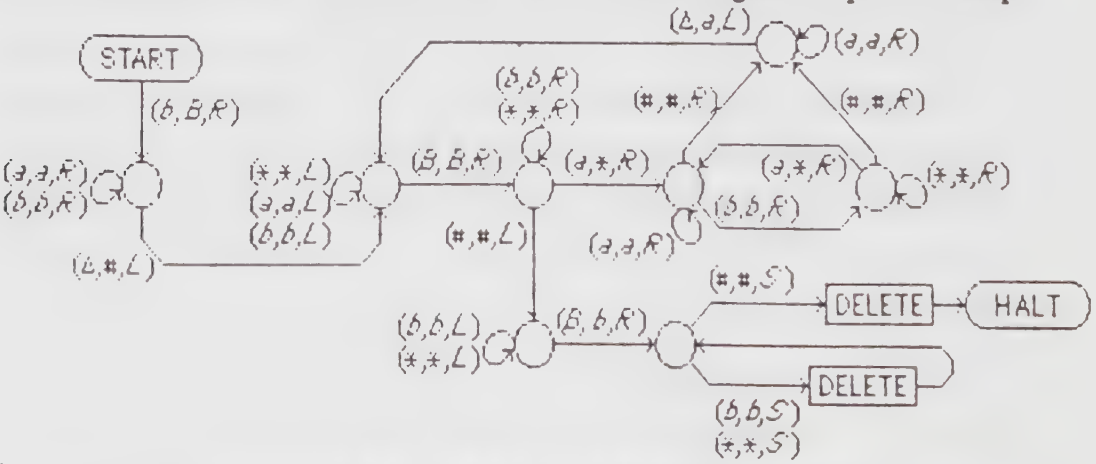
(iii) START *aabaa* →1 *Aabaa* →1 *Aabaa* → 2 *Aabaa* → 2 *Aabaa*
→2 *Aabaa*△ →3 *Aabaa* →5 *Aabay* → 5 *Aabay* → 6 *Aabay*
→7 *Axbay* →8 *Axbay* →8 *Axbay* → 3 *Axbay* → 5 *Axbyy*
→6 *Axbyy* →6 *Axbyy* →9 *axbyy* → 9 *aabyy* → 9 *aaayy*
→9 *aaa*△*y* →9 *aaa*△△△ →11 *aaa*△△ →11 *aaa*△△ →11 *aa*△△
→HALT *aa*△△△

(iv - vi) Let us call the first number *m* and the second *n*. If *n* > *m* (and *m* > 0) the TAPE HEAD is in cell *n*+2 when the machine reaches HALT. If they are equal it is in cell *n*+2; if *m* > *n* the TAPE HEAD is in cell ii when the machine reaches HALT.

9. Again, this machine assumes that the inputs are the same length.



10. This machine interprets any string in $\mathbf{b(a+b)^*}$ as n numbers (where n is the number of b's in the input and the value of the i'th number is the number of consecutive a's following the ith b) and leaves as output some string in $\mathbf{ba^*}$, where the value of the number is the number of a's in the longest input clump.



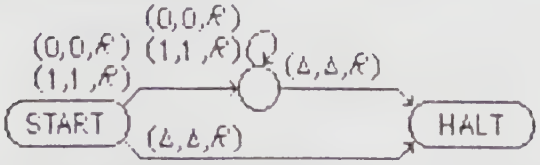
11.(i) IDENTITY START $\underline{a}a \rightarrow 1\ a\underline{a} \rightarrow 1\ aa\underline{\Delta} \rightarrow$ HALT $aa\underline{\Delta}\Delta$

SUCCESSOR START $\underline{a}a \rightarrow q\ a\underline{a} \rightarrow q\ aa\underline{\Delta}$ HALT $aaa\underline{\Delta}$

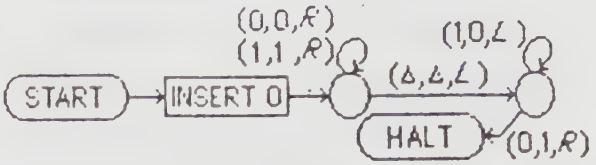
(ii) IDENTITY START $\underline{a}aaaba \rightarrow 1\ a\underline{a}aaba \rightarrow 1\ aa\underline{a}aba \rightarrow 1\ aaa\underline{b}a$ CRASH

SUCCESSOR START $\underline{a}aaaba \rightarrow q\ a\underline{a}aaba \rightarrow q\ aa\underline{a}aba \rightarrow q\ aaaaba$ CRASH

12. IDENTITY

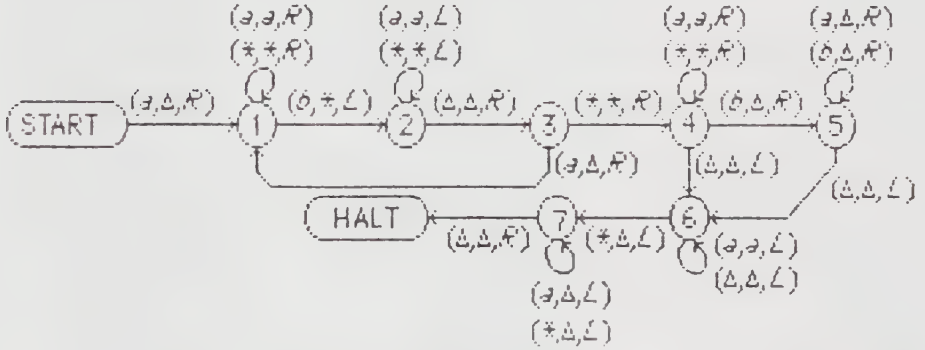


SUCCESSOR



13. START $\underline{b}baaababaaaba \rightarrow 3\ {}^{*}\underline{b}aaababaaaba \rightarrow 4\ {}^{*}\Delta\underline{a}aababaaaba$
$\rightarrow 4\ {}^{*}\Delta a\underline{a}ababaaaba \rightarrow 4\ {}^{*}\Delta aa\underline{a}babaaaba \rightarrow 4\ {}^{*}\Delta aaa\underline{b}abaaaba$
$\rightarrow 5\ {}^{*}\Delta aaa\Delta\underline{a}baaaba \rightarrow 5\ {}^{*}\Delta aaa\Delta\Delta\underline{b}aaaba \rightarrow 5\ {}^{*}\Delta aaa\Delta\Delta\Delta\underline{a}aba$
$\rightarrow 5\ {}^{*}\Delta aaa\Delta\Delta\Delta\Delta\underline{a}ba \rightarrow 5\ {}^{*}\Delta aaa\Delta\Delta\Delta\Delta\underline{b}a \rightarrow 5\ {}^{*}\Delta aaa\Delta\Delta\Delta\Delta\Delta\underline{a}$
$\rightarrow 5\ {}^{*}\Delta aaa\Delta\Delta\Delta\Delta\Delta\Delta\underline{\Delta}$ Note that this input represents 6, not 5, inputs. This machine does not check for form.

14.(i)



(ii) START $aabaaaabaabaaba$ $\rightarrow 1\ \Delta\underline{a}baaaabaabaaba$

$\rightarrow 1\ \Delta\underline{a}baaaabaabaaba \rightarrow 2\ \Delta\underline{a}^{*}aaabaabaaba \rightarrow 2\ \underline{\Delta}a^{*}aaabaabaaba$
$\rightarrow 3\ \Delta\underline{a}^{*}aaabaabaaba \rightarrow 1\ \Delta\Delta^{*}\underline{a}aabaabaaba \rightarrow 1\ \Delta\Delta^{*}\underline{a}aabaabaaba$
$\rightarrow 1\ \Delta\Delta^{*}a\underline{a}abaabaaba \rightarrow 1\ \Delta\Delta^{*}aa\underline{a}baabaaba \rightarrow 1\ \Delta\Delta^{*}aaa\underline{b}aabaaba$
$\rightarrow 2\ \Delta\Delta^{*}aa\underline{a}^{*}aabaaba \rightarrow 2\ \Delta\Delta^{*}a\underline{a}a^{*}aabaaba \rightarrow 2\ \Delta\Delta^{*}\underline{a}aa^{*}aabaaba$
$\rightarrow 2\ \Delta\Delta^{*}\underline{\ }aaa^{*}aabaaba \rightarrow 2\ \Delta\Delta^{*}aaa^{*}aabaaba \rightarrow 3\ \Delta\Delta^{*}\underline{\ }aaa^{*}aabaaba$
$\rightarrow 4\ \Delta\Delta^{*}\underline{a}aa^{*}aabaaba \rightarrow 4\ \Delta\Delta^{*}a\underline{a}a^{*}aabaaba \rightarrow 4\ \Delta\Delta^{*}aa\underline{a}^{*}aabaaba$
$\rightarrow 4\ \Delta\Delta^{*}aaa^{*}\underline{\ }aabaaba \rightarrow 4\ \Delta\Delta^{*}aaa^{*}\underline{a}abaaba \rightarrow 4\ \Delta\Delta^{*}aaa^{*}a\underline{a}baaba$
$\rightarrow 4\ \Delta\Delta^{*}aaa^{*}aa\underline{b}aaba \rightarrow 5\ \Delta\Delta^{*}aaa^{*}aa\Delta\underline{a}aba \rightarrow 5\ \Delta\Delta^{*}aaa^{*}aa\Delta\Delta\underline{a}ba$
$\rightarrow 5\ \Delta\Delta^{*}aaa^{*}aa\Delta\Delta\Delta\underline{b}a \rightarrow 5\ \Delta\Delta^{*}aaa^{*}aa\Delta\Delta\Delta\Delta\underline{a} \rightarrow 5\ \Delta\Delta^{*}aaa^{*}aa\Delta\Delta\Delta\Delta\Delta\underline{\Delta}$
$\rightarrow 6\ \Delta\Delta^{*}aaa^{*}aa\Delta\Delta\Delta\underline{\Delta} \rightarrow 6\ \Delta\Delta^{*}aaa^{*}aa\Delta\Delta\underline{\Delta} \rightarrow 6\ \Delta\Delta^{*}aaa^{*}aa\Delta\underline{\Delta}$
$\rightarrow 6\ \Delta\Delta^{*}aaa^{*}aa\Delta\underline{\Delta} \rightarrow 6\ \Delta\Delta^{*}aaa^{*}a\underline{a} \rightarrow 6\ \Delta\Delta^{*}aaa^{*}\underline{a}a$
$\rightarrow 6\ \Delta\Delta^{*}aaa^{*}\underline{\ }aa \rightarrow 7\ \Delta\Delta^{*}aa\underline{a}\Delta aa \rightarrow 7\ \Delta\Delta^{*}a\underline{a}\Delta\Delta aa$
$\rightarrow 7\ \Delta\Delta^{*}\underline{a}\Delta\Delta\Delta aa \rightarrow 7\ \Delta\Delta^{*}\underline{\ }\Delta\Delta\Delta\Delta aa \rightarrow 7\ \Delta\underline{\Delta}\Delta\Delta\Delta\Delta\Delta aa$ HALT

Chapter Twenty Five

15.(i) START $\underline{b}abaa \to 1\ b\underline{a}baa \to 2\ ba\underline{b}aa \to 3\ bab\underline{a}a$
$\to 4\ baba\underline{a} \to 4\ baba\underline{a}\triangle \to 5\ baba\underline{a}\# \to 5\ bab\underline{a}a\#$
$\to 5\ ba\underline{b}aa\# \to 6\ b\underline{a}baa\# \to 6\ \underline{b}abaa\# \to 7\ b\underline{a}baa\#$
$\to 8\ b\triangle\underline{b}aa\# \to 9\ b\triangle b\underline{a}a\# \to 10\ b\triangle bA\underline{a}\# \to 10\ b\triangle bA\underline{a}\#$
$\to 10\ b\triangle bAa\#\triangle \to 11\ b\triangle bAa\#\underline{a} \to 11\ b\triangle bAa\#\underline{a}a \to 11\ b\triangle bA\underline{a}\#a$
$\to 11\ b\triangle b\underline{A}a\#a \to 12\ b\triangle bA\underline{a}\#a \to 10\ b\triangle bAA\#\underline{a} \to 10\ b\triangle bAA\#\underline{a}$
$\to 10\ b\triangle bAA\#\underline{a}\triangle \to 11\ b\triangle bAA\#\underline{a}a \to 11\ b\triangle bAA\#\underline{a}a \to 11\ b\triangle bA\underline{A}\#aa$
$\to 12\ b\triangle bAA\#\underline{a}a \to 13\ b\triangle bA\underline{A}\#aa \to 13\ b\triangle b\underline{A}a\#aa \to 13\ b\triangle\underline{b}aA\#aa$
$\to 14\ b\triangle\underline{b}aa\#aa \to 15\ b\triangle\underline{b}aa\#aa \to 15\ b\triangle\triangle\underline{a}a\#aa \to 15\ b\triangle\triangle\triangle\underline{a}\#aa$
$\to 15\ b\triangle\triangle\triangle\triangle\#\underline{a}a \to 18\ b\triangle\triangle\triangle\triangle a\underline{a}a \to 18\ b\triangle\triangle\triangle\triangle aa\underline{a} \to 18\ b\triangle\triangle\triangle\triangle aa\underline{a}\triangle$
$\to 19\ b\triangle\triangle\triangle\triangle aa\underline{a} \to 20\ b\triangle\triangle\triangle\triangle a\underline{a}\triangle \to 20\ b\triangle\triangle\triangle\triangle\underline{a}a \to 20\ b\triangle\triangle\triangle\triangle\triangle aa$
$\to 18\ b\triangle\triangle\triangle aaa \to 18\ b\triangle\triangle\triangle aaa \to 18\ b\triangle\triangle\triangle aaa\triangle \to 19\ b\triangle\triangle\triangle aa\underline{a}$
$\to 20\ b\triangle\triangle\triangle a\underline{a}\triangle \to 20\ b\triangle\triangle\triangle\underline{a}a \to 20\ b\triangle\triangle\triangle\triangle aa \to 18\ b\triangle\triangle a\underline{a}a$
$\to 18\ b\triangle\triangle aaa \to 18\ b\triangle\triangle aaa\triangle \to 19\ b\triangle\triangle a\underline{a}\triangle \to 20\ b\triangle\triangle\underline{a}a$
$\to 20\ b\triangle\triangle\triangle aa \to 18\ b\triangle a\underline{a}a \to 18\ b\triangle aaa \to 18\ b\triangle aaa\triangle$
$\to 19\ b\triangle aa\underline{a} \to 20\ b\triangle a\underline{a}\triangle \to 20\ b\triangle\underline{a}a \to 20\ b\triangle\triangle aa$
$\to 18\ baa\underline{a} \to 18\ baaa \to 18\ baaa\triangle \to 19\ baa\underline{a}$
$\to 20\ baa\triangle \to 20\ b\underline{a}a \to 20\ \underline{b}aa \to \text{HALT}\ b\underline{a}a$

(ii) START $\underline{b}aaaba \to 2\ b\underline{a}aaba \to 2\ ba\underline{a}aba$
$\to 2\ baa\underline{a}ba \to 2\ baaa\underline{b}a \to 3\ baaab\underline{a}$
$\to 4\ baaab\underline{a}\triangle \to 5\ baaab\underline{a}\# \to 6\ baaa\underline{b}a\#$
$\to 6\ baa\underline{a}ba\# \to 6\ ba\underline{a}aba\# \to 6\ b\underline{a}aaba\#$
$\to 6\ \underline{b}aaaba\# \to 7\ b\underline{a}aaba\# \to 8\ b\triangle\underline{a}aba\#$
$\to 8\ b\triangle a\underline{a}ba\# \to 8\ b\triangle aa\underline{b}a\# \to 9\ b\triangle aab\underline{a}\#$
$\to 10\ b\triangle aabA\#\underline{\ } \to 10\ b\triangle aabA\#\triangle \to 11\ b\triangle aabA\#\underline{a}$
$\to 11\ b\triangle aab\underline{A}\#a \to 11\ b\triangle aabA\#\underline{a} \to 13\ b\triangle aab\underline{A}\#a$
$\to 13\ b\triangle aa\underline{b}a\#a \to 14\ b\triangle a\underline{a}ba\#a \to 16\ b\triangle\underline{a}aba\#a$
$\to 16\ b\triangle aaba\#a \to 17\ b\triangle\underline{a}aba\#a \to 8\ b\triangle\triangle\underline{a}ba\#a$
$\to 8\ b\triangle\triangle a\underline{b}a\#a \to 9\ b\triangle\triangle ab\underline{a}\#a \to 10\ b\triangle\triangle abA\#\underline{a}$
$\to 10\ b\triangle\triangle abA\#\underline{a} \to 10\ b\triangle\triangle abA\#\underline{a}\triangle \to 11\ b\triangle\triangle abA\#\underline{a}a$
$\to 11\ b\triangle\triangle abA\#\underline{a}aa \to 11\ b\triangle\triangle ab\underline{A}\#aa \to 12\ b\triangle\triangle abA\#\underline{a}aa$
$\to 13\ b\triangle\triangle ab\underline{A}\#aa \to 13\ b\triangle\triangle a\underline{b}a\#aa \to 14\ b\triangle\triangle\underline{a}ba\#aa$
$\to 16\ b\triangle\triangle\underline{\ }aba\#aa \to 17\ b\triangle\triangle\underline{a}ba\#aa \to 8\ b\triangle\triangle\triangle\underline{b}a\#aa$
$\to 9\ b\triangle\triangle\triangle b\underline{a}\#aa \to 10\ b\triangle\triangle\triangle bA\#\underline{a}a \to 10\ b\triangle\triangle\triangle bA\#\underline{a}a$

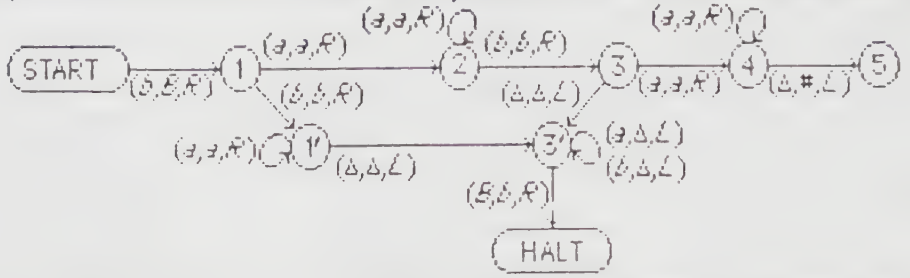
→ 10 *bΔΔΔbA# aa*
→ 11 *bΔΔΔbA# aaa*
→ 12 *bΔΔΔbA# aaa*
→ 14 *bΔΔΔba# aaa*
→ 15 *bΔΔΔΔΔ# aaa*
→ 18 *bΔΔΔΔΔ aaaa*
→ 18 *bΔΔΔΔΔ aaaaΔ*
→ 20 *bΔΔΔΔΔ aaaΔ*
→ 20 *bΔΔΔΔΔ aaa*
→ 18 *bΔΔΔΔ aaaa*
→ 18 *bΔΔΔΔ aaaaΔ*
→ 19 *bΔΔΔΔ aaaa*
→ 20 *bΔΔΔΔ aaa*
→ 18 *bΔΔΔ aaaa*
→ 18 *bΔΔΔ aaaaΔ*
→ 19 *bΔΔΔ aaaa*
→ 20 *bΔΔΔ aaa*
→ 18 *bΔΔ aaaa*
→ 18 *bΔΔ aaaaΔ*
→ 20 *bΔΔ aaa*
→ 18 *bΔ aaaa*
→ 18 *bΔ aaaaΔ*
→ 20 *bΔ aaa*
→ 18 *baaaa*
→ 18 *baaaaΔ*
→ 20 *baaa*
→ HALT *baaa*

→ 10 *bΔΔΔbA# aaΔ*
→ 11 *bΔΔΔbA# aaa*
→ 13 *bΔΔΔbA# aaa*
→ 15 *bΔΔΔba# aaa*
→ 18 *bΔΔΔΔΔ aaaa*
→ 18 *bΔΔΔΔΔ aaaaΔ*
→ 20 *bΔΔΔΔΔ aaa*
→ 18 *bΔΔΔΔ aaaa*
→ 18 *bΔΔΔΔ aaaaΔ*
→ 20 *bΔΔΔΔ aaaΔ*
→ 20 *bΔΔΔΔ aaa*
→ 18 *bΔΔΔ aaaa*
→ 18 *bΔΔΔ aaaaΔ*
→ 20 *bΔΔΔ aaaΔ*
→ 20 *bΔΔΔ aaa*
→ 18 *bΔΔ aaaa*
→ 18 *bΔΔ aaaaΔ*
→ 19 *bΔΔ aaaa*
→ 20 *bΔΔ aaa*
→ 18 *bΔ aaaa*
→ 19 *bΔ aaaa*
→ 20 *bΔ aaa*
→ 18 *baaaa*
→ 19 *baaaa*
→ 20 *baaa*

→ 11 *bΔΔΔbA# aaa*
→ 11 *bΔΔΔbA# aaa*
→ 13 *bΔΔΔba# aaa*
→ 15 *bΔΔΔΔ# aaa*
→ 18 *bΔΔΔΔΔ aaaa*
→ 19 *bΔΔΔΔΔ aaaa*
→ 20 *bΔΔΔΔΔ aaa*
→ 18 *bΔΔΔΔ aaaa*
→ 18 *bΔΔΔΔ aaaaΔ*
→ 20 *bΔΔΔΔ aaa*
→ 18 *bΔΔΔ aaaa*
→ 18 *bΔΔΔ aaaaΔ*
→ 20 *bΔΔΔ aaa*
→ 18 *bΔΔ aaaa*
→ 18 *bΔΔ aaaa*
→ 20 *bΔΔ aaaΔ*
→ 20 *bΔΔ aaa*
→ 18 *bΔ aaaa*
→ 20 *bΔ aaaΔ*
→ 20 *bΔ aaa*
→ 18 *baaaa*
→ 20 *baaaΔ*
→ 20 *baaa*

16. We can do this modification to the first part of the MPY routine (on page 785) by altering one label and adding two new states: First, change the instruction from start to state 1 to (*b*, *B*, *R*), marking cell i. Then add the instruction (*b*, *b*, *R*) from state 1 to new state 1′, to account for the possibility that the first input is zero, and the instruction (Δ, Δ, *L*) from state 3 to new state 3′, to account for a second input of zero. At state 1′ there is a loop labeled (*a*, *a*, *R*) (to check the form of the input) and an edge to 3′: (ΔΔ, *L*). In state 3′ the form of the input is correct and one is zero, so we erase everything: (*a*Δ, *L*) and (*b*Δ, *L*). We have

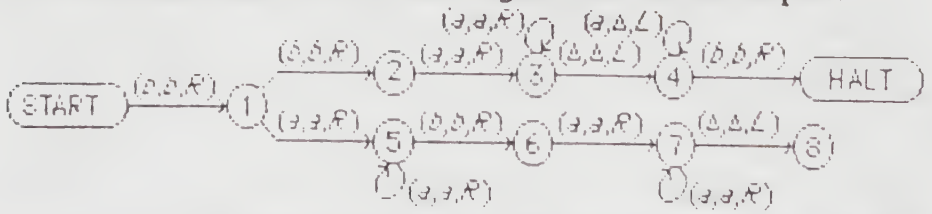
finished when we reach the *B* that marks cell i. We change it back and move right and into HALT . (The rest of theTAPE is now blank, so we need not delete anything.) The whole routine is



17. It is probably easiest to describe the shift and add method for a 3TM, multiplicand in track 1, multiplier in track 2, product in track 3. If the least significant bit of the multiplier is 0, just shift the product; if it is 1, shift the product and to it add the multiplicand in track 1. Delete the least significant bit on track 2 and, if there are more nonblank multiplier cells, repeat. To avoid the necessity of pushing everything down the TAPE at each step, either allow for the product as many cells as there are bits in the two inputs combined or process from right to left, with the least significant bits at the beginning of the TAPE.

18. Use the Euclidean algorithm.

States 1 - 7 : Check the input, rejecting strings not of the form **ba*baa*** . If the dividend is 0, reach state 3, erase the divisor leaving the *b*'s to represent the answer 0, and go to HALT. If the dividend is not 0 verify the form and reach state 8, where processing begins from the right end of the input.



States 8 - 11: New subtraction: Change the last *a* of the divisor to *A* and match it by changing the first *a* of the dividend to *.

States 12 - 14: Find the character in the divisor left of the leftmost *A*.

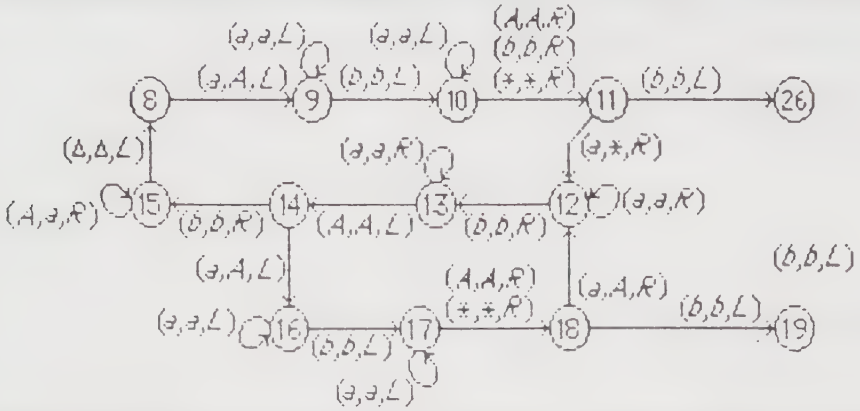
States 14, 16, 17, 18, 12: Subtraction continues: Pair an *a* from

the divisor with one from the dividend (changing both to *A*).

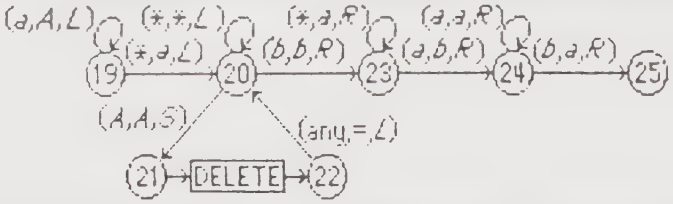
States 14,15, 8: Subtraction complete: Restore the divisor to lower case and begin a new subtraction.

States 11, 26: The dividend has been entirely marked and a new subtraction is just beginning; i.e. the remainder is zero.

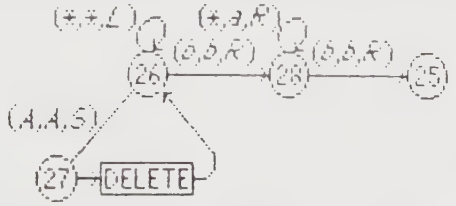
States 18, 19: The dividend has been entirely marked and a subtraction is in process. There is a nonzero remainder.



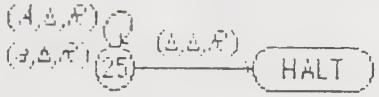
States 19-20-21-22-23-24: Change the remainder portion back to *a*. (The rightmost * is the last character to change.) Delete all *A*'s left of that; each * remaining counts one whole subtraction. Change these to *a*. Change the first *a* of the remainder to *b* to separate it from the quotient; leave the rest of the remainder; change the *b* following to *a* to replace the other *a* changed.



States 26-27-28: No remainder. Delete all *A*'s and change * to *a*.



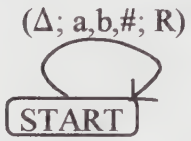
State 25: Erase the rest of the string.



Chapter Twenty Five

19. To determine whether a number is prime, we need to know if the number has any divisors. So we would like to test integers 2 through the object number. We can use repeated 'subtraction' to simulate the division process. When subtracting the a's on the right from the a's on the left we change them into b's so that we can restore the original a^n if desired. Next we need to decide whether
(i) the process is incomplete, subtract again.
or (ii) the process is complete and the number is a divisor - the number is not a prime.
or (iii) the 'division/subtraction' of the second number is complete but not a divisor of the first. In this case, we need to decide whether there is still a possible divisor or not, in the latter case the first number is a prime.

20. Consider the following TM.



It may print out *any* language, but we cannot *know* which.

ISBN 0-471-17305-3

John Wiley & Sons, Inc.
New York · Chichester · Brisbane
Toronto · Singapore · Weinheim