

**Solutions to Selected Exercises for Chapter 7 of R. Gregory Taylor, *Models of Computation and Formal Languages* (Oxford University Press: New York, 1998)**

**© Oxford University Press**

**Solutions to Exercises for § 7.2**

- 7.2.1 (a) is a vector of length 4 representing +9.  
(b) is not a vector.  
(c) is a vector of length 5 representing +28.  
(d) is a vector of length 5 representing -15.  
(e) is a vector of length 8 representing +186.  
(f) is a vector of length 8 representing -27.

- 7.2.2. (a) ...000000000000010101 or +10101  
(b) ...111111111111010101 or -010101  
(c) ...111111111110010101 or -0010101  
(d) not possible  
(e) ...11111111111000000 or -000000

- 7.2.3. (a) ...00000000000000010001  
(b) ...00000000000000011011  
(c) ...111111111111111101  
(d) ...1111111111111111101  
(e) ...11111111111111110101  
(f) ...0000000000000001010  
(g) ...0000000000000001010

- 7.2.4 (a)  $\neg B$  (b)  $A \vee B$  (c)  $\neg(A \& B)$

7.2.5.  $bval(A)$  is +19,  $bval(B)$  is +25,  $bval(\neg A)$  is -12, and  $bval(\neg B)$  is -6. It should now be obvious that Boolean  $\neg$  as applied to vectors is not in any sense equivalent to unary minus as applied to integers. We shall be able to implement unary minus, however (see below).

**Solutions to Exercises for § 7.3**

7.3.1. Register  $V_2$  comes to contain vector  $+10^{n-1}$ .

7.3.2.  $V_2 := V_2 \ll_0 V_3$

7.3.4.  $V_2 := V_2 \ll_0 V_3$   
 $V_2 := V_2 \ll_0 1$

### Solutions to Exercises for § 7.4

7.4.1.

**Input**  $V_1$   
 $V_2 = +I^n$  for  $n \geq 1$

**Output**  $V_3 = \begin{cases} + & \text{if the } n^{\text{th}} \text{ bit from the right is } 0 \\ +I & \text{if the } n^{\text{th}} \text{ bit from the right is } I \end{cases}$

**Algorithm**

```
begin
     $V_2 := V_2 \gg 1$ ;
     $V_4 := \text{Convert}(V_2)$ ; {applying the Unary-to-Binary Converter
                           (Example 7.4.2)}
     $V_5 := +I$ ;
     $V_6 := V_1 \gg V_4$ ;
    if  $V_6 \& V_5 \neq +$  then  $V_3 := +I$  else  $V_3 := +$ 
end.
```

7.4.2. The machine described below computes in  $O(\log_2 n)$  parallel steps for  $n = |w|$ .

**Input**  $V_1 = +w$   
 $V_2 = +I^{|w|}$

**Output**  $V_3 = +ww$

**Algorithm**

```
begin
     $V_4 := \text{Convert}(V_2) = B(|w|)$ ; {applying the Unary-to-Binary Converter
                                     (Example 7.4.2)}
     $V_3 := (V_1 \ll_0 V_4) \vee V_1$ 
end.
```

7.4.4. Machine  $M$  computes in  $O(n)$  time. To see this, note that the while-loop of lines (3)–(6) will be iterated precisely  $n-1$  times so that  $O(n)$  steps are required to complete all iterations. The instructions before and after the while-loop require an additional  $O(1)$  steps only. In any case, the earlier solution of Exercise 7.4.1 computes in  $O(\log_2 n)$  steps and hence is to be preferred. In particular, that solution has a call to the Unary-to-Binary Converter and hence is in accordance with Programming Hint 7.4.3, whereas  $M$  ignores Programming Hint 7.4.3 and, consequently, has no advantage over sequential machines. For example, it is easy to see that a multitape Turing machine might use what is in essence the algorithm embodied in  $M$  to identify the  $n^{\text{th}}$  bit from the right in  $O(n)$  steps.

7.4.5.

```
 $m_{50} = +10101010101010101010101010101010$ 
 $m_{51} = +11001100110011001100110011001100$ 
 $m_{52} = +11110000111100001111000011110000$ 
 $m_{53} = +11111111000000001111111100000000$ 
 $m_{54} = +11111111111111110000000000000000$ 
```

7.4.6. We use lines (1)–(3), which precede (7.4.2) in the text. To construct  $m_{41}$ , we can use

- (1)  $W := W \gg 1;$        $\{W \text{ is now } +10.\}$
- (2)  $Z := Y \gg W$        $\{Z \text{ is } +11110000111100.\}$
- (3)  $Y := Y \oplus Z;$        $\{Y \text{ is } +1100110011001100, \text{ which is } m_{41}.\}$

To construct  $m_{40}$ , we can use

- (1)  $W := W \gg 1;$        $\{W \text{ is now } +1.\}$
- (2)  $Z := Y \gg W$        $\{Z \text{ is } +110011001100110.\}$
- (3)  $Y := Y \oplus Z;$        $\{Y \text{ is } +1010101010101010, \text{ which is } m_{40}.\}$

- 7.4.7 (a)  $V_3$  is +10001100 and  $V_4$  is +10001111.  
 (b)  $V_3$  is +1011111001100000 and  $V_4$  is +1011111001111111.  
 (c)  $V_3$  is +11100011 and  $V_4$  is +11100011.  
 (d)  $V_3$  is  $+(10)^{50} \underline{0}^{28}$  and  $V_4$  is  $+(10)^{50} \underline{1}^{28}$ .  
 (e)  $M$  in effect calculates the difference  $d$  between  $|w|$  and the least power of 2 that is greater than or equal to  $|w|$ . Then,  $V_3$  will be  $V_1 \hat{\ } 0^d$  and  $V_4$  will be  $V_1 \hat{\ } 1^d$ . (Only  $V_3$  would be used in conjunction with Example 7.4.7.)

7.4.8. We first show that if  $f$  is partial recursive, then  $f$  is vector-machine-computable. So suppose that  $f$  is partial recursive. First, by Theorem 5.1(a), we have that  $f$  is register-machine-computable. Let  $M_{RM}$  be a register machine that computes  $f$ . Since the adopted conventions regarding function computation for vector machines (Definition 7.7) are modeled on those for register machines (Definition 5.2), we have only to show that each register machine instruction occurring within  $M_{RM}$ 's program can be simulated by some vector machine instruction. This turns out to be easy. Any register machine instruction of the form

$$R_i ++$$

within  $M_{RM}$ 's program will correspond to a vector machine instruction

$$V_i := V_i \ll 1$$

Assignment  $R_i := 0$  will correspond to  $V_i := +$ . Finally, any conditional branch instruction of the form

$$R_i = R_j ? \text{ goto } \mathcal{L}$$

occurring within  $M_{RM}$ 's program can be mapped onto

$$\neg(V_i \leftrightarrow V_j) = + ? \text{ goto } \mathcal{L}$$

That leaves just the Start and Halt instructions and we are done. The result of replacing each instruction within  $M_{RM}$ 's program with its vector-machine equivalent is the program of a vector machine  $M_{VM}$  that computes  $f$  in the sense of Definition 7.7. It follows that  $f$  is vector-machine-computable.

As for the other direction, suppose that  $f$  is a  $k$ -ary vector-machine-computable function. Let  $M_{VM}$  be a vector machine that computes  $f$  and suppose that  $M_{VM}$  has  $j$  registers. (By Definition 7.7, we have  $j \geq k+1$ .) We describe a Turing machine  $M_{TM}$  with at least  $j+2$  tapes that computes  $f$  in the sense of § 2.3. It will be convenient to assume that these tapes are one-way-infinite, possessing a *rightmost* tape square and endmarker  $\rho$ .  $M_{TM}$ 's worktapes will in general contain finite, unbroken strings of 0s and 1s starting at the far right. As usual, we assume that the  $k$  arguments of  $f$  appear initially on the input tape of  $M_{TM}$ . Machine

$M_{TM}$  first copies them to  $worktape_1$  through  $worktape_k$ , respectively. Afterward,  $M_{TM}$  proceeds to simulate  $M_{VM}$  on  $worktape_1$  through  $worktape_j$ . For example, an assignment of the form

$$V_i := V_i \ll 5$$

will cause  $M_{TM}$  to in effect append five  $I$ s to the unbroken string of  $O$ s and  $I$ s currently on  $worktape_i$ . An assignment of the form

$$V_i := V_i \gg 5$$

will cause  $M_{TM}$  to erase up to five digits from the right end of the unbroken string of  $O$ s and  $I$ s currently on  $worktape_i$  and then to shift the remaining symbols to the far right. A conditional branch of the form

$$V_i \vee V_j =+? \text{ goto } \mathcal{L}$$

will require that  $M_{TM}$  implement the indicated Boolean operation on the strings currently on  $worktape_i$  and  $worktape_j$ , respectively, storing the results on some  $worktape_{i'}$ . This is no problem, even in the case where the strings on  $worktape_i$  and  $worktape_{i'}$  are of unequal length. Afterward,  $M_{TM}$  determines whether the string currently on  $worktape_{i'}$  consists of  $O$ s only. If so, then  $M_{TM}$  proceeds with its simulation of the vector machine instructions at  $\mathcal{L}$ . If and when  $M_{TM}$ 's simulation of  $M_{VM}$  halts,  $M_{TM}$  copies the (relevant portion of the) contents of  $worktape_{k+1}$  to its output tape before halting. It should be clear that  $M_{TM}$  computes  $f$  and that, as a consequence,  $f$  is Turing-computable and hence partial recursive.

### Solutions to Exercises for § 7.5

7.5.1. Suppose, first, that  $L$  is a vector-machine-acceptable language. Let  $M_{VM}$  be a vector machine that accepts  $L$  and suppose that  $f$  has  $j$  registers. (By Definition 7.9, we have  $j \geq 3$ .) We describe a Turing machine  $M_{TM}$  with at least  $j+2$  tapes that accepts  $L$  in the sense of § 2.3. As in the solution to Exercise 7.4.8, it will be convenient to assume that these tapes are one-way-infinite, possessing a rightmost tape square and endmarker  $\rho$ .  $M_{TM}$ 's worktapes will in general contain finite, unbroken strings of  $O$ s and  $I$ s starting at the far right. As usual, we assume that the input word  $w$  appears initially on the input tape of  $M_{TM}$ . Machine  $M_{TM}$  first copies  $w$  to  $worktape_1$ . Next,  $M_{TM}$  produces  $I^{hm}$  starting at the far right on its  $worktape_2$ . Afterward,  $M_{TM}$  proceeds to simulate  $M_{VM}$  on  $worktape_1$  through  $worktape_j$  precisely as described in the solution to Exercise 7.4.8. If and when  $M_{TM}$ 's simulation of  $M_{VM}$  halts,  $M_{TM}$  copies just the rightmost symbol of  $worktape_3$  to its output tape before halting. It should be clear that  $M_{TM}$  accepts  $L$  and that, as a consequence,  $L$  is Turing-acceptable.

As for the other direction, suppose that  $L$  is Turing-acceptable. First, by Theorem 5.2, we have that  $L$  is register-machine-acceptable. Let  $M_{RM}$  be a register machine that accepts  $L$ . We describe the operation of a vector machine  $M_{VM}$  that simulates  $M_{RM}$  and thereby accepts  $L$  in the sense of Definition 7.9. We shall focus here on Read and Write instructions, having already described, in the solution to Exercise 7.4.8, how most other register machine instructions will be simulated by  $M_{VM}$ . We must suppose that, initially, input word  $w$  is present in vector  $V_1$ .  $M_{VM}$  then uses  $V_1$  to produce vector  $+|w|$  in register  $V_2$ . If  $M_{RM}$  reads  $w$  from its input tape into registers  $R_i$  through  $R_{i+n-1}$ , say, then  $M_{VM}$  will read  $w$  from  $V_1$  into  $V_i$  through  $V_{i+n-1}$  one character/bit at a time. (For this purpose, it might be convenient to think of  $w$  being transformed into  $w^R$  first, although this is not essential.) So an instruction of the form

$$\text{Read}(R_i)$$

will involve  $M_{VM}$ 's ascertaining the rightmost character/bit in  $V_1$  and then copying it into  $V_i$  before shifting  $V_1$  one position to the right. In any case, we see that  $L$  is vector-machine-acceptable.

7.5.2. Suppose, first, that  $L \in P$ . By Definition 1.11 and Theorem 5.3(a), it follows that there exists a register machine  $M_{RM}$  that accepts  $L$  such that  $time_{M_{RM}}(n)$  is  $O(p(n))$  for some polynomial  $p(n)$ . But now the construction of the solution to Exercise 7.4.8 can be modified to produce a vector machine  $M_{VM}$  that accepts  $L$ , where  $time_{M_{VM}}(n)$  is  $O(time_{M_{RM}}(n))$  and hence  $O(p(n))$ . Moreover, by the nature of the simulation of  $M_{RM}$  by  $M_{VM}$ , we have that  $space_{M_{VM}}(n)$  will be bounded above by  $time_{M_{VM}}(n)$ . It follows that  $space_{M_{RM}}(n)$  is  $O(p(n))$  as well.

As for the other direction, suppose that  $L$  is accepted by vector machine  $M_{VM}$  such that both  $time_{M_{VM}}(n)$  and  $space_{M_{VM}}(n)$  are  $O(p(n))$  for some polynomial  $p(n)$ . We show that  $L \in P$ . By Exercise 0.5.13 essentially, we may assume without loss of generality that  $time_{M_{VM}}(n) \leq p(n)$  and that  $space_{M_{VM}}(n) \leq p(n)$  for sufficiently large  $n$ . The solution to Exercise 7.4.8 can be readily modified so as to describe the construction of a multitape Turing machine  $M_{TM}$  that uses one worktape to simulate each of  $M_{VM}$ 's registers and thereby itself accepts  $L$ . Moreover, the nature of the construction of  $M_{TM}$  will cause it to be the case that each of the at most  $time_{M_{VM}}(n)$  instructions executed by  $M_{VM}$  other than shift operations will be simulated by  $M_{TM}$  in  $O([space_{M_{VM}}(n)])$ .

Consider, on the other hand,  $M_{TM}$ 's simulation of a left-shift instruction of the form

$$V_i := V_j \ll_0 V_k$$

By hypothesis,  $|V_i|$ —even after instruction execution—is bounded above by  $p(n)$ . Moreover, execution of this left-shift instruction may be described as writing a certain number of 0s at the far right on worktape<sub>*i*</sub> and then copying the contents of worktape<sub>*j*</sub> off to the left. It follows that  $M_{TM}$ 's simulation of this instruction requires  $O([p(n)]^3)$  steps. Essentially, this is because the writing of the 0s at the far right amounts to doubling  $O(p(n))$  0s a total of  $O(p(n))$  times—actually  $O(\log_2(p(n)))$  times, where each doubling of  $O(p(n))$  0s requires  $O([p(n)]^2)$  steps.

From the preceding two paragraphs, it follows that  $M_{TM}$  can simulate  $M_{VM}$ 's entire computation in  $O([p(n)]^4)$  steps. By Corollary 2.1, some single-tape Turing machine  $M_1$  can simulate  $M_{VM}$ 's computation in  $O([p(n)]^8)$  steps. But  $[p(n)]^8$  is itself a polynomial in  $n$ , from which it follows that  $M_1$  computes in polynomial time in the length of input. Q.E.D.

7.5.3 (a) See the machine at icon **Exercise 7.5.3a**.

(b) Hint: use the output of the machine at (a) to generate mask  $m_{k0} = (10)^{2^k}$  in logarithmic time. The machine in `genmask.vm` may be used for this purpose. There is then a little more to do but not much.

(c) Use the output of the machine at (a) to generate mask  $m_{k0} = (10)^{2^k}$  in logarithmic time, where  $k = \lceil \log_2 n \rceil$ . Then apply the machine of Exercise 7.5.5(b) to  $V_1$  &  $m_{k0}$ . If the result is identical with  $V_1$  itself, then  $V_1$  was of even length.

7.5.5 (a) See the machine at icon **Exercise 7.5.5a**.

(b) See the machine at icon **Exercise 7.5.5b**.

### Solutions to Exercises for § 7.7

7.7.1 (a)  $V_1 \ll_0 V_1$  is +101100000000000.

(b)  $V_2 \ll_0 V_2$  is -.

7.7.2 (a) We describe a single-tape Turing machine  $M$  that converts from unary to binary by repeatedly dividing by successive powers of 2. To this end,  $M$  maintains on its single tape the current divisor (always a power of 2), the current dividend (initially the input string  $I^n$ ), and a string consisting of

those digits in the binary numeral naming  $n$  that have been generated so far—the least significant digits, in fact. We might imagine the divisor on the left, the dividend in the center, and the incipient binary numeral off on the right.  $M$  repeatedly uses asterisks in the usual way to divide the current dividend by the current divisor, writing the remainder—either 0 or 1—off on the right. Each such division requires  $O(n^2)$  steps. Afterward, the divisor on the left is doubled in length. This also requires at most  $O(\log_2^2 n)$  steps. This process is repeated until the dividend is 0. This means that the process will be repeated  $O(\log_2 n)$  times. Consequently, the time analysis of  $M$  itself is  $O(n^2 \cdot \log_2 n)$ .

(b) Referring to the solution of 2(a), imagine now that the dividend, divisor, and incipient binary numeral are maintained on separate worktapes. The process of dividing and writing the remainder must yet be iterated  $O(\log_2 n)$  times. However, the division process itself requires no asterisks and only  $O(n)$  steps worst case. This makes for an overall time analysis of  $O(n \cdot \log_2 n)$ .

7.7.3 (a)

**Input**  $V_1$  containing an arbitrary vector  $w$

**Output**  $V_2 = \begin{cases} + & \text{if } V_1 \text{ has negative sign} \\ +I & \text{if } V_1 \text{ has positive sign} \end{cases}$

**Algorithm**

```
begin
   $V_3 := V_1$ ;
   $V_4 := -V_1$ ;
  while  $V_3 \neq +$  and  $V_4 \neq +$  do begin
     $V_3 := V_3 \gg 1$ ;
     $V_4 := V_4 \gg 1$ 
  end;
  if  $V_3 = +$  then  $V_2 := +I$  else  $V_2 := +$ 
end;
```

The foregoing machine determines the sign of  $w$  in  $O(\log_2 |w|)$  steps. Since no left shifts occur, this machine computes in  $O(|w|)$  space as well. We shall think of this machine as requiring exactly  $|w|$  processors.

(b) The following vector machine determines the sign of  $w$  in  $O(1)$  steps but requires  $O(2^{|w|})$  space.

**Input**  $V_1$  containing an arbitrary vector  $w$

**Output**  $V_2 = \begin{cases} + & \text{if } V_1 \text{ has negative sign} \\ +I & \text{if } V_1 \text{ has positive sign} \end{cases}$

**Algorithm**

```
begin
   $V_3 := V_1$ ;
   $V_3 := V_3 \gg V_3$ ;
  if  $V_3 = +$  then  $V_2 := +I$  else  $V_2 := +$ 
end;
```

Equivalently, one could use

**Input**  $V_1$  containing an arbitrary vector  $w$

**Output**  $V_2 = \begin{cases} + & \text{if } V_1 \text{ has negative sign} \\ +I & \text{if } V_1 \text{ has positive sign} \end{cases}$

**Algorithm**

```
begin
   $V_3 := V_1$ ;
   $V_3 := V_3 \ll_0 V_3$ ;
  if  $V_3 = +$  then  $V_2 := +$  else  $V_2 := +I$ 
end;
```

The space analysis may require additional comment. Let us focus on the second of the two machines here. Worst case,  $w$  might be positive and its nonconstant part consist of  $I$ s only. In that case, the execution of  $V_3 := V_3 \ll_0 V_3$  at line (2) causes a left shift-0 by  $2^m$ .

7.7.4. integer division by (a power of) 2