University of Central Florida
School of Computer Science
COT 4210 Fall 2004

Prof. Rene Peralta
Homework 6 Solutions (by TA Robert Lee)

1(a). (3.12 from textbook) A ***Turing machine with left reset*** is similar to an ordinary Turing Machine except that the transition function has the form

$$\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\text{R}, \text{RESET}\}.$$

If $\delta(a, b) = (r, b, \text{RESET})$, when the machine is in state q reading an a, the machine's head jumps to the left-hand end of the tape after it writes b in the tape and enters state r. Note that these machines do not have the usual ability to move the head one symbol left. Show that Turing machines with left reset recognize the class of Turing-recognizable languages.

Solution Given a Turing-recognizable language, by definition there exists a standard Turing Machine M_1 that recognizes it. We need to convert M_1 into a TM with left reset; call it M_2. Since it can't move left and often gets confused, M_2 will keep track of where its head is supposed to be by marking that square with a dot. After each transition is complete, there is exactly one dot on the tape, but during the transition procedure, there can be more than one dot on the tape.

For rules in M_1 of the form $x \to y, R$, M_2 performs the following procedure:

Step 1. Read an x, overwrite it with a dotted y (this dot will help it find the correct final position and will be erased later), and move right.

Step 2. Mark the current square (one square to the right of the original square) with a dot, but then it must move (because it can only move right or reset left, not stay in one position), so reset left.

Step 3. To get its head back to the correct place, move right until you read a dotted square (this will be the square with the dotted y). Erase the dot and move right. Now the head is over the correct square, and this square already has a dot on it (it was marked in Step 2).

If you thought that was complicated...

For rules in M_1 of the form $x \to y, L$, M_2 performs the following procedure:

Stage 1 (during which M_2 marks the eventual destination square, i.e., the square to the left of the original square, with a dot)

Step 1. Read an x, overwrite it with a dotted y, then reset left. If this square (the square at the left end of the tape) is dotted, then it must be the dotted y, so quit (because M_1 would also have failed to move left, so we have successfully simulated the behavior of M_1). If this square is not dotted, mark this square with a dot; the move right that must accompany this marking is part of Step 2.

Step 2. Move right. Read the current square; if it is dotted, it must be the dotted y, so erase the dot over the y, reset left, and go to Stage 2, Step 1 (because the destination square has been successfully marked). If this square is not dotted, mark it with a dot (it is our new candidate for the destination square) and reset left.

Step 3. Move right until you read a dotted square. (Note that this first dotted square keeps track of the extent of your search thus far.) Erase the dot and move right (onto the current candidate square). Go to step 2.

Stage 2 (during which M_2 positions its head on the destination square by first dotting the square to the left of the destination square)

Step 1. (Note that we have arrived here from Stage 1, Step 2, so the current head position is the square at the left end of the tape.) Read the current square; if it is dotted, it is the destination square, so reset left and quit. If it is not dotted, mark it with a dot (this is our first candidate for the square-to-the-left-of-the-destination-square, or STTLOTDS for short); again, the move right that must accompany this marking is part of Step 2.

Step 2. Move right. Read the current square; if it is dotted, it is the destination square, so reset left and go to step 4. If it is not dotted, mark it with a dot (this is the new candidate for the STTLOTDS) and reset left.

Step 3. Move right until you read a dotted square (this is the square marking the extent of our search so far). Erase the dot and move right. Go to step 2.

Step 4. Move right until you read a dotted square (this is the STTLOTDS). Erase the dot, move right, and quit.

1(b). (3.13 from textbook) A ***Turing machine with stay put instead of left*** is similar to an ordinary Turing machine except that the transition function has the form

$$\delta : Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\text{R, S}\}.$$

At each point the machine can move its head right or let it stay in the same position. Show that this Turing machine is *not* equivalent to the usual version. What class of languages do these machines recognize?

Solution To show that a Turing machine with stay put instead of left is not as powerful as (and therefore, not equivalent to) the usual version, we must show there exists a language that can be recognized by a standard Turing machine but cannot be recognized by a Turing machine with stay put instead of left. One such language is the context-free language

$$\{a^n b^n \mid n \geq 0\}.$$

A Turing machine with stay put instead of left cannot go back to read what it wrote before; as a result, it cannot keep track of how many a's it has read in order to match them with the b's. In fact, this kind of Turing machine is no more powerful than a DFA, and can only recognize the class of regular languages.

2. Write C-like pseudocode to recursively enumerate all polynomials in $\mathbb{Z}[x, y]$ which have a positive integral root.

Solution The main function that performs the enumeration is called enumerate().

The following are descriptions of the functions:

traverse_xy(n) returns the n-th pair of values for x and y in a traversal order similar to the one used in Figure 4.3 on page 162 of the textbook.

test(int i, polynomial p) evaluates p, a polynomial in $\mathbb{Z}[x, y]$, using each of the first i values of x and y given by traverse_xy() to check whether p has a positive integral root.

lexi_poly(n) returns the n-th polynomial in $\mathbb{Z}[x, y]$ in a lexicographical order. (Polynomials in $\mathbb{Z}[x, y]$ can be sorted by number of terms and the coefficient and exponents of x and y in each term.)

enumerate() calls test() on every polynomial in $\mathbb{Z}[x, y]$ and prints out the ones that are recognized.

```
bool test(int i, polynomial p) {
   for (j = 0; j < i; j++) {
      evaluate p with x and y set to traverse_xy(j);
      if the value of p is 0, return TRUE;
   }
   return FALSE;
}

void enumerate() {
   i = 1;
   while(i > 0) {
      for (j = 1; j <= i; j++) {
         if (test(i, lexi_poly(j))) {
            print (lexi_poly(j));
         }
      }
      i++;
   }
}
```

On the next page, Professor Peralta provides details of how to write the traverse_xy() and lexi_poly() functions. The latter cleverly uses the Gödel numbering system for sequences of integers.

ENUMERATING POLYNOMIALS

RENE

1. basic idea

```
for (S = 0; true; S++)
  for (r = 0 ; r <= S; r++)
    for (i = 0; i <= S; i++)
      if ((i+r) == S)
        if r is a root of ith polynomial print the polynomial
```

The above is written so that it is easy to see that any polynomial with an integer root will eventually be printed. Some polynomials will be printed more than once, but that is not a problem.

2. the ith polynomial

We would like to associate a unique positive integer with each bivariate polynomial. For simplicity we will ignore the 0 polynomial. Here is an example of a polynomial:

$$17x^2y + xy^2 - 3x^2 + 2y - 13$$

We will order the terms by degree and then lexicographically. We will also add the terms with coefficient 0. The above polynomial is thus

$$(-13) + 0x + 2y + (-3)x^2 + 0xy + 0yx + 0y^2 + 0x^3 + 1x^2y + 17xy^2$$

The point of doing the above is that the polynomial is uniquely identified by the sequence of integers

$$-13, 0, 2, -3, 0, 0, 0, 0, 1, 17.$$

So we have reduced the problem of defining a correspondence between integers and polynomials to the problem of defining a correspondence between integers and finite integer sequences. There are many ways to do the latter. Here is one:

- encode a non-negative integer by i by $2i$ and a negative integer i by $2i + 1$. This reduces the problem to that of representing finite sequences of non-negative integers.
- the sequence $s_1, s_2, ..., s_k$ of non-negative integers is uniquely encoded by the integer
$$2^{s_1}3^{s_2}5^{s_3}...p_k^{s_k}$$
where p_k is the k^{th} prime.

So the polynomial

$$(-13) + 0x + 2y + (-3)x^2 + 0xy + 0yx + 0y^2 + 0x^3 + 1x^2y + 17xy^2$$

is represented by the integer

$$2^{27}5^4 7^7 (23)^2 (29)^{34}.$$

3(a). (4.10 from textbook) Let

$$A = \{\langle M \rangle \mid M \text{ is a DFA that doesn't accept any string containing an odd number of 1's}\}.$$

Show that A is decidable.

Solution Given input $\langle M \rangle$, our TM will construct the DFA M_2 that recognizes the language

$$L(M) \cap L(M_1),$$

where $L(M_1) = \{w \mid w \text{ contains an odd number of 1's}\}$. Notice that $L(M_1)$ is a regular language. It follows that $L(M) \cap L(M_1)$ is regular, because the regular languages are closed under intersection. Thus, by definition of a regular language, M_2 exists.

By Theorem 4.4, we can give our TM the ability to decide whether $L(M_2) = \emptyset$. If so, it returns "yes"; if not, it returns "no". Therefore the language

$$A = \{\langle M \rangle \mid M \text{ is a DFA that doesn't accept any string containing an odd number of 1's}\}$$

is decidable.

3(b). (4.12 from textbook) Show that the problem of testing whether a CFG generates some string in 1^* is decidable. In other words, show that

$$\{\langle G \rangle \mid G \text{ is a CFG over } \{0,1\}^* \text{ and } 1^* \cap L(G) \neq \emptyset\}$$

is a decidable language.

Solution Given input $\langle G \rangle$, our TM will construct the CFG G_2 that generates the language

$$L(G) \cap L(M),$$

where $L(M) = 1^*$. Notice that $L(M)$ is a regular language. It follows that $L(G) \cap L(M)$ is context-free, because by problem 2.17(a) the context-free languages are closed under intersection with a regular language. Thus, by Theorem 2.6, G_2 exists.

By Theorem 4.7, we can give our TM the ability to decide whether $L(G_2) = \emptyset$. If so, it returns "no"; if not, it returns "yes". Therefore the language

$$\{\langle G \rangle \mid G \text{ is a CFG over } \{0,1\}^* \text{ and } 1^* \cap L(G) \neq \emptyset\}$$

is decidable.