

## A useful fact

We first state and prove a fact that will be quite useful in the solutions below.

*Proposition:* Given integers  $m$  and  $s$  coding a Turing machine  $M$  with instantaneous description  $S$ , it is primitive recursive (i.e., computable by a FOR program) to determine the code  $s'$  of the ID  $S'$  of  $M$  after it executes for 1 step, or set  $s'$  to  $s$  if  $M$  has halted.

*Proof:* We will describe a FOR program to compute  $s'$ . It first decodes<sup>1</sup>  $s$  to determine the tape  $\tau$ , the state  $q$  of the finite control of  $M$ , and the symbol  $c$  under the read head (and its position  $p$ ) in the tape. Then,  $m$  is decoded to determine the transition that applies. From this, we can determine the new position  $p'$  of the head, the new state  $q'$  of the finite control, and the new tape  $\tau'$ . Encode these to form  $s'$ . (If  $q$  was a final state, the program can just execute  $s' \leftarrow s$ .) This completes the program.

We'll refer to this program by  $P$  and assume that it takes inputs in two variables  $m$  and  $s$ , and puts its output in a variable  $s'$ .

## Exercise 1: Partial recursive functions

Since anything computable by a WHILE program can be computed by a Turing machine, we assume that we have a Turing machine  $M$  that is equivalent to the given WHILE program  $W$ . Without loss of generality, assume that  $W$  takes inputs in variables  $a_1, \dots, a_n$  and put its output in variables  $b_1, \dots, b_k$ . We can assume that the input to  $W$  is turned into a string  $a_1Ba_2B \dots Ba_n$  (where  $B$  is the blank symbol) to form the input to  $M$ ; we do similarly for the output. Determine the integer  $x$  that codes  $M$ .

We can now describe a WHILE program that is equivalent to  $W$  but contains only 1 *while* loop. This program will use the same input and output variables as  $W$ . The first thing it does is set variable  $m$  to zero, and then increment it  $x$  times. So  $m$  contains the code of  $M$ . Then, based on  $a_1, \dots, a_n$ , it computes the code  $s$  of the instantaneous description of  $M$  in its initial state. This doesn't require *while* loops since decoding  $m$  and encoding  $s$  is primitive recursive. The program then does the following:

```
P;  
while s ≠ s' do  
  s ← s';  
  P  
done
```

The last thing the program does is to decode  $s$  into the output variables  $b_1, \dots, b_k$ . This program clearly has only one *while* loop, and it is equivalent to  $W$  since it essentially runs  $M$  on its inputs until  $M$  halts (which does not necessarily happen).

---

<sup>1</sup>It was given that the encoding and decoding operations done throughout this program are primitive recursive

## Exercise 2: Primitive recursive functions

### Part 1: Turing machine $\rightarrow$ primitive recursive function

We're given a Turing machine  $M$  with a primitive recursive time-bound  $f$ , where  $M$  computes some function  $F$ . As in Exercise 1, we can determine the integer  $x$  that codes  $M$ , and then construct a program to simulate  $M$  on input  $x$ . The key difference is that since we have  $f$ , we essentially have a FOR program that we can use to compute a bound  $B = f(|x|)$ . (Note that we have to compute  $B$  as part of the program since it depends on the input.) So instead of a *while* loop, we can use a *for* loop, i.e.

```
P;  
for B do  
  s  $\leftarrow$  s';  
  P  
done
```

Since FOR programs are equivalent to primitive recursive functions, this shows that  $M$  computes a primitive recursive function.

### Part 2: Primitive recursive function $\rightarrow$ Turing machine

We're given a primitive recursive function  $F(x)$ . By induction on the derivation of  $F$ , we can construct a Turing machine time-bounded by a primitive recursive function  $f$ . (We omit a lot of the grungy details.)

*Base cases*

For the successor, zero, and projection functions, it is straightforward to construct Turing machines to compute each of these functions, and we can construct these Turing machines so that they have a primitive recursive time bound. (We don't want the machines to do something silly like compute Ackermann's function, throw away the result, and then compute, say, the zero function.)

*Inductive cases*

Here, we have some function  $f$  that is defined by either composition or primitive recursion. By the inductive hypothesis, we can assume that the functions  $f$  is derived from can be computed by Turing machines with a primitive recursive time bound. It's straightforward then to construction a machine that computes  $f$ , but it takes some bit of work to show that the constructed machine actually has a primitive recursive time bound.