Models of Computation II, Exercises 3: TM & Comp. Functions

1. (a) M begins in the configuration $(start, \epsilon, 011␣111␣110)$. This means the machine looks like this:

		start ↓												
···	␣	0	1	1	␣	1	1	1	␣	1	1	0	␣	...

The state of the machine is "*start*", the only symbols to the left of the head are blank, and from the head onwards we have the data 011␣111␣110. Notice that the symbol currently under the head is 0. We take the state ("*start*") and the symbol under the head (0), and look them up in the program table. It says $(skip1s, 0, R)$. So, we should replace the symbol currently under the head with 0[1], we should move the head to the Right, and we should proceed in state "*skip*1*s*". M is now in this configuration: $(skip1s, 0, 11␣111␣110)$ which looks like:

			*skip*1*s* ↓											
···	␣	0	1	1	␣	1	1	1	␣	1	1	0	␣	...

The table tells us $(skip1s, 1, R)$, so we transition to $(skip1s, 01, 1␣111␣110)$ which looks like:

				*skip*1*s* ↓										
···	␣	0	1	1	␣	1	1	1	␣	1	1	0	␣	...

You don't have to explain every step of the machine in this amount of detail. But for a question like this, you do have to tell us what those steps are. Your answer should look

[1] this is easy, since it is already 0

like the following. M performs the computation:

$$\begin{aligned}
(start, \epsilon, 011\sqcup 111\sqcup 110) &\to_M (skip1s, 0, 11\sqcup 111\sqcup 110)\\
&\to_M (skip1s, 01, 1\sqcup 111\sqcup 110)\\
&\to_M (skip1s, 011, \sqcup 111\sqcup 110)\\
&\to_M (skip{\ast}s, 011{\ast}, 111\sqcup 110)\\
&\to_M (shift, 011, {\ast\ast}11\sqcup 110)\\
&\to_M (shift, 01, 1{\ast\ast}11\sqcup 110)\\
&\to_M (append, 011, {\ast\ast}11\sqcup 110)\\
&\to_M (skip{\ast}s, 0111, {\ast}11\sqcup 110)\\
&\to_M (skip{\ast}s, 0111{\ast}, 11\sqcup 110)\\
&\to_M (shift, 0111, {\ast\ast}1\sqcup 110)\\
&\to_M (shift, 011, 1{\ast\ast}1\sqcup 110)\\
&\to_M (append, 0111, {\ast\ast}1\sqcup 110)\\
&\to_M (skip{\ast}s, 01111, {\ast}1\sqcup 110)\\
&\to_M (skip{\ast}s, 01111{\ast}, 1\sqcup 110)\\
&\to_M (shift, 01111, {\ast\ast}\sqcup 110)\\
&\to_M (shift, 0111, 1{\ast\ast}\sqcup 110)\\
&\to_M (append, 01111, {\ast\ast}\sqcup 110)\\
&\to_M (skip{\ast}s, 011111, {\ast}\sqcup 110)\\
&\to_M (skip{\ast}s, 011111{\ast}, \sqcup 110)\\
&\to_M (skip{\ast}s, 011111{\ast\ast}, 110)\\
&\to_M (shift, 011111{\ast}, {\ast\ast}10)\\
&\to_M (shift, 011111, {\ast\ast\ast}10)\\
&\to_M (shift, 01111, 1{\ast\ast\ast}10)\\
&\to_M (append, 011111, {\ast\ast\ast}10)\\
&\to_M (skip{\ast}s, 0111111, {\ast\ast}10)\\
&\to_M (skip{\ast}s, 0111111{\ast}, {\ast}10)\\
&\to_M (skip{\ast}s, 0111111{\ast\ast}, 10)\\
&\to_M (shift, 0111111{\ast}, {\ast\ast}0)\\
&\to_M (shift, 0111111, {\ast\ast\ast}0)\\
&\to_M (shift, 011111, 1{\ast\ast\ast}0)\\
&\to_M (append, 0111111, {\ast\ast\ast}0)\\
&\to_M (skip{\ast}s, 01111111, {\ast\ast}0)\\
&\to_M (skip{\ast}s, 01111111{\ast}, {\ast}0)\\
&\to_M (skip{\ast}s, 01111111{\ast\ast}, 0)\\
&\to_M (tidyup, 01111111{\ast}, {\ast}\sqcup)\\
&\to_M (tidyup, 01111111, {\ast}\sqcup\sqcup)\\
&\to_M (tidyup, 0111111, 1\sqcup\sqcup\sqcup)\\
&\to_M (finalise, 01111111, \sqcup\sqcup\sqcup)\\
&\to_M (finalise, 0111111, 10\sqcup\sqcup)
\end{aligned}$$

Note: If, during the course of a computation, the machine were to remain in the same state, with the same symbol under the head for more than three steps, you could write out the first and last step in full, and put "..." in the middle. You can see this in Slide 11 of your lecture notes. The machine M in this question only ever stayed in the same state for at most 3 steps, so you wouldn't have saved any space with that trick here.

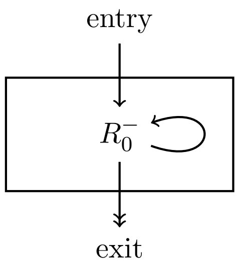
(b) In your computation for part (a), you observed the machine taking the list $[2, 3, 2]$ and calculating the final value $[7]$. In fact, the machine M takes a non-empty list of n natural numbers $[x_1 \dots x_n]$, and produces a singleton list containing only the sum of the numbers in the input list $[\sum_{i=1}^{n} x_i]$.

If you're ever confronted with a TM and need to know what it does, you can apply many of the debugging techniques you're used to in other programming languages. Try it with a variety of inputs. Try it with "corner case" inputs, like the empty list[2]. If you trust the author of the TM, and if the states or symbols have suggestive names (such as "tidy up" or "1"), then pay attention to the clues those names gave you[3]. Consider the control flow of the code. In a programming language such as Haskell or Java, this often means a graph of function calls. In a system such as register machines, this means the graph of transitions from one program location to another. In Turing machines, this means the transitions from one state to another. Spot the loops, the entry points, and the exit points. Try to figure out what properties the loops maintain.

2. Define $M = (Q, \Sigma, s, \delta)$ where $\Sigma = \{\sqcup, 0, 1\}$, $Q = \{s, t\}$, and δ is given by:

δ	$\sqcup$	0	1
s	$(t, 1, R)$	$(t, 1, R)$	$(s, 0, R)$
t			

3. (a) To implement the successor machine with Minsky machines, we define gadgets to implement each of the instructions. Clear R_0:



Increment R_0:

[2]pat yourself on the back if you spotted that M sums non-empty lists but leaves empty ones empty

[3]On the other hand, if you don't trust the author of the code (if you're checking for malware for example) then absolutely do not pay attention to any of their variable names, function names, comments, etc. If you don't understand why not, see the following URLs: `http://www.ioccc.org/2000/primenum.c` `http://www.ioccc.org/2000/primenum.hint`

$$\begin{tikzcd}
\text{entry} \arrow[d] \\
R_0^+ \arrow[d] \\
\text{exit}
\end{tikzcd}$$

Test $R_0 = R_1$:

$$\begin{tikzcd}
 & & \text{entry} \arrow[d] & & & & \\
T^+ \arrow[rr] & & R_0^- \arrow[dl] \arrow[dr, two heads] & & & & \\
 & R_1^- \arrow[ul] \arrow[d, two heads] & & R_1^- \arrow[r, two heads] \arrow[d] & T^- \arrow[rr, two heads] \arrow[d] & & \text{equal} \\
 & R_0^+ \arrow[dr] & & R_1^+ \arrow[dl] & R_1^+ \arrow[r] & R_0^+ \arrow[ul] & \\
 & & T^- \arrow[rrrr, two heads] \arrow[d] & & & & \text{not equal} \\
 & R_0^+ \arrow[ur] & R_1^+ \arrow[l] & & & &
\end{tikzcd}$$

Conversely, we can implement the Minsky machine instructions using Successor machine gadgets. Increment R_0:

$$\begin{tikzcd}
\text{entry} \arrow[d] \\
R_0^+ \arrow[d] \\
\text{exit}
\end{tikzcd}$$

Test R_0 for zero and decrement:

$$\begin{tikzcd}
 & & \text{entry} \arrow[d] & & & & \\
R_0 \text{ is zero} & & R_0 = T? \arrow[ll, "\text{yes}"'] \arrow[d, "\text{no}"] & & & & \\
 & & T^+ \arrow[d] & & & & \\
 & S^+ \arrow[ur] & R_0 = T? \arrow[l, "\text{no}"] \arrow[d, "\text{yes}"] & & R_0^+ \arrow[d, bend right] & & \\
 & & \text{Clear } R_0 \arrow[r] & \text{Clear } T \arrow[r] & R_0 = S? \arrow[u, bend right, "\text{no}"'] \arrow[r, bend right, "\text{yes}"'] & \text{Clear } S \arrow[r] & R_0 \text{ decremented}
\end{tikzcd}$$

(b) The Church-Turing thesis is that anything that is algorithmically computable is computable by a Turing machine. Anything that is computable by a Minksy machine is computable by a Turing machine, and hence anything that is computable by a Successor machine is also computable by a Turing machine. The Successor machine can reasonably seen as a formalisation of an algorithm, which could be implemented. If it could compute anything that is not Turing-computable, it would falsify the Church-Turing thesis. The fact that it cannot therefore supports the thesis.

(c) If the halting problem for Successor machines were decidable, so too would be the halting problem for Minsky machines, which we have seen is not decidable. Hence the halting problem is not decidable.

4. (a) Given a register machine M, we know that the partial function of one argument that it computes is also Turing computable (Slide 19). This means that we can construct a Turing machine T that, on the input 0_0 will halt exactly when M halts with all registers initially 0. It is easy to construct a Turing machine T' that writes 0_0 onto an empty tape and then starts executing T with the head at the left-most 0. In fact, the function that given the code of M computes the code of the corresponding T' should be computable. (Effectively, such a program is a compiler.)

Suppose that we have a decision procedure[4] for the set of numbers corresponding to the Turing machines which eventually halt when run on the empty input. Then we can use this to decide the halting problem for register machines by combining it with the register-

[4] A decision procedure for a set is a register machine (or equivalently a Turing machine) that computes the characteristic function for the set.

machine-to-Turing-machine compiler above. But the halting problem is undecidable, so there can be no such decision procedure, and the set is undecidable.

(b) The set of satisfiable first-order sentences must be undecidable. If it were decidable, we would have a procedure for deciding the halting problem for Turing machines: first compute the formula corresponding to the Turing machine, then determine if it is satisfiable, and hence whether the Turing machine halts. Since the halting problem is not decidable, there can be no decision procedure for the set of satisfiable first-order sentences.

(c) The set of valid first-order sentences is semidecidable. Consider a machine that, given a sentence of first-order logic (encoded as the number s) starts at $p = 0$ and determines if (s, p) is a valid sentence-proof-pair. If it is, it halts; if not, it increments p and loops, checking the next candidate proof.

If s does have some proof, which must have some number, say p', the machine will eventually halt. If s has no proof, the machine will continue to run forever.

The set of valid first-order sentences is not decidable. A sentence F is satisfiable if and only if the sentence $\neg F$ is not valid. Therefore, if we had a decision procedure for first-order sentences, we would also have a decision procedure for satisfiable sentences (by simply feeding in the complement). From the first part, we know that this is not possible.

5. One problem with the idea of a perfect virus scanner is that the concept of a virus is poorly defined. For instance a program that overwrote your master boot record (MBR) might be considered a virus; however, there are legitimate reasons for overwriting the MBR, so when should a program that overwrites the MBR be considered a virus? However, let's simplify things and make the following assumptions:

- There is some operation such that, if a program performs it then it should be considered a virus.
- A program can run for ever, and use an unbounded amount of memory, without being considered a virus.

Given a register machine, we can write a program that simulates that machine. We make our program perform the illegal operation once the register machine halts. If the register machine never halts, the program should not be considered a virus, since it will never execute the illegal operation. If it does halt, then the program should be considered a virus. A perfect virus scanner would therefore be able to solve the halting problem for register machines, which should be impossible by the Church-Turing thesis.

For a more direct approach, assuming that the virus scanner is not itself a virus(!), we could construct a program P using the virus scanner code V to check if V identifies P as a virus. If it does, then it simply exits normally; otherwise it performs the illegal action. Thus P is a virus if and only if V does not identify it as one, and so V cannot be a perfect virus scanner.