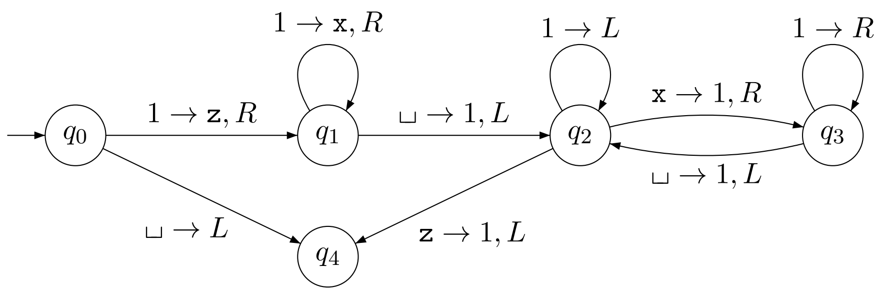
THE UNIVERSITY OF MELBOURNE
SCHOOL OF COMPUTING AND INFORMATION SYSTEMS
COMP30026 MODELS OF COMPUTATION

Selected Tutorial Solutions, Week 12

84. Here is the Turing machine D:



Note that, for the two transitions into q_4, the 'L' has no effect, since, at that point, the tape head is positioned over the leftmost tape cell. The machine doubles the number of ones that it finds on the tape.

85. Assume that H is a decider for the language

$$\mathit{Halt}_{TM} = \{\langle M, w\rangle \mid M \text{ is a Turing machine and } M \text{ halts when run on input } w\}$$

The following is a decider for A_{TM}:

```
On input $\langle M, w\rangle$:
    run $H$ on $\langle M, w\rangle$;
    if $H$ rejects, reject;
    else run $M$ on $w$ and accept $x$ if $M$ accepts $w$;
    else reject.
```

Since H halts for all input, the above Turing machine also halts, and it accepts if and only if M accepts w. Since we know that A_{TM} is undecidable, we conclude that Halt_{TM} is undecidable as well.

86. Yes, $\{\langle G\rangle \mid G \text{ is a CFG over } \{\mathtt{0}, \mathtt{1}\} \text{ and } L(G) \cap \mathtt{1}^* \neq \emptyset\}$ is decidable. Since $\mathtt{1}^*$ is regular, the intersection $L(G) \cap \mathtt{1}^*$ is context-free, and we can build a context-free grammar G' for it. We then run the emptiness decider from Lecture 20 on G'. If that decider accepts, we reject, and vice versa.

87. Here is how the 2-PDA recogniser for B operates:

   (a) Push a \$ symbol onto stack 1 and also onto stack 2.

   (b) As long as we find an `a` in input, consume it and push an `a` onto stack 1.

   (c) As long as we find a `b` in input, consume it and push a `b` onto stack 2.

   (d) As long as we find a `c` in input, consume it and pop both stacks.

   (e) If the top of each stack has a \$ symbol, pop these.

   (f) If we got to this point and the input has been exhausted, accept.

If the 2-PDA got stuck at any point, that meant reject.

88. To simulate M running on input $x_1 x_2 \cdots x_n$, the 2-PDA P first pushes a $\$$ symbol onto stack 1 and also onto stack 2. It then pushes $x_n, x_{n-1}, \ldots x_2, x_1$ onto stack 2, in that order. P is now ready to simulate M. Note that it has consumed all of its input already, but it is not yet in a position to accept or reject.

    For each state of M, P has a corresponding state. Assume P is in the state corresponding to M's state q.

    For each M-transition $\delta(q, a) = (r, b, R)$, P has a transition that pops a off stack 2 and pushes b onto stack 1. If stack 2 now has $\$$ on top, P pushes a blank symbol onto stack 2. Then P goes to the state corresponding to r.

    For each M-transition $\delta(q, a) = (r, b, L)$, P has a transition that first pops a off stack 2, replacing it by b. It then pops the top element off stack 1 and transfers it to the top of stack 2, unless it happens to $\$$. And then of course P goes to the state corresponding to r.

    If this seems mysterious, try it out for a simple Turing machine and draw some diagrams along to way, with snapshots of the Turing machine's tape and tape head next to the 2-PDA's corresponding pair of stacks. The invariant is that what sits on top of the 2-PDA's stack 2 is exactly what is under the Turing machine's tape head at the corresponding point in its computation.

89. Assume $\mathcal{B}$ is countable. Then we can enumerate $\mathcal{B}$:

Element								
b_1	$\boxed{0}$	0	1	0	1	1	1	...
b_2	1	$\boxed{0}$	1	1	1	0	1	...
b_3	1	0	$\boxed{1}$	1	1	0	1	...
b_4	0	1	0	$\boxed{0}$	1	0	0	...
$\vdots$								

    However, the infinite sequence which has

$$i\text{'th bit} = \begin{cases} 0 & \text{if the } i\text{th bit of } b_i \text{ is } 1 \\ 1 & \text{if the } i\text{th bit of } b_i \text{ is } 0 \end{cases}$$

    is different form each of the b_i. Hence no enumeration can exist, and $\mathcal{B}$ is uncountable. This should not be surprising, because the set $\mathcal{B}$ is really the same as (or is isomorphic to) $\mathbb{N} \to \Sigma$.

90. First, simulating a Turing machine move $x \to y, R$ is easy; simply dequeue x and enqueue y. For example, assume the Post machine's queue holds `a c $ b a` and we want to simulate the TM move `a` $\to$ `b`, R. After dequeueing `a` and enqueueing `b`, the queue holds `c $ b a b`. This reflects the fact that the TM tape contents changed from `b a a c` to `b a b c` and the tape head moved from being positioned over the last `a` to the `c`.

    A move $\sqcup \to x, R$ needs special treatment; we only need to enqueue x, nothing more.

    Simulating a move $x \to y, L$ is more tricky. We won't give a formal definition here; we just sketch the idea through an example. The Post machine will need to use a couple of internal symbols for its own bookkeeping.

    Again assume the Post machine's queue holds `a c $ b a` and now we want to simulate the TM move `a` $\to$ `b`, L. We want to change the `a` in the front of the queue to `b` and bring the `a` from the end of the queue up to the front. That is, we want this transformation:

$$\texttt{a c \$ b a} \quad \rightsquigarrow \quad \texttt{a b c \$ b}$$

However, we need to make that happen with enqueueing and dequeueing operations only. We could dequeue `a`, enqueue `b`, and then make three "left rotations", where a left rotation dequeues symbol x and enqueues it. But how does the Post machine know how many left rotations to make?

Here is the idea for how that is done. Start by adding another marker, #. Ideally we would like to put this marker just before the last `a` in `a c $ b a`, to indicate where left rotations should stop. However, we can't put the marker there. All we can do is to enqueue it, that is, place it after the last `a`:

`a c $ b a #`

Then we dequeue `a` and enqueue `b`:

`c $ b a # b`

Now the trick is to do the left rotations, but in a "delayed" fashion. That is, we dequeue the `c` and go to a state whose purpose is to remember that we still need to enqueue a `c`. When we next see a symbol which is not the #, we do this enqueueing:

`$ b a # b c`

A $ can just be rotated:

`b a # b c $`

On a `b`, again we do "delayed" rotation, that is, dequeue it and go to a state which remembers that we are holding a `b`:

`a # b c $`

Once we see the `a`, we know that we should just have enqueued the `b`, so we go ahead and do that:

`a # b c $ b`

Again, on an `a`, we do "delayed" rotation, that is, dequeue it and go to a state which remembers that we are holding an `a`:

`# b c $ b`

Now we see the value of the delayed enqueueing. At this point we know that we are at the new tape head position, or rather, we *were*—the `a` should not really have been dequeued. However, now we are in a position to fix things: Dequeue the # (it gets thrown away), enqueue a different marker, say !, *before* enqueueing the `a`, to get to this queue:

`b c $ b ! a`

Now all we need to do is perform a sequence of left rotations until the ! comes to the front of the queue, then throw it away. That leaves us with

`a b c $ b`

as desired.