Due: Thursday 10/13/2022

1. Create a Turing Machine for the language of $b = w$ over alphabet $\Sigma = \{0, 1, =, a\}$ where b is a binary number and w is a sequence of as who length is the binary number. For example $101 = aaaaa$ is in the language as 101 is 5 in binary and $aaaaa$ has exactly 5 as. Likewise, $0 =$ is in the language, as there are 0 as on the right side of the equal sign. We leave it up to you whether $=$ by itself is in the language (*i.e.*, you can choose to treat ε on the left of the $=$ as 0, or as invalid input). Note that this language is infinite, as it includes binary numbers b of any bitlength, as long as there are an appropriate number of as after the $=$, so *e.g.*, $10^{50} = a^{2^{50}}$ (*i.e.* a 1 followed by 50 0s, then an $=$ then 2^{50} as) is in the language.

   For this problem you should *either* draw a clearly labeled state diagram OR show the transitions as a table. You are welcome (and encouraged but not required) to submit a brief description of your algorithm/explanation of how your TM works, but are not required to. Doing so may help your TA understand your TM and give partial credit if needed.

   Your TM may be multi-tape if you wish, and may use any reasonable number of tapes (I don't see how you would want more than 4). You may use "stay" (S) and "don't write anything" (we used $\bigcirc$ in class, but you can use any notation you want as long as you are clear).

2. Consider for a moment the language F which is the set of strings that are syntactically valid Python code for a function requiring no parameters (*e.g*, `def f():...`). Observe that F is decidable

(the Python interpreter decides this all the time). Now consider $R \subset F$ which is the subset of Python program code for zero-parameter functions, such that execution of the function (*e.g.*, calling `f()`) throws an exception (of any type).

- Show that R is recognizable by explaining how to recognize this language.
- Show that R is not decidable by showing an appropriate reduction from a well-known undecidable problem.

3. Are *Un*decidable languages closed under intersection? Put another way, is it true that

$$\forall L_1, L_2.\text{Undecidable}(L_1) \wedge \text{Undecidable}(L_2) \Rightarrow \text{Undecidable}(L_1 \cap L_2)$$

   If this is true, then write a proof. If this is not true, then give a counter example (show me two languages that are undecidable, and show me that their intersection is decidable).

4. **Implementation** Your first implementation task is to write Python code which executes a Turing Machine. As with the previous homework, we have given you the code (in `tm.py`) to parse the Turing Machine description from a file. Your goal is to write `executeTM` which takes three parameters:

   tm is the TuringMachine object which contains the states and transition information.

   s is the input string to execute this TuringMachine on

   verbose is a boolean indicating if the execution should give verbose output. If this boolean is true, you should print the state and contents of the tape before each step. Note that we give you `printTmExec` which prints this information in

the proper format. If you structure your tapes as a list of lists, and track the tape positions as a list of integers, you can just call this function. If you do something else, please adjust this function but keep the formatting.

As you learned in class, there are many equivalent variations of Turing Machines. You are going to implement at "fancy" variation with a few "features" that make it much easier to write down the TM specifications:

- Multi-tape. Your Turing Machine supports an arbitrary number of tapes. How many tapes it has is specified at the start of the input fill with *e.g.*, `Tapes:3`. Note that as is common with multi-tape Turing Machines, the movement of each tape is independent and can be either "Left", "Right" or "Stay".
- Don't Write. Your Turing Machine supports "don't write anything". This feature is really just for simplicity in writing down the input specification, as it is equivalence to having rules that say "if there is an A, write an A. if there is a B, write a B..."
- Don't Cares. Our TM specification will let us say that we don't care what value is on a particular tape. *E.g.*, we might write `A,,C` to mean "there is an A on tape 0 and a C on tape 2, but we don't care what is on tape 1 (match anything)". This feature is just a simplicity in writing down the input specification as we don't have to go write down every possibility for Tape 1 to do the same action.
- In Order Matches (first found priority) with default Reject. When we write down our TM specification, we can write the

matches in order, with the first ones having priority over later ones. For example, if we wrote a rule for `A,,C` (as above) then `A,,` ("Tape 0 has an A, but we don't care about the other two tapes"), Then if Tape 0 has an A, and Tape 2 has a C, we use the `A,,C` rule which came first, but if Tape 0 has an A and Tape 2 has any other symbol, we use the later `A,,` rule. If no rules match we default to "Reject". This is also equivalent to a regular TM as it is just shorthand for writing down many equivalent rules, and for skipping writing down many "go to reject state" rules.

At this point, you might want to take a moment to look at three provided files:

- `tm.py` This is the Python file you will be working in. You can see that we wrote several classes (think of them as structs: we'll use them more as actual classes once you learn OOP in 551), and the code to parse the TM file.
- `doublePalindrome.tm` This is an example TM specification file. It makes a 19 state Turing Machine (states 0–18) which decides the language of "double palindromes" (ww^Rww^R) that contain only a, b, and c. Note that there is documentation on this file specification at the end of this homework writeup.
- `tm-example-exec.txt` is the result of executing the double palindrome Turing Machine with input `abccbaabccba`. In particular, I ran:

```
python3 tm.py doublePalindrome.tm abccbaabccba verbose > tm-example-exec.txt
```

A few things about your TM implementation:

- The Turing Machine should always start in state 0.

- Tape 0 of the Turing Machine should always start with the input string on it.
- Any tapes other than Tape 0 should start with a single blank.
- Attempting to move left past the end of the tape does nothing (it is not an error, and the tape position does not move).
- Attempting to move right past the end of the tape adds a `blank_char` and the tape position moves onto it.
- The Turing Machine should do any writes that it needs to *before* moving the tape position.
- In my implementation, I made a variable `tapes` which is a list of lists of characters. *E.g.*,

  ```
  tapes = [ ['A', 'b', 'c'], ['d', 'e'] ]
  ```

  would mean that Tape 0 has "Abc" and Tape 1 has "de". You can represent your tape some other way if you want, but this is what I did.
- In my implementation, I made a variable `tape_pos` which is a list of integers. This list keeps the position (list index) of each tape. *E.g.*,

  ```
  tape_pos = [2, 1]
  ```

  Would mean that Tape 0 is positioned at 2 (pointing at the 'c' in the above example tapes), and Tape 1 is positioned at 1 (pointing at the 'e' in the above example tapes). Again, you may do it some other way if you want.
- I wrote three helper functions that I found useful—these are described briefly in comments in `tm.py`. As with the other points, you don't have to abstract things this way, but I found it useful to do so.

Note that your TM's execution should match ours *exactly*. We should be able to run the same TM with the same input in "verbose" mode and see exactly the same states and tape contents at each step.

5. **Implementation** Rather than asking you to describe a Turing Machine in words, we are going to ask you to *implement* the Turing Machine itself—writing out the states and their actions in the file format accepted by your Python code above! As a quick reminder, there is a reference for the input file format at the end of this assignment

   In particular, you are going to write a Turing Machine that **decides** the language of strings containing As, Bs, and Cs such that num(A)s = num(Bs) + 2 * num(Cs). For example, if there are 3 Bs and 5 Cs in the string, there would have to be exactly 3 + 2 * 5 = 13 As in the string. If there is any other character in the input string, your Turing Machine should reject the input. Examples of some strings your TM should accept (with the math as to why in parenthesis after):

   ```
   (epsilon)           (0 = 0 + 2 * 0)
   AB                  (1 = 1 + 2 * 0)
   BA                  (1 = 1 + 2 * 0)
   CAA                 (2 = 0 + 2 * 1)
   ACAAAABAACC         (7 = 1 + 2 * 3)
   ```

   Examples of some strings your TM should accept (and why in parenthesis):

   ```
   ab                  (only A,B, C allowed)
   ABADAC              (only A,B, C allowed)
   ```

```
ABA                    (2 > 1 + 2 * 0)
ABAC                   (2 < 1 + 2 * 1)
```

For this part, you should submit a file `aEqBPlus2C.tm` which can be run in your Python program from the previous step. *E.g.*, we should be able to run

```
python3 tm.py aEqBPlus2C.tm ACAAAABAACC
```

and get a result of "True"

Your Turing Machine may use however many tapes you want. I found 3 to be convenient.

TM file specification The following is the specification for the `.tm` files read by your `tm.py` (and which you will write for the second implementation section).

- Lines containing only whitespace are ignored
- Lines *starting* with a `#` sign are comments to the end of the line
- The first non-comment line should be `Tapes:N` where N is the number of tapes.
- The second non-comment line should be `Blank:c` where c is the symbol you want to treat as "blank" (added to the end whenever you move off the right end of the tape).
- Each other line describes a state. All states start with `N:` where N is the number of that state. After the : can either be "Accept", "Reject" or a description of the actions in that state.
- The actions in a state are described by a semi-colon (`;`) delimited set of `match=transition` pairs. For example

`_,=18,,L;a,=3,R_,;b,=5,R_,;c,=7,R_,`

specifies 4 match=transition pairs.

1. `_,=18,,L;`
2. `a,=3,R_,`
3. `b,=5,R_,`
4. `c,=7,R_,`

- The `match` part of the match=transition pair specifies one symbol per tape in a comma-separated list. A symbol may be omitted to indicate "don't care" for that tape. In the first example above (`_,` the match specifies that Tape 0 must have an `_` and we do not care what is on Tape 1 (match anything). Note that the right number of commas are required when symbols are omitted for don't cares—if you have an N tape Turing Machine, you need (N-1) commas in the match.
- The `transition` part of the match=transition pair specifies the state to go to (by number), and then a comma, and one "tape action" per tape. The tape action consists of one character that must be either R, S, or L to specify in which direction to move on the tape, and a second character specifying what to write on the tape (before moving). If only one character is present, nothing is written to the tape (the character that is there is unchanged). If no characters are present, the movement is implicitly 'S'