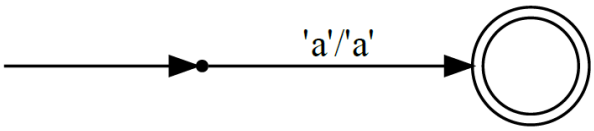
Final Report for the Universal Turing Machine

Author: Hongyu Zeng, Junnan Wang

1. Test cases that work:

a) TM1: x

```
x =
  TM [1, 2] "a" "a !" id ' ' '!' t2 1 [2]
  where
    t2 = goRight 1 'a' 'a' 2
```

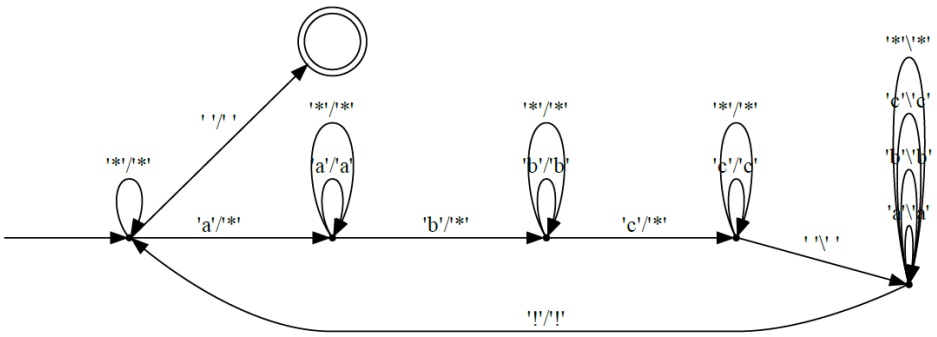


i. Test cases:

```
if accepts utm (inputU2 x "a") -- accepts
  then putStrLn "accepts"
  else putStrLn "rejected"
if accepts utm (inputU2 x "") -- rejects
  then putStrLn "accepts"
  else putStrLn "rejected"
if accepts utm (inputU2 x "b") -- rejects
  then putStrLn "accepts"
  else putStrLn "rejected"
```

1. Tested “a”, “b”, and “”, all behave as expected (See output of Main)

b) TM2: tripletm



i. Test cases:

```
if accepts utm (inputU2 tripletm "") -- accepts
  then putStrLn "accepts"
  else putStrLn "rejected"
if accepts utm (inputU2 tripletm "abc") -- accepts
  then putStrLn "accepts"
  else putStrLn "rejected"
if accepts utm (inputU2 tripletm "aabbcc") -- accepts
  then putStrLn "accepts"
  else putStrLn "rejected"
if accepts utm (inputU2 tripletm "bc") -- rejects
  then putStrLn "accepts"
  else putStrLn "rejected"
if accepts utm (inputU2 tripletm "a") -- rejects
  then putStrLn "accepts"
  else putStrLn "rejected"
if accepts utm (inputU2 tripletm "aab") -- rejects
  then putStrLn "accepts"
  else putStrLn "rejected"
```

ii. Tested “”, “abc”, “aabbcc”, “bc”, “a”, “aab”, all behave as expected (See output of Main)

c) Output of Main

```
➜  theOneTM git:(jw) ghc --make -O2 Main.hs
➜  theOneTM git:(jw) ./Main
accepts
rejected
rejected
accepts
accepts
accepts
rejected
rejected
rejected
➜  theOneTM git:(jw)
```

2. Debug functions (ntcs):

```haskell
-- debugging tool - ntcs
nt :: Config Integer Char -> [Config Integer Char]
nt = newConfigs utm -- take a config and step the list of next configs
-- obtain the next config (get rid of the bracket since we are having deterministic UTM)

ntc :: Config Integer Char -> Config Integer Char
ntc config = head (nt config)

-- debugging fxn:
-- input: number of steps as n; the config to begin with
--note: the iterate function actually can run to the end:  iterate ntc config
--for example: iterate ntc initial1  -- a little weird though...
ntcs :: Int -> Config Integer Char -> Config Integer Char
ntcs n config = last $ take (n + 1) (iterate ntc config) -- the first element after iterate function is the current config

-- input: number of steps and the input tape
ntcsFromStart :: Int -> [Char] -> Config Integer Char
ntcsFromStart n input = ntcs n (initialConfig utm input)
```

a) The function **ntcs** takes a number n and a specific configuration of tape.
   i. n: number of steps for utm
   ii. config is any configuration of the utm with Integer as states and Char as symbols

b) Functionality: this method is able to step forward a certain number of steps to check the next configuration of our utm.

3. Tests that are not working:

a) We have not encountered a failed case.

4. High-level Description and Walk-Through:

a) Tape composition and Encoding Scheme with justifications:

i. UTM tape symbols: " !;:$,._tuvwxyz10"

ii. The UTM tape consists of 4 parts. Encoded blank symbol of the input TM, encoded final states of the input TM, encoded transitions of the input TM, and encoded configurations of the TM on each step.

iii. The transitions are 5-tuples (encoded): current state, current symbol TM pointing at, next symbol to put-on, direction to move, next state

iv. The configs are 4-tuples (encoded): left part of the TM, current state, current symbol, right part of the the TM

v. Each tape symbol of the input TM is encoded in binary. Note that 1s and 0s are followed by underscore "_", and each symbol has an additional "." followed by "_". This is helpful for later use when we want to form new configs based on transition and current config. We need a way to get the previous symbol and the next symbol when we move left/right

vi. Each state symbol of the input TM and the direction are also encoded in binary and has "_" inserted after each 1/0. Note that there is not additional "._" in front of these two.

vii. Separation of each part: Final states each starts with "$_", transitions all have ";_" at the beginning, and configs are separated with transitions by ":_" and each has an ":_" at the beginning.

viii. Blank symbol is needed as if we keep moving right on input TM, we need to append blanks from the infinite-list-of-blanks to the last part of the input config.

b) Key ideas:

i. Label the current state q and tape read head S.

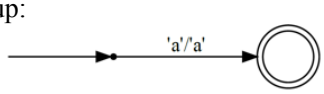
ii. Compare to all transitions. If any matches, execute the transition and form a new config (append to the end), go from step i again

iii. If no transition matched (dead end), check if it reaches a final state by comparing to all final states. We assume that there is no more transitions if reached the final states on the input TM.

c) More details with a sample run:

i. Example TM x:

a) Setup:




```
x =
  TM [1, 2] "a" "a !" id ' ' '!' t2 1 [2]
  where
    t2 = goRight 1 'a' 'a' 2

ex1 = inputU2 x "a"
```

b) Config explanation

"._1_0_0_0_0_0_$_1_0_;_1_,._1_1_0_0_0_0_1_,._1_1_0_0_0_0_1_,0_,1_0_:_:_._1_0_0_0_0_0_1_,1_,._1_1_0_0_0_0_1_,._1_0_0_0_0_0_"

	symbol before read head	current state	read head	symbol after read head
original	!	1	a	blank
encode	1_0_0_0_0_1_	1	1_1_0_0_0_0_1_	1_0_0_0_0_0_

ii. Step0: enter the start state

```
[*Test> ex1                                                                    ]
"._1_0_0_0_0_0_$_1_0_;_1_,._1_1_0_0_0_0_1_,._1_1_0_0_0_0_1_,0_,1_0_:_:_._1_0_0_0
_0_1_,1_,._1_1_0_0_0_0_1_,._1_0_0_0_0_0_"
[*Test> initial1                                                               ]
[1: "!." "_1_0_0_0_0_0_$_1_0_;_1_,._1_1_0_0_0_0_1_,._1_1_0_0_0_0_1_,0_,1_0_:_:_.
_1_0_0_0_0_1_,1_,._1_1_0_0_0_0_1_,._1_0_0_0_0_0_"]
```

iii. Step1: Label current state q and read head S
   1. Locate the state and read head at the current configuration part

```
[*Test> config_mark_qS                                                    ]
[5: "!._1_0_0_0_0_0_$_1_0_;_1_,._1_1_0_0_0_0_1_,._1_1_0_0_0_0_1_,0_,1_0_:_:_._1_
0_0_0_0_1_" ",1y,.y1y1y0y0y0y0y1y,._1_0_0_0_0_0_"]
```

      a) The current configuration (newest) is always at the end. So we can find the current state and current read head from the current config part.
      b) The current state and read mark are marked as y.

iv. Step2: Match the current state and read head to one of the transitions
   1. The idea is that if matched config with a transition, we can update the config. If no match, then check if reaching the final state (Assume input TM is deterministic)

   2. 2-1: mark the state and tape symbol for a transition

```
[*Test> config_mark_trans                                                 ]
[9: "!._1_0_0_0_0_0_$_1_0_;_1x,.x1x1x0x0x0x0x1x,." "_1_1_0_0_0_0_1_,0_,1_0_:_:_.
_1_0_0_0_0_1_,1y,.y1y1y0y0y0y0y1y,._1_0_0_0_0_0_"]
```

      a) The state and tape symbol for a transition is marked as x
   3. 2-2: compare by checking back and forth

```
[*Test> config_matched                                                    ]
[26: "!." "_1_0_0_0_0_0_$_1_0_;_1_,._1_1_0_0_0_0_1_,._1_1_0_0_0_0_1_,0_,1_0_:_:_
._1_0_0_0_0_1_,1_,._1_1_0_0_0_0_1_,._1_0_0_0_0_0_"]
```

      a) This is after the comparison. In this case, the transition matched the current config.
      b) The labels are erased, and ready to do the update of config

v. Step3: Update config if matched in step 2
   1. The idea here is to obtain the new state and new tape symbol from the matched transition, then form a new configuration. We append the new configuration at the end of the tape.
   2. 3-1: mark the transition and current config:

```
[*Test> config_update                                                     ]
[44: "!._1_0_0_0_0_0_$_1_0_;_1_,._1_1_0_0_0_0_1_,.u1u1u0u0u0u0u1u,0u,1y0y:_:_.x1
x0x0x0x0x1x,1_,._1_1_0_0_0_0_1_,.z1z0z0z0z0z0z: " ""]
```

      a) As shown in here, the matched transition is marked by u and y for new tape symbol, direction, and new state.
      b) Also the current config is marked by x and z
   3. 3-2: update the config (appending to the end)

```
[*Test> config_finish_update                                              ]
[156: "!._1_0_0_0_0_0_$_1_0_;_1_,._1_1_0_0_0_0_1_,._1_1_0_0_0_0_1_,0_,1_0_:_:_._
1_0_0_0_0_1_,1_,._1_1_0_0_0_0_1_,._1_0_0_0_0_0_:_._1_0_0_0_0_1_._1_1_0_0_0_0_1_,
1_0_,._1_0_0_0_0_0_,._1_0_0_0_0_0_ " ""]
```

      a) Here you can see tat a new config starting with : (last part of the 2nd line) is appended to the tape.
   4. Transition analysis for the encoded tape:

	tape	state
initial	!a_	1
updated	!a_	2

a) The encoded TM initially has “!a” on the tape. The starting state is 1. Then the input string is “a”, which matches the transition to update the configuration. Then the updated configuration is with tape “!a”, and state is 2.

vi. Step4: Clean symbol t (after inspection we found that the states regarding this clean up is unnecessary as symbol t is cleared while copy-pasting blanks, but we didn’t bother try to remove this as we write this report)

```
*Test> config_clean
[1: "!." "_1_0_0_0_0_0_$_1_0_;_1_,._1_1_0_0_0_0_1_,._1_1_0_0_0_0_1_,0_,1_0_:_:_.
_1_0_0_0_0_1_,1_,._1_1_0_0_0_0_1_,._1_0_0_0_0_0_:_._1_0_0_0_0_1_._1_1_0_0_0_0_1_
,1_0_,._1_0_0_0_0_0_,._1_0_0_0_0_0_"]
```

vii. Step5: no transition matched, check for finals

```
*Test> config_final
[165: "!._1_0_0_0_0_0_$_1_0_;z1_,._1_1_0_0_0_0_1_,._1_1_0_0_0_0_1_,0_,1_0_:_:_._
1_0_0_0_0_1_,1_,._1_1_0_0_0_0_1_,._1_0_0_0_0_0_:_._1_0_0_0_0_1_._1_1_0_0_0_0_1_,
1y0y,.y1y0y0y0y0y0y,._1_0_0_0_0_0_" ""]
```

1. The idea in here is that if no transition matches the current configuration, we need to check the state to see if a final state is arrived.
2. Mark the config and check for each final states:
   a) Here we can simply compare the state in the current config and check if this state matches any states in the final states.
   b) The final result as shown below is that the utm reaches the final state 185.

viii. Step6: accepted

```
*Test> config_end
[185: "!." "_1_0_0_0_0_0_$_1_0_;z1_,._1_1_0_0_0_0_1_,._1_1_0_0_0_0_1_,0_,1_0_:_:
_._1_0_0_0_0_1_,1_,._1_1_0_0_0_0_1_,._1_0_0_0_0_0_:_._1_0_0_0_0_1_._1_1_0_0_0_0_
1_,1_0_,.y1y0y0y0y0y0y,._1_0_0_0_0_0_"]
```

ix. Summary:
1. Step0: 0
2. Step1: 155
3. Step2-1: 127
4. Step2-2: 2602
5. Step3-1: 384
6. Step3-2: 11666
7. Step4: 188
8. Step5: 2067
9. Step6: 1311
10. Total: 18500

Possible Future Improvement:

Our UTM is too slow. We may modify our algorithm so that after we append new configs, we delete the redundant ones so that we only keep the newest. But this requires additional runtime on marking and copy-pasting. We are not sure if that would improve or not.

Another possible way is that we may also program our UTM to get rid of redundant blanks for the configs.

Reference:

https://plato.stanford.edu/entries/turing-machine/