Computer Science Assignment 11

Yiping Deng

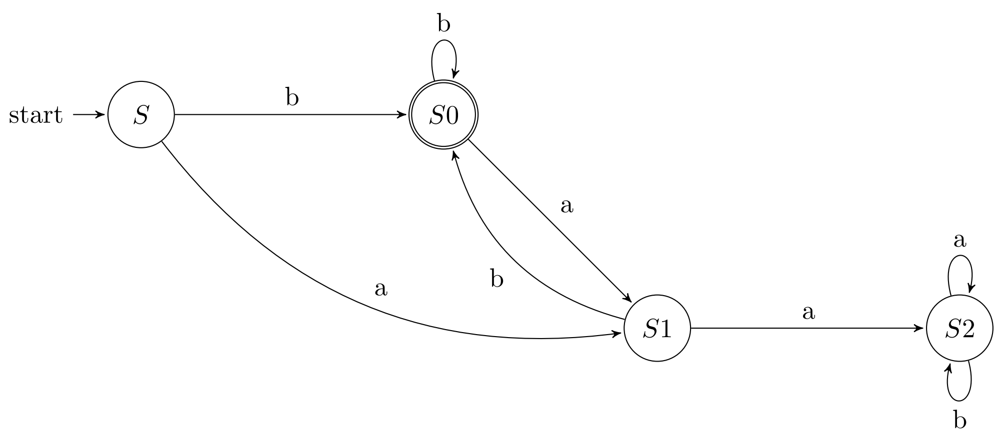
December 4, 2017

Problem 11.1

1)

$$\begin{aligned}
\delta = \{&((S,a),S1),((S,b),S0),\\
&((S0,b),S0),((S0,a),S1),\\
&((S1,a),S2),((S1,b),S0)\\
&((S2,a),S2),((S2,b),S2)\}\\
S &= \{S0,S1,S2,S3\}\\
s0 &= S\\
F &= \{S0\}
\end{aligned}$$

2)



As shown in the figure, S is the inital state, S0 is the accepting state.

3)

The following Haskell code will represent the finite state machine above.

```
-- the states of FSM
data State = S | S0 | S1 | S2

-- executing the delta function
-- takes a State, retrives the top Char from String, execute
-- the finite state machine
-- until it ends
accepts :: State -> String -> Bool
```

```
-- initial state
accepts S ('a':xs) = accepts S1 xs
accepts S ('b':xs) = accepts S0 xs
-- move out of initial state to S0

-- S0 state movement
accepts S0 ('a':xs) = accepts S1 xs
accepts S0 ('b':xs) = accepts S0 xs
accepts S0 [] = True
-- this is a accepting state

-- S1 state movement
-- failing case
accepts S1 ('b':xs) = accepts S0 xs
accepts S2 ('a':xs) = accepts S2 xs

-- S2 and others
accepts _ _ = False
-- covering up all the failure case, including S2
-- this also covers non-accepting state case
-- also covers the case the string is not in
-- the char set1

-- use this function to test
decide :: String -> Bool
decide = accepts S
```

4)

The regular grammar that generates the patterns is described below.

$$
\begin{aligned}
G &= (N, \Sigma, P, S) \\
N &= \{S, T, U\} \\
\Sigma &= \{a, b\} \\
P &= \{S \mapsto aT, S \mapsto bU, \\
&\quad U \mapsto \epsilon, U \mapsto bU, \\
&\quad U \mapsto aT, \\
&\quad T \mapsto bU\}
\end{aligned}
$$

Problem 11.2

a)

The following is the mathematical definition of increment Turing machine.

$$
\begin{aligned}
s0 &= S0 \\
\Gamma &= \{0, 1, \$\} \\
b &= \_ \\
F &= \{\} \\
\delta &= \{(S0, \$, S1, \$, R), \\
&\quad (S1, \$, S2, \$, L), (S1, 0, S1, 0, R), (S1, 1, S1, 1, R), \\
&\quad (S2, 0, S3, 1, L), (S2, 1, S2, 0, L), \\
&\quad (S3, 0, S3, 0, L), (S3, 1, S3, 1, L)\}
\end{aligned}
$$

S0 is the initial state, just skip a $ symbol.
S1 is the skip state, skipping until hit the last $ symbol.
S2 is the carried state, do the carried addition.
S3 is the non-carried state.

The following is the Haskell code validates the Turing machine.

```haskell
import Prelude hiding (head)

-- give enough states to Turing machine
data State = S0 | S1 | S2 | S3 deriving (Show)

-- the tape is a string with index of current head
data Tape = Tape String Int deriving (Show)

-- Tape S
-- return the raw String under the Tape
tapes :: Tape -> String
tapes (Tape s _) = s
-- read from turing machine
head :: Tape -> Char -> Bool
head (Tape xs i) c = xs !! i == c

-- move the tape left
left :: Tape -> Tape
left (Tape xs i)
    | i == 0 = Tape ("_" ++ xs) 0
    | otherwise = Tape xs (i - 1)

-- move the tape right
right :: Tape -> Tape
right (Tape xs i)
    | i + 1 >= length xs = Tape (xs ++ "_") (i + 1)
    | otherwise = Tape xs (i + 1)

-- write to the tape
write :: Tape -> Char -> Tape
write (Tape xs i) c = Tape (replaceAt i c xs) i
    where replaceAt 0 nc (y:ys) = nc:ys
          replaceAt n nc (y:ys) = y:replaceAt (n - 1) nc ys

accepts :: State -> Tape -> Tape
-- initial state S0, move to skipping state
-- directly move
accepts S0
    | head tape '$' = accepts S1 (right tape)

-- skipping state, move to the least signficant digit
accepts S1 tape
    | head tape '$' = accepts S2 (left tape)
    | head tape '0' = accepts S1 (right tape)
    | head tape '1' = accepts S1 (right tape)

-- carried add state, write down 1 if 0 on tape
-- write down 0 if 1 on tape
-- carry of 1 on tape
-- moving to the left
accepts S2 tape
    | head tape '0' = accepts S3 (left (write tape '1'))
    | head tape '1' = accepts S2 (left (write tape '0'))
```

```
-- none carried state, simply do nothing, just move until the end
accepts S3 tape
    | head tape '0' = accepts S3 (left tape)
    | head tape '1' = accepts S3 (left tape)

-- if no instruction, halt
-- in this case, it will be S2 or S3 hit £ symbol, which is not defined
-- it will hit the halt and return the tape
accepts _ tape = tape

-- increment function
-- takes a string like £010102£ in binary
-- increase the binary int by 1
-- return £01010£ like binary string
increment :: String -> String
increment x = tapes $accepts S0 (Tape x 0)
```

b

Mathematical definition of decrement Turing machine

$$
\begin{aligned}
s0 &= S0 \\
\Gamma &= \{0, 1, \$\} \\
b &= \_ \\
F &= \{\} \\
\delta &= \{(S0, \$, S1, \$, R), \\
&\quad (S1, \$, S2, \$, L), (S1, 0, S1, 1, R), (S1, 1, S1, 0, R), \\
&\quad (S2, 0, S3, 1, L), (S2, 1, S2, 0, L), \\
&\quad (S3, 0, S3, 0, L), (S3, 1, S3, 1, L) \\
&\quad (S4, 1, S4, 0, R), (S4, 0, S4, 1, R)\}
\end{aligned}
$$

S0 is the inital state, just skip a $ symbol.
S1 is the right flip state, flip the bits until hit $.
S2 is the carried state, do the carried addition.
S3 is the non-carried state. S4 is the flip state, flip the bits back until hit $.

This is the validating Haskell code.

```
import Prelude hiding (head)

-- give enough states to Turing machine
data State = S0 | S1 | S2 | S3 | S4 deriving (Show)

-- the tape is a string with index of current head
data Tape = Tape String Int deriving (Show)

-- Tape S
-- return the raw String under the Tape
tapes :: Tape -> String
tapes (Tape s _) = s
-- read from turing machine
head :: Tape -> Char -> Bool
head (Tape xs i) c = xs !! i == c

-- move the tape left
```

```haskell
left :: Tape -> Tape
left (Tape xs i)
    | i == 0 = Tape ("_" ++ xs) 0
    | otherwise = Tape xs (i - 1)

-- move the tape right
right :: Tape -> Tape
right (Tape xs i)
    | i + 1 >= length xs = Tape (xs ++ "_") (i + 1)
    | otherwise = Tape xs (i + 1)

-- write to the tape
write :: Tape -> Char -> Tape
write (Tape xs i) c = Tape (replaceAt i c xs) i
    where replaceAt 0 nc (y:ys) = nc:ys
          replaceAt n nc (y:ys) = y:replaceAt (n - 1) nc ys

accepts :: State -> Tape -> Tape
-- initial state S0, move to flip state
-- directly move
accepts S0 tape
    | head tape '$' = accepts S1 (right tape)


-- flip state, move to the least signficant digit
-- flip 0 to 1, 1 to 0
accepts S1 tape
    | head tape '$' = accepts S2 (left tape)
    | head tape '0' = accepts S1 (right (write tape '1'))
    | head tape '1' = accepts S1 (right (write tape '0'))

-- carried add state, write down 1 if 0 on tape
-- write down 0 if 1 on tape
-- carry of 1 on tape
-- moving to the left
-- hit the end, move to flip state again
accepts S2 tape
    | head tape '0' = accepts S3 (left (write tape '1'))
    | head tape '1' = accepts S2 (left (write tape '0'))
    | head tape '$' = accepts S4 (right tape)

-- none carried state, simply do nothing, just move until the end
-- hit the end, then move to flip state
accepts S3 tape
    | head tape '0' = accepts S3 (left tape)
    | head tape '1' = accepts S3 (left tape)
    | head tape '$' = accepts S4 (right tape)

-- flip the state, 1 to 0 , 0 map to 1
accepts S4 tape
    | head tape '0' = accepts S4 (right (write tape '1'))
    | head tape '1' = accepts S4 (right (write tape '0'))

-- if no transition defined, halt
-- in this case, it will be S4  hit £ symbol, which is not defined
-- it will hit the halt and return the tape, which is decreased value
accepts _ tape = tape
```

```
-- decrement function
-- takes a string like £010102£ in binary
-- decrease the binary int by 1
-- return £01010£ like binary string
decrement :: String -> String
decrement x = tapes $accepts S0 (Tape x 0)
```

c)