Universal Turing Machine
We both did the course evaluations :)

YAFAN HUANG & SHIHUI SONG, The University of Iowa, USA

Universal Turing Machine (UTM), of which principle is considered to be the original idea of a stored-program computer, simulates an arbitrary Turing machine (TM) on an arbitrary input. In this report, we aim to design and implement a UTM as the final project based on provided Haskell code in this course. The report is organized as follows. In Section 1, we will formulate the UTM problem, mapping its function into a TM. In Section 2, a high-level algorithm design of our UTM is provided. In Section 3, we will introduce the implementation-level algorithm design, which contains our modified encoding strategy and two TM-level algorithms. In Section 4, we will present the detailed design along with debug phase of our UTM. At last, we will provide the appendix of this project code.

CCS Concepts: • **Theory of Computation**;

Additional Key Words and Phrases: Turing Machine as a Problem Solver

ACM Reference Format:
Yafan Huang & Shihui Song. 2021. Universal Turing Machine We both did the course evaluations :). In *Course Report'21: CS:4330:0001 Fall21 Theory of Computation, Aug 23–Dec 10, 2021, Iowa City, IA.*

1 PROBLEM FORMULATION

Though almost every algorithm we know are Turing-computable, we do not always think questions in a TM way. It may be hard to directly come up with a TM idea that can realize the function of UTM. So first of all, we formulate the UTM to a general problem, which is determining if a string can be accepted by a given language. Such problem can be formulated as following mathematical formula:

$$\text{UTM} = \{< M, s > | M \text{ is a Turing machine and } M \text{ accepts string } s.\} \tag{1}$$

Once a encoded TM and input string can be accepted by this language, it also indicates the input string matched the given TM.

2 HIGH-LEVEL ALGORITHM DESIGN

We design the UTM step by step, starting with a human-readable high-level algorithm design. The high-level algorithm design can be seen in Algorithm 1. An encoded TM & input string $< M, s >$ contains initial state, final state, and transition functions of TM and all symbols of input string. We make two initialization: (1) we mark the initial state as current state and it will be overwrote to simulate the TM header moves; (2) we mark the first symbol in input string s as current symbol to simulate the tape is read from left to right every time. If current state does not match final state, that means the TM has not reached its final and is still in a intermediate state (or just halt). Note that in the TM provided in the Haskell code, only final state contains accept result – reject will be shown as halt in an arbitrary intermediate state. Otherwise, it will go into a loop, which is moving header by target transition functions. To find which transition function guide its movement, we traverse all transition functions. If current state matches input state of current transition function and read symbol in the transition function matches current symbol, we will do the overwrite and update. Then break from the transition function traversal loop. Or else, if the traversal reaches the end of the transition function, which indicates all transition function can not match current state and symbol, TM will then reject the input string. At last, if current state matches final state, TM accepts the input string. This is our UTM high-level design.

```
Input: Encoded TM & input string < M, s >;
Output: Accept or Reject;
Function start UTM
    Mark initial state as current state;
    Mark first input symbol as current symbol;
    while current state != final state do
        forall transition function do
            if input state = current state then
                if read symbol = current symbol then
                    overwrite current symbol;
                    overwrite current state;
                    update current symbol;
                    Break;
                end
            end
            else if transition function ends then
                Reject;
            end
        end
    end
    Accept;
end
```

Algorithm 1: High-level Algorithm Design

3 IMPLEMENTATION-LEVEL ALGORITHM DESIGN

After designing the high-level algorithm, we have to map all the function into a TM-level, and here the implementation-level algorithm design comes. In this section, we addressed two issues before proposing the detailed methodology. Firstly, we will understand the encoding strategy provided by the Haskell code, and slightly modify it based on our requirements. Then, we will address the difficulties in Implementing UTM along with dividing it into several small blocks.

3.1 Understanding and Modifying the Encoding Format

Below highlighted texts shows the original encoding example of a simple TM. Note that no line shifts are contained in the encoded string, we add this just for a clearer explanation. By observing the

given encoding strategy and reviewing our algorithm, we conclude several requirements on modifying the encoding function:

- Divide TM and input string: In original encoding strategy, each part in TM and input string are divided by '#'. We aim to use '$' to divide TM and input string to avoid some unnecessary header movement.
- Avoid right shift in overwriting symbol: Each symbol in input string is encoded to length of 6 or 7. Thus we will add one '_' add the end of every symbol, to avoid right shift of all tape.
- Label current transition function: Because there is a transition function traversal phase in our algorithm, we use '' to label the current transition function.
- Label current symbol: Because the we want to simulate how header is moving left and right by the TM on the input string, we use '^' to label current symbol in the input string. Also to avoid unnecessary right shift, we add '_' in front of each input symbol for labeling.
- Label current state: There is a start state part in every encoded TM. Here we directly regard it as the current state, and overwrite it when the header moves. Once the start state matches one state from final state sets, we will give accept results. As a result, we use '%' and '+' to label current and final state respectively. Because the current state overwrites for every steps, we add some '_' to avoid some unnecessary right shifts.

Here is an example of how original encoding strategy works:

```
100001#                     /*Leftend*/
000001#                     /*Blank Symbol*/
1#                          /*Start State*/
011,#                       /*Final State*/
1.000001.0.011.000001.,     /*Transition Func1*/
1.010101.0.1.010101.,#      /*Transition Func2*/
1000011,0100011,#           /*Input String*/
```

Here is an example of how our encoding strategy works:

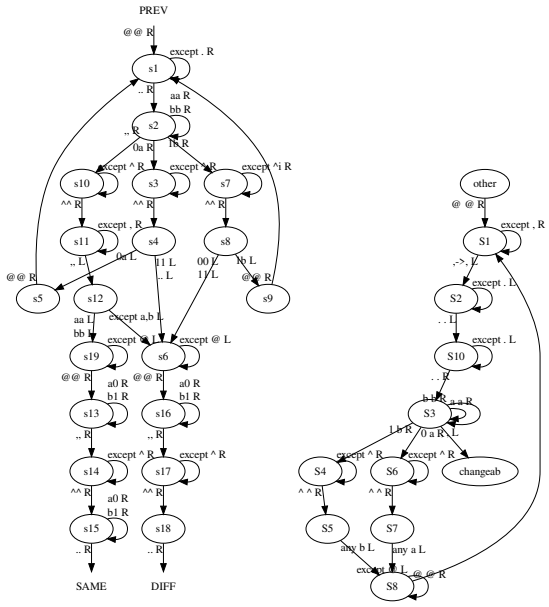
```
100001#                     /*Leftend*/
000001%                     /*Blank Symbol*/
1_____+                     /*Start State*/
011,#                       /*Final State*/
@1.000001.0.011.000001.,    /*Transition Func1*/
1.010101.0.1.010101.,$      /*Transition Func2*/
^_1000011_,_0100011_,#      /*Input String*/
```

Our modified encoding strategy Haskell code is shown as below:

```
addString :: UEncode a => String -> a -> String -> String
addString u x rest = bitenc x ++ u ++ rest

percent :: UEncode a => a -> String -> String
percent = bitencSep '%'

underline :: UEncode a => a -> String -> String
underline = addString "_____+"

transitStart :: UEncode a => [a] -> String -> String
transitStart xs rest = foldrGlue comma xs ("#@" ++ rest)

inputStart :: UEncode a => [a] -> String -> String
inputStart xs rest = foldrGlue comma xs ("$^" ++ rest)

inputSpace :: UEncode a => [a] -> String -> String
inputSpace xs rest = foldrGlue spaceComma xs ('#' : rest)
```

3.2 Understanding the Difficulties in Implementing UTM

After designing the encoding strategy, we try to address the core parts in this UTM, which are state/symbol/transition function comparison and overwrite state/symbol. The two parts' designs visualized by graphViz can be shown in Figure 1.



(a) Comparing two state, symbol or transition function (b) Overwriting state or symbol

Fig. 1. Basic logic of two core part in our designed UTM, shown as a simple TM.

We integrates these two core algorithms into our final UTM, which is described in details in Section Methodology.

4 METHODOLOGY

4.1 Detailed Design of UTM

Our detailed design of UTM can be seen as Figure 2. Specifically, our model can be divided into several parts, which are listed as below:

- **Checking** if current state matches final state. This function is realized mainly by state 1-21.
- **Traversing** all transition functions, and determine if current symbol and state matches start state and read symbol in current transition function. This function is realized mainly by state 21-46.

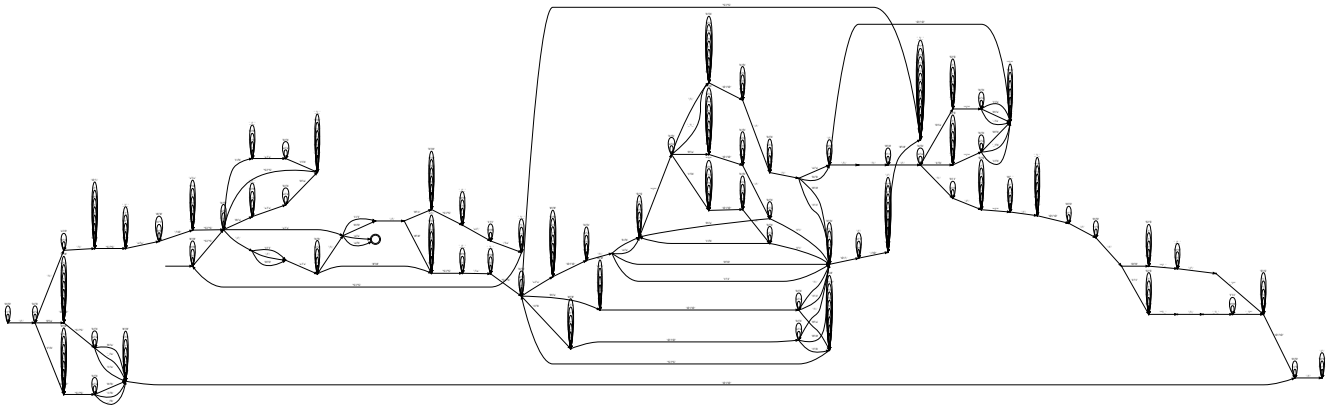


Fig. 2. The detailed design of our UTM. The raw and a clearer version of visualized UTM can be found in the project folder /raw-design/.

- **Overwriting** current state, symbol, and transition function based on the information provided by the current transition function in traversal phase. The function of this part is roughly realized by state 47-84.

Specifically, there are some implementation details. In part 1 (**Checking** phase), the initial start state starts with a symbol "%", while the initial final state starts with a symbol "+" respectively. We move the tape header from the start symbol on the start state. If header is pointing '0', we change to 'a'. Else if header is pointing '1', we change to 'b'. Then we move the header to the first symbol of final state (i.e. the symbol after '+'). If this symbol matches the first symbol of start state, we replace the symbol same as the previous strategy. And we repeat this procedure until the header on start state is pointing a '_' or '+' (i.e. reaching the end of the start state). If the matching false during this process, we move the '+' to the next final state, as the final state is a set containing a series of accepted final states. Finally, we will check the last character of the final state and see if there are still character except 'a' or 'b' at the end of the final state. If such circumstance happens on all final states in final state set, this indicate no current state matches final state, and we continue to the **Traversing** phase. We will also change 'a' and 'b' to '1' and '0' after this process. In part 2 (**Traversing** phase), current symbol starts with a '$\hat{}$' and current transition function starts with an '@'. We use the similar strategy to match start state and reading symbol of transition function with current state and current symbol. But we temporarily keep the 'a' and 'b' during this process. If we still not find the appropriate transition function that matching the current state and symbol, we will reject the encoded TM and string. We move '@' to the first transition function after we jump out the **Traversing** phase. In part 3 (**Overwriting**), we follow the same strategy while overwriting the current state and current symbol (i.e. using 'a' and 'b' to temporarily replace '0' and '1'). Note that the '$\hat{}$' of current symbol is moving according to the direction provided by transition function, and the string of current symbol is also overwrote by the write symbol in the same transition function (i.e. the current one).

4.2 Debugging Phase

For part 1 **Checking** phase, we use 1-21 state for debugging its functionality. As Figure 3 shows, when we use the debugging tool with step number = 15, we can find the first symbol of start state (i.e. current state in our UTM) changes from '1' to 'b'.

```
*TMExamples> ntcsJunan (initialConfig utm $ inputU utm "abc") 15
[4: "!100001#000001%b_" "____+1101,#@1.100011.0.1.100011.,1.000011.0.1.0000
11.,1.110001.0.1.110001.,1.101001.0.01.101001.,01.100011.0.001.0100011.,01.
000011.0.11.1000011.,01.101001.0.111.101001.,01.110101.0.1001.110101.,01.11
11101.0.01.1111101.,01.1000011.0.01.1000011.,01.0100011.0.01.0100011.,01.00
```

Fig. 3. Debugging results of step=15 in **Checking** phase.

In step number = 22, shown as Figure 4, because the first symbol of final state is also '1', we also change it to 'b'.

```
*TMExamples> ntcsJunan (initialConfig utm $ inputU utm "abc") 22
[7: "!100001#000001%b_____+" "b101,#@1.100011.0.1.100011.,1.000011.0.1.0000
11.,1.110001.0.1.110001.,1.101001.0.01.101001.,01.100011.0.001.0100011.,01.
000011.0.11.1000011.,01.101001.0.111.101001.,01.110101.0.1001.110101.,01.11
11101.0.01.1111101.,01.1000011.0.01.1000011.,01.0100011.0.01.0100011.,01.00
```

Fig. 4. Debugging results of step=22 in **Checking** phase.

In step number = 57, shown as Figure 5, because there are still symbols after we scan the final state, we change them to '1' because of the mismatch.

```
*TMExamples> ntcsJunan (initialConfig utm $ inputU utm "abc") 57
[16: "!100001#000001%" "1_____,1101,#@1.100011.0.1.100011.,1.000011.0.1.000
011.,1.110001.0.1.110001.,1.101001.0.01.101001.,01.100011.0.001.0100011.,01
.000011.0.11.1000011.,01.101001.0.111.101001.,01.110101.0.1001.110101.,01.1
111101.0.01.1111101.,01.1000011.0.01.1000011.,01.0100011.0.01.0100011.,01.0
```

Fig. 5. Debugging results of step=57 in **Checking** phase.

For part 2 **Traversing** phase, we add another 22-46 states and them debugging, shown as nodes in the visualized UTM graph.

In step number = 7782, header and tail picture are shown as Picture 6 and 7. The matching strategy of current state in start state

```
*TMExamples> ntcsJunan (initialConfig utm $ inputU utm "abc") 7782
[31: "!100001#000001%b____+1101,#@b.100011.0.1.100011.,1.000011.0.1.000011
.,1.110001.0.1.110001.,1.101001.0.01.101001.,01.100011.0.001.0100011.,01.00
0011.0.11.1000011.,01.101001.0.111.101001.,01.110101.0.1001.110101.,01.1111
```

Fig. 6. Debugging header results of step=7782 in **Traversing** phase.

```
1.,011101.0111101.1.011101.0111101.,011101.1111101.1.011101.1111101.,011101
.101001.0.01101.101001.,$^1000011_,_0100011_,_1100011_,_#"]
```

Fig. 7. Debugging tail results of step=7782 in **Traversing** phase.

of current transition function is similar as what we did in **Checking** phase.

However, in step number = 60000, we can also see, in Figure 8 and 9, read symbol of transition function is matching with the current symbol in input string. We can predict that because the next symbol here is '0', we can predict that, this matching is unsuccessful.

```
*TMExamples> ntcsJunan (initialConfig utm $ inputU utm "abc") 60000
[36: "!100001#000001%b____+1101,#@b.baa011.0.1.100011.,1.000011.0.1.000011
00011.0.01.1000011.,01.0100011.0.01.0100011.,01.001101.0.0111.001101.,11.00
.0.001.000011.,001.1111101.0.001.1111101.,001.110101.0.011.110101.,101.1000
```

Fig. 8. Debugging header results of step=60000 in **Traversing** phase.

```
1.,011101.1111101.1.011101.1111101.,011101.101001.0.01101.101001.,$^baaa011
_,_0100011_,_1100011_,_#"]
```

Fig. 9. Debugging tail results of step=60000 in **Traversing** phase.

We feel regret that because of the time limitation and the extremely larger step numbers in **Overwriting** phase, we did not finish all the debugging phase in Phase 3. But based on our observations in the debugging results of first 2 phases, our UTM works basically correct. Although there may be some mistakes in Sections 3, our UTM design is somehow promising.

5 APPENDIX

5.1 UTM Haskell Code

We provided our UTM Haskell code at the end of this document.

```
utm =
  TM [1 .. 84] "10.,#+@$^_%" "10.,#+@$^_%ab" id ' ' '!' trans 1 [11]
  where
    trans = -- Checking Phase --
            loopRight 1 "10#" ++
            goRight 1 '%' '%' 2 ++
            goRight 2 '1' 'b' 4 ++
            goRight 2 '0' 'a' 3 ++
            goRight 2 '%' '%' 7 ++
            goRight 2 '+' '+' 9 ++
            loopRight 2 "_ab" ++
            goRight 2 ',' ',' 14 ++
            loopRight 3 ",10_" ++
            goRight 3 '+' '+' 5 ++
            loopRight 4 ",10_" ++
            goRight 4 '+' '+' 6 ++
            loopRight 5 "ab" ++
            goLeft 5 '0' 'a' 7 ++
            goLeft 5 '1' '1' 8 ++
            goLeft 5 ',' ',' 8 ++
            loopRight 6 "ab" ++
            goLeft 6 '1' 'b' 7 ++
            goLeft 6 '0' '0' 8 ++
            goLeft 6 ',' ',' 8 ++
            loopLeft 7 ",+01ab_" ++
            goRight 7 '%' '%' 2 ++
            loopLeft 8 "_+01,L" ++
            goLeft 8 'a' '0' 8 ++
            goLeft 8 'b' '1' 8 ++
            goRight 8 '%' '%' 17 ++
            loopRight 9 "01ab" ++
            goLeft 9 ',' ',' 10 ++
            goLeft 9 '#' '#' 16 ++
            goRight 10 'a' 'a' 11 ++
            goRight 10 'b' 'b' 11 ++
            goRight 10 '0' '0' 12 ++
            goRight 10 '1' '1' 12 ++
            goRight 12 ',' ',' 15 ++
            goRight 13 '+' '+' 9 ++
            goRight 14 ',' ',' 13 ++
            goRight 14 '0' '0' 13 ++
            goRight 14 '1' '1' 13 ++
            goLeft 15 '#' '#' 16 ++
            goLeft 15 '0' '1' 8 ++
            loopLeft 16 "01,_" ++
            goLeft 16 'a' '0' 16 ++
            goLeft 16 'b' '1' 16 ++
            goLeft 16 '+' ',' 16 ++
            goRight 16 '%' '%' 20 ++
            loopRight 17 ",10_" ++
            goRight 17 '+' ',' 18 ++
            loopRight 18 "01" ++
            goLeft 18 ',' '+' 19 ++
            loopLeft 19 "01,_" ++
            goRight 19 '%' '%' 2 ++
            loopRight 20 "10_" ++
            goLeft 20 ',' '+' 21 ++
             -- Traversing Phase --
            loopLeft 21 "10_" ++
            goRight 21 '%' '%' 22 ++
            loopRight 22 "_ab" ++
            goRight 22 '1' 'b' 24 ++
            goRight 22 '+' '+' 28 ++
            goRight 22 '0' 'a' 23 ++
            loopRight 23 ",01_+#" ++
            goRight 23 '@' '@' 25 ++
            loopRight 24 ",_01+#" ++
            goRight 24 '@' '@' 26 ++
            loopRight 25 "ab" ++
            goLeft 25 '0' 'a' 27 ++
            goLeft 25 '1' '1' 43 ++
```

```
goLeft 25 '.' '.' 43 ++
loopRight 26 "ab" ++
goLeft 26 '1' 'b' 27 ++
goLeft 26 '0' '0' 43 ++
goLeft 26 '.' '.' 43 ++
loopLeft 27 ",01+_ab@#" ++
goRight 27 '%' '%' 22 ++
loopRight 28 ",01+_ab#" ++
goRight 28 '@' '@' 29 ++
loopRight 29 "01ab" ++
goLeft 29 '.' '.' 30 ++
goLeft 30 '0' '0' 43 ++
goLeft 30 '1' '1' 43 ++
goRight 30 'a' 'a' 31 ++
goRight 30 'b' 'b' 31 ++
loopRight 31 "ab01.,$" ++
goRight 31 '^' '^' 35 ++
loopLeft 32 "ab01.,$^" ++
goRight 32 '@' '@' 33 ++
loopRight 33 "01ab" ++
goRight 33 '.' '.' 34 ++
loopRight 34 "ab" ++
goRight 34 '1' 'b' 31 ++
goLeft 34 '0' '0' 43 ++
goLeft 34 '.' '.' 43 ++
loopRight 35 "ab" ++
goLeft 35 '1' 'b' 32 ++
goLeft 35 '0' 'a' 36 ++
goLeft 35 ',' ',' 39 ++
goLeft 35 '_' '_' 39 ++
loopLeft 36 "01,.$^ab" ++
goRight 36 '@' '@' 37 ++
loopRight 37 "01ab" ++
goRight 37 '.' '.' 38 ++
loopRight 38 "ab" ++
goLeft 38 '1' '1' 43 ++
goLeft 38 '.' '.' 43 ++
goRight 38 '0' 'a' 31 ++
loopLeft 39 "01,.$^ab" ++
goRight 39 '@' '@' 40 ++
loopRight 40 "01ab" ++
goRight 40 '.' '.' 41 ++
loopRight 41 "01ab" ++
goLeft 41 '.' '.' 42 ++
goLeft 42 '0' '0' 43 ++
goLeft 42 '1' '1' 43 ++
goRight 42 'a' 'a' 47 ++
goRight 42 'b' 'b' 47 ++
loopLeft 43 "10.ab" ++
goRight 43 '@' ',' 44 ++
goRight 44 '0' '0' 44 ++
goRight 44 '1' '1' 44 ++
goRight 44 'a' '0' 44 ++
goRight 44 'b' '1' 44 ++
loopRight 44 "." ++
goRight 44 ',' '@' 45 ++
goRight 45 'a' '0' 45 ++
goRight 45 'b' '1' 45 ++
loopRight 45 "10._$^," ++
goLeft 45 '#' '#' 46 ++
goLeft 46 'b' '1' 46 ++
goLeft 46 'a' '0' 46 ++
loopLeft 46 "10.,#+@$^_" ++
goRight 46 '%' '%' 22 ++

-- Overwriting Phase--
loopRight 47 "10." ++
goLeft 47 ',' ',' 48 ++
goLeft 48 '.' '.' 49 ++
loopLeft 49 "10" ++
goRight 49 '.' '.' 50 ++
loopRight 50 "ab" ++
goRight 50 '1' 'b' 51 ++
goRight 50 '0' 'a' 53 ++
goLeft 50 '.' '.' 56 ++
loopRight 51 "_10.,$" ++
goRight 51 '^' '^' 52 ++
loopRight 52 "ab" ++
goRight 52 '1' 'b' 55 ++
goRight 52 '0' 'b' 55 ++
goRight 52 '_' 'b' 55 ++
loopRight 53 "_10.,$" ++
goRight 53 '^' '^' 54 ++
loopRight 54 "ab" ++
goRight 54 '1' 'a' 55 ++
goRight 54 '0' 'a' 55 ++
goRight 54 '-' 'a' 55 ++
loopLeft 55 "_10,.$^" ++
goRight 55 '@' '@' 47 ++
goLeft 56 'a' '0' 56 ++
goLeft 56 'b' '1' 56 ++
goRight 56 '.' '.' 57 ++
loopRight 57 "_10,.$" ++
goRight 57 '^' '^' 58 ++
goRight 58 'a' '0' 58 ++
goRight 58 'b' '1' 58 ++
goRight 58 '1' '_' 58 ++
goRight 58 '0' '_' 58 ++
goRight 58 ',' ',' 59 ++
loopLeft 59 "10.,$^_" ++
goRight 59 '@' '@' 60 ++
loopRight 60 "10" ++
goRight 60 '.' '.' 61 ++
loopRight 61 "10" ++
goRight 61 '.' '.' 62 ++
goRight 62 '0' '0' 63 ++
goRight 62 '1' '1' 64 ++
loopRight 63 "01,.$" ++
goRight 63 '^' '_' 68 ++
loopRight 64 "01,.$" ++
goLeft 64 '^' '_' 65 ++
goLeft 65 ',' ',' 66 ++
```

```
goLeft 66 '_' '_' 67 ++
loopLeft 67 "01" ++
goLeft 67 '_' '^' 70 ++
loopRight 68 "10_" ++
goRight 68 ',' ',' 69 ++
goLeft 69 '_' '^' 70 ++
loopLeft 70 "01,.$" ++
goRight 70 '@' '@' 71 ++
loopRight 71 "10" ++
goRight 71 '.' '.' 72 ++
loopRight 72 "10" ++
goRight 72 '.' '.' 72 ++
loopRight 73 "10" ++
goRight 73 '.' '.' 74 ++
loopRight 74 "ab" ++
goLeft 74 '0' 'a' 75 ++
goLeft 74 '1' 'b' 78 ++
goLeft 74 '.' '.' 80 ++
loopLeft 75 "@10.,_+#" ++
goRight 75 '%' '%' 76 ++
loopRight 76 "ab" ++
goRight 76 '1' 'a' 77 ++
goRight 76 '0' 'a' 77 ++
goRight 76 '_' 'a' 77 ++
loopRight 77 "01,_+#" ++
goRight 77 '@' '@' 71 ++
loopLeft 78 "@10.,_+#" ++
goRight 78 '%' '%' 79 ++
loopRight 79 "ab" ++
goRight 79 '0' 'b' 77 ++
goRight 79 '1' 'b' 77 ++
goRight 79 '_' 'b' 77 ++
goLeft 80 'b' '1' 80 ++
goLeft 80 'a' '0' 80 ++
goLeft 80 '.' '.' 81 ++
loopRight 81 "01,_+#." ++
goLeft 81 '@' '.' 81 ++
goRight 81 '%' '%' 82 ++
goRight 82 'a' '0' 82 ++
goRight 82 'b' '1' 82 ++
goRight 82 '0' '_' 82 ++
goRight 82 '1' '_' 82 ++
goRight 82 '_' '_' 82 ++
goRight 82 '+' '+' 83 ++
loopRight 83 "01#" ++
goLeft 83 ',' '@' 84 ++
loopLeft 84 "_10,#+" ++
goRight 84 '%' '%' 2
```

5.2 Debugging Function

Inspired and followed by Junnan's presentation in the last course, we understand the functionality and write a similar one. This is the reason why we name the debug function as ntcsJunnan. The Haskell code of this function can be seen as below.

```
ntJunan :: Config Integer Char -> [Config Integer Char]
ntJunan = newConfigs utm
ntcJunan :: Config Integer Char -> Config Integer Char
ntcJunan config = ntJunan config !! 0
ntcsJunan ::  Config Integer Char -> Int -> Config Integer Char
ntcsJunan config n = last $ take (n+1) (iterate ntcJunan config)
```