Turing Machine Implementation

by Sina Rostami

1. Turing Machine

Re-introduction to the Turing Machine.

2. Implementation concepts

Basis for Implementing the TM.

3. Implementation

Implementing the TM in C++ programming lang.

– Turing Machine

Formally, a Turing machine is a 7-tuple,

$$(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}}),$$

$$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$$

Members of δ AKA Rules.

The heart of a Turing machine is its transition function δ, as it tells us how the machine gets from one configuration to another.

A Turing machine configuration is described by its current state, the current tape contents, and the current head location.

Tip

We use _ (underscore) to show TM's blank symbol in this lecture.

– TM Configuration example

For a state q and two strings u and v,
over the tape alphabet Γ,
we write uqv :
for the configuration where the current state is q,
the current tape contents is uv,
and the current head location is the first symbol of v.

–Turing Machine Configuration

For example

1011 q7 0111

represents the configuration when the tape is 10110111,

the current state is q7,

and the head is currently on the first 0 after q7.

–Turing Machine Configuration

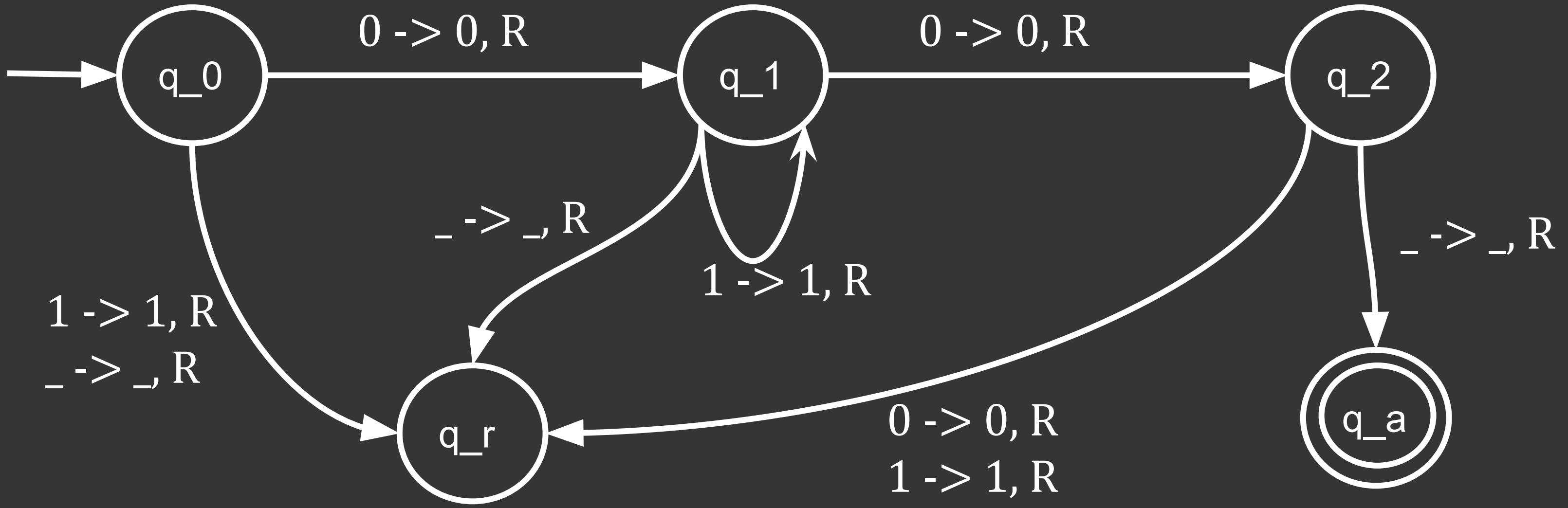
An accepting configuration is a configuration in which the state is the accept state.

A rejecting configuration is a configuration in which the state is the reject state.

Accepting and rejecting configurations are halting configurations.

–Turing Machine example

For L = {01*0}



Implementation Basis

We need following concepts to implement TM.

- **TM 's 7-Tuple**

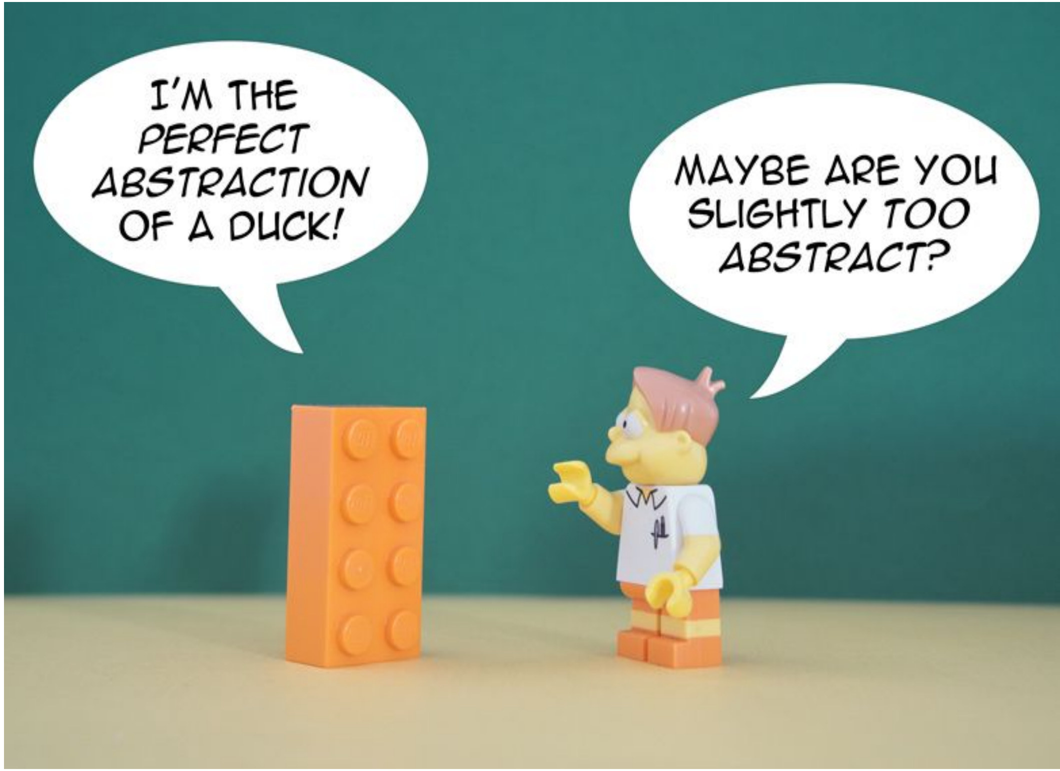
  For implementing we need the 7-tuple which we discussed earlier.

- **I/O**

  We need some input and output feature to give the initializing fields like 7-tuple and input string, and also see the result of the TM.

But How to Implement our 7-Tuple in code !!!???

Abstraction!



Let’s Start

We start from abstracting some basis,

and then go for some coding and test our idea.

Concepts

- **Direction**

  Left, Right

- **Rule**

  a -> b, R

- **State**

  q_0, q_1, .., q_a, q_r

- **Result**

  ongoing, accepted, rejected

- **TM!**

– Direction

Only can have 2 values, LEFT and RIGHT

So we Use a simple enum to abstract the direction.

```
enum class Direction
{
 LEFT,
 RIGHT,
};
```

– Rule

a -> b , R

In a single rule we must have 2 characters:

- The character that TM reads from tape
- The character that TM writes in tape

And also a Direction, which we already defined.

```
struct Rule
{
 char read_symbol, write_symbol;
 Direction direction;
};
```

– State

Every state has a name (like q_0, q_1, ..., q_a, q_r)
and also some transitions to other states with specified Rule!
So we need rules that've been mapped to states.

```
struct State
{
 string name;
 map<Rule, State> transitions;
};
```

– Result

Like the Direction, Result also can have only 3 values:

1. ON_GOING
2. ACCEPTED
3. REJECTED

```
enum class Result
{
  ACCEPTED,
  REJECTED,
  ON_GOING,
};
```

TM

Turing should have followings:

Some states and also q_0, q_a, q_r

Some chars as input alphabet

Some chars as tape alphabet

```
struct TuringMachine
{
 vector<State> states;
 State q_0, q_a, q_r;
 vector<char> input_alphabet;
 vector<char> tape_alphabet;
};
```

Tip

We use std::vector as container

Milestone

TM intro.

Abstract concepts

Run and test codes

Turing’s work

Our work

Implementation basis

Implement required and auxiliary functions.

Turing’s work is much greater than our’s!

What is left ?

Write auxiliary functions for connecting structures together and checking given string to the TM

parse the TM inputs.

In other word handle to make an instance of our TM.

Test our TM with some examples.

And try to break it

– Check input string func.

```
Result check_input_string(string input_string)
{
  size_t current_index = 0;
  State current_state = q_0;
  Result result = Result::ON_GOING;
  print_current_config(input_string, current_index, current_state);
  while (result == Result::ON_GOING)
  {
    handle_current_char(input_string, current_index, current_state);
    print_current_config(input_string, current_index, current_state);
    if (current_state == q_a)
      result = Result::ACCEPTED;
    else if (current_state == q_r)
      result = Result::REJECTED;
  }
  return result;
}
```

- Handle current char func.

```
void handle_current_char(string &input_string, size_t &current_index,
                         State &current_state)
{
  char &head_symbol = input_string[current_index];
  for (auto &pair : current_state.transitions)
  {
    if (pair.first.read_symbol == head_symbol)
    {
      head_symbol = pair.first.write_symbol;                 // a -> b
      current_index = (pair.first.direction == Direction::LEFT) // R, L
                       ? current_index - 1
                       : current_index + 1;
      current_state = pair.second;
      return;
    }
  }
}
```

We're Almost Done!

Now let's parse and test the example we just checked out in TM intro.

_ Parsing and testing an example func.

Parse Rules

```
Rule r1('0', '0', Direction::RIGHT);
Rule r2('1', '1', Direction::RIGHT);
Rule r3('_', '_', Direction::RIGHT);
```

_ Parsing and testing an example func.

Parse States

```
State q_0("q_0");
State q_1("q_1");
State q_2("q_2");
State q_a("q_a");
State q_r("q_r");
```

_ Parsing and testing an example func.

Parse Transitions(connect States to each other)

```
q_0.transitions.emplace(r1, q_1);
q_0.transitions.emplace(r2, q_r);
q_0.transitions.emplace(r3, q_r);
q_1.transitions.emplace(r2, q_1);
q_1.transitions.emplace(r1, q_2);
q_1.transitions.emplace(r3, q_r);
q_2.transitions.emplace(r3, q_a);
q_2.transitions.emplace(r1, q_r);
q_2.transitions.emplace(r2, q_r);
```

_ Parsing and testing an example func.

Create a TM, Call the chek func. on a string and see the output

```
TuringMachine tm({q_0, q_1, q_2, q_a, q_r},
                 {'0', '1'},
                 {'0', '1', '_'},
                 q_0, q_a, q_r);

Result result = tm.check_input_string("0111110___");
switch (result)
{
case Result::ACCEPTED:
    cout << "ACCEPTED" << endl;
    break;
case Result::REJECTED:
    cout << "REJECTED" << endl;
    break;
}
```

The result !

```
"0111110___"
```

```
q_00111110___
0q_1111110___
01q_111110___
011q_11110___
0111q_1110___
01111q_110___
011111q_10___
0111110q_2___
0111110_q_a__
ACCEPTED
```

The result on wrong input!

`"01111101___"`

```
q_001111101___
0q_11111101___
01q_1111101___
011q_111101___
0111q_11101___
01111q_1101___
011111q_101___
0111110q_21___
01111101q_r___
REJECTED
```

Thanks!

I hope this gave you a newer and deeper perspective to the TM.

You can find this presentation

and the source files

in my github profile with link bellow:

https://github.com/sina-rostami

Course : Theory of Languages and Automata
Instructor : Dr. Khasteh

Fall 2020