計算機数学期末レポート

A1778594 池田 力

期末レポート課題例の問6

ローマ字変換を行うような**有限状態変換器**を考える。

この計算モデルを定式化する。

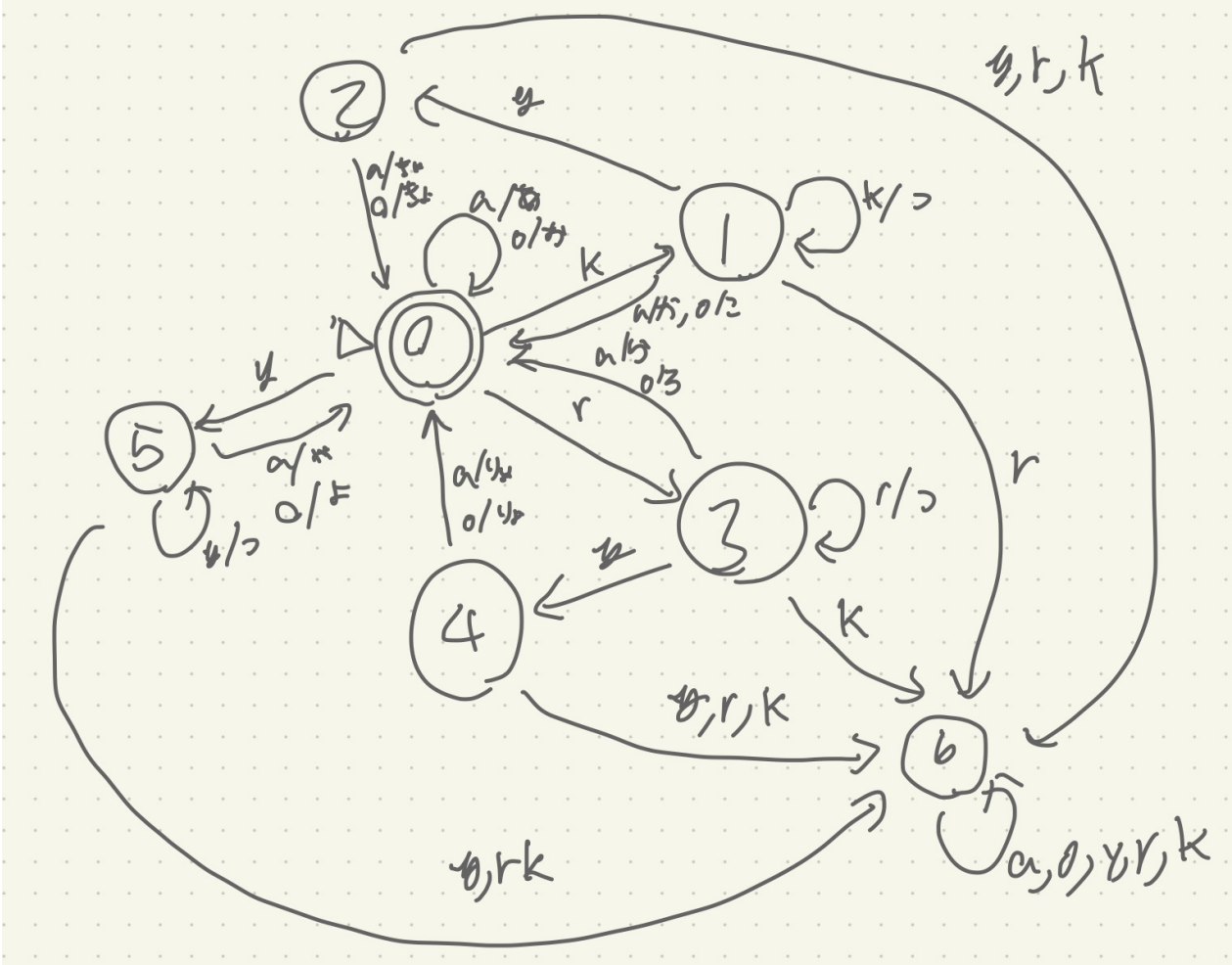
出力付き有限オートマトンMを以下のように定める。

$$M = (Q, \Sigma, \Gamma, \delta, s, F)$$

ただし、それぞれの文字は以下のような意味を持つ

- Q : 有限集合 ・・・ オートマトンの状態の集合
- Σ : 有限集合 ・・・ 入力文字の集合
- Γ : 有限集合 ・・・ 出力文字の集合
- $\delta : Q \times \Sigma \to Q \times \Gamma$ ・・・ 状態遷移関数。現在の状態と入力文字から次に遷移する状態と出力する文字への写像である。
- $s \in Q$ ・・・ 初期状態
- $F \subset Q$ ・・・ 受理状態(正しく変換できた状態)の集合

具体的な出力付き有限オートマトンを構成する。



ローマ字かな変換を行うプログラムをRustで作成した。

コードについて

※コードの全体は https://github.com/Tsutomu-Ikeda/rust-dfa/blob/master/src/main.rs へアップロードしたので適宜参照されたい。 まず、上図の状態遷移図の状態を `enum` で列挙した。

```
enum States {
  S0,
  S1,
  S2,
  S3,
  S4,
  S5,
  S6,
}
```

次に、 `impl`トレイト を用いて初期状態、受理状態、状態遷移関数(`next`)を定義した。

```
impl States {
  // 初期状態
  fn new() -> States {
    States::S0
  }

  // 受理状態か判定する
  fn is_accept_state(self) -> bool {
    match self {
      States::S0 => true,
      _ => false,
    }
  }

  // 状態遷移関数(出力付き)
  fn next(self, i: char) -> States {
    match self {
      States::S0 => match i {
        'k' => States::S1,
        'r' => States::S3,
        'y' => States::S5,
        'a' => {
          print!("あ");
          States::S0
        }
        'o' => {
          print!("お");
          States::S0
        }
        _ => States::S6,
      },
      States::S1 => match i {
        'k' => {
          print!("っ");
          States::S1
        }
        'y' => States::S2,
        'a' => {
          print!("か");
          States::S0
        }
        'o' => {
          print!("こ");
          States::S0
        }
        _ => States::S6,
      },
      States::S2 => match i {
        'a' => {
```

```
          print!("きゃ");
          States::S0
        }
        'o' => {
          print!("きょ");
          States::S0
        }
        _ => States::S6,
      },
      States::S3 => match i {
        'r' => {
          print!("っ");
          States::S3
        }
        'y' => States::S4,
        'a' => {
          print!("ら");
          States::S0
        }
        'o' => {
          print!("ろ");
          States::S0
        }
        _ => States::S6,
      },
      States::S4 => match i {
        'a' => {
          print!("りゃ");
          States::S0
        }
        'o' => {
          print!("りょ");
          States::S0
        }
        _ => States::S6,
      },
      States::S5 => match i {
        'y' => {
          print!("っ");
          States::S5
        }
        'a' => {
          print!("や");
          States::S0
        }
        'o' => {
          print!("よ");
          States::S0
        }
        _ => States::S6,
      },
      States::S6 => match i {
        _ => States::S6,
      },
    }
  }
}
```

以上の有限オートマトンの定義から状態の遷移を行えば、期待した通りの動作となる。

```
fn get_input_value() -> String {
  let mut buffer = String::new();
  let stdin = io::stdin();
  let mut handle = stdin.lock();

  handle.read_to_string(&mut buffer).unwrap();
  return String::from(buffer.trim());
}
```

```
fn translate(input: String) {
  print!("{}: ", input);
  let mut state = States::new();

  for i in input.chars() {
    state = state.next(i)
  }

  print!("\n");

  println!("最終遷移状態: {}", state);

  if state.is_accept_state() {
    println!("正常に変換されました。")
  } else {
    println!("正常に変換されませんでした。")
  }
}

fn main() -> io::Result<()> {
  let input = get_input_value();

  for line in input.split("\n") {
    translate(String::from(line));
  }
  Ok(())
}
```

動作例

`input.txt` として以下のような入力を用意した。

```
kokyoyakoryokoo
yokoyokokokkyo
koyookak
aaakkkka
aaakkrr
yokohama
```

この入力を用いて実行したところ以下のような結果となった。

```
kokyoyakoryokoo: こきょやこりょこお
最終遷移状態: S0
正常に変換されました。
yokoyokokokkyo: よこよこここっきょ
最終遷移状態: S0
正常に変換されました。
koyookak: こよおか
最終遷移状態: S1
正常に変換されませんでした。不正な入力です。
aaakkkka: あああっっっか
最終遷移状態: S0
正常に変換されました。
aaakkrr: あああっ
最終遷移状態: S6
正常に変換されませんでした。不正な入力です。
yokohama: よこ
最終遷移状態: S6
正常に変換されませんでした。不正な入力です。
```

正常な入力では、最終遷移状態が `S0` になりしっかりとひらがなに変換されていることがわかる。また、最終遷移状態が `S0` 以外の場合、不正な入力として受理されていないことがわかる。特に `kr` という入力や、`kryao` 以外の文字の入力をすると、 `S6` へ遷移し変換が途中から行われていないことがわかる。

期末レポート課題例の問8

逆ポーランド記法で記述された数式を計算して返すようなプログラムを作成した。 なお、実数に対応させ、小数点付きの数値も受け付けるように実装した。

コード

```
use std::io::{self, Read};

fn get_input_value() -> String {
  let mut buffer = String::new();
  let stdin = io::stdin();
  let mut handle = stdin.lock();

  handle.read_to_string(&mut buffer).unwrap();
  return String::from(buffer.trim());
}

fn calc(formula: String) {
  // プッシュダウンスタック
  let mut stack: Vec<String> = vec![];
  // 数字が入力中かどうか判定するためのフラグ
  const NUMBER_INPUT_FLAG: char = '!';

  for c in formula.chars() {
    if NUMBER_INPUT_FLAG
      .to_string()
      .eq(stack.last().unwrap_or(&String::from(" ")))
    {
      stack.pop();
      match c {
        '0'..='9' | '.' => {
          let last_number = stack.pop().unwrap();
          stack.push(last_number + &c.to_string());
          stack.push(NUMBER_INPUT_FLAG.to_string());
        }
        '+' | '-' | '*' | '/' | '^' | ' ' => {}
        _ => {
          break;
        }
      }
    } else {
      match c {
        '0'..='9' => {
          stack.push(c.to_string());
          stack.push(NUMBER_INPUT_FLAG.to_string());
        }
        '+' => {
          let right: f64 = stack.pop().unwrap().parse().unwrap();
          let left: f64 = stack.pop().unwrap().parse().unwrap();
          stack.push(format!("{}", left + right));
        }
        '-' => {
          let right: f64 = stack.pop().unwrap().parse().unwrap();
          let left: f64 = stack.pop().unwrap().parse().unwrap();
          stack.push(format!("{}", left - right));
        }
        '*' => {
          let right: f64 = stack.pop().unwrap().parse().unwrap();
          let left: f64 = stack.pop().unwrap().parse().unwrap();
          stack.push(format!("{}", left * right));
        }
        '/' => {
          let right: f64 = stack.pop().unwrap().parse().unwrap();
          let left: f64 = stack.pop().unwrap().parse().unwrap();
          stack.push(format!("{}", left / right));
        }
```

```
          '^' => {
            let right: f64 = stack.pop().unwrap().parse().unwrap();
            let left: f64 = stack.pop().unwrap().parse().unwrap();
            stack.push(format!("{}", left.powf(right)));
          }
          ' ' => {}
          _ => {
            break;
          }
        }
      }
    }
    if stack.len() == 1 {
      let result: f64 = stack.pop().unwrap().parse().unwrap();
      println!("{} => {}", formula, result);
    } else {
      println!("{} => 文法が誤っています。計算できませんでした。", formula);
    }
}

fn main() -> io::Result<()> {
    let input = get_input_value();

    for line in input.split("\n") {
      calc(String::from(line));
    }
    Ok(())
}
```

工夫した点/苦労した点

一桁の自然数のみを受け付けるプログラムから、二桁以上の数字も計算できるようにする方法を工夫した。

数字が連続して入力されている状態か保持する方法を考えた結果、入力された数字をスタックに入れる際は次のスタックに `!` を追加し、計算結果を入れる場合は追加しないという方針で作った。 自然数全体を受け付けられるようにすれば、実数への拡張は小数点を入力可能文字にするだけであったため、簡単に行えた。

popで演算子の左の項と右の項を取り出す順番を間違えた。

右から先に取り出さなければならないのに、左から取り出していた。 `+` や `*` の可換な演算子ではたまたまうまく行ったが、 `-` や `/` の可換ではない演算子でうまく行かないことに気が付き修正した。

実行サンプル

```
89 10 + 22 * => 2178
355 113 / => 3.1415929203539825
1.6 1.6 * 0.4 0.2 / - 100 * 1 + => 57.00000000000005
2 1 2 / ^ => 1.4142135623730951
2 3 1 * => 文法が誤っています。計算できませんでした。
```