１、目的

プログラミング言語である LISP の方言 Scheme の方言である Racket もプログラミング言語であるからチューリング完全である。本レポートでは Racket 上でチューリング完全である言語の BrainFuck（以下 BF）を実装できることを示すことで改めて Racket がチューリング完全であることを確認する。

２、BF の仕様

BF はデータを格納する配列、配列のいずれかを指すポインタ、BF のプログラム、プログラムのどこを読んでいるのかを指すポインタ、入力と出力ストリームを持つ。命令は以下の８つである。

１、「>」ポインタをインクリメントする
２、「＜」ポインタをデクリメントする
３.「＋」インタが指す値をインクリメントする
４、「－」ポインタが指す値をデクリメントする
５、「.」ポインタが指す値を出力に書き出だす
６、「,」入力から１バイト読み込んでポインタが指す値に代入する
７、「[]ポインタが指す値が 0 なら対応する]の直後にジャンプする
８、「]」ポインタが指す値が 0 でないなら対応する[の直にジャンプする

今回の実装ではプログラム実行中に入力を受け取る機能は実装しない。そのため、上記のうち「,」は実装しないことに注意。

３、BF の実装

最初に、BF における配列に当たるものを Racket のリストで表現する。必要な機能としてはポインタの指す要素へのアクセスと書き換えができればよい。この時、ポインタが既存のリストの先を指した場合は都度 0 で埋めてリスト拡張するようにした。

```
(define (access_list list p) ;リストとポインタを入力してその指す値を出力する
  (cond
    [(null? list) (access_list (cons 0 '()) p)]
    [else
     (cond
       [(= 0 p)  (car list)]
       [else (access_list (cdr list) (ptrmm p))])]))
(define (use_list list p func);リストとポインタと1引数の関数を入力として、ポインタの指す値をその値を関数に入力した結果に書き換えたリストを出力する
   (cond
    [(null? list) (use_list (cons 0 '()) p func)]
```

```
    [else
     (cond
       [(= 0 p) (cons (func (car list)) (cdr list))]
       [else (cons (car list) (use_list (cdr list) (ptrmm p) func))])]))
```

上記を用いて BF の関数の 1 ～ 5 を実装したのが以下

```
(define ptrpp (lambda (x) (+ x 1)));インクリメント
(define ptrmm (lambda (x) (- x 1)));デクリメント
(define listpp (lambda (list ptr) (use_list list ptr ptrpp)));リストとポインタを入力して指定した値をインクリメント
(define listmm (lambda (list ptr) (use_list list ptr ptrmm)));リストとポインタを入力して指定した値をデクリメント
(define output (lambda (list ptr) (integer->char (access_list list ptr))));リストとポインタを入力して指した値を出力
```

本プログラムでは、文字列形式で受け取った BF のプログラムを一文字ずつのリストに入れたものを解釈していく。ループが存在しなければ、先頭の要素を読んだ後その後ろの要素のリストをまた BF プログラムの解釈にかければよい。リストの先頭の要素とそれ以外の要素へアクセスするのは LISP の基本であるから簡単に実装できる。そして、命令リストが空になったところで終了である。これを実装したのが以下

```
(define mainloop (lambda (tape ptr bat)
                   (cond
                     [(equal? (car bat) #¥>) (mainloop tape (ptrpp ptr) (cdr bat))]
                     [(equal? (car bat) #¥<) (mainloop tape (ptrmm ptr) (cdr bat))]
                     [(equal? (car bat) #¥+) (mainloop (listpp tape ptr) ptr (cdr bat))]
                     [(equal? (car bat) #¥-) (mainloop (listmm tape ptr) ptr (cdr bat))]
                     [(equal? (car bat) #¥.) (begin
                                               (display (string (output tape ptr)))
                                               (mainloop tape ptr (cdr bat)))]
                     [(equal? (car bat) #¥[) (forloop tape ptr bat 1)]
                     [else (car bat)]
                   )))
```

ループの始点を検知したら、そこからはループを考慮した処理へ移行する。つまり、BF の命令リストのどこを読んでいるのかを指すポインタを導入した処理である。こちらは C や Java のような実装である。

```
(define forloop (lambda (tape ptr bat batptr)
                        (cond
                          [(equal? (access_list bat batptr) #¥>) (forloop tape (ptrpp ptr) bat (ptrpp batptr))]
                          [(equal? (access_list bat batptr) #¥<) (forloop tape (ptrmm ptr) bat (ptrpp batptr))]
                          [(equal? (access_list bat batptr) #¥+) (forloop (listpp tape ptr)  ptr bat (ptrpp batptr))]
                          [(equal? (access_list bat batptr) #¥-) (forloop (listmm tape ptr)  ptr bat (ptrpp batptr))]
                          [(equal? (access_list bat batptr) #¥.) (begin
                                                                   (display (string (output tape ptr)))
                                                                   (forloop tape  ptr bat (ptrpp batptr)))]
                          [(equal? (access_list bat batptr) #¥]) (cond
                                                                   [(= 0 (access_list tape ptr)) (forloop tape ptr bat (ptrpp batptr))]
                                                                   [else (forloop tape ptr bat (forstart bat (ptrmm batptr) 0))])
                                                                   ]
                          [(equal? (access_list bat batptr) #¥[) (cond
                                                                   [(= 0 (access_list tape ptr)) (forloop tape ptr bat (forend bat (ptrpp batptr) 0))]
                                                                   [else(forloop tape ptr bat (ptrpp batptr) )]
                                                                   )]
                          )))

(define forend (lambda (bat batptr n);[]を抜ける処理
                       (cond
                         [(equal? (access_list bat batptr) #¥[) (forend  bat (ptrpp batptr) (+ n 1))]
                         [(equal? (access_list bat batptr) #¥])
                          (cond
                            [(= 0 n) (ptrpp batptr)]
```

```
                [else (forend bat (ptrpp batptr) (- n 1))])]
            [else (forend bat (ptrpp batptr) n)])))
(define forstart (lambda (bat batptr n);[]の先頭に戻る処理
            (cond
              [(equal? (access_list bat batptr) #¥]) (forstart bat (ptrmm batptr ) (+
n 1))]
              [(equal? (access_list bat batptr) #¥[)
               (cond
                 [(= 0 n) (ptrpp batptr)]
                 [else (forstart bat (ptrmm batptr) (- n 1))])]
              [else (forstart bat (ptrmm batptr) n)])))
```

BF のプログラムを文字列で受け取って解釈へ投げる本プログラムの初期化処理が以下

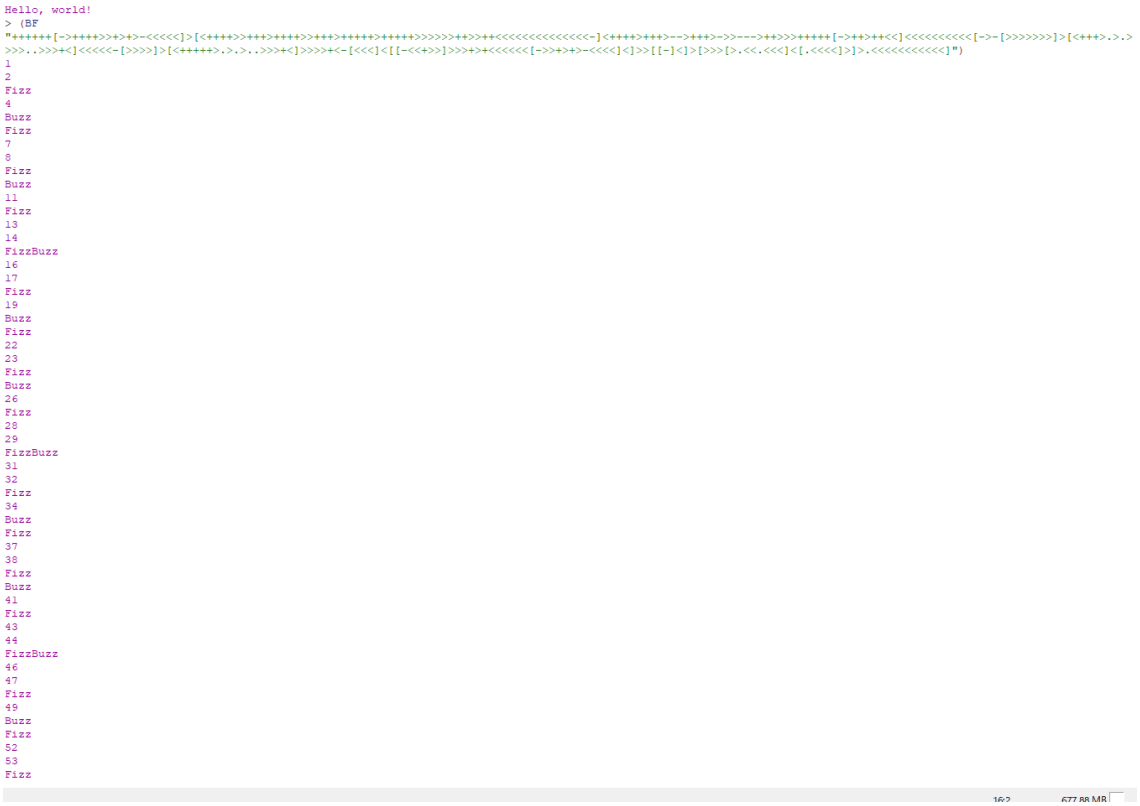
```
(define (BF str)
  (begin
    (let ([ptr 0])
      (let ([tape '()])
        (mainloop tape ptr (string->list str))))))
```

４、実行結果

```
ようこそ DrRacket, バージョン 7.2 [3m].
言語: racket, with debugging.
> (BF "+++++++++[>++++++++>++++++++++>+++++<<<-]>.>++.+++++++..+++.>-.------------.<++++++++.--------.+++.------.--------.>+.")
Hello, world!
>
```

Hello,world!

問題なく実行されている。



FizzBuzz

多重ループを含む命令でも問題なく動作していることがわかる。

これらのサンプルプログラムはサイト
http://hakugetu.so.land.to/program/brainfuck/1-4.php
から引用させていただいた。

５、結論

Racket 上で BF を実装することができた。改めて、Racket がチューリング完全であることを示すことができた。

６、ソースコード全文

```
#lang racket
(define (BF str)
  (begin
    (let ([ptr 0])
      (let ([tape '()])
        (mainloop tape ptr (string->list str))))))
```

```
(define mainloop (lambda (tape ptr bat)
                        (cond
                          [(equal? (car bat) #¥>) (mainloop tape (ptrpp ptr) (cdr bat))]
                          [(equal? (car bat) #¥<) (mainloop tape (ptrmm ptr) (cdr bat))]
                          [(equal? (car bat) #¥+) (mainloop (listpp tape ptr) ptr (cdr bat))]
                          [(equal? (car bat) #¥-) (mainloop (listmm tape ptr) ptr (cdr bat))]
                          [(equal? (car bat) #¥.) (begin
                                                    (display (string (output tape ptr)))
                                                    (mainloop tape ptr (cdr bat)))]
                          [(equal? (car bat) #¥[) (forloop tape ptr bat 1)]
                          [else (car bat)]
                        )))

(define forloop (lambda (tape ptr bat batptr)
                       (cond
                         [(equal? (access_list bat batptr) #¥>) (forloop tape (ptrpp ptr) bat (ptrpp
batptr))]
                         [(equal? (access_list bat batptr) #¥<) (forloop tape (ptrmm ptr) bat
(ptrpp batptr))]
                         [(equal? (access_list bat batptr) #¥+) (forloop (listpp tape ptr)  ptr bat
(ptrpp batptr))]
                         [(equal? (access_list bat batptr) #¥-) (forloop (listmm tape ptr)  ptr bat
(ptrpp batptr))]
                         [(equal? (access_list bat batptr) #¥.) (begin
                                                                  (display (string (output
tape ptr)))
                                                                  (forloop tape  ptr bat
(ptrpp batptr)))]
                         [(equal? (access_list bat batptr) #¥]) (cond
                                                                 [(= 0 (access_list tape
ptr)) (forloop tape ptr bat (ptrpp batptr))]
                                                                 [else (forloop tape ptr
bat (forstart bat (ptrmm batptr) 0))])
                                                                 ]
                         [(equal? (access_list bat batptr) #¥[) (cond
```

```
                                                    [(= 0 (access_list tape
ptr)) (forloop tape ptr bat (forend bat (ptrpp batptr) 0))]
                                                    [else(forloop  tape  ptr
bat (ptrpp batptr) )]
                                                    )]
                        )))
```

```
(define ptrpp (lambda (x) (+ x 1)));インクリメント
(define ptrmm (lambda (x) (- x 1)));デクリメント
(define listpp (lambda (list ptr) (use_list list ptr ptrpp)));リストとポインタを入力して指定した値をインクリメント
(define listmm (lambda (list ptr) (use_list list ptr ptrmm)));リストとポインタを入力して指定した値をデクリメント
(define output (lambda (list ptr) (integer->char (access_list list ptr))));リストとポインタを入力して指した値を出力
(define forend (lambda (bat batptr n);[]を抜ける処理
                    (cond
                      [(equal? (access_list bat batptr) #¥[) (forend  bat (ptrpp batptr) (+ n 1))]
                      [(equal? (access_list bat batptr) #¥])
                       (cond
                         [(= 0 n) (ptrpp batptr)]
                         [else (forend bat (ptrpp batptr) (- n 1))])]
                      [else (forend bat (ptrpp batptr) n)])))
(define forstart (lambda (bat batptr n);[]の先頭に戻る処理
                    (cond
                      [(equal? (access_list bat batptr) #¥]) (forstart bat (ptrmm batptr ) (+ n 1))]
                      [(equal? (access_list bat batptr) #¥[)
                       (cond
                         [(= 0 n) (ptrpp batptr)]
                         [else (forstart bat (ptrmm batptr) (- n 1))])]
                      [else (forstart bat (ptrmm batptr) n)])))
```

```
(define (access_list list p) ;リストとポインタを入力してその指す値を出力する
  (cond
    [(null? list) (access_list (cons 0 '()) p)]
    [else
     (cond
       [(= 0 p)  (car list)]
       [else (access_list (cdr list) (ptrmm p))])]))
(define (use_list list p func);リストとポインタと1引数の関数を入力として、ポインタの指す値をその値を関数に入力した結果に書き換えたリストを出力する
   (cond
    [(null? list) (use_list (cons 0 '()) p func)]
    [else
     (cond
       [(= 0 p) (cons (func (car list)) (cdr list))]
       [else (cons (car list) (use_list (cdr list) (ptrmm p) func))])]))
```