１、序論

本レポートの目的は、関数型言語と手続き型言語が同程度の計算力を持っていることを証明することである。そのために、Haskell と Python 上で Brainfuck（以下 BF）を動作させ、その実装過程と結果について考察する。また、関数型言語の方が手続き型言語よりもメモリ使用量について調査する。

２、計算力とチューリング完全

万能チューリングマシンと等しい能力を持っている計算機はチューリング完全であるという。一般に、コンピュータのプログラム言語はチューリング完全であるといわれているが、ここで改めてそれを示す。

チューリング完全であることを示すには、チューリング完全であることが証明されている計算機を実装すればよい。今回は BF を実装する。

BF は一つの配列と配列上の位置を表すポインタから構成されていて、入力はこのポインタと配列を操作する命令を並べる。命令は以下の８つ

＞、ポインタの値を一つ増やす

＜、ポインタの値を一つ減らす

＋、ポインタの指す配列の値を一つ増やす。

―、ポインタの指す配列の値を一つ減らす。

．、ポインタが指す値を出力に書き出す。

，、入力から１バイト読み込んでポインタが指す先に代入する。

[、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

] 、ポインタが指す値が 0 でないなら、対応する[の直後にジャンプする。

入力で読み込ませたい内容も命令に予め書き込んでおくことができるので、入力を読み込む機能の有無で計算力は変わらない。今回は最低限の実装のため、入力を読み込む機能は実装しないこととする。

３、Haskell 版 BF

Brainguck.hs

```
import Data.Char

data Command = Next | Back | Puls | Minus | Input | Output | Roopstart | None
|Roopend deriving(Eq)

brainfuck ::[Int] ->Int ->[Command] ->Int -> [Int]->[Int]
brainfuck t p com i output
 |length com==i =output
```

```
 |otherwise= exec t p com i output

exec ::[Int] ->Int -> [Command]  ->Int ->[Int] ->[Int]
exec t p com i out
  |(com !! i) ==Next =brainfuck  t (p + 1) com (i + 1) out
  |(com !! i) ==Back =brainfuck  t (p - 1) com (i + 1) out
  |(com !! i) ==Puls =brainfuck  (puls t p) p com (i + 1) out
  |(com !! i) ==Minus =brainfuck  (minus t p) p com (i + 1) out
  |(com !! i) ==Output =brainfuck  t p com  (i + 1) (t !! p : out)
  |(com !! i)==Roopstart =roopstart t p com i out
  |(com !! i)==Roopend =roopend t p com i out
  |(com !! i) ==None =brainfuck  t p  com (i + 1) out



puls :: [Int] -> Int ->[Int]
puls t p =(take p t) ++ ((t !! p + 1) :[])++ drop (p+1) t

minus :: [Int] -> Int ->[Int]
minus t p =(take p t) ++ ((t !! p - 1) :[])++ drop (p+1) t

roopstart :: [Int] ->Int -> [Command] ->Int->[Int] ->[Int]
roopstart t p com i out
 |(t !! p) ==0 =endroop t p com i out
 |otherwise =brainfuck t p com (i+1) out

endroop ::[Int] ->Int -> [Command] ->Int->[Int] ->[Int]
endroop t p com i out
 |(com !! i)==Roopend =brainfuck t p com (i+1) out
 |otherwise =endroop t p com (i+1) out

roopend ::[Int] ->Int -> [Command] ->Int->[Int] ->[Int]
roopend t p com i out
 |(t !! p) ==0 =brainfuck t p com (i+1) out
 |otherwise =startroop t p com (i-1) out

startroop ::[Int] ->Int -> [Command] ->Int->[Int] ->[Int]
```

```
startroop t p com i out
 |(com !! i) ==Roopstart =brainfuck t p com (i+1) out
 |otherwise =startroop  t p com (i-1) out

cTob :: Char ->Command
cTob '+' =Puls
cTob '-' =Minus
cTob '>' =Next
cTob '<' =Back
cTob '[' =Roopstart
cTob ']' =Roopend
cTob '.' =Output
cTob _ =None

initbf com =map chr (reverse $ brainfuck (repeat 0) 0 com 0 [] )

stTobf :: String->[Command]
stTobf st =map cTob st

testBF =stTobf "+++++++++[>++++++++>++++++++++>+++++<<<-]>.>++.+++++++..+++.>-.-
-----------.<++++++++.--------.+++.------.--------.>+."
bfhoge=initbf                                   $                                stTobf
"++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
++++++++++++++++++++++++.+++++++.--------.--."
bfHelloworld =initbf testBF
bf str=initbf (stTobf str)
```

実行画面

```
*Main> bf "+++++++++[>++++++++>++++++++++>+++++<<<-]>.>++.+++++++..+++.>-.------
------.<++++++++.--------.+++.------.-"
"Hello, world!"
```

本プログラムでは一つの BFMachine に当たる値を更新していくのではなく、値を更新した BF 関数を返す再帰的な処理をしている。実装に当たって参考文献[1]からコードを一部引用している。

```
brainfuck ::[Int] ->Int ->[Command] ->Int -> [Int]->[Int]
brainfuck t p com i output
```

```
 |length com==i =output
 |otherwise= exec t p com i output
```

これがメイン関数である。BF の配列に当たる[Int]、ポインタに当たる Int、命令列に当たる[Command]、命令列のポインタに当たる Int と出力に当たる[Int]を入力として、最終的に出力の[Int]を返す。
命令列のポインタが命令列の最後に来ていなければ命令を実行する exec を呼び出す。

```
exec ::[Int] ->Int -> [Command]  ->Int ->[Int] ->[Int]
exec t p com i out
  |(com !! i) ==Next =brainfuck  t (p + 1) com (i + 1) out
  |(com !! i) ==Back =brainfuck  t (p - 1) com (i + 1) out
  |(com !! i) ==Puls =brainfuck  (puls t p) p com (i + 1) out
  |(com !! i) ==Minus =brainfuck  (minus t p) p com (i + 1) out
  |(com !! i) ==Output =brainfuck  t p com  (i + 1) (t !! p : out)
  |(com !! i)==Roopstart =roopstart t p com i out
  |(com !! i)==Roopend =roopend t p com i out
  |(com !! i) ==None =brainfuck  t p  com (i + 1) out
```

exec では BF の配列に当たる[Int]、ポインタに当たる Int、命令列に当たる[Command]、命令列のポインタに当たる Int と出力に当たる[Int]を入力として、最終的に出力の[Int]を返す関数である brainfuck を呼び出す。また、本プログラムは BF の命令列が間違っていることを考慮していない。

roopstart はループ開始記号を読んだ時の命令で、ループを飛ばすかどうかを判断する。endroop はループの終了記号の先まで飛ばす。逆に、roopend はループ終了記号を読んだ時の命令で、ループするかどうかを判断する。startroop はループの開始記号に戻る

```
initbf com =map chr (reverse $ brainfuck (repeat 0) 0 com 0 [] )
```

[Command]を受け取って String を返す。命令列を BF に投げた後、帰ってきた Int のリストを文字列に変換する[2]。

```
stTobf :: String->[Command]
stTobf st =map cTob st
```

```
cTob :: Char ->Command
cTob '+' =Puls
cTob '-' =Minus
cTob '>' =Next
cTob '<' =Back
cTob '[' =Roopstart
cTob ']' =Roopend
cTob '.' =Output
cTob _ =None
```

stTobf は String を受け取って[Command]を返す。変換は単純に、一文字ずつ取り出して Command に相当するものがあればそれを、なければ None を返す。

４、Python 版 BF

```
class bfMachine:
    hairetu = [0]
    pointa=0
    command=[]
    step=0
    output=[]
    def __init__(self, str):
        self.command=list(str)

def main():
    print("Brainfuck のコマンドを入力してください")
    inputcommand=input('>> ')
    bfm=bfMachine(inputcommand)
    while(True):
        if(len(bfm.command)==bfm.step):
            break
        else:
            bfm=exec(bfm)
    str=list(map (chr,bfm.output))
    anser=""
    for c in str:
```

```
            anser+=c
        print(anser)

def exec(bfm):
    com=bfm.command[bfm.step]
    if(com==">"):
        return Next(bfm)
    elif(com=="<"):
        return Back(bfm)
    elif(com=="+"):
        return Puls(bfm)
    elif(com =="-"):
        return Minus(bfm)
    elif(com=="."):
        return Output(bfm)
    elif(com=="["):
        return Roopstart(bfm)
    elif(com=="]"):
        return Roopend(bfm)
    else:
        return bfm

def nextstep(bfm):
    bfm.step+=1
    return bfm

def Next(bfm):
    bfm.pointa=bfm.pointa+1
    if len(bfm.hairetu)<=bfm.pointa:
        bfm.hairetu.append(0)
    else:
        pass
    return nextstep(bfm)

def Back(bfm):
    bfm.pointa=bfm.pointa-1
```

```
    return nextstep(bfm)

def Puls(bfm):
    bfm.hairetu[bfm.pointa]+=1
    return nextstep(bfm)
def Minus(bfm):
    bfm.hairetu[bfm.pointa]-=1
    return nextstep(bfm)

def Output(bfm):
    bfm.output.append(bfm.hairetu[bfm.pointa])
    return nextstep(bfm)

def Roopstart(bfm):
    if(bfm.hairetu[bfm.pointa]==0):
        for num in range(bfm.step,len(bfm.command)):
            if (bfm.command[num]=="]") :
                bfm.step=num
                break
            else:
                pass
    return nextstep(bfm)

def Roopend(bfm):
    if(bfm.hairetu[bfm.pointa]!=0):
        for num in reversed(range(0,bfm.step)):
            if (bfm.command[num]=="[") :
                bfm.step=num
                break
            else:
                pass
    return nextstep(bfm)

main()
```

実効画面

```
Brainfuck のコマンドを入力してください
>>      +++++++++[>++++++++>++++++++++>+++++<<<-]>. >++. +++++++. . +++. >-. ----------
-. <++++++++. --------. +++. ------. --------. >+.
Hello, world!
```

Python での実装は非常に単純である。

1,状態を保持する BFMachine を用意する。

2,BFMachine から読み込んだ命令を実行し、命令の結果を BFMachine に書き込む。

3,2 を命令列の最後に行くまで While で繰り返す。

4,While を抜けたら BFMachine の Output を参照して画面に出力する。

５、両者の比較

５－１、実装上の比較

両者の一番の違いは、BF マシンの状態の扱いである。Haskell はその都度新しい関数を作っていくが、Python ではあくまで一つのマシンの値を参照し代入して処理を継続する。BF 上の命令の実行方法自体はどちらも似たようなものである。

５－２、実行時の比較

両者の実行時のメモリ使用量について調査した。単純のため、どちらも実行した瞬間に
BF に文字列

```
+++++++++[>++++++++>++++++++++>+++++<<<-]>.>++.+++++++..+++.
>-.------------.<++++++++.--------.+++.------.--------.>+
```

を与えて実行するように一部変更を加えている。

まず、Haskell について。これは GHC の設定を変更し、prof ファイルを出力させた[3]。

prof ファイルの前半部分（一部）

```
    total time  =        0.00 secs   (1 ticks @ 1000 us, 1 processor)
    total alloc =     138,056 bytes  (excludes profiling overheads)
```

```
                                                                                      individual     inherited
COST CENTRE        MODULE                    SRC                        no.    entries  %time %alloc   %time %alloc

MAIN               MAIN                      <built-in>                  41          0    0.0    0.4   100.0  100.0
 CAF               GHC.IO.Encoding.CodePage  <entire-module>             65          0    0.0    0.1     0.0    0.1
 CAF               GHC.IO.Encoding           <entire-module>             60          0    0.0    0.1     0.0    0.1
 CAF               GHC.IO.Handle.Text        <entire-module>             54          0    0.0    0.1     0.0    0.1
 CAF               GHC.IO.Handle.FD          <entire-module>             51          0    0.0   25.3     0.0   25.3
 CAF               Main                      <entire-module>             48          0    0.0    0.0   100.0   66.3
  main             Main                      BF2.hs:(68,1)-(69,136)      82          1    0.0    0.0   100.0   66.3
   bf              Main                      BF2.hs:66:1-26              84          1    0.0    0.0   100.0   66.2
    initbf         Main                      BF2.hs:58:1-64              85          1    0.0    1.2   100.0   52.4
     brainfuck     Main                      BF2.hs:(6,1)-(8,34)         87        376  100.0    0.0   100.0   51.2
      exec         Main                      BF2.hs:(11,1)-(19,53)       88        375    0.0    9.0     0.0   51.2
       ==          Main                      BF2.hs:3:97-98              89       1144    0.0    0.0     0.0    0.0
       puls        Main                      BF2.hs:23:1-57              92        249    0.0   35.6     0.0   35.6
       minus       Main                      BF2.hs:26:1-58              94         44    0.0    6.6     0.0    6.6
       roopend     Main                      BF2.hs:(39,1)-(41,40)       93          9    0.0    0.0     0.0    0.0
        startroop  Main                      BF2.hs:(44,1)-(46,41)       95        256    0.0    0.0     0.0    0.0
         ==        Main                      BF2.hs:3:97-98              96        256    0.0    0.0     0.0    0.0
       roopstart   Main                      BF2.hs:(29,1)-(31,40)       91          1    0.0    0.0     0.0    0.0
    stTobf         Main                      BF2.hs:61:1-22              86          1    0.0   13.8     0.0   13.8
     cTob          Main                      BF2.hs:(49,1)-(56,12)       90        119    0.0    0.0     0.0    0.0
 main              Main                      BF2.hs:(68,1)-(69,136)      83          0    0.0    7.8     0.0    7.8
```

prof ファイルの後半部分

トータルで使用したメモリ量に対して、main 以下が使用したのは 66%程であるから、BF で消費したメモリ使用量はおよそ 91kb である。

続いて Python について。こちらは memory_profiler を用いて main 関数のメモリの使用状況について 1 行ずつ表示させた[4]。

```
Line #    Mem usage    Increment   Line Contents
================================================
    12   16.469 MiB   16.469 MiB   @profile
    13                             def main():
    14                                 # print("Brainfuck のコマンドを入力してください")
    15   16.473 MiB    0.004 MiB       inputcommand="+++++++++[>++++++++>+++++++++++>+++++<<<-]>.>++.+++++++..+++.>-.------------.<++++++++.--------.+++.------.--------.>+."#input('>> ')
    16   16.473 MiB    0.000 MiB       bfm=bfMachine(inputcommand)
    17   16.473 MiB    0.000 MiB       while(True):
    18   16.477 MiB    0.000 MiB           if(len(bfm.command)==bfm.step):
    19   16.477 MiB    0.000 MiB               break
    20                                     else:
    21   16.477 MiB    0.004 MiB               bfm=exec(bfm)
    22   16.477 MiB    0.000 MiB       str=list(map (chr,bfm.output))
    23   16.477 MiB    0.000 MiB       anser=""
```

```
24   16.477 MiB    0.000 MiB       for c in str:
25   16.477 MiB    0.000 MiB           anser+=c
26   16.484 MiB    0.008 MiB       print(anser)
```

こちらはメイン関数内で増加したメモリ量は 0.016MiB であるから、BF の使用したメモリはおよそ 17kb である。プログラム上の BF の動作で使用したメモリの使用量は Haskell の方が大きいことがわかった。これは Python のサンプルでは状態を 1 つの変数で表現しているが、Haskell のサンプルでは再帰のためにすべての状態を保持しているからである。Haskell でも末尾再帰を用いることでその問題を回避できる。

６、まとめ

BF を実行できたことにより、Haskell と Python のどちらもチューリング完全であることは示された。一方で、実行時のメモリの使用量は実装によることがわかった。

７、引用

◎Haskell
[1] Lipovaca,Miran,田中/英行,村主/崇行,”すごい Haskell たのしく学ぼう！ Kindle 版”, オーム社 ,2012 年
[2] Haskell で文字と数値の変換したんだよっ
https://chakku.hatenablog.com/entry/2016/02/22/014737
[3] GHC における多彩な情報の出力方法
https://haskell.jp/blog/posts/2017/12-ghc-show-info.html
[4] Python memory_profiler で実行中のメモリ消費量を確認する
http://blog.roy29fuku.com/tips/python-check-memory/