平成 20 年 11 月 22 日
京都工芸繊維大学コンピュータ部

Lime 38

はじめに

こんにちは。コンピュータ部部長の湯浅です。今年も無事に Lime を発行することができた……そういっても過言ではないかもしれません。いえ、ずばり過言ではないでしょう。要するに今年も無事に Lime を発行することが出来たということです。

Lime はコンピュータ部の部員の活動の一部を記したものです。つまり「”コンピュータ部”という名前からして怪しげな集団が普段何をしているか」という今世紀最大の謎を解き明かすヒントがここに隠されているわけです。その謎を解き明かした時あなたの身に何が起こるのかはまだわかりません。しかし、未知の物に対して恐れを抱いている場合ではありません。今こそ恐怖を振り払い前進すべき時なのです。そう、この謎を解き明かせるのはこの Lime38 号を手に取ったあなたしかいないのですから。

人間五十年。Lisp も今年で 50 周年。それはさて置き、Lime が誕生してから今年で 27 年が経つそうです。まだ 21 年しか生きていない私には 27 年という時間は非常に長く感じますが、この Lime38 号に書かれている α-β 法の解説が実は Lime 創刊号にも書いてあったりすることを考えると案外短い時間なのかもしれません。車が空を飛び、人がチューブの中を光の速さで移動する時代を未だに迎えていません。昔の未来が訪れていないということは今は昔そのものなのです。竹取の翁です。

この文章を躊躇いなく載せてくれた編集の荒木君に感謝をしつつ、これをはじめの挨拶とさせていただきます。

平成 20 年 11 月 9 日
京都工芸繊維大学コンピュータ部部長 湯浅 信吾

目 次

1 シンタックスが無ければ作ればいいじゃない

情報工学課程 3 回生 湯浅 信吾

1.1 はじめに

1.1.1 前置記法はお嫌い？

「Lisp？ “1+1” をわざわざ “(+ 1 1)” とか書かないといけないんだろ？　さらには “x=2” は “(setq x 2)” だっけ？ 面倒でやってられないよ。」Lisp をちょこっとだけかじった人からこんな言葉を聞くことがあります。Lisp を書きなれてる人はこれに対して「だからそれが分かりやすいんじゃないか。」と反論しますが、大抵の場合は納得してもらえず、無駄な言い争いが始まってしまいます。

しかし、Lisp 方言の代表格である Common Lisp では “1+1” のような中置記法が絶対に使えないのかといえばそんなことはありません。元々書けないのであれば、書けるようにするまでです。

「シンタックスが無ければ作ればいいじゃない」

本記事ではシンタックスを新たに作れるという Common Lisp の脅威の力を紹介します。

1.1.2 目標

Common Lisp のリーダマクロ (reader macro) を用い、Common Lisp のプログラムに中置表記の式を書くことができるようにすることを、ここでの目標とします。

具体的には

```
(let (x y)
  #[ x = (y=5*(4+3)) - 2 ]
  (format t "~&x=~A~%y=~A~%" x y))
```

こんなプログラムを動かしたときに、

```
x=33
y=35
```

と出力されるようにすることとします。これなら前置記法が嫌いで中置記法しか愛せない方々からも文句は言われないでしょう。多分。

1.2 リーダマクロ

1.2.1 マクロ文字

Common Lisp はプログラムを読み込む際に、マクロ文字 (macro character) があれば特別な動作をしながら構文木 (syntax tree) を作ります。例えば「’(a b c)」という文字列が読み込まれると「(quote (a b c))」のような

構文木が出来上がります (「'(a b c)」の括弧はただの文字に過ぎず、「(quote (a b c))」の括弧はリストを表すS式の表現であることに注意してください)。これは、「'」が「(quote 次に来るもの)」という形の構文木を作るようなマクロ文字であり、「(」が「)」に来るまで式を読み込み、それをリストにするマクロ文字であるためです。

マクロ文字はプログラマが新たに設定することができ、その動作を Common Lisp のプログラムを書くことにより指定できます。例えば、文字「!」に「'」と同様の動作をさせるようにしてみましょう。

```
(defun !-reader (stream char)
  (declare (ignore char))
  `(quote ,(read stream t nil t)))
(set-macro-character #\! #'!-reader)
```

これで、今後「'(a b c)」と書く代わりに「!(a b c)」と書くことが出来るようになりました[1]。

1.2.2 リードテーブル

しかし、一度マクロ文字を設定してしまうと、今後ずっとその影響が残ります。マクロ文字の影響する範囲を限定するためにはリードテーブル (readtable) を一旦コピーしてからそこにマクロ文字を設定することになります。

マクロ文字の情報はリードテーブルというところに保存されます。現在のリードテーブルはスペシャル変数*readtable*に入っています。*readtable*はスペシャル変数なので let で新たに値を設定して、その let の中で set-macro-character を使うとその有効範囲は let の中だけになります。また、現在のリードテーブルをコピーするには関数 copy-readtable を使えばいいので、先ほどのマクロ文字「!」の有効範囲を限定するには次のようにします。

```
(defun !-reader (stream char)
  (declare (ignore char))
  `(quote ,(read stream nil nil t)))

(let ((*readtable* (copy-readtable)))
  (set-macro-character #\! #'!-reader)
  (defvar *my-list* !(a b c)))
```

こうすると、*my-list*の値は (a b c) になりますが、let を抜けた後は「!」の効果はなくなるようになります。

1.3 演算子順位構文解析

1.3.1 アルゴリズム

ここでは中置表記の式を読み取るために演算子順位構文解析 (operator precedence parsing) を行います。演算子順位構文解析はその名の通り、演算子に順位を付け、それに基づいて構文解析を行う手法です。

そのアルゴリズムの基本方針は「新たに読み込んだ演算子が前に読み込んだ演算子より優先順位が低ければ、前に読み込んだ演算子を還元 (reduce) する、そうでなければ新たに読み込んだ演算子をシフト (shift) する」というものです。詳細な説明は参考文献 [1] に譲るとして、ここでは次のようなアルゴリズムを利用しました。

```
# s はスタックトップを表す
# a<<s は a より s の方が優先順位が高いことを表す
# a>>s は s より a の方が優先順位が高いことを表す
# s=a は s と a の優先順位が等しいことを表す
```

[1] 実際には既にマクロ文字に割り当てられているリーダマクロ関数を get-macro-character 関数を使って取得できるので、「(set-macro-character #\! (get-macro-character #\'))」とだけ書けば同じことが出来ます。

表 1.1: 「3*2+1」の構文解析

字句	スタック	動作 (理由)	結果
3	[$]	reduce(演算子以外)	[3]
*	[* $]	shift(* ≫ $)	[3]
2	[* $]	reduce(演算子以外)	[2 3]
+	[$]	reduce(+ ≪ *)	[(* 3 2)]
1	[+ $]	reduce(演算子以外)	[1 (* 3 2)]
$	[]	reduce($ ≪ +)	[(+ (* 3 2) 1)]

```
# 文字列終端記号$と開き括弧の優先順位は最も低いとする
1．スタックに$を PUSH
2．字句を 1 つ読み込み a に代入
3．a が演算子と括弧でなければ
        a を還元
        2 に戻る
4．a<<s または (a=s かつ s が左結合) ならば
        s をスタックから POP して還元
        4 に戻る
5．a>>s ならば
        a をスタックに PUSH
6．a が閉じ括弧ならば
        対応する開き括弧を POP するまで、スタックから POP しながら還元
7．a が$ならば
        終了
8．2 に戻る
```

ここで言う「文字列終端記号」とはストリームの終端で「(read stream nil :eof)」を評価した際に得られる「:eof」のことです。文脈自由文法における「終端記号」とは無関係です。また、このアルゴリズムでは簡単のため文字列終端記号と括弧を演算子と同様に扱ってますが、後ほど作成するプログラムでは特別扱いしています。

還元の処理ですが、結果格納のためにもう一つスタックを用意して、演算子以外はそのまま (必要であれば何か処理を施して)PUSH する、演算子は結果格納用のスタックを好きにいじってから何かを PUSH する (場合によっては何も PUSH しない) ことにします。

このアルゴリズムを用い「3*2+1」の構文解析を行い、構文木「(+ (* 3 2) 1)」が出来上がる様子を表 1.1 に示します。

1.4 Common Lisp での実装

1.4.1 字句解析

構文解析を行う前に字句解析 (lexical analysis) を行う必要がありますが、ここでは字句解析を read 関数に任せることにします。read は現在のリードテーブルの規則に沿って 1 つだけ字句 (token) を読み込みます。

そうすれば、演算子に自身を表す文字[2]を返すようにマクロ文字を設定するだけで、字句解析を行うことが出来ます[3]。ただし、この方法では演算子は一文字という制限が掛かってしまいますし、二項演算子の「-」(引く) と単項演算子の「-」(マイナス) の区別が出来なくなります。本記事では簡単のためこの制限を受け入れることにします。

[2]演算子+なら文字#\+といった具合です
[3]これは set-macro-character の第 3 引数を指定しなければその文字が区切り文字に設定されるためです。

1.4.2 全体の構成

さて、最初に立てた目標を達成すべくソースを書き始めます。まず、ストリームから字句を読み取り構文解析を行う関数 (仮に my-parser とします) を定義し、そして、set-dispatch-macro-character 関数を用い「#[」が読み込まれたら、「]」を読み込むまで ny-parser に読み込みを任せるようにします。つまり以下のようなソースとなります[4]。

```
(defun my-parser (stream &rest rest)
  (declare (ignore rest))
  ;; 区切り文字を設定
  ;; 実際に構文解析し構文木を返す
  )
(set-dispatch-macro-character #\# #\[ #'my-parser)
```

あとは、my-parser をガリガリと書いていけば一応完成するのですが、せっかく Common Lisp を使うのですからマクロを用いて、必要なものを指定すれば勝手に関数を生成してくれるようにしましょう。

構文解析器に必要なパラメータは

- リテラル (演算子と括弧以外の字句) に対する処理
- 演算子とその優先順位、結合性、還元時の処理
- 開き括弧と閉じ括弧、還元時の処理
- 文字列終端記号

の 4 つなので、これら 4 つを指定と関数名を指定すると構文解析を行う関数を定義するマクロ def-op-parser を作ります。こうしておくことで複数の構文解析器が欲しくなったときに楽が出来ます。

ここでは、演算子以外に対する処理は特になく、演算子は「*, /, +, -, =」の五種類とし、優先順位は「=」が一番低く、「*, /」が一番高いように設定し、「=」のみを右結合としそれ以外は左結合とします。それぞれの演算子は還元時に「(* x y)」「(/ x y)」「(+ x y)」「(- x y)」「(setq x y)」という式を生成するようにします。また、括弧は「()」のみを用いて、還元時には括弧の中身をそのまま返すことにします。文字列終端記号は「]」です。以上の要求を満たす構文解析を行う関数を以下のように定義できるようにします。

```
(def-op-parser my-parser (input stack acc)
  ()
  (((#\+ 2 :left)
    (let ((x (pop acc)) (y (pop acc)))
      (push `(+ ,y ,x) acc)))
   ((#\- 2 :left)
    (let ((x (pop acc)) (y (pop acc)))
      (push `(- ,y ,x) acc)))
   ((#\= 1 :right)
    (let ((x (pop acc)) (y (pop acc)))
      (push `(setq ,y ,x) acc)))
   ((#\* 3 :left)
    (let ((x (pop acc)) (y (pop acc)))
      (push `(* ,y ,x) acc)))
   ((#\/ 3 :left)
    (let ((x (pop acc)) (y (pop acc)))
      (push `(/ ,y ,x) acc))))
  ((( #\( #\) ) ))
  #\])
```

[4] my-parser は 1 つ以上の引数を受け取り、第 2 引数以降を無視するように定義していますが、これは、set-macro-character と set-dispatch-macro-character の両方に使えるようにするためです。詳しくは参考文献 [2] の両関数の項目をお読み下さい。

それでは、def-op-parser を作っていきましょう。

1.4.3 def-op-parser の引数

上記 def-op-parser の使用例を振り返ってください。第 1 引数は定義する関数の名前です。第 2 引数はというと、演算子順位構文解析の節で出てきた「読み込んだ字句」「スタック」「結果用のスタック」の 3 つを表す変数です。これらは次に説明するリテラル、演算子、括弧の指定において使用します。

さて、第 3 引数ですが、リテラルに対する処理の指定です。これは次の様な文法で指定することにします。

```
((test-form form1 form2 ...) ...)
```

ここで、「...」は「0 回以上の繰り返し」を表します。test-form を評価し、それが真であれば form1 以降の式が左から順に評価されます。これは cond の節とほぼ同じであり、次のように使います。

```
(((stringp input) (push (intern input) acc))
 ((consp input) (error "bad expression"))
 ((symbolp input) (push input acc)))
```

全ての test-form が偽になった場合、その記号は acc に push されます (つまり上の例の ((symbolp input) (push input acc)) は本当は不要です)。

第 4 引数は演算子に対する処理の指定です。これは次のような文法で指定します。

```
(((op-char precedence association) form1 form2 ...) ...)
```

op-char には演算子を表す文字を指定し、precedence には優先度を表す 1 以上の整数 (大きい方が優先度が高い) を指定、association には:left か:right を指定します。form1, form にはこの演算子が還元するときに行う処理を指定します。

第 5 引数は括弧に対する処理の指定です。これは次のような文法で指定します。

```
(((open-char close-char) form ...) ...)
```

open-char は開き括弧を表す文字、close-char は閉じ括弧を表す文字を指定し、form には還元時の処理を書きます。form は省略可能です (そうすると、括弧の中身が結果用スタックのトップに残ることになります)。

第 6 引数は文字列終端記号を表す文字を指定します。

第 3 引数から第 5 引数まではなんだか cond の文法と似ていますね。これは後で cond の中に突っ込みやすくするための工夫です。それはさておき、それらの文法から特定のものを取り出しやすいように以下のマクロを定義します。

```
(defmacro op-char (spec)
  `(first (car ,spec)))
(defmacro op-precedence (spec)
  `(second (car ,spec)))
(defmacro op-association (spec)
  `(third (car ,spec)))
(defmacro op-body (spec)
  `(cdr ,spec))
(defmacro lt-test (spec)
  `(car ,spec))
(defmacro lt-body (spec)
  `(cdr ,spec))
(defmacro pr-open (spec)
  `(first (car ,spec)))
(defmacro pr-close (spec)
```

```
  `(second (car ,spec)))
(defmacro pr-body (spec)
  `(cdr ,spec))
```

1.4.4 区切り文字の生成

構文解析を始める前に、全ての演算子、括弧及び文字列終端記号を区切り文字に設定します。まず、「指定された文字を “その文字自身を返すマクロ文字” に設定する式を返す」関数 set-self-macro-character を定義し、演算子、括弧は map 系の関数を用い全ての記号に適用し、文字列終端記号は直接適用することにします。

```
(defun set-self-macro-character (char)
  `(set-macro-character ,char
                        #'(lambda (s c)
                            (declare (ignore s c))
                            ,char)))

(defun set-op-macro-character (operators)
  (mapcar #'(lambda (op)
              (set-self-macro-character (op-char op)))
          operators))

(defun set-pr-macro-character (parentheses)
  (mapcan #'(lambda (pr)
              `(,(set-self-macro-character
                   (pr-open pr))
                ,(set-self-macro-character
                   (pr-close pr))))
          parentheses))
```

1.4.5 シフトと還元の処理

さて、次にスタックトップと読み込んだ字句を比較して、シフトか還元を行う処理を作ります。演算子順位構文解析の肝となる部分であるだけに、少々長いコードとなっていますが、演算子順位構文解析のアルゴリズムとあわせてみたらそれほど難しいものではありません。

```
(defun make-action-table (op operators parentheses stack)
  (let ((i-prec (op-precedence op))
        (i-char (op-char op)))
    `(ecase (car ,stack)
       ,@(mapcar
          #'(lambda (op2)
              (let ((s-prec (op-precedence op2))
                    (s-assoc (op-association op2))
                    (s-char (op-char op2)))
                `((,s-char)
                  ,@(if (or (> i-prec s-prec)
                            (and (= i-prec s-prec)
                                 (eq s-assoc :right)))
                        `((push ,i-char ,stack)
                          :shift)
                      `((pop ,stack)
                        ,@(op-body op2)
                        :reduce)))))
          operators))))
```

```
        operators)
      (,(mapcar #'(lambda (pr) (pr-open pr))
                parentheses)
       (push ,i-char ,stack))
      ((:bottom)
       (push ,i-char ,stack)
       :shift))))
```

この関数は第 1 引数に指定された演算子を読み込んだ際に、スタックトップにあわせた動作を行うコードを生成します。スタックには演算子か開き括弧しか PUSH されないので、ecase を使って全ての場合に対する場合分けを行います。また、演算子順位構文解析のアルゴリズムの説明では最初にスタックに文字列終端記号を PUSH していましたが、:bottom というシンボルを PUSH するように若干アルゴリズムを変更し、それに対する処理も書いています。また、シフトが行われた際には:shift、還元が行われた際には:reduce が評価されます。これら 2 つの値は後ほど使用します。

1.4.6 リテラルの指定を cond の節に展開

ここからしばらくは、読み込んだ字句に対する場合分けのコード生成を考えて生きます。この場合分けは cond で行うことにします。

リテラルの指定はそのまま cond の節として利用できるのですが、一応、関数を一段挟むことにしましょう。

```
(defun make-literal-cond-clause (literals)
  (mapcar #'(lambda (l)
              `(,(lt-test l)
                ,@(lt-body l)))
          literals))
```

色々してますが、現時点の仕様では literals と同じ形のリストが返されます。

1.4.7 演算子の指定を cond の節に展開

次に、演算子が入力された場合の処理を作ります。入力が演算子と同じ文字であれば、シフトまたは還元を行いますが、シフトと還元を行うコードの生成は既に作ってあります。還元は可能な限り連続して行うので次の様なソースになります。

```
(defun make-operator-cond-clause
       (operators parentheses input stack)
  (mapcar #'(lambda (o)
              `((eql ,(op-char o) ,input)
                (do ()
                    ((not (eq ,(make-action-table
                                o operators
                                parentheses stack)
                              :reduce))))))
          operators))
```

1.4.8 括弧の指定を cond の節に展開

同様に、括弧が入力された場合の処理を作ります。開き括弧が入力されればシフト、閉じ括弧が入力されれば対応する開き括弧を見つけるまで、スタックを POP しながら還元を繰り返しますが、この処理はすぐ後で定義する reduce-to 関数に任せ、先に cond の節に展開するコードを書いてしまいます。

```
(defun make-parenthesis-cond-clause
       (parentheses operators input stack)
  (mapcan #'(lambda (p)
              `(((eql ,(pr-open p) ,input)
                 (push ,input ,stack))
                ((eql ,(pr-close p) ,input)
                 ,(reduce-to (pr-open p) operators
                             parentheses stack)
                 ,@(pr-body p))))
          parentheses))
```

1.4.9 特定のものを見つけるまで還元

さて、先ほど使用した関数 reduce-to を作ります。この関数はスタックから特定のものを POP するまで還元を繰り返します。還元を行うために優先順位 0 (def-op-parser で指定できるのは 1 以上なのでもっとも優先順位が低い) の演算子に対する処理を生成し、利用しています。

```
(defun reduce-to (to operators parentheses stack)
  `(do ()
       ((or (eq (car ,stack) ,to) (null ,stack))
        (when (null ,stack)
          (error "Syntax error! (by op-parser)"))
        (pop ,stack))
     ,(make-action-table '((#\Null 0 :left) nil)
                         operators parentheses stack)))
```

この関数は括弧の対応を取る以外にも、最後に文字列終端記号を読み込んだ際に使用します。

1.4.10 いよいよ完成

必要なパーツは全て出来上がりました。あとはこれらを組み合わせるだけです。そう、ついにマクロ def-op-parser を書くときが来たのです。

```
(defmacro def-op-parser (name (input stack acc)
                         literals operators parentheses
                         &optional (delimiter :eof))
  (let ((stream (gensym))
        (rest (gensym)))
    `(defun ,name (,stream &rest ,rest)
      (declare (ignore ,rest))
      (let ((*readtable* (copy-readtable)))
        ,@(set-op-macro-character operators)
        ,@(set-pr-macro-character parentheses)
        ,@(when (characterp delimiter)
            (list (set-self-macro-character delimiter)))
        (do ((,input (read ,stream nil :eof nil)
                     (read ,stream nil :eof nil))
             (,stack (list :bottom))
             ,acc)
            ((or (eq ,input :eof)
                 (eql ,input ,delimiter))
             ,(reduce-to
                :bottom operators parentheses stack)
```

```
      (car ,acc))
    (cond ,@(make-literal-cond-clause literals)
          ,@(make-operator-cond-clause
              operators parentheses input stack)
          ,@(make-parenthesis-cond-clause
              parentheses operators input stack)
          (t (push ,input ,acc))))))))
```

まず、区切り文字を設定して、後は文字列終端記号を読み込むまでループ。ループの内容はというと、cond を使ってリテラル、演算子、括弧に対する場合分けを行い、どれにも当てはまらなければシフト。以上の処理を行う関数を定義。

最も長いマクロではありますが、大したことはやってません。何はともあれこれで完成です。def-op-parser に関するプログラムを全てロードし、以前出てきた def-op-parser の使用例をコンパイルしてロードすると、目的を満たす構文解析を行う関数 my-parser が出来上がるので、これまた以前出てきたとおりに set-dispatch-macro-character で登録してやると、全ての準備が整います。

1.4.11 いざ実行！

高まる気持ちを抑えつつ、ゆっくりと目標であった式を入力。そして、Enter を押し式を評価。

```
CL-USER> (let (x y)
           #[ x = (y=5*(4+3)) - 2 ]
           (format t "~&x=~A~%y=~A~%" x y))
x=33
y=35
NIL
```

見事動作しました[5]。もう「Lisp が中置記法が使えなくて不便な言語」なんて言わせません!!!

1.5 終わりに

1.5.1 効率

「へー、Lispってこんなことも出来るんだ。けど遅いんじゃない？」苦し紛れにこんな反応を示す方もいるでしょう。確かに私が書いたコードの効率がいいとはとても言いがたいです。

しかし、中置記法が本来の Lisp の式に変換される処理は「リード時」に行われます。具体的には、インタプリタにソースが読み込まれるときや、ソースをコンパイルするために読み込むときです。インタプリタで実行する際には若干のロスが生まれてしまいますが、コンパイルしてしまえば、コンパイル時間が若干遅くなるだけで、実行時には何一つ余計な時間は掛かりません。

言語自体のシンタックスを追加するという高度な抽象化を行いながら、実行時に余計な時間が掛からないというのは、Common Lisp ならではでしょう。

1.5.2 今後の展望

今回作ったのは「Common Lisp のコードに別のシンタックスを埋め込む」といったものでしたが、私には「全く別の言語を Common Lisp のコードとしてリードする」という野望があります。リードさえ出来てしまえばコンパイルは Common Lisp の処理系ががんばってくれるので、実質コンパイラが出来てしまうことになります。

[5] 実際には REPL を使って関数やマクロをいくつか作るごとに試していたため、ほぼ確実に動くのが分かっていたんですけどね (笑)

実行時に Common Lisp の処理系が必要なことが気になるのであれば、ECL[6]のような処理系を使えば C のソースを吐かせたり、C からコンパイル済みのファイルを読み込ますことが出来てしまうので、そういった手段を使うのもいいかもしれません。

だらだらと長い記事になってしまいましたが、最後までお読み頂きありがとうございました。本記事により少しでも Common Lisp の面白さが伝われば何よりです。

```
(defvar lisp '#1=(#1# is SUGOKU powerful!))
```

[6]http://ecls.sourceforge.net/

参考文献

[1] 中田育男 『コンパイラの構成と最適化』 朝倉書店

[2] Kent Pitman, X3J13 Project Editor 「Common Lisp HyperSpec」
〈http://www.lispworks.com/documentation/HyperSpec/Front/index.htm〉

2 バレルシフタ

電子システム工学課程 3 回生 小宮山 敦史

2.1 シフタって何？

バレルシフタについて話す前に、シフタがどのようなものか、からはじめた方がよさそうだ。

例えば、8bit 入力を 1 bit 左シフトなら、といった回路を作ればよい。しかし、1bit シフトだけでは、複数 bit

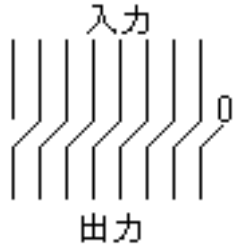

図 2.1: 1bit シフタ

シフタする場合に時間がかかる。たとえば 5bit シフトしたい時には 5 回命令を実行しなければならず、5 倍の時間を所用してしまう。

というわけで、1 度に複数 bit シフトする回路も作ってみる。2bit シフトはこんな感じで、

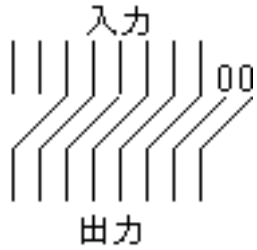

図 2.2: 2bit シフタ

この調子で 3bit シフタ、4bit シフタと作っていけばよい。8bit なら 7bit シフタまで作れば大丈夫だ[1]。

しかし、こうすると回路が 7 つできる上に、そのうちのどれを使うかを選択するものを作る必要がある。

これは場所をとるし面倒だ。シフト部分は IC を使わないからそれほどでもないかもしれないが、16bit、32bit となってくるとこれでは大変なことになる。15 や 31 の回路から 1 つ選択となり、14 や 30 個も無駄な回路ができてしまう。

[1] 8bit シフトすると全部 0 になってしまう

2.2 バレルシフタ

というわけで、1 つの回路で済む方法を取ってみる[2]。

8bit 入力なら、以下のような回路を作れば、シフトするかしないかを 3 回選べば 0 から 7bit のシフトができる。選択部分はデマルチプレクサを使う[3]。ここで「8」は、線 8 本をまとめたという意味である。さすがにこの量となると 1 本 1 本書いていたのでは大きすぎなので省略した。つまり、この図で 1 つの OR もしくはデマルチプレクサは、2 入力 1 出力が 8 個分ある。

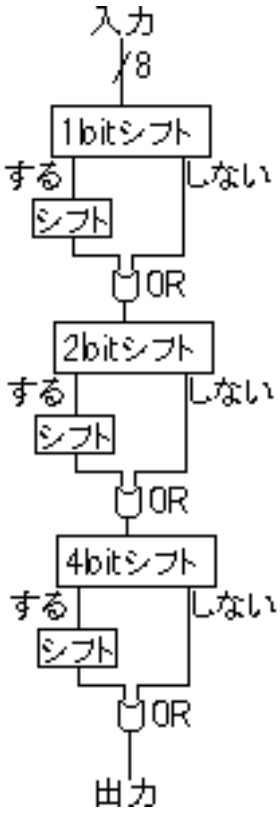

図 2.3: 8bit 入力バレルシフタ

2.3 バレルシフタ改

しかし、これはデマルチプレクサの代わりにセレクタを使えば、OR 回路が不要となる。具体的には、この方法で行けば、16bit、32bit と増えるたびに、セレクタの段が 1 つずつ増える。最初の方法だと、2 倍+1 ずつ回路が増えるのにだ。バレルシフタは入力 bit が大きくなるほど有利なようだ。

ところでちょっとぐぐったらバレルシフタってローテートするものが普通なのか。今回ではシフタといってるので空きには 0 が入るようなものを想定しているが[4]、問題は無いだろう。

8bit なので、バレルのありがたみが薄いが、入門としてはいいのではないだろうか。

[2]まあ、その 1 つが少々大きいのだが
[3]最初に私が考えていた回路である
[4]そうか、シフト部分にセレクタを用意すれば、シフトとローテートを切り替えられるのか！

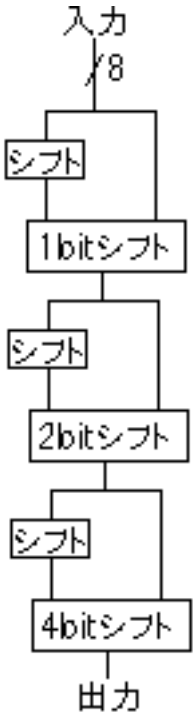

図 2.4: バレルシフタ改

2.4 おわび

これは、私が作る羽目になった[5]バレルシフタについて、基本的な構造について言及したものである。

実際の製作物について軽く触れると、前記の基本構造に、右シフト、左シフトができるように 3-state バッファを加えたものである。

CPU に追加、という形になるため、向こうにもシーケンサの作り直しなどで手間をかけさてしまった。にもかかわらず、動く保証はまだない。執筆段階では、一通り半田付けは終わったのだが、ショートしていたり、間違ってつけた部分があったりするかどうかの確認がまだできていない。そうでなくても、配線がごちゃごちゃ、設計図と全く違うなどひどい有様である。実際の回路製作の大変さが身に染みた。

新聞やワイヤストリッパーを貸してくれた小長谷氏、半田付けの姿勢が悪いと気遣ってくれた[6]楠氏ら、効率低下を防ぐために代わりにニッパーを持ってきてくれた林氏に、深く謝意を示したい。

[5]小長谷製作中の CPU に付け足すよう提案したら、言いだしっぺがやることになった。
[6]結局最後の方まで直さなかったので、しばらく腰が痛く寝れなかった。

参考文献

[1] 〈www.icsd2.tj.chiba-u.jp/ namba/lecture/lab6/notes7.htm〉

3 電光掲示板について

電子システム工学課程 3 回生 小長谷 拓

3.1 はじめに

部室にはコンピュータ部 OB が製作した、発光ダイオードが大量についているプラスチックの板 (以下、LED マトリクス基板と呼ぶことにします) が 6,7 枚あり、使われずに置いてありました。使わずに置いておくのはもったいないので、それらを用いて電光掲示板を作りました (といっても、2 枚しか使っていませんが)。その原理と回路の解説をします。

ただし、LED マトリクス基板 1 枚だけは、OB が制御回路を製作し、毎年新入生向けのクラブ紹介のときに使用されています。

3.2 電光掲示板の点灯方式

一般の電光掲示板や数字を表示する 7 セグメント LED を多数点灯させる場合、主に 2 つの点灯方式があります。一つはダイナミック点灯方式、もう一つはスタティック点灯方式と呼ばれるものです。

例えば、4×8 ドットの LED マトリクスボードがあるとします。これは、4 行 8 列の LED が並んだ板だと考えてください。LED の数は合計 $4 \times 8 = 32$ 個です。この LED マトリクスボードに”A”を表示させるとします。

3.2.1 ダイナミック点灯方式

図 3.4 にダイナミック点灯方式で”A”を表示する方法を示します。図中の 2,3,4,5 を高速で繰り返し表示すると人間の目には図中の 1, のように見えます。つまり、ある一時点では、LED は 1 行しか光っていませんが、点灯させる行を高速に切り替えることによって、全ての行の LED が同時に光っているように見せているのです。ダイナミック点灯方式は、スタティック点灯方式に比べて、配線の本数が少なくてすみます。しかし、ちらつきがあるという欠点もあります。行を切り替える速さをはやくすると、ちらつきは感じにくくなります。

3.2.2 スタティック点灯方式

スタティック点灯方式は単に図 3.4 の 1, のように、”A”という形に LED をずっと光らせておく方法です。”A”を表示するのに必要な LED は同時にずっと光っています。よって、ちらつきはありません。しかし、ダイナミック点灯方式に比べて配線の本数が多くなります。LED が 32 個あれば、それらが個別に点灯、消灯できるようにしなければなりませんので、スイッチが 32 個必要になるということです。

私が作った電光掲示板は、ダイナミック点灯方式で光らせています。なにしろ、LED が 16×64 個も付いているのですから。ダイナミック点灯方式の場合、LED マトリクスボードに接続される線の本数は 16+64 = 80 本だけですみます。スタティック点灯方式の場合 16×64+1 本必要になります。

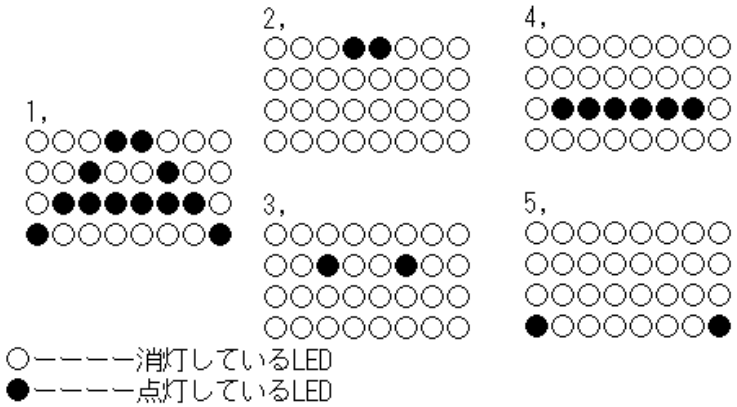

図 3.1: ダイナミック点灯方式

3.2.3 LED マトリクスボードの回路

ダイナミック点灯方式で用いる LED マトリクスボードの回路図を図 3.2 に示します。ただし、図 3.2 は 4 × 8 ドットの LED マトリクスです。実際は 16 × 64 ドットですが、回路は基本的に同じです。

例えば、図 3.2 の 3 行 4 列の LED を光らせたいときは、図 3.3 のように、row3 に電源の +、col4 に電源の − をつなげるとよいことがわかります。また、3 行 4 列の LED と 3 行 6 列の LED を同時に光らせるには、row3 に電源の +、col4 と col6 に電源の − をつなげればよいのです。しかし、3 行 4 列の LED と 4 行 3 列の、2 つの LED だけを同時に光らせることはできるでしょうか？row3 と row4 に電源の +、col3 と col4 に電源の − をつなぐと 3 行 4 列、4 行 4 列、4 行 3 列、4 行 4 列の、合計 4 つの LED が同時に光ってしまいます。よって、違う行の LED を同時に、しかも目的どおりに光らせることはできません。そこで、ダイナミック点灯方式を用いて、1 行ずつ光らせます。

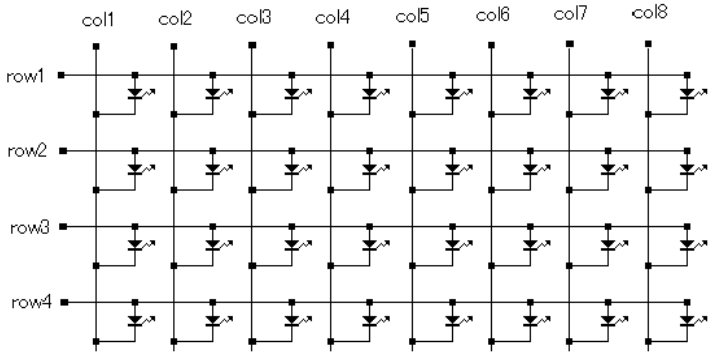

図 3.2: LED マトリクスボードの回路図

3.3 制御回路

16 × 64 個の LED を思い通りに点灯させなければなりません。電光掲示板のブロック図を図 3.4 に示します。ダイナミック点灯方式を使って LED マトリクスを点灯させるので、まず、どれか一つの行を選んで、その一

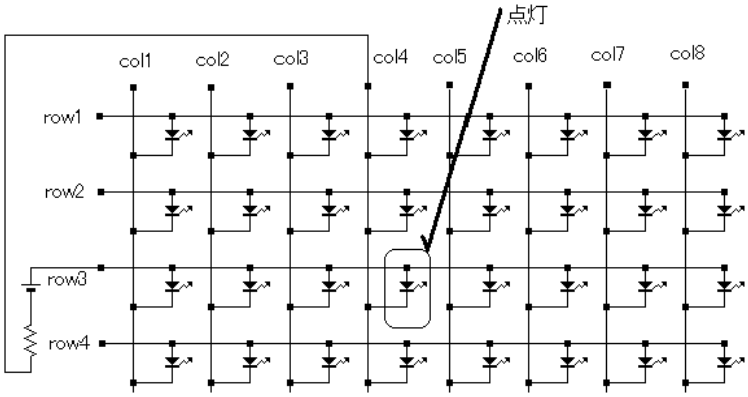

図 3.3: 3 行 4 列を光らせる

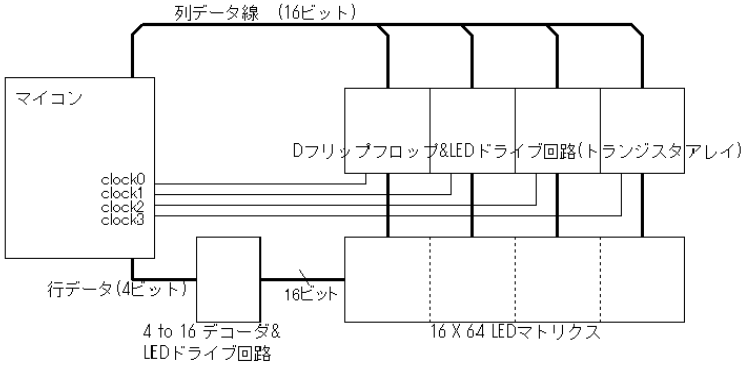

図 3.4: 電光掲示板のブロック図

行のみ光るようにします。図 3.4 では、行データと書いてある線に、4 ビットの 2 進数で光らせたい行の番号を送ります。例えば、5 行目を選ぶときは、0101 を行データとして送ります。そのデータは 4to16 デコーダでデコードされます。4to16 デコーダにはロジック IC 74HC154 を使用しており、例えば、先の例では、74HC154 の出力の 5 番目のピンのみが L レベルになります。

しかし、行を選んだだけでは LED は点灯しません。次は、列データに点灯させたいパターンを送ります。

私の製作した回路では、マイコンから出る配線の本数をできるだけ少なくしたかったので、LED マトリクスを縦に 4 つの区画に分けて、個別に列データを送るようにしました。合計 64 列あるので、一つの区画は 16×16 ドットです。列データを送る線は、16 ビットのバスのようになっています。列データは、1 区画のうちの 1 行でひとかたまりになったデータであり、これをマイコンのメモリに書き込んでおきます。行データは、どの区画でも共通です。そのままでは列データ線に送ったデータは全ての区画にたどり着いてしまうので、目的の区画を clock0 ～ 3 を使って選びます。clock 入力を L レベルから H レベルにすると、列データが D フリップフロップに書き込まれます。同時に、D フリップフロップに書き込んだデータの通りに LED が点灯します。ここには、D フリップフロップが 8 つ入った 74HC273 を使用しました。1 個では 8 ビットしか記憶できないので、2 個まとめて、16 ビットの記憶ができるようにしました。列データを受け取る D フリップフロップとドライブ回路の回路図を図 3.5 に示します。ただし、図は 1 区画分の回路で、図中の U1、U2 が 74HC273、U3、U4 が NPN トランジスタアレイ TD62083AP です。

一つの区画への書き込みが終わったら、次の区画を書き込みます。4 つの区画全ての書き込みが終われば、少

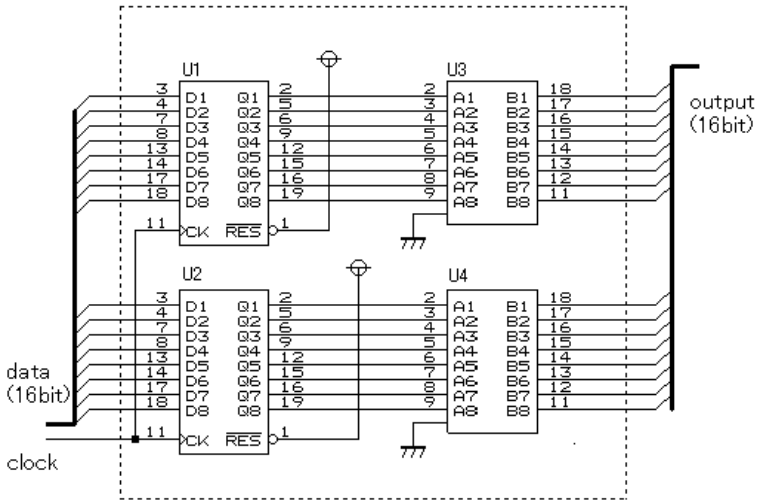

図 3.5: D フリップフロップと LED ドライブ回路

しの間そのままの状態を保ち、次は、1 つ目の区画から順に LED を消灯します。それから、行データに 1 を加算し、次の行を点灯できるようにします。これを繰り返します。

このような回路構成にした結果、必要なマイコンの汎用入出力ポートは合計 24 本程度で済みました。列データをシリアルで送れば、もっと配線の本数を少なくできるのですが。

マイコンは Interface 誌についていた SH7144F です。ちょうど部室にあったので、使いました。このマイコンで LED を光らせるだけというのは、かなりもったいないという気もしますが、マイコンのみ取り外して別の用途にも使えるので、そのマイコンを使用しました。

3.4 最後に

製作した回路を動かすソフトウエアに関しての説明が全くしませんでしたが、それも間単にできてしまいます。行データの方は 0 から 15 まで 1 ずつ加算してゆくだけです。列データは、列データ線に一区画一行のデータを出力し、D フリップフロップの clock 信号を入力してやればよいだけです。文字を表示してスクロールするのもソフトウエア次第でできます。もう少し行を増やせば、ちょっとしたドット絵なども表示できたかもしれません。

参考文献

[1] 東芝 「TD62083AP データシート」
〈http://www.semicon.toshiba.co.jp/docs/datasheet/
ja/LinearIC/TD62081AF_TD62084AP_ja_datasheet_060612.pdf〉

4 人工無能

情報工学課程 3 回生 田村 真司

4.1 はじめに

はい、どうも未だに初心者を抜けていない田村です。説明するのは人工無能について。

人工知能ではありません (ここ重要！)。なんだよ人工知能じゃねぇのかよ、という人にはごめんなさい。それでもいい、という人は少しの時間お付き合いください。

構成としては大きく分けて

1. 人工無能について
2. 人工無能を作成

の 2 つとなっています。

4.2 これから作る人工無能について

4.2.1 人工無能って何？

まずは人工無能について。結構な人は人工知能 (AI) については多少なりとも聞き覚えがあると思います。ではこれから説明する人工無能とはなんなのか？以下ははてなキーワードからの引用です。

> 主にチャット等で発言の中のキーワードに反応して適当な対応を返すプログラムのこと。従来の人工知能研究とは異なり、会話における『表象の現象だけ』を考えて会話をシミュレートしようとするアプローチを取る。
>
> 古くは Eliza から最近では「どこでもいっしょ」のトロに至るまで、様々な人工無能がある。

以上が人工無能についての説明です。まあ簡単にまとめると、「人工無能は人間からみるとまるで考えているように見えるが実はプログラムに従って受け答えしてるだけ」です。ようするに事前に決められた行動を行うだけで何も考えてない＝人工無能です。

4.2.2 どういう人工無能がある？

人工無能には大きく分けてハッシュ型、ログ型、マルコフ連鎖型の 3 種類があります。

ハッシュ型は入力文字列の中に辞書のキーにヒットするものがあるかどうかを検索し、ヒットした場合その辞書記憶の内容を出力します。

ログ型はチャットのログをそのまま辞書に転用し、入力文字列と辞書の記憶が似ていればヒットし、一部の名詞等を替えて出力するものです。

マルコフ連鎖型はチャットのログからランダムに文書らしきものを生成する辞書を起こしておき、辞書のキーにヒットするものを選んでから文書を生成させる。

4.2.3 ここではどんな人工無能を作るの？

今回作るプログラムでの動作は「1 人以上」が相手ではなく、1 対 1 の会話プログラムで、目標は「それっぽい」会話をしてもらうこと。たまに間違うのはご愛嬌。

わかり易くプログラミングしやすいハッシュ型の考え方からを始め、マルコフ連鎖にも手を出してみます。その名も「むのー」君。そのまんまです。シンプルイズベストってことでひとつ（おい）。

4.3 人工無能を作ってみよう

4.3.1 最初に

まずは人工無能の基本的な枠組みを作って、そのプログラムを実行してみたのが以下です。

```
入力をどうぞ
あなた：どうも
むのー：どうも
あなた：元気ですか？
むのー：元気ですか？
あなた：あなたに聞いているのですが
むのー：あなたに聞いているのですが
あなた：……
むのー：……
```

はい、これが簡単な枠組みです。い、痛い、石を投げないで。このプログラムは人が話しかけ、むの一君が答える。その答えが鸚鵡返しになっているだけです。これから「それっぽい」会話をするようにむの一君を改造していきます。

4.3.2 「それっぽい」会話に向けて　その 1「キーワードに反応させてみる」

人間同士の会話では形式通りの会話というものがあります。具体的に言えば挨拶に関することです。

「ありがとう」には「どういたしまして」のように定型通りの答えを返せばぱっと見では人間のように見える、はずです。

でもこれには誰でも気づくように欠点があります。予想外のことを言われたときにどうするか、という問題です。先ほどと同じように鸚鵡返しにするのも芸がないですし、ここでは登録した相槌を無作為に選んで表示させるようにしてみます。

そのプログラムを実行してみたのが以下です。

むのー：メッセージをどーぞ
あなた：こんにちは
むのー：こんちゃー
あなた：ありがとう
むのー：どういたしまして
あなた：挨拶はしっかりできましたね
むのー：いいともー
あなた：でも相槌は意味不明ですね
むのー：ですです

上の例では「こんにちは」には「こんちゃー」、「ありがとう」には「どういたしまして」が登録されています。

また、適当に相槌 (のようなもの) を登録しておきランダムに出てきたのが下の二文です。ランダムですので一つ目の「いいともー」は意味が通じていないように見え、二つ目の「ですです」はまるでこちらの言葉に反応したかのように見えている、という訳です。

これで少しはマシになってきたような気がしてきませんか。今は 2 つしかキーワードとして登録していませんが、キーワードの数を増やせばまるで人間と会話しているように見える、かもしれません。

しかしよく考えてみるとこのままでは同じ語を入力すると毎回同じ返事しかしない、まるで機械のような返答しかしません。実際機械なんだしいーじゃん、という意見もあるかもしれませんが「それっぽい」会話を目指しているので解決策を考えます。

これも力技で解決するなら、一つのキーワードに対して何個も回答を用意しておきその中からランダムに一つを返答として返すという方法があります。具体例を挙げると、「こんにちは」という 1 つのキーワードに対して「こんちゃー」、「はろー」、「昼ですね」と 1 つ以上の返答を登録しておくやり方です。

当然、他にも欠点があります。単語に間違った反応をする可能性です。例えばキーワード「雨」に対して「濡れてしまうから嫌ですね」と返すようにしてしまうと、「今日は雨じゃなくてよかったよ」という文に対して「濡れてしまうから嫌ですね」と返してしまいます。

じゃあもっとキーワードの条件を絞ればいいじゃないか？と言われるかもしれません。そうすると登録するキーワードの量が尋常な量じゃなくなってしまう上に今度はなかなかヒットしてくれません。

間違ってもいいから学習データを入れて自動で作ってみるのもいいかもしれませんが、今回はある程度のキーワードを手動で入力しておいてキーワードに合致しないときにはマルコフ連鎖での文章生成を行い、返答させてみます。

4.3.3 「それっぽい」会話に向けて その 2「マルコフ連鎖を使おう」

マルコフ連鎖とは、ある出来事がいつでもその直近の出来事のみに影響を受けて確率的に生じるとする考え方です。

これだけでは何が言いたいのかさっぱりなので実際に文を作ってみましょう。まず例として文章を二つ。

「今日は晴れで暑いです」、「私は元気です」

まずこの 2 文に対して形態素 (文法的に意味のある最小の言語要素) の切り出しを行います。

「今日　は　晴れ　で　暑い　です」、「私　は　元気　です」

次に切り出した形態素から文字の連鎖を考えます。

(先頭) →今日　私
今日→は
私→は
は→晴れ　元気
晴れ→で
で→暑い
暑い→です
元気→です

(先頭) からは 50 %の確率で「今日」、「私」のどちらかが選ばれます。それと同様に「は」からは「暑い」か「元気」が 50 %の確率で連鎖します。

文頭からはじめ、次に続く形態素の中からランダムに一つ選んで文章を伸ばしていきます。

例)「今日　は　元気　です」

しかし必ずしも正しい文になるわけではなく、むしろこの方法だけではほとんど正しい文はできないでしょう。さっきの例だけでも「私　は　晴れ　で　暑い　です」という意味不明な文になる可能性があるのが分かります。

4.3.4 「それっぽい」会話に向けて　その 3「今までの考えを改良してみる」

前節でマルコフ連鎖だけではまともな文にならないだろう、と判断しました。じゃあこれからどうするか？それの解決策の内のいくつかをこの節で紹介しようと思います。

1. 文脈を意識してみる

前のマルコフ連鎖では前の入力文がどんなものであっても (先頭) から無作為に文章を作っていました。これではいきなり意味不明のことを話し始める危ない人 (?) になってしまいます。

文脈を意識させるには前の入力文を使います。前の入力文の中にある名詞を一つ無作為に選び、その名詞をマルコフ連鎖の開始文字にすればまるでむの一君が文脈を考えて返事してくれるように見えます。

2. マルコフ連鎖を二重にしてみる

まあ字をみればある程度予測はつくと思いますが、前の 2 形態素から次の形態素を選びます。前節の例を使って同じように表を作ってみます。

(先頭) →今日　私
(先頭)+今日→は
(先頭)+私→は
今日＋は→晴れ
私+は→元気
は+晴れ→で
晴れ+で→暑い
で+暑い→です
は+元気→です

この表をみれば分かりますが、前節で例として出した「私は晴れで暑いです」という意味不明な文が出てきません。手前 2 形態素を見るので単純マルコフ連鎖に比べて正しい文を作る可能性が高くなります。

ではこれを 4 重、5 重にすればより正しい文ができるのか？と疑問を持つ人もいると思います。確かにより正しい表現の文はでると考えられますが、その反面プログラミング時のデータの取り扱いが難しくなったり必要とされるデータ量が増加したりなどの問題が出てくる可能性があります。

4.3.5 これからのアイディア

これまでの考えは大体実装してみましたが、以下に書くのはこれからしようと思っていることです。自分が完全に理解しているとは言えませんが、人工無能を作ってみようと思った人に参考程度になれば嬉しいです。

1. ログ型の考え方も導入してみる

マルコフ連鎖だけではデータが増えたときになかなか文が終わらず意味不明な文章になってしまうことがあります。これの解決法としてログ型の考え方を使います。ログ型は最初に少し説明しましたが、事前に登録した文と似たような文がきたときに登録した文の名詞や動詞だけを替えて文章を作る方法です。

事前に品詞の連鎖を登録しておき、それに従ってマルコフ連鎖を行います。例えば前の例と同じ「私は元気です」を辞書に登録する場合は「名詞」＋「助詞」＋「名詞」＋「助動詞」となります。品詞だけを登録しておくことによりある程度の自由性が確保され長すぎる文にもならない、かもしれません。

2. 文脈をもっと読んでみる

前節で紹介したときは、一つ前の相手の文の名詞を選んで次の文を作っていました。しかし、もしかしたらもっと前からその話題が続いているかもしれません。これを解決するのに二つ前の相手の文からも名詞を選んでみればより文脈を考えてくれるかもしれません。

また、名詞だけではなく動詞もマルコフ連鎖のキーとして選んでみるのも効果的だと思います。ただ、この場合前節で紹介したマルコフ連鎖は一方向しか流れがないのでそのままでは使えない可能性が高いです。これの解決策としてはマルコフ連鎖を双方向にすれば動詞から文章が作れるようになると思われます。

3. キャラ (性格) 付け

ある程度しっかりした受け答えができるようになってからですが、辞書に登録する時に使う学習データをいじることによってその人工無能の性格を作ることができます。しかし、これは人との受け答えをしているときなどリアルタイムで辞書に登録する作りかたをしていると人と話しているうちに性格が変わっていってしまいます。

つまり、キャラ (性格) 付けしたければ自分が決めた性格に合う台詞だけを載せた学習データを手動で作る必要が出てくるわけです。自分の計画通りの性格になるかは人工無能の性格に懸ける愛しだいです。

4.4 最後に

マルコフ連鎖を説明するときに簡単に「形態素に切り出して〜」とか言ってますが、これはかなり難しく自分ではできませんでした。形態素に切り出すことを形態素解析というのですが、どうやって助詞と動詞の送り仮名を見分けるのかさっぱり分かりません。しかし心配は無用です。頭のいい方たちが開発した形態素解析エンジンをフリーで公開していらっしゃいます。自分は MeCab(和布蕪) を使わせて頂きましたが、他にも茶筌や KAKASI などいくつかあるのでいろいろ試してみてください。

最後になりますが、こんな文にお付き合い頂きありがとうございました。これを読んで人工無能への興味が少しでも出てきていれば幸いです。しかしこれは人工無能を作ろうと思ったときの初歩的な知識です。試行錯誤して自分だけの人工無能に挑戦してみてください。

参考文献

[1] 小高知宏 『はじめての AI プログラミング』 オーム社

[2] しまりす「人工無脳は考える」
〈http://www.ycf.nanet.co.jp/ skato/muno/〉

[3] Jyakky「人工無能のつくりかたー文章生成の初歩の初歩ー」
〈http://npca.my-sv.net/npca/bulletin/200605.pdf〉

[4] Jyakky「はてなキーワード (人工無能)」
〈http://d.hatena.ne.jp/keyword/%BF%CD%B9%A9%CC%B5%C7%BD〉

[5] 京都大学情報学研究科-日本電信電話株式会社コミュニケーション科学基礎研究所共同研究ユニットプロジェクト 「MeCab: Yet Another Part-of-Speech and Morphological Analyzer」
〈http://mecab.sourceforge.net/〉

5 OpenMPで遊んでみた

情報工学課程 2 回生 出原 真人

5.1 はじめに

5.1.1 OpenMPってなんなのさ

OpenMP とは「共有メモリ型の並列プログラミングのための標準 API」のこと、だそうです。共有メモリ型は「並列プログラミングに参加する複数の処理の流れがメモリを共有している」という意味らしいんですが、詳しいことはよく知りません。とりあえず、プログラムが書ければいいです。OpenMP は逐次プログラムに OpenMP のための行を足していくだけでプログラムを並列化することができます。聞く限り便利そうな感じです、気に入りました。並列プログラミングに興味はあるし、ちょうど開発環境もある、渡りに船ですね。と、いうことで今回の目的に移ります。

5.1.2 今回の目的

思い起こせば 1 年前、学園祭に展示したシューティングゲーム。ロード画面にも何か動きを持たせたい、でもロード中はそっちにかかりきりになって画面更新できないし……。と、よくわからない葛藤を経て最終的には「起動時に全てロードしておいて、ロード画面を入れたい場所ではロード画面っぽい何かを表示させる」という地点に着地。見た感じロード画面なんだけど実際には何もしていないという不思議な形態をとりました。未だにどうすればいいのかよくわかりませんが、今回は OpenMP を使って「ロード画面を表示する」部分と「ロードする」部分にスレッド分けして並列実行させることで無理矢理解決させてみようと思います。

5.2 主要な指示文

5.2.1 指示文の基本

本編は一瞬で終わってしまいそうなので、折角ですから OpenMP の指示文について少し解説しておきます。今回は C 言語から OpenMP を使うので**”#pragma omp 指示文”** の形が基本です。では、指示文の解説に移ります。

5.2.2 #pragma omp parallel

次に続く文、またはブロックを複数スレッドで並列に実行しますよ、ということ。これをしないと何も始まらない。
”#pragma omp parallel num_threads(スレッド数)”のようにすれば、並列実行するスレッドの数も指定可能です。スレッド数がコアの数を超えると高速化目的で並列プログラミングをしていた場合ほとんど意味がなくなるけど、目的が目的なんで今回は関係ないです。マルチコアじゃない CPU では動きませんなんて意味がない。

5.2.3 #pragma omp for

parallel で指定された部分で使うと、後続のループ文を分割して並列化してくれる指示文。parallel 指示文のみでは基本的に全スレッドで同じコードを実行しますが、for 指示文で指定された for 文は各スレッドで異なるループ部分を実行してくれます、便利ですね。ループの分割方法は **scedule** 指示節を付け加えることで細かく制御できます。代表的なもので以下のようなものがあります。
・schedule(static, チャンクサイズ)
繰返しをチャンクサイズ毎に分割して、各スレッドに順番に割り当てる。
全スレッドに割り当て終えたら最初のスレッドに戻って割り当てる。
・schedule(dynamic, チャンクサイズ)
繰返しをチャンクサイズ毎に分割して、暇なスレッドに動的に割り当てる。

5.2.4 #pragma omp sections , #pragma omp section

parallel による並列実行ブロックの中、もしくは parallel 指示文と同じ行に指定して使います。sections 指示文で指定されたブロックの中に、section 指示文で指定された文あるいはブロックが複数あればそれぞれを別のスレッドで実行してくれます。

5.2.5 #pragma omp barrier

並列実行ブロックの中で使うと、全スレッドを同期してくれます。つまり、並列実行されているスレッドが全部このポイントに到達するまで待機してくれるということ。全部到達したら残りの処理を並列実行開始。こんな感じの同期をバリア同期と呼びます。

5.2.6 #pragma omp single

並列実行ブロックの中で使うと、後続の文、もしくはブロックを 1 スレッドでのみ実行しれくれます。この指示文があった場合、暗黙的なバリア同期が発生して全てのスレッドが到達するまで待ち合わせをします。

5.3 変数とか関数とか

5.3.1 shared , private

parallel 指示文では、指示文以前に定義されている、またはブロック内で static で定義されている変数はスレッド間で自動的に共有されます。ブロック内で宣言された自動変数は各スレッドでローカルの変数になります。これらの挙動を変えたい場合は指示文の後に”shared(共有変数にしたい変数のリスト)”や”private(ローカル変数に

したい変数のリスト)”をに付け加えます。
例：#pragma omp parallel private(i,j)
できるだけ明示的に示しておかないと予期せぬ共有などが起こってバグの原因になったりするので要注意です。

5.3.2 関数たち

omp.h をインクルードすることで指示文の他にも便利な関数が使えるようになります。そのうちのいくつかを紹介しておきます。

・void omp_set_num_threads(int)
使うスレッドの数を指定できます。

・int omp_get_num_threads()
スレッドの数を返します。

・int omp_get_thread_num()
現在実行中のスレッドの番号を返します。

・int omp_max_thread()
スレッドの最大生成数を返します。

・int omp_get_num_procs()
プログラムで使用可能なプロセッサの数を返します。

5.4 実際にやってみた

面白くなって色々試してみたのだけれど、本編とあんまり関係ないので割愛。はたしてロード画面はうまくいくのか。意気揚々とソース製作開始！
……の、ハズだったのだが、流石に 1 年前のコード、読める気がしない。
混沌としすぎている、おまけに並列化にむいていない書き方……。
仕方がない、諦めよう。と、いうわけにもいかないので新たなプロジェクトを作成し、ロード画面だけでも作ることに。
無事関数の移植が終わり、いよいよ実行。肝心の関数はこんな感じ。

```
void ompTest()
{
    flag=true;
    //グローバル変数の flag を用意しておく。
    #pragma omp parallel num_threads(2)
    {
        #pragma omp sections
        {
            #pragma omp section
             load();
        //ロード関数。全部ロードしたら flag を false に変える。

            #pragma omp section
             while(flag==true)loadDisp();
        //ロード画面の表示処理をする関数。
        }
    }
    /* Hello,OpenMP World! と表示する処理 */
}
```

……結果、ロード画面をしばらく表示した後、
ロードした画像・音楽と共に Hello,OpenMP World! と表示された。なんだか当初の目的とは微妙にずれてし

まった気がしないこともないけれど、無事にロード画面を作成することができた。Mission Complete !

5.5 終わりに

なんだかよくわからない駄文にここまでお付き合いいただき有難うございます。お、こいつバカなことやってんな、とか思ってもらっても構いません。でも、これを機に少しでも OpenMP に興味を持って頂ければ幸いです。

参考文献

[1] 安田絹子, 小林林広, 飯塚博道, 阿部貴之, 青柳信吾 『マルチコア CPU のための並列プログラミング』 株式会社秀和システム

6 STGで全方位武器を作ってみたよ

情報工学課程 2 回生 中井 道

6.1 はじめに

6.1.1 はじめにのはじめに

本学のコンピュータ部では Lime という部誌を出しているそうです。はい、この冊子のことですね。私は 1 回生の春休みから入部したのでこれの記事を書くのは初めてなのですが書いてみようと思います。なんでそんな微妙な時期に入部したのかというと、ノート PC を買ったので早く入りたかったからだと思います。もともと 1 回生の時に入部しなかった理由がノート PC を持っていなかったというものですから。ちなみに隠された理由として、後輩よりも先に入りたかったというチープなものもあります。

6.1.2 STG

それで、1 年遅れで入部した私ですが、主に STG（シューティングゲーム）を作っていました。動機はなんでしょう？……去年友人が作っているのを見ておもしろそうだと思ったことでしょうか。まあ、何かゲームが作りたかったのでしょう。あまりよく覚えていません。開発言語は C 言語です。また、山田　巧氏の DX ライブラリを使わせて頂いています。また、今回の内容には直接関係はしませんが、STG のアルゴリズムは「14 歳からはじめる C 言語わくわくゲームプログラミング教室」という書籍を参考にさせていただきました。あと、スクリーンショットの画像の一部もその書籍に付属していた画像を使っています。

昔に某社の「RPG が簡単に作れるソフト」を使って RPG を作っていたことがありますが、「ああ、こんなこともできるんだ〜。」「ここをこうすればすごいのが出来るんじゃないか？」の連続で結局最後まで完成しませんでした。ちゃんと初めに全体を設計してしまわないといけないですね。技の追加とかエフェクトの強化とかは後回しにすべきだと思います。大学も設計工学域[1] に入ったのですから今度こそ設計は大事にしないといけないですね。計画的に作成すると何処かの誰かに誓っておきましょう。

6.1.3 今回の内容

STG で全方位武器を作ってみました。はい、誓いは破られましたね。そう、誓いなんて好奇心の前には無力なんです。ちなみに、STG 自体はボス戦は試作してみましたがスクロールする通常のステージには全く手をつけていません。このゲームは学園祭で展示を予定していますが、こんなことで本当に間に合うのでしょうか。

で、内容なのですが、「全方位武器」ってなんでしょう。オールレンジ武器とも言うのでしょうか。実は名前がよくわからなかったのでなんかイメージでそう言ってみました。どんなものかというと分離式自律機動型の無線式兵装端末群です…といっても分かりにくいですね。某ロボットアニメの進化した人類が使っている敵機を囲

[1] 本学は平成 18 年 4 月より、従来の 2 学部（工芸学部、繊維学部）7 学科を工芸科学部として 1 つに統合し、3 学域 10 課程に再編しました。(http://www.kit.ac.jp/03/03_020100.html)

んで攻撃できる武器を想像してもらえればいいと思います。まあ、「漏斗」なアレのことです。以後は何となく意味が通じそうな気がする「ビット」という名称を使うことにします。

6.2 実際にやってみた

6.2.1 目標の動きを考える

まず目標の動きは考えましょう。まあ、理想は某ロボットアニメの…なわけですがね。あちらと違ってこっちは 2D で製作しているので奥行きは考えなくていいので多少は楽ですね。まあ、自分的に実装してみたい動作を書き出してみましょう。

1. 自機の位置から展開
2. 最後は自機に格納
3. ランダムに移動する
4. 各ビットが複数回同じ敵機を攻撃
5. 複数機を同時攻撃
6. 1 機を集中攻撃
7. 及び弾を避ける
8. 耐久力をつける
9. 残像というか軌跡をつける
10. 自機の周りに展開してバリアを張る
11. 流れるような動き
12. フェイントをかける

まあ、こんな感じですかね。簡単そうなのやら難しそうなのやらが混じっていますね。順に考えていきましょう。

1，2 ですがこれは可能でしょう。3 , 4 もできそうです。5 と 6 は今回はどちらか 1 つにしますってことで 5 を実装したいと思います。といっても敵機が 1 機になれば同じですが。7 は今の私には少し無理ですね。8 は 7 が無理なので、もし実装したとすると我先にと敵機に突貫してすぐ自滅するので実装できません。9 は難しそうですけど実装したいです。10 は実装したいですがちょっと無理ですね…また次の機会に。11 も難しいですので、今回は直線的な動きでいきたいと思います。12 は別に簡単につけられるのですが…気分で今回は見送ります。では残ったのを書き出してみますね。

1. 自機の位置から展開
2. 最後は自機に格納
3. ランダムに移動する
4. 各ビットが複数回同じ敵機を攻撃
5. 複数機を同時攻撃
9. 残像というか軌跡をつける

6.2.2 動作をまとめてみた

全体的な動作処理を大まかにまとめましょう。

1. 展開
2. 敵機の選択
3. 敵機の周囲にランダム移動
4. 攻撃（一定回数攻撃するまでは 3 に戻る）
5. 格納

まず、上の動作処理に入る前に自機、敵機、ビット（6 機にします）の座標などの情報を記録しておく必要があります。自機は 1 つですので構造体を用意しておきます。ビットの場合は複数存在しますので各ビットの情報を記録するために構造体の配列を用意します。敵機も複数のデータを扱っていますので構造体の配列を用意しますが、今回は特に触れません。

では、ビットの構造体のメンバで今回の説明に関わるものを箇条書きにしておきます。

1. 現在の x, y 座標
2. 1 フレーム前の x, y 座標
3. 移動先の座標
4. 角度
5. 出現 flag(初期値:FALSE、画面上に存在しているかを示す、TRUE で存在)
6. 状態 pat（数ある動作状態を管理、pat は pattern の略、関係する定数を大小関係とともに 示すと、RECEIVE < FIRST < EXPAND < STOP < ATTACK）

各変数及び定数は使用時に適宜説明します。以下はビット 1 つの処理です。ですので、以下のものを 6 回ループするのですね。ではアルゴリズムを考えていきましょう。

6.2.3 展開

まずは、展開ってことで、展開っていうからには拡がるんでしょうね…自機を中心として。拡がるのはビットであってバグでないことを祈ります。

まあ、攻撃なんでボタンが押されたら射出ですよね。というわけで、ボタンが押されたらビットを出現させることにします。今回はビットの構造体配列の出現していないデータ（「出現 flag」が FALSE のもの）を見つけるまで探す処理をループさせるようにします。見つけたら「出現 flag」を TRUE にして現在の座標に自機の座標を代入します。それから展開するする方向ですが今回はビットは 6 機ですので綺麗に等間隔で展開してもいいのですが、いろいろ試したいのでランダムの方向にします。で、その角度を記録しておきます。

展開するのですから先ほどの角度に一定フレーム動きます。この「一定フレーム」ですが毎フレームごとに+1される「状態 pat」（初期値 FIRST）を作って記録しておけばそれにしたがって動作を決められます。以後の処理もこの変数で処理を決めたいと思います。

展開はこの変数が定数 EXPAND になるまで繰り返します。EXPAND になればある程度拡がったと判断して展開は終了し、次の状態に移行します。

この展開等これからのビットの状態の判定は if 文をとにかく並べました。基本的にはこの展開のように「状態 pat」の値による判定ですが、違う時もあります。

6.2.4 敵機の選択

次はどの敵機を狙うかを決めますが、展開してすぐに敵機に向かうのもあれなので…回してみましょう。もうくるくると、くるくるくるくるくるくると。まあ、約 1 回転だけですが。「状態 pat」は定数 STOP まで。

STOP になったら、回すのを終了しそろそろ敵機を選びましょう。一時利用の配列を作り、対象となる敵機の構造体配列（これも「出現 flag」を含む）をすべて回し「出現 flag」が TRUE のもののアドレスをすべて記録します。ちなみに後の攻撃を考えてこの敵機を選ぶ処理は先に選んだ敵機が存在していないときにのみ行います。存在しているなら新たに敵機を選ぶ必要がありませんから。そして、一時利用配列の中からランダムで 1 機を選び、その敵機周囲の座標をランダムに取り、座標とビットからその座標への角度を記録しておきます。

6.2.5 敵機の周囲にランダム移動

ランダムに移動するということで…さっきからやたらと「ランダム」って単語を使いすぎているような気がしてきました。「ラ」を「ガ」に置き換えたくなるような単語ですが、「乱雑に」とか、「乱数で」とかの方がいいのでしょうか。まあ、今回は「ランダム」のままでいきます。

で、ランダム移動とのことですが、ランダムなところは実は上の項で終わってます。ということで、この項の題名は嘘っぱちです。ただのノイズだと思えばいいかもしれません。ですので、ただ移動します。ひたすらに。先程の座標まで。ただ、1 フレームで割と移動しますので飛び越すかもしれません。というか非常に高確率で飛び越します。ですので、目標座標のある程度近くまで行けばいいということにします。

あと、この動作の間は「状態 pat」に関係なく動きどれだけの間動くか分からないので、この変数は変化させないことにします。ということは、+1 する処理は最後に一括ではなく各状態の最後に書くということになります。回避方法もないことはないですがこのままでいきます。見づらいです。

6.2.6 攻撃

移動したらいよいよ攻撃です。if 文で進んできたので目標座標までついたのなら「状態 pat」が STOP という条件だけで次の動作に移れます。

攻撃ですが今ビットは敵機の周囲にいますので、まず、そこから敵機に照準を向ける必要がありますので角度を敵機の方向に向け、その角度に多少の誤差を付加します。で、すぐに発射しては強くなりすぎるので一定フレーム待つことにします。「状態 pat」が ATTACK になるまででいいでしょう。そして、肝心の攻撃なのですが実はあまり満足していません。今のところ、「状態 pat」が ATTACK になったら、細長い画像（これがレーザーのつもり…）を射線上に表示するのですが、これだと発射してから 1 フレームで敵機まで到達するのですよ。「レーザーは速いんだ！」ってことでもいいですが、というか初めは私もそう思ってこのように実装しました。が、しかし実際に実装してみるとやっぱりいまいちな感じがしました。ちゃんと発射してから相手に向かって行ってほしいです。

ビットが一定回数敵機を攻撃したら、「状態 pat」を RECEIVE に設定しておきます。一定回数の攻撃を終えていないときは、「状態 pat」を EXPAND に戻し、またくるくるまわってもらい同じ処理を繰り返してもらいましょう。

6.2.7 格納

さて、最後は格納です。これは簡単ですね。この格納の処理を if 文の一番初めに挿入しておき、「状態 pat」が RECEIVE になった時に発動するように設定しておけばいいです。ここで、格納の処理を 1 番初めに挿入する理由及び、RECEIVE が関連する定数の中で 1 番小さい理由ですが、関連する変数の値はビットの動き方をいろいろ試す上でよく値が変わりましたので、ATTACK よりも大きくしておくと修正が面倒でした。格納は特に変更がなかったのでそれを考えて初期値 FIRST よりも小さくしました。けれど、そうすると初めの展開の処理で「状態 pat」が EXPAND よりも小さい時という条件と被り、展開が誤作動しましたので一番最初に挿入しました。格納処理自体は自機が動くことを考えて毎フレーム自機の位置に照準を向けて移動すればいいです。あと、

移動の時と同じで座標に幅を持たせないと飛び越します。

6.2.8 残像というか軌跡

さて、上に書いた処理は全部終わりましたが、まだ残像というか軌跡の処理が残っています。「残像というか軌跡」ってさっきからしつこいですよね。まあ、一応意味があって…って言うか無かったらただの阿呆です。これは実体験から来ていまして、当然というか当たり前というか…その 2 つは同じ意味だというか、むしろ当たり前でないかもしれませんが、この文章を書く前にプログラムは何度も書いて試してみました。それで、はじめは残像をつけようとしたのですよ。ビットの 1 フレーム前の座標を保持しておき、その座標に色を変えたり透明度を変えたりしたものを表示してみましたのですが、ビットがある程度早いので重ならず飛び飛びに表示されました。あまり綺麗でありません。元々早く動くの残像がなくてもそう見えるときがあるのに、それに加えて残像とは…ビット多すぎます。辺りがビットビットビット…で、画面世界がビットに支配されました。

ということで、残像をやめ軌跡にしました。軌跡はビットの現フレームの座標と 1 フレーム前の座標とを線で結んだもので、フレームごとに透明度を変化させました。なかなか、綺麗になり満足です。これは DX ライブラリ本家の web ページにあるサンプルプログラムの「ホーミングレーザー 1」を参考にさせていただきました。残像の方もサンプルプログラムの残像処理を若干参考にさせていただきました。

6.2.9 動かしてみた

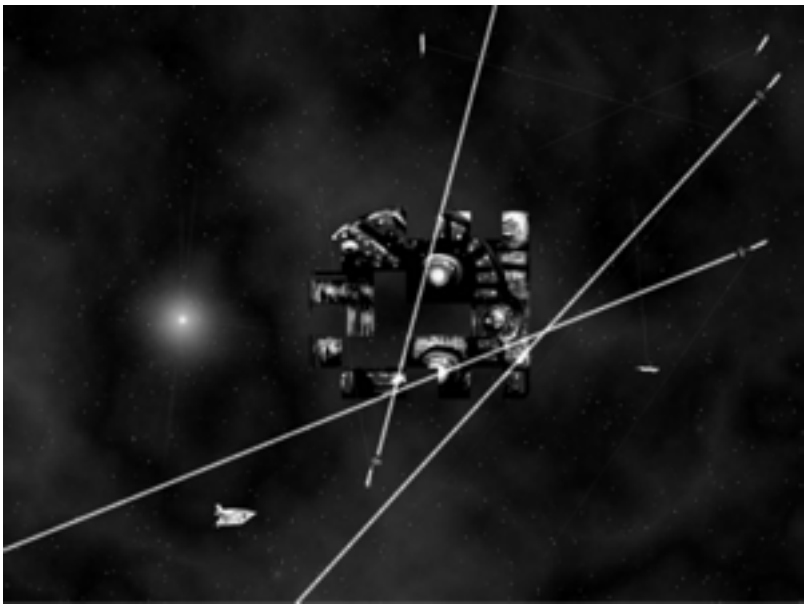

図 6.1: スクリーンショット

前の項でも若干動かした処理が入っていましたが、とにかく動かしてみました。割とうまく動きました。よかったです。そこで、重大な発見が !! 設計段階からある程度予想はしていましたが、格納処理の時だけビットが切り捨てた目標の「流れるような動き」をします。おお、いいです。敵機の周囲に動く時も敵機の動きに合わせて動けばいいかもしれませんが、敵機がビットよりも速く遠ざかっていく場合は攻撃できないとか考えたので今回は見送りました。

6.3 終わりに

悪いところも多々ありますが割とうまくいってよかったです。ただ、すごくプログラムが汚くなったので、今度また整理して 1 から作り直してみたいです。if 文の羅列より良い書き方も探してみたいです。まだ、C の文法で書いているので、C++の文法を勉強してクラスとかを使ってみたいと思います。切り捨てた目標も実現できるようにしたいですし、レーザーも改良してみたいです。このように改善すべき点は多々あるのでこれからもがんばって改善していきたいです。

それにしても、本当にこのゲームは学園祭に間に合うのでしょうか。皆さんも「誓い」は大切にしましょう。まあ、頑張ってみます。

それでは、DX ライブラリ作者の山田 巧 氏、参考書著者の大槻 有一郎 氏、それにこんな文章を最後までじっくり読んでくださった方、飛ばし飛ばし読んで下さった方、「終わりに」のみを読んで下さった方、どうも有り難うございました。

あと、最後にちょっと改造してみました。

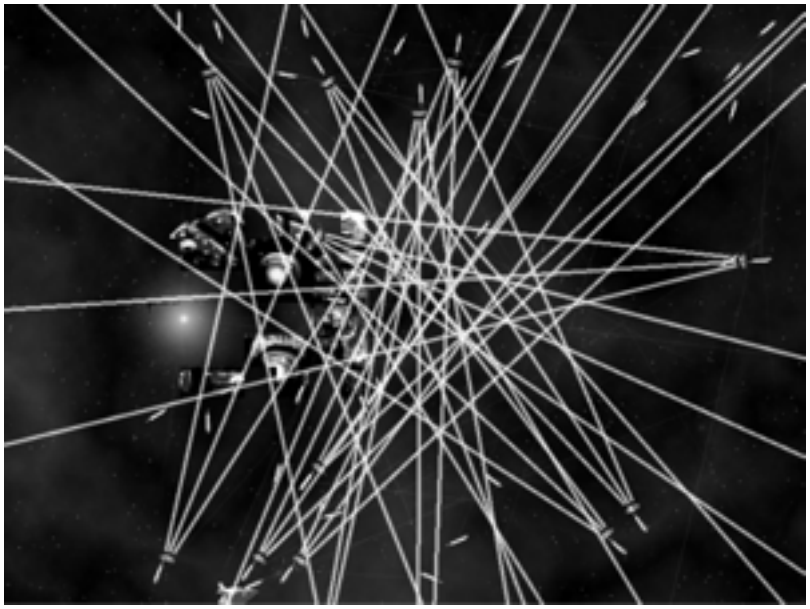

図 6.2: 俺つえー！

参考文献

[1] 山田巧 「ＤＸライブラリ置き場」
〈http://homepage2.nifty.com/natupaji/DxLib/〉

[2] 大槻有一郎 『14 歳からはじめる C 言語わくわくゲームプログラミング教室』 株式会社ラトルズ

7 Java勉強記

情報工学課程 2 回生 米井 将二

7.1 はじめに

最近、Java をひっそりとやってます。夏の合宿では「15 歳からはじめる　わくわくゲームプログラミング」を読みながら、[1]GUI でゲームを作っていたのですが、GUI 用のクラスの扱い方が勉強のメインになってしまい、物足りなかった感じを受けました。またその作業も本読みながらなので、身に付いたとは言えませんでした。なので、今回はオセロを作ってみようかと思い、とりあえず対人用で[2]CUI でできるまでには作ってみました。「習うより慣れよ」の精神で自分なりにやったので、ソースコードを見ると結構、力押しな書き方をしていますが、これが今の自分の実力 … 。まだまだ経験と勉強がいりますね。これから地道に頑張っていこうと思います。

前置きが長くなりましたが、これから Java と自分が作ったオセロについて説明していこうと思います。

7.2 Javaって？

7.2.1 言語の種類

Java は端的に言えばオブジェクト指向の高級言語です。これでは説明になってないので、もう少しまじめに説明します。プログラミング言語には高級言語と低級言語があります。わかりやすく言うと高級言語が人間の考えとか言葉に近い命令とか構文のプログラミング言語で低級言語は機械、コンピュータよりの命令や構文を備えたものです。高級言語は普段、僕たちが使っているような形式の言葉で機械に対して命令を行う言語で低級言語は機械語やそれとほぼ 1 対 1 対応したアセンブリ言語で機械に対して命令を行う言語という認識を僕はしています。おそらくこれであってるんじゃないかと思います。

この高級言語の中でもいろいろと種類が分かれていて、例を挙げると「手続き型」や「オブジェクト指向」などがあり、Cのような手続き型言語は「Aの命令を実行したら、B という命令を実行し、その後にCを … 」といったようにどういう風に命令を行っていくかの”手続き”を書いていきます。これに対して今回、使った Java のようなオブジェクト指向言語はオブジェクトという中にデータや行うべき命令を持った一つの部品ようなものを作って、その部品同士の間で情報のやり取りを行ってプログラムを作るものです。

例を挙げると『車』というオブジェクトと『人』というオブジェクトを作って「人が車を運転する」という内容のプログラムを作る時、『車』の中には”アクセルを踏む”、”ハンドルを〇へ切る”といった風に行うべき[3]メソッドといったものを持たせます。この時、『人』がこのメソッドを車から呼び出して、「人が車を運転する」というプログラムを実現します。

---

[1]Graphical User Interface の略:ユーザーへの情報や操作がグラフィックを多用するもの

[2]Character-based User Interface の略:別名コマンドラインインターフェース、ユーザの操作はキーボードからでユーザーへの情報は文字で行われる

[3]C 言語でいうところの関数、行うべき処理が記述されている

7.2.2 Java の便利な点

Java の良い所は使ってみた感想としては次のように感じました。

1. クラスごとに設計していくのでプログラムの全体像を把握しやすい
2. GUI のクラスが事前に Java の言語に存在するので、GUI でものを作るときは楽
3. [4]継承や[5]オーバーロードといった機能が便利

(1.) に関してはCなどの手続き型言語に比べて、”部品”単位でプログラムを書いていくので感覚としてはプログラムの全体像を把握しやすかったと思います。ただ部品を作っていく時に事前にどの部品がどういった情報を持っているべきかどういった行動を行えるべきかということを念頭に置いていないと頭の中で爆発する気がします。というか、実際、僕は今回の Java オセロ書き始めた時、無計画に書いては参考書籍を見るという行為を繰り返していたので、事前に全体像の設計をすることがオブジェクト指向言語では重要なのかもしれないと感じました。

(2.) については夏合宿の方で作ってたゲームで随分お世話になりました。Cだと DirectX を扱ってウィンドウを作成/表示しようとするとかなりソースを書かないといけなかった気がしますが、Java だと java.awt パッケージの中の Frame クラスや javax.swing クラスの JFrame クラスを使えば[6]インスタンスを作って、メッソドを用いれば比較的簡単に作れるので便利でした。実際は大きさ指定などで処理をいろいろ書かねばなりませんが、それでもC/C++で作るよりは大分楽でした。こういったクラスが事前に Java の言語の中に存在しているのは有難いと思います。

(3.) はCでは似たような関数を作る時に前に作ったやつを流用するのが、困難でした。これも結構便利な機能だと思います。でも、クラスの継承を行うとクラス同士で親と子の関係ができるので、それを気をつけるのが少し苦労する点なのかもしれません。

7.3 Java でオセロ

だらだらと Java の説明をしていても面白くないので、そろそろ自分の書いた Java オセロの説明をしようかと思います。Java は上で説明したように”部品”を作っていく言語なので、オセロに必要な”部品”を考えてみました。

- ボード (盤面)
- 石 (白, 黒)
- オセロを行う人

必要な”部品”はこんな感じだと思います。これら三つの部品をクラスとして作っていったのですが、オセロとしての機能のほとんどはボードの部分が持ち、人のクラスが main メソッドを持って必要に応じてボードの機能を呼び出す形になりました。石のクラスはデータとして『座標』と『色』を持っています。ソースを全て載せると長くなるので一部だけ載せて、説明していこうと思います。

7.3.1 Stone クラス

```
public class Stone extends Coordi{
```

[4]あるクラスの機能を別のクラスに受け継がせること
[5]同じ名前のメソッドであっても引数が違えば別のものと認識する機能
[6]クラスから作られる実体、クラスが型抜きとするとインスタンスは型抜きから作られたクラスと能力の同じ物体

```
        public static final int EMP = 0;
        public static final int WHITE = 1;
        public static final int BLACK = -1;
        public static final int WALL = 2;
        public static final int CANPUT = 3;

        public int color;

            public Stone(){
                  super(0,0);
                  this.color = EMP;
               }

            public Stone(int x,int y,int color){
                  super(x,y);
                  this.color = color;
               }

            public Stone(String co) throws IllegalArgumentException{
                if(co == null || co.length()<2)

 throw new IllegalArgumentException("The argument must be Reversi style coordinates!")

                x = co.charAt(0)-'a'+1;
                y = co.charAt(1)-'1'+1;

            }

            public String toString(){
               String co = new String();

                 co += (char)('a'+ x-1);
                 co += (char)('1'+ y-1);

                 return co;
         }
}
```

これはStoneクラスのソースコードです。短いので全体を載せました。Coordiクラスを継承していますが、Coordiクラスは主に石の座標を取るために作られたクラスです。では、Stoneクラスの説明をしていこうと思います。まず、石の色、盤面の状態を数字で表現するために対応する定数を public static final int ··· の部分で決めています。これにより Stone クラス、その他クラスで各変数名と定数は同一のものとして扱われています。

Stone クラス内に Stone(···){ ··· } と記述されている関数のような部分がいくつかありますが、これはコンストラクタといってStoneクラスのインスタンスが作られた時に同時に行われる処理です。与えられる引数により行う処理が異なってます。Stone(String co) のコンストラクタは与えられた二文字の文字列をオセロの座標用に変換している部分です。文字列が無い or 文字列が二文字未満の時、throw new IllegalArgumentException ···

という部分で例外発生した部分を探しに行き、例外処理を行います。

7.3.2 Board クラス

Board クラスはいろいろな機能を持たせたので、長くなっています。なので、ここでは主にオセロを実現する上で重要だと思う機能についてのみ説明しておきます。

自分の石で相手を挟める方向の取得

以下はそのソースコードです。

```
public int checkMove(Stone stone){
            stone.color = currentcolor;

      if(Board[stone.x][stone.y]!=EMP && Board[stone.x][stone.y]!=CANPUT)
                        return NONE;

               int x,y;
               int dir=NONE;

            /*上判定*/
               if(Board[stone.x][stone.y-1]==-stone.color){
                       x=stone.x;
                       y=stone.y-2;
                   while(Board[x][y]==-stone.color){
                               y--;
                      }
                   if(Board[x][y]==stone.color)
                               dir |= UPPER;
                  }
            /*下判定*/
               if(Board[stone.x][stone.y+1]==-stone.color){
                   x=stone.x;
                   y=stone.y+2;
                while(Board[x][y]==-stone.color){
                           y++;
                     }
                if(Board[x][y]==stone.color)
                           dir |= LOWER;
               }
            /*左判定*/
               if(Board[stone.x-1][stone.y]==-stone.color){
                   x=stone.x-2;
                   y=stone.y;
                while(Board[x][y]==-stone.color){
```

```
                    x--;
                }
                if(Board[x][y]==stone.color)
                        dir |= LEFT;
            }
        /*右判定*/
            if(Board[stone.x+1][stone.y]==-stone.color){
                x=stone.x+2;
                y=stone.y;
                while(Board[x][y]==-stone.color){
                        x++;
                    }
                if(Board[x][y]==stone.color)
                        dir |= RIGHT;
                }
        /*左上判定*/
            if(Board[stone.x-1][stone.y-1]==-stone.color){
                x=stone.x-2;
                y=stone.y-2;
                while(Board[x][y]==-stone.color){
                        x--;
                        y--;
                }
            if(Board[x][y]==stone.color)
                        dir |= UPPER_LEFT;
                }
        /*右上判定*/
            if(Board[stone.x+1][stone.y-1]==-stone.color){
                x=stone.x+2;
                y=stone.y-2;
                while(Board[x][y]==-stone.color){
                        x++;
                        y--;
                    }
                if(Board[x][y]==stone.color)
                        dir |= UPPER_RIGHT;
                }
        /*左下判定*/
            if(Board[stone.x-1][stone.y+1]==-stone.color){
                x=stone.x-2;
                y=stone.y+2;
                while(Board[x][y]==-stone.color){
                        x--;
                        y++;
```

```
            }
          if(Board[x][y]==stone.color)
                      dir |= LOWER_LEFT;
           }
/*右下判定*/
     if(Board[stone.x+1][stone.y+1]==-stone.color){
          x=stone.x+2;
          y=stone.y+2;
      while(Board[x][y]==-stone.color){
             x++;
             y++;
          }
       if(Board[x][y]==stone.color)
              dir |= LOWER_RIGHT;
        }

       return dir; //設置可能方向
  }
```

8 方向のそれぞれを bit で対応させて、フラグとして挟める方向に対応する bit の位に 1 と論理和をとることで 1 を立てます。ある情報を bit の列で符号化するって方法は大学の課題で二分木を符号化する時にも使った記憶があります。よく使われる常套手段なのかはわかりませんが、僕は便利だと思いました。

表 7.1: 方向と対応する bit 列

名前	対応する bit 列	十進数
NONE	00000000	0
UPPER	00000001	1
UPPER __ LEFT	00000010	2
LEFT	00000100	4
LOWER __ LEFT	00001000	8
LOWER	00010000	16
LOWER __ RIGHT	00100000	32
RIGHT	01000000	64
UPPER __ RIGHT	10000000	128

挟んだ石をひっくり返す

```
public void flipStone(Stone stone){
                    int x,y;
                    int dir=checkMove(stone);

                    Board[stone.x][stone.y]=currentcolor;
                /*上*/
                    if((dir & UPPER)!=NONE){
                            y = stone.y;
                            while(Board[stone.x][--y]!=currentcolor){
                                Board[stone.x][y]=currentcolor;
                            }
                       }
                /*下*/
                    if((dir & LOWER)!=NONE){
                            y = stone.y;
                            while(Board[stone.x][++y]!=currentcolor){
                                Board[stone.x][y]=currentcolor;
                            }
                       }
                /*左*/
                    if((dir & LEFT)!=NONE){
                            x = stone.x;
                            while(Board[--x][stone.y]!=currentcolor){
                                Board[x][stone.y]=currentcolor;
                            }
                       }
                /*右*/
                    if((dir & RIGHT)!=NONE){
```

```
                    x = stone.x;
                    while(Board[++x][stone.y]!=currentcolor){
                        Board[x][stone.y]=currentcolor;
                    }
                }
            /*右上*/
                if((dir & UPPER_RIGHT)!=NONE){
                    x = stone.x;
                    y = stone.y;
                    while(Board[++x][--y]!=currentcolor){
                        Board[x][y]=currentcolor;
                    }
                }
            /*左上*/
                if((dir & UPPER_LEFT)!=NONE){
                    x = stone.x;
                    y = stone.y;
                    while(Board[--x][--y]!=currentcolor){
                        Board[x][y]=currentcolor;
                    }
                }
            /*左下*/
                if((dir & LOWER_LEFT)!=NONE){
                    x = stone.x;
                    y = stone.y;
                    while(Board[--x][++y]!=currentcolor){
                        Board[x][y]=currentcolor;
                    }
                }
            /*右下*/
                if((dir & LOWER_RIGHT)!=NONE){
                    x = stone.x;
                    y = stone.y;
                    while(Board[++x][++y]!=currentcolor){
                        Board[x][y]=currentcolor;
                    }
                }
        }
```

これは挟んだ相手の石をひっくり返すメソッドです。アルゴリズムとしてはまず最初に先ほどのひっくり返せる相手のいる方向を得る checkMove メソッドでその方向 dir を得ます。後は各方向と対応した bit 列と dir で論理積をとることでその方向にひっくり返せる相手がいるかいないかを判定します。論理積の結果が 0 でなければ、その方向にひっくり返せる相手がいるということなので、自分と同じ色が出てくるまで盤面をその方向に進み、その道の上にある石を自分の色にしていきます。

終了判定

始まりにかなり力押しの書き方をしたとお話しましたのがこの部分です。このメソッドでは[7]boolean 型として以下の点を満たせば、true を返して終了していることにしました。

- 限界の手数に達しているか？
- 自分の色の石で置ける場所が盤面に無いか？
- 相手の色の石で置ける場所が盤面に無いか？

最初の限界手数を考えるのは手数が限界の 60 手に達しているかを判定すればいいだけなので、力押しをしてるとは思わないのですが、残りの二つの判定では for の二重ループで盤面全体に対して checkMove メソッドを使い、ひっくり返せる相手がいるという結果を一度でも得られれば false を返して呼び出し元へ return しています。どうにかもっと賢くできないかと考えて見たのですが、結局、力押しの結果となったのは残念 … 。

7.3.3 Main クラス

このクラスは”人”に当たるクラスです。この Main クラスが Board クラスの実体であるインスタンスを作り、その中にある上で述べたような機能を呼び出してオセロを実現するわけです。僕のソースでは CUI で盤面を表示するために ConsoleBoard クラスを作り盤面表示ための print メソッドを書き、Board クラスを継承することで Board クラスの機能を付加することでコマンドライン用のオセロのボードの部分を作りました。そのソースを下に載せておきます。

```
class ConsoleBoard extends Board{
    public void print(){
       System.out.println("  a b c d e f g h ");
          for(int y=1;y<=8;y++){
            System.out.print(" "+y);

             for(int x=1;x<=8;x++){

              switch(getColor(new Coordi(x,y))){
                    case Stone.BLACK:
                        System.out.print("●");
                        break;
                    case Stone.WHITE:
                        System.out.print("○");
                        break;
                    case Stone.CANPUT:
                        System.out.print("◇");
                        break;
                    default:
                        System.out.print("  ");
                        break;
                }
```

[7]真と偽の二つの値を持つ型

```
            }
            System.out.println();
        }
    }
}
```

後はこの ConsoleBoard クラスのインスタンスを作って、それを使うことで Main クラスが必要に応じて ConsoleBoard の機能を呼び出していけば完成です。

7.4 今後へ向けて

本文を読んで「AI 入れてないの？馬鹿なの？」という感想を抱いた方が多いと思われます。残念ながら AI の勉強/実装には間に合いませんでした。今後の予定としては今年が終わるまでには AI の実装をしようと目論んでいます。参考書籍のまま書いたら勉強にならないのは目に見えてるので今回は我流で書きました。書籍のソースを参考にはしましたが、我流で書いてるので結構、自分の頭で考えることができたのではないのかと思います。最初は Java と今まで使ってた C との違いに面食らいましたが、最近、Java の使い方もなんとなくわかってきたように思えるので僕としては良い感じです。これからも地道に Java をひっそりとやっていこうと思います。

参考文献

[1] Seal Software 『リバーシのアルゴリズム』 工学社

[2] 畠中晃弘, 江原良典 『ゼロからはじめる Java』 株式会社アスキー

8 CASLIIと私

情報工学課程 2 回生 村上 明男

8.1 はじめに

8.1.1 きっかけ

部室に置いてあった CASLII の本を手に取ったのがきっかけで CASLII に触ってみました。まずプログラミング言語 CASLII とは一体何なのかについて説明します。そして CASLII にはどのような命令があるのかについてほんの一部だけ説明します。

8.1.2 CASLII とは？

CASLII とは、IPA(独立行政法人 情報処理推進機構) が実施する基本情報技術者試験 (FE 試験) で出題対象となっているアセンブリ言語で、仮想のコンピュータである COMETII を使用するためのソフトウェアです。COMETII と CASLII は FE 試験の一部問題において受験者の有利不利の差がつかないように経済産業省がそれぞれ仕様を定めています。
ちなみに、2008 年 10 月に行われた FE 試験では C、COBOL、Java、アセンブリ言語 (CASLII) の 4 言語から選択して問題を解く形式になっていました。

8.2 仕様・コーティング形式

8.2.1 COMETII の仕様

ここでは COMETII の仕様の一部を紹介します。

- COMETII は 1 語 16 ビットのワードマシン

記憶領域の大きさは 65536 語です。

- COMETII は中央処理装置内に GR(汎用レジスタ)、PR(プログラムレジスタ)、FR(フラグレジスタ)、SP(スタックポインタ) の 4 種類のレジスタを持つ

GR(汎用レジスタ) には GR0、GR1、GR2、GR3、GR4、GR5、GR6、GR7 の 8 種類のレジスタがあります。これらは算術演算、論理演算、比較演算、シフト演算に使用されます。
PR(プログラムレジスタ) はプログラムの実行および制御を行うレジスタです。

FR(フラグレジスタ) は演算結果後の GR の状態 (オーバーフローが発生したか否かなど) を保持するレジスタです。
SP(スタックポインタ) はスタックの最上段のアドレスを保持するレジスタです。

- COMETII では数値は 1 語 16 ビット、文字は 1 文字 8 ビットで表現する

その他の仕様については参考文献などを参考にしてください。

8.2.2 CASLII のコーティング仕様

ここでは CASLII のコーティング仕様の一部を紹介します。
CASLII に命令にはアセンブラ命令、マクロ命令、機械語命令の 3 種類があり、コーティングの時にはこれらの命令はラベル欄 (最大 8 文字)、命令コード欄、オペランド欄、注釈欄、未定義欄 (通常は空白) を持ちます。

(例)

```
ラベル      命令コード      オペランド
REI         START
            LD              GR0,INU ;load
            RET
INU         DC              4
            END
```

その他、

- ラベル名は第 1 文字目から英大文字もしくは数字で記述する
- 命令コード欄にはアセンブラ命令、マクロ命令、機械語命令の命令コードを記述する
- オペランド欄には GR、アドレス (ラベル名) を記述する
- オペランドの記述範囲は 72 文字まで
- 注釈欄には処理系で許す任意の文字を書ける
- 注釈行の開始には;(セミコロン) を用いる

などの仕様があります。

8.3 CASLII の命令

8.3.1 開始・終了命令など

ここからは CASLII の命令の一部を見ていきます。くどいですが、この記事で紹介する命令以外の CASLII の命令に関しては参考文献などを参考にしてください。

(例 1)

```
ラベル      命令コード      オペランド
EX1         START
            LAD             GR1,5
            RET
KANI        DC              8
            END
```

START 命令はプログラムの先頭に書かなければならず、ラベルも記述しなければなりません。また、START 命令の次の行からすぐにプログラムを実行する場合はオペランド欄には何も記述しません。
LAD 命令は実効アドレスを GR に設定する命令です。なお、この命令の実行で FR の値が変化することはありません。例 1 での LAD 命令では GR1 に 10 進数の 5 を設定することになります。
DC 命令はプログラムで使用する定数 (10 進数、16 進数、文字、文字列、アドレスが使用可) を定義する命令です。例 1 での DC 命令では 10 進数の 8 を定義することになります。またラベル名を KANI と記述しているのでプログラム処理で参照することが可能になります。このとき、10 進数の 8 は 1 語の領域内に 2 進数表現で格納されます。
RET 命令と END 命令はともに終了命令なのですが、使用目的が違います。アセンブラのプログラムの中には原始プログラムの終了とプログラム実行終了という 2 つの終了命令が存在します。RET 命令はプログラムの実行を終了する命令であり、END 命令は原始プログラムの命令を定義する命令です。

8.3.2 フラグを立てる

CASLII には様々な分岐命令が存在します。ここでは JZE 命令を取り上げました。

(例 2)

```
ラベル      命令コード      オペランド
EX2         START
            LD              GR2,NEKO
            SUBA            GR2,KANI
            JZE             FINISH
            LAD             GR2,666
            ST              GR2,NEKO
FINISH      RET
NEKO        DC              5
KANI        DC              8
            END
```

以下の説明では「番地」という表現を用いますが、これはラベルで指定したアドレスのことを指します。
では例 2 で新しく登場する命令について説明します。

- LD 命令:GR 同士の値の設定、または実効アドレスの内容を GR に設定する命令

例 2 では 8 行目の DC 命令で NEKO 番地の内容に 10 進数の 5 が定義されています。そして、2 行目の LD 命令で NEKO 番地の内容 (10 進数の 5) が GR2 に設定されます。

- ST 命令:GR の内容を実効アドレスが示す番地に格納する命令

例 2 の 6 行目にある ST 命令では、GR2 の内容が NEKO 番地の内容に格納されます。

- SUBA 命令:2 つの GR 同士の減算または GR の内容から実効アドレスが示す番地の内容を減算する命令

例 2 の 3 行目の SUBA 命令では (GR2 の内容 - KANI 番地の内容) が GR2 に設定されます。また、このとき FR は演算終了後の GR の内容によって新しい値が設定されます。

- JZE 命令:FR の値を変える演算命令実行後、GR に設定された値が 0 なら分岐する命令

例 2 の 4 行目の JZE 命令では、3 行目の SUBA 命令によって GR3 に設定された値が 0 なら分岐するのですが、例 2 では GR3 は 0 ではないので分岐しません。もし分岐した場合は分岐先はラベル名 FINISH になり、JZE の下 2 行の命令は未実行の状態で RET 命令を実行 (プログラム実行を終了) することになります。

プログラム実行の要点　SUBA 命令により NEKO 番地の内容と KANI 番地の内容とが一致していれば JZE 命令で分岐先のラベル名 FINISH にジャンプすることになりますが、一致していないので GR2 の内容と NEKO 番地の内容がさらに変化します。

8.4　おわりに

ここまでかなり粗雑な内容になってしまいましたが、CASLII がどのような言語かは少しは伝わったのではと思います。これから基本情報技術者試験を受ける予定がありましたら、CASLII を勉強してから受験してみてはいかがでしょうか？最後に、これを読んで少しでも CASLII に興味を持ってもらえたら幸いです。

参考文献

[1] 平井利明 『CASLII– 仕様と演習』 ムイスリ出版

[2] 情報処理推進機構 「試験で使用する情報処理用語」
〈http://www.jitec.jp/1_02annai/annai_yogo_jis.html〉

9 占いとプログラム

デザイン経営工学課程 1 回生 中野 秀規

9.1 占い

> 占いとはさまざまな方法で、人の心の内や運勢や未来など、直接観察することのできないものについて判断することや、その方法をいう。[1]

手相、人相、風水、血液型、占星術等々、現在テレビや雑誌にも載っている占いですが、大きく分けると命・ト・相（めい・ぼく・そう）の三種類ありますが、今回作ったプログラムでは命にあたる誕生日占い、その中でも西洋占星術を用いました。もちろん私がしたのではありません。
参考 URL と本を最後に載せておきます。
こんな話ばかりでもダメなのでそろそろプログラムの話に移ります。

9.2 プログラム

まずこのプログラムを作ろうと思ったときに、誕生日なので 365 通りに分けないといけないな、と考えました。まだまだC言語に関しては門を叩いたばかりだったので、頭にまず浮かんだのが if 文です。C言語をやっている人にとってはアホらしい話ですが、if 文というのは

if(条件式){ 宣言* 文*}

という形になっていて、if 文が一個の場合、条件文が真であれば実行文を実行し、そうでなければ実行しない (又は他の分を実行する)、と二通りの分岐を作ることができます。また、この後にさらに if 文を続けることで分岐を増やすことができます。if 文の後に else if という用に if 文を置いていくと、if 文の数が 1、2、3、4 … と増えていくにつれて分岐は 2、3、4、5 … と増えていきます。つまり、365 個の分岐をつくろうとすると 364 個の if 文が必要となります。仮に if 文一個で一行としても分岐だけで 364 行、また 365 種類の占い結果を打ち込まなくてはいけないので非常に面倒で、偉そうなことは言えませんがあまり良くないです。

そこで、ファイル操作関数を使うことにしました。そのために、少し面倒ですが各日の占い結果を書いたファイルを月毎に分けて保存しました。それぞれのファイル名は month1.txt〜month12.txt となっていて、まず scanf で占う人の月を読み込んで、その数字からファイルを選びます。ファイルを選んだら次は占う人の日を読み込みます。そういえば、占い結果を保存したファイルの中身は、1 月 1 日を例にとると

1 月 1 日
性格：我が強く …
恋愛：かなり不器用な …
相性のいい人：従順なタイプ …

[1] Wikipedia より

相性の悪い人：わがままな人 …
ラッキーポイント：えび茶色 …

のように 6 行ずつになっていて、この下に同じように 1 月 2 日、3 日と 6 行ずつ続いていきます。この 6 行ずつ続いているというのを利用しました。fgets で 6 行分ファイルから読み取って表示するのですが、占う人が打ち込んだ日の数の回数分 6 行読み込みます。すると前に読み込んだ 6 行に新たな 6 行が上書きされるので、ちゃんと打ち込んだ日の占い結果が表示されるというわけです。分かったような口調で書いていますが、手伝っていただいてやっとできました。ここでお礼を言います。ありがとうございます。

参考文献

[1] 「誕生日占い」
〈http://www.amitaj.or.jp/ m-nakao/uranai/uranai.htm〉

[2] ムッシュムラセ 『[誕生日別] 性格事典 365 の性格・運命・つきあい方がわかる』PHP 出版社

[3] 「占い - Wikipedia」
〈http://ja.wikipedia.org/wiki/%E5%8D%A0%E3%81%84〉

10 カットカットカットカットオオオオ！

京都大学情報学科 1 回生 森下 耕平

10.1 はじめに

かつてチェスや将棋といったゲームは上流階級の人々にとって知性を示す為の物だったと聞きます。そして現在、それらのゲームにおいてコンピュータの実力はかなりのものになってきており、特にチェスでは人間のチャンピオンをコンピュータが打ち負かすまでになりました。そうすると、現代のコンピュータをその時代に持っていけば、そのコンピュータは賢人などと呼ばれたりするのでしょうか……？　そんな訳ありませんよね。

なんだかよくわからない始め方をしてしまいましたが、何の話をするかと言うと、チェスや将棋などのゲームをコンピュータに打たせる時に使われるアルゴリズムの一つ、α-β 法という物の実装に挑戦してみたという話です。といっても、いきなりチェスや将棋のようなルールが複雑なゲームで作るのは大変そうだったので、まずはもっとルールが単純なリバーシ、つまりオセロで作ってみました。

10.2 α-β 法とは

α-β 法とはオセロ、チェス、将棋のように、理論上では全ての手を読み切ることが可能な、完全情報ゲームと言われるゲームにおいて最善手を見つけるためのアルゴリズムの一つで、minimax 法と呼ばれるアルゴリズムを改良した物です。

まず minimax 法について。ゲームをする時、最終的に目指すのは勝利です。言い換えると「相手に勝利している局面」に辿り着くことです。完全情報ゲームでは起こりうる全ての展開が明らかになっていますから、それを一つずつ調べていけば「勝利している局面」までの手筋がどのような物かも明らかになります。ただ、相手も勝利を目指して打っているはずです。相手が悪手を打つのを期待しても仕方がないので、相手も最善手を打ってくるものとして考える事になります。さて、具体的にある局面での、自分側のある手を評価する方法です。評価値は自分側に有利であるほど高くなるとします。

1. その手を打った後の盤面を考えます。
2. その盤面での相手の応手を全て考え、それぞれ評価します。
3. 評価した中で最も評価値の低い手の評価値を今評価している手の評価値とします。

最も評価値が低いということは、最も自分側に不利という事なので、相手側はそれを打ってくると考えます。それ以外の手を打たれた場合は想定より善い状況になったという事なので問題ありません。このようにして、出来る限り状況が悪くならないように手を考えるのが minimax 法です。相手の応手の評価は応手をするのを自分にし、評価値は最も評価値の高い手の評価値として同じ手順を行います。そしてその自分の応手の評価を……というようにして、起きうる展開を先へ先へと調べてゆきます。本当ならこれを開始から終局まで続けて勝てる展開を見つけたい所ですが、途方もない計算が必要になる為事実上不可能です。そこで序盤では先読みの手数を何手までと定め、その手数まで読んだ時点で探索を打ち切り、その場面の有利不利を評価値とすることになり

ます。もし途中に勝てる展開があれば、その評価値は正の無限大、負ける展開なら負の無限大と定めれば良いでしょう。

実際にチェスなどで人間のチャンピオンに勝利しているくらいですから、このアルゴリズムは優秀です。しかし問題になるのは計算量です。先読みの手数を一つ増やすごとに計算量は級数的に増えていくため、終盤以外ではたいした数の先読みが出来ません。この点を多少改良したのがα-β法です。

minimax 法では先読み数の限界になるまで、どんな手だろうと読み進めていきます。そのため明らかに読む必要の無い手筋も読んでしまい、その無駄な計算量も先読み数を増やすごとに飛躍的に増えていきます。α-β法では、もう選ばれない事が確定した手の探索を途中で打ち切ることで計算量を減らし、結果的に先読みできる手数を増やして精度を上昇させる事が出来ます。

それでは具体的なα-β法の手順についてです。基本的には minimax 法と同様に評価を進めてゆきます。そしてたとえば自分手番のある局面で、ある手 A の評価値が x になったとします。そして別の手 B の評価を開始し、B に対する相手のある応手の評価値が x より低い値 y になったとします。相手側の応手は最も評価値が低いものが選ばれるため、この時点で B の評価値は y 以下となり、A の評価値より低くなるので B が自分側の手として選ばれることはなくなります。ここで、これ以上 B の評価を続けるのは無駄になる為、評価を打ち切ります。α-β法では局面にもよりますが大幅な計算量の削減が期待できます。そしてその効果は先読み数を増やすほど大きくなります。

10.3 実装

さて、実際にわたしが書いたソースはこんな感じです。

```
int search(int tmp_player,int depth,int limit){
  int tmp_value,i,j,directions,value=-10000;
  depth--;
  if(depth==0){
    value=point(tmp_player);
  }
  else{
    for(i=0;i<=7;i++){
      for(j=0;j<=7;j++){
        directions=check(i,j,tmp_player);
        if(directions>=1){
          tmp_reverse(i,j,tmp_player,directions,depth);
          tmp_value=search(tmp_player*(-1),depth,value*(-1))*(-1);
          if(tmp_value>=limit){
            recovery(i,j,tmp_player,directions,depth);
            return tmp_value;
          }
          else if(tmp_value>=value){
            value=tmp_value;
          }
          recovery(i,j,tmp_player,directions,depth);
        }
      }
    }
  }
  return value;
}
```

point 関数は盤面の状況を評価して値を返す関数。check 関数は指定されたマスにコマを置いた時、相手のコマをひっくり返せる方向を 8 方向分数値にして返す関数[1]。tmp_reverse 関数は指定されたマスに自分のコマを置いて相手のコマをひっくり返す関数で、recovery 関数がそれを元に戻す関数です。評価値に-1 をかけて比較する事により、自分側の手の評価と相手側の手の評価を一つの関数で行えるようにしています。

10.4 今後の展望

今後は、この文を書いている時点では盤面を評価する関数が全く出来ていない為、まずはそれを作るつもりです。そして後は GUI まで出来ればいいなと思っています。

[1] 8 方向それぞれの置けるか置けないかを 8bit の 1,0 に対応させて一つの数値として返しています。

編集後記

Lime38 号はお楽しみいただけしょうか。

どうも、はじめまして。今回、編集を担当した荒木です。今年の Lime の編集というと、なかなか記事が集まらなくて編集できない印象がありました。締め切りが、繰り返す延長により、当初よりも 1 か月以上伸びていたりして、そうしているうちに、この文を書いている日付から、2 日後には印刷所に行くとかどうとかになってたり。ともあれ、無事こうして、今年も Lime を完成することができました。

是非、来年の Lime もご期待を!

平成 20 年 11 月 9 日 編集担当 荒木 修

Lime Vol.38

平成 20 年 11 月 22 日 発行 第 1 刷

発行 京都工芸繊維大学コンピュータ部
http://www.kitcc.org/