平成 22 年 11 月 20 日
京都工芸繊維大学コンピュータ部

Lime42

はじめに

はじめまして、部長の中野です。本年も無事に Lime を発行することができ、少し肩の荷が下りた気分でございます。今回の Lime の中身はゲーム制作などソフトな活動が多く、創作活動が活発なのは喜ばしいことですが、今後は電子工作などのハードな活動にも力を入れ、また外部の団体との交流を積極的にしてほしいと思います。部の創立以来最も頼りない部長ではありましたが、多くの新入部員の勢いや、部員の助けがあってここまで来れました。この場を借りて心から感謝申し上げます。

では最後までどうぞお楽しみください。
男勝りな編集長、鴛渕さん。編集お疲れさまでした。

平成 20 年 11 月 20 日
コンピュータ部部長 中野 秀規

目 次

1 Common Lisp に継続(みらい)を

京都大学大学院 情報学研究科 修士課程 1 回 林 奉行

1.1 はじめに

Scheme [1] や Ruby [2]，Javascript 1.7 [3] などのプログラミング言語ではプログラムが自身の**継続**を扱えるようになっています。継続はいくつかのプログラミング言語の制御構造をよく一般化していて，これを扱えることによってコルーチンや，それを利用したグリーンスレッド，大域脱出などをプログラム側で記述することが可能です。これらの機能がプログラムを簡潔に記述するのに優れているかは場合によるでしょうが，少なくとも I/O 等々の要請で「続き」を管理する必要があるようなプログラムは書きやすいような気がします。のちに Yahoo Store になった Viaweb のシステムがセッション管理に継続 (のようなもの) を使っていたのは有名な話です。

さて，Common Lisp [4] はコミケカタログにも勝る重さと厚さの仕様書を持つ，(後ろから一発で仕留めるのに) 有用なプログラミング言語ですが，継続を扱う機能は言語仕様に含まれていません。なぜ Common Lisp では継続を扱えないのかは筆者も議論が追えていないところですが，おそらく実行効率の面で問題があったのではないでしょうか。しかしながら，少々効率は落ちても継続を取得したり，呼び出したりしたくなることもあるでしょう。この記事では，そのような要求を満たすべく，Common Lisp で継続を扱う技法について考えます。

1.2 けいぞく!!

継続をどのように扱うかは Scheme を参考にしましょう。Scheme では次のように `call-with-current-continuation` (`call/cc`) 関数で継続を取得 (capture) します。

```
(define call/cc call-with-current-continuation)
(call/cc (lambda (cc) (cc 'a) 'b)) ; => A
```

`call/cc` の引数に関数を渡すと，`call/cc` が呼ばれた時点の継続を引数としてその関数が呼ばれます。Scheme では，継続は 1 引数の関数として実現されています [5]。継続が呼び出されると，呼び出された継続への引数が `call/cc` の値として返されます。つまり，`call/cc` の次の処理に戻ります。上の例では 2 行目を評価するとラムダ式の末尾に`'b` あるにもかかわらず，その前に `(cc 'a)` として継続を呼び出しているため，`call/cc` の返り値として `A` が返ります。

先の例では継続を C 言語でいう `return` 文のように使っていますが，Scheme では継続はファーストクラスのオブジェクトですのでクロージャの外側への持ち出し，外側からの呼び出しも許されます。継続オブジェクトを持ちまわすことでさまざまな制御機構を実装することができるでしょう。例えば，図 1.1 で示した Scheme のプログラムでは `call/cc` を用いて並行処理を実現しています。

```
1  (define cont '())
2
3  (define (enqueue v)
4    (set! cont (append cont (list v))))
5
6  (define (dequeue)
7    (let ((head (car cont)))
8      (set! cont (cdr cont))
9      head))
10
11 (define (yield)
12   (let ((cc (call/cc (lambda (cc) cc))))
13     (if cc
14         (begin (enqueue cc)
15                (let ((next (dequeue)))
16                  (if (not (null? next))
17                      (next #f)))))))
18
19 (define (start)
20   (let ((head (dequeue)))
21     (if (not (null? head))
22         (head #f))))
23
24 (define (sample x)
25     (for-each (lambda (x) (display x) (yield))
26               '(1 2 3 4 5 6 7 8 9 10))))
27
28 (set! cont (list sample sample))
29 top-level> (start)
30 11223344556677889910 10
```

図 1.1: 並行処理 in Scheme

1.3 継続渡し形式

では Common Lisp で call/cc を実装するとして，継続をどうやって取得するのでしょうか。たとえば次のようなプログラムがあったとします。

```
(progn (func1) (call/cc #'(lambda (cc) ...)) (func2) (func3))
```

この場合，cc に渡されるべき継続は,

```
#'(lambda () (func2) (func3))
```

と表せます。次の場合はどうでしょうか。

```
(defun foo ()
  (func1) (call/cc #'(lambda (cc) ...)) (func2) (func3))
```

この場合も先ほどと同じで良いような気がしますが，foo の呼び出され方によっては困ったことになります。例えば次のような場合です。

```
(progn (func4) (foo) (func5) (foo))
```

1 回目に foo が呼ばれたときに foo 内の call/cc が取得すべき継続は,

```
#'(lambda () (func2) (func3) (func5) (foo))
```

ですが 2 回目に呼ばれたときは,

```
#'(lambda () (func2) (func3))
```

です[1]。単純に後ろの部分をパッケージしただけではうまくいかないようです。

ところで，筆者はよく人から夢見がちな性格だと言われますし，自分でも多少そう思っている節があります。例え現実逃避的でも夢を見ることは楽しいですし,「もしも」を思うことは未来への原動力です。ということで,ここでも「もしも」を考えてみることにします。もし，foo を呼び出すときに以降の継続を取得できているとしたらどうでしょう。foo を呼び出すときに foo に以降の継続を教えてやることができるようになります。foo を書き換えてその継続を受け取ってみます。

```
(defun foo2 (cont)
  (func1) (cal/cc #'(lambda (cc) ...)) (func2) (func3))
```

今度は foo2 がどこで呼ばれても，継続は

```
#'(lambda () (func2) (funcall cont (func3)))
```

と表せます。cont の引数に (func3) を評価した値を渡しているのは，(func3) の結果が (foo2) を評価した値になるからです。foo が,

```
(func1 (foo2))
```

のような呼ばれ方をされた場合，この値が使われることになります。

さて，書き換えた foo2 は呼び出された時点で，foo2 が値を返した以降の継続を cont として受け取っています。なので，次のように書き換えても問題ないはずです。

```
(defun foo3 (cont)
  (func1) (call/cc #'(lambda (cc) ...)) (func2) (funcall cont (func3)))
```

[1] スコープのことは今は考えません。

呼び出し側では次のようにします。

```
(progn (func4) (foo #'(lambda () (func5) (foo #'(lambda (x) x)))))
```

`#'(lambda (x) x)` は `#'identity` とも書けて，こちらの方が簡潔ですので以降はこちらを使います。

さらに，`foo3` の中で呼び出されている `func1`〜`func3` も，中で `call/cc` を呼び出しているかもしれないので同様に定義しなおします。すると，`foo3` も次のように書き換える必要があります。

```
(defun foo4 (cont)
  (func1-cps #'(lambda ()
                 (call/cc #'(lambda (cc) ...))
                 (func2-cps #'(lambda () (func3-cps cont))))))
```

「もしも」の話をしていたわけですが，このような変換をプログラム全体に施していけば,「もしも」をとっても良くなります。ここで，晴れて `call/cc` が取得すべき継続は,

```
#'(lambda () (func2-cps #'(lambda () (func3-cps cont))))
```

であると言えるようになります。継続は `call/cc` の第 1 引数として渡すことにしましょう。

```
(defun foo-cps (cont)
  (func1-cps #'(lambda ()
                 (call/cc-cps #'(lambda ()
                                  (func2-cps #'(lambda () (func3-cps cont))))
                              #'(lambda (cc) ...)))))
```

`foo-cps` のような形式を**継続渡し形式** (continuation passing style; CPS) と呼び，通常のプログラムから継続渡し形式に変換することを **CPS 変換** (CPS transformation) と言います [6]。Common Lisp で `call/cc` を実装するために，まずプログラムを CPS 変換することにしましょう。

1.4 CPS 変換

CPS 変換は，慣れるとそう複雑でもないのですが，手で書くとなると長ったらしくて手間です[2]。なので，ここでは自動で CPS 変換するようにしてみたいと思います。

1.4.1 CPS 変換マクロ

受け取った本体を CPS 変換してくれるマクロ `=defun` を定義してみました。

```
(defmacro =defun (name parm &body body)
  (let ((sym (declare-cps-func name)))
    `(progn (defun ,sym (cont ,@parm)
              ,(transform-prg body 'cont))
            (defun ,name (&rest rest)
              (apply (function ,sym) (cons #'identity rest))))))
```

CPS 変換された関数は継続を受け取るための引数がひとつ増えますが，マクロを使う側がそれに気を遣わなくてもよいように 1 枚ラッピングしています。

`declare-cps-func` は名前の通りシンボルに束縛された関数が CPS 変換された関数であることを，宣言するための関数で，次のように定義されています。

[2] てまー。

```
(defparameter *cps-transformed* (make-hash-table))

(defun declare-cps-func (sym)
  (let ((cpsfun (gethash sym *cps-transformed*)))
    (if (null cpsfun)
        (setf (gethash sym *cps-transformed*) (gensym))
        cpsfun)))
```

CPS 変換の際，ある関数が CPS 変換されたものか否かを判断しなくてはならないことがあります。すでに =defun を用いて定義済みであればそれを覚えておけばよいのですが，相互再帰している場合など，これから定義するつもりの関数を呼ぶ場合に困ります。パッケージで区別するなどの方法も考えられますが，今回は CPS 変換された関数であることを，declare-cps-func を用いて先に宣言しておくことにしました。

transform-prg が関数本体を CPS 変換する関数です。定義は次のようになっています。

```
(defun transform-prg (form cont)
  (transform-exp (car form)
                 (if (null (cdr form))
                     cont
                     `(function (lambda (v)
                         (declare (ignore v))
                        ,(transform-prg (cdr form) cont))))))
```

form には CPS 変換すべきフォームが，cont には CPS 変換されたフォームが受け取るべき継続 (ラムダ式) がそれぞれ渡されます。困った時には分割統治法でトップダウンに考えていくとわりとなんでもうまくいくものです。transform-prg でも問題を細かく分割していきます。

CPS 変換は基本的には，

1. CPS 変換するコードの中で最初に評価される部分を CPS 変換する。
2. CPS 変換した部分に残りのコードを CPS 変換したもの (継続) を渡す。

という手順で行われます。1 が終わった時点で，CPS 変換された最初に評価される部分は継続 (continuation) を渡せる (passing) ようになっています。それに残りの部分を CPS 変換して継続として渡す，ということを再帰的に繰り返すことで変換を進めていきます。

CPS 変換する対象のフォームの種類によって,「どの部分が最初に評価されるか」が変わってきますので，その種類に合わせて変換を進めていく必要があります。その場合分けは，transform-exp に記述してあります。

```
(defun transform-exp (form cont)
  (when (atom form)
    (return-from transform-exp `(funcall ,cont ,form)))
  (case (car form)
    (quote `(funcall ,cont ,form))
    (progn (transform-prg (cdr form) cont))
    (if (transform-if form cont))
    (cond (transform-cond form cont))
    (let* (transform-let* form cont))
    (function (transform-fn form cont))
    (lambda (transform-lambda form cont))
    (apply (transform-apply form cont))
    (funcall (transform-funcall form cont))
```

```
    (=funcall (transform-=funcall form))
    (otherwise (transform-func form cont))))
```

変換対象がアトムやクォートされた式だった場合は値を返しているということですので，継続に自身を渡しています。progn の場合はそのまま transform-prg に渡してしまいます。そのほかのフォームについては，とりあえず，if，cond，let*，function，lambda，apply，funcall に対応してみました。以下，それぞれについて見ていきます。

1.4.2 関数呼び出しの CPS 変換

関数呼び出しでは，呼び出されている関数を CPS 変換すればよいのですが,「最初に評価される部分」がその関数であるとは限りません。

```
(func1 (func2) (func3))
```

というような呼び出し方がされている場合は，最初に評価されるのは (func2) です。よって上の関数呼び出しは,

```
(func2 #'(lambda (v1)
   (func3 #'(lambda (v2)
      (func1 cont v1 v2)))))
```

のように変換されなければなりません。よって関数呼び出しの CPS 変換をする関数 transform-func は次のようになります。

```
(defun callp (x)
  (and (consp x)
       (not (macrop (car x)))
       (not (eq (car x) 'quote))))

(defun transform-func-sub (args form gen)
  (if (null form)
      (funcall gen (nreverse args))
      (if (callp (car form))
          (let ((sym (gensym)))
            (transform-exp
             (car form)
             `(function (lambda (,sym)
                ,(transform-func-sub (cons sym args) (cdr form) gen)))))
          (transform-func-sub (cons (car form) args) (cdr form) gen))))

(defun transform-func (form cont)
  (let ((fun (car form))
        (rest (cdr form)))
    (transform-func-sub
     nil rest
     #'(lambda (args)
         (let ((cpsfun (gethash fun *cps-transformed*)))
           (if cpsfun
               `(,cpsfun ,cont ,@args)
               `(funcall ,cont (,fun ,@args))))))))
```

transform-func-sub が高階関数になっているのは，後に apply や funcall などでこの関数を再利用するためです。transform-func から transform-func-sub に gen として渡す関数では，呼び出されている関数が CPS 変換されているか (される予定か) を*cps-transformed*に登録されているか否かで判断しています。CPS 変換されていない関数が呼び出される場合は継続を渡せませんので，そのまま呼び出して戻り値を継続に渡すようにしています。

引数を再帰的に CPS 変換するか否かは callp 関数で判断しています。引数がクォートされた式だった場合の他に，マクロの呼び出しだった場合も CPS 変換しないようにする必要があります。マクロの呼び出しでは引数として書いたフォームが評価されずにそのままマクロに渡されるので，上の変換例のように変換してはプログラムの意味が変わってしまうためです。あるシンボルに束縛されているのが関数かマクロかを判断するための関数は残念ながら Common Lisp では提供されていませんが，各実装ごとに方法は違うものの，一応やり方はあるようで，今回は次のようにしてみました。

```
(defun macrop (sym)
  (when (fboundp sym)
    #+sbcl (sb-impl::closurep (symbol-function sym))
    #+clisp (eq (type-of (symbol-function sym)) 'system::macro)
    #+allegro (eq (type-of (symbol-function sym)) 'excl::closure)))
```

1.4.3 if，cond の CPS 変換

変換対象のフォームが if の場合は，最初がテストフォーム (分岐条件) が評価されますので，transform-if は次のようになります。

```
(defun transform-if (form cont)
  (transform-exp (second form)
                 (let ((sym (gensym)))
                   `(function (lambda (,sym)
                      (if ,sym
                          ,(transform-exp (third form) cont)
                          ,(transform-exp (fourth form) cont)))))))
```

cond は if に簡単に変換できますので，if に変換した後 transform-if に丸投げしてしまいましょう。

```
(defun transform-cond-to-if (form)
  (if (null form)
      nil
      (let ((test (caar form))
            (occa (cdar form)))
        (if (null occa)
            (let ((sym (gensym)))
              `(let* ((,sym ,test))
                 (if ,sym ,sym ,(transform-cond-to-if (cdr form)))))
            `(if ,test
              (progn ,@occa)
              ,(transform-cond-to-if (cdr form)))))))

(defun transform-cond (form cont)
  (transform-if (transform-cond-to-if (cdr form))
                cont))
```

1.4.4 let* の CPS 変換

let の CPS 変換はシンボルの置換をする必要があって手間なので[3]，さしあたって let* を作ります。let* で最初に評価されるのは変数の初期値を求める部分ですので，関数呼び出しの場合と同様順々に入れ子にしていきましょう。

```
(defun transform-let*-sub (vars body cont)
  (if (null vars)
      (transform-prg body cont)
      (cond ((symbolp (car vars))
             `(let (,(car vars))
                ,(transform-let*-sub (cdr vars) body cont)))
            ((consp (car vars))
             (let ((sym (first (car vars)))
                   (init (second (car vars))))
               (if (or (atom init)
                       (eq (car init) 'quote))
                   `(let (,(car vars))
                      ,(transform-let*-sub (cdr vars) body cont))
                   (transform-exp
                     init
                     `(function (lambda (,sym)
                       ,(transform-let*-sub (cdr vars) body cont)))))))
            (t (error "Invalid variable name was specified.")))))

(defun transform-let* (form cont)
  (transform-let*-sub (cadr form) (cddr form) cont))
```

1.4.5 function, lambda，apply，funcall の CPS 変換

後で apply や，funcall などで呼ばれることを考えて，#'foo などの関数やラムダ式も CPS 変換しておきましょう。

```
(defun transform-lambda (form cont)
  (let ((sym (gensym)))
    `(funcall ,cont
      (function (lambda (,sym ,@(second form))
        ,(transform-prg (cddr form) sym))))))

(defun transform-fn (form cont)
  (let ((arg (second form)))
    (cond ((symbolp arg)
           (let ((sym (gensym))
                 (transformed (gethash arg *cps-transformed*)))
             (if transformed
                 `(function ,transformed)
```

[3] てままー。

```
                ‘(function (lambda (&rest ,sym)
                   (funcall (car ,sym)
                            (apply (function ,arg) (cdr ,sym))))))))
          ((consp arg) (transform-exp arg cont))
          (t (error "Error in transform-fn")))))
```

apply や funcall 側では第 1 引数に継続と取るように書き直してやります。

```
(defun transform-apply (form cont)
  (transform-func-sub
   nil (cdr form)
   #’(lambda (args)
       ‘(apply ,(car args) (cons ,cont ,(cdr args))))))

(defun transform-funcall (form cont)
  (transform-func-sub
   nil (cdr form)
   #’(lambda (args)
       ‘(funcall ,(car args) ,cont ,@(cdr args)))))
```

1.4.6 変換例

一通り CPS 変換するコードが書けましたので試してみましょう。次の例,

```
(=defun fib (n)
  (cond ((= n 0) 0)
        ((= n 1) 1)
        (t (+ (fib (- n 1)) (fib (- n 2))))))
```

はおなじみの，n 番目のフィボナッチ数を計算するものですが，これは図 1.2 のように展開されます。……とても手で書く気にはなれませんね。自動化することにして正解でした。

1.5 call/cc の実装

前節までで CPS 変換はできましたので，いよいよ call/cc を実装してみたいと思います。といっても CPS 変換が終わった時点で準備はほとんど終わっています。call/cc は次のようなとても短い関数です。

```
(defun call-with-current-continuation (cont fun)
  (funcall fun cont cont))
```

CPS 変換時にこの関数が呼ばれるように*cps-transformed*にあらかじめ登録しておきましょう。

```
(defparameter *cps-transformed*
  ’(call-with-current-continuation call-with-current-continuation
    call/cc call-with-current-continuation))
```

これで，完成と言いたいところですがまだ一つ問題が残っています。call/cc で取得した継続を呼び出す場合です。継続は引数に前式の結果をとる 1 引数の関数ですが，プログラム中で funcall や apply を用いると CPS 変換時に継続が渡されるように書き換えられて，2 引数で呼び出されてしまします。今回は，継続を呼び出すときには特別な関数=funcall を用いるようにしてみました。

```
(defun #:g01 (cont n)
  (funcall
   #'(lambda (#:g02)
       (if #:g02 (funcall cont 0)
         (funcall
          #'(lambda (#:g03)
              (if #:g03 (funcall cont 1)
                (funcall
                 #'(lambda (#:g04)
                     (if #:g04
                         (funcall
                          #'(lambda (#:g08)
                              (#:g01
                               #'(lambda (#:g05)
                                   (funcall
                                    #'(lambda (#:g07)
                                        (#:g01
                                         #'(lambda (#:g06)
                                             (funcall
                                              cont
                                              (+ #:g05 #:g06)))
                                         #:g07))
                                      (- n 2)))
                               #:g08))
                          (- n 1))
                         (funcall cont nil)))
                 t)))
          (= n 1))))
    (= n 0)))

(defun fib (&rest rest)
  (apply #'#:g01 (cons #'identity rest)))
```

図 1.2: CPS 変換されたフィボナッチ数を求める関数

```
(defun transform-=funcall (form)
  (transform-func-sub
   nil (cdr form)
   #'(lambda (args)
       `(funcall ,@args))))

(defmacro =funcall (&rest rest)
  `(funcall ,@rest))
```

下の=funcall の定義は CPS 変換されていない場所から継続を呼ぶために定義しています。funcall を用いてもよいのですが統一性があった方がよいと思ったので定義してみました。

これで，call/cc が実装できました。

1.6 使ってみる

実際に call/cc を使ってみます。図 1.3 のプログラムは図 1.1 のプログラムを今回実装した call/cc を使って Common Lisp で書き直したものです。期待通り動いています。

1.7 まるめ

Common Lisp で継続を扱う技法として，CPS 変換を用いて Scheme 風の call/cc を実装してみました。Common Lisp のコードを CPS 変換する機会がありまして，CPS 変換するプログラムの書き方を覚えたので，ふと思い立って作ってみましたがなかなか使えそうです。今回用意したフォーム，関数以外にも let や map などあるとより便利になるでしょう。

それでは，ここまでご覧いただいた貴方に良い継続(みらい)を。

1.8 参考文献

[1] R. Kelsey, W. Clinger, and J. Rees (eds.). Revised5 report on the algorithmic language schemea. *ACM SIGPLAN Notices*, 33(9), September 1998.

[2] Dave Thomas, Chad Fowler, and Andy Hunt. *Programming Ruby: The Pragmatic Programmers' Guide, Second Edition.* Pragmatic Bookshelf, October 2004.

[3] Rhino. http://www.mozilla.org/rhino/.

[4] Guy L. Steele and Thinking Machines Inc. *Common Lisp the Language, 2nd Edition.* Digital Press, 1990.

[5] 湯浅 太一. *Scheme* **入門** *(***岩波コンピュータサイエンス***).* 岩波書店, 1991.

[6] Daniel P. Friedman, Mitchell Wand, and Christopher T. Haynes. *Essentials of Programming Languages, 2nd Edition.* MIT Press, 2001.

```
1  (defparameter *cont* nil)
2
3  (defun enqueue (v)
4    (setq *cont* (nconc *cont* (list v))))
5
6  (defun dequeue ()
7    (let ((head (car *cont*)))
8      (setq *cont* (cdr *cont*))
9      head))
10
11 (=defun yield ()
12   (let* ((cc (call/cc #'(lambda (cc) cc))))
13     (cond (cc (enqueue cc)
14               (let* ((next (dequeue)))
15                 (if (not (null next))
16                     (=funcall next nil)))))))
17
18 (=defun start ()
19   (let* ((head (dequeue)))
20     (if (not (null head))
21         (=funcall head nil))))
22
23 (=defun sample (x)
24   (cond ((null x) nil)
25         (t (format t "~A" (car x))
26            (yield)
27            (sample (cdr x)))))
28
29 (defparameter *sample* #'(lambda (x) (sample '(1 2 3 4 5 6 7 8 9 10))))
30 (setq *cont* (list *sample* *sample*))
31
32 top-level> (start)
33 1122334455667788991010
```

図 1.3: 並行処理 in Common Lisp

2 Nave; train☆Project!【なヴぇ；とれいん☆ぷろじぇくとぉ〜！】

電子システム工学課程 3 回 渡邉 翔一郎

2.1 電子工作

2.1.1 電子工作の魅力

私が始めてこの言葉を耳にしたのは小学生のときで、“電気で動くおもちゃを作る” くらいに思っていました。けれど実際は、電子工作はミニ四駆の改造から研究室の測定器の製作にまで及び、この言葉に親しむ人もそれこそ小学生からお年寄りまでいるわけです。それだけに私はこの「電子工作」という言葉には何かしら魅力というか、夢があると思います。

私はこの Lime の記事で電子工作の楽しさ、夢を伝えられたらいいなぁと思います。さぁ、電子工作に情熱を。MMQ[1]。

2.2 Nave;train ☆ Project!について

2.2.1 鉄道模型ってなぁに？

実際の鉄道車両を模型化したものを鉄道模型といいます。この鉄道模型では車両を線路の上で電気でコントロールして走らせたり、その線路の沿線で踏み切りや信号などを動作させて楽しむことができます。鉄道模型にも種類があり、主に縮尺で区別されています。私は 1/150（新幹線は 1/160）スケールの N ゲージで遊んでいます[2]。

毎年恒例！実写と模型の比較写真を載せておきます。ぜひ見比べてみてください。

図 2.1: 実車と模型車両の比較。実車は京都駅にて撮影した 700 系のぞみ。

[1] 萌え萌えきゅん

[2] N は Nine の意味で線路幅が 9mm になっています。

2.2.2 昆布鉄道

昨年度、私はこの松ヶ崎祭で昆布鉄道と題して鉄道模型に関連した電子工作の作品「マスターコントローラー」を展示しました。実車同様に 2 ハンドルで操作でき、またちょっとした “おまけ” で模型列車にカメラを内蔵して運転席からの光景を画面で見ながら運転できるようにしていました。

今年は要素を追加してより運転を楽しんでもらおう！と思い、プロジェクトの名前も Nave;train ☆ Project!(略してなヴぇ☆とれ) に変えて半田ごてを握ってがんばりました。二十歳を過ぎてもなお、純粋な子供心を忘れていないせいか自然とニヤニヤしてきます。ぐふふ。

2.3 製作品

2.3.1 半田付けしてつくったもの

私が今までに昆布鉄道に関連して製作したものを紹介します。

- マスコン回路
- レベルメータを使った電圧表示
- 最高速度超過警告
- 自動列車制御装置 (ATC)
- 自動列車停止装置 (ATS)
- 電流超過警告・遮断
- 信号機
- 踏み切り
- ポイント切り替え器
- トランジスタブザー
- 場内放送再生

一部、別の普通の用途に使ってしまいました (笑)。今回は「信号機」と「自動列車制御装置 (ATC)」、「自動列車停止装置 (ATS)」について述べたいと思います。

2.4 信号機の製作

2.4.1 鉄道信号機について

鉄道の信号機は車の信号と違っていくつか種類があります。車の信号はランプが 3 つですが、鉄道では 2〜5 つの信号機があります。ここ JR 京都線 (JR 西日本) では 4 灯式信号という、4 つのランプの信号機がよく見られる[3]一方、私の地元である JR 予讃線 (JR 四国) では 2 灯式が見られます。ランプの数に違いがあるのは進行速度に制限をかけるためです。2 灯式では “進んでよい” と “止まれ” の意味しか持ちませんが、4 灯式になると、黄色信号を交えて “45km/h で進行してよい” の意味を持たせます。

ここでは私が実際に製作した 5 灯式信号機を説明していきます。

[3] 原くん談

2.4.2 鉄道 5 灯式信号機

5 灯式信号機は青から始まって、列車が通過して赤 → 黄 ×2 → 青+黄 → 青とすすみます。これを再現するには LED を 5 つ並べて、順序よく点灯させなければなりません。次の図のように動作します。

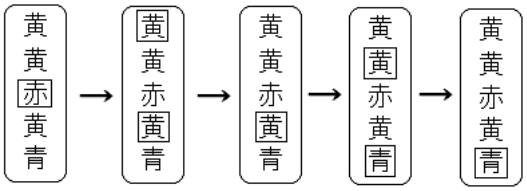


図 2.2: 5 灯式信号機の動作

2.4.3 動作原理

回路図は著作権の都合上ここに載せるわけにはいきませんが、動作原理を述べたいと思います。(回路図は参考文献 [1] を参照願います。)

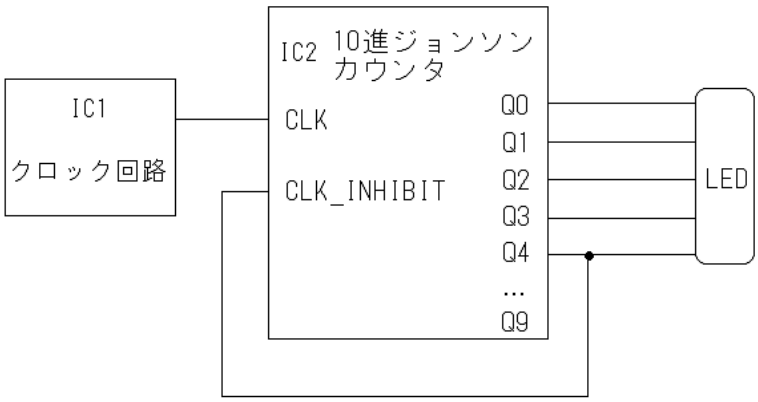


図 2.3: 5 灯式信号機の回路のブロック図

今回製作した回路では 2 つの IC を使用しています。図 2.3 に回路のブロック図を示します。IC1 はインバータ、IC2 は 10 進ジョンソンカウンタです。IC1 で信号機の点灯時間を測るタイマの役割を果たします。

次に IC2 はまず赤、次に黄、といったようにランプの光る順番を決めています。図 2.4 は IC2 のタイミング図です。順序良く出力しているのが分かると思います。

ここで、一巡して青に戻ったときにはそのまま青が点灯していないとまずいので、青信号の出力をバイナリカウンタの CLK_INHIBIT に入れてやります。すると CLK は青信号が点灯中に変化しないので青がずっとついたままになります。

そして列車が通過した際は全ての IC をリセットして初期状態に戻してやればまた赤 → 黄…と光り始めるのです。

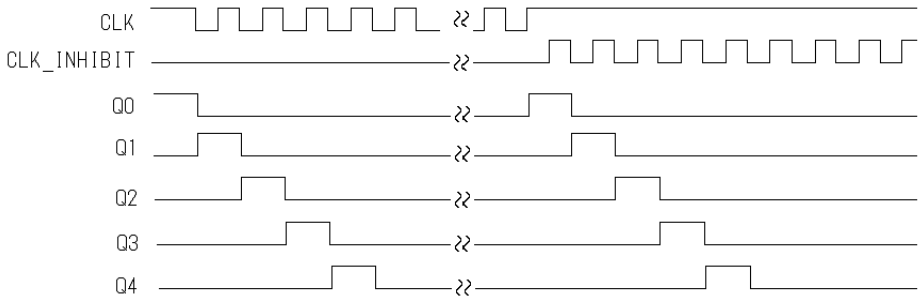


図 2.4: 10 進バイナリカウンタのタイミング図

2.5 車両検出回路

2.5.1 車両検出方法

踏み切りも信号も、車両を検知しないと始まりません。車両検知にはいろいろな方法が考えられますが、今回は光センサを利用することにしました。光センサだと模型自体に物理的接触も無く、影響を与えることはありません。

具体的には線路の真上に LED とフォトトランジスタを並べて、列車が通過した際に反射する光で検知します。真上から光を照射するので、室内の電灯の明かりなどを気にしなくてもすみます。

2.5.2 回路図

光センサによる車両検知の回路図を図 2.5 に示します。見た目のとおり、部品も少なく回路としては簡単です。順を追って説明していきます。

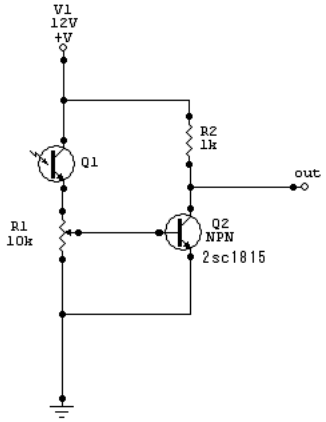


図 2.5: 光センサ回路

Q1 はフォトトランジスタです。これは受光部にあたる光の強さで電流の流れる量を制御する部品です。ここで通過した列車の光を検出します。列車が通過して反射光がこのフォトトランジスタに差し込むと R1 の半固定抵抗に電流が流れます。このとき、この抵抗には電位差が生じるので、Q2 のトランジスタのベースに電流が流れます。ただし、この半固定抵抗を Q1 の方にいっぱいに振ると、反射光がフォトトランジスタに差し込んだ際に Q2 のベース・エミッタ間電圧が限界値 5V を超えてしまうので気をつけなければなりません。ここで R2 に注目すると、反射光が差し込む前は Q2 に電流が流れていないため R2 に電流が流れません。そのため R2 に

は電位差が生じず、out には 12V(Hi) を指します。しかし、ここで Q2 のトランジスタがオンになると R2 に電流が流れるため out の電位が下がります (Lo)。こうして光センサとして動くわけです。

2.6 検出信号の伝達回路

2.6.1 信号を送る

2.4.3 の車両検出回路で「列車が来た！」というのがわかったら、その情報を処理部 (各信号機・踏切を動かす回路) に伝達しなくてはなりません。各検出回路から処理部まで導線でつなげば一番早いですが、検出数が多くなると線の数が膨大になってしまいます。そこで、複数の検出信号を 1 本の導線で伝達できるようにするために、D/A コンバータを使います。

2.6.2 D/A コンバータ

D/A コンバータとはデジタル信号をアナログ信号に変換する回路のことです。ちなみに、アナログ信号をデジタル信号に変換する回路を A/D コンバータといいます (後述)。各検出回路からは Lo もしくは Hi の信号が出ています。これらの信号をまとめて 1 つの電圧で表現してしまえば、1 本の導線で情報が伝達できます。たとえば、次の表 2.1 のように規則をつくってしまいます。

表 2.1: D/A コンバータ出力規則

検出器 1	検出器 2	出力電圧 [V]
Lo	Lo	0
Lo	Hi	3
Hi	Lo	6
Hi	Hi	9

いま、この例では検出器 1 が Hi のときは 6V 出力、検出器 2 が Hi のときは 3V 出力としています。こうすれば、いまどこの検出器を列車が通っているのかが電圧だけで、しかも 1 本の導線だけでわかります。この規則をより細かくしていけば、1 本の導線でさらに複数の検出器の信号を伝達することが可能です。独立した検出器の信号 (デジタル信号) がひとつの電圧 (アナログ信号) で表現されました。

2.6.3 R-2R ラダー回路

2.6.2 のような D/A 変換を実現するために R-2R ラダー回路を利用します。次の図 2.6 を見てください。いま、S1 と S2 がともに GND につながっています。抵抗には電流が流れないので out は 0V が出力されます。

次に、S1 を上側に倒すと抵抗に電流が流れます。S2 は依然 GND につながっているので、R2、R3、R4 の合成抵抗は 2kΩ となり、結果 out には 12V の半分の 6V が出力されます。

次に、S1 を下に倒して S2 を上に倒すと、out 両端の抵抗比が 3:1 になり、out には 3V が出力されます。

最後に S1 と S2 両方を上に倒すと 9V が出力されます。

この S1、S2 の ON/OFF を検出器の Hi/Lo に割り当ててやれば、2 つの独立した信号がひとつの電圧になって伝わります。ちなみに、本来は out の部分にアンプをつけますが、今回は無しでもうまく動作するのではずしています。

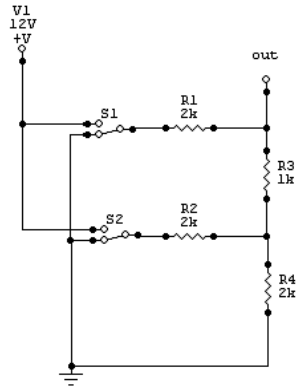


図 2.6: R-2 R ラダー回路

2.7 車両検出 & D/A コンバータ回路

2.7.1 設計した回路図

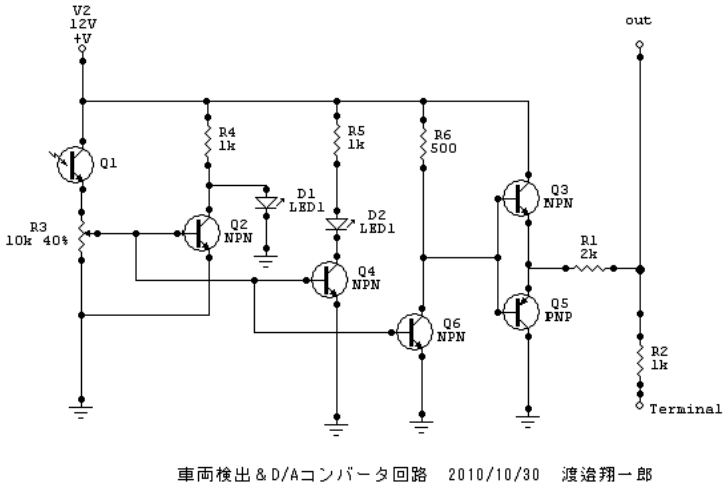


図 2.7: 車両検出& D/A コンバータ回路

2.6.2 と 2.6.3 を踏まえて私の設計した回路図を図 2.7 に示します。2 つの回路を組み合わせた回路になっていると思います。先程と同じく、半固定抵抗の調整にはベース・エミッタ間電圧が限界値 5V を超えないよう気をつけなければなりません。Q3、Q5、Q6 が Hi/Lo を切り替えるトランジスタです。トランジスタ Q6 のベースに電流が流れると Hi、電流が流れていないときは Lo になります。R-2R ラダー回路のスイッチングには Hi と Lo の中間値があっては困るので、コンプリメンタリの Q3 と Q5 を配置しました。また、この回路にはちょっとしたおまけをつけています。

列車が検出器を通過中の時は赤 (D1)、通過していないときは青 (D2) が点灯するようにしています。

2.7.2 赤/青 LED の点灯

赤/青を点灯させるために Q2 と Q4 を付け足しました。列車が通過しているとき Q2 と Q4 にはベース電流が流れています。このとき赤色 LED(D2) には電流が流れる一方、青色 LED(D1) には両端に電位差が生じない

ため光りません。

逆に列車が通過していないときはベース電流が流れないため赤色 LED は光りませんが、青色 LED の両端には電位差が生じるために光ります。

2.8 自動列車制御装置 (ATC)・停止装置 (ATS)

2.8.1 安全にカーブを曲がるために

列車がカーブに差しかかるとき、最高速度で突入すると脱線する恐れがあります。このように列車が特定区間を走行する際、列車の速度がちゃんと指定した速度以下になっているかを確認しなければなりません。次のような手順で確認・制御を取ります。

1. 車両検出器をカーブ手前に設置
2. 列車が車両検出器を通過
3. 列車に供給している電圧と指定した電圧を比較
4. 指定した電圧を超えている場合、警報機を鳴らす、列車に供給する電圧を指定した電圧まで下げる

4 のとき、列車を指定速度以下に制御する装置を自動列車制御装置 (ATC) といいます。

2.8.2 動作原理

上記 3 から説明します。列車に供給している電圧と指定した電圧を比較するのにコンパレータを使います。このとき、コンパレータの基準電位に指定した電圧をあてます。列車に供給している電圧が指定した電圧 (基準電位) を超えると、コンパレータの出力が Lo から Hi になるので、これをきっかけに警報機のスイッチをオン、そしてブレーキをかけます。このブレーキにより列車に供給される電圧は下がりますが、これは指定した電圧 (基準電位) を下回るまで働くことになります。

2.8.3 列車を停止させる

赤信号など列車が停止しなければならない状況にもかかわらず、止まらず走行を続けようとしている場合、運転者の操作に関係なく列車を停止させなければなりません。このとき、上記と同じ原理を使って指定の電圧を 0 にしておけば列車を停止させることができます。列車を自動的に停車させる装置のことを自動列車停止装置 (ATS) といいます。

2.8.4 緊急停車信号機

これは回路図には書いていませんが、紹介だけしておきます。

列車を運転する側に何ら問題がなくても、走行線路の先に障害物があった場合や、電気ショートがあった場合は走行している列車を止めなければなりません。各信号機を赤色に切り替えるのはもちろん、各検出器にも緊急停車を指示する緊急停車信号機を設置して、停車を促します。

この場合、制御部から各信号機と検出器の緊急停車信号機に停止信号を送らなければなりません。しかし信号線を増やしては元も子もありません。そこで、停止信号は制御部から信号線に 12V の電圧を掛けて緊急停車信号機を作動させます。

2.8.5 緊急停車信号機の動作原理

緊急停車信号機の動作は 5 つの LED が右回りに回転するように点灯します。これを実現するために 2.4.3 で使った 10 ビットバイナリカウンタとインバータを使います。信号機と同様に、Q0〜Q5 を順番に点灯させればいいわけです。ちなみにこの回路については信号機、踏み切りと同様に参考文献 [2] に詳しく載っています。興味のある方はプリント基板も付属で付いていますので購入を検討されてはいかがでしょうか。

2.9 おわりに

昨年もそうだったのですが、この Lime の記事を書いている時点では設計した回路全ての動作テストができていないため、なヴぇ☆とれのすべてを書くことができません。ですが、この文章を読んでいただいたころには、きっとすべてが完成しているはずです。ぜひごらんください。

ちなみにちょうど同じころ、JR 西日本が新型車両を導入します。先日先着 1000 名限定の見学会に参加してきましたが、いい電車でした。

最近、私は部の仲間に「紳士」と呼ばれています。本当にそのままの意味で呼ばれているかどうか知りませんが、仲間と仲良くいられることはうれしく思います。これからもここ京都で和気あいあいと半田ごてを握りながら、その「紳士」を磨きあげていきたいを思います。たとえ変態と言われても。

2.10 参考文献

[1] エレキジャック No6 **CQ 出版社** p62〜83 智田聡丞

[2] トランジスタ技術 Special No84 **CQ 出版社**

3　特盛！LED マトリクス

電子システム工学課程 3 回 鴛渕 真理

3.1　はじめに

「LED マトリクス」は、電光掲示板などに利用されている LED の集合体です。LED マトリクスは、秋月電子通商で購入できる 2 色 LED マトリクスを用います（図 3.1）。この商品は、縦横に 16 個 LED が並べられており、さらに赤と緑の 2 色表示を行う事ができます。しかしただ光らせるだけではシンプルすぎるので、今回は 8 枚の LED マトリクスを使って、4096 個の LED を一気に制御することを目指します。

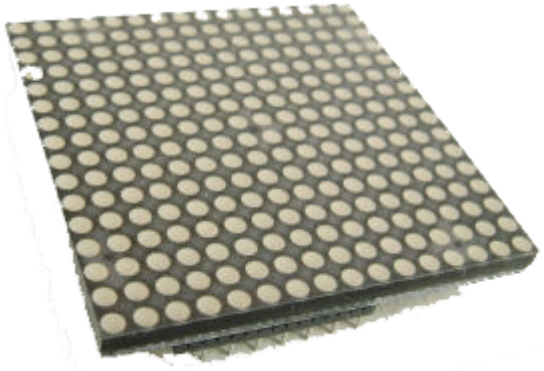


図 3.1: 2 色 LED マトリクス

3.2　仕様

製作にあたっての仕様を以下のように設定します。

・安価であること

社会でも部活でも外す事のできない課題です。今回は最後に市販品との比較内容を記載します。

・1 つのマイクロコンピュータ（以下マイコン）で全ての LED を制御できること

複数個のマイコンを用いて通信を行えば、多くの数の LED を制御することができます。しかし、この方法を取るとコストが高くなるため、今回は 1 つのマイコンだけで制御を行ないます。

・一定の明るさを保てること

今回は、LED1 つに対して信号線を 1 本使う事をせず、ダイナミック点灯と呼ばれる点灯方式を採用します。この方式は光の残像を利用した表示方法で、一列ごとに点灯を行います。しかし、この回路を直感的に組むと、列の中で多く LED を点灯させるほど 1 個あたりの電流が減り、暗くなってしまいます。これを回避するために、専用の IC ををを利用し、点灯する LED の数に関わらずそれぞれに一定の電流を供給します。

3.3 回路の構成

LED の点滅を制御するマイコンですが、今回は制御対象の数が多ので、LED に直接端子を繋ぐとするとマイコンの端子が足りません。そこで、マトリクスから伸びている端子を減らすような回路を製作しました。製作した回路は、マイコンと受信部で構成されており、1 つのマイコンから数個の受信部へ、2 つのシリアルデータ（時間的に変化するデータ）を送信します。各受信部では、受信したシリアルデータを読み取り、数十個の端子に出力を行います。

3.3.1 マイコンボード (Aki-7125)

Aki-7125 マイコンボードは、汎用出力に使用できる端子が 35 本あります。ここで、本当に 8 個のマトリクスボードを制御できるか検証するために、受信部へデータを送信する際に用いる端子の数を考えます。まず、すべての LED マトリクスに共通に接続する端子は、1. 列データ送信クロック用 2. 行データ送信用クロック用 3. データ更新用 の 3 本になります。次に、16 個それぞれに使用する端子は、1. 列データ用 2. 行データ用 の 2 つになります。これより、1 つのマイコンボードで制御できる LED マトリクスの枚数は、$(35\text{-}3)/2 = 16$ 枚となります。今回の目標は 8 枚なので、十分に制御できることがわかります。

3.3.2 受信部

LED マトリクス (LT-5016M1)

縦 16 ×横 16 × 2 色（赤と緑）、アノードコモンの LED マトリクスで、命令を受け取る端子が 48 本（行 16 本、列 32 本）あります。1 枚で 16 × 16 × 2=512 個の LED が載っていますのでれを 8 枚使用し、LED の総数は 4096 個になります。

シフトレジスタ (TC74HC164AP)

1 本の信号線に時間的に変化する信号を入力すると、8 個の出力から信号を出す IC です。シリアル-パラレル変換器 (直並列変換器) とも呼ばれます。LED マトリクスの縦の列を、１列ずつ点灯させるために使用します。

ソースドライバ (TD62783)

ソースドライバは、電流を吐き出す（出力する）役目を持ちます。入力信号があれば出力を行う、というシンプルな IC です。前述したシフトレジスタのパラレル出力をこのソースドライバの入力とし、マイコンに定格以上の電流が流れる事を防ぎます。

LED ドライバ (TB62706)

250 円と値ははりますが、16bit シフトレジスタ・ラッチ・定電流回路が全てパッケージされている優れものです。LED 駆動用と銘打ってあり、5[mA]〜90[mA] の電流を吸い込む事ができます。また、1 つの外部抵抗をつけることによって 16 個の端子の出力電流を一括して調整する事ができます。今回は、この外部用抵抗に可変抵抗を接続し、10[mA]〜90[mA] まで出力電流を調整できるようにします。

受信回路の全体図

図 3.2 のように各素子を接続します。前述した素子を配置しています。図中央にある 50 の端子がある部品が、LED マトリクスに繋がるコネクタです。また、図中央上側にある 10 の端子がある部品は、マイコンに繋がっており、各種命令が送られてくる部分となります。16 ある行の端子にはシフトレジスタとソースドライバを接続し（図左側）、つねに 16 行のうちの 1 行だけが点滅するように制御します。32 ある列の端子には LED ドライバを接続し（図右側）、点灯させたいデータを LED マトリクスに送ります。

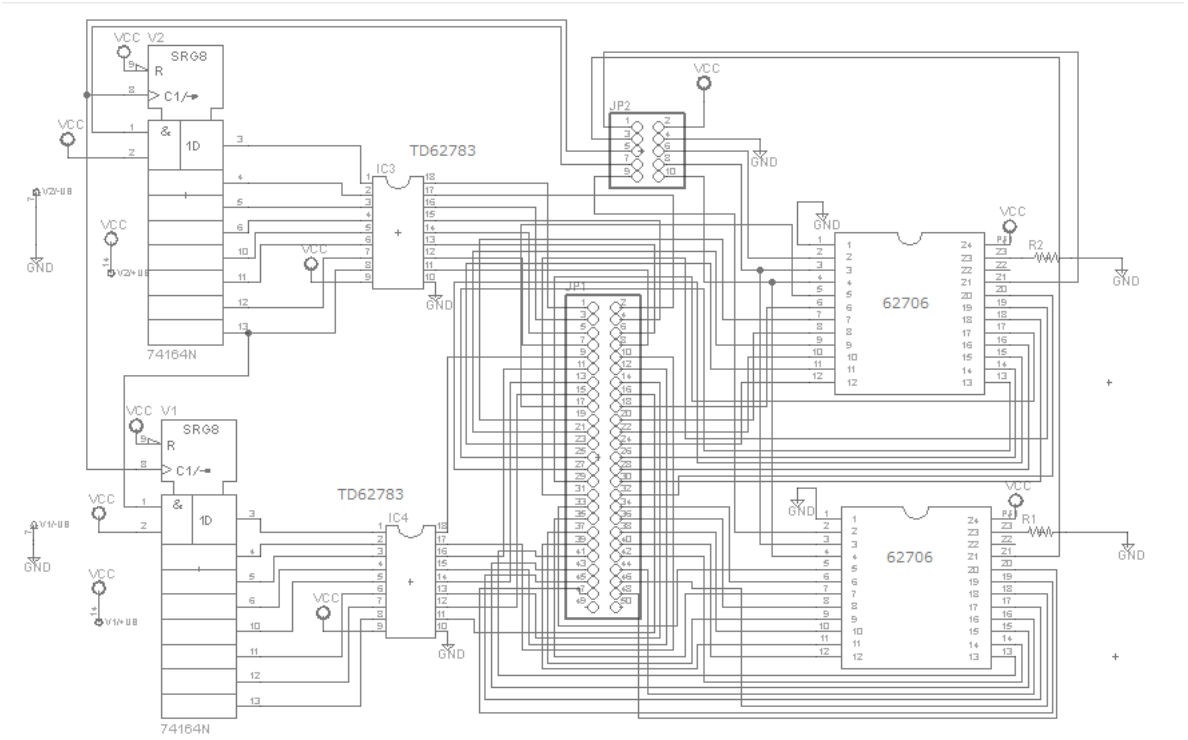


図 3.2: 受信回路全体図

3.4 プログラム

テストプログラムとして、2 色の LED で格子柄を表示するプログラムを作成しました。

3.4.1 プログラムの流れ

display[32] という整数型配列の中身を表示させるプログラムを作ります。このプログラムでは一定時間が経つと INT_MTU2_1_TGIA1 関数にジャンプし、display[32] のデータにしたがって LED マトリクスの端子の電圧を制御します。

また、INT_MTU2_1_TGIA1 関数の中の処理は、

1. 32 個の列データを順番に格納
2. 光らせる行を 1 行ずらす

という流れになっています。

3.4.2 プログラム本文

```
#include "iodefine.h"                    //レジスタの宣言
#include <machine.h>                     //CPU 処理の記述

#define IN164  PA.DRL.BIT.B0             //端子の再宣言
#define IN706  PA.DRL.BIT.B1
#define LT706  PA.DRL.BIT.B2
#define EN7061 PA.DRL.BIT.B4
#define EN7062 PA.DRL.BIT.B5
#define CLR164 PA.DRL.BIT.B8
#define CLK164 PE.DRL.BIT.B0

void initIO(void);                       //入出力端子の設定
void initMTU21(void);                    //PWM による時間制御のための設定

volatile long int cnt=0,col=0,row=0;
volatile unsigned long int data[16]={0xaaaaaaaa,0x55555555,0xaaaaaaaa,0x55555555,
                                     0xaaaaaaaa,0x55555555,0xaaaaaaaa,0x55555555,
                                     0xaaaaaaaa,0x55555555,0xaaaaaaaa,0x55555555,
                                     0xaaaaaaaa,0x55555555,0xaaaaaaaa,0x55555555
                                     };
                                        //格子柄を表す 16*32 のデータ

void main(void)
{

    STB.CR4.BIT._MTU2=0;                 //省電力設定を解除
    INTC.IPRD.BIT._MTU21G = 15;          //割り込み優先度

    initIO();
    CLR164=1;
    initMTU21();                         //MTU20 PWM 出力
    set_imask(0);

    while(1){
    CLR164=1;
    }
}

void initIO(void){

    PFC.PECRL2.BIT.PE4MD = 1;            //TIOCA1
    PFC.PEIORL.WORD = 0xFFFF;            //PE=出力
    PFC.PAIORL.WORD = 0xFFFF;            //PE=出力
}
```

```
void initMTU21(void){
    MTU2.TSTR.BIT.CST1=0;

    MTU21.TCR.BIT.CCLR=1;               //TGRA クリア
    MTU21.TCR.BIT.TPSC=2;               // 20000000/16 Hz でカウント 31250
    MTU21.TMDR.BIT.MD=2;                //PWM MODE 1
    MTU21.TIOR.BIT.IOA=2;               // 01
    MTU21.TIOR.BIT.IOB=1;               // 00
    MTU21.TGRA=50;                      //DEFAULT 100Hz
    MTU21.TGRB=25;
    MTU21.TIER.BIT.TGIEA = 1;           //コンペアマッチ　割り込み on

    MTU2.TSTR.BIT.CST1=1;
}


void INT_MTU2_1_TGIA1(void){
    int i;

    IN706 = (data[row]>>col)&1;             //行データを代入

    col++;                                  //列をずらす
    if(col>=32){                            //32 列分のデータを格納したら
        col=0;                              //列を示す変数を初期化
        EN7061=1;
        EN7062=1;
        LT706=1;                            //データをラッチにセットする
        EN7061=0;

        EN7062=0;
        if(row == 0)                        //0 行目に到達したら
            IN164 = 1;                      //行データ端子を立ち上げる
        row++;
        if(row>=16)                         //16 行分データを格納したら
            row=0;                          //列を初期化
        CLK164= 1;                          //行のクロック端子を立ち上げる

    }else{                                  //列データ格納中だったら
        LT706 =  0;                         //ラッチをセットしない。
        CLK164 = 0;                         //行のクロック端子を立ち下げる
        IN164 = 0;                          //行データ端子を立ち下げる
    }

    MTU21.TSR.BIT.TGFA = 0;                 //割込フラグクリア
}
```

3.5 動作確認

プログラムを実行すると、無事、格子状に LED が光りました。図 3.3 に実際の写真を示します。図の右側上が受信回路の配線面（裏には図 3.2 のとおりに IC が載っています）図の下側にあるのがマイコンです。

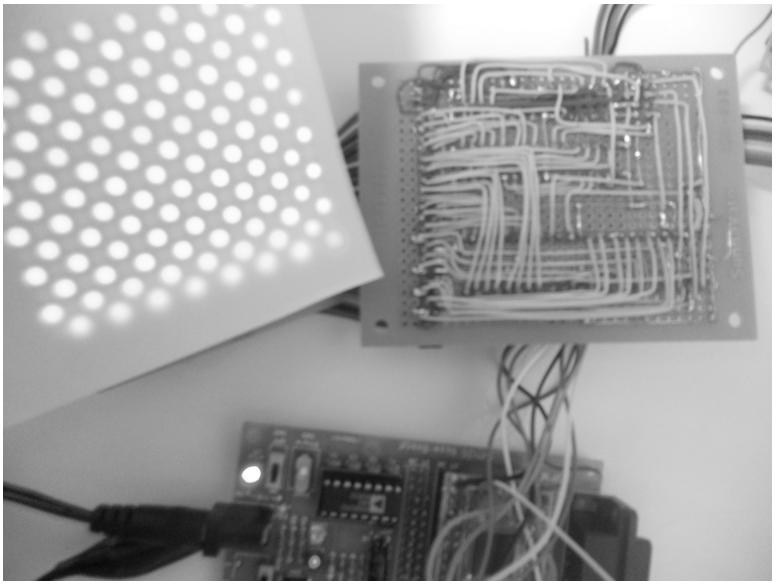


図 3.3: 動作確認写真

3.6 おわりに

今回の製作で苦労したのは、おわかりかもしれませんが、回路製作の配線部分です。次の製作からは、ユニバーサル基盤ではなく基盤加工機を用いて、配線作業を減らそうと強く決意しました。こんな配線（図 3.3）は二度としたくありません。今回の課題を達成するには、あと 7 枚、受信部の回路を作る必要がありますが、残り 7 枚は基板加工機を導入し、機械に配線を彫ってもらう事にします。今後も鋭意製作に励みたいと思います。

4　ルイージと対戦できる、五目並べを作ってみた。

情報工学課程 1 回 葛西 響子

4.1　はじめに

私は、ルイージが大好きで、ルイージが中心となるゲームを作ってみたいと昔から思っていました。でも、プログラムに関する知識が皆無の状態で大学に入学したので、どのジャンルのゲームが作りやすいのか全く分かりませんでした。初めは夏休み中にオセロを作ろうと思ったのですが、自分の実力不足のために、途中で行き詰ってしまいました。そこで、プログラムの基本といわれる五目並べを少しアレンジしたもの（ルイージ流五目並べ）を作ってみました。

4.2　プログラムの流れ

ルイージ流五目並べがどのように作られたかを抜粋して書いていきたいと思います。ルイージ流五目並べは、ＤＸライブラリ [1] とＣ言語 [2] を使って作りました。

4.2.1　マウス入力の受け付け方

ルイージ流五目並べは、マウスをクリックすることで石の置く位置を決定できます。その部分のソースは以下のようになります。

```
// マウスを表示状態にする
SetMouseDispFlag( TRUE );
int MouseInput;
// マウスの入力を待つ
MouseInput = GetMouseInput();
while( ( MouseInput & MOUSE_INPUT_LEFT ) == 0 ){
  // メッセージ処理
  if( ProcessMessage() == -1 ){
    break; //エラーが起きたらループから抜ける
  }
  // マウスの入力を得る
  MouseInput = GetMouseInput();
}
if( ( GetMouseInput() & MOUSE_INPUT_LEFT ) != 0 ){
  // 押されている
  // マウスの位置を取得
  GetMousePoint( &x, &y );
}
```

SetMouseDispFlag 関数でマウスカーソルの表示の有無を設定します。( ) の中に TRUE と書くとマウスカーソルが表示され、 FALSE と書くと表示されません。
GetMousePoint 関数でマウスカーソルの画面上での座標を int 型変数 に格納し、マウスカーソルの位置を取得します。ボタンを押した瞬間マウスがどこにいたかを得るのに使用します。
GetMouseInput 関数でマウスの入力状態値を得ます。戻り値を MOUSE_INPUT_LEFT とＡＮＤ演算し、結果が０でなければマウス左ボタンが押されていることになります。

4.2.2 ルイージのセリフを表示する（私が一番やりたかったことです☆）

ルイージ流五目並べの醍醐味（……と私が勝手に設定しています)、ルイージのセリフ表示について説明します。この部分のソースは、以下のようになります。

```
int Message;
Message = GetRand(11);
if(Message==0){
  DrawString(0, 430, "僕の置く石は、あえてピンク色だよ♪", White);
}
else if(Message==1){
  DrawString(0, 430, "兄さんだったら、きっとここに置くはずだ！", White);
}
～～～～～～～～～～～～～～～～～～省略～～～～～～～～～～～～～～～～
else if(Message==11){
  DrawString(0, 430, "帽子は手洗いに限ると僕は思うんだ♪", White);
}
```

GetRand 関数で０～指定した値までの乱数を得ます。その値を Message に代入し、Message の値によって、表示するルイージのセリフを変えます。DrawString 関数で文字列を描画できるので、これを使ってルイージのセリフを表示します。

4.2.3 ルイージの思考ルーチン

ルイージがどのように自動で石を置いているのかについて説明します。
この部分のソースを以下に示します。

```
//no は盤上に石を置く位置を記憶する配列
//X は x をキリのよい数字に書き換えたものを記憶する変数
//Y は y をキリのよい数字に書き換えたものを記憶する変数
//judge は勝ち負けを判定する関数
goal:
int Luigi;
Luigi = GetRand(7);
if(Luigi==0){
  if(no[X-1][Y-1]==0){
    DrawCircle( x-50, y-50, 15, Pink, TRUE);
    no[X-1][Y-1]=10;
    judge(no);
```

```
    }else{
       goto goal;
     }
  }
  else if(Luigi==1){
    if(no[X][Y-1]==0){
      DrawCircle( x, y-50, 15, Pink, TRUE);
      no[X][Y-1]=10;
      judge(no);
    }else{
       goto goal;
     }
  }
〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜省略〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜〜
  else if(Luigi==7){
    if(no[X+1][Y+1]==0){
      DrawCircle( x+50, y+50, 15, Pink, TRUE);
      no[X+1][Y+1]=10;
      judge(no);
    }else{
       goto goal;
     }
  }
```

プレイヤーが置いた石の周りのどこか一か所にランダムで置くという思考ルーチンになっています。周りが全て埋まっていた場合は、フリーズします。本当は、プレイヤーの石が３つ並んだらプレイヤーの勝利を阻止するというプログラムをこのソースの前に書きたかったのですが、うまく実行されなかったので、断念しました。……つまり、この状態ではルイージは非常に弱いです。

4.2.4 勝ち負けの判定

石が５つ並んだら、プレイヤーが勝ったか、ルイージが勝ったかを判定します。この部分のソースは、以下のようになります。

```
  //プレイヤーの判定
  //横
  for(i=0;i<=7;i++){
    for(j=0;j<=3;j++){
      if(no[i][j]==1&&no[i][j+1]==1&&no[i][j+2]==1&&no[i][j+3]==1&&no[i][j+4]==1){
        DrawBox   (  1, 421, 399, 469, Black, TRUE);
        DrawString(0, 430, "おめでとう！君の勝ちだよ☆", White);
        WaitKey();
        exit(0);
      }
    }
  }
```

```
  //縦
～～～～～～～～～～～～～～～～～～～～省略～～～～～～～～～～～～～～～～～～～～
  //右斜め上から左斜め下
  if(no[0][4]==1&&no[1][3]==1&&no[2][2]==1&&no[3][1]==1&&no[4][0]==1){
    DrawBox   (  1 ,  421 , 399 , 469 , Black , TRUE) ;
    DrawString(0,430,"おめでとう！君の勝ちだよ☆", White);
    WaitKey() ;
    exit(0);
  }
  if(no[3][7]==1&&no[4][6]==1&&no[5][5]==1&&no[6][4]==1&&no[7][3]==1){
    DrawBox   (  1 ,  421 , 399 , 469 , Black , TRUE) ;
    DrawString(0,430,"おめでとう！君の勝ちだよ☆", White);
    WaitKey() ;
    exit(0);
  }
～～～～～～～～～～～～～～～～～～～～省略～～～～～～～～～～～～～～～～～～～～
  if(no[2][5]==1&&no[3][4]==1&&no[4][3]==1&&no[5][2]==1&&no[6][1]==1){
    DrawBox   (  1 ,  421 , 399 , 469 , Black , TRUE) ;
    DrawString(0,430,"おめでとう！君の勝ちだよ☆", White);
    WaitKey() ;
    exit(0);
  }
  if(no[3][4]==1&&no[4][3]==1&&no[5][2]==1&&no[6][1]==1&&no[7][0]==1){
    DrawBox   (  1 ,  421 , 399 , 469 , Black , TRUE) ;
    DrawString(0,430,"おめでとう！君の勝ちだよ☆", White);
    WaitKey() ;
    exit(0);
  }

  //左斜め上から右斜め下
～～～～～～～～～～～～～～～～～～～～省略～～～～～～～～～～～～～～～～～～～～
  //ルイージの判定
～プレイヤーの判定のソースの、「==1」を「==10」にして、「おめでとう！君の勝ちだよ☆」を「ラッキー、僕の勝ちだ～♪」にしたもの～
```

縦、横、斜めにそれぞれ５つ並んだ状態か否かの判定を１マスずつずらして行っていき、５つ並んでいたら、プレイヤーの勝ちかルイージの勝ちかに応じて、ルイージのセリフが表示されます。縦と横の判定は for ループで簡潔に表せたのですが、斜めの判定が簡潔に表せず、ソースが冗長になってしまいました。そのため、右斜め上から左斜め下の判定の一部だけ載せます。なお、縦の判定は、横の判定とほとんど同じなので省略します。

4.3　ゲーム動作画面

このようなゲーム画面になります。

4.4 今後の課題

板の一番外側の部分に石を置くと、たまに誤動作するところや、プレイヤーが置いた石の周りがすべて埋まっていた時にフリーズするところ、ルイージの思考ルーチンが弱すぎるところを直していきたいと思います。あと、できれば斜めの判定部分のソースをもっと簡潔にしたいです。

4.5 おわりに

ゲームのプログラミングは、思ったよりも難しいことが身にしみてわかりました。でも、プログラムに関する知識を少しずつ身につけていけば基本的なゲームをある程度作れることがわかりました。コンピュータ部で行われたＣ言語勉強会で学んだＣ言語が、ゲーム制作によって、かなり自分のものになってきたと感じています。今後も頑張って勉強して、五目並べもどきをパワーアップさせたり、もう一歩進んだゲームを作ったりしたいと思います。

4.6 参考文献

[1] Remical Soft
Ｃ言語～ゲームプログラミングの館～[ＤＸライブラリ]
< http://dixq.net/g/index.html >

[2] TOMOJI
初心者のためのポイント学習Ｃ言語
< http://www9.plala.or.jp/sgwr-t/index.html >

5　C言語でSTGをつくってみた

情報工学課程 1 回 山田 晃久

5.1　はじめに

コンピュータ部の C 言語勉強会で C 言語の基礎を学び, 何か自分でゲームを作ってみたいと思いました。しかし, 何から始めて良いのかもわからず先輩に尋ねてみたところ、DX ライブラリというライブラリがあることを教えてもらったので、この DX ライブラリを使ってC言語でシューティングゲーム (STG) を作ってみることにしました。

5.2　DX ライブラリとは

DX ライブラリとは山田巧氏が作成した「DirectX や Windows 関連のプログラムを使い易くまとめた形で利用できるようにした C++言語用のゲームライブラリ」です。今回は C++言語用のライブラリを使いますが、ゲーム自体は C 言語のスタイルで書いています。

5.3　プログラミング

キャラクターの動かし方などのゲームの基本的な処理を紹介したいと思います。

5.3.1　基本

メイン関数の中でまず初期化処理をして、while 文でメインループを作ります。そのメインループの中でゲームのほぼ全ての処理を行います。ゲームでキャラクターや弾などを移動させるには、それぞれの画像の位置を少しずつずらしながら表示するという処理を繰り返し行います。

5.3.2　自機の移動

自機とは自分が操作するキャラクターのことです。最初に自機の位置や画像などのデータを入れておくための構造体を作ります。自機を表示するには LoadGraph 関数を使い用意した画像ファイルをメモリに読み込みます。そして、メモリに読み込んだ画像を DrawGraph 関数を使い表示します。次に key という変数を作り、GetJoypadInputState 関数を使ってジョイパッドとキーボードの入力状態を得ます。そして、上ボタンが押されていれば自機の座標を上にずらします。このような処理を上下左右の４方向についてさせます。また、自機が画面外に出てしまわないように移動可能な範囲も設定しておきます。自機の x 座標と y 座標を jiki.x, jiki.y とすると自機の移動のプログラムは次のようになります。

```
int WINAPI WinMain(HINSTANCE hI, HINSTANCE hp, LPSTR lpC, int nC)
{
  ChangeWindowMode(TRUE);                //ウィンドウモードで起動
```

```
  if(DxLib_Init() == -1)return(-1);    //ＤＸライブラリ初期化
  SetDrawScreen(DX_SCREEN_BACK);
  struct charadata{
    int x;
    int y;
    int cx;
    int cy;
    int r;
    int image;
  };
  struct charadata jiki;                  //自機の構造体
  jiki.image = LoadGraph("image.bmp");//自機の画像読み込み
  while(ProcessMessage() == 0){
    ClsDrawScreen();
    //自機の移動
    int key = GetJoypadInputState(DX_INPUT_KEY_PAD1);
    if(key & PAD_INPUT_UP && jiki.y > 20){
        jiki.y -= 5;
      }
    if(key & PAD_INPUT_DOWN && jiki.y < 465){
        jiki.y += 5;
      }
    if(key & PAD_INPUT_LEFT && jiki.x > 30){
        jiki.x -= 5;
      }
    if(key & PAD_INPUT_RIGHT && jiki.x < 415){
        jiki.y += 5;
      }
    DrawGraph(jiki.x, jiki.y, jiki.image, TRUE);
    ScreenFlip();
  }
  DxLib_End();
  return 0;
}
```

5.3.3 自機の弾

次に自機の弾ですが、とりあえずは直線軌道のみです。直線軌道の弾はショットキーが入力された時に自機に描画し、それを前に移動させていくだけです。また、複数の弾を処理するには for 文を使います。しかし、このままだとショットキーを押しっぱなしにしていると弾が重なって描画されるので、reload という変数を作り初期値を 0 にします。弾を 1 発発射するごとに reload をある値にしてから 0 になるまで徐々に減らしていきます。そして、ショットキーが入力されていて、かつ reload の値が 0 の時にだけ弾を発射するようにします。実行結果は図 1 のようになります。▲が自機で●が弾です。

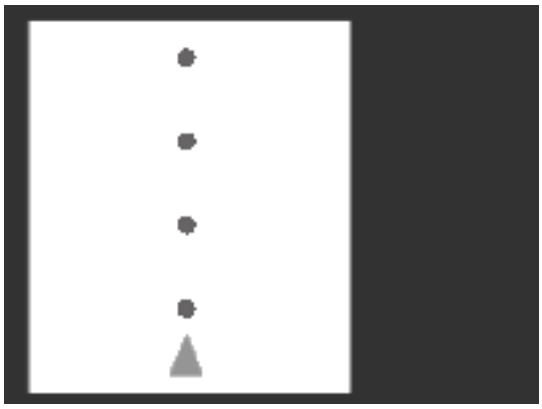


図 5.1: 自機の弾

5.3.4 当たり判定

当たり判定とは、自機や敵のショットが対象に命中したとみなされる範囲のことです。まず自機や敵、弾のそれぞれの構造体にそれぞれの中心の座標、および当たり判定の半径を作っておきます。そして、お互いの中心の距離がそれぞれの当たり判定の半径の合計より小さくなったときに命中するようにします。例えば、自機の中心座標と当たり判定の半径を jiki.cx, jiki.cy, jiki.r、敵の弾の中心座標と当たり判定の半径を tama.cx, tama.cy, tama.r として、命中すれば１、それ以外の場合は０を返す hit 関数を作ります。その関数のプログラムは次のようになります。

```
int hit(void)
{
  int x, y, r;
  x = jiki.cx - tama.cx;
  y = jiki.cy - tama.cy;
  r = jiki.r + tama.r;
  if(x*x + y*y <= r*r){
      return 1;}
  return 0;
}
```

5.3.5 敵の弾

敵には自機と同じ構造体を使います。基本的なことは自機とあまり変わりません。とりあえずは敵の弾も直線軌道のみです。下の図は敵が弾を発射している図です。▲が自機で■が敵、●が敵の弾です。

今はまだまっすぐにしか撃ってきませんが少し工夫すると、３方向に撃たせることや自機を狙って撃たせることもできます。それはこれから実装していこうと思います。

5.4 最後に

今回は本当に基礎的な部分しか出来ていませんが、今後は複数の敵を動かしたり、敵の弾の動きの種類を増やしたりして、ステージも作ってみたいです。あと、動作させるだけではなく効率的なプログラムを書けるように頑張ります。

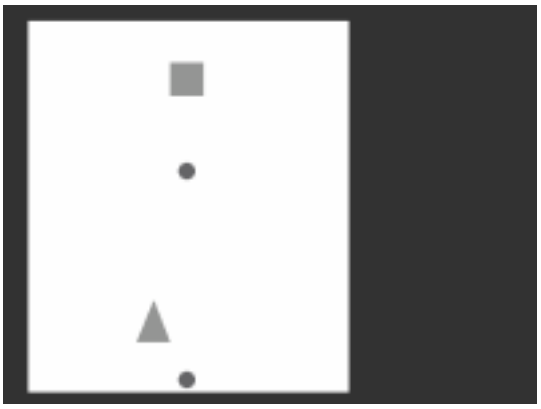


図 5.2: 敵の弾

5.5 参考文献

[1] 山田 巧
「ＤＸライブラリ置き場」
＜http://homepage2.nifty.com/natupaji/DxLib/＞

[2] Dixq
「龍神録プログラミングの館」
＜http://dixq.net/rp/＞

6 Rubyで数独を解こう!!

情報工学課程 1 回 山田 基晴

6.1 はじめに

入学してから、およそ半年の間 C 言語について、いろいろなことを自分なりに勉強してきました。C 言語を触るのは確かに楽しいですが、他のプログラミング言語にも触れてみたいという欲が出てきたので、まつもとゆきひろさんにより作られた”Ruby”[1] というプログラミング言語を勉強することに決めました。プログラミング言語の勉強をするときは、何かしらの物を作って勉強した方が記憶に留まり易いことを、この半年で気づいたので、数独を解くプログラムを Ruby で作ろうと思い立ちました。何故数独かというと……単純に私が好きだから。

6.2 数独の基本ルール

- 3 × 3 のブロックに区切られた 9 × 9 の正方形のマス[1]があり、空いているマスに 1〜9 のいずれかの数字を入れる。
- ただし、縦・横の各列及び、太線で囲まれた 3 × 3 のブロック内に同じ数字が複数入ってはいけない。

6.2.1 今回の問題

数学者が考えた”世界で最も難しい数独” [2] というものを解くことにします。

		5	3					
8							2	
	7			1		5		
4					5	3		
	1			7				6
		3	2				8	
	6		5					9
		4					3	
					9	7		

そして、この問題は、次のようにソースファイルとは別に記述しておきます。

[1]応用ルールで、16 × 16 や 25 × 25 などがありますが、今回は考えません。
[2]IT medeia Games
解けたら天才!フィンランドの数学者が作った「世界一難しい数独」
http://gamez.itmedia.co.jp/games/articles/1008/23/news093.html

```
$ cat questio.txt
**53*****
8******2*
*7**1*5**
4****53**
*1**7***6
**32***8*
*6*5****9
**4****3*
*****97**
```

6.3 アルゴリズム

問題を解くに当たって用いたアルゴリズムは、バックトラック法 [3] というものです。バックトラック法とは、系統的に探査を進めて、矛盾が生じるとそれ以降の処理はしないで、一つ前の状態に戻して、次の試行をする、というものです。

今回の数独の話に置き換えると以下のようになります。マスをチェックして、1〜9 の数字がすでに入っていれば、次のマスをチェックします。もし空マスならば、同列・行・ブロックで使用されていない数字を当てはめて、次のマスに移動します。そして、当てはめれる数字が無くなり、矛盾が生じたら、一つ前の状態に戻して、再度別の数字を当てはめていきます。

6.4 Ruby による実装

制作した数独を解くプログラムをコード片を示しながら説明していきたいと思います。

6.4.1 問題を配列に格納する

```
mas=Array.new
file_name=ARGV.shift                    #ファイル名を取得
File.foreach(file_name) do |line|       #1 行毎読み込み line に代入
  mas << line.chomp.split(//).map{|it| it.to_i}
end
```

```
$ ruby sudoku.rb question.txt
```

プログラムが解く問題をファイルから読み込み、配列に格納します。読み込むファイルのパスはコマンドライン引数から取得できるようにします。Ruby のプログラムを実行するとき、

```
$ ruby <Ruby プログラムファイル> <引数 1> <引数 2> ...
```

のようにプログラムファイルのパスの後ろに指定した引数は、Ruby 側では配列 ARGV として受け取ることができます。

今回のプログラムでは、1 番目の引数で読み込むファイルのパスを指定するので、ARGV の 0 番目の要素が、問題が書き込んであるファイルのパスとなります。ここで、shift メソッドを利用します。shift メソッドは、配列の先頭要素を取り除いて、それを返します。その返されたもの、即ち問題が書き込んであるファイルのパスが file_name に代入されています。そして、File.foreach メソッドにより、ファイルの open、1 行読み込んでそれを変数 line に繰り返し代入、ファイルの close を行っています。

4 行目は、3 行目で代入された line を以下の手順で変換します。chomp メソッドにより、変数 line の末尾にある改行コードを消します。split メソッドにより、一文字づつに分けます。それらが map メソッドのブロック内にある it に代入され、to_i メソッドにより整数に変換 (”*”は整数 0 になる) されます。そして、それを配列 mas に格納しています。よって、以下のように問題が格納されています。
mas=[[0, 0, 5, 3,…, 0],[8, 0,…, 0],……,[0, 0,…, 9, 7, 0]]

6.4.2 クラスの定義とオブジェクトの生成

```
class Sudoku
  def initialize(mas)
    @mas=mas
    @rows=3                 #rows:行:ブロックのサイズ
    @cols=3                 #columns:列:ブロックのサイズ
    @block=@rows*@cols      #3*3 のブロック 1 つ分
  end
end
a=Sudoku.new(mas)
a.show                       #問題表示
a.solve(0)
```

上のようにすることで、Sudoku クラスを定義することが出来ます。オブジェクト指向言語は、クラスからオブジェクトを生成し、クラス内で記述されているメソッドをオブジェクトから呼び出して、さまざまな処理をさせます。クラスからオブジェクトを生成するには、クラスメソッドである new メソッドを利用します。クラス名.new とすることで、クラスからオブジェクトを生成できます。new メソッドの引数 (上のソースなら mas) は initialize メソッドに自動的に渡されます。

@mas や@rows など、先頭に@が付いているものをインスタンス変数と呼びます。これは生成されたオブジェクト固有の変数で、クラス内のどのメソッドでも用いることが出来ます。クラス内で記述されているメソッドは生成されたオブジェクト固有のものであり、そのオブジェクトでしか利用できません。そのようなメソッドをインスタンスメソッドと呼びます。インスタンスメソッドである show メソッド (後述) で問題を表示し、さらに solve メソッド (後述) に引数 0 を渡しています。

6.4.3 数独の計算部分

```
class Sudoku
  def solve(p)                #調査位置 p。p=0 から始まる
    if p>=81                  #p が 81 を超えたら計算終了
      puts "find!!"
      show                   #解答表示
      exit
    end
    row=p / @block            #調査位置 p を行と列に変換
    col=p % @block
    if @mas[row][col]!=0      #すでに数字が確定してる→ solve(p+1)
      solve(p+1)
    else                      #空マスに適当な数字 k を置く→ solve(p+1)
      for k in 1..9           #for ループ。9 を含む
```

```
        if put_check(row, col, k)==true
          @mas[row][col]=k
          solve(p+1)
          @mas[row][col]=0
        end
      end
    end
  end
  def put_check(row, col, k)
    for gyo in 0...9        #for ループ。9 を含まない
      if @mas[gyo][col]==k
        return false        #既に同じ数字が行で使われてる
      end
    end
    for retsu in 0...9
      if @mas[row][retsu]==k
        return false        #既に同じ数字が列で使われてる
      end
    end
    row_block=row / @rows * @rows
    col_block=col / @cols * @cols
    for gyo in 0...@rows
      for retsu in 0...@cols
        if @mas[gyo+row_block][retsu+col_block]==k
          return false        #既に同じ数字がブロックで使われてる
        end
      end
    end
    return true               #適当な数字 k が決まる
  end
end
```

まず、solve メソッドの定義の、「空マスに適当な数字 k を置く→ solve(p+1)」というコメントがなされている else 文の中で、バックトラック法を用いています。

put_check で、同列・同行・同ブロックで使用されていない適当な数字 k を調べます。@mas に格納されてる 0 の箇所に数字 k を当てはめていき、再帰呼び出しをして矛盾が生じるまで進めていきます。もしも矛盾が生じたならば一つ前の状態に戻すために、その場所を 0 にして、再度別の数字 k を当てはめます。この作業を p が 81 を超えるまで繰り返します。そして、p が 81 を超えたら、計算終了です。

6.4.4 表示メソッド

```
class Sudoku
  def show
    print "+-------+------+------+","\n"
    for i in 0...9
      for j in 0...9
        if j%3==0
```

```
          print " |"
        else
          print " "
        end
        if @mas[i][j]==0
          print "*"
        else
          print @mas[i][j]
        end
      end
      print " |","\n"
      if i%3==2
        print "+-------+------+------+","\n"
      end
    end
  end
end
```

6.5 実行画面

```
$ ruby sudoku.rb question.txt

+-------+------+------+
 |* * 5 |3 * * |* * * |
 |8 * * |* * * |* 2 * |
 |* 7 * |* 1 * |5 * * |
+-------+------+------+
 |4 * * |* * 5 |3 * * |
 |* 1 * |* 7 * |* * 6 |
 |* * 3 |2 * * |* 8 * |
+-------+------+------+
 |* 6 * |5 * * |* * 9 |
 |* * 4 |* * * |* 3 * |
 |* * * |* * 9 |7 * * |
+-------+------+------+
find!!
+-------+------+------+
 |1 4 5 |3 2 7 |6 9 8 |
 |8 3 9 |6 5 4 |1 2 7 |
 |6 7 2 |9 1 8 |5 4 3 |
+-------+------+------+
 |4 9 6 |1 8 5 |3 7 2 |
 |2 1 8 |4 7 3 |9 5 6 |
 |7 5 3 |2 9 6 |4 8 1 |
+-------+------+------+
 |3 6 7 |5 4 2 |8 1 9 |
```

```
 |9 8 4 |7 6 1 |2 3 5 |
 |5 2 1 |8 3 9 |7 6 4 |
+-------+------+------+
```

6.6 おわりに

クラス?オブジェクト??メソッド???という様にチンプンカンプンの状態でリファレンスや Ruby の本を読みながら、今まで進めてきました。実際、それらを曖昧にしか捉えれてなかったりします……。とはいえ、一応キチンと動作するものが作れてほっと一安心しています。(計算時間は長いけど)

C 言語のときファイル操作に四苦八苦した分、Ruby ではそれが容易に扱えるのが、本当に嬉しかったです。

Ruby の特徴として、文字列を扱うことに長けているという記述をよく目にしますが、今回のプログラムでは、あまりその周辺のことが学べませんでした。だから、次は何か文字列をいろいろとイジルものを作りたいです。

6.7 参考文献

[1] 原信一郎　著　まつもと　ゆきひろ　監修
Ruby プログラミング入門

[2] Ruby リファレンスマニュアル
http://www.ruby-lang.org/ja/man/html/index.html

[3] 近藤嘉雪　著
定本 C プログラマのためのアルゴリズムとデータ構造

[4] 有限会社メディアチップス
http://mediatips.co.jp/puzzle-9x9.html

[5] WATANABE Tetsuya HOME Page
Ruby で数独
http://homepage1.nifty.com/~tetsu/ruby/puzzle/sudoku.html

7　シューティングゲーム作成

情報工学課程 1 回 河端 駿也

7.1　はじめに

自分のコンピュータ部での活動というと、C 言語勉強会で C 言語の基礎を知ったり、ICPC という C 言語も使えるプログラミングコンテストの予選で惨敗したりだったのですが、C 言語を使って作品を作るということはしていなかったので、今回 SDL[1] というライブラリを使ってシューティングゲームを制作しました。

7.2　SDL とは

SDL はクロスプラットフォームのマルチメディアライブラリです。クロスプラットフォームとは異なる OS にも対応していること、つまり Windows で動かせたことを Unix でも同じように動かすことができるのです。自分は普段、Ubuntu という Linux ディストリビューションを使っており、Linux でもゲーム作りに使えるライブラリはないかと探したところ SDL をみつけしました。

7.3　ゲーム全体の流れ

```
メイン関数 {
        初期化処理
  メインループ {
            画像表示
            入力処理
            自機の移動
            自弾の移動
            敵の出現
            敵の移動
            敵弾の移動
            当たり判定
            画面出力
            イベント処理
            終了判定
        } 終了処理
  }
```

まず SDL の初期化をします。そしてゲームの中心となるメインループに入り、敵の出現や当たり判定といったイベントの処理を終了条件を満たすまで繰り返します。終了条件を満たせばメインループから抜け出し終了処理をする、これが全体の流れです。この中から初期化処理、画像の表示、移動、当たり判定、終了処理について説明します。

7.4 初期化処理

まずは初期化です。SDL のリファレンスは日本語に翻訳されています [2]。SDL の初期化には SDL_Init 関数を使います。

```
int SDL_Init(Uint32 flags);
```

flags には SDL のどの機能を使用するか指定します。ビデオ系なら SDL_INIT_VIDEO、タイマ系なら SDL_INIT_TIMER を指定します。SDL_Init は失敗すると-1、成功すると 0 を返すので、if 文で初期化できたかを判断します。ウィンドウの初期化には

```
SDL_Surface *SDL_SetVideoMode(int width, int height,
                              int bpp, Uint32 flags);
```

を使います。ここで、横幅・高さ・ピクセル濃度・描画にシステムメモリを使うかビデオメモリを使うかといったビデオモードを指定します。SDL の初期化の他に，自機の初期位置やゲームで使う乱数も初期化しておきます。

7.5 メインループ

1. 画像表示

   画像はネット上のフリー素材を使用しました。着弾時の爆発エフェクトはフリーの生成ソフトもあります。今回は発色弾というソフトを使用しました。これは exe ファイルですが wine を入れれば Unix 上でも動作します。画像の読み込みには IMG_Load 関数を使います。

   ```
   SDL_Surface *IMG_Load(const char *file);
   ```

   SDL には画像読み込みに SDL_LoadBMP 関数がありますがこれは BMP 形式のファイルしか読み込めません。そこで補助ライブラリ sdl_image の関数 IMG_Load を使います。この関数は BMP だけでなく GIF, PNM, XPM などが読み込めます。

   次は矩形領域を定義します。矩形領域を表現するために，SDL Rect 構造体を使います。次のように定義されます。

   ```
   typedef struct{
       Sint16 x, y;
       Uint16 w, h;
   } SDL_Rect;
   ```

   x, y が短形の左上端の位置、w, h が短形の横幅と高さです。SDL のウィンドウでは左上が原点になります。そしてそして読み込んだ画像を画面に表示するには SDL_BlitSurface 関数を使います。

   ```
   int SDL_BlitSurface(SDL_Surface *src, SDL_Rect *srcrect,
                       SDL_Surface *dst, SDL_Rect *dstrect);
   ```

   この関数は転送元サーフェス (src) から転送先サーフェス (dst) へ高速転送を行います。srcrect, dstrect には各サーフェスの転送する部分を指定します。全体を転送する場合は srcrect を NULL にします。

   ```
   void DrawShot(void) { /* 自弾の表示 */
       SDL_Surface *pSurface = IMG_Load("001.png");
   ```

```
    int i;
    for (i = 0; i < SHOT_MAX; i++) {
        if(!Shot[i].view)
            continue;
        SDL_Rect dest;
        dest.x = (int) Shot[i].pos.x;
        dest.y = (int) Shot[i].pos.y;
        SDL_BlitSurface(pSurface, NULL, screen, &dest);
   /* screen:ウィンドウのスクリーン; グローバル変数として定義 */
    }
}
```

自弾の位置 Shot[i].pos には浮動小数点数 (double 型) を使っています。理由は浮動小数点数は整数 (int 型) より精度が高いので、自弾の位置をより細かく計算できるからです。

2. 入力処理

上下左右、弾のショットのボタンを決めます。ゲームの操作はキーボードから行うことにしました。キーボードからの入力は SDL_GetKeyState 関数で取得できます。例えば，J キーが押されたとき、自機を下に動かしたければ

```
Uint8 *keys=SDL_GetKeyState(NULL);
```

とし、どのキーが押されているか取得

```
if(keys[SDLK_j] == SDL_PRESSED)
    input.down=1;
    else
    input.down=0;
```

押されていれば 1、それ以外は 0 を input 構造体のメンバに代入して覚えておきます。他のボタンについても同様に、それぞれ input のメンバ input.up, input.left, input.right, input.shot に代入しておきます。

3. 移動関連

自機の移動は if 文を使います。もし、そのフレームに移動ボタンが押されていたら、ボタンに応じて自機の位置を更新します。さきほどの例なら

```
if(input.down == 1)
    Fighter.pos.y += 4.0
```

とします。

次に移動範囲の制限についてです。これをしないと場外に飛び出てしまいます。下に行き過ぎないためには (ウィンドウの高さ - 画像のピクセル数) の場所で停止させます。

```
if(Fighter.pos.y > (480.0-64.0))
    Fighter.pos.y = (480.0-64.0);
```

敵や弾の移動も同じよう if 文を使います。ある Y 座標を過ぎたら敵の動きを変えるという動作もあれば、弾丸の挙動に三角関数を使うという手もあります。複雑な数式を試してみるのも面白いと思います。

4. 当たり判定

当たり判定とは、ある物体（自機）が他の物体（弾、敵機など）に当たっていると判定される領域のことです。今回は当たり判定を矩形で考えることにしました。画面表示の時に SDL_Rect という矩形領域を表現する構造体を定義しましたが、またこれを使おうと思います。まず二つの矩形 a, b があります。これらは左上と右下の座標の組で表すことができます。$a_L(x0, y0)$, $a_R(x1, y1)$, $b_L(x2, y2)$, $b_R(x3, y3)$。SDL の原点は左上です。ここで a, b が相手の矩形に重なっている場合は、

(a) a が左側で a_L のほうが b_L よりも x 軸への垂直距離が短い時

(b) a が左側で b_L のほうが a_L よりも x 軸への垂直距離が短い時

(c) a が右側で a_L のほうが b_L よりも x 軸への垂直距離が短い時

(d) a が右側で b_L のほうが a_L よりも x 軸への垂直距離が短い時

という全部で 4 通りが考えられます。ここから矩形が重なっている場合に最後まで残る条件は、$(x0 < x3) \wedge (x2 < x1) \wedge (y0 < y3) \wedge (y2 < y1)$ です。よって当たり判定の関数は次のようになります。

```
int HitRect(SDL_Rect *a, SDL_Rect *b) {
    if ((a->x) < (b->x + b->w)) &&
        ((b->x) < (a->x + a->w)) &&
        ((a->y) < (b->y + b->h)) &&
        ((b->y) < (a->y + a->h)) )
            return 1;
    return 0;
}
```

7.6 終了処理

SDL を終了させるには SDL_Quit() を書きます。SDL_Quit() はすべての SDL サブシステム（SDL_INIT_VIDEO や SDL_INIT_TIMER）を停止し、それらが確保したリソースを解放します。

7.7 プレイ

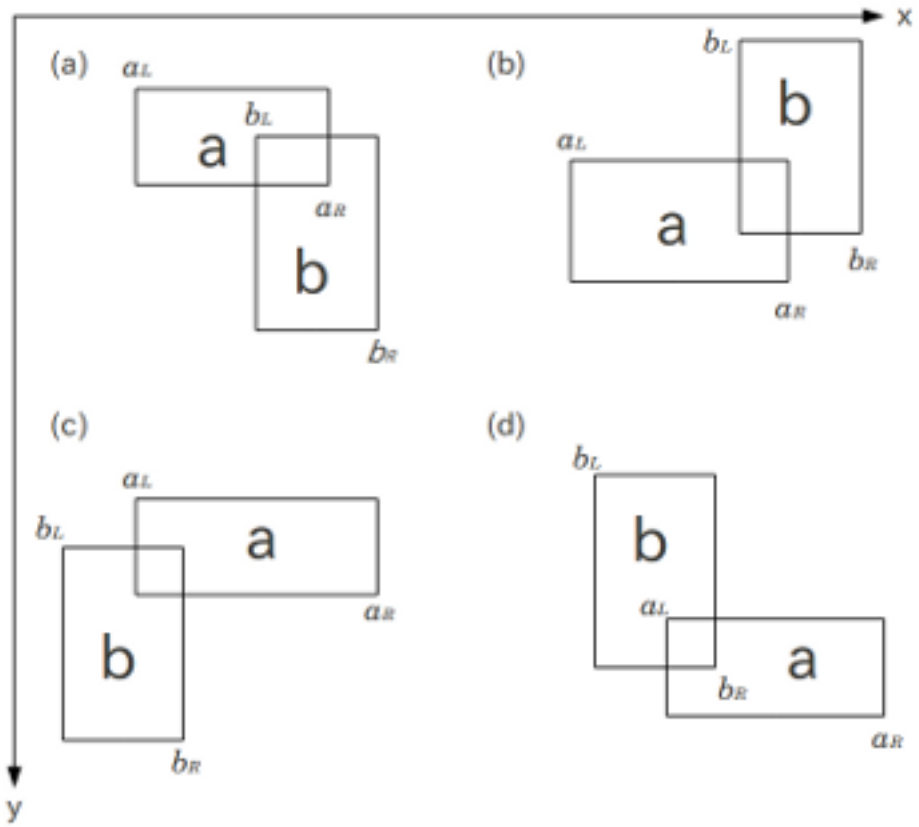
やってみると、欠けることなく画像が表示されます。キーボードによる入力も動作しています。当たり判定の動作もしっかりです。

7.8 まとめ

シューティングゲームは作りやすい、と聞いた記憶があります。確かに慣れると、動作の変更も容易です。

7.9 今後の課題

今まではシューティングゲームの基礎となる部分を作ってきました。そのため面白みがありません。今後は、背景を付けてそれをスクロールさる、敵のバラエティを増やす、自弾などが変わるアイテムを出現させるなどの制作に力を入れていこうと思います。



7.10 参考文献

[1] 「Simple DirectMedia Layer」
＜http://www.libsdl.org/＞

[2] 「SDLdoc 日本語翻訳プロジェクト」
＜http://www.tacoworks.jp/software/SDLdoc-jp/＞

[3] 「初心者の初心者による初心者の為の SDL」
＜http://tokyo.cool.ne.jp/sdl/＞

8　初めて作る電源装置

電子システム工学課程 1 回 松本 駿

8.1　製作の背景

大学に入ってから少しずつ電子工作をやるようになりました。当然ながら作ったものを動かそうとすると電源が必要です。最初の内は電池を使用していましたが、電池では 5V など 1.5 の整数倍でない電圧を出す際にその都度その都度レギュレータ等を用いて調整しなければならず、面倒でした。そこで自分がよく使う電圧を出せる電源装置を作ろうと考えました。電池と異なり、使用していて電圧が落ちたり消耗することが無いので交換する必要もありません。ただ、本格的なトランス等を用いたものは技術面からも予算面からも厳しかったので、電子工作初心者なりに、ある程度簡単かつそこそこ使える性能を持つ電源を目指すこととしました。

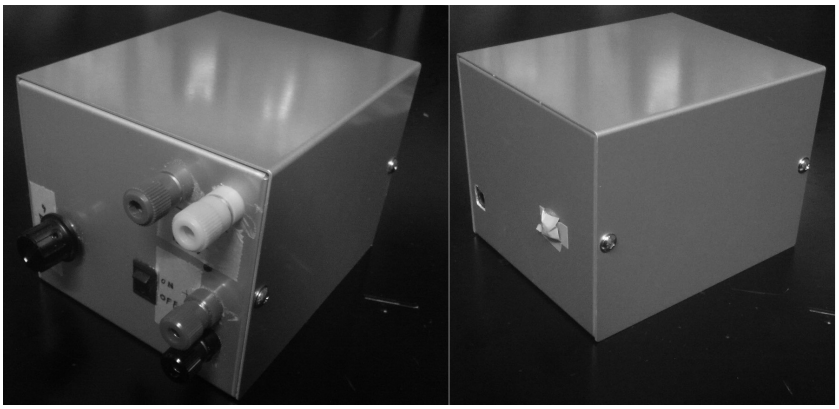


図 8.1: 電源装置外観

8.2　設計指針

1. 入力: AC100V

   使い勝手を考え、一般の家庭用電源から供給できるようにします。

2. 出力: DC3〜9V

   最大電流 1A 程度を想定　出力電圧は自身の必要十分なものです。

3. 大きさと重量: 片手で扱える範囲

   極端に大きすぎたり重すぎたりすると使いづらいのでこのように。実際、製作においてこれは問題にはなりませんでした。

8.3 構成

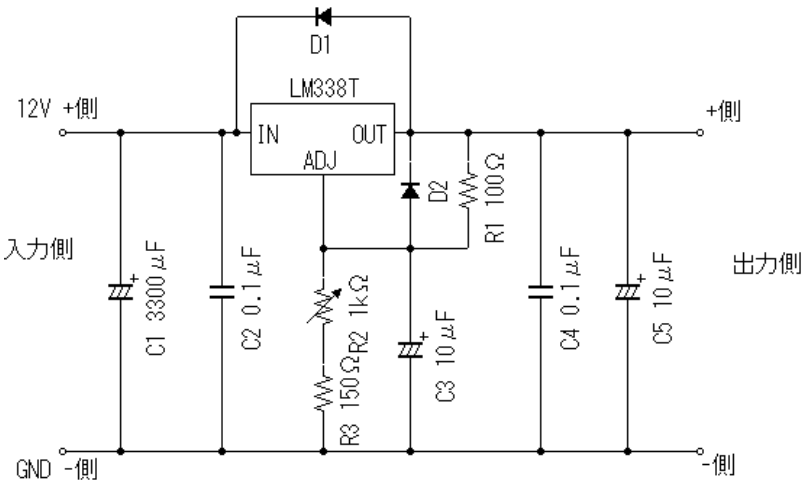


図 8.2: 回路図

メインの回路図は図 8.2 の様になりました。秋月電子のデータシートをベースに、一部抵抗値を変更しています。

8.3.1 変圧と整流

トランスとダイオードブリッジの使用も考えましたが、熱処理とパーツ費用の問題があり断念しました。代わりに 12V1A 出力の特価品 AC アダプタを利用します。これを使うことにより AC-DC 変換と変圧の為の機構は気にせずに済みます。

8.3.2 電圧制御部

前述のアダプタから出力された直流 12V を 3〜9V とするにはレギュレータを用います。レギュレータの出力電圧は R1、R2、R3 の抵抗値で決まります。R2 に可変抵抗を用いることによりその出力電圧を連続的に変化させることが可能です。

その計算式は次式で表されます。

$$V_{OUT} = V_{ref}(1 + \frac{R2 + R3}{R1}) + I_{ADJ}(R2 + R3) \tag{8.1}$$

ここで、I_{ADJ} はレギュレータの ADJ ピンを流れる 50μA 程度のかなり微弱な電流であるため、0 とみなせば I_{ADJ} を含む項は無視できます。また V_{ref} はレギュレータの ADJ と OUT の電位差で、ほぼ一定値をとります。今回これが約 1.3V であり、R1 を 100Ω、R2 を可変 1kΩ、R3 を 150Ω として各値を (8.1) 式に代入すると、

$$V_{OUT} = 1.3(1 + \frac{R2(\text{可変}) + 150}{100}) \tag{8.2}$$

となります。ここで、R2 の抵抗値の幅が

$$0 < R2 < 1000\,(\Omega) \tag{8.3}$$

なので、(8.2)、(8.3) 式より出力可能な電圧の値は概算で、

$$3 < V_{OUT} < 16\,(\mathrm{V}) \tag{8.4}$$

と表すことができます。(8.4) 式だけ見れば 16V まで電圧を上げることが可能であるようにも思えますが、使用する AC アダプタの出力が 12V の為どんなに R2 の抵抗値を上げても 12V 以上にはなりません。ちなみに、似たモデルの 24V 出力 AC アダプタを用いると 16V の出力は可能でした。

また、正常動作させる場合、出力は入力より 3V 程度低くすること、とレギュレータのデータシートに記載されているため実質的に 12-3=9(V) 程度までを限度としています。実際に抵抗値を上げれば約 12V の出力が得られることは確認しましたが、安定動作を考えて可変抵抗 (ボリューム) のツマミに突起を設け、9V まで回すとケース側のピンにぶつかるように調整してそれ以上の出力ができなくなるようにしました。

8.3.3 電源ランプ

メインの動作とは関係ありませんが、テスト時にスイッチのみでは電源が入っているかどうか見た目で分かりにくかったので確認用として LED を 1 つ設置しました。些細な追加点ではありますが使い勝手に影響する部分です。

1kΩ の抵抗と 3mm 径赤色 LED を直列にして主回路と並列に繋いであり、電源を入れると点灯します。

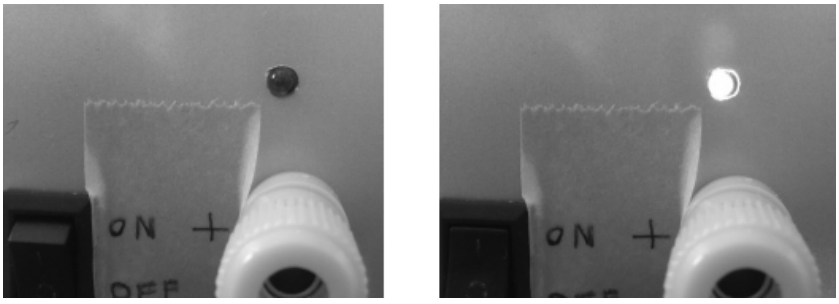


図 8.3: 電源ランプの動作 左:電源 OFF 右:電源 ON

8.3.4 12V 出力

12V のアダプタを繋げているのに 9V までしか出力できないのも妙です。そこで上記の LED にさらに並列に端子を接続し、12V をそのまま利用できるようにしました。前面上部の 2 本の端子はその 12V 出力の＋と－にあたります。

8.4 筐体 (ケース) と組立

寺町通りのパーツショップで AC アダプタや基板等と共に 500 円程度のネジ止め式アルミ (？) 製ケースを購入しました。放熱と固定のためケースに穴を開けて基板上のレギュレータをネジ止めします。その際、テスタで当たるとケース自体は伝導しなかったのですが念のためゴムワッシャと絶縁シートを使いました。

また、1A を超えるような大電流を流すことは考慮していない為、放熱板を設けたりはしていません。現状ケース自体での放熱で十分のようですが、もし今後改造してより大きな電流を流すようにするのであれば放熱板も設置することになると思います。

工作部分はケースに穴を開けてツマミやスイッチ、AC アダプタ端子の為の開口部を作ったのみです。ただ今回ケース加工に一番時間を取られた気が……。

それぞれのパーツを載せてみて、ケーブル長や絶縁等に問題が無いことを確かめてから固定して完成です。余談ですがホットボンドはかなり使い勝手がよく、製作時に重宝しました。

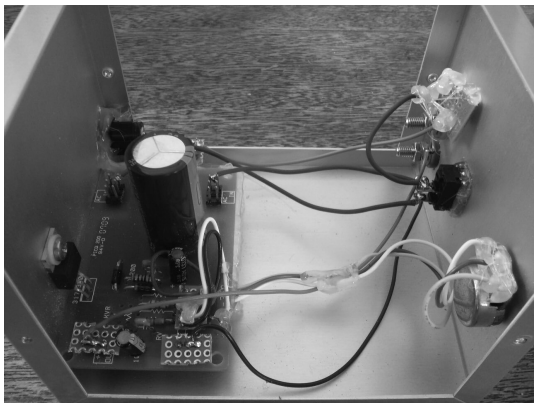


図 8.4: 中身

8.5 蛇足

8.5.1 USB 端子を用いた電源供給

少しはコンピュータにちなんだことを。

ケース加工用のリュータ[1]の電源確保にノート PC(USB2.0 対応) の USB 端子を利用しました。USB には 4 本の線があり、コネクタのメス側を開口部側から見て左から Vcc(赤) Data-(白) Data+(緑) GND(黒) です。白と緑が信号線で、赤が+5V、黒がグラウンドを示しています。実際に端子が色分けされているわけではありませんが、USB 接続の機器のケーブルを分解するとこの色の被膜が使われていることが多いです。

オス型 USB 端子から両端の 2 本を引き出せば、手軽に使える 5V の電源とみなすことができます。ただし複数 USB 端子があっても GND は共通なので直列に繋いで 10V を出す、ということは出来ません。短絡の恐れがあります。

出力できる最大の電流は USB の規格[2]で決まっているため大電流を流す用途には使えませんし、そもそも機種により実際に流せる電流の最大量も異なるようなので過信はできませんがそれほど電流を必要としない時には便利です。USB2.0 であれば 5 ミリ径の赤色 LED を 10 から 20 個程度を光らせるなら余裕 (のはず) です。

8.6 終わりに

8.6.1 完成品仕様

寸法: 横 10cm 高さ 9cm 奥行き 12cm (ツマミ等の突起物除く)
重量: 258g (AC アダプタ除く)
入出力は設計指針に準じます

[1]電動の回転式切削工具。先端部を交換することで様々な切削を行える。
[2]ローパワーデバイス:100mA ハイパワーデバイス及び USB2.0:500mA USB3.0:900mA

8.6.2 感想

今回作った電源装置ですが、回路図はほぼデータシート通りでパーツ構成も秋月電子の電源キットのものにほぼ準じており、更に AC アダプタを利用することで工作としてはかなり難易度の低いものでした。初心者級の私が挑戦してみるには都合はよかったです。このようなものでも文章の執筆時点では不具合もなくきちんと動作しています。製作自体もですが、電源装置を手に入れることによりできることが広がったのがうれしいです。これを使ってもっといろんなものを作りたいと思います。

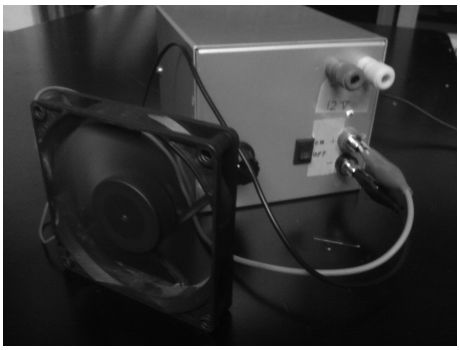


図 8.5: 実際の動作 5V でファン回転中

8.7 参考文献

[1] 安定化電源キット マニュアル (秋月電子通商)
<http://akizukidenshi.com/>

[2] LM338 データシート (ナショナルセミコンダクタ)
<http://www.national.com/JPN/>

9　迷路の自動生成

情報工学課程 1 回 津田 啓史

9.1　はじめに

みなさん、迷路ってご存知ですか。そうです、あの迷路です。小学生のときに自由帳にいっぱい作りましたよね。共感してくださる方がそれなりにいると思うのですが…。僕は、この迷路に何故か思い入れがあるようで、C 言語勉強会[1]で初めてゲームを作ったときから “迷路のゲームを作りたい！” と思っていました。ということで良い機会なので挑戦してみました。

9.2　迷路の自動生成

9.2.1　迷路生成

迷路の自動生成には、棒倒し法、穴掘り法 (道延ばし法)、壁のばし法などいくつか方法があります。今回は、比較的楽にプログラミングできる棒倒し法で作っていきます。参考にしたサイトは Samayou Oharikui[1] です。なお、壁のばし法については過去の Lime vol.40 に掲載されていますので、興味のある方は是非ご覧になってください。

9.2.2　棒倒し法

棒倒し法の手順について説明します。

1. 基本となる四角の外壁と、その中に 2 マスおきに等間隔に内壁を配置します。
2. 内壁の一段目から棒に見立てて倒していきます。ここは上下左右にランダムに倒します。ただし、棒が倒れるところが重ならないように避けて倒します。
3. 二段目以降は、上に倒さないようにして、下左右の三方にランダムに倒します。こちらも倒すところが重ならないように避けて倒します。
4. これをすべての内壁に適用すると迷路が完成します。

このように棒倒し法は非常に単純で作りやすいのですが、短い袋小路が多くなったり、スタートからゴールまでのルートが分かりやすかったりなど、悪癖が数多くあるのです。このあたりは目を瞑ることにして作っていきます。

[1] コンピュータ部では年にいくつか勉強会が開かれます。

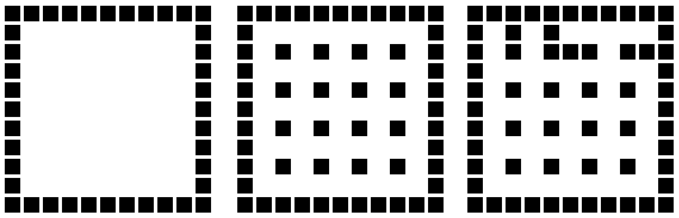


図 9.1: 棒倒しの様子

9.3 プログラム

9.3.1 それでは早速始めよう

プログラミングにあたって、『プログラミング言語 C』[2] という本で復習しながら組みました。
まず必要となるものを定義、宣言しておきます。

```
#define KABE 0
#define MICHI 1
#define START 2
#define GOAL 3
#define SizeX 39
#define SizeY 39

int masu[SizeY][SizeX];
```

ここで SizeY, SizeX は縦横のマス目の数を表します。今回は 39 × 39 マスで作ります。方眼紙の各マスに壁や道を配置していくイメージです。道は白のまま、壁は黒くする感じで作っていきます。

9.3.2 初期壁生成

```
//初期壁生成
int i, j;
for(i = 0; i <= SizeY-1; i++){
  for(j = 0; j <= SizeX-1; j++){
    masu[i][j] = MICHI;
  }
}

for(i = 0; i <= SizeX-1; i++){
  masu[0][i] = KABE;
  masu[SizeY-1][i] = KABE;
}

for(i = 0; i <= SizeY-1; i++){
  masu[i][0] = KABE;
  masu[i][SizeX-1] = KABE;
}
```

```
for(i = 1; i <= (SizeY-3)/2; i++){
  for(j = 1; j <= (SizeX-3)/2; j++){
    masu[2*i][2*j] = KABE;
  }
}

//スタートとゴールの配置
masu[1][1] = GOAL;
masu[SizeY-2][SizeX-2] = START;
```

はじめにすべてのマスを真っ白な道にします。次に大きく四角の外壁を生成し、その中に２マスおきに内壁を配置します。迷路っぽくするためにスタートとゴールも作りました。

9.3.3 棒を倒す

ここで棒を倒していくのですが、ランダムに倒すために疑似乱数を使います。疑似乱数とは、コンピュータ内である規則に従って生成される値のことです。今回は rand 関数を使います。

【書式】

```
#include <stdlib.h>
int rand(void);
```

rand 関数は、０～RAND_MAX の間から疑似乱数を返します。RAND_MAX は stdlib.h の中でマクロ定義されています。

```
int r;

for(i = 1; i <= (SizeX-3)/2; i++){

  r = rand();

  if(r%4 == 0){
    if(masu[2][2*i-1] == KABE){
      r = rand();
      if(r%3 == 0){
        masu[2][2*i+1] = KABE;
      }
      else if(r%3 == 1){
        masu[3][2*i] = KABE;
      }
      else if(r%3 == 2){
        masu[1][2*i] = KABE;
      }
    }
    else masu[2][2*i-1] = KABE;
  }
```

```
  else if(r%4 == 1){
    masu[2][2*i+1] = KABE;
  }
  else if(r%4 == 2){
    masu[3][2*i] = KABE;
  }
  else if(r%4 == 3){
    masu[1][2*i] = KABE;
  }
}

//2 段目以降 (上以外の三方向に分かれる)
for(i = 2; i <= (SizeY-3)/2; i++){
  r = rand();
  for(j = 1; j <= (SizeX-3)/2; j++){
    r = rand();
    if(r%3 == 0){
      if(masu[2*i][2*j-1] == KABE){
        r = rand();
        if(r%2 == 0){
        masu[2*i][2*j+1] = KABE;
        }
        else if(r%2 == 1){
          masu[2*i+1][2*j] = KABE;
        }
      }
      else {masu[2*i][2*j-1] = KABE;
    }
    else if(r%3 == 1){
      masu[2*i][2*j+1] = KABE;
    }
    else if(r%3 == 2){
      masu[2*i+1][2*j] = KABE;
    }
  }
}

//描写
for(i = 0; i <= SizeY-1; i++){
  for(j = 0; j <= SizeX-1; j++){
    if(masu[i][j] == KABE){
      printf("■");
    }
    else if(masu[i][j] == START){
      printf("入");
    }
```

```
    else if(masu[i][j] == GOAL){
      printf("出");
    }
    else{
      printf("  ");
    }
  }
  printf("\n");
}
```

試行錯誤しながら組んだのですが、力技になってしまいました。ここで気をつけなければいけないことが、すでに倒したところに倒してはいけないという点です。左から右、上から下の内壁から順に倒していくので、左側に倒すときにそこに壁があった場合、左以外に倒れるようにしました。最後にマスを順に描写して完成です。

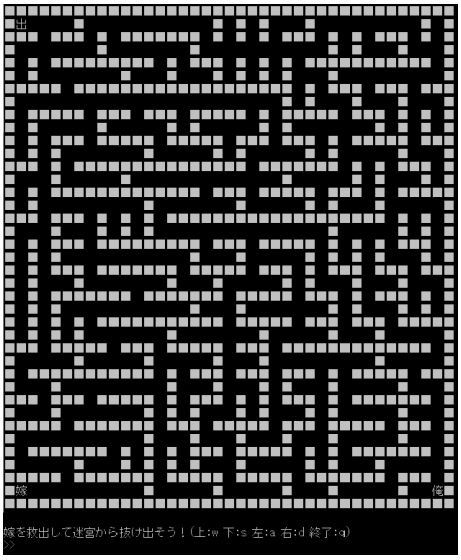


図 9.2: ゲーム画面

9.4 応用

これだとただの迷路なので、この迷路をもとにゲームを作りました。

プレイヤーは迷宮に捕らわれた嫁を救出して脱出するという単純明快なゲームです。迷宮を徘徊するモンスターを出すともっとおもしろくなると考えたのですが、力量不足でうまく動かせませんでした。

9.5 おわりに

改善点が多々ありますが、ちゃんとした迷路ができて一安心しています。次はワイヤーフレームを使って、よりダンジョンっぽくしてみようと思います。

参考文献

[1] Ishida So
Samayou Oharikui
＜http://www5d.biglobe.ne.jp/stssk/＞

[2] B. W. カーニハン/D. M. リッチー 著/石田晴久 訳
『プログラミング言語C』 第2版 共立出版

編集後記

Lime43 号、お楽しみいただけたでしょうか。「てっくす?それ美味しいの？」という、編集能力ゼロの状態から始まった編集作業でしたが、作業に知恵をかしてくださいました部員達の力で、なんとか形にする事ができました。この場をかりてお礼を言わせていただきます。また、編集にあたり歴代の Lime を読んでいると、記事にかける情熱はもちろん、後年の後輩がより簡単に編集できるようにと、スタイルファイル等を作っていて下さいました数年前の先輩方のお気遣いに、とても励まされました。

そして今 Lime を手にとって下さっています読者様。ここまで読んでいただきまして、ありがとうございました。来年の編集担当は、もっとうまくやってくれると思います。どうぞご期待下さい。

平成 22 年 11 月 20 日 編集担当 鴛渕 真理