百萬石

Rubyにおける手続きの抽象化
やはり俺のirbシェル化計画はまちがっている。
液晶制御欲が高まったわけだが
PostgreSQLで単回帰分析
いろいろなiPhone5sの買い方

MMA

目次

第 1 章　Ruby における手続きの抽象化　alstamber　1

1.1　手続きとは 1
1.2　block つきメソッド 4
1.3　proc 5
1.4　lambda 7
1.5　`Method` オブジェクト 8
1.6　`Symbol#to_proc` メソッド 9
1.7　`Fiber` 9

第 2 章　PostgreSQL で単回帰分析　hayakawa　11

2.1　データの準備をする 11
2.2　単回帰分析 12
2.3　感想 13

第 3 章　やはり俺の irb シェル化計画はまちがっている。　KA ☆ SI ☆ O　15

3.1　夏休みが終わったのにこの猛暑はどう考えてもおかしい。 15
3.2　きっとどこにでも異形の花は咲いている。 16
3.3　誰だってポイントフリーという病を通過点としている。 17
3.4　だから俺はスターシステムを支配する。 21
3.5　ようやく進捗確認と共に物語は始まる。 23
3.6　けれど彼には DT 力が足りない。 24
3.7　つまるところ俺たちの魔法に複雑な呪文はいらない。 26
3.8　どんなに長い夜でも必ず朝はおとずれる。 28

第 4 章　情報開示請求 簡易報告　KHe7　33

4.1　はじめに 33
4.2　前記事以降の手続き 33
4.3　開示文書の内容 34
4.4　おわりに 35

4.5 参考文献 35

第 5 章 いろいろな iPhone5s の買い方 Mernao 37

5.1 注意事項 · 免責事項 37
5.2 用語解説 38
5.3 iPhone5s について 39
5.4 MNP 優遇について 39
5.5 いろいろな買い方 39
5.6 結局どうすれば良いのか 42
5.7 おわりに 42
5.8 参考文献 43

第 6 章 液晶制御欲が高まったわけだが mznh 45

6.1 マイコン 45
6.2 キャラクタ LCD 45
6.3 参考文献 48

第 1 章

Ruby における手続きの抽象化

alstamber

Ruby は Web 開発を中心に現在広く使われているプログラミング言語の一つである。Ruby は block や Proc、lambda などによって手続きを抽象化して柔軟に取り扱うことができる。この記事では、Ruby における手続きの抽象化について例を挙げながらまとめたいと思う。

1.1 手続きとは

プログラミング言語において、手続きという概念はそれなりに広い意味を持っている。大雑把に定義するのであれば「プログラムの小さなかたまり」と定義するのが妥当であろう。プログラムを書いていると、同じような処理が何度も登場することに頻繁に出くわすことは、プログラムを書いた経験がある人なら誰もが承知しているだろう。このような何度も登場する処理を予めまとめておき、そのような処理が必要となったらそれを呼び出すという仕組みを用意しておけば、同じ処理を何度も記述しなくてすむようになる。

手続きを形式的に定義するとすれば、0 個以上の入力を受け取って、0 個か 1 個の出力をするような存在であるとすることができる。手続きに出力が存在すれば、その出力は入力が存在するならば入力に依存して決定する。ここで重要なのは、手続きの出力は入力に「のみ」依存して決定するとは限らない事である。一般に手続きの出力は入力以外の外部要因に依存して決まる可能性を残している。例えば標準入力からの入力を返すような手続きはその一例といえる。受け取る入力の数が 0 個であるような手続きは、外部要因によって出力が決定されるか、または一切の動的な要因に影響されない、言い換えれば常に同じ出力を行う手続きであるといえる。出力を持たないような手続きも先ほど示した定義に従えば存在し得る。このような手続きは、手続きの出力としてではなく他の何らかの手段で外部に影響をおよぼすことで、その手続きの役割を果たす。標準出力に出力するような手続きはその一例だろう。

手続きは、数学的な観点から見れば、0 個の入力および 0 個の出力を許すようにした関数であるともいえる。そういった理由から、手続きのことを「関数」と呼ぶプログラミング言語も存在する。C はその一つである。一方で、厳密に数学における関数の定義に従うようにのみ手続きを定義できるようにしている言語も存在する。純粋関数型言語と呼ばれる言語のグループがそれに当

たる。

手続きの入力を引数と呼び、出力を戻り値 (返り値) と呼ぶ。

1.1.1 ファーストクラスとしての手続き

ファーストクラスとは

以降の話をすすめるために、ファーストクラスという概念を導入したいと思う。ファーストクラスという概念の定義はいくつか存在しているが、ここではかの古典的名著である SICP(計算機プログラムの構造と解釈) における定義 [?] を引きたいと思う。

> 一般にプログラム言語には、計算要素を扱う方法にいろいろな制限があるものだ。制限の殆どない要素は第一級 (first-class) 身分を持つという。第一級要素の「権利と特権」は：
> ・変数として名前がつけられる
> ・手続きに引数として渡せる
> ・手続きの結果として返される
> ・データ構造に組み込める
> である。Lisp は他の通常のプログラム言語と違い、手続きに完全な第一級身分を授与した。

広く用いられているプログラミング言語では、整数や文字といった基本的なデータ構造はファーストクラスである。たいていの言語で、整数や文字は変数として名前をつけて使うことができるし、手続きに渡すことができる。さらには、手続きの結果として返すことができ、他のもっと大きなデータ構造の要素にすることができる。

基本的なデータ構造を組み合わせたデータ構造については、必ずしもファーストクラスであるとは限らない。例えば C の配列を考えよう。C において配列名とは連続的に確保されたメモリ領域の先頭のアドレスに付けられた名前である。言い換えれば、C の配列は線形に並んだデータの集まりそのものを表すものではなく、その集まりがある場所を表すための仕組みでしかない。簡単な C のプログラムからその事実を確認することができる。

```
int *a = calloc(100, sizeof(int));
int *b;
b = a;
```

最後の代入文で配列 a の内容は b に対してコピーされるわけではない。単に、a という配列 (= ポインタ) が指す場所を b というポインタが指すようになるという意味しかない。単純な int 型の変数同士の代入文において値のコピーが行われるのとは全く意味が異なっている。

このような性質は、配列を返すような関数を C で定義したいと考えた時に問題となる。結論を述べれば、直接配列を返すような関数を C で定義することは不可能である。次のような C のコードはエラーとなる。

```
int a[2];

int f()[2]{
    return a;
```

```
}
```

これを回避する方法はいくらかある。1 つは戻り値に使いたい配列を引数にとって、それを返すことである。2 つ目は配列ではなくポインタを代用する方法である。しかしこの方法では配列の大きさという情報が欠落してしまう。最後の方法は配列に対するポインタを返すようにする方法である。先程の例は最後の方法を用いれば次のように書き換えられる。

```
#include <stdio.h>
int a[2];

int (*f())[2] {
    return &a;
}

int main(){
    printf("%d\n", (*f())[1]); /* 配列の2番目の要素を取り出す */
    return 0;
}
```

しかし、いずれの方法も本質的には解決にはなっていないのである。なぜならどの方法も配列を直接返すような関数を定義したわけではないからである。このように C においては配列を返すような手続きを定義できないので、配列は C においてはファーストクラスではない。

もっとも、この事実は他のあらゆる言語で配列がファーストクラスではないということを意味しているわけではない。Java や Ruby においては、配列はファーストクラスになっている。

ファーストクラスな手続き

手続きというしくみは、一般に広く用いられているプログラミング言語にはたいてい実装されている。手続きは、プログラミング言語では普遍的な構造であるが、これがファーストクラスになっているかどうかは言語によってまちまちである。

C は手続きがファーストクラスになっていない言語の一例である。C では手続きを関数という形で表すが、関数は変数に代入したりできないし、他の関数の引数にしたり戻り値にしたりすることができない。また関数の配列をつくったりすることもできない[*1]。

手続きがファーストクラスになっている言語としては、関数型言語と呼ばれる言語のグループを挙げることができる。関数型言語では、プログラムの主役が数学的な観点での関数になっている。C でいうところの関数は手続きの C 流の呼び名であるが、関数型言語での関数は入力が 1 つ以上で出力が 1 つあるという意味で、厳密に数学の関数と等価である。関数型言語における関数は、変数に束縛することができ、また他の関数に渡したりすることにも何ら制約がない。このような事実から、関数型言語の関数はファーストクラスである。

今回扱う Ruby においても手続きはファーストクラスとして取り扱うことができる。しかし、Ruby における手続きのファーストクラス化は関数型言語のそれほど徹底していない。これについては次の項で詳述する。

[*1] 関数へのポインタを用いればこれらの要件を満たすことは一応できる

1.1.2　Ruby における手続き

オブジェクト指向

Ruby はオブジェクト指向というスタイルをとっている言語である。オブジェクト指向とは、プログラムの主役としてオブジェクトというものを据える戦略のことである。ここでいうオブジェクトは、プログラムで取り扱うデータと、そのデータに関連する手続きをひとまとめにしたもののことを言う。オブジェクト指向では、オブジェクトが相互に関連しオブジェクトに含まれる手続きを呼び出すことで処理が進行していく。

オブジェクト指向には、徹底したオブジェクト指向とそうでないものがある[*2]。徹底したオブジェクト指向とは、プログラム中に登場しうるあらゆるデータ構造がオブジェクトとして取り扱われるようなものを言う。Ruby は徹底したオブジェクト指向を採用している。

メソッド

Ruby では手続きをメソッドという単位で表す。これは C でいうところの関数にほぼ相当する存在である。Ruby ではメソッドも `Method` オブジェクトとして表現されているが、これについては後で詳述する。

Ruby では、メソッドとは別の手続きを表す方法を提供している。まず、その別の方法について、次項以降で説明していく。

1.2　block つきメソッド

Ruby におけるファーストクラスな手続きの話をする前に、Ruby に特徴的な block つきメソッドというものについて説明する。これはユーザが定義した手続きを引数として渡すことができるようなメソッドである。例を示す。

```
(5..10).each{|num|
  puts num
}
```

ここでは `each` というものが block つきメソッドである。このメソッドは `(5..10)` というものに対して呼び出されている。この `(5..10)` は 5 から 10 までの整数の列を表しているオブジェクトである。Ruby では `(a..b)` と書くことで `a` から `b` までの整数の列を表すオブジェクトを作ることができる。そしてこのように作られた整数列を表すオブジェクトは `each` というメソッドを必ず持つようになっている。

`each` は後ろに手続きを表す引数をとる。この手続きは 1 つの引数をとる。今の場合は `num` がそれに当たる。`num` にはどのような値が実際に入力されるかというと、先ほどの整数列の要素が前から順番に与えられる。この例だと `5,6...,10` という順番で `num` に入力されることになる。`puts`

[*2] この言葉はこの記事の中だけで通用する造語である

というのは与えられた引数を標準出力に出力するメソッドなので、このプログラムを実行すると

```
5
6
7
8
9
10
```

という結果が得られることになる。

今 each メソッドに与えた手続きのことを Ruby では block と呼ぶ。block は、ある限定的な条件においてはファーストクラスであるように振る舞う。今見たように、block つきメソッドに対しては引数として与えることができるからである。しかし block はオブジェクトではない。つまり、変数に代入して使いまわすということはできない。block は限定的なファーストクラス手続きという存在であるといえる。

1.3 proc

先程も述べたとおり、block は限定的な局面ではファーストクラスであるように振る舞うが、オブジェクトではないため、ファーストクラスの厳密な定義には当てはまらない。Ruby には完全にファーストクラスとして振る舞う手続きを表すためのオブジェクトが用意されている。それが proc である。以下のプログラムは、先ほどの block を使った例と全く同じように動作する。

```
p = Proc.new do |num|
  puts num
end

(5..10).each(&p)
```

Proc.new メソッドを block を引数に渡して呼び出すことで、proc を作ることができる。今の例では、何らかの引数を受け取って、その引数を標準出力に出力するような手続きを表す proc をつくっている。5 行目で作った proc を each メソッドに渡している。本来 each メソッドは block つきメソッドであるが、与えたい proc に&をつけることで、block つきメソッドに proc を与えることができる。

proc は Proc というクラスのオブジェクトである。オブジェクトであるので、メソッドの引数としたり、メソッドの返り値としたり、また変数に代入もできる。proc の配列を作ることも何ら問題ない。よって proc はファーストクラスな手続きであるといえる。

1.3.1 block と proc

block と proc は手続きの塊を取り扱う手法という点では共通しているが、一方でもちろん異なる部分も存在している。これら 2 つについて実際に block を用いたプログラムを実装し、細かな部分について調べていくことにしよう。

map!の実装

map!は Ruby に実装されている block つきメソッドのひとつである。map!がどのようなメソッドであるのか、例を示して説明する。

```
a = [1,2,3]

a.map!{|i| i**2}
```

map!メソッドは、block に配列の各要素を順々に与える。block は配列の各要素を受け取って、それをもとに処理を行う。block の戻り値 (上の例の場合は i の 2 乗である) が得られると、その戻り値によって元の要素が上書きされる。結果として、上のプログラムの場合は、各要素が 2 乗された配列 a が生まれる。

今、map!を自力で実装することを考える。その例として my_map!を以下に示す。

```
class Array
    def my_map!
        for i in 0..self.length do
            n = self[i]
            self[i] = yield(n)
        end
    end
end
```

my_map!では、for 文を用いて配列の要素を 1 つずつ n に取り出し、それを引数として yield というものを呼び出している。そして、その結果を再び配列に代入しなおしている。

yield というものはなんだろうか。yield はメソッドに渡されている block に対応する特殊なメソッドの名前である。これを呼び出すと、メソッドについてきた block を呼び出して実行することができる。yield の引数は、block の引数となる。

proc を引数にとるメソッド

次に、my_map!を block つきメソッドではなく、proc をとるように書きなおすことを考える。このように定義したメソッド my_map2!を以下に示す。

```
class Array
    def my_map2!(&p)
        for i in 0..self.length do
            n = self[i]
            self[i] = p.call(n)
        end
    end
end
```

違いは&のついた引数をとるようになっていることと、yield ではなく p.call を呼び出していることだけである。

実際、この my_map2!に block を渡すことができる。

```
a = [1, 2, 3]
```

```
a.my_map2!{|i| i**2}
```

今まで見てきたとおり、block と proc には互換性とでも言うべきものが存在しているようである。実は block と proc は、見た目は多少異なるが、実体は同じである。

```
def a(&p)
    p.class
end

puts a {}
```

上のプログラムを実行すると、`Proc` と出力される。つまり、block は見た目こそ違えど、実際には proc の特殊な場合でしかないといえる。また、少し前で見てきたとおり、`&`を proc につけることで block として利用することができる。

1.4 lambda

proc と同じように手続きの塊を取り扱う手法として lambda というものも Ruby には用意されている。

```
a.my_map2!(&lambda {|i| i**2})
```

あるいは Ruby1.9 以降であれば、次のようにも記述できる。

```
a.my_map2!(&-> (i) {i**2})
```

lambda は proc と同じく `Proc` クラスのオブジェクトである。なので、proc と lambda は内部的には区別されていないのではないかという気もするが、実際には `Proc#lambda?`メソッドを用いることでこの 2 つは区別することができる。

lambda は proc とよく似ているが、proc との違いも存在する。それは lambda は引数の数が一致していないとエラーが起きるということである。proc では仮引数の数が多い場合、余分な変数に対して `nil` を入れることで帳尻を合わせる。逆に実引数が多い場合、余分な実引数が切り捨てられる。lambda では数が一致していない時、`ArgumentError` が発生し、プログラムは停止する。

lambda と proc の違いとしてもうひとつあげられるのは、`return` の挙動の違いである。

```
def proc_return
  Proc.new { return "proc"}.call
  return "proc_return finished"
end

def lambda_return
  lambda { return "lambda" }.call
  return "lambda_return finished"
end
```

上のプログラムを実行すると、`proc_return` の場合は、1 番目の `return` が実行され、2 番めの `return` は実行されないが、`lambda_return` の場合は 1 番目の `return` が無視され、2 番めの `return` が実行される。

これは proc があくまで block の拡張的存在であるのに対して、lambda はメソッドに近い挙動をとるように設計されているためである。proc の中で `return` されたとき、proc 自体が終了するのではなく、proc を呼び出したメソッド自体が終了するようになっているのである。一方で lambda の中で `return` されたときは lambda の処理が終わるだけなので、lambda を呼び出したメソッドには何ら影響しない。

1.5　Method オブジェクト

Ruby ではメソッドそのものも、特殊な位置づけではあるが、オブジェクトとして取り扱うことができる。それらのオブジェクトには `Method` オブジェクトという名前が与えられている。`Method` オブジェクトを生成するためには

`Method` オブジェクトを生成するには、以下のように記述する。

```
str = "hogehoge"
rvs_obj = str.method(:reverse!)

puts rvs_obj.call()
```

最後の行の `rvs_obj.call()` を実行することは、`str.reverse!()` を実行することと全く同じ効果となる。また `rvs_obj` は `str.reverse!()` をそのままオブジェクトにしたものであるので、この実行によって `str` 自体にも変更は反映される。

1.5.1　proc と Method オブジェクトの違い

proc と `Method` オブジェクトの違いとして挙げられるのは、クロージャであるかどうかという点である。Ruby でメソッドを定義する際には、定義の外側にあるローカル変数を参照することができない。よってメソッドを普通に定義してもクロージャにはできない。一方で、`Proc` クラスのオブジェクトおよび block は、自分の外側にあるローカル変数を参照することができるため、容易にクロージャのように使用することが可能である。

例えば、次のようなプログラムを考える。

```
def closure( &block )
  local_var = "hoge"
  block.call()
end

def run()
  local_var = "fuga"
  closure { puts local_var }
end

run()
```

`closure` メソッドに渡されている block は、自分が定義された `run` メソッドの中で変数を探そうとするので、このプログラムを実行すると `fuga` が出力される。

また Method オブジェクトはレシーバと不可分の存在であるということも特徴的だろう。例えば Method#receiver[*3] というメソッドがあり、これは Method オブジェクトの対応するメソッドのレシーバを返すようなメソッドである。

かと言って Method オブジェクトと proc の間になんの関連もないのかというとそうでもない。Method#to_proc メソッドというものを用いると、Method オブジェクトを proc に変換できるためである。

1.6 Symbol#to_proc メソッド

Method#to_proc メソッドを前項で紹介したが、Symbol#to_proc メソッドというものも存在する。これは Ruby1.8.7 以降で追加されたメソッドである。Symbol#to_proc メソッドの使用例を示す。

```
["aaa", "b", "cc"].map(&:length)
```

注目するべきなのは map の引数である。map の引数としては本来 block が期待される。実際に渡されているのは&:length というものなので、:length という proc にあたるものに&をつけることで、block とみなして渡していることがわかる。しかし、実際には:length はシンボルであるので、暗黙的に proc への変換が行われる。この暗黙の変換の時に呼ばれるのが Symbol#to_proc である。これが呼び出されることで、シンボルが表している手続きが proc に変換されることになる。つまり、次のようなプログラムと同じような挙動をするといえる。

```
["aaa", "b", "cc"].map{|w| w.length}
```

このプログラムのような block の引数に対して一度だけメソッドを呼び出すような block のことをワンコールブロックと呼ぶ。Symbol#to_proc メソッドを用いることで、ワンコールブロックを簡潔に記述することができる。

1.7 Fiber

これまでに述べてきたものとは少々毛色が違う手続きの抽象化として、Fiber を取り上げたい。Fiber はコルーチンと呼ばれる仕組みを Ruby に取り入れるために導入されたものである。コルーチンとは、一旦中断したあと、その中断した場所から再開できるような機能が付加されている手続きの塊のことである。コルーチンを利用することで、複数の手続きが協調して動作するようなプログラムを記述することができる。

複数の手続きが併存しているようなプログラムを記述する方法として、スレッドというものも存在する。スレッドは OS からの支援が得られるため、並列な処理を実装するとき、最もよく用いられる手法である。一方でスレッドの実行の切り替えは、ユーザ側が指定できず、OS や処理系の実装に依存している。このような方式をプリエンプティブと呼ぶ。

[*3] Method オブジェクトの receiver メソッドという意味である

一方で Fiber は、ユーザ自身が中断や再開を決定することができる。これをノンプリエンプティブと呼ぶ。ノンプリエンプティブである利点は、切り替えを自身で決定できるため、排他制御などについて考慮する必要がなくなることである。また、切り替えのオーバーヘッドも小さくしやすい。このような利点から、Ruby のリファレンスにおいては「ノンプリエンプティブな軽量スレッド」と紹介されている。

Fiber は、並列処理を記述する以外にも使うことができる。例えば次のようなプログラムがあったとする。

```
def count(n)
  Fiber.new do
    loop do
      Fiber.yield n
      n += 1
    end
  end
end

c = count(0)
c.resume
c.resume
```

c = count とすることで変数 c に count メソッドで定義された Fiber が代入される。Fiber は Fiber#resume メソッドを呼び出すことで実行できる。Fiber は proc などと同じように実行されるが、Fiber.yield メソッドを呼び出すと、実行を中断して元の場所に戻る。その後 Fiber#resume メソッドを呼び出すと、再びその場所から実行が再開される。

このような値の列を 1 つずつ生成するような手続きのことをジェネレータと呼ぶ。配列やリストを返すような関数と異なり、ジェネレータは一度に一つの値しか返さないので、占有するメモリを減らすことができる[*4]。

[*4] ちなみに、筆者はジェネレータを改良したようなものを言語処理系に導入することで、特定の問題を解く際の計算量を減らすという研究をしている

第 2 章

PostgreSQL で単回帰分析

hayakawa

昨今、ビッグデータやデータサイエンティストと呼ばれる職業が注目されてきている。その動きの中で、統計処理を得意とする R 言語が注目されている。この言語を用いることによって、高度な統計的手法を容易に実践することが可能になっている。しかしながら、R 言語は、データを csv ファイルもしくはデータベースなどから取得してから、基本的にはオンメモリで利用しなければならない。もし、簡易的な処理を行うだけであれば、SQL などのクエリ言語のみで完結させたい場合がある。例えば、算術平均や分散、n 数 (何行のデータか) を把握したい場合である。また、データベース上で、データを予め加工した状態にした方が高速に処理が行える場合がある。

今回は、統計的手法の中でも実装を簡易に行うことが出来る単回帰分析を PostgreSQL で行う。

2.1 データの準備をする

R 言語に組み込まれてるデータセットである cars を利用する。R 言語に組み込まれているデータを利用するならば、データベースを利用する意味が無いなどと言わないでください。(汗

2.1.1 データの準備

R で以下を実行する。

```
data(cars)
write.csv(cars, file="cars.csv", row.names=F)
```

で cars のデータを csv ファイルに出来る。このデータセットには、

speed(車の速度)
dist(停止距離) の 2 変数がある。

2.1.2　PostgreSQL にデータをインポートする

まずは、table の作成を行う。

```
create table cars(
speed numeric,
dist numeric);
```

次に、csv ファイルからデータをインポートする。

```
copy cars from '/home/hayakawa/cars.csv' with csv header;
```

これで、データをインポートすることが出来る。確認のために、

```
select * from cars;
```

などと実行すると良いかもしれない。

2.2　単回帰分析

2.2.1　どんなものか

回帰式は、y = ax+b という形で表現される。x は説明変数で、入力データである。y は目的変数で求めたい値である。a は回帰係数、b は切片である。a や b を推定するには、最小二乗法という手法を用いる。これは、当てはまりが一番良くなるように a や b を推定する方法である。次の節より、SQL を用いて、a,b, 決定係数 (当てはまり具合を示す) を求める。

2.2.2　直線の傾きを求める

```
select covar_samp(speed, dist)/variance(speed) from cars;
```

2.2.3　y 切片を求める

```
select avg(dist) - covar_samp(speed, dist)/variance(speed) * avg(speed)
    from cars;
```

2.2.4　まとめる

係数と切片を求める

```
select covar_samp(speed, dist)/variance(speed)  as coef_speed,
avg(dist) - covar_samp(speed, dist)/variance(speed) * avg(speed)   as
    intercept
from cars;
```

2.2.5 決定係数を求める

```
select
sum(power(speed*coef_speed + intercept - avg_dist,2)) /
sum(power(dist - avg_dist,2)) as r_squared,
avg(coef_speed) as coef_speed,
avg(intercept) as intercept
from
cars
cross join
(
select covar_samp(speed, dist)/variance(speed)  as coef_speed,
avg(dist) - covar_samp(speed, dist)/variance(speed) * avg(speed)   as
    intercept,
avg(dist) as avg_dist
from cars
) as lm
;
```

出力結果は、

```
     r_squared     |    coef_speed    |    intercept
-------------------+------------------+------------------
 0.651079380758251 | 3.93240875912409 | -17.579094890511
```

となる。

2.2.6 結果

これで、dist = 3.9 * speed - 17.6 という式を得ることが出来た。決定係数は 0.65 で 65% の当てはまり具合である。この値が大きいほど、精度の高いことを意味する。

2.3 感想

SQL で単回帰分析を行う事は面倒だと思っていたが、想像していたよりは利便性が高そうである。機会を見つけて利用したい。

第 3 章

やはり俺の irb シェル化計画はまちがっている。

KA ☆ SI ☆ O

3.1 夏休みが終わったのにこの猛暑はどう考えてもおかしい。

10 月、夏休みが終わり後学期が始まった。つまるところ最高の夏が終わった。

思えば夏休み中ほとんどガストにいた記憶しかない。やはり真のプロファミ[*1]はガストを好むと思う。サイゼも俺の縄張りであることに変わりはないのだが、如何せん駅の南にあるサイゼは客層が悪い。なんかコンビニの書籍コーナーで血液型の説明書とか買っちゃいそうな頭の悪い人種の集会所になっている。それにサイゼだとどう頑張ってもミラノ風ドリアとドリンクバーの組み合わせが価格最安で 400 円は必要になる。その点ガストは違う。初回こそオトクーポンの半額プレミアムカフェと 313 円のポテトフライで 450 円ほど掛かるが、会計時に貰えるプレミアムカフェ無料クーポン (一週間の期限付き）を使用すれば次回からは 313 円で利用することができる。しかもこの無料クーポンはクーポン使用時にも貰えるので週 1 で通えば半永久的にリサイクルできるのだ。立地も国道沿いで夜はあまり人が来ない。そして 24 時間営業。

別に好きでファミレスを多用していたわけではない。下宿の電気代が 10k を超えてしまったため、涼を取りながら勉強するスペースを模索した結果だ。日中は大学図書館で過ごしているが、17 時で閉館してしまうためその後は居場所がないのだ。え？トモダチ？ナニソレ？どこの新興宗教の教祖だよ。それとも俺と二人でいるとすげー静かなのに、他の友達がいるときは笑顔がすげー眩しかった吉田のことか？あれまじなんなの…？二重人格？予備校にいるときの俺かよ。部室にはクーラーがあるので部室が開いていればよかったのだが、8 月末からサークル棟の耐震工事で使用できない状態になっていた。

[*1] プロひとりファミレスラーの略

3.2　きっとどこにでも異形の花は咲いている。

部室、というのはこの俺、比企谷八幡が所属する胞子部の部室のことである。胞子部は主に子実体[*2]を形成するタイプの菌類全般について研究するサークルだ。もともとはフィールドワークを積極的に行うアウトドア派のサークルだったのだが、ここ十数年で計算機を導入し、シミュレーションとデータ分析を用いた手法に傾倒した結果、今やもはやただのプログラミングサークルになっている。まあ、ほらあれだ。あるだろ、x68k 向けに開発しないゲーム同好会とか、ライブやってるだけのシンセ研究部とか、マイコン使わないマイコンサークルとか。あ、最後のは花火サークルでした。

今現在、俺が 2 限講義出席確認後速攻退出 TA をキメて向かっているのもその部室である。ちなみに TA とはなんか講義中に後ろに座ってるだけの給料泥棒という意味ではなく Time Attack のことである。この時間は必修のプログラミング演習なのだが大学の計算機室のような酸素の薄い空間でわざわざ解く必要もないだろう。

部室前に着くと、ドアの擦りガラスの向こうから明かりが零れていた。既に先客がいるようだ。ノブを回し、ドアを開けるとそこには見慣れた光景が広がっていた。入って右側には長机が数台壁付けされ、その上に”山”で採取された端末とモニタが並べられている。左側には事務棚が数台あり、図鑑やオライリー[*3]などが収納されている。その部室の最奥に雪ノ下雪乃はいた。雪ノ下は俺に気づき一瞥するがすぐに作業、もといきのこの盆栽作りに意識を傾ける。今取り組んでいるのは電子機器と菌類の融合らしく、中古のリアルフォースのキーの隙間から何やら紫色の物体が顔を覗かせている。にしてもなんというか、夏の間掃除していなかったからか埃なのか胞子なのかわからないものが宙を舞っていてやばい。おいこれスエヒロタケ[*4]とかじゃないだろうな。

「なんかこの部屋全体的にカビっぽくねーか、ファボリーズとか撒いて除菌した方がいいんじゃねーの？」

「ダメよ、そんなことしたらこの子達や比企谷君が死滅してしまうじゃない。」

「おい、さらりと俺を菌類にカテゴライズするな。」

子実体は間違いなく単生である。分類的にマツタケとかと同じ、レアキャラだ。保健室の常連じゃないと会えない感じの奴。

「あら、違うの？じゃあ冬人夏草…、いえ冬虫夏草[*5]とか？」

「なんでわざわざ虫の方で言い直しちゃったの？人であってるよ？俺には五分の魂しかないの？」

「じゃあ年中クズ？」

それもはやただのダメな人だから。でも人として認識してもらえてよかった。

特に訂正しないまま、俺はファボリーズをカーテンに吹き付け窓を開けた。日本のシュッシュを世界レベルへ。室内の空気が一頻り浄化された後、俺は窓を閉めてエアコンの電源を入れて冷房

[*2] しじつたい。ようするにきのこ

[*3] 攻撃力が高そうな分厚い技術書を発行する出版社。およびその出版社から発行された図書。

[*4] 人間の体内から生えたことで有名なきのこ

[*5] 虫に寄生するタイプのきのこ。冬の間は虫。夏はきのこになるので冬虫夏草と呼ばれる。

設定にした。それから特に雪ノ下と言葉を交わすこともなく、いつものように廊下側の一番端の席に座り鞄からラップトップを取り出し現実からログアウトした。
ちなみに今の内に断っておくと、この SS に由比ヶ浜は登場しない。やっはろーは死んだ、時代はにゃんぱす〜。

胞子部にはこれをしなければいけないという活動らしい活動というものは特にない。講義の空き時間や昼飯時に訪れて適当にラップトップを叩きつつ 2L ペットボトルでお茶を飲むというのが普段の俺の過ごし方だ。2L ペットボトルを持ち歩いてると奇異の目で見られることが多いんだけどなんなの。経済的にも優しいし飲み終わって空いたボトルは筋トレやタヌキモ[*6]の栽培にも利用できて超便利なんだけどな。学内で携帯しても捕まらない理想的な武器なんだよ？

そんな胞子部であるが、年に一度部員強制参加の活動がある。名を同定祭といい、秋の文化祭に合わせて胞子部が開催するイベントである、もちろん略称は DT 祭だ。内容は、近隣の愛好家や胞子部 OB できのこを持ち寄って同定した後、可食なものを OB 参加の鍋会で美味しくいただくというものだ。ちなみに OB はオールドボーイ、オーバーボーイではない。持ち込まれたブツには明らかに毒々しい色をしたものも含まれており、現役部員の DT 力を鍛えようという OB の熱意が込められている。ちなみに自身の研究のためにあえて図鑑で食毒不明となっているものを持ち込む危険な輩もいるようだ。ていうかそれが盛られていたのは俺の皿で盛ったのは雪ノ下だった。お前何やってんだよ。しかも食毒不明だが近縁種はすべて毒という種をわざわざチョイスしてきているあたり確信犯である。まあ俺ははがねタイプで雪ノ下はどくタイプなので毒は無効である。

で、話戻るがこの鍋の鍋敷きに使用するために発行されるのが通称百幡石という部誌だ。
内容は部員のコラム集のようなものであり、テーマは各自自由となっている。鍋の方は OB に任せきりとなっている現在、この部誌を書くのが現在の俺のタスクというわけだ。俺は未だ決まらぬ記事のテーマに頭を悩ませる振りをしつつ孵化厳選をしていた。ポ○モンなど数年ぶりにやったがグラフィックの進化には驚く。んん wwwwwww 男主人公はありえない wwwwwwwwww

3.3　誰だってポイントフリーという病を通過点としている。

どれだけ時間が経ったのだろうか、チラと左手に止めた腕時計を見るとお昼の 1 時を回り 3 限が始まったところであった。腹は減っているのだが昼間のらんちたいむ[*7]の学生弁当は作り置きだからな…。もう夕方まで待つか。その時、後方から視線を感じた俺は思わず振り返った。爆発した。いつの間にか背後に雪ノ下が移動している。俺は 3DS のスライドパッドにパンの袋閉じる奴（名前知らん）を差し込み、椅子ごと体を雪ノ下の方へ向けた。いや、これ考えた奴まじで神だろ。
「どした？」
「依頼が入ったわ、今月から始めた例のアレよ。」
どうやら視線で進捗確認していたわけではないようだ。
依頼というのは、俺達の新たな活動である学内横断お悩みメールに寄せられた相談である。持つ

[*6] 水生の食虫植物。主にヤフオクに生息している。
[*7] UDP を採用している弁当屋

ものが持たざるものに慈悲の心をもってこれを与える。人はそれをディプログラミングという。Emacs で小指を痛めた人には Vim を、副作用に苦しむものには純粋関数型言語を。

本日の相談内容、<<PN:さがみんさんのお悩み >>

『必修の FP の fizzbuzz という問題が分かりません。解き方を教えてください、ソースコードの例とかもあるとうれしいです。』

fizzbuzz というのはプログラミングの初歩問題だな。書道でいうとこの永字八法。ドス Hello, World のようなものだと思っておけばいい。簡単にルールを説明すると、

1. 1 から 100 までの自然数を順番に出力していく。
2. ただし 3 で割り切れるものは数値の代わりに fizz、5 で割り切れるものついては buzz を出力する。両方で割り切れるものは fizzbuzz と出力する。

みたいな感じだ。FP というのは間違いなく Functional Programming のことなので Lisp や Haskell で解くべきであろう。俺はこんな感じのコードを書き始めた。

```
solve = map fizzbuzz [1..100]

fizzbuzz :: Int -> String
fizzbuzz i
  | divisible15 i = "fizzbuzz"
  | divisible3  i = "fizz"
  | divisible5  i = "buzz"
  | otherwise     = show i
  where divisible3  = divisibleN 3
        divisible5  = divisibleN 5
        divisible15 = divisibleN 15
```

こいつが動くように divisibleN を定義すればいい。割りきれるかどうかの判定だがこれには剰余演算を用いる。剰余演算っていうのは割った余りを返す演算だ。

例えば偶数人のクラスで一人欠席者が出ている状態で二人組を作ると一人余る。それが俺だ。ちなみに偶数人の場合でもなぜか三人組ができていて俺が余る。

割った余りが 0 ならそれは綺麗に割り切れるということになる。Haskell には mod という関数があるのでこれを利用すれば簡単に divisibleN が実装できるだろう。

「なんだか酷く冗長な定義に見えるのだけれど、こんな感じになるのかしらね。」

雪ノ下がホワイトボードに続きを書いていく。

```
divisibleN n m = m `mod` n == 0
```

「まあそんな感じだな。だが Haskell 的に面白みのないコードだ、この定義はポイントフリーになっていない。」

「ドヤ顔で解説したそうな比企谷君にあえて聞いてあげるのだけれど、ポイントフリーとはなんのことなの？」

え、俺今そんな顔してた？やめろよ同級生と話してるのになんかカウンセリングみたいになっちゃってるだろ。お前今日のヒキタニ君係なの？

ポイントフリーというのは引数が明示されていないタイプの関数定義のやり方だ。雪ノ下の

divisibleN の定義では引数の m,n が付いているのでポイントフリーではないというわけだ。ちなみにポイントフリーではない形式はポイントワイズという。
「ふうん。大体分かったわ。けれど、ポイントフリーにすることで一体どんなメリットがあるのかしら。」
「いや、これといってない。一部の関数はポイントフリースタイルで定義することによって本質が浮き彫りになることもあるが、そもそもポイントワイズで書けるものをあえてフリーで書き直すってのは一種の遊びみたいなものだな。」
「意味はないのね…コードゴルフみたいな感じかしら。ちなみにこの関数をポイントフリーで書き直すとどうなるのかしら。」
ふむ、こういった無意味な変形も Haskell の醍醐味の一つではあるので俺は試しに取り組んでみることにした。ラムダ式を用いる方法が一番てっとり早いがそれではあんまりだろう。まず、俺はこの関数で二項演算子として使われている関数 (mod と ==) を通常の関数の表記に直すことにした。

```
divisibleN n m = (==0) (mod m n)
```

次に、引数を取り除くために、定義の右辺で使われている m,n の並び順を引数の並び順に合わせる必要がある。これには flip :: (a -> b -> c) -> b -> a -> c を使えばいいだろう。flip は受け取った関数の第一引数と第二引数の順序を入れ替えた関数を返す関数だ。

```
divisibleN n m = (==0) (flip mod n m)
```

ここまではいいだろう。最終的に定義の右辺が何かしらの関数 f を用いて f n m の形になれば n と m が消去できるので、f を作るために (==0) と flip mod を合成する必要がある。試しに俺は

```
divisibleN = (==0).(flip mod)
```

のような安直なコードを書いてみた。結果はエラー。どうやら型が問題なようだ。(==0) と flip mod の間に挟まっている. は関数合成演算子である。つまるところ

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f(g(x))
```

である。(.) は (b -> c) 型の関数と (a -> b) 型の関数を受け取って合成して (a -> c) 型の関数を返す。それに対して、(==0) と flip mod の型は次のようになる。

```
(==0) :: (Eq a, Num a) => a -> Bool
flip mod :: Integral c => c -> c -> c
```

問題は flip mod である。こいつは二引数関数だ。Haskell における二引数関数というのはカリー化された関数のことなので厳密には flip mod の型は c -> (c -> c) となる。flip mod の返す値の型は (c -> c)、それに対して (==0) の引数の型は数値でなければならないのでこの 2 つはどう足掻いても合成できない。(==0) をどうにかして関数を受け取る形に変形する必要がある。
一般化すると二引数関数 bi

```
bi :: a -> (b -> c)
```

に対して一引数関数 mono が存在する。

```
mono :: c -> d
```

今、なんらかの関数 g を使って mono を次のように変換したい。

```
g mono :: (b -> c) -> e
```

g mono と bi を合成すると型は a -> e になる。最終的に全体が二引数関数になるので e には一引数関数が入る。そして e が b を受け取る関数であることは自明であり。e の返り値の型は mono の返り値の型と同じ d になる。よって g の型は

```
g :: (c -> d) -> (b -> c) -> (b -> d)
```

となる。この g を探せばいい。俺は Hoogle に型をそのまま入力して同じ型の関数がないか検索した。
それでヒットしたのがこちらの関数になります。

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

なんと同じ型を持っているのは関数合成である。ためしに g として (.) を使用してみた。

```
divisibleN = ((==0).).(flip mod)
```

なんか知らんが、これでうまく動作した。このように Haskell は型を合わせてコンパイルが通れば大体うまくいく、オールオッケー。これでポイントフリー化ができた。

　というわけでこの divisibleN 関数の定義を送りつけて今回の依頼は完了となる。魚の釣り方は教えずに魚だけを投げつけておくというのが俺のスタイルだ。偉い人は言いました。教えるは一時の優越感、教えないは一生の優越感。
「おい、できたぞ雪ノ下。」
「あら、まだやっていたの？既に Java で解き終えて回答するところなのだけれど。」
なん…だと？
「Java はよく謂れのない非難を受けているけど、30 億のデバイスで走っている実績があるわ。エラーメッセージも読みやすいし、初心者にはこちらの方が優しいでしょう。」
try、catch、finally 俺は死んだ。
「まあ、ソースは出さないほうがいいんでねーの？この質問者写したの提出する気まんまんだぞ多分。」
「それもそうね…」

　<< 胞子部からの回答 >>
Scala で解いて jdc でデコンパイルするといいと思います。

一応 scalac の吐いた class ファイルも添付しておいた。俺超優しい。

　一仕事終えた俺は生茶をラッパ飲みしつつ作業へと戻った。またしても綾鷹は選ばれない。こんな感じで胞子部の一日は過ぎて行く。思えば入部して一年半、この空間で日常生活におよそ必

要無い知識を蓄え続けている。俺はスクリーンセーバーの生み出すフラクタル模様をぼんやりと眺めながら入部当時のことを思い返していた。

3.4　だから俺はスターシステムを支配する。

暗黒の高校生活から脱した俺は自分の学力よりワンランク上の某国立大学に入学した。新たな季節、新たな生活、新たな学校。変わるのはいつも環境ばかりで俺自身は何一つ変わらないが。4月の時点ではここでなら俺は変われるなどという柄にもない淡い期待に胸を膨らませていた。だがあえて学友を作ることはしなかった。これは俺のコミュニケーション能力の欠如によるものではないと一言添えておく。オリエンテーション、入学式、ガイダンスを兼ねた新入生合宿、友達作りのチャンスはいくらでもあっただろう。だがあえてしなかった。重要なことなので二回言いました。大体偶然居合わせただけの見知らぬ奴と仲良くなろうって発想がどうしようもなく不自然で気持ち悪いんだよ、お前ら電車の中で友達作るの？作らないだろ？まあそんなとこで友達作ってもその後は一度も連絡取ることもないだろうし、講義とかで偶然居合わせると気まずい空気になるだけだ。ソースは俺…の知り合い、電話帳に残ってるメアド交換してそれっきりの奴の連絡先とか見つけるとフラッシュバックで吐きそうになる。らしい。ともかく運命力も必然力も俺は信じない。

となると残る人との出会いの機会はサークルくらいだ。
いくつか趣味嗜好の合いそうなサークルに顔を出してみたが、どうにも（俺のようなぼっちにも分け隔たりなく親切に接してくれるような）ピンとくるサークルがなかった。別に新歓の席で誰にも相手にされなかったからトイレに行く振りして帰ったとかそういうわけではないんだからね？うん、あのサークルは新年度に 1 人だけ”いないもの”を作らなきゃいけない風習とかそういうのがあったんだよ多分。今頃部員総勢俺に感謝しているに違いない。あ、でもその後フェードアウトしたから今頃全滅ですね分かります。

まあ新入部員の中に一人だけアレな奴がいるとその年の新入部員全員が不仲になりかねないからなあ、ソースは俺。いやまあ、仮に学年みんな仲良くしてても俺はその輪の中にいないからどうでもいいんだけどね？誰しもが俺といるとありのままの姿をさらけ出さずにはいられない、なんというラーの鏡。なんでいじられキャラって俺がいじった時だけ本気で嫌そうな顔するんだろうな本当。

まあ別段サークル活動自体に興味があったわけではない、サークルに入って得られるであろう恩恵が目当てだった。恩恵と書いてギフトと読む。具体的には過去問とか、お昼の居場所とか。そんな不純な動機でサークルを選別していた俺にとって胞子部はまさにベストな選択であった。カードキーで常時入室可能。気を使うべき先輩は全員 B4 であまり部室に顔を出さない。過去問こそあまり残されて無いがなぜかオライリーは揃っている。その年の新入部は雪ノ下と材木…ドス生肉だけだった。ちな今年の入部希望者は話の都合により 0。

俺は基本的にコミュニティの人口も、ギャグ漫画の登場人物も少ないに越したことはないと思う。キルミーとかやばいだろ、ほとんど二人だけで話回してて面白いし。いや、面白いよね？そ

れに比べて某 λ カ娘の軟弱ぶりときたら見てて寒々しい。ギャグ漫画はキャラではなくギャグで笑わせなければいけない。安易な新キャラ投入など持ってのほかだ。新キャラってすごく扱い辛いと思うんだけどな。それまでうまくやっていたコミュニティの中に一人だけぼっちが参入するわけで。読者としてはぽっとでの新キャラがレギュラーメンバーにタメ口なんてきいた日には激おこである。空気の読めない新人芸人のようなものだ。おいそこ、ドキプリの赤の悪口はやめろ。ただしゲストキャラが多いギャグ漫画は面白い。これは間違いない。

結局漫画でもなんでもキャラクターが少なければ少ないほど面白いんだ。つまるところぼっちの俺は面白くて馴れ合ってる糞共には微塵のコンテンツもない。俺こそがエンターテイメントの源流、ミスターオリジン。

あまりのキルミーの面白さに負の感情に捕らわれてはっと目が覚めた。キルミー危険だな、規制しろ。窓の外を見るとすっかり日は暮れていた。どうやらあれから寝てしまったらしい、時計を確認するともう 18 時だった。らんちたいむを食べ逃してしまった。雪ノ下はまだ何やら作業をしている。

「あら、起きたのね。おはようハス谷君。」

「誰だよハス谷君、それだと俺副作用禁止おじさんみたいになっちゃうだろ。」

「確かに…あなたちっとも純粋じゃないし、目の中なんて参照透明とはとても言えないくらい濁りきってるものね。訂正するわ、unsafePerformIO 谷君?」

いや、それもはや原型ないから。谷取ったらただの副作用だから。

「ばっかお前、俺めっちゃ純粋だぞ。どれくらい純粋かっていうとな。話せば長くなるんだが。」

俺が小学校 4 年の時の話をしよう。俺の小学校では学年が偶数の時に学芸会、奇数の時に展覧会を行っていたんだ。でそのときは学芸会が開かれたわけだ。俺は公立だったし他の小学校のことはよく知らないが俺の学校の学芸会は所謂演劇的なものだった。当然演劇なのでクラス内で配役を決める。クラス内ヒエラルキー。誰が決めたわけでもなくいつの間にか作り上げられる 1 軍・2 軍・3 軍。こういう行事ものでは決まってそうだが、クラスで一番人気の男子と女子が主役 or ヒロインを務め、あとはまあそれぞれのカーストごとに適当な役が割り振られていくものだ。

「話はもう読めたわ。あなたはどうせ背景の木役か村人 A あたりに収まったのでしょう?」

「いや、それが違うんだなあ。話はそう簡単でもない。」

俺に与えられたのは役ではなく仕事、舞台音響だ。場面ごとの BGM の選曲と劇進行時の音響装置の操作の両方を任された。まあ所謂裏方さんであるが、音楽というのはエンターテイメントにとって非常に重要な要素であるということは認識してたので、俺は大役を任されたと素直に喜んだ。今思うと吐き気がするほどの純粋さである。そう、もう聡明な真のぼっちの諸君はお気付きであるだろうが、俺のこの配置はクラスで役決めをする前から職員会議での決定事項だった。つまり俺は予め”問題を起こす可能性のある生徒”としてキャストの外へ排除されていたのだ。うおああああああああああああああああああああ。思い出したらまじで死にたくなってきた。殺して！!!! 誰かあのころの純粋な僕を殺して!!!！

「それはなかなかになかなかね、クラスの不和の原因であるところの比企谷君を未然に処理した担任の優秀さを賛辞すべきなのかしら。ちなみに私も音楽担当だったけれどもピアノの伴奏だった

から少し意味合いが異なるかもしれないわね。」
確かに、技術を要するか否かで状況が全然違って見える。俺はもう二度と技術職とそれ以外の仕事の積集合の仕事にしか就かないと決めた。働いたら、死あるのみである。

3.5　ようやく進捗確認と共に物語は始まる。

「話は変わるけれど、記事の進捗はどうかしら。締切りまでまだ 4 週間ほどあるけれど。」
「進捗？ダメだよねえ。前編後編新編、すべて最高だったと思うよ。でも、記事は一文字も書いてないしテーマも決まってないよ。」
「確か、夏休み前に Haskell を用いてシェルを作成したいみたいなことを言ってなかったかしら？自分はシェルになりたいとまで抜かしていたわよね？」
「あーそうだったな、バグが、バグ達が跳梁跋扈する俺のプログラムを、皆さんに見せたいね。」
「ま、ようするにあれだ。trifecta[*8]で構文解析まで書いたとこで飽きた。あと一ヶ月で完成させるのは無理だな。」そもそも何か作ってみた系で記事書いてもあまり面白くないんだよな…。殊更情報系の内容は実物があるわけでもないので地味だ。
「と、なると。途中経過までで記事にするか、プロジェクトを縮小させて完成させるかね。」
あれれ〜？おっかしいぞー？書かないっていう選択肢が表示されていない。
「今から再開させるとして Haskell はないな、もう暫く Haskell 書いてないし。ていうかさっき三カ月ぶりくらいに Haskell 書いたわー。三カ月ぶりに書いたわー。」
やっぱ Haskell って糞だわ。
「まず言語選びからということね、比企谷君は Haskell 以外だと何が使えるの?」
「まあメジャーどころは押さえている、よく使うのは C, Java, Scala, Ruby あたりだな。普段あまり書かないが Perl や JavaScript も大分無理すれば書けないこともない。」
「後半のそれは書けないというのよ。いるわよね、見栄はって一度しか使ったことのない言語を経験言語に挙げる人。」
おい、こっち見ていうな。なんで俺の履歴書の特技欄知ってるんだよ。ちなみにどんなに使えても PHP は書かない。
「今挙げた中で一番使えるのは Ruby だな、あとは大体同程度だ。」
「C 系にあまり自信がないのなら既存のシェルを拡張して構文を追加するのは難しそうね。」
「シェルってほとんど C なのか。探せば色々ありそうだけどな。」
とりあえず俺は wikipedia でシェルを検索する。案の定、そこにはシェルの一覧の項が設けられていた。ふむふむ。昔からある Unix シェルは当然の如く C だな。しかし Scheme で書かれた scsh なんてのもある。WM やデスクトップ環境もシェルとして挙げられている、俺は無意識にシェル=テキストシェルに限定していたがよく考えればこれらもシェルだな。俺が普段一番よく使うシェルである。同点で zsh と ssh。
「Ruby で書かれたシェルは載ってなさそうね…。」

[*8] Haskell のパーサコンビネータライブラリ

「ぽいな、正直 Ruby で一から作りたいと思わないし。」
念のために俺はページ内を Ruby で検索した。すると一件だけ Ruby で引っかかる行が見つかる。irb、それがそのシェルの名前だった。

3.6　けれど彼には DT 力が足りない。

「比企谷君、私は Ruby に関しては全く知らないのだけれど。この irb というのは本当にシェルなのかしら?irb…という名前から連想するに irc の親戚か何か？」
いや、RFC1459 は関係ないから。断じて関係ない。だからもうその名を口にするんじゃあない。
「irb は Ruby の対話型環境だよ。python の対話モードとか Haskell の ghci みたいなもんだ。」
「じゃあシェルではないのね。シェルのような物体といったところかしら。」
その言い方だとラスボスの八面体しか想像できない。強そう（確信）。
「ああ、まあシェルライクなインターフェースが利用できるという点で irb を下敷きにシェルを作ることは可能だ。いやまてよ…。」
おれは天啓を得た。この考えがうまくいけば、ほとんど労せずに Ruby 対話型インタプリタとしても使えるシェルが誕生することになる。
「とりあえず、irb を利用して ruby で作るか。」
俺はおもむろに terminal で vim を立ち上げた。
「まず、Ruby からシステムのコマンドを呼び出す方法についてだが。3 つ方法がある。Kernel.#system, Kernel.#‘, Kernel.#exec だ。」
「Kernel……というのはクラスの名前かしら。.# というのはなんなの？」
「Kernel はモジュールだな。Java でいうところの static class と abstract class を融合させたようなものなんだが、まあ一言で説明するのは難しい。モジュール名の後ろの記号については大体次のような感じになっている。」

C#method #method は C のインスタンスメソッド
C.method #method は C のクラスメソッド
C.#method #method は C のモジュール関数

この表記すげー便利なんだけどなあ、Ruby 界隈でしか使われているのみたことないんだが。
「Kernel.#‘と Kernerl.#exec の 2 つについてだが、Kenerl.#‘は ‘ls -l‘のようなリテラル形式で引数を取る。受け取ったコマンド文字列の実行結果が文字列として返る。よって呼び出しがすぐに返らないコマンド、エディタなんかの起動には使えない。Kernel.#exec は現在のプロセス上でコマンドを実行するので irb が死んでしまうから使えない。残った Kernel.#system は渡されたコマンド文字列をサブシェル上で実行する。よって今回はこの system 関数を使う。」
「irb が取得した入力に対してこの関数を適用すればいいのね、システムコマンドと Ruby の構文の区別はどう付けるの?」

「いや、そもそも俺は今回 irb の元ソースに対して一切手を付けるつもりはないぞ。必要なことはすべて irb の設定ファイルである.irbrc に書く。」
「設定の変更だけで目的が達成できるの？」
雪ノ下が疑問に思うのも当然だ、しかし Ruby に不可能はない。
「.irbrc は ruby のソースファイルそのものだからな。irb のソースに手を加える代わりに.irbrc に書いて実行時に irb の挙動を変更する形で実装していく。」
「驚いたわ……そんなことが可能なのね。その.irbrc にシステムのコマンドを 1 つ 1 つ定義していくのかしら。普段使用するものだけでも膨大な数になりそうなのだけれど。」
「いや、実装すべきメソッドは 1 つでいい。method_missing だ。」
method_missing というのは存在しないメソッドが呼ばれた時に呼ばれるメソッドである。通常時はこいつが呼ばれると例外が投げられ、メソッド探索が失敗する。しかし、method_missing は通常のメソッドと同様にオーバーライドできる。つまり method_missing 上で system を呼び出せば。任意のコマンド名に対して一度にメソッドを定義できるのだ。method_missing の引数はメソッド名のシンボルと与えられた引数リストになる。まあとりあえず書いてみよう。

```
def method_missing(name,*args)
  super if name.to_s =~ /^to_/
  system(name.to_s)
end
```

先頭が to_で始まるメソッドは super でオーバーライド前の method_missing に投げている。わずか 4 行で書けてしまった。またしても勝利してしまったか…。ためしに irb を起動して ps やら ls やらをタイプして見るとコマンドの実行結果が表示された。vim とかも問題なく起動できるようだ。
「ここまでは狙い通りだな。問題はオプションをどうするかだ。」
「普通に引数として渡すのではだめなの？ system 関数は任意の数の引数が取れるようだけれど。」
「オプションが 1 つならそれでもいいんだけどな。2 つ以上あるとカンマで区切って渡す必要がある。構文はできる限り実際のシェルに近づけたい。」
「.irbrc から irb が取得した入力行に対しては干渉できるのよね？だったら構文そのものを書き換えることも可能なんじゃないかしら。」
「いや、それはダメだ、アンフェアすぎる。」
なぜならそれをやると、もう irb のソースいじればよくね？ってことになり虚しさが半端ないからな。それは禁止事項にしておこう。
とはいえ、この問題に対処するのはなかなか至難の技である。妥協案として

```
cmd %{opt1 opt2 opt3 ... }
```

みたいな書き方もできるがこれは最後の手段にしたい。
「まあ前提として、オプション自体も method_missing で呼ぶ必要があるな。」
「あくまでも裸のワードに拘るというならそれしかないのでしょうね。」
裸のワードって何それエロい。Ruby には裸のワードが 4 種類ある。変数名、定数名、メソッド

名、キーワードである。この内実行時にあれこれできるのは method_missing のあるメソッド名と const_missing のある定数名だけだ。Ruby で裸のワードを並べた場合。メソッドと変数、両方の可能性で探索が行われる。今は method_missing をオーバーライドした状態なので

```
cmd opt1 opt2 opt3
```

というワードの並びは

```
cmd (opt1 (opt2 opt3))
```

として解釈される。オプションが 3 つの場合、評価の順は opt3,opt2,opt1,cmd となる。
「末端から評価が始まることに注意して method_missing でコマンド文字列を構成していけばよさそうね。」
「だな。」
俺はコード、もとい.irbrc を書き始めた。

```
def method_missing(name,args=nil,*rest)
  super if name.to_s =~ /^to_/
  args ? name.to_s + " " + args : name.to_s
end
```

これで cmd opt1 opt2 opt3 という行を与えれば”cmd opt1 opt2 opt3”が返る。しかし…。
「この定義だとどこで system 関数をはさめばいいの分からないな…。」
system は存在しないコマンドを与えた時は何もせずに nil を返す。だから method_missing の呼び出しごとに system を試すというのもありなんだが…。getent passwd みたいなコマンドを呼び出すと、まず passwd が実行されて次に getent passwd が実行される。これは望んでいる挙動ではない。
時刻は既に夜の 20 時を回っていた、腹も減りに減り果てた。続きは飯でも食いながら考えることにしよう。とりあえず今日はここで解散となった。

3.7　つまるところ俺たちの魔法に複雑な呪文はいらない。

俺は○川駅から 1,2 分のところにあるラーメン二郎に向かっていた。二郎イエローの看板を発見し、つい加速したくなる衝動をなんとか抑える。この日は数人の並び、すぐに店に入ることができ、コールを聞かれた。
「ニンニク入れますか？」「あ、全部で」
わざわざ長い呪文を好んで使うジロリアンもいるが、俺は呪文は極力短くてシンプルなものを好む。プログラミングでも同じことだ、ヤサイニンニクアブラカラメをコールしたいなら”全部”と一言で言えばいい。フルでコールしちゃう奴は絶対 Java とか好きそう。

```
YasaiNinnikuAburaKarame yasaiNinnikuAburaKarame = new
    YasaiNinnikuAburaKarame();
```

ってタイプして悦に入ってそうなイメージ。

ちなみに「あ、全部で」の「あ、」が重要である。これは TCP というプロトコルの機能の一部である。この無意味にも思える母音で自分の声の音量を調節、相手の注意を引きつけると同時に自分の使用する音階を相手に意識させ、確実にオーダが届くようにする。別に普段人と話してないから思わず出ちゃう第一声とかではない。また二郎といえばマシマシのイメージが強く浸透しているがマシマシの通る店舗というのはそんなに多くない。少なくとも本店ではマシマシは通らないと聞く。俺のホームである○川店ではマシすら通らないことが多い。それどころかマシをコールすると目に見えてノーコールの麺バーよりトッピング量が減るくらいである。

ドン！と目の前のカウンターの上に二郎が置かれる。戦闘開始。麺、野菜とバランスよくそこそこに味わいつつ胃袋にぶち込んでいく、今日の豚は当たりな方だな、けして固すぎず、可食部が多めだ。ピッ、っと左隣の席から跳ねてきた油が俺の手の甲を刺激する。予想外の温度にたじろぐが、まあ稀によくあることなので気にしない。心頭滅却すれば火もまた涼し。気を取り直し味変えの胡椒を取ろうとしたそのとき、油が今度は俺の手と顔に連続ヒットする。アツゥイ！なんかもう飛び散る油が弾幕のようになっていた。お隣さんどんだけの勢いですすってんだよ。食事をする時は救われていないとだめなんだよ?孤独のグルメ読んでないの?ソレイジョウイケナイ。アームロックを掛けたくなる衝動を抑えつつ、どんな大豚トリプルが食ってるのかと隣の席に一瞥をくれる。そこにいたのは意外にも妙齢の女性。ていうか胞子部顧問の平塚先生だった。あ、ちなみに妙齢というのよく誤用されるが微妙な年齢という意味ではなく若くて美しい年ごろを指すらしい。後で知った。向こうに気付かれる前に店を出るため俺はペースを早める。あっという間に完食。終結 -K.O-、完勝 -PERFECT-。うん、なんというかもう二郎当分いいわー。一生分食ったわー。丼をカウンターの上に戻して飛び散った油を台拭きで拭き取り俺は席を立って店主に軽く会釈して店を出た。以上レポっす、チラシの裏すんません。

店を出て夜風に当たる。あ、二郎食いてえ。

「お、比企谷じゃないか。こんなとこで会うなんて奇遇だな。君も二郎かね。」

げ、この人食うのどんだけ早いんだよ、俺の方が 1 ロット近く猶予あったのに。

「うす」

俺は隣にいたことを悟られないよう、目を逸らして適当な対応でごまかす。

「おい、人と話す時は相手の目を見て話したまえ。失礼だろう。」

それで実際見ると、なんだその反抗的な目はみたいな反応されるんだぜ？完全に詰んでるだろ。小学校 3 年の時の担任の永井、お前だけは絶対に許さない。

俺は仕方なしに顔をそちらに向ける。

「平常時ですら濁りきっている君の眼球が呪いよりもおぞましい色になっているな…。部活で何かあったのか？話たまえ。」

眼球俺の本体だったのかよ。確かにアニメだと原作の 30 倍くらい濁ってる。

俺は今抱えている問題についてほげほげがふがふがといった感じでざっくり説明した。

「ふむ、それは難しい問題だな。文字列を構成する場合と system を呼び出す場合が前後の文脈で変わるから綺麗な関数にするのは無理だな。大域的な状態を保持してそれにアクセスするのはどうだろう。」

```
global flag = true
method_missing cmd opts = if flag then system (show cmd ++ " " ++  opts
    ); true = false else show cmd ++ " " ++ show opts
```

「Ruby のことはよく知らないので Haskell っぽい擬似コードで書かせてもらったが、flag というのがここでいう大域的な状態だな。」
なるほど、Haskell でこれが可能なのかという問題はともかく。状態によって関数?の振る舞いを変えるという発想はいい。
「あー Ruby は正格評価だからそれは無理ですね。遅延評価ならよかったんですが。」
ようするに Ruby は関数に与えられた引数をすべて評価してからでないと関数を呼び出せないのだ。
「となると、手詰まりだな…。そもそものアプローチが間違っている気がしないでもないが。」
状態の変更、ようするに呼び出しが終了したという状態が定義できればいい。呼び出しの始めならともかく終了か…。
「しかし、学生は時間を持て余していていいな。無駄なことに打ち込めるというのは素晴らしいことだ。」
「先生が学生の頃は何に打ち込んでいたんですか?学生運動とか?」
「なあ、君は私をいくつだと思ってるんだ？ちょっとこれからの単位の話をしよう。」
「ひ、ひや規模は小さくなりましたけど今でもやってるとこはやってるらしいですよ?」
いや、ほんといくつなんだろう。面倒くさかったのでそこら辺の設定は曖昧だ。まあ見た目通りの年齢だとするとポストは助教とかだろうか。
「まあ、残り少ないモラトリアムを精々楽しみたまえ。」
そう言い残して平塚先生は駅の方角へ消えた。

社会に出ることを遅延されているのに、新卒だけが高く評価されるシステムが理不尽で謎だ。いっそもう死ぬまで遅延評価でいい。ついでに大学からも卒業を遅延されそうだ。

しかし、時間ね。ありとあらゆるカードが平等に分配されていない社会でも時間だけは平等だ。みんな聞け、提案がある。一度 0 に戻すんだ。みんなのスキルや資産を集めてシャッフルして配りなおすんや。早くしろ、間に合わなくなっても知らんぞ。そう、こうしている間にも時間は絶えず流れている。時間……。その時、俺に天啓が舞い降りた。神[*9]をも冒涜する 12 番目の理論、それが時間。これが世界の、そして俺の選択だ。

3.8　どんなに長い夜でも必ず朝はおとずれる。

次の、次の日。
「で、その選択とやらを見つけたのはいいのだけれど、なぜ昨日は部室に来なかったのかしら。」
「昨日雨降ってたし。全休安定でそ。いや、別に雨が嫌いなわけじゃないんだ。雨が降ってると自転車が使えなくなるのが嫌なんだ。お前、あれだぞ徒歩通学とか歩いてるだけで体力消費するし、

[*9] matz

ドラクエでいうと毒の沼の上をトラマナ無しで歩くようなもんだぞ。」
「放射能汚染の影響はそこまで深刻に出てないと思うのだけれど。あ、汚染されているのはあなたの脳内の方なのね?」
「ばっか、放射脳とか反原発とか死ぬほど嫌いだよ。」
思想とかじゃなくて人種が気にくわない、B 級ホラーで最初に死にそうなイメージ。
はあ、俺も TKB 大生みたいにセグウェイで通学したい。
「それで、問題は解決できたの?」
「ああ。突破口は時間だ。つまりタイムアウトだ。」
コードを見せた方が早いだろう、とりあえず昨日書いてきたものをラップトップのモニタに表示する。

```
TIME_OUT = 0.01
class ShellCmdInvocation < Exception;end
class HikiProc < Proc; end
def hikiproc
  HikiProc.new
end

def method_missing(sym,arg=hikiproc{|x| x.to_s},*rest)
  case arg
  when HikiProc
    cmd = arg.call(sym)
  when String
    cmd = [sym,arg].join(" ")
  end

  super if /^to_/ =~ sym

  t = Thread.new do
    sleep(TIME_OUT)
    Thread.main.raise(ShellCmdInvocation,cmd)
  end

  hikiproc {|x| t.kill; x.to_s + " " + cmd}
end

module IRB
  class Irb
    alias :old_eval_input :eval_input

    def eval_input
      old_eval_input
    rescue ShellCmdInvocation => cmd
      puts
      system(cmd.message)
      retry
    end

    alias :old_output_value :output_value

    def output_value
      return if @context.last_value.is_a? HikiProc
      old_output_value
    end
  end
end
```

「まず、sleep 関数を使って各 method_missing の呼び出しでタイマーをセットするんだ。タイマーは処理を止めてはいけないので別スレッドで実行する。戻り値としてコマンド文字列とタイマーを内包した。プロックを返す。プロックは Ruby のブロックのオブジェクト版みたいなものだな、クロージャになってるのでタイマーと文字列を束縛しておくことができる。」

ちなみに、デフォルトの Proc を汚染しないように HikiProc という名前にしてある。なぜだか逆に汚染されたような気がするのは気のせいだ。

「で、method_missing はタイマーをセットすると同時に受け取ったタイマーをリセットするんだ。タイマーがリセットされたということは後続の呼び出しがあるということだから、そこはコマン

ドの先頭ではないということになる。」
困惑した雪ノ下を無視して俺は続ける。
「タイマーがリセットされずにタイムアウトしたらどうなるか？その時はそこがコマンドの先頭ということになるので system 関数を使ってコマンドを実行すればいい。ここで気をつけなければいけないのは、タイマーがタイムアウトするのは別スレッドだということだ、最後の method_missing の呼び出しは既にリターンしているからここでコマンドを実行すると、コマンドの出力が予期せぬ形でプロンプトに割り込んでしまうことになる。だから、system 関数はメインスレッドで実行しなければいけない。別スレッドに処理を割り込ませるにはどうしたものかと考えたが、例外を使うことで解決した。」
Thread.main.raise(ShellCmdInvocation,cmd) というのがそれだ、ここでメインスレッドで例外を起こすと同時にコマンド文字列も渡している。
「例外を投げたら拾わなければいけない、どこで拾うかだが、無難に irb で入力を評価している部分で拾えばいいだろう。」
Ruby のクラスというのは実行時にいつでもオープン、つまり再定義できる。だから irb で定義されているクラスをオープンしてメソッドをオーバーライドしてやればいい。irb で入力行を取得して eval している部分が IRB::Irb#eval_input だ、この中で例外を拾えばいい。ここでは alias を使って元のメソッドを再利用している。このように、例外を拾うためだけにメソッドをオーバーライドするやり方をアラウンドエイリアスという。
「と、まあ駆け足になったがこんな感じだな。」
多分ほとんどの人間は hikiproc しか記憶に残っていないだろうな。まあもうそれでいいや。
「極めて非決定的論なプログラムだということは分かったわ。TIME_OUT の時間しだいではとんでもないことになりそうね。」
「ま、ともかくやっとスタート地点に立ったというところだな。残す課題は山積みだ。オプションのハイフンとかはマイナス単項演算子でなんとかするとして、その他の記号の類、&, $, =, >, < あたりをなんとかしないといけない。それらと method_missing を組み合わせると相当複雑になりそうだ。」
「残念だけれど、続きはないわ。もう記事の〆切の時間よ。」
「え？まだ 4 週間近くあるんでそ？」
「こちらではそうなのだけれど、現実世界の方で〆切が来てしまったのよね。」
どうりで今年は秋がなかったわけだ。ファッキンコールド。ていうかもう〆切過ぎてるんだけどな。
そう、時間は誰にでも平等に、残酷に流れる。明けない夜が無いように〆切の無いタスクも存在しない。やはり何もかも、俺の irb シェル化計画は間違っている。 了

第 4 章

情報開示請求 簡易報告

KHe7

本記事は私の行った情報開示請求について簡単な報告を行うものである。

4.1 はじめに

私は、国立大学法人 電気通信大学 (以下、電通大とする) を相手として『独立行政法人等の保有する情報の公開に関する法律』による情報公開制度を利用し、請求する法人文書として “平成 23、22、21 年度の電気通信大学の学部、大学院で行われた全ての科目 (時間割コード別) の成績の人数分布が分かる資料” と指定して 2011 年 11 月 9 日に法人文書開示請求を行った。

以降 2 年間にわたって私が行ってきた手続きや本制度の導入経緯やその趣旨などは前の百萬石[*1]に寄せた記事において十二分に記述したため本記事では扱わない。

この記事ではその後の手続きと最終的に開示された資料について記述する。

4.2 前記事以降の手続き

4.2.1 手続きの日時

2013.10.07。情報公開・個人情報保護審査会 (以下、審査会とする) において審議。
2013.10.09。審査会、諮問に対する答申を行い、答申書の写しを私に対して送付。
2013.11.01。電通大、審査会の答申を受けて異議申立てに対する決定を行い私に通知。
2013.11.11。私、法人文書の開示の実施方法等申出書を電通大に提出。
2013.11.15。電通大、法人文書の開示を実施し私に対象資料を開示。

4.2.2 手続きの具体的説明

まず、前記事の執筆時点における状況について説明する。

私の情報開示請求に対して電通大は法人文書開示請求の対象となる文書を保有していないとし

[*1] 百萬石 2013 年 夏号、コミックマーケット 84 にて頒布。

て全部を開示しないという決定を行った。

その法人文書不開示決定に対して私は異議申立てを行った。電通大から審査会に対して行われた諮問の答申において、ある会議資料の一部が対象の法人文書に相当すると認定された。

昨年、電通大によってその資料の開示決定が行われたが表中の科目名、担当教員氏名、受講者の人数、評定平均、秀 %、優 %、良 %、可 %、不可 % が不開示となる部分開示の決定であった。

私はその決定に対して異議申立てを行った。その異議申立ては電通大によって審査会に対して諮問されており答申が行われるのを待っていたのが前記事時点での状況である。。

次に、前小節に挙げた手続きについて説明する。

審査会は今年 3 月からの数度に渡る審議の結論として 2013 年 10 月 9 日に審査会は答申を行った。そして、この答申において担当教員の氏名、及び受講者の人数は法第 5 条 4 号に該当するため不開示としたことが妥当であるが、科目名並びにクラス別の評定平均、秀 %、優 %、良 %、可 %、不可 % の数値は同条 3 号及び 4 号のいずれにも該当せず開示すべきとした。

電通大はこの答申を受けて当該文書の担当教員氏名並びに受講者の人数を不開示部分とする法人文書開示決定を行いその旨を私に通知した。

私は電通大に対して写しの交付による開示を希望する旨の法人文書の開示の実施方法等申出書を提出し、先日、電通大から法人文書の写しを受け取った。

4.3　開示文書の内容

請求から 2 年の期間を経て入手した、「第 8 回大学教育センター教育推進部門会議 (平成 22 年 12 月 16 日) 資料 1,3 のうち「クラス別」の表」は次のような内容で構成されていた。

表 4.1　第 8 回大学教育センター教育推進部門会議 (平成 22 年 12 月 16 日) 資料 1,3 のうち「クラス別」の表

■ 表 3-1 クラス別

001.	▼ Academic Spoken English I							
002.	▽ 2009	人数	評定平均	秀 %	優 %	良 %	可 %	不可 %
003.	黒塗り	黒塗り	3.0	33.3	36.7	30.0	0.0	0.0
004.	黒塗り	黒塗り	3.0	30.6	47.2	13.9	5.6	2.8
005.	黒塗り	黒塗り	2.9	27.0	51.4	16.2	0.0	5.4
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
022.	黒塗り	黒塗り	1.7	2.2	22.2	24.4	46.7	4.4
023.	黒塗り	黒塗り	1.6	8.8	8.8	29.4	41.2	11.8
024.	▽ 2010	人数	評定平均	秀 %	優 %	良 %	可 %	不可 %
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
286.	黒塗り	黒塗り	2.1	5.4	41.1	30.4	7.1	16.1

このような構成で 10 科目、2 年分の全クラスの成績分布が記載されていた。

4.4　おわりに

初めての情報開示請求は 2 度の異議申立てを挟んだ 2 年間の期間を有する長大なものとなった。

受け取った文書の内容もさることながら、手続きを行うにあたって調査した結果や手続きそのものも非常に楽しく有意義な活動であったと考えている。

また、対象文書不存在の不開示決定が異議申立てと審査会への諮問、答申を経て不開示部分ばかりの開示決定へと変わり、再びの異議申立てと審査会への諮問、答申を経て最終的に有意義な内容に満ちた文書が開示されるという経験ができたことは稀に見る幸運であったと思う。

今後も興味のあることについて、自ら調べてまとめるだけでなく、詳しいであろう組織の持つ情報を利用することを積極的に取り入れていきたい。

思うところがあることが幾つかあるので、一段落した頃に再び活動を始めたいと思う。最長 2 年かかることを考えると複数を並行して進めることは必要になるだろう。ゆっくり進められる間に物事を進めておきたい。

また、今回触れなかった資料の数値に関する細かい話については自身の余裕を考えつつ春の百萬石ないし次回の Dentoo.LT にて話題にできればと思う。

4.5　参考文献

- KHe7 「情報公開制度」 百萬石 2013 年 夏号 電気通信大学 MMA, pp.83-89, (2013).
- 情報公開・個人情報保護審査会 答申書 平成 25 年度 (独情) 答申第 32 号.

第 5 章

いろいろな iPhone5s の買い方

Mernao

携帯電話の販売形態は複雑だ。であるから多種多様な買い方が存在する。定価で携帯キャリアショップで買う以外にも買い方はある。今回は、遂に 3 キャリアから発売された iPhone5s の買い方についてゆる〜く考えてみる。

5.1 注意事項 · 免責事項

この記事を読んだことにより発生したいかなる損害も筆者は保障致しません。中には大きなリスクを背負うものもあり、場合によっては二度と携帯電話会社と契約を結べなくなったり、予期せぬ請求が発生する可能性があることをご承知ください。

5.2 用語解説

携帯を安く買うために必要な用語について軽く説明する。

MNP　モバイルナンバーポータビリティの略。番号はそのままで他社へ乗り移れるシステム。他社の顧客を奪い取ることができるため、各社躍起になって MNP で転入してくる顧客に対して様々な優遇措置をとっている。つまり客にとっては MNP は狙い目。逆に長期利用顧客が不利に思えることもしばしば。MNP には事務手数料 2,100 円がかかる。

弾　MNP の転出に使うための電話回線。これが無いと転出はできない。通常は今使っている電話回線を弾にするが、弾を作る方法もある。

違約金　契約解除料などと通常は書かれる。所謂「2 年縛り」を破った時に発生する違約金の事。更新月でなければ 9,975 円かかる。

BL　ブラックリストの事。携帯電話の短期解約を繰り返したり、支払いを滞納するとブラックリストに入れられて、契約ができなくなる。BL 入りの期間を喪中と呼んだりもする。docomo は喪中期間が非常に長いことで有名で 10 年超えの場合もあるらしい。しかも喪明けしても預託金数万円が必要になったりと相当厳しい。

養分　携帯電話会社に無駄金を払うこと。

定価　そのまんま。携帯の定価は普通 6-9 万円と非常に高い。だが機種によっては下手すれば 2 ヶ月程度で 0 円に値下げされるの。特に au の場合これが顕著なので、どうしても発売と同時に欲しいのでなければ数ヶ月待つことをオススメする。

頭金　頭金は分割金の一部ではなく、携帯電話ショップ (キャリアではない) の懐に入るものである。つまり払うだけ無駄。頭金 1 万円の店が普通にあるが、全部ショップの懐に入っている。

実質 0 円　実際は 0 円ではない。これについては面倒なのでここでは詳しく述べないが、全くもって 0 円ではないという事に注意してほしい。逆に一括 0 円は本当の 0 円。

コン・オプ　コンテンツ、オプションの事。これらに加入することで端末代金を安くしたりしてもらえる店が多い。安い場合は大抵加入が強制で、とてつもなく安い場合はこれらに大量 (場合によっては 30 個以上) に加入する必要があるので注意が必要。30 個もコンテンツを付けられると外すのも面倒である。(一発で外す方法もあることはある)

純新規　MNP 新規契約でない新規契約の事。回線が 1 つ増える。

シュート　端末を売る事。高額で買い取ってくれる店もある。

キャリア　携帯電話会社のこと

k　キロ=1,000 の事。10k の場合は 10,000 となる。キャッシュバックの表記によく使われる。

CB　キャッシュバックの事。2,3 ヶ月後に指定の口座に振込む方式や、後日店頭引渡しであったりする。稀に当日にもらえる場合もある。キャッシュと言いつつ実際は商品券という店もあるので注意が必要である。

5.3 iPhone5s について

iPhone5s はこの記事執筆時点で Apple 社の最新端末である。日本では日経に散々過度な期待をされ続けた docomo からも遂に発売されたため、docomo·au·Softbank の 3 大キャリアのいずれでも契約が可能となった。各社、月々の利用料金から一定額を割り引く施策を行なっているため、2 年間使い続けると安くなるようになっている。この割引は通信料などから割り引かれる訳であって、端末代金から割り引いてくれるわけではない。なので端末本体は安く買ったほうが良いに決まっているということである。本記事では、au か Softbank の iPhone5s を購入することを考える。docomo の購入を考えないのは docomo の端末代金が他 2 社よりも高く設定されていて比較が面倒だからである。

5.4 MNP 優遇について

MNP 優遇には色々なものがあり、例えば基本料金 980 円程度が 2 年間無料になったり、キャッシュバックがあったりなどがある。基本料金無料はキャリアがやっている事なのでどこの店で買っても同じである。しかしキャッシュバックは各店舗が勝手にやっている事なのでその額はまちまちである。仮にやたら高いキャッシュバックを謳っている場合があっても、多くはコンテンツ等に大量加入することが必須であったり、変なものを抱き合わせで買わされたりするので結局高くつくので注意が必要である。悪質な店舗の場合は基本料金無料の 980 円 x24 ヶ月=23520 円を堂々とキャッシュバック額に含めて表示しているので、キャッシュバックの内訳の確認は要確認である。特に「最大○○円キャッシュバック」というのは要注意で、基本料金無料に加えて学割を使った時の割引額を含めて[*1]表示をしている場合がある。

5.5 いろいろな買い方

それでは早速 iPhone5s のいろいろな買い方を紹介して行こう。ここでは一番安い 16GB モデルを買う事を考える。

5.5.1 定価購入

発売日に購入しようとしたらまずこの方法しか無い。諦めよう。定価は 68,040 円なので、純新規の場合はそれに新規事務手数料 3,150 円が追加で必要となる。もちろん月々の通信料の支払いも必要だ。ということで定価で買うと 71,190 円必要となる。

[*1] それは割引であってキャッシュバックではないじゃないかと店に喝を入れてやりたい。

5.5.2　MNP

先に述べたように MNP を使うことで優遇を受けることができる。本体代金の値引は、購入する店によって多い少ないが変わってくるので吟味が必要である。筆者がざっと市場調査をした限りでは MNP 購入者に 20k 円程度の優遇を設けている店が数店舗あった。例えば数カ月後に 20k 円の CB(商品券) がもらえる店や、20k 円分のポイントがもらえる大手家電量販店などが見られた。CB の代わりに本体代金からその場で値引きできるという店もあった。一番条件の良い店ではなんと MNP なら本体代金が 35k 円という店もあった。なんと定価の約半額である。

さて、この様に MNP には非常に美味しい面もあるがもちろんデメリットもある。1 つは今使っているキャリアを解約したり、MNP 事務手数料がかかってしまうことである。大抵の場合は 2 年間使う代わりに基本料を安くしますよという所謂「2 年縛り」のプランに加入しているため、更新月でない場合は違約金として 9,975 円が必要となる。ここに MNP 事務手数料 2,100 円と転出先での新規事務手数料 3,150 円が必要となるので合計で約 15k 円が MNP での購入に必要となる。更に MNP で安く購入ができる店ではオプションやコンテンツへの加入が条件になっている場合が大抵なので注意が必要である。よくあるのは同一キャリア内での通話が無料になるオプションや、パケット定額プランへの加入である。少なくとも来月まではオプションを解約してはいけないという条件になっている事が多い。パケット定額への加入はまともに使うのであれば必須であるので、ここでは計算には入れないことにする。ここで筆者の経験上、オプションとコンテンツにかかる費用をざっと 3k 円と見積もっておく。すると最終的に購入に必要な額は 53k 円となる。なんだかんだあったが定価で買うよりも 18k 円程度安く買えることになる。また、MNP なら基本料無料になるというキャンペーンをやっていた場合、仮に 1 年間使って買い換え[*2]たとしても 980 円 x12 ヶ月=11,760 円安くなるので機種変更などで買うよりもオトクということになる。

5.5.3　MNP+ 下取り + 機種変

さて、ここまでの買い方は割と普通の買い方なので、お店で相談すれば伝授してもらえるだろう。ここからはちょっとイレギュラーな買い方を紹介していく。もちろん手間も危険度も上がるので注意が必要である。

まず下取りとは何かというと、Softbank が最初に実施したもので、今使っているスマートフォンを電話会社で買い上げる代わりに、月々の利用料金から割引ますよというものである。例えば iPhone5 16GB なら最大で 1,000 円 ×21 ヶ月=21,000 円の値引きとなる。下取りの最大の欠点は、月々の料金から割り引くため、途中で解約してしまった場合は売ったはずなのに割引が受けれずに損をするという事である。一括の価格を安くするという趣旨の記事なのにおかしいじゃないかというお叱りを受けそうだがそこは許して頂きたい。

しかし MNP+ 下取りとは一体どういうことかと疑問に思われると思う。どういう仕組みかと

[*2] 1 年間という数字の根拠は、1 年経ったら新しい iPhone 出るし買い換えるよね、というものである。

いうと、MNP をしてまず旧世代の iPhone5 を安く購入し、それを下取りに出してその回線で機種変更を行うというものである。

ではまず iPhone5 を購入することを考える。iPhone5 なら今なら MNP で 16GB が一括 0 円の 25kCB ぐらいで買えるだろう。機種変更は多分定価での購入になるだろうから、本体代金 68,040 円 + 機種変更手数料 2,100 円がかかる。MNP にかかる諸費用を 18k 円と見積もっておくと、大体 42k 円で購入できるという事になる。普通に MNP で購入するよりも安く購入できるという訳である。

この方式の心配な部分は、果たして iPhone5 を購入した直後に機種変更を行えるかという事である。よくある機種変更が格安でできる案件では、端末利用期間が 3 ヶ月以上でないと機種変不可能というものがある。もしこの制限に引っかかってしまうと iPhone5 を買ったものの機種変更ができないという事になる。ここからは筆者の憶測だが、端末利用期間 3 ヶ月以上というのは安くなっている端末だけに適用される条件で、定価で買う分には影響しないのではないかと思う。この辺は要確認である。

もう一点注意が必要なのは、au の場合は下取りが可能なのは端末購入から 3 ヶ月経過したもののみという点である。つまり au ではここで述べたやり方は使えないというわけである。Softbank の場合はざっと見た感じでは書いていないので可能ではないかと思う。

5.5.4　MNP+ シュート + 機種変

前章で MNP+ 下取りを紹介したが、この方式は正直お勧めしない。というのも下取り価格があまりにも低いからである。しかも au の場合は 3 ヶ月経過していないと下取りしてもらえないという問題がある。更に言えば、2 年間使わないといけないという問題[*3]がある。そこで、下取りに出すのではなく普通に中古で売ってしまう (シュート) 事を考えてみる。

iPhone5 なら新機種が出た今でも高額買取の店なら White16GB を 45k 円で買い取ってくれる。この場合 5.5.3 の時の下取り価格が 45k 円に上がったと考えればいいので、なんと 18k 円で iPhone5s が買えるという事になる。どう考えても下取りよりもこの方式の方が良いだろう。

5.5.5　多重 MNP+ シュート + 機種変

ここまで iPhone5 をシュートするとオトクに 5s が買えるという話をしてきた。となると、iPhone5 を 1 台ではなく複数台売ったらどうなるか気になるだろう。しかし、iPhone5 を 1 台買った時点で、すでに 1 回 MNP をしているのでもう MNP をすることができない。MNP した直後にもう一回 MNP をすれば良いと思うかもしれないが、まず確実に BL 入り[*4]するだろうし、25k 円の CB が貰えない。さらに言えば、販売店に多大な迷惑[*5]が掛かるので絶対に即解約・即 MNP 転出はやめて頂きたい。

[*3] iPhone は新機種が毎年出るのに 2 年間も使わないといけないとはどういうことだと筆者は思ってしまう。

[*4] Apple 端末は他の端末よりも BL 基準が厳しいという噂もある

[*5] 最悪閉店

こうなると回線を増やす他に方法はなくなる。簡単に回線を増やす方法としてプリペイド携帯がある。今回は Softbank のプリペイド携帯である、プリモバイルを使用することを考える。プリモバイルの回線を作るためには Softbank の端末を Softbank ショプに持ち込み、プリモバイル用の SIM を発行してもらわなければならない[*6]。SIM の発行自体は無料だが、チャージをしないと発信できないし、MNP もできないので 3000 円チャージをする必要がある。プリモバイルは昔は転出の際に違約金を取られたりしなかったが、弾に使われまくった結果、対策として違約金を設けるようになった。これがかなり高額で 20,790 円取られてしまう。更に MNP 手数料もバカ高くて 5,250 円する。つまりプリモバイルにかける金額だけで 29,040 円となる。では転入した後の事を考えてみる。BL 入の基準はよく分かっていないが 6 ヶ月とか 3 ヶ月とか言われているので 6 ヶ月持つことを考える。au ならパケットフラット必須の契約でも、月末に契約すれば日割りが効くので安く抑えられる。店の条件にもよるが、翌月フラットを外してもいいならば維持費は LTE プランの基本料金 980 円に抑えることができる。事務手数料で 3k、解約時の違約金で 10k、コンテンツやオプション (フラットなども含む) でざっと 5k 円かかると見積もって、更に維持費 980 円 ×6=6k 円を加えると、転入先で 24k 円かかることになる。CB+ シュートで 70k なので手元に残るのが 16k 円となる。この分が単純に iPhone5s の本体代金に充てられるとすると、2k 円で iPhone5s が買えることになる。

長々と書いてきたが、かなりややこしいしリスクもかなり大きいのでこの方法はおすすめできない。

5.6　結局どうすれば良いのか

で、結局どの買い方が良いのか。正直筆者にも分からない。安くしようとすればするほど条件は厳しくなるし、リスクも大きくなる。一歩間違えば多額の請求がポンと来ることも覚悟しなければならない。一番安全かつオトクなのはやはり 5.5.2 の単純に MNP で買うやり方ではないだろうか。頑張って良さそうなお店を探してみて欲しい。

ちなみに筆者はどうやって iPhone5s を買おうと思っているかというと、早急に欲しい訳でも無いので MNP 一括 0 円で CB が 10k ぐらい乗り始めたら買おうと思っている。一括 0 円の時期が何時頃になるかというのに関しては諸説あるが、年度末である来年 3 月という説を筆者は信じている。

5.7　おわりに

ここに書かれている情報は 2013 年 11 月中旬現在の情報で、筆者が適当に情報収集したものである。従ってそれほど信頼できるものではない。参考程度に、こんな馬鹿げた購入方法があるのかという程度に見て頂きたい。また金額の計算もかなり適当で、端数処理もかなりざっくりである。もしも何を思ったか本当に色々やろうと思っている人がいるならばしっかりと調べて自己責

[*6] 発行してくれない店もある模様。端末の種類は何でもいいのかどうか分からない。

任で行なって欲しい。繰り返しになるが、この記事を読んで損害などが発生しても筆者は一切責任を負わない。

5.8 参考文献

[1] [プリモバイル] 契約解除料はかかりますか？ — ソフトバンクモバイル
http://faq.mb.softbank.jp/detail.aspx?id=72716

[2] スマホ下取り割 — モバイル — ソフトバンク
http://www.softbank.jp/mobile/campaign/list/sp_trade_in/

[3] 下取りプログラム — スマートフォン・携帯電話 — au
http://www.au.kddi.com/mobile/trade-in/

第 6 章

液晶制御欲が高まったわけだが

mznh

ここ最近 AVR を使ってマイコンプログラミングを色々やっている。マイコンプログラミングは最初のうちは LED をチカチカさせたりドットマトリクスをチカチカさせたりとひたすらチカチカさせがちだ、そこで今回はキャラクタ LCD を制御して適当な文字列を表示させてみようと思う。

6.1 マイコン

制御に使うマイコンは ATTiny2313 を用いた。ライターとか開発環境とかの説明は割愛します。

6.1.1 超々簡易マイコンプログラミング入門

とても大胆な意見だがマイコンプログラミングとは単純にいうと如何に器用にマイコンの足の ON/OFF を切り替えるかだと筆者は思っている。ここでは AVR マイコンにおけるそのやり方をこの上なく大雑把に説明する。なお各マイコンのデータシート等を合わせて見るとよいかもしれない。

```
DDRD=0b00110011;
// マイコンの足はDDRA~Dでグループわけされている。
// このビットを立てるとそのピンは入力ピンになる。（ON/OFFを読み取る
PORTD=0b11110000; // このビットを立てると対応するピンがONになる。
PIND；// この値が読み取られたON/OFFの結果と対応している。
```

6.2 キャラクタ LCD

今回は秋月で売られている SC1602BSLB を制御することにした。これは英数字を用いて 16 文字 2 行の表示ができる。マイコンとの通信には 8 本のピンをデータ用に使うモードとそれが 4 本のピンですむモードがある。マイコンの足をたくさん使うと汎用性が失われ配線も面倒になるので今回は 4 本のモードを選択した。

図 6.1　SC1602BSLB

6.2.1　タイミングチャート

早速制御するべくデータシートを読む。データシートには大抵タイミングチャートというものが載っておりこれのとおりにピンを ON/OFF してやると諸電子部品とマイコンが円滑に通信できるようになるのだ。なのでタイミングチャートにしたがって 4bit のデータを送信する関数、及びそれを用いて 8bit のデータを送信する関数をゴリゴリ実装した。その前に後々ピン配置変えてもいいように色々と define。

```
#define LCD_RS_DDR DDRD    /* LCDのRSピンが接続されているいるピンのDDR
    */
#define LCD_RS_PORT PORTD /* LCDのRSピンが接続されているピンのPORT */
#define LCD_RS_BIT 6     /* LCDのRSピンがつながっているピンの番号 */
#define LCD_RW_DDR DDRD/* RWピンについて同様 */
#define LCD_RW_PORT PORTD
#define LCD_RW_BIT 0
#define LCD_EN_DDR DDRD/* ENピンについて同様 */
#define LCD_EN_PORT PORTD
#define LCD_EN_BIT 1
#define LCD_DATA_DDR DDRD/* データピン(4本)について同様 */
#define LCD_DATA_PORT PORTD
#define LCD_DATA_PIN PIND /* データピンが接続されているPINも定義 */
#define LCD_DATA_BIT 2
/* データピン4本の先頭番号(この場合データピンはPortD 2〜5) */
```

```
void __lcd_send4bit(uint8_t data){
  cbi(LCD_EN_PORT,LCD_EN_BIT);          //EをOFFに.
  cbi(LCD_RS_PORT,LCD_RS_BIT);          //RSの制御レジスタを選択.
  cbi(LCD_RW_PORT,LCD_RW_BIT);          //書き込みモードに設定.
  LCD_DATA_PORT = (LCD_DATA_PORT&(~(0x0F<<LCD_DATA_BIT)))|data<<
      LCD_DATA_BIT;
  //送信データのビット書き込み
  sbi(LCD_EN_PORT,LCD_EN_BIT);          //EをONに.
  _delay_us(10);                  //待機
  cbi(LCD_EN_PORT,LCD_EN_BIT);          //EをOFFに
  _delay_us(10);                  //待機
}
//4bitモード時に8bit命令を送信する関数　dataの書式は0b76543210な感じで
//rs_flag=RSのON/OFF,rw_flag=R/WのON/OFF,data=書き込むデータ
```

```
void __lcd_send8bit(bool rs_flag,bool rw_flag,uint8_t data){
  uint8_t top = (data&0xF0)>>4;//データの上位ビットを0b0000****な感じで
  uint8_t bottom = (data&0x0F);//データの下位ビットを0b0000****な感じで
  //上位ビット送信
  cbi(LCD_EN_PORT,LCD_EN_BIT);//ENをオフに
  if(rs_flag){sbi(LCD_RS_PORT,LCD_RS_BIT);}
  else{cbi(LCD_RS_PORT,LCD_RS_BIT);}//RSの値よりレジスタを選択
  if(rw_flag){sbi(LCD_RW_PORT,LCD_RW_BIT);}//RWを操作
  else{cbi(LCD_RW_PORT,LCD_RW_BIT);}
  LCD_DATA_PORT=((LCD_DATA_PORT&(~(0x0F<<LCD_DATA_BIT)))|top<<
      LCD_DATA_BIT);
  //送信データのビット書き込み
  sbi(LCD_EN_PORT,LCD_EN_BIT);//EをON
  _delay_us(10);
  cbi(LCD_EN_PORT,LCD_EN_BIT);//ENをオフに
  //下位ビット送信
  cbi(LCD_EN_PORT,LCD_EN_BIT);//ENをオフに
  LCD_DATA_PORT =( (LCD_DATA_PORT&(~(0x0F<<LCD_DATA_BIT)))| bottom<<
      LCD_DATA_BIT );
  //送信データのビット書き込み
  sbi(LCD_EN_PORT,LCD_EN_BIT);//EをON
  _delay_us(10);
  cbi(LCD_EN_PORT,LCD_EN_BIT);//ENをオフに
  lcd_waitbusy();//ビジー待機
}
```

6.2.2 ビジーフラグを読み取ろう

この液晶はマイコンからデータを受け取るだけじゃなくて液晶側が処理中かどうかをマイコンに教えることもできる。そのためには液晶からデータを読み出す関数を作ってやる必要がある。というわけでゴリゴリ実装した。

```
//4
    bitモード時に8bit命令を受信する関数　返り値の書式は0b76543210な感じで

//rs_flag=RSのON/OFF
//なお返り値は受信された"8bit"のデータである。
uint8_t __lcd_receive8bit(bool rs_flag){
  uint8_t top = 0;
  uint8_t bottom = 0;
  //データ受信準備
  LCD_DATA_DDR &= ~(0x0F<<LCD_DATA_BIT);//出力ピン全てを入力ピンに
  cbi(LCD_EN_PORT,LCD_EN_BIT);//ENをオフに
  if(rs_flag){sbi(LCD_RS_PORT,LCD_RS_BIT);}//RSの値よりレジスタを選択
  else{cbi(LCD_RS_PORT,LCD_RS_BIT);}
  sbi(LCD_RW_PORT,LCD_RW_BIT);//RWをオンに
  LCD_DATA_PORT&=~(0x0F<<LCD_DATA_BIT);//内部プルアップをOFFに
  _delay_us(10);
  //上位ビット受信
  sbi(LCD_EN_PORT,LCD_EN_BIT);//EをON
  _delay_us(10);//待機
  top=0x0F&(LCD_DATA_PIN>>LCD_DATA_BIT);//上位ビットを受信
  _delay_us(10);
  cbi(LCD_EN_PORT,LCD_EN_BIT);//ENをオフに
  _delay_us(10);
  //下位ビット受信
```

```
  sbi(LCD_EN_PORT,LCD_EN_BIT);//Eを ON
  _delay_us(10);//待機
  bottom=0x0F&(LCD_DATA_PIN>>LCD_DATA_BIT);//下位ビットを受信
  _delay_us(10);//待機
  cbi(LCD_EN_PORT,LCD_EN_BIT);//ENをオフに
  //後始末
  LCD_DATA_DDR|=(0x0F<<LCD_DATA_BIT);//全て出力ピンに戻す
  cbi(LCD_RW_PORT,LCD_RW_BIT);//RWをオフに
  _delay_us(1);
  return (top<<4|bottom); //受信された8bitのデータを返す。
}
```

6.2.3　できた

これでマイコンと液晶の間で自由にデータが送受信できる環境が整った。後はデータシートの通りに制御してやればいいだけだ。というわけで出来た。

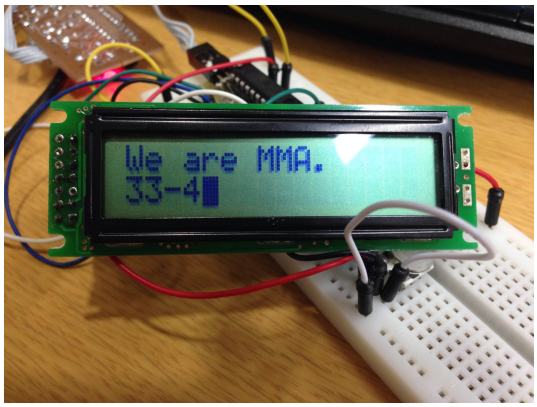

図 6.2　できた

6.3　参考文献

ＬＣＤキャラクタディスプレイモジュール（１６×２行バックライト付）: ディスプレイ関連 秋月電子通商 電子部品 ネット通販データシート http://akizukidenshi.com/download/ds/sunlike/SC1602BSLB-XA-GB-K.pdf

百萬石 2013年 秋号 ©電気通信大学 MMA

2013 年 11 月 21 日　初版第一刷発行　【本書の無断転載を禁ず】

著　者　alstamber, hayakawa, KA ☆ SI ☆ O
　　khe7, mernao, mznh
編集者　hiyakashi
発行者　電気通信大学 MMA
発行所　電気通信大学 MMA 部室
　　〒182−8585 東京都調布市調布ヶ丘 1−5−1 サークル会館 208 号室
　　http://www.mma.club.uec.ac.jp/
印刷所　電気通信大学 MMA 部室
製本所　電気通信大学 MMA 部室
表　紙　電気通信大学 MMA

2013 秋号