囲碁的分析・設計手法

佐原伸
URL: http://www.sra.co.jp/people/sahara
E-Mail: sahara@sra.co.jp

1章 はじめに

前々から、ソフトウェアの設計は囲碁のプレーに似ていると感じていた。囲碁の定石[1]や手筋は、ソフトウェアの設計で言えば「データ構造とアルゴリズム」などに相当し、囲碁のルールやヨセの一部が数学的に厳密に定義されている[2]のは、ソフトウェアの設計で言えば「形式仕様で問題の一部が記述されている」ことに相当しよう。

プロの棋士は、形式に定義されたルールや手筋、あるいは非形式に定義された定石や手筋を場面に合わせて使い、既存の定石や手筋がないところは自分で考えて碁を打つ。もちろん、形式に定義されたルールや手筋の中身を完全に知っているわけではない。形式定義から導かれた実践的な定理を使って打っているにすぎない。

ところが、現在の開発現場の設計者の大部分は、「ルールを知らず、定石を知らず、手筋を知らず」あるいは「間違ったルールと定石と手筋を使って」ソフトウェアの設計を進めている。

そこで、本稿では形式仕様とデータ構造とアルゴリズムなどの定石を使い、「囲碁的パラダイム」で実際にプログラムが開発できることを検証してみようと思う。もちろん、形式仕様の研究では、実用的な手法やツールが開発され「実際に使えること」は分かっているのだが、それと「開発現場で使えること」とは次元が異なるので、開発現場からの視点で検証してみようというわけである。

対象のプログラムは、日付計算にしてみた。有価証券評価システムをSmalltalkで作っているチームが私の隣にいるので、少しでも手伝おうと言う趣旨である。また、問題がさほど大きくならないので、本稿で説明するにはちょうど良い大きさであろうという判断もある。

2章 問題の説明

事務処理に必須な「２つの日付の間の、日曜日の回数を求める」を行うことを考えてみよう。この機能は、「休日を除く日数の加算や減算」を行うことや「第２土曜日や第４金曜日を求める」場合に必須の機能である。この時、休日を考慮しない日付間の加減算や曜日を求める関数は存在するものとする。これらの機能も形式仕様で分析・設計・実現したのだが、天文計算のドメイン知識を調べるなどの面倒はあるものの、最終的にはプログラムというよりさほど複雑でない計算式で実現されるので、今回の説明からは除いた。ともかく上記のような関数を使えば、毎年年末に日本の各地で繰り返される「不完全な日付処理」によるシステム・トラブルが少しは減るだろうし、昭和が平成に変わったときの混乱

[1]定石は約２万個あると言われている。

[2]エルウィン・バーリカンプ／デビッド・ウルフ著、吉川／小林／石原訳「囲碁の算法　ヨセの研究」トッパン, 1994, ISBN=4-8101-8929-5

は防げたであろうし、無事2000年を迎えることができるだろう。

2.1 要求仕様記述言語

要求仕様は、形式仕様記述言語であるRSL[3]（RAISE Specification Language）を使ってみよう。この言語は、ロボットの仕様記述や、中国の鉄道システムの要求仕様記述言語として用いられた実績があるから、日付計算くらいは簡単に記述できると考えたからである。

2.2 行き当たりばったり仕様

まず「ソフトウェア開発の現場」の伝統に則って「行き当たりばったり」法で仕様を考えてみよう。

```
日曜日回数: Date×Date→Int
/* fromとtoの間の日曜日の発生回数を返す。 */
variable days : Int

日曜日回数（from, to）≡
  days := subtractDate(to, from);
  if isSunday(from)
    then days / 7
    else (7 - dayNumber(from) + days) / 7
  end
```

仕様1　行き当たりばったり仕様

「日曜日回数」が関数名で、２つのDate型の引数（from, to）を持ち、整数を答えとして返す。'//'は整数の商を計算する演算子である。subtractDateは日付の引き算を行う関数であり、isSundayは日曜日か否かを返す関数であり、dayNumberは曜日を0..6の整数（0が日曜日）で返す関数で、いずれも、あらかじめ定義されているものとする。

この仕様は、すぐにfrom日付が日曜日であると答えがおかしいことに気が付く。そこで、以下のように修正した。

---

[3]The RAISE Language Group著、「The RAISE Specification Language」、Prentice Hall、1992, ISBN=0-13-752833-7

```
日曜日回数: Date×Date→Int
/* fromとtoの間の日曜日の発生回数を返す。 */
variable days : Int

日曜日回数（from, to）≡
  if from = to then
    if isSunday(to) then 1 else 0 end
  end;
  days := subtractDate(to, from);
  if isSunday(from) then
    if days > 0 then days / 7 else abs(days) / 7 end
  elsif days > 0
    then (7 - dayNumber(from) + days)  / 7
    else dayNumber(from) - days / 7
  end
```

仕様2　行き当たりばったり仕様修正版

この仕様でも欠陥があるのだが、それは読者への宿題としよう。いずれにせよ、これらの仕様には以下の欠点がある。

(1) プログラミングとテストの工数が増大する
(2) プログラミングがいつ終わるか分からない
(3) 正しいことの保証がない
これが、最大の問題なのだが、「行き当たりばったり」法で作ってしまったプログラムの正しさを証明するのはきわめて困難である。実行時にいくらテストしてみても、「すべての組み合わせをテストした」と胸を張って言える場合以外、正しいことの保証はない。そして、「すべての組み合わせをテストする」のは、実用的なプログラムの場合、ほとんど不可能である。

2.3 目的指向の段階的詳細化

「ではどうするか？」ということになるが、形式手法では「目的指向型」で開発する。具体的には、プログラムの作成と証明を同時に行い、段階的に詳細化していくという方法で、以下の手順で仕様からプログラムへと変換していく。

(1) 問題は何であるかを把握する
ちょっと面倒なことになるので、後述する
(2) 事後条件・事前条件を見つける
プログラムは事前条件を満たすどんな状態の下で実行しても、有限時間内に実行が終わり、事後条件を満たす状態をもたらす
(3) 再帰または繰り返しにより事前条件を事後条件に近づけていく
ここからは設計工程になる。実際には、事後条件を緩めて、事前条件P_0から事後条件へ至る途中の「条件」（P_1, P_2など）を導出する

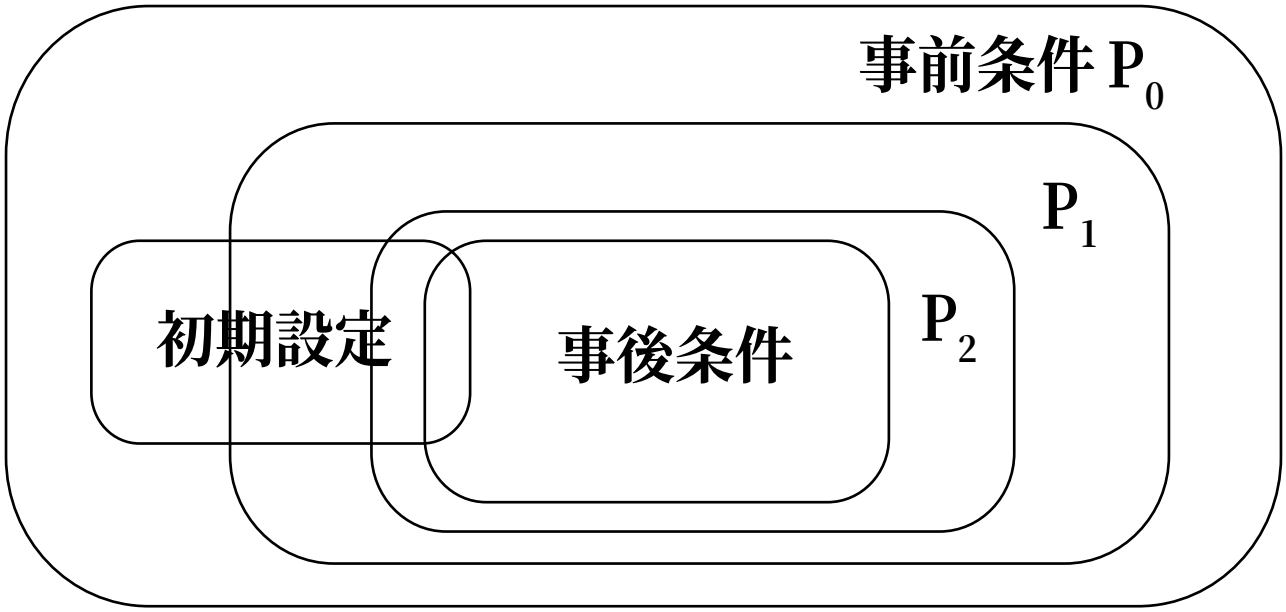

図 1　事後条件・事前条件・初期設定の関係

(4) 最後に、効率の考慮

効率の問題は、本稿ではあまり触れない

2.3.1 事後条件を求める

ここでは、「問題は何であるか」をひとまず考えないことにして、事後条件の洗い出しへ進もう。事後条件は以下のようになる。

曜日回数（dayOfTheWeek, from, to）≡
/* fromとtoの間の曜日dayOfTheWeekの発生回数を返す。 */

post
∃DW:曜日-set, ∀d∈DW・from ≦ d ≦to∧dayNumber(d) = dayOfTheWeek⇒
曜日回数（dayOfTheWeek, from, to）≡ card(DW)

仕様3　事後条件

ここでは、事後条件を考えるついでに、「２つの日付の間の、日曜日の回数を求める」のではなく、「２つの日付の間の、指定された曜日の回数を求める」ように仕様を拡張した。日曜日だけを求める特殊な機能ではなく、問題を一般化したわけである。そのようなことは「最初から指定せよ」とお考えかも知れないが、実際の開発現場では、このように「要求」は「真の要求でない」形で出てくることが多い。ユーザーの言うとおりに要求を実現すれば「毎日が日曜日」ということなりかねない。

この事後条件の定義は「ある曜日の集合DWがあって、その任意の要素dがfrom ≦ d ≦ toかつdayNumber(d) = dayOfTheWeekなる制約条件を満たしているならば、求める曜日回数は集合DWの要素の個数（cardinality）である、とまことに素気ない。曜日-setは未定

義のまま記述しているが、例えば、９６年7月8日（月）から7月３１日（水）までの月曜日の回数は、{7月8日, 7月１５日, 7月２２日, 7月２９日}という集合の要素数５である、ということである。

「その要素の個数を求めるのが問題なんだろう」とおっしゃるだろうが、「そうです」と答えるしかない。「行き当たりばったり仕様」では「問題の解き方」を書いていたが、形式手法では「まず、問題を記述する」のが目的なのである。解くのは設計工程よりあとの仕事というわけである。

2.3.2 事前条件を求める

次に事前条件を求める。事前条件は「初期設定」で簡単に設定できるか、あらかじめ問題が満たしている条件を考えることになる。ここでは、一応、以下のような事前条件を考える。

```
pre
 ∃D:日-set, ∀d∈D・ from ≦ d ≦to⇒DW⊆D
```

仕様4　事前条件

ここでは、日の集合Dがあって、その要素dがfrom ≦ d ≦toならば、事後条件で出てきた集合DWがDの部分集合になると言っている。

2.3.3 事後条件を弱める方法

事後条件を緩めて、事前条件との中間になる条件を求めるには、以下の方法がある。

(1) 積項を取り除く

項A, B, Cがあって、A∧B∧Cのような形をしているとき「積項」と呼ぶ。そのうちのひとつの項を取り除いて条件を緩めようと言うのだ。ここでは、事後条件からd ≦toを取り除いてfrom ≦d ∧dayNumber(d)=dayOfTheWeekとしてみよう。

(2) 定数を変数に置き換える

例えば、1≦d≦10というような条件があったら、1≦d≦i∧1≦i≦10に置き換えてみるという方法である。

(3) 変数の領域を広げる

例えば、1≦d≦10という条件があったら、0≦d≦100に置き換えてみるという方法である。

(4) 和項を追加する

これは、Aという項があったら、A∨Bを考えてみようという方法であるが、試行錯誤になってしまい、行き当たりばったりのプログラムと同様になりかねないので避ける。

2.3.4 不変条件と限度関数の開発

さて、dをfromからtoまで反復させてプログラムを作ろうと決心すると、反復の上限を決める「蓋」の条件は、先ほど取り除いた積項の否定を使えばよいことが分かっている。

今の場合、d > toである。また、反復の最中変わらない不変条件は、残りの積項すなわちfrom ≦d ∧dayNumber(d)=dayOfTheWeekとなる。反復が終了することを保証する「限度関数」tはt : to - dとすれば、tは常に正であり、かつdは増加するのだから、段々0に近づいていくことが分かり、反復が終了することを保証できる。

プログラムの大筋は以下のようになるだろう。

```
d := from;
do
  ？
  d := d + 1;
until  d ＞ to end
```

プログラム1　大筋プログラム

上のプログラムで？の部分は、dの曜日が指定された曜日と等しいことを保証するコードということになる。すなわち以下のようになる。

```
曜日回数（dayOfTheWeek, from, to）≡
/* 事前条件：... */
/* 事後条件：... */
variable sum : Int := 0
d := from;
/* 不変条件：∃DW:曜日-set, ∀d∈DW・
from ≦d ∧dayNumber(d) = dayOfTheWeek */
/* 限度関数 t : to - d */
do
  if dayNumber(d) = dayOfTheWeek then
    sum := sum + 1
  end
  d := d + 1;
until  d > to end;
sum
```

プログラム2　一応完成したプログラム

2.4 問題は何であるかを把握する

さて、一応正しいプログラムは完成したわけであるが、どう見てもエレガントなプログラムとは言えない。原因を考えてみると、どうも「問題は何であるか」を考えて、問題に内在する性質をうまく使っていないことにその原因があるようだ。

例えば、有名な「ケーニヒスベルグ[4]の７つの橋」の問題は、仕様５で示した事後条件（頂点n_1とn_2が繋がっていて、任意の頂点の弧が偶数個である）を満たしていれば一筆書き可能であることを示している。この場合、事後条件を発見した時点で、問題の解答も見つけ

[4]現在は、ロシアのカリーニングラードである

ていることになる。

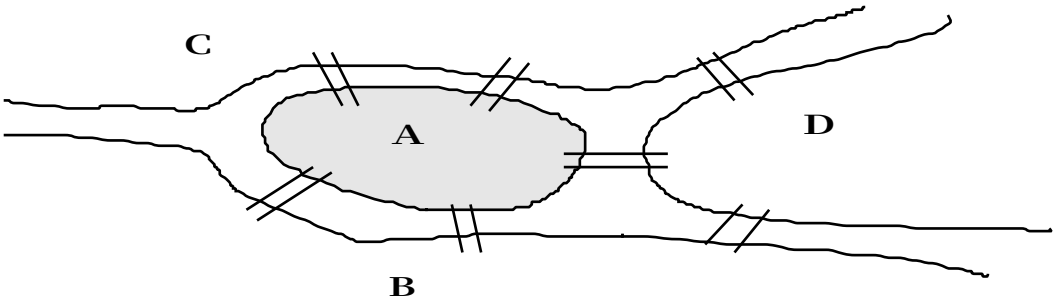

図 2　ケーニヒスベルグの７つの橋

$\forall n, n_1, n_2 \in Node$・
$card(arc(n))\backslash 2 = 0 \wedge n_1 \leftrightarrows n_2$
⇒一筆書き可能

仕様5　ケーニヒスベルグの７つの橋の答え

2.4.1 そもそも、問題は何であったか？

曜日問題の場合、ケーニヒスベルグの７つの橋のようにうまくいくであろうか？
事後条件のfromを0に、toをnに対応させ、曜日を0..6に対応させてみると...
ある性質（７で割った余りが指定されたものと同じ）を持った自然数を数えることと同じになる。ならば、漸化式で問題を定義し、数学的帰納法で証明することができるはずで、仕様ないしプログラムを再帰的に書くことができる。

2.4.2 漸化式

漸化式は以下のようになるだろう。

d0 = 1（同じ性質のとき）｜ 0（同じ性質でないとき）
dk＋1 = dk＋1 （同じ性質のとき）｜ dk（同じ性質でないとき）

仕様6　漸化式による定義

従って、仕様は再帰的に以下のように書ける。

```
曜日回数: dayOfTheWeek×Date×Date→Int
/* fromとtoの間の曜日dayOfTheWeekの発生回数を返す。*/
曜日回数（dayOfTheWeek, from, to）≡
  let
    isIt(d) ≡  if dayNumber(d) = dayOfTheWeek
      then 1 else 0 end
  in
    case to - from of
      0 → isIt(from)
```

```
      _ → isIt(from) + 曜日回数 (dayOfTheWeek, from + 1, to)
    end
  end
  pre from ≦to
```

仕様7　再帰的な仕様

ここで、「_」は0以外の場合を表す。

2.4.3 数学的帰納法

仕様７は、数学的帰納法を使って容易に[5]証明できる。

なお、脇道にそれるが、数学的帰納法による証明には納得行かない向きもあろう。特に、高校や大学の教養課程では、以下に示すペアノの公理を示さずに数学的帰納法だけを教えるので、厳密に言えば「数学的帰納法で証明できた」というのはインチキである。

ペアノの公理で自然数の定義を以下のように「帰納的」に行っているので、自然数に対応させることのできる問題に数学的帰納法が使えるのだ。

```
value
      zero : N,
      succ : N → N /* 「次」を返す関数 */
axiom ∀n, n1, n2 : N・
      [first_is_zero]  /* ゼロの次はゼロでない */
        ～ (succ(n) ≡ zero) ,
      [linear_order]  /* 「次」同士が等しければ、前も等しい */
         (succ(n1)≡ succ(n2) ) ⇒ (n1 ≡ n2) ,
      [induction]  /* 「帰納法の定義そのもの */
        ∀p : N → Bool・
         (p(zero) ∧ (∀n : N・p(n) ⇒p(succ(n))) ) ⇒ (∀n : N・p(n))
```

仕様8　ペアノの公理

2.4.4 再帰から反復への変換

末尾再帰のプログラムは、スタックを必要としない反復型のプログラムに変換できる。仕様７で表すプログラムは末尾再帰ではないが、末尾関数が結合則を満たせば、同様の変換ができることが分かっている[6]。従って、以下のプログラムが得られる。

```
variable sum: Int := 0
n := to - from;
```

[5]**通常、こう書く場合証明はさほど簡単ではない :-) が、「力仕事でできる」というニュアンスがある。**

[6]**なお、一般の再帰プログラムも、スタックに途中の結果を積めば反復型のプログラムに変換できる。**

```
while n≠0 do
      sum := sum + isIt(n);
      n := n - 1
end;
sum
```

プログラム3　反復型のプログラム

なお、一般の再帰プログラムも、スタックに途中の結果を積めば反復型のプログラムに変換できる。

2.5 問題をもう少し考えると

さて、プログラム3は、プログラム２よりも簡単に証明しつつプログラムに落とせたものの、本質的にプログラム２と同じである。「日付特有の知識」を使っていないし、エレガントではないし、n = to - fromとすれば計算量がO（n）であり、キリストが生まれた日から今日まで金曜日が何回あったかなどを計算しようとすると、言語処理系によっては難儀する。

問題をもう少し考えてみよう。実は忘れていた不変条件があった。

∀d0, d1, di∈曜日-set・
dayNumber(d0)=dayNumber(d1) ∧ from≦d0≦d1≦to⇒
∃d : Int・d1 - d0≡7×d ∧ max(di) - min(di)≡7×（card(曜日-set)－1）

仕様9　忘れていた不変条件

要するに、指定された曜日の日付同士の差は７の倍数であり、指定された曜日の日付で最もto側に近いものと、最もfrom側に近いものとの差は7×（card(曜日-set)－1）である。

変換すると

card(曜日-set)≡(max(di) - min(di)) / 7＋1

となり、求めたい答えはもう不変条件の中に出ている！

しかも、max(di)と min(di)が見つかれば、途中の反復計算は必要なくなるので、計算量はO(1)になる。実は、min(di)さえ見つかれば、max(di)はtoで代用してよい。

to - max(di) < 7∧d1 - d0≡7×d なので
(max(di) - min(di)) / 7≡(to - min(di)) / 7 なのである

さらに、実はmin(di)はもう見つかっている。プログラム３で、実行時にdoの後の最初のif文を満たすdがmin(di)である。従って、プログラムは以下のようになる。

```
曜日回数（dayOfTheWeek, from, to）≡
d := from;
while dayNumber(d)≠dayOfTheWeek do
  d := d + 1
end
(to - d) / 7＋1
```

プログラム4　効率の良いプログラム

Smalltalkで実現したプログラムは以下のようになる。

```
  occurrencesOfDayOfTheWeek: aDayNumber Between: aDate

    ¦ date fromDate toDate ¦
    fromDate := self min: aDate.
    toDate := self max: aDate.
    date := fromDate.
    [date dayNumber = aDayNumber]
      whileFalse: [date := date addDays: 1].
    ^(toDate subtractDate: date) // 7 + 1
```

プログラム5　形式手法を使ったSmalltalk版完成プログラム

実は、行き当たりばったりプログラムにも、最終版があり、以下のようになった。

```
occurrencesOfDayNumber: aDayNumber Between: aDate

¦ days ¦
days := self subtractDate: aDate.
self dayNumber = aDayNumber ifTrue: [^7 + days abs // 7].
self dayNumber > aDayNumber
  ifTrue: [days > 0
    ifTrue: [^7 - (aDayNumber - self dayNumber) abs + days abs // 7]
    ifFalse: [^(aDayNumber - self dayNumber) abs + days abs // 7]]
  ifFalse: [days > 0
    ifTrue: [^(aDayNumber - self dayNumber) abs + days abs // 7]
    ifFalse: [^7 - (aDayNumber - self dayNumber) abs + days abs // 7]]
```

プログラム6　行き当たりばったりSmalltalkプログラム最終版

2.5.1 ２つのプログラムの比較

形式手法と行き当たりばったりの２つのプログラム（プログラム５と６）を比較してみよう。

正常データのテストは両方とも一応正しい答えを返すが、異常データに対して形式手法版は無限ループに陥ることにより異常を検出した。一方、行き当たりばったり版は、ときどき間違った答えを返す。もちろん、商品としてのソフトウェアはパラメータをチェックして無限ループに陥らないようにした方がよいが、少なくとも間違った答えを返すよりは良い。

正しさを主張できるか？という観点で見ると、形式手法版は正しさを主張できるが、行き当たりばったり版は、今となってはなぜ動くかも分からない。

効率はどちらもO(1)で互角であろう。

2.6 またまた、問題を考える

さて、ここまでで一応形式手法に軍配が上がったと思うが、プログラム４（あるいは５）はまだまだエレガントではない、と思える。「反復を消せないのか」と誰でも直感的に感じるであろう。筆者は怠け者なので、「計算量O(1)だし、まあ良いか」と思っていたのだが、SEA代表幹事の山崎さんから、以下のようなエレガントな解を頂いた。以下、筆者の「翻訳ミス」があるかも知れないが、紹介する。

2.6.1 今度こそ最後のプログラム？

まず、問題を良く考えると事前条件・事後条件は以下のようになる。

```
pre
type R = {|rng [n →n / 7 | n∈Int]} /* 7で割った商の集合 */
f, t∈Int, w∈R, 0≦f≦t,
h: Int → R /* 環準同型（ring homomorphism） */

post
S = dom h(w) ∩ {f..t}・A ≡ card(S) /* Aが答え （dom h(w)≡h-1(w)） */
```

仕様10　事前条件・事後条件最終版

上の条件を見ると、もはや問題は日付問題ではなくなった。すなわち整数系を環（ring）と見て、その商環（quotient ring）への準同型写像があり、その代数系上で事後条件を満たすプログラムを作るということになる[7]。

n / 7のところに少し日付問題の面影が残っているが、ここも任意の整数に変えて問題を拡張できる。が、とりあえず、以下の文中では「日付問題」の用語を使うし7という定数も使う。

---

[7]かどうかは、この原稿を書き上げてからチェックする予定である。こうなってくると代数仕様記述言語CafeOBJとその背景にある理論（多ソート代数、カテゴリー）あたりに出動して頂くのが良いかも知れない。が、筆者の手には余るので、IPAのCafeOBJホームページ（http://www.ipa.go.jp/STC/CafeP/cafeproject.html）からマニュアルとソフトを手に入れて頂きたい。

また、そこまで行かなくとも、今はやりの秋山仁氏らが訳した「C.L.リュー著。成嶋弘、秋山仁訳。コンピュータサイエンスのための離散数学入門。マグロウヒル、1986年、ISBN=4-89501-087-2」あたりでも確認できるだろう。

さて、次に事後条件をより詳細に分析していく。

```
I = {f..t}
d = t - f + 1 /* = card(I) */
q = d / 7
r = d \ 7 /* 7で割った余り */
```

とすると、

```
q ≦ A ≦ q+1
```

が成り立つ。なぜなら、

(1) 任意の連続する７日間には、必ずw曜日がちょうど１日存在する。
(2) card(I) = 7×q + r (0≦r＜7)であるから、Iには少なくともq個の連続する７日間が存在するが、q+1個は存在しない。
(3) 余りのr日間にw曜日が存在するかも知れない。

次に、

```
x ++ y = (x + y) \ 7
x ⊥ y = max(x - y, 0)
```

として、

```
T = {h(f)..h(f) ++ (r ⊥ 1)}
```

を考える。Tは余りr日間の曜日に対応する（card(T) = r）。
すると、

```
A ≡ if w∈T then q + 1 els q end
```

ここで、

```
x -- y = if x ≧ y then x - y els x - y + 7 end
```

とすれば、

```
w∈T ⇔ (w -- h(f)) + 1 ≦ r
```

である。なぜならば

```
w∈T ⇔ {0..(r ⊥ 1)}∋w' = w -- h(f)
  ⇔ r ⊥ 1 ≧ w'
  ⇔ r ≧ (w -- h(f)) + 1
```

従って、プログラムは以下のようになる。

```
A(f, t w)≡
     let
          d ≡ t - f + 1
          q ≡ d / 7
          r ≡ d \ 7
          delta ≡ if (w -- h(f)) + 1 ≦ r then 1 els 0 end
          x -- y ≡ if x ≧ y then x - y els x - y + 7 end
     in
          q + delta
     end
```

プログラム7　形式仕様によるプログラム最終版

補助関数はあるにせよ、反復は消滅し、プログラム６までよりはエレガントなプログラムに見える。

3章 おわりに

プログラム７の中の定数７を一般の整数に拡張した場合、何に役立つのかはまだ見えてこないが、どうも問題はこれで終わりそうにない予感がしている。しばし、代数の勉強にいそしみ、後日報告したい。

いずれにせよ、開発の現場に転がっていた問題をプログラムするのに意外に手間取った。が、手間取らなければ何が起こるか？数年後に欠陥が発生し、そのころには開発したメンバーは居なくなっているか、居ても、もう問題も解き方も忘れていて、かなりの大トラブルになるであろう。実際に、昔、証券業務プログラムの開発でこのような経験を何度もしている。たかが10行ほどのプログラムでも、場合によっては数百万円の損失を招く。まして、数百万行のプログラムでは...

それに比べ、形式手法による開発は高校程度の数学知識をベースにして、最初に考えなければならないことがやや多いにしても、開発工程の後の方の手間と手戻りが少なくて済むと言えるだろう。

今回対象にした日付問題は、別に「特に難しい」問題を選んだわけではない。実際の開発現場では「もっと難しい問題」がごろごろと転がっている。そのような問題に徒手空拳で挑むのは、囲碁の名人に目をつぶって挑むようなものである。

囲碁の定石は２万個ほどあり、アマチュア初段でもこれをほぼ使いこなしている。それでもアマチュア４段には歯が立たない。しかし、アマチュア初段と４段の差などは、プロから見れば誤差の内である[8]。

形式手法の「定石」はRAISEの場合で約3000である。ソフトウェアの開発は囲碁よりは少し難しいとしても、これくらいを使いこなしてアマチュア初段格ということになるだろうか？筆者自身、まだこの程度も使いこなせないので、アマチュア５級というところだと

[8]小林元名人の筆者への忠告から。

考えているが、それでも、何とかプログラムの開発の一部に適用することができた[9]。「従って、開発現場でも形式仕様は使える」という結論が導けるが、やや強引だろうか？:-)

日付計算プログラム全体[10]は９５年１月１日から開発が始まり９６年６月末に一応完成したが、全体の工数は0.5人月ほどである。Smalltalkの行数にして717行であり、天文計算やビジネスに使える日付処理を実現している。Smalltalkにあらかじめ備わっているDateクラスより、プログラムの適用性・保守性・再利用性がかなり高いという評価結果が出ている。[11]

なお、本稿は９５年９月に直江津で行われたSEAソフトウェア・デザイン・ワークショップの夜の成果を元に、９６年７月に行われた南山大学講義およびSEA名古屋支部セミナーで話した内容を基にしている。

---

[9]筆者は、大学時代「代数」で追試を受け、教官のお情けでなんとか単位を取った。

[10]http://www.sra.co.jp/people/sahara/softwareから入手できる。

[11]IPAによるオブジェクト指向評価プロジェクトの成果として、SRAのftpサーバーでβ版を無償公開しているツールOOMによって評価した。