http://www.sal.tohoku.ac.jp/~tsigeto/ssm/05r12b.html
NCID= BA85451442 (http://webcat.nii.ac.jp)

2008-03-10

Editor= Tadahiko MAEDA.
2005 SSM Research Series 12:
Problems in Measurement and Analysis in Social Surveys.
2005nen SSM Tyousa Kenkyuukai (Sendai, Japan)
pp. 21–45 (2008)

編集= 前田 忠彦.
2005 年 SSM 調査シリーズ 12:
社会調査における測定と分析をめぐる諸問題.
2005 年 SSM 調査研究会 (仙台, 日本)
pp. 21–45 (2008)

Development of Modules for Data Reduction

— for efficient analysis of occupational history —

TANAKA Sigeto

(Graduate School of Arts and Letters, Tohoku University)

データ・リダクションのための汎用モジュールの開発

— 効率のよい職歴分析のために —

田中 重人

(東北大学大学院文学研究科)

データ・リダクションのための汎用モジュールの開発

―効率のよい職歴分析のために―

田中重人
（東北大学）

【要旨】

本稿は、複雑な構造を持つデータに対する“toolbox”アプローチを提唱する。このアプローチは、データ加工から統計分析までを統計分析パッケージひとつでおこなう“all-in-one”アプローチに代わるものであり、プログラミング言語を使ってデータ加工をおこなうプロセスを統計分析から切り離そうとするものである。統計分析パッケージの貧弱なデータ加工機能を使うよりも、本格的なプログラミング言語を使うほうが効率がよい。一般的に必要とされる機能をそろえた「モジュール」を用意しておくことで、効率をさらにあげることができる。本稿では、職歴データを加工するためのオブジェクト指向モジュールに特に焦点をあわせる。著者はこれまでに Perl を利用したモジュールを作成してきており、それらのモジュールの概要を紹介する。また、階級分類と“person-year”データを作成するためのプログラム例も紹介する。くわしい情報は http://www.sal.tohoku.ac.jp/~tsigeto/dredu/ を参照されたい。

キーワード：データ構造, Perl, 職歴分析, オブジェクト指向プログラミング

1. はじめに

1.1. All-in-one アプローチから toolbox アプローチへ

データ分析においては、「統計分析パッケージ」を使って、データの加工から統計分析まですべてを ひとつのソフトウェアでこなすのがふつうである。しかし、統計分析パッケージにおいては、主たる機能は統計分析のためのものであり、データ加工のための 機能は貧弱である。複雑な構造をもつデータの分析を効率よく進めるためには、統計分析パッケージは統計分析のために使い、データ加工には本格的 (full-fledged) なプログラミング言語を使うのがいい。さまざまな道具の入った道具箱 (toolbox) から必要な道具をその都度えらんで使うように、ソフトウェアも必要に応じて使いわけるべきなのである。

この報告では、統計分析の前処理としてのデータ加工 (data reduction) をおこなうためのシステムの設計について、基本的な考えかたを述べる。また、現在作成中の、SSM データの加工をおこなうためのモジュール群について、概要を報告する。

1.2. 歴史

筆者は、1990 年代前半に、1985 年 SSM 調査データ分析用のプログラムを SAS System の

マクロ機能を使ってつくっていた。

その後、1995 年 SSM 調査に参加した際に、このプログラムを Perl (version 4) に移植した。このプログラムは筆者の WWW ページ http://www.sal.tohoku.ac.jp/~tsigeto/ssm/script.html から入手できる。

1.3. プラットフォーム

今回のプログラム開発にあたっては、Perl version 5 (Wall, Christiansen, and Orwant 2000) を用いる。Perl は version 5 へのヴァージョン・アップの際にポインタをあつかえるようになり、OOP (オブジェクト指向プログラミング) にも対応した (Conway 2001)。これらの追加機能をふまえて、過去に Perl verson 4 で書いたプログラムをもとに、全面的に書きあらためることにした。

プログラムのテストに使ったのは Windows XP (Service Pack 2) 上の ActivePerl (http://www.activestate.com) v5.8.7 である。

2. データ・リダクション・システムの要件

2.1. Data reduction という概念

社会調査のデータはケース×変数からなる行列で表現される。統計分析パッケージのデータ・ハンドリングにおいては、通常、先頭から 1 ケースずつ処理を進めることになり、そこでは、データは単純なスカラ値を持つ変数の集合として見えることになる。

この単純な外見は、見かけ上のものである。調査票はもっと複雑な構造を持つ。たとえば、職業のデータは、職業を構成するさまざまな要素を測定した変数が集まったものである。さらに、職業の経歴 (職歴) データの場合、調査対象者が経験してきた職業が順番に並んで職歴の全体を形作るデータとなる。ほかにも、家族調査における世帯表 (日本家族社会学会 2005) や妊娠・出産等の履歴 (国立社会保障・人口問題研究所 2007) など、類似の構造を持つデータは多い。

本来複雑なデータであるにもかかわらず、形式上は単純な 1 次元のデータであるかのようにあつかわざるをえないことが、分析上の大きな困難をもたらしている。たとえば SPSS (2005 年 SSM 調査のデータは SPSS のデータ・セットのかたちで配布されている) では、単純なスカラ量以外のデータ型はサポートされていない。職歴データのような複雑なデータ構造は、あつかいようがないのである。

データ・リダクション (data reduction) とは、そのままでは集計できないような複雑なデータから、集計可能なデータを抽出してくるプロセスをいう (Wall and Schwartz 1993)。本稿の文脈では、データを統計分析にかける前に必要な前処理をおこなって、その結果を統計分析

パッケージに渡すまでのプロセスのことである。システムにデータを投入する時点では、調査票の構造を反映したかたちであつかえるようにしておき、１次元のデータとしてあつかえるところまで複雑性を落としてから 統計分析パッケージに渡すのである (図 1)。

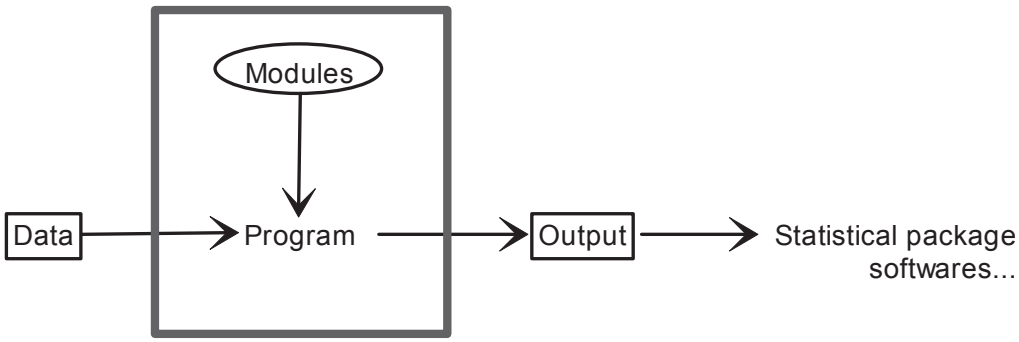

図 1. Data reduction system

2.2. コードの再利用とモジュール

こうしたシステムは複雑なものになるので、ひとつのファイルを最初から最後まで直線的に実行するようなプログラムにはできない。再利用可能なコードはプログラム中で何度も再利用することになるだろう。プログラムの見通しをよくするには、再利用するコードは別のファイルにまとめておき、必要に応じて呼び出すのがいい。こうしたコードのまとまりを「モジュール」(module) と呼ぶ。

モジュールの設計において重要なことは、モジュール同士が不用意に干渉しないように お互いの活動空間を分離しておくことだ。相手を呼び出したり、呼び出しに応答したりするための「窓口」(interface) を用意しておき、それ以外の部分では、各モジュール内部でなにがおこなわれようと、他のモジュールに影響がおよばないようにしておくのがいい。

2.3. データ型

本格的なプログラミング言語は、さまざまな「型」(type) のデータをサポートする。代表的な例はつぎのようなものである

スカラ (scalar): 単一の数値または文字列

ハッシュ (hash): 「名前」から「値」への写像

関数 (function): コンピュータに実行させる操作を記述したもの

配列 (array): データが一列に並んだもの

構造体 (structure): 一定のテンプレートにしたがって組み立てられたデータ

ポインタ (pointer): 他のデータがおかれている「場所」(address) への参照

データの加工を効率よくおこなうには、これらのデータ型をうまく使いわける必要がある。たとえば、職業のデータは「従業上の地位」「産業コード」「職業コード」などの名前とそれに 対応する値を対にして保持した「ハッシュ」としてあらわすことができる。そして、職歴は、職業を一列にならべたものなので「ハッシュの配列」である。

2.4. カプセル化と前後関係非依存性

いろいろなデータ型をあつかえるのは便利である。しかしそれは同時に面倒なことでもある。さまざまなデータ型が混在していると、いま操作の対象としているデータがどのような型のものであるかを 常に意識しながらプログラムを書かなければならなくなる。

たとえば、学歴の変数から教育年数を求める場合を考えてみよう。2005 年 SSM 日本調査のデータでは、学歴はつぎのような値をとる。

中学校: 1
高等学校: 2
高等専門学校: 3
短期大学: 4
大学: 5
大学院: 6
学歴なし: 8

学校教育における標準的な年限にしたがった場合の教育年数を求める標準的な技法は、テーブル参照 (table look-up) だろう。つぎのような変換テーブルを用意しておき、これにしたがって値を変換する。

1 → 9
2 → 12
3 → 14
4 → 14
5 → 16
6 → 18
8 → 0

Perl でこの操作をおこなうためには、つぎのように書けばいい ($ED_SSM は学歴の変数である)。

```
%eduyear = (
    1,  9,
    2, 12,
    3, 14,
    4, 14,
    5, 16,
    6, 18,
    8,  0,
) ;

$Eduyear = $eduyear{$ED_SSM};
```

このコード中、$ はスカラを、% はハッシュを、{ } はハッシュから値を取り出すことをそれぞれあらわしている。こういう妙な記号を使うのは Perl の文法の問題である。他の言語では、別のやりかたが使われるだろう。いずれにせよ、データ型を区別するためになんらかの識別子を使う言語では、これらを書きまちがえるとプログラムはまともに動かない。あるいは、初出時に「宣言」をおこなって変数等の「型」を確定する決まりになっている言語もある。そのような言語では、その変数等の「型」に応じて利用可能な操作が決まっている。その規則に違反するプログラムは実行できない

データ型の区別をつけることは、モジュールを開発しているときなら苦にならないかもしれない。そのときには、データをどのような型で表現してどのように処理すべきかが主要な関心事になっているだろうから。だが、モジュールを使用してデータを加工しているときには、データ型そのものにはあまり注意が向かない。そのような状況で、型をまちがえるたびにエラー・メッセージをみるのは、相当なストレスである。

この問題に対処するためには、モジュールの利用者がデータ型を意識しないですむようにインターフェイスを工夫する。あらゆることをスカラと関数だけで表現できるようにするのである。

上の教育年数変換のコード例は、つぎのようにも書ける

```
package SSM;

    # 関数の定義
sub Eduyear {
    ($e) = @_;    # 呼び出し元からわたされた引数
    return  9 if 1==$e;
    return 12 if 2==$e;
    return 14 if 3==$e;
    return 14 if 4==$e;
    return 16 if 5==$e;
    return 18 if 6==$e;
    return  0 if 8==$e;
    return '' ;    # どれにもあてはまらなければ空文字列を返す
}
```

この内容を SSM.pm というファイル名で保存する。このコードは、必要な時につぎのようにして呼び出せる。

```
use SSM;    # SSM.pm ファイルを読み込む
......
$Eduyear = SSM::Eduyear($ED_SSM);
```

呼び出された関数がおなじ結果を返すことを保証できるのであれば、モジュール内部での仕事のこなしかたはどのように実装してもいい。上とおなじように、モジュール内に用意した変換テーブルを参照して変換するには、つぎのようにする。

```
package SSM;

    # ハッシュの定義
%eduyear = (
    1,  9,
    2, 12,
    3, 14,
    4, 14,
    5, 16,
    6, 18,
    8,  0,
) ;

     # 関数の定義
sub Eduyear {
     ($e) = @_;    # 呼び出し元からわたされた引数
     return  $eduyear{$e} ;
}
```

モジュール内部における処理の手続きは全然ちがうものになっている。(たぶんこちらのほうがすこし速い。そのかわり記憶領域を多めに使う。) だがそのことをモジュール利用者が気にかける必要はない。関数 SSM::Eduyear() を呼び出した時にえられる出力はかわらないからだ。利用者が知っておかなければならないのは、入力と出力との関係——モジュール SSM が提供する関数 Eduyear() に教育の変数 $ED_SSM をあたえると何を返してもらえるか——だけである。

カプセル化がもたらすもうひとつの望ましい効果は、前後関係からの独立ということである。個々の関数への入力と出力との関係が常に一定であるということは、呼び出す場所の前後でどのような操作がおこなわれていようと、それはその関数の挙動に影響しない、ということだ。プログラムの一部の変更が他の部分に波及してしまうことを防げるから、エラーが起きにくくなる。また、既存のプログラムの一部を切り取って再利用することが簡単にできるようになる。

2.5. 分業の弊害

このような関数を提供するモジュールをたくさんつくっていくと、大きな問題が生じる。

どの関数がどのモジュールで定義されているかをおぼえきれなくなるのである。たとえば、一連の SSM 調査に共通するデータの操作をモジュール SSM で定義し、各々の調査に独自のデータの操作は SSM2005, SSM1995, ... などで定義することにしよう。2005 年 SSM 日本調査の教育年数のデータを操作するには、SSM::Eduyear() を呼び出すべきだろうか? それとも SSM2005::Eduyear() を呼び出すべきだろうか?

この問題を解決する方法はふたつある。ひとつはモジュールから関数を輸出／輸出 (export/import) する方法、もうひとつはオブジェクト指向プログラミング (object-oriented programming: OOP) である。

「輸入」とは、モジュールを特定せずに関数の名前を指定するだけで関数を呼び出せるようにする、ということの比喩的な表現である。たとえばモジュール SSM から関数 Eduyear() を輸入したとしよう。それ以降、プログラム中で Eduyear() と書けば、それは SSM::Eduyear() と書いたのとおなじことになる。この仕組みをうまく利用すれば、複数のモジュールのどこで何が定義されているかを知らなくてもよくなる。

オブジェクト指向プログラミング (OOP) においては、操作の対象となる「オブジェクト」に指示をあたえると、その指示を実行するためのメソッド (特殊な関数と考えてよい) をそのオブジェクトが探してきて実行する (これも比喩的な表現である)。オブジェクトは、特定の「クラス」(class) に所属していて、そのクラスが定めている規則にしたがってメソッドを探す。たとえば SSM2005::Eduyear() を探してみて、それでみつからなければつぎに SSM::Eduyear() を探す…… というふうに。クラスの設計がきちんとなされていれば、この方法も、問題の解決に役立つ。

輸出／輸入と OOP のどちらがふさわしいかは、状況 (あるいは利用者の思想) による。

操作の対象となるデータの性質が重要であるにもかかわらず、利用者側でそれを把握するのがむずかしい場合がある。たとえばファイルから多数の数値を読み込んで、それを一定の順序で変数に代入していく、というような操作 (後述の Dredu モジュールを参照) を考えよう。この場合、読み込むべき数値がいくつあるかとか、どの数値をどの変数に代入すべきかとかは、いちいちおぼえておけるようなことではなかろう。そのような情報はオブジェクトに格納しておき、それと読み込んできたデータとを一体のものとしてあつかうのがよい。そうすれば、どのような操作をすべきであるかはオブジェクト自身が決められる。このような場合には、OOP を採用するのが自然である。

一方、上でとりあげた教育年数の変換のような操作は、ふつうは利用者の頭の中に入っているであろう。学歴について分析しようとするのであれば、それぞれの学歴を取得するのにどれだけの年数がかかるかは 基礎的な知識として当然知っておくべきことである。そのようなことについて、オブジェクトをいちいち設計するのは労力の無駄である。通常は、関数をどこから呼び出すかという手続きを簡略にするために、「輸入」が適切におこなわれていれば

じゅうぶんであろう。

3. モジュール群の概要

3.1. モジュール間の関係

開発中のモジュール群は、データ・リダクション一般において必要となる汎用の操作を集めたもの (Dredu モジュール群) と SSM 調査に特有の操作を集めたもの (SSM モジュール群) にわかれている。

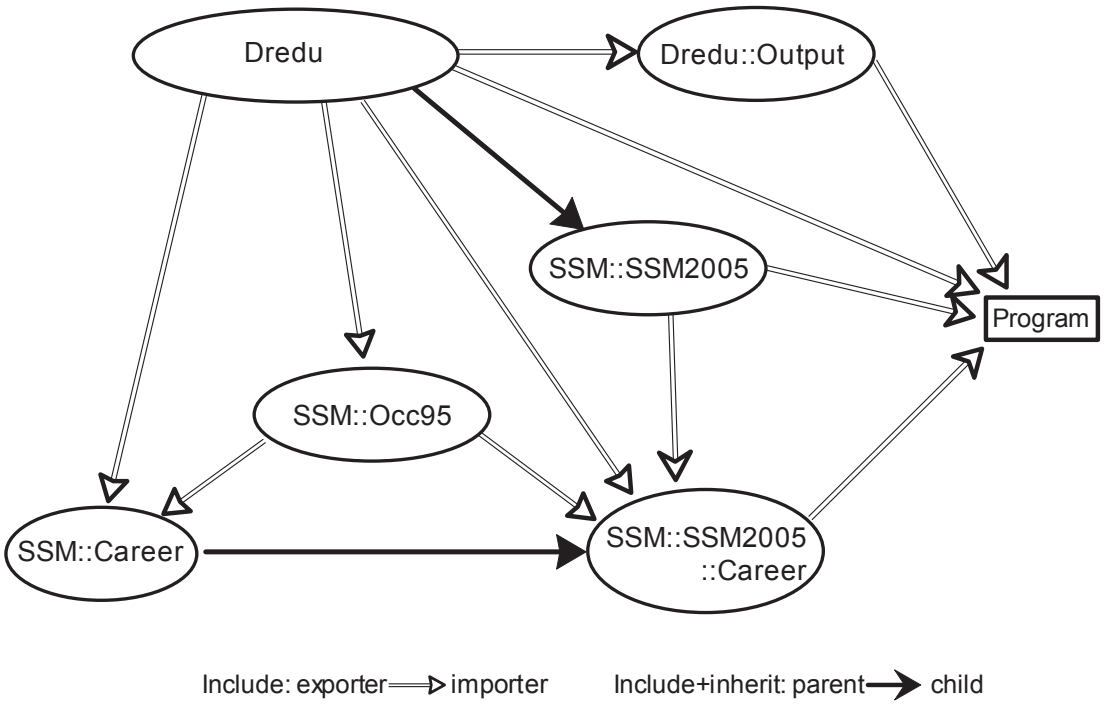

図 2. Dependency amoung modules

前者については、つぎのモジュール群を用意している

Dredu —— データの読み込み、変数への値の割り当て、欠損値の処理をおこなう。クラス Dredu を定義し、OOP にしたがったインターフェースを持つ。

Dredu::Output —— データの出力や単純集計をおこなう。クラスは定義せず、関数の輸出のみをおこなう。

後者については、つぎのモジュール群を用意している

SSM::SSM2005, SSM::SSM1995 —— 2005 年, 1995 年 SSM データについて、職業以外の分析をおこなう。Dredu クラスを「継承」(inherit) したうえで必要な機能を追加する。

SSM::Occ95 —— 1995 年版 SSM 職業分類 (1995 年 SSM 調査研究会 2006) 関連の関数を輸出する。SSM::Career などから利用される。

SSM::Career, SSM::SSM2005::Career, SSM::SSM1995::Career —— 職業・職歴の分析をおこなう。クラス SSM::Caeer を定義する。OOP にしたがったインターフェースを持つ。

これらのモジュール間の依存・継承関係を図 2 に示す。

3.2. Dredu とその派生クラス

Dredu モジュールは、データ・ファイルから 1 ケースずつデータを読み込んで、それを格納するためのオブジェクトを定義するとともに、オブジェクト操作のための各種メソッドを提供する。

データ・ファイルから読み込んだ各変数の値は、変数名に使われている文字種類にしたがって、つぎのような規則で格納される

(1) 大文字と数字と下線だけからなる変数名の場合、そのまま Perl の変数として格納される。これにアクセスするには $ を変数名の先頭につけて $Q01_1A などとする。

(2) 上記にくわえて小文字がふくまれる場合、Dredu::Var という名前空間に隔離して格納される。これにアクセスするには $Dredu::Var:: を変数名の先頭につけて $Dredu::Var::q01_1a などとする。

(3) これら以外の場合は、Dredu オブジェクト内部に格納される。

データ・ファイルからあたらしいケースを読み込むたび、これらの変数はあたらしい値に書き換えられる。

いずれの場合も、Dredu オブジェクトは、各変数へのポインタを保持しており、Value メソッド (後述) などを通じてアクセスすることができる。ただし、この記法はずいぶん長いものになる。変数名をすべて大文字 (および下線と数字) だけからなるものにしておくと、通常の Perl の変数として (短い記法で) コードできるので便利である。

SSM::SSM2005 と SSM::SSM1995 は、いずれも Dredu を継承するかたちでつくられている。基本的な部分は Dredu の機能を利用し、各調査に独自の部分をそれぞれのモジュールで

実装している。ほかのさまざまな調査についても、Dredu で定義されるメソッドを継承あるいは再定義 (override) しながら、それぞれの調査に特化したモジュールをつくっていくことができる。

3.3. SSM::Career とその派生クラス

SSM::Career は職歴データ分析に特化したモジュールである。このモジュールでは職歴のオブジェクトを定義しており、そのオブジェクトを操作するための共通のメソッドを提供する。各調査の職歴データ分析に特化したメソッドは、各調査のためのモジュール SSM::SSM2005::Career や SSM::SSM1995::Career で定義されている。詳細は 6 節で述べる。

なお、親職・配偶者職なども、職歴と共通した形式のデータである。このため、一段のみからなる職歴のようなものとみなせる。これらには、職歴用のメソッドをそのまま適用できる。

3.4. 慣習的な約束

上で述べたように、モジュールが提供する機能はすべて関数またはメソッド呼び出しによる。関数やメソッドが返す値のほうも、できるかぎり、通常の (単一の) スカラ値だけにしたいところである。しかし、現実には、通常のスカラ以外の値を返さざるをえないこともある。そこで、つぎのルールにしたがって関数を命名することで、データ型に関する示唆をあたえるようにしてある。

- 通常の単一のスカラ値 (数値または文字列または真偽) を返す関数の名前は大文字ではじめる
- それ以外の値 (配列、ハッシュ、ポインタなど) を返す関数の名前は小文字ではじめる

たとえば、valid($X, $Y) は引数のうちで欠損値でないものの配列、miss($X, $Y) は欠損値であるものについて欠損値の種類をあらわす文字列の配列を返す。どちらも、単一のスカラ値を返す関数ではないので、小文字からはじまる。

一方、All_valid($X, $Y) は、$X と $Y の少なくとも一方が欠損値であれば 0 (=偽) を、両方が非欠損値であれば 1 (=真) を返す。単一のスカラ値を返すので、大文字ではじまる関数名にしてある。

4. データ分析用の汎用モジュール

社会調査データの分析では、一般に、データを 1 ケース分ずつ読み込んで加工する方式がとられる。このようなデータ加工のプロセスで使う一般的な機能をまとめたものが Dredu モジュールである。

4.1. 変数の定義とデータ入力

モジュール Dredu を使うには、プログラム冒頭に次のように書いておく。

```
use Dredu;
$data = Dredu -> new; # データ格納用のオブジェクトの作成
```

冒頭の use は、モジュールを読み込むことをあらわす。2 行目で Dredu クラスに属する新しいオブジェクトを作成し、以下 $data として参照する。演算子 -> はクラスまたはオブジェクトを指定してメソッドを呼び出すことをあらわす。

テキスト形式 (タブ区切り) のファイルからデータを読み込むには、これに続けてつぎのように書く。DATAFILE にはデータ・ファイル名が入る。メソッド fs は、テキスト・ファイルから入力する際の変数の区切り文字を指定するメソッドである。

```
$data -> fs("¥t");     # タブ区切りに
$head = $data -> datafile( 'DATAFILE' , 1 ) ;  # ファイルを指定して 1 行読む
chomp $head;         # 末尾の改行記号を除去
@var = split( $data->fs , $head );  # 配列に分解
$data -> variables(@var);     # 変数名を定義
while( $data -> read_text ){  # 1 ケースずつ読み込む

     # ここに必要な処理を書く

}
```

変数名等の情報は、定義ファイルであたえることもできる。このファイルには、変数の定義、欠損値、データ・ファイルの名前などを一定の規則にしたがって書いておく。メソッド configfile() に「定義ファイル」のファイル名をわたして起動すると、Dredu オブジェクトに定義ファイルの内容が格納され、それ以降の処理で利用されるようになる。この機能を使うと、つぎのように書ける。

```
$data -> configfile( 'FILENAME' ); # 読み込むファイルや変数を定義
$data -> read_text;              # 1 行読み捨て
while( $data -> read_text ){ ....  # 1 ケースずつ読み込んで...
```

ただし、巨大なテキスト・ファイルを読み込んで処理するには、かなり時間がかかる。このことを考慮して、固定長のバイナリ・ファイルを読み書きする機能も用意してある。

2005年SSM日本調査データについてこの操作をおこなうためのプログラム ssm2005.pl を用意した。カレント・ディレクトリに素データのファイル (ssm2005.dat) を置いて ssm2005.pl を 実行すると、バイナリ・ファイル ssm2005.bin ができる。このときに定義ファイルの内容が標準出力に出てくるので、これを適当なファイル名 (たとえば ssm2005.ini) にリダイレクトして使うとよい。1995年SSM調査A票についても、同様のプログラム ssm1995a.pl を作成している。いずれのプログラムにおいても、欠損値の処理 (4.4節) は自動的におこなわれて定義ファイルに書き出される。変数名にふくまれるアルファベットはすべて大文字に変換される。

バイナリファイルから1ケースずつ読み込むには、メソッド read_binary を使う。

```
$data -> configfile( 'ssm2005.ini' ); # 読み込むファイルや変数を定義
while( $data -> read_binary ){ ....   # 1ケースずつ読み込んで...
```

4.2. 変数の操作

読み込んだデータは、メソッド Value('変数名') で読み出せる。ふたつ引数をあたえて Value('変数名' , 値) のようにすると、第1引数の変数へ第2引数の値を代入する。なお、変数名がすべて大文字 (と下線と数字) からなる場合、その変数名の先頭に $ をつけて参照することができるので、通常はこちらを使うほうが便利だろう。

```
while( $data -> read_binary ){       # 1ケースずつ読み込む
   print $data -> Value( 'Q01_2A' ) ; # 年齢を出力
   print $Q01_2A ;                   # 上におなじ

   $data -> Value( 'Q01_2A' , 30 ) ;  # 年齢を30歳に
   $Q01_2A = 30;                     # 上におなじ
}
```

メソッド values(変数名のリスト) によって、複数の変数の値のリストをえることができる。この場合、変数に値を代入する機能は用意していない。

```
print $data -> values( 'Q01_2YW' , 'Q01_2M' ) ; # 生年月を出力
```

変数名のリストを指定せずに values メソッドを呼び出した場合、全変数の値のリストが返る。変数の順番は、定義ファイル (あるいは variables メソッド) で変数を定義したときの順番による。この変数順は、variables メソッドを引数なしで呼び出すことで出力できる。

4.3. 別名

変数には、正規の名前以外に「別名」(alias) をあたえることができる。

```
$data -> alias( 'Q01_1' , 'Sex' );
$data -> alias( 'Q01_2A', 'Age' );
$data -> alias( 'Q24_1' , 'MarStat' );
```

```
$data -> alias( 'DANSU' , 'NStage' );
```

メソッド alias() の引数には、正規の変数名と別名の組をあたえる。

定義ファイルにつぎのように書いておいても同様のことができる (空白ではなくタブで区切ること)。

```
[alias]
Q01_1    Sex
Q01_2A   Age
Q24_1    MarStat
DANSU    NStage
```

ssm2005.pl の出力する定義ファイルには、これらの別名の指定がデフォルトで入っている。SSM::SSM2005 などの提供するメソッドには、これらの別名が定義されていることが前提になっていることがある。

4. 4. 欠損値

欠損値には、適当な値をあたえておく。この「適当な値」はなんでもよいのだが、通常は絶対値の大きい負の整数にしておくのがいいだろう。そうしておくと、たいていの演算の結果が変な値になるため、欠損値処理のまちがいをみつけやすくなる。

欠損値の定義には missing() メソッドを使う。

```
$data -> missing(
    'NA' ,    -32767,
    'OUT' ,   -32766,
    'BLANK',  -32765,
    'GHOST',  -32764,
    'YET' ,   -32763,
)
```

あるいは、定義ファイル中につぎのように書いておき、configfile() メソッドで定義ファイルを読み込んだ際に、欠損値が定義されるようにしてもよい。空白ではなくタブで区切ること。

```
[miss]
NA      -32767
OUT     -32766
BLANK   -32765
GHOST   -32764
YET     -32763
```

これらの欠損値は、つぎのような種類わけによる。

NA: 無回答

OUT: 非該当

BLANK: 素データファイルにおける空白

GHOST: 技術的な理 9 由により存在しない値 (初職の転職理由など)

YET: まだ経験されていない項目

各種メソッド・関数は、これらの欠損値種類が定義されていることを 前提にしてつくられている場合がある。

種々の演算において、欠損値が例外あつかいされる仕様にはしていない。普通に計算すると、ただの数値として処理されてしまう。必要であれば、欠損か否かを前もって判断して条件分岐させること。そのために関数 miss()、valid()、All_valid() が用意されている。たとえば年齢を 10 歳刻みにする場合は、つぎのようにすればよい。

```
if( valid($Q01_2A) ) {           # 欠損値でなければ
   $age10 = int( $Q01_2A / 10 );    # 10 で割って切り捨てる
   $age10 = 7 if 6==$age10 ;        # 70 代は 60 代あつかい
} else  {                       # 欠損値なら
   $age10 = $Q01_2A;               # 元の値をそのまま
}
```

欠損値に対応する欠損値種類を調べるには関数 Misstype() を使う。この逆の変換を行うのが Missval() である。

```
    # $FOO が非該当なら $BAR も非該当にする
if( Misstype($FOO) eq 'OUT' ) { $BAR=Missval('OUT') } ;
```

4. 5. データ出力

Dredu::Output モジュールは、データ出力のための機能を提供する。輸出される関数 output() は、変数名と変数値の組を受け取り、テキストを書き出す。区切り記号は Perl の特殊変数 $, によって指定する。

```
use Dredu:Output;
$, = "¥t" ;  # タブ区切り
.................
output(
   'NAME1', $VAR1,
   'NAME2', $VAR2,
   .................
) ;
```

右側の変数 ($VAR1, $VAR2...) の値がタブ区切りテキストとして書き出される。その際、第 1 行目には、左側の各文字列 (NAME1, NAME2...) が出力されるので、これを統計パッケージで変数名として読み込むとよい。この 1 行目がいらないなら、Perl 組み込みの print 文をふつうに使えばよい。

```
print $VAR1, $VAR2, $VAR3, .... ;
```

出力先は標準出力 (standard output: STDOUT) である。デフォルトでは、この出力は画面に表示される。ファイルに出力するには、起動時のコマンド・ラインで出力をリダイレクトすること。または、Perl の組み込みの関数 select() を使ってもよい。

欠損値はそのまま出力される。統計分析にあたって欠損値処理をおこなえるようにするには、統計分析パッケージの側で欠損値の割り当てをおこなうこと。たとえば SPSS の場合は、「シンタックス」につぎのような一文を入れておけばよい

```
missing values ALL (-32767 thru -32763).
```

関数 convert_missing_to_alphabet を実行してから output() を実行すると、欠損値はアルファベットに (値のちいさいほうから A, B, C, ... の順に) 変換されてから出力される。

```
convert_missing_to_alphabet;
output( "....... );
```

この関数は、SAS System での分析を念頭においてつくったものである。SAS プログラムの最初につぎのように書いておくと、アルファベット 1 文字からなる値を「特殊欠損値」としてあつかうことができる。

```
missing A B C D E;
```

もし欠損理由を区別しなくてよいなら、欠損値をすべて単一の値に置き換えればよい。関数 convert_missing_to_null は欠損値をすべてブランク (空文字) に、関数 convert_missing_to_dot は欠損値をすべてピリオド (.) に置き換える指示である (そのあとに出てくる output() 関数に有効)。

4. 6. 集計

Dredu::Output モジュールは、単純出力のための関数 freq(), frequencies(), class_freqlist() も輸出する。これらには、引数としてハッシュをあたえる必要がある。前 2 者に対しては、値→出現回数の 1 次元ハッシュ、class_freqlist() に対しては、値の組→出現回数の 2 次元ハッシュをあたえる。

```
while( $data->read_binary ){
    .......
    $sex{$Q01_1} ++ ;           # ハッシュ作成 (数え上げ)
    $marstat{$data->Value('MarStat')} ++;
    $sex_marstat{$Q01_1, $data->Value('MarStat')} ++;
}
freq( %sex );                  # 度数分布 (人数のみ)
frequencies( 8,  %marstat );     # 度数分布 (人数, %, 累積%)
class_freqlist(8,%sex_marstat);  # 男女別度数分布 (人数, %, 累積%)
```

集計のために大した機能を用意する予定はない。output() で出力したテキストファイルを統計分析パッケージで読み込んで分析するのが本来の使いかたである。

4.7. その他のデータ加工関数

Dredu モジュールは、つぎの関数を輸出する。

関数 Common_question() は、調査票のちがいによって同一内容の変数が別の変数名で格納されている場合 (2005 年 SSM 日本調査では留置 A/B 票にそのような質問項目がある)、それらの変数を統合するための簡便な方法を提供する。このメソッドは、引数のリストのなかから、非該当でないものをひとつ抜き出す。

```
$Household_size = Common_question( $A20_1, $B07_1 );
```

関数 Sum() は、引数のリストのうち、欠損値でないものについての合計を求める。

```
$Sum_of_valid = Sum( $A, $B, $C, ... );
```

関数 dummy() は、第 1 引数の値について、第 2 以降の引数と照らし合わせてダミー変数を作成する。

```
@foo = dummy($FOO, 0, 1, 2, 3 );
output(
    'FOO_0', $foo[0],  # $FOO==0 なら 1, そうでなければ 0
    'FOO_1', $foo[1],  # $FOO==1 なら 1, そうでなければ 0
    'FOO_2', $foo[2],  # $FOO==2 なら 1, そうでなければ 0
    'FOO_3', $foo[3],  # $FOO==3 なら 1, そうでなければ 0
);
```

5. SSM データ分析のための基本操作モジュール

5.1. オブジェクトの作成

モジュール SSM::SSM2005 と SSM::SSM1995 は Dredu を継承する。つまり Dredu とおなじかたちのオブジェクトを処理対象としており、SSM 調査データの分析に必要な特殊なメソッドをそれに追加するものである。

このモジュールを使う場合、データを読み込むためのオブジェクトは、Dredu クラスではなく、SSM::SSM2005 あるいは SSM::SSM1995 クラスに属するものとして作成する。

```
use Dredu;        # 関数を輸入
use SSM::SSM2005;
$data = SSM::SSM2005 -> new; # データ格納用のオブジェクトの作成
.....
```

5.2. メソッド

メソッド Id は、支局番号 ($NUMS), 地点番号 ($NUMP), 対象番号 ($NUMI) を合成して

ID 番号を作成する。

メソッド Age_marriage, Age_first_birth, Age_last_birth は、それぞれ 結婚時、第 1 子誕生、末子誕生の時点での回答者の年齢を求める。子供の誕生年を求めるにあたっては、田中 (1997) とおなじ計算方法を使う。

メソッド children_age, age_births, household_member は、それぞれ 子供年齢、子供誕生時の回答者年齢、世帯構成について配列のかたちにして返す。これらのメソッドは、2005 年 SSM 日本調査についてだけ用意した。子供誕生時年齢の計算は、上と同様、田中 (1997) の方法による。household_member は、もとのデータは 1 か 2 の値をとる 2 値データであるが、これを 1 (=該当者あり) と 0 (=該当者なし) に変換する。

世帯構成に関しては、自由回答データの情報も利用して、世帯類型を作成するためのプログラム SSM/SSM2005/Household.pl を作成した。このプログラムは関数 household_type を定義する。モジュールではないので、つぎのようにして呼び出す。

```
require 'SSM/SSM2005/Household.pl';
........
household_type;
```

この関数は、呼び出す側と分離されておらず、おなじ「空間」で動作する。世帯類型をあらわすさまざまな変数をつくって呼び出し側の空間を「汚染」するので、注意されたい。

5.3. 関数

SSM データ分析用の関数としては、教育年数を求める Eduyear() と、所得を 1 万円単位の値に変換する Income_10000yen() を輸出する。

```
$Eduyer = Eduyear ($ED_SSM);
$Pinc_metric = Income_10000yen($Q33A);
```

6. 職歴分析のためのモジュール

SSM::Career, SSM::SSM2005::Career, SSM::SSM1995::Career の各モジュールは、職業の変数処理のためのメソッドを提供する。これらのうち、SSM::Career は 1995 年 SSM 調査と 2005 年 SSM 調査の両方に共通のメソッドを提供する「親」のクラスを定義している。あとのふたつは、それぞれ 1995 年調査と 2005 年調査に特殊なメソッドを提供するものであるが、SSM::Career から共通処理のためのメソッドを継承しているため、必要があれば共通処理のためのメソッドを「親」クラスから自動的に呼び出す。これらのモジュールは、OOP のインターフェースにしたがって設計されているため、何も輸出しない。

6.1. オブジェクトの設計

SSM 調査における「職歴」(career) のデータは、初職から現職までの職業の変化を全て記録したものである。職業上の移動 (従業先の移動と同一従業先内での異動の両方をふくむ) があるたびにひとつの「段」(stage) として記録される。2005 年データでは最大 16 段、1995 年データでは最大 17 段のデータが記録されている。

個々の「職歴段」は、つぎのような変数の集合体である。

page: 職歴段の番号
number: 従業先の番号
industry: 産業 (SSM 産業分類コード)
size: 企業規模 (従業員数のカテゴリー)
status: 従業上の地位
job: 仕事の内容 (SSM 職業小分類コード)
post: 役職
start: 開始年齢
reason: 転職理由 (2005 年調査のみ; 従業先移動の場合のみ)
reason7: 転職理由の「その他」自由記述 (2005 年調査のみ; 従業先移動の場合のみ)
isic: 国際産業分類 ISIC (2005 年調査のみ)
isco: 国際職業分類 ISCO (2005 年調査のみ)
income: 収入の変化 (2005 年調査のみ; 従業先移動の場合のみ)

これらの変数の実体は、たとえば $Q08D02 などの名前の変数として格納されている。「職歴オブジェクト」はそれらの変数へのポインタを持つだけである。また、職歴オブジェクトは元データのオブジェクトに関する情報を持っており、職歴以外の変数の情報も呼び出せる。

職歴オブジェクトは、メソッド new() によって構築する。引数として、元データの変数へのポインタを渡すと、new() は元データから職歴データの格納されている変数名を探して、職歴を組み立てる。その際、職歴の最初に、「第 0 段」を追加する。これは初職より前の職歴にあたるものであり、全員無職、段番号や従業先番号や開始年齢は 0 にしておく。

メソッド empty によって、空の (つまり段数 0 の) 職歴オブジェクトをつくることができる。

SSM::Career (およびその派生クラス) が提供するその他のメソッドは、「オブジェクト抽出メソッド」と「メンバ配列メソッド」の 2 種類に分かれる (図 3)。以下、それぞれについて解説する。

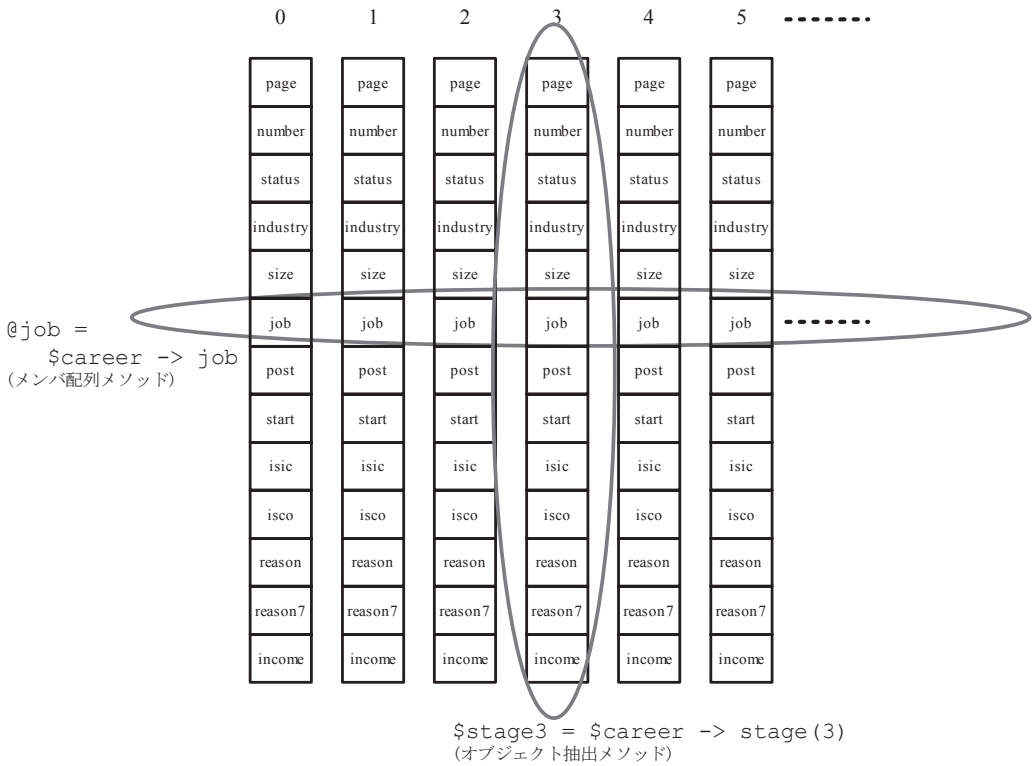

図 3. Manipulation of a career object

6.2. オブジェクト抽出メソッド

「オブジェクト抽出メソッド」は、職歴オブジェクトからいくつかの職歴段を抜き出すものである。抜き出した結果もまた職歴オブジェクトとしてあつかわれるので、メソッドを重ねて使うことができる。

メソッド valid_range は、そのケースに対して有効な職歴段の範囲を抜き出す。職歴のデータは、2005 年データでは 16 段、1995 年データでは 17 段存在する。しかしこれは最大の段数をもつケースにあわせたものであり、ほとんどのケースはこれよりもすくない段数だけが記録されていて、現職に到達したあとはすべて欠損値となっている。valid_range は、職歴データの末尾についているこのような欠損値部分を切り落とし、「有効」な範囲の職歴段だけを抜き出す。多くの場合、ケースごとにあらかじめこの処理をしてから分析するほうが混乱がすくないだろう。

```
.....
use SSM::SSM2005::Career;
$career0 = SSM::SSM2005::Career -> new($data); # 職歴オブジェクトを定義
.....
while( $data -> read_binary ){
```

```
    $career = $career0 -> valid_range;          # 有効部分のみ
    # いろいろ分析する
}
```

メソッド stage() は、引数であたえた数字に該当する番号の段を抜き出す。引数をふたつ以上あたえてもよい。

```
$third_stage= $career -> stage(3);       # 3番目の職歴段
$odd_stages = $career -> stage(1,3,5,7);  # 1,3,5,7番目の職歴段
```

メソッド first, last は、それぞれ最初・最後の職歴段を取り出す。

```
$first_stage= $career -> first;           # $career -> stage(0) と等価
$last_stage = $career0->valid_range->last; # $career0->stage($DANSU) と等価
```

firstjob は「初職」を取り出すメソッドである。就業経験がない場合は、職歴の「第 0 段」(上述) を返す。それ以外の場合は、たいていは stage(1) とおなじである。ただし、職歴第 1 段の従業上の地位が無職または欠損である場合がありえるため、それらについて例外処理をおこなう。第 1 段が無職であれば、有職の段がみつかるまで 2 段目以降を探索する。従業上の地位が欠損であれば、空の職歴オブジェクト (empty メソッドとおなじ) を返す。

結婚前の初職を取り出すのが firstjob_single メソッドである。結婚前に就業経験のない者については、第 0 段 (無職) を返す。ただし、結婚前に第 1 子が誕生している場合には、結婚時年齢ではなく第 1 子誕生時年齢 (田中 (1997) と同様の計算方法で調整する) を使って計算する。もし結婚時年齢が無回答なら、空の職歴オブジェクト (empty メソッドとおなじ) を返す。これら以外の場合は firstjob とおなじ結果を返す。

メソッド at_age (下限年齢, 上限年齢) はふたつの引数をとり、下限年齢から上限年齢までの間に経験された職歴段を抽出する。

```
$career_young = $career -> at_age(20,29);  # 20代の職歴
```

引数をひとつだけ指定することもできる。その場合には、その年齢に該当する職歴段を抽出する。ただし、移動がおこった年齢にあたっている場合、複数の職歴段が抽出されることがある。職歴段をひとつに限定したいなら、first (移動前の職歴段を採る) や last (移動後の職歴段を採る) などを利用する。

```
$Age_lc = $data->Age_last_birth;  # 末子誕生時年齢
$career_lc = $career -> at_age($Age_lc) -> last;
```

メソッド each は、職歴オブジェクトを個々の職歴段に分解し、それぞれを要素に持つ配列を返す。この配列を操作することによって、個々の職歴段を操作することができる。

```
foreach $x ($career->each) {
```

```
    print $x -> status;  # 従業上の地位を出力（後述の status メソッドをみよ）
}
```

6.3. メンバ配列メソッド

「メンバ配列メソッド」は職歴オブジェクトにふくまれる職歴段から各フィールドの値をとりだし、あるいはそれらを操作して一定の値に変換したものの配列を返す関数である。返値はオブジェクトではなく単なるスカラ値の配列なので、メソッドを重ねて使うことはできない。

各職業変数の値を取り出すメソッドには、上述の各フィールドの名前 (page, status, industry, job, size など) がそのままつけてある。これらの名前を指定すれば、その名前に該当する変数の値が配列になって返る。

```
print $career->status;  # 従業上の地位の変化を観察する
```

上述の each メソッドを使ってつぎのように書いても同様の結果がえられる。

```
print map {$_->status} $career->each;  # 職歴段個別に status メソッドを適用
```

オブジェクト抽出メソッドと重ねて使うと、次のような操作が可能である。

```
$Age_lc = $data->Age_last_birth;
$career -> at_age($Age_lc) -> last -> status ;
```

このコードは、つぎのように動作する

- 末子誕生時の職歴段をとりだす
- 取り出された職歴段が複数である場合には、最後のものを採る
- その職歴段の「従業上の地位」の値を取り出す

メソッド have_job は、status で抜き出した「従業上の地位」の値のうち、有職のものだけについての配列を返す。この値を条件分岐に使うことができる。

```
if( $career->at_age($Age_lc)->last->have_job ){
    # 末子誕生時に有職だった者だけについての処理を何か書く
}
```

メソッド job8598 は、1995 年版 SSM 職業分類 (1995 年 SSM 調査研究会 2006) について、1985 年以前の SSM 調査データとの整合をとるために 管理職に関する変換 (2005 年社会階層と社会移動調査研究会 2007, pp. 89–94) をおこなった職業小分類コード (3 桁) を返す。

メソッド class_job8 は、「SSM 職業大分類」(1995 年 SSM 調査研究会 2006, pp. 101–5) の 8 分類を返す。

```
$j8 = $career -> firstjob -> class_job8;
print '専門' if $j8==1;
print '管理' if $j8==2;
print '事務' if $j8==3;
print '販売' if $j8==4;
print '熟練' if $j8==5;
print '半熟' if $j8==6;
print '非熟' if $j8==7;
print '農林' if $j8==8;
print '無職' if $j8==Missval('OUT');
# これら以外の場合は職業コード欠損
```

職業小分類コードについては、あらかじめメソッド job8595 によって管理職に関する変換がおこなわれる。もし変換前のコードに基づいた大分類が必要であれば、SSM::Occ95 で定義されている関数 occ8() を呼び出して使うこと。

```
........
use SSM::Occ95;
........
while( $data->read_binary ){
    $career = $career0->valid_range;
    print map { occ8($_) } $career->job;  # メソッド job の返値を occ8()で変換
}
```

メソッド class_new8() は、総合職業分類 (原・盛山 1999) を作成する。引数には、「小企業」の上限を指定する (「企業規模」の変数値を使用する)。6 (300 人未満まで) を指定すると、原・盛山 (1999) の使用している分類とおなじ結果がえられる。値のあたえかたは次のとおりである。

1: 専門
2: 大企業・官公庁の管理・事務・販売
3: 小企業の管理・事務・販売
4: 自営業の管理・事務・販売
5: 大企業・官公庁の熟練・半熟練・非熟練
6: 小企業の熟練・半熟練・非熟練
7: 自営業の熟練・半熟練・非熟練
8: 農林漁業
Missval('OUT'): 無職
その他の欠損値: 職業変数のどれかが欠損

メソッド class_status4 は、従業上の地位 (および一部は職業分類) に基づいて 4 つの階級に分類したものを返す。

1: 経営者・自営・家族従業者
2: 常時雇用されている一般従業者
3: 臨時雇用・パート・アルバイト・派遣社員・契約社員・内職

9: 無職

これらのメソッドを組み合わせて、さまざまな職業階級分類をつくることができる。たとえば、無職を 0、自営を 9、非正規雇用を 10 として、常時雇用の場合のみ総合職業分類を適用するあたらしいメソッドをつくるには、つぎのようにする。

```
sub class10 {
    my ($self) = @_;                # 操作の対象になるオブジェクト
    my $N8 = $self -> class_new8(6);  # 総合職業分類
    my $S4 = $self -> class_status4;  # 従業上の地位 4 分類
    my $class;
    if( miss($S4) ) { $class =$S4; }  # 欠損値
    elsif( 1==$S4 ) { $class = 9; }   # 自営など
    elsif( 2==$S4 ) { $class =$N8; }  # 常時雇用されている一般従業者
    elsif( 3==$S4 ) { $class =10; }   # 臨時雇用・パート・内職など
    elsif( 9==$S4 ) { $class = 0; }   # 無職
    else{ warn( "ERROR: Illegal value $S4" ); }  # 以上で全部のはず
    return $class;
}
```

7. 使用例

7.1. 結婚前初職と末子誕生時の職業移動

結婚前の初職と末子誕生時との間の職業移動をとらえるには、つぎのようにするとよい。

```
$Fjs10 =$career -> firstjob_single -> class10;  # 上記の class10 メソッド
$Age_lc = $data->Age_last_birth
$Lc10 = $career -> at_age($Age_lc) -> last -> class10;
output (
    'Id', $data->Id,
    'Sex', $data->Value('Sex'),
    'Age', $data->Value('Age'),
    'MarStat', $data->Value('MarStat'),
    'MarAge', $data->Age_marriage,
    'Age_lc', $Age_lc->Age_last_birth,
    'Fjs10', $Fjs10,
    'Lc10', $Lc10,
);
```

ID, 性別, 年齢, 婚姻状況, 結婚時年齢, 末子誕生時年齢, 結婚前初職分類, 末子誕生時職分類がファイルに出力される。

7.2. 職歴の “parson-year” データ

保田時男氏による SPSS シンタックス ssm05py.sps (2007-07-18) の職歴部分 (およそ 400 行のコード) と同等のことをおこなう。15 歳から 70 歳までの各年齢時の職業に関する変数 (12 個) を書き出したファイルが出力される。出力されるデータは 1 行が回答者ひとり分である。各回答者を特定するための変数 3 個と各年齢時の職業に関する変数 (70−15+1)×12 = 672 個

をあわせた 675 個のフィールドが各行に出力される。

```
....... 〔省略〕 ......
# 変数名のリスト
@field = qw(
    page number reason industry isic size
    status job isco post start income
);
@vname = qw(ISNUM ISN ISA ISB ISBI ISC ISD ISE ISEI ISF ISPA ISG);

# 先頭に変数名を出力
@header=();
foreach $a(15..70){
    push @header, map{ $_ . $a }@vname;
}
print 'NUMS' , 'NUMP' , 'NUMI', @header;

# 1 ケースずつ処理
while( $data -> read_binary ){
    $career = $career0 -> valid_range;   # 有効な職歴段のみ
    @result=();
    foreach $a(15..70){
        $j = $career->at_age($a)->last;  # 各年齢時の職歴段
        foreach $f( @field ){
            push @result, $j -> $f;     # 各フィールドの値を配列に格納
        }
    }
    print $NUMS , $NUMP , $NUMI, @result;   # 出力
}
```

付録: Perl 語彙

Perl で使われる記号と予約語について、簡単に説明する。本稿を理解するための最小限の説明にとどめているため、不正確な場合がある。

```
# ―― ここから行末までコメント
; ―― ひとつの「文」の終端
$ ―― 単一のスカラ値を持つ変数
% ―― ハッシュ全体
@ ―― 配列変数
@_ ―― 関数・メソッドへわたされた引数の配列
:: ―― モジュール名の区切り
-> ―― メソッドの呼び出し
'...' ―― 文字列
"¥t" ―― タブ記号
++ ―― 数値をひとつ増やす
= ―― 左辺に右辺の値を代入
== ―― 両辺が (数値として) 等しい
eq ―― 両辺が (文字列として) 等しい
foreach $i(...){...} ―― () 内のリストを順番に $i に代入して {} 内のコードを繰り返す
map {...} ―― 配列の全ての要素に対して {} 内のコードを実行
my ―― { } で囲まれたブロック内だけで通用する名前を宣言する
package ―― モジュール等が使う「名前空間」を定義
print ―― 標準出力に出力
```

push —— 配列の末尾にあたらしい要素を追加
qw(...) —— ( ) 内の文字列を、空白で区切られたリストに変換
require —— 他ファイルの Perl プログラムを読み込む
return —— 関数・メソッドの処理を終え、「返値」を返す
sub —— 関数・メソッドの定義
use —— モジュールの読み込み
warn —— 警告を表示する
while(...){...} —— ( ) 内の条件式が真である限り {} 内のコードを繰り返す

文献

1995 年 SSM 調査研究会. 2006. 『SSM 産業分類・職業分類 (95 年版)』(修正版).
2005 年社会階層と社会移動調査研究会. 2007. 『2005 年 SSM 日本調査 コード・ブック』.
Conway, Damian; 山根ドキュメンテーション (訳). 2001. 『オブジェクト指向 Perl マスターコース』ピアソン・エデュケーション.
原純輔・盛山和夫. 1999. 『社会階層: 豊かさの中の不平等』東京大学出版会.
国立社会保障・人口問題研究所. 2007. 『わが国夫婦の結婚過程と出生力: 出生動向基本調査 第 3 回』厚生統計協会.
日本家族社会学会 全国家族調査委員会. 2005. 『第 2 回家族についての全国調査 (NFRJ03): 第 1 次報告書』.
田中重人. 1997. 「高学歴化と性別分業」『社会学評論』48: 130–42.
Wall, Larry, and Randal L. Schwartz; =近藤嘉雪 (訳). 1993. 『Perl プログラミング』ソフトバンク.
Wall, Larry, Tom Christiansen, and Jon Orwant. 2000. *Programming Perl* (3rd ed). Sebastopol, CA: O'Reilly.

Development of Modules for Data Reduction: For Efficient Analysis of Occupational History

Sigeto TANAKA

Tohoku University

This paper proposes a "toolbox" approach for analyses of data with a complex structure. In this approach, you use any programming language for preprocessing before statistical analyses. It is an alternative to the "all-in-one" approach in which you use a statistical package to conduct both data processing and statistical analyses. It is efficient to use a full-fledged programming language instead of poor data processing functions of statistical packages. And it is more efficient to utilize "modules" prepared to meet common needs. The paper focuses on the data structure and the object-oriented modules to process occupational history. The author has developed some modules in Perl, and introduces the basic feature of the modules. This paper also includes some programs to make class categories and parson-year data of personal histories. Further details are available from http://www.sal.tohoku.ac.jp/~tsigeto/dredu/.

Keywords and phrases: data structure, Perl, career analysis, object-oriented programming