2006 年 6 月 22 日

C のソースからプロトタイプ宣言を作成

新潟工科大学 情報電子工学科 竹野茂治

1 はじめに

本稿では、C 言語のソースから関数のプロトタイプ宣言を作成するツールを考察します。

awk は元々行単位のフィルタですから、[2], [3] のような対話型処理よりも、このような文書処理、テキストデータ処理を得意としています。しかも、今回のように入力 1 行だけでは判断できない場合はそれなりに難しく、多少アルゴリズムを考え、ちゃんとプログラミングする必要がありますので、そのような一例として紹介します。

2 プロトタイプ宣言について

ここでは、少し C 言語のプロトタイプ宣言について説明します。

AWK や C 言語では、自分用の関数 (サブルーチン) を作ってそれを自分で使うことができますが、その場合 C 言語では、

1. 関数を使う場合は、それを使う前にその関数が定義されていないと、その返値は整数型 (int) であると見なされてしまう。
2. 関数の定義を前に置く代わりに、空の宣言で代用してもよい。

のようになっています。よって、少なくとも int 以外の値を返す関数は、

```
double f1(double x)
{
    ....
}

int main(void)
{
    double x;

    x=f1(1.5);
    ....
```

のように、関数を使う前に定義するか、

```
double f1(double x);

int main(void)
{
    double x;

    x=f1(1.5);
    ....
}

double f1(double x)
{
    ....
}
```

のように、使う前に空の宣言だけを置く必要があります。この空の宣言を プロトタイプ宣言 といいます。

プロトタイプ宣言は、関数の記述位置を考えなくていいようにするだけでなく、コンパイル時に引数のチェックを行うようにもなっていて、よって、引数の数が足りないとか、引数の型が不適切である、といったバグを検出することを可能にする仕組みで、現在の C では標準的な書き方となっています。

なお、通常 C 言語のソースの先頭に “#include <stdio.h>” のようなインクルードファイルの指定を書くと思いますが、インクルードファイルの目的の一つは標準ライブラリに対するプロトタイプ宣言です[1]。

プロトタイプ宣言は、ソースの先頭に書くか、あるいは別なファイルに書いておいてそのファイルをインクルードするか、のどちらかの方法が取られます。

プロトタイプ宣言を作るには、関数の定義の先頭の部分をコピーして、最後に ‘;’ をつければいいのですが、たくさん関数がある場合は、それを手作業でいちいちやるのは面倒です。そこで、そのような作業を行うツールを AWK で書く、というのが本稿の目標です。

[1] 実際に、いくつかのインクルードファイルを見てみると、多くのプロトタイプ宣言が書かれていることがわかるでしょう。

3 関数定義部の取り出しについて

人によって C のソースの書き方は色々あるので万人向けのツールを作るのは難しいですが、自分の (今の) ソースの書き方に限定すれば[2]関数定義の先頭を見つけるのはそれほど難しくはないと思います。

私は、C の関数の書き方は、例えば以下のようにしています。

```
int f1(int x,int y)
{
    int p;
    ....
}

int f2(double *long_name_variable_1, int *long_name_variable_2,
       char *long_name_variable_3, float *long_name_variable_4,
       long *long_name_variable_4)
{
    int p;
    ....
}
```

おおまかには、次のようなルールで書いています。

1. 関数の返り値の型と関数名、およびそれに続く小カッコ '(' は同じ行に書く。
2. 関数の返り値の型は、インデントなしに行の先頭から書き始める。
3. 関数の引数部が長い場合、それはすぐ次の行にインデント (行頭にタブやスペースを入れること) をして書き続ける。
4. 引数部の終わりの小カッコ ')' は、引数部の行末につける。
5. 引数部の行の次の行に、インデントなしで中カッコ '{' のみの行を書き、関数本体は、その次の行からインデントをつけて書く。
6. 関数本体の終わりは、インデントなしで中カッコ '}' のみの行を書く。

この私の (現在の) ルールによれば、行頭にインデントなしで (中カッコでなく) 文字が始まっていればそれは関数定義部となりそうですが、実際の C のソースでは必ずしもそうではなく、これ以外にも行頭から文字が始まる場合があります。私が関数定義部以外に行頭から書くものとしては、以下のようなものがあります。

[2]ソースの書き方は時間とともに変わるので、そうなったらこのツールもそれに合わせて書き直せばいいでしょう。

- 外部変数宣言

```
int external_variable1;
double ex1=0.0,ex2=1.0;
```

- マクロ定義、インクルードファイルなどのプリプロセッサ命令

```
#include <stdio.h>
#define add(x,y) ((x)+(y))
#ifdef HOGE
```

(長いマクロ定義では、2 行目以降にインデントを入れて書く)

- 構造体、共用体、enum の定義

```
typedef struct mylist {
  int n;
  struct mylist *next;
} Mylist;
enum Error_number { Err_file, Err_input, No_Err=0 };
```

- コメント

```
/* これはコメント */
/* これもコメント
 * これもコメント
 */
```

- (既にある) 関数のプロトタイプ宣言

これらがあるので、単純に行頭が文字列でも関数定義とは限りませんが、コメントやプリプロセッサは、行頭が記号 (‘#’, ‘/’) ですからこれは考慮する必要はなく、構造体などは、その行に小カッコ ‘(’ が含まれないのでそれで判別できます。外部変数は、小カッコが含まれるもの、例えば

```
char (*a)[10];
double (*f)(double);
```

みたいなものを決して使わないとはいえませんが、これは行末が ‘)’ では終わらず、‘;’ で終わりますのでそれで判別できます。プロトタイプ宣言が既にある場合も同様です。

よって、今回は、

1. 行頭がアルファベット、または ‘_’ (アンダースコア) で始まっていて、その行に ‘(’ が含まれ、

2. その行末が、‘)’ であるか、またはそれ以下にインデントされている行がいくつか続き、その最後の行の行末が ‘)’ となっていて、

3. その次の行が ‘{’ のみの行

である場合にこの部分が関数定義であると見なし、それを抜き出して ‘;’ を付加して出力することにします。

結局、この関数定義部分の判別は複数の行によってなされることになるので、AWK でも結構面倒です。

4 行のチェック

今回のプログラムでは、AWK の正規表現によるパターンマッチ機能を利用して、入力行がどのような行であるかを調べながら進行するものを考えますが、行のチェックは次のようなことを行います。

- ア: 行の先頭文字がアルファベットかアンダースコアで始まり、‘(’ が含まれる行
- イ: 行の最後が ‘)’ で終わる行
- ウ: 行の最初がスペースかタブで始まる行 (インデントされている行)
- エ: 行の最初が ‘{’ で、それだけの行

「ア」は関数定義の先頭、「イ」は定義の終わりを意味します。「イ」は「ア」と同じ行かもしれませんし、その後かもしれません。「ウ」は「ア」の関数定義の続きの 2 行目以降の部分で、その終わりは「イ」になっているはずです。逆にそうでなければ関数定義ではないと見れます。そしてさらに、その「イ」の次の行が「エ」になっている場合に関数定義と見なして出力を行なうことにします。

ただし、例えばソースにすでにプロトタイプ宣言が含まれている次のような場合、

```
int f1(char *name, double x);
int f2(char *name)
{
    ...
```

この先頭行で「ア」のチェックを行った後で「イ」のチェックを行いますが、これは「イ」ではないので、今度は次の行を読んで「ウ」のチェックを行うことになります。そうす

るとこの 2 行目は「ウ」ではないので、この 1 行目が関数定義ではないことがわかります。

しかしこの方法だと、2 行目を読んだ後で 1 行目が関数定義でないとわかり、しかもこの 2 行目を元に戻って「ア」の処理にかけないといけないことになってしまいます。

よって、それよりも 1 行目の段階でこれが関数定義ではないと判別してしまう方が楽ですから、必要に応じて、

- オ: 行末が ';' である行

の判別を行って、現在考えている部分が関数定義ではないことを判断してやることにします。

5 getline

行単位のフィルタとしての AWK スクリプトは、

```
{
  実行コード
}
```

のように書くと、基本的に AWK は入力を 1 行ずつ読み込んで、その各行に対してこの実行コード部分を適用します。すなわち、入力の 1 行目を読み込んでそれに対してこのコード部分を適用し、入力の 2 行目を読み込んでそれに対してこのコード部分を適用し、ということを入力の最終行まで行うわけです。

しかし、今回のような、複数行で一つの意味を持つなどの場合にはその形式ではプログラムを書きにくいこともあるため、コードの途中で現在行を捨てて次の行を読み出す `getline` という命令も用意されています。

- **getline**: 何も指定しないで実行すれば、現在入力中のファイルから次の行を読み込む ($0 等のセットも行う)。ファイルを指定して現在の入力とは別のファイルから行を読み込んだり、外部のプログラムを実行してその標準出力から行を読み込むことも可能。

ある意味で行単位のフィルタとしての AWK は、getline を実行してはコード部分を実行し、また getline を実行してはコード部分を実行し、という動作を繰り返していると見ることもできます。つまり、コードの実行部分の前に必ず暗黙の getline を実行して

いる、という形式になっています。よって、その暗黙の getline も含めてアルゴリズムの考察を行うことにします。

また今回は、getline 同様、行単位フィルタとして特徴的なジャンプ命令である next も利用します。

- **next**: 現在の入力行に対する処理をそこで中断し、次の入力行を読み込んで (暗黙の getline)、コード部分の先頭に戻る。

6 単純なアルゴリズム

まずは、今回のプログラムを最も単純に考えてみることにします。

(1) 1 行入力する (暗黙の getline)
(2) 現在行が「ア」の行かチェック
  (3) 「ア」ならば、その行を保存して
  (4) その行が「イ」であるかチェック
    (5) 「イ」ならば、次の行を取得しそれが「エ」であるかチェック
      (6) 「エ」ならば、保存したものに ';' をつけて出力する
      (7) 「エ」でなければ、(2) に戻る
    (8) (4) が「イ」でなければ「オ」であるかチェック
      (9) 「オ」ならば、(1) に戻る
      (10) 「オ」でなければ、次の行を取得しそれが「ウ」であるかチェック
        (11) 「ウ」ならば、その行を保存し (4) に戻る
        (12) 「ウ」でなければ、(2) に戻る
  (13) (2) が「ア」でなければ、(1) に戻る

これをフローチャートにしたのが図 1 です。(7) と (12) のところでは、新しい行を取得するのではなく、現在持っている直前に取得した行に対して「ア」のチェックを行うため、先頭、すなわち暗黙の getline の前に戻るのではなく、getline と「ア」の間に戻るようにしています。

これを AWK の疑似コード (おおまかなコード) として書いてみると、例えば次のようになります。

```
{
    while(1){
        if(!「ア」) next
        (リセットして保存)
        while(1){
```

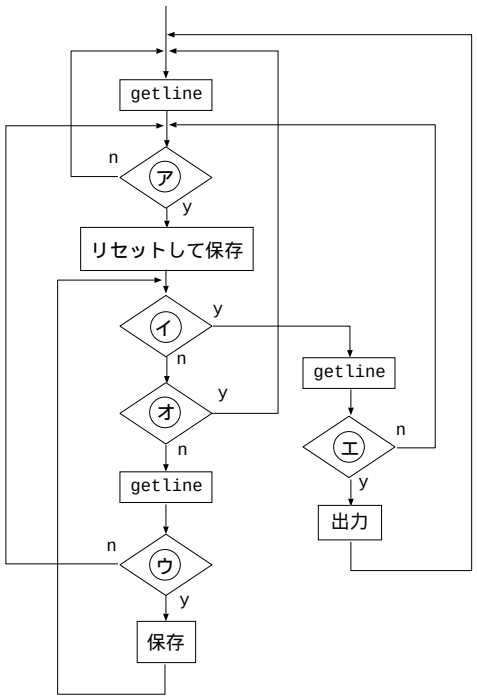

図 1: 単純なアルゴリズムのフローチャート

```
            if(「イ」){
                getline
                if(「エ」){ (出力); next }
                else break
            }
            else if(「オ」) next
            else{
                getline
                if(!「ウ」) break
                (保存)
            }
        }
    }
}
```

このコードでポイントとなるのは、2 つの while 文です。“while(1)” は、C 言語でもよく使われる手法ですが、条件判断が常に真 (つまり 1) であるループなので、無限ループを作っていることになります。

「ア」の条件判断をこの while ループの外に出して先頭の行に書いてしまうと、「ア」の判断の前には必ず行の取得 (暗黙の getline) が行われることになってしまい、(7) や (12) を実現できません。よって、それを while(1) で囲むことで行を取得せずに「ア」の判断部分に戻れるようにしていて、行を取得してから「ア」の判断をしたい場合は

next を使うようにしてその while 文から抜け出すようにしています。

2 つ目の while(1) 文は、「ウ」のループ構造の実現のためです。フローチャートを見ればわかりますが、「ウ」で保存した後は次の行や「ア」の上に戻るのではなく、「イ」の上に戻りますので、この流れがひとつのループ構造になっています。よって、そこをなんらかのループとして書く必要があるわけです。ここでは、「ウ」でない場合にこの 2 つ目の while ループから抜けて「ア」の上に戻る、つまり (12) を実現するために break 文を使って内側の while ループから抜け出しています。

「イ」の後の処理である (5),(6),(7) も while ループの中に書きましたので、(7) では break 文で内側のループから抜けるようにしていますが、「イ」だった場合にまず break して内側の while ループから抜け、その外でその後の処理 (5),(6),(7) の処理を書く、という手もあります。ただし、その場合その部分は、(12) で break したのではないことを区別するためのコードを書く必要があります。

7 パターンマッチ中心のアルゴリズム

6 節のプログラムの構造は、単純なフローチャートから書いているので考えやすいかもしれませんが、getline を内部で 2 回使用していますし、while(1) ループを 2 重に使っていて、AWK スクリプトとしてはあまり読みやすいものではなく、処理のわかりにくいものになっています。今度は、より AWK らしいアルゴリズムを考えてみます。

「ア」～「オ」は、行の種類の判別になっていますが、そのような行の性質ごとに処理を書くのはより AWK らしい書き方だと言えます。そもそも、AWK の行処理部分は、

```
{
    コード部分
}
```

以外にも、

```
条件 1 {
    コード 1
}
条件 2 {
    コード 2
}
...
```

のように、条件を満たす行に対してのみ実行するコードを別々に書くことができるようになっていて、さらにその条件が行全体 ($0) に対するパターンマッチングである場

合は、「$0 ~ /パターン/」という条件式を、単に「/パターン/」のように書けばよいことになっています。

これを利用して、「ア」~「オ」のパターン毎の処理を考えてみることにします。このうち、「ア」「ウ」「エ」は行頭に関係していてお互いに排他的[3]ですし、「イ」「オ」は行末に関係し排他的なので、それらに分けて考えてみます。

なお、「ア」や「ウ」の処理の後には、同じ行を「イ」(や「オ」) のチェックにも回さなければいけないことに注意します。よって、そのために「フラグ」(フラグ 1) を使って、「イ」に「ア」や「ウ」の処理の後であることをわからせることにします。

(1) 「ア」ならば、リセットして保存してフラグ 1 をセット

(2) 「ウ」ならば、現在保存した行がなければ next、あればこの行も保存してフラグ 1 をセット

(3) 「エ」ならば、それが「イ」の次の行であれば (フラグ 2 がセットされている場合) 保存しているものに ';' をつけて出力してリセットして next、そうでなければリセットして next

(4) 「イ」ならば、「ア」か「ウ」の次ならば (フラグ 1 がセットされていれば) フラグ 2 をセットして next、されていなければ next

(5) フラグ 1 でなければリセット

ここでの「リセット」は、保存しているものをクリアすることとフラグ 2 をクリアすることを意味します。これをフローチャートにしたのが図 2 です。この場合は、6 節にはほとんど必要なかった「リセット」という作業、および「フラグ」というものが含まれていて、逆に明示的な (暗黙のもの以外の) getline はなく、「オ」もありません。

フラグは上にも説明しましたが、次のような意味を持つ 0 か 1 の値を取る変数です[4]。

- フラグ 1 : 「ア」か「ウ」を通過したかどうかを表す
- フラグ 2 : 前の行が「イ」であったことを表す

この行毎の処理の場合、例えば「イ」である行を見つけたとしても、それ単独では、その行が「ア」を通過してきたものなのか、「ウ」を通過してきたものなのか、またはそのいずれも通過しなかったものなのかが判別できません。しかし、「ア」か「ウ」を通過した場合のみ「イ」は意味がありますので、そのためにフラグ 1 が必要になります。

[3] 排他的とは、同時に成立しないことを意味します。

[4] このような役割の変数は、立てるか立てないかで 2 つの状態を表現する旗にちなんで、よく フラグ (旗) と呼ばれます。

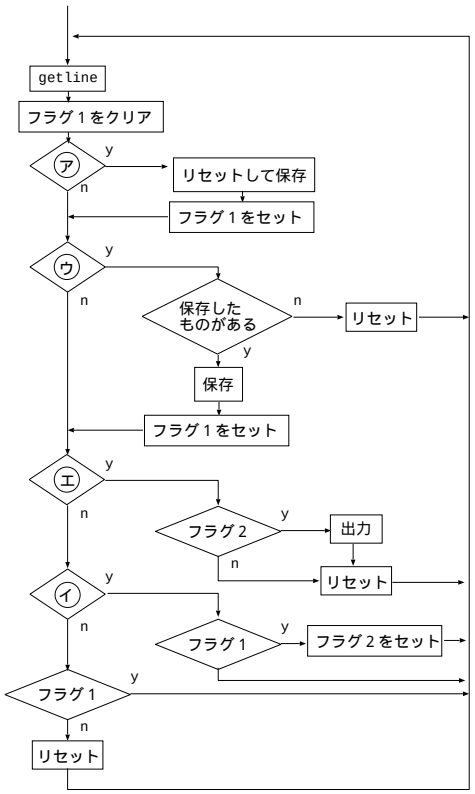

図 2: パターン中心のフローチャート

同様に「エ」は「イ」の次の行でのみ意味を持ちますので、それを示すためにフラグ 2 が必要となります。

「ウ」や「エ」のリセットは、関数定義ではないと判別された場合に各変数を初期化して、また一から関数定義を探す、という目的の為に入れてあります。そのリセットがないと、関数定義ではないところで「エ」が出力してしまう恐れがあります。最後の「フラグ 1 でなければリセット」の部分も同様の理由で入れてあります。

これを疑似コードにしてみると、例えば以下のようになります。

```
{ flag1=0 }
「ア」{ (リセットして保存) ; flag1=1 }
「ウ」{
    if( (保存したものがある) ){ (追加して保存); flag1=1 }
    else{ (リセット) ; next }
}
「エ」{
    if(flag2==1){ (出力) }
    (リセット); next
}
「イ」{ if(flag1==1) flag2=1; next }
```

```
(flag1==0){（リセット）}
```

この最初の実行コードブロックには条件がついていませんが、条件のついていないコードブロックは、すべての行が実行対象になります。

先頭のブロックや「ア」、「ウ」の実行ブロックは、next がない、あるいはあっても必ずしも実行されるとは限りませんので、そのブロックが終ったら、同じ入力行に対して次のブロックの条件判断に動作が移ります。これに対し、「エ」、「イ」のブロックでは最後に必ず next が実行されますので、これらのブロックに入った場合はその下のブロックは実行されずに次の入力行に動作が移ります。

フラグが 2 つあるので少しわかりにくいかもしれませんが、特にループも使用しておらず、AWK スクリプトとしてはすっきりした構造になっています。

なお、このスクリプトだと「ア」のチェックをしたあとで「ア」でない場合にも「ウ」であるかどうかをチェックしていますが、「ア」「ウ」「エ」は互いに排他的なので、これは無駄です。つまり、フローチャートは、図 2 よりもむしろ図 3 の方が望ましいかもしれません。これは、AWK の 「if～ else if～ else」のような書き方を使えば実現で

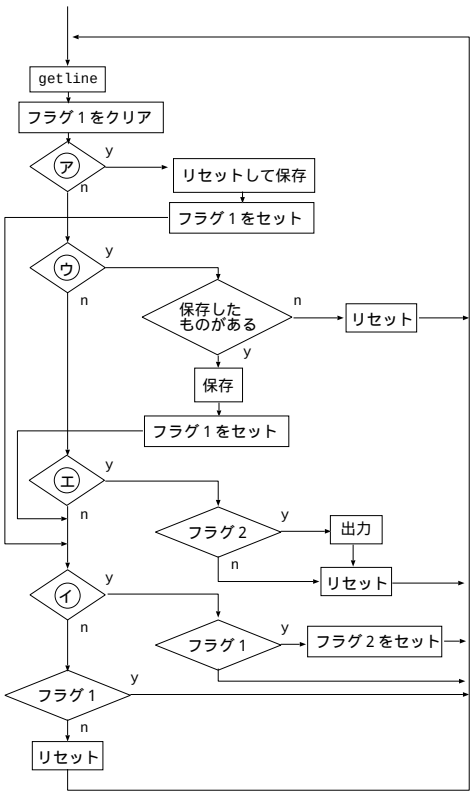

図 3: パターン中心のフローチャート (改良版)

きます。つまり、図 3 の疑似コードは以下のようになります。

```
{ flag1=0 }
```

```
{
    if(「ア」){ (リセットして保存) ; flag1=1 }
    else if(「ウ」){
      if( (保存したものがある) ){ (追加して保存); flag1=1 }
      else{ (リセット); next }
    }
    else if(「エ」){
      if(flag2==1){ (出力) }
      (リセット); next
    }
}
「イ」{ if(flag1==1) flag2=1; next }
(flag1==0){ (リセット) }
```

このように、「ア」「ウ」「エ」を一つの実行ブロック内に書いてしまえば if else 文でその無駄を排除することができます。ただ、ここまでするなら、先頭ブロックや「イ」のブロック、最後のブロックを外に出しているのもあまり意味はありませんので、いっそ全部を一つの実行ブロックにして、以下のようにしてもいいでしょう。

```
{
    flag1=0
    if(「ア」){ (リセットして保存) ; flag1=1 }
    else if(「ウ」){
      if( (保存したものがある) ){ (追加して保存); flag1=1 }
      else{ (リセット); flag2=0; next }
    }
    else if(「エ」){
      if(flag2==1){ (出力) }
      (リセット); flag2=0 ; next
    }
    if(「イ」){ if(flag1==1) flag2=1; next }
    if(flag1==0){ (リセット); flag2=0 }
}
```

8 状況を保存するアルゴリズム

7 節のプログラムの構造は行単位の処理形式になっていて、行の性質毎にその処理を分けて書き、現在、あるいは直前の状態は、2 つのフラグによって受け渡しをする、という形になっていました。

この状態を変数で受け渡す、という考え方をさらに進めると、現在の状態をより詳細に伝える変数を用意することで、6 節と 7 節のプログラムの中間のようなコードを書く、つまり、行単位の処理形式なんだけれども、処理の流れは単純なアルゴリズムに近い、というものを作ることができます。

ここでは、その状態を表す変数を mode と書くことにします。ここでは mode は 0,1,2,3 の値を取るとして、その意味は以下の通りであるとします。

- mode=0: リセットされた状態 (保存している行はない)
- mode=1: 現在行が「ア」であって、まだ「イ」「オ」のチェックはされていない状態
- mode=2: 前の行が「ア」か「オ」であって、「イ」ではなかった状態 (それらの行が保存されている)
- mode=3: 前の行が「イ」であった状態 (よって現在行の「エ」のチェックを行う)

この場合は、mode が 0 であることによりリセットされていることがわかるので、リセットという作業もほとんどいりません。

これを使えば、以下のようにすればできることになります。

(1) 1 行入力 (暗黙の getline)
(2) 「ア」ならばリセットして保存して mode=1
(3) mode の値をチェック
  (4) mode が 1 か 2 ならば、「ウ」であるかチェック
    (5) 「ウ」ならば、その行を保存する
    (6) 「ウ」でなければ mode が 1 でなければ mode=0 にして next
    (7) (5),(6) の後「イ」ならば mode=3 として next
    (8) 「イ」でなくて「オ」ならば mode=0、
      「オ」でなければ mode=2 として next
  (9) (3) の mode が 3 でなければ (i.e. mode=0) next
  (10) 「エ」ならば出力
  (11) mode=0 として next

フローチャートは、図 4 のようになります。疑似コードは、例えば以下の通りです。

```
「ア」{ (リセットして保存) ; mode=1 }
(mode==1 || mode==2){
    if(「ウ」){ (保存) }
    else if(mode!=1){ mode=0; next }
```

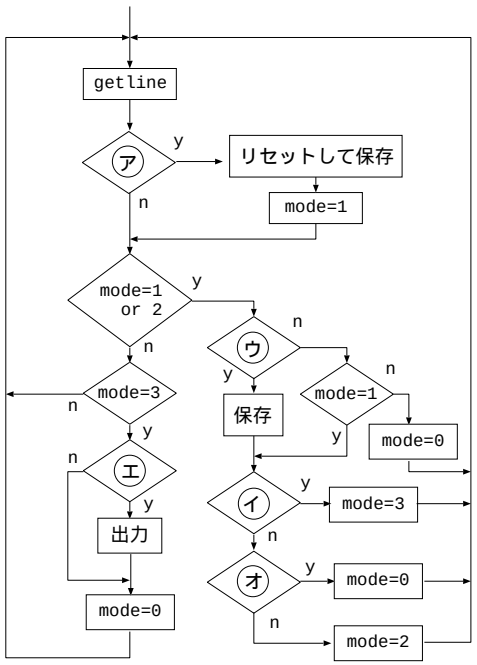

図 4: 状況保存パラメータ利用の場合

```
    if(「イ」) mode=3
    else if(「オ」) mode=0
    else mode=2
    next
}
(mode==3){
    if(「エ」){（出力）}
    mode=0
}
```

例外的なものを先に処理する、という考え方であれば次のようにも書いてもいいでしょう。

```
「ア」{（リセットして保存）; mode=1 }
(mode==0){ next }
(mode==3){
    if(「エ」){（出力）}
    mode=0
    next
}
{
    if(「ウ」){（保存）}
```

```
    else if(mode!=1){ mode=0; next }
    if(「イ」) mode=3
    else if(「オ」) mode=0
    else mode=2
}
```

9 ソースコード全体

後は、「保存」、「リセット」、「出力」などの部分を作成すれば済みますが、行の保存は、例えば

```
h[++lines]=$0
```

のようにすればいいでしょう。そしてこの “lines” という変数で現在保存している行数を管理すれば、リセットも “lines=0”、および必要ならばフラグのクリアで済みます。

現在保存している行があるかどうかも、この “lines” の値を見ればいいだけ (正ならば保存している行がある) ですし、出力部分もこれを利用すれば、

```
for(j=1;j<lines;j++) print h[j]
print h[lines] ";"
```

でできます。

また、「ア」～「オ」のパターンは、正規表現を使って、以下のように書けます。

- 「ア」: `/^[_a-zA-Z].*\(/`
- 「イ」: `/\)[ \t]*$/`
- 「ウ」: `/^[ \t]/`
- 「エ」: `/^\{[ \t]*$/`
- 「オ」: `/;[ \t]*$/`

これで今までの疑似コードをすべて具体的なコードとして書くことができます。それらを上げてみます。

まず、単純なアルゴリズムの場合 (cf. 6 節):

```
{
    while(1){
        if($0 !~ /^[_a-zA-Z].*\(/) next
        lines=0; h[++lines]=$0
        while(1){
            if($0 ~ /\)[ \t]*$/){
                getline
                if($0 ~ /^\{[ \t]*$/){
                    for(j=1;j<lines;j++) print h[j]
                    print h[lines] ";"
                    next
                }
                else break
            }
            else if($0 ~ /;[ \t]*$/) next
            else{
                getline
                if($0 !~ /^[ \t]/) break
                h[++lines]=$0
            }
        }
    }
}
```

次は、パターンマッチ中心のコード (cf. 7 節):

```
{ flag1=0 }
/^[_a-zA-Z].*\(/ {
    lines=0; flag2=0; h[++lines]=$0
    flag1=1
}
/^[ \t]/ {
    if(lines>0){ h[++lines]=$0; flag1=1 }
    else{ lines=0; flag2=0; next }
}
/^\{[ \t]*$/ {
    if(flag2==1){
        for(j=1;j<lines;j++) print h[j]
        print h[lines] ";"
    }
    lines=0; flag2=0
    next
```

```
}
/\)[ \t]*$/ { if(flag1==1) flag2=1; next }
(flag1==0){ lines=0; flag2=0 }
```

また、この無駄を省いたもの:

```
{ flag1=0 }
{
    if($0 ~ /^[_a-zA-Z].*\(/){
        lines=0; flag2=0; h[++lines]=$0
        flag1=1
    }
    else if($0 ~ /^[ \t]/){
        if(lines>0){ h[++lines]=$0; flag1=1 }
        else{ lines=0; flag2=0; next }
    }
    else if($0 ~ /^\{[ \t]*$/){
        if(flag2==1){
            for(j=1;j<lines;j++) print h[j]
            print h[lines] ";"
        }
        lines=0; flag2=0
        next;
    }
}
/\)[ \t]*$/{ if(flag1==1) flag2=1; next }
(flag1==0){ lines=0; flag2=0 }
```

および、いっそ一つのブロックで書いたもの:

```
{
    flag1=0
    if($0 ~ /^[_a-zA-Z].*\(/){
        lines=0; flag2=0; h[++lines]=$0
        flag1=1;
    }
    else if($0 ~ /^[ \t]/){
        if(lines>0){ h[++lines]=$0; flag1=1 }
        else{ lines=0; flag2=0; next }
    }
    else if($0 ~ /^\{[ \t]*$/){
```

```
        if(flag2==1){
            for(j=1;j<lines;j++) print h[j]
            print h[lines] ";"
        }
        lines=0; flag2=0
        next;
    }
    if(/\)[ \t]*$/){ if(flag1==1) flag2=1; next }
    if(flag1==0){ lines=0; flag2=0 }
}
```

そして状態を保存するアルゴリズムの場合 (cf. 8 節):

```
/^[_a-zA-Z].*\(/ { lines=0; h[++lines]=$0; mode=1 }
(mode==1 || mode==2){
    if($0 ~ /^[ \t]/) h[++lines]=$0
    else if(mode!=1){ mode=0; next }
    if($0 ~ /\)[ \t]*$/) mode=3
    else if($0 ~ /;[ \t]*$/) mode=0
    else mode=2
    next
}
(mode==3){
    if($0 ~ /^\{[ \t]*$/){
        for(j=1;j<lines;j++) print h[j]
        print h[lines] ";"
    }
    mode=0
}
```

それの例外処理を先にしたもの:

```
/^[_a-zA-Z].*\(/ { lines=0; h[++lines]=$0; mode=1 }
(mode==0){ next }
(mode==3){
    if($0 ~ /^\{[ \t]*$/){
        for(j=1;j<lines;j++) print h[j]
        print h[lines] ";"
    }
    mode=0
    next
```

```
}
{
    if($0 ~ /^[ \t]/) h[++lines]=$0
    else if(mode!=1){ mode=0; next }
    if($0 ~ /\)[ \t]*$/) mode=3
    else if($0 ~ /;[ \t]*$/) mode=0
    else mode=2
}
```

10 最後に

今回作成した C 言語のプロトタイプ宣言作成ツールは、元の C 言語のソースの書き方を指定して、しかもバグがない、という前提で作られていますので、違う書き方、あるいはその C のソースにバグがある場合は正しく動作しない可能性がありますし、いくつか紹介したスクリプトも異なる結果を生ずるかもしれません。

いくつか紹介した中では、最後の「状況を保存するアルゴリズム」によるものが一番おすすめのもので、慣れてくると今回のような問題では最初からそれを選択して作る、ということもできるようになります[5]。

今回の問題のように、複数の行を見て判別する処理は、例えばメールヘッダの処理などもそうですし、割りとあるような気がします。その場合、どのように考えたらいいのか、という一つの指針としてこのレポートが使えれば、と思います。

参考文献

[1] 竹野茂治、「AWK に関する基礎知識」(2006)

[2] 竹野茂治、「AWK による簡単なタイプ練習ソフト」(2006)

[3] 竹野茂治、「AWK による数合てゲームの作成」(2006)

[4] A.V. エイホ、B.W. カーニハン、P.J. ワインバーガー (足立高徳訳)、「プログラミング言語 AWK」、新紀元社 (2004) (元版は 1989)

[5] D.Dougherty、A.Robbins (福崎俊博訳)、「sed & awk プログラミング 改訂版」、オライリー・ジャパン (1997)

[6] 志村拓、鷲北賢、西村克信、「AWK を 256 倍使うための本」、アスキー出版 (1993)

[5] 実際、今回のサンプルスクリプトとして最初に作ったのがこの型のものでした。