プログラミング言語における日本語サポート方式
― 多言語サポートの国際標準化の脈絡から ―

長谷部　紀元
図書館情報大学

プログラミング言語を日本語の文字が扱えるように拡張されるようになって久しい．しかしこれは言語個別，オペレーティング・システム個別に行われている．このため日本語化に関する規格の標準化には大きな困難があった．ここでは日本語を含む多言語サポートの普及・標準化の可能性を探ることを目標として，次のような話題について報告する．

1) プログラミング言語の日本語化の問題点．
2) C 言語の多言語サポートの標準化の動き．
3) プログラミング言語の日本語化基本仕様の設定の動き．
4) ISO における 16 ビット・コード標準化の動き．

Extending Programming Languages to Process Japanese Characters
— In the Context of International Standardization —

Kigen HASEBE

University of Library and Information Science

1-2 Kasuga, Yatabe-mati, Ibaraki-ken, 305 Japan

There has been a long history of efforts to enhance programming languages to handle Japanese characters, as well as alphabets. But they were done in language dependent and/or operating system dependent ways, which made it difficult to establish standards for Japanization of languages. In this paper, the author reports on related topics in order to research the feasibility of pervasion and standardization of multi-lingual support. Such topics include

1) problems in Japanizing programming languages,
2) proposals for support of multiple languages including Japanese in C,
3) the activity to set up basic specification to Japanize programming languages in general, and
4) the ISO activity to standardize 16 bit code structures which is necessary for Asian languages.

1. はじめに

本稿ではプログラミング言語の日本語化のあるべき姿を検討するために，まずCとUNIXシステムでの日本語化の現状を見た後，多言語情報処理のためのコードの国際標準化の動きと，プログラミング言語日本語化の一般的方針設定の動きを眺める．

2. C言語における日本語化・国際化

2.1 C言語とUNIXシステムの標準化

C言語はUNIXオペレーティング・システムの中で生まれ，成長してきた．また一般にプログラミング言語が取り扱う文字データのコードは，オペレーティング・システムと密接な関係がある．その様な観点から，UNIXシステムとC言語の標準化の動きを見ておくことにする．

2.1.1 UNIXシステムの標準化

UNIXシステムはAT&Tのベル研究所で開発されたものであるが，大学・研究所ばかりでなく商用にも使用されるようになるとともに，早い段階で標準化の努力が始められた．手をつけたのはアメリカのUNIXシステムに関連する各種事業者の団体の/usr/groupである．1981年に活動を始めたこのグループは主としてUNIXの最初の商用の版であるSYSTEM IIIに基づいて，カーネルとC言語と，その関数ライブラリにたいする要求仕様をまとめ，その成果を1984年に発表した[1]．その内容は早い時期から次の版であるSYSTEM Vに反映されている．

AT&Tは自社の製品の標準を明確にして普及を図るために，1985年からSVID (System V Interface Definition)[2),3)]を発行している．

その後/usr/groupのUNIXシステムの標準化に関する活動はIEEEのPOSIX (Portable Operating System for Computer Environment) に引き継がれた．そのP1003.1の成果は1986年に米国標準草案として出版された[4]．その後も改訂作業は続けられており，1987年中に最終案ができることになっている．内容は基本的にはSVIDの第2版に一致している．その後の製品であるSYSTEM V Rel. 3で実現されたネットワークなどの新しい機能は入っていない．4BSD (Berkeley Software Distribution) の機能の一部 (Job Controlなど) がオプションとして入っている．C言語に関連してはカーネルとのインタフェースとなる関数の定義があるだけで，基本的にANSI X3J11の標準化に委ねている．一方，シェルと基本コマンドの標準化がP1003.2で行われている．

ヨーロッパとアメリカの情報処理システム・メーカ10社からなるX/OPENは，ソース・コード・レベルでの移植性を保証することによってソフトウェアの可用性を高めることを目的に活動している．そのための共通の応用システム環境をSVIDを基礎として定義している[5]．

一方，UNIXシステムの並立した流れになっている4BSDとSYSTEM Vとの関係がよく問題になる． 4BSDの機能強化に積極的であるSun MicrosystemsとAT&Tは1985年から統合の方向で共同作業を行っている．重要な課題はネットワーク分散型ファイル・システムと，それによって問題になる異種ファイル・システムの同時サポートとのことである．

2.2 C言語の国際標準化

C言語の標準に当たるものは長期にわたってKernighanとRitchieの教科書[6]しかなかった．それもad hocな処理系に影響されているため，標準としては不十分であった．処理系はその後も各種作られそれに連れてC言語も成長を続けたために，この傾向はますます助長された．この様な状態の中で1983年，ANSIにCの標準化作業のためにX3J11が設けられた．標準化作業の内容は過去に行われた拡張の集成が多くの部分を占めている．関数ライブラリについては，/usr/groupの結果を基礎としている．標準の草案が公表されており[7]，1988年の前半までには正式のANSI規格になる予定である．その活動の途中経過については本研究会でも報告されている[8]．最近の変更としては国際化がある．ほとんどがヨーロッパの言語を対象としたものであるが．

国際化に関する基本的な方針を下にまとめておく．

a. 8ビット・コードの使用を可能にする．
b. 文字の分類と変換 (cf. `isalpha`) と照合順序 (cf. `strcoll`)，小数点 (cf. `printf`)，日付の表記法 (cf. `strftime`) の各言語・国の習慣をサポートする．

c. これらの言語・国によって変動する処理環境 (locale) をプログラムの実行時に設定できるようにする (cf. `setlocale`).

ISO でも C の標準化を TC97/SC22/WG14 で取り上げており, ANSI の結果を採用するであろうといわれている.

2.3 日本語 UNIX 諮問委員会の提案

上のような状況の中でにおいて 1984 年に, 日本語 UNIX 諮問委員会は AT&T UNIX パシッフィック社から UNIX システムの日本語化に関する諮問を受けた. ベル研究所との意見交換も含めて検討を行い, 結果を 1985 年に提案として公表した[9]. この提案は AT&T が開発している UNIX SYSTEM V の日本語機能である JAE (Japanese Application Environment) さらには国際機能の設計の基礎となっている[10),11)]. 諮問委員会の提案についての解説は既に存在する[12)] ので, ここでは本稿の論点に関わる部分に話を絞り, その後の AT&T での製品化などを交えて, その特徴をまとめて紹介することとする. 個人的な意見・興味によっていることをお断りしておく.

2.4 UNIX システムと C の日本語化の基本方針

プログラムの言語, 具体的には C, による記述のレベルで, 日本語の文字をアルファベットと同等に取り扱えることを目標にした. その実現方式はアジア・ヨーロッパの各言語のための多言語化・国際化と互換性を持つように留意した. そのためには文字コードの体系の設定が基本的な問題であった.

元来の UNIX システムでは文字データ処理の全ての局面において, 1 バイト, もっと細かくいうと 7 ビットで表現できる ASCII が無反省に近い形で使用されていた. これは文字の表現に最低 2 バイトを必要とする日本語への応用には適さない. それで次の 3 個の局面を区別し, 各局面に応じてコードの取り扱いを切り替えることにした.

2.4.1 ファイル・コード

プロセスと UNIX カーネルとの間のデータ交換に使用するコード. 全ての文字はバイト列で表現される. ファイルに格納するデータのコードと同一である.

2.4.2 処理コード

使用者レベルのプログラムが論理的な処理のために使用するコード. 日本語の文字を ASCII で表されるアルファベットなどと全く同等に取り扱うことが可能である.

2.4.3 外部コード

端末や各種入出力装置に固有のコード. 理念的にはハードウェアを直接に制御する UNIX カーネルの入出力制御の部分だけが使用する. 必要によってハードウェアに関わる特殊な使用者プログラムが使用する.

2.5 ファイル・コードの拡張

ASCII 以外の文字コードの使用を可能にするため, EUC (Extended UNIX Code) を導入した. これは任意の文字セットを UNIX システムで取り扱うための, コード・システム設定のための方針ともいうべきものである. 将来の UNIX SYSTEM V では標準的な機能になる予定である. これの特徴をまとめると次のようになる.

a. 4 種までの別個の文字セットを同時に取り扱う. 各セットの文字は固有の長さのバイト列で表現される.

b. ASCII とそれ以外の文字セットの文字をバイト単位に区別できる.

c. 複数文字セットを取り扱うためのコードの拡張のエスケープ・シーケンスが省略などによって不要である.

d. コーディング方式は国際標準である ISO 2022 に準拠している.

EUC コードの表現を表 1 に示す[10)]. ここで SS1 と SS2 の値は 8E と 8F (16 進数) であるので, セット 0 以外のコードを構成するバイトは全て最上位のビットが 1 となる. セット 0 には固定的に ASCII を割り当てる. 他のセットには任意の文字コードを割り当てて, (最上位ビットの変更を別にして) その標準のコードのままで使用する. このように ASCII とそれ以外の文字の区別が極めて容易である. jae .].

EUC の応用として日本語の場合には諮問委員会提案によって, セット 1, 2, 3 の (SS2, SS3 を除く) バイト数は 2, 1, 2 で, それぞれ漢字 (JIS C 6228), カナ文字 (JIS C 6220) それに外字を割り当てることになっている. ヨーロッパではセット 1 のバイト数を 1 にして各国の文字コードを割り当てる. これは

コード・セット	EUC表現
コード・セット0	0xxxxxxx
コード・セット1	1xxxxxxx 1xxxxxxx 1xxxxxxx 1xxxxxxx 1xxxxxxx 1xxxxxxx
コード・セット2	SS2 1xxxxxxx SS2 1xxxxxxx 1xxxxxxx SS2 1xxxxxxx 1xxxxxxx 1xxxxxxx
コード・セット3	SS3 1xxxxxxx SS3 1xxxxxxx 1xxxxxxx SS3 1xxxxxxx 1xxxxxxx 1xxxxxxx

表1　EUCコード・セットの表現

ASCIIと併せて8ビットの1バイト・コードを使用するのと同じになる．

日本語文字とそれ以外の文字との間にシフト・コードを必要とする場合の，言語仕様の設定の困難については強く指摘されているところである[13]．EUCはその解決手段として有効である．パソコンでよく使用されているシフトJISもシフト・コードは不必要であり，UNIXシステムのコードとの関係がよく問題にされる．長期的な観点からは，国際的な互換性と拡張可能性からEUCの方が優れていることが明らかであろう．

2.6 処理コード

諮問委員会提案はファイル・コードでの表現に複数バイトを必要とする文字をASCIIと同等の論理性で取り扱うために，新しいデータ型である `long char` を導入した．全く新しい着想というわけではないが，汎用システムの標準としては意欲的な試みであったと思う．これは `char` が8ビットの幅であるのに対して，日本語の文字の表現を可能にするために16ビットの幅を持っている．他の性質は `unsigned int` と同様である．`long char` 型の導入によって，実現される利点を下にまとめておく．

a. 日本語などの文字を人間の意識に合わせて，1文字をプログラム上でも1個のデータ単位として取り扱うことができる．

b. 日本語などの文字をアルファベットと同一のプログラム・コーディング手法で取り扱うことができる．これに対して，バイト列として日本字の処理を行う場合には，プログラム論理をアセンブラ的なレベルに落とすことになり，記述性も実行効率も悪くなる．

c. 文字列定数ばかりでなく，文字定数も使用することができ，記述性を改善することができる．前者はバイト列として取り扱う場合にも可能であるが，文字定数の方はそうはいかない．

d. ファイル・コードとの間の変換はCの標準入出力関数ライブラリのレベルで自動的に，つまり使用者がコーディングするプログラムの外で行う．

e. そのために，`char` 型の文字・文字列を取り扱う関数に対応させて，`long char` 型の各種の関数を用意した．

f. 従来のCの特色であったシステム記述機能を生かすため，`char` 型の機能はそのまま残っている．

ファイル・コードと処理コードの対応を表2に示す[10]．処理コードの定義の上でのポイントはセット0について，EUC表現と整数としての値が同じになるようにしたことである．これによって基本コードであるASCIIに関して，比較演算に問題が生ずることを避けることができた．

2.7 処理コードの重要性

蛇足ながら，処理コードと `long char` の概念の導入は筆者の提案に基づいている．日本語の処理のための実用的なソフトウェアを効果的に開発するためには，文字・文字列の定数の直感的な表記を許す `long char` の方式が大変に重要であると考えている．現在のところ，この案はANSIにおけるCの標準化には反映されておらず，広く普及するには至っていない．

しかしながら `long char` の評判は良く，早い時期にNEC[14]とNTT[15]で処理系が試作された．後者は商品化されている．他でも日本語ウィンドウシステムの開発などに使用されている[16]．

ファイル・コードから処理コードへの変換のオーバヘッドが問題とされるかも知れない．実験によると単純なファイル・コピーはともかく，字句解析のようにテキスト処理を行うものにおいてはプログラム・サイズと処理時間が相当に減少することが示されている[15]．コードの論理性によるプログラムの単純化が効果を示していると考えられる．

コード・セット	処理コード表現
0	0 0 0 0 0 0 0 0 0 x x x x x x x
1	1 0 0 0 0 0 0 0 1 x x x x x x x 1 x x x x x x x 1 x x x x x x x
2	0 0 0 0 0 0 0 0 1 x x x x x x x 0 x x x x x x x 1 x x x x x x x
3	1 0 0 0 0 0 0 0 0 x x x x x x x 1 x x x x x x x 0 x x x x x x x

表2　処理コード表現

但し,提案で採用されたファイル・コードの定義では4種類以上の文字セットを取り扱うことはできない．処理コードの概念をもっと徹底させるならば,任意の数の文字セットが同時に可能であるような仕様を作ることもできると考えられる．**long char** のビット幅16が十分でなければ,32に拡大することも考えられると思う．記憶素子の価格の低下は目ざましいばかりである．論理的な機能の強化とソフトウェア開発の容易化につながるのであるから,記憶容量のオーバヘッドを必ずしも絶対的な制約と考える必要なない．またファイル・コードにシフト・コードを使用することが必要になるであろう．しかし文字データに対する論理的な処理を全て処理コードによって行うならば,大きな困難は生じずに済むように思われる．

2.8 JAE

AT&Tは諮問委員会の提案に基づいて日本語化の製品であるJAE 2, 0を開発している[10]．処理コードの機能のC言語の組み込みについてはANSIの様子見というところでまだ実現していない．ライブラリでのサポートについては,**wchar_t** という **typedef** によるデータ型によるものを実現している．各文字セットのバイト数など,EUCコードと処理コードとの対応は実行環境の設定によって動的に制御できるようになっている．

JAE 2.0での興味ある機能としては,交換可能な日本語入力機能がある[11]．これはSYSTEM V 3.0のストリームTTYによって,カーネルの内部で実現されており,外部コードの処理もなされる．

2.9 処理コードによる日本語処理プログラムの例

処理コードである,**long char** 型を使用するCプログラムの例題を見ることにしよう．処理コードの言語への組込みがどのような効果を持つかを実感して頂けると有難い．これはNECの試作コンパイラによって実際に実行したものである．

2.9.1 処理コードでの入出力

処理コードでテキストを入力し,“文字”数を勘定する簡単なプログラムを例1に示す．**getlchar()** は **getchar()** に似ていて,バイト(**char**)単位にではなく文字単位に読み取る点で異なる．ファイル・コードの文字セット毎のバイト数の差を意識する必要がないのがポイントである．なお **printf()** の書式を指定する文字列定数において,形式が旧来のCのものと同じであるのに日本字を混在できている．これはこの形式ではプログラム・テキスト中の日本字を表すバイト列がそのまま実行プログラムに現れるからである．他の **long char** 型の文字列や文字列定数では,バイト列から各々のデータ型に変換される．

2.9.2 文字の分類

テキスト中の‘。’(句点)と‘.’(ASCIIのピリオド)を勘定するプログラムを例2に示す．バイト列としては2バイトである。’も処理コードでは1文字であるので,**case** 文のラベルのところ(*09*)に文字定数としてそのまま書くことができる．日本語の文字と **ASCII** をプログラム・ロジックのうえで区別する必要は全くない．

2.9.3 バイトでの日本字処理

例2と同じことをするプログラムを **long char** 型を使わないで,**char** 型だけで書くと例3のように実に嫌らしいものになる．まず入力する文字のバイト数を区別する(*07*)．ここでファイルコードの具体的な形式を知っていることがまず必要である．次に文字種確認のやり方が日本字とASCIIとで違ったものになる．ASCIIに関するコーディング(*10–11*)を日本字でのやり方(*08*)にしたいと思うプログラマは少ないだろう．最も悪いことは,日本字のコードのビット・パターンを具体的に書く必要のあることで,記述性・可読性を酷く損なってしまう．更に二次的な作用として,日本字のコードの変更に関する移植性も失われる．日本字のコードがリテラルとしてだけ使用されているならば,単純にコード変換をすれば済む訳だが．

2.9.4 文字列処理

最後に日本字文字列の操作を例4にしめす．**getls()** は1行分のテキストを **long char** 型の

```
01 #include <stdio.h>
02 main()
03 {
04   int count = 0;
05   while (getlchar() != EOF)
06     count++;
07   printf("文字数: %d\n", count);
08 }
```

例 1

```
01 #include <stdio.h>
02 main()
03 {
04   int c, kuten = 0, period = 0,
05       other = 0;
06   long char lc;
07   while ((c = getlchar()) != EOF) {
08     switch (lc = c) {
09     case '。': kuten++; break;
10     case '.': period++; break;
11     default: other++; break;
12     }
13   }
14   printf("句点: %d\n", kuten);
15   printf("period: %d\n", period);
16   printf("その他: %d\n", other);
17 }
```

例 2

```
01 #include <stdio.h>
02 main()
03 {
04   int c0, c1, period = 0, other = 0,
05       kuten = 0;
06   while ((c0 = getchar()) != EOF) {
07     if (c0 & 0x80) c1 = getchar();
08     if (c0 == 0xa1 && c1 == 0xa3)
09       kuten++;
10     else switch (c0) {
11     case '.': period++; break;
12     default: other++; break;
13     }
14   }
15   printf("句点: %d\n", kuten);
16   printf("period: %d\n", period);
17   printf("その他: %d\n", other);
18 }
```

例 3

```
01 #include <stdio.h>
02 #include <string.h>
03 main()
04 {
05   long char ls0[41], ls1[41];
06   long char *kpos;
07   int le0;
08   while (getls(ls0) != NULL &&
09       getls(ls1) != NULL) {
10     le0 = lstrlen(ls0);
11     lstrcat(ls0, ls1);
12     printf("入力と結果の文字数は ");
13     printf("%d, %d と %d です.\n",
14       le0, lstrlen(ls1), lstrlen(ls0));
15     printf("結果は \"%ls\"\n", ls0);
16     kpos = lstrchr(ls0, '字');
17     if (kpos != NULL)
18       printf(
19       "'字' は %d 文字目にあります.\n",
20           kpos - &ls0[0] + 1);
21   }
22 }
```

例 4

```
01 % cat lstrdata
02 cat lstrdata
03 文字列中での ASCII と
04 日本字の混合
05 % lstr < lstrdata
06 入力と結果の文字数は 14, 6 と 20 です.
07 結果は "文字列中での ASCII と日本字の混合"
08 '字' は 2 文字目にあります.
```

例 5

文字列に変換して入力する (08). 結果は引き数のバッファに格納される. 2 個の文字列を入力し (*08–09*), 連結して (*14*) 結果を出力する (*15*). 確認のために文字列の長さを出力しているが (*12–14*), 連結の結果が元の文字列を壊すので (*11*), 文字列バッファの一つを先だって調べている (*10*). これらの操作は日本字を必ずバイトの列として取り扱うシステムでも可能である. 根本的に異なる事情は文字を個別に取り扱うときに生ずる. 16 は文字列中で特定の文字を探索し, その結果を出力している (*17–20*). このような探索処理はの記述は, 日本語のがバイト列であるときには不可能である.

これの実行結果を例 5 にしめす. プログラム **lstr** が例 4 のソースに対応する実行プログラムである (*05*). これに入力するデータファイル

lstrdata の内容は *03* と *04* の 2 行に示されている．これらの文字のうち英数字と記号が EUC のセット 0 で，仮名漢字はセット 1 で表記されている．

3. 多言語処理用コードの国際標準化

文字コードの国際標準化は現在まで，主として通信における情報の交換の円滑化を目的として行われてきている．全ての文字・制御コードを 8 ビットからなるバイトあるいはオクテット (octet) の列で表現する習慣，また 8 ビットのうち 7 ビットだけを意味のあるものにしようとする傾向はこのためである．これは対象をヨーロッパ系の言語に限って，1 バイトで全ての図形文字が表現できる場合には問題は生じない．

一方アジアで広く使用されている漢字のように文字種の多い表記手段を使用する場合には大きな困難が生じる．同様の問題は，朝鮮のハングル文字などにも存在する．これは大きな欠陥である．多数の言語の同時処理を考慮するならば更に問題は大きくなる．

このような問題を解決するためには，情報処理の局面での文字のコーディングの方式を変更することが必要となる．このような観点から，コンピュータなどの情報処理システムの内部におけるコード構造に関する標準化が ISO/TC97/SC2 で検討されている．

現在 2 種の提案が行われているので，ここではそれらを紹介する．何れの方法によっても，既存のコード系

a. 96 文字集合 (cf. JIS C 6220)
b. 191 文字集合
c. 94×94 文字集合 (cf. JIS C 6228)

などを埋め込む (slot in) ことが容易にでき，複数種類のコード系を併存させることができる．

複数バイトのコードでしか標準化されていなかったアジア系の表記システムを 1 個のデータ単位としてコード化しようとしていることは大きな前進といえるであろう．これによってアジア系の言語を処理するプログラムの記述を改善するための，プログラミング言語の仕様の改善に大きな可能性が生まれたことは大きな希望である．

3.1 8+8 2-オクテット・コード

これは 8 ビット・コード (octet) のうちの図形文字に相当する部分を 2 個組合せた形のものを，1 個の図形文字のコードとして使用する．これを図 1 に示す．この提案では既存のコード化の習慣と通信ハードウェアとの互換性を重視している．7+7 ビットの 2 バイト・コードによる図形文字 (ideographic character) は同時に 4 種まで (最大 36 K 文字種) 埋め込むことができる．つまり図 1 の I00 から I11 までである．これには JIS C 6228 (日本) や GB 2312 (中国) それに KS C 5619 (韓国) などがある．

7 あるいは 8 ビットの 1 バイト・コードによる図形文字 (Alphabetic character) は，

a. 0 の列
b. 上記の 2 バイト・コードで ideographic character としては使用していない部分，つまりアルファベットなどと重複する部分 (32–47 と 160–175 の列で A00 から A11 まで)

に埋め込むことができる．

このコード表が溢れた場合にはロッキング・シフトの手法によって，コード表の枚数を増やす形で拡張を行うことができる．最大 7 M 文字種．

このコード構造は仕組みが単純で受け入れられ易いためか，標準化はこれを採用する方向で動いているそうである．しかし 36K 文字種という制限からロッキング・シフトを使うようになった場合には，現状と同じような問題が生ずる恐れがあろう．

3.2 16 ビット・コーディング

16 個のビットの全体を使用する．全てのビット・パターンをコードに使うことができるので，16 ビット・コードの利益を最大限に生かした形で利用することができる．

16 ビットの幅を持つコードの表は目的毎に区分して使用する (表 X)．

a. 制御機能文字 (0.7 K 文字種)
b. アルファベット及び記号 (3 K 文字種)
c. 図形文字一般 (54 K 文字種)
d. 私的使用その他

既存の各種のコードは各々の区分に，数値的に変換して埋め込まれる．16 ビットのコード表が溢れた場合にはシングル・シフトの手法などによって，文字を 32 ビットでコーディングすることができる．最大 1.5M 種．図形文字一般の区分は大きいのでこの拡

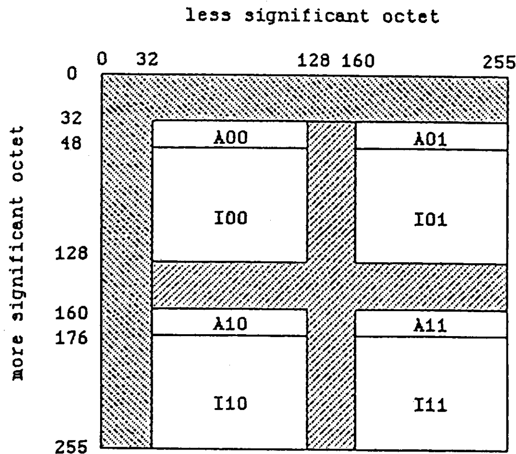

図1　8+8 2-オクテット・コード

張の必要性は比較的少ない.

このコード構造によるならば, 多数の種類の文字セットを同時に使用できることになるので, プログラム言語の国際化の観点から大変に興味を引かれる. ただ既存のコード系の埋め込みをコード表の上で隙間なくやるようにしているため, それら2種の間のコード変換が複雑になる傾向がある. その結果数値的な変換のパラメタの決定とその維持には難しいところがあるように思われる.

4. プログラミング言語の日本語化の動向

4.1 日本語機能専門委員会

過去においてはプログラミング言語の日本語化は, 各言語毎に個別に検討されてきた. 更にいうなら, 日本語化の方式はメーカ毎にばらばらであり, 各言語の日本語化の方式を標準化することも困難であった. たとえば, 日本電子工業振興協会の JIS 日本語 FORTRAN 原案作成委員会でも決定案に到達することができていない. これには複数バイト文字の取り扱いについて, 国際的な標準化が進んでいないことも原因になっている. このような事態を打開するため, 一貫した方針に基づいて各種言語における日本語化を調整し, 同時にこれを国際標準に反映させることを目的に情報処理学会情報規格調査会の日本語機能専門委員会[†1]が活動を行っている. ここではこの委員会で検討されている方針の一端を紹介する[17]. なおここでの記述は審議の途中における資料に基づくものであって, 当委員会で決定された方針ではないこと, また記述に誤りがあった場合には筆者の責任であることをお断りする.

4.2 プログラム言語における日本語の扱いに関する案

4.2.1 文字型の種類

文字を表現するためのコードのバイトの数に応じて, 次の2種類の文字型を併用する. 文字種としては後者 (第2種) が前者を包含する.

4.2.1.1 第1種 1バイトであり, 通常は ISO 文字あるいは ANSI 文字をこれに当てる.

4.2.1.2 第2種 2バイトであり, 通常はこれが日本語文字である.

4.2.2 文字 / 文字列のプログラムでの操作

a. データには文字型のものと文字列型のものがありうる.
b. データの長さは, バイトではなく, 文字数で数える.
c. 文字型の混在は, 1個の文字列中では許さない. 外部ファイルでは混在が可能である.
d. ライブラリ関数を2種の文字型について別個に用意する.
e. 2種の文字型の間での変換関数を用意する.
f. 異なる文字型の文字列の値の比較も可能とする. 必要ならば第1種から第2種への変換を自動的に行う.
g. 文字の印字幅は文字型とは独立である. 文字種 (cf. カタカナ) に基づいて, あるいは書式中の指定によって印字幅を決定する.

4.2.3 プログラム記述のための文字のデータ型

a. コメントや識別子に第2種文字を許すときは, それらが全て第2種文字からなるものとする.

4.2.4 ISO 仕様の案

4.2.4.1 文字型の種類 文字型を複数設定できるようにする.

†1 委員長は筑波大学電子情報工学系の池田克夫教授.

4.2.4.2 新規の文字型の仕様範囲 コメントと識別子において，第2種に相当する文字の使用を許す．キーワードにおいては許さない．

4.3 個別言語の日本語化への適用

日本語機能専門委員会の直接の活動ではないが，上記の方針に合致している，あるいは近似していると考えられる標準化の試みに次のようなものがある．

a. FORTRAN 8X に関する ANSI X3J3 へのコメント

b. C 言語の多国語化に関する ANSI への提案[18]．

c. Common Lisp の日本語化[19]．

5. まとめ

以上，日本語 UNIX システム諮問委員会の提案において筆者が重要であると考える3個の局面における文字コードの体系的区分，更にその中の処理コードに関わる日本の内外における標準化の動きを見てきた．

ファイル・コードである EUC は問題なく根付く方向にあるといえよう．外部コードは当然の概念ではあるが，UNIX SYSTEM V 強化された機能によって，日本語機能を手際良く整理された形で組み込むのに役だっている．残る処理コードの概念は，C 言語の関数ライブラリの機能として定着することは確からしくなってきた．これが標準化されればそれなりの効果があることは確かである．ただ C 言語そのものの拡張に至るかどうかは確かでない．

一般的にいって日本語処理のプログラム記述性を良くし，ソフトウェア開発の促進を図るには言語のデータ型の追加による文字・文字列の定数の記述手段が不可欠であると考えている．上でみた通り処理コードにあたるものの国際標準化の作業が進行している．更にプログラミング言語一般の日本語化に対してこれを適用しようとする動きもある．これらの動きが成果を挙げることを期待したい．これによって利益を得るのは日本ばかりでなく，アジアなどの多くの国も同様である．

(謝辞)

資料を提供して頂いた東京大学工学部の和田英一教授，筑波大学電子情報工学系の中田育男教授，AT&T ユニックス・パシフィックの門田次郎氏，NTT の山田伸一氏と鈴木幸市氏，日本語 C 処理系を貸与して頂いた日本電気の藤林信也氏に感謝します．

参考文献

1) *1984 /usr/group Standard*, /usr/group Standards Committee (1984).

2) *System V Interface Definition, Issue 2, Volume I, II*, AT&T (Jan 1986).

3) *System V Interface Definition, Issue 2, Volume III*, AT&T (Jan 1987).

4) *Draft American National Standard, IEEE Trial Use Standard Portable Operating System for Computer Environments, IEEE Std 1003.1*, Technical Committee on Operating Systems of the IEEE Computer Society (Apr 1986).

5) The X/OPEN Group Members: *X/OPEN Your Guide to Portability, Portability Guide II*, p. 28, X/OPEN (Jan 1987).

6) Kernighan, B. W. and Ritchie, D. M.: *The C Programming Language*, p. 228, Prentice-Hall (1978).

7) X3J11: *Draft Proposed American National Standard for Information Systems — Programming Language C, X3J11/86-151*, p. 191, The American National Standard Institute (Oct 1986).

8) 石畑清: C の ANSI 規格案について，情報処理学会 プログラミング言語研究会資料，pp. 8 (1985).

9) 日本語 UNIX システム諮問委員会: UNIX システム日本語機能の提案に当たって，プレス・リリース (1985).

10) UNIX SYSTEM V 日本語アプリケーション・エンバイロメント　リリース 2.0 機能説明書，p. 178, AT&T ユニックス・パシフィック (1987).

11) Kogure, H. and McGowan, R.: *A UNIX System V STREAMS TTY Implementation for Multiple Language Processing*, *Proceedings of the Summer 1987 USENIX Conference*, pp. 323-36 (Jun 1987).

12) 小野芳彦: UNIX の日本語化の実現方法，情報処理，Vol. 27, No. 12, pp. 1393-1400 (1986).

13) 木下恂: 標準プログラミング言語における日本語処理，情報処理，Vol. 26, No. 3, pp. 226-232 (1985).

14) 石田晴久: UNIX の日本語機能と日本語対応 C コンパイラ，*bit*, Vol. 19, No. 5 別冊　最新 UNIX, pp. 132-38 (1987).

15) 鈴木幸市，尾田政臣，川瀬登，加藤哲郎: UNIX における日本語プログラミング支援ソフトウェ

ア，研究実用化報告，Vol. 35, No. 12, pp. 1353-61 (1986).

16) 萩谷昌己: GMW ウィンドウ・システムについて, *bit*, Vol. 19, No. 3, pp. 226-241 (1987).

17) 中田育男: プログラミング言語における日本語の扱い, 第 4 回 日本語専門委員会資料 (Apr 1987).

18) Japan C Language Committee: *A Proposal for an Addition to the ANSI C Language Standard* (1987).

19) 元吉文男: Common Lisp アラカルト (9) 日本語化, *bit*, Vol. 19, No. 5, pp. 593-596 (May 1987).