記　号　処　理　36－2
プログラミング言語　5－2
（１９８６．３．１０）

Ａｄａにおける日本語テキスト処理パッケージの構築とその使用

松尾篤弥　　牛島和夫
（九州大学 工学部）

１．まえがき

プログラミング言語Ａｄａは、１９７０年代のソフトウェア工学の成果を取り入れた数々の優れた機能を持っている。しかし、我国におけるソフトウェア開発にＡｄａを使用しようとすると、既定の文字型として日本語文字を使えず、日本語テキスト処理が困難という問題がある。

普通、日本語文字は２バイトにコード化し、英数字は１バイトにコード化する。日本語テキストとして日本語文字と英数字の２種の文字群が任意に混在するテキストを考えた場合、文字の数とコード化するのに必要なバイト数との間に一定の関係は成り立たない。このことにより、日本語テキスト処理自体にも多くの問題がある。

そこで、Ａｄａにおいて日本語文字・テキストを扱うことを可能にし、更に日本語テキスト処理時に起こる問題を解決するパッケージを開発した。

ここで述べる方式は、規格からの逸脱を極力避けるという方針のもとに、処理系には直接手を加えず、パッケージを付加することにより日本語テキスト処理機能を実現した。なお、このパッケージは米国防総省が実施する検定を通ったECLIPSE MV/10000Ａｄａ上で実現している。

２．用語の定義

ここでは、本論文で使用する用語の説明を行なう。

◆ＡＳＣＩＩ文字
ＡＳＣＩＩコードが定義する図形文字あるいは書式制御文字。本論文においてはＡＳＣＩＩコード、ＩＳ０７ｂｉｔコード、ＪＩＳ６２２０コードを特に区別しない。

◆日本語文字
ＪＩＳ６２２６漢字コードが定義する図形文字

◆混在文字列
日本語文字とＡＳＣＩＩ文字が任意に混在した文字列

◆日本語テキスト
混在文字列によって構成したテキスト

◆シフトコード
混在文字列において、ＡＳＣＩＩ文字集合と日本語文字集合を弁別するための特殊なコード列

３．日本語テキスト処理とＡｄａ

この章では、Ａｄａに日本語テキスト処理機能を実現する方法について述べる。

３．１　使用するシステム

この実現に使用したシステムは、九州大学情報工学科の教育用計算機システムデータゼネラルECLIPSE MV/10000上のＡＤＥ(Ada Deveropment Environment Ver. 2.20,2.30)である。ＡＤＥが提供するＡｄａ処理系は米国防総省が行なう検定を通ったものである。また、MV/10000は日本語情報処理システムIKISを持っている。IKISは日本語端末からのかな漢字変換による日本語文字入力、端末・プリンタへの日本語文字出力などを提供する。

IKISにおける文字コード（IKIS漢字コード）は、英数字はＡＳＣＩＩコード、日本語文字はＪＩＳ6226漢字コードに16進で8080H を加えたものである。日本語テキストを１バイトごとのコード列とみたときコード値が16進で00H ～7FH のときＡＳＣＩＩ文字、A0H ～FFH のときは日本語文字と、コード値を表わすバイト列の各先頭ビットを見れば２つの文字集合の区別がつく（図１）。従って、シフトコード等の特別な弁別子を使うことなく、ＡＳＣＩＩ文字と日本語文字の混在が可能である。

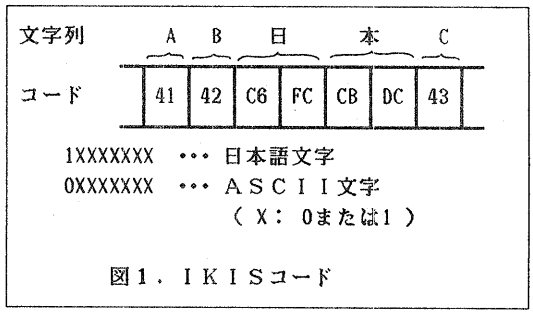

図１．ＩＫＩＳコード

３．２　日本語テキスト処理

プログラミング言語で日本語処理機能と言う場合、日本語化の程度によって、次の３つのレベルに分けることができる。

a)日本語文字、あるいは日本語文字を含むテキストをデータとして処理することができる
b)プログラム中の識別子（変数名など）に日本語文字を使うことができる
c)プログラム中の予約語に日本語文字を使うことができる

この論文で言う「日本語テキスト処理機能」とは、上記a)の日本語文字や日本語文字を含むテキストをデータとして処理する機能のことであり、以下の要

求機能を満たすものとする。

1)日本語文字を算体(object)として扱い得ること
2)ファイル・端末等で日本語テキストの入出力が行なえること
3)日本語文字リテラルをＡｄａのプログラム・テキスト中に書けること

３．３　Ａｄａにおける文字型

Ａｄａは既定の文字型 CHARACTER型を図２の様に宣言している。

```
type CHARACTER is
  ( nul,soh,stx,etx,eot,enq,ack,bel,
    bs ,ht ,lf ,vt ,ff ,cr ,so ,si ,
    dle,dc1,dc2,dc3,dc4,nak,syn,etb,
    can,em ,sub,esc,fs ,gs ,rs ,us ,
    ' ','!','"','#','$','%','&',''',
    '(',')','*','+',',','-','.','/',
    '0','1','2','3','4','5','6','7',
    '8','9',':',';','<','=','>','?',
    '@','A','B','C','D','E','F','G',
    'H','I','J','K','L','M','N','O',
    'P','Q','R','S','T','U','V','W',
    'X','Y','Z','[','¥',']','^','_',
    '`','a','b','c','d','e','f','g',
    'h','i','j','k','l','m','n','o',
    'p','q','r','s','t','u','v','w',
    'x','y','z','{','|','}','~',del);
```

図２．CHARACTER の宣言

図２中、要素 nulからusまで、および delの計33個は書式制御文字、' 'から '~'までの95個は図形文字である。これから明らかなように、CHARACTER型の文字集合は、ＡＳＣＩＩ文字群（厳密にはISO 7bitコード文字群）のみで、日本語文字を含まない。一部の日本語ＦＯＲＴＲＡＮ等で行なっている様に、文字型を適当に拡張解釈し文字集合に日本語文字を加えるのは、Ａｄａにおいては明らかな文法違反である。実際、今回の実現に使用したＡｄａ処理系では CHARACTER型の拡張解釈を行うことはできなかった。

また、CHARACTER 型を拡張せず、ある特定のＡＳＣＩＩコード列を日本語文字とみなす方式にも問題がある。例えば、ＪＩＳ6226漢字コードを使い、日本語文字を表現する２バイトコードの各バイトに対応するＡＳＣＩＩ文字の列を使って日本語文字を表現することを考える。この様な方式では、ＡＳＣＩＩ文字と日本語文字を弁別するためシフトコードが必要になる。ところが、列挙型 CHARACTER型が文字リテラルとして宣言する要素は、図形文字のみである。このことは、通常１つ以上の書式制御文字を含むシフトコードを文字リテラルとして表わせないことを意味し、プログラムテキスト中に日本語文字リテラルを表現できない（図３）。従って要求を満たさない。

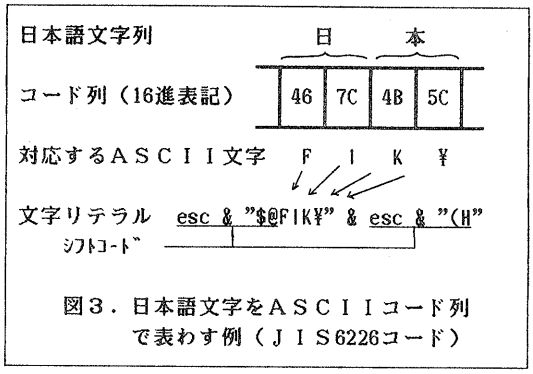

図３．日本語文字をＡＳＣＩＩコード列で表わす例（ＪＩＳ6226コード）

３．４　混在文字型

前節で示した様に既定の文字型は使えないので、ＡＳＣＩＩ文字と日本語文字とが任意に混在する混在文字列を表わすための新しい型、混在文字型（NCHARACTER型）および混在文字列型(NSTRING型）を導入する。なお、ここでは文字コードとして、３．１節で説明したIKIS漢字コードを使うものとするが、２つの文字集合の混在を弁別するためにシフトコードを使わず、日本語文字コードの各バイト値がＡＳＣＩＩコードの特殊記号と重複しないコード系なら（例えば文献[1] の汎用日本字コードなど）、何れのコード系にも適用できる。

混在文字列を文字コードの列とみて、それに直接対応する算体の列を考える。すると、コードの長さが違う日本語文字コードとＡＳＣＩＩ文字コードの両方に混在文字型算体を無理なく対応させる必要がある。そこで、２バイトである日本語文字コードを先行する第１バイトと後続の第２バイトとに分け、各々のコード値に１個づつ混在文字型の算体を対応させる。また、１バイトであるＡＳＣＩＩ文字コードには混在文字型の算体１個を対応させる。すると、混在文字型算体は１バイトのコード値に対応できればよいことになる[1] 。

そこで、混在文字型NCHARACTERを１バイトのコード数に対応する 256個の要素をもつ列挙型として宣言し、さらに、混在文字列型 NSTRINGをその要素にNCHARACTER型算体を持つ配列型として宣言する。混在文字列の要素を表わす算体の列は NSTRING型として表現できる。

混在文字型算体とコード値の対応付けは入出力手続きやコードに関係する関数などで行なえばよい。従って、この対応をつけるのみなら、列挙型NCHARACTERの要素はどのように設定しても、３．２節の要求1)2)を満たすことができる。

混在文字列リテラルをプログラムテキスト中で使えるようにするには、混在文字型算体とコード値の

間の対応のみでなく、混在文字列リテラル中のすべての図形文字が混在文字型の値と対応する必要がある。すなわち、混在文字列リテラル内のすべての図形文字が、列挙型NCHARACTERの要素と対応しなければならない。

プリプロセッサを使ってすべての図形文字と要素間の対応をつけるのも一つの方法である[2]。本論文では、この対応を直接つける方法を与える。すなわち、混在文字列リテラルの内、ＡＳＣＩＩ文字リテラルはそのままの形で、日本語文字リテラルは、第１バイト、第２バイト別々の特殊な文字リテラルが並んだものとみなし、これらの１バイトでコード化可能な文字リテラルすべてを列挙型の要素として宣言するものである。（図４）

なお、NCHARACTER型、NSTRING 型の２つの型はパッケージNCHAR_DEF において宣言している。

３．５　入出力手続き

NCHARACTER型算体および NSTRING型算体を入出力するために TEXT_IOに相当するパッケージNCHAR_IOを用意した。このパッケージにはファイルのOPEN、CLOSE、ファイルや端末での入出力、その他の手続きや関数等が宣言してある。

混在文字型算体の内容と実際の文字コードの対応はこのパッケージでつける。出力においては、混在文字型算体の内容を文字コードに変換し、さらに、外部装置が受けつける形にしたあと出力のためのシステムコールをおこなっている。入力の場合は逆に、入力のためのシステムコールのあと、入力した文字コードを混在文字型の値に変換している。

なお、パッケージNCHAR_IOの入出力手続きは、パッケージ TEXT_IOを用いて作製したファイルも読み書きできるようになっている。これによって、今までに蓄積してきたＡＳＣＩＩ文字のみのテキストも混在文字型として扱うことができる。

３．６　規格への適合性

ここで、今まで述べた方法がＡｄａの規格に適合しているか考察する。

入出力のためのパッケージNCHAR_IOは規格上問題ない。なぜなら、入出力の実現は各処理系に任されていて、規格の範囲外となっているからである。

次に、NCHARACTERと NSTRINGを宣言しているパッケージNCHAR_DEF では、混在文字列型 NSTRINGを、単にNCHARACTER型の配列型として宣言しているだけなので、問題はない。問題があるとすれば、NCHARACTER型の宣言である。

列挙型NCHARACTERに 256個の要素を宣言すること自体は、問題がない。しかし、その要素にＡＳＣＩＩ文字以外の文字リテラル（日本語文字の片割れ）がある点が問題である。Ａｄａの基準文法書（ＬＲＭ）に次のような表現がある　：”The only characters allowed in the text of a program are the graphic characters and format effectors. Each graphic character corresponds to a unique code of the ISO seven-bit coded character set ...”[3]。これは、いかなる方法においてもＡｄａに日本語文字リテラルを書けないことを意味する。日本語テキストを処理する場合これでは大変不便である。そこで、やむを得ず、若干の文法拡張を行なうことにする。プログラムテキスト中に日本語文字を書くことを許すには、基準文法書でいう「graphic characters」を拡張しなければならない。 graphic characterの定義は次のようになっている。

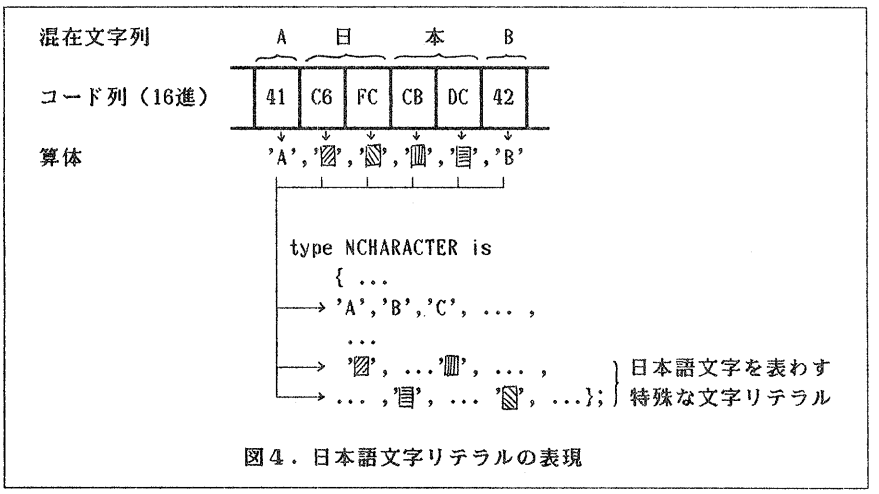

図４．日本語文字リテラルの表現

```
graphic_character::= basic_graphic_character
                   | lower_case_letter
                   | other_special_character
```

ここで、basic_graphic_character は英大文字、数字、空白やプログラムテキストにおいて特別な意味を持つ特殊記号を表わす。また、other_special_character は、特に意味を持たず文字リテラルおよび注釈のみで使用できる記号文字群を表わす。そこで、この other_special_characterを以下のように拡張する。

●other_special_character の定義を拡張し、日本語文字を表わすために必要なリテラルを含める。

なお、この拡張は文字型 CHARACTERの拡張を意味しない。これは、拡張を最小限に留めるためである。

4．混在文字列の正規化

この章では、混在文字列における問題点とその解決法として正規化法の適用について考察する。

4．1　混在文字型の問題点

3章で述べた方法で日本語テキスト処理は可能になるけれども、混在文字型では、ＡＳＣＩＩ文字は算体1個、日本語文字は算体2個と言うように、算体の数と文字数の関係が一定でない。これにより、我々は、混在文字列を操作する場合、現在着目している文字がＡＳＣＩＩ文字か日本語文字かを常に意識していなければならず、数々の不都合が起こる。

例えば、ある混在文字列の中のある文字をポインタが指しているとする。このポインタを更新して次の文字を指示したい場合、現在ポインタが指している文字がＡＳＣＩＩ文字なら＋1、日本語文字なら＋2と場合分けしてポインタを進めなければならない。

この場合分けを怠ると、ずれ読みが起こる可能性がある。ずれ読みとは、日本語文字の第1バイトと第2バイトのシーケンスがずれて他の文字に化ける現象をいう。例えば、図5の様なテキストで「娘」という文字を算体1つごとに捜したとき、「般若」のところでマッチしてしまう。

4．2　正規化日本語文字列

ＡＳＣＩＩ文字か日本語文字かを常に意識したプログラムを書くことは大変面倒である。そこで、このプログラミング上の面倒を取り除くため、我々の研究室では、「正規化日本語文字列」を提案して来た[4]。正規化日本語文字列とは、1バイトで表現するＡＳＣＩＩ文字コードにはNULLコードを付加し、すべての文字を2バイトで表現するものである。（図6）

この表現によれば、全ての文字は2バイトでコード化されているとみなせるため、ポインタ更新の際の場合分けは必要なくなる。

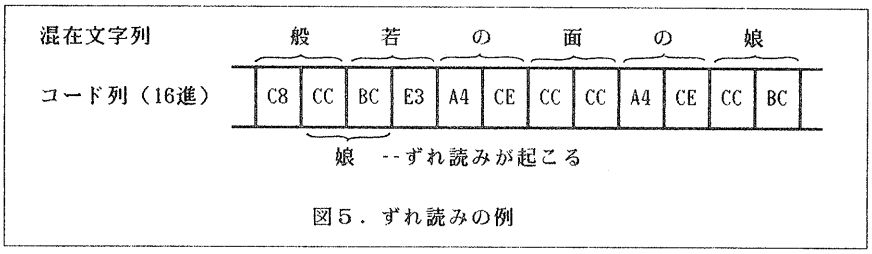

図５．ずれ読みの例

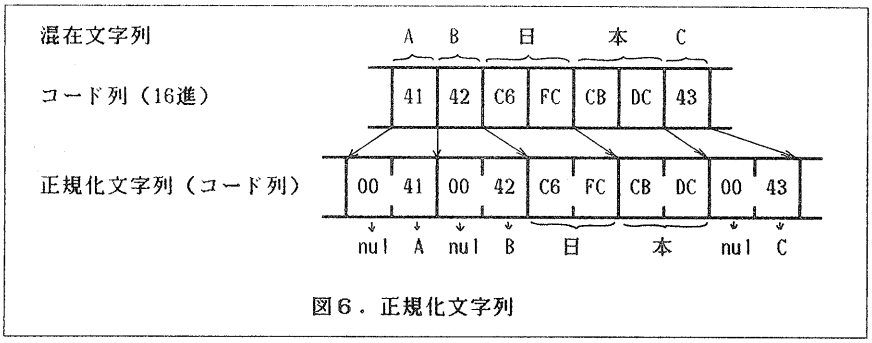

図６．正規化文字列

４．３　正規化文字型の導入の経緯

日本語文字を扱うことができるＦＯＲＴＲＡＮにおいて正規化日本語文字列を実現するには大抵の場合に文字型の算体を使っている。つまり、連続する文字列の要素２つで日本語文字１つを表現する。

Ａｄａにおいて、この正規化文字列をどう表現するか、その決定に至るまで紆余曲折があった。この経緯を順を追って述べる。

i)文字を１つの算体で表わす

上記のＦＯＲＴＲＡＮにおけると同様に、正規化日本語文字列の１文字を混在文字列型の要素２つを使って表わすことはＡｄａの言語設計の背景にあるデータ抽象の概念に反すると考え、１文字を１つの算体としてみなせるような正規化文字型NORM_NCHおよび正規化文字列型NORM_NSTを宣言することにした。この型の実現は次のようなものである。

◇文字を表わす２バイトのコードを１６ビットの数値とみて、数値型の派生型として宣言する。

```
type NORM_NCH is new INTEGER ;
type NORM_NST is array( POSITIVE
                range <> ) of NORM_NCH ;
```

さらに、この型をprivate 型とすることで、数値型であることを隠すことにした。

ii）文字を混在文字列の２つの要素で表わす

一方、我々の研究室で行なっている混在文字列におけるパターンマッチの研究で、正規化日本語文字列を１バイトごとのコード列とみてパターンマッチを行なうと効率のよい結果を得ることが分かっている[5]。Ａｄａでプログラムを書く場合にもこの手法を使いたいという要求があった。データ抽象と効率とのトレードオフの結果、正規化文字を１つの算体で表わすことはあきらめ、上記のＦＯＲＴＲＡＮの場合と同じく、混在文字列の要素２つで表わすことにした。

iii)文字を１つの構造型算体で表わす

１文字を混在文字列の２つの要素で表わす表現法を採用したのは、効率面を重視した結果であるが、データの抽象化やプログラムの読解性を妨げる。そこで、この表現法が妥当なものかどうか、他方面からの意見を求めた。その結果、「１文字は１つの算体で扱えるようにすべきである」とのことであった。この意見を取り入れ、さらに、ii）で述べたような、１バイトごとのコード列とみなす手法を用いる場合にも効率の低下を極力抑えるような型を設けることにした。それは、以下のようなものである。

◇文字をレコード型の算体とし、その中に混在文字型の要素２つを宣言する。

```
type NORM_NCH is record
    HIGH : NCHARACTER ;
    LOW  : NCHARACTER ;
end record ;
type NORM_NST is array( POSITIVE
                range <> ) of NORM_NCH ;
```

１文字は１つの構造算体とみなして操作できる。しかし、１バイトごとのコード列と見る場合は、正規化文字コードの第１バイト（奇数番目）に着目したいならばHIGH要素、第２バイト（偶数番目）に着目したいならば LOW要素と、場合分けして要素指定しなくてはならない。

４．４　相互変換パッケージ

混在文字型算体と正規化文字型算体を相互に変換する手続きを定義したパッケージＣＯＮＶＥＲＴを用意した。このパッケージにより、外部との入出力には混在文字型、内部処理には正規化文字型と使い分けることができる。

また、このパッケージは、２つの型の算体を相互に変換するのみでなく、入出力デバイスにおいて、混在文字型の入力文字列を正規化文字型に変換して入力する手続きや、正規化文字型の算体を混在文字型に変換して出力する手続きも宣言している。文字単位の処理のみしか行なわない場合には、これらの手続きが、混在文字型やコード体系などの知識をカプセル化し、混在文字型などを全く意識することなくプログラミング可能である。

４．５　日本語テキスト処理パッケージ

日本語テキスト処理機能は以下の３つのパッケージにより実現している。（付録参照）

1.パッケージNCHAR_DEF
　混在文字型、混在文字列型の宣言
2.パッケージNCHAR_IO
　混在文字型、混在文字列型に対する操作、入出力手続きの宣言
3.パッケージCONVERT
　正規化文字型、正規化文字列型の宣言
　混在文字（列）型、正規化文字（列）型の相互変換手続き、正規化文字（列）型の入出力手続きの宣言

５．使用経験

これまで述べた３つのパッケージは、九州大学情報工学科の教育用計算機システムECLIPSE MV/10000上のＡｄａで実現している。このパッケージを使用

して実際にソフトウェアの開発を行なっているので、その使用経験について述べる。

我々の研究室では日本語文章推敲支援ツール「推敲」[6] を開発している。この「推敲」をECLIPSE MV/10000上に実現するのにＡｄａを用いた。この過程でいくつかの結果を得て、その一部はパッケージの仕様にフィードバックした。

５．１　パッケージと可視性の問題

指定したパッケージを直接可視にする use節は、パッケージを示す拡張名を省くために使うが、乱用するとプログラムの読解性を損なう。しかし、 use節は指定したパッケージ内で暗黙のうちに宣言される基本操作（各種演算など）への可視性にも影響するため、全く使わないことにすると、かえって読解性を損なうこともある。

例えば、with節のみで NCHAR_DEFを指定する場合とuse 節においても指定する場合を考えると、演算子"&" の表記は次のように変わってくる。

```
with NCHAR_DEF ;
procedure TEST is
  WORK : NCHAR_DEF.NCHARACTER ;
  NBUF : NCHAR_DEF.NSTRING( 1 .. 10 ) ;
  begin
    WORK := NCHAR_DEF.NCHARACTER'VAL(164);
    NBUF(1..2) := NCHAR_DEF."&"(WORK,WORK) ;
        ...
```

◇with節のみで指定した場合

```
with NCHAR_DEF ;  use NCHAR_DEF ;
procedure TEST is
  ...
    NBUF(1..2) := WORK & WORK ;
      ...
```

◇with節、use 節の両方で指定した場合

ここで示した "&"演算子は基本操作の１つで、NCHARACTER型や NSTRING型の型宣言の直後暗黙に宣言される。両者をみても明らかなように、直接可視の場合（ use節で指定している場合）のほうがプログラムが読みやすい。また、混在文字リテラルもパッケージNCHAR_DEF が直接可視でなければ使えない。従って、パッケージNCHAR_DEF やパッケージCONVERT を使う場合には、with,use両方で指定した方がプログラムを書く上でも、また読む上でも実用的である。

５．２　正規化文字は構造算体である

正規化文字型はレコード型として宣言した。つまり、正規化文字は、その値として構造値を持つ。構造値は一般の数値型や列挙型の値と違い、既定の順序関係や部分範囲などの定義がない。従って、順序比較演算が暗黙のうちに宣言されることはなく、制約を含んだ部分型(制約付きの subtype)を宣言することができない。具体例を次に挙げる。

▼文字の順序比較演算がない

混在文字列の "あ"< "い" のように正規化文字を比較し、そのコード値において順序判断をする不等号演算が暗黙に宣言されていない。

これはコード値と文字との関連を知っているユーザにとって不便である。また一方で、文字の表現をカプセル化し、ユーザには内部を見せないという立場からは、この関数は必要ないものである。ここでは実用を重んじ、不等号による順序比較関数をパッケージCONVERT 内に陽に宣言している。

▼字種に関する制約付き部分型を宣言できない

CHARACTER 型においては以下のように、制約付き部分型を宣言することができ、この型を利用して文字を弁別することができる。

```
subtype NUMBER is CHARACTER range '0'..'9';
CH : CHARACTER ;
  begin
    ...
    if CH in NUMBER then

      --CHの内容が数字であった場合の処理

    end if;
    ...
```

◇ CHARACTER型の部分型とその使用例

正規化日本語文字列から平仮名列だけを取り出すなど、字種を弁別したい場合は多い。しかし、上記のような制約付き部分型を正規化文字型において宣言し文字種を弁別することはできない。この解決法として関数により文字種の弁別を行なう。正規化文字型に対して文字種を弁別する関数の実現は、比較的容易である。なぜなら、文字種の判断は、日本語文字コードの第１バイトを参照するだけでよいことが多く、第１バイトと第２バイトの区別が明確である正規化文字型はこの参照が容易に行なえるからである。

▼正規化文字型の文字定数を直接記述できない

正規化文字型定数（正規化文字リテラル）を直接、混在文字リテラルで表現することはできない。混在文字リテラルは混在文字（列）型であり、正規化文字型にするには、パッケージCONVERT の手続きMK_NORM を使って変換する。しかし、手続きMK_NORM だけでは、定数宣言ができないため、関数MK_NORM を用意した。

```
NNBUF:NORM_NST( 1..100 ) ;
RE   :NORM_NCH ;
begin
  MK_NORM( "れ",RE ) ;
  ...
  if NNBUF(I) = RE then
    ...
    ◇手続きMK_NORM のみしかない場合
```

```
NNBUF:NORM_NST( 1..100 ) ;
RE   :constant NORM_NCH:=MK_NORM("れ");
begin
  ...
  if NNBUF(I) = RE then
    ...
    ◇関数MK_NORM がある場合
```

５．３　混在文字型と正規化文字型の使い分け

正規化文字型は抽象物としての文字を処理する場合に適しているが、入出力主体の処理には入出力するコードと直接対応する混在文字型が適していると考えていた。ところが、必ずしもそうではない場合があることがわかった。

６．あとがき

Ａｄａに日本語テキスト処理機能を加える方法、および使用に際して正規化文字型の有用性について述べた。もともとＡｄａの言語設計にあたって、多字種を有する文字集合の存在は考慮に入っていなかったものと考える。パッケージを利用することにより、多字種を有する言語のテキスト処理にＡｄａを使う道を、比較的無理なく拓くことができた。ここで述べた事柄は、単に日本語文字を使うためのみでなく、中国語や韓国語などの多字種を扱う場合や、マルチフォント処理を行なう場合にも応用できるであろう。しかし、多字種の存在を最初から考慮したプログラミング言語の設計が重要と考える。その際に、ここでの検討事項が種々参考になるものと期待している。

なお、この研究の一部は昭和60年度科学研究費補助金一般研究（Ｂ）60460228の援助を受けた。

参考文献

[1]木下：新しい日本字コード系による日本語処理の実現，情報処理学会マイクロコンピュータ研究会資料，No.37-1（1985）

[2]福山、小林、伊集院、木下、落合：Ａｄａ日本語処理機能の実現法，情報処理学会第30回全国大会講演論文集，pp.459-460.（1985）

[3]ＡＮＳＩ：”Reference manual for the Ada programming language,” ANSI/MIL-STD-1815A-1983

[4]川副、吉田、牛島：日本語テキスト処理プログラムの流通性を上げるためのプリミティブ，電子通信学会電子計算機研究会資料　EC84-59（1985）

[5]尹、高木、牛島：日本語テキストにおけるパターンマッチング手法の比較と改善，情報処理学会第38回記号処理・第５回プログラミング言語合同研究会資料，（1986）

[6]牛島、日並、尹、高木：日本語文章推敲支援ツールのプロトタイピング，コンピュータソフトウェア，Vol.3,No.1(1986),pp.35-46.

※Ａｄａは米国政府Ada Joint Program Officeの登録商標である。

付録

```
package NCHAR_DEF is

    type NCHARACTER is
  ( nul,soh,stx,etx, eot,enq,ack,bel,
    bs ,ht ,lf ,vt , ff ,cr ,so ,si ,
    dle,dc1,dc2,dc3, dc4,nak,syn,etb,
    can,em ,sub,esc, fs ,gs ,rs ,us ,
    ' ','!','"','#', '$','%','&',''',
    '(',')','*','+', ',','-','.','/',
    '0','1','2','3', '4','5','6','7',
    '8','9',':',';', '<','=','>','?',
    '@','A','B','C', 'D','E','F','G',
    'H','I','J','K', 'L','M','N','O',
    'P','Q','R','S', 'T','U','V','W',
    'X','Y','Z','[', '¥',']','^','_',
    '`','a','b','c', 'd','e','f','g',
    'h','i','j','k', 'l','m','n','o',
    'p','q','r','s', 't','u','v','w',
    'x','y','z','{', '|','}','~',del,

    X80,X81,X82,X83, X84,X85,X86,X87,
    X88,X89,X8A,X8B, X8C,X8D,X8E,X8F,
    X90,X91,X92,X93, X94,X95,X96,X97,
    X98,X99,X9A,X9B, X9C,X9D,X9E,X9F,
    '。','「','」','、','・','ヲ','ァ','ィ',
    'ゥ','ェ','ォ','ャ','ュ','ョ','ッ','ー',
    'ア','イ','ウ','エ','オ','カ','キ','ク',
    'ケ','コ','サ','シ','ス','セ','ソ','タ',
    'チ','ツ','テ','ト','ナ','ニ','ヌ','ネ',
    'ノ','ハ','ヒ','フ','ヘ','ホ','マ','ミ',
    'ム','メ','モ','ヤ','ユ','ヨ','ラ','リ',
    'ル','レ','ロ','ワ','ン','゛','゜','  ' )

    for NCHARACTER'SIZE use 8 ;

    type NSTRING is array ( POSITIVE
                range <> ) of NCHARACTER;
    pragma PACK( NSTRING ) ;

end NCHAR_DEF ;
```

１．パッケージ NCHAR_DEF

```
with NCHAR_DEF ;
with COMMON_IO ;
package NCHAR_IO is

    use NCHAR_DEF ;

    subtype FILE_TYPE is COMMON_IO.FILE_TYPE ;

    type FILE_MODE is ( IN_FILE , OUT_FILE ) ;

procedure NOPEN( FILE : in out FILE_TYPE ;
                 MODE : FILE_MODE ; NAME : STRING ) ;
procedure NCLOSE( FILE : in out FILE_TYPE ) ;

function END_OF_FILE( FILE : FILE_TYPE ) return BOOLEAN ;

-- NCHARACTER  Input - Output
procedure NGET( FILE : FILE_TYPE ; NCH : in out NCHARACTER ) ;
procedure NGET( NCH  : in out NCHARACTER ) ;
procedure NPUT( FILE : FILE_TYPE ; NCH : NCHARACTER ) ;
procedure NPUT( NCH  : NCHARACTER ) ;

-- NSTRING   Input - Output
procedure NGET( FILE : FILE_TYPE ; NBUF : in out NSTRING ) ;
procedure NGET( NBUF : in out NSTRING ) ;
procedure NPUT( FILE : FILE_TYPE ; NBUF : NSTRING ) ;
procedure NPUT( NBUF : NSTRING ) ;

-- NSTRING line Input - Output
procedure NGET_LINE( NBUF : in out NSTRING ;
                     LENGTH : in out INTEGER ) ;
procedure NPUT_LINE( NBUF : NSTRING ) ;

-- others
procedure NL ;

-- Convert CHARACTER <--> NCHARACTER
function NCHAR_TO_CHAR( NC   : NCHARACTER ) return CHARACTER ;
function NCHAR_TO_CHAR( NBUF : NSTRING )    return STRING ;
function CHAR_TO_NCHAR( C    : CHARACTER )  return NCHARACTER ;
function CHAR_TO_NCHAR( BUF  : STRING )     return NSTRING ;
end NCHAR_IO ;
```

2．パッケージ NCHAR_IO

```
with NCHAR_DEF ;   use NCHAR_DEF ;
with NCHAR_IO ;
package CONVERT is

   type NORM_NCH is record
       HIGH : NCHARACTER ;
       LOW  : NCHARACTER ;
   end record ;

   type NORM_NST is array ( POSITIVE range <> ) of NORM_NCH ;

------- convert Normalized type with NSTRING --------
   procedure MK_NORM( NBUF  : NSTRING ;
                      NRMBF : out NORM_NST ; LENGTH : out INTEGER) ;
   procedure MK_NSTR( NRMBF : NORM_NST ;
                      NBUF  : out NSTRING  ; LENGTH : out INTEGER) ;
   function MK_NORM( NBUF : NSTRING ) return NORM_NCH ;

-------- I/O with the Normalized type ----------
   procedure PUT_NORM( FILE  : NCHAR_IO.FILE_TYPE ;
                       NRMBF : in NORM_NST ) ;
   procedure PUT_NORM( NRMBF : in NORM_NST ) ;
   procedure PUT_NORM_LINE( FILE  : NCHAR_IO.FILE_TYPE ;
                            NRMBF : in NORM_NST ) ;
   procedure PUT_NORM_LINE( NRMBF : in NORM_NST ) ;

   procedure GET_NORM( FILE  : NCHAR_IO.FILE_TYPE ;
                       NRMBF : out NORM_NST ; LENGTH : out INTEGER ) ;
   procedure GET_NORM( NRMBF : out NORM_NST ; LENGTH : out INTEGER ) ;
   procedure GET_NORM_LINE( FILE : NCHAR_IO.FILE_TYPE ;
                            NRMBF :out NORM_NST ; LENGTH :out INTEGER );
   procedure GET_NORM_LINE( NRMBF :out NORM_NST ; LENGTH :out INTEGER );

----------------- code function ------------------------------
   function CODE( CHR : NORM_NCH ) return INTEGER ;
   function CODE_HIGH( CHR : NORM_NCH ) return INTEGER ;
   function CODE_LOW ( CHR : NORM_NCH ) return INTEGER ;
   function NORMCHR( CODE : INTEGER ) return NORM_NCH ;
   function NORMCHR( HIGH , LOW : INTEGER ) return NORM_NCH ;

   function ">"( LEFT , RIGHT : NORM_NCH ) return BOOLEAN ;
   function "<"( LEFT , RIGHT : NORM_NCH ) return BOOLEAN ;
   function ">="( LEFT , RIGHT : NORM_NCH ) return BOOLEAN ;
   function "<="( LEFT , RIGHT : NORM_NCH ) return BOOLEAN ;
---------------- CONSTANT ---------------------
   NORM_LF  : constant NORM_NCH :=
                  ( HIGH => NCHARACTER'( NUL ) ,
                    LOW  => NCHARACTER'( LF )  ) ;
end ;
```

3．パッケージ CONVERT