記 号 処 理 36－4
プログラミング言語 5－4
（１９８６．３．１０）

ＴＡＯにおける日本語文字列処理

ＮＴＴ 電気通信研究所
杉村 利明 奥乃 博 竹内 郁雄

１．まえがき

我々は、LispマシンELIS上に、知能的な統合プログラミング環境ＮＵＥ（New Unified Environment)を構築する研究を進めている[1,2]。そして、ＮＵＥの「核言語」であるＴＡＯを開発している。

ＴＡＯは、記号処理言語Lisp、論理型言語Prolog、対象指向型言語Smalltalk-80の機能を同一枠組みの中で使えるようにした統合言語である。プログラマはアプリケーションに適した機能（言語）を選んで効率的にプログラムを作ることができる。

さらに、ＴＡＯは文字列処理についても、一つの枠組みの中に最大限の自由度を与えるという考えに基づき、日本語文字とASCII文字とを自由自在に処理できる統合した文字列処理機能を有している（以下、統合文字列処理と呼ぶ）。

本稿では、統合文字列処理に対する基本的な考え方、実現方法、処理効率、使用経験について報告する。

２．基本的な考え方

ＴＡＯにおける、文字列処理に対する基本的な考え方は次の２点である。

（１）日本語文字とASCII文字の統合文字列処理の実現

日本語文字とASCII文字の文字列処理を一つの枠組みに統合化して、両者を区別なく自由自在に処理できるようにする。これにより、従来からのASCII文字を基本にしたプログラム作りが、日本語についてもそのまま適用可能となり、日本語ソフトウェアの実現が容易になる。

（２）文字列処理の効率化

文字列処理がアプリケーション・プログラムの中で大きな割合を占めている。エディタやかな漢字変換プログラムにおいても文字列処理の割合が大きい。したがって、文字列処理機能はアプリケーション・プログラムを作る上で重要な機能の一つであり、効率化が重要である。

３．文字列処理実現上の基本的な考え方

（１）文字列処理の基本操作は、次の３つの操作であると考え、これを効率よく実行できることを設計の基本とした。

(i)文字列の先頭の部分文字列を取り出す shead

```
ex. (shead "abcd")   --> "a"
    (shead "abcd" 3) --> "abc"
```

(ii)文字列の末尾からの部分文字列を取り出す stail

```
ex. (stail "abcd")   --> "bcd"
    (stail "abcd" 3) --> "d"
```

(iii)文字列の連結を行う string-append (sconc とも呼ぶ)

```
ex. (string-append "abcd" "xyz") --> "abcdxyz"
```

この操作をlist処理との対応でとらえると以下の様に考えることができる。

list処理		文字列処理	
car	(car '(a b c d)) --> a	**shead**	(shead "abcd") --> "a"
cdr	(cdr '(a b c d)) --> (b c d)	**stail**	(stail "abcd") --> "bcd"
append	(append '(a b) '(c d)) --> (a b c d)	**string-append**	(string-append "ab" "cd") --> "abcd"

（２）文字列のＧＣ（Garbage Collection)が１文字単位で行え、メモリが効率的に利用できることを方針とした。

（３）日本語文字については、次の点を方針とした。

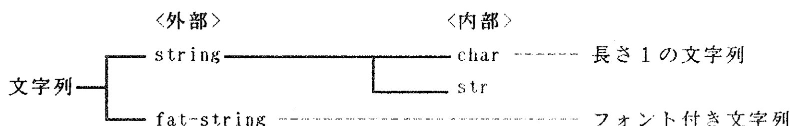
(i)日本語文字とASCII文字が混在してもメモリ容量が増大しない。

(ii)日本語文字を処理するにあたって、従来のASCII文字の処理が遅くならない。

４．文字列の構造

文字列は外部から見ると、フォント情報を持たない通常の文字列であるstringと、フォント情報を持った文字列のfat-stringの２種類がある。

stringは内部的には、str型とchar型から構成される。メモリの節約のために、１文字のstringがchar型で表現される（システムのモードの切り替えによって、実際に長さ１のstr型で表すことも可能）。



４．１ char型の構造

char型のlisp objectの構造を図4.1に示す。ＴＡＯでは、tagがlisp pointerに付随しており、pointer部の指すデータの型を示している（char型や数値の場合は、pointer部自身がその値を表している）。図では、tagの値はchar型を示すtagcharである。codeはASCII文字の場合、bit15-7はすべて０である。日本語文字の場合は、bit15-0 にシフトJIS（JIS C 6226 漢字コードの各バイトのbit7を１にしたもの）が入る。

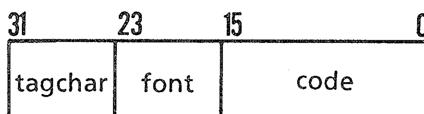


図４．１　char型の構造

４．２ str型の構造

str型のlisp objectの構造を図４．２に示す。str型は、lisp pointerがstring headというpointerを介して、str-memblkというメモリ領域の中の、実際の文字列が入っているstring bodyを指すという、２段階のデータ構造を構成している。

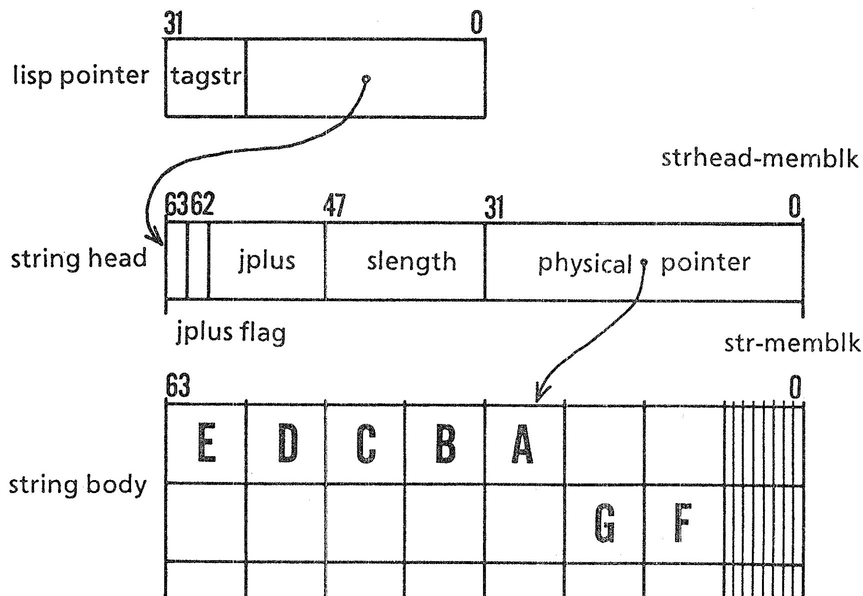


図４．２　str型の構造

（１）string header

string headerには、文字列の長さ、文字列中の日本語文字の個数、string bodyへのpoiterが入っている。構造は以下の通りである。

bit 63	gcmark	--- ＧＣのためのマークビット
bit 62	jplus flag	--- 日本語文字が含まれる時に１となる
bit 61-48	jplus	--- stringの中の日本語文字の個数
bit 47-32	slength	--- stringの長さ（２byteの日本語文字を１文字と数える）
bit 26- 0	physical pointer	--- lisp objectでないものを指すpointer

（２）string bodyとstr-memblk

str-memblkのbyte7-1の７バイトが、string bodyとして実際の文字コードを収容するのに使われる。文字コードは、ASCII文字の場合、bit7が０の１バイトで表され、日本語文字の場合、bit7が１の２バイトで表される。byte0はＧＣのためのマークビットとして使われる。

４．３　新しく文字列を作る時の方法

string-appendやread関数によって新しく作られる文字列のstring bodyは、最後に作られた文字列のstring bodyのすぐ隣から始まる。しかし、最後に作られた文字列のstring bodyの入っているword(64bit)の残りのバイト数が、２または３より小さい場合、新しい文字列のstring bodyは以下の条件によって次のwordから始まることがある。

(1)新しい文字列の長さが残りのバイト数より小さい場合は、すぐ隣から始まる。

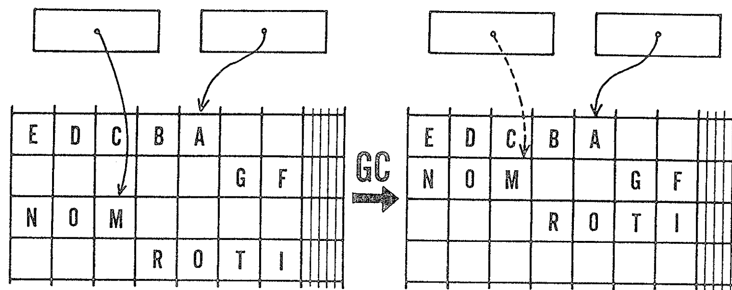
(2)新しい文字列の長さが8より小さく、残りのバイト数が3より小さい場合は、次のwordから始まる。

(3)それ以外は、すぐ隣から始まる。

一方、部分文字列がsheadやstailで求められる場合は、新しい文字列が作られるのではなく、string headだけが作られて、既に存在する文字列の一部がstring bodyとして指される。

4.4 文字列のＧＣ(Garbage Collection)

文字列のＧＣでは、最初にstring headerがマークされ、その後str-memblkがマークされword単位でcompactionされる。"MONITOR"が上にcompactionされ、その後にphysical pointerの付け替えが行われる様子を図4.3に示す。



上にcompactionされ、その後、physical pointerの付け替えがおこなわれる。

図4.3 文字列のＧＣ

5．ＥＬＩＳのＭＧＲ（Memory General Register）とＳＤＣ（Source Destination Register）による文字列処理の高速化

ＭＧＲはメモリからの読み込みと書き込みのできるregister fileであり（図5.1参照）、以下の機能がある[3]。

(1)メモリと64bit幅で読み込みと書き込みができる。

(2)ＡＬＵとは32bit幅、16bit幅、8bit幅で読み込みと書き込みができる。通常は32bit幅である。

	CAR側 (31 … 0)	CDR側 (31 … 0)
MGR0	car0	cdr0
MGR1	car1	cdr1
MGR2	car2	cdr2
MGR3	car3	cdr3

図5.1 ＭＧＲの概要

ＳＤＣはＭＧＲを指す一種のindex registerで、auto-incrementも可能で、これを用いてＭＧＲを任意の8bit、16bit単位でアクセスできる。このことにより、ＭＧＲは文字列処理用のcache registerとして使えることになる（7バイト分のcache）。これが文字列処理の高速化の鍵になっ

ている。図５．２はＳＤＣを用いてauto-increment機能により、ＭＧＲの文字列をたどる例である。日本語文字とASCII文字の判別については、水平型マイクロにより、ほとんどオーバーヘッドが生じていない。

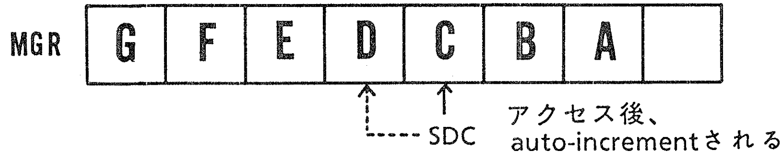


図５．２　ＳＤＣを用いた文字列のアクセス

６．文字列処理関数の概要

６．１　基本関数

日本語文字とASCII文字とを区別しないで自然に使える基本的な関数の主なものを以下に示す。

（１）Operation

- (shead "abcと日本語") --> "a"
  (shead "abcと日本語" 4) --> "abcと"
- (stail "abcと日本語") --> "bcと日本語"
  (stail "abcと日本語" 4) --> "日本語"
- (string-append "abc" "と" "日本語") --> "abcと日本語"
- (substring "abcと日本語" 2 5) --> "cと日"
  nsubstringはsubstringと同じ機能であるが、string headだけを作る。
- (slength "abcと日本語") --> 7
  (slength "") --> 0

（２）Predicate

- (charp "a") --> "a"
  (charp "あ") --> "あ"
  (charp "aa") --> nil
- (stringp "abcと日本語") --> "abcと日本語"

（３）Comparison

- (string= "abcと日本語" "abcと日本語") --> "abcと日本語"
  (string= "abc" "日本語") --> nil
- (string< "abc" "日本語") --> "日本語"

（４）その他

- (string-reverse "abcと日本語") --> "語本日とcba"
- (string-search "日本語" "abcと日本語") --> 4
- (exploden "abcと日本語") --> (97 98 99 42184 50940 52183 47340)

６．２　日本語文字のための関数

- (jcharp "あ") --> "あ"　　　…… 日本語文字か調べる
  (jcharp "日本語") --> nil
  (jcharp "a") --> nil
- (jstringp "abcと日本語") --> 4　　　…… 日本語文字を含むか調べる
  (jstringp "abc") --> nil
- (string-byte-count "abcと日本語") --> 11 …… 文字列のバイト数

6.3 検討を要する関数

(1) 日本語文字の中の英数字の扱いについて検討を要する関数の例を以下に示す。

graphic-char-p, alpha-char-p, upper-case-char-p, lower-case-charp
both-case-p, digit-char-p, alphanumeric-char-p
char-upcase, char-downcase, string-upcase, string-downcase

(2) 日本語文字とASCII文字の間の変換関数の例を以下に示す。

char-kanji …… ASCII文字を対応する日本語文字に変換する
char-standard …… 日本語文字を対応するASCII文字に変換する

7. 文字列処理の速度

(1) TAOの文字列処理の速度を基本的な関数について以下に示す。

①string compare

(string= "a" "a") … 13μsec
(string= "abcde" "abcde") … 22
(string= "abcdefghij" "abcdefghij") … 30

②sheadとstail

(shead "abcde") … 25
(stail "abcde") … 28

③string search

(string-search "bbbbb" x) … 36.4msec
ただし、x は "aaa … aaabbbbb" で長さ10000である。

(2) 各lisp処理系の文字列処理の速度比較を文献4より引用して表7.1に示す(string-taraiの定義を図7.1に示す)。TAOは、tarai4 と string-tarai4 の速度比が一番小さい。また、string-tarai4 の速度についても大型計算機の処理系に近い速度が得られている。このことから、TAOがlist処理と同様に文字列処理についても強力であることがわかる。

表7.1 tarai4とstring-tarai4の速度比較(単位はmsec)

処理系	tarai4	string-tarai4	速度比
FRANZ LISP (M-280H)	70.7	2948	42
CAMBRIDGE LISP (M-280H)	66.5	490	73
FLATS LISP (FLATS)	18.0	3862	214
FRANZ LISP (VAX-11/780)	4397	37370	8
HLISP (MELCOM COSMO 800/3)	2014	24360	12
INTERLISP-D (XEXOX 1100 SIP)	990	64006	65
KYOTO COMMON LISP (MV10000)	510	8431	17
TAO* (ELIS)	628	1780	3
LISPVM (3081K)	137	838	6
SYMBOLICS COMMON LISP (3600)	85	6921	81
ZETALISP (3600)	119	25167	211
UTLISP (M-280H)	15.2	350	23
ZETALISP-PLUS (LAMBDA)	682	25250	37

*はInterpreted codeの速度である。それ以外は、Compiled codeの速度である。

```
(DEFUN STRING-TARAI (X Y Z)
  (COND ((STRING-LESSP
          (SUBSTRING X 0 1) (SUBSTRING Y 0 1))
         (STRING-TARAI (STRING-TARAI (SUBSTRING X 1) Y Z)
                       (STRING-TARAI (SUBSTRING Y 1) Z X)
                       (STRING-TARAI (SUBSTRING Z 1) X Y) ))
        (T Y) ))
(STRING-TARAI "ABCDEFGHIJ" "EFGHIJ" "IJ") <--- string-tarai4
    ; where string-tarai is called 12605 times.
    ; stail is called 9453 times.
```

図７．１ string-taraiの定義

８．文字列処理の使用例

アプリケーション・プログラムにおける文字列処理の使われ方、日本語文字とASCII文字の統合文字列処理の効果を以下の例で示す。

８．１ かな漢字変換プログラムの作成

（１）プログラムの構造

かな漢字変換プログラムは大きく分けて、辞書との照合により単語・付属語の抽出を行う部分、抽出された文法情報により文法検定を行う部分、検定結果の順位づけを行う部分からなる。この中で、文字列処理を基本にしているのは、入力文字列のすべての部分文字列について辞書との照合を行う単語・付属語抽出処理である。それ以外の部分は、通常のlist処理が基本となっている。

（２）文字列処理関数の使われ方

かな漢字変換プログラムで使われている文字列処理関数の内訳を、ソースプログラム上での出現の割合（staticな性質）として以下に示す。特に、文字列処理では、list処理と異なり長さを求める関数が多く使われている。

〈関数の種類〉		〈割合〉
文字列の長さを求める関数（slength）	…	５０％
部分文字列を求める関数(char, shead, nsubstringなど）	…	２７％
比較を行う関数（string<, string=, string/=など）	…	２０％
文字列を連結する関数（string-append）	…	３％
部分文字列の置き換え関数（string-fillなど）	…	０％

（３）統合文字列処理の効果

日本語文字とASCII文字を処理するためには、従来はプログラマが１バイトの文字なのか２バイトの文字なのかを自分で意識しながらプログラムを作成する必要があり、プログラム作成の効率を低下させていた。しかし、ＴＡＯでは、統合文字列処理により両者の違いを特に意識してプログラムを作る必要がなく、かな漢字変換のプログラム作りが自然に効率よく行えた。

（４）その他

単語辞書の収容語数は数万語程度で、これをlispの基本的なデータobject（例えば、listを使う）で表すと、general purposeなデータ構造であるために、メモリを大量に使い効率が悪い。言語処理のために、大量の辞書データを効率よく実現できる機能がlispに必要であると考える。

また、辞書をファイルに入れる場合は、必要な単語をそのつどＳ式に変換して読み込む必要があるので、変換にかかる時間が問題になる。これには、ファイルのデータを変換なしに直接読み込める機能と、これを表現できるデータobjectが必要と考える。

8.2 アプリケーション・プログラムの日本語化

（1）Ｇｒｅｐ

GrepはASCII文字を対象に作られた、文字列のパターン照合プログラムであり、日本語文字のことは全く意識しないで作られた。しかし、プログラムはそのままで日本語文字についてもASCII文字の場合と同様に正常に動作した。このことは、統合文字列処理により、ASCII文字を対象にしたプログラムの日本語化が容易であること、新たに作るプログラムも従来と同じ方法で作ることができることを示していると言えよう。

（2）Ｆｅｄｉｔ

プログラムの変更は、(i)かな漢字変換を組み込むためにプログラムを修正した、(ii)画面に実際に表示される文字列の長さを求めるために、表示を行う関数のslengthをstring-byte-countに変更した、の２点であった。(i)は新たな機能を追加するための作業でありやむおえない。(ii)が日本語化のための純粋な作業である。

すなわち、Feditの内部の文字列処理の変更の必要はなく、画面表示の部分だけが変更の必要があったことになる。統合文字列処理により、日本語化のための変更は少なく済んだと言えよう。今後、入力と表示について日本語文字とASCII文字の統合がさらに進めば、入力から表示まで両者を特に意識しないでプログラムを作ることができるようになろう。

9．まとめ

ＴＡＯの統合文字列処理について、実現方法、処理効率、使用経験を述べた。ＴＡＯの文字列処理の実行効率については、他のLisp objectの処理の場合と同様に、効率が高いことを示した。ＴＡＯの日本語文字とASCII文字の統合文字列処理の効果については、既存のプログラムの日本語化が容易であること、新たにプログラムを作る場合も従来と同じ作り方で効率よく行えることを示した。今後は，ＮＵＥ全体としての日本語文字とASCII文字の統合化を進めて行きたい。

〈謝辞〉　本研究を進めるにあたり御指導いただいた入力装置研究室酒井室長、第２研究室日比野室長、大野主幹研究員、ならびにGrepを使わせていただいた大井研究主任、Feditを使わせていただいた天海氏、有益な御助言、御協力をいただいたＮＵＥグループの各位に感謝いたします。

〈参考文献〉

1) Ikuo Takeuchi, Hiroshi Okuno, and Nobuyasu Ohsato: TAO --- A harmonic mean of Lisp, Prolog and Smalltalk, ACM SIGPLAN Notices, vol.18, no.7, (1983), 65-74.
2) 竹内,奥乃,大里,渡邊,日比野: New Unified Environment(NUE)の基本構想,記号処理 18-1, 1982.
3) 日比野,渡邊,大里: LispマシンELISのアーキテクチャ---メモリレジスタの汎用化とその効果,記号処理 24-3, 1983.
4) Hiroshi G. Okuno: The Report of The Third Lisp Contest and The First Prolog Contest, 記号処理 33-4, 1985.