Vol. 46 No. 11 情報処理学会論文誌 Nov. 2005

スクリプト言語Rubyの拡張可能な多言語テキスト処理の実装

松 本 行 弘[†,††] 縄 手 雅 彦[†††]

テキスト処理において文字集合間の変換は，さまざまな問題を引き起こす可能性がある．そのような問題を回避するため，スクリプト言語 Ruby に対して，複数の文字集合とそのエンコーディングを変換を行うことなく直接扱うことができる多言語テキスト処理の枠組みを実装した．この多言語テキスト処理の枠組みを利用することで EBCDIC，UTF-16，UTF-32，GB18030 をはじめとする各種 CES に比較的容易に対応できる．この多言語化により，Ruby を用いたテキスト処理の可能性が大きく広がる．また，この多言語テキスト処理のための枠組みは Ruby に依存しないため，ライブラリとして他のプログラムにも応用できる．

Multilingual Text Manipulation Method for Ruby Language

Yukihiro Matsumoto[†,††] and Masahiko Nawate[†††]

The character conversion between character code sets can cause various problems. To avoid these problems, we developed a multilingual text manipulation framework for the Ruby language, which the author designed. Users can easily add support for the new character encoding scheme (CES), such as EBCDIC, UTF-16, UTF-32, and GB18030. The multilingual text manipulation will broaden the application domain of the Ruby language. In addition, the framework for the multilingual text manipulation can be used for other programs, since they are idependent from the Ruby interpreter.

1. はじめに

各国語テキストを表現する文字集合は数多く存在しており，また，その文字集合によるテキストを表現する文字エンコーディングスキーム（Character Encoding Scheme: CES）も数多く存在している．たとえば，日本語だけに限定しても，文字集合としては JIS X0208[1] で定義されているものや，ISO 10646[2]（国内標準は JIS X0221[3]）で定義されているものなどがあり，CES には JIS X0208 による文字集合を表現するものとして ISO-2022-JP[4]，EUC-JP，Shift_JIS があり，JIS X0221 による文字集合を表現するものとしては UTF-8[5]，UTF-16，UTF-32 などがある．

このようにさまざまな CES が長い間使われてきた経緯を考慮すると，すでに存在する膨大なテキストデータの CES を統一することは現実的ではない．また，インターネットを介した国境を越えたデータやアプリケーションの流通を考えると，現代におけるテキストを処理するアプリケーションには，なんらかの方法で複数の文字集合，およびそれらの CES をサポートする機能を持たせること，すなわち多言語化（Multilingualization: M17N）が要求される．

本論文では，特にスクリプト言語を対象として，そのような多言語化アプリケーションを記述するための多言語テキスト処理機能のあり方について考察し，スクリプト言語 Ruby[6] に対して行った多言語テキスト処理機能の実装について述べる．

本研究における多言語化はテキスト処理だけを対象とし，さまざまな CES のテキストの入力，出力，CES の検出・推測，相互変換などは扱わない．

2. スクリプト言語における多言語テキスト処理機能

2.1 多言語テキスト処理機能の必要性

スクリプト言語は，ファイル処理，テキスト処理を主たる対象とするので，ユーザが日常的に使う CES を自然に処理できる必要がある．ユーザが使う CES をあらかじめ強制することはできないので必然的に多言語化に対する強い要求があることになる．現在利用中

† 株式会社ネットワーク応用通信研究所
Network Applied Communication Laboratory, Inc.
†† 島根大学大学院総合理工学研究科
Interdisciplinary Graduate School of Science and Engineering, Shimane University
††† 島根大学総合理工学部
Interdisciplinary Faculty of Science and Engineering, Shimane University

の CES の物理的な構造を意識しなければアプリケーションが記述できないようでは，十分な多言語化が行われているとはいえない．スクリプト言語のユーザが広がるにつれ，対応すべき CES も増加している．

2.2 多言語テキスト処理の実装方式

多言語化を実現する方法は，大きく分けて以下の 2 つの方式がある[7]．

- UCS（Universal Character Set）方式
- CSI（Character Set Independent）方式

UCS 方式は，テキストデータを読み込むタイミングでより大きな文字レパートリを持つ内部的な文字集合（Universal Character Set）に変換してから文字列処理を行う．内部的には単一の CES しか扱わないので，プログラムがシンプルになる．新しい外部 CES への対応は，その外部 CES から UCS への変換ルーチン（あるいはテーブル）を用意することで行う．

UCS 方式は実装がシンプルで，実行モデルが理解しやすいという利点があるが，扱える文字の範囲に対して，UCS として選択する文字集合とその CES に由来する越えがたい制約を導入することにもなる．UCS としては ISO 10646 で定義されるもの（以下 Unicode[8]）が用いられることが多く，UCS の物理表現である CES としては UTF-8 あるいは UTF-16 が採用されることが多い．Unicode による UCS 方式は，Ruby 以外のスクリプト言語の代表である Perl[9]，Python[10] によって採用されている．内部 CES としては Perl は UTF-8 を，Python は UTF-8 と UTF-16 からコンパイル時に選択するようになっている．スクリプト言語以外では Java がやはり Unicode による UCS 方式を採用している（CES は UTF-16）．より広い範囲の多言語化を目指すケースでは各種文字集合を明示的に包含した独自の CES を使うこともある．

もう一方の CSI 方式は，文字，文字列をオブジェクトと見なし，テキスト処理は抽象化したインタフェースを通じ，内部データ表現を参照することなく行う．実際に特定の CES の物理表現を参照する基本的な処理（プリミティブ）を定義し，すべての文字列処理はそのプリミティブを通じて行う．新しい CES への対応は，その CES に対するプリミティブを定義することで行う．CSI 方式はプリミティブを定義することで，対応する CES を動的に増やすことができるので，国際化の観点からは理想的ではあるが，必要十分なテキスト処理インタフェースを定義することが困難であることなどが課題としてあげられる．

2.3 Unicode を用いた UCS 方式の問題点

Unicode による UCS 方式には，先に述べたとおり，実装がシンプルになる，実行モデルが理解しやすい，などの利点があるが，その一方で以下のような問題点がある．

- テーブルによる使用メモリの増大
  Unicode と他の文字集合との相互変換は，コードポイントからの計算によって行うことは不可能で，テーブルを参照して変換する必要がある．その結果，実行時にテーブルをメモリに読み込む必要があるため，使用メモリが増大する．組み込みシステムなどメモリ制約が厳しい条件下では，Unicode 以外の CES 対応をあきらめるか，Unicode の全機能を必要としなくても変換用の比較的大きなテーブルをリンクするかの選択を迫られる．
- 文字集合変換の問題
  歴史的な事情から，同じように見えるが微妙に異なった文字集合が存在している．これらの文字集合のテキストを UCS に正規化するときにユーザの期待とは異なる正規化を行う可能性がある．具体的な例としては，漠然と Shift_JIS として分類されている CES は，実際には MS932，CP943，Shift_JIS-JISX0213 など複数の CES のいずれかである可能性があり，それぞれ Unicode へのマッピングが微妙に異なる．また，テキストに Unicode に直接マップできないベンダ定義文字，ユーザ定義文字（外字）を含む場合には，正しい UCS 正規化は定義できない．Unicode による UCS 方式を採用している Java では，相互運用性のある方法ではベンダ定義文字，ユーザ定義文字を処理できない．
- 大文字集合に対応不能
  多言語を表現する文字集合として，Unicode は最も広く使われているが，唯一のものではなく，Unicode よりもはるかに大きい文字レパートリを持つ文字集合が存在している．たとえば，TRON コード[11),12] や今昔文字鏡[13),14] がそれである．また，中国の国家規格である GB18030 は Unicode よりも広いコードポイント空間を持ち，Unicode に（まだ）割り当てられていない文字を含む．それらの文字集合を用いたテキストは，情報を失うことなく Unicode に変換することができず，結果として Unicode を利用した UCS 方式を採用した言語では処理することができない．

対応する文字集合をあらかじめ限定できる個別のアプリケーションが UCS 方式を選択することには大きな問題はないものの，プログラミング言語がテキスト処理方式として Unicode を用いた UCS 方式を選択

してしまうと，その言語で記述されるすべてのアプリケーションが Unicode による UCS 方式を強制されることになり，UCS 方式がふさわしくないケースでは，その言語の利用そのものをあきらめざるをえない．

具体的例としては以下のようなケースが考えられる．

- 近い将来 Unicode で対応されることが期待できないアジア系の言語のテキスト処理
- 古典研究などのための Unicode の範囲を越える文字集合のテキスト処理
- TRON など Unicode より大きな文字集合をサポートする OS 上でのアプリケーション記述
- 組み込みなどリソースの制約がある状況でのマルチバイト対応アプリケーション記述

アプリケーションを制限しないという観点からは，上記のようなケースにも対応できる CSI 方式がスクリプト言語の多言語テキスト処理方式として望ましい．

2.4 Ruby の多言語テキスト処理の現状

日本で開発されたスクリプト言語である Ruby の現在のバージョン（1.8.2）は，日本で広く用いられている CES である EUC-JP，Shift_JIS，UTF-8 の 3 つの CES に対応している．ただし，EUC-JP 対応は，日本語固有の処理を含まないので，中国で用いられている同種の CES である EUC-CN，韓国で用いられている EUC-KR も処理可能である．

Ruby では文字列を表現する String クラスはテキストをバイト列として扱い，文字単位の処理は原則的に正規表現を用いて行う．Ruby の正規表現エンジンは，文字種テーブルの切替えによって上記 3 種の CES に対応している．この文字種テーブルは構文解析部，文字表示部などでも用いられ，Ruby インタプリタ全体で上記 3 種の CES を切り替えて用いることができ，文字列リテラルや識別子にマルチバイト文字を直接用いることができる．ただし，識別子やリテラルに用いる CES はインタプリタ全体で統一する必要があり，混在はできない．テキストデータについては CES を明示的に指定することで，対応している複数 CES を同時に処理することができる．

3. 多言語テキスト処理機能の設計

本研究は情報処理振興財団（IPA）の 2000 年度「未踏ソフトウェア創造事業」の支援による「オブジェクト指向スクリプト言語 Ruby 次期バージョンの開発」[15] の一部として開発された Ruby の多言語テキスト処理機能を発展させたものである．以後，IPA の支援により開発された Ruby の多言語テキスト処理機能を「第 1 版」と呼び，本研究によって開発された多言語テキスト処理機能を「第 2 版」と呼ぶ．

3.1 多言語テキスト処理機能の設計方針

文字種テーブルの代わりに，いくつかのプリミティブ集合を用意することで複数の CES に対応できる CSI 方式の多言語テキスト処理機能を提供するフレームワークを実装した．このフレームワークを実装するにあたり，その設計方針を以下のように定めた．

- 変換を必須としない
  Unicode を用いた UCS 方式の問題の多くは変換により発生している．変換により今まであいまいに済ませてきた（済ませられた）問題が顕在化する．入出力とも 1 つの CES しか扱わないアプリケーションでは，できるだけ今までどおりテキスト処理のための内部 CES への変換を行わずに処理を行うことを可能にする．
- 固定的な内部文字集合を仮定しない
  どのような内部文字集合を用いても，将来それがカバーできない新たな文字集合が発生することを避けることはできない．究極の UCS と期待された Unicode 以後も，それを超える文字集合（たとえば GB18030）が登場している．将来の大文字集合への対応の可能性を消さないため，Ruby の多言語化にあたっては，固定的な内部文字集合（およびその CES）を仮定せず，ユーザが新規 CES を定義・追加することを可能にする．
  より広い範囲の文字集合をサポートするために，独自の CES を採用しているアプリケーションがある．たとえば Emacs の多言語対応である Mule は内部的には ISO-2022 に対応する Mule コードと呼ばれる内部 CES を採用している．このようなアプリケーションでは独自内部 CES を C のような低レベル言語ですべて実装しているが，CSI 方式の多言語化で Ruby がこれらの CES に直接対応できれば，このようなアプリケーションの開発効率を高めることができ，Ruby の適用範囲を広げることにもつながる．
- 文字列の正規化を行わない
  ある特定の文字のマルチバイト表現が一意に決まらない CES において文字列の正規化はテキスト処理において重要なトピックであるが，要求される正規化方法はアプリケーションごとに大きく異なるため，プログラミング言語としてはその機能を提供しない．ただし，アルファベットの大文字小文字の同一視だけは用意する．
- Ruby のテキスト処理を実現する
  一般に，CSI 方式では必要なプリミティブ集合の

定義が困難である．これは事前にどのような文字列処理が必要になるか決定することが難しいからである．今回は，対応すべきテキスト処理機能をRuby 言語のテキスト処理機能（具体的には，文字列クラスのメソッドと正規表現マッチ）に限定する．Ruby はテキスト処理言語として全世界で広く使われており[16)]，長期間新たなテキスト処理機能追加の要求もないことから，Ruby の持つテキスト処理機能を組み合わせることで実世界で必要とされるテキスト処理のほとんどをカバーできると考える．

- フレームワークは Ruby に依存しない
  この多言語テキスト処理フレームワークは，Ruby の処理系に依存しない．Ruby 以外のアプリケーションでもこのフレームワークを利用して多言語テキスト処理を実装することを可能にする．

3.2 Ruby の多言語テキスト処理機能の仕様

多言語テキスト処理機能を実現するため，Ruby 言語レベルで以下のような仕様変更を行った．

- 文字列オブジェクト（String クラス）のメソッドを CES 情報を参照して文字単位で動作するようにした．
- 単一の文字の表現をコードポイントを表す整数から，1 文字を含む文字列に変更した．
- 文字列どうしの演算で CES が互換でないとき例外が発生するようにした．2 つの CES は互いに等しいとき，またはいずれかが ASCII であり，他方が ASCII 互換であるとき互換であるとする．
- 文字列オブジェクト（String クラス），正規表現オブジェクト（Regex クラス），ファイル入出力オブジェクト（IO クラス）に CES を明示的に指定する．メソッド（encoding=）を追加した．
- Regex クラスを CES 情報を参照してマッチを行うようにした．
- sprintf メソッドで “%s” 指定子が CES の文字幅情報を参照してフォーマットを行うようにした．

これらの変更により，文字列オブジェクトおよび正規表現オブジェクトごとに対応する CES を指定し，CES に応じたテキスト処理を行うことができるようになった．第 1 版で導入されたこの多言語テキスト処理機能の仕様は第 2 版でも変更されない．

3.3 多言語テキスト処理フレームワーク

多言語テキスト処理フレームワークは，Ruby のテキスト処理を CES に依存しない形で実現するための必要な基本的機能を提供する．フレームワークのテキスト処理関数は，プリミティブをメンバとして持つ C の構造体（CES 構造体，m17n_encoding）を引数としてとり，内部的にプリミティブを呼び出し，実際の処理を行う．第 2 版のフレームワーク API を**表 1** に示す．CES 構造体については後述する．

本研究の一部として，Ruby に付属する正規表現ライブラリ「鬼車」[17)] にもこのフレームワーク API を用いるように修正を加えた.「鬼車」は Ruby に依存しない独立したライブラリなので，CES 拡張可能な正規表現ルーチンを本フレームワークの一部として一般のアプリケーションから利用可能となる．

このフレームワークを用いて，以下のように Ruby のテキスト処理を再実装した．

- 文字列を実現する String クラスのオブジェクトと正規表現を実現する Regex クラスのオブジェクトに上記の C の構造体を関連づける機能を追加した．
- 複数のオブジェクトを取り扱う処理に，取り扱うそれぞれのオブジェクトに関連づけられた CES がすべて互換であるかどうか判定し，互換でなければ例外を発生するコードを追加した．
- CES 依存の処理をフレームワーク API 関数で置き換えた．

フレームワーク対応後のコード例を**図 1** に示す．この例は Ruby の文字列オブジェクトの長さを求めるメソッドの実装する関数 rb_str_length() を示している．例中の rb_m17n_get_encoding 関数は，オブジェクトに関連づけられた CES 構造体を取得する関数である．この関数は，取得した CES 構造体と m17n API の 1 つ m17n_strlen() を用いて文字列の長さを求めている．

4. 多言語テキスト処理機能の実装

4.1 多言語テキスト処理機能第 1 版の実装

第 1 版の実装の詳細は割愛するが，第 1 版では，CES に対応するための 16 個のプリミティブが抽出された．第 1 版では従来の Ruby が対応していた EUC-JP, Shift_JIS，UTF-8 の各 CES に対応したうえ，新しい CES 対応の試作として iso8859-1，big5 に対応した．iso8859-1 対応は 113 行，big5 対応は 136 行（いずれも空行，コメントを含まない）で実装可能であった．

4.2 多言語テキスト処理機能第 1 版の制限

多言語テキスト処理機能第 1 版には以下の制限があった．

- 対応する CES は ASCII 互換であり，ASCII 文字しか含まないテキストは，いずれの CES においても同一の物理構造を持たなければならない．
- マルチバイト文字の先頭 1 バイトを参照するだけ

表 1 多言語テキスト処理フレームワークの API
Table 1 API for the M17N text processing framework.

CES 構造体管理関数	
関数名	機能
m17n_define_encoding	CES 構造体に名前を付けて登録
m17n_find_encoding	名前から CES 構造体を取得
m17n_encoding_name	CES 構造体から名前を取得
m17n_encoding_to_index	CES 構造体から番号を取得
m17n_index_to_encoding	番号から CES 構造体を取得
テキスト処理関数	
関数名	機能
m17n_asciicompat	この CES の物理表現が ASCII 互換かどうか
m17n_mbminlen	この CES のマルチバイト文字の最小長
m17n_mbmaxlen	この CES のマルチバイト文字の最大長
m17n_encode	マルチバイト文字列長
m17n_nth	n 番目のマルチバイト文字先頭位置の取得
m17n_strlen	マルチバイト文字列長
m17n_codelen	コードポイントに対応する文字のバイト長
m17n_mbclen	ポインタの指すマルチバイト文字の長さ
m17n_mbcput	コードポイントに対応するマルチバイト文字をバッファに
m17n_cwidth	マルチバイト文字出力幅
m17n_swidth	マルチバイト文字列出力幅
m17n_casecmp	マルチバイト文字列の比較（大文字小文字を同一視）
m17n_isprint	コードポイントは印字可能な文字か
m17n_isspace	コードポイントは空白文字か
m17n_ispunct	コードポイントは記号文字か
m17n_isupper	コードポイントは大文字か
m17n_islower	コードポイントは小文字か
m17n_isalnum	コードポイントは英数字か
m17n_isalpha	コードポイントは英文字か
m17n_iswchar	コードポイントは空白文字か
m17n_isdigit	コードポイントは数字か
m17n_isxdigit	コードポイントは 16 進数字か
m17n_iscntrl	コードポイントは制御文字か
m17n_toupper	対応する大文字のコードポイント
m17n_tolower	対応する小文字のコードポイント
m17n_codepoint	ポインタの指す文字のコードポイントの取得
m17n_ladjust	ポインタ位置に最も近い左側の文字境界位置の取得

```
VALUE
rb_str_length(str)
    VALUE str;
{
    /* オブジェクトから CES 構造体の取り出し */
    m17n_encoding *enc = rb_m17n_get_encoding(str);
    int len;

    /* m17n_strlen API を用いて文字列の長さを得る */
    len = m17n_strlen(enc, RSTRING(str)->ptr,
                      RSTRING(str)->ptr+RSTRING(str)->len);
    /* 整数を Ruby オブジェクトに変換する */
    return INT2NUM(len);
}
```

図 1 フレームワーク対応例
Fig. 1 An example of a framework applied method.

でマルチバイト文字の占める長さが分からなければならない．
- マルチバイト文字の解釈に状態遷移を必要としてはならない．

第1の制限により，EBCDIC のような ASCII とはまったく異なった文字配列を持つ CES に対応できない．また，UTF-16 や UTF-32 は ASCII の範囲でコードポイントは同一だが，物理表現が異なる．UTF-16 や UTF-32 への対応には，文字の構成単位が 1 バイトではないので第2の制限も問題となる．この制限はまた，2 バイト目を参照することではじめてマルチバイト文字全体の長さが決定できる GB18030 の対応にも障害となる．第3の制限により ISO-2022 のような状態遷移を持つ CES に対応できない．

現在ではあまり使われない EBCDIC や，主に通信用として用いられ，内部処理にはあまり用いられない ISO-2022 はともかく，Unicode の表現形式として重要な UTF-16，UTF-32 や，中国国家規格として 2001 年以降中国国内での使用が義務づけられている GB18030 に対応できないのは望ましくない．

4.3 多言語テキスト処理機能第 2 版の開発目標

第2版の開発にあたっては，第1版の制限を緩和すべく以下のことを開発目標とした．
- UTF-16，UTF-32 のような文字を構成する基本単位が 1 バイトよりも大きな CES に対応可能であること
- GB18030 のような先頭 1 バイトだけではマルチバイト文字の長さが分からず，2 バイト目以降を必要とする CES に対応可能であること
- EBCDIC，UTF-16，UTF-32 のように物理表現が ASCII 互換でない CES に対応可能であること

第1版の制限のうち，状態遷移のある CES には対応しないこととする．これは主に性能上の理由からである．状態遷移のある CES を状態遷移のない CES に変換することは比較的容易であることから，この制限が問題になることは少ないと判断した．

4.4 多言語テキスト処理機能第 2 版の実装

第2版の開発目標を達成するため，第1版に対して以下の点を改善した．
- UTF-16，UTF-32 のような文字の単位となる長さが 1 バイトよりも大きい CES をサポートするため，マルチバイト文字の最小長の情報を与えることにした．
- 第1版ではマルチバイト文字列の 1 バイト目からマルチバイト文字の長さを求めるプリミティブがあったが，文字を指すポインタを与えて長さを求めるプリミティブで置き換えた．
- 第1版では ASCII 互換の CES のみを対象にしていたため，改行文字，空白文字などをそのまま指定していた．第2版では ASCII と互換性のない CES も取り扱えるようにするため，ASCII 文字から CES へのマッピングを行う機能を新たに追加して，処理を行うようにした．
- 正規表現ルーチンを従来の Ruby 1.8 が採用していた GNU Emacs 由来のものから，開発版 Ruby 1.9 で採用されている「鬼車」ライブラリをベースにしたものに置き換えた．「鬼車」は独自の CES 対応を持つが，これを本フレームワークによって置き換えた．
- 第1版ではプリミティブであった nth（n 番目のマルチバイト文字先頭位置を取得する機能）および strlen（マルチバイト文字列長を求める機能）を他のプリミティブを用いてフレームワーク内で実装した．これにより定義すべきプリミティブ数が減少した．マルチバイト文字の最小長の情報を用いて最適化を行ったため性能のペナルティはない．
- 第1版ではプリミティブであった swidth（マルチバイト文字列出力幅を求める機能）を cwidth プリミティブを用いてフレームワーク内で実装した．swidth は使用頻度がさほど高くないので独立したプリミティブを用意する必要性は低い．

第2版における各 CES 対応用の構造体を**図 2** に示す．新 CES への対応はこの構造体を用意し，m17n_define_encoding() 関数を呼び出すことによって行う（**図 3**）．このような定義を含む Ruby の DLL を用意することで動的に Ruby インタプリタに新たな CES 対応を組み込むことができる．

表 2 に第2版の各プリミティブの機能を示す．これらのプリミティブを用意することにより，GB18030，UTF-16，UTF-32 などへの対応が可能になった．サンプルとして UTF-16BE（ビッグエンディアン）および UTF-16LE（リトルエンディアン）への対応を用意した．UTF-16 対応はビッグエンディアン（UTF-16BE）に対するものと，リトルエンディアン（UTF-16LE）に対するものの双方を合計して 152 行（コメントおよび空行を含まない）で実現されている．

これらの改良により，第2版では第1版にあった制限が緩和され，Unicode の CES として重要性が増している UTF-16 および UTF-32 に直接対応することが可能になった．また，中国において重要な国家規格である GB18030 や，ASCII と互換性のない EBCDIC のような CES への対応も可能になった．

```
#define M17N_ASCIICOMPAT 1
typedef struct m17n_encoding {
    int flags;         /* M17N_ASCIICOMPAT or 0 */
    int mbminlen;
    int mbmaxlen;
    /* c: コードポイント, p: 文字列ポインタ, s: 文字列先端, e: 文字列終端, code: ctype code type */
    unsigned int (*encode)(unsigned int c, m17n_encoding* enc);
    int (*codelen)(uint c, m17n_encoding* enc);
    int (*mbclen)(uchar *p, uchar *e, m17n_encoding* enc);
    int (*ctype)(uint c, uint code, m17n_encoding* enc);
    uint (*toupper)(uint c, m17n_encoding* enc);
    uint (*tolower)(uint c, m17n_encoding* enc);
    uint (*codepoint)(uchar *p, uchar *e, m17n_encoding* enc);
    void (*mbcput)(uint c, unsigned char *p, m17n_encoding* enc);
    int (*cwidth)(uchar *p, uchar *e, m17n_encoding* enc);
    uchar *(*ladjust)(uchar* s, uchar* p, m17n_encoding* enc);
} m17n_encoding;

/* code type for ctype primitive */
#define M17N_U      01      /* Upper case */
#define M17N_L      02      /* Lower case */
#define M17N_N      04      /* Numeral (digit) */
#define M17N_S      010     /* Spacing character */
#define M17N_P      020     /* Punctuation */
#define M17N_C      040     /* Control character */
#define M17N_W      0100    /* non alpha-numeral Word character */
#define M17N_X      0200    /* heXadecimal digit */
```

図 2 Ruby の CSI プリミティブ構造体（第 2 版）
Fig. 2 CSI primitives structure for version 2.

5. 性能評価

5.1 正規表現検索性能評価

まず，正規表現の検索性能を測定する．プログラムはファイルを読み込んでキーワードを検索する単純な grep プログラムである．与えるテキストデータは ASCII 英文（284 KB: Ruby の ChangeLog）と EUC-JP の日本語文（約 1 MB：日本聖書協会・口語訳新約聖書）を用いる．検索キーワードとしては英文に対しては「ruby」（ASCII），日本語文に対しては「主」（EUC-JP）を用いた．英文に対する検索は，(1) 大文字小文字を同一視しないものと，(2) 同一視するものの双方を行った．測定誤差を減らすためプログラムの 1 回の実行につき，テキストデータは 5 回繰返して読み込んだ．Ruby インタプリタのコンパイルには gcc 2.95.4 を用い，実行は Intel Pentium III 600 MHz 上の Linux 2.4.26 で行った．性能評価プログラムは 10 回連続で実行し，最も成績の良いものを記録した．測定結果を**表 3** に示す．

この結果から，以下のことが分かる．

- 英文テキストで大文字小文字を同一視しない場合，多言語化による性能の変化はほとんどない．
- 英文テキストで大文字小文字を同一視する場合，多言語非対応版と比較して，第 1 版，第 2 版の順に性能が低下する．特に第 2 版では 20%も低下している．
- 日本文テキストの場合，その差はわずかであり，多言語テキスト処理機能の導入による性能低下が少ない．

```
static m17n_encoding eucjp_encoding = {
    M17N_ASCIICOMPAT,            /* flags */
    1,                           /* mbminlen */
    3,                           /* mbmaxlen */
    0,                           /* encode */
    eucjp_codelen,               /* codelen */
    eucjp_mbclen,                /* mbclen */
    asc_ctype,                   /* ctype */
    asc_toupper,                 /* toupper */
    asc_tolower,                 /* tolower */
    eucjp_codepoint,             /* codepoint */
    eucjp_mbcput,                /* mbcput */
    eucjp_cwidth,                /* cwidth */
    eucjp_ladjust                /* ladjust */
};

...
m17n_define_encoding(``euc-jp'', &eucjp_encoding);
```

図 3 CES の対応
Fig. 3 Defining new CES.

性能低下が小さいことから，これは多言語対応版で増加した関数ポインタ経由のプリミティブ呼び出しの

表 2 プリミティブの機能（第 2 版）
Table 2 Primitive functions for version 2.

名称	動作	目的
asciicompat	この CES の物理表現が ASCII 互換かどうか	ASCII 互換チェック
encode	コードポイントに対応する ASCII コード（なければ 0）	ソースコードリテラルとの比較
mbminlen	この CES のマルチバイト文字の最小長	固定長文字に対する最適化
mbmaxlen	この CES のマルチバイト文字の最大長	確保する文字領域長取得
mbclen	ポインタの指すマルチバイト文字の長さ	次の文字位置の取得など
codelen	コードポイントに対応する文字のバイト長	文字長の取得
ctype	コードポイント文字種判定（アルファベットか，数字かなど）	文字種判別，正規表現文字クラス
toupper	対応する大文字のコードポイント	大文字小文字変換
tolower	対応する小文字のコードポイント	大文字小文字変換
codepoint	ポインタの指す文字のコードポイントの取得	論理表現への変換
mbcput	コードポイントに対応するマルチバイト文字をバッファに	物理表現への変換
cwidth	マルチバイト文字出力幅	フォーマット出力（sprintf）
ladjust	ポインタ位置に最も近い左側の文字境界位置の取得	逆方向検索時に前の文字位置の取得

表 3 正規表現性能測定結果
Table 3 Result for regular expression search.

テキスト	処理時間 [sec]（相対比）		
	非対応	第 1 版	第 2 版
英文 284 KB（1）	0.364(1.00)	0.367(1.01)	0.384(1.05)
英文 284 KB（2）	0.368(1.00)	0.394(1.07)	0.443(1.20)
日本語文 1 MB	0.879(1.00)	0.896(1.02)	0.897(1.02)

表 4 文字列メソッド性能測定結果
Table 4 Result for string methods.

テキスト	処理時間 [sec]（相対比）			
	非対応	jcode.rb	第 1 版	第 2 版
英文 284 KB	0.404(1.00)	—	0.413(1.02)	0.437(1.08)
日本語文 1 MB	0.523(1.00)	4.844(9.26)	0.585(1.12)	0.598(1.14)

コストが，非対応版の処理で行われている文字種テーブル参照のコストと比較して極端に大きくないことが分かる．もともと Ruby の正規表現検索は EUC，SJIS，UTF-8 の 3 種類の CES に対応していたため，本研究による CES 対応の一般化によって導入されたコストがさほど大きくなかったことが原因であろう．

第 2 版は英文テキストの大文字小文字を同一視した正規表現検索の性能低下が比較的大きかった．多言語非対応版では，同一視する文字の占める長さは 1 バイトであると仮定しており，同一視のためにはバイト単位で文字種テーブルを参照するだけで十分である．また第 1 版では ASCII 文字だけを同一視の対象にしている．それに対して，マルチバイト文字も同一視の対象にしている第 2 版では，同一視する文字の長さが仮定できず，毎回 1 文字分の変換結果を配列に書き込む必要があり，このことが性能低下の原因であると考えられる．

結論として，正規表現検索については多言語対応による性能低下はあるものの，実用上問題となる可能性は低いことが分かる．

5.2 文字列クラスメソッド性能評価

次に，文字列クラスのメソッドが操作の単位をバイトから文字に変更した際の性能低下を測定する．正規表現検索の測定と同じテキストデータを用いて，各行に対して以下の処理を行い，実行時間を測定する．

- 行の長さを文字単位で求める．
- 2 文字目から 6 文字目までを切り出す．
- 切り出した文字列を文字単位で逆順にする．

測定実行の条件は正規表現検索の測定と同様である．日本語文テキストにおいては，多言語非対応版は文字の代わりにバイト単位での処理を行うため，多言語対応版とは実行結果が異なることに留意する必要がある．参考のため，String クラスのメソッドを再定義することで多言語非対応版でも多言語対応版と同じ結果を得るためのプログラム（jcode.rb）を用意し，これを用いて実行したものも参考として測定した．測定結果を**表 4** に示す．

この結果から，以下のことが分かる．

- 英文テキストにおいては，同じ結果を得るための文字列処理の性能低下は 10%以下である．

- 日本文テキストにおいては性能が 10%以上低下するが，多言語非対応版で同じ結果を得るための処理時間と比較するとはるかに向上している．
  これは jcode.rb が文字単位の処理を行うために正規表現を用いていることによる．文字単位の処理を行うためには正規表現を用いるよりも，多言語テキスト処理機能を用いた方が性能が高いことを意味している．

この人工的な例題では，処理時間のほとんどが文字列クラスのメソッドで文字単位の処理を行うものだけで占められている．実際のアプリケーションでは，ここまで文字単位の処理が集中することは考えにくいので，文字列クラスの変更による性能低下も許容範囲内であると推測できる．

6. まとめ

対応する文字列処理を Ruby の文字列メソッドと正規表現マッチに限定することで，Ruby 言語における CSI 方式の多言語テキスト処理フレームワークを実装した．これにより今後 EBCDIC，UTF-32，GB18030 をはじめとする各種 CES に比較的容易に対応できる．この多言語化により，従来独自のテキスト処理プログラムを用意するしかなかった TRON などのような独自の文字集合と CES を用いているプラットフォームでも Ruby の利用が可能になり，テキスト処理の可能性が広がる．この多言語テキスト処理のためのプリミティブは Ruby に依存しないため，ライブラリとして他のプログラムにも応用できる．

このテキスト処理機能の多言語化の性能評価を行ったところ，正規表現検索の性能においても文字列メソッドの性能においても，性能的ペナルティが小さいことが観測された．また，さらに本研究による多言語テキスト処理が，UCS 方式の多言語テキスト処理と比較して性能的競争力を持つことを確認した．

謝辞 多言語テキスト処理機能第 1 版の開発は，情報処理振興財団（IPA：現在の名称は独立行政法人情報処理推進機構）の 2000 年度第 1 回未踏ソフトウェア創造事業の支援により行われた．

参考文献

1) 日本規格協会：7 ビット及び 8 ビットの 2 バイト情報交換用符号化漢字集合，JIS X 0208-1997.
2) International Organization for Standards: Universal Multiple-Octet Coded Character Set (UCS), ISO 10646-2003.
3) 日本規格協会：国際符号化文字集合（UCS），JIS X 0221-2001.
4) Murai, J., et al.: Japanese Character Encoding for Internet Messages, RFC 1468 (June 1993).
5) Yergeau, F.: A transformation format of ISO 10646, RFC 3629 (Nov. 2003).
6) まつもとゆきひろ，石塚圭樹：オブジェクト指向スクリプト言語 Ruby，アスキー出版局 (1999).
7) 樋浦秀樹：国際化と文字コード，インターネット時代の文字コード，bit 2001 年 4 月号別冊，pp.172–183, 共立出版 (2001).
8) Unicode Consortium: The Unicode Standard, Version 4.0, Addison-Wesley (2003).
9) Wall, L., et al.: *Programming Perl*, 3rd edition, O'Reilly & Associates (2000).
10) http://www.python.org/
11) 久保田淳市，樺澤 哲，櫛木好明：多国語環境における日本語処理の実現，情報処理学会研究報告 — 計算機アーキテクチャ，Vol.1988, No.014 (1988).
12) Lunde, K.: *CJKV Information Processing*, pp.595–596, O'Reilly & Associates (1999).
13) 谷田貝常夫，谷本玲大：ネットワーク・コラボレーションによる大規模フォントコンテンツの移植と応用情報基盤整備の試み，情報処理学会研究報告 — 人文科学とコンピュータ，Vol.1999, No.085 (1999).
14) 谷本玲大：JIS 以外の文字コード — 今昔文字鏡，インターネット時代の文字コード，bit 2001 年 4 月号別冊，pp.103–119, 共立出版 (2001).
15) 松本行弘：オブジェクト指向スクリプト言語 Ruby 次期バージョンの開発，2000 年度未踏ソフトウェア創造事業. http://www.ipa.go.jp/NBP/12nendo/12mito/mdata/12-10h/12-10h.pdf
16) Thomas. D., et al.: *Programming Ruby*, 2nd edition, Pragmatic Bookshelf (2005).
17) http://www.geocities.jp/kosako3/oniguruma/

(平成 16 年 9 月 2 日受付)
(平成 17 年 9 月 2 日採録)

松本 行弘（正会員）

1990 年筑波大学第三学群情報学類卒業．同年（株）日本タイムシェア入社．1994 年トヨタケーラム（株）入社．1997 年（株）ネットワーク応用通信研究所入社．オープンソースソフトウェアの開発に従事．プログラミング言語の設計と実装に興味を持つ．2003 年より島根大学大学院総合理工学研究科博士後期課程．ACM 会員．

縄手　雅彦（正会員）

1981年広島大学工学部第二類（電気系）卒業．1987年広島大学大学院工学研究科博士課程後期材料工学専攻修了．同年4月名古屋大学工学部電気学科助手．1990年5月広島大学工学部第二類（電気系）助手．1996年12月島根大学地域共同研究センター助教授．1999年4月同大学総合理工学部助教授．現在に至る．博士（工学）．1987年より光磁気記録材料，磁気記録ヘッド材料等の磁性材料の研究に従事．2001年より福祉情報工学分野の研究も開始．最近では肢体不自由者のためのコンピュータ操作支援ツールの開発に従事．日本応用磁気学会，日本物理学会，応用物理学会，日本金属学会，電子情報通信学会，ヒューマンインタフェース学会各会員．