情報処理学会論文誌　プログラミング　Vol. 2　No. 2　27–36 (Mar. 2009)

Rubyにおける実用的な多言語処理の実装

松　本　行　弘[†1,†2]

多くのスクリプト言語において多言語テキスト処理はUnicodeを固定的な内部文字コードとして採用しているが，その場合，Unicode以外の文字集合で表現されたテキストを処理するためには文字集合間の変換が必要になり，文字集合間の互換性や文字集合における歴史的な事情などによりさまざまな問題を引き起こす可能性がある．そこで筆者が開発しているスクリプト言語Rubyに対して，固定的な内部文字集合を持たない文字集合独立方式を採用し，文字集合間の変換をできるだけ行わないテキスト処理機能を実装した．本論文で述べるRubyの多言語テキスト処理機能は，Unicodeを固定的な内部文字集合とする他スクリプト言語（PerlおよびPython）と比べて，テキスト処理におけるプログラムの簡潔さおよび性能において劣らない実用的なものであることを示す．本論文で述べる多言語テキスト処理機能はRubyバージョン1.9として公開されている．

Implementation of Practical Multilingual Text Manipulation for Ruby

Yukihiro Matsumoto[†1,†2]

Many scripring languages of present days use Unicode as their universal internal character set to manipulate multilingual text processing. But due to character set compatibility and other historical issues, text conversion to/from the universal character set may cause various problems. We designed and implemented character set independent multilingual processing, which avoids character set conversion as much as possible. We show that multilingual text processing in Ruby is practical in both productivity and performance, comparing other scripting languages, e.g. Perl and Python. The work described in this paper is publicly available in Ruby version 1.9.

†1 株式会社ネットワーク応用通信研究所
Network Applied Communication Laboratory Ltd.
†2 島根大学
Shimane University

1. はじめに

旧来，欧米で開発されたソフトウェアは英数文字および若干の記号から構成されるUS-ASCII文字集合に基づいたテキストデータを処理対象とし，欧州の一部や日本などのアジア諸国のようにUS-ASCII以外の文字を含むテキスト処理を必要とする地域では，ISO-8859-1（欧州諸国）やJIS X0208（日本）など，当該地域で用いられる文字集合とその符号化方式（Character Encoding Scheme，以下CES）に対応するようソフトウェアの改修を行うことが一般的であった．しかし，社会のグローバル化にともない，個別のソフトウェアに対して地域ごとに改修を行うことは次第に現実的ではなくなってきている．このように複数の地域で用いられる複数の文字集合をソフトウェアの改修なしに対応することを「多言語処理」と呼ぶ．

多言語処理を実現する方法は大きく分けると以下の2つの方式がある[1)]．

- 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，以下UCS）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に由来する超えがたい制約を導入することにもなる．UCSとしてはISO 10646[2),3)]で定義されるもの（以下Unicode[5)]）が用いられることが多く，UCSの物理表現である内部CESとしてはUTF-8あるいはUTF-16が採用されることが多いが，まれにUTF-32が用いられることもある．

UCS方式では，すべてのテキストデータに対して内部CESへの変換を行うため，変換コストが必要となる点と，変換にともなう情報喪失が発生する危険性が課題としてあげられる．

CES変換では以下のような理由により，情報喪失が発生することがある．

- 変換前のCESの混同により誤った変換が行われる．
  テキストデータにはそのテキストがどのCESで符号化されているかを示す情報が付加されていないことがしばしばあり，その場合にはなんらかの推測を行うしかないが，そ

の推測が間違っていると，致命的な情報喪失が発生する．

- 対応する文字の不存在
  変換前後の文字集合が異なる場合に変換対象となる文字が変換先に存在することは保証されず，そのような場合には変換不能である．たいていは存在しない文字を「〓」や「？」のような置換文字で表現することになる．
- 1 対多の文字対応
  歴史的な事情その他から文字集合間での文字の対応関係が 1 対 1 ではない場合がある．その場合には文脈情報なしに「正しい」変換を行うことは不可能である．
- 正規化による情報喪失
  文字集合によっては，1 つの文字の表現形式が複数ある場合があり，変換にあたって正規化が行われることがある．その場合，逆変換において元の情報が復元できない．

複数の CES および文字集合が共存してきたわが国ではこの情報喪失問題は「文字化け」として広く知られているが，グローバル化にともない海外でもこの種の問題に遭遇する機会が増えており，「Mojibake」という呼称が定着しつつある．

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，多言語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プリミティブ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．また，特定の CES に依存したチューニングが容易な UCS 方式に比べて実行性能的に不利である．一方で，CSI 方式では CES 変換をともなわないため，変換にともなう情報喪失が存在しない．

CSI 方式の実装は困難であるとの指摘が一般的[1] であったが，筆者らは Ruby[6] に対して CSI 方式のテキスト処理機能の実装に成功している[7]．本研究はその実装をベースに機能・性能ともに実用レベルまで発展させたものである．前論文では文字列および正規表現の多言語化までしか実装されていなかったが，本研究では，

- 入出力
- ファイル名など外部との界面
- コマンドラインオプション
- CES 変換

などもカバーしている．また，性能についても改善を行い，他言語と比肩するものとなっている．

2. 多言語処理アプリケーションのモデル

旧来のアプリケーションの固定的に単一の文字集合と CES だけをサポートするモデルはいわば「単言語（単文字集合）モデル」である．世界に 1 つだけしか文字集合が存在しなければ，この最もシンプルなモデルで十分である．Unicode は世界中の文字集合すべてのスーパセットとなるような文字集合を定義しようという野心的な試みであり，世界中のテキストデータがこれに統一されれば「単言語モデル」で十分である．

確かに Unicode はバージョンが進むにつれ包含する文字数も増え，現時点でかなりすぐれた文字集合を提供しているが，Unicode 自身が複数の CES を提供するうえ，既存の CES を用いた大量のテキストデータの存在はいかんともしがたい．近い将来において Unicode によるテキストデータ統一は期待できない．

そこで，必然的に 1 つのアプリケーションで複数の CES を扱う多言語処理への要求が発生する．多言語処理を行うアプリケーションのモデルとして以下のようなものが考えられる．

- 単エンコーディングモデル
- UCS モデル
- 多エンコーディングモデル

2.1 単エンコーディングモデル

アプリケーションが対応すべき文字集合および CES はユーザごとに異なるが，1 つのアプリケーションが同時に複数の CES に対応する必要性はさほど高くない．たとえば C では locale という仕組みが提供され，ユーザはプロセス単位で使用する CES を 1 つだけ選択する．つまり，C は 1 つのアプリケーションは同時に 1 つの CES だけを扱うことを支援している．このような状況をふまえて，ユーザごとに CES を CSI 方式で切り替えることはできるが，同時に処理する CES を 1 つだけ選択するアプリケーションモデルを「単エンコーディングモデル」と呼ぶことにする．

単エンコーディングモデルのメリットは CES 間の変換が行われないため，変換コストが不要である点と，変換にともなう情報喪失が存在しないことである．

また，同じ CSI 方式を採用したものでも，後述する「多エンコーディングモデル」と比較して，アプリケーション中での CES の混在がないため，アプリケーション開発者の負担が少ないこともメリットである．

一方，単エンコーディングモデルのデメリットとしては，本質的に CSI 方式を要求するので，CSI 方式の実装の困難さがあげられる．特に ISO-2022-JP のような状態のある（stateful）CES を効率良く処理することは困難で，処理可能な CES に対するある種の制限を必要とするケースが多い．筆者らによる CSI 方式の実装も stateful CES は扱うことができない．

2.2　UCS モデル

UCS 方式を採用し，データ入力時に外部 CES から内部 CES（多くの場合，UTF-8 または UTF-16）に変換し，出力時に外部 CES に戻すアプリケーションモデルを「UCS モデル」と呼ぶことにする．

UCS モデルのメリットは，モデルの構造がシンプルであり，アプリケーション実行中に複数 CES が混在しないので，開発者の負担が少ないことがあげられる．また，内部 CES が固定なので，内部 CES に依存した手法でテキスト処理をチューニングできる余地があり，性能的にも有利である．

一方，デメリットとしては，入出力にともない CES 変換が発生するので，変換コストと，情報喪失の危険性がある．

2.3　多エンコーディングモデル

CSI 方式を採用し，文字列 1 つ 1 つが自分の CES 情報を保持し，必要に応じて明示的に変換を行うことで，1 つのアプリケーションが同時に複数の CES を処理することができるモデルを「多エンコーディングモデル」と呼ぶことにする．

多エンコーディングモデルの最大のメリットは，表現力と自由度の高さである．多エンコーディングモデルは他のアプリケーションモデルのスーパセットであるため，このモデルを提供できる枠組みは他のモデルも提供できる．多エンコーディングモデルを実現できる枠組みで，ただ 1 つの外部 CES を用いるならば，それは単エンコーディングモデルであり，読み込み時に外部 CES を 1 つの CES に変換して処理を行うならば，それは UCS モデルである．多エンコーディングモデルは CSI 方式の自然な適用なので，CSI 方式を実現できているならば，このモデルを採用することによる困難さの増加は発生しない．

多エンコーディング方式のデメリットとしては，1 つのアプリケーション中に複数の CES で表現された文字列が混在することにより，テキストデータの CES をトラッキングする必要があり，アプリケーションの複雑化と開発者の負担が高まることである．また，性能的に不利な点も CSI 方式の課題としてあげられる．

3.　Ruby における多言語処理

筆者らは，CSI 方式を採用し，2 章で述べた 3 種類のアプリケーションモデルのいずれにも対応できるような多言語処理機能を Ruby に対して実装した．本章では Ruby の多言語処理機能の概要について述べる．

3.1　文　字　列

多言語処理機能の中心は文字列（`String`）オブジェクトである．Ruby バージョン 1.8 以前では Ruby の文字列はあくまでもバイト列であり，文字単位の操作はマルチバイトを解釈する正規表現を経由することを求められたが，本研究の成果により，Ruby 1.9 では文字列オブジェクトに対する処理はすべて文字単位で動作するようになっている．

文字列オブジェクトに対する処理が文字単位に動作するとは

- 文字列の長さ
- 文字列に対するオフセット

の指定の単位がバイト単位ではなく，文字単位であるということである．Ruby 1.8 以前はこの指定がバイト単位であったので，この点は大きな非互換性である．

文字列オブジェクトは物理表現（バイト列）と，そのバイト列を解釈するための CES 情報とを持つ．文字列に対する操作は CES 情報が示すプリミティブを経由して行われる．これはバイト列という物理表現を CES という「レンズ」を経由して眺めることにたとえられる．文字列オブジェクトが持つ CES 情報は `encoding` メソッドにより取り出すことができる．文字列が持つ CES 情報は，`force_encoding` メソッドで強制的に書き換えることができる．その際，物理表現と CES の整合性は保証されない．CES との整合性をチェックする場合には明示的に `valid_encoding?` メソッドを呼ぶ．`valid_encoding?` メソッドは物理表現が CES として正当な符号化表現であるとき，真を返す．

複数の文字列オブジェクトをともなう操作，たとえば文字列の結合において，それぞれの文字列の CES 属性が異なる場合，以下の条件が満たす文字列は互換性があると見なされる．

- 双方の CES の物理表現が US-ASCII 上位互換であること
- 双方の文字列を構成する文字が US-ASCII の範囲内であること

文字列オブジェクトの CES 属性が異なる場合，UTF-16 のように物理表現が US-ASCII とは異なる CES 属性を持つ文字列や，マルチバイト文字を含む文字列は，互換性があるとは見なされない．文字列が互換性を持たない場合には例外が発生する．

CES 間の変換は `encode` メソッドを用いる．`encode` メソッドは CES 名を示す文字列

または Encoding オブジェクトを引数にとる．レシーバとなる文字列オブジェクトの CES が変換元となる．文字列オブジェクトをバイト列として読み込んだなどの理由で変換元の CES を明示的に指定したい場合は第 2 引数に CES 名または Encoding オブジェクトを渡す．UCS 方式を採用した言語では外部 CES から UCS への変換に encode メソッド，UCS から外部 CES への変換に decode メソッドを用いるものが多いが，Ruby では特定の UCS を特別扱いすることなく，CES 変換はすべて encode メソッドで行い，いちいち区別する必要はない．

3.2 CES 変換

先に述べたように CES 変換は文字列オブジェクトの encode メソッドを経由して行う．encode メソッドを実現している変換エンジンは単なる CES 変換以外にも，改行の正規化，エラー処理方法の指定，XML エスケープなどの機能も持つ．それらの機能は encode メソッドへのキーワード引数で指定する（**表 1**）．

3.3 文　　字

Ruby の多言語処理機能において，文字に相当するものは 1 文字を含む文字列で代用される．これは Unicode におけるサロゲートペアや合成文字のように，複数の「文字」から構成される論理上の「文字」に自然に対応するためである．さらに大文字小文字変換など文字種変換においては，ドイツ語における「ß」と「ss」のように 1 文字を変換した結果が複数文字に対応する場合もあり，これも 1 文字の文字列を文字として取り扱った方が都合が良い．

3.4 Encoding

各 CES を表現するオブジェクトとして Encoding オブジェクトを導入した．Encoding オブジェクト自身にはほとんど機能がない．あるのは名前や文字列表現を得るメソッドとダミーエンコーディングであるかどうかをチェックする dummy? メソッドだけである．

ダミーエンコーディングとは stateful などの理由で直接文字単位での処理を行うことができない CES に対して割り当てられる Encoding オブジェクトである．ダミーエンコーディングに属する文字列オブジェクトはバイト列として振る舞う．これはそのような CES を持つ文字列が変換結果として登場することがあるためである．

本論文執筆時点で Ruby が対応する CES は 81 種類である（別名含む）．なお，ここには 3 種類のダミーエンコーディングを含む．

表 1　encode のキーワード引数
Table 1　Keyword argument for encode.

キーワード	値	意味
invalid:	nil	不正なバイト列でエラー（default）
invalid:	:replace	不正なバイト列は置換
undef:	nil	存在しない文字でエラー（default）
undef:	:replace	存在しない文字は置換
replace:	文字列	置換文字列
universal_newline:	true	CRLF と CR を LF に変換
crlf_newline:	true	LF を CRLF に変換
cr_newline:	true	LF を CR に変換
xml:	:text	XML CharData として置換
xml:	:attr	XML AttrValue として置換

3.5 正 規 表 現

正規表現は多言語処理機能の根幹である．Ruby では「鬼車」[8] と呼ばれる CSI 方式を採用した正規表現ルーチンを採用している．鬼車と Ruby の多言語処理機能は，CES 対応のプリミティブを共有している．

3.6 リ テ ラ ル

プログラム中に登場する文字列および正規表現リテラルにも CES 情報が必要である．CES 情報がなければ正しくプログラムを字句解析することができない．

Ruby ではマジックコメントと呼ばれる形式でプログラムの CES を指定する．すなわち，プログラムの先頭 2 行以内に

```
# -*- coding: euc-jp -*-
```

という形式のコメントが登場した場合，そのコメントがプログラムが記述されている CES を示す．また，ファイルの先頭が UTF-8 における BOM（Byte order mark–「0xef 0xbb 0xbf」の 3 バイト）で始まっていた場合，そのプログラムは UTF-8 で記述されていると見なす．そのファイルに登場するリテラルには指定した CES 情報が付加される．

3.7 入　出　力

Ruby においてファイルなどとの入出力は IO クラスおよびそのサブクラス（File, Socket など）がつかさどる．各 IO オブジェクトは external_encoding と internal_encoding の 2 つの CES 関連の属性を持つ．external_encoding は外部 CES を示す属性であり，internal_encoding は内部 CES を示す属性である．

外部 CES とは入力あるいは出力するデータの CES であり，open 時に指定するが，コマンドラインオプションによりアプリケーションごとのデフォルト値を指定することもできる．内部 CES とはプログラム内部に読み込んでくるデータ（文字列オブジェクト）の CES であり，外部 CES と同様 open 時に指定する．

```
# CES 情報を明示しない
open(path, "r")
# external_encoding だけを明示的に指定
open(path, "r:euc-jp")
# external_encoding と internal_encoding 両方を指定
open(path, "r:euc-jp:utf-8")
# CES 情報をキーワード引数で指定
open(path, "r", encoding: "euc-jp:utf-8")
# 内部で呼ばれる open のためにキーワード引数を指定
File.read(path, encoding: "euc-jp")
```

図 1　open メソッドによる CES 指定
Fig. 1　CES specification for open method.

表 2　open のキーワード引数
Table 2　Keyword argument for open.

キーワード	意味	例
mode:	モード	`mode: "r:euc-jp:utf-8"`
perm:	作成時パーミッション	`perm: 0622`
encoding:	CES 属性	`encoding: "euc-jp"`
textmode:	テキストモード	`textmode: true`
binmode:	バイナリモード	`binmode: true`

IO オブジェクトの CES 属性は open メソッドの引数で指定する（**図 1**）．File.read クラスメソッドのように内部的に open を呼ぶものはキーワード引数を受け付けるようになっており，CES などを明示的に指定できる（**表 2**）．

IO オブジェクトに明示的に CES 属性が指定されなかった場合，デフォルトの CES が用いられる．デフォルトの外部 CES は Encoding.default_external で参照される．コマンドラインオプションで明示的に指定されない場合，default_external の CES は locale（Windows ではコードページ）情報に基づいて設定される．

デフォルトの内部 CES は Encoding.default_internal で参照される．Encoding.default_internal として設定された CES は UCS モデルの内部 CES として動作する．Ruby インタプリタのコマンドラインオプション -E または -U で指定されない限り Encoding.default_internal の値は nil である．

IO オブジェクトに対して外部 CES 属性だけが指定されているとき，入力時に読み込んできた文字列に外部 CES 情報を付与する．出力時にはデータの物理表現をそのまま出力する．内部 CES が指定され外部 CES と異なる場合には，入出力時に暗黙に CES 変換を行う（入力時：外部 CES→内部 CES，出力時：内部 CES→外部 CES）．暗黙の CES 変換は情報喪失の危険性があるが，CSI 方式を採用した Ruby においても複数 CES を同時に扱うアプリケーションを現実的なコストで実装するためには UCS モデルを忌避することはできず，その際の開発者の手間を削減することは重要であると考えた．

3.7.1　入　　力

IO オブジェクトから読み込みを行うとき，そのデータの CES は external_encoding であると見なして読み込みを行う．external_encoding が明示的に指定されていない場合，Encoding.default_external が用いられる．

internal_encoding が明示的に指定されていない場合には，Encoding.default_internal を参照する．いずれかが設定されている場合には外部 CES から内部 CES への変換が暗黙に行われる．いずれも設定されていない場合には，外部 CES を読み込んだ文字列に付与する．

3.7.2　出　　力

IO オブジェクトに対して書き込みを行う場合，明示的に external_encoding が指定されていない場合，出力する文字列の物理表現をそのまま出力する．external_encoding が指定されている場合，文字列を外部 CES に変換する．出力の場合には default_external および default_internal は参照しない．

3.8　外界との界面

入出力以外で外界から与えられる文字列の CES は default_external が尊重される．ただし，ファイルシステムの用いる CES が固定されているプラットフォーム（具体的には Windows と Mac OS X）では，そちらが優先される．そのような外部情報としては，以下のようなものがあげられる．

- 環境変数（ENV）
- コマンドライン引数（ARGV）
- パス名（File）
- ディレクトリエントリ（Dir）

UNIX のような環境では，ファイルシステムがパスの CES を規定しておらず，C での文字列終端の「'\0'」や，パスにおけるディレクトリ区切り記号「/」など特定の文字を含まなければどのような CES であっても許される．実際，1 つのファイルシステムに UTF-8, Shift_JIS, EUC-JP のファイル名が混在することも決して珍しくない．このような環境では，外部から得たパスの CES を正しく得ることは困難で，CES の指定を間違った場合，CES 変換により深刻な情報喪失の危険性がある．そのような情報喪失はすなわち「存在するはずのファイルを見つけられない」という問題に直結する．

表 3　-E コマンドラインオプション
Table 3　Command line option -E.

オプション	default_external	default_internal
-E *e*	e	指定なし
-E *e*:*i*	*e*	*i*
-E :*i*	指定なし	*i*
-U	locale	UTF-8

Ruby の多言語処理機能では，内部 CES を明示的に指定しなければ，パス文字列に CES を付加するものの，CES 変換は行わない．それにより，パスを失う事態は避けられる．

内部 CES を明示的に指定した場合には，仕様上 CES 変換は避けられない．しかし，このような外部からの入力で CES 変換に失敗した場合，壊れたテキストを返すのではなく，変換元のパス（の物理表現）をバイト列（CES として ASCII-8BIT を指定したもの）を与えることで，プログラム内部で「文字の列」として扱うことはできなくても，情報喪失によりパスの情報を失うという最悪の事態を避けようとする．

3.9　コマンドラインオプション

Ruby のコマンドラインオプションのうち，多言語処理に関連があるものは `-E` と `-U` である．`-E` はデフォルトの CES を設定する（**表 3**）．

表 3 において「locale」とあるものは CES が locale 情報に基づいて選択されることを意味し，「指定なし」とあるものはこのオプションによって内部 CES が設定されないことを意味する．複数の `-E` オプションが指定された場合には，指定内容が矛盾する場合にエラーになる．

すべてのコマンドラインオプションの指定が完了した後，最終的に `Encoding.default_external` が指定されていなかった場合には，locale に基づいて `Encoding.default_external` の外部 CES が設定される．`Encoding.default_internal` が指定されなかった場合には，内部 CES は指定されず，入出力にともなう暗黙の変換が発生しない．

`-U` は外部 CES を locale（Windows ではコードページ）情報に基づいて設定し，内部 CES を UTF-8 に設定する．このオプションは実行するプログラムが，UCS モデルで動作することを指定するものである．

4.　Ruby における多言語処理の実装

CSI 方式を採用した多言語処理の実装の基本については前論文[7] を参照されたい．前論文の内容は，プリミティブの名称や機能などに若干の変化はあるが，基本的な事項については最新バージョンでも同様である．現在の多言語処理機能が要求するプリミティブを**表 4** に示す．

表 4　多言語処理プリミティブ
Table 4　Primitives for M17N processing.

プリミティブ	処理内容
precise_mbc_enc_len	文字の長さ
max_enc_len	文字の最大長
min_enc_len	文字の最小長
is_mbc_newline	改行文字の判定
mbc_to_code	文字からコードポイント取得
code_to_mbclen	codepoint から文字長取得
code_to_mbc	codepoint から文字取得
mbc_case_fold	大文字小文字の正規化
apply_all_case_fold	関数による大文字小文字の正規化
get_case_fold_codes_by_str	大文字小文字を反転させた文字取得
property_name_to_ctype	プロパティ名から文字種表取得
is_code_ctype	文字種判定
get_ctype_code_range	プロパティ範囲取得
left_adjust_char_head	前の文字の先頭取得
is_allowed_reverse_match	左方向マッチ可能か判定

本章では性能が犠牲になりがちな CSI 方式で，速度を達成するためのいくつかの最適化について述べる．

4.1　CODERANGE 最適化

各文字を構成するバイト列の長さが一定ではないマルチバイト文字列の処理では，文字列長の取得や特定文字へのオフセット位置の計算は文字列の先頭からのスキャンが必要で，計算量は $O(n)$ になる．

しかし，実際に処理を行う文字列の多くはその内容の文字列すべてが ASCII の範囲内であり，文字の長さは結局 1 バイトである．そこで，文字列を構成する文字の範囲を文字列オブジェクトに記憶させておくことで，ASCII しか含まない文字列の処理を高速化できることが期待される．

具体的には文字列に**表 5** に示すフラグを設定する．`ENC_CODERANGE_7BIT` が設定されている文字列は，このような文字列ではその内容がすべて ASCII 文字であることが明らかなので，それに応じた最適化を行うことができる．たとえば長さやオフセットの取得が $O(1)$ でできる．特に ASCII 文字列を多く処理するプログラムでこの最適化の効果は高い．

表 5　ENC_CODERANGE フラグ
Table 5　ENC_CODERANGE flags.

フラグ	意味
ENC_CODERANGE_7BIT	ASCII のみ
ENC_CODERANGE_VALID	マルチバイト文字列
ENC_CODERANGE_BROKEN	CES 的に不当な文字列
ENC_CODERANGE_UNKNOWN	未判定

4.2　search_nonascii 最適化

ENC_CODERANGE フラグの設定には文字列のスキャンが必要になる．文字列が長くなればこの判定のためのコストも無視できない．そこで，文字列を構成するバイト列が 7 ビットで表現できる 0〜127 までの範囲に収まっていない場合にはマルチバイト文字を含むという CES 共通の性質を利用して，最適化を図っている．

具体的には文字列をバイト単位にスキャンする代わりにワード単位でスキャンする．アドレスのアラインメント不整合によるバスエラーを回避するためのアラインメントの調整を行った後，ワード単位に 8 ビット目をオンにしたマスクとの論理和をとり，ゼロであった場合にはそのワードはマルチバイト文字を含まないと判定する．マスクとしては 1 ワードが 32 ビットである場合には 0x80808080 を，1 ワードが 64 ビットである場合には 0x8080808080808080 を用いる．メモリからのデータのフェッチ回数が減るうえ，メモリアクセス時のアラインメント調整が不要になるので高速なスキャンが実現できる．

4.3　utf8_lead_bytes 最適化

UTF-8 にはマルチバイト文字の先頭バイトを見るだけでその文字を構成するバイト長が分かるという性質がある．そこで，

- CES が UTF-8
- 内容が CES 的に正当

であることが明らかなときに，この性質を利用して文字列スキャンを削減している．

4.4　Ruby 1.8 との互換性

Ruby 1.8 と Ruby 1.9 では多言語処理において大きく異なっている．具体的には以下の点に注意する必要がある．

- 文字列のオフセットが文字単位
- 文字の表現がコードポイントを示す整数から文字列に変わった
- 文字リテラル「?a」が整数から文字に変わった
- マルチバイトを含むプログラムにはマジックコメントが必要になった

しかし，もともとバイト単位でしかオフセット処理を行っていなかった Ruby 1.8 以前では文字列のオフセットをとることはまれであり，かつ，文字列処理の基本となる正規表現による処理は，（いくつか新しい機能が増えているが）基本的に上位互換であるので，非互換性の深刻度はあまり高くない．ある経験によれば，1.8 対応のプログラムの 7 割はそのまま動作し，残り 3 割については，あまり大規模でない修正が必要であったということである[9]．

また，Ruby によって記述された Web 日記システムである tDiary[10] では，Ruby 1.9 に対応させるために変更が必要であった行数が変更前 Ruby プログラム 11,654 行に対して 123 行であった（空行およびコメントも含む）．このうち，多言語処理に関連する変更はわずか 10 行であった．tDiary のような実用的なプログラムの多言語処理に対する対応がわずか 10 行で済むということは多言語処理により導入された非互換度が深刻でないことを示している．

5.　他スクリプト言語における多言語処理

Ruby と類似の領域をカバーするスクリプト言語としては Perl[11] および Python[12] がある．本章ではそれらがどのように多言語処理を実現しているかについて述べる．

5.1　Python

Python は言語として UCS 方式を採用しており，内部 CES として主に UTF-16 を用いる．Python はコンパイル時の設定で UTF-32 を内部 CES として選ぶこともできるが，実際の使用例はほとんど報告されていない．内部 CES として UTF-16 を選んだ場合でも，サロゲートペアなどの特別扱いはされておらず，単純に 2 文字として処理される．

Python では UCS に変換されたテキストは通常の文字列（string）と互換ではあるが独立した unicode オブジェクトに格納される．string はバイト列と解釈される．入出力の対象は基本的にバイト列であり，unicode オブジェクトを得るためには，明示的に decode メソッドによってバイト列から unicode 文字列に変換するか，codecs モジュールが提供する unicode 文字列を入出力するためのラッパを利用する必要がある．unicode 文字列から出力用に外部 CES で符号化されたバイト列を得るためには encode メソッドを用いる．

近い将来リリースが計画されている Python 3.0 では，文字列はすべて unicode 文字列となり，バイト列を表現する bytes オブジェクトが導入される．

5.2　Perl

Perl も UCS 方式を採用しており，内部 CES として UTF-8 を用いる．Perl では文字列には utf8 フラグという属性を保持しており，文字列が内部 CES に変換された「文字の列」

であるか，バイト列であるかを区別する．utf8 フラグがセットされている文字列は CES として UTF-8 を使っている．

ファイル入出力は open 時に明示的に指定しない限り，utf8 フラグのセットされていない文字列を返し，これはバイト列であると見なされる．utf8 フラグがセットされた文字列はマルチバイト文字を正しく解釈し，セットされていない文字列は 1 バイトが 1 文字を構成すると解釈される．ただし，言語としては双方とも同じ文字列であるため，文字列の内容が US-ASCII の範囲内であれば，同じ振舞いをする．utf8 フラグのセットされた文字列を直接出力しようとするとエラーになる．

CES の変換は Encode モジュールを用いて行われる．Encode モジュールが提供する decode メソッドで外部 CES から UTF-8 への変換と utf8 フラグのセットを行い，encode メソッドで UTF-8 文字列から外部 CES への変換が行われる．

6. ベンチマーク

各言語による多言語処理の性能を測定するため，EUC-JP テキストを読み込んで加工するアプリケーションを Ruby（**図 2**），Python（**図 3**），Perl（**図 4**）を用いて記述した．このプログラムはテキストを読み込み，その中で同じひらがなが 2 文字連続している部分をブラケットで囲むという処理を行う．正規表現によるマルチバイト文字列の探索と置換の速度を問うアプリケーションである．

単純なプログラムではあるが，同じ振舞いをするプログラムを比較することで，Ruby での記述が UCS モデルを採用した他の言語での記述と比較して，同程度に簡潔であることがうかがえる．

CSI 方式は UCS 方式のスーパセットであるので，Ruby では UCS モデルを用いることもできる．UCS モデルを採用して Python や Perl とまったく同じように内部的に Unicode に変換してから処理を行う Ruby によるプログラムを**図 5** に示す．これにより UCS モデルを採用した場合の記述力とまったく同じ処理（読み込み，Unicode への変換，処理，Unicode からの変換，出力）を行った場合の性能を測定することができる．図 5 のプログラムと Python や Perl のプログラムを比較するとほとんど記述力に差はないことが分かる．Ruby の多言語処理機能では Perl や Python と同様の UCS モデルのプログラムもほぼ同程度の手間で実装可能であり，CSI 方式の表現力の高さを示している．

これらのプログラムの実行結果を**表 6** に示す．テキストデータとしては EUC-JP の日本語文（約 4 MB–日本聖書協会・口語訳新旧約聖書）を用いた．実行は Intel Core2 Duo T7500 2.20 GHz，メモリ 2 GB の計算機で行った．OS は Linux 2.6.26，Python は 2.5.2，Perl は 5.10.0，Ruby 1.8 は 1.8.7（2008-08-11 patchlevel 72）を用いた．Ruby は ruby 1.9.0（revision 19599）を用いた．性能評価プログラムは 10 回連続で実行し，最も成績の

```
# -*- coding: euc-jp -*-
fp = open("in.txt", "r:euc-jp")
fout = open("out.ruby.txt", "w")
while line = fp.gets
  line.gsub!(/([あ-ん])\1/, '[\1\1]')
  fout.print line
end
```

図 2　テキスト加工プログラム（Ruby）
Fig. 2　Text processing sample (Ruby).

```
# -*- coding: utf-8 -*-
import codecs
import re

fout = codecs.open('out.python.txt', 'w', 'euc_jp')
pat = re.compile(u"([あ-ん])\\1")
for line in codecs.open('in.txt', 'r', 'euc_jp'):
    fout.write(pat.sub("[\\1\\1]", line))
```

図 3　テキスト加工プログラム（Python）
Fig. 3　Text processing sample (Python).

```
use encoding "utf-8";

open(FP, "<:encoding(euc-jp)", "in.txt");
open(FOUT, ">:encoding(euc-jp)", "out.perl.txt");
while (<FP>) {
    s/([あ-ん])\1/[\1\1]/g;
    print FOUT;
}
```

図 4　テキスト加工プログラム（Perl）
Fig. 4　Text processing sample (Perl).

```
# -*- coding: utf-8 -*-
fp = open("in.txt", "r:euc-jp:utf-8")
fout = open("out.ruby.txt", "w:euc-jp")
while line = fp.gets
  line.gsub!(/([あ-ん])\1/, '[\1\1]')
  fout.print line
end
```

図 5　テキスト加工プログラム（Ruby UCS 版）
Fig. 5　Text processing sample (Ruby UCS).

表 6　テキスト加工プログラム実行時間
Table 6　Execution time of text processing programs.

言語	実行時間（秒）
Ruby	0.541
Ruby 1.8	0.590
Python	1.240
Perl	1.282
Ruby（UCS）	1.399

```
fp = open("in.txt", "r:euc-jp")
fout = open("conv.ruby.txt", "w:shift_jis")
while line = fp.gets
  fout.print line
end
```

図 6　CES 変換プログラム（Ruby）
Fig. 6　CES convsersion sample (Ruby).

良いものを記録した

この測定結果から，単エンコーディングモデルを採用した場合，Ruby によるテキスト処理が Python や Perl のものと比較して 2 倍以上高速であることが分かる．これは単エンコーディングモデルを採用したプログラムが，コストのかかる内部 CES と外部 CES の間の相互変換を回避できることが理由であると考えられる．単エンコーディングモデルのプログラムは Python と比較して 129%，Perl と比較して 137%，UCS モデルを採用した Ruby と比較して 159%高速である．Unicode 以外の CES を用いたレガシーなテキストデータの処理において，単エンコーディングモデルを用いることができる限り，Ruby が採用した CSI 方式に相当の性能優位性があることが分かる．また，Ruby 1.8 との比較によって，少なくとも今回測定した範囲内では拡張性のある CSI 方式の導入によって性能の低下が発生していないことが分かる．

複数の外部 CES を 1 つのプログラムで扱いたいなどの理由で UCS モデルを採用せざるをえない場合がある．そのような場合を想定して，同じ UCS モデルのプログラムで比較するとき，Ruby は Python と比較して 13%，Perl と比較して 10%程度遅い．現時点では UCS モデルを用いた Ruby プログラムの性能は Python および Perl と比較して優れてはいないが，その性能差はさほど大きくない．

CES 変換の効率を測定するため，上記で用いたテキストデータを EUC-JP から Shift_JIS に変換する測定も行った（Ruby **図 6**，Python **図 7**, Perl **図 8**）．測定結果を**表 7** に示す．プログラミング言語以外に独立した CES 変換プログラムである nkf，lv，iconv も交えて測

```
fout = open('conv.python.txt', 'w')
for line in open('in.txt', 'r'):
    fout.write(line.decode('euc-jp').encode('shift_jis'))
```

図 7　CES 変換プログラム（Python）
Fig. 7　CES convsersion sample (Python).

```
open(FP, "<:encoding(euc-jp)", "in.txt");
open(FOUT, ">:encoding(shift_jis)", "conv.perl.txt");

while (<FP>) {
    print FOUT;
}
```

図 8　CES 変換プログラム（Perl）
Fig. 8　CES convsersion sample (Perl).

表 7　CES 変換プログラム実行時間
Table 7　Execution time of CES conversion programs.

言語	変換時間（秒）
Ruby	0.375
Python	0.227
Perl	0.403
nkf	0.539
lv	0.264
iconv	0.112

定した．この測定から Ruby の CES 変換が他処理系と比べて遜色ないことが分かる．

7. 今後の課題

本研究では Ruby に対して CSI 方式を採用した多言語処理機能を実装し，それが Python, Perl などの対象領域が近い他言語と比較しても，表現力，性能ともに実用レベルに到達していることを示した．

本開発による多言語処理機能の性能は，単エンコーディングモデルでアプリケーションを開発できる場合には他言語を凌駕する性能が得られるが，UCS モデルを用いて他言語とまったく同じ処理を行った場合にはやや性能的に見劣りがあり，改善が期待される．

改善の可能性の 1 つとして，プリミティブ `mbc_to_code` と `code_to_mbclen` がつねに連続して呼ばれることが観測されており，`mbc_to_code` がその実装で `code_to_mbclen` を内部的に呼び出していることを考慮するとこれらの呼び出しを融合させることが可能である．`mbc_to_code` は文字列処理において相当頻度で呼び出されており，プリミティブ融合

によるさらなる性能改善が期待される．

プログラムの記述力という観点からは，CSI 方式が物理表現とは独立した「文字の列」という抽象的な概念を扱えることから，Ruby の多言語処理機能は合成文字など Unicode で grapheme cluster [13] と呼ばれる概念を自然に取り扱える可能性がある．これはあくまでもコードポイントをベースにする他言語との差別化につながる機能であり，今後研究の余地があると考えられる．

8. ま と め

Ruby に対して，CSI 方式を採用し多言語処理アプリケーションの各種モデルを実現できる，実用的な多言語処理機能を実装した．この多言語処理機能は API は簡潔であり，性能も十分競争力がある．今回測定したテキスト加工アプリケーションでは，単エンコーディングモデルで，UCS 方式を採用した Python および Perl で実装した同じ振舞いのアプリケーションと比較して，2 倍以上の性能向上が見られた．UCS モデルでは，CSI 方式の利点が活かせず性能は他言語に比べてわずかに劣るが，記述力については十分に競争力がある．UCS モデルでの性能については今後の改善が期待される．

本研究で開発された多言語処理機能は，最新版 Ruby に取り込まれて公開されている．

謝辞　本研究の多言語処理機能をはじめとして，Ruby はその開発コミュニティの数多くのメンバの協力によって実現された．特に田中哲（産業総合研究所），Martin Dürst（青山学院大学），田中隆裕（ソフトバンク・テクノロジー株式会社），小迫清美の各氏に感謝する．

参 考 文 献

1) 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).
2) International Organization for Standards: Universal Multiple-Octet Coded Character Set (UCS), ISO 10646-2003 (2003).
3) 日本規格協会：国際符号化文字集合（UCS），JIS X 0221-2001 (2001).
4) Murai, J., et al.: Japanese Character Encoding for Internet Messages, RFC 1468 (June 1993).
5) Unicode Consortium: The Unicode Standard, Version 5.0, Addison-Wesley (2006).
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) 松本行弘，縄手雅彦：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，情報処理学会論文誌，Vol.46, No.11, pp.2633–2642 (2005).
8) http://www.geocities.jp/kosako3/oniguruma/
9) Thomas, D.: personal communication.
10) http://www.tdiary.org/
11) Wall, L., et al.: *Programming Perl 3rd edition*, O'Reilly & Associates (2000).
12) http://www.python.org/
13) Unicode Consortium: Unicode Text Segmentation, Unicode Standard Annex #29. http://www.unicode.org/reports/tr29/

(平成 20 年 9 月 29 日受付)
(平成 20 年 12 月 24 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．ACM 会員．