Unicodeの最新動向（その2）

木戸彰夫　　白鳥孝明
日本アイ・ビー・エム株式会社

本稿は、Unicodeの文字符号としての特徴、他の漢字圏の符号化文字集合との関連、及びUnicodeにまつわる様々な議論を技術的観点からを解説を行った『Unicodeの最新動向』(参考文献1)の続編であり、Unicodeのアプリケーション分野での利用のされ方、及び利用上の問題点を解説する。本稿は、1章と3章を木戸が、2章を白鳥が執筆した。

Current Status of Unicode (Part 2)

Akio Kido　　Takaaki Shiratori
IBM Japan, Ltd.

This report explains current usage of Unicode in several application domains and discuss about technical issues on those applications. This paper is the second part of the "Current Status of Unicode" (reference document 1), presented IPSJ/Digital Document interest group meeting's 12th meeting.

１．プログラム言語での利用

Unicodeがアプリケーション分野でどのように利用されつつあるかを見て行く上で、まずプログラム言語での利用のされ方を議論するのは、有用なことであろう。仮にプログラム言語がUnicodeを「文字」として取り扱う機能を全くもたなかったとするならば、その言語で記述されるアプリケーションでのUnicodeの利用が、かなりの制限を受けることが容易に想像されるからである。

プログラム言語を、その実装という観点ではなく、言語仕様の観点から見るならば、「文字」及び、「文字符号」に関連する部分は、

- ソースコード表現に許される文字集合
- 文字型のサイズ
- 文字型が表せる文字集合
- 文字ストリング型内での文字データ境界
- 文字型データのリテラル表記
- 文字型データの内部表現

及び、

- 識別子に許される文字集合

となる。衆知のように、従来の多くのプログラム言語の言語仕様は、ソースコード表現に許される文字集合、文字型のサイズ、文字型が表せる文字集合、及び文字型データの内部表現に規定を設けず、一見、コメントおよび文字データとして如何なる文字の使用をも自由に許しているように見受けられるが、その実、１文字1バイト（８ビット）という暗黙の了解事項に縛られていた。その事は、文字型データリテラルの数値表現が２桁のヘキサ表現に限られていることや、文字ストリング型内での文字データの境界が必ずしもデータ型の境界と一致しない為、Ｃ言語のmbrlen() 関数のように、文字ストリングデータ型内での文字データの境界を調べるAPIが用意されていることなどから見て取れる。プログラム言語におけるデータ型の思想から言うならば、本来、データ型はその型が格納し得るデータの集合によって規定されるべきものであるので、任意の１文字を格納できぬ文字型は、本来の文字型ではなく、オクテット型として規定されるべきものではあるのだが、プログラム言語が取り扱うべき文字は、２５６未満の文字数で全て表現できるというラテンスクリプトを想定した暗黙の了解事項によって、それらのデータ型は、CHARACTER型と称されてきた。ところが、Unicodeの登場によって、ラテンスクリプトの世界においても、その暗黙の了解事項が破綻をきたすことになった。

単純に考えるならば、多くのプログラム言語の言語仕様は、文字型のサイズ（ビット長）を規定していないのであるから、２５６以上の文字数をもつ文字集合をサポートしようとする場合においても、プログラム言語が新たなデータ型を導入する必要はない。実装のレベルで文字型のサイズを１６ビット以上として実装すればよいだけである。唯一プログラム言語の言語仕様の変更を必要とするものは、数値表現を用いた文字型データのリテラル表記の桁数に関する変更であるが、これ自体も文字型データの内部表現がプログラムの実行環

境を依存せず一定であること、およびその内部表現をプログラマが知っていることを前提とした、プログラムのポータビリティーを失わさせる好ましくない表現法であるので、過去の遺物として無視してしまっても大きな問題にはなり得ない。しかし実際には、プログラム言語の多くは、例えばC言語のwchar_tやCOBOLのPIC(N)型の導入に見られるように、従来の「文字型」のサイズの制約を取り去るかわりに、本来の意味での文字型を新たに導入することによって、Uniocdeのような２５６以上の文字数をもつ大きな文字集合のサポートの要求に答えようとしている。この選択は、異なったプログラム言語で書かれたモジュールのリンクを容易にすることと、共用体を用いたデータ型のサイズに依存した既存のプログラムのアップワードコンパチビリティーに留意したものであると言える。

これらの異種言語で書かれたモジュールのリンケージ、および共用体の問題を鑑み、筆者がプロジェクトエディターを務めたISO/IEC TR 10176 Second edition（参考文献２）では、Unicode（ISOでは、ISO/IEC 10646-1として規格化されている）への対応の為に以下のようなガイドを追加した。

- ソースコードに許される文字集合としては、少なくともISO/IEC 10646に規定される全ての文字を許すこと。
- 少なくともISO/IEC 10646の全ての文字を要素集合としてもつことができる文字型を用意すること。
- オクテットストリング型を利用して文字を表現する機能を用意してもよいこと。ただしその場合は、オクテットストリング内での文字の境界を判定する機能と、オクテットストリング型と文字ストリング型の間での型変換の機能を用意すること。
- 文字型データの数値表現を用いたリテラル表記では、ヘキサ表現で８桁までの表現を可能にすること。
- 文字名の短縮形（エスケープ文字列に続く、ISO/IEC 10646での文字符号値のヘキサ表現）を用いた、文字データ型内でのデータの内部表現に依存しないリテラル表現を用意すること。
- 利用者定義識別子として、ASCII文字集合外の文字も許すようにすること。ただし記号類は除くものとする。
- ASCII外の文字を用いた識別子のASCII文字をだけを用いた代替表現として、文字名の短縮形を用いた表現を許すようにすること。
- 識別子、予約語、区切り文字としてASCII外の文字集合を許し、かつその集合が、全く、若しくはほとんど同形の印字字形をもつ複数の文字を含む場合、プログラム言語の実装は、その実装が言語のシンタクス上同一と判断する文字のリストを文書化すること。
- ソースレベルのポータビリティーに加えて、実行モジュールレベルでのポータビリティーに対する要求を持つJavaのような言語の場合、文字型データの内部表現としては、Unicodeを推奨すること。

上記のISO/IEC TR 10176のガイドの最後の項目にも関係することでもあるが、JavaやECMA Scriptなどの最近のプログラム言語の仕様においては、文字型の要素集合をUnicode文字として、符号化方法をも含めてUnicodeを直接参照する言語仕様が増えつつある。これらの言語では、外部ファイルが記述されている様々な既存の文字符号とプログラム言語内部で使用される文字型データの内部表現であるUnicodeとの文字符号の相互変換を、プログラム言語のI/O境界で執り行うことにより、プログラム言語内の文字表現を一つの文字符号に絞っている。Unicodeの最新版であるUnicode Version 2.1までを視野に入れる限り、文字型の内部表現をUnicodeに限定すること自体は、上記のISO/IEC TR 10176が提供するガイドにも従ったものであり、特に大きな問題を生じない。しかし、現在改訂作業が進められているUnicode Version 3.0の環境においては、いくつかの問題を引き起こすことが予想される。ISO/IEC JTC1/SC2が現在規格化を進めている、ISO/IEC 10646の基本多言語面以外の面を使用

した文字種の拡張を受ける形で、Unicodeでもそれらの拡張面に定義される文字のサポートを予定している。ISO/IEC 10646の場合、UCS4と呼ばれる32ビット表現を用て、それらの文字を固定長で表現する手法があるのだが、Unicodeは16ビットというデータ長に依存した符号系であるので、それらの文字はISO/IEC 10646では、UTF-16と呼ばれる変形表現で表されることになる（Unicodeではこの表現法をサロゲートと呼んでいる）。つまり、基本多言語面以外の面を使用して符号化された文字は、特定域の符号値をもつ16ビットデータのペアにより表現されることになる。この１６ビットペアによる１文字の表現をもつことにより、Unicodeは16ビット固定長の文字符号から、16ビット単位の可変長の文字符号へと変遷をとげる。この可変長となったUnicodeをJavaやECMA Scriptなどのプログラム言語がどのように取り扱うかについては、まだ結論が出されていない。

２．データベースでの利用

データベースでUnicodeを使う要求が増えてきた背景には、ビジネスのGlobalizationやボーダーレスなネットワーク・アプリケーションの開発が盛んになってきたことが挙げられる。Unicodeを利用する利点として、容易にマルチリンガル・データベースを構築することができることがある。その理由としては、Unicodeを使わないで構築した場合に比べ、以下のような現実の開発・運用を困難にする要因に直面せずに済むことがあげられる。

- 各国語の文字符号をスキーマ定義に関連付ける複雑さ。
- それらを念頭にプログラミングをしなくてはならない複雑さ。
- 多くの文字符号変換を伴う場合の実行時のパフォーマンスの低下。
- 特殊な実装社独自の文字符号体系を使った場合、外部の世界とのデータの共用やポーティングに関する問題。

ここでUnicode利用の必然性とサポートにあたってのハイレベルな構図を述べておきたい。データベースは元来、その稼動するプラットフォームでサポートされるネイティブな文字符号体系をそのまま文字データの格納用にサポートしてきた。今日ではJavaやWindowsオペレーティング・システムなどに代表されるように、プラットフォームにおけるネイティブ・コードがUnicodeになりつつあり、こうしたプラットフォームの動きに呼応する形でUnicodeをサポートしだすのは当然の動きと言える。先に第１章で、プログラミング言語でのUnicodeの利用が述べられているが、データベースにおけるデータベース・アクセス言語ＳＱＬがホスト・プログラミング言語に埋め込まれることでその一部であると考えれば、プログラミング言語へのガイドが基本的にＳＱＬにもあてはまる。その他に、データを扱う上でデータベース言語のもつ大きな特色は、データを永続的に保存できかつ再利用できるようにその仕組みと一体になっていなければならないということが挙げられる。すなわち、１章で述べた他のプログラム言語モジュールとのリンケージ、および共用体の上位互換への考慮がデータベース・システムにとって本質的課題として存在していると言えよう。このため、「データ・アクセス」という果たすべき役割から従来のオクテット型というべきCHARACTER型に加えて、Unicodeのためのいわゆる文字型が新たに必要という構図が必然的に見えてくる。

もう少し内部コードにまで立ち入って、実装面での考慮点に触れておきたい。データベースでUnicodeを実装するには周知のとおり、２通りのサポートの仕方がある。一つは文字単位の処理を前提とした文字型を用意する方法で、処理のし易さから内部エンコーディングとしてUCS2を使うのが一般的となっている。UCS2は現存する文字セットの中で可能な限り世界各国の文字を均一のサイズでサポートしている文字セットでUniodeの本来の特質をもっとも忠実に具現化している。しかし、Unicode V3では文字数に関する更なる要求や言語識別などの要求を満たすため以下のような拡張が

議論されており面０１から面１６までが近い将来使われていく可能性が取り沙汰されている。

- 面02を使ってのCJK統合漢字2万字の追加。
- JCS（符号化文字集合調査研究委員会）が進めているJIS X0208の拡張により追加された第3水準、第4水準の文字のうち、CJK統合文字2万字の中に含まれてない漢字の追加要求。
- 面14を使っての新しい言語タグの導入。
- 面15、面16を使っての外字の割り振り。

このような将来起こりうる拡張まで考慮するならば、16ビット固定長から16ビット可変長への変遷というものを今から意識しておく必要があるだろう。具体的には内部文字符号体系に関する情報をデータベースのスキーマ内に記録しておく際に、コード変換が主な目的ならUCS2という標識よりUTF16を使うことがある。一方、UTF16のUCS2以外の部分についても文字型としての処理を施すのは実装のレベルにおいて検討すべき課題であると考えている。処理のし易さという観点からはUCS4も文字型の内部文字符号体系としてあてはまりUCS2以外の拡張された文字に対しての文字型としての処理を行なう解決策になるが、現時点では一文字につき２オクテットを余分に消費するので、プログラミングにおける文字処理のプロセス・コードとして有効性が認められても、データを大量に保持しなくてはならないデータベースの内部コードとしては有効性が下がるのが一般認識であろう。

もう一つのサポートの仕方は従来のオクテット型を利用しながら、かつ文字セットの要求を満たすことに重点をおくニーズに応えるもので、オクテット型というデータ型への親和性と、そもそもこの種のデータ型で暗黙のうちに想定されてきたラテンスクリプトの共通部分である７ビットASCII文字集合の符号と8ビット部分が同一符号であるという利点をもつUTF8を文字符号として使うことが必然的であろう。ただし、これには以下のような短所があげられる。

1) 文字データの境界が必ずしもデータ型の境界と一致しない。従って、ホスト変数へのアサイメントの際に起こりうるマルチバイト文字の途中での切り捨てなどの問題が発生。
2) ある一定の文字数に対するオクテット数が定まらないため、入力フィールドが文字数を長さの単位として用いている場合に入力したはずのデータが格納できないという問題が発生。

しかしながら、UTF8の特質による次のような長所とは言わないまでも事実があることも忘れてはならない。

- 文字ストリング全体を格納したり引っ張り出してくるオペレーションをする限りにおいては問題ない。
- Forwardサーチのアルゴリズムを用いた検索処理にも問題を引き起こさない。

短所である１）に関しサポートの行き届いたデータベースでは、文字ストリングの終わりに発生する無意味なオクテット単位は切り取ったりスペースで置き換えたりするなどの処理をしている。また、２）のようにある一定の文字数に対するオクテット数が定まらないという現象によっておこる問題については、格納されるデータ項目に対し十分の長さを定めたデータベースのフィールドのオクテット数を許容される最大値とし、結果としてそれを超えさせないようなユーザーの入力フィールド側での処理（Validation）が必要であろう。オクテット型によるUnicodeサポートは、プログラミング言語とのバインディングを厳格に規定している、すなわち両者の間で文字符号が一致していることを前提にしている処理系においてかつそういう処理系におけるオクテット型をあえて選びたいユーザーのプログラミング環境においては、データベースがUnicodeをサポートする上での解になりそうだ。ただし、Unicodeの特質を十分に利用した文字型でのサポート、にデータベース・ベンダー（開発・提供者側）とプログラマー（使用者側）の双方からみたUnicode利用面での優位性は否めないので、前者の方が将来のUnicode利用経の大きな方向性と言って間違いない。

オクテット型、文字型というデータ型によって従来の各国語別のプラットフォームのネイティブ・

コードを既にサポートしている処理系においては、ＤＤＬ（データベース定義言語）の中でそういう従来のものとUnicodeを区別する手段が必要になってくる。

データベースの処理系が大規模であればある程、マルチリンガルへの要求が高まり、使用者の目からみれば上位互換という観点から処理できる文字符号体系のバリエーションは必要十分であってほしい。さらに、自由なレベルで文字符号体系を設定できるかどうかにも関心が払われる。これらが実際の使い勝手を決める重要な要因になるであろうが、データベースの処理系が動くオペレーテシング・システム環境や処理系の設計思想がもつ制約に多分に左右されるのが現実である。議論のためここで、Unicodeという可能な限りの全世界の文字セットを一つの文字符号体系で網羅している世界だけに特化して考えてみる。すなわち、Unicode以外をサポートするプログラム言語とデータベースとの間での文字符号変換によるパフォーマンスの劣化をあえて容認し、先にあげたバリエーションや設定のレベルを文字レパートリーに関するものとして考えれば、これら要因に関する実装上の考慮は最小限で済む。例えばSQL2（JIS X3005 1995）の場合、ここにおける「CHARACTER型」ではスキーマのレベルにおいてまずデフォルトの文字レパートリーを定義でき、それを個々の列のレベルでオバーライドできる仕様を唱えている。ある一つのUnicodeの符号体系が唯一内部コードであれば、異なる文字符号体系間の比較やアサイメントのオペレーションにおける暗黙のコード変換に伴う多くの作業が必要なくなるので、可能な限りの文字集合を網羅したUnicodeを内部的に利用することによってSQL2準拠のための条件を満たす近道であるともいえなくはない。ただし、現実には従来からの文字符号体系へのサポートを切り捨てる訳にいかない。列レベルでの自由な文字符号体系指定の実装面での難しさが存在する。

また、その規格水準の中級ＳＱＬ以上では標準文字レパートリー名の中にはJIS X0221、すなはちUCS2でサポートされる全ての文字というものを挙げているが、ここではUnicodeサポートが準拠の条件となるにまで至ってはいない。

JIS X3005 1995における「CHARACTER型」の位置付けと解釈についてもう少し触れたい。ここでの「CHARACTER型」はオクテット型とも文字型とも処理系の実装のレベルで決めうちしてない。正確には、スキーマ定義の段階において使用者側が指定できる仕組みになっている。NCHAR型は、CHARACTER SET句によって処理系定義の文字セットをした「CHARACTER型」の一バリエーションと言えるようになった。特に日本（JIS X3005 1995）では、JIS X3005-1990との互換性からNCHAR型を漢字列型と呼び、この論文で論じているいわゆる文字型に属する。ここで気づくことは確かにかつてのNCHAR型の独自性や特有の意義が薄れていることだが、そこには利用者側からみたある一定の価値が覗える。それは、データベースでの文字ストリングの実体がオクテット型であろうと文字型であろうとも統一したＳＱＬ文で扱える、すなはちプログラムに変更を加える必要がなくなるということである。実装のレベルにおいて、複数の異なる文字符号体系が採用されている処理系、例えば一般のユニバーサルといっていい「CHARACTER型」と漢字列型をサポートしているような処理系においてこの利点を現実に実装するためにはかなりの作業と考慮点が必要とみられるが、オクテット型と文字型を文字ストリング処理において融合させることのメリットは大きい。これは、オクテット型と文字型の両方でUnicode文字集合が使えることによってデータとして使用できる文字集合の違いが長さを気にするオペレーション以外にはユーザの目から見えなくなるためだと言える。こういうもとで、利用者側がオクテット型と文字型のオペレーションの違いを必要としない、２つのデータ型でオペレーションを融合させたＳＱＬプログミングを行うには次のような問題を取り除いておかなくてはならない。

- CHARACTER型定数（''）とNCHAR型定数（N''）の間のSQL文のシンタックス上の違いにより引き起こる問題。

- 詰め込み（Padding）文字が全角スペースかまたは半角スペースかにより引き起こる問題。
- 異なる文字符号体系間のオペレーションのために必要なコード変換の問題。

シンタックス上の違いに関しては、差分となっている部分をオプショナルにするやり方が一番妥当であろう。詰め込み文字の問題に関しては、固定長のデータの場合には詰め込み文字があたかもデータの一部となるオペレーションがupdatesや代入における様々な段階で起こることが問題を複雑なものにしている。解決策としては全角または半角スペースを比較オペレーションの前段階で詰め込み文字（Trailing Blanks）に関してだけどちらかにフォールディングしたり取り払う前処理を施すことが考えられる。この辺りに関してSQL2では明示的な照合順序の指定で解決しようとしているが、2種類の詰め込み文字を同じに扱う昨日はデフォルトとしてもっていてもいいものと考えている。

コード変換に関しては、実装のレベルで細部には独自の手法とコード変換規則が取り入れられていることが現状のようで、本稿ではこれ以上詳しく触れない。

3．Ｗｅｂでの利用

World Wide Webの世界は、最も積極的にUnicodeの適用が進められているアプリケーション分野の一つである。その理由は、Webの性格上、世界中に点在する各国語の言語で書かれた電子文書を同時に取り扱わなければならない、というところからきているように思われる。言い換えるならば、Webは、文字種も文字符号も異なる複数の電子文書を同時に取り扱う統合環境であり、その効率的な実現の為には、電子文書処理システムが採用する内部文字符号は、各国語で書かれた電子文書で使用されている全ての文字を含んでいる必要がある。現在この要求を満たす文字符号系としては、ISO/IEC 2022の符号拡張法を用て、各国の文字符号を切り替えながら文字の符号化を行う方法と、世界中のコンピュータで頻繁に使用されているほとんどの文字を、１つの符号表内に含むUnicodeが存在する。電子文書の静的な表現としては、この２つの符号化表現に優位差はない。もしくは、前者の方が既存の符号表現からの変換の容易さにおいて優位であるとも言える。しかし、Workable Textとして、文書処理を前提とした動的文書の表現としては、文字符号表現が状態をもたない分、Unicodeの方に利点が大きい。そのため、WebではUnicodeの方が積極的に利用される傾向にある。その例としては、HTTPやHTMLが文字符号としてUnicodeやUTF-8を許していることに加えて、XML (eXtended Markup Language)ではUTF-16をデフォルトの文字符号として規定したり、DOM (Document Object Model: Web Objectにアクセスする為のAPI群)が文字型の内部表現としてUTF-16を採用し、また、国際化URLとしてUTF-8がURLの%エスケープで用いられる文字符号として採用されようとしていることが挙げられる。

さて、Unicodeで書かれた電子文書の処理を考える上で問題となる事項の一つには、「電子文書上で文字の同一性」問題がある。Unicodeが複数の空白文字をもっていたり、また、非合成文字と合成文字の列による文字合成シーケンスをもっていることに起因して、異なった二つのUnicode文字列が、全く同一、もしくは人間の目では差異の判別が難しいほとんど同一の印字表現をもつことがある。前述のプログラム言語や、データベースというアプリケーションの観点からは、文字列を構成する要素である「文字」自体、およびその並びが異なるのであるから、それらの文字列は、たとえ印字表現が似通ったものであるにせよ、文字符号データとしては全く異なったものであると考えられる。しかし、文書としては、人間が目で判別が不可能である、もしくは困難な「文字列」表現を、異なったものとして取り扱うことは、ナンセンスであると言わざるをえない。たとえば、ラテン文字の'A'にアキュート・アクセントの付いた「文字」をもつ既存の符号系で表された電子文書では、その「文字」は一つ要素からなる文字符号で表現されるであろう。しかし、'A'にアキュート・アクセン

トのついた「文字」を要素としてもたない文字符号で表現された電子文書では、同じ「文字」は文字合成制御シーケンスを用いた、'A'という文字の文字符号と、単独のアキュート・アクセントの文字符号という二つの要素からなる文字符号列として表現されるているかもしれない。それらの「文字」の符号表現の違いは、符号化文字集合の制約からくるものであり、電子文書が表現する「情報」としての違いは認められない。よって、Webの利用者の利便性を考えるならば、'A'にアキュートアクセントの付いた「文字」を含む単語を含む電子文書を、Webで公開されている文書群の中から検索するという処理を行う場合、検索処理は、「文字」の符号表現の違いを無視して、利用者にとって有為な印字字形に注目して執り行われるべきである。UnicodeもしくはISO/IEC 10646では、この印字字形に対応する情報単位をCC Data Elementとして、Unicodeで言うところの文字とは区別して考えている。また、World Wide Webコンソーシアムでは、Character Model for Webと称して、Webにおける「文字」の取り扱いについてRequirements for String Identity Matching and String Indexing（参考文献4）というWeb上での「文字列の取り扱い」についての要求項目をまとめた技術レポートを発行し、現在その要求項目をWeb関連技術で、どのように実装するかについて、検討を進めている。

異なった文字符号列で表現されたCC Data Elementの同一性を検査する手法としては、比較実行時に、適切に定義された照合順序テーブルを参照し、文字列を照合順序を表す値に変換した後、その値同士を比較する方法（C言語で言うならば、strxfrm() 関数もしくはstrcol()関数を利用する方法）と、事前にCC Data Elementを構成する文字符号列の正規化を行い、文字符号列中のCC Data Elementの表現方法の統一を行ってしまう方法が考えられる。前者は、比較される文字符号列自体の変更を伴わないので、スマートであり、また一度Unicodeに変換された文字符号列を元の文字符号を用いた符号列に再変換しようとした場合にも、多くの場合可逆変換が保証されるので安全な方法であるのだが、文字符号列を照合順序を示す値に変換する為には、テーブルを参照しながら文字列を照合順序値に置き直す計算を複数回（最大６回程度）行わなければならず、実行時の計算コストが高つく。それに引き換え、後者の方法は文字列自体の変換を行ってしまうので、変換前の状態への可逆変換は不可能になり、また変換の際に文字列の長さまで変わってしまう危険性があるのだが、その変換は電子文書の処理系が電子文書を作成するとき、およびWeb上に存在するUnicode以外の文字符号で表された既存の文書をUnicodeに変換する際に一度だけ実行されればよく、Webというシステム全体をでの効率を考えた場合、その変換、検索処理に要する実行時コストはかなり小さくなる。この文字符号情報の保存と、実行時コストとのトレード・オフの結果、Web上での実装技術としは文字符号の保存がさほど重要ではない（サーバーとクライアントとの間の電子文書の転送に際して、両者がともに理解可能な文字符号への自動的な変換がなされることがある）との理由により、後者の手法が選ばれようとしている。World Wide Webコンソーシアムではこの手法を、Early Normalizationと呼び、電子文書がWeb上でUnicodeで符号化される最も早いタイミングで（エディターでUnicodeで書かれた電子文書を作成する時、または既存の文字符号で書かれた電子文書を、通信経路もしくは電子文書処理システムがUnicodeに変換する時）にこの処理を執り行い、以降Unicodeのバイナリ値での文字列の検索、比較が可能になるようにしようと考えている。

ところで、このようなUnicode文字列の正規化を行う際に問題になるのは、どのような方法でその正規化を行うかということである。それについてはUnicode Technical Report #5（参考文献3）がUnicodeコンソーシアムで検討されている手法について詳しく述べている。簡単にその手法を解説するならば、正規化は以下に述べる３つのフェーズで行われる。

1) 既に合成済の文字(pre-composed form)の合成シーケンスへの分解

2) 合成シーケンス内での合成文字の並び換え
3) 正規化された合成シーケンスの先頭からの合成シーケンスの合成済の文字への再合成

この３つのフェーズの内、技術的に難しいのは第２フェーズの合成文字の並び換えのフェーズの実装法である。この並び換えは、合成文字ごとの並べ順を示す重み付けを定義したテーブルを参照して行われる。合成文字には、合成シーケンス中に現れる合成文字の出現順序によって印字字形が影響を受ける種類のものと（ある非合成文字の上部に二つのアクセントが付く場合、アクセント文字の出現順によって、どちらのアクセントがより上位に付くかが変わることがある）、影響を受けない種類のものと（ある非合成文字の上部と下部に一つづつアクセント文字が付く場合、どちらのアクセントが先に出現しようと、印字字形に変わりはない）が存在する。前者の場合、テーブル上それらのアクセント文字の重みとして、同じ値を定義することにより、同じ重みを持つアクセント文字同士の出現順に関しては、元の文字符号列での並びを正規化後も保存することにする。一方後者の場合、それらの合成文字の重みとして、異なった値を定義することにより、その重みに従った順への当該合成シーケンス内における合成文字の出現順の正規化が行われるようにする。問題は、その合成シーケンスが含まれる単語の言語情報、及びUnicodeに変換される前の符号化文字集合の情報に依存しないで、正規化ルールのテーブルを作ることが出来るのか、そのテーブルでルール化できない例外事項をいかに減らせるか、さらに、将来のUnicode改訂（文字の追加）に、そのテーブル、および第３フェーズの再合成のためのルールセットが追随できるかということであるが、筆者は多国語の専門家ではないので確証をもてずにいる。

話をアジア圏の漢字スクリプトに移すと、Web上における文字のアイデンティティの問題として、異体字の解釈が浮かび上がってくる。Unicodeの文字・文字符号はUnified CJK Characterと呼ばれるように、アジア圏（日本、台湾、中国、香港、韓国，ベトナム）で使われている文字を統合して、その統合された文字に対して符号値を割り当てたものである。その統合・包摂の基本ルールは、Unicode（正確には、ISO/IEC JTC1/SC2/WG2 Ideographic Rapporture Group）が作り出したものであるが、そのルールの例外事項として、各国の文字符号規格がその包摂基準上別の文字としているものについては、Unicodeの包摂基準でいえば同じ文字と同定できるものであっても別の文字と判断して、別の符号値を割り振るというものがある。このルールは、各国毎に異なる包摂基準を尊重し、既存の文字符号との間での相互変換を可能にする為に設けられたものであるが、別の見方をすれば、文字の同一性の検査の為にはその文字を含んでいる単語の言語情報を必要とする、ということを意味している。つまり、日本語の包摂基準から言えば、Unicode上の二つの文字は同じ文字であり、その文字を含む単語を日本語の単語として検索する際には、それらの文字を同じものとして取り扱ってほしいが、その文字を含む単語が中国語の単語であるならば、それらは別の文字であり、別の文字として取り扱われるべきである、ということがありえる。また同じ言語で書かれた電子文書であろうと、片方がたとえば、JIS X 0208漢字集合を用いて書かれ、もう一方がそれに加えて、JIS X 0212補助漢字集合をも用いて書かれていた場合、JIS X 0208の包摂基準では同じ文字と同定されていたものが、JIS X 0212を加えた場合、別の文字として区別されるということが起き得る。これらの文字の包摂基準の違いを吸収して、利用者が思った通りの検索が行えるようにするためには、異体字関係を定義したシソーラスのような辞書を利用者ごとに作成し、利用者にとって好ましい異体字関係を教え込むことによって、ファジー検索を行う際、処理系がそれらの異体字を同一の文字として取り扱えるようにしてやらねければならない。しかし現在のところ、このような異体字関係を表す辞書、およびその辞書を利用したファジー検索の技術は確立されていない。

参考文献

[1] 織田哲治，木戸彰夫，小田明：Unicodeの最新動向，情報処理学会デジタル・ドキュメント研究会資料12-3，pp17-24，1998

[2] ISO/IEC TR 10176 Second edition: Information Technology – Guidelines for preparations of programming language standards, 1998

[3] Unicode Technical Report #15 (Draft), (http://www.unicode.org/unicode/reports/tr15/).

[4] Requirements for String Identity Matching and String Indexing, (http://www.w3.org/TR/1998/WD-charreq-19980710)