

## 1 歴史を少し

計算機利用の黎明期 (1962)、カリフォルニア工科大学で博士課程を終えようとしていた一人の男がある本を書きはじめた。男の名は Donald E. Knuth という。そしてその本 *The Art of Computer Programming* はコンピュータアルゴリズムの基本的な教科書の一つとなった。他の著者と同じように、彼も自分の本がきれいに組まれて欲しいと願った。他の著者と違っていたのは、スタンフォード大学にいてその本の第2版を発行しようとしていた70年代終わりに、そのために必要なツールも開発しようと思ったことである。彼が学部生向けの一年間のプロジェクトだと思ったことを完成させるためには、彼本人が丸十年かかった。彼はこのプロジェクトを  $\text{T}_\text{E}\text{X}$  と命名した。 $\text{T}_\text{E}\text{X}$  は組版のためのプログラミング言語、適切なコンパイラ、必要なフォントやユーティリティなどからなっている。それからほぼ30年がたつが、 $\text{T}_\text{E}\text{X}$  はまだ使われている。そして PostScript 言語のような「産業用」技術と共に使われて強力な組版ツールとなっている。それにもかかわらず、 $\text{T}_\text{E}\text{X}$  プロジェクトには一つ特殊な点がある。それは1990年に Knuth が  $\text{T}_\text{E}\text{X}$  の開発を止めたという意味で凍結されているということである。もちろん今でも、 $\text{T}_\text{E}\text{X}$  プログラミング言語でプログラムを書くという形での貢献は可能である。しかし  $\text{T}_\text{E}\text{X}$  コンパイラは 1990年の状態で不朽なままである。

もし  $\text{T}_\text{E}\text{X}$  が英文の数学やコンピュータ科学の本、たとえば *The Art of Computer Programming* を書くためだけに使われるのであれば、これは些細な問題だろう。しかし、実際はそうではない。もし  $\text{T}_\text{E}\text{X}$  を他の状況、たとえば英語以外の言語での組版、に適用しようとしたとたん、重大な限界にぶつかってしまう。計算機の世界では、ソフトウェアは普通、英語指向である。ほとんどの計算機やソフトウェア開発は、マーケティングと同様、米国で行われているのだから。しかし、他の言語で当然と思われていること（右から左、上から下に書くことやダイアクリティカルマークの組み合わせ、多くの種類の文字を使うことなど）が、英語指向のソフトウェアでは大きな問題となる。

著者らも含め多くの人々が、複雑な  $\text{T}_\text{E}\text{X}$  のコードを書いたり、外部のプリプロセッサやポストプロセッサを適用したりすることで、 $\text{T}_\text{E}\text{X}$  の欠陥を回避しようとしてきた。これらの手法は  $\text{T}_\text{E}\text{X}$  が何でもできるということを証明はしたものの、その結果はしばしばみっともなく不自然なものとなった。 $\text{T}_\text{E}\text{X}$  を拡張し、問題を解決できる新しいシステムが必要なが明らかとなったのだ。

このような理由から、著者らは1994年以来、多言語組版やXML文書処理といった新しい領域を探求する、 $\text{T}_\text{E}\text{X}$  に基づいた新しいシステムを構築している。システムの名前は  $\Omega$  という。 $\Omega$  はギリシャアルファベットの最後の文字であり、西洋文化では頂点、究極、至上をしばしば意味する。

$\Omega$  は、すべての言語に対して、全システムがその言語を念頭において設計したかのように、できる限り自然であることを目標の一つとしている。たとえば  $\Omega$  では、どの方向へ書く場合にも、同じように、同じ特徴と同じ機能を利用できる。多種多様な文字も  $\Omega$  にとっては問題ではない。 $\Omega$  の内部表現は 16ビット (もうすぐ31ビット) だから。ダイアクリティカルマークの組み合わせはたいへん低レベルのプロセスで処理されるため、ユーザが書く高レベルの  $\text{T}_\text{E}\text{X}$  コードと共存できるだけでなく、ユーザが組み合わせをコントロールすることができる。

1994年以来今日まで着実に、著者らは  $\Omega$  を開発し、新しい機能を追加してきた。表記法のように人間の本質に関わる問題に対する解決をみつけることはすばらしい

経験である。もっとも興味深い挑戦は、プログラミングは普通一定の厳密さを必要とされ、表記方法はそうとは限らないという点にある。典型的な例はタイ語である。極東の言語の多くと同じように、タイ語の文を構成する語は、見た目では別れて表記されていない。語は音節からできている。タイ語の段落を行に詰めていくときには、行末は単語の間か、単語の音節の間に来なくてはならない。音節の間に行末が来た場合にだけ、ハイフンが組版システムによって追加されなくてはならない。つまり、組版システムは単語の切れ目を知る必要がある。どんなタイ語話者でも単語の切れ目を知っているが、タイ語文書を打つ時に単語の切れ目を何らかの方法で明示するように強いることはΩの「自然原理」に反している。人々が自分の言語をタイプする際の習慣を長い間持っているのなら、そのやり方が「もっとも自然な」方法なのであり、Ωはその習慣を続けられるようにしなくてはならない。つまり、タイ語の単語の切れ目が、たとえ切れ目が明示されていない文書の中であっても認識できるように、Ωに教えなくてはならない。これは文書の形態素解析を要する言語学的問題となる。

Ωに取り組むことは、計算機科学、言語学、組版、フォントデザイン、言語やその他分野の歴史に関わることである。Unicodeによる符号化<sup>1</sup>の成功は、Ωプロジェクトにも非常に役に立った。Unicodeは、Ωプロジェクトの適切な基盤となるだけではなく、計算機関連の人たちに国際化の必要を気づかせてくれた。Ωの目標は、Unicodeに含まれるより多くの言語、より多くの部分をカバーすることによって、ユニコード文書の*de facto*の組版システムとなることである。しかし「組版システム」と「レンダリングエンジン」を混同してはならない。残念ながらこの混同はしばしば起こっているのだけれど。Unicodeは情報交換用符号（情報交換用エンコーディング）であり、データの交換とデータの印刷上の表現の間には根源的な違いがある。内容と形式、文字とグリフの違いである。この違いは、英語のように文字とグリフの間にほとんど一対一の対応がある言語の場合には、根本的な問題であるとはいえ、理論上のものにとどまっている。しかし他の言語や表記法においてはより重大である。アメリカ製のデスクトップパブリッシングソフトがどんどん世界中の組版に使われるようになってきたため、組版もまた画一化されつつあり、地域の印刷の伝統はアマゾンの森林と同じ勢いで消滅しつつあることは言うまでもない。

Ωは世界中のどこでも、どんな文脈でも使用できるほどオープンであり、強力である。実際、Ωはパブリックドメインであるため、経済障壁を越えることができている。世界のすべての組版の伝統を実装するという長い道のりを、Ωは進んでいる。

## 2 フォント

活版印刷の中心概念は「動かせる活字」(movable type)の利用である。これは文書を記号（文字、句読点など）に分解することである。記号は繰り返され、組み合わせられ、並べられて行となり、行は縦に並べられてページとなる。フォントはこのような記号の集まりである。コンピュータ上のフォントのパラダイムは、グーテンベルグの印刷機(1470)のホットメタルの活字やグーテンベルグに先立つこと数世紀(970)中国で発明された木版活字に呼応している。

<sup>1</sup><http://www.unicode.org>

Opt $\TeX$ はこのパラダイムを以下の方針のもとに実装しようとした。

- 組版は2次元である。
- 本はページから、ページは行から、行は語と句読点から、語は文字から構成される。
- はじめは、文字は単なる箱である。この箱は垂直方向に二つの次元（ベースラインに対する高さheightと深さdepth）と水平方向に一つの次元（幅 width）を持つ。
- 第2段階では、これらの箱はピクセルまたはベジエ曲線で表される形状で満たされる。この段階は実際には $\TeX$ の外で起こる。
- 文字の相互作用には二種類ある。文字は互いにくっついたり離れたりするか（カーニング）または合字（リガチャ）と呼ばれる新しい記号に変わる。

Opt $\TeX$ フォントはこれらの方針に基づいて設計されている。 $\TeX$ が最初に開発されて以降に現れた他の制約が無かったとしたら、このやり方はは充分なものだっただろう。

<http://www.rakuten.co.jp/iseshima/295582/414382/>制約の一つはエンコーディングである。これはキャラクタ（文字、数字、句読点等）とフォント中のその順序を示す番号（フォントは表として表され、個々のキャラクタはその座標で表されることが多いので「位置」と呼ばれる）の対応関係である。 $\TeX$ が開発された際、D. E. Knuth はフォントの大部分に ASCII と呼ばれるエンコーディングを採用した。ASCIIは今も存在するのだから、これは賢明な決断だった。しかしASCIIはUnicode同様、情報交換用符号であり、したがって定義により組版に充分なものではあり得ない。 $\TeX$ フォントには、ASCIIにはないが組版上必要な追加の記号が含まれていた。このオリジナルの $\TeX$ フォントエンコーディングは現在‘OT1’ $\LaTeX$ エンコーディングと呼ばれている。

ここ30年の間に、何百という異なるエンコーディングが定義されてきた。‘OT1’ $\LaTeX$ エンコーディングのようなフォントエンコーディングが、さまざまなフォントデザイナー、コンピュータメーカ、ソフトウェア会社によって定義され、公式非公式を問わず多くの情報交換用符号化が定義されてきた。多くのエンコーディングが非英語圏のユーザの特別な必要に応じるために定義された。

問題は、 $\TeX$ ユーザが $\TeX$ 以外のフォントを使いたいと思ったときや、逆に $\TeX$ フォントを他のプログラム（たとえばグラフィックデザインソフト）で使いたいと思ったときに起こる。後者の問題が本当の意味で解決されたことは無い。（実のところ、 $\TeX$ 風の表現を作り出す他の方法が見つかっている。たとえばEPSファイルの中のテキストを $\TeX$ で組まれたテキストに置き換えるpsfragパッケージのように。）前者の方は興味深い新しい概念であるバーチャルフォントに繋がった。バーチャルフォントは「抽象的なフォント」であり、キャラクタは空の箱でピクセルもベジエ曲線も含んでいない。バーチャルフォントは $\TeX$ と外界のあらゆるエンコーディングのフォントとの間の中間的ステップとして用られる。

バーチャルフォントは既存のフォントのエンコーディングを変更するだけでなく、異なるフォントからのグリフを組み合わせて合成文字を作り出すこともできる。これは大変役に立つ。たとえばアクセントの付いた文字が必要な場合、文字とアクセントをそれぞれ含んでいる「本当の」フォント（本当のピクセルかベジエを

含むフォント)があれば、バーチャルフォントがそれらをアクセント付きの文字に合成することができる。

「多言語」組版を行おうとすれば、つまりシステムが異なる言語の文脈に対して同じように適切な結果を出さなければならないとすれば、さらに多くの問題が起こってくる。典型的な例はラテン文字の上に付加される分音記号 (dieresis = トレマ = ウムラウト) である。ドイツ語の組版では、フランス語の組版よりアクセントは低い位置に組まれる。このような場合、「多言語」フォントは複数のしばしば矛盾する制約を満たさなくてはならなくなる。解決方法は明らかに二通りある。可能な場合をすべてカバーできるようにフォントに含まれる文字数を大きくするか、文脈によってキャラクターの特性のいくつかが変化するような動的なものにしておくかである。いいかえれば、フォントが充分大きいか、その場で必要な合成が行えるほど賢いかのどちらかであればよい。

分音記号の例は、非ラテンアルファベット言語での要求に比べれば、ほんの小さな問題に過ぎない。それらの言語では、グリフは組み合さってどんどん複雑なまとまりを構成していく。日本語の振り仮名はこのような例である。ほとんどの場合、表意文字の上 (縦書きに組んでいる場合には右) に一つか二つの振り仮名を置けばよい。しかし時には一群の表意文字の上 / 右にたくさんの振り仮名をつけなくてはならない。この振り仮名の付け方は非常に特殊で、はっきり決まっている。T<sub>E</sub>Xの場合ならばこの問題は、すべてのキャラクター (ベースの表意文字と振り仮名) を別々の単語ととらえ、それらを二次元に組み立てることで解決することになるだろう。このやり方はある程度しか有効ではない。それというもの、テキストがその統一性を失い、キャラクターを含んだそれぞれの箱の集まりになってしまうためである。

OpT<sub>E</sub>Xのもう一つの欠陥は、日本語や他のCJK (中国・日本・韓国) 言語を組版しようとする時に明らかになる。T<sub>E</sub>Xフォントは横組みに使うように設計されている。もちろん標準のT<sub>E</sub>Xフォントを使って縦に組版することはできるが、文字間の縦の距離は好ましいものにはならない。縦組みの場合には、CJKのアクセント (傍点など) はベースキャラクターの右や左に、水平軸に沿って中央に置かれる。つまり、「アクセントを置く」(あるいは“diacriticizing”。これは「識別する」を意味するギリシャ語の動詞) διακρίνωから来ている) 行為は、テキストの方向とは直行する軸を常に使うことになる。これらの直行する軸は文字の丁度半分の位置を通るとは限らない。非対称なグリフでは、アクセントの軸は中心からそれていることもある。言いかえれば、多言語、多方向の組版をしかるべく行うためには、各フォントの各キャラクターごとに、軸の位置を垂直であろうと水平であろうと知っておく必要がある。(図 1参照)。

Unicode standard bookをちょっとめくってみれば、キャラクターの上下左右に置かれるダイアクリティカルマークだけでは充分でないことがすぐにわかる。ヘブライ語のdageshのようにグリフの中に置かれる場合すらある。同じ種類や違う種類のダイアクリティカルマークが様々に組み合わせられることは言うまでもない。

キャラクターにダイアクリティカルマークを付けることは面白いし、実際、特にたくさんそれを使う言語もある (たとえばベトナム語。) しかしダイアクリティカルマークは、キャラクターの値や振る舞いの他の側面に影響を与えるべきではない。たとえば、ダイアクリティカルマークがはみ出していない限り、マークが付いた文字もそうでない文字も全く同じようにカーニングされるべきである。ÂVÂTÂR (悪いカーニング) とÂVÂTÂR (よいカーニング) を比べてほしい。別の例をあげてみよう。文法の本では、ある文法規則を強調するために語の特定の文字だけにア

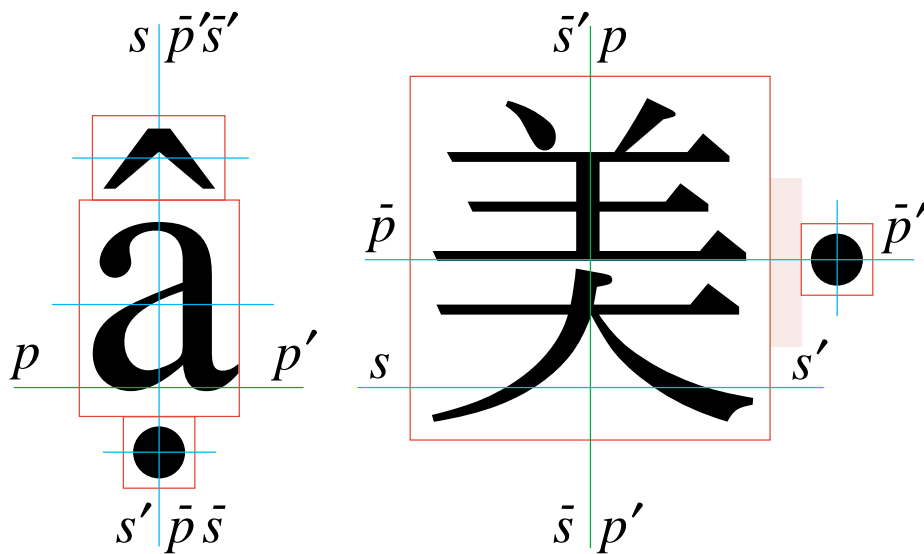


Figure 1: 一次、二次ベースラインとアクセントの方向。ラテンスクリプトの場合には一次ベースライン $p \rightarrow p'$ は水平であり、一次アクセント $\bar{p} \rightarrow \bar{p}'$ は垂直上向きである。二次ベースライン $s \rightarrow s'$ は垂直下向きで二次アクセント $\bar{s} \rightarrow \bar{s}'$ も垂直だが、上向きになる。表意文字の場合、一次ベースライン $p \rightarrow p'$ は垂直下向き、一次アクセント $\bar{p} \rightarrow \bar{p}'$ は水平右向き、二次ベースラインは $s \rightarrow s'$ は水平右向き、二次アクセントは $\bar{s} \rightarrow \bar{s}'$ は垂直上向きとなる。

アンダーラインが引かれることがある。しかし、ハイフンを付ける必要がある場合には、この語もアンダーラインが無い場合と全く同じように扱われるべきである。

これらの問題を解決するために、 $\Omega$ は $\text{T}_\text{E}\text{X}$ の基本的なパラダイムに新しい方針を導入した。

- フォントは組版に必要なすべてのグリフが含まれるよう大きくなくてはならない。
- 文字は単なる箱である。 $\text{T}_\text{E}\text{X}$ でも使われていた三つの値（高さ・深さ・幅）は、テキストの主方向に関する情報となる。箱はこれに加えて、テキストの二次方向に関する高さ・深さ・幅、主方向と二次方向での対称軸という情報を持つ。そしてグリフ内にアクセントを置く場合のような特別な場合には、追加の情報（追加の座標点）を持つこともできる。
- 組版は複数のレベル（ページ > 行 > 語 / 文字 > アクセント > アクセントのアクセント）で行われ、レベルの変更ごとに組みの方向も変更される（主方向から二次方向へ、またはその逆）。各文字は、主方向用に $\text{T}_\text{E}\text{X}$ で持っていた属性すべてを二次方向用にも持つ。
- $n + 1$  レベルで起こることは、はみ出しの問題がない限り $n$  レベルに影響しない。

最後の方針を示す例をみてみよう。ページは行からなる。これを「レベル1」と呼ぼう。行は語からなる。これを「レベル2」としよう。これらの単語にアセンダ（訳注：xの高さより上にはみ出す部分）のある文字が使われているかどうかにかかわらず、行の間の距離（「leading」）は常に同じである。語に含まれる文字の一つが大きすぎて、leadingが同じであると前の行に重なってしまうような不幸な場合をのぞいて。文字にアクセントをつけることは「レベル3」である。前述のように、文字にアクセントをつけることによって、カーニングが影響されてはならない。付けたアクセントが回りの文字やそれらのアクセントと重なってしまうような場合をのぞいて。つまり、(a)レベル $n$ の組版中にはレベル $n+1$ を無視し、(b)(a)の最中、レベル $n+1$ の処理が重なりを起こすかどうかを管理し、(c)重なりが起こる際にはレベル $n$ の振る舞いを変更する、ことができなくてはならない。

前段落を再読してみてほしい。ここではページの構造について書かれているが、縦組みか横組みかどちらであるかについては書かれていないことに気が付くだろう。実際、書字方向の所でのべたように、 $\Omega$ は組みの方向を平等に扱っており、すべての方向で同じ機能と属性が利用できる。

この新しい組版モデル(文書をレベルに分解し、レベルごとに方向が変わる)によって、 $\Omega$ はどれほど複雑な組版の問題であろうと解決できるようになるだろう。たとえそれがアジアの言語であろうと、たくさんのダイアクリティカルマークを持つ人工/自然言語であろうと、必ずしも言語ではないが二次元の構成を持つもの(音楽の譜面や技術図面など)であろうと。

### 3 さまざまな方向へ書くこと

ほとんどの文書処理ソフトは、文は横書きで左から右に書かれ、行は上から下へ流れるということを基本的に想定している。世界の人口の多くがこの書字法を使っているが、決して全員がではない。

アラビア、ヘブライなど中東発祥のスク립トは、右から左へ横書きされる。日本を含む極東の伝統的な書字/印刷方法は縦書きであり、最初の行がページの右側に来る。ウイグルやモンゴルでは縦書きであるが、最初の行がページの左側になる。

さらに混乱を深めることに、「自然な」書字方向が異なる複数のスク립トを一つのページに混在させることもできる。この時には、一つのスク립トを回転させて組んで、全体のスク립トの流れに合わせることになる。

$\Omega$ はこれらの問題を一般性を保ちつつ解決している。 $\Omega$ では、書字方向は3つの属性で指定され、それぞれの属性は、Top, Bottom, Left, and Rightのどれかの値を取るとする。これらの値は、実際のページの上下左右の端のうち一つを示すものである。さて書字方向は

主方向 ページの「上」

第二方向 ページの「左」

第三方向 各文字の「上」

を指定する。第二方向は主方向と直行しなくてはならないが、第三方向は4通りのうちのすべての値を取ることができる。つまり3 2通りの書字方向が可能である。良く用いられているものを挙げておこう。

Vertical CJK: CJK scripts  $\begin{matrix} \text{use} \\ \text{R} \\ \text{T} \\ \text{T} \end{matrix}$ , LR scripts  $\begin{matrix} \text{use} \\ \text{RTR} \\ \text{R} \end{matrix}$ , RL scripts  $\begin{matrix} \text{use} \\ \text{RBR} \\ \text{R} \end{matrix}$  (RBR), and MU scripts  $\begin{matrix} \text{use} \\ \text{RTL} \\ \text{T} \end{matrix}$  (RTL).

Horizontal: LR and CJK scripts use TLT, and RL and MU scripts “ТЯТ ээв” (TRT).

Vertical MU: MU and RL scripts  $\begin{matrix} \text{use} \\ \text{LTL} \\ \text{T} \end{matrix}$  (LTL), LR scripts  $\begin{matrix} \text{use} \\ \text{LTR} \\ \text{R} \end{matrix}$  and CJK scripts  $\begin{matrix} \text{use} \\ \text{LTL} \\ \text{T} \end{matrix}$ .

Figure 2: 様々な方向のテキストの例。テキスト全体は左から右へ流れているが、違う向きに書かれるいろいろな部分を含んでいる。各行は書字方向が混在するもっともありふれた場合の構造を説明している。

**TLT** — 左から右へ書く (LR) スクリプト、横書き CJK。

**TRT** — 右から左へ書く (RL) スクリプト。

**RTT** — 縦書き CJK, 縦書きCJK中に縦に書かれた LR スクリプト。

**LTL** — モンゴル、ウィグルスクリプト。

**RTL** — 縦書き CJK 中のモンゴル、ウィグルスクリプト。

**RTR** — 縦書き CJK 中の回転された LR スクリプト。

**LTR** — モンゴル、ウィグルスクリプト中の回転された LR スクリプト。

**LTT** — モンゴル、ウィグルスクリプト中の縦書き CJK。

方向パラメータは、ページ (ヘッダとフッタ用)、ページ内容、パラグラフ、テキスト、数式の 5 種類について設定でき、それぞれどのような値をとることもできる。

## 4 Ω翻訳プロセス

Ωの設計方針のひとつは、組版をどんな文脈でもできる限り自然なものとする事である。最初の節で見たような歴史的な理由から、「自然な」組版とは、しばしばΩがタイプライター入力のようなもの (あるいは「論理の入力」と組版との間の中間ステップを補うことを意味している。たとえば、フランス語の組版ではセミコロンはいつも小さなスペースの後に来る。(このスペースは幅は可変だが、せいぜい 1 em の 1/4 ~ 1/6 程度の薄いもので、語間のスペースよりは小さくなくてはならない。語間のスペース自体も可変である。) フランス語の文書を入力する際、人々はセミコロンの前にスペースをタイプする習慣がある。つまり、製版の「もっとも自然な」方法とは、セミコロンの前にスペースを入力し、結果としては前述の適当な薄いスペースを出力として得ることなのである。

これらの規則に沿って薄いスペースを定義するためには、明らかに $\TeX$ コマンドがいくつか必要である。これらを一つのマクロコマンドに圧縮できるかも知れないが、それより小さくはならない。したがって、 $\Omega$ はこのマクロコマンドをすべてのセミコロンの前に挿入する必要がある（もちろん数式中やフランス語の文脈以外で使われているセミコロンを除いて）。

このような変換を簡単にするために、 $\Omega$ では $\Omega TP = \Omega$  Translation Process（ $\Omega$ 翻訳プロセス）という概念を導入している。 $\Omega TP$ とは、テキスト（より正確にはデータストリーム）を読んで文脈に応じて適切な動作を行うフィルタのようなものである。もう一つの良くある例は、大文字 / 小文字の変換である。ラテン、ギリシャ、キリル、アルメニアアルファベットを用いる言語では、大文字と小文字の区別がある。組版システムは、時に大文字 / 小文字を変換しなくてはならないことがある。たとえばヘッダーを大文字で書く本など。この変換は単純なものではない。第一に、何を変換し何を変換しないかを、正確に知っていないといけない。「A device working with 100 mW current」という文が「A DEVICE WORKING WITH 100 MW CURRENT」に変換されれば大惨事である（Mは $10^6$ だが、mは $10^{-3}$ だから）。第二に、小文字から大文字、また逆の変換がどのように行われるかを知らなくてはならない。たとえばトルコ語では、ラテン文字 'i' の大文字は 'İ' であり、他のラテン系言語のように単なる 'I' ではない。ギリシャ語では、大文字に変換するとアクセントと気音符（breathing）は消えるが、曖昧さを無くするため隠れていた分音符が現れることがある。

繰り返すが、これは明らかに  $\Omega TP$  向けの仕事である。では、フランス語の文章を大文字に変換するという仕事はどうだろう？ 明らかに、 $\Omega TP$  は組み合わせができる必要があり、入れ子にできればもっとすばらしい。これは $\Omega$ がやっていることそのものである。 $\Omega TP$  は組み合わせ、入れ子にして使うことができるので、どのような文脈でも、グローバルでもローカルでも、適切な結果を得るために適切な $\Omega TP$ を起動することができる。

前の節で、エンコーディングについて述べた。これらが負担となるのは、フォントエンコーディングに何通りもあるからだけではなく、組版しなくてはならない文書データが、地域、言語、OS、コンピュータメカなどによって異なるエンコーディングによってキー入力されるせいでもある。異なるエンコーディングを変換する $\Omega TP$ を書くことは確かに可能ではある。しかし、 $n$ 個のエンコーディングがあれば、すべての可能な組み合わせを尽くすためには $2n^2$ の異なる $\Omega TP$ が必要となるため、不毛であろう。一つのエンコーディングには存在するが、他にはない文字をどう扱えば良いかという問題があることはいうまでもなく。

この問題にはすばらしい解決法がある。Unicodeである。Unicodeの設計方針の一つは、既存の情報交換用符号中のすべての文字を含むべきであるということである。このことは、どんなエンコーディングでデータが入力されようと、常に、情報を失うこと無く、Unicodeに変換できるということを意味している。そしてもちろん、必要とされる $\Omega TP$ の数もたいへん減少する。 $n$ 個のエンコーディングに対し、 $n$ 個の $\Omega TP$ で済み、すでにUnicodeエンコーディングになっている文書は変換する必要すらない。

今まで挙げてきた例は、 $\Omega TP$ の三つの主なカテゴリを示すものである。すなわち、データをUnicodeに変換するもの（データの「標準化」または「浄化」プロセス）、データを言語学上または組版上の必要に応じて変換するもの、（これはUnicodeエンコーディングだと一番うまくいく）、そしてUnicodeのデータをフォントエンコーディングに変換するものである。Unicodeはフォントエンコーディング

ではなく、情報交換用符号であるため、三つ目のカテゴリが明らかに必要である。(「Unicode エンコーディング」であると称するフォントもたくさんあるけれど、単に組版には適していないという意味でしかない...)

$\Omega$ TPは $\Omega$ の隠れた機構であり、(ミクロな)組版の大半の問題を解決できる。多分もっとも重要な点は、エンドユーザから隠されてはいるものの、 $\Omega$ TPはユーザが完全にアクセスでき、修正できるということであろう。実際 $\Omega$ TPのシンタックスは大変簡単であり、どんなユーザでも、プログラマでなくても、自分の問題を解決するための $\Omega$ TPを書くことができる。たとえば、日本の $\Omega$ ユーザがふりがながたくさんついた長い文書を打たなくてはならないとしたら、ふつうのかなを特別な方法でマークして入力し、望みの結果を得るためのシンタックスを作り出すことができる。どうやってやるかは重要ではない、 $\Omega$ TPは書かれたものを $\text{T}_{\text{E}}\text{X}$ コマンドに変換するのだから。もちろん直接 $\text{T}_{\text{E}}\text{X}$ コマンドを書くことはできる。しかし(a)時間がかかり(b)文書の可読性を下げる。この方法の効率が良いのは、 $\Omega$ TPを書くことが簡単な仕事の場合である。

しかし $\Omega$ TPにはとても複雑なものもある。たとえばアラビア語、シリア語、モンゴル語アルファベット(文字が互いに繋がったり完全に形を変えたりすることがある)の文脈解析も $\Omega$ TPで行うことができる。通常のソフトウェアのほとんどでは、この仕事はオペレーティングシステム(MacOSでの Arabic Language Kit, Arabic MacOS, Arabic Windows 95/98, Windows 2000)が行っている。これらのオペレーティングシステムの一つでアラビア語を組版し、文字のうち一つだけに色を付けてみて欲しい。文字の間に挿入された追加情報を文脈解析エンジンが捨てられないというだけの理由で、文脈解析が失敗するのが分かる。

これらのオペレーティングシステムの文脈解析エンジンを変更することは、難しいし、不可能かもしれない。そして通常のエンドユーザにはできそうにない。 $\Omega$ TPを使って文脈解析を行うことによって、オペレーティングシステムが文脈解析を提供していたとしても、どんなユーザでも自分の文書を完全に制御することができるのである。

話はこれだけではない。 $\Omega$ TPが導入され広く使われだして数年がたち、 $\Omega$ TPのシンタックスは強力ではあるものの、複雑なタスクには充分でないことが明らかになった。実際の例を見よう。最初の節で見たように、タイ語の単語は、空白や他の視覚的な区切りなしに、くっついて入力される。しかし $\Omega$ は、行が単語の境界以外で切れたときにハイフンを補うために、単語の切れ目を知らなくてはならない。タイ語の語の切れ目を見つけることは簡単な仕事ではない。タイの言語学者がこれを行うソフトウェアをC言語で書いている。このソフトウェアを $\Omega$ TPのシンタックスで書き直すことは恐ろしい時間の浪費となるだろう。

このため、非常に最近、「外部」 $\Omega$ TPと呼ばれる新しい種類の $\Omega$ TPが導入された。「外部」 $\Omega$ TPは、他の $\Omega$ TPと同様、 $\Omega$ の中で用いられる。違いは書き方にある。実際のところ、それはオペレーティングシステム上の実行可能ファイルにすぎない。STDINとSTDOUTのデータストリームを使うどのような実行可能ファイルも、 $\Omega$ TPとして使うことができる。つまり、前述のCで書かれたタイ語の実行可能ファイルも、何の変更もなく $\Omega$ 中で使うことができる。ソースコードを持つ必要もない。したがって将来、「 $\Omega$ プラグイン」、つまり特定のプラットフォームのための、 $\Omega$ の機能を拡張して特定の組版の問題を解決するバイナリ形式の実行可能ファイルにお目にかかれるかも知れない。

タイ語だけでなく、すべての言語に当てはまる例も挙げる。パブリックドメインのスペルチェッカー spellの国際化バージョンは ispell という。著者らは

ispellを呼び、ΩTPとして使われた際には ispellで認識されない単語に色を付ける短いPerl スクリプトを書いた。これは、 $\TeX$ ファイルの最初の一行、「マクロコマンドパッケージ」をロードする部分を変えるだけで実行できる。このようにして、ユーザは  $\TeX$ ファイルを準備し、「ispell プラグイン」とともに走らせ、どの単語に色が付けられるかを調べ、間違いをなおすことができる。その後プラグイン無しにファイルを処理すれば、全体が適切な色で出力される。 $\TeX$ ファイルに ispellをかけることは大変やっかいなため、このやり方が役に立つ。文書と $\TeX$ コマンド、数式などを区別するのは困難なので、「 $\TeX$ だけが $\TeX$ プログラミング言語を理解できる」という言い回しは、ここにも適用できる。 $\TeX$ を効率的にスペルチェックする唯一の方法は、 $\TeX$ 、あるいは我々の場合Ωで行うことなのである。

この短い節はΩTPの2、3の例を示しただけである。Ωは、Ω内部に閉じていて書きやすく、プラットフォームに依らないΩTPと、外部の複雑で強力ではあるがプラットフォーム依存のΩTPという二通りのアプローチを続けていく。この両方を持つことが、どのような文脈でも効率の良い多言語組版を行う鍵となる。

## 5 Ωと自然言語処理

Ωは組版する文書にフィルタを選択的に適用することができる。（選択的とは、どんなフィルタをいつでも使ったり止めたりできるということである。たとえば言語ごと、構成要素ごとに違うフィルタを適用することができる。）この能力は、NLP（自然言語処理）の手法を用いて、組版の新たな地平を拓くこととなった。じっさい、文書の変形には特別な自然言語向けソフトウェアを必要とする場合があるだろう。その言語の単語のコーパスですむ場合も、本当の形態素解析を要する場合もあるけれど。

この章では、いくつかの違う言語で、外部のツールをΩの中から用いることによって効率の良い組版を行っている例のうち、比較的簡単なものを示す。

### 5.1 日本語順序付け

日本語の順序付け (Sorting) は、複数のレベルの情報が同時に使われるために大変複雑である。情報のレベルとは、書かれている文字や文字のまとまり（かなと漢字）とその読みである。両方とも重要であり、両方とも日本語の単語リストを並べる際には考慮される。文書がΩに与えられる際には、たいてい最初のレベルの情報、すなわちかなと漢字と句読点の列だけが与えられる。

日本語文字列照合順番, JIS X 4061-1996 Standard (see [?])によれば、日本語の単語は4つの違った方法で順序付けされる。

順序付けをする前に、文字列のトークンをいくつかのカテゴリに分類する必要がある。カテゴリとは、漢字、かな、ラテン文字、ギリシャ/キリル文字、数字、句読点、空白文字等である。(規格の§4.3と4.4を見て欲しい)。かなの順序付けについてはすべての方法で一致している。かなの順序付けは以下による。

(a)規格の§4.4.10.1の表に沿って長音記号ーはその前の母音と同じ値を取るという特別な規則が適用される。(カ+ー=カ+ア, など) (b)濁点、半濁点によって(は<ば<ぱ, etc.); (c)大きさによって(よ<><よ); (d)種類によって(ひらがな<かたかな).

(疑似) 数学的に言えば、キャラクタの部分集合  $K$  (漢字),  $k$  (かな),  $O$  (その他) を定義する。  $k$  と  $O$  は半順序集合である。

当然、残る問題は漢字の順序付けである。(つまり文脈に応じた  $K$  の半順序を見つけることである。) 4つの方法それぞれについて次に説明する。

1. 第一のやり方(規格の §5.1)では、漢字を JIS X 03092JIS X 0208 エンコーディングでの位置によって順序付ける。これは計算機にとってはやさしいが、JIS X 0208を暗記していない人間にとってはつらいものとなる。

数学的に言えば、 $\varepsilon: K \rightarrow \mathbb{N}$  が漢字に JIS X 0208 でのコードポイントを対応させる関数であるとき、 $K$  の順序は  $\mathbb{N}$  の順序の  $\varepsilon^{-1}$  として定義される。

2. 二つ目のやり方(規格の §5.2)は一步進んだ物であり、漢字をその読みによって順序付ける。より正確には、漢字はかなに変換され、前述のかな順序規則が適用される。二つの単語が同じ読みを持つ際には第一のやり方が用いられる。

数学的に言えば、 $\varphi: K \times \dots \times K \rightarrow k \times \dots \times k$  が漢字(あるいは後述する漢字のクラスタ)すべての読みを与える関数であり、 $\hat{\varphi}$  が文脈に依存した漢字(あるいは漢字のクラスタの)唯一の読みを与える関数であるとき、 $K$  の半順序は、 $k$  の順序の  $\hat{\varphi}^{-1}$  として与えられる。

この方法は、ある漢字がどのあたりに置かれるかを人間が読みから大体知ることができるという点で、前のものより優れている。問題は、読みがほんの少しだけ異なる同じ漢字を使った単語が、順序付け結果のリスト中では大変離れた位置に現れるかもしれないことである。

3. 三つ目のやり方(規格の §5.3.1)は最もリソースを要求する。まず第一に、単語を「照合要素(collation elements)」を構成するクラスタごとに分けなくてはならない。「照合要素」とは、個々の漢字の読みに分割できない読みである。最も単純なクラスタは一つの漢字である。しかしそれより大きなクラスタもある。たとえば今日 = きょう のように。(今日 = こんにちは二つの単漢字クラスタ 今日 = こん と 日 = にちからできた単語である。) 数学的には、 $K_1, \dots, K_n \in K$  であり  $\hat{\varphi}(K_1 \dots K_n) \notin \varphi(K_1) \dots \varphi(K_n)$  であるとき、 $K_1 \dots K_n$  は漢字クラスタである。

漢字やクラスタが明らかになったら、それぞれに代表的な読みを割り当てる。たとえば音読み一つと訓読み一つ、あるいは二つの(音韻的にも意味的にも)大きく異なる音読みと一つの訓といったように。数学的には、これは同値関係  $\mathcal{E}$  が  $\varphi(K_1 \dots K_n)$  (漢字あるいは漢字クラスタの可能な読みすべての集合)に適用されることを意味している。この割り当てのやり方は規格には述べられていない。こうして新しい関数  $\mathcal{E} \circ \hat{\varphi}$  が得られ、この関数は漢字クラスタを(同値類の代表元の)読み、文脈に依存して与える。 $K$  の半順序は  $k$  の順序の  $(\mathcal{E} \circ \hat{\varphi})^{-1}$  となる。

このやり方の困難な点は、網羅的で人間の直感に近い同値関係  $\mathcal{E}$  を見つけることにある。

4. 四つ目のやり方は(規格の §5.3.2) 二つ目と三つ目の妥協点ともいえる。漢字クラスタごとに、最初の読みを取る。もし二つの単語の最初の読みが同じ

であれば、次の漢字クラスタに移る。二番目のやり方との違いは、単語の生の読みではなくクラスタを対象としている点であり、三番目のやり方との違いは文脈によって明示的に与えられている読み（本当の読みとは非常に違っているかも知れない同値元の代表元ではなく）を用い、同値類を取る際に、読みの最初のかなを取るといふ、ユニークに定義された方法を用いている点である。

数学的に言えば、 $\hat{\varphi}$  を  $p_1: k^n \rightarrow k$  (文字列から最初のかなへの射) と合成し、 $k$  の順序に  $(\hat{\varphi} \circ p_1)^{-1}$  を適用したものを  $K$  の半順序とする。

順序付け方法を簡単に説明をしたところで、 $\Omega$  と既存のツールで何ができるかをみてみよう。考えられるツールは茶筌(Chasen)<sup>2</sup> (日本語形態素解析器) とかかし(Kakasi)<sup>3</sup> (漢字かな変換器。日本語 Altavista のような、ウェブ探索エンジンで用いられている) である。前者では、 $\varphi$  (漢字が漢字クラスタのすべての読みの集合) と  $\hat{\varphi}$  (文脈によって与えられる唯一の読み) の両方を得ることができる。後者のツールでは、 $\hat{\varphi}$  だけしかが得られないが、ずっと高速である。

かかしを用いると、追加のツール無しで、順序付けの二つ目のやり方を実現することができる。日本語の本の索引を作るには、索引項目となるすべての単語（とその文脈）にかかしを適用し、 $\text{\TeX}$  コマンド中にもとの語（漢字とかなを含む）とそのかかしによるかな表記を保持すればよい。

しかし三番目か四番目の方法が適用できる方がずっと望ましい。このためには「漢字クラスタの同定」という問題が解かれなくてはならない。我々はかかしと茶筌の開発者にこの機能を追加するよう依頼する予定である。それまでの間でも、クラスタが分割可能かどうかは、クラスタの読みを個々の漢字の読みの組み合わせとと比較することによってチェックすることができる。

この問題が解ければ、4番目の方法は簡単に実装することができる。3番目の方法の実装には、さらに一つ課題が残る。充分完全な辞書上での、同値類  $\mathcal{E}$  の実装を見つける（あるいは、作り出す）課題である。これは数千時間の人手と高度な言語学的な作業を要しかねない、壮大な仕事である。

一方、順序付ける項目数がとても多くはないのであれば（言い換えれば、作ろうとしている本の索引項目が東京の電話帳より少ないのであれば）、結局のところ4番目の方法で多分充分である。

結局のところ、まだ述べていない最大の問題は茶筌やかかしによる間違った読みの割合の多さである。これが文脈情報が充分に与えられないためにしろ、単に日本語の本来的な複雑さのためにしろ。

## 5.2 他の言語での自然言語処理の利用

自然言語処理ツールが日本語以外の組版機能を強化する他の例を3つ見てみよう。これらの例は多様性と、それぞれの言語を使わない人にも理解しやすいという点で選ばれた。

<sup>2</sup>日本語形態素解析システム茶筌開発部 <http://cl.aist-nara.ac.jp/lab/nlt/chasen/>

<sup>3</sup>漢字かな(ローマ字)変換プログラム <http://kakasi.namazu.org/index.html>. ja

### 5.2.1 ドイツ語の複合語

日本語同様、ドイツ語も複数の言葉をつなぎ合わせて複合語を作る。たとえば *Berg* = 山, *besteigen* = 登る, *Bergsteigen* → 山登り。これは大変よく起こる現象である。興味深い点は、複合語の要素は独立した語として持っていた特徴の多くを持ち続けるということである。特に、複合語は独立した語と同じように発音される。例としてとても珍しい場合を考えてみよう。 *Wachstube* という語は “wax-tube” と “wahn-shtube” とも発音される。はじめの場合、複合語を構成するもとの要素は *Wachs* (wax) と *Tube* (tube) であり、二つ目の場合には *Wach* (awake) と *Stube* (room) が要素である。

何世紀もの間、ドイツの組版ではこういった場合を区別する大変効率的な方法を持っていた。組版は言語的現象の視覚的表現であり、これはまさに二つの言語的実体（語）を分離するという言語的現象である。この現象を自然に表現する方法は、印刷した語を視覚的に分離することである。これには二つのやり方がある。

1. 「語末形の s」を構成要素の最後に用いる。
2. 要素間にはどんな合字（リガチャ）も使わない。

この二つのルールの背景を説明しておこう。1943年まで、ドイツ語は基本的には *broken script*、もっとも典型的にはジャーマンフラクチュールで印刷されてきた。Broken script には、二通りの文字 s がある。末尾の短い s (s) と語頭と語中に現れる長い s (ſ) である。語の終りには末尾用の s が用いられ *ſaus* のようになるが、語頭や語中では、語中形の s が使われて、*ſehen, alſo* となる。この規則は複合語の要素にも同様に適用され、次のようになる。 *ſaus + Arbeit = ſausarbeit* (*ſaufarbeit* という形は間違っている)。もちろん、前述の例 (*Wachstube*) は *ſſachstube* と *ſſachtube* の二通りに書くことができる。両方とも正しい書き方であるが、それぞれ全く違った意味を持つ異なる単語である。

Broken script には、最小限の合字 *ſz* (sz), *ch* (ch), *ck* (ck), *ff* (ff), *fi* (fi), *fl* (fl), *ft* (ft), *ll* (ll), *ſi* (ſi), *ſſ* (ſſ), *ſt* (ſt), *tt* (tt), *tz* (tz) も含まれている。これらの合字は複合語の要素間に現れることはない。たとえば *trotzdem* (*trotz* は一語なので、*tz* は合字になる) という語はあるが、*Brotzange* の *t* と *z* は合字にならない (*t* は *Brot* という語に、*z* は *zange* という語に属するため)。

今日の人々は、「broken script は読みにくい」と言う。しかし実際は、複合語を視覚化するうまい仕組みのある broken script は、ドイツ語を読むには理想的であった。

1943年に broken script が禁止されて以来 (詳細は [?] を参照されたい)、長い s とほとんどの合字はドイツ語には使われていない。ほとんどであってすべてではないことに注意しておこう。実際 *fi*, *fl*, *ch*, *ck* の合字はドイツ語の高品質の組版にはまだ使われている。そして同じ規則がまだ適用されている。“*Auflage*” や “*aufimmer*” と書き、同時に “*fliegen*” や “*finden*” と書かなくてはならないのである。( *ch* や *ck* の合字はこの規則に影響されない。というのもこれらは組版上の合字というより、文法的な二重字だから。)

このような場合普通そうであるように、この種の情報は生の原稿には含まれない。伝統的な本製作モデルである「著者-編集者-印刷者」モデルでは、合字は印刷者によって入れられ、編集者によって修正され得る。今日では、平均的な計算機ユーザは自分自身をこのモデルでの「著者」だと考え、計算機や、より正確には組版ソフトウェアに「編集者」と「印刷者」の役目を果たさせたいと考えている。

これは、 $\Omega$ が $\Omega$ TPを用いて行う典型的な変換である。唯一の問題は、複合語とその要素は自明ではなく、自然言語処理の仕事であるという点である。実際、それぞれの語の形態素解析を要する。ドイツ語の場合は、新しい複合語が自由に作られることから特に形態素解析が必要となる。新しい複合語ももちろん誤りではなく、辞書に存在しなかったとしても見つけ出さなくてはならないのだから。

我々はDMM (*Deutsche Malaga-Morphologie*) システムを用いることで、 $\Omega$ ([?]<sup>4</sup>参照)でこれを成し遂げた。*Malaga* (= *Merely a Left-Associative-Grammar Application*)とは、形態素解析用プログラミング言語であり、Oliver Lorenz<sup>5</sup>によって作られた。このシステムに複合語を分解してもらうことができるので、自動的に短い *s* を使ったり、broken script での組版の際には構成要素間の合字を分けたり、ロマンスク립トでの組版で *fi* や *fl* の合字を分離することができる。DMM は 100 正しいわけではないので、別の $\Omega$ TPを追加し、この規則に影響される可能性のあるすべての *s* の字(*n*語の単語中の 2, ..., *n* - 2 番目のもの)をユーザがすばやくスクリーン上でチェックできるように組版している。

### 5.2.2 ギリシャ語のグレーブ/アキュートアクセントのチェック

ギリシャでは1954年と1980年に言語の改革が行われ、アクセントの数をそれぞれ3つから2つへ、2つから1つへと減らした。しかし高品質の本はいまだに伝統的な方法で組まれている。つまり3つのアクセント(と2つの気音符、分音符と下付きイオタ)を使って。しかし、1954年と1980年の改革には根本的な違いがある。1954年の改革はグレーブアクセントを消し去るものであり、グレーブは組版にのみ使われていた。(今もつかわれている。)学校ではグレーブアクセントについて説明されることすらない。手書きのギリシャ語ではグレーブアクセントは使わない。タイプライターにグレーブアクセントのキーはない。実際のところそれは(著者-編集者-印刷者モデルの)印刷者の時点で挿入されていたのである。

グレーブアクセントを置く規則は実際には次のようにとても単純なものである。単語の後に終止符、セミコロン、カンマ、感嘆符、疑問符、コロンが続かない場合に、単語の最後の音節のアキュートをグレーブに置き換える。たとえば *τὸ παῖδι* と書く際には、最初の語はグレーブアクセントになる。これはアクセントが最後の音節にあり、句読点が後ろに付いていないからである。二つ目の単語は、その直後に終止符があるため、アキュートアクセントのままである。

この規則にはいくつか例外がある。第一に、以下の場合には句読点が後ろに続かなくてもアキュートアクセントのままになる。

- 前接の副詞(前の語の接尾辞のように振る舞う)があとに続いているとき。*τὸ δικό μου, ἡ μέθοδός μου* (二つ目の例ではアクセントが二つついている。これは最後から二番目の音節にアクセントが付いた単語のあとに前接の語が続くときに起こる。)
- ダブルクォート、かっこ、かぎかっこ、長いダッシュ、省略記号は、アクセント付けの際には無視される。単語にこれらのうちのひとつがついたものの後ろに空白が来れば、その単語にはグレーブアクセントがつく。もし、後ろに終止符、セミコロン、カンマ、感嘆符、疑問符、コロンがつけば、その単

<sup>4</sup><http://www.fluxus-virus.com/en/research.html>で入手可能

<sup>5</sup><http://www.linguistik.uni-erlangen.de/örlorenz/DMM/DMM.html>

語はアキュートアクセントのままである。: τὸ «μικρὸ» παιδί となるが ἦταν «μικρός»<sup>6</sup>であるなど。

- 単語 τί ('that' or 'what') は常にアキュートアクセントを取る。
- 単語 ποιός, ποιά, ποιό, γιατίは、疑問文のなかではアキュートアクセントとなる。ποιός τὸ εἶπε αὐτό; ('who said that?')となるが、ξέρω ποιός τὸ εἶπε ('I know who said it')である。

第二に、単語の後にカンマが続いてもアキュートアクセントがグレーブに変わる、大変例外的な場合がある。

- 単語 γιατί ('why', 'because') は「なぜなら(because)」という意味の時だけ、カンマの前でもグレーブアクセントになる。αὐτὸ ἐγίνε γιατί, ἐνῶ τὸ ἤξερε, δὲν ἤθελε νὰ τὸ πιστέψει.

このような例外はあるにしても、アキュートアクセントをグレーブに変換する基本的な基準は、前述の句読点が直後にあることである。ΩTPでこれを実装するのは簡単である。Ioannis Kanellos (École nationale supérieure des télécommunications de Bretagne<sup>6</sup>, France)による仕事に加え、我々はいくつかの例外をカバーし、ユーザによる確認が必要な部分を赤で表示するようにΩTPを拡張した。これはギリシャ語の文書を入力する際にはとても役に立つ。というのも、ギリシャ人はグレーブアクセントを入力することに慣れておらず、間違いはしばしば起こるためである。

我々のプロジェクトには、ギリシャ語のアクセントと気音符すべてに関するスペルチェッカーを作ることにも含まれている。しかしこれにはとても多くの言語学的リソース、すなわち形態的、統語的、時には意味的解析が必要となるだろう。現在、Institute for Language and Speech Processing<sup>7</sup> などいくつかのチームが、monotonic Greek (アクセント1つだけで気音符なし)用のスペルチェッカーを研究している。我々の知る限りでは、まだ、完全なアクセント付きのギリシャ語について作業しているところはない。

### 5.2.3 タイ語の文の分離

タイ語は、他の東南アジア言語、そして日本語同様、視覚的には文中の語を分離しない。しかし、日本語と異なり、行を送る際にはどこで行を切っても良いわけではない。行は単語の切れ目で切れていることが好まれる。単語が長すぎる場合には、単語の音節の間で切ることが好まれる。後者の場合にはハイフンが(他の多くの言語同様)用いられるが、容易に想像がつくように、前者の場合にハイフンは使われない。

Opt<sub>TEX</sub>は語を音節に分けることが大变得意だし、そのアルゴリズムはもちろんタイ語にも適用可能である。しかしよりよいやり方は単語の間で切ることであり、これは<sub>TEX</sub>でもΩでもほとんど不可能である。これには二つの理由がある。第一に二つのハイフネーションアルゴリズム(単語用と単語内の音節用)を同時に適用することはできないため、第二に、このような単語の同定は単純な仕事ではなく自然言語処理ツールを必要とするためである。

<sup>6</sup><http://http://www-iasc.enst-bretagne.fr>

<sup>7</sup><http://www.ilsp.gr/>, アテネ

幸い、バンコクの *Center for Research in Computational Linguistics*<sup>8</sup> の一員が、どんなタイ語文書の文字列も単語に分けることができる、タイ語分節エンジンを開発している。例をみてみよう。文字列

## บริการตัดคำสำหรับโฮมเพภาษาไทย

は次のように分節される。

## บริการ|ตัด|คำ|สำหรับ|โฮมเพ|ภาษา|ไทย

この外部バイナリは、Ωの中では外部ΩTPとして使うことができる。

しかしタイ語の組版の品質に悪影響を与えずにタイ語単語の切り出しを行うためには、もう一つ別の問題を解決しなくてはならない。タイ語の表記法の特徴には、左側に大きくはみ出す文字があるという点がある。たとえばโ, โย, โยのように。これらの文字が他の文字に続く場合には、充分な量のカーニングが必要である。nl は nl よりずっと良く見える。

しかし単語を分節するための追加の情報を(TeXコマンドという形で)文字の間にいれることによって、カーニングができなくなる。どうしたらΩはTeXコマンドを挟んだ二つの文字をカーニングできるのだろうか？

この問題は、*ghost*と呼ばれる新しいTeXコマンドを導入することで解決した。ある文字の *ghost* とは、幅ゼロの見えないグリフであり、カーニングに関しては元の文字と同じように振る舞う。*ghost* は、カーニングされるべき文字の間に別のオブジェクトが挿入されてカーニングができないときに用いられる。たとえば、単語のうち一つの文字だけに色をつけるとしたら、色をつけるコマンドがカーニングを妨害することにもなってしまう。(色つきでない) *ghost* を色を付ける文字の左右に挿入することによって、Ωは周囲の文字と *ghost* をカーニングし、視覚的に正しい結果が得られる。

タイ語の場合には、

```
<letter 1><word-break><letter 2>
```

を

```
<letter 1><ghost of letter 2><word-break><letter 2>
```

に置き換えることによって、組版上最適な結果を得ることができる。

## 6 Conclusion

世界中の言語を組版することは偉大な挑戦である。たくさんの異なる問題を同定し、研究し、解決を見つけ、実装しなくてはならない。これらの解決にはΩ固有の機能や、ΩTPとして実装できる追加のアルゴリズムや、外部ΩTPsとして使うことのできる自然言語処理ツールが必要となるだろう。実際のところ、Ωは著者-編集者-印刷者 書籍処理モデルの計算機上での実装でありこの連鎖の各リンクが持っている知識とノウハウをモデル化しなくてはならない。著者の言語知識と効率的文書構

<sup>8</sup><http://www.cyberc.com/crcl/software/segment.htm>

成と、編集者の組版規則や文書の構造を組版に反映させる能力、表現の一貫性と、印刷者の高度な組版、文書の組版への変換のモデルなのである。

Ωプロジェクトはこれらの問題を解決し続ける。Ωを拡張し、新たなリソース(フォント, ΩTP, L<sup>A</sup>T<sub>E</sub>Xスタイルファイル)を作り、自然言語処理ツールのような既存のリソースを調整することによって。我々は、ΩがT<sub>E</sub>Xの後継者として、高品質の文書組版のための標準的なパブリックドメインツールとなることを望んでいる。