第9章　テキスト処理

長い長い道のりだった計算機科学基礎もようやく最終回を迎えることができた。ここで振り返って見ると、皆様が計算機を使うのはほとんどが文字 (テキスト) を読んだり書いたりするためであって、数値の計算などめったにしないはずである。(でしょう?) 最終回にあたってはそのテキスト処理を取りあげ、特に修論やレポートを書くのに欠かせないツールである文書整形系について学んで頂くことにする。

9.1　計算機とテキスト処理

9.1.1　文字の処理の歴史

そもそも、計算機が最初に作られたときはその目的は文字通り「計算する」ことであって、扱うのは数値が中心であった。しかし、文字も符合化すればビットの列で表せ、計算機に取り込めることはご存知の通りである。ただし最初のころは符合化というのは入出力装置が CPU と文字をやりとりするのに符合化コードを使うから必要、という程度のものであった。というわけで、最初のころは文字を扱う機能を中心に据えたプログラム言語というのは「異端」とされ「文字列処理言語」だとか「記号処理言語」だとか特別な名前が奉られてきたものである。

しかし、事務計算の場合などでは帳票を出力する際には計算した数値といっしょに顧客名や商品名なども打ち出したい、ということで事務処理言語 (もちろん COBOL がそうである) には中心に文字を扱う機能がいろいろつけ加えられ、計算機で文字を扱うことは当たり前になってきた。それでも、77 規格になる前の Fortran では変数に文字型がなかったくらいである。

しかし結局、人間が扱う情報のうち数値だけ、というのは比較的少なく、一方文字は文章 (テキスト) の形をとればどんな情報でも扱えるから、その処理の比重が高まるのは当然のことだった。というわけで現在では汎用の手続き言語では文字が自由に扱えることは当たり前になっている。その辺の歩みを図 9.1 に示した。

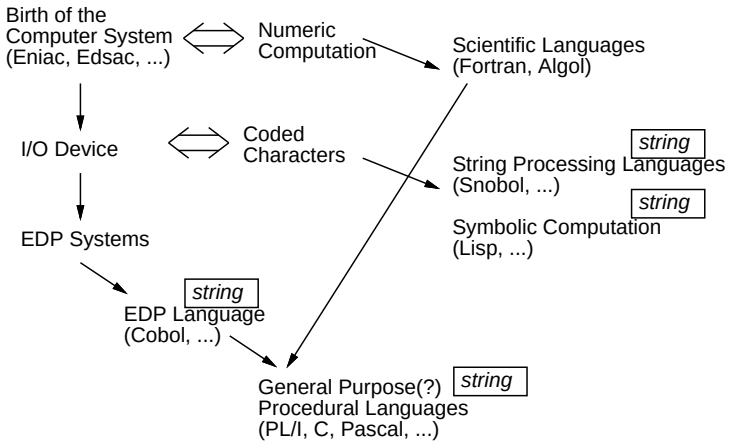

図 9.1: プログラム言語と文字列処理の歩み

9.1.2　テキスト処理

では、テキストを扱うプログラムとして、具体的にどんなものがあるだろう? 応用ベースで考えてみよう。

- テキストエディタ: 何はともあれテキストファイルを計算機に打ち込んだり編集したりするのにはエディタのお世話になるわけであり、従ってエディタは立派なテキスト処理ソフトである。
- スペルチェッカ: 英文の場合特に綴り間違いを直すのは計算機の力を借りるととても楽である。日本語スペルチェッカというのはあんまり聞いたことがない。
- 整形系: 文書をきれいに右そろえしたり、タイトルなどを大きなフォントで打ち出したり、表をきれいに作成したりするための処理を行なうソフトのこと。
- レイアウトソフト: 複数の記事などをページの上で配置して「組版」を行なうソフトのこと。
- 参照データベース: 文献その他の参照を管理してくれるような文書作成向け特殊目的データベース。

これらはいいかえれば「人間がきれいな文書を作り出すための手助けをする」ものと考えてよい。ホワイトカラーの仕事というものは外見上は大部分が文書をつくり出すことで成り立っているので、文書作成を手助けするソフトの必要性は昔から高いわけである。

ところで、この中におなじみ「ワープロ」が出てこなかった、と思った方もおいでかと思う。実はワープロというのはエディタと整形系を一体化した

ソフトのこと、と思えば良い (高級なのだともっと機能がついていたりすることもあるが)。つまり 1 つのソフトで両方できるからきれいな文書の作成がこれで一通りこなせる、というわけである。

これにさらに各種のフォントが自在に使えてレイアウトソフトの機能まで備わると「DTP」などと呼ばれることもある。DTP(Desktop Publishing) とは要するに、町の印刷やさんに頼んでいた程度の美しさの文書を自前で手軽にできるようにする、という意味だと考える。なお、「卓上」ではなくプロの印刷 (新聞や書籍など) を扱うシステムのことは CTS(Computer Typesetting System、電算写植) と呼ぶ。DTP のための整形系については後でまた述べる。

9.1.3 テキスト処理のための言語機能

さて、数値に対しては四則演算や三角関数など各種の「演算」があるのと同様、テキスト (文字列) に対しても各種の「演算」があって、特に文字列処理言語の場合はそれが効率よく扱えるような言語機構が備わっている。具体的な「演算」としては次のようなものが挙げられよう:

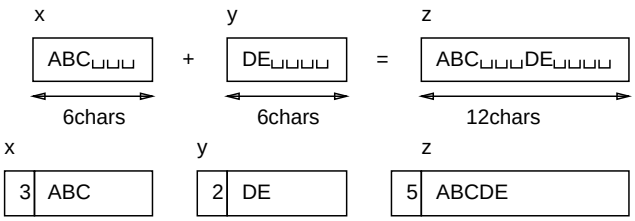

図 9.2: 固定長文字列と可変長文字列

- 文字列と文字列を連結する。
- 文字列の中から部分文字列を取り出す。
- 文字列の中に、指定した部分文字列があるかどうか探索する。
- パターンマッチング
- 各文字を規則に従って別の文字に置き換える (mapping)。
- 探索やマッチで見つかった部分文字列を他の文字列に置き換える。

これらは一見 (パターンマッチを除いては) 別に難しくないように思えるかも知れないが、実際にやると色々問題が起きる。例えば文字列の連結であるが、これが意味を持つためにはまず文字列の長さが可変長である必要がある (なぜか?)。ちなみに、可変長というのは、ある変数に入る文字列の長さが可変である、という程度の意味である。例えば標準 Pascal や Fortran や COBOL の文字列は可変長ではなく固定長である (図 9.2)。

次に可変長であると、その長さは連結していくとどんどん長くなる、という問題もある。適当な上限を設けるとしても、それぞれの変数にその上限まで入れられるだけ領域を用意する、というのはあまりよい方法でない (なぜか?)。で、結局柔軟に文字列を扱うためには文字列をヒープ (ある程度の大きさのメモリ領域を確保しておき、実行時にそこから必要なだけ切り取って来て利用すること) に入れてごみ集め (不要になった切り取り部分を回収して来てあとで再利用できるようにすること) をするようなシステムが必要になってくる (図 9.3)。

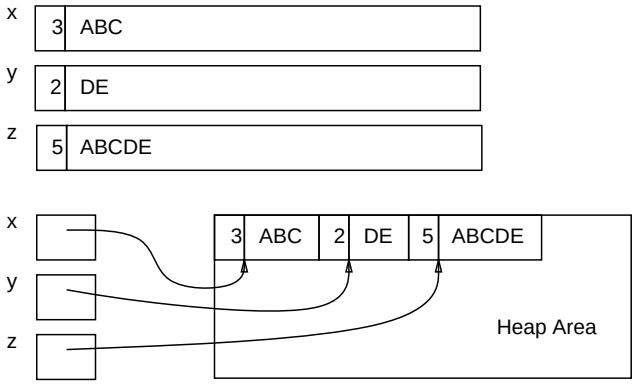

図 9.3: 変数領域方式とヒープ方式

このような方法が性能の面でも有利な場合が多い。例えば部分文字列の取りだしにしても、変数領域に直接入れる方法だと毎回コピーが必要だがヒープを使う場合には「どの場所」ということだけ覚えておけば済むように工夫することもできる。

次に、部分文字列の探索だが、これは素直に考えれば図 9.4 上のように探したい文字列を 1 番目、2 番目、... の位置に置いてみてそれであてはまるかどうかを調べればよい。しかし、この素朴な方法だと最悪の場合「探される文字列の長さ×探す文字列の長さ」回の比較が必要になり、ひどく効率が悪くなることがある (図 9.4 下)。これを避けるため様々な「高速文字列探索アルゴリズム」が研究されている。パターンマッチの場合も同様である。

とはいえ、これらの機能は文字列処理言語の場合その処理系内部に実現済みなわけであり、利用者はその言語でプログラムを書けば良いだけだから話は簡単である。実は、Awk には文字列の連結も検索やパターンマッチも備わっていて (ですね?)、その処理系の中ではヒープを活用している。だから Awk は立派な文字列処理言語である。

なお、文字列処理は常に文字列処理言語で書かなければならない、というものではなく、例えば C など普通の言語でももちろん可能である。ただしその場合には上で述べたような問題を自分で何とかする必要が生じて結構大変

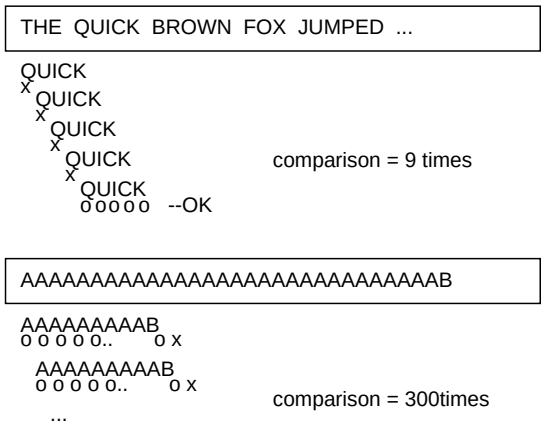

図 9.4: 2 次マッチングによる探索

である。[1] 一方、よく行なうパターンの文字列処理については既にならった `sed`, `grep`, `tr`, `sort`, `uniq` などを駆使すればわざわざプログラムを書かなくても済む、という面もある。

9.1.4 テキストエディタによるテキスト処理

テキストエディタはテキストファイルを打ち込んだり修正したりするのに欠かせない道具だ、という話はこれまでにも繰り返ししてきた。とすると、テキストエディタも立派なテキスト処理ツールである…というのは、いくらか抵抗がありますね? つまり、基本的にテキストエディタでテキストを処理するのにはいちいち人間がそう指示しないといけないので、「自動的に」という意味あいが薄いからいまいちなわけである。

しかし、実際にはテキストエディタでも「自動的な」処理をかなり行なうことができる。ここではおなじみ mule を例にとってみてみよう。

パラグラフのつめ合わせ

Unix のテキストファイルは行の長さがあまり長いと困るので、適当なところで改行しなければならない。しかし人間にとっていちいち決まった位置で改行を打ち込むのは結構面倒である。そこで、打ち込む時には適当な場所で改行しておいて、1 パラグラフ打ち終わったところで

`ESC q` ← `fill-paragraph` 指令

[1] 実際にはわざわざ書かなくてもライブラリルーチンとして用意されている機能で大体用が足りる。

を実行するとそのパラグラフを揃えてくれる。または、いくつかのパラグラフをまとめて揃えたい時には揃えたい最初の行の先頭で

```
Ctrl-Space  ← set-mark
```

を実行したあと、揃えたい最後の行に行って

```
ESC X fill-region RET
```

を実行するとまとめて揃えてくれる。なお、プログラム例や実行例など追い込みの場所でないころを揃えてしまうと悲惨! なので注意。その時はあわてず、

```
^X u   ← undo
```

をやるともとに戻る。

置き換え

ある文字列を別の文字列で置き換える、というのは sed でできるのだったが、mule の中でもできる。それには

```
ESC X replace-string RET 文字列 1 RET 文字列 2 RET
```

とやればよい。文字列 1、2 として漢字まじりも使えるので便利である (通常とおり打ち込む時に^\でローマ字モードに入れる)。

なお、対話型エディタならではの機能として、置き換えながらそのつど確認を求める指令 query-replace もある。起動方法は上と同じだが、置き換え文字列がある度にそこを表示してどうするか聞いて来る。利用者は

```
y、Space   -- 置き換える
n          -- ここは置き換えずに先へ行く
q          -- 中止する
!          -- 以後は一括して置き換える
```

から選んで応答する。さらに、sed のように正規表現で置き換えを指定するものとして query-replace-regexp もある (使い方は同じで、ただし文字列 1 としてパターンを指定できる)。

キーボードマクロ

ここまでに説明したのはどれも予め用意された動作であって、利用者が動作をプログラムするというものではなかった。しかし、ファイルを編集していて、規則的な変更のため同じ指令パターンを繰り返し打ち込むことはよくある。例えば、

```
1  久野
2  寺野
3  大木
4  ...
```

というファイルがあったとして、これを隣り合う 2 行どうしで入れ換えたいものとする。それには、1 行目の先頭で^K^K をやって消し (これで 2 行目が先頭になる)、^N で 3 行目の先頭に行き、^Y でさっきの 1 行目をその前に差し込む、というのを繰り返し実行することになる。

そこで、いちいち同じ打鍵列を繰り返す代りに 1 行目の先頭で

```
^X ( ^K ^K ^N ^Y ^X )
```

と打つと、^X(と^X) で囲まれた部分が、キーボードマクロとして取り込まれる。あとは、^Xe を打つと取り込まれた打鍵列があたかも今打ち込まれたかのように実行される。これを繰り返して行なうには、

```
^U 回数 ^X e
```

により回数指定してやればよい。

プログラマブルエディタ

キーボードマクロは確かにプログラムみたいだが、しかし決まったことの繰り返ししかできない。そこで、もっとちゃんとしたプログラム言語をエディタの内部に用意して、これで好きなプログラムが書けるようにする、というのを考えた人がいる。これを「プログラマブルエディタ」と呼ぶ。実は mule は内部に Lisp 言語処理系を持つプログラマブルエディタである。例を見ていただこう。

```
(defun unindent ()      ←コマンド (関数) の名前。
  (interactive)         ←対話的コマンドである。
  (beginning-of-line)   ←行の先頭に行く。
  (while (not (char-equal (following-char) 10)) ←※ 1
    (while (char-equal (following-char) 32) (delete-char 1))
    (next-line 1)))
```

※ 1 のところは、「カーソルの所の文字が文字コード 10(改行) でない間ループする」を意味する。ループの中身は次行以下で、「カーソルの所の文字が文字コード 32(空白) である限りそれを消す」「次の行へむ」を意味する。従って、このプログラムは空白行が現れるまで次々に行の先頭の空白を取り除く。

以上の内容を持つファイルを unindent.el という名前のファイルに用意し、mule の中から

```
ESC X load-flie RET unindent.el RET
```

で読み込む。すると、あとは編集中に行頭の空白を取りたいところで

```
ESC X unindent RET
```

とやれば上のプログラムが動いてくれる。

実は mule のカスタマイズはこのような各種プログラムを読み込むことで自由に行なえる。mule は実行開始直後に各ユーザのホームディレクトリにある `.mule` という名前のファイルを読み込むので、この中に上の内容を入れておけば、いちいち load しなくても unindent コマンドが使えるようになる。

9.2　日本語の入力

9.2.1　日本語入力の難しさと歴史

さて、我々日本人にとっては、テキストといえばその多くが「日本語」ということになる。しかしアルファベット 26 文字程度ならキーボードに直接その数だけキーを用意すれば足りるが、日本語では多数の漢字が必要なため、そう簡単には済まない。

実際、1980 年代中頃まで、計算機で漢字まじりのちゃんとした日本語を扱うというのは夢のまた夢というか、相当困難なことだった。そのため、せめてカタカナだけでも使えるように、ということが行なわれた。カタカナだけだったら、キーボードでも何とか入力できるし、プリンタの活字も用意できる。この結果今でもカタカナ出力の計算機伝票は結構残っている。

やがて出力の方はレーザープリンタやドットマトリックスプリンタのように細かい点の集まりで文字を表すものが現れて漢字でも問題なく打てるようになってきたが、どうやって入力するかという方は結構後まで試行錯誤が行なわれた:

- 特殊キーボード: 昔からあった「漢字テレタイプ」のように漢字入力専用のキーボードを使用する。訓練が必要な上特別な装置がいるので敷居が高くて普及しなかった。
- 漢字タイプ方式: 大きな板にあらゆる漢字が並べてあって、ペンで拾っていく。わかりやすいので一時流行ったが、ひどくのろくてつかれるのですたれた。
- コード方式: 各漢字ごとに符号を定めて暗記し、普通のキーボードで符号の羅列を打ち込んでいく。速いけれど訓練がつらいので普及しなかった。
- バッチ変換: ローマ字やかなで打ち込み、それを一括して漢字変換する。変換間違いを直すのに手間取るので普及しなかった。

- かな漢字変換: ローマ字やかなで読みを打ち込んで、その場で漢字に変換していく。結局、これが一番普及して勝者となった。

9.2.2 かな漢字変換の原理

皆様はかな漢字変換でさんざん日本語を入力しているので、それがどんなものであるかは改めて説明するまでもない。しかし、それではその処理内容が具体的にはどうなっているか、ということになると、ふだんかな漢字変換をしながらよほど注意深く観察している人でないと的確に答えるのは難しいのではないだろうか? ここでかな漢字変換の処理過程について、順を追って見てみよう。

1. まず、利用者はキーボードからローマ字表記を入力する。入力された文字は順にかな表記に変換されていく。ローマ字かな変換は、こういうローマ字表記はこういうかな、という対応表を調べながら順に置き換えて行くことで行なえる。ただし、ローマ字表記も人によって/ システムによって違いがあるのでいろいろと問題。また、「za」と打ったとたんにただちに「ざ」にしたり、そこで DEL を押すとまた「z」だけの状態に戻ったりする処理は結構面倒である。なお、かなキーボードで打ち込む人はこの処理がいらない。
2. ある程度打ち込んだところで「変換」を指示すると、まずひらがな列を適当な位置で区切る。例えば「きょうはよいてんきです」だと「きょうは/よい/てんき/です」のように。
3. 区切られたそれぞれの部分について、辞書を引いて適切と思われる候補単語の漢字表記に置き換える。このとき、助詞の処理を行ない (たとえば辞書に「今日」しかなくても「きょうは」を「今日は」にする)、また活用語尾の変化にも対応する (辞書に「返す」しかなくても「返せば」「返さない」などをうまく変換する) 必要がある。そして、日本語には同音異語 (「花」と「鼻」など) が多いので、どれが適切かを選ぶのも大変である。だいたいは、一番最近使われた候補とか、単語の利用頻度などを参考に決める。

2'. そして実は、辞書を引いてみないとひらがな列をどこで区切るのがいいかは分からない。従って、辞書を引いてうまくあてはまる語があったらそこで区切る、というのを左から順にやっていくのが普通である。しかし先の方へ行ってみたら実は別の区切り方が正しいと分かったりするので、その時はやり直すひつようがある。(たとえば「きょうはながとどいた」は「きょう/はなが/とどいた」になるべきですね。)

4. そうやって頑張っても区切り方や選んだ候補が正しくないことは常にある。そこで、利用者に「どこの区切りをどう直せ」「この語は別の候補

に取り換える」などの指示を仰いで直さなければならない。ソフトも大変だが、利用者にとってもいちいち表示されている候補を見ながら指示を出すのはおおきな負担である。

5. 利用者の修正が終わって「確定」するとようやく、最終的な日本語文字列をプログラムに渡すことができる。

こうしてみると、とても大変でしょう? しかも、どのプログラムでもキーボードから日本語を入力するところではすべて、本来の処理に加えてこの処理をしなければならない。こうして見ると単に「a」というキーを打てば「a」という字が入ってそれでおしまいで済む英語圏がうらやましい、という気分になりませんか?

9.2.3　かな漢字変換のソフトウェア構造

前節で説明したように複雑な機能を持つソフトウェアを、どのような形で日本語入力を必要とする任意のプログラムに接続させるかというのは難しい設計問題である。ここでは単純なものから初めて代表的な方法を説明する。

直接組み込み方式

かな漢字変換機能のすべてを、「1 行日本語を入力する」というサブルーチンとその下請け、孫請けの集まりとして用意し、日本語入力を必要とするプログラムに一緒に組み込んでおく (図 9.5)。

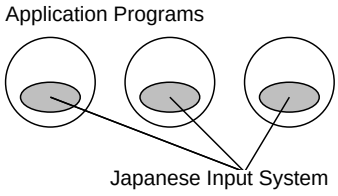

図 9.5: 直接組み込み方式

この方法は単純明解ではあるが、次のような欠点がある。

- 個々のプログラムを日本語入力用に改造してやる必要がある。
- 個々のプログラムに巨大なサブルーチン群を組み込むので各プログラムが巨大になる。
- かな漢字変換の新版が出たりしてもすでにプログラムに組み込まれたものは新しいのに取り換えられないし、利用者の好みに応じて変換方式を選ぶのも難しい。

- プログラムごとに組み込んだものが違うと、かな漢字変換の方式がバラバラになってしまう。

フロントエンド方式

上のような欠点を解消するために、PC の世界で広まったのが漢字フロントエンドプロセッサ (FEP) 方式である。これは、OS のキーボード処理部分に独立したプログラムを「差し込める」仕掛けを用意し、そこにかな漢字変換プログラムを差し込んで変換処理を行なわせるものである (図 9.6 のようにアプリケーションの「手前に」差し込んで使うことから「フロントエンド」と呼ばれるわけである)。

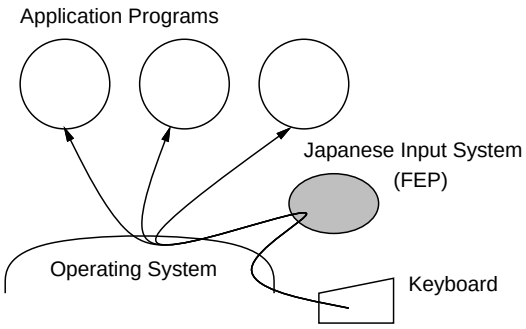

図 9.6: 日本語入力フロントエンド方式

このやり方であれば、プログラムそのものは直さなくても済むし、FEP も好みのものを選んだり新版に入れ換えたりすることが自由にできる。ただし、FEP にも次のような弱点がある。

- FEP の制御用に一部のキーを横取りしてしまうためそのキーがアプリケーションプログラムからは使えなくなったりする。
- FEP とやりとりする (候補選択などの) モードとアプリケーションとやりとりするモードを切替えるのがわずらわしい。
- ひらがな列や漢字候補などを表示したり応答するために表示画面の一部を使ってしまう。

ただし日本で販売されている PC ソフトはだいたいがこれらの問題をできるだけよけるように設計されてはいる。

サーバ方式

PC 用の FEP は変換用の辞書をハードディスクやフロッピーに置いているので、あるマシンで単語登録しても別のマシンでは全然使えないし、逆に誰か

が自分の入れておいた単語を削除してしまったりといったトラブルも起きる。

そこで、Unix システムのようにネットワークで接続された複数システムを運用する場合には、図 9.7 のようにネットワーク上に 1 つ、かな漢字変換サーバを置いて、そこでまとめて変換を受け持つことが多い。こうすれば、どのマシンで単語登録をしてもそれはサーバ内の辞書に入るので、あとで別のマシンで日本語入力するときに問題なく参照できる。また、サーバ上ではユーザごとに辞書を管理するので、他人と定義がまざってしまうこともない。

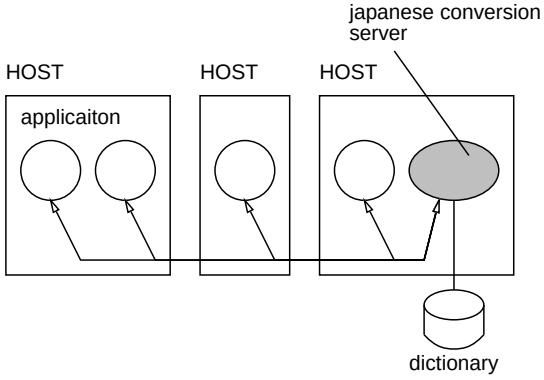

図 9.7: 漢字変換サーバ方式

この方式を取った場合、漢字変換処理の大部分はサーバが受け持つので、日本語入力ライブラリの方はローマ字かな変換と候補表示などの機能程度で済み簡単である。そのため直接組み込み方式と組み合わせることも多く行なわれる。サーバのみバージョンアップすることも問題ない。実は我々が mule で使っているのはこの方式である。(かな漢字変換サーバとして canna サーバと呼ばれるものを使っている)。

一方、サーバ方式とフロントエンドを組み合わせることもできる。例えば、smb で

```
% canuum mule-nw
```

とやるとフロントエンド canuum が動き出し、その中で mule が動く。canuum はキーボードからの入力を横取りして canna サーバと通信しながら漢字変換を行ない、確定するとその結果を mule に引き渡す。なお、canuum では漢字 on/off(フロントエンドの接続と切り離し) には`^O` キーを使っている。

入力マネージャ方式

フロントエンド方式は端末や PC の画面の一部を横取りして候補表示に使うので、たとえば X のソフトのような GUI アプリケーションには適用でき

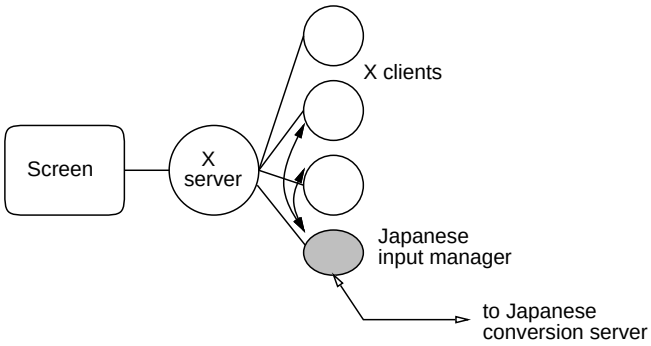

図 9.8: 漢字入力マネージャ方式

ない。そこで、これらのシステムでは入力マネージャと呼ばれるプログラム (図 9.8) を使用する。

まず、漢字入力を使うプログラムは入力マネージャを呼び出したり、確定した結果を受け取るための通信機能を備えるようにする (だいたいはそれ用のライブラリを組み込むことで対応する)。そして、入力マネージャはプログラムから呼び出されると画面上に自分の窓を開き、キー入力をすべてその窓で受け取って候補表示などの入力作業を行なう。確定すると、入力マネージャは日本語文字列をプログラムに送信する。

我々のところでは、kinput2 と呼ばれる入力マネージャが用意されている。これを動かす場合には (処理速度の都合から)smb の窓で単に

```
% kinput2 &
```

といってバックグラウンドで起動する。それだけだと何の変化もなさそうだが、この状態で端末窓の上で Shift-Space を押すと kinput2 の窓が現れ (図 9.9)、そこでローマ字入力してスペースキーを押すと変換が始まり、さらにスペースを押すと次々と候補が現れる (何回か変換してまだ正しい候補が見つからないと全候補一覧の窓が開く — 図 9.10)。正しいものが現れたら RET で確定する。kinput2 の窓を閉じたい (入力マネージャを切り離したい) 場合には、再度 Shift-Space を押す。

ここで見たように、kinput2 の窓は端末窓のカーソル付近に現れ、候補は本体の窓に重なって見える (over the spot 入力) ので、視線を大きく動かさなくても変換が行なえるという利点がある。kterm 以外にも、tcl/tk の入力欄、k2d の漢字テキスト欄なども kinput2 に対応しているが、これらの場合には kinput2 の窓はやや離れた位置にできる (off the spot 入力)。その様子を図 9.11 に示す。なお、これらのツールの場合は kinput2 への接続は Shift-Space ではなく (mule と同様の) `Ctrl-\`による。しかし kinput2 の切り離しの方はさっきと同じ Shift-Space なので注意されたい。

図 9.9: kterm の窓で kinput2 による入力

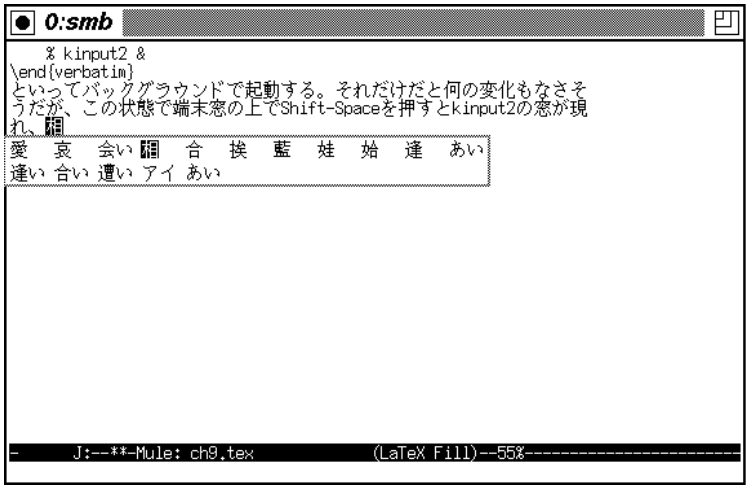

図 9.10: kinput2 による候補表示

このように、かな漢字変換自体がとても複雑である上に、それをシステムに組み込むのも非常込み入った設計を必要とするので、日本語を使用する我々は計算機利用において (ソフトが簡単に日本語を使えるようにできないとか、使えたとしても操作が難しいなど) 多くの不利な点を負っていたわけである。しかし今後の計算機のユーザインタフェースを考えてみると、ペン入力/タッチ入力など利用者とのやりとりを必要とする入力方式が主流になる可能性があり、その時日本の培ってきた入力技術の蓄積がプラスに作用するといいなあ…と思う訳なのであった。

9.2.4 かな漢字変換のカスタマイズ

かな漢字変換はとても複雑なシステムなので、それを自分の好みに合うようにカスタマイズするのはなかなか難しい。しかし、だいたいの利用者は

- かな漢字変換辞書に自分の必要とする単語を登録する
- ローマ字表記を自分の好みに合わせて変更する

の 2 つをやればそれなりに済むはずである。単語登録は準備コースでやったはずだから、ここではローマ字表記の変更をやってみる。

まず、標準のローマ字かな変換テーブルは

```
/usr/local/canna/lib/dic/default.kpdef
```

にあるので、これを自分のホームディレクトリにコピーしてくる。その内容は見ればあきらかで、

```
1       l
2       2
...
a       あ
i       い
u       う
e       え
o       お
ka      か
ki      き
...
```

のように、ローマ字とかなの表記が並んだものになっている。ここで、例えば自分は大文字の「N」でただちに「ん」が出るようにしたい、という場合にはどこでも好きなところに

```
N       ん
```

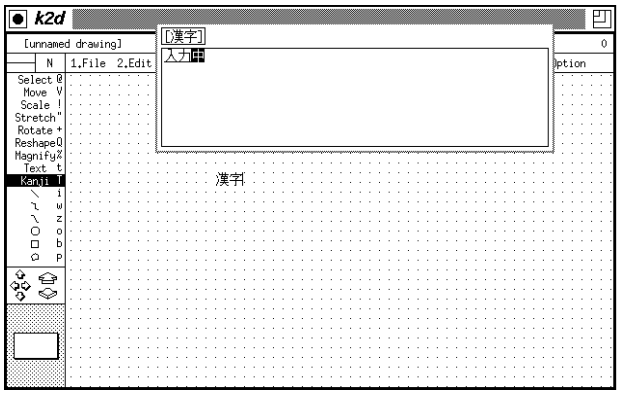

図 9.11: k2d と kinput2 の組み合わせ

を追加してファイルに書き出す。その後で、

```
mkromdic  default.kpdef
```

とやると、このファイルを変換したファイル `default.kp` ができる。我々のところでは Canna はホームディレクトリに `default.kp` があるとそれがシステムのものに優先されるようになっているので、これで mule でも canuum でも kinput2 でも「N」と打つと「ん」が入るように変更されたはずである。

なお、このような設定ファイルが「.」で始まらないのは気持ちが悪いと思う人もいるかも知れない。その場合には、ホームディレクトリの `.canna` というファイルの中に

```
(setq romkana-table "default.kp")
```

というところがあるので、このファイル名とさっきのファイルをともに「.」つきのファイル名に直せばよい。なお、`.canna` ではその他にもいくつかのカスタマイズが指定できる。

9.3　文書整形系と DTP

9.3.1　見たまま方式とマークアップ方式

先にも述べたように、現在では計算機の利用目的のうち、文書を作成する、というのが重要な位置を占めるようになっている。その場合、作成する文書はテキストエディタで打ち込んだ地の文のような味気ないものではなく、様々な大きさやフォントの文字を含み、図なども豊富な美しいものである必要がある。

そのような文書を組み立てるソフトウェアの方式として、次の 3 種類 (後の 2 つはいっしょと考える人もいる) がある:

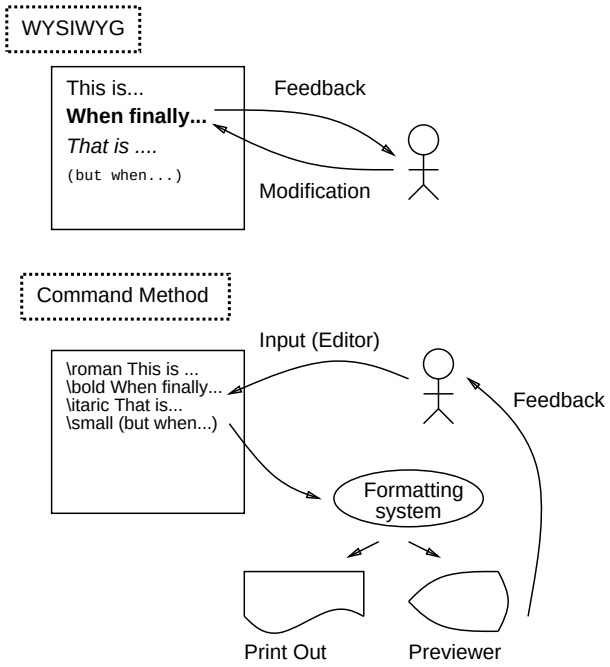

図 9.12: WYSIWYG 方式と指令/意味づけ方式

- 見たまま方式、または WYSIWYG(What You See Is What You Get) 方式。これは画面の上に本当に紙に打つのに使うようなフォント (といっても解像度が違うから美しさは劣る) を用いて、紙に打つのと同じレイアウトで表示するものである。そこで指令やマウス操作を用いて配置やフォントを変更したり、文章を直したりすると、その結果は直ちに画面に反映される。直接的で分かりやすいといえる。JStar, MacWrite, EgWord などが代表的である。ワープロもまあこちらの仲間といえる。
- 指令方式。これはエディタを用いて作成するファイルの中に出力する文章に混ぜて「ここからこういうフォント」「ここで改ページ」などの「指令」を入れておき、これに従ってフォーマッタソフトが整形を実行するものである。指令を使うことにより非常に細かい制御まで行なうことができる一方、指令を覚えるのが大変で、フォーマッタを掛けて紙に出すまでどういう出力になるか見えないという欠点がある (紙に出す代わりに画面で確認するプレビュアソフトを使えることは多い)。troff, TeX などが代表である。
- 意味づけ方式。これも文章に指令を混ぜるのは同じだが、ただし「どう整形する」という指令を混ぜるのではなく「ここは章の切れ目」「ここからここまでは箇条書き」など文章の意味にそったマークを入れてお

く。指令方式より使いやすく、また文書のスタイルを変更しても文書の意味で指定してあるから影響されないという利点を持つ。Scribe, latex などがこれである。

後 2 つの方式は文書の中に出力される文章に加えて各種の指示をいれることから「マークアップ方式」と呼ばれる。

さて、皆様にとって敷居が低いのは当然ワープロソフトに近い見たまま方式だろうが、ここではマークアップ方式である latex を体験していただく。(見たまま方式をやったことがない人は E408、E437 の Macintosh でマックライト II や EgWord を体験して見られることを強くおすすめする。)

またぞろ、マークアップ言語なんか覚えたくないというご意見が目にみえるようである…が、見たまま方式はそんなにいいだろうか?

- 見たまま方式は、分かりやすい代りに大量の文書を扱うのは苦しいものがある (CPU の負担、ソフトの操作性の両方から)。例えばマックライトで 50 ページのレポートはかなりつらいが、latex なら 500 ページの本 1 冊でも楽勝である。
- 見たまま方式はある 1 つの体裁に合わせて文書を作ってしまうので、あとから体裁を変更するのは不可能に近い (そのために最近はスタイルシート機能というのが使われるが、そうやって段々使い方が面倒になっていく)。マークアップなら、整形時のパラメタを変えるだけで多様な体裁に対応できる。
- 見たまま方式はある特定のツールに依存するので、ツール間でテキストを流通しにくい。テキストファイルにしてしまうと、文字飾りなどはすべて失われてしまう。マークアップはもともと全部テキストファイルなのでテキストの流通が容易である。
- 見たまま方式ので文字飾りなどの体裁を変更するにはマウスとメニューと格闘してかなりの手間が掛かり、大量にやるのは苦痛である。マークアップではテキストと一緒に指令を入れるだけなので、一旦指令を覚えてしまえば労力は小さい。
- マークアップ方式では章節番号の自動わりふり、目次、索引などの自動作成など各種のサポートを提供してくれる。

9.4　jlatex 入門

本節では意味付け方式の文書整形システムの一つである latex の日本語版である jlatex について実例中心で一通り解説することを通じて、意味付け方式 (と広義の指令方式) というのはどんなものかの感触を持っていただきたいと考える。より詳しくは「楽々latex」などの参考書を見ていただきたい。[2]

[2]野寺、楽々latex、共立出版、1990、247p.

9.4.1 文書の基本構造

まずともかく、jlatex の文書に最低限必要な基本構造の例を見ていただこう。次のようなものは、一つの完結した latex 文書である (内容も読んでね)。

```
\documentstyle[epsf,11pt]{jarticle}
\begin{document}
ここに示すように、jlatex の文書はまずこの文書がどんなスタイルの文書である
かを宣言する部分から始まる。スタイルの例としては「記事 (jarticle)」、

「本 (jbook)」、「報告 (jreport)」などがある。ここでは一番簡便な
jarticle を例に使用している。これに加えて字の大きさ、図形の取り込み
機能などをオプションとして指定できる。

続いて「文書開始」の宣言があり、この中に文書の本体が入る。この
部分には様々な記述が可能だが、一番簡単にはここに示すように段落ごとに 1 行
あけて次々に文章を書いて行くだけでも地の文が普通にできる。つまり、
「特に指定がない」ならば「地の文」である。

あとは文書の本体が終わったら最後に必ず「文書終了」の宣言がある。
最低限必要なのはたったこれだけである。
\end{document}
```

これを jlatex に掛けて打ち出したものを付録につけておく。ところで、ここで少し補足しておくと、まず宣言 (指令) は

```
\指令名 [オプション指定]{パラメタ}
```

のような形をしている。ここでオプション指定がないときは [...] はなく、またパラメタがないときはさらに{...}も不要である。ということは\はそのままでは文書に含められない。そういう特別な記号がいくつか予約されているのでちょっと不便ではある。

9.4.2 表題、章、節

さて、いきなり本文が始まって延々と続くだけではあんまりだから、最低限表題と章建てくらいはやりたいと思われるのも当然である。その場合の例を次に示す。

```
\documentstyle[epsf,11pt]{jarticle}
\begin{document}

\title{あなたがつけた表題}
\author{書いた人の名前}
\maketitle

\section{表題について}
表題をつけるには title、author など必要な事項を複数指定した後最後に
maketitle というとそれらの情報をもとに表題が生成される。その際指定
されなかった事項は出力されないかまたは適当なものが自動生成される
(たとえば日付を指定しないと整形した日付が入る。)

\section{章建てについて}
ここにあるように section 指令を使って各節の始まり、およびその表題を
指定する。節の中でさらに分ける場合には section というのも使え、
さらに subsection というのまで可能である。ちなみに jbook/jreport スタイル
では section の上位に chapter があるが jarticle では section からである。
ところで、節番号 etc. は自動的に番号付けされるのに注意。

\section{より細かいことは}
より細かいことは、参考書を参照してください。

\end{document}
```

9.4.3　いくつかの便利な環境

表題と章建てができればこれだけで結構普通の文書は書けてしまうはずであるが、しかし全部地の文ではめりはりがないことおびただしい。それにプログラム例など行単位でできているものまできれいに詰め合わされてしまうのでは困る。このように部分的にスタイルが違う部分を指定するのに

```
\begin{環境名}
....
....
\end{環境名}
```

というのを使う。よく使う環境について、以下で説明する。

まず、verbatim(そのまま) というのは文字通り入力をそのまま整形せずに埋め込む。例えば

```
\begin{verbatim}
This is a pen.
That is a dog.
\end{verbatim}
```

は、つぎのような出力になる。

```
This is a pen.
That is a dog.
```

従ってこれはプログラム例などを入れるのに適しているし、その他よく作り方が分からないスタイルはすべて手で整形してこれで用意してしまっても良い。ところで、わざわざ別の行にするのでなく、文章のなかに一部「そのまま」を埋め込みたい場合もある。そのような時には verbatim の類似品で `\verb|....|` というのを使えば良い。これで縦棒にはさまれた部分がそのまま出力できる。ちなみに中に縦棒を含めたい時は、両端に縦棒以外の適当な記号を使えば良い。

つぎに、itemize(箇条書き) について説明する。これの場合は環境のなかに複数「`\\item ....`」というのがいくつか並んだ格好になっていて、その一つずつが箇条書きの 1 項目になる、というものである。例えば

```
\begin{itemize}
\item あるふぁはギリシャ文字の一番目です。
\item ベータはギリシャ文字の二番目です。
\item ガンマはギリシャ文字の三番目です。
\end{itemize}
```

は、つぎのような出力になる。

- あるふぁはギリシャ文字の一番目です。
- ベータはギリシャ文字の二番目です。
- ガンマはギリシャ文字の三番目です。

さて、この itemize を enumerate(数え上げ) に変更すると、出力の際 1, 2, 3... と自動的に番号付けされるようになる。入力はほとんどまったく同じだから出力のみ示そう。

1. あるふぁはギリシャ文字の一番目です。
2. ベータはギリシャ文字の二番目です。

3. ガンマはギリシャ文字の三番目です。

点や番号でなくタイトルをつけたい場合には description(記述) を使う。これは

```
\begin{description}
\item[あるふぁ] これはギリシャ文字の一番目です。
\item[ベータ] これはギリシャ文字の二番目です。
\item[ガンマ] これはギリシャ文字の三番目です。
\end{description}
```

のように、各タイトルを [] の中に指定する。結果は次のようになる。

あるふぁ これはギリシャ文字の一番目です。

ベータ これはギリシャ文字の二番目です。

ガンマ これはギリシャ文字の三番目です。

この他にもいくつか環境があるが、参考文献などを参照されたい。

9.4.4　脚注と図

脚注の作り方はとても簡単で、単に好きなところに`\footnote{.....}`というのはさんでおけばそれがページの下に集められて脚注になり、そこへの参照番号は自動的につけられる。

図のほうはそもそも何で図を描くか、というところから様々なやり方がある。実は jlatex の出力は後で述べるツールで PostScript に変換されるので、そのなかに PS で描かれた図を差し込んで一緒に出力することができる。PS で図を用意するには、前回やった k2d で描くとか、前前回やった xwd の結果を xwd2ps で変換したりすればよい。latex の方では、図を入れたい箇所に次のものをはさめばよい。

```
\begin{figure}[htbp]
\begin{center}
\includegraphics[width=12cm]{my.ps}
\caption{図の表題}
\end{center}
\end{figure}
```

つまり、はさみたいファイルの名前、その大きさ、タイトルの 3 つを指定する。ただし図はちょうど文章のその場所に入るとは限らず、ページの残りが少なかったりすると次のページに回されたりする。図の大きさは幅などの変わりに `scale=0.7` のようにもとの大きさに対する倍率でも指定できる。

9.4.5 数式

実は、TeX 属のフォーマッタはもともとは数式を美しく打ち出したい、という目的のもとに開発されたものなので、数式機能にはひどく充実したものがある。で、凝り出すと大変だからここでは簡単に説明する。まず、数式を文中にはさむときは$で囲むたとえば

と書くと`$x^2 - a_0$`のような具合になる。

と書くと $x^2 - a_0$ のような具合になる。ここで出てきたように、肩字は^、添字は_で表せる。その他、数学に出てくる記号は

```
\equiv \partial \subseq \bigcap
```

などのように書くと $\equiv \partial \subset \bigcap$ のように出てくる。また、文中ではなく独立した行にしたければ$の代わりに$$ではさめばよい。たとえば

とかくと`$$f(t) = \sum_{j=1}^m a_j \exp i\lambda_j t$$`になる。

とかくと

$$f(t) = \sum_{j=1}^{m} a_j \exp i\lambda_j t$$

になる。このように、肩字や添字のグループ化には{}を使う (なぜ () でないかは分かりますね?) より詳しくは参考文献を参照のこと。

9.4.6 動かし方、その他

このほか jlatex にはまだまだ様々な機能があるのだが、とりあえずはここまでに述べたもので十分以上だと思う。使いこなすには、まず自分で文書を作成し打ち出して見ることが重要である。というわけで、最後になったが打ち出し方を説明しておく。

jlatex を動かすには、まず上で述べたような形式のファイルを .tex で終わる名前のファイル名に格納する。例えば sam.tex だとしよう。次にすることは

```
jlatex sam.tex
```

によりこのファイルを整形することである。入力があると途中で?が出て処理が止まることがあるが、そういう場合にはとりあえず「x[ret]」を打ち込んで実行を終わらせる。そして、エラーメッセージをよく見て間違っているところを直し、再度走らせればいいわけである。成功すると .dvi で終わる名前のファイル、この場合だと sam.dvi ができているはずである。

これでいきなり紙に打ってもいいのだが、普通はまず画面でできばえを確認する。それには X-Window を動かした状態で

```
xdvi sam.dvi
```

という。Xdvi が起動すると縮尺やページめくりを指定するボタンを伴った窓が現れるので、これらのボタンをマウスで操作して各ページが意図どおりかどうか確認する。

ただし、xdvi だと PS で描かれた図のところは真っ白で何も表示されない。図も見たい場合には

```
div2ps -r sam.dvi >sam.ps
```

のようにして dvi ファイルから PS ファイルを生成し、あとは前回やったように

```
gs sam.ps
```

により ghostscript で見ることができる。ただし、gs は 1 ページ目から順番にしか見られないのであんまり使いでがよくない。そこで、ページや拡大率を自由に操作できるよう gs に「皮」をかぶせたツール ghostview を使う方がよいかも知れない (ただし、いくぶん遅い)。使い方は

```
ghostview sam.ps
```

である。最後に、プリンタに出す場合には PS ファイルを単に

```
lpr -Plw sam.ps
```

で打ち出せばよいわけである。漢字フォント転送のため 1 ページ目をうち始めるまで 2〜3 分掛かるので注意。

9.5　演習

9-1. mule でキーボードマクロと lisp 関数を作成使用してみて、その違いについて述べよ。

9-2. 通常の mule でのかな漢字変換に加えて、canuum と kinput2 による漢字入力を試してみて、その比較を行なえ。

9-3. Canna のローマ字変換規則を自分の好みに合わせて調整して、使い心地を報告せよ。どのように調整したかも述べること。

最終回はニュースへの投稿ではなく、レポートを作成して頂きます。その内容を以下に述べます。

- レポート作成には jlatex を使用すること。
- レポートの名称は「計算機科学基礎'96 最終レポート」です。

- 著者欄に学籍番号と氏名を記入して下さい。
- 第 1 節として、latex を使ってレポートを書いてみてどうだったかという報告を述べてください。(もちろん、この節は最後に書くことになるでしょう。)
- 前回までの各章の演習問題すべての中から、まだ報告していないものを合計 3 課題 (1 つの課題が選択方式になっている場合にはその中の 1 つでよい) 選び、やってください。
- 第 9 章については 9-1〜9-3 から 1 つ選んでやってください。
- 以上 4 課題についてそれぞれを 1 セクションとして latex 文書を作成してください。つまりこれが 2〜5 節となります。
- 各セクションには必ず 1 個以上の図が入ること。図は k2d で描くか、X の画面例であれば「`xwd -screen -frame | xwd2ps >`出力ファイル」により生成した PS ファイルを取り込むのが簡単でしょう。
- 各セクションには必ず 1 箇所以上の箇条書き (itemize、enumerate、description のいずれか) と verbatim 環境が含まれること。
- 最後にまとめの節を設け、本科目を通じて学んだこと、学びたかったけど学べなかったこと、その他感想、反省、改善提案など自由に記してください。

〆切は、7 月 6 日 (土曜日) 一杯 (厳守) とします。久野のメールボックスまで提出してください。なお、この日は事務指定の成績報告期限より遅いので 2 学期始めの成績リストには載りません。あらかじめご了承ください。自分の成績が知りたい人は久野まで直接聞きにきてください。

計算機講義室に'94 年と'95 年の先輩たちの提出したものがありますので参考にしてください (ただし'95 年は時間数が少なく課題が少ないのでレポートも薄いです)。