ＲＥＸＸ入門

広田　豊彦*

１．はじめに

ＲＥＸＸとは、再構造化拡張実行プログラム言語（ REstructured eXtended e Xecutor language ）のことである。そしてＲＥＸＸは、仮想計算機／システム・プロダクト（ＶＭ／ＳＰ）システム・プロダクト解釈プログラム（以後、単に解釈プログラムと呼ぶ）によって、解釈・実行される。しかし、本解説では、言語および解釈プログラムをどちらも区別することなく、ＲＥＸＸと呼ぶことにする。ふつうは文脈から、ＲＥＸＸという語が、言語を指すのか、あるいは解釈プログラムを指すのか、容易に区別がつく。（あるいは区別しなくても差し支えないであろう。）

すでにＣＭＳに慣れ親しんでいるユーザであれば、ＲＥＸＸについてもおよそのことは知っているかもしれない。ＣＭＳで使えるコマンドの中にはＲＥＸＸで書かれているものも多数ある。そのようなコマンドは中身をエディタで覗いてみることができる。多少プログラミングに関する知識があれば、たとえＲＥＸＸのことを知らなかったとしても、およそのことはわかるだろう。それを適当にいじって自分専用のコマンドを作ったユーザもいることと思う。

ＲＥＸＸでプログラムを書く目的は、大きく２通りに分けることができる。すなわち、

１．端末から入力する一連のコマンドを１つにまとめる。

２．何らかのデータ処理を行う。

である。はじめてＲＥＸＸを使おうとするときには、最初の目的で使うことが多いであろうが、ＲＥＸＸに慣れ親しんでくると、２番目の目的にも有用であることがわかってくるだろう。特に文字処理を主な内容とするようなデータ処理であれば、かなり高度な処理をＲＥＸＸプログラムで書くことができる。ＲＥＸＸは、他のプログラミング言語、たとえばＰＬ／Ｉなどと違い、ソースプログラムが直接、解釈・実行される。そのために同じ程度の規模のプログラムであれば、ＲＥＸＸの方がはるかに容易に開発できる。このことがＲＥＸＸの最大のセールス・ポイントとなっている。

しかし、そのためにＲＥＸＸは、その文法の解説だけで１冊のマニュアルにな

---

* 情報工学部知能情報工学科、３月まで情報科学センター

る。実際のところＲＥＸＸに関しては、次の２つのマニュアルがある。

・ＶＭ／ＳＰシステムプロダクト解釈プログラム使用者の手引き

N:SC24-5238

・ＶＭ／ＳＰシステムプロダクト解釈プログラム解説書　　N:SC24-5239

「使用者の手引き」は、ＲＥＸＸについて基礎から学ぶためのテキストになっている。例題や演習も多数含まれている。これらを実際にこなせば、ＲＥＸＸの一通りの機能を身に付けることができる。また、ＲＥＸＸの特定の機能について知りたいときには、それに関する部分だけを読んでもよい。一方、「解説書」の方は、ＲＥＸＸの命令などについて、厳密に説明している。ＲＥＸＸをより高度に使いこなしたいとき、あるいは特定の命令の機能や文法を確認したいときに参照するマニュアルである。

本解説は、「使用者の手引き」と同じような形式で、はじめてＲＥＸＸを学ぼうとする人を主な対象として書いている。「使用者の手引き」は２５０ページを越えるぶ厚いマニュアルになっており、これを手元に置いてじっくり勉強できる人は限られているであろう。そこで本解説では、上で述べたＲＥＸＸの１番目の目的に焦点を絞って、簡単な使い方を説明している。

ＲＥＸＸを本格的に使いこなしてデータ処理をするには、やはり「使用者の手引き」などで勉強する必要がある。それでも本解説の内容を理解できていれば、あとは目的に応じて、「使用者の手引き」の中の必要な項目だけを抜き出して読むことができるであろう。

２．ＲＥＸＸの基本

まずは、ＲＥＸＸでプログラムを書いてみよう。プログラムを書くためには、もちろん計算機にログオンしなければならない。そしてエディタを使う必要がある。エディタは何を使ってもよいが、ＲＥＸＸのプログラムを書くときには、システム・プロダクト編集プログラム（いわゆるＸＥＤＩＴ）を使うのがふつうである。いずれにしてもこの資料では、ログオンの方法や、エディタの使い方などは説明しないので、別の資料で勉強してもらいたい。

というわけで、次に示すＲＥＸＸのプログラムを、ファイル名 HELLO、ファイルタイプ EXEC で作ってみよう。ファイルモードは A を指定するのがふつうである。この話がよくわからない人は、この資料を読む前に、ＶＭ／ＳＰ　ＣＭＳの基本的な使い方を学んでもらいたい。

```
/* Example 2.1: A conversation */
say "Hello! What's your name?"
```

```
pull who
if who = "" then say "Hello stranger"
else say "Hello" who
```

このプログラムを実行するには、ＣＭＳの READY 状態で、ファイル名 HELLO を入力する。プログラムの実行例を次に示す。下線部はユーザが端末から入力する箇所である。そのほかはシステムから出力される。

```
R; T=0.02/0.05 09:55:20
hello
Hello! What's your name?
Hiroshi
Hello HIROSHI
R; T=0.01/0.01 10:01:39
```

最初の行は、システムからの READY メッセージである。もしかすると、「R;」ではなくて、「Ready;」かもしれない。

２行目で、端末から「hello」を入力している。大文字で「HELLO」と入力してもかまわない。「hello」が入力されると、ＣＭＳは「HELLO EXEC」というファイルを捜す。そのファイルが見つかると、ＣＭＳはファイルの１行目を調べる。この場合、１行目は「/*」で始まり、「*/」で終っている。そこでＣＭＳは解釈プログラムを呼び出して、ファイル「HELLO EXEC」を実行させる。Pascal やＣで書いたプログラムのように、コンパイルやリンクなどの処理をする必要はない。

「HELLO EXEC」が実行されると、３行目のメッセージが出力されて、入力待ちになる。

４行目で、端末から「hiroshi」を入力している。ここはどんな文字列を入力してもかまわない。（ただし、日本語は受け付けてくれないかもしれない。）

５行目では、「Hello」に続いて、４行目で入力された文字列がそのまま出力されている。ただし、入力が小文字であっても大文字に変換されて出力される。しかし、「Hello」の方は大文字へは変換されていない。

「HELLO EXEC」の実行が終って、６行目には再びＣＭＳの READY メッセージが出力されている。

もしもうまくいかないときには、もう１度エディタを呼び出して、ファイル「HELLO EXEC」の中を調べてみてもらいたい。たぶんどこかに入力ミスがあるはずだ。それを修正してもう１度実行してみる。それでもだめなときには、だれかに相談してみるのがよいだろう。おそらくあなたには見つけられなかったエラーを見つけてくれるだろう。

それでは次に、「HELLO EXEC」の中身について説明しよう。もう１度「HELLO EXEC」の中身を書いておく。

```
/* Example 2.1: A conversation */
say "Hello! What's your name?"
pull who
if who = "" then say "Hello stranger"
else say "Hello" who
```

各行の説明は以下の通りである。

1 : /* A conversation */

/* と */ で囲んだものは注釈である。ＲＥＸＸのプログラムの１行目は必ず注釈でなければならない。注釈の中身は何でもかまわない。実は、ＣＭＳが HELLO というコマンドを受け取ったとき、ファイル・タイプを調べて、どのように処理するかを決定するのだが、EXEC というファイル・タイプは処理を一意に決定することができない。ファイル・タイプが EXEC のときには、中身がＲＥＸＸのプログラムであるとは限らず、ＥＸＥＣ（という言語）あるいはＥＸＥＣ２のプログラムかもしれないのである。そこでＣＭＳは、ファイルの１行目が注釈であるときには、ＲＥＸＸのプログラムであると解釈して、解釈プログラムを呼び出してくれることになっている。もちろん、１行目以外に任意の位置に注釈を入れることができる。

2 : say "Hello! What's your name?"

say は文字列を端末に表示する命令である。引用符（ ' または " ）で囲まれた文字列はそのまま出力される。引用符で囲んでいない文字列は変数名として解釈される。変数名については下で説明している。

3 : pull who

pull はプログラム・スタックから文字列を取り出す命令である。プログラム・スタックが空のときには、端末に入力要求を出す。プログラム・スタックの操作については４節で説明するが、特別な操作をしていないかぎりは、pull は端末から文字列を入力する命令であると考えてさしつかえない。who は変数名であり、端末から入力された文字列が who に代入される。

4 : if who = "" then say "Hello stranger"

if は条件を判定する命令であり、キーワード then や else と一緒に使われる。who = "" は条件を示している。この場合、文字列 who が ""（空文字列）であるときに条件が成立する。pull 命令に対して端末から文字列を入力するときに、何も入力せずに実行キーを押すと、who は空文字列になる。

then は、if 命令の条件が成立したときに実行すべき命令を指示するキーワードである。この場合には、say 命令が実行され、端末に文字列 "Hello stranger" が表示される。

5 : else say "Hello" who

else は if 命令の条件が成立しなかったときに実行すべき命令を指示するキーワードである。この場合には say 命令が実行される。

この say 命令では、文字列 "Hello" はそのまま端末に表示されるが、who の方はそのまま文字列 "who" が表示されるのではなくて、who に代入されている文字列が表示される。すなわち、さきほどの pull 命令のときに端末から入力した文字列がそのまま表示される。

if 命令においては、キーワード then は必須であるが、キーワード else の方はなくてもよい。else がないときには、if 命令の条件が成立しないと何も実行しない。

以上が「HELLO EXEC」の中身の説明であるが、それぞれの命令については、以下の節でもう少し詳しく説明する。

3．会話

ＲＥＸＸのプログラムでユーザと会話するには、say 命令と pull 命令を使う。この２つの命令は、すでに前節の例題で一通り説明したが、ここでさらにいくつかの点について補足する。

say 命令は、引用符で囲まれた文字列をそのまま端末に表示する。英小文字は小文字のままだし、途中の空白の数も変わらない。たとえば、

```
/* Example 3.1: cases and spaces (1) */
say "The     Lord     of     the     Rings"
```

を実行すると

```
The     Lord     of     the     Rings
```

が端末に表示される。もしも、引用符を書き忘れたらどうなるであろうか？　次の例を考えてみよう。

```
/* Example 3.2: cases and spaces (2) */
say  The     Lord     of     the     Rings
```

これを実行すると次のようになる。

```
THE LORD OF THE RINGS
```

最初の例との違いは、1) 小文字が大文字に変換されている、2) 語間の空白が1つだけになっている、の2点である。

2つの例の違いは、次のようなＲＥＸＸの解釈ルールによるものである。

1：引用符で囲まれた文字列は、そのまま扱われる。

2：そのほかの文字列は、キーワードや変数の羅列として解釈される。

Example 3.2 では2番目の解釈ルールが適用されて、The、Lord、of、the、Rings のそれぞれが変数名とみなされている。そしてこのときには、さらに次のルールが適用されることになる。

3：キーワードや変数名はすべて大文字に変換される。

つまり、Example 3.2 は次のように書き換えても全く同じだということになる。

```
/* Example 3.2b: cases and spaces (2) */
SAY THE LORD OF THE RINGS
```

ここで2節の Example 2.1 に話を戻してみたい。Example 2.1 の一部を抜き出すと、次のようになっていた。

```
/* Example 2.1: A conversion */
・・・（省略）
else say "Hello" who
```

これを実行したときには、端末に WHO が表示されたわけではなく、その前に端末から入力した文字列（たとえば Hiroshi ）が大文字に変換されて HIROSHI のように表示されていた。

それでは、どうして Example 3.2 では THE LOAD OF THE RINGS が表示されたのだろうか？　それは、Example 2.1 では pull 命令によって変数 who に値が代入されていたのに対して、Example 3.2 の4つの変数 THE、LORD、OF、RINGS には何の値も代入されていないからである。このとき、次のような解釈ルールが適用される。

4：変数に1度も値が代入されていないときには、その変数名自体が変数の値になる。

変数 THE の値は THE、変数 LORD の値は LORD、といったようになるわけである。しかし、変数に1度も値が代入されていないというのは、空文字列が代入されることとは違う。

たとえば、Example 2.1 を次のように変更してみよう。このプログラムを動かすとどうなるのか、各自で考えてみてもらいたい。

```
/* Example 3.3 (2.1b): A conversation */
```

```
say "Hello! What's your name?"
pull who
if whi = "" then say "Hello stranger"
else say "Hello" who
```

4行目の if 命令では、条件式は whi = "" となっている。ところが、変数 whi には1度も値が代入されていないので、変数 whi の値は WHI になり、条件式はいつも成立しない。そこで必ず else say "Hello" who が実行されることになる。一方、変数 who には、3行目の pull 命令によって何らかの文字列が代入されている。たとえば、端末から Hiroshi を入力したとすると、

```
Hello HIROSHI
```

と出力される。これは Example 2.1 と同じである。しかし、空行を入力したときには、

```
Hello
```

となり、Example 2.1 とは異なる。これは4行目の if 命令の条件式で who ではなくて、whi と書いたためである。

以上が変数の代入についての話である。1度も代入されていないときに、変数の値が変数名自体になることを除けば、プログラムで使う変数と同じように使える。

次に、端末からの入力を指示する pull 命令について説明する。

pull 命令のもっとも簡単な使い方は、Example 2.1 をはじめとしてこれまでのいろいろな例題に登場している。すなわち、

```
pull 変数名
```

という形式である。これで、端末から入力した文字列が指定された変数に代入されるわけである。ただし、これまでに示した実行例からもわかるように、たとえ端末から小文字で入力したとしても、すべて大文字に変換される。

実は pull 命令は

```
parse upper pull 変数名
```

の省略形だからである。3つのキーワード parse、upper、pull のうちで、upper が大文字への変換を指定している。この upper を省略して、

```
parse pull 変数名
```

とすれば、端末から入力した文字列は、大文字に変換されることなく、そのまま

変数に代入されることになる。たとえば Example 2.1 を書き直してみると次のようになる。

```
/* Example 3.4 (2.1c): A conversation */
say "Hello! What's your name?"
parse pull who
if who = "" then say "Hello stranger"
else say "Hello" who
```

これを実行すると次のようになる。

```
R; T=0.02/0.05 09:55:20
hello
Hello! What's your name?
Hiroshi
Hello Hiroshi
```

さて、parse 命令は pull 以外にもいろいろなキーワードと組み合わせて使われるわけであるが、この命令の機能は、その名の通り構文解析をすることである。（厳密に言えば字句解析かもしれないが） これまで Example 2.1 やその派生例題において、端末から Hiroshi という1語しか入力しなかったが、複数語を入力したらどうなるであろうか？

```
R; T=0.02/0.05 09:55:20
hello
Hello! What's your name?
Hiroshi Nakai
Hello Hiroshi Nakai
```

この場合には、変数 who に Hiroshi Nakai が代入される。それでは、端末から入力された2つの語を別々の変数に代入するにはどうすればよいかと言えば、単に parse 命令（あるいは pull 命令）に2つの変数を並べればよいのである。すなわち、

```
pull first last
```

のようにすると、変数 first に Hiroshi が、変数 last に Nakai が代入される。

このように単語を分離して、変数に代入するという機能が、parse 命令の基本である。なお、入力された単語の数が変数の数よりも少ないときには、余分の変数には空文字列が代入される。逆に入力された単語の数が多いときには、最後の

変数に残りのすべての単語が代入されることになる。parse 命令の機能の詳細についてはマニュアルを参照してもらいたい。

4．コマンド

ＲＥＸＸプログラムの中でＣＭＳやＣＰのコマンドを実行することができる。ふつうはプログラム中にコマンドをそのまま書けばよい。たとえば、次のプログラムは指定されたファイルのバックアップを作成する。

```
/* Example 4.1: Backup */
say "Enter filename and file type"
pull fn ft
copy fn ft a fn backup a "(" rep
```

４行目の copy はＲＥＸＸの命令ではないので、ＣＭＳあるいはＣＰのコマンドと解釈されて、実行される。基本的には、copy コマンドを端末から入力した場合と同じである。

変数 fn と ft には、端末から入力されたファイル名とファイルタイプが代入されているので、copy コマンドには、fn と ft ではなくて、入力されたファイル名とファイルタイプが渡される。a には何も代入されていないので、そのままドライブ名として a が渡される。"(" はＲＥＸＸの演算子としても使われるので、それと区別するために、引用符で囲んでいる。最後の rep は a と同じく何も代入されていないので、そのまま copy コマンドに渡される。

以上述べたように、ＲＥＸＸ命令でないものはＣＭＳ（あるいはＣＰ）コマンドと解釈され、何も代入されていない変数はそのままの文字列が渡される。しかし、ＣＭＳコマンドを間違いなくコマンドとして実行させるには、引用符で囲んでおくほうがよい。また、変数についても、変数として代入された値を使うのでなければ、引用符で囲んでおくべきである。そこで Example 4.1 を書き直すと次のようになる。

```
/* Example 4.1: Backup */
say "Enter filename and file type"
pull fn ft
"copy" fn ft "a" fn "backup a ( rep"
```

Example 4.1 ではＣＭＳコマンドを使っているが、もちろんＣＰコマンドを使うこともできる。次の例は、ＣＰコマンドでディスクをリンクし、さらにＣＭＳコマンドでそのディスクをアクセスできるようにしている。

```
/* Example 4.2: Set up environment */
'link maint 403 403 rr'
'access 403 m'
say '** Ready **'
```

一般にＣＭＳコマンドやＣＰコマンドを実行すると、コマンドからメッセージが出力されることがある。そのようなメッセージは、コマンドを入力した端末ユーザに対して情報を提供する役割を果している。しかし、ＲＥＸＸのプログラム中でコマンドを実行したときには、そのようなメッセージは不必要であることが多い。Example 4.2 を実行すると次のようなメッセージが出力される。

```
M (403) R/O
** Ready **
```

１行目はＣＭＳの access コマンドに対するシステムからのメッセージである。２行目はＲＥＸＸの say 命令によって出力したものである。

そもそもＲＥＸＸプログラムを実行するユーザにとっては、そのプログラムが全体としてどんな機能を果たすかが重要なのであって、プログラム中の個々の命令やコマンドについては知る必要はないはずである。そしてＲＥＸＸプログラムからは say 命令でメッセージ出力をすればよいので、コマンドからのメッセージは不必要ということになる。

ＣＭＳコマンドからのメッセージを出力しないようにするには、set cmstype コマンドを使う。Example 4.2 に set cmstype コマンドを入れると次のようになる。

```
/* Example 4.2b: Set up environment */
'set cmstype ht'
'link maint 403 403 rr'
'access 403 m'
'set cmstype rt'
say '** Ready **'
```

２行目の set cmstype コマンドではオペランド ht を指定している。これによってＣＭＳコマンドからのメッセージは端末に表示されなくなる。５行目の set cmstype コマンドではオペランド rt を指定している。これによって、今度はメッセージが端末に表示されるようになる。もしもこの５行目のコマンドがないと、６行目の say 命令によるメッセージの表示までも消されてしまうことになる。say 命令自体はＲＥＸＸの命令であって、ＣＭＳコマンドではないが、実際の端末

入出力はＣＭＳによって行なわれているからである。

set cmstype コマンドはＣＭＳのコマンドであるが、このコマンドが意味を持つのは、ＲＥＸＸプログラムの中だけである。端末から直接 set cmstype ht を入力したからといって、以後のＣＭＳからのメッセージが表示されなくなるわけではない。また、ＲＥＸＸプログラムの開始時には rt にリセットされるので、事前に端末から set cmstype ht を入力しておいても効果がない。

さて、set cmstype コマンドによってＣＭＳからのメッセージを消すことができるが、このコマンドではＣＰからのメッセージを消すことはできない。たとえば Example 4.2 を次のように書き換えたとする。

```
/* Example 4.2c: Set up environment */
'set cmstype ht'
'link maint 404 403 rr'
'access 403 m'
'set cmstype rt'
say '** Ready **'
```

以前のプログラムと違うのは、３行目だけである。３行目の link はＣＭＳではなく、ＣＰのコマンドである。ユーザ maint の 404 ディスクを自分の 403 ディスクとしてリンクすることを指示している。ところが、正しくは maint の 403 ディスクであって、404 ディスクは存在しないものとすると、

```
MAINT 404 NOT LINKED; NOT IN CP DIRECTORY
** Ready **
```

というようなエラーメッセージが出力される。（エラーチェックをしていないので、たとえエラーが発生しても、Ready が出力される）　ＣＭＳはＣＰの下で動いているＯＳであり、ＣＰからのメッセージは、ＣＭＳが関知しないレベルで出力されるために、set cmstype コマンドでは消すことはできない。

ＣＰコマンドからのメッセージを消すには、ＣＭＳの execio コマンドを使う。execio コマンドはファイル入出力などにも使うが、ＣＰコマンドからのメッセージを制御するためにも使う。この場合、次のような形式になる。

```
execio 0 cp (string ＣＰコマンド
```

最初のオペランド 0 はメッセージを保存しないことを指示している。あとでそのメッセージを利用したいときには、メッセージを保存する必要があるが、いまの場合には、単にメッセージを消したいだけなので、保存の必要はない。

次のオペランド cp はＣＰコマンドを実行させることを指定している。そして

オプションで string を指定すると、それ以後の文字列がＣＰコマンドと見なされることになる。ただし、この場合に限って、ＣＰコマンドは大文字で書かなければならない。たとえば、link コマンドを実行させる場合には、

```
execio 0 cp (string LINK MAINT 404 403 RR
```

のようになる。すなわち、Example 4.2c は次のように書き直すことができる。

```
/* Example 4.2d: Set up environment */
'set cmstype ht'
'execio 0 cp (string LINK MAINT 404 403 RR'
'access 403 m'
'set cmstype rt'
say '** Ready **'
```

このように書き直すことによって、ＣＰからのメッセージを消すことができる。

ところで、上の Example 4.2 を実行すると、余分なメッセージが表示されなくなるのはいいとしても、コマンドの実行が成功したかどうかがわからなくなる。Example 4.2c ではリンクに失敗すると、エラーメッセージが出力されていたが、Example 4.2d では、たとえリンクに失敗しても、エラーメッセージは出力されずに、最後の ** Ready ** が出力されることになる。

端末から同じコマンドを実行した場合には、エラーメッセージが表示されるのはもちろんであるが、リターン・コードが０以外の値になり、それがＣＭＳの Ready メッセージに表示される。たとえば次のようになる。

```
R; T=0.02/0.05 09:55:20
link maint 404 403 rr
MAINT 404 NOT LINKED; NOT IN CP DIRECTORY
R(00107); T=0.01/0.01 09:55:36
```

ＲＥＸＸのプログラムの中では、このリターン・コードが rc という変数に代入されることになっている。したがって、コマンドの実行が終了したあとで、変数 rc が０であるかどうかを調べれば、コマンドが正常に終了したかどうかがわかるわけである。そこで Example 4.2d を書き直してみると、次のようになる。

```
/* Example 4.2d: Set up environment */
'set cmstype ht'
'execio 0 cp (string LINK MAINT 404 403 RR'
if rc ^= 0 then do
```

```
        say '!! Cannot link !!'
        exit
        end
'access 403 m'
'set cmstype rt'
say '** Ready **'
```

4行目から8行目まで、新たに4行を挿入している。4行目の if 文の最後にある do はＲＥＸＸの命令であり、8行目のＲＥＸＸ命令 end と対になっている。ふつうは then の後ろに1つの命令またはコマンドしか書くことができないが、do と end で囲むことによって、多数の命令やコマンドを書くことができる。

7行目の exit は、ＲＥＸＸプログラムを途中で終了させるための命令である。したがって、link コマンドの実行が失敗して、リターン・コードが0でない値になったときには、

```
!! Cannot link !!
```

を端末に表示して、プログラムを終ることになる。このときには、** Ready ** は表示されないので、ユーザはエラーが発生したことに気が付く。

5．おわりに

最初にも述べたように、ＲＥＸＸを使いこなせば、かなりのデータ処理をＲＥＸＸプログラムですませることができる。この解説では全く触れなかったが、ファイル入出力もできる。コマンドの入出力に対して、プログラム・スタックなるものが用意されている。ふつうは端末に応答を要求するようなコマンドを実行するとき、事前にプログラム・スタックにその応答を入れておくことができる。一方、コマンドからの出力を端末に表示しないための方法については4節で説明したが、出力をプログラム・スタックに保存することができる。そしてプログラム・スタックからその出力を取り出して処理することができる。変数についても、単純変数しか説明しなかったが、配列や構造化された変数を使うことができる。また、本文で簡単に紹介した parse 命令はかなり高度な解析機能をもっている。この機能を、ＰＬ／Ｉなどの一般のプログラミング言語で実現しようとすると、かなりの行数のプログラムを自分で書かなければならないであろう。

以上、簡単に項目だけ紹介したが、ＲＥＸＸの機能の豊富さを感じとってもらえたであろうか？　これらはすべて「使用者の手引き」に例題とともに親切に説明されている。本解説だけで終るのではなくて、ぜひとも「使用者の手引き」へも手を伸ばしてみてもらいたい。