ソフトウェア工学 22－8
（1982. 2. 9）

データ抽象化言語CLUによる中規模システム作成の経験

久野　靖、遠城秀和、木村　泉
（東京工業大学理学部）

1．はじめに

本報告では、データ抽象向き言語CLU［1］を用いて日本語文書処理システムを作成した経験をもとに、モジュール化とデータ抽象を生かしたソフトウェア作成手法について論ずる。

使用した処理系はFACOM230－45S用処理系［2］（CLU45Sと記す）である。作ったシステムの概略は図1に示すとおりである。すなわち、かなわかち書き（またはローマ字わかち書き）日本語文を入力し、漢字かなまじり文および単語一覧表を出力する。かなファイルおよび単語一覧表を適宜編集して再度システムを走らせれば、辞書が必要に応じて更新された上で再処理がなされる。システム作成に要した労力は約6人月であり、完成したシステムは実用に供しつつある［3］。原プログラム行数は約6000行であり、これはCLUが表現力の大きい言語であることからみて、相当大きなプログラムといえる。高水準言語を使用したため資源消費はある程度大きくなったが、それでも論文1編が120KB程度の主記憶内で20000字程度のもので経過時間約20分ほど、10000字程度のもの（本稿）で15分ほどで処理可能である。さらにシステム全体をHITAC M－200H計算機用処理系（CLU－HMと記す）に移植する作業も進行中である。

なおここでは言語CLUを用いたが、本文で示すアイデアの大部分はデータ抽象とモジュール化の機能を持つ他の多くの言語（たとえばAda）にも適用可能と考えられる。

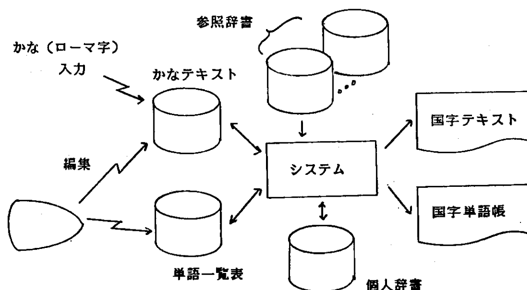

図1．　システムの概略

2．モジュール環境

今回のようなシステムを作成する場合には、システムのそれぞれの機能を実現することよりむしろ

1）計算機のアーキテクチャ

2）オペレーティングシステムの提供する環境

3）システム自身に含まれる他のモジュール

などと「接続」することに労力を食われてしまいがちである。そしてこの問題は、作成されるシステムの機能、OSの機能、およびシステムに求められる通用力の程度などが増大するにつれていっそう増大する傾向がある。

そこで、本システムの開発にあたっては全体を

1）界面モジュール群

2）道具モジュール群

3）機能モジュール群

の三階層にわけ、それらを並行して作成して行く、という方針をとった。以下にそのあらましを述べる。なおここでモジュールとはCLUのクラスタ（Adaのパッケージと同様のもの）をいい、外部からは見えないデータとそれを扱うための操作群をひとまとめにしたものである。

2．1　界面モジュール

実際に開発をはじめるにあたって、何よりもまず金物やオペレーティングシステムになるべく依存しないようにしたいと考えた。そこで、システム全体が外部環境とどこで接続するかを洗い出し、そこに統一的な界面を作り出すためのモジュール（界面モジュール）群を作成することにした。これにより機械独立なモジュール環境とでもいうべきものを作りあげることができた。

具体的には界面モジュールとして次のものを作成した。

num　－　整数型（下記）
cval　－　文字コードの定義
text　－　テキストファイル
sfile　－　順アクセス辞書ファイル
ifile　－　直接アクセス辞書ファイル

numは機能的には普通の整数型と類似しているが、内部表現に十分多くのビット数を持たせてある。このようなものを特に用意したのは、標準の整数型で扱える値の範囲が外部環境に依存するためである。実際にはCLU45Sでは非標準の型として「長い整数」型が用意されているのでそれを利用した。CLU－HMでは標準整数で十分用がたりるのでそれをそのまま用いている。もし扱う値の範囲がさらに大きくなった場合にはnumの内部に多倍長の計算を組み込むことによって、numを使用する他のモジュールに変更を加えることなしに対応できる。

cvalは、モジュール環境の中で独自に定義した文字セットを使うためのモジュールであり、外部コードからの変換操作や制御文字の値を返す関数などを含んでいる。わ

ざわざ独自の文字セットを使っているのは、外部の文字セットには大抵

1）必要なだけの図形文字がそろっていない

2）文字の並び順が不自然である

3）制御文字が予約されている

などの不都合があって使いにくいこと、および外来の文字セットを用いると多少とも外部環境に依存したプログラムになってしまうことを考慮したものである。ｃｖａｌ以外のモジュール中に文字定数を書く場合は、下に示すように、初期設定の際にｃｖａｌ中の適当な関数を呼び出し、その結果返された値をしまっておいて用いる。

ｎｌ：ｃｈａｒ：＝ｃｖａｌ＄ｎｌ（）

ｏｋ：ｓｔｒｉｎｇ：＝ｃｖａｌ＄ｓｃｏｎ（"ｏｋ"）

残りの三つはすべてファイルに関するモジュールである。われわれのシステムでは大きくわけてテキストと辞書の２種類のファイルを用いる。そのちがいは、テキストファイルが文字単位の入出力を基本とするのに対し、辞書ファイルは項目単位で読み書きする点にある。さらに辞書ファイルに対しては見出し項目をキーとした直接アクセスも用意した。

界面モジュール群の大きさは合計で６００行たらずであり、一人で十分作成／管理できた。このようにファイルに関する界面が小さくてすんだのは、外部環境の多様な機能のうちから扱う応用領域に必要な機能だけを注意深く選んだことによる。これにより、移植の際に書きなおす範囲も小さく限定できる。もしファイルに関してもとのオペレーティングシステムが持つ機能をひととおり全部取り入れようとすればずっと大変なはずである。

２．２　道具モジュール

界面モジュールの作成に続いて、文書処理に必要と思われるまとまとった機能を選び出し、それらを実現するモジュール群（道具モジュール）の作成をおこなった。「道具」ということばを使ったのは、これらのモジュールが特定の「主プログラム」の下請けではなく、汎用に使われるものだからである。これらのモジュールによって作られる環境を文書処理モジュール環境と呼んでよかろう。

実際には道具モジュールとして次のものを用意した。

ｃｓｅｔ　－　文字の集合

ｓｔａｂｌｅ　－　文字列をキーとするハッシュ表

ｔｏｋｅｎ、ｔｔｅｘｔ　－　綴りとその入出力

ｃｏｍｍａｎｄ、ｃｔｅｘｔ　－　指令とその入出力

ｖｉｆｉｌｅ、ｑｆｉｌｅ　－　ｉｆｉｌｅの最適化版

ｒｅｍａｒｋ　－　メッセージ出力

ｓｏｒｔｂｕｆ　－　文字列配列の外部整列

ｃｓｅｔはＰａｓｃａｌにおけるｓｅｔ　ｏｆ　ｃｈａｒ（文字の集合型）に相当するもので、操作としては空な集合を作る関数、集合に文字を追加する関数、二つの集合の和を作る関数、ある文字が集合に属するかどうか調べる関数、および英字や数字など一般的に使う集合を返す関数などを持つ。たとえば

ｋａｎａ：ｃｓｅｔ：＝ｃｓｅｔ＄ｈｉｒａ（）＋
　　　　　　　　　ｃｓｅｔ＄ｋａｔａ（）

のようにして「かな文字の集合」を定義しておき、

ｉｆ　ｃｓｅｔ＄ｍｅｍｂｅｒ（ｋａｎａ，ｃｈ）・・・

により文字がかなであるかどうかを調べることができる。特定の文字のコードを「知って」いるのはｃｓｅｔとｃｖａｌだけなので、コード体系を手なおしするときはこの二つを修正すればよい。

ｓｔａｂｌｅは文字列をキーとするハッシュ表の機能を持ち、渡された文字列をあらかじめ指定された範囲内の一意的な整数に変換する。実際にいろいろなデータ型の表を使うには、その整数を添字としてデータ項目の配列から

ｖ：＝ａｖ［ｓｔａｂｌｅ＄ｍａｐ（ｔ，ｓ）］

のようにして要素を取り出すことになる。

ｔｏｋｅｎやｃｏｍｍａｎｄは文書ファイルの中にあらわれる綴りや（整形プログラムなどに与える）指令に対する抽象データ型であり、ｔｔｅｘｔやｃｔｅｘｔはこれらを読み書きするためのモジュールである。これらを共通に使うことでモジュール間の接続の誤りを減らし、利用者から見たシステムの統一性を高めることができた。

そのほか、ｖｉｆｉｌｅやｑｆｉｌｅはｉｆｉｌｅにバッファリングの機能をかぶせることでディスク参照を減らしたものであり、ｉｆｉｌｅを下請けに呼ぶことで機械独立性を確保している。ｒｅｍａｒｋはメッセージや虫取り情報などを打ち出すためのファイルである。ｓｏｒｔｂｕｆは整列（ソーティング）機能を提供している。ｓｏｒｔｂｕｆについてはあとでもう一度触れる。

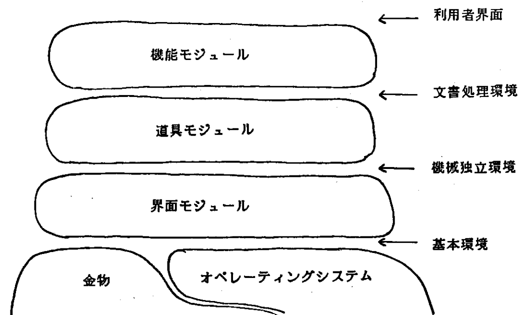

図２．　モジュール環境の階層構造

２．３　機能モジュール

ある程度道具がそろってきたところで、システムの利用者が直接利用する諸機能を実現するためのモジュール群（機能モジュール）の作成に取りかかった。機能モジュールの具体的な内容については本稿では省略するが、全

体としては１０個ほどのプログラムとそれぞれの局所的な下請けのモジュールから成り、システムの全行数の７割ほどを占めている。

これらのモジュールは、必要な道具モジュールがそろっていたため比較的気楽に書くことができ、大幅な変更にもほとんど心理的ていこうがなかったのみならず、実際にいくつかの異なる版を作成してしょう／ためす検討することも容易であった。その段階で必要ならば新しい道具モジュールを追加し、そのために外部環境に依存する新しい機能が必要なら対応して界面モジュールの拡張や追加もおこなった。

このようにして、できあがったシステムは図２に示すような３種のモジュール環境とモジュール群から成階層構造を持つものとなった

２．４　他の構成法との比較

プログラム作成にモジュール化やデータ抽象の機能を持った言語を用いただけで、よいプログラムができるわけではなく、モジュール構造をきちんと設計することが大切である。そのよく知られているやりかたに「下向き設計法」（ｔｏｐ－ｄｏｗｎｄｅｓｉｇｎ）がある。その要点はまずシステム全体の機能を規定し、続いてそれを細分化しながら各分割レベルにモジュールを配置して木構造を作るところにある。しかし、特に大きな（あるいは実験的な）システムを作ろうとしたとき下向き設計法を適用しようとしても教科書どおりには行かないことが多い。おそらくその真の原因は「システム全体の機能」をはじめからきっちりと規定することが本質的にできないことにあろう。

われわれの日本語文書処理システムの場合も、試作版やシステムの機能に対するおおまかなイメージはあったが、作成の途上でさまざまな代案をためして大幅な変更をおこなったため、実際にできあがったものはそれとはだいぶちがったものになった。そしてそのようなことが楽におこなえたのは、モジュール環境のわく組みを設定してできるところから段階的に作っていったことが大きな理由であったといえる。

４．応用システムの通用力

応用システムを作る際重要な考慮点の一つに移植／変換の容易さがあげられる。そのためにはプログラムからできるだけ外部環境に依存する部分をなくすことが望ましいが、高級言語とコンパイラを使ったとしても、いつもうまく行くとは限らない。そこで、大切なのはプログラムを無変更で移し換えられるようにすることではなく（おそらく多くの場合それは不可能であろう）、移し換える際に変更しなければならない部分を限定するように工夫する方が順当であろう。

われわれが応用システムのうちで外部環境に依存する部分を界面モジュールとしてまとめたのは　　　この理由による。すなわち、いったん外部環境に独立なモジュール環境ができれば、比較的少量の界面モジュールのみを変換する（言い換えればモジュール環境を移植する）ことでその中にあるシステム全体を移植できる。その場合、変換する部分が少ないことだけではなく、モジュール環境の中にある残りのすべての部分について虫取りのやりなおしがいらないことも移植コストを引き下げる効果をもたらす。さらに、新しい外部環境でもモジュール環境が移されると同時にこれまでに作られたすべての道具の恩恵をこうむることができること、およびもとの環境との間で自由にモジュールのやりとりができ、仕事の成果をわかちあえることも大きな利点である。

```
words = proc()
  sb:sortbuf := sortbuf$create(astr$["1"],
          astr$["u01","u02","u03","u04"])
  uin:ttext := ttext$input("uin")
  while true do
    t:token := ttext$gettok(uin)
    if token$is_wd(t) then
      sortbuf$put(sb, astr$[t.wd])
    end
  end
    except when end_of_file: end
  ttext$close(uin)
  sortbuf$sort(sb)
  uout:text := text$output("uout")
  last:string := ""
  while true do
    a:astr := sortbuf$get(sb)
    if a[1] ≠ last then
      text$putl(uout, a[1]); last := a[1]
    end
  end
    except when end_of_data: end
  text$close(uout)
  sortbuf$close(sb)
end words
```

ＣＬＵのプログラム

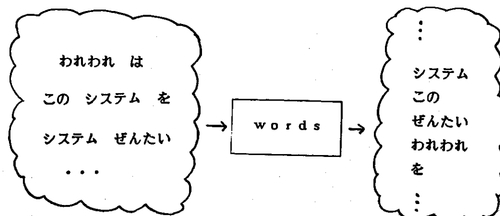

図３．　単語のリストの作成

５．道具箱

５．１　道具箱の効果

機能モジュールから見た場合、そのモジュール環境はちょうど文書処理に必要なさまざまな道具を集めた道具箱のようなものである。この道具箱を利用すれば、機能モジュールを書く仕事はかなり簡単なものになる。たとえば、文書の中に出てくる綴りの一覧表を作る、という作業を考えてみよう（図３）。そのためにはまず綴り単位の入出力をおこなうｔｔｅｘｔモジュールによって文書を読み、各「単語」を整列をおこなうモジュールｓｏｒｔｂｕｆに渡し、文書をすべて読み終ったらｓｏｒｔｂｕｆから整列された単語を取り出しながら重複を取り除いてファイルに書

き出せばよい。（ｕｉｎ、ｕｏｕｔ、ｕ０１・・・などはＦＡＣＯＭ２３０－４５Ｓにおける標準的な入出力および作業ファイルの名前である。）この例からわかるように、機能モジュールの仕事の大部分は必要な道具を正しい形で組み合わせてやることで、このためほとんどの機能モジュールはごく少ない行数ですんでいる（５０行を越えるものは３つしかない）。

このように、道具箱が整備されてくれば機能モジュールはかなり気楽に作ったり修正したりできる。その気楽さはＵｎｉｘオペレーティングシステムにそなわっているさまざまなソフトウェアの道具をｓｈｅｌｌコマンドを使ってつなぎ合わせて使うときの気楽さに似ているが、さらにＣＬＵでは記述能力が高く、型検査その他の検査機能があるので、いっそう有利な点も少なくない。

５．２　道具箱の整備

このように作成したモジュールを道具箱に入れて使えるのは、それらをはじめから共通に使う汎用の道具として作ったことによる。もし各モジュールが特定のプログラムの「部品」として作られたものなら、それらをいくら集めても道具箱にはならないであろう。さらに、われわれの道具箱には文書処理に必要な最小限の道具を選んで入れてあるので道具箱の管理がしやすく、必要な道具が簡単に選べる。したがって、ちがう目的のためにはたとえば統計解析用、構造計算用、といったいくつもの道具箱を用意しておいて、その中から必要なものだけをもってきて使う、といった形に環境を整備するのが望ましいであろう。

さらに、道具箱の中味を固定する必要もない。先に述べたように、システム開発の主要部分はそれぞれの道具箱を並行してじゅうじつさせて行くことにほかならないので、必要な道具を取捨選択して行くことが大切である。新しく作るモジュールに必要な道具を追加したり、状況の変化から不要になった道具を取り除いたりして、断えず環境を整備し続けることで整った環境を保つことができる。

５．３　道具の基準

新しい道具を道具箱に加えるとき、必要なものは何でも入れてしまうのではなく、何らかの基準を設けてそれにあったものだけを追加するべきであろう。基準の例としては

　１）その道具は広く汎用的に使われるか？

　２）既にある道具と働きが重複しないか？

　３）その道具を加えても統一性が保たれるか？

などが考えられる。これらの条件に合わないモジュールがあれば、それはある機能モジュールの局所的な下請けにとどめておいた方が無難である。われわれのシステムでも道具箱に入っているものと同数かそれ以上のモジュールが局所的な下請けとして使われている。これらについても状況に応じて道具箱に移したりその逆にするなどの整理をおこなうべきであろう。

６．外部の道具との協調

６．１　外部の道具の問題

整列はわれわれのシステムでは頻繁に使われる機能である。そこでｓｏｒｔｂｕｆを用意したのであるが、別の方法としてオペレーティングシステムの一部として備え付けの整列ユティリティ（メーカー版と記す）を利用する、という手も考えられた。しかしそれを使おうとすると、データを外部環境のファイルに移し換えなければならないこと、パラメタの指定がわずらわしいこと、別プログラムとして走らせなければならないこと、など種々の難点があり、特にキーやフィールドが固定長であることは致命的と思われた。たとえばｓｏｒｔｂｕｆでこれまでに扱ったもっとも長い「単語」は５２文字のもの（実は“ａｂｃｄ・・・ｘｙｚＡＢ・・・ＸＹＺ”）であったが、だからといって欄の幅を一律に５２字分としたのでは大変なことになる。結局ごく簡単なアルゴリズムによって外部整列をおこなうモジュールｓｏｒｔｂｕｆを作成したのはそうした考慮による。

ｓｏｒｔｂｕｆでは、整列されるものは「文字列の配列」型のデータであるとしている。（これは辞書ファイルを扱うモジュールに合わせたもので、このようなデータ型の統一はモジュール環境の統一性、わかりやすさ、および使いやすさの点で有効であった）。したがって欄の幅を気にする必要はなく、キーの指定も「ｎ番目の要素」とさえいえばよいので、「ｎバイト目から長さｍバイト」という指定方法より誤りの可能性がずっと少ない。

６．２　外部の道具の取り込み

われわれはｓｏｒｔｂｕｆを作るのにあまり労力を掛けるのは得策でないと考え、できるだけ単純なものを作成したので、当然速さはメーカー版よりだいぶ劣ったものとなった。実際には整列すべきデータの量が少ないかまたははじめからほとんど並んでいる場合はさほど問題ではないが、本当に大きいデータを整列しなければならないこともあり、そのときはメーカー版の速さは魅力的であった。

そこでわれわれはモジュール環境の中から呼べるある種の「皮」を用意することにした（図４）。すなわち渡される各データを調べて長すぎる欄を含むものがあればそれはｓｏｒｔｂｕｆに渡し、それ以外は固定長形式のファイルに固定欄にそろえて書き出してメーカー版に整列させ、最後に両者を併合して返すことにした。ほとんどのデータはそれほど長い欄を含まないので、大量のデータの大部分は高速なメーカー版に渡され、ｓｏｒｔｂｕｆはごく少しのデータを整列すればすむことになった。このように、外部

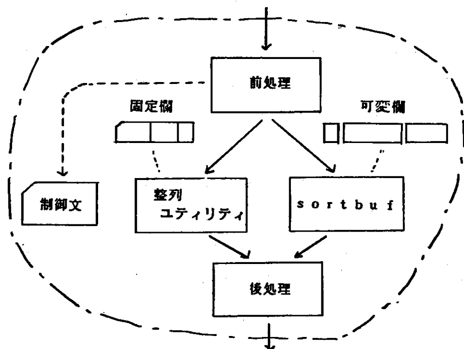

図4． 整列プログラムの取り込み

のプログラムでもモジュールの皮をかぶせることで、ファイルによる受け渡しや別プログラム起動の手間は残るにせよ、モジュール環境の一部として利用することができる。

7．チームによる作業

7．1　モジュール環境による分割と情報隠蔽

モジュール環境の考えかたによれば、システム全体は複数の階層から成り、その中に個々のモジュールが存在する。システム全体の、各階層およびその中の各モジュールへの分割は均等であり、各モジュールはある程度の自己完結性を持つので、各プログラマに自由にそれらを割り振ることができる。そして、階層のちがうモジュール間の参照関係は（内側に向って）1方向であり、モジュール群間の関係がはっきりしやすい。さらに各モジュールはそれを囲むモジュール環境によって「守られて」いて、ずっと内側のモジュールが外からむやみに呼ばれることはないので、プログラムの局所性を増すことができる。さらに、上位モジュールを受け持つプログラマは高レベルのモジュール環境の中に「住んで」いるので、低レベルの詳細や外部環境について知る必要がない。

実際、われわれのシステムは二人（久野、遠城）で開発したが、上記の利点は非常にはっきり感じられた。たとえば界面モジュールは一方だけが分担したので、もう一人の方は外部環境の細部、低レベルの入出力やバッファの管理、外部の文字セット、などについてまったく知らずにすんだ。また、システム全体の大きさにくらべればプログラマ同志が連絡しなければならないことがらの数は大変少なかったと感じている。

7．2　プログラマ間の連絡

今回われわれの使ったＣＬＵ処理系では分割コンパイルのために各モジュールごとにそこに含まれる操作の名前、パラメタや返される値の型などを記述した「仕様ファイル」（Ａｄａにおけるパッケージ仕様に相当）を用意するようになっている（図5にｔｅｘｔモジュールの仕様を示す）。これに含まれている情報はごく限られているが、構文が明確に定まっており、かつコンパクトにまとまっているので、プログラマにとっては自分の必要なモジュールや操作を探すのにきわめて有用であった。

われわれは単に仕様ファイルのありかをきめてそれぞれが自分の担当のモジュールの外部仕様を作成／管理するというやりかたをとった。もちろん一つのファイルを複数人が同時に更新しようとすれば問題がおきるが、端末がたまたま一つの部屋に集中していたこと、人数が二人と少なかったことなどから特にまずいことはおこらなかった。また、他人の作成したモジュールを使いたいときも、ほとんど仕様ファイルを見るだけで間に合った。

このことから見て、仕様ファイルを第1レベルの文書として整備することはかなり効果があると思われる。今回のわれわれの場合は人数が少なく断えず顔をあわせていたため、わからないところは口頭で聞けばすんだが、より大きいシステムを多人数で開発する場合には当然もっときちんと文書を作って管理する必要があろう。その点については文献［4］および［5］が参考になろう。

```
text = cluster is input, output, close,
    geta, getl, getc, back, puta, putl, puts, putc
  input = proctype(string) returns(text)
  output = proctype(string) returns(text)
  close = proctype(text)
  geta = proctype(text) returns(astr)
                        signals(end_of_file)
  getl = proctype(text) returns(string)
                        signals(end_of_file)
  getc = proctype(text) returns(char)
                        signals(end_of_file)
  back = proctype(text) signals(invalid)
  puta = proctype(text, astr)
  putl = proctype(text, string)
  puts = proctype(text, string)
  putc = proctype(text, char)
end text
```

図5． 仕様ファイルの例

8．プログラム言語の機能

本節では今回われわれが使用したＣＬＵを中心としてプログラム言語のいくつかの側面について考察する。プラス面は次のとおりである。

データ抽象、型検査、モジュール化、分割コンパイル

これらの機能は大きなプログラムを書くには不可欠といってよいと思われる。特に型検査は不注意なモジュール間の接続誤りによる時間の浪費を防いでくれた。

動的オブジェクト

ＣＬＵのこの機能はプログラムに大きな融通性を与えてくれた。さらに、データの量がふえたときでもプログラムを手なおしする必要はなく、実行時に多くの主記憶を割り付けるだけですんだ。

システムプログラミング機能

ＣＬＵは見かけは高レベルの言語であるが、かなりシス

テムプログラムに近い部分も十分ＣＬＵで記述することが可能であった。たとえば、われわれが使った低レベル入出力は単なるオペレーティングシステムのデータ管理機能を呼び出す皮にすぎず、アセンブラとデータ管理マクロを用いて書けることはすべてＣＬＵでも記述できた。

基本環境の置き換え

われわれの用いた処理系では基本環境を容易に変更できた。標準環境をあらかじめ整理しておけることは非常にありがたかった。

一方マイナス面は次のとおり。

効率

ＣＬＵでは動的オブジェクトを管理するためのオーバーヘッドが避けられない。しかしこれは融通性と表裏一体の関係にあり、システムの使われかたにもよるがある程度はやむを得ないことと思われる。

外部のプログラムの取り込み

外部のユティリティなどをモジュール環境の一部として自由に使えるようになることが望ましい。

終期設定（ｆｉｎａｌｉｚａｔｉｏｎ）

プログラムがアボートした場合、自分で管理しているバッファなどに残っているデータは失われてしまう。そこで、あらかじめあと始末をおこなう手続きをシステムに教えておいて、必要なら自動的に走らせてくれることが望ましい。データ抽象言語と終期設定の関連については文献［６］に議論がある。

９．まとめ

モジュール環境の考えかたは特にモジュール化／データ抽象機能を持つ言語によくマッチし、実際に応用システムを開発する上で役にたった。この手法のおもな利点は応用システム作成において問題となる外界や他モジュールとの接続のしかたをうまく制御できることにあり、また融通性、通用力、共同作業、などの面でも改善をもたらすことができる。また実例として日本語文書処理システムの開発に伴って作られたモジュール環境について説明し、このやりかたを実際のシステム作りに応用する上での問題点などについても検討した。

謝辞　システム作成のためにＣＬＵ処理系を使用させてくださった佐渡一広氏、およびさまざまなアドバイスをしてくださった木村研究室の皆様に感謝します。

参考文献

［１］Ｌｉｓｋｏｖ　ｅｔ　ａｌ．、ＣＬＵ Ｒｅｆｅｒｅｎｃｅ Ｍａｎｕａｌ、Ｓｐｒｉｎｇｅｒ－Ｖｅｒｌａｇ、Ｂｅｒｌｉｎ Ｈｅｉｄｅｌｂｅｒｇ、Ｎｅｗ Ｙｏｒｋ、１９８１

［２］佐渡一広、プログラム言語ＣＬＵの実用的処理系とその使用経験、情報処理学会論文誌、Ｖｏｌ．２２、Ｎｏ．４、ｐｐ．２９５－３０３

［３］久野、遠城、マクロ方式かな漢字変換のシステム化、情報処理学会第２４回全国大会予稿集

［４］Ｔ．Ｒ．Ｈｏｒｓｌｅｙ、Ｗ．Ｃ．Ｌｙｎｃｈ、Ｐｉｌｏｔ：Ａ Ｓｏｆｔｗａｒｅ Ｅｎｇｉｎｅｅｒｉｎｇ Ｃａｓｅ Ｓｔｕｄｙ、４ｔｈ Ｉｎｔ．Ｃｏｎｆ．ｏｎ Ｓｏｆｔｗａｒｅ Ｅｎｇｉｎｅｅｒｉｎｇ、１９７９

［５］Ｈ．Ｃ．Ｌａｕｅｒ、Ｅ．Ｈ．Ｓａｔｔｅｒｔｈｗａｉｔｅ、Ｔｈｅ Ｉｍｐａｃｔ ｏｆ Ｍｅｓａ ｏｎ Ｓｙｓｔｅｍ Ｄｅｓｉｇｎ、４ｔｈ Ｉｎｔ．Ｃｏｎｆ．ｏｎ Ｓｏｆｔｗａｒｅ Ｅｎｇｉｎｅｅｒｉｎｇ、１９７９

［６］Ｒ．Ｌ．Ｓｃｈｗａｒｔｚ、Ｐ．Ｍ．Ｍｅｌｌｉａｒ－Ｓｍｉｔｈ、Ｆｉｎａｌｉｚａｔｉｏｎ Ｏｐｅｒａｔｉｏｎ ｆｏｒ Ａｂｓｔｒａｃｔ Ｔｙｐｅｓ、５ｔｈ Ｉｎｔ．Ｃｏｎｆ．ｏｎ Ｓｏｆｔｗａｒｅ Ｅｎｇｅｎｅｅｒｉｎｇ、１９８０