プログラム言語 CLU の実用的処理系とその使用経験†

佐渡一広††

抽象化機能を言語の基本的機能として持つプログラム言語 CLU の処理系を，比較的小規模な（バッチ処理向き）計算機システムに，実用性を念頭において作成し，使用してみたので報告する．

使用計算機は一応 FACOM 230-45S であるが HITAC M シリーズへの移植もほぼ完了している．処理系作成にあたっては効率について特に注意した．本論文では，処理系作成上問題となる動的対象物，例外処理，繰り返し子，パラメタつきモジュール，分割コンパイル等の諸機能を中心に，上記の条件のもとで採用した処理系作成技法について論ずる．処理系はおもに Pascal で記述し，機械依存部分はマクロアセンブラによって作成した．

本処理系はすでに実用品としてプログラム作成に用いられている．その利用経験をもとに CLU 言語の長所，短所についても検討する．

1. まえがき

Fortran や Pascal のような伝統的な言語を使用してすっきりと筋の通ったプログラムを書こうとした場合，モジュール分割や情報隠蔽 (information hiding)[1] などの考えにもとづいた抽象化手法[2] を支援する機能が欠けているために不満を感ずることが多い．それらの言語でも，抽象化技法を間接的に取り入れることは不可能ではないが，プログラムの書きかたやまとめかた，コンパイル時の誤り検出などで不都合な点が多く，保守性や読みやすさといった点でもそれほど改良されない．そこで，抽象化機能を持つプログラム言語 CLU[3),4)], Alphard[5)], Euclid[6)], Mesa[7)] などの中から，言語としての完成度が高く，比較的単純で，筆者らの要求にもっとも適すると考えられる CLU を選び，FACOM 230-45 S/O SII 用処理系を作成した[8),9)].

CLU は Liskov により開発された抽象化機能を言語の基本的機能としてもつプログラム言語であって，モジュール単位のコンパイルが可能であって，データをすべて動的な対象物 (dynamic object) として表現し，大域変数 (global variable) がなく，例外処理機能 (exception handling) を持つといった特徴がある．ここで採用した言語仕様は文献 4) により，わずかの制限，変更，追加があるが，ほぼ完全な実現となっている．

使用した計算機のユーザ用主記憶は 192K バイトであるが，ほかのプログラムと並列に動く場合は個々のプログラムにはより少ない主記憶しか与えられない．オンラインテキストエディタが常時動いているため，実用となるには処理系は大体 100K バイト以下で動かなければならず，目的プログラムにしてもなるべく小さくする必要がある．また使用計算機はバッチ処理向きである．

本処理系は比較的小規模な計算機に，実用となるよう作成されており，使用記憶域を小さくすることを第一に考え，しかし実行速度についても十分に配慮されている．このため，M.I.T. の CLU 処理系[4),10)]のような大型の計算機で TSS 向きに作成されているものと作成方法は異なっている．

本論文では前記の条件下での CLU 処理系作成において問題となる動的対象物，例外処理，繰り返し子 (iterator)，パラメタつきモジュール (parameterized module)，分割コンパイル，およびプログラム管理用ライブラリ (CLU library) について実現方法を述べる．

この処理系は日常のプログラミングに用いられており，すでに実用的ソフトウェアが種々でき上っている．そこで，本論文では CLU 処理系使用上の経験に基づいて CLU 言語の長所，短所についても論ずる．

2. 処理系の構成と分割コンパイル

CLU ではコンパイルはモジュール単位でおこなわれる．モジュールには手続き (procedure)，繰り返し子 (iterator)，およびクラスタ (cluster) の3種があり，それぞれ手続き (procedural)，制御 (control)，およびデータ (data) の抽象化を表現する．手続きは一般言語での手続き，関数と同じものであり，繰り返し子は後章で述べるが，データ生成と文の実行を繰り返すものであり，クラスタはデータ型と操作 (operation)

---

† A Practical Processor of CLU and Experiences with Using It by KAZUHIRO SADO (Department of Information Science, Tokyo Institute of Technology).

†† 東京工業大学理学部情報科学科

を組み合わせたものであり抽象データ型を定義する．手続き，繰り返し子は内部に手続き，繰り返し子，およびクラスタの定義をもたない．クラスタは手続きと繰り返し子を用いて操作を定義するほか，クラスタ内に局所的な手続きや繰り返し子の定義を持てる．

モジュールをコンパイルするには，そのモジュールが使用するほかのモジュールのインタフェース仕様(interface specification)があれば足りる．ただし，インタフェース仕様としては引数および結果の，個数およびデータ型である．ただし，発生しうる信号(signal)があればその信号名を含む(**3.3**節参照)．これはCLUには大域変数がなく，対象物の考えを使っているので，ほかのモジュールの内部仕様を知らなくてすむためである．なお，インタフェース仕様はCLU言語本来のものではない．

図1に処理系の構成を示す．

CLUコンパイラはコンパイルするモジュールの実現部(implementation)とインタフェース仕様を入力し，アセンブラ語のコードを出力する．アセンブラはアセンブラ語コードを相対形式コードに変換する．ここまでがコンパイルの段階であり，各モジュールごとの相対形式コードが生成される．

すべてのモジュールの相対形式コードに基本データ型操作，モニタなどの支援ルーチンを加え，結合編集プログラムを起動して実行形式プログラムを得る．

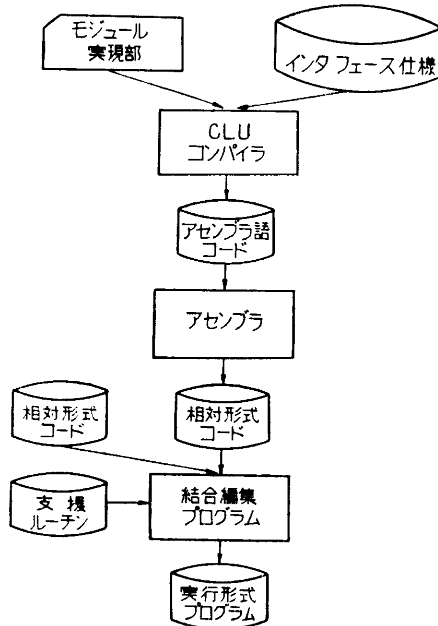


図1　処理系の構成
Fig. 1 System organization.

ただし，CLUコンパイラは一度に複数のモジュールをコンパイルすることができ，このときは同時に与えられるモジュールのインタフェース仕様はなくてもよい．あればインタフェース仕様と実現部との一致が確かめられる．プログラム全体が小さく，分割してコンパイルする必要がなければインタフェース仕様は与えなくてもよいものとしてある．これは処理速度を速めるためである．

以上で実験的プログラミングには十分であるが，これでは後章で述べるパラメタつきモジュールの処理がめんどうであり，また(大きなプログラムでは)処理とモジュールの管理が複雑となるので，CLUライブラリを置く．このときは，処理系は**図2**で示す構成になる．

ライブラリにはモジュール情報(インタフェース仕様，実現部，外部仕様，内部仕様，注釈など)，モジュール間の関係(呼び出し関係)，およびパラメタつきモジュール用の情報を入れる．

本処理系ではライブラリへのアクセスは前／後処理プログラムによりおこない，CLUコンパイラではお

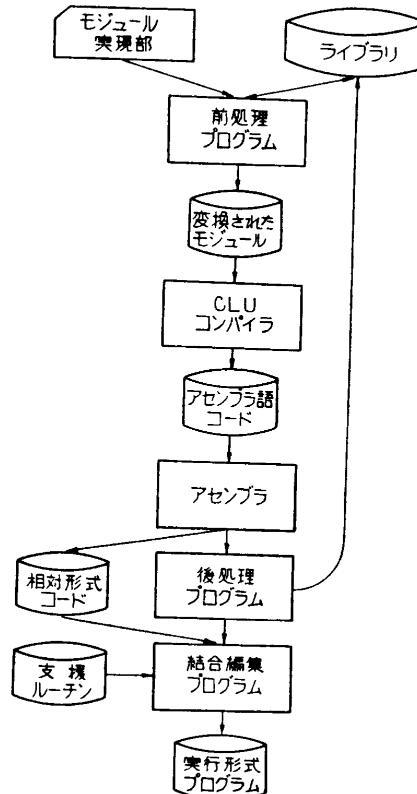


図2　ライブラリつき処理系の構成
Fig. 2 System organization with library.

こなわない．モジュールは前処理プログラムによりライブラリに貯えられ，前処理プログラムは必要なモジュールを CLU コンパイラに与える．CLU コンパイラとアセンブラにより相対形式モジュールが作成される．これが無事終了すると，ライブラリに対し後処理プログラムが正常にコンパイルされたことを知らせる．前処理プログラムについては**3.5**節でより詳しく述べる．

ライブラリにはすべてのモジュールが格納されることが建前ではあるが，実現部のないインタフェース仕様のみのモジュールや，誤りのあるモジュールを格納することも許してある．このときを含め，モジュールはインタフェース仕様を中心に管理する．インタフェース仕様の変更がない限り実現部の変更は自由である．(ほかのモジュールには影響がない)．インタフェース仕様の変更は無条件には許さない．

また，一般に1つの抽象に対し複数の実現方法が存在し得るが，現時点では1つの実現部しか許さないことにしてある．複数の実現部を許すためには，ユーザが特別の指示を与え，処理系がその制御をせねばならないからである．

なお，CLU コンパイラは Pascal で書かれ約 8,000 行，支援ルーチンはほとんどマクロアセンブラ語で書かれ約 5,000 行である．支援ルーチンにはモニタ（記憶域管理，例外処理など），基本データ型操作，入出力ルーチンがある．前／後処理プログラムは CLU 自身で書かれ約 4,500 行である．

コンパイラ作成に Pascal を使用したのは言語仕様と処理系の性能から作成時点で最もすぐれていると判断したからである．支援ルーチンの作成にアセンブラを使用したのは，効率の問題を含めこの種の低レベルのプログラミングに適する十分な言語がなかったからである．また，処理系としてコンパイラがアセンブラ語のコードを出すのは，入出力やオペレーティングシステムとの関係で結合編集プログラムを用いる必要があり，そのためには相対形式コードを作らなければならずこれにはアセンブラを使用する必要があったからである．

3. 処理方法

本章では CLU に特徴的な諸機能を実現するため，どのような手法を採用したか説明する．

3.1 基本データ型

CLU には **bool** 型，**int** 型，**real** 型，**char** 型，**string** 型（可変長）のような通常の演算子つき基本データ型 (basic type)，および **array** 型（動的に伸縮)，**record** 型のような基本型生成子 (basic type generator) があり，各データ型は豊富な操作を持つ．概念的にはこれらもクラスタによって定義された抽象データ型である．そこで，本処理系ではこれらについても一般のクラスタと同じく，モジュールインタフェース仕様をコンパイル時に受け取る．ただし，**record** 型のように選択子 (selector) があるためクラスタでは十分表現できないものや，**array** 型のようにパラメタを要するものは，型が正しく判明していればよい目的プログラムを生成できるので，特別な扱いをする．また **int** 型の加減算など，頻繁に使われる操作は目的プログラム中に展開し，効率を高めた．

3.2 対象物

CLU 言語ではすべてのデータは概念上は動的な対象物により表現される．つまり，変数や配列の要素などは直接には値 (value) を持たず，対象物への一種のポインタとして扱われる．対象物にはひとたび生成されたあとは持つ値を変えることのできない不変対象物 (immutable object，たとえば **bool** 型，**int** 型，**string** 型）と，生成後，その値を変更することが可能な可変対象物 (mutable object，たとえば **array** 型や **record** 型）がある．**array** 型や **record** 型などの対象物の各要素の値はほかの対象物を指す．また，変数あるいは対象物の要素として対象物を共有 (share) できる．

対象物を用いることによりポインタの概念を使わずにリストなどを表現できるが，実行効率にはかなり影響する．そこで，本処理系では基本データ型の不変対象物では共有が問題とならないので，直接に値を持てる限りは対象物を使用しないことにする．ただし，クラスタの実現のための表現型 (representation) として不変対象物のデータ型が使われた場合，そのクラスタとしては分割コンパイルのため特に対象物を使用する．

対象物は記憶管理ルーチンにより管理され，廃品回収 (garbage collection) の対象となる．廃品回収のためには実行時スタック上のデータおよび対象物内の要素が対象物を指すか値であるかを区別せねばならない．このために，実行時スタッフおよび対象物の内部を，対象物へのポインタを入れる部分と値を入れる部分に分け，両者を区別している．旗を用いて値と対象物へのポインタとを区別する方法もあるが，旗を置くための領域を要すること，値の大きさがデータ型によ

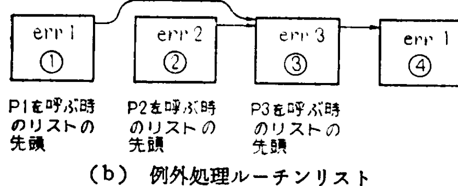
って異なること (**char** 型と **real** 型など) などから不適当と判断した．また記憶域を節約するため，対象物内には廃品回収用の専用作業領域はとらない．対象物の記憶域からの取り出しは発見順はめこみ法 (first fit) で，廃品回収は mark-sweep 法でおこなっている[11]．圧縮 (compaction) は必要なときにのみおこなっているが，実際はほとんどおこらない．特に入出力用領域や語境界 (boundary) を持つ対象物があるため圧縮はめんどうであり，出力用領域さえあらかじめ確保しておけば実際のプログラムの性質から圧縮は多くの場合なくても十分実用となる．

3.3 例外処理

まず CLU における例外処理機能について説明しておく．図3(a)において，手続きPが手続きP1を呼んだとする．P1からPに復帰するには正常復帰と信号 err1 または err3 を出す方法とがある．正常復帰なら単純に戻り次の文 (P2の呼び出し②) の実行に移る．信号 err1 を出せばPの中の err1 という名の例外処理ルーチン (exception handler) のうち，P1を呼び出した文に付けられたものに飛ぶことになる (①の行の **when** から **end** まで)．err3 を出した場合はP1を呼び出した文に直接に付けられた例外処理ルーチンはないので1つ外側のレベルにある③の例外処理ルーチンに飛ぶ (P2とP3の呼び出しは起こらない)．もしこのような例外処理ルーチンがなければP自体がエラーとなり，**failure** という名の信号が発生する．例外処理ルーチンが実行され，その処理が終るとPの中の次の文の実行に移る．ただし，例外処理ルーチン中で **return** (正常復帰) が実行されたり，新たに信号が出されたり，ループからの抜け出しなどが起ったりすればそれに従う．

```
P=proc (x: int)
    begin
       P1(x) except when err1: … end ①
       P2(x) except when err2: … end ②
       P3(x)
      end except when err3: …           ③
                 when err1: … end       ④
    end P
P1=proc (x: int) signals (err1, err3)
    if x<0      then signal err1
    elseif x=0 then signal err3
    …
    end P1
P2=proc (x: int) signals (err2)
P3=proc (x: int) signals (err1)
```

(a) 例外処理のプログラム



(b) 例外処理ルーチンリスト

図3 例外処理プログラムと例外処理ルーチンリスト
Fig. 3 A sample program of exception handling and its exception handler list.

本処理系における処理方法は次の通りである．手続きが正常に復帰するなら呼んだところに戻る．信号が出たなら，信号名に対応する例外処理ルーチンを捜す．例外処理ルーチンは図3(b)で示すようにリストにし，有効な例外処理ルーチンをつないでおく．また，リストの順序は内側の例外処理ルーチンから外側の例外処理ルーチンに向けてつなげばよい．つまり，PがP1を呼ぶには図3(b)の最左端のルーチンをリストの先頭とし，P2なら2番目，P3なら3番目を先頭とする．この方式によればリストは小さくてすみ，リストの実行時での保守も先頭のみでよい．

この方法の利点は，単純であって主記憶をあまり要しないことである．欠点としては，例外処理を使用するプログラムでは信号が発生しなくても多少は実行時の負担があり，一方信号が発生すれば例外処理ルーチンを捜すために時間がかかるという点があげられる．

ほかに，手続きごとに例外処理ルーチンの表 (信号名と有効範囲) を設け，信号が発生した時点で表から例外処理ルーチンを捜し出すという方法があり，M.I.T. の CLU 処理系[10]で使われている．この方がわずかに効率がよいが，信号の発生する番地と例外処理ルーチンを対応させること (特に支援ルーチン内で発生したとき) がめんどうになる．

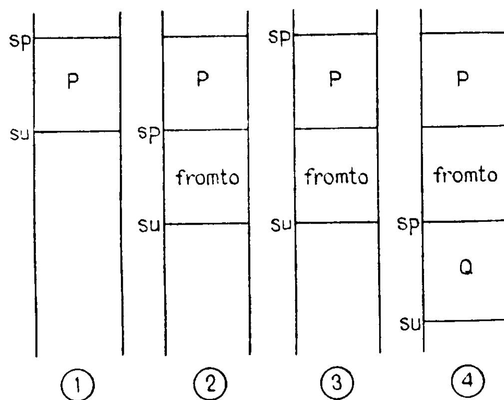
筆者らの経験によれば，例外処理機能は頻繁には使用されるものではなく，信号はめったに発生しないので，ここに述べた方法で十分であると考えられる．

3.4 繰り返し子

まず CLU における繰り返し子について説明しておく．繰り返し子は手続きと似た形式で定義されるが (図4(a)の fromto)，**for** 文から呼ばれることにより (図4(a)の2行目) 対象物 (複数個でもよい．図4(a)では **int** 型対象物) を繰り返し送り出すものである．繰り返し子は呼ばれると手続きと同じようにして実行される．繰り返し子本体の **yield** 文が実行されると **yield** 文で与えられた値 (図4(a)では f の持つ対象物) を持って帰る．その結果 **for** 文では，その値が変数に代入され (図4(a)では i に対象物が与

```
P＝proc ( )
   for i: int in fromto (1, 10) do
      Q(i)
      end
   end P
fromto＝iter (f, t: int) yields (int)
      while f<=t do
         yield (f)
         f:=f+1
         end
      end fromto
Q＝proc (i: int)
   ...
   end Q
```

（a） 繰り返し子のプログラム



（b） 実行時スタック

図 4 繰り返し子とスタック

Fig. 4 A sample program of iterator and its stack movement.

えられる)，**for** 文の本体（図 4（a）では Q(*i*)）が実行される．これが終わるとまた繰り返し子に戻り（図 4（a）では fromto)，**yield** 文の次の文（図 4（a）の *f*:=*f*+1）が引続き実行される．以後これを繰り返す．繰り返し子の実行が（本体の終わりまたは **return** 文の実行によって）終了すると **for** 文も終了する．図 4（a）では Q(1) から Q(10) までが実行されるわけである．

本処理系における処理方法は次の通りである．実行時スタック上に現在実行中の手続きまたは繰り返し子が使用しているスタックの枠 (stack frame) へのポインタ (SP) と，現在プログラムが使用している最大のスタック領域へのポインタ (su) の 2 つを置く．手続きまたは繰り返し子の呼び出しに際しては su の次に新しく枠をとり，sp を su の位置に移し，su は枠の大きさだけ動かす（図 4（a）の **for** 文中で fromto を呼ぶとスタックは図 4（b）の①から②になる．また Q を呼ぶと③から④になる)．手続きからの正常復帰と繰り返し子の終了による復帰では，sp と su をもとに戻す（図 4（a）の Q が終るとスタックは図 4（b）の④から③に，fromto が終ると②から①になる)．**yield** 文の実行では sp のみを繰り返し子が呼ばれる前の状態に戻す（図 4（a）の **yield** 文を実行するとスタックは図 4（b）の②から③になる)．**for** 文から繰り返し子に戻るには同じく sp のみ移動させる（図 4（a）の Q(*i*) 終了後 fromto に戻ると，スタックは図 4（b）の③から②になる)．基本的にはこの方法で十分だが，復帰番地や sp，su の退避，引数と結果の受け渡しがこれに加わる．本処理系ではさらに細かい操作をおこなって効率を上げている．

このような単純な方法で処理可能なため，繰り返し子は効率よく実行できる．**yield** 文のオーバヘッドは手続き呼び出しの 1/4 以下である．なお，M. I. T. の CLU 処理系[10]もこれと同様の方法をもちいている．

3.5 パラメタつきモジュールと CLU ライブラリ

たとえば，スタックを定義するのにスタックに積まれる要素のデータ型ごとに別々の定義を書かなければならないのでは不便である．処理の本質的な形式は同じなのだから統一した書きかたがしたい．そこで CLU では要素のデータ型をパラメタとすることで **int** 型対象物を要素とするスタック stack[**int**] や，**string** 型対象物を要素とするスタック stack[**string**] などを一挙に定義できるようになっている．すなわち，スタックのクラスタ定義において

stack＝**cluster** [*t*: **type**] **is** …

のように，stack 定義のクラスタ内では要素の型として *t* を使用し，stack を使用する側で stack[**int**] や stack[**string**] として *t* をパラメタとして与える．同様な方法で手続きや繰り返し子にもパラメタをつけることができる．パラメタのついたモジュールをパラメタつきモジュールと呼ぶ．（クラスタのときのみ type generator とも呼ぶ．）パラメタは手続きや繰り返し子の引数とは別であり，たとえば手続きでは 2 数の最大値をもとめる関数を max[**int**](*x*, *y*) のように書き表わされる．パラメタとしてはデータ型のほか，基本データ型の定数を与えることも可能である（上例に *t*: **type** とあるのはパラメタがデータ型であることを示す)．

この機能の処理方法としては，コンパイル時に一種のマクロ機能として処理する方法[12]，実行時にモジュール結合をおこなう方法[10),13]などがある．

コンパイル時処理による方法では，パラメタごとに

モジュール例 (module instance) を生成する．たとえば前記の stack では stack[int] および stack[string] に対応して2つの実行用コードを生成する．この方法では目的プログラムの効率はよいが，プログラムの段階的な作成がしにくく，コンパイルが複雑になり，また複数のモジュール例が生成されることもあり，目的プログラムが全体として大きくなるおそれがある．また，再帰的にパラメタつきモジュールを使用することは不可能である．

実行時にモジュールを結合する方法では，モジュール単位のコンパイルが容易であり，プログラム作成やコンパイルが楽であるが，常に実行時の処理をするため目的プログラムが大きくなり，実行速度も遅くなる．特にこのためにはデータ形式を統一せねばならず，これでは **3.2** 節で述べたような対象物の実現方法が採用できない．

以上のことを考慮して，本処理系ではコンパイル時にモジュール例を生成する方法を採用した．前述のように，この方法によるとパラメタつきモジュールの再帰的利用はできなくなるが，これで処理不可能となるプログラムは実用上ほとんどなく，禁止しても問題はないと考えた．また，複数のモジュール例が生成されることは多いが，2モジュール例程度であれば実行時結合と比べても目的プログラムの大きさはそれほど変わらないし，stack[int] と stack[char] のように事実上同一のデータ形式として処理可能なものは共用モジュールとして処理することも考えられる．

コンパイル時による方法でパラメタつきモジュールをコンパイルするには，そのモジュールの使われかた(パラメタの値)に応じてモジュール例を生成する必要がある．このため，パラメタつきモジュールを参照するモジュールがコンパイルされる時点で必要なモジュール例がわかる．パラメタつきモジュールのコンパイルは各モジュール例ごとに参照された時点でコンパイルする必要がある．そこでライブラリを利用し，これに使用された実パラメタすべてと，モジュールの実現部を格納し管理する．こうすることにより分割コンパイルと段階的プログラム作成が可能となる．

モジュールの実現部は前処理プログラムによりライブラリに貯えられる(図2)．前処理プログラムは入力したモジュールのうち，パラメタのないモジュールについてはコンパイルするために実現部を出力する．その中でパラメタつきモジュールが使用されていれば名前を置き換えておく．パラメタつきモジュールが入力されたら，すでに参照されている形式と新しく参照された形式についてモジュール例を出力する．この際モジュール名は新しくつけ，パラメタは使用されたデータ型あるいは定数に置き換えてパラメタのないモジュールとしてモジュール例を生成する．前処理プログラムが出力したモジュールは図1での入力モジュールとインタフェース仕様にあたるが，この段階ではパラメタについて一切考えなくてよい．

3.6 機械依存部分の取り扱い

われわれは時に計算機に依存するプログラムを作成する．そこで基本的入出力をおこなう入出力データ型や，バイトやワードといった単位でデータを扱うデータ型（ほとんど入出力と関係する）を用意した．したがってアセンブラ語を使用せずに CLU でシステムプログラムの作成をおこなうことができる．

これらのデータ型の操作はすべてアセンブラ語で書き，CLU のプログラムからこれを呼び出す形式をとった．CLU のモジュールをコンパイルするにはそのモジュールで使用するほかのモジュールのインタフェース仕様があれば十分である．アセンブラ語で書かれたモジュールもこれに準じた扱いをし，CLU コンパイラにはインタフェース仕様のみを与える．アセンブラ語のモジュールは別にアセンブルをし，相対形式コードとして用意する．ただし，インタフェース仕様とアセンブラ語のモジュールの照合はおこなわない．

4. 効　率

本処理系の実行効率について簡単に述べておく．

英文用のテキストフォーマッタと行列の固有値を求めるプログラムを，基本的アルゴリズムを変えずに CLU, Pascal および Fortran で書き，実行させてみたときの実行時間を**表1**に示す．テキストフォーマッタでは，CLU に文字列データ型があることから Fortran よりも速くなっている．CLU 版の行列固有値計算プログラムでは，行列型データを導入した．また，実行時間中の対象物の生成，廃品回収，手続き呼び出しのオーバヘッドに関する部分を**表2**に示す．対象物の生成と手続き呼び出しは実行回数と平均所要時間か

表1 実行時間の例
Trble 1 Execution times of sample programs.

	CLU	Pascal	Fortran
テキストフォーマッタ	27.0 sec	20.0 sec	36.4 sec
行列の固有値	9.1 sec	6.9 sec	3.5 sec

表 2 オーバヘッド時間
Table 2 Overhead measurement results.

	総実行時間	オブジェクト生成	廃品回収	手続き呼び出し
テキストフォーマッタ	27.0 sec	3.6 sec	2.0 sec	3.4 sec
行列の固有値	9.1 sec	—	—	2.6 sec

ら求めた値であり，廃品回収時間は実時間である．

実行時間はプログラムの性質や書きかたに大きく左右されるが，一般的な目安としては Pascal や Fortran と比べて 1～4 倍を要する．また対象物の生成に 5～20%，廃品回収に 5～20%，手続き呼び出しに 10～40% 要している．このオーバヘッドを除くとほぼ Pascal と同じ実行速度になる．

これまでの実験では，ほとんど **string** 型対象物の生成の多さが問題になっている．**array** 型や **record** 型では，対象物の生成が多くてもその後に多くの操作が施されるため（配列の要素の読み出し書き込みなど）全体としてはそれほどひどい効率低下は起こらないが，**string** 型では連結や切り出しのため無用な中間結果としての対象物が生成される．記憶管理ルーチンの効率改善，無用な対象物を生成しない方法（3 個の文字列の連結を一度におこなう，定数要素のみの **array** 型や **record** 型対象物を生成しないなど）をとることなどでかなりの効率向上が見込まれる．

手続き呼び出しでのオーバヘッド軽減には Scheifler[14] がいうようにインライン展開 (inline substitution) が有効である．これは現処理系では実施していないが，前処理プログラムに展開機能をつけることで導入できる．

これらのことが十分におこなわれれば実行効率の問題も解消されると考える．

なお，処理系の大きさはコンパイラ，前／後処理プログラムとも約 110K バイトである．目的プログラムの大きさは，支援ルーチンが最低 12K バイト（記憶管理ルーチンが 4K バイト，例外処理や手続き呼び出しなどの処理ルーチンが 3K バイト，基本出力に 3K バイト，デバッグ用に 2K バイト），で目的プログラムは約 20 バイト／行（実行時チェックなし）または約 30 バイト／行（実行時チェックあり）である．

5. 使用経験からみた CLU 言語の長所・短所

筆者の周辺ではすでに本文の CLU 処理系をもちいてテキストエディタ，漢字マクロプロセッサ，Backus 風関数的プログラミング処理系[15]，CLU 処理系の前／後処理プログラムなどが作成されている．本章ではこれまでの経験に基づいて CLU の長所，短所を論ずる．なお，文献[16]に抽象化手法によるプログラミングの経験が論じられている．

CLU 語でよいプログラムを作成するには上手な設計が必要である．上手な設計であればモジュール分けが楽で保守性や読みやすさにすぐれたプログラムとなる．CLU では完全なモジュール分けが要求されるため，CLU に慣れないうちはプログラムの作成は必ずしも楽ではないが，慣れるに従い比較的設計のよいプログラムが作成できるようになる．

CLU でプログラムを書くと，抽象化によりまとまった考えをすなおに表現できる．プログラムの誤りのほとんどはコンパイル時に取れ，実行時になって判明する誤りは少ない．実行時に起こる誤りも，モジュールが閉じておりほかのモジュールの影響を受けにくいので，誤りの場所を局所化できるため楽に修正できる．またプログラムの変更も容易である．

一般に CLU で書かれたプログラムは，完全にモジュール分けがなされること，クラスタにすべての操作を明白に記さなければならないこと，大域変数がないこと，暗黙の型変換やデータ型操作の自動選択 (overloading) はごく一部しかおこなわないことなどから，プログラムが長くなる傾向にあるが，その結果コンパイル時の誤り検出の機会が増し，プログラムが読みやすくなっている．さらに，モジュール間の依存度が小さく，それぞれのモジュールの独立性が高いため，モジュールの再利用が容易である．また，大域変数がないのでモジュールの使用に制限を置く必要のないことがある（モジュールの有効範囲を規定する必要がない）*．これらはこの種の言語の予想される特長としてしばしば言及されているが，これらのことは筆者らの経験ではたしかに実際にいえることである．

Pascal, Euclid, Mesa 等ではポインタ型により動的変数の使用が可能であるが，すべてのデータを動的対象物で表現すればデータ型の種類を増さずにすみ，消滅した変数を参照することもない (dangling reference)．たまたま筆者の周辺ではテキスト処理を多くおこなっているので，文字列 (**string** 型) が可変長であることは大きく役立っている．また CLU では配列 (**array** 型) を伸び縮みさせることも可能である**．

* CLU の最新の言語仕様[17]では占有変数 (own variable) が導入された．

** 実行時に **array[int]$addh**($a$, 10) で配列 a の上限を 1 つ増しその要素として 10 を入れる，あるいは **array[int]$remh**($a$) で上限の要素を捨て上限を 1 へらすなどができる．

これにより配列の上限下限を定め，有効な範囲を示す変数を別に置く必要もない．筆者の周辺での配列の使用実績では，配列の約 3/4 が伸び縮みしている．

制御の抽象化をおこなう繰り返し子は，データを順に取り出し処理する場合は便利である．繰り返し子をもちいることで使用側に規制ができ，誤りを起こしにくくなる．たとえば，ライブラリから順にモジュールを取り出す繰り返し子を用意しておくことで，ライブラリの印刷，複写はその繰り返し子を呼び出すことによってモジュールを取り出し，処理するというだけですむ．よって，終りの判定をおこない，モジュールを取り出す操作を明示的に繰り返すようなことは必要でない．もっとも全体としては繰り返し子の使用度はあまり多くない．複雑な処理，たとえば一度に 2 つのモジュールを要する時など，新しく繰り返し子を定義せねばならず，この時は繰り返し子を使用しないことが多いからである．

例外処理機能を使用すると，手続き実行中での誤り発見後の処理が楽になる．手続きの呼び出し側が誤りを意識せずとも正常に実行ができる（誤りがあればプログラムが停止してくれる）．特にプログラム作成中は有効である．全体として例外処理機能によりこれまでないがしろにされがちであった誤り検出や処理を正しくおこなう傾向がでている．ただ CLU では手続きを多重に飛び出すには間にあるすべての手続きで信号を捉え再度発生させねばならず，また例外処理ルーチン実行後信号を発生した所に戻れないといった不便さはある（たとえばエラーメッセージを出し，処理を続行したい時など）．

ほかに欠点として，内部モジュールがなく大域変数がないためモジュール数が増加することがある．もっともこれは不便ではあるが初めに述べた利点に貢献してもいる．対象物が共有可能であることは，ときに誤りの原因となる．手続きの副作用を禁止しようとして，プログラムの書きかたが複雑となり，設計の悪いプログラムになることがある．また最近の言語の多くにある数え上げ型 (enumeration type) がないことも不便である．

総じて CLU の設計目標は実効を持っているといえる．

6. む す び

われわれは CLU 処理系を作成し，使用しているが，これまでのところ実用のプログラム作成用言語処理系として十分な結果を得ている．本処理系は FACOM 230-45 S/OS II 専用として作成したが，HITAC M シリーズの移植もほぼ完了している[18]．

いずれの計算機も CLU のような対象物指向の言語に向いていないための不利はある．しかし，ファームウェア化あるいは専用計算機[19]などの方法で不利をなくすこともできよう．

なお，これまでに得られた結果などから CLU の言語仕様を変更することも考えており，最終的にはわれわれの要求にさらに近い言語にしていくつもりである．この中には並列処理機能があり，現在その外部仕様を検討している．

このほか，デバッグ用，プログラム開発用ツールの開発，最適化機能の作成が必要であると考えている．

謝辞 日ごろご指導いただき本稿作成での細部にわたる助言をいただいた木村泉助教授，米澤明憲助手，数々の議論と助言をいただいた角田博保氏，関根裕氏はじめ木村研究室の方々に感謝いたします．

参 考 文 献

1) Parnas, D. L.: A Technique for Software Module Specification with Examples, Comm. ACM, Vol. 15, No. 5, pp. 330-336 (1972).
2) Liskov, B. H. and Zilles, S. N.: Programming with Abstract Data Types, Proc. of ACM SIGPLAN Conf. on Very High Level Languages, SIGPLAN Notices, Vol. 9, No. 4, pp. 50-59 (1974).
3) Liskov, B. H., Snyder, A., Atkinson, R. and Schaffert, C.: Abstraction Mechanisms in CLU, Comm. ACM, Vol. 20, No. 8, pp. 564-576 (1977).
4) Liskov, B. H. et al.: CLU Reference Manual, Comput. Structures Group Memo 161, Lab. for Computer Science, Massachusetts Inst. of Tech. (1978).
5) Shaw, M., Wulf, W. A. and London, R. L.: Abstraction and Verification in Alphard: Defining and Specifying Iteration and Generators, Comm. ACM, Vol. 20, No. 8, pp. 553-564 (1977).
6) Lampson, B. W. et al.: Report on the Programming Language Euclid, SIGPLAN Notices, Vol. 12, No. 2, pp. 1-79 (1977).
7) Geschke, C. M., Morris, J. H. and Satterthwaite, E. H.: Early Experience with Mesa, Comm. ACM, Vol. 20, No. 8, pp. 540-553 (1977).
8) 佐渡一広：FACOM 230-45 S 電子計算機用 CLU 処理系の作成・使用経験について，修士論文，東

京工業大学 (1979).
9) 佐渡一広：CLU 処理系の作成について，第 21 回プログラミングシンポジウム報告集，pp. 153-160 (1980).
10) Atkinson, R., Liskov, B. H. and Scheifler, R. W.: Aspects of Implementing CLU, Proc. of the ACM 1978 Annual Conference, pp. 123-129.
11) Knuth, D. E.: The Art of Computer Programming Vol. 1 Fundamental Algorithms, p. 634, Addison-wesley Publishing Company (1973).
12) Gries, D. and Gehani, N.: Some Ideas on Data Types in High-Level Languages, Comm. ACM, Vol. 20, No. 6, pp. 414-420 (1977).
13) Yuasa, T.: Module-wise Compilation for a Language with Type-parameterization Mechanism, RIMS preprint-280, Research Inst. of Mathematical Sciences (1979).
14) Scheifler, R. W.: An Analysis of Inline Substitution for a Structured Programming Language, Comm. ACM, Vol. 20, No. 9, pp. 647-654 (1977).
15) 石原　哉：関数的プログラミング言語処理系の試作とその使用経験，卒業論文，東京工業大学 (1980).
16) 角田博保，佐渡一広，関根　裕：CLU を使ってみて（抽象化のしかたについて），よいプログラムを作るにはシンポジウム報告集，pp. 7-10 (1979).
17) Liskov, B. H. et al.: CLU Reference Manual, Rep. TR-225, Lab. for Computer Science, Massachusetts Inst. of Tech. (1979).
18) 吉田健一：CLU 処理系の HITAC M-180 への移植について，卒業論文，東京工業大学 (1980).
19) Snyder, A.: A Machine Architecture to Support an Object-Oriented Language, Ph. D. thesis, Massachusetts Inst. of Tech. (1979).

（昭和 55 年 8 月 11 日受付）
（昭和 55 年 12 月 18 日採録）