# 23. ＣＬＵ処理系の作成について

東京工業大学 理学部　佐渡一広
Kazuhiro Sado

## 1．はじめに

本論文はプログラム言語CLU の処理系作成上[14]の問題点およびCLU の使用経験について議論するものである。

われわれがプログラムを作成するのにFortranやpascal [10] を使用しても、モジュール分けや情報隠ぺい (information hiding) [8] などにより、抽象化手法 [5] を用いることが多い。このようなアプローチを取ってもプログラムの書きかたやまとめかた、コンパイル時のエラーチェック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良くならない。そこで、抽象化機能を持つプログラム言語CLU [6]、Alphard [9]、Euclid [4]、Mesa [2] などの中から、言語としての完成度が高くかつわれわれの要求により適すると考えられるCLU を選びFACOM 230-45S (以下45Sと記す) にその処理系を作成した。

CLU はLiskovら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る。手続き (procedual)、データ (data)、制御 (control) の３種の抽象化を基本とした、モジュール単位のコンパイルをする。データをすべて動的なオブジェクトとして表現し、広域的変数を許さず、例外処理機能を持つといった特徴がある。

処理系作成上の問題点としてCLU の諸機能 (動的オブジェクト、例外処理、繰り返し子、パラメタつきモジュール) の取り扱い、およびプログラム管理の問題があり、本論文ではこれらについて述べる。また、CLU によるプログラミングの経験についても触れる。

## 2．処理系作成にあたって

CLU 処理系作成の目的は、抽象化手法によるプログラミングの実験、日常用いるプログラムをCLU で書くこと、およびプログラムの管理・蓄積をはかることである。

作成した処理系は CLU Reference Manual [7] に基づくもので、基本的部分の変更はなるべく避けた。変更点はいくつかあるが、これらについては後に述べる。

作成手順は、まずCLU の部分言語の処理系を作成し、その後使用経験をつみながら順次機能拡張をおこなった。現在大部分が完成しており、日常のプログラミングに用いられている。未完成の機能としてはパラメタつきモジュールと一部の基本データ型操作である。またCLU ライブラリについても作成中である。

処理系を作成した重要な目的は日常使用することであるから、効率について特に重視した。使用している45S はユーザ用主記憶は192 K byteであるが、他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などが常時動いているため、実用となるにはコンパイラは大体100 K byte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。よって、処理系作成には使用記憶域を小さくすることを、実行速度より重視した。

現処理系は対象にする計算機は45S のみを考え、他の計算機への移植性はあまり考慮しなかった。ただし、後に触れるように、現在M-180 に移行中である。

## 3．処理系の構成

CLU ではコンパイルはモジュール単位におこなわれる。モジュールとは手続き (procedure)、繰り返し子 (iterator)、クラスタ (cluster) の３種があり、それぞれが手続き、制御、データ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、クラスタは新しいデータ型に操作 (operation) を定義することで抽象データ型を定義するものである。手続き・繰り返し子は内部に手続き・繰り返し子・クラスタの定義をもたない。クラスタは手続き・繰り返し子を用いて操作を定義するが、ほかにクラスタに局所的な手続き・繰り返し子を定義できる。

あるモジュールをコンパイルするのに必要な情報は、そのモジュールの使用する他のモジュールのインターフェース仕様 (interface specification) である。現処理系ではインターフェース仕様として引数・結果の個数とデータ型、例外処理の情報のみを要求する。またこれだけで十分である。

45S のCLU 処理系の構成を図１に示す。CLU コンパイラはコンパイルするモジュールの実現部 (implementation) とインターフェース仕様を入力し、FASP* 語のモジュールを出力する。FASPはFASP語のモジュールを相対形式モジュールに変換する。ここまでがコンパイルの段階であり、各モジュールの相対形式モジュールが生成される。

* FASPは45S アセンブラ。

プログラムに必要なだけの相対形式モジュールに基本ルーチン・モニタなどの支援ルーチンを加えて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。

ただし、プログラムが小さく分割にコンパイルをしない場合はインターフェース仕様を必要としない。

各モジュールにはインターフェース仕様、実現部、相対形式モジュールの3状態がある、現時点ではこれらの管理はすべてプログラム作成者自身がおこない、インターフェース仕様と相対形式モジュールとの対応などは処理系は管理しない。これではモジュールの不一致などの事故も起るので、後章で述べるライブラリ機能によって、たとえばMesaの処理系でおこなわれているような、モジュール管理をする。

なお、コンパイラはPascal**で書き約7000行である。支援ルーチンはほとんどFASPで書き約5000行である。支援ルーチンにはモニタ、基本データ型操作、入出力ルーチンがある。この他約2000行のFASPで書いた特殊なルーチンが用意してある。

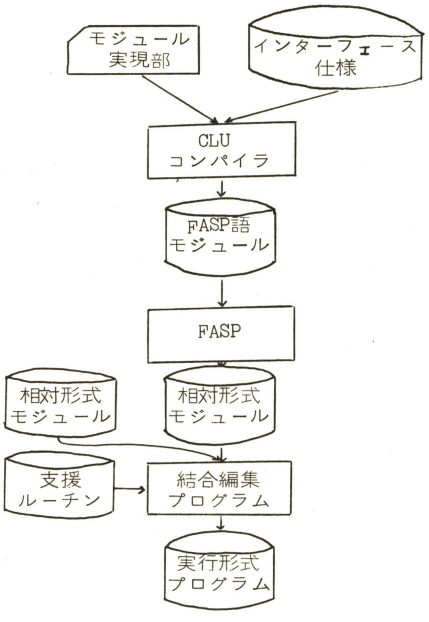


図1 処理系の構成

* 結合編集プログラムとFASPはOSのサポートする基本ソフトウェア。
** Pascalは東京大学の和田研究室からいただいたものをわれわれが改良したものを使用した。

## 4・処理方法

本章ではCLUに特徴的な機能を実現するための方法について述べる。

### 4・1基本データ型

CLUにはboolean、integer、real、character、string（可変長）といった通常の演算子を持った基本データ型、array（特殊な操作を持つ）、recordといった基本型生成子（basic type generator）があり、各データ型は豊富な操作を持つ。基本的にはこれらもすべてクラスタにより定義された抽象データ型である。そこで、コンパイラはこれらについても一般のクラスタと同じく、モジュールインターフェース仕様をコンパイル時に受け取る。しかし、recordのように選択子（selector）のためクラスタでは十分表現できないものやarrayのようにパラメタを要するものは、型が正しく判明しておれば良い目的プログラムを生成できるので、特殊な扱いをする。またinteger の加減算などひん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。この方法で基本データ型の操作の追加変更を容易にした。

### 4・2オブジェクト

すべてのデータは概念上は動的なオブジェクト（dynamic object、対象物）により表現される。つまり、変数などは直接値（value）を持たず、オブジェクトへの一種のポインタとして扱われる。オブジェクトには生成された後は一生オブジェクトの持つ値を変えない不変オブジェクト（immutable object、boolean型やinteger 型などのオブジェクト）と、生成後も値の変更が可能な可変オブジェクト（mutable object、array 型やrecord型などのオブジェクト）がある。array 型やrecord型などのオブジェクトは要素の値として他のオブジェクトを指す。またオブジェクトは複数の変数やオブジェクトで共有（share）できる。

このため、分割コンパイルが容易であり、ポインタの概念を考えずにリストなどを表現できるのだが、実行効率にはかなり影響する。そこで、基本データ型の不変オブジェクトは共有が問題とならないのでオブジェクトを使用せず、直接値を持たせる。ただし、これらの型であってもクラスタ実現のための表現型（representation）として使われた場合、そのクラスタとしてはオブジェクトを使用する。オブジェクトの使用を減らすことで、オブジェクトの生成が減り、記憶域の使用量が減り、さらにデータの参照が速くなることで、効率が良くなる。

オブジェクトは記憶管理ルーチンにより管理され、廃品回収（garbage collection）がおこなわれる。ここでの方法は記憶域を節約するため、オブジェクトに

余分な領域を取らずにおこない、またオブジェクトの生成回収とも単純なアルゴリズムを採用した。今後他の良いアルゴリズムへの変更と、オブジェクトの使用が少ないように目的プログラムの最適化をすることで効率のより以上の向上をはかる予定である。

### 4．3例外処理

CLU の例外処理機能（exception handling）についてまず説明をする。図2．1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（signal）err1またはerr3による方法とがある。正常復帰なら単に戻り、次の文（P2を呼び出す）を実行する。信号err1を出すならばP の中のerr1という名の例外処理ルーチン（exception handler）で、P1を呼び出した文を含む最も内側のものに飛ぶ（図2．1で①のところのwhenからend）。err3の時も同じであるが、図2．1では③のところの例外処理ルーチンに飛ぶ。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、エラーが外側に伝ぱする。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。つまりerr1が起こったとすると、例外処理ルーチン実行後はP2の呼び出し実行する。

処理方法としては、手続きが正常に復帰するならば呼んだところに戻る。信号を出したならば信号名により例外処理ルーチンを捜す。例外処理ルーチンは図2．2（図2．1のP）で示すようにリストにし、常に有効な例外処理ルーチンのみをつないでおく。またリストの順序は内側のルーチンから外側のルーチンを指す。つまり、P がP1を呼ぶには図2．2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。これによりリストは小さく、リストの実行時での保守も先頭のみでい。

この方法の利点は単純な方法で、主記憶をあまり必要としないことである。欠点として、例外処理を使用するプログラムは信号が起こらなくても実行時間に多少の負担が必要であり、信号が起これば例外処理ルーチンを捜すために時間がかかることである。

別の方法として、MIT のCLU 処理系［1］では例外処理ルーチンの表（信号名と有効範囲）を設け、信号が起きた時点で表から例外処理ルーチンを捜す。この方が効率が良いが、表の作成で目的プログラムとうまく対応させることで複雑になる。

これまでのところ、例外処理機能はひん繁に使用されるものではなく、信号はめったに起こらないので、ここに述べた方法で十分であろう。

```
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)
```

図2．1例外処理のプログラム

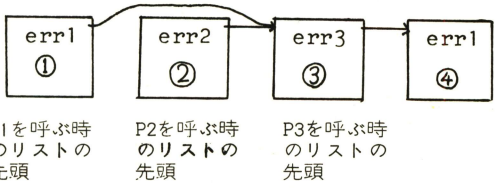


図2．2例外処理ルーチンリスト

### 4．4繰り返し子

繰り返し子についてまず説明する。繰り返し子は手続きと似た形式で定義されるが（図3．1のfromto）、for 文から呼ばれることにより（図3．1の2行目）データ（オブジェクト、図3．1ではinteger 型データ）を繰り返し生成するものである。繰り返し子は呼ばれると手続と同じように実行する。その時yield 文を実行するとyield 文の値（図3．1ではf の持つオブジェクト）を持ち、呼ばれたfor 文に戻り値を変数に代入する（図3．1ではi にオブジェクトが与えられる）。そしてfor 文の中の文（図3．1のQ(i)）を実行する。これが終るとまた繰り返し子のyield 文に戻り、yield 文の次の文（図3．1のf：＝f＋1）以降が実行される。以後これを繰り返す。繰り返し子が終ると（図3．1ではwhile 文が終る時）for 文も終る。図3．1ではQ(1) からQ(10)までが実行される。

処理方法は、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上の先頭（spで示す）と、現在プログラムの使用している最大領域（suで示す）の2つのポインタでおこなう。手続きまたは繰り返し子の呼び出しはsuの次に新しく領域をとり、spをsuの位置に移し、suは使用する大きさに

合わせて移動させる（図３．１の for 文中でfromtoを呼ぶとスタックは図３．２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３．１のQ が終るとスタックは④から③になる。また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に戻す（図３．１のyield 文を実行するとスタックは図３．２の②から③になる）。for 文から繰り返し子に戻るには、やはりspのみを移動させる（図３．１のQ(i)終了後fromtoに戻ると、スタックは図３．２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・結果の受け渡しがこれに加わる。現処理系ではさらに細かい操作をおこない、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、繰り返し子はかなり効率よく実行できる。なお、同様な方法がMIT のCLU 処理系［１］でも用いられている。

```
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
```

図３．１繰り返し子のプログラム

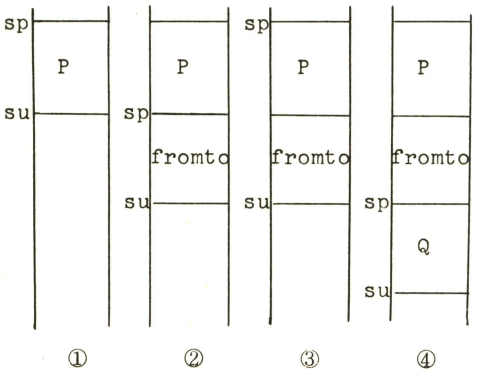


図３．２実行時スタック

## ４．５パラメタつきモジュール

たとえばスタックというデータ型を定義するのにスタックの要素のデータ型によらず同じである。そこでCLU では要素のデータ型をパラメタとすることでinteger型オブジェクトを要素とするスタックstack[int] や、string型オブジェクトを要素とするスタックstack[string]を一度に定義できる。定義の方法は

```
stack = cluster [t : type] is ...
```

とし、要素の型としてstack 定義のクラスタ内では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。このようにモジュールには手続き・繰り返し子・クラスタのいずれにもパラメタをつけられ、これをパラメタつきモジュール（parameterized module、クラスタではtype generatorとも呼ぶ）と呼ぶ。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sum[int](x,y) のように書き表わされる。パラメタとして先の例のデータ型のほか、基本データ型の定数も可能である（上の例のtype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を調べてみると、多くの処理系［３など］ではコンパイル時での一種のマクロ機能として処理されている。一部MIT のCLU 処理系［１］やイオタ処理系［12］では実行時にモジュール結合をおこなっている。

コンパイル時による方法では目的プログラムの効率は良いが、プログラムの段階的な作成がしにくく、コンパイルが繁雑になるといった問題がある。さらにCLU ではこの方法では不可能なプログラムを書ける。（再帰的にパラメタつきモジュールを呼ぶこと、つまり、

```
gt = cluster [t : type] is ...
     ...
     gt[array[t]]
     ...
```

といったプログラムでは無限にマクロ展開することになる。さらにCLU では任意のデータ型オブジェクトを持てるデータ型any があり、any を用いると再帰的用法の有用なプログラムを書ける。）また、先の例でstack[int]やstack[string] のように、stack というモジュールに対して複数のモジュール例（module instance）が生成されることもあり、目的プログラムが大きくなることもある。

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、常に複雑な処理をするため目的プログラムが大きくなり、実行速度も悪くなる。

以上のことを考慮して、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。この方法で処理不可能なプログラムは実質上ほとんどなく、禁止して問題がないと考えた。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あまり変わらず、より小さいことも多い。

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）で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をコンパイルするときにその参照されるパラメタモジュールをコンパイルできる。さらに、その時パラメタつきモジュールのインターフェース仕様のみでなく、実現部が必要となる。そこで次節で述べるライブラリを利用し、これに必要とするモジュール例の種類とモジュールの実現部を格能する。この方法でまた分割コンパイルと段階的プログラム作成の問題を解決する。

## 4．6 ライブラリ

モジュールを管理するためにライブラリ（library）を置く。ライブラリにはモジュールの情報（抽象化情報、インターフェース仕様、実現部）、モジュール間の関係、およびパラメタつきモジュール例の情報を入れる。

処理系の構成としては図4のように、3章で述べた構成（図1）の前にプリプロセッサを置く。モジュールの実現部はコンパイルされる前にまずプリプロセッサによりライブラリに蓄えられる。プリプロセッサはさらに、入力したパラメタのないモジュールはそのまま出力するが、入力したパラメタつきモジュールはすでに参照された形式があれば、パラメタを実際の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。プリプロセッサから出力するモジュールは、もとのモジュール実現部中で参照するパラメタつきモジュールをパラメタごとプリプロセッサの生成した別の名前に置き換えたものである。プリプロセッサ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ー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。コンパイラとFASPによりこのモジュールは相対形式モジュールに変換される。プリプロセッサはまた相対形式モジュールがライブラリのモジュールと対応するよう、相対形式モジュールファイルも管理する。このため図1では個々の相対形式モジュールを別々のファイルに入れたが、図4では1つのファイルにすべて入れる。

ライブラリにはすべてのモジュールが格能されるが、インターフェース仕様のみ（抽象化情報のみ）のモジュールもある。パラメタのないモジュールなら実現部が与えられた段階でコンパイルをすることでよいが、パラメタつきモジュールであれば実現部が与えられるまではそのモジュールの使われかた（パラメタの値）をライブラリに入れ、実現部が与えられた時点で必要なだけのモジュール例を生成し、相対形式モジュールとする。それ以後は参照されるたびに新しいモジュール例が必要であれば生成する。

一般に一つの抽象に対し複数の実現方法が存在し得るが、現時点では一つの実現部しか許さない。複数の実現部を許すには抽象と実現部との対応、実現部の選択といった問題がある。CLU では言語として抽象と実現を分けていないため、何らかの方法で区別できるようにしなくてはならない。当分の間はこの点は処理しない予定である。ただし、プログラム作成者がライブラリ以外で別の実現部を持つことは問題がない。

ライブラリからの抽象化情報の取り出し・変更、プログラム解読・保守のための情報提供、モジュールの取り出し、不必要モジュールの削除などのため、プリプロセッサのほかにライブラリ操作プログラムも作成している。また、テキストエディタで直接ライブラリを扱うことも予定している。

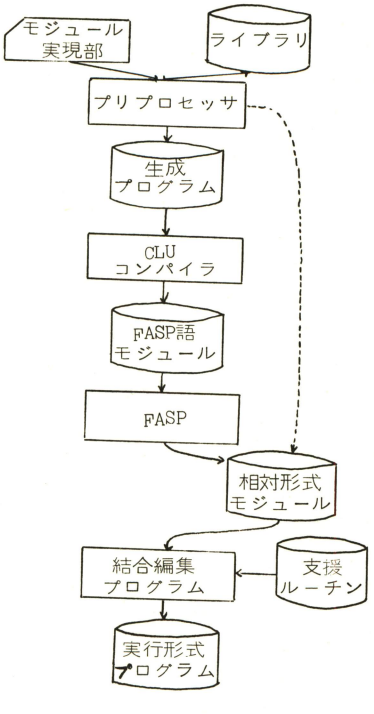


図4 ライブラリつき処理系

プリプロセッサではライブラリのほか、別個のファイルを置く。このファイルにはsortや数値計算プログラムなどを入れておく。プリプロセッサは入力したモジュール中にライブラリにないモジュール参照があれば、このファイルから自動的にライブラリに取り込む。この方法でプログラムの蓄積をする予定である。

なお、プリプロセッサやライブラリ操作プログラムはすべてCLU で書いている。

## 4．7 MIT CLUとの相異点

45S のCLU 処理系は CLU Reference Manual [7] を基にしており、基本的に違いはないが、多少の拡張をおこなった程度で、この点について述べておく。

広域的データ型宣言を導入した。CLU ではモジュール名を除けばすべての名前は各モジュールごとに局所的である。このためモジュール間で情報転送をするにはすべて引数または結果として渡す。この時の転送されるオブジェクトの型はrecord型など複雑なデータ型においても各モジュールごとに型宣言をすることになる。また、他のデータ型と異なる新しいデータ型を作成するにはクラスタにより新データ型を定義する以外にない。この点について、プログラム作成を楽にし、また効率も向上することから、クラスタの省略形という意味で広域的な型宣言を許した。この宣言はインターフェース仕様としておこなうか、ライブラリに抽象として与える。この広域的な型宣言をクラスタに定義しなおすことは、オブジェクト生成（定数と構成子、constractor）を除けばまったく他のモジュールを変更する必要はない。オブジェクト生成はクラスタで操作を定義できないため、定義可能な操作に変更することになる。

予約語の一部について別名を与えた。ifやdoなどの制御文を閉じるにはすべてend が要求される。これらがすべて同名称では読みにくく、誤りも起りやすいので、fi、odなどをend の代りに使用することを許した。その他の予約語についても別名を与えたものがある。

初期のCLU にはあるが、CLU Reference Manualでは削られているrepeat-until文やcase文などを45S では残した。また、Alphard のfirst 文やModula [11] のloop文を取り入れてある。これらはすべてプログラムを書きやすく、見やすくするため、また繰り返し途中での飛び出しを避けるために入れた。

## 4．8計算機に依存するプログラム。

われわれは時に計算機に依存したプログラムを作成する。すでに45S での基本的入出力をおこなう入出力データ型や、bit やbyte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が）を用意してあり、アセンブラのプログラムを書かずにCLU でシステムプログラムの作成がおこなえる。

これらのデータ型の操作はすべてアセンブラで書き、CLU のプログラムからこれを呼び出す形式である。CLU のモジュールをコンパイルするには、そのモジュールの使用する他のモジュールのインターフェース仕様があれば完全なチェックをするのに十分である。そこでアセンブラのプログラムについてもCLU でのモジュールに準じた扱いをし、CLU のコンパイラにはアセンブラのプログラムのインターフェース仕様を与える。またアセンブラのプログラムは別にアセンブルをし、相対形式モジュールとして用意する。ただし、インターフェース仕様とアセンブラのプログラルとの照合はおこなわない。

アセンブラを使用するのは入出力に関することか、bit やbyteの単位でデータを扱うためか、特にプログラムを速くするためである。最後の場合を除けば、これらを扱うことのできる（抽象）データ型があればアセンブラをほとんど使用する必要がない。そこで、上で述べたデータ型を用意したわけだが、そのデータ型を設けてから後はまったくアセンブラを使用していない。

基本データ型は最も計算機に依存するデータ型であると考えられる。上で述べたbit やbyte単位の扱いのできるデータ型をさらに強化し、オブジェクトをかなり自由に扱うことのできるデータ型があれば、基本データ型の操作についてもCLU での記述が可能となる。このためにはほかにrecord型のようなデータ型をCLU のクラスタで記述できることが条件だが、この問題を解決し、CLU の基本データ型の操作をCLU で書けば、CLUの支援ルーチンのFASP語部分は2000行程度になる。

## 5．効率

現時点での処理系の実行効率について簡単に述べておく。実行時間は廃品回収のため実行時のプログラムの大きさにかなり影響し、正確には求まらない。だがこれまでのところ、一般にPascalのプログラムと比べておよそ1～4倍の時間を要する。ほとんどの場合、オブジェクトの使用度に比例する。string型データを扱うプログラムはstringの性質からオブジェクトを多く生成するため効率が悪いが、array 型やrecord型ではオブジェクトの生成は多くともその後の操作ではオブジェクトの生成をしないためそれほど悪くならない。効率を向上させるには記憶管理ルーチンの効率を向上させるほか、オブジェクトの生成回数を減らす方法もある。後者については3個の文字列の連結を一度のオブジェクト生成ですますとか、定数要素のみのarrayやrecord型のオブジェクトを生成しないことなどがある。これらを手作業による最適化の実験や効率の計算などの結果から、CLU によるプログラム作成のしやす

さを考慮すれば、現処理系の効率を多少向上させることで実行効率の問題はなくなると考える。

なお、これ以外のたとえば手続き呼び出しの損失はどの言語にも共通であり、CLU の欠点とはならない。

## 6. CLUを使用して

これまでにCLU で書いたプログラムはテキストエディタ（約1000行）、漢字マクロプロセッサなどがある。CLU を使用しているものは現在数人であるが、序々に増えている。使用者の多くはまだ始めたばかりであり、小さなプログラムが多いが、今語前述のCLU プリプロセッサを含め 2〜5000行のプログラムが作成されるはずである。

本章ではこれまでの経験に基づいてCLU の検討をする。なお、文献［13］も参照されたい。

プログラムを作成するには上手な設計が必用である。上手な設計であればモジュール分けが楽で保守性・読みやすさにすぐれたプログラムとなる。CLU では特に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、慣れるに従い自然にプログラムを作成できる。

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、実行時での誤りは少ない。実行時に起こる誤りも、モジュールが完全に閉じており、他のモジュールの影響を受けにくので、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要易である。一般にCLUで書くプログラムは完全にモジュール分けをし、特にクラスタではすべての操作を明確に記すこと、および暗黙の型変換やデータ型操作の自動選択しないことなどから、プログラムが長くなる傾向にあるが、コンパイル時の誤り発見の機会が増し、プログラムが読みやすくなる。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。

だがそのままでは不便なところもある。まず、内部モジュールがなく、広域的変数もないため、モジュール数が増加することである。不便ではあるが上で述べた利点に貢献しており、ライブラリと4．7節で述べた広域的データ型宣言を許すことで不便さを解消している。

抽象データ型の定数が作れない不便さもある。CLU に数え上げ型（enumeration type）がないことにもよるが、いくつかの状態を示す抽象データ型を定義するにはオブジェクトを生成するための操作を数多く定義せねばならない。また、抽象データ型で一定の状態を表わすオブジェクトは同じく操作を定義し、それを実行することで使用することになる。定数は操作の代用とも考えられ、この方が保守しやすく修正も楽であるが、読みやすさと効率の点で劣る。そこでAlphard や Mesaの共有定数（shared constant）のような方法を検討している。

総じてCLU は強力な機能を持つが、信頼性を高めるため常に制限をした用法のみを許す言語であるといえる。

## 7. おわりに

われわれはCLU 処理系に作成し、使用しているが、今までのところ実用のプログラム作成用言語として十分な結果を得ている。これまでに述べたことのほか、以下に述べる作業をおこなっている。

### 7．1 HITAC M-180への移植

現処理系は移植性をあまり考慮せず作成したが、現在M-180 への移植作業をおこなっている*。この作業では45S での処理系を大幅に変更することになるが、45Sでの処理系と同程度の機能を持ち、入出力や整数の大きさなど一部を除き45S と互換性のあるプログラムが作成できる処理系となるよう考慮している。4章で述べたプリプロセッサはそのままM-180 に移行できるわけである。

### 7．2 並列処理の導入

今後のプログラムには並列処理が要求されることもあり、CLU に並列処理機能を入れるため、現在この外部仕様を検討中である。

### 7．3 新処理

4章で述べたプリプロセッサはCLU で書いている。これに目的コード生成ルーチンなどを組み込むことで新コンパイラを作成する予定である。この作業はプリプロセッサ完成以後となるので、まだほとんどおこなっていない。

この場合、コンパイラは移植性を考慮し、基本データ型操作などもアセンブラで書くことはせず、中間言語あるいはCLU で直接書く。またこの時、今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。

CLUでコンパイラを書くことにより10000 行程度のCLU プログラムを作成してみるわけでもある。

謝辞

日ごろ御指導いただいている木村泉助教授、本稿作成での細部にわたる助言をいただいた米沢明憲助手、角田博保氏、また処理系作成での御協力をいただいた木村研究室の方々に感謝いたします。

* この作業は吉田健一［15］がおこなっている。

参考文献


[ 1] Atkinson, R., Liskov, B., and Sheifler, R., Aspects of Implementing CLU, *Proceedings of the ACM 1978 Annual Conference.*

[ 2] Geschke, C. M., Morris, J. H., and Satterwaite, E. H., Early Experience with Mesa, *Comm. ACM* 20, 8 (Aug. 1977), 540-553.

[ 3] Gries, D., and Gehani, N., Some Ideas on Data Types in High-Level Language, *Comm. ACM* 20, 6 (June 1977), 414-420.

[ 4] Lampson, B. W., et al., Report on the Programming Language Euclid, *SIGPLAN Notices* 12, 2 (Feb. 1977), 1-79.

[ 5] Liskov, B. H., and Zilles, S. N., Programming with Abstraction Data Types, *Proc. ACM SIGPLAN Cong. on Very High Level Languages, SIGPLAN Notices* 9, 4 (April 1974), 50-59.

[ 6] Liskov, B. H., Snyder, A., Atkinson, R., and Schaffert, C., Abstraction Mechanisms in CLU., *Comm. ACM* 20, 8 (Aug. 1977), 564-576.

[ 7] Liskov, B. H., et al., *CLU Reference Manual,* Comput. Structures Group Memo 161, Lab (July 1978).

[ 8] Parnas, D. L., A Technique for the Specification of Software Modules with Examples, *Comm. ACM* 15, 5 (May 1972) 330-336.

[ 9] Shaw, M., Wulf, W. A., and London, R. L., Abstracton and Verification in Alphard: Defining and Specifying Iteration and Generators, *Comm. ACM* 20, 8 (Aug. 1977), 553-564.

[10] Wirth, N., The Programming Language PASCAL: *Acta Informatica* 1 (1971), 221-227.

[11] Wirth, N., Modula: a Language for Modular Multiprogramming, *Software Practice & Experience* 7, 1 (1977).

[12] Yuasa, T., *Module-wise Compilaton for a Language with Type-parametrization Mechanism,* RIMS-280 (1979).

[13] 角田、佐渡、関根、CLU を使ってみて、よいプログラムを作るにはシンポジウム報告集（1979年）。

[14] 佐渡一広、FACOM230-45S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（1979年3月）。

[15] 吉田健一、CLU処理系のHITAC M-180への移植について（仮題）、卒業論文（1980年3月提出予定）。

本 PDF ファイルは 1980 年発行の「第 21 回プログラミング・シンポジウム報告集」をスキャンし、項目ごとに整理して、情報処理学会電子図書館「情報学広場」に掲載するものです。

この出版物は情報処理学会への著作権譲渡がなされていませんが、情報処理学会公式 Web サイトの https://www.ipsj.or.jp/topics/Past_reports.html に下記「過去のプログラミング・シンポジウム報告集の利用許諾について」を掲載して、権利者の捜索をおこないました。そのうえで同意をいただいたもの、お申し出のなかったものを掲載しています。

過去のプログラミング・シンポジウム報告集の利用許諾について

情報処理学会発行の出版物著作権は平成 12 年から情報処理学会著作権規程に従い、学会に帰属することになっています。

プログラミング・シンポジウムの報告集は、情報処理学会と設立の事情が異なるため、この改訂がシンポジウム内部で徹底しておらず、情報処理学会の他の出版物が情報学広場 (=情報処理学会電子図書館) で公開されているにも拘らず、古い報告集には公開されていないものが少からずありました。

プログラミング・シンポジウムは昭和 59 年に情報処理学会の一部門になりましたが、それ以前の報告集も含め、この度学会の他の出版物と同様の扱いにしたいと考えます。過去のすべての報告集の論文について、著作権者（論文を執筆された故人の相続人）を探し出して利用許諾に関する同意を頂くことは困難ですので、一定期間の権利者捜索の努力をしたうえで、著作権者が見つからない場合も論文を情報学広場に掲載させていただきたいと思います。その後、著作権者が発見され、情報学広場への掲載の継続に同意が得られなかった場合には、当該論文については、掲載を停止致します。

この措置にご意見のある方は、プログラミング・シンポジウムの辻尚史運営委員長 (tsuji@math.s.chiba-u.ac.jp) までお申し出ください。

加えて、著作権者について情報をお持ちの方は事務局まで情報をお寄せくださいますようお願い申し上げます。

期間：2020 年 12 月 18 日 〜 2021 年 3 月 19 日
掲載日：2020 年 12 月 18 日

プログラミング・シンポジウム委員会

情報処理学会著作権規程
https://www.ipsj.or.jp/copyright/ronbun/copyright.html