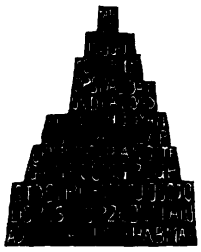

19. 抽象データ型言語†

佐渡一広††　米澤明憲††

1. はじめに

抽象化(abstraction)は信頼性が高く，理解・変更・保守が容易なプログラムを作成する上で，極めて有効な手段を提供するものである．本解説では，抽象化の概念と，その機能を持つ言語について述べる．なお，この分野の解説としては文献31)がある．また文献18)には抽象化にもとづくプログラミング方法論が手ぎわよくまとめられ，よい例題も示されている．

2. 抽象化

2.1 抽象化について

1970年前後に，

- Hoare[8]，Morris[21]らのデータ構造に対する概念，
- Dijkstra の構造的プログラミング (structured programming)[4]，Wirth の段階的詳細化 (stepwise refinement)[32]，
- 構造的なデータ型 (data type) の概念を持つ簡素な言語 Pascal，
- データと手続きをまとめるカプセル化機能 (encapsulation)，およびそのまとまりを対象物 (object) として扱う概念を持つ言語 SIMULA，
- Algol N[30] などの拡張可能言語[29]

などがあらわれてきた．これらを背景にして，構造的プログラミングにもとづき問題を段階的に詳細化するときの各段階を「抽象化のレベル (levels of abstraction)」と呼ぶようになった[13]．さらに複数のデータとこれらにアクセスする複数の手続きを1つにまとめるカプセル化機能を持つモジュール構造に抽象化のレベルを導入することで，データの抽象化(data abstraction)，すなわち抽象データ型 (abstract data type) の概念が作られた．また抽象データ型は情報隠蔽 (information hiding)[24] や局所化 (locality) の効果も合わせて持っている．この抽象データ型をプログラム言語に取り入れることが Liskov によって提案された[13]．

抽象データ型とは，データの内部構造を外部から隠し，目的に応じて定義された操作だけをそのデータに対する外部からのアクセス手段として許し，これら以外の方法で内部構造をアクセスすることを禁止したデータ型である．これによりプログラムの設計・実現にデータの内部構造を考慮せず，外部的なふるまいだけを注目して問題分割 (problem decomposition) をする手段を与える．同時に内部のデータが不当に書き替えられることがないから，安全性も保証される．この結果，抽象化レベルの各階層ごとに正しく具体化すれば，全体として正しいプログラムを作成することができる．このために，下位レベルの挙動は上位レベルの状態によらずに，陽に渡された引数等のみによって結果が定められることが原則である．

例としてスタックを考えてみる．スタックの機能はデータをしまう入れ物と，それにアクセスする push と pop という2つの操作で特徴づけられる．これらを1つにまとめたものが抽象データ型 stack となる．つまり入れ物は stack 型 (の対象物) として扱い，入れ物にアクセスするには push または pop のいずれかの操作によってのみおこなう．この結果，stack 型と push, pop の役割さえ知ればスタックの機能を利用する場合十分であって，スタックの実現方法は何にも知らなくてよい．またこれ以外の方法でスタックの内部構造にアクセスできない．

抽象データ型によるプログラム作成とは，抽象データ型を中心にしてプログラムのモジュール化，階層化をおこなうものである．

一方，Fortran の副プログラム，Algol の手続きといった，プログラムの手続き部分を別に置き，引数をつけてそれを呼び出し実行する，という機能は古くから使われている．このような機能は手続きの抽象化 (procedural abstraction) と呼ぶことができる．

† A Tutorial on Abstract Data Type Oriented Languages by Kazuhiro SADO and Akinori YONEZAWA (Department of Information Science, Tokyo Institute of Technology).

†† 東京工業大学理学部情報科学科

```
stack=cluster [t: type] is create, push, pop
      rep=array [t]
  create=proc ( ) returns (cvt)
         return (rep$new( ))
         end create
  push=proc (s: cvt, x: t)
       rep$addh(s, x)
       end push
  pop=proc (s: cvt) returns (t) signals (empty)
      if rep$empty(s)
        then signal empty
        else return (rep$remh(s))
        end
      end pop
end stack
```

a. CLU

```
type Stack (StackSize: unsignedInt)=module
     exports (Pop, Push)
     var IntStack: array 1..StackSize of signedInt
     var StackPtr: 0..StackSize=0
     procedure Push (X: signedInt)=.....
     procedure Pop (var X: signedInt)=.....
     end Stack
```

b. Euclid (方法 1, 概略)

```
type Stack=module
     exports (Stk, Pop, Push)
     type Stk (StackSize: unsigendInt)=record
          var StackPtr: 0..StackSize=0
          var Body: array 1..StackSize of signedInt
          end Stk
     procedure Push (var Istk: Stk (parameter),
                     X: signedInt)=.....
     procedure Pop (var Istk: Stk (parameter),
                    var X: signedInt)=.....
     end Stack
```

c. Euclid (方法 2, 概略)

図-1 Stack の定義

2.2 抽象データ型の言語表現

本節では CLU 語[15] および Euclid 語[3] における抽象データ型の表現について簡単に述べる．詳しくはそれぞれの文献を参考にされたい．

CLU における抽象データ型 stack の例を図-1 a. に示す．抽象データ型の定義は，**cluster** という，データ型の内部構造とそのデータ型に対してアクセスできる操作をひとまとめにして定義する構文機構を用いておこなう．図-1 a. の1行目は stack という名と外部から見える3つの操作 create, push, pop を提示している．2行目は内部構造（名前 **rep** に定義されたデータ型）を配列として定義しているが，外部からは見えない．3行目，6行目，9行目で3つの操作の実体を定義している．(addh, remh は配列を伸縮させる操作である．) 外部からは抽象データ型 **stack** とその3つの操作のみ見える．stack 型の変数 **s** は

s: stack[**int**]

のように定義する．**int** はスタックの要素の型を示している*．CLU では変数は対象物を指す名前として扱われており**，stack 型対象物は **create** という操作で作られる．s の指す対象物へのアクセスは push または pop を stack[**int**]$push(s, x) のように記すことでおこなう．また stack 型は組み込みのデータ型とまったく同じに扱える．このような cluster が1つのモジュールをなし，1つの抽象データ型を定義している．

Euclid では CLU と同じくモジュールが抽象データ型を定義するのに使われる．図-1 b. に Stack 型の定義の一部を示す．モジュール内のルーチン，変数，データ型宣言は **exports** 文で宣言することによって外部から使えるようにすることができる（2行目）．変数は

var S: Stack(100)

として宣言することで最大長が 100 の Stack 型変数 S とその対象物が作られる (CLU とは異なっている)．変数 S へのアクセスは S. Push(x) のように記す．また Euclid では図-1 c. に示すように Stack を定義することもできる．この場合はまず

var S: Stack

と宣言し，次に

var A: S. Stk(100)

として A という最大長 100 のスタックが作られる．この場合 S. Stk が抽象データ型となっている．外部からは S. Stk の構造は見えない．変数 A へのアクセスは S. Push (A, x) のように記す．図-1 c. の方法のようにデータ型をモジュールの外から見えるようにすれば，1つのモジュールで複数の抽象データ型を定義することが可能である．

2.3 制御の抽象化

抽象化の対象として，これまでに述べてきた手続きおよびデータのほかに，プログラムの制御がある．これは if 文や while 文等のようにプログラムの流れを制御するもののことで，制御の抽象化 (control abstraction) と呼ばれる．使用者が制御の抽象化をおこなう手段としては，繰り返しに対する抽象化 (iterator

* 2.5 節参照
** 3 章参照

```
from_to=iter (from, to: int) yields (int)
        while from <= to do
              yield (from)
              from := from+1
              end
        end from_to

for i: int in from_to(1, 10) do
  a[i] := b[i]+c[i]
  end
```

図-2 CLU の繰り返し子

abstraction) がある.

CLU の制御の抽象化機能である繰り返し子 (iterator) の例を図-2に示す. 繰り返し子は **for** 文の入口で呼び出され, 手続きのように実行する. **yield** 文が実行されるたびに呼び出し側にデータを送り返すことにより **for** 文の本体を実行する, という過程を繰り返すコルーチン (coroutine) のような機能である.

2.4 効用と評価

抽象データ型によるプログラム作成の利点としては次のことが上げられる. 抽象データ型によりプログラムの構造をデータを中心として自然に構成できる. たとえば文書整形を行うプログラムを作成するのに page, line, word といった抽象データ型を定義しておけば, これらを用いて問題に即した自然な形でプログラムの階層化とモジュール分割をおこなえる. さらに, 抽象データ型ではデータの内部構造は外部から隠されており, またデータへのアクセスは特定の操作を用いることによってのみおこなわれることから, データの外部的な挙動のみからモジュールの性質を把握できる. このため, プログラム作成では各モジュールの仕様を段階的に決めていき, あとは各モジュールごとに仕様にあわせて実現させていけばよい. 同時に各モジュールの相互依存関係が明らかになり, モジュールを単独に扱うことが容易である. プログラムの正当性についても個々のモジュールごとに調べればよいことから検証しやすく, 信頼性も高くなる. また変更や保守も必要な部分に関する少数のモジュールに対しておこなうだけですむ.

一方, 完全なカプセル化やモジュール化のために, データへのアクセスがその抽象データ型に随伴する操作のみを使用するので, 冗長性やオーバヘッドが多くなり, 実行効率が悪くなることがある. しかし, CLU に関して文献 27) に示されているように, 動的対象物を扱う場合においても Fortran や Pascal と比較して実行効率は一般に 1～3 倍程度であるので, 上で述べた抽象化にもとづくプログラム設計・作成の効用が損われることは少ない.

2.5 抽象データ型に関連する事柄

● データ型のチェック

抽象データ型を支援するプログラム言語は, コンパイル時にデータ型の不一致をチェックすることでデータの誤った使用の検出をおこなう.

さらにデータ型の変数にアクセスの制御 (access control) をつけることでデータの不正使用をコンパイル時に検出することを強化する試みもある[9]. これは変数ごとに, そのデータ型に適用できる操作を制限することでデータ保護などをおこなうものである. Alphard[33] には取り入れられているが, その他の言語には実装されていない.

● 対象物

2.1 節における抽象データ型の説明にあるように, 抽象データ型を持つデータあるいはモジュールは対象物として扱われることがある. これは ACTOR 理論[7),34)]における対象物から並列性を除いたものの具体化とも考えられる. また対象物を動的に扱うことでデータを統一して扱うことができ, 分割コンパイルやパラメタつきモジュールの処理に有効である.

● パラメタつきモジュール

抽象化をより有効に利用するため, パラメタつきモジュール (parameterized module), 汎用モジュール (generic module), 型生成子 (type generator) などと呼ばれる, モジュールにパラメタ (データ型や定数等) を陽にあるいは陰につける機能を持つものが多い. 図-1 a. の stack では要素の型 t をパラメタに, 図-1 b. と c では大きさとして StackSize をパラメタにしている.

● 例外処理

構造的にプログラムを作成する場合, 異常事態 (入力の誤り, ファイルの終りなど) の処理が問題となる. このために例外処理機能 (exception handling)[12] を持つものがある. (図-1 a. の **Signal**, **Signals** はその例.) 例外処理機能を用いることで抽象化がより効果的におこなえる.

● 仕様記述

抽象化を用いて段階的に詳細化をしていく場合, 各レベルの仕様を記述することが重要な仕事である[14]. このため, 抽象データ型に関する仕様技法の研究は活発で, これまでに, 仕様記述方法として抽象モデル法[33),35)], 公理的方法[6),36)], 状態機械法[24)]が考えられ

ている.

3. 抽象データ型言語

- CLU

抽象化を支援するために M. I. T. の Liskov を中心として設計された. 初期の設計思想は文献 13) にある. その後言語仕様は変更され, 最終版は文献 16) に発表された. 言語仕様の概略を知るには文献 15) がよい.

CLU は手続きの抽象をする procedure, 制御の抽象をする iterator, およびデータの抽象化をする cluster の3種の抽象化を用いてプログラム作成をする. このとき, 個々の抽象化を定義するプログラムが1つのモジュールを形成している.

変数はすべて実行時に生成される対象物を指す名前と考え, 変数への代入はその指示対象物を与えることを意味する. これによって対象物の共有 (sharing) が起こる.

基本データ型は抽象データ型と同様に扱われる. 通常のインフィックス (infix) 記法や, 配列・レコードに対する操作は, それぞれ特定の名前の操作の略記法 (syntax sugaring) として使用できる (たとえば T$add (a, b) は a+b とも記せる). それゆえ使用者の定義した抽象データ型でもインフィックス記法が使用できる. この部分には多重定義 (overloading) が可能である. このほか, 例外処理機能, 分割コンパイル機能などがある.

CLU の最終言語仕様[16]は初期[13]のものから抽象化機能の強化(繰り返し子, 略記法の拡張など), 効率の向上, といった方向で改良されてきた. 最終的には使い方によっては, 「外部から見た抽象データ型の特性は一定である」という原則を破ることが可能な占有変数 (own variable) が取り入れられた. この理由は, 資源の排他利用の制御などに占有変数が必要となるためである.

- Euclid

Pascal をもとにし, 検証可能性を重視して設計したものである. 言語仕様は文献 11) に示されている. 抽象データ型については文献 3) に記述がある.

カプセル化を支援するためのモジュール定義がこの言語の中心的機能であり, モジュールの可視部の制御ができる. また値の別名 (aliasing) を禁止していることも特徴である.

可視部の制御の方法は, **exports** 文にモジュール内のルーチン, 変数, あるいはデータ型の名前を記すことで外部からアクセス可能にし, またアクセスする側は **imports** 文で使用するルーチン, 変数, データ型の名前を明記する. このように可視部を制御することで抽象データ型が定義できるが, 大域変数を使用したり, モジュール内の変数を外部からアクセスしたりできるため, 抽象データ型にとらわれないプログラム作成ができる.

制御の抽象化として generator を持つ. また各モジュールには初期設定用と後始末用のルーチン (データ生成時あるいは消滅時に実行される) が書ける. ルーチンのインライン展開 (inline substitution) の指定ができる. さらにシステム記述用に機械依存部分を記述することもできる.

- Mesa

システム記述用に Xerox の Palo Alto 研究所で開発された. 言語仕様は文献 20) にあるが, 概略としては文献 5) が参考になる.

カプセル化されたモジュールを持ち, モジュールはインタフェース部と実現部の2つに分かれる. インタフェース部と実現部との対応は C/Mesa と呼ぶモジュール間の制御言語で記述する. 1つのインタフェース部に対し, 複数の実現部を置くことができる.

インタフェース部に記述された部分 (手続き, 変数, データ型, 例外等) のみ外部からアクセスできる. モジュールは暗黙には1つの対象物となっているが, 実行中に新しくモジュールの対象物を生成したり, すでにある対象物の複製を作ることもできる. これらの機能を用いることでモジュールを抽象データ型として扱える.

またインタフェース部にはデータ型名だけを書くことが可能なので, **図-1 c.** に示した Euclid と同じ形式で抽象データ型を定義することもできる.

このほか, 型をもたないデータやビットパターンとしてデータ型変換をおこなえるようにするため, 厳格なデータ型からの抜け穴 (loophole) を持つ. これはシステム記述のため (たとえばプログラムの目的コードを扱うとき), 極端な効率の低下を防ぐため, あるいは1回抜け穴を通ることでより厳しいデータ型チェックをおこなえることがあるからである. さらに, コルーチン機能, 例外処理機能, モニタ (monitor) による並列処理機能も備わっている.

一般に, Mesa では一様参照性 (uniform reference) を重視したデータ構造を採用している. これにより手

続き，データ，例外処理，並列処理を統一して扱っている．

- Iota[22]

プログラムを抽象化のレベルに従って階層的に記述して検証することを目的として設計された．仕様・検証には多ソート一階述語論理 (many sorted first order logic) を用いている．仕様記述とプログラム記述の2つの言語からなる．

- Alphard[33]

CLU と並行して CMU の Wulf, Shaw らによって設計され，抽象化支援のほか仕様記述，検証規則を与えているが，いまだ未完成で処理系もない．

4. 処理系と使用状況

CLU の処理系は，M. I. T. で PDP 10 の上にまず Lisp によって実現された．その後 CLU 自身によるセルフコンパイル (self compile) をする処理系が作られ，現在 M. I. T. ほか数ヵ所で稼働している (DEC 20 シリーズで動く処理系は M. I. T. から磁気テープで入手可能)．この処理系における例外処理，繰り返し子，パラメタつきモジュールの処理方法が文献[2]にある．また CLU で書かれたプログラムに対する最適化の手法の研究もある[1),28)]．日本では FACOM 230-45 S/OS II の上に Pascal によって記述され稼働しており[26),27)]，日常的なプログラム作成に利用されている[10)]．

Mesa の処理系は Alto 計算機上でマイクロプログラムを用いて実現されている．データや手続きの目的コードをはじめ，実行用のスタックも対象物として扱われている．また Mesa で書かれた OS として Pilot[25)] がある．Pilot 作成によって得られた Mesa の使用経験の報告[19)]がある．

Iota の処理系は DEC system 20 の上に Utah Lisp を用いて作成されており[23)]，教育用などに使用されはじめている．また IBM 360/370 系の計算機用処理系もある．

5. 今後の見通し

CLU に関してはその開発はほぼ終っている．Liskov らは CLU をもとにして，分散処理プログラム支援のための言語機能[17)]と，抽象データ型による通信手段の研究をおこなっている．また移植性の高い CLU 処理系の作成や，分散処理のためのマイクロコンピュータ用処理系の作成もおこなっている．

Mesa は Xerox におけるプログラム開発用として実用化されている．Iota はプログラム作成支援環境の作成がなされている．

今後設計されるプログラム言語は，大なり小なり抽象化の概念を取り入れていくことは確実であろう．ちなみに Ada でも擬似的ではあるが抽象データ型を作る機能がある．

現在の計算機のアーキテクチャでは抽象データ型の処理をするのに不都合な点がある (対象物の動的な処理から派生する諸問題)．これには現在研究・開発の盛んな対象物指向型の計算機アーキテクチャが有効である[25)]．この結果，完全な抽象化によるプログラム作成は十分に実用的なものになろう．また，これまでに作られた抽象データ型言語の成果は，今後のプログラミング方法論にさらに大きな影響を与えるものと考えられる．

参考文献

1) Atkinson, R. R.: Optimization Techniques for a Structured Programming Language, S. M. Thesis, Dept. of Electr. Eng. and Comptr. Sci., M. I. T. (Jun. 1976).
2) Atkinson, R. R., Liskov, B. H. and Scheifler, R. W.: Aspects of implementing CLU, Proc. of the ACM 1978 Annual Conf., pp. 123-129 (1978).
3) Chang, E., Kaden, N. E. and Elliott, W. D.: Abstract Data Types in EUCLID, SIGPLAN Notices, Vol. 13, No. 3, pp. 34-42 (Mar. 1978).
4) Dijkstra, E. W.: Notes on Structured Programming in Structured Programming, Academic Press (1972).
5) Geschke, C. M., Morris, J. H. and Satterthwaite, E. H.: Early Experience with Mesa, Commun. ACM, Vol. 20, No. 8, pp. 540-553 (Aug. 1977).
6) Guttag, J. V., Horowitz, E. and Musser, D. R.: Abstract Data Types and Software Validation, Commun. ACM, Vol. 21, No. 12, pp. 1048-1064 (Dec. 1978).
7) Hewitt, C.: Viewing Control Structure as Patterns of Passing Messages, J. Artificial Intelligence, pp. 324-364 (1977).
8) Hoare, C. A. R.: Note on Data Structuring in Structured Programming, Academic Press (1972).
9) Jones, A. K. and Liskov, B. H.: A Language Extension for Expressing Constraints on Data Access, Commun. ACM, Vol. 21, No. 5, pp. 358-367 (May 1978).
10) 角田博保，佐渡一広，関根 裕：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, pp. 7-10 (1979年).
11) Lampson, B. W. et al.: Report on the Programming Language EUCLID, SIGPLAN Notices, Vol. 12, No. 2, pp. 1-79 (Feb. 1977).
12) Levin, R.: Program Structure for Exceptional Condition Handling, Ph. D. thesis, Comptr. Sci. Dep., Carnegie-Mellon Univ. (1977).
13) Liskov, B. H. and Zilles, S. N.: Programming with Abstract Data Types, Proc. ACM SIGPLAN Conf. on Very High Level Languages, SIGPLAN Notices, Vol. 9, No. 4, pp. 50-59 (Apr. 1974).
14) Liskov, B. H. and Zilles, S. N.: Specification Techniques for Data Abstractions, IEEE Trans. Software Eng. SE-1, No. 1, pp. 7-19 (Mar. 1975).
15) Liskov, B. H., Snyder, A., Atkinson, R. R. and Schaffert, C.: Abstraction Mechanisms in CLU, Commun. ACM, Vol. 20, No. 8, pp. 564-576 (Aug. 1977).
16) Liskov, B. H. et al.: CLU Reference Manual, Rep. TR-225 Lab. for Comptr. Sci., M. I. T. (1979).
17) Liskov, B. H.: Primitives for Distributed Computing, ACM Operating System Conference, pp. 33-42 (1979).
18) Liskov, B. H.: Modular Program Construction Using Abstractions, in Abstract Software Specifications, Lect. Notes in Comptr. Sci., Vol. 86, pp. 354-389, Springer-Verlag (1980).
19) Lauer, H. C. and Satterthwaite, E. H.: The Impact of Mesa on System Design, Proc. of 4th Inter. Conf. on Software Eng., pp. 174-182 (1979).
20) Mitchell, J. G., Mayburg W. and Sweet, R.: Mesa Language Manual, XEROX Palo Alto Research Center CSL-78-1 (Feb. 1978).
21) Morris, J. H.: Toward More Flexible Type Systems, Lect. Notes in Comptr. Sci., Vol. 19, pp. 377-384, Springer-Verlag (1974).
22) Nakajima, R., Honda, H. and Nakahara, H.: Hierarchical Program Specification and Verification—A Many Sorted Logic Approach, Acta Informatica, Vol. 14, Fasc. 2 (Aug. 1980).
23) Nakajima, R., Yuasa, T. and Kojima, K.: The IOTA Programming System—A Support System for Hierarchical and Modular Programming, Proc. of IFIP Cong. 80, Tokyo, pp. 299-304 (1980).
24) Parnas, D. L.: A Technique for Software Module Specification with Examples, Commun. ACM, Vol. 15, No. 5, pp. 330-336 (May 1972).
25) Redell, D. D. et al.: Pilot: An Operating System for a Personal Computer, Commun. ACM, Vol. 23, No. 2, pp. 81-92 (Feb. 1980).
26) 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年1月).
27) 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, 情報処理論文誌, 22巻4号 (掲載予定).
28) Scheifler, R. W.: An Analysis of Inline Substitution for a Structured Programming Language, Commun. ACM, Vol. 20, No. 9, pp. 647-654 (Sep. 1977).
29) Schuman, S. A. and Jorrand, P.: Definition Mechanisms in Extensible Programming Languages, Proc. of the AFIPS, Vol. 37, pp. 9-19 (1970).
30) 島内剛一: プログラム言語論, 共立出版, p. 205 (1972).
31) 鳥居宏次, 二木厚吉, 真野芳久: プログラミング方法論の展望, 情報処理, 20巻1号, pp. 22-43 (1979年1月).
32) Wirth, N.: Program Development by Stepwise Refinement, Commun. ACM, Vol. 14, No. 4, pp. 221-227 (Apr. 1971).
33) Wulf, W. A., London, R. L. and Shaw, M.: An Introduction to the Construction and Verification of Alphard Programs, IEEE Trans. Software Eng. SE-2, No. 4, pp. 253-265 (Dec. 1976).
34) 米澤明憲: ACTOR 理論について, 情報処理, 20巻7号, pp. 580-589 (1979年7月).
35) Yonezawa, A.: Specification Techniques for Abstract Data Types with Parallelism in Mathematical Studies of Information Processing, Lect. Notes in Comptr. Sci., Vol. 75, pp. 127-150, Springer-verlag (1979).
36) Zilles, S. N.: An Introduction to Data Algebras, in Abstract Software Specification, Lect. Notes in Comptr. Sci., Vol. 86, pp. 248-272, Springer-Verlag (1980).

(昭和56年3月2日受付)