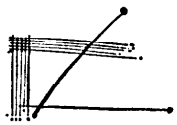
展　望



プログラミング方法論の展望†

鳥居　宏次††　二木　厚吉††　真野　芳久††

1. まえがき

システム作成，ソフトウェア作成，プログラム作成などと呼ばれて，何とはなしに区別して使われている作成技術のうち本稿では，プログラム作成に焦点を当てる．ここで，これらの区別として規模の大きさを1つの目安と考えている．前二者を議論するには，要求定義やその分析，管理，保守の他に，純粋に技術論では片付けられない人的資源の運用に至るまでの広範囲の要素を考慮しなければならない．本稿の対象範囲はこれらの要素以外についてであり，歴史的流れをふり返りながら，基本概念，プログラミング方法論，仕様およびプログラム記述法，さらに検証法について最近の成果と近い将来の動向を展望する．

2.ではまずプログラム作成における重要な基本概念について述べる．次にそれらの諸概念と密接に関連付けられる方法論をよく知られた「電報の編集問題」を例題として具体的に示す．

3.では，**2.**で述べた基本的な概念がプログラミング言語の上にどのように反映されているかを考慮しつつ，方法論を意識して設計された最近の言語の特徴を考える．言語の表記法が必ずしも単なる思いつきから決められるものではないことが，うかがい知れるであろう．

4.では仕様記述法とくに，形式的仕様記述法について述べる．現実のプログラム作成における仕様は未だ自然言語によっている場合が多い．しかし，プログラムが正しく動くことを保証（検証(verify)）しようとするとき，何に対して正しく記述されているかの「何」，すなわち「原器」が存在しなくてはならない．その原器に当たるものが仕様であり，当然のことながら仕様は，正確かつ厳密に述べられていなくてはならない．このような背景から注目されているのが形式的仕様記述である．

プログラム作成過程（あるいはソフトウェアのライフサイクル）から考えれば，プログラムの設計やプログラミング言語による表現（コーディング）の前段階として，仕様記述が位置する筈である．しかし，本稿では**2.**と**3.**との密接な関係を強調すること，および，仕様記述が**5.**の検証法と直接関連することから，**3.**と**4.**とが内容的に前後している．

5.では代表的ないくつかの検証法を述べる．やはり**2.**で基本概念としてとりあげた抽象化が，最近の検証法の中でとくに重視され，抽象データ型に基づくものが多くなっている．

2. 方法論

2.1 諸概念

理想的なプログラムと呼ばれるものが備えるべき要因には信頼性 (reliability)，わかり易さ (understandability)，効率 (efficiency)，変更し易さ (modifiability)，が基本的と言われる．細かいことをいえば，更に多くのキーワードがつけ加えられるべきであろう．一般的表現をすれば，プログラムの構造が重要であり，1つは独立性の高い要素群へ階層的に分割すること (hierarchical decomposition) で，今1つは各要素がいわゆる，整構造 (well structured) になっていることである．ここで，階層構造および整構造を持つプログラムは全体として，最低限上記要因を満たすものでなければならない．

この目的に向かって，諸技術が開発されてきた．しかも，それらに本質的な原理，又は，概念が明確になってきている．むしろ，うまく本質を表現した言葉が発見されてきたという方が適当かもしれない．目下のところでは，

（a）　モジュール化 (modularization)
（b）　抽象化 (abstraction)
（c）　情報隠蔽 (information hiding)
（d）　局所性 (localization)

が代表的である．現実には，この原理を促進するため

† Trends in Programming Methodology by Koji TORII, Kokichi FUTATSUGI, Yoshihisa MANO (Computer Science Division, Electrotechnical Laboratory).

†† 電子技術総合研究所　ソフトウェア部

に，それぞれ，モジュール化技法，抽象化技法，情報隠蔽技法，局所化技法が提案されている*.

次に，各原理を簡単に説明する.

(i) モジュール化

モジュール化という考えは古くから存在している．できるだけ独立した部分に分割するようにしたいとの考えである．モジュラー・プログラミング技法などと呼ばれて久しいが，サブルーチンに分割することも思想的には同一視できるし，そこまでさか上れば 1950 年代に存在していたといえよう．ソフトウェア関連の用語が一般にそうであるように，モジュールの定義は定かでなく，中には階層的に分割することまでも意味して使われている．しかし階層性はむしろ抽象化に関連づけ，モジュール化は，決められたインタフェースを通じてのみ情報のやりとりとする，相互に独立性の高い要素に分割することと考えるのが妥当であろう.

モジュール化技法としては，モジュラー・プログラミング，Parnas の情報隠蔽原理と関連したモジュール分割法[Parnas 71]などがある.

(ii) 抽象化

10 年前に Dijkstra により構造的プログラミング (structured programming，以下 S. P. と略称する) の提案で使われて以来，最近では最もよく知られ，かつそのための技法が提案されている原理といえよう．枝葉末端のことには目をつぶり，その時点で考えている問題の本質的な性質，特徴のみを考慮の対象にしようとするものである.

プログラムの作成では抽象化された内容は，最終的にはプログラミング言語で記述されねばならず，それに至る道程には逐次的な詳細化が存在する．すなわち抽象化の密度が階層的に粗になっていく．これらの階層のことを [Liskov 72] は抽象化のレベル (levels of abstraction) と表現した．この詳細化は，モジュール化の原理と併用され，上位レベルが下位レベルの抽象像になっている．もちろん複数個の詳細化が可能であり，従って，幾通りものプログラム化が存在しうる.

各階層ごとに上位レベルの概念を，正しく具体化していることが保証されればプログラムが全体として検証されたことになる.

抽象化に関する技法の提案は極めて多い．プログラムは手続きとデータとから成ることを思えば，手続きの抽象化とデータの抽象化とが必然的に存在する．特に後者については抽象データや抽象データ型があり，仕様言語，プログラミング言語，さらに検証においても非常に重要な位置を占めている.

更に，プログラミング方法論にはモジュール化技法と抽象化技法との組合せとして考えられるものが多い．S. P.[Dijkstra 72]，下降型 (topdown) プログラミング法[Mills 71]，段階的詳細化法 (stepwise refinement) [Wirth 71a]などはそうである．また，アクション・クラスタ法[Naur 69] は動作と論理の流れとの分離は存在するものの，詳細化の過程を経ることから抽象化原理に従っているといえる.

(iii) 情報隠蔽

[Parnas 71] で初めて使われた言葉であり，不必要な情報は他へ与えてはならないことを主張している.

ここで，前段の抽象化原理における本質的という言葉も非常に似た意味で理解されていることは注意に値する．情報隠蔽の思想は抽象化原理における本質の抽出という考え方をさらに積極的に推し進め，必要最小限の情報だけを提供し，その他の情報は他から見ようとしても見えなくしようとするものである.

Parnas 自身はこの思想を仕様記述言語の提案に採用したり，モジュール分割の基準に用いようとした．また，プログラミング言語において，従来のアクセス権をより明確な形で記述しようとするのもこの思想の反映であり，最近の構造的言語の大きな特徴の一つといえよう.

(iv) 局所化

この原理は，いろいろな意味で近い関係にあるものはできるだけひとまとまりにしようとするものである．構造的な制御要素の使用や go to 文の不使用は明らかに局所化技法の例とみることができる.

これらの原理は**表-2.1** にも示すように現実面として言語要素にも採用されている．なお表では上記 4 つ以外の原理的な表現をも括弧付きで付け加えている.

2.2 具体的な諸提案

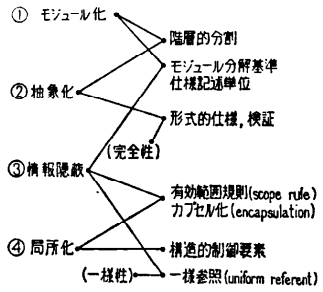
設計からコーディングの段階に対して提案されているいくつかの方法論について概観する**．なお，例題を用いる場合には，「電報の編集問題」[Henderson 72] を多少変更した次の問題を使用する.

---

* 本稿でのこの理解の仕方は我々の 1 つの見方の提案でしかない．例えば [Ross 75] では，原理について検討しているが，そこでは上記 4 つの他に一様性 (uniformity)，確証性 (confirmness)，完全性 (completeness) も並べている.

** プログラミング方法論は，それへの関心の高さから，これまで様々な解説がなされ，また翻訳も少なくない．ここでは十分に紹介がなされていると考えられる方法論については，その基本的な考え方のみに触れる.

表-2.1



「処理される電報は文字（英数字，空白記号）列であり，定められた大きさのブロック単位で読み込まれる．電報内の各語（空白を含まない文字列）は1つのブロックに含まれ，ブロック内の各語は1つ以上の空白記号によって区切られる．各電報は語 "ZZZZ" によって区切られる．処理は空の電報の出現によって終了する．各電報について，課金される語数と 13 文字以上の長すぎる語の有無が調べられ，その情報が出力される．"ZZZZ" のみが課金されない．」

この問題は Henderson らによって S.P. を用いてもバグがあったとされ，以後しばしば例題[Ledgard 74], [Noonan 75], [Jackson 75]として使われてきたものである．

(i) モジュール設計

複雑なシステムの設計における有力な方法は部分に分割してから処理する，いわゆる分割統治 (divide and conquer)法によってその見かけ上の複雑さを減少させることであり，古くからモジュール化技法として知られてきた．

Parnas[Parnas 71]は，システムの構造は（a）モジュールの集合，（b）モジュールの特性，（c）モジュール間の連絡 (connection) により示されるとし，モジュール間の連絡はモジュールが互いに行う協定 (assumption) とした．

Parnas の情報隠蔽はこの協定を必要最小限にすべきとするもので，彼が与えた仕様記述法[Parnas 72]は，それを用いて情報隠蔽を実践しようとしたものであった．

彼の方法に現われるモジュールは，システムの状態を与える V 関数，状態を変更させる O 関数，の2種類から成る．状態の変更は，関数呼出しの前後での状態（を表わす V 関数の値）の関係式で表わされる．これら関係式は，実現法には触れていずモジュールの使

```
V 関数 WORD
  取りうる値: 語 (初期値 不定)
  効果: なし
O 関数 NEXTWORD
  取りうる値: なし
  効果: WORD' が不定ならば
          WORD=入力ファイル中の最初の語
        さもなくば
          WORD=入力ファイル中の WORD' の次の語
        WORDLENGTH=WORD 中の文字数
        WORD が "ZZZZ" ならば EOTEL=真
        WORD', WORD がともに "ZZZZ" ならば
          EOF=真
        EOF が真のときの呼出しは誤り
V 関数 WORDLENGTH
  取りうる値: 整数 (初期値 不定)
  効果: WORD が不定のときの呼出しは誤り
V 関数 EOF
  取りうる値: 論理値 (初期値 偽)
  効果: なし
V 関数 EOTEL
  取りうる値: 論理値 (初期値 偽)
  効果: なし
```

図-2.1 電報編集問題の語に関連する部分の Parnas 風の仕様記述

用者と作成者間における十分小さな協定であると言える．

電報編集問題の中で語に関連する部分を Parnas の仕様記述法に基づいて記述すると，図-2.1 のようになるであろう．ここで ' は関数呼出し前の状態を表わす．この部分（語モジュールと呼んでよいかもしれない）は，状態を表わす4つの関数と，それらの状態の変化を記述する1つの関数から成る．他の部分が語を扱う場合は，ここに記された情報のみを利用でき，他の情報（表現法やアルゴリズム）を利用することはできない．

[Liskov 72] は良いモジュール分割は抽象化レベルと S.P. に基づくとし，次のように定義している．「システムは階層構造に分割される．各部分は抽象化の1つのレベルを表わし，共通の資源を共有する1つ以上の機能から成る．また，部分間の制御の連絡は構造的であるための制限を受け，データの連絡も陽に渡されたものに限られる．」

そのためのいくつかの指針を示しており，データ抽象の考えの芽生えが見られる．

この種の指針を現実の設計に応用しようとしたものに複合設計 (structured design または composite design)[Myers 75]がある．その記述は，視覚に訴えるモジュール構造図と非形式的なインタフェースの記述とから成る．それを得るための主要データの流れを解析する手順が与えられている．また，モジュール内の要素の関連度を与える基準（例えば，偶然同じモジュー

ル内にあるか，機能面としては同じの2つの要素であるか等関連度が強い程良いとされる）と，モジュール間の結合度を与える基準（例えば，一方が他方の一部を利用するか，共通領域中のデータを共有するか等結合度が弱い程良いとされる）があり，モジュール構造の良し悪しを決める評価基準となっている．ただし，あくまでも実践での実用性を重視した手法であり，理論的裏付けに乏しい．

(ii) 階層的プログラミング

階層的に展開する諸方法は思考の時間的過程に抽象化のレベルの考えを適用した方法と言える．S. P.[Dijkstra 72]，段階的詳細化法[Wirth 71a]，下降型プログラミング法[Mills 71]等多少の相違はあるが，この部類に入る．

Dijkstra はこれを，プログラミングの任意の時点で存在するものは，抽象機械の説明書とその抽象機械上で動くプログラムであるという形で明確に表現している[Dijkstra 72]．

この階層的プログラミングは，最近重視され出したプログラムの検証にうまく適合し，プログラミングと検証（又は証明）の並行という新しい型のプログラミングが提案され始めた(**5.4** 参照)．Wulf はこのプログラミングの流れを**図-2.2** のような規範として表現した[Wulf 77]．これはプログラミング法というものではないが，抽象データ型と抽象制御構造とを用いたプログラミングの流れの大枠として典型的と考えられる*．

(iii) 抽象データの利用

抽象データとは，その実現法（データ構造やデータ構造へのアクセス・アルゴリズム）と独立なデータであって，その振舞いによって完全に規定されるものである．この振舞いはその上で定義されるいくつかの演算(operation)によって定められる．抽象データを実現している部分とそれを利用している部分とは，振舞いを定めるいくつかの演算のみを仲介としているため，当該データに関するそれぞれの局所性は十分高いと言える．

これをデータ型に適用した抽象データ型もまた重要な考えである(**4.3** 参照)．Wulf の規範に見られるように，抽象データ型はモジュール化のための基準として設計時に使用できる技法である．また，コンパイル時のタイプチェックを重視する大幅にタイプ化された

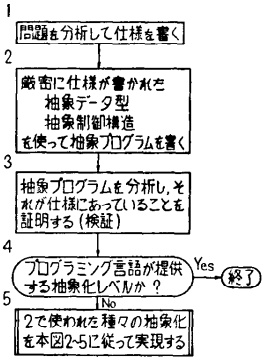


図 2.2 Wulf によるプログラミングの規範

言語を用いて抽象データ型を利用するプログラミングは，データの使用に関する安全性を高めることになる．

ここで抽象データ型を使って電報の編集問題を考えてみよう．

処理される電報の概念上の構成単位は「語」である．しかし記述すべきプログラム言語には一般に語に相当するデータ型は存在しない．そこで語という抽象データ型を考え，このデータ型の振舞いをその実現法に関係しないように，演算のみで規定する．

例えば次のようになる．

データ型名：Word
operation ReadNextWord：Word — 電報のファイルから次の語を1つ読み取る．
Length (W：Word)：integer—与えられた語の長さを求める．
IsZZZZ (W：Word)：Boolean — 与えられた語が“ZZZZ”か否かを判定する．

この抽象データ型 Word はその振舞いが同じであればどのように実現してもよいし，他に影響を与えることなくあとで実現部分だけを変更することもできる．

(iv) Jackson の方法

S.P. で代表されるような機能分割を行ういわゆる機能的接近法 (functional approach) に対し，Jackson [Jackson 75] は入出力のデータ構造に着目したプログラミング法を示した．電報のようなテキストや事務計算で使われるデータは一般に構造を持っており，この方法の適用範囲は狭くない．

Jackson の方法の基本的な考え方は，「データの構造に密接に結び付いたプログラムの構造」にある．その手順を次に示す．

(1) 入出力データの構造を定める．

(2) (1)の結果からプログラムの構造を定める．

* Wulf らによって設計された言語 Alphard はもちろんこの流れを支援することを主目的としている．

（3） 基本的な演算を抽出し，その実行回数などを考慮しつつ各演算を（2）の結果に重ねる．

（4） （3）で得られた演算を含むプログラム構造を，実行可能な形式でプログラムとして表現する．

データの構造およびプログラムの構造は共に連接 (sequence), 繰返し (iteration), 選択 (selection) によって表わされ，同名の構造はそれぞれ対応するものと考える．

電報の編集問題のプログラムの構造を求めるために，上記手順の（1），（2）を実践してみよう．

まず入力データと出力データの構造が分析され，**図-2.3** のようになる．

この例では入力，出力のデータ構造間に不調和又は不一致 (structure clash) がある．それはブロックと電報との概念単位間に包含関係がないこと (boundary structure clash) によると考えられるが，その場合の解決策が用意されている．それは中間ファイルを設定して，処理を2段階に分けることである．中間ファイルの構成単位としては，入力，出力の両データ構造の中でメモリ中に入る最大の共通成分が適当とされる．この例では，語が最大の共通成分である．こうして処理前半部，および後半部から見た中間ファイルは**図-2.4** のようになる．

プログラムの全体的な構成は，**図-2.5** となる．

前半部，後半部とも入力，出力データ構造間に不調和は存在しないので，それらのプログラムの構造は前記データ構造を反映したものになる．中間ファイルを実際に実現すれば，あるいはコルーチンや並行プロセスの可能な環境では，こうして書かれたプログラムはそのままで実行可能となる．ただし，効率上の問題から Jackson は中間ファイルを使わない通常の順次プログラムへの機械的な変換 (program inversion) を提案している．変換して得られたプログラムはもとの構造をかなり保ってはいるが，理解しづらいものとなる．

別の見方をすればこの変換を行ってくれるツール，あるいは並行プロセス（の特殊な場合）を効率良く支援してくれる言語が，Jackson 法を適用する際に望まれる．

また，Jackson 法は，2つの動作（ここでは語の読込みとその利用）の時間的関係をきっちり規定する必要がなければ，ある種の問題のプログラム構造は単純な形で書けることを示唆している．

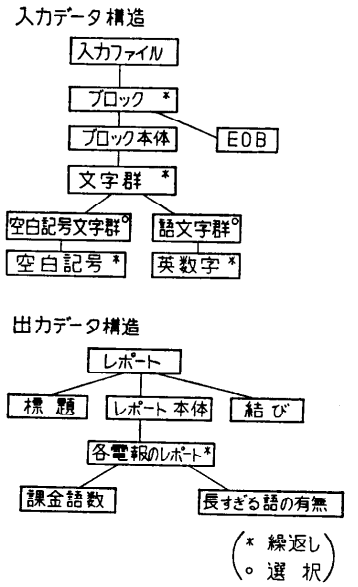


図-2.3 入力データと出力データの構造

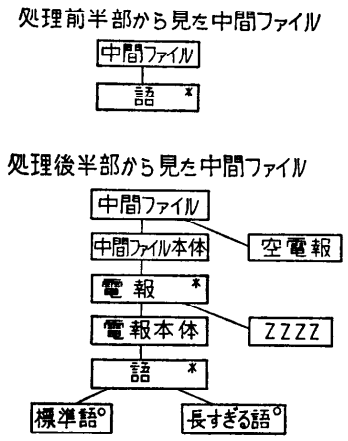


図-2.4 中間ファイルの構造

図-2.5 プログラムの構成

3. プログラム記述法

3.1 方法論からの影響

2. では，人間の能力の限界を認識して問題の見かけ上の複雑さを減少させるための方法論上の様々な概念，提案を示した．そのようなプログラミング上の概

表-3.1 プログラミング方法論と関係深い主なプログラミング言語

言語	(主要) 用途	言語の主な特徴	参考文献
Simula 67	シミュレーション	class の概念を初めて導入した. subclass を利用した階層構造プログラムが書ける. コルーチン機能を持つ.	[Dahl 68] [Dahl 72]
Pascal	システム記述, 教育	構造的で簡潔な制御構造, 豊富なデータ構造化機構を持つ. 言語の簡潔さと効率の良さに特徴がある.	[Wirth 71b] [Jensen 76]
Clu	抽象化支援	抽象データ型を初めて取り入れた. iterator による制御の抽象化. 例外処理機構を持つ.	[Liskov 74] [Schaffert 75] [Liskov 77]
Alphard	ソフトウェア工学研究の思考上の道具, システム記述	プログラムは仕様部とインプリメント部からなる. 言語に含まれる少ない基本的概念から標準的前設定 (standard prelude) や利用者の定義によって拡張される.	[Wulf 76] [Wulf 78]
Concurrent Pascal	OS 記述	データ抽象のための class, データの共有と同期のための monitor, プロセスのための process なるデータ型がある. 簡潔な言語構造.	[Brinch Hansen 75] [Brinch Hansen 77]
Mesa		カプセル化機構としての module (インタフェース用と実現用の2種類あり). 一様参照 (uniform reference) 問題について考案.	[Geschke 75] [Geschke 77]
Modula	ミニコンの OS 記述	名前の可視性を制限する module 構造. 特に機械依存部分, 同期部分を一箇所に閉じ込める.	[Wirth 77]
Euclid	システム記述	検証を可能とするように Pascal を改造. 名前の可視性の制御, 値共有 (aliasing) 禁止, カプセル化支援の module 等を特徴とする.	[Lampson 77] [Popek 77]
Gypsy	汎用および通信処理システム記述	静的, 動的両者のチェックによる証明をねらう. Algol 風の入れ子構造を放棄.	[Ambler 76] [Ambler 77]
Dijkstra's language	プログラミング法例示	最弱事前条件(weakest precondition) により定義された非決定的な制御構造, 初期設定, 非可視的なブロック構造, 関数としての配列, 手続きなし.	[Dijkstra 76]
ι language	教育用	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のが最大の目的. 多ソート一階論理 (many sorted first order logic) に基づく仕様言語, Pascal を拡張したプログラミング言語の2つがプログラム記述の基本となる.	[Nakajima 77]

念がプログラム・テキストに反映されていないとすれば, 保守, 変更といったもう一つの重要な段階での複雑さを克服していないことになる. つまりプログラムを記述する道具であるプログラミング言語は, 本来方法論の諸概念を支援すべきものである. 逆に, プログラミング言語は, そこに含まれる機能や制限のために, 問題を分析しプログラムを作り出す諸段階での我々の思考方法に多くの影響を与えている (再帰呼出し (recursion) の有無はその代表的な例であろう).

このように言語設計者のプログラミングに対する考え方は, 利用者の思考法, プログラムのわかり易さ等と大きな関連を持ってくる.

こうした認識が広まるにつれ, 最近の言語は方法論を強く意識して設計されるようになった. ここではその種の言語のみに焦点を絞る. 表-3.1 にその代表的な言語について特徴等を示す. これらの言語の多くに見られるいくつかの傾向をまとめてみよう.

(a) モジュール構造のプログラムが書ける. モジュール内の詳細やモジュール外の詳細は, 特に指定しない限りモジュールの外や内に知らされない. 知らせる手段が用意されていない場合もある. 機能的に独立であるように分割された単位という従来のモジュールの概念に対してその考えを補強する意味で, 内や外の詳細を区切る壁としてのモジュールの役割が強調され出したわけである. これは情報隠蔽の1つの具体化と言える.

(b) データの抽象化を積極的に支援している. プログラムを計算過程の動作として手続き的に捉えようとすることが多かったのに対し, 処理の対象であるデータを中心に持ってきて, データの振舞いとして捉えようとする面も出てきた.

(c) プログラムの証明/検証を支援しようとして言語の構造にも影響が現われだした. 具体化法としての1つはプログラム中に仕様や表明 (assertion) を書かせる方法であり, 1つはプログラムの証明/検証が容易になるように言語機能を制限する等の形で言語仕様を注意深く定める方法, の2つに大きく分かれる.

(d) 不必要に詳細な部分は記述しなくてもよい. あるいは別のレベルで記述させる. この例としては非決定的なアルゴリズム記述や, 繰返し制御の定義がある. これらは検証を容易にする手段でもあり, 後者は抽象化のレベルの具体化にもなっている.

3.2 概念上の基礎としての Simula 67 と Pascal [Wegner 76]

(i) Simula 67 の class

Simula 67 の class は Algol 60 のブロックと全く同じ構造をしている. つまりデータの宣言, 手続きの宣言, 実行文, の集まりである. Algol 60 のブロッ

ク中のデータは，制御がブロック内にあるときのみ意味を持ち，制御が外に出ると意味を失う．一方 class は object (実現値) を作るための枠組であって，object の作成，消去は class の外側で行われる．従って，class 内に制御がない場合にも，object は存在し続けることになり，手続きをも伴ったデータ構造と考えられるものである．

この考え方は単にシミュレーションの対象を記述するためだけでなく，データ型のようにその時点での状態を表わすデータ構造とそれを扱う手続きとから成る対象を記述するための有力な手法であった．

class の概念はデータと手続きとを対にした新しいタイプのモジュールを実現するすぐれたものであり，その後の object 指向型 (object oriented) 言語*等の概念上の基礎となっているが，object を保持する reference 変数という一種のポインタの持つ次のような強力な機能は受け継がれなかった．すなわち，ある object を指す reference 変数を持っている側は，その object に局所的なデータや手続きをすべて参照することができ，更にデータの値を変更することまでもできる．これを防ぐ手段は class 側に用意されていない．

このような情報の筒抜けは，その後の方法論，特に情報隠蔽の考えに合致しなかったわけである．

(ii) Pascal の簡潔さとデータ型チェック

Pascal の特徴は，制御構造とデータ構造の言語機能のバランスをとりながら，簡潔さを追求した点にある．また，プログラミングの複雑さが十分認識された今日 Pascal は言語のあり方としてより適した方向付けをしたと考えられる．

Pascal の簡潔さは主としてコンパイラの効率，目的プログラムの効率という実用性に動機付けられているが，簡潔さの故に比較的簡単な公理で言語の多くの部分の形式的な定義が可能となった．このことはプログラムの形式的な証明／検証を容易にするもので，実用的な言語によるプログラムの証明／検証への見通しを明るくするものであった．

Pascal のもう1つの特徴は，データの誤った使用をコンパイル時にチェックしようとする設計思想である．データ構造化機構を一通りそろえ，自由にデータ型を定義できることがその裏付けとしてあり，コンパイル時チェックを可能とするような制約が付け加えられている．

コンパイル時チェックの充実は，誤りの早期発見や実行時チェックの減少により効率の上昇をもたらしてくれる．それをできるだけ多く可能とするような言語機能や構文が，主としてデータ抽象機能と絡んで追求され多くの言語に取り入れられている．

一方 Pascal への批判もいくつかある[Habermann 73] [Welsh 77]．データ型に固有な演算の集合を定義できないという最近の視点からの批判の他に，データ型に関するものとして，

(a) タイプチェックを重視しながら，データ型の同等性の定義がなく処理系まかせになっている．処理系の定める同等性もすっきりした形では書けない．

(b) 様々な大きさの配列を処理する手続きを書けない．

(c) subrange 型の変数を左辺に持つ代入文をどう考えるか問題がある．

(d) set 型の定数のデータ型が定まらない．

等がその主なものであり，Pascal に基づいて言語設計を行う場合克服しなければならない点となっている．

3.3 いくつかの言語での実際例

(i) データ抽象の具体化

データ抽象をインプリメントするには，group[Shigo 75]のように単一のデータを抽象化する機構と，抽象化されたデータを枠組として定義する抽象データ型機構の2つが考えられる．

抽象データ型の利用は Liskov によって提案され，彼女らによって設計された Clu に組み込まれた．(抽象) データ型を定義する機能を持つ言語での典型的な使用法を**表-3.2** にまとめる．ここで T が定義される (抽象) データ型で U なるデータ型の2つの成分を内部表現として持ち，T に固有な演算として P を含んでいるとしている．

ここでは抽象データ型を忠実に実現している Clu と，完全ではないが抽象データ型の使用を支援している Euclid と Modula での場合を検討する．

Clu の抽象データ型は cluster によって定義される．この cluster によるスタックの定義を**図-3.1** に示す．

cluster の外側では，抽象データ型名の他には **is** リスト中の演算名のみが データ型名$演算名 の形で参照できる．また，cluster の内側で参照できる外側の名前も (抽象) データ型名や前記のような演算名に限られている．この意味で cluster は抽象データ型を定義する (というよりは定義してしまう)．

---

* データをメモリの集合体としてではなくメモリと不可分のものとして捉え，その操作はいくつかの限られたアクセスのみによるようにした言語を object 指向型言語と呼ぶことがある．

表-3.2 (抽象) データ型の様々な実現法

	(抽象) データ型定義	宣言	object 生成	初期設定	演算呼出し
Simula 67	class T; begin U u1, u2; procedure p (v); ～ S end	ref (T) a, b	a:－new T		a.p (v)
Clu	T=cluster is p; rep=record [u1, u2:U]; create=oper ( ) returns (cvt); ～ p=oper (a: cvt, v: V); ～ end T;	a, b: T			T$p(a, v)
Euclid	type T=module export (p); var u1: U:=～; var u2: U:=～; procedure p (v: V)=～ end T;	var a, b: T			a.p(v)
	type T=module export (T1, p); type T1=record var u1: U:=～ var u2: U:=～ end T1 procedure p (var a: T1, v: V)=～ end T	var a1: T var a, b: a1. T1			a1.p(a, v)
Modula	module M; define T, p; type T=record u1, u2: U end; procedure p (var a: T; v : V); ～ end M;	var a, b: T			p(a, v)
Alphard	form T= specification ～ p (a:T, v:V) ～ representation unique u1: U init ～ u2: U init ～ ～ implementation body p=～ endform	local a, b: T			p(a, v) a.p(v)
Concurrent Pascal	type T=class; var u1, u2: U; procedure p (v:V); ～ begin init S end;	var a, b: T	init a		a.p(v)

```
Stack=cluster [t: type]
  is create, Push, Pop;
  rep=array [t];
  create=oper ( ) returns (cvt);
        return (array [t] $ create(1));
  end create
  Push=oper (S: cvt, x: t);
    rep $ extendh (S, x);
    return;
  end Push;
  Pop=oper (S: cvt) returns (t) signals (empty-stack);
      if rep $ size (S)>0
         then return rep $ retracth (S)
         else signal empty-stack;
  end Pop;
end Stack;
```

図-3.1 Clu によるスタック記述

rep は抽象データ型の内部表現を表わす．また **cvt** は，cluster の外側ではこの cluster によって定義される抽象データ型，内側では内部表現 **rep** として扱われるべきデータ型であり，必要な時点で型変換がなされることを示す．抽象データ型と内部表現との型変換を行うための基本演算 **down** と **up** も用意されている．

これらの型変換は視点の変更であって，実際には実行時の計算を必要としない．しかし cluster の内と外で異なる使われ方をされており，異なるデータ型とみなされていることになるので，これらは明確に区別されなければならない．上記型変換はこの考えに基づいて導入されている*．

Euclid の module はカプセル化 (encapsulation)**

* Alphard ではこのような型変換を用意していない．
** 方法論での局所化の概念に対応する言語上の概念である．

のための機構であり，抽象化されたものの表現やインプリメント法をその外から隠すことができる．moduleは1つのデータ型を定めるのでmoduleを使って抽象データ型を作ることができる．

Euclid で抽象データ型 Stack を実現する方法の1つを下に示す[Chang 78]．

```
type Stack (StackSize: unsignedInt)=module
  exports (Pop, Push)
  var IntStack: array 1..StackSize of signedInt
  var StackPtr: 0..StackSize:=0
  procedure Push (X: signedInt)=
    imports (var IntStack, var StackPtr,
             StackSize)
    begin
    procedure Overflow= … end Overflow
    if StackPtr=StackSize then
      Overflow
    else
      StackPtr:=StackPtr+1
      IntStack (StackPtr):=X
    end if
    end Push
  procedure Pop (var X: signedInt)=
    imports (var IntStack, var StackPtr)
    begin
    procedure Underflow= … end Underflow
    if StackPtr=0 then
      Underflow
    eise
      X:=IntStack (StackPtr)
      StackPtr:=StackPtr−1
    end if
    end Pop
end Stack
```

Euclid では module やルーチン*間の名前の可視性をプログラマの制御下に置いている．module やルーチン間では何の指定もなければ，外にある名前にアクセスできない．上例のように module で自分の中にある名前を外に知らせる場合には **exports****，module やルーチン内で外の名前にアクセスする場合には **imports**** を使ってプログラムテキスト中に明確に書くことになる．

module 内の変数をその値の変更というアクセス法も認めて export したり，外側の名前を変数扱いで import したりできるので，柔軟性のある使い方はできる．ただし抽象化の原理は簡単に破れるわけで，言語がプログラマに課す拘束は Clu ほどでないと言える．

module が定義するデータ型の変数を，module 内の手続きに引数として渡すことはできない．すなわち，Clu の **cvt** に相当するものはない．Clu での演算呼出しは，データ型名 $ 演算名 であるのに対し，Euclid では 変数名.手続き名 であり***，後者では変数は暗黙の引数として引き渡されている．

この方式では，例えば object の同値性の判定のように，その抽象データ型の値を2つ以上引数として取る演算を書くことができない．Euclidではそのような演算を持つ抽象データ型を定義する機構も用意されている（表-3.2 参照）が，自然な表現とは言えず言語の複雑さを増している．

Modula の module は可視性を制御するための壁の役割のみを持つ．しかし module 内でデータ型が定義されそれが export されても，その構造は外から見えないという規則がある．これにより，exportされたデータ型の変数は，そのデータ型を定義している module 内でしか操作できなくなる．また module 内の局所的な変数も export はできるが値の参照しかできない．これは関数手続き定義の省略形とみなせるものである．こうして初期設定に関するわずらわしさは残るが，抽象データ型を他の言語要素との調和を保って実現している．

(ii) 制御の抽象化

抽象化されたデータの集まりを対象にして繰返し処理する場合，抽象化のレベルを整え，繰返し法の詳細を隠す意味で，データの処理を行う繰返し本体部分と対象となるデータの取出しを行う部分とに分離することは有用である．

Liskov はこの概念を制御抽象(control abstraction)と呼び[Liskov 77]，取出し部分を繰返し部分とコルーチン的に動く iterator として Clu 中に組み込んだ．次の例は **string** 型のデータ中の文字についての繰返しを行えるようにする iterator である．

```
string_chars=iter (s: string) yields (char);
  index: int:=1;
  limit: int=string $ size (s);
  while index<=limit do
    yield string $ fetch (s, index);
    index:=index+1;
    end
  end string_chars;
```

繰返し処理部分は次のように書く．

```
for c: char in string_chars (s) do
    (繰返し処理の本体)
```

繰返し処理の本体とデータの取出し部分がコルーチン的に動くこと，**yield** によって次の対象が返される

---

* Euclid では procedure と function を総称してルーチンと呼ぶ．

** Modula では **define** と **use** が対応する．

*** object の成分として手続き/演算も含めていないかどうかの概念的な違いによる．

こと，iteratorの実行終了によって **for** 文も終了することで両者の動作は理解できよう．

Alphard と Euclid では，特別な名前の演算と変数を generator と呼ばれる抽象化機構の中に作っておき，ある種の制御構文はそれらの演算と変数とを用いて定められる制御構造の簡略形として定義している．こうして繰返し処理とデータ取出しとの分離を実現している．

(iii) 抽象データ型と組込みデータ型との調和

言語または言語システムに組み込まれた基本的なデータ型と，プログラマによって定義された抽象データ型との間には取扱い方法の制約面で違いが生じ易いが，概念上の一貫性を保つことが一様性 (uniformity) の立場から要求される．

Clu では，基本データ型も利用者定義の抽象データ型と同様に演算で定義されており，通常使われるインフィックス記法や配列操作等の記法を特定の名前の演算の略記法として捉えることで，両者の統一化を計っている．

Alphard も考え方は同じである．組込みのデータ型を最小にし (rawstorage と boolean のみ)，他のデータ型はすべて単一の抽象化機構 **form** で定義されるものとした．少し詳しく見てみよう．

演算は関数呼出しとして書かれるのが原則であるが，いくつかのもの(≒, =, >, +, or 等) は，インフィックス記法の演算として使うことができる．β を二項演算子とすれば

〈term〉$_1$ β 〈term〉$_2$

は

&β(〈term〉$_1$, 〈term〉$_2$)

のことであり，&β なる演算が **form** 中で定義される．

また，点表記法

〈qualname〉.〈identifier〉

は，

〈identifier〉 (〈qualname〉)

であり，角括弧表現

〈term〉 [〈expression list〉]

は

&subscript (〈term〉, 〈expression list〉)

である．

integer や vector 等は標準的前設定 (standard prelude) 中の **form** の中で &β や &subscript を用いて定義されていると考えるわけである．プログラマが定義するデータ型に対して，&β はインフィックス記法の演算，&subscript はアクセス・アルゴリズムを定義することに使える．

(iv) 検証のための工夫

プログラム中に仕様を書かせ，プログラムの検証を積極的に支援している言語として Alphard や ι (イオタ) 言語がある．これらは，抽象データ型の検証法 (**5.3** 参照)に基づいている．Alphard によるスタック定義例は**図-5.3** にある．

一方 Euclid は，現在の検証法の範囲内で検証を行うに適したプログラムを作れるよう設計されたと言われる．検証を容易にするいくつかの性質がどのような形で Euclid に組み込まれているか見てみる．

そのような性質として，関数(function) の副作用の禁止，aliasing*の禁止を取り上げる．これらは単に文法書に「禁止する」と書かれるのではなく，コンパイル時に効率良くチェックされる形で言語に組み込まれているのが望ましい．

関数の副作用，aliasing，全域変数の危険性等は変数へのアクセスが許されすぎているとして Euclid では制限を付け加えている．最大の制限はルーチンと module に対しての閉じた有効範囲 (closed scope) の導入であり，閉じた有効範囲内で外の名前にアクセスするときは import リストを書くことになっている．

関数の副作用の禁止は，直接的には関数への引数や import される変数が **var** (通常の変数扱い) であってはならないという規則に現われている．間接的には有効範囲 S 内にルーチンを import し，そのルーチンがある名前を (**var** として) import しておれば，S もその名前を (**var** として) import していなければならないという規則によって保証されている (関数の中から呼び出したルーチンが副作用を行うのを防いでいる)．これらの規則はコンパイル時にチェック可能なものである．

aliasing は，引数の引渡しやポインタ変数を使用する場合しばしば起こることであるが，Euclid では副作用の危険性を持つものとして除こうとしている．引数の引渡しによる aliasing のいくつかの例を示す．

(a) **procedure** P (**var** J: signedInt)
　　imports (**var** I)
　　.....
　　なる P に対して呼出し P(I)
(b) 呼出し Q(I,I)
(c) 配列 A を用いて呼出し R(A, A(I))

* 2つ以上の異なる名前が同じ又は重なる変数を参照すること．

(d) 配列 A を用いて呼出し S(A(I), A(J))
(ただし I=J の場合)

(a)～(c) はコンパイル時にチェックされる. (d) では I≒J という表明が生成され実行時にチェックされる.

ポインタによって生ずる aliasing は難しい問題であるが, ポインタを隠れた配列 (implicit array) の添字と考えることで, (証明規則も含めて)配列での問題に帰着させている. すなわち, 各ポインタは隠れた配列として振舞うある collection 変数の要素を指すものと考える. Cを collection 変数, p を ↑C のポインタとすると, p↑ は C を配列と考えて C(p) を表わす.

この考えは Pascal の最初の版にあり改訂版から消えた class 変数に対応するものである.

(v) アクセス法の制御

データへのアクセス権は現在の言語ではほぼ全か無かの原則に基づいている. データの保護や情報の共有をより制御された形で行うために OS で既に使われていたアクセス制御機構を言語の中に埋め込もうとする動きがある[Jones 76][Wulf 76]. 抽象データ型を定義でき, いくつかの演算のみが object への可能なアクセスとなっている object 指向型言語は, アクセス制御機構を考察するための良い基底言語と言える.

アクセス制御を組み込む利点としては[Jones 76],

(a) アクセスの方法が正しく書ける (データ共有の制約が守られる).

(b) 宣言風に記述されることでプログラマの意図が示される.

(c) コンパイル時にアクセスの正しさが保証される.

(d) JCL 等の別個の機能を使う必要がない.

があげられる.

Alphard のデータ型記述の一般形は

T(仮引数)〈演算〉

であり, 宣言時にアクセス制限を〈 〉内の演算のみに規定できる. 例えば

a) **local** i : integer〈+, −, =, ←〉

b) **function** $f(a : T\langle h\rangle)$

で, a) は変数 i に ＊ や ／ 等が使えないことを示し, b) は仮引数 a への演算を h のみに限定している. また,

local i : integer
.....
$f(i\langle +, -\rangle)$

のように実引数に〈 〉表現を付けてより制限されたアクセス権のみを呼ばれる手続き側に渡すことができる.

これらのアクセスの正しさはコンパイル時にチェックされる.

4. 形式的仕様記述法

4.1 形式的仕様の重要性

(i) 非形式的仕様と形式的仕様

プログラム作成者の最大の関心事は, でき上がったプログラムが意図通り正しく動くことである (2.1 参照). このプログラム作成者の意図を述べたものが仕様である. 仕様には, 大きく分けて, 図, 表などを伴った(規格化された)自然語で書かれた非形式的仕様と, 構文及び意味が形式的に厳密に定義された人工言語(例えば一階述語論理)で書かれた形式的仕様の2つがある.

現在ソフトウェアの生産現場で使われている仕様(外部仕様, 内部仕様など)は, ほとんど非形式的なものであり, 仕様記述のレベルでの無矛盾性のチェック, でき上がったプログラムが仕様を満たすことの検証などを厳密に行うことはできない. このためプログラムの正しさの確認は, 代表的な入力に対する実行による出力のチェック (テスト) という本質的に不完全な方法に頼らざるを得ない.

これに対し作成者の意図を形式的仕様として記述することには, 次のような利点が期待できる.

(a) プログラムの検証を厳密に行う基礎が与えられる.

(b) 仕様のレベルで無矛盾性, 完全性などのチェックができるので設計段階での誤りをなくせる.

(c) 形式的仕様は非形式的なものに較べてあいまい性が少ないので, 意思疎通用の媒体として有用である.

このような利点を持つ形式的仕様が現実に使われなかったのは, 実際的な規模のプログラムについての記述が困難であったことによる. しかしながら最近のプログラミング方法論の研究の結果, プログラムの本質的な構造に対する理解が深まり, 抽象データ型といった重要な仕様記述の単位(specification unit) が陽に認識されるにつれて, それらに対する形式的仕様記述法がいくつか提案されてきている. ここでは, 形式的仕様に対象を絞り, 代表的な記述法について述べる.

(ii) 仕様記述 (法) に望まれる性質

仕様記述（法）がプログラミングにおいてその役割を十分に果たすためには，次のような性質を持たなければならないと考えられる[Liskov 75]．

（a） 形式性(formality)：数学的に厳密な定義が与えられ検証の基礎となり，形式的（機械的）な取扱いに耐えること．

（b） 記述性 (constructibility)：書き易いこと．

（c） 理解性 (comprehensibility)．

（d） 最小性 (minimality)：必要最小限の情報だけを含むこと．できる限りプログラムの機能 (what) を述べ，具体的な操作手順 (how) は述べないこと．

（e） 広適用性 (wide range of applicability)：特定の領域だけでなく広い範囲に適用できること．

（f） 拡張性 (extensibility)：プログラマの意図の変化に伴う，修正や拡張が行い易いこと．

上述の性質の中でも，形式性は高信頼性を保証したいプログラム作成においては不可欠の要件である．

(iii) 仕様記述の単位

仕様が上述したような良い性質を持つためには，その仕様がプログラム化しようとする問題の本質的な構造を明確に表現したものでなければならない．つまり仕様は明確な抽象化に基づいて適度の複雑さ（単純すぎても意味がない）を持った概念単位をひとまとまりとして記述したものでなくてはならない．さらに，その概念単位（仕様記述の単位）は当然プログラム全体の構成要素としても基本的で意味のある必要がある．

このような性質を持った概念単位としてよく知られたものに機能の抽象化から得られた手続き (procedure 又は function) がある．実際それはプログラミングにおいて最も基本的な役割を果たしてきている．

一方，データの抽象化から得られた"抽象データ型"の概念の重要性が最近認識されるようになった．抽象データ型は，ある特定の要素の集まりの上に働く，互いに関連し合った複数の演算の集まりとして考えるのが普通である．つまり要素の振舞いはそれら演算を適用することによってのみ知ることができる．この抽象データ型の考え方の最も重要な点は，互に関連し合った複数個の手続き（演算）をひとまとまりの単位として捉え，それが仕様記述の単位として，つまりはプログラムの構成単位として基本的であることを陽に認識したことにあるといえる．

以下では，手続きと抽象データ型の仕様記述について概観する*．

4.2 手続きの仕様記述法

単一の手続きを1つのモジュールと見ることは，プログラミングにおいて基本的であり，その仕様記述法も数多く提案されている[McCarthy 62][Naur 66][Floyd 67][Hoare 69][Manna 69][Manna 72]．これらはいずれも，プログラムの検証に際し，検証の基礎となる，そのプログラムが何をするものであるかということを形式的に記述しようとする努力から生まれてきたものである．

手続きの仕様記述法には，入出力仕様記述法と操作的 (operational) 仕様記述法の2つの代表的な方法がある[Liskov 76]．

（i） 操作的仕様記述法

この方法は，仕様記述したい手続きの機能をアルゴリズムを示すことにより定義するものである．ただし，この仕様として与えたアルゴリズムは，その手続きの機能を計算機上にインプリメントするアルゴリズムとは異なるものであることを十分認識する必要がある．つまり仕様としてのアルゴリズムは，できるだけ簡単でプログラマの意図の抽象化として得た手続きの機能の本質的な構造を忠実に反映することに主眼がおかれ効率などは問題にされない．これに対し，手続きを計算機上で実際に動かすアルゴリズムは効率を考慮することが大切である．

このような仕様を記述するための（アルゴリズム記述用）言語は出来るだけ単純であることが望まれる．このような仕様記述法として有望なものに，再帰方程式 (recursive equation) による関数の定義法[McCarthy 62, 63]がある．操作的仕様記述という考え方からすれば，他のプログラミング言語も仕様言語として使えないことはないが，形式性において再帰方程式がすぐれているといえる．たとえば，2つの整数の最大公約数を求める手続き gcd の再帰方程式による操作的仕様は次のようになる．

インタフェース：
gcd (integer, integer) → integer
方程式：
$\gcd(x, y)$=if $x \leq 0 \vee y \leq 0$ then error ("unexpected input") else $f(x, y, \min(x, y))$
$f(x, y, z)$=if $\mathrm{rem}(x, z)=0 \wedge \mathrm{rem}(y, z)=0$ then z else $f(x, y, z-1)$

（ii） 入出力仕様記述法

この方法は，入力と出力が満たすべき性質を表明 (assertion) の組として表現することにより，入出力の関係を記述するもので，手続きの仕様記述としては最も一般的なものである．Hoare の記法[Hoare 69]を使う

* この他の仕様記述の単位と考えられる並行プロセスについては 6. で簡単に触れる．

とこれは,

P {M} Q

のように表現できる．ここで P，Q はそれぞれ入力および出力に関する表明であり，通常一階述語論理で表現される．また M は手続きを示す．

このように書かれた仕様の解釈としては次の2つがある．

（a） 命題Pが成立していて，手続き M が実行されたとき，もし M の実行が停止するのなら Q が成立する．

（b） 命題Pが成立していて，手続き M が実行されたとき，M の実行は停止してかつQが成立する*．

（a）は，Hoare がこの記法を導入したときの本来の解釈であり，M の（P,Q に対する）部分正当性(partial correctness) といわれる．

（b）は，（a）の条件に M の実行が停止する条件を付加したもので，M の全正当性 (total correctness) といわれる．

たとえば，2つの整数の最大公約数を求める手続き gcd の入出力仕様は次のようになる．

インタフェース：
gcd (integer, integer) → integer
入出力仕様：
in(x, y){gcd (x, y)} out(x, y)
ただし，
in(x, y)≡$x>0 \wedge y>0$
out (x, y)≡gcd(x, y)|$x \wedge$ gcd(x, y)|y
& $\forall i$: integer [$i|x \wedge i|y \Rightarrow i \leq$gcd($x, y$)]
$x|y \equiv \exists i$: integer [$y = x * i$]

4.3 抽象データ型の仕様記述法

抽象データ型の振舞いを仕様として厳密に述べるためには，その抽象データ型の値の集合と，その集合上に定義されたいくつかの基本的演算の働きを明らかにする必要がある．これはある意味で自然数，群，行列といった数学的な体系を定義することに類似している．

抽象データ型の仕様は，構文部 (syntactic part) と意味部 (semantic part) に分けて考えるのが便利である．構文部では，仕様記述の対象となるデータ型を特徴付けるいくつかの演算の入出力領域が明らかにされる．意味部では，構文部で導入された各演算に対してその働きが明らかにされる．

抽象データ型の仕様記述法は，主にその意味部の記述方法の違いにより，抽象モデルを用いる方法と公理的方法に大別される[Liskov 75,76]．

（i） 抽象モデルを用いる方法

この方法では，仕様記述したい抽象データ型は，すでに形式的仕様が与えられ，性質もよく知られた他の抽象モデルを用いて表現されることにより意味付けされる．つまり，仕様記述されるべき抽象データ型を特徴付ける各演算を1つの手続きとみて，抽象モデル上の演算を用いてその仕様を記述することになる．これは手続きの仕様をアルゴリズムを示すことにより記述する操作的仕様記述に対応するものであり，やはり効率などは考慮されず形式性，読み易さ，簡明さが重視される．

抽象モデルを用いた仕様記述の例として，整数を要素とし，長さに制限のあるスタックの，列(sequence) という抽象モデルを用いた仕様記述[Wulf 76]を示す（図-5.3 **specifications** 部分参照）．この仕様では，構文部は陽には示されていないが，それは次のようになる．

構文部：

push (istack, integer)	changes istack
pop (istack)	changes istack
top (istack)	returns integer
empty (istack)	returns boolean

istack の仕様は，istack と列 $\langle \cdots x \cdots \rangle$ を対応させる **let** 節での宣言と，I_a, β_{pre}, β_{post}, β_{req}, β_{init} といった命題により記述されている．ここで，

a） I_a：istack の要素を示す不変命題．つまり，$\{x|I_a(x)\}$ が istack の値の集合になる．

b） β_{pre}, β_{post}：各演算の入出力を規定する命題．

c） β_{init}：初期条件．

d） β_{req}：**form**（表-3.2 参照）の引数に対する条件．

である．これらの命題は，列という抽象モデルを用いて記述されている．むろん列に対して定義されている nullseq（空列を示す），length（列の長さを示す），last（列の最後の要素を示す），leader（last を取り去った列を示す）といった演算に対してはあらかじめ厳密な定義が与えられているものとしている．

（ii） 公理的仕様記述法

この方法では，データ型を特徴付けるいくつかの演算の間に成立する関係を公理として述べることにより，そのデータ型の振舞いを記述する．公理を述べる言語の違いに応じて，述語論理的と代数的の2つの代表的な方法があるが，ここでは，最近仕様記述法としての研究が盛んである代数的仕様記述法[Zilles 74][Guttag 75][Goguen 78]について述べる．

* Dijkstra の最弱事前条件 (weakest precondition) の考え方[Dijkstra 75]は，この条件を満たす最も弱い命題を wp(M,Q) と書いて構成的検証法(5.4 参照) に利用しようとするものである．

表-4.1 各仕様記述法の特徴

			文献	特徴
手続きの仕様記述法	入出力仕様記述法（述語論理による）		[Naur 66] [Dijkstra 75] [Floyd 67] [Hoare 69]	・検証法との関連において最もよく使われる． ・形式性，最小性，広適用性は良い． ・記述性，理解性はおとる． ・仕様記述法としては，蓄積もあり最も一般的．
	操作的仕様記述法（再帰方程式による）		[McCarthy 63] [Manna 72]	・fixed point 理論に基づいた理論的基礎付けがある． ・理解性，記述性は良い． ・最小性はおとる． ・仕様言語として研究は新しく，今後の課題は多い．
抽象データ型の仕様記述法	公理的方法	述語論理的	[Hoare 72a, 73] [Standish 78] [Spitzen 75]	・形式性，最小性，広適用性は良い． ・記述性，理解性はおとる． ・主として組込みデータ型の記述に使われてきた．
		代数的	[Zilles 74] [Goguen 78] [Guttag 75, 78]	・形式性，最小性は良い． ・記述性，理解性は，述語論理的なものよりは良い． ・広適用性は述語論理的なものにおとる．
	抽象モデルを用いる方法		[Earley 71] [Wulf 76] [Hoare 72a]	・理解性，記述性，形式性は良い． ・広適用性，最小性はおとる． ・モデルの選び方が仕様の質に決定的に影響する．
	状態機械モデルを用いる方法		[Parnas 72] [Robinson 77b]	・理解性，記述性は良い． ・広適用性，形式性はおとる． ・比較的大きな例題に対して，仕様記述の実験が行われ，この方法に基づいた仕様言語も設計されている．

代数的方法では，仕様記述の対象となる抽象データ型の値は，そのデータ型を出力領域とする演算（この演算を構成的演算 (constructor operation) と呼ぶ）の適用の系列として認識される．具体的には，構成的演算を表わす演算記号から生成される項 (term) の集合を，公理により定められる同値関係*で分類して得られる商集合が，抽象データ型の値の集合になる．構成的演算以外の演算は，これらデータ型の値から何らかの情報を引き出す．公理は，これら非構成的演算がデータ型の任意の値に対して定義されるように記述されなければならない．

例として，蓄えられる要素のデータの型や数に制限のないスタックの代数的仕様記述を図-5.4 に示す．

NEWSTACK, PUSH, POP, REPLACE が構成的演算であり，POP, REPLACE に関する公理から抽象データ型 Stack の値は必ず PUSH(…PUSH(NEWSTACK, i_1), …, i_n という形で表わせることが分かる．他の公理はこのような任意の要素に対して ISNEW, TOP の値を定めている．

上述した2つの方法以外で注目されるべき仕様記述の方法としては，Parnas の状態機械モデルを用いるものがある[Parnas 72](図-2.1 参照)．これは，抽象データ型の要素を状態機械の状態に対応させるもので，演算の間に成立する関係を公理で記述するという意味では公理的方法に近い．この方法に基づいた仕様言語 SPECIAL も提案されている[Robinson 77b]．

4.4 各仕様記述法の特徴

4.2, 4.3 で述べた仕様記述法を含め，現在提案されている代表的な仕様記述法の特徴をまとめて表-4.1 に示す．

5. 検証法

5.1 検証法の背景と階層的検証法

プログラムの検証 (verification of program) とは，プログラムが仕様通りに動くことを確かめることである．つまり，プログラムの検証はあくまでプログラマの意図を厳密に述べた仕様があって初めて意味を持つものであり，正確にはプログラムの仕様に対する検証というべきものである．プログラムの（正当性の）証明 (proof of correctness) という言い方が同様な意味で使われることも多いようであるが，証明 (proof) という言葉にはプログラムの正しさを（仕様など考えずに）無条件に確立するという響きがあり，過去においても少なからず無用の誤解を招いて来た事情を考慮し，ここでは検証 (verification) という術語に統一している．

現実的な規模を持つプログラムの検証を行おうとするときの最大の難点は，その仕様記述および検証が著しく複雑なものになるということである．この難点を

* この同値関係を公理からどのように定義するかについては [Guttag 76] と [Zilles 74] [Goguen 78] の立場は異なる．

克服するためには，大きなプログラムを多数の小さなモジュールに分割し，仕様記述および検証をそれらモジュールごとに独立に実行できるようにして全体としての作業量を軽減することが不可欠と考えられる．このような要求を満たすようなモジュール化法の1つとして，抽象化のレベルに従った階層的モジュール化法がある（2.2 参照）．

この方法でモジュール化されたプログラムは，図-5.1のような構造を持つ[Nakajima 77][Robinson 77a]．ここで，各箱は1つのモジュールを示し，仕様部（specification part）と実現部（realization part）から成る．各モジュールの仕様部はある抽象化のレベル*に属し，その抽象化のレベルで抽象化されたデータ型または手続きを定義している．実現部は，それが属するレベル中の仕様部で定義されたデータ型または手続きを使って，あるプログラミング言語（最終的に実行可能な言語）で書かれている．

このように構造化されたプログラムの検証は，各モジュールに対し実現部が確かに仕様部を満たしていることの検証に帰着される**．このようにプログラムを抽象化のレベルに従ってモジュール化して検証する方法を階層的検証法と呼ぶ．

階層的検証法では，プログラム全体の検証は，各モジュールの検証に帰着する．このモジュールは，4.1で述べた仕様記述の単位に対応するものであり，現在のところ機能を抽象化して得られる単一の手続きで規定されるモジュール，データを抽象化して得られる複数の手続きで規定されるモジュールの2つが基本的なものとして認識されている．

検証は，これらモジュールごとに考えられなければならない．手続き（単一の手続からなるモジュール）に対する検証法は，帰納表明（inductive assertion）法，不動点（fixed point）法など，いわゆるプログラムの検証法としてよく知られているものであり，既にすぐれた入門書（[Manna 74] の3章および5章，また最近の話題を知るためには [Manna 78] が良い）が出ている点も考慮し，ここでは複数の手続きから成る抽象データ型の検証法を中心に述べていく．

5.2 手続きの検証法

手続きの検証（プログラムの検証）に関しては，過去に多くの研究が行われ数多くの方法が提案されている．

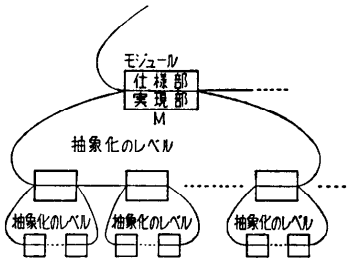


図-5.1 階層的検証法の概念図

るが，大きく次の2つに分類される．

（a） [Floyd 67] の帰納表明法に始まる，繰返し（iteration）を基礎に置く Algol 風なプログラムの検証法[Manna 69][Hoare 69][Igarashi 75]．検証はプログラム変数の間に成立する表明に対して行われる．

（b） [McCarthy 62] の再帰帰納（recursive induction）法に始まる，再帰呼出し（recursion）に基礎を置く Lisp 風な再帰プログラムの検証法[DeBakker 71][Morris 71][Burstall 69][Manna 73]．検証は再帰プログラム相互の同値性，再帰プログラムが定義する関数の種々の性質などに対して行われる．この方法は，再帰プログラムが計算する関数を，その再帰プログラムの最小不動点（least fixed point）として定義する考え方[Manna 72,73]との関連から，不動点法と呼ばれることも多い．

4.2-(ii) の入出力仕様に対する検証は，プログラムの要所に不変表明（invariant assertion）を付け加えることにより（a）の方法で行われる．

4.2-(i) の操作的仕様に対する検証は，結局2つのプログラムの同値性を証明することに帰着する．2つのプログラムがともに再帰プログラムのときは，これは(b)の方法に従って検証される．

5.3 抽象データ型の検証法

（i） 一般的手法

抽象データ型の検証とは，抽象度の高い抽象データ型の抽象演算を，抽象度の1段低い具体データ型の具体演算を用いて書かれたプログラムが正しくシミュレートしていることを示すことである．この事情を概念的に示すと図-5.2 のようになる．ただし抽象度の高低は相対的なものであり，次の段階では，具体データ型が抽象データ型として扱われることになる．

抽象データ型の実現は，

（a） 抽象データ型の各値がどんな具体データで表現されているかを示す，具体データの集合から抽象デ

---

* 図から明らかなようにプログラム全体は抽象化のレベルを要素とする木構造を成す．

** モジュールごとの検証がプログラム全体の検証を意味することは，木の高さに関する帰納法により容易に示される．

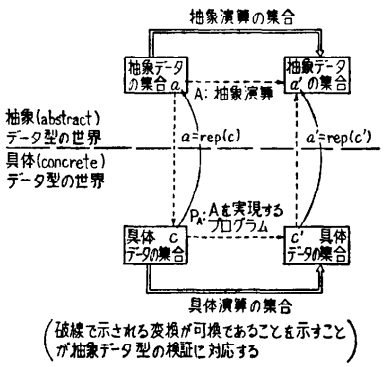


図-5.2 抽象データ型の検証の概念図

ータ*の集合の上への表現関数（rep. function；一般には多対1）.

（b） 各抽象演算 A に対し，それを実現する具体演算を使って書かれたプログラム P_A.

の2つにより示されている．さらに，抽象データ型及び具体データ型の振舞いはそれぞれ仕様により規定される．このような状況で，抽象データ型が確かに仕様通り振舞うように具体データ型により実現されているということの意味は次のようになる．つまり，具体データ型が仕様通り振舞うことを仮定すれば，各 P_A は表現関数 rep で示される対応のもとで抽象演算 A の仕様を満たすということである．もっと具体的には，抽象データ a に A を適用して $a'=A(a)$ が得られるのなら，$a=\mathrm{rep}(c)$ なる c に P_A を適用すれば $a'=\mathrm{rep}(c')$ なる c' が得られるということである（図-5.2 参照）．抽象データ型の検証とは，これを示すことになる．

仕様がどのように書かれているかは，検証法を考えていくうえで本質的である．抽象データ型の仕様記述法には，そのデータ型を特徴付けるいくつかの演算に対して，

（イ） それら各演算に対してその入出力仕様を示す（たとえば，**4.3-(i)** の抽象モデルを用いる方法）．

（ロ） それら各演算の間に成り立つ関係を公理として示す（たとえば，**4.3-(ii)** の代数的方法）．

---

* 3,4 では“抽象データ型の値”と呼んでいた．

** たとえば，演算 F と G が可換であるという公理 $F(G(x))=G(F(x))$ が仕様として述べられている場合（上記（ロ）），F，G を実現する Algol 風のプログラム P_G, P_F から直接これを示すことはむずかしい．この問題を解決するために，P_G, P_F の入出力仕様を述語論理を用い $\phi_F(x)\supset\psi_F(x, P_F(x))$, $\phi_G\supset\phi_G(x, P_G(x))$ のように記述し，これを仲介にして公理 $F'G(x))=G(F(x))$ を示す方法が考えられる．ここで，ϕ, ψ はそれぞれ入力表明，出力表明と呼ばれるものである．[Nakajima 77] ではこの入出力仕様のことを pre-spec と呼んでいる．

という2つの異なるやり方がある．ここでは，（イ）の仕様記述に基づくものの代表例として，Hoare の検証法[Hoare 72b]を発展させた Alphard における方法を，また，（ロ）の仕様記述に基づくものの代表例として，[Guttag 76] の代数的仕様を用いた検証法を紹介する．

その他の抽象データ型の検証法としては，Parnas 流の仕様記述に基づいた Robinson らの方法[Robinson 77a]が評価されている．また，ι-システムにおける検証法[Nakajima 77]では，pre-spec** という考え方を導入して，上記（イ），（ロ）を混用した仕様記述が可能となっている．

（ii） Alphard における検証法[Wulf 76]

図-5.3 に見るように，**form** の **specification** 部は

```
form istack(n: integer)=
  beginform
  specifications
βreq→requires n>0;
      let istack=⟨…xi…⟩ where xi is integer;
  Ia→invariant 0≤length(istack)≤n;
βinit→initially istack=nullseq;
      function
                                        βpre
        push (s:istack, x:integer) pre 0≤length(s)<n
                                   βpost      s' は実行前の値を示す
                                  post   s=s'~x,
        pop(s: istack) pre 0<length(s)≤n post s=leader(s'),
        top(s: istack) returns x: integer
          pre 0<length(s)≤n post x=last(s'),
        empty(s: istack) returns b: boolean
          post b=(s=nullseq);
  representation
    unique v: vector(integer, l, n), sp: integer init sp←0;
repの関数→rep(v, sp)=seq (v,l,sp);
      Ic→invariant 0≤sp≤n;
          states
            mt when sp=0,
            normal when 0<sp<n,
            full when sp=n,
            err otherwise;
  implementation                          s. v' の s. sp 番目の要素を x
                                          に置換えたベクトルを示す.
                          βout
    body push  out (s. sp=s. sp'+1∧s. v=α(s. v', s. sp, x))=
    βin→          mt, normal:: (s. sp←s. sp+1; s. v [s. sp]←x);
                  otherwise:: FAIL;
    body pop out (s. sp=s. sp'−1)=
              normal, full:: s. sp←s. sp−1;
              otherwise:: FAIL;
    body top out (x=s. v [s. sp])=
              normal, full:: x←s. v [s. sp];
              otherwise:: FAIL;
    body empty out (b=(sp=0))=
                normal, full:: b←false;
                mt:: b←true;
                otherwise:: FAIL;
    endform;
```

図-5.3 istack を定義する **form**（[Wulf 76] から引用）

抽象モデルを用いた入出力仕様で与えられる．**representation** 部ではやはりこの抽象モデル (istack の場合は列) を媒体として，プログラム化のための具体データ (istack の場合は v: vector と sp: integer) と抽象データの間に rep 関数を定義する．istack の場合には，rep (v, sp)=seq$(v, 1, sp)$，つまり，ベクトル v と整数 sp の組は，その抽象データとして v の先頭の sp 個の整数を並べてできる列を示すことになる．さらに，具体データに対する不変命題 I_c も示されている．**implementation** 部では抽象データ上の各演算 (push, pop, など) に対し，それを実現するプログラムが具体データを用いて示され，その入出力を規定する命題 β_{in}, β_{out} も付け加えられている．

form の随所に記述された β_{req}, I_a, β_{init}, β_{pre}, β_{post}, I_c, β_{in}, β_{out} といった命題と rep 関数を使えば，**form** により定義される抽象データ型の検証は次の4種類の命題が真であることを証明することに帰着する．

form に対して

(1) 表現 (representation) の正しさの証明

$I_c(x) \supset I_a(\mathrm{rep}(x))$

(2) 初期設定の正しさの検証

$\beta_{req}\{\mathrm{init}\}\beta_{init}(\mathrm{rep}(x)) \wedge I_c(x)$

各抽象演算に対して

(3) 具体演算を使って書かれたプログラムの検証

$\beta_{in}(x) \wedge I_c(x)\{P_A\}\beta_{out}(x) \wedge I_c(x)$

(4) 抽象演算の入出力仕様とそれを実現するプログラムの入出力仕様の関係が正しいことの証明

a) $I_c(x) \wedge \beta_{pre}(\mathrm{rep}(x)) \supset \beta_{in}$

b) $I_c(x) \wedge \beta_{pre}(\mathrm{rep}(x')) \wedge \beta_{out} \supset \beta_{post}(\mathrm{rep}(x))$

(1) では，x が具体不変命題 I_c を満たす正当な具体データならば，x が表現する抽象データ $\mathrm{rep}(x)$ もまた抽象不変命題 I_a を満たす正当なものであることが証明される．

(2) では，β_{req} が真であれば，init 節 (istack に対しては $sp \leftarrow 0$) を実行して生成される具体データ x に対して I_c が真であり，かつその x が表現する抽象データ $\mathrm{rep}(x)$ については β_{init} が真であることが証明される．

(3) では，各抽象演算 A を実現するプログラム P_A (図-5.3 参照) が β_{in} と β_{out} で記述された通りに働くことおよび I_c が各プログラムに対して不変命題であることが証明される．この証明の際に，具体データ型 (istack の場合は vector) が仕様を満たすことが仮定される．一般にはこの具体データ型が仕様を満たすことの検証が，プログラム全体の検証の次の段階になる (istack の場合，もし vector というデータ型がプログラミング言語が提供しているものであれば，これはその言語のコンパイラの検証に帰着する)．

(4) では，抽象演算を実現しているプログラムが，rep 関数で定義される対応のもとで確かに抽象演算をシミュレートしていることが証明される．

たとえば istack の検証は，

form に対して

(1) $0 \leq sp \leq n \supset 0 \leq \mathrm{length}(\mathrm{rep}(x)) \leq n$

(2) $n > 0\{sp \leftarrow 0\}\mathrm{rep}(v, 0) = \mathrm{nullseq} \wedge 0 \leq sp \leq n$

抽象演算 push について

(3) $(0 = s.sp \vee 0 < s.sp < n) \wedge 0 \leq s.sp \leq n$
$\{s.sp \leftarrow s.sp + 1;\ s.v[s.sp] \leftarrow x\}$
$s.sp = s.sp' + 1 \wedge s.v = \alpha(s.v', s.sp, x)$
$\wedge 0 \leq s.sp \leq n$

(4) a) $0 \leq s.sp \leq n \wedge 0 \leq \mathrm{length}(\mathrm{rep}(s.v, s.sp)) < n \supset 0 \leq s.sp < n$

b) $0 \leq s.sp \leq n \wedge 0 \leq \mathrm{length}(\mathrm{rep}(s.v', s.sp')) < n \wedge s.sp = s.sp' + 1 \wedge s.v = \alpha(s.v', s.sp, x) \supset s = s' \sim x$

(他の演算については略)

といった命題が真であることを証明すればよい．

istack に対しては，これらの命題が真であることは明らかである．

(iii) 代数的仕様記述に基づく検証法[Guttag 76]

代数的仕様記述では，演算の間に成立する関係を公理として述べることにより，データ型の振舞いが記述される (**4.3-(ii)** 参照)．従って，抽象データ型の検証は，具体データ型を使って抽象データ型を実現するプログラムが，確かにその抽象データ型の公理を満たすことを示すことになる．ただし，実現に用いた具体データ型はそれ自身の仕様を満たすことを仮定する．

図-5.5 は，**図-5.4** に仕様が示された Stack を Array と Integer の対で実現するプログラムを示している．ここで，Array は**図-5.6** に示されるような仕様を満たす．**representation** 部では，STAK が Array と Integer の対からなる具体データ型を抽象データ型 Stack に対応づける関数 (**図-5.2** の rep 関数) であることが示されている．**programs** 部では，STAK の定義及びその STAK で決定される対応づけのもとで，NEWSTAK, PUSH, POP などの演算を実現するプログラムが，再帰方程式の形で記述されている．

```
type Stack [elementtype: Type]
                    要素のタイプは何でもよいことを示す.
syntax
                  NEWSTACK→Stack,
構成的演算        PUSH (Stack, elementtype)→Stack,
                  POP (Stack)→Stack,
                  REPLACE (Stack, elementtype)→Stack.
非構成的演算      TOP (Stack)→elementtype∪{UNDEFINED},
                  ISNEW (Stack)→Boolean,
semantics
  declare stk: Stack, elm: elementtype;
          POP (NEWSTACK)=NEWSTACK,
          POP (PUSH (stk, elm))=stk,
          TOP (NEWSTACK)=UNDEFINED,
          TOP (PUSH (stk, elm))=elm,
          ISNEW (NEWSTACK)=TRUE,
          ISNEW (PUSH (stk, elm))=FALSE,
          REPLACE (stk, elm)=PUSH (POP (stk), elm).
```

図-5.4 Stack の代数的仕様 ([Guttag 76]) から引用)

```
representation STAK (Array [Integer, elementtype], Integer)
  →Stack [elementtype],
programs
  declare arr: Array, t: Integer, elm: elementtype;
NEWSTACK=STAK (NEWARRAY, 0)
  PUSH (STAK (arr, t), elm)=STAK (ASSIGN (arr, t+1, elm),
                                  t+1),
  POP (STAK (arr, t))=IF t=0 THEN STAK (arr, 0)
                      ELSE STAK (arr, t−1),
  TOP (STAK (arr, t))=ACCESS (arr, t),
  ISNEW (STAK (arr, t))=(t=0),
  REPLACE (STAK (arr, t), elm)=
          IF t=0 THEN STAK (ASSIGN (arr, 1, elm), 1)
                 ELSE STAK (ASSIGN (arr, t, elm), t),
```

図-5.5 Stack の (Array, Integer) 対による実現 ([Guttag 76] から引用)

```
type Array [domaintype: Type, rangetype: Type]
syntax
  NEWARRAY→Array,
  ASSIGN(Array, domaintype, rangetype)→Array,
  ACCESS (Array, domaintype)→rangetype.
semantics
  declare arr: Array, dval, dval1: domaintype, rval: rangetype;
  ACCESS (NEWARRAY, dval)=UNDEFINED,
  ACCESS (ASSIGN (arr, dval, rval), dval)=
          IF dval=dval THEN rval ELSE ACCESS (arr, dval).
```

図-5.6 Array の代数的仕様 ([Guttag 76]) から引用)

programs 部を記述する言語が，仕様の **semantics** 部で演算間の関係を公理として述べるものと全く同じであるということが，この仕様記述に基づいた検証法が比較的簡単である大きな原因となっている．

抽象データ型の検証は，各公理の両辺が等しいことを，具体データ型の公理および **programs** 部の再帰方程式を書き換え規則として使って示すことになる．

たとえば POP に関する2番目の公理，

POP(PUSH(stk, elm))=stk

を示すには，まず stk=STAK (arr, t) なる arr と t の存在を仮定して，

POP(PUSH(STAK(arr, t), elm)=STAK(arr, t)

を示すことになる．次に，PUSHプログラムを使えば

POP(STAK(ASSIN(arr, t+1, elm), t+1))
=STAK(arr, t)

となり，さらに POP プログラムと (t+1)−1=t を使えば

IF t+1=0 THEN STAK (ASSIGN(arr,
t+1, elm), 0)
ELSE STAK(ASSIGN(arr, t+1, elm), t))
=STAK(arr, t)

となる．ここで，stk=STAK(arr, t) なる t に対しては t>0 であることが示されるから（生成帰納法の項参照)．これは結局，

STAK (ASSIGN(arr, t+1, elm), t)
=STAK (arr, t)

に帰着する．これが成立することは，STAKの定義を使い，t に関する数学的帰納法により確かめられる．

この簡単な例からも推測できるように，代数的仕様に基づいた抽象データ型の検証は比較的単純な式の操作に帰着する部分が多いという特色がある．

(iv) 生成帰納法 (generator induction) の原理[Wegbreit 76]

生成帰納法の原理は，抽象データ型を実現する（一般には複雑な）データ構造が満たさなければならない性質を保証するごく自然かつ強力な方法として，[Wegbreit 76] により定式化されたものである．それは，抽象データ型を，その値の集合上に定義されるいくつかの演算で定義するという基本原理から必然的に導びかれるものであり，抽象データ型の検証法における重要な原理である．

生成帰納法の原理は次のように述べることができる．

抽象データ型 T を特徴付ける演算の中で，タイプ T の値を返す演算又はタイプ T の値を変化させる演算の集合を F_1, F_2, $\cdots F_t$ とする．$P(X)$ をタイプ T の値(またはそれを実現するデータ構造)上のある命題とする．もし，任意の $F_i(1\leqq i\leqq t)$ および F_i の入力の内タイプ T を持つすべての値 X に対し $P(X)$ が成立するという仮定のもとに，F_i の出力又は F_i により変化させられた値 Y に対して $P(Y)$ が成立することが証明されれば，タイプ T の任意の要素 x に対して $P(x)$が成立する．

たとえば，Alphard による istack の記述の例題

における，I_a，I_c が常に真であることの証明はこの原理に拠っている．また，図-5.5 の Stack の実現において $t>0$ が常に成立することも，この原理に基づいて容易に示される．

5.4 構成的検証法

今後プログラミングとの関連において重要性が増すと思われる検証法に構成的検証法がある．これは，Dijkstra の検証先行型プログラミング[Dijkstra 76]に代表される方法である．従来の「プログラムを作り上げてからそれが仕様にあっていることを検証する」という考え方からの発想の転換もさることながら，検証をプログラミングの最も基本的な構成要素の1つとして捉えている点が注目される．5.1 で示した階層的検証法は，この構成的検証法の枠組みを与えるものである．この話題については，Dijkstra 自身の [Dijkstra 75, 76] に詳しい．また，再帰プログラムの自動合成との関連においてこの話題について述べている [Manna 77] も興味深い．

6. あ と が き

プログラミング方法論といってもいろいろの内容が想定できる．本稿では，基本的な考え，展開過程を重視した(設計) 方法論，記述法(プログラミング言語)，形式的仕様記述法，検証法に的を絞ってみた．また，個々の説明というより，諸提案の特徴を相互に比較する立場をとった．しかしながら紙面の制限にもまして我々の能力のなさから，内容のレベルが一定していないことは気掛りな点であり，容易に理解し得ない部分も多いのではないかと懸念する．かなり対象を専門的に書きすぎているかもしれないが，参考文献などで舌足らずの点は補って頂くことを期待する．

ここではまとめを兼ねて，今後重要性が増すと思われるが紙面の都合で十分議論できなかった話題について概観する．

並行処理に関しては，単にシステム・プログラムだけではなく単純化されたリーダ・ライタ問題のようにコルーチン機能があれば便利になる場合が現在のユーザ・プログラムにおいても少なくない．本来人間の思考方法は完全に順序的なものではなく並列性をも許容している．しかし今日の計算機の処理の主流が順序的であることが，計算機関係者の思考法を順序的なもののみに制限している危険性がある．

4. でも触れたが，データを共有する複数プロセスが並行的に動作する状況も，外から見たときそのままひとまとまりとして表現するのが自然で，問題の構造を的確に示すことが多く，1つの仕様記述の単位として重要性が認められつつある．

上述の人間の思考方法を支えるものの一つとして，並行処理とくに共有データと同期処理を行うためにセマフォ以来の記述法やメカニズムに多くの提案がある．中でも [Brinch Hansen 73] や [Hoare 74] の monitor は一つの抽象データ型とみなせよう．また，これらの検証法の提案もいくつかある[Habermann 72] [Howard 76][Flon 76]．さらに，[Hoare 78] では入力，出力，並行性がプログラムの構造化に対する基本要素と考え，[Dijkstra 76] の言語 (表-3.1 参照) におけるガーデッド・コマンド (guarded command) の積極的利用と，入力コマンド自身をもガード (条件式) の中に組み入れるなど，新しい記述法の提案を行っている．

もっと一般的な仕様記述／検証法としては，データ上の演算といった事象 (event) 間の半順序関係 (時間的前後関係) を与えることにより複数並行プロセスの仕様記述を行う方法[Greif 77]，新しい状態変数を導入してそれら変数上の表明として仕様記述を行い，Hoare 流の公理を拡張して検証系を構成する方法[Owicki 76]などがある．

この他に注目される動きとして Hewitt の actor 理論[Hewitt 73, 77]を基礎にし並行プロセスの挙動を記述しようというものがある．actor 理論は actor と，その actor 間で行われる“メッセージのやりとり”(message passing) との2つを基礎にして計算の世界を統一的に記述しようとするもので，Hewitt 自身の提案になる人工知能用言語 PLANNER[Hewitt 69]の意味記述の研究から発展してきたものと伝えられる．この方向の研究としては，monitor を拡張した同期の機構 serializer の提案[Atkinson 77]，並行性を持った抽象データ型の仕様記述／検証法の提案とその分散処理システムの(形式的) 仕様記述への応用[Yonezawa 77a, 77b]などがある．

現実的な対策としては，ツールや支援システムが今後充実されねばなるまい．本稿で述べたプログラミング言語の諸性質の多くは記述されたプログラムの形式に影響を与えるものであった．計算機の支援はプログラムを読みこむことから始まり，何らかの冗長な記述を支援のよりどころとしている．しかし APL や INTERLISP の例を引くまでもなく会話型で支援されるプログラミング*もまた魅力的である．この場合に用

いられる言語は，冗長さに基づいた支援を期待するものではなく，簡潔で十分強力な表現が望まれる．今後の使用環境を考えると，この種の言語の研究も更に必要と考えられる．また，プログラミング・システムをすべてソフトウェアでまかなおうとする従来方式のみならず，マイコン利用の量的機動力，大型機の分散処理や計算機網の充実によるハード的支援などを積極的に利用することも考慮すべきであろう．

TSSにしても廉価な標準的機器や目的別特殊機器などに支えられた知的端末と，その上でのソフトウェアシステムの開発などを考え得る現在では夢を自由に現実化できる状況にあるといえる．

末筆ながら，本機会を与えて頂くきっかけとなるとともに平素から御指導賜わる石井治博士に感謝する．

参 考 文 献

[Ambler 76] A.L. Ambler, D.I. Good and W.F. Burger, "Report on the language Gypsy version 1.0", The University of Texas at Austin, ICSCA-CMP-1 (Aug. 1976).

[Ambler 77] A.L. Ambler, D.I. Good, J.C. Browne, W.F. Burger, R.M. Cohen, C.G. Hoch and R.E. Wells, "Gypsy: A language for specification and implementation of verifiable programs", SIGPLAN Notices 12, 3 (Mar. 1977).

[Atkinson 77] R. Atkinson and C. Hewitt, "Syncronization in actor systems",SIGPLAN-SIGACT Symp. on Principles of Programming Languages, Los Angeles (Jan. 1977).

[Boyer 75] R.S. Boyer and J.S. Moore, "Proving theorems about LISP functions", JACM, 22, 1,129-144 (Jan. 1975).

[Brinch Hansen 73] P. Brinch Hansen, "Operating system principles", Prentice-Hall (1973).

[Brinch Hansen 75] P. Brinch Hansen, "The programming language Concurrent Pascal", IEEE SE-1, 2, 199-207 (Jun. 1975).

[Brinch Hansen 77] P. Brinch Hansen, "The architecture of concurrent programs", Prentice Hall (1977).

[Burstall 69] R.M. Burstall, "Proving properties of programs by structural induction", Comput. J. 12, 1, 41-48 (Feb. 1969).

[Chang 78] E. Chang, N.E. Kaden and W.D. Elliot, "Abstract data types in EUCLID", SIGPLAN Notices 13, 3, 34-42 (Mar. 1978).

[Dahl 68] O.J. Dahl, B. Myhrhaug and K. Nygaard "The SIMULA 67 common base language", Norwegian Comp. Center (May 1968).

[Dahl 72] O.J. Dahl and C.A.R. Hoare, "Hierarchical program structures", in "Structured programming", Academic Press (1972).

* このような会話型プログラミングに対するサーベイとしては[Sandewall 78] が興味深い．

[DeBakker 71] J.W. DeBakker, "Recursive programs", Mathematical Center, Amsterdam, Holland (1971).

[Dijkstra 68] E.W. Dijkstra, "Co-operating sequential processes", in "Programming language", Academic Press, (F. Genuys, Ed.) (1968).

[Dijkstra 72] E.W. Dijkstra, "Notes on structured programming" in "Structured programming", Academic Press (1972).

[Dijkstra 75] E.W. Dijkstra, "Guarded commands, nondeterminacy and fomral derivation of programs", CACM 18, 8, 453-457 (Aug.1975).

[Dijkstra 76] E.W. Dijkstra, "A discipline of programming", Prentice-Hall (1976).

[Earley 71] J. Earley, "Toward an understanding of data structures", CACM 14, 10, 617-627 (Oct. 1971).

[Flon 76] L. Flon and A.N. Habermann, "Towards the construction of verifiable software systems", SIGPLAN Notices, 11, 2, 141-148 (1976).

[Floyd 67] R.W. Floyd, "Assigning meanings to programs", AMS Applied Math. Symp. 19, 19-32 (1967).

[Geschke 75] C.M. Geschke and J. Mitchll, "On the problem of uniform references to data structures", IEEE SE-1, 2, 207-219 (Jun. 1975).

[Geschke 77] C.M. Geschke, J.H. Morris Jr. and E.H. Satterthwaite, "Early experience with Mesa", CACM 20, 8, 540-553 (Aug. 1977).

[Goguen 78] J.A. Goguen, J.W. Thatcher and E.G. Wegner, "An initial algebra approach to the specification, and implementation of abstract data type", in "Current trends in programming methodology, Vol. Ⅳ: Data structuring", (R.T. Yeh, Ed.), Prentice-Hall (1978).

[Greif 77] I. Greif, "A language for formal problem specification", CACM 20, 12, 931-934 (Dec. 1977).

[Guttag 75] J.V. Guttag, "The specification and application to programming abstract data types" (Ph. D. Thesis), University of Toronto, CSRG-59 (1975).

[Guttag 76] J.V. Guttag, E. Horowitz and D.R. Musser, "Abstract data types and software validation", USC-ISI, ISI/RR-76-48 (Aug. 1976).

[Guttag 78] J.V. Guttag, E. Horowitz and D.R. Musser, "The design of data type specifications", in "Current trends in programming methodology, Vol. Ⅳ: Data structuring",(R.T. Yeh, Ed.), Prentice-Hall (1978).

[Habermann 72] A.N. Habermann, "Synchronization of communicating processes", CACM 15, 3, 171-176 (Mar. 1972).

[Habermann 73] A.N. Habermann, "Critical comments on the programming language Pascal", Acta Informatica 3, 47-57 (1973).

[Habermann 75] A.N. Habermann, "Path expressions", CMU-CS report (1975).

[Henderson 72] P. Henderson and R.A. Snowdon, "An experiment in structured programming", BIT 12, 1, 38-53 (1972).

[Hewitt 69] C. Hewitt, "PLANNER: A language for manipulating and proving theorems in a robot",

IJCAI-69, Washington, D.C. (1969).
[Hewitt 73] C. Hewitt et al., "A universal modular actor formalism for artificial intelligence", IJCAI-73, Stanford (1973).
[Hewitt 77] C. Hewitt, "Viewing control structure as patterns of passing messages", J. of AI 8, 324-364 (1977).
[Hoare 69] C.A.R. Hoare, "An axiomatic basis for computer programming", CACM 12, 10, 576-580, 583 (Oct. 1969).
[Hoare 72a] C.A.R. Hoare, "Notes on data structuring", in "Structured programming", Academic Press (1972).
[Hoare 72b] C.A.R. Hoare, "Proof of correctness of data representation", Acta Informatica 1, 4, 271-281 (1972).
[Hoare 73] C.A.R. Hoare and N. Wirth, "An axiomatic definition of the programming language PASCAL", Acta Informatica 2, 4, 335-355 (1973).
[Hoare 74] C.A.R. Hoare, "Monitors: An operating system structuring concept", CACM 17, 10, 549-557 (Oct. 1974).
[Hoare 75] C.A.R. Hoare, "Parallel programming: An axiomatic approach", Comput. Languages, 1, 2, 151-160 (Jun. 1975).
[Hoare 78] C.A.R. Hoare, "Communicating sequential processes", CACM 21, 8, 666-677 (Aug. 1978).
[Howard 76] J.H. Howard, "Proving monitors", CACM 19, 5, 273-279 (May 1976).
[Igarashi 75] S. Igarashi, R.L. London and D.C. Luckham, "Automatic program verification I: A logical basis and its implementation", Acta Informatica, 4, 2, 145-182 (May 1975).
[Jackson 75] M.A. Jackson, "Principles of program design", Academic Press (1975).
[Jensen 76] K. Jensen and N. Wirth, "Pascal user manual and report", Springer-Verlag (1976).
[Jones 76] A.K. Jones and B.H. Liskov, "An access control facility for programming languages", CMU-CS report (May 1976).
[Lampson 77] B.W. Lampson, J.J. Horning, R.L. London, J.G. Mitchell and G.J. Popek, "Report on the programming language EUCLID", SIGPLAN Notices, 12. 2 (Feb. 1977).
[Ledgard 74] H.F. Ledgard, "The case for structured programming", BIT 14, 45-57 (1974).
[Liskov 72] B.H. Liskov, "A design methodology for reliable software systems", FJCC-72, 191-199 (1972).
[Liskov 74] B.H. Liskov and S.N. Zilles, "Programming with abstract data types", SIGPLAN Notices 9, 4, 50-59 (1974).
[Liskov 75] B.H. Liskov and S.N. Zilles, "Specification techniques for data abstractions", IEEE SE-1, 1, 7-19 (Mar. 1975).
[Liskov 76] B.H. Liskov and V. Berzine, "An appraisal of program specifications", MIT Lab. for Comp. Sci. CSG-Memo 141, (Jul. 1976).
[Liskov 77] B.H. Liskov, A. Snyder, R. Atkinson and C. Schaffert, "Abstraction mechanisms in Clu", MIT Lab. for Comp. Sci. CSG-Memo 144-1 (Jan. 1977).
[Manna 69] Z. Manna, "The correctness of programs", JCSS 3, 2, 119-127 (May 1969).
[Manna 72] Z. Manna and J. Vuillemin, "Fixpoint approach to the theory of computation", CACM 15, 7, 528-536 (Jul. 1972).
[Manna 73] Z. Manna, S. Ness and J. Vuillemin, "Inductive methods for proving properties of programs", CACM 16, 8, 491-502 (Aug. 1973).
[Manna 74] Z. Manna, "Mathematical theory of computation", McGraw-Hill (1974).
[Manna 77] Z. Manna and R. Waldinger, "The axiomatic synthesis of systems of recursive programs", IJCAI-77, Cambridge (1977).
[Manna 78] Z. Manna and R. Waldinger, "The logic of computer programming", IEEE SE-4, 3, 199-229 (May 1978).
[McCarthy 62] J. McCarthy, "Towards a mathematical science of computation", IFIP-62, 21-23 (1962).
[McCarthy 63] J. McCarthy, "A basis for a mathematical theory of computation", in "Computer programming and formal systems" (P. Braffort and D. Hirschberg, Ed.), North-Holland, 33-70 (1963).
[Mills 71] H.D. Mills, "Top-down programming in large systems", in "Debugging techniques in large systems", Prentice Hall, 41-55 (1971).
[Morris 71] J.H. Morris, "Another recursion induction principle", CACM 14, 5, 351-354 (May 1971).
[Myers 75] G.J. Myers, "Reliable software through composite design", Mason/Charter Pub. (1975).
[Nakajima 77] R. Nakajima, M. Honda and H. Nakahara, "Describing and verifying programs with abstract data types", IFIP Working Conf. on the Formal Description of Programming Concept, New Brunswick (1977).
[Naur 66] P. Naur, "Proof of algorithms by general snapshots", BIT 6, 310-316 (1966).
[Naur 69] P. Naur, "Programming by action clusters", BIT 9, 250-258 (1969).
[Noonan 75] R.E. Noonan, "Structured programming and formal specification", IEEE SE-1, 4, 421-425 (Dec. 1975).
[Owicki 76] S. Owicki and D. Gries, "Verifying properties of parallel programs: An axiomatic approach", CACM 19, 5, 279-285 (May 1976).
[Parnas 71] D.L. Parnas, "Information distribution aspects of design methodology", IFIP-71, 339-344 (1971).
[Parnas 72] D.L. Parnas, "A technique for software module specification with examples", CACM 15, 5, 330-336 (May 1972).
[Popek 77] G.J. Popek, J.J. Horning, B.W. Lampson, J.G. Mitchell and R.L. London, "Notes on the design of Euclid", SIGPLAN Notices 12, 3, 11-18 (Mar. 1977).
[Robinson 77a] L. Robinson and K.N. Levitt, "Proof techniques for hierarchically structured programs", CACM 20, 4, 271-283 (Apr. 1977).
[Robinson 77b] L. Robinson and O. Roubine, "SPECIAL-A specification and assertion language",

Technical Report CSL-46, Stanford Research Institute (Jan. 1977).

[Robinson 77c] L. Robinson, K. N. Levitt, P. G. Newmann and A. R. Saxema, "A formal methodology for the design of operating system software", in "Current trends in programming methodology, Vol. I: Software specification and design", (R. T. Yeh, Ed.), Prentice-Hall (1977).

[Ross 75] D. T. Ross, J. B. Goodenough and C. A. Irvine, "Software engineering: Process, principles and goals", IEEE Computer, 17-27 (May 1975).

[Sandewall 78] E. Sandewall, "Programming in an interactive environment: The LISP experience", Computing Surverys 10, 1, 35-71 (Mar. 1978).

[Schaffert 75] C. Schaffert, A. Snyder and R. Atkinson, "The Clu reference manual", MIT Project MAC (Sep. 1975).

[Shigo 75] O. Shigo, T. Shimomura, K. Iwamoto and T. Maejima, "Implementing the abstraction technique in software development", 2nd USA-Japan Comp. Conf., 517-522 (1975).

[Spizen 75] J. M. Spizen and B. Wegbreit, "The verification and synthesis of data structures", Acta Informatica 4, 2, 127-144 (1975).

[Standish 78] T. A. Standish, "Data structures—An axiomatic approach", in "Current trends in programming methodology", vol. 4: Data structuring", (R. I. Yeh, Ed.), Prentice-Hall, (1978).

[Wegbreit 76] B. Wegbreit and J. M. Spizen, "Proving properties of complex data structure", JACM 23, 2, 389-396 (Apr. 1976).

[Wegner 76] P. Wegner, "Programming languages—The first 25 years", IEEE C-25, 12, 1207-1225 (Dec. 1976).

[Welsh 77] J. Welsh, W. J. Sneeringer and C. A. R. Hoare, "Ambiguities and insecurites in Pascal", Software-Practice and Experience 7, 685-696 (1977).

[Wirth 71a] N. Wirth, "Program development by stepwise refinement", CACM 14, 4, 221-227 (Apr. 1971).

[Wirth 71b] N. Wirth, "The programming language Pascal", Acta Informatica 1, 1, 35-63 (1971).

[Wirth 77] N. Wirth, "Modula: A language for modular multiprogramming", Software-Practice and Experience 7, 1, 3-35 (1977).

[Wulf 76] W. A. Wulf, R. L. London and M. Shaw, "Abstraction and verification in Alphard: Introduction to language and methodology", CMU-CS report (Jan. 1976).

[Wulf 77] W. A. Wulf, "コンピュータ・コンプレックス・システムとソフトウェア・エンジニアリングに関する動向, 第4章 Alphard", 昭和51年度特別セミナー講演録, (社) 日本電子工業会, (1977).

[Wulf 78] W. A. Wulf Ed., "(Preliminary) An informal definition of Alphard", CMU-CS report (Feb. 1978).

[Yonezawa 77a] A. Yonezawa, "Specification and verification techniques for parallel programs based on message passing semantics", (Ph. D Thesis), MIT, Lab. for Comp. Sci., TR-191 (Dec. 1977).

[Yonezawa 7b] A. Yonezawa and C. Hewitt, "Modelling distributed systems", IJCAI-78, Cambridge (1978).

[Zilles 74] S. Zilles, "Algebraic specification of data types", Project MAC Progross Report, MIT, 52-58 (1974).

(昭和53年10月20日受付)