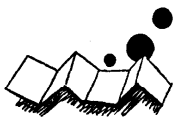
解　説



Ada「批　判」†

中　島　玲　二††

1. はじめに

過去20年間のプログラム言語設計の流れを眺めるにその一側面は，第1の要件である実用性（アルゴリズムを表わす道具としての便利さや効率性）を尊びつつ，いかに第2の要件たる信頼性（その言語で書かれるプログラムが正しく働くことを保証しやすいこと）を向上させるかの知恵比べの歴史として解釈できる．高水準言語の黎明期と比較して第2の要件がより重視されるようになった理由には，ソフトウェアが包含する原理的な困難が認識されるようになったことや，技術的，学問的進歩が第1の要件をそれほど損なわずに第2の目的へ近づく可能性を高めたことが考えられるであろう．しかし現在の発達段階のもとで双方の要件の十全な両立が可能になったわけではない．「発明」されては消えていく類々としたプロトタイプ型プログラム言語（その多くは第2の要件により好意的である．）の山がこの事実を物語る．

この困難は汎用言語においてはなはだしい．PL/1を失敗とみなすかいなかは立場により異なるところであろうが，多くの論争と批判があったことは否めない．巨大な万能言語の存在自体が疑問視され，OS 360への反省もあいまって，ヨーロッパの学者を中心とするプログラム方法論，意味論，証明論の展開があり，その影響のもとで Algol 68 批判派による Pascal が生まれてから10年の年月が経過した．コンピュータ・サイエンスが経験したこのような試行錯誤の歩みを度外視して，これからのプログラム言語の発展を論ずることは不可能であり，アメリカ国防省（以下 DOD と略す．）により進められている統一言語 Ada の試みも例外ではない．

Ada の開発の動期となったいわゆる組み込み型計算機システム (embedded computer systems) に関して DOD が直面する困難の多くは，軍事目的という反進歩的側面は別として，これからの先進的なソフトウェア・システムの一般が克服しなくてはならない課題であり，その深刻さが加速度的に増大しているという認識も妥当なものと考えてよいだろう．しかし単一の汎用言語を開発することでこの問題の解決へ近づけるかいなかは，とりわけ今日のわれわれのソフトウェアに対する理解と水準のもとでは，別の問題である．ソフトウェア的困難さから，その解決のために言語や方法論を考案することの成否は，設計者の技術的，学問的力量とともに，なにがなぜ困難かについて正確で原理的な認識がなされているかいなかに依存するであろう．DOD の抱える問題の解決のために，単一汎用言語の開発が正解であるのか，またたとえ単一汎用言語の立場を承認するとしても，Steelman Requirement [2] に指定される“言語要求仕様”が適当かどうか，さらにこれを容認しても，Ada の設計は妥当であるのか，詳細かつ全面的な検討が必要であることは間違いない．

筆者は決してプログラム言語を主たる専門とするわけではなく，Ada に関する DOD 側の文献を網羅的に調べたわけでもない．Ada の「解説」や「紹介」をするなら部分的知識と理解による記述も許されようが，いやしくも「批判」は設計者の立場を全面的に知り尽さずして軽々に試みるべきものでないことを十分承知したうえで，筆者なりの「批判的コメント」を提示することにした．断定的な否定的論評を目標とするのではなく，むしろ筆者の思いつくところを読んでいただいて，読者の客観的な観察のきっかけになることが少しでもあれば，望外の幸せとする．

本論は自己完結的な記述ではない．批判点の例証として引用する Ada の構成のおのおのを解説することはしない．これについては文献1),3) や本特集に含まれるほかの Ada に関する記事などを参照されたい．

2. Ada 設計批判

DOD の統一言語の試みが決して容易なことでないことは，Steelman に如実に示されていると思う．冒

† Critcal Comments on Ada by Reiji NAKAJIMA (Research Institute for Mathematical Sciences, Kyoto University).
†† 京都大学数理解析研究所

頭「一般設計規準」に挙げられる8項目の美しいスローガンとこれに続く微に入り細をうがつ超具体的な言語要求仕様はきわめて両立しがたい．これは Reference manual [1] において前文に「設計目標」として謳われている理念と，それに続く実際の Ada の定義との間の隔絶としてそのまま引き継がれている．

2.1 統一的設計理念の欠如，概念整理の不徹底

言語は巨大なものとなっている．これは汎用性という目的からやむをえないとしよう．汎用言語は雑多な対象を扱うためのもので Steelman「一般設計規準」1E の簡潔性 (simplicity) の要求が裏切られるのは当然だとする論者もあるだろうが，言語の設計思想に一定の統一的思想があれば言語定義の混乱は汎用言語でもある程度は避けられるはずである．(ここで統一的思想というのは必ずしもフォーマルなものを指すわけではない．) 残念ながら Ada の設計には統一的理念が薄弱で，全体的理解をきわめて困難にしていると考える．この徴候は随所に見られる．例として Ada のモジュール概念である **package** を見てみよう．

package とはデータ対象，データ型，手続きと関数をひとまとめにするための言語構成で，外から見える仕様部*と，その実現**を与える **body** とからなり，いわゆる密封機能***を与えることが目的だが，少々得体の知れぬものとなってしまった．それは必ずしも雑多な対象を含みうるからではなく，もっと原理的なところ——設計者たちがモジュール，抽象化，データ型，ブロック構造などの概念構成を徹底していないこと——に由来すると思われる．以下なぜ筆者がそう考えるかの説明を試みる．

まず Pascal の手続きを思い出そう．それは定義体が大域変数を含まず呼び出しの効果が実引数に対してのみ現れる純粋型と，大域変数で表わされる特定のデータ対象を変化させる副作用型に分けられる．後者の場合，手続きはその特定データ対象の上に固有な演算と考えることもできる．この考え方を進めると SIMULA の class となる．class は1つのデータ対象を表わし，その上には固有の演算群が定義される．class に操作するのはこれらの演算を通してのみ可能である．(Algol や Pascal ではこの制限を設けるのは不可能である．) A を整数のスタックを表わす class とすると，その上の演算 PUSH(X: **integer**) の呼び出しはたとえば A.PUSH(3) となる．A は PUSH のパラメータと見なしてもよい．A.PUSH(3) により A というデータ対象は変化する．

CLU などの抽象データ型は，SIMULA の class を押しすすめて，データ型概念とした．ここではスタックは特定のスタックではなく，PUSH, POP などを演算として持つデータ対象の全体，つまりデータ型，である．**var** X: STACK と宣言すれば，変数Xは任意のスタックを値としうる．Pascal 式に書くと

```
Package STACK is
    procedure PUSH(X: REAL);
    function POP return REAL;
end;
package body STACK is
    MAX: constant=100;
    s: array (1..MAX) of REAL;
    PTR: INTEGER range 0..MAX;
    procedure PUSH(X: REAL) is
    begin
        PTR:=PTR+1;
        s (PTR)=X;
    end PUSH;
    function POP return REAL is
        ⋮
    end POP
```

図-1 **Package** STACK の仕様部と実現部 ([3] より借用)

```
begin
    STACK. PUSH (X);
        ⋮
    X=STACK. POP ( );
end
```

図-2 STACK の使用例

```
package COMPLEX_NUMBER is
type COMPLEX is private
  function "+" (X, Y: COMPLEX) return COMPLEX;
  function "−" (…
      ⋮
  function IM_PART (X: COMPLEX) return REAL;
private
  type COMPLEX is
    record
      RL: REAL;
      IM: REAL;
    end record;
end;
```

図-3 **Package** COMPLEX_NUMBER の仕様部

```
declare
    X: REAL;
    Y: COMPLEX;
begin
    X=COMPLEX_NUMBER. IM_PART (Y);
      ⋮
end
```

図-4 COMPLEX_NUMBER の使用例

* specification
** impeementation
*** encapuslation

PUSH (N: **integer**: **var** X: STACK) はどんなスタック X にも適用できる.

図-1～4 は設計者の1人による Ada 紹介記事[3]から借用した. ここで図-1の **package** STACK は1つのデータ対象を表わし, SIMULA でいえば1つの class に対応するが, 抽象データ型としてはデータ型そのものではなく, むしろその1つの要素または変数に相当する. 一方図-3の COMPLEX_NUMBER の **package** は抽象データ型とも言うべき COMPLEX を定義する.

STACK の表わすスタックは **body** に隠された構造 s と PTR により与えられる. s や PTR は Algol の手続きでは **own** 変数に相当するとしてよい. PUSH や POP を呼び出す前後にも s や PTR の値は保持される. PUSH や POP の効果は STACK なる **package** の定めるデータ対象に対する副作用とみなされる. 一方 COMPLEX_NUMBER なる **package** は抽象データ型 COMPLEX とその上の純粋型手続き※や IM_PART を定義する. STACK.PUSH* の呼び出しは, SIMULA の class A で言えば, A.PUSH に対応し, COMPLEX_NUMBER.IM_PART の呼び出しとは論理的に全く異なるものと考えるべきである.

問題はこの異なった対象が同一の構成要素**package** で与えられ, しかも外側から見ることができる唯一の部分である仕様を眺めても区別がつかぬことである. (図-1と図-3を比べるといかにも区別がつきそうであるが, たとえば STACK を拡張して仕様部に COMPLEX のようなデータ型の定義とその上の手続きを追加してみよ.) Algol や Pascal でも純粋型手続きと副作用型手続きは同じ構文のもとに与えられるが, ここで注意すべきは Algol のようなブロック構造式言語と抽象化機能をもつモジュール式言語の設計思想の基本的相異である. Algol の手続きはあるブロックに局所的に定義され, その副作用はこのブロックの大域変数に対して与えられる. しかも手続きの定義で仕様と実現は未分化である. 定義体 (definition body) という用語が示すように, 冠頭部 **procedure** P (…) とその本体 **begin**…**end** を2つ合せて手続きの定義となす. 本体はあくまでも定義の一部でありこの手続きが副作用を持つことも, **own** 変数を持つことも, 手続きを使う (呼び出す) 側へ知られている. ところが, モジュール式言語では, 仕様部と実現部は峻別される. モジュールを外から使う上で論理的に知られなくてはならない性質はすべて仕様部を見るだけで知られるはずである. **package** が SIMULA の class のようなデータ対象を表現すると同時に抽象データ型のような透明な template を定義でき, しかも仕様部からは相異を判別できないようなことは設計上の誤りと言うより, 設計者が基本的な概念を十分に整理していないことに由来するように思える. このような例は無数にあって, いちいち挙げると際限ないが, 図-3の **"private** type" もその1つである. COMPLEX は **"private"** なデータ型であり, 仕様部に与えられた COMPLEX の **record** によるデータ表現は実は外からは見てはいけないとする. これはおかしな話で, データ表現を隠したければ **body** で行えばよい. 設計者たちは **own** の構造とデータ表現を区別したいのだろうがそれはむしろ, 大きなわく組 (template 的 **package** と class 的 **package** を別の構文とするなど) で行うべきである.

ブロック構造, 抽象化, モジュールなどはそれぞれ独自の基本的理念があって導入された. それらを表面的類似性だけを頼りに十分に消化せずに1つの言語のもとに寄せ集めても成功するはずはない. 文字どおり **"寄せ集め"** となってしまった **package** は氷山のほんの一角にすぎない.

2.2 アルゴリズム記述概念, 言語処理概念, 支援管理概念の混同

Ada のプロジェクトでは Stoneman Requirement に示されるように支援システムなどプログラミング環境の整備が強調されている. これは Ada という特定の言語にあつらえるものであり, Stoneman の内容は別として, 基本的な考え方は当を得ていると思う. おのおののプログラム言語に対し, 支援システムを含む総合的なプログラミング環境を用意することによりプログラム言語をアルゴリズムの記述という本来の目的へ純化させる可能性が高まる. 言語の処理やプログラム作成の支援に属する指定や命令はアルゴリズムの記述から分離されて, それぞれに相応しいコマンドや言語のレベルで行われるようになる. これは言語の設計理念の統一性の保障, ひいてはプログラムの読みやすさ, 簡潔さ, 検証のしやすさにつながる.

ところが残念にも Ada の言語設計にはこのようなプログラミング環境に組み入れられるという仮定が積極的に用いられているようには少しも思われない, 実

* PUSH の正式名は, それを定義する **package** 名とつないで, STACK.PUSH である.

際異なったレベルの概念の混入は言語に無用の複雑さを加えている．(たとえば "**pragma**" の使用例として Reference Manual に与えられている機能の多くや，分割コンパイルなどの翻訳処理機能やライブラリに関する指定など，この観点から整理する必要があろう．)

2.3 意味論上の欠陥

Steelman「一般設計規準」1H には，「言語は完全にかつ曖昧さなく定義されるべし」(この定義とは構文と意味の両方の記述を含む．)「言語はフォーマルに定義されるべし」とある．しかし Reference Manual を読むかぎり，formal semantics はおろか，広義の意味記述すら存在していない部分がある．意味記述とはその言語で文法上正しいプログラムが実行されるときいかなる効果を生じるかの指定である．言語設計はたとえ自然言語で書かれたインフォーマルなものであるにしろ完全な意味論をともなうべきで，それは明快かつ簡潔であることが望ましいというのがここ10年のプログラム言語の学問的常識ではなかったか．(このような認識が常識として受け入れられるようになったこと自体がプログラム意味論や方法論の研究の間接的な成果であると考える．)

ところが Ada では構文上許されるが意味指定をしない構成が多く存在する．たとえば手続きや関数の引数として配列やレコードが用いられると，実引数と仮引数の関係は定義されない ([3]の6-3)．(英語ではまさに "The language does not define…" とある．)言語の実現上の実際的観点による便法であろうと想像できないことはないが，あまりにも安易な解決の仕方ではないだろうか．

一方意味記述が与えられている箇所でも，その多くは冗長で不明確であることは否めない．Reference Manual 前文の「concise and complete」という約束は見事に裏切られて，随所に各項目が数行に及ぶ，長大な箇条書きが与えられている．(たとえば，宣言の "elaboration" の途中で例外事象*が発生する場合の結果の記述 [3の 11.42] や "**task**" の "**wait**" 命令の効果の説明 [3の 9.10] を見よ．(これが concise でないのは一目瞭然だが，complete，つまり箇条書きがあらゆる場合を尽しているかいなかの判定も，不可能でないとしても非常に困難になりうることは確かである．

マニュアルはその言語の設計の質的水準をよく反映する．筆者が Ada の Reference Manual を調べる際，一つの言語構成の意味や規則をきわめるのに何度もページを往復する必要があったり，どうしても満足な記述を見い出せずついに断念することもあった．これは筆者の能力のせいだけによるものでもなかろう．あるいは百科辞典のように大切に詳細に調べればすべて判るのかも知れないが，そのようにマニュアルを書かねばならぬような言語設計自体に問題がある．たとえば各所で意味記述が "データ型が配列である場合"，"データ型がレコードである場合" と各論的に提示される．([3の 12-3] を見よ．)これは設計者によってデータ型という概念の構成が十分にはなされていないことを示すにすぎず，2.1 で述べた欠陥の一つの証拠である．

2.4 階層的プログラミングを支持する言語構成について—批判と提言

Ada のプログラム構成概念で Pascal と本質的に異なる点は **package** や並列処理のための **task** のようなモジュールを取り入れたことである．階層的モジュラ・プログラミングを言語的に支持することが目的であるが，設計方針が一般性を重んじることに傾きすぎ，しかも 2.1 で指摘したような概念整理の不徹底から，むしろプログラム構造を読みづらく複雑にすることにならぬかと危惧する．実際モジュール，手続きや関数，ブロック構造などの構造概念が各々の目的に合せた整理もなしに入り混り，加えてタイプ・パラメータ化*，可視性の命令，ライブラリ概念が混入され不気味な雰囲気をかもしだす．手続きや関数を定義するブロックとモジュールはいくらでも入れ子になれるから，ブロックで局所的に **package** が定義できる．モジュールの本来の目的である自己完結性が全くないような **package** も定義できる．(たとえば，2つの異なった **package** で定義される関数が相互再帰できる．このような関数は当然1つのモジュールに収納されるように規定する構文規則があって然るべきである．) **package** の **body** で，その **package** を含むブロックで宣言される大域変数をいじくることも禁止されていない．**generic** の定義も大域変数を含みうる．よってその instance が異なったブロックで用いられると，大域変数はいつのまにか **generic** が定義されるブロックから instance が使われるブロックへ輸出される ([3の 12.7])．ブロック構造とモジュールの併用は **generic** の使用とあいまって，奇怪なスコープや可視性の規則を生み出すことを設計者たちはいかに考えているのだろうか．

* exception

* type-parametrization

ここでプログラム構成概念の改良についての筆者の提案を2点挙げる．まず第1点として，アルゴリズムを表現する単位としての手続きと関数，その上に階層階造化や密封機能のためのモジュール概念，さらにその上にライブラリ概念の3つのレベルにプログラム構成の仕組みを分離整理すべきである．モジュールは手続きや関数（やデータ型，データ対象）の集合体として定義し，関数や手続きはモジュールの構成要素としてしか導入できないものとする．(overloading* は異なったモジュールの定義する関数の間でのみ許し，同一のモジュール内では許さないことにする．)またライブラリはモジュールの集合体として性格づける．現在の Ada では相手かまわず適用できる **generic** は，モジュールのタイプ・パラメータ化としてのみ許すことにする．(手続きや関数のタイプ・パラメータ化はそれらを含むモジュールをタイプ・パラメータ化することで成就される．)一方ライブラリに関する指定は **2.2** で述べたように異なる言語レベルで行うのが適当であろう．

第2の提案は，ブロック構造の廃止もしくは，使用の制限を導入することである．現在の Ada のようにブロック構造とモジュールが全く自由に入れ子になれるような自由さは構造上の複雑さを増すだけで，益は少ないことはすでに述べたが，基本的にブロック構造とモジュールはなじみにくい．ブロック構造とは文脈的にプログラム構成要素間の依存関係を指定するのに便利だが，モジュールの導入の目的は依存関係を文脈から独立に指定することを含んでいる．実際，モジュールがあればブロック構造は不用になるし，たとえブロック構造を残すにしても，モジュール内に局所的に許すだけで止めるべきである．モジュールの実現部で必要な手続や関数は局所的にも定義できることとし，モジュールを越えて使われる大域変数はデータを定義するモジュールとして与えればよい．要は **"use"**([3の8.4]) を拡張して，モジュール間の依存関係を明確に指定する機能を導入することである．ブロック構造の廃止や制限により，分割コンパイルやライブラリ管理も当然簡単化される．

以上のような規制がプログラマの表現上の自由度を損なうことはないものと考える．不便を感ずるとすればブロック構造に慣れ親しんでもモジュールの本来的使用目的を体得していない場合であろう．

3. むすび

Ada が Steelman の「一般設計規準」に謳われたスローガンのほとんどを満たしていないこと*，ここ10年のプログラミング研究の成果を十分に踏まえているとは思えないことを述べてきた．これは Ada 設計の不備だけでなく，Steelman の欠陥の指摘でもある．(DODの契約を得るために設計された言語であるのだから当然である．) これは DOD の単一万能言語の試みが抱える本質的困難さに由来するところが多いだろう．統一言語が窮極的目標であることをたとえ認めるとしても，未熟な段階での開発とその事実上の押し付けは将来に禍根を残すことになりはしないか，筆者は共通言語の代りに，むしろ**おのおの専門化された機能を持つ複数の言語を共通の設計理念のもとに**（たとえば異種言語によるプログラムの間の接続インタフェースは共通に定義されなければならない．)開発する方が正しい選択ではなかったかと空想する．ハードウェア革命の進行の過程でソフトウェアも多くの新しい概念を生み出していくにちがいない．その中で DOD の問題にも Ada より優れた解決策が考え出されるであろう．Ada がその芽を摘むことがないよう願うほかはない．

DOD という "お上"の用達の方針によって産業界が Ada になびくのは全く無理からぬこととしても，学界としては冷静に見守って，Steelman 前文の美しいスローガン（それは Ada に実現されているとは思えない．)が一人歩きしてソフトウェアの技術，学問，教育の方向が歪められることがないように対処したいものである．

謝辞　本稿で述べた意見（特に **2.2** と **2.4** の内容）は筆者らによる76年以来のイオタ語とその検証・支援・管理・処理システムであるイオタ・プログラミング・システムの設計の経験（による偏向）に基づく．したがってイオタ・プロジェクトの共同研究者に負うところが多い．筆者が Ada に接したのは，協同システム開発株式会社の JADA 委員会に参加させていただいたことに始まる．同会社ならびに同委員会関係者諸兄に深くお礼申し上げる．また本稿の成立については柴山悦哉，島崎真昭，萩野達也，米澤明憲の各氏にお世話になったことを感謝をもって特記する．

* モジュール A で定義される関数 F の正式名を A.F と表わすことにすると，overloading とはモジュール A, B がおのおの関数 A.F, B.F を定義することにすぎない．("A.", "B." の省略を許せば結果的に overloading になる．)

* 巨大な言語だから，1C, 1F の効率性もあやしいがこれは PL/1 の登場のときも同様であった．

参 考 文 献

1) Reference Manual for the ADA Programming Language, DOD (1980).
2) Steelman: Requirement for High Order Computer Languages, DOD (1978).
3) Barnes, J.: An Overview of ADA, Software-Practice and Experience, Vol. 10 (1980).

ほかの文献，特に和文のものについては本特集のほかの Ada 記事の参考文献を見られたい．

(昭和 55 年 12 月 9 日受付)