Ⅳ プログラミング言語の歴史と展望†

中　田　育　男††

1. はじめに

プログラミング言語の歴史を振り返り，最近の成果や今後の課題についてふれてみたい．プログラミング言語の歴史は，より良いプログラムをより容易に作成するための努力の歴史を示している．勿論，言語だけでそれが達成されるものではない．FORTRAN や COBOL が作られた時代は言語が主役であったが，最近は言語以外の各種の方法論の重要性が認識され，ソフトウェア工学という名のもとで研究が盛んである．しかし，それらの方法論はまた言語にも影響を与えている．

ここで取り上げる言語は，特定の問題向き言語ではなく，汎用言語であり，計算機で実行すべきアルゴリズムを記述する言語である．その最も原始的なものは機械語であるが，それをより人間向きに，書き易く，分かり易くするためにプログラミング言語が発展してきた．それは，計算機の生の機能をそのまま使って表現するものから始まり，より抽象化した表現を可能とする方向へと進んできた．プログラミング言語の歴史は一言で言えば抽象化の歴史と言えるかも知れない．しかし，ただ抽象化すればよいというものでもない．その代償として計算機の実行効率の低下をまねくことになる．ただしその度合いは言語だけでなく，言語の翻訳系の作成技術にも依存する．また，言語には，各種の応用に適用できるように豊富な機能が欲しいという要求もあれば，逆に，使いこなすためにはできるだけ単純であってほしいという要求もある．さらに大きな問題として，過去に蓄積されたソフトウェア財産との連続性や，現在および将来のその言語に対する支援体制の問題がある．これらの相矛盾する要求の間でバランスをとることが大切である．

プログラミング言語の歴史に関しては Wegner の優れた評論[1]がある．本稿ではそれより対象を狭くし，情報処理学会の20周年に合せて，この20年間のプログラミング言語の歴史，すなわちアセンブリ言語から最近の Ada にいたるまでの発展過程の中で，主として上記のような点に着目して，いささか個人的に片寄った内容になるかと思うが，私見を述べてみたい．

2. アセンブリ言語

情報処理学会の発足した 1960 年頃は我が国ではアセンブラの開発時代であった．それは「自動プログラミングの研究」という名のもとで行われていた．米国で自動プログラミングのシンポジウム[2]が開かれたのは 1954 年であるから，それに遅れること 5,6 年である．

アセンブリ言語で機械語より進歩したのは命令コードの暗記コード，記号番地，番地式，10 進定数などである．これらは機械語より一歩抽象化されたものと言えよう．特に記号番地は，プログラムの作成／修正時の煩わしい番地計算からプログラマを解放し，番地の内容を示唆する名前によってプログラムを読みとり易くする効果が大きい．

しかし，それだけで自動プログラミングと呼ぶのは今日考えれば言いすぎの感がしないでもない．もっとも 1954 年当時米国ではアセンブラだけでなく，ライブラリからのサブルーチン自動取込みや，浮動小数点機構やインデックス・レジスタのない機械でそれがあたかもあるがごとくに実行するプログラムシステムなども含めて自動プログラミングと呼んでいたようである[3]．当時の我が国のアセンブリ言語の特徴は標準化に努力が傾注されたことである．その言語は SIP (Symbolic Input Program) と呼ばれた．これは記号で書かれたプログラムを直接メモリに読み込むプログラムであった．

3. FORTRAN

FORTRAN 言語で実現された抽象化は，データに

† A Historical Review of Programming Languages by Ikuo NAKATA (Institute of Information Sciences and Electronics, University of Tsukuba).

†† 筑波大学電子・情報工学系

関しては，まず整数型，実数型，その後，複素数型，論理型，さらに最近の文字型というデータ型，および配列と配列要素などがある．制御の流れに関してはFORTRAN の名前のもとになった式 (mathematical FORmula TRANslation system)，関数形式と副プログラム形式のサブルーチン，DO ループ，IF 文などがある．初期の FORTRAN 言語には計算機のスイッチやレジスタ構成，命令語などに影響された機能もあったが，それらは順次より抽象化された一般的なものへと変ってきている．

FORTRAN では，例えば次のような式が書ける．

```
X1=F(I*J, M1)*DAD-(C+1.0)/Y
```

これらの抽象化はプログラム言語の歴史の中でも画期的なものであり，それが勿論 FORTRAN が広く受け入れられていった主なる理由ではあるが，コンパイラ側からの努力も忘れてはならないと思う．

我が国で FORTRAN コンパイラが最初に開発されたのは 1962 年であるが[4),5)]，米国 IBM で最初の FORTRAN 言語が作られたのが 1954 年 10 月であり，IBM 704 のコンパイラの完成は 1957 年 4 月である．コンパイラの完成時期から言えばやはり 5 年の差であるが，質的にも大きな差があったと思う．

IBM 704 FORTRAN の開発者は，言語の仕様もさることながら，それ以上に目的プログラムの効率に重点を置いた．FORTRAN を広く使ってもらうためには，人手によるコーディングの結果と同程度に効率のよい目的プログラムが作成されることを示す必要があると考えたからである[3)]．それはほぼ実現され，そのことが FORTRAN の普及に確かに大きな役割をはたしたと言えるであろう．我々も何度かその目的プログラムを見て感心したものである．それを実現するために，コンパイル時に原始プログラムのシミュレーションをして最も頻度高く実行されそうな所に優先的にインデックス・レジスタを割当てることまでやっている．その頻度をコンパイラに教えてもらうために，初期の FORTRAN 言語には FREQUENCY という文があった．

我が国で最初に作られた FORTRAN コンパイラはとにかく作ることに精一杯で最適化までは考えていなかった．

IBM 704 の FORTRAN は素晴らしい目的プログラムを作り出したが，その代償としてコンパイルに長時間を要するのが問題であった．そこで，実際に大きなプログラムの開発時には，コンパイルの回数を減らしてバイナリ・デックで修正することも行われたようである．FORTRAN はその後 FORTRAN IV へと発展したが，その当初，目的プログラムの最適化には初期ほどの熱意がないように筆者に感じられたのはそのためもあったのであろう．

我が国の HITAC 5020 の最初の FORTRAN IV コンパイラ[6),7)]は，コンパイル時間と実行時間のどちらも適当に速くすることを狙って開発された．その後，コンパイル／デバッグを速くするものと，実行を速くするものとの両方が開発されるようになった．IBM のワトソン研究所で研究開発され，後に商品化された FORTRAN-H コンパイラ[8)]は，データの定義／参照関係の流れを解析して目的プログラムの最適化をはかるものであり，それ以後の，プログラムの解析／最適化の技術発展のもとをなしたものである．

以上のようなコンパイラ技術とメーカの支援体制に支えられて，FORTRAN は普及し続けたとも言えるであろう．

4. ALGOL 60

ALGOL で導入された主なものは，ブロック構造と名前の有効範囲規則，再帰的サブルーチンなどである．例題として n の階乗を求めるプログラムを文献 9) からとる．これは ALGOL 言語の種々の機能を示すために少し冗長に書かれている．

```
integer procedure factorial (n);
  value n; integer n;
  begin integer i;
    if n=0 then factorial:=1
      else begin i:=factorial (n-1);
                 factorial:=n*i
           end
  end
```

ALGOL 以後のプログラム言語ではこのブロック構造が標準的なものとなったが，70 年代に入って，プログラムの信頼性などの見地から，入れ子形式の有効範囲，存在範囲だけでは不十分だと言われてきている．ALGOL の処理系では実行時のメモリ領域がスタック形式で使われる．これは FORTRAN でコンパイル時にメモリ領域の割当てを固定してしまうのと対照的である．

ALGOL は以後の言語設計，処理系の理論や設計にはかり知れない影響を与えたが，普及に関しては，ヨーロッパで比較的普及したものの，米国や日本ではあ

まり普及しなかった．その理由には，メーカの姿勢もあったかと思うが，実行時の効率のよい処理系が作りにくい言語であったことも一つの理由であろう．

5. COBOL

FORTRAN が科学技術計算用の言語であるのに対し，COBOL は事務計算用の言語として作られたものであり，事務計算用の種々の抽象化がなされている．データは，例えば，売上伝票なら文字型データの商品名，数値データの単価などのデータの集合を一つのデータとみなし，次のように書くことができる．

```
01 URIAGE.
  02 HIZUKE PICTURE 9(6).
  02 TOKUISAKI PICTURE X(4).
  02 SYOOHIN.
    03 HINMEI PICTURE X(8).
    03 TANKA PICTURE 9(7).
    03 KOSUU PICTURE 9(5).
```

COBOL が世に普及した大きな理由は，上記のレコード型データなどのデータ記述と操作の機能が強力なことであるが，そのほかに，米国の最大のユーザである米国国防省 (DOD) が言語仕様の設計時からスポンサになり[10]，COBOL が使えることを計算機購入の条件にしたことが大きかったと思われる．

6. PL/I とシステム記述言語

PL/I は IBM が System/360 という 360° の適用範囲を考えた計算機システムの開発にあたって，それにふさわしい言語として開発した言語であり，それまでの主たる言語，FORTRAN, ALGOL, COBOL を集大成したものである．さらに，データ型としてビット列や，番地を抽象化したポインタ，ポインタで参照される基底付変数，例外処理や多重タスクの機能などが加えられた．

それだけ豊富な機能があるにもかかわらず，PL/I の普及速度はあまり速くなかった．その理由には，言語が複雑で学ぶのが容易でないこと，実行効率のよい処理系の作成が容易でないことなどがあるであろう．確かに初期のコンパイラではあまり効率はよくなかったようである．最近はデバッグ用のコンパイラと最適化コンパイラが開発されているが，最適化コンパイラの大きさは約 400 K ステップもある．大型機の通常の FORTRAN コンパイラが 20～50 K ステップ，最適化 FORTRAN で 100 K ステップ (400 K バイト) であるから，その 4 倍である．この数字は PL/I 言語が作られた年 (1964 年で，まだ NPL と呼ばれていた) に予測した[11]ものとぴったり一致する．

ソフトウェアの生産性，信頼性，保守性などの問題から，システムプログラムの記述にも従来のアセンブリ言語に代る高級言語の必要性が高まってきた所へ，ビットやポインタまで扱える PL/I が出現したので，この PL/I またはその方言が各所でシステム記述言語として使われるようになった[12],[13]．もっとも，最初の NPL ではシステムプログラムのことはあまり考えていなかったようで，ポインタの機能は，PL/I コンパイラを PL/I で書こうとして必要になって入れたものらしい[11]．

MIT の Multics では PL/I 言語をほとんどそのまま使ったが，その後コンパイラの負担軽減をはかってサブセットをとったり[14]，実行時の効率を良くするために機械に依存した機能がそれぞれ独自に導入されたりしている[12]．これは抽象化の方向には逆行するものであるが，効率を重んじ，機械の全機能の利用を必要とするシステムプログラマに普及するにはやむを得ないことであろう．

現在ではほとんどの計算機メーカで，PL/I サブセット版に限らず，このような言語が使われていると思うが，その詳細は公表されていないし，ユーザに提供もされていない．企業秘密という面もあるであろうが，一旦公開してしまった言語は，その言語のユーザがいる限り勝手な変更はできず，保守しつづける必要がある，というメーカの事情にもよるのであろう．

7. ALGOL 68, ALGOL N

ALGOL 68 ではユーザがデータ型を定義でき，それらの型のデータに対する演算も定義できる．これはユーザに抽象化の機能を与えたものである．データの型の整合性のチェックにも配慮している．これはプログラムの信頼性，保守性の向上を助けるものとして以後の言語でも重要視されていく．特にポインタに関しては，PL/I がポインタの指す先のデータの型のチェックは全然しないのに対し，そのデータの型をポインタ (**ref**) の型に含めてチェック可能としている．

FORTRAN の後継者として PL/I が作られたように，ALGOL 68 は ALGOL 60 の後継者として作られた．PL/I が従来の言語の機能の合成であるのに対し，ALGOL 68 では直交性を重んじ，できるだけ一般化し，特殊ケースをなくすようにしているが，それ

でも仕様書は分厚く，特殊な記述方式とあいまって，解読は容易でない．またその一般性は効率よい処理系を困難としているようである．それらがやはりこの言語の普及を妨げているのであろう．

日本で ALGOL 68 を検討していたグループは，これが世界の標準になるのは好ましくないと考え，代案として ALGOL N[15] を開発した．その仕様書は薄く，記述は厳密であり，ユーザによる構文規則記述機能はより強力であったが，やはり一般の人には理解が容易ではなかった．

8. 拡張可能言語

ALGOL 68 も ALGOL N も，ともに，まず言語の骨格を定義し，次にその骨格を使って，普通にユーザの使うデータの型や演算，およびその書き方を定義している．それが仕様書を薄くし，定義を厳密にするのに役立っているのであるが，この定義の機能自身をユーザの目から見れば，ユーザが自分に都合のよい機能を自分で定義して使えるようにできる機能である．

これらの言語が開発された頃，このような拡張可能言語の研究が行われるようになり，ALGOL 68 や ALGOL N の定義機能はそのような言語にも応用された[16]．その研究の最盛期[17]には，一つの基本言語とそれを基にした拡張機能さえあれば，プログラミング言語の問題はすべて解決するなどという意見もあったが，実際に拡張可能言語を開発してみると，それを使って作れる言語には限界があるし，開発者には有効に利用できても，一般ユーザには難しく，夢は実現しなかった．

抽象化という言葉で言えば，今までの抽象化は各言語で固定してなされていたものであったが，拡張可能言語はユーザに抽象化の機能を与えたことになる．言語全体をこの機能で構成していくのは実用的でなかったわけであるが，ユーザによる抽象化の機能はプログラムの分かり易さや階層構造を実現するのに重要であり，それらは次の世代の言語に取り入れられるようになる．

9. PASCAL

ALGOL 68 の複雑さに反対し，ALGOL 60 程度の単純さを保ち，ALGOL 60 で処理上問題のあった点をすっきりさせ，データ型の記述機能を豊富にしたものが PASCAL であろう．データ型の中には勿論レコード型やポインタ型もあるが，さらにスカラー型やセット型もある．

従来の言語で

```
RED=1;
BLUE=2;
YELLOW=3;
  ⋮
IF COLOR=RED THEN …
```

と書いていたのをスカラー型として抽象化して

```
var COLOR : (RED, BLUE, YELLOW);
```

とし，さらに

```
COLORSET:=[RED, YELLOW]
```

は従来の

```
COLORSET='101' B;
```

の抽象化である点に筆者は感心したものである．ただし余談であるが，処理系のことを考えて，集合の要素数を1語のビット数以内と制限しているのは惜しい気がする．

PASCAL は，まず各国の大学を中心に普及し，さらに産業界に広まりつつある．その普及の原因は，言語が単純でありながら，豊富なデータ型やストラクチャード・プログラミング向きの制御機能を備えているからであろうが，忘れてならないのは処理系が単純明快で使い易く，他機種に移植するのが容易であったことである．PASCAL の設計者 Wirth は，言語設計の時から処理系を十二分に考慮している．1パス・コンパイルが可能なこと，構文解析は分かり易いトップ・ダウン解析が可能なように構文を LL(1) 文法の範囲内にすること，文頭のキーワードでエラー処理もやり易くすること，などである．その結果，コンパイラは小さく，分かり易く，処理効率もよく，それを読んで移植するのも容易になったわけである．ただし，また余談であるが，そのために文の名札に整数しか許さない[18]のは，GOTO 文がストラクチャード・プログラミングの敵であるとしても，やはり時代に逆行するものであると思う．

10. Ada

1970年代に入って，ソフトウェア工学という名のもとにプログラミングの方法論が盛んに研究され，それを反映したプログラム言語も種々開発されてきた[19]~[29]．そこでの主題は信頼性の高いプログラムを作るための言語であった．

Adaは，毎年巨額の費用をソフトウェアに費やしている米国国防省 (DOD) が，信頼性，保守性の高いソフ

トウェアを，適切な費用で開発するための新言語を求めて[30]，プログラミング言語に関する多数の学者，実務家を動員して設計したものである．それは最近のこの種言語の集大成と言えるかも知れない．

Ada 言語はまだ preliminary version の段階であり，不完全な部分もあるようである．今後はさらに改良されていくであろうし，ユーザが本格的に使うのはこれからであるから，まだ評価できる段階ではないが，ここでは文献31),32) からいくつかの話題を取り上げてみたい．

10.1 データ型

データの型の整合性をコンパイル時または実行時にチェックすることによってプログラムの信頼性，保守性を高めることが強調されている．

型の同等性は構造 (structural equivalence) でなく，名前 (name equivalence) による．例えば

```
type COMPLEX is
  record
        RE : INTEGER;
        IM : INTEGER;
  end record;
type INT_COMPLEX is
  record
        RE : INTEGER;
        IM : INTEGER;
  end record;
type RATIONAL is
  record
        NUMERATOR : INTEGER;
        DENOMINATOR : INTEGER;
  end record;
```

の三者は同じ構造ではあるが，型はすべて違うと考える．その方がより強いチェックができるからである．またコンパイラも単純になる．ユーザに型定義を許した初期の言語 (SIMULA 67, ALGOL 68, PASCAL) では構造が同じものは同じ型とみる「弱い型チェック」であったが，その後の言語 (MESA[23], ALPHARD[24], CLU[25] など) では名前による「強い型チェック」である．そのような型チェックが実際に信頼性，保守性の向上に大きな効果があったという報告もされている[33),34)]．

Adaではさらに，同じ特性の型に違う型名を簡単につけられるようにして，型チェックの機能を使い易くしている．

```
type B is new A;
```

とすればBはAと同じ特性を持つが違う型である．

PASCALの部分範囲型 (subrange, 例えば1..100で1から100までの範囲を示す) に対応するものは，型に対する制限 (constraint)，および型と制限を一緒にした副型 (subtype) として一般化された．

```
subtype LETTER is CHARACTER range
  "A"..."Z";
```

で副型 LETTER は型 CHARACTER で値が "A" から "Z" までのものである．これはよりきめ細かな型チェックを可能にする．

レコード型についてはPASCALの可変部(variant)に対応するものがあるが，Adaでは型チェックをきちんとやることがうたわれている．例えば，PASCALの

```
type MF=(M,F);
     PERSON=
       record
         BIRTH : DATE;
         case SEX : MF of
           M : (ENLISTED : BOOLEAN);
           F : (PREGNANT : BOOLEAN)
       end
```

に対応するものは

```
type PERSON is
  record
    BIRTH : DATE;
    SEX : constant (M,F);
    case SEX of
      when M=>ENLISTED : BOOLEAN;
      when F=>PREGNANT : BOOLEAN;
    end case;
  end record;
```

であり，さらに

```
subtype MALE is PERSON (SEX=>M);
subtype FEMALE is PERSON (SEX=>F);
ANYONE : PERSON;
HE : MALE;
SHE : FEMALE;
```

と宣言（ここで副型 MALE は SEX をMに固定したもの，HE は MALE 型変数）したものに対して，

```
HE:=SHE;
```

はコンパイル時エラーとなり，

```
SHE:=ANYONE;
```

は実行時にチェックされる．このようなチェックは実行効率を落すであろうが，それよりも Ada では信頼性を重視しているのである．その効果の例として引用されている PASCAL コンパイラ[35]においては，テストのある段階でのエラー 64 個の内，18 個 (28%) が可変部の型の違いによるもの，13 個 (20%) が nil pointer による参照，32 個 (50%) が添字や case の選択で範囲を越えたものである (残りの 1 個は無限ループ)．Ada ではこれらのチェックを強化しているわけである．

数値データや演算結果の精度を機械に依存しない形で定義するのは大変難しいので，ほとんどの言語はそれを避けている．PL/I でそれを試みたが，言語を複雑にした一つの原因ともなっており，成功とは言えなかったと思う．Ada では再びそれに挑戦している．例えば

```
type MY_FLOAT is digit 8
  range MIN..MAX;
type F is delta 0.01 range −100.0..100.0;
```

で MY_FLOAT は有効数字 8 ケタ，F は誤差限界が 0.01 である．PL/I の場合より，より人間向きな記述になっている．ただし，その評価は，処理系でどのように効率よく，忠実に実現されるかにかかるであろう．

10.2 副プログラム

ここでもプログラムの信頼性に対する考慮がいくつかされている．

引数に関しては，副プログラムでその値を参照するだけ (**in**) か，それに値を与える (**out**) か，その両方 (**in out**) かを指定できる．例えば

```
procedure PUSH (E: in ELEMENT_TYPE;
  S: in out STACK);
```

といった書き方である．

別名 (aliasing) を禁止している．別名とは例えば

```
procedure P (X: in out INTEGER) is
  begin X=X+A; end
```

なる P を P(A) で呼んだときに，P の中では X も A も同じものを意味してしまうことを言う．別名がなければ，引数の受け渡し方が by reference でも by copying でも，同じ結果が得られる．別名があるとプログラムの解析も困難になるので，プログラムの検証に重点を置いて設計された言語 EUCLID[21] でも禁止していたものである．

副プログラムでは副作用 (side effect) の有無が問題になる．Ada では副作用なしで値を返す **function**，副作用ありで値を返す **procedure**，副作用ありで値を返さない **procedure** の三つを区別することによってユーザに明確に意識させようとしている．

プログラムの読み易さにも種々の考慮がされている．

副プログラムの宣言と実体とを切り離すことができる．従来の ALGOL 系言語ではそれができず，大きなプログラムを読みにくくしていた，宣言と実体の分離はまた分割コンパイル，さらには，インタフェースを明確にした上でのプログラムの分割開発を可能にするものである．この効果は MESA の経験[34]でも強調されている．この考えはより一般化されて，副プログラムだけでなく，データや副プログラムの集合としてのモジュールにも適用されている．

関数名に演算記号を使えるようにしたのも読み易さに効果があろう．例えば

```
functisn "*" (A:COMPLEX; Y:COMPLEX)
  return COMPLEX;
C, D: COMPLEX;
  ⋮
C=C*D;
```

と書くことができる．このような機能は拡張可能言語では豊富に持っていたがユーザには使いきれなかったという反省がある．Ada で，このような演算記号は Ada 言語が本来持っているものだけに制限しているのは賢明であろう．

上の例では，"*" が実数型などの掛算のほかに複素数の掛算としても使えるように多重定義 (overloading) されたことになる．この多重定義はスカラー型の要素名や一般の副プログラム名についても許される．それは読み易くする効果があるが，あまり紛らわしい使い方まで許すことは，かえって読みにくくし，処理系も複雑になる[36]ので避けるべきであろう．

10.3 モジュール

モジュールについては近年多くの議論がされてきたが[19]，それらの成果を，実用性を考慮してうまくまとめているという感じがする．

Ada のモジュールは次の 3 種類のものを表現するものである．①データの集合，②副プログラムの集合，③データ型とそれに対する演算の集合，この最後のものが抽象データ型あるいはカプセル化されたデータ型などと呼ばれて研究されてきたものである．①,② は実用上の必要性からであろう．筆者等も同様のことを

考えたものである[28),29)].

モジュールや副プログラムに関連して，不当なアクセスのチェックや情報隠蔽のために，モジュールの内外での名前の可視性の制御についても多くの議論がされてきた．Adaでは，モジュールの外から参照してもよいものを可視部 (visible part) としてまとめて宣言し，モジュールの外のものを参照するのに，参照したいモジュール名を宣言する．後者には可視性の制限という意味で **restricted** というキーワードが使われる．後者で，モジュール名としたのは，個々の変数や関数の列挙では，読みにくく，書くのも煩わしくなるのを考慮してのことである．

ところで従来の可視性の制御は，いずれもモジュールの入れ子構造による可視範囲を基本として，より細かく制御できるようにしたものであるが，筆者等[29)]は，入れ子構造を越えた可視性も場合によって必要ではないかと思っている．例えば，ある二つのモジュールの間だけでデータの受け渡しをする場合でも，それらのモジュールの共通の祖先でそのデータの宣言をしなければならないのが気になる．Ada[32)]では，その必要性は明記されていないが，その指定も可能なようである．

10.4 そ の 他

並列制御の機構としては，Dijkstra のセマフォ[37)]，さらに Brinch Hansen や Hoare によるモニタ[38)~40)]からランデブー[41),42)]へと抽象化が進んできていた．Adaの方式はランデブーの拡張と言われているが，並列処理されるプログラムが，従来の言語より分かり易く表現されていると思う．

例外 (exception) の扱いは PL/I の ON 条件に似ている．また総称プログラム単位 (generic program unit) があるが，これは制限されたマクロ機能のようなものであり，使い易さを考えて単純化したものであろう．これはライブラリ開発に有用と思われる．余談であるが PL/I の総称関数は Ada では多重定義された関数にあたる．

高級言語を設計するときは，言語のレベルを上げて生産性，信頼性の向上をはかるのと，それにより機械の機能が十分生かせなくなることと天秤にかけて悩むものであるが，Adaではそれを補うために，機械レベルの記述の機能を備えている．これは Ada の開発者が以前に開発した2レベル言語 LIS[43)] での経験を生かしたものであろう．

以上，Adaの個々の機能はそれぞれ良く考えられ適切な選択がなされていると思うが，全体としてはやはり大きな言語となっており，処理系の負担の大きさや，ユーザが十分使いこなせるかが心配である．PL/I や ALGOL 68 よりも複雑になると感じられているようである[44)]が，どうであろうか．ユーザは言語の設計者の思い通りにはなかなか使ってくれないものであることは筆者も経験している．教育が必要である．

Ada の普及にあたっては DOD の力は大きいであろう．Ada の試作処理系は現在 ARPANET を通して全米どこからでも使うことができる．この ARPANET も Ada の普及に一役買いそうである．

11. おわりに

以上，プログラミング言語の歴史を概観してきた．Adaは今までのプログラミング言語の研究，経験の成果を，実用的な言語として集大成したものと言えるかも知れない．しかし，その本当の評価はこれからである．

今後のプログラミング言語の課題としてはいくつかのものが考えられる．一つは設計言語との結びつきである．最近はプログラムを記述する以前の段階の重要性が認識され，要求仕様を記述する言語やプログラム構造を記述する言語が各種提案されている．設計段階の言語とプログラム記述言語とのスムースな繋りは今後の重要な課題である．

もう一つはプログラミングを助ける言語である．プログラミング言語における抽象化は勿論プログラミングを助けるものであるが，Adaを含めて，最近の言語ではプログラムの信頼性，保守性により重点が置かれている．それが重要であることは言をまたないが，プログラミングという人間の知的活動を助けることにもっと目を向けることも必要ではないだろうか．昔のように自動プログラミングとは言わないが，プログラマがよく使うパターンを自動化したり，Floyd がいうようにプログラムの paradigm[45)] (軌範) を自然に表現できる言語である．最近はデータの流れによるプログラムの表現も考えられている．確かにそれが自然な表現である場合も多い．それを言語にとり入れることも考えられよう．

参 考 文 献

1) Wegner, P.: Programming Languages—The First 25 Years, IEEE Vol. C-25, No. 12, pp. 1207-1225 (1976).

2) Symposium on Automatic Programming for Digital Computers, May 13-14, 1954, Washington DC, The Office of Naval Research.
3) Backus, J.: The History of FORTRAN I, II, and III, SIGPLAN Notices, Vol. 13, No. 8, Proc. History of Programming Languages Conference, pp. 165-180 (1978).
4) Takeshita, T.: Survey of Programming Languages in Japan, Proc. 1st USA-JAPAN Computer Conference (1972).
5) 嶋田正三他: HARP 103 について, 第4回プログラミングシンポジウム報告集 (1963).
6) 中田育男他: HITAC 5020 ソフトウェアシステム(2) HARP 5020 第5回プログラミングシンポジウム報告集 (1964).
7) Nakata, I.: On Compiling Algorithms for Arithmetic Expressions, Comm. ACM, Vol. 10, No. 8, pp. 492-494 (1967).
8) Lowry, E. S., et al.: Object Code Optimization, Comm. ACM, Vol. 12, No. 1, pp. 13-22 (1969).
9) Gries, D.: ALGOL 60 Language Summary, SIGPLAN Notices, Vol. 13, No. 8, Proc. History of Programming Languages Conference, p. 1 (1978).
10) Sammet, J. E.: The Early History of COBOL, SIGPLAN Notices, Vol. 13, No. 8, pp. 121-161 (1978).
11) Radin, G.: The Early Histoy and Characteristics of PL/I, SIGPLAN Notices, Vol. 13, No. 8, pp. 227-241 (1978).
12) 中田育男: システム記述言語の最近の傾向, ソフトウェア工学シンポジウム「ソフトウェアツール」報告集, pp. 65-74 (1979).
13) 井田昌之他: 基本ソフトウェアの記述ツール, 情報処理, Vol. 20, No. 6, pp. 519-526 (1979).
14) 中田育男: HITAC 5020 TSS の記述言語としての PL/I サブセット, 昭和43年電気四学会連合大会 2529 (1968).
15) Simauti, T.: ALGOL N, Proc. 1st USA-JAPAN Computer Conference (1972).
16) 中田育男他: 増殖型言語 SELF について, 第12回プログラミングシンポジウム報告集 (1971).
17) Proc. of the Int. Symp. on Extensible Languages, SIGPLAN Notices, Vol. 6, No. 12 (1971).
18) Wirth, N.: The Design of a PASCAL Compiler, Software-Practice and Experierce, Vol. 1, pp. 309-333 (1971).
19) 鳥居宏次他: プログラミング方法論の展望, 情報処理, Vol. 20, No. 1, pp. 22-43 (1979).
20) Brinch Hansen, P.: The Programming Language Concerrent Pascal, IEEE Trans. Soft. Eng., Vol. 1, No. 2, pp. 197-207 (1975).
21) Lampson, B. W., et al.: Report on the Programming Language Euclid, SIGPLAN Notices, Vol. 12, No. 2 (1977).
22) Wirth, N.: Modula: a Language for Modular Multiprogramming, Software-Practice and Experience, Vol. 7, No. 1, pp. 3-35 (1977).
23) Geschke, C. M., et al.: Early Experience with Mesa, Comm. ACM, Vol. 20, No. 8, pp. 540-553 (1977).
24) Wulf, W. A., et al.: Abstraction and verification in Alphard: Introductions to language and methodology, CMU-CS report (1976).
25) Liskov, B., et al.: Abstraction Mechanism in CLU, Comm. ACM, Vol. 20, No. 8, pp. 564-576 (1977).
26) 岩元莞二他: SPOT-6, 高信頼性ソフトウェア開発のための言語システム, 情報処理学会ソフトウェア工学研究会資料 3 (1977).
27) 紫合 治他: 高信頼性ソフトウェア開発のためのプログラミングシステム, 情報処理学会論文誌, Vol. 20, No. 4, pp. 322-329 (1979).
28) 林 利弘他: 制御用ストラクチャード・プログラミング言語 SPL の開発思想他, 情報処理, 17回大会, No. 1-4 (1976).
29) 野木兼六他: トップダウン・プログラミング言語 SPL におけるモジュール概念について, 情報処理学会ソフトウェア工学研究会資料 3 (1977).
30) 上條史彦: プログラミング言語への期待——米国国防省の HOL プロジェクトについて——, 情報処理, Vol. 19, No. 3, pp. 266-274 (1978).
31) Preliminary ADA Reference Manual, SIGPLAN Notices, Vol. 14, No. 6, part A (1979).
32) Ichbiah, J. D., et al.: Rationale for the Design of the ADA Programming Language, SIGPLAN Notice, Vol. 14, No. 6, part B (1979).
33) Mitchell, J. G.: Mesa: A Designer's User Perspective, COMPCON Spring 78, pp. 36-39 (1978).
34) Lauer, H. C., et al.: The Impact of Mesa on System Design, Proc. 4th Int. Conf. on Soft. Eng., pp. 174-182 (1979).
35) Hartmann, A. C.: A Concurrent PASCAL Compiler for Minicomputers, Lecture Notes in Computer Science 50, Springer-Verlay (1977).
36) 近山 隆: プログラム言語 ADA 処理系の試作, 第21回プログラミングシンポジウム報告集, pp. 137-142 (1980).
37) Dijkstra, E. W.: Cooperating Seqnential Processes, in Programming Languages (ed. F. Genuys), Academic Press (1968).
38) Brinch Hansen, P.: Operating System Principles, Prentice Hall (1973).
39) Brinch Hansen, P.: The Programming Language Concurrent Pascal, IEEE Trans. Soft. Eng., Vol. 1, No. 2, pp. 199-207 (1975).

40) Hoare, C. A. R.: Monitors: An Operating System Structuring Concept, Comm. ACM, Vol. 17, No. 10, pp. 549-557 (1974).
41) Hoare, C. A. R.: Communicating Seqnential Processes, Comm. ACM, Vol. 21, No. 8, pp. 666-677 (1978).
42) Brinch Hansen, P.: Distributed Processes: A Concurrent Programming Concept, Comm. ACM, Vol. 21, No. 11, pp. 934-941 (1978).
43) Ichbiah, J. D., et al.: The two-level approach to data independent programming in the LIS system implementation language, in Machine Oriented Higher Level Languages (van der Poel/Maarssen, Eds.), North-Holland, pp. 29-47 (1974).
44) 徳田雄洋: サンタ・クルーズ報告記: プログラミング方法論連続講演会. bit, Vol. 11, No. 12, pp. 76-79.
45) Floyd, R. W.: The Paradigms of Programming, Comm. ACM, Vol. 22, No. 8, pp. 455-460 (1979).

(昭和55年2月5日受付)