

解説

ソフトウェア工学とプログラム言語*

和 田 英 一**

1. はじめに

計算機をハードウェアとよんでいるのを文献でみて、それならと冗談半分にプログラムをソフトウェアとよんだりしていたのは、東大、物理の高橋研で最初のパラメロン計算機 PC-1 ができた直後、はじめて実際にプログラムを計算機にかけられるようになって楽しんでいた頃だから、もう 17 年も前になる。

その後アメリカからきた文献にソフトウェアという言葉を見つけたときは、我と我が眼を疑い、誰もおなじことを考えるものだと感心したものだが、そのソフトウェアも、プログラミングがサイエンスになったのとなんだか同様な過程で、いつのまにか工学になったようで、気がついてみたらこの頃では、東大の情報工学の大学院で森口先生とご一緒に「ソフトウェア工学」という講義を担当することになっていた。

Knuth の分類による科学への道はまだ遠いとしても、せめて芸術でない技術の方のアートにでもなっていれば、工学とよべるかもしれないが、まだブラックアートの要素も方々にみられる現状では、ソフトウェア工学はなんとか上手にソフトウェアを作りたいという願望を表わしているようにも見える。アートにするには、計算機におしえるのは無理でも、少なくとも他人にどうすればよいかわせられる必要があろう。また工学とするためには信頼性や経済性も不可欠の要素であろう。どうすればよいかわについては、プログラムを書くには、処理系を作るには、そしてプログラム言語を設計するにはどうするか、特にプログラム言語に関連したものであろう。プログラムの書き方に関しては、構造的プログラミング（これについての文献は今や枚挙にいとまがない）をはじめとし、なにになによるプログラミングという標題の文献がずいぶん分眼につくようになった。

処理系の作り方は Translator Writing System の登場で、ある種の言語については構文解析はかなり科学のレベルまで来たといえよう。参考書も一応で揃っている。

本稿ではプログラム言語の設計とその周辺の話について、他説自説と混ぜてのべたいと思う。最近我々（森口研）のまわりで比較的好く話の種になる Pascal と BCPL にかたよりすぎるかもしれないが、ご了承ください。

なお空間的制限から 100 編程用意した参考文献の表は思い切り短くした。関心のある向きは直接筆者に問い合わせられたい。

2. プログラム言語の設計方針

この方面の文献で近頃筆者の眼にとまったもののなかに、Hoare¹⁾ と Wirth²⁾ のがある。前者は 1973 年の ACM プログラム言語プリンスiplシンポジウムでの話だが、報告集にはでていない。一方 Wirth は Algol W, Euler, PL 360, Pascal³⁾ 等のプログラム言語の設計でつとに著名だが、後者はかれが昨年の IFIP 大会で行った招待講演である。まず Hoare の方から簡単に紹介するとそれは次のような論調である。

かれによると、プログラム言語で大切なのは、機械と独立であること、規定が安定していること、なじみの記号が使われること、ライブラリーが揃っていること、世間に広く知られていることや強力な組織が支援していることよりなにより、プログラマーの仕事、つまりプログラミング、ドキュメンテーションとデバッグを助けることである。

プログラミングについては、各部分ごとにプログラムのなすべきことや、各部分間の能率のよいインターフェイスがうまく記述でき、いかにプログラムが動くかより、なにをプログラムが実現したいかがのべられなければならない。またよいプログラムは読み易いように書け、コメントが気楽につけられるようでありたい。デバッグは、プログラムがわかり易く書け、ドキ

* Software Engineering and Programming Languages by Eiiti WADA (Faculty of Engineering, University of Tokyo).

** 東京大学工学部計数工学科

コメントーションが注意深くできていれば、そうでない場合に比べて作業量は少いであろうが、プログラム言語はコンパイルの時点でできるだけ多くの虫がとれるようになっていくべきである。そのためには処理系が100%完璧で、処理速度も目的プログラムの速度も十分大きい必要がある(特に処理速度はコンパイル中に型のチェックをするため、サブルーチンは毎回親プログラムと一緒にコンパイルしなおすので重要である)。これらの目標を達成するためには単純である以外に方法はない。

人間のあらゆるいとなみで、その道の上手とは自分の道具を完全に理解している人であり、プログラミングも例外ではありえない。その言語を十分理解し、必要に応じて適切な機能を使えるようにするには、言語の設計方針は単純が第一、そうなれば文法の記述にも矛盾が入りにくく、処理系も完全にし易く、よいことだらけだという(IFIPのアルゴル作業グループにあてた最近の手紙でも、Hoareはonly very simple languagesを設計すべきだと勧告した)。

つぎにWirthの方はAlgol 60の文法書をよんだとき眼からうろこが落ちたような気がしたとのべたあと、言語の設計とはどういうことか、言語の設計にはどうすればよいかを続けてのべているが、言語の設計とは抽象の選択だとかれはいう。これは別の言葉でいえばやはり一種のパターン認識だと筆者には思えるのだが、要するにこういう形はよくでてくるなあとか、こういうふうに考えてみるとよくわかったようだなあとかいう経験を十分につんで、それらのものの中から適当なものを言語の要素として集めることである。

Wirthによると、そのためには設計者が言語の目的と用途の分野について精通していなければならない、また抽象のレベルを揃えるように精選しなければならない(レベルということを考えて、高級言語の中の唯一の機械語命令のようなgo to文は、たしかにちぐはぐである)。そして言語は学ぶにも使うにも容易であれ…から始めて、多くの要求がリストしてあるが、リストの約半分は面白いことに処理系に対するものである。いわく、能率のよいオブジェクトコードができること、いわく、処理系は速かつコンパクトであること、環境の変化に容易に追従でき、他の計算機に簡単に移植できること、処理系開発とドキュメンテーションの時間と費用が小さいこと、そこでとうとう言語の設計とはとりまなおさず処理系づくりであるという結論に達してしまう。

この結論からWirthの足跡をふりかえてみると、かれは言語の設計者であると同時に、いつも優れた処理系製作者であったことに思いあたる。特にPascalではいちやくオブジェクトコードの検討をおこない、処理系をこしらえた。EulerやPL 360でも言語の設計は処理系の作製と一心同体であった。また招待講演にもどるが、最後に言語の設計や保守はひとりで行うに限るといっている。これで連想されるのはどこに書いてあったか思い出せないのだが、van der Poelが「Trac, これはほとんど1個人つまりMooreによって設計された」とのべ、また別のところで全く同じ構文を使って「Lisp, これはほとんど1個人つまりMcCarthyによって設計された」とのべているのを見つけて、van der Poelは面白いことを強調すると感じたことがあった。幾分余談になるが自動車でもF. Porscheのフォルクスワーゲン、A. Issigonisのモリスミニ、A. Citroenのシトロエンなどは「ほとんど1個人によって設計された」ユニークなものであった。

ところでHoareは「単純」ということをくどくくつかえしたが、この単純がどういうものかは大問題で、かれとて決してTuring機械や後述するINTCODEなどは考えていないであろう。Sussmanたちはμ-Plannerの序文で、高級言語というのはプログラミングのスタイルについて仮定をもっていて、そのスタイルでプログラムする限り、指定事項はごく少なくてすむ、アセンブラーは簡単だけど、指定すべき事項はやたらに多くなる。一方高級言語でも、プログラマーのスタイルを仮定していない言語は、かえって使いにくい、Plannerはこういう意味でまた別の新しいスタイルだといっているが、Hoareのいいたいことは筆者が察するにこのスタイルが、Wirth流に言えば抽象が、単純であるような高級言語を設計せよというのであると思う。

これからの言語の設計に影響を及ぼすのではないかなと思われる最近の言語には、Pearl, Clu, Alphardなど、考え方にはclass, monitor, tagなどがある。このうちPearlは構造的プログラム用の道具を与える他、Floyd流の仮定をつけてプログラムのチェックをおこなうことを考えている(この機能はAlgol Wにもある)。Cluはデータの型をそれに対するオペレーションで規定しようとしている点でclassやmonitorとよく似ている。Alphardはレコードのセクターのブロックの外への持ち出しを制御してtag方式のよう

な保護機構をねらっている。これらの考え方が実用になるかどうかは、我々がデータの型にいつのまにか馴染らされたようになるかどうかにかかっているように思われる。

3. システム記述言語

プログラム言語に主要な関心を持っている人たちは特殊用途の言語にはあまり興味がないようだが、特殊目的のものの中なかでも、システム記述言語は別扱いである。それはプログラム言語の処理系を、ゆくゆくはそのプログラム言語で書くにしても、まずは適当な(システム記述)言語で書かなければならないからであろう。言語の処理系をふくめて、システムプログラムといっても決してそれは特殊なプログラムではないのだといっている Brinch Hansen のような人でも、一般利用者にも使えるとはいえず、システム記述言語 Concurrent Pascal を提案している。システム記述用言語にどんなものがあるかは、竹下^{4),5)}や Sammet^{6),7)}のサーベィを参照されたい。こんなに作られ、またこの方面の研究集会もひらかれたが、それでも適当なものがないと嘆いている人がいる。

一般的なシステム記述言語はどうしても機械依存部分がうまく書けず、そこはアセンブラで逃げるということをやったりするのは面白くないと、反対に機械を大いに意識して高級言語の方へよじ登ろうという反動もみられ、この陣営を MOHLL (machine oriented higher level language) とよぶ。MOL Bulletin という印刷物を 1973 年 10 月までに 3 号ほど出版したが、IFIP の研究集会を開催して以来、休止している。

ところで、システム記述言語として比較的話題にのぼるのは Bliss^{8),9)}と BCPL^{10),11)}である。Bliss は PDP-10 のために開発された MOHLL であるが、PDP-11 のものも作られた。MOHLL であるせいか、いくつかの処理系、APL, Simula ごとくシミュレーターなど書いたとはいうものの、「操作システムはまだ書いたことがないらしい」と Sammet にいわれ、どちらかといえば世の中をにぎわしているのは go to 論争に便乗したためかもしれない。それに対して BCPL は Strachey の CPL を Project MAC に滞在していた Richards が CTSS のため単純化したもので、そのため Multics にも開発用語として一応まだ残っている。しかし M. I. T. ではそう使われていず、本国のケンブリッジをはじめ英国の方で人気がある。

まずオックスフォード大学の OS 6 は BCPL で記述されていて、BCPL のシミュレーターを作ったため、計算機が運び込まれてから 45 時間で最初のバージョンは動いたという。ケンブリッジで試作されているケンブリッジケイパビリティ計算機のシステムプログラムも Needham によれば BCPL で記述する予定だそう。その他 MOHLL のひとつ ALIAS も BCPL で開発されているし、M. I. T. の PAL コンパイラーもそうだという。また Snobol 3 やエディターなど文字処理言語のシステム、ケンブリッジの数式処理などにも使われ、他に何に使われたかはもう Richards でさえわからないといっている。最近ではベル研究所にただ B とか C という名前のシステム記述用言語が登場したが、これもベースは BCPL と聞いている。

BCPL がこのようによく使われたのは、やはり CPL を単純にした Basic CPL であることと、後述する移植性のよさであろう。左手の値と右手の値という考え方は、少しなれないと複雑なようだが、一旦わかっしまえばかえって好きなことがいろいろでき、データの型が 1 種類しかないことと調和して、利用者が自由に使う気をおこす。それでいてブロック構造をもち、リカーシブコールをはじめ高級言語なみの制御機構はあり、ケンブリッジ版の文法書もタイプで 40 ページしかないのだから、そう抵抗はなく使えるようである。この文法書の厚さは大変重要で、最近イリノイ大学の人が発表したシステム記述言語 Cleopatra は、文法書が 180 ページもあるから、それだけで食欲がなくなってしまう。

BCPL も丁度手頃なプログラムには丁度手頃なのであって、これで記述された操作システム OS 6 もシングルプロセスであることが重要である。もし複数プロセス用になると、恐らく何か別の機構を入れるか設計しなおすべきかもしれない。Pascal をベースとしたシステム用言語でも、前記 Brinch Hansen の Concurrent Pascal やトロント大学の Sue 言語のように Pascal を拡張しているのは、やはり一般教育用 Pascal もシステム記述には「スタイル的」に不十分だからであろう。そういうわけで Cleopatra が大型言語なのも設計する対象の要求からそれ相応の理由があるのかもしれない。

不十分といえば操作システムを書くにあたって、他のシステム記述言語はどう処理しているのかよく調べていないけれど、どうしても欲しいものが動的コンパ

イル¹²⁾である。これは一時データであったプログラムを処理し、次にそれにプログラムとして制御を移さなければならぬという操作システムの宿命を記述するためである。Algol 68 の報告書は後方の章に入出力と若干の操作システムの機能をプログラムで示しているが、これも大変歯切れが悪い。データをプログラムに変身させるのに Lisp や GPM, Pascal-S などインタプリティブ形式のものだと可能なのだが、コンパイル方式（言語はどちらの方式をとっても構わないのだが）にはあまり例をみない。積極的にプログラムを作りだす機能を持っている言語は Pop-2 で、それには popval という関数が用意されている。プログラムをデータとして扱うという意味では、Simula の class もそれができるといえるが、この方はプログラムを最初から書いておかなければならない点で動的コンパイルとは異なる。

システム記述言語の章の最後では、筆者は Multics の開発者の 1 人 Corbató の述懐にふれたと思う。Multics を開発するに際して高級言語を採用した理由、またそれが PL/I になったいきさつをのべたあと、報告の最後でかれはいう。「もしまたシステムを作るとしたら？ きっとまずシステム記述言語から作るでしょう。EPL や PL/I を使うとしたらもっと裸のものにするでしょう。もし言語を設計するひまがなかったらきっと FORTRAN を使うでしょう」と。

4. ソフトウェアの移植

見出しの「移植」は英語でいうポータビリティとかトランスファビリティのつもりである。すなわち一度ある計算機用にできあがったプログラムを他（機種）の計算機で使うのになるべく不精をする方法である。腕を組んで何もしないでもそのまま移ってくれば申し分ないが、たとえ機械に依存する部分があって書きかえるにしても、はじめからすっかり書き直すより手間がかからない方法なら、ポータブルという資格がありそうである。

一般利用者が書いているようなプログラムは、まず FORTRAN や COBOL などのプログラムを使って書いておけば、そして新機械の方にそれらの処理系が用意してあれば、一応移し得たようなものである（コマンドについては後述）。ようなものというのは、例えば語長や記憶容量など旧計算機の機能を精一杯に使っていると、新計算機に移ったように見えてもうまくゆかない場合がある。Dahlstrand によれば、そのとき

おきる状況は大体つぎのよつつのいずれかである。ある人が 4 ミリオン×4 ミリオンはいくつかと 4 人の男にたずねたら、男 1 は正しく答えた。男 2 はまちがって 16 ミリオンと答えた。第 3 の男はにやっと笑っただけ。そして男 4 は俺にはむずかしすぎるといったそうだ。プログラムを移植して、うけつけた計算機も、利用者からみるとこの 4 人の態度のどれかに相当するので、決して安心はできないという。

システムプログラムとなると FORTRAN や COBOL のとき般用言語では書いてないから、一応もうまくはいかない。操作システムの方はまずあきめの境地だが、言語処理系の方はそれでもいろいろ工夫されている。ACM の研究集会の報告集をみると（般用言語へ？）逆コンパイルするのはどうかとか、エミュレーションを利用せよというような発言がある。この逆コンパイルは成功すると思えない。また処理系を移そうとしても、その処理系が操作システムを利用しているだろうから、移植する範囲がエスカレートして、Hewlett がピーナッツを動かそうとしてゾウまで動かすことになることひやかした。始めから移すつもりでなく作ったものを移そうとすると、このように大さわぎになる。

言語の処理系の移植を計画的におこなうと次のようになる。いま誰かが新しいプログラム言語 L_1 を発明し、計算機 M_1 に処理系を開発したとする。次に M_2 用、 M_3 用と順々に開発し、 M_m まで開発した。FORTRAN などはこの手で開発したのであろう。言語がひとつのうちはこれで結構。しかしプログラム言語 $L_2, L_3, \dots, L_l$ が次々と発明され、そのたびに m 個ずつ処理系を作ったのでは、工学的センスある者、ん？と思うに違いない。もうちょっと経済的にする方法はないかと、当然誰でも考えつくのがいわゆる中間言語 UNCOL 方式である。すなわちすべての言語は L_i から UNCOL へおとす処理系だけを持ち、すべての計算機は UNCOL から M_j へおとす処理系だけを開発する。そうすると $l \times m$ の処理系を作るかわりに $l+m$ だけを作ればよいから、普通にははるかに経済的だという論法である。一般のプログラムからみると、ふたつの処理系をはしごで通らなければならないから、それだけ遅くなるのはたしかだが、そこは我慢してもらう。心配性の人とはまた次のように考えて沈み込むかもしれない。つまり L_i から UNCOL へおとす処理系は所詮 M_j の上で働くのだから、金物がちがえば m 個必要、それなら結局 $l \times m$ の $L \rightarrow$ UNCOL 処理系が必

要で、全然改善にならないと、UNCOL派はこれにはこう答える。図-1 (a) の記号を見てほしい。これは言語 S で書いてあるプログラムを言語 T に変換する言語 C で書いてあるプログラム ($S \rightarrow T$ in C と書く) を意味している。例えば図-1 (b) は、計算機 M_j 用の FORTRAN コンパイラである。 M_j の言語というものは直接走れる機械語のことである。同様に各計算機が用意しておくべき UNCOL から M_j へおとす処理系は c である (FORTRAN や UNCOL は F や U と書く)。UNCOL 派の主張によれば、各言語は $L_i \rightarrow U$ の処理系を U で書いたものをひとつだけ用意する。つまり計算機に無関係で、これは d のものである。心配性の人が必要ではないかと考えたのは、 e のものだった。ところがこれは人間が用意しないでよい。つまり c へ d を入力すると f のような関係で e が作れるのである。 f のような図はなれないとちょっとわかり難いかもしれないが、中央の c と示されたところは、 U で書いてあるプログラム (今は d) を読み込んで M_j で書いてあるプログラム (今は e) に変換するというのだから、 d と e は実行する内容 ($L_i \rightarrow U$, これはしかし c は知らなくてよい) は同じで、記述してある言語が U から M_j にかわるのである。かようなわけで $l+m$ でよい。一般の利用者のプログラム $? \rightarrow !$ が L_i で書かれ (g), U におち (h), M_j におち (i) プロセスを j の絵に示そう。もちろんこのうち左下の d と c は e が一旦できれば毎回使う性質のものではない。これが実はどのくらいまじめに実験されているか不明だが、少なくともコロラド大学には JANUS という中間言語があるそうだ。JANUS の意味は辞書を引いてほしい。この方式がむずかしいのは UNCOL がソース、オブジェクト、システム記述用と3拍子揃って能率よくない

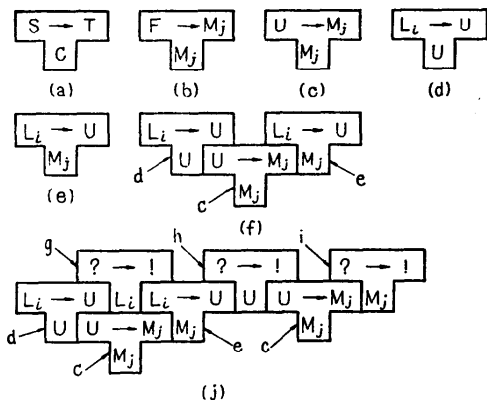


図-1

ればならないからで、実用的には先ほどの杞憂が本当になって結局また $l \times m$ (正確には $l+l \times m$) の処理系を作る事態に戻る。 U を L_i 依存のものにし、そのかわり非常に単純な言語にして、 U_i から M_j への $l \times m$ 個処理系の開発を、 L_i から M_j へのそれよりはるかに容易にしようというものである。そのため言語 L_i の l 発明者により l 個の $L_i \rightarrow U_i$ in U_i が作られ、 U_i の文法書とともに $U_i \rightarrow M_j$ の開発者のところへ開発用キットとして供給される。これは別の見方をすれば、 $L_i \rightarrow M_j$ の処理系のうち、言語に依存した構文解析部分だけを $L_i \rightarrow U_i$ が担担し、オブジェクトコード発生部分を $U_i \rightarrow M_j$ が担当しているのである。最近そのようにして開発され、文献が比較の出まわっているのは毎度おなじみの BCPL と Pascal である。

BCPL の B は Sammet によると Bootstrap の B であって、OCODE とよばれる U_i を開発用キットにして仲間をふやした¹³⁾。RichardsによればこのOCODEはマクロ命令であって、 $U_i \rightarrow M_j$ は各マクロ命令の定義を書くことに相当し、移植はマクロ展開の形でおこなわれるが、そのためにはマクロプロセッサを作るか、これも別の機械から移植しておかなければならない。いずれにしる開発者はマクロを定義し、操作システムとのインターフェイスを作り、それから $L_i \rightarrow U_i$ in U_i をマクロ展開し、 $L_i \rightarrow U_i$ in M_j を作る。あとは一般のプログラムをこれで処理し、オブジェクトをマクロ展開してやればよい。一旦この態勢になったら、マクロ展開の部分を $L_i \rightarrow U_i$ に組み込めば、直接 M_j への処理系が作れるわけである。なお OCODE は(マクロ)命令の数が 56 のスタック機械である。

マクロというのはオープンサブルーチン・サブルーチンの機能は開いていても閉じていてもほとんどかわらない。だからマクロ命令は閉じたサブルーチンのコールと思ってよい。ソースがマクロ命令だけでできているなら、サブルーチンから戻るたびにまたとびだすので、そのくらいなら解釈ルーチンにしても同じで、そのような処理系も作りうる。

さて Richards は移植の経験から、OCODE はブートストラップのためには大きすぎ、移植の手間が予想外に大変だと悟った。そしてもっと簡単なブートストラップ用インタプリタ用言語 INTCODE^{14), 15)} を発明する。これは命令の種類が8, レジスタが6, サブルーチンのリカーシブコールができるほかは、Hoareの喜びそうな単純至極の言語で、筆者は TX-0 や Ten Mini Languages, D 32 などとを連想した。今や BCPL

の開発用キットは INTCODE のものが入ってくる。INTCODE の処理系作りは、インタプリタ、アセンブラの両方で2日といわれているので、そのうち挑戦してみたいと思っている。インタプリタだけあって、移植は早いのが実行のおそいの欠点で、移植がすんだら早く直接 M_i への処理系へ移行する必要がある。東大では井田により INTCODE の方法で BCPL が Tosbac 40 へ移植され、また板野によりダイナミックマイクロプログラムの金物が作られて INTCODE がその上で走ったりしている。

Pascal もほぼ同様な考えで処理系の移植がおこなわれている。もっとも最初の移植、チューリッヒの CDC 6000 からベルファストの ICL 1900 は少しちがった方法であった¹⁶⁾。すなわちチューリッヒには図-2 の a と b があつた。ベルファストではほしいのは d であるが、ベルファストの連中は a を入手してオブジェクトコード発生部を 1900 用に書きかえた c を用意した。そしてチューリッヒへ出かけ、e の要領で不思議な f なるものを得た。この e にもう一度 c を入力して目指す d を得、ベルファストへ帰還したのであつた。

同じような試みをハンブルグのグループが PDP-10 に対して実施した。この方はしかしチューリッヒには旅行できない、近所には CDC が無いというので、ベルファストより悲壮であつた。そこでかれらはなんと PDP-10 の上に CDC-6000 のシミュレーターを作り、長時間かけてベルファストの連中がチューリッヒでやったようなことをおこなつたのである。しかしそのうちチューリッヒで新しい処理系が作られた。これは JANUS や OCODE の影響をうけ、中間言語としてスタック計算機 SC を考え、Pascal → SC in Pascal (これは順次精密化の方法で作られた) と Pascal → SC in SC を開発用キットとして供給するもので、ハンブルグは以後の移植は SC 経由で直接 M_i へおとす処理系まで作りあげた。この SC を使う Pascal は Pascal P^{7),8)} と呼ばれ、現在は IBM 370 のアセンブラで書かれた SC のアセンブラと SC のインタプリタが

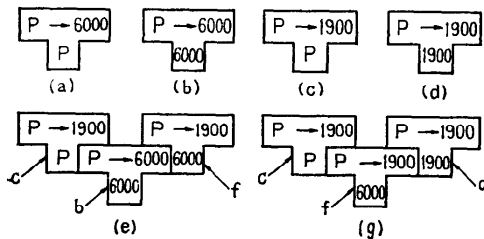


図-2

広くでまわっている。Pascal が BCPL より複雑であるためか、SC も OCODE よりは厄介だが、面白いことに SC も命令の種類は 60 ほど、ただしこちらは 2 アドレスである。SC による移植を図-3 に示す。a は c の部分が SC のインタプリタ、b が開発用キットのコンパイラである。OCODE では任意のソースプログラム(d)をマクロ展開したが、こちらはコンパイルして e が得られ、これをふたたびインタプリタすればよい。これを第1段の移植と呼ぼう。Pascal 国際本部はこれでは演習問題くらいにしか実用にならず、図-3(f)に示す第2段の移植を奨励している。すなわちキットにある $P \rightarrow SC$ in P のオブジェクトコード発生部を改造して g を作り、第1段移植の要領で h を得る。この h に g を入力し i を得るのである。これは SC を 6000、 M_i を 1900 と変更すれば図-2 とまったく同様である。6000 のインタプリタを作るかわり SC のを作るだけで作業はハンブルグのとちがわない、東大では武市により Melcom 7700¹⁹⁾へ、正田、安村により Hitac 8800 へ、更に武市により Facom 230-38 へ第1段移植がおこなわれた。Pascal 国際本部では第2段の作業用としてオブジェクト発生部の組み込み易い Trunk Pascal の供給をはじめた。そこで目下 Facom 230-38 への第2段移植がすすんでいる。

こう書いてくると、移植は簡単と思われるかもしれないが、実は第2段のは $L_i \rightarrow M_i$ よりはましとしてもかなり大変である。Pascal P の文献¹⁸⁾には次のような一節がある。「我々の経験では移植もブーストラップも（この種の文献からつい信じてくなる程度の）わずかな出費では決してできない。高級のマクロプロセッサ、超高級機械語、あるいはホスト計算機などの重装備がなければ、コンパイラのごとき、計算機やシステムに依存するソフトウェアを移すに必要な努力はネグリジブルでない。しかしその努力が、無からソフトウェアを書く努力より、約1桁少ければ、この方法は移植の目的にかなっていると考え、我々の方式は実行可能なものと思っている」。

筆者の思うに、この方法のもうひとつの長所は構文

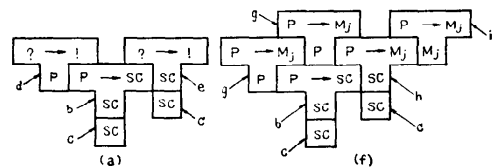


図-3

解析部が共通だから、その意味でどの機械でもコンパティブルな処理系が得られることである。

5. 文 法 書

前の BCPL と Cleopatra の文法書のページ数の引用のし方から、ページ数の少い方が言語が上等のような印象を与えたかもしれないが、そうではない。文法書は言語に便利な機能を追加しても厚くなるし、親切に例を沢山いれても厚くなる。また会話型で利用できる言語などでは、操作システムの説明と思われるものまで記述してあって、いよいよ厚さは急増する。Lisp の例でいえば、最初の Lisp 1.5 のマニュアルは 106 ページ、それが MacLisp では 276 ページ (厚さ 17 mm) となり、Interlisp では 2 次元のページ付けなので全ページ数は不明だが、厚さは 5 cm ほどになっている。このくらい厚くなってくると、文法の一部改訂で、まったく新しく一冊入手するわけにはいかないので、アップデートのアルゴリズムをきめておく必要がある。改訂された部分のマージンに印がつくのは IBM 7040 のマニュアルあたりで最初に見た。しかしこれは活字の印刷で、360 になってからは計算機で編集してラインプリンターで印刷するようになった。CTSS の第 2 版のマニュアルでは、コンソールから改訂の日付と改訂箇所を昔の方へと印刷してみることができ、それで必要なページ数をうちだして、自分のハードコピーファイルに入れておくというようになった。計算機で編集し、これをタイプセットの機械で印刷原版を作ろうとしているのは Algol 68 であり、したのはベル研の C である。Algol 68 は、しかしタイプセット機械にトラブルがあるようでおくれている。タイプセット機械ほどきれいではないが、いろいろなフォントで印刷できる XGP (ゼロックス・グラフィック・プリンター) があちこちの計算機学科や研究所におかれ、それでマニュアルを印刷するのも新しい流行である。前述の MacLisp, Interlisp 共に XGP 製である。

文法の記述については構文はほとんどが BNF で書ける限り BNF を使うようになった。Euler や PL 360 では BNF の中に変数が使われたりもしたが、Algol 68 では 2 レベルの記述を徹底的に使って構文上の制限事項まで盛り込もうとした。馴れると面白いけれどもあまり一般むきとはいえない。

意味の方をフォーマルに書く努力は Lisp, Euler, Algol N, Algol 68 などで試みられ、最近では VDL で

書いてみたというのがときどきあるが、VDL もあまり一般むきではなく、意味を記述したのはこれしかないといわれたらいささか困るであろう。Pascal には Axiomatic Definition と称するものもあるが、実数や goto 文など面倒なところは省略したからあまり意味はない。Pascal は文法書や Axiomatic Definition やマニュアルをみてもまだよくわからないところが少なく、Habermann が文句をつけ、Lecarme が反論する一幕もあった。文法上の問題点をこまかく論じる風潮は Algol 60 以来かと思う。この一幕が Pascal に注目している人が多いせい、それとも文法書が、やや不明確なせいかわからないが、恐らくこの両方であろう。文法書を精密に書いて欲しいのはやまやまであるが、読みにくくなっては困る。しかし世の中のプログラム言語に対する常識も進化したので、昔よりは書き易くなったと思われる。

6. コマンド言語

最後にコマンド言語について簡単にのべる。一般的なプログラム言語は沢山の機種に処理系が作られると標準規格が用意され、プログラムの互換性が一見向上したように見える。それを大きく妨げているのがコマンド言語、ジョブ制御言語 (JCL) である。これは操作システムへの指令言語と解釈されるが、操作システムの設計者の操作システムと言語に対する哲学の個々の相異から、JCL はあたかも機械語のごとくばらばらに発展してきた。操作システムが機械そのものより複雑であるため、なお始末が悪い。せっかく一般的言語 FORTRAN や Lisp で書いたプログラムも、他の機種にもってゆくには、4 章の 4 人の男の例に気をつける他に大量に JCL カードをとりかえなければならない。

昨年スウェーデンで開かれた IFIP のコマンド言語の集会では、大体つぎの三つの方向でこれに対処しようとするように見える。

- i) コマンドをもっとわかり易くする
- ii) コマンドを (つまり操作システムを) 標準化する
- iii) プログラム言語で代用する

このうち i) を実現するには操作システムも簡単にしなければならないであろう。さもないと、ある人の批判のように「ひとつの不可能から他の不可能へ進むだけ」ということになりかねない。ii) もアメリカやイギリスで考えられているようだが、こう無政府状態のコマンド界が統一できるかどうか疑わしい。一応 IBM

流の考えに従って模倣しているシステムが少なくないからその方向にまとまるかとも思うが、それならそれでまたそれが本当によかったかという反省も生じるに違いない。iii) これもここでくわしく紹介するのは不可能だが、例えば Lauesen たちの Algol のインタプリタをコマンドインタプリタにしようというアイデアで、

execute (<program file> <parameter list>)

というような文を入力すると、ライブラリーからこのプログラムを読みだし、パラメーターをそのプログラムに渡して実行させるというのである。しかしこれには Algol の文法を変更しなければならず、また書き方だけに気をとられていると、今の JCL くらい難解なシステムにすぐなる危険性があり、現時点ではやはり少し自由を束縛しても簡単な操作システムだけを用意し、わかり易いコマンド大系をその上に作る努力をあちこちで試みるより仕方がないようである。

次に引用するのは、最近の bit ある記事の結びの句である。

「結論はこうである。ジョブ制御言語は再検討の余地がある。そしてそれにはユーザがもっと積極的に文句を言わなければいけない。莫迦話にしてはまともすぎる結論だが…」

よいシステムは、優れた洞察力を持った人、ほとんど1人で作ればよいことないが、コマンド、ないし操作システムにはそのような例が皆無といわないまでも非常に少い。そして使いにくいシステムができたとしたら利用者が沢山意見をだし、それをとり入れて改造するのがもちろん望ましいが、その改造の仕方もソフトウェア工学の一番むずかしい部分のひとつであろう。

参 考 文 献

- 1) C. A. R. Hoare: Hints on Programming Language Design, p. 29, STAN-CS-73-403, Stanford University (1973)
- 2) N. Wirth: On the Design of Programming Languages, Information Processing 74, pp. 386~393 (1974)
- 3) N. Wirth: The Programming Language PASCAL, Acta Informatica, Vol. 1, No. 1, pp. 35~63 (1971)
- 4) T. Takeshita: Survey of Programming Languages in Japan, First UJCC Proceedings, pp. 231~240 (1972)
- 5) 竹下享: 日本におけるプログラミング言語, bit 臨時増刊プログラミング言語, 1974年8月号, pp. 264~270
- 6) J. E. Sammet: Brief Survey of Languages Used in Systems Implementation, SIGPLAN Notices, Vol. 6, No. 9, pp. 2~19 (1971)
- 7) J. E. Sammet: Roster of Programming Languages for 1973, SIGPLAN Notices, Vol. 9, No. 11, pp. 18~31 (1974)
- 8) W. A. Wulf, D. B. Russel & A. N. Haberman: BLISS: A Language for System Programming, Comm. ACM, Vol. 14, No. 12, pp. 780~790 (1971)
- 9) W. A. Wulf 他: The Design of an Optimizing Compiler, p. 103, Department of Computer Science, CMU (1973)
- 10) M. Richards: BCPL: A Tool for Compiler Writing and System Programming, SJCC 1969, pp. 557~566 (1969)
- 11) M. Richards: The BCPL Programming Manual, p. 40, The Computer Laboratory, University of Cambridge (1973)
- 12) 和田英一, 筭捷彦: 動的コンパイル, 昭和47年度情報処理学会第13回大会講演予稿集, pp. 42~44 (1972)
- 13) M. Richards: The Portability of the BCPL Compiler, Software-Practice and Experience, Vol. 1, No. 2, pp. 135~146 (1971)
- 14) M. Richards: Bootstrapping the BCPL Compiler Using INTCODE. Machine Oriented Higher Level Language, pp. 265~270 (1974)
- 15) M. Richards: INTCODE—An Interpretive Machine Code for BCPL, p. 8, The Computer Laboratory, University of Cambridge (1972)
- 16) J. Welsh & C. Quinn: A Pascal Compiler for ICL 1900 Series Computer, Software-Practice and Experience, Vol. 2, No. 1, pp. 73~77 (1972)
- 17) K. V. Nori, U. Anmann, K. Jensen & H. H. Nägeli: The Pascal <P> Compiler: Implementation Notes, p. 57, Berichte des Instituts für Informatik, Eidgenössische Technische Hochschule Zürich.
- 18) N. Wirth: Pascal and Portability, in Pascal Newsletter No. 2, SIGPLAN Notices, Vol. 9, No. 11, p. 17 (1974)
- 19) 武市正人: PASCAL コンパイラの portability について, 第16回プログラミング・シンポジウム報告集, pp. 90~96 (1975)

(昭和50年6月20日受付)