システム記述言語の記述性

井田昌之（青山学院大学）

§1. はじめに

現在においても進んだ概念を持っているといえるMulticsは主にPL/Iで書かれている。その開発者の1人であるCorbatóは報告の中で，

『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。』

という述懐を述べているという。[1]

また，実用化された最適化コンパイラの先駆の1つであるFortran Hの設計・製作者達は，

『もし高級言語を使わなかったなら，Fortran Hの特徴であるグローバルな最適化機能を組込み，かつデバッグできたかというかは疑わしい。』

と述べ，Fortranをその記述言語に採用した点について感想をもらしている。[2]

オブジェクトコードの実行効率がアセンブリ言語による場合に比べて悪いとか，高級言語だけではすべての機械機能を直接表現することはできないのではないか？　という直観的な疑問で思考をやめてしまったのなら，高級言語でシステムプログラムが書かれることはなかったし，上記のような述懐が今から約10年程前になされることはなかっただろう。

そんなわけで，当然のことながら，

『高級言語でシステムプログラムを書けるか否か？』

という議論をする必要はない。いくつかの実例がこれを裏付けている。

使用実績のある言語を大別するならば次のように分けることができよう。

Ⓐ Fortran，PL/I，Algolなどの普及した汎用言語（ないしはそのサブセット，または拡張版）。例えば2)，3)，4)など。

Ⓑ システム記述を意識して開発された言語。例えば，UNIX[5]を記述したC[6,7]，Pascalその他のコンパイラを記述しているB[8]，実験・研究の段階としては，ミニコンのためのポータブルな記述言語Er[9]，OS記述用のBLISS[10]，Concurrent-Pascal[11]，また，オックスフォード大学。OS 6やSnobol 3その他多数を記述したと文献1に記されているBCPL[12]などがある。
更に，高級言語とはいえないが，Ⓑより機械記述性を重視した意味を持つ，

Ⓒ 中間水準言語。13)など。

ここでは，この3つのタイプの言語のうち，ⒷとⒸについて具体例を用いて説明する。（部分的にアセンブリ言語の助けをかりた）PL/I系の記述言語が商用機においてはよく用いられているが，

・PL/Iそのものはどうしても重い。（同一処理のFortranと比べて，オブジェクトサイズで2倍，実行時間で6倍という報告もある。またコンパイルも遅い。）

・外観的なSyntaxだけがPL/Iに似ているだけで実質的にはⒷにはいるべきPL/I系言語も多い。

などの点から，筆者はPL/Iをはなれた記述言語をもっと積極的に評価し，使用していくべきだと思っている。

標題の「記述性」という言葉には，その意味でこれからのシステム記述言語は，その質が問われるであろうという願いがこめられており，具体的には，

「機械機能のアクセス能力」と「プログラマにとっての書き易さ・読み易さ」

という両面を一語にまとめている。これは，生産性・保守性・効率などについての大きな議論そのものであり，本質的には，デザインの段階からのエイドという話や，使用する機械上に構築されているアーキテクチャの良否などとも関連して研究することが必要と思われる。

§2. システム記述言語の持つべき性質

2.1 端的な例

筆者が過去において経験したあるメインフレームメーカでのコンパイラ開発プロジェクトでの話から始めることにしたい。

言語仕様の設計と並行して記述言語の選定が行なわれた。メーカ側の希望はアセンブリ言語である。我々はそれを用いたくないと思っていた。そこで諸々の状況を考えあわせて，「システム記述言語が整っていないなら，Fortranを使うべきだ。」との提案を行なった。これに対してどのような返事がなされたのであろうか？

記号表の処理の部分をモデルとして，アセンブリ言語によるコーディングと，そのプログラムをFortranに置きかえたもの，そしてそのコンパイルされたオブジェクトのうつの対照表が作成された。それを手に，アセンブリ言語によるコーディングとFortranのオブジェクトとを対比し，“いかにFortranのオブジェクトがアセンブリ言語によるそれに比べて悪いか”をとうとうと説明し，「故に，Fortranを使うべきでない。」と言ったのである。

この話から我々が得るべき事に次の2つがあるだろう。

① 局所的な（小さな）ルーチンの実行効率のみにより使用言語を選定していいのだろうか？

少なくとも数万ステップにわたることが予想されるシステムプログラムの開発には全体を見通しやすく，保守性の高い言語を用いるべきである。事実，この説明のまっ最中にアセンブリ言語によるコーディングの中で，表中のある情報を引き出すための命令はどれかを彼に尋ねたところ，彼（コーディングした本人）は，最初からトレースしなおすことによりようやくそれを発見することができた！

（また，筆者はあまり重い言語を好まないのでそうした方向を重視しないが，前述したFortran Hに関する文献[2]では，「高級言語記述がオブジェクト効率に難があるという指摘は，充分な最適化を行なうことにより対処できる。」ことが述べられている。同様の指摘は他でも見られる。）

② アセンブリ言語によるコーディングのアルゴリズム・手法，そしてデータ構造

はそのまま高級言語に移しても最適でない．
この比較においては同機能を実現するのにFortranでは約4倍かかるとされていたが，そのコーディングはアセンブラによるそのままのコードをFortran化したものである。
i) 表の構造の変更（これにより表の大きさが数割増しになる。）
ii) 処理アルゴリズムの変更
iii) サブルーチン呼び出し時の引数をCOMMON変数に。
iv) 記述方法の改良。（オブジェクトを意識した）
などにより約2倍の大きさまで縮めることができる。
（また，ループ関係のオブジェクトが多少吟味されたものになっていれば，更に2,30%縮めることができる．）

もちろん現在においてFortranを記述言語に採用することが最適な決定であるという事態は少なくなっている。また，Fortran 77[14]では，文字型変数の整数からの分離や，ビット演算関数の組込規定がない，などの明確化（?）が行なわれているので，今後はFortranをスーパーマクロアセンブラのように使うなどという状況はなくなるかもしれない。とはいえ，上述した事例は記述言語の選択に際して行なわれるべき議論をその中に含んでいる。
コンパイラ開発プロジェクトをいかに進めたらよいか。この事についてJ.J. Horningは次のようなゴールを設定しまとめている。[15]

「1) CORRECTNESS：正しく動かなければ，効率や生産コストその他の議論は全く意味がない。しかし，完全に正しいコンパイラを作るのは多くの場合，難しいので，「Reliability」がその代替ゴールとなるだろう。
2) Availability：正しいコンパイラであっても使えなくては仕方がない。
3) Generality & Adaptability：標準性。要求と仕様はたえず変わる可能性がある。
4) Helpfulness：bare-bone compilerとtruly useful oneとは違う。この特徴は偶然含められることではない。エラー診断，開発用ツールの具備など。
5) Efficiency：効率という言葉は，しばしば述べられるが簡単に誤解されてしまう。①コンパイラ開発の効率，②そのコンパイラを用いたプログラム作成の効率（コンパイル効率を含む），③そのコンパイラにより生成されたオブジェクトの効率。」

そして，その技術的な適用立てとして考えられるツールを次のように列挙している。

「i) コンパイラ・コンパイラ：各種の長所をもっているが，実用的なものはまだあまりない。
ii) Standard Design：開発をスクラッチから始める必要はない。例えばMckeeman[16]やGries[17]などを調べて利用せよ。
iii) Design Methodologies：例えばLiskov[18]らの設計手法を用」

いよ。

IV) OFF-THE-SHELF components & Techniques : 過去に使用され，永さいてきたアルゴリズムを利用せよ。

V) Structured Programming : 問題の体系的な整理。例えばDijkstra[19]を参照せよ。

VI) Structured Programs : Structuredな制御構造を使え。また，マクロやモジュール化を利用せよ。

VII) APPROPRIATE Languages : 記述言語の選択は重要な鍵となる。適切な言語の使用により，コンパイラのソースは短くなり，またエラーが減少する。次のような条件を満たす言語がよい。

——読み易く，理解しやすい
——適切なデータオブジェクト（ブーリアン，整数，文字等）とそれらに対するオペレータがある。
——簡潔かつ強力な制御構造とデータ構造がある。（文字，ベクトル，選択等）
——充分なコンパイル時のチェック機能
——モジュール化支援機能（マクロ，手続，データ型定義など），特に独立したコンパイルができること。
——モジュールインタフェイスの分離した，そしてチェック可能な仕様記述を許す。
——機械語へ効率よく写像する。

2.2. 汎用高級言語によるシステム記述の実例

ここでは，過去の商用システムにおいて実際に用いられた汎用の高級言語について紹介を行なう。（文献20，21などにはいくつかの報告がまとめられている。）

(1) Algol

バローズ社のOSはMCP (Master Control Program) と呼ばれる。MCPの開発にはAlgolが用いられている。文献22を見るとB5000の開発の当初より，既にAlgolを意識したハードウェア構造と，システム記述言語としてAlgolを採用した経緯などが述べられている。B5000にはアセンブリ言語は存在しなかった。MCPの初版は手作業により機械語におこされ，作られた。それを用いてAlgolコンパイラが作られ，MCPのAlgol版が製作された。Algol版のMCPは手作業による初版よりも小さく，そして高速であったという。この事はシステムに与える機能の問題よりも，ハードウェアの構造によるとされる。

B6500/6700のMCPの開発には，Extended Algolが用いられているが，これはAlgol 60にビット処理，ストリング処理，リスト処理，イベント処理，非同期処理等の機能を追加したものである。また機械へのアクセス機能を持つ同様の言語ESPOLにより，MCPの機械依存部は記述されているという。

(2) Fortran

NASAでのHoneywell 516, 832のOSと支援システムの開発に利用されている。またIBM社のFortran HはFortranで書かれており，初期開発はIBM709X上で行なわれ，3回のブートストラップにより/360にのせられている。文献2によれば，第1回のブートストラップは709Xから/360へ。第2回，第3回は/360上

でのself-bootstrapである。第2回のブートにより/360上の所要メモリが550Kから約400Kバイトに減り，またビット処理などの言語拡張が組み込まれた。第3回のブート最適化により，コンパイルタイムを約35%減らすことができ，処理能力が2倍になった。そして256Kバイトで約700ステートメント程度をコンパイルできるようになった。データフロー解析に基づく大域的最適化やレジスタの有効利用その他の特徴的機能は，冒頭で述べたように記述言語としてFortranを採用したことにより実現可能になったことが述べられている。

(3) PL/I

GE645上のTSSシステムMulticsはPL/Iにより記述された。[3] PL/Iはそのモジュラリティ，機能の豊富さ，機械独立性などの利点により選ばれている。性能を重視した部分はアセンブリ言語で記述されている。後に保守性のためにPL/Iでコーディングし直した部分もある。全体の規模は/500モジュールで，その内アセンブリ言語のモジュールは250である。（1モジュールは平均200ステートメント） PL/Iコンパイラのオブジェクトはアセンブリ言語記述に比して2倍程度効率が劣るが，これはコンパイラの最適化処理が充分でないことが大きな原因であるとされている。

また，PL/Iそのものではないが，サブセットとなっているもの，一部の機能を強化したものなどがシステム記述用に利用されている。

IBM社のPL/Sなどはその例である。国内においてもいくつかのものが発表されている。ブートストラッピングにより作成され，PL/Iその他のコンパイラの開発などに利用された日電のBPL[23]，DIPS/IのOS等の記述に利用されたSYSL[24]などは「情報処理」に発表された論文での好例である。

8ビットマイクロプロセッサのために開発されたPL/M[25]はマイコンのシステムプログラム作成のためにも利用されている。

メーカーは本格的のPL/I系のシステム記述言語を社内用として，現在でも，使用している。

(4) SP機能の付加された軽装言語

特記すべき程のものではないが，FortranやPL/IにSPステートメント・自動段付などを付加したものが各所で使用されている。プリプロセッサの範囲にはいるものが多い。筆者の近辺でもモジュラープログラミングを意識したもの[26]などが作成され，利用されている。

2.3 既存のPL/I系言語におめられた機能と記述性

アセンブリ言語ではなく，PL/Iをシステム記述言語として採用された理由として，文献23で次のものをあげている。（筆者により若干並べかえられている。）

(1) ビット列の処理，文字列の処理が書ける。
(2) 構造体が扱える。Based変数がある。
(3) メモリの動的割付けができる。
(4) モジュラーに書ける。コンパイルタイムファシリティ（ソースの書き換えなど）がある。各種デバッギング機能がある。ソース自身にドキュメント性がある。
(5) 機械独立性がある。
(6) 他言語と結合可能。

また欠点として次のようなものをあげている。（一部省略）

(a) Syntax ruleはオブジェクトの高効率化に結びつきにくい。また，最適化

コンパイラの作成は実際上難しい。

(b) 使用するステートメントによりオブジェクト効率が変わり，作業者の熟練度により，製品の品質が不均一になるおそれがある。

(c) syntax, semantics が複雑すぎるきらいがある。

(d) 実質的にデータの型変換を必要としない場合の入出力に関するPL/Iの機能はアセンブラのI/Oマクロ以上に"高級"というわけでもない（筆者註：高級でもないのにステップを浪費する。低級なら少々の煩雑さも分かるので"易しくなる"。）

(e) PL/Iコンパイラは大きい。

これらの指摘は，1970年以前の時点でのそのであり，また OS記述ではなくコンパイラ記述用としてのPL/Iについてである点に留意する必要があるが，おおむね理解できるものである。

その後のPL/Iサブセットとなる言語ではアセンブリ言語でしか扱いえなかった細かな指示方法などがつけ加えられているものが多い。

例えば SYSL[24] はOS記述をその目標としており，フォルトその他の割込み記述とか，変数のレジスタ割付などのいくつかの工夫がなされている。

マイコン用のPL/Iとして知られているPL/M[25]はメインフレーム・ミニコンピュータのためのPL/Iとは異なり，その機械の能力にあわせてかなり「軽く」作られている。

例えば変数の型としては，

BYTE（1バイト変数）と ADDRESS（2バイト変数）

の二つだけであり，正常の BIN, DEC, CHAR その他の定義は存在しない。また，構造体の定義も，

```
DECLARE  ENTRY  STRUCTURE (
        A  BYTE ,  B (5) ADDRESS , C  ADDRESS ) ;
```

などのような形式になっている。変数の初期化は一番外側の手続きに属するそのに対してのみ許されるとか，再帰呼出しを行なえる手続名は一番外側のそのだけに対してのみであり，その中には内部手続を含んではならないなどの制約がある。しかし，ポインタ（BASED変数）は一応備わっているし，ENABLE文・DISABLE文とかポートI/O・BYTE ⇄ ADDRESS変換・ビットシフト・その他機械語的な役割をする組込関数が具備されており，実用上は比較的問題がない。

これらのPL/I系の言語は使用する機械に依存したコーディングを可能とする点にむしろ特徴を持ち，syntaxがどうかといえばPL/I流であると解釈し，各々独立した言語として認識すべきであるかもしれない。

2.4 システム記述とタイプレス言語

前述したPL/Mの「型」はBYTEとADDRESSだけであるとしたが，これは記憶域の割付け情報であり，長さ属性の指示である。各変数は符号なし一定数として，文字を格納するため，論理演算の対象とするため，など任意の用途に用いてよい。式の評価に際しては長さの調整だけが行なわれ，型変換は行なわれない。しかし，本来のPL/Iであるならば，例えば I+J という式においてはIとJに対して宣言された型と属性がしかるべく調整され，加算される命令が生成される。後述するBでは，'+'という演算子は整数加算命令を，'#+'という演

算子は浮動小数点加算命令を各々キカイ的に生だせる。PL/Mには+とPLUSの2種の加算がある。

変数の型がそれを使用する演算子によって仮定される言語をタイプレス言語(Typeless Language)という。PL/Mにはタイプレス性がある。これに対して、その変数宣言に型情報を含む言語を型付き言語(Typed language)という。

ほとんどの機械語はタイプレスである。しかし、タグ付き記憶を持つ機械の機械語はTypedである。例えばBurroughs 6700の語は52ビットよりなり、インパリティビット、3フラグビット、48データビットを持つ。ここでの加算命令の対象となる二数はどちらも整数であっても実数であってもよい。また、制御語を演算対象にしようとするとフォルトが生じる。[27]

タイプレス言語の場合、型変換は明示的に行なわなければならない。しかし、多くの機械の機械語はタイプレスであるので機械語への見通しもよく、そのコンパイラも結果として小さくなる利点がある。

Horningが指摘したような、簡潔かつ強力な制御構造その他の機能がTyped言語と同様に装備できるのであれば、

- 機械語との対応がとりやすい
- コンパイラを小さく作成できる

などの点で立つべきものがあると思われる。

◎ブロック構造言語はシステム記述に向いているのだろうか?

いきなりこうした疑問符付きの文を記すのはちょっと唐突かもしれない。しかし、システム記述言語の持つべきQualityとマシンに対する記述性を考えるための一つの問題意識を投げかけてくれる。

システムプログラムはいうまでもなく応用プログラムとは異なり、使用者にとってはオーバーヘッドとなるものである。その開発・保守の効率を上げるためにも、より高級な言語を使用することは望ましいが、実験・研究用は別としても、だからと言って実行効率をいたずらに低下させることは避けなければならない。(ここでの議論に直接関係はないが、はやりのSPのために効率を軽視する傾向がなかっただろうか? StructuredなProgramにしたらメモリもふえ、実行速度も低下したと胸を張って報告する気には単為はなれない。)

アーキテクチャへのシステムプログラムの浸透(各種のファームウェア化、アーキテクチャの再構成)を手段として選択できないここでの立場を考えるならば、必然的にstack frameとheapをその記憶域形態として必要とし、またStackをつなぐ静的・動的連鎖構構を要求するブロック型言語は市場にはまれ、ている多くの機械の構造に直接なじまないものである。(/360型のマルチレジスタの場合、手続きの入れ子の数を制限することにより、定数レジスタ[27]の代用を汎用レジスタにさせるなどのことができ、実行効率を上げることができるが本質的ではない。)ころして処理系の大型化やその他の諸々の問題に影響してくる。

システム記述言語の記述性を分析的に見るならば、Wait[28]やPoole[29]が指摘したコンパイラの2つの機能、すなわち

LDT (Language Dependent Translator):言語構造に依存した翻訳部分。
MDT (Machine Dependent Translator):機械構造に依存した翻訳部分。

の名々について考える必要がある。

LDTは、その出力に機種独立なコード(Abstract Machineの命令)を用いるこ

とにより，言語仕様の議論のみをすることができる。そしてここでの記述性は使用者に対する概念となる。

しかしMDTは異なった体系間のインタフェイスであるので当然Target側の性能（この場合アーキテクチャ）によりその能力も制限される。ここでの記述性はその対応処理能力となる。

このLDTとMDTを意識して設計・実装された言語は最近ふえているが，10年以上前に考えられ普及しているそのそしてIBM社のDOS FORTRAN IVがある。[30] そのコンパイラはPOPと呼ばれる擬似スタックマシンの命令を用いて書かれている。文献30)にはPOPの説明が詳細に述べられている。

この結果，グローバルな最適化はしにくくなるが，いくつかのメリットが考えられる（/360/370のアーキテクチャがかわっても Fortran が直さに使える!?）。

MDTの設計に際しては，従って対象となる機械の構造を吟味する必要がある。例えば2項演算が実際に行なわれるハードウェアリソースには次のようなものがある。

①メモリーメモリ。IBM 1401など。演算レジスタがない。
②単一の演算用レジスタ。いわゆるアキュムレータを持つもの。
③マルチ汎用レジスタ。IBM /360，U1108等。
④階層的レジスタ。レジスタはメモリのバッファという認識で，オペランドはすべてレジスタ中になければならない。CDC 6600等。
⑤スタック。TOS（Top of Stack）レジスタ＋フレームスタッキング。
B 5000～，HP3000等

また，サブルーチンの戻り番地の格納先にはおよそ次の5つがある。

①スタック：（この場合の機構については文献22やコラを参照のこと）
②pure スタック；8080等
③オペランドに指定したハードウェアレジスタ：IBM/360 その他
④特殊なレジスタ：1401等
⑤メモリ：1130等。サブルーチンの先頭番地に戻り番地をいれ，実行はその次の番地から進められる。

引数の受け渡しも同様の方法がある。②から⑤の順に再帰手続や再入手続の構成が困難になる。

（OSも含めた）1つのJobが使う空間には次の2通りがある。

①単一論理アドレス空間：多くの計算機システム
②多重論理アドレス空間：Multicsなどのセグメンテーション。MVS。

前者の場合，動的なメモリの割付け・解放やマルチタスキング，アクセス保護その他にあまり好ましくない。

またどんな機械語命令が用意されているかということも，実際の作成に影響がある。例えばBCPL系では，Iに1を加えることを行なうのに，

```
++I
```

と書ける。これは他の言語の I=I+1 に等しい。1加える命令のある機械にとってその言語が，I=I+1 としか書けないのなら，しっかりした局所最適化の行なわれるコンパイラでなければ，その1加える命令が算術代入文（ずに非ず）のコードに生成されることはない。しかし，++I であれば「軽い」コンパイラでも1加える命令を生成できる。

ここでHorningのかかげたgoalを思い出そう。

1. Reliability (← correctness), 2. Availability
がその筆頭にある。

「確実な動作をし」　かつ　「ちゃんと使える」
ことがシステム記述言語の前提条件であり，その上に「マシン記述性」と「開発・保守性能」がのっているのである。

タイプレスな言語はこれらを満足するだろうというのが筆者の意見である。次章では筆者が研究用に利用している2つのタイプレス言語について紹介する。そしてそれらは商用システムでの使用実績がある。

§3. 2つの例

3.1. GMAP-S[3]

GMAP-Sは日本電気の寺本・小金丸氏らにより開発された言語で，冒頭に述べた中間水準言語に属する。その処理系の初版はSP付加のPL/Iで約4Kステップで書かれ，それに基づき第2版をGMAP-S自身で作成している。現在約5Kステップを要し，初版の3倍以上の速度となっている。アセンブラのプリプロセッサの形式をとり，処理系は小さくなっている。データ定義機能はアセンブラのものをそのまま用いる。また，必要ならばインラインに機械語命令を書き込むことができ，機械記述性は完備しているといえる。

ソースは当然のことながらライブラリ化し編集することができる。またリスティングには自動段付けが行なわれる。（実例等は席上で示す。）使用できるステートメント及びいくつかの特徴の説明を以下に記す。

(i) 手続の構成のための文：PROC，ENTRY（副入口を定義），RETURN，STOP，IPROC（入れ子になった内部手続きの宣言），ICALL（内部手続きの呼出し）
PROCとIPROCとは生成されるシーケンスが異なる。

(ii) 実行制御文：IF，ELSE，CASE（OTHER付），DO UP（制御変数又はインデクスレジスタを増しながらのfor文），DO DOWN（減らしながらのfor文），DO UNTIL；LOOP（repeatとかdo-foreverと同じ），BEGIN，LEAVE（ループの脱出），EPILOG（ループ脱出時の処理），EXIT（後処理指定を含むループからの脱出），GOTO，END

(iii) 代入文：変数，定数及びレジスタに対する二項演算までの代入文，SUBSTR代入文，部分語指定代入文．

(iv) その他文法の要約：a)文はセミコロンで区切る。セミコロンのないものはアセンブリコードとみなされそのまま出力される。b)注釈は，行の先頭に*または;のあるもの。c)ロケーションラベル，パラメータ名，インデクスレジスタ名，配列要素，定数が変数名等となる。d)ワード処理を基本とするがストリング処理記述などにおいては，ハードウェアでも存在する文字列処理命令が生成される。

3.2. B[8]

B言語はBCPLの発展として，ベル研のD.M.RitchieとK.L.Thompsonにより設計された言語である。[31] このBはインタプリタ版であったようである。

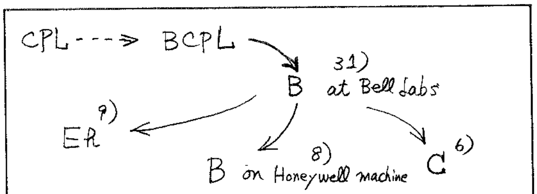

図1 BCPL系言語の流れ

Bell研では，このBを発展させC[6),7)]を作り，UNIX[5)]を製作している。(かしBはそのタイプレス性を生かして Honeywell 6000/66 用にコンパイラが作成され実用に供されている。後者のBはベル研のものに比べて switch 文の拡張，浮動小数点演算子の追加，論理演算子 && 及び || の追加その他が施こされている。ここではこの拡張されたコンパイラ版のBについて述べる。Cはタイプレス言語ではなく，型宣言がある。また structure がはいっている点などがBとの特徴的な違いである。Eれは16ビットミニコンを意識したBといえる。割込制御文の追加などの違いがある。Eれもこともその基本はBとは大差ない。(対比は後上で行なう。)

また，もととなるBCPLはそのポータブルな設計に興味深いものがあるが，紙面に限りがあるのでここでは省略する。文献1や29を参照されたい。

◎ Honeywell版 B[8)] の特徴

a)関数形式の手続きで構成する。変数は大局的と局所的の2レベルがある。もちろん型宣言はない。局所変数は auto 文により確保される。auto 変数のスコープは静的にその定義を含むBody内だけである。手続きの入れ子構造はない。大局変数は extrn 文により参照が可能となる。手続中の変数名は auto中，extrn中，その関数の仮引数リスト中になければならない。関数の値は使わなくてもよい。

b)基本となる記憶単位は語(cell)である。外部名表・ベクトル域・automatic域・手続域を想定している。配列名はポインタである。例えば a[1] は a の内容と1を格納した番地の内容(つまり1)をかえあわせた番地(lvalue)とその値(rvalue)を表わしている。従って a[b]=b[a] であり indirection operator * を使えば *(a+b) とも等しい。この lvalue, rvalue, *演算子などの概念は特徴的である(最近のものでは，文献32などの説明をみるとよい。)。

c)Full-ASCII 使用。識別名の長さは任意だが最初の8文字だけが有効(外部名は6文字まで)。大文字と小文字の区別はない。キーワードは小文字のみ。

d)15のキーワードがある。auto, extrn, if, else, for, while, repeat, switch, do, return, break, goto, next, case, default。次ページの例に使われていないものを説明する。for (expr1; expr2; expr3) statement はいわゆる for 文になっている。repeat statement は break があるまで statement を永久にくりかえす。do statement while (expression); はいわゆる do until。

e)定数には，10進定数，8進定数，浮動小数点定数(単精度)，キャラクタ定数(1～4文字の，1語にはいる文字列，single quote ' で囲む。右づめでパディングはゼロ)，ストリング定数(double quote " で囲まれた文字列。任意長である。コンパイラにより最後にターミネイト文字 null がつけられる。この定数の値はこの文字列へのポインタである。)。またキャラクタ定数，ストリング定数の中には escape sequence があり，*を前につけて任意のオード，8進数などをいれられる。

f)文は式をセミコロンで区切ったもの。あるいは{と}でかこんだ複文。

g)compile 時に name結合される manifest がある。これにより定数や式に名をつけることができる。

h)簡潔かつ強力な演算子群。①単項演算子：#(実数化)，##(整数化)，~(1's comp)，-(2's cmp.)，#-(float neg)，!(logical not)，*(indirection)，&(address gen.)，++(pre & post inc)，--(pre & post dec)。②二項演算子：<<，>>(シフト)，&，~，|(ビット演算)，&&，||(論理演算)，%(整除剰余)，+，-，*，/，#+，#-，#*，#/，==，!=，<，<=，>，>=，#==，#!=，…，#>=。③代入演算子：lvalue=expr，lvalue <op>= expr。<op>は*，/，%，+，-，<<，>>，&，~，|。

(Cとはopの位置が逆)。④Query演算子： expr1 ? expr2 : expr3 ; expr1が真ならばexpr2, 偽ならばexpr3。

ⅱ)コンパイラは4Kステップと小さいが，約36Kのライブラリ関数が用意されている。また外部ファイルからのソースの部分的なとりこみ機能やデバッガをある。

```
100 /* recursive descent translator written by m.ida */   ← 注釈は/*と*/で囲む
110 token (0);   ← 大局変数 tokenの宣言。初期値0をもつ。
120 emit(i)
130   { putchar(i); putchar(' '); }   ← putcharは引数を端末に文字として印字する組込関数
140 scan()
150   { extrn token; token = getchar(); }   ← 大局変数 tokenに端末入力からの1字をもつ
160 main()   ← 主手続。コンパイラはmainという名の手続を探しだしこれを主手続とする。
170   {
180     scan();
190     expr();
200   }
210 expr()
220   {auto t;
230    extrn token;
240    if (token == '-')
250      {scan();term();emit('neg');}
260    else term();
270    while (token == '-' || token == '+')
280      {t = token;
290       scan();
300       term();
310       if(t == '-') emit('neg');
320       emit('add');
330      }
340   }
350 term()
360    {auto tt;
370     extrn token;
380     factor();
390      while (token == 'x' || token =='/')
400      {tt = token;
410       scan();
420       factor();
430        if(tt == 'x') emit('mul');
440        else emit('div');
450       }
460   }
470 factor()
480   {                          "tokenが'0'から'9'までならば"という意味
490    extrn token;
500    switch (token)
510      {case '0':'9' : {emit('lit');emit(token);
520                       scan();  break;}
530       case '(' : {scan(); expr();
540                   if(token == ')' )
550                     {scan(); break;}
560                   else exit();}
570       default :                                   ← caseを満たさなかった時
580              {emit('load'); emit(token); scan();}
590      }
600   }
```

◎switch文による多重分岐は，breakがなければその下の記述へと進むようになっている。

◎++の使用例。図2のプログラムのうちscan及びmainは最初次のように作成していた。

```
110 token (0);
120 pointer (0);   ← 外部名として宣言
130 ibuf [63];

170 scan()
190   {
190     extrn token,pointer;
200     extrn ibuf;
210     token = ibuf[pointer++];
230   }
240 main()
250   {                    ibuf[pointer]をtokenにいれたのち，pointerを1加える
260     extrn ibuf,pointer;
270     while((ibuf[pointer++] =
              getchar()) != '.') ;
280   pointer = 0;
290   scan();
300   expr();
310 }
```

[図2] Bによるプログラム例

```
SYSTEM ?b *
SYSTEM ?time go
a+(b-c)/d-ex(-f);
load a load b load c neg add load d div add load e load f neg mul neg add

On  12/18/78 at 13:42:21.361
End 12/18/78 at 13:42:51.007
elapsed time = 0:00:29.646, processor = 0:00:00.075
key i/o = 88, file i/o = 4

SYSTEM ?go
-5+b/c.
lit 5 neg load b load c div add
```

[図3] 図2のプログラムの実行例(下線が使用者の入力)

§4. おわりに

実用に供しているシステム記述言語についての紹介を行なった。また，PL/I型でない言語の可能性について追及した。お等にこだわるつもりはないが，手引書も50ページ程と薄く，気軽に端末に向かえる点などには魅力がある。話は変わるが1年程前のBYTE誌に「C: A Language for Microprocessors?」などという記事がでていたのを思い出した。

最後に，資料収集その他に協力・御配慮を頂いた諸氏に深謝いたします。

参考文献

1) 和田英一：ソフトウェア工学とプログラミング言語；情報処理 Vol.16 No.10
2) E.Lowry & C.Medlock: Object Code Optimization; CACM Vol.12 No.1 pp13~22 (1969)
3) F.J.Corbató: PL/I as a Tool for System Programming; Datamation pp68 73~76 (May/1969)
4) Burroughs Corp.: Master Control Program Reference Manual; (1970)
5) D.M.Ritchie & Ken Thompson: The UNIX Time-Sharing System; Bell Labs. (1974)
6) D.M.Ritchie: C Reference Manual; Bell Labs. (Jan. 1974)
7) B.W.Kernighan: Programming in C—A Tutorial; Bell Labs (May 1974)
8) R.P.Gurd: User's Reference to B for Honeywell series 6000/66; Univ. of Waterloo (Jan.1977)
9) Reinaldo S.C.Braga: EH Reference Manual; Univ. of Waterloo (1977) (1971)
10) W.Wulf etal.: BLISS: A Language for System Programming; CACM Vol14. No.12 pp780~790
11) P.B.Hansen: The Architecture of Concurrent Programs; Prentice-Hall (1977)
12) M.Richards: BCPL: A Tool for Compiler Writing and System Programming; SJCC 1969 pp 557~566 (1969)
13) 寺井・小笹丈；GMAP-S 解説書；日本電気（社内用）(1978)
14) ANS FORTRAN 77 unofficial Document X3J3/90.5 (1978-6-1)
15) J.J.Horning: Structuring Compiler Development; in Lecture Notes in Computer Science 21, pp498-513, Springer Verlag (1976)
16) W.M.Mckeeman, J.J.Horning: A Compiler Generator; Prentice-Hall (1970)
17) D.Gries: Compiler Construction for Digital Computers; Wiley & Sons (1971)
18) B.Liskov: A Design Methodology for Reliable Software Systems; proc. FJCC pp191~199(1972)
19) E.W.Dijkstra etal.: Structured Programming; Academic Press (1972)
20) J.E.Sammet: Brief Survey of Languages used in systems implementation; ACM SIGPLAN Vol 9 No.9 pp2~9 (1971)
21) 竹下亨：日本におけるプログラミング言語；bit Vol 6. No. 9 pp268~270(1974)
22) D.M.Bulman: Stack Computers; IEEE computer pp14~16 (May/1977)（那須はbit誌に近々掲載される。）
23) 小久保靖世他：コンパイラ記述用言語BPL；情報処理 Vol.11 No.6 pp342~349
24) 寺島信義他：システム製造用言語SYSL-2の設計：情報処理 Vol16. No.8 pp692~697
25) Intel: PL/M-80 プログラミングマニュアル; 1977
26) 間野浩太郎：Fortran-MDL/I；私情協教育ソフトウェア研究委員会資料(Oct.1978)
27) E.I.Organick: 計算機システムの構造（土居範久訳）；共立出版 1978. pp 98~99
28) W.M.Waite: Relationship of Languages to Machines; in L.N.in C.S. 2) pp170-194 SpringerVerlag (1976)
29) P.C.Poole: Portable and Adaptable Compilers: in L.N.in C.S.21 pp427~497 SpringerVerlag (1976)
30) IBM: /360 DOS Fortran IV Program Logic Manual; No. GY28-6394-2 (1975)
31) Johnson.S.C. & B.W.Kernighan: The Programming Language B; Bell Labs (1972)
32) A.Aho & J.Ullman: Principles of Compiler Design; Addison Wesley (1970) pp 50~72