解　説

プログラム・トランスファラビリティ*

松　下　　温** 山崎晴明** 丹下栄二**

1. はじめに

ソフトウェアの危機ということがいわれるようになってから久しい．グロシュの法則によれば，コンピュータを大型化すればするほどそのコスト・パフォーマンスは向上するといわれてきた．しかし，それに伴ってソフトウェアが巨大化し，またそのためにソフトウェアの開発，メインテナンスに困難さが生じてきた．そのうえ，巨大化したソフトウェアを作成したりするためには非常に大きな工数を必要とし，プログラマ管理面においても大きな問題が起るようになってきた．さらに，その作成費用も巨額化してきたために，そのコスト削減が叫ばれるようになってきた．

このようにソフトウェアに費す費用が増大してきたために（図-1参照），これをできるだけ減少させようという考えが発生することは自然である．その1つの方法として，たとえばソフトウェアの一部をハードウェア化する，すなわちマイクロ・プログラムを用いてファームウェア化するということがあげられる．

そのもう1つの方法として，プログラムにトランスファラビリティを持たせるという方法がある．これには作成したプログラムをできるだけ有効に使用しようということが基本としてある．すなわち，特定のマシンでしか動けないようなプログラムのつくり方ではソフトウェアの作成費用がかさむだけである．ゆえに，作成したプログラムの使用環境の制限をできるだけゆるめれば，そのプログラムを他に流用できたりするから，それによって費用の節減を計ることができる．しかもソフトウェアに投資する費用の大部分は，メーカ間での競争によって出現する新機種などへの既存のプログラムの焼き直し，バージョン・アップに費やされているといわれている．プログラムにトランスファラビリティを持たせる方法が非常に有効な方法であるゆえんである．

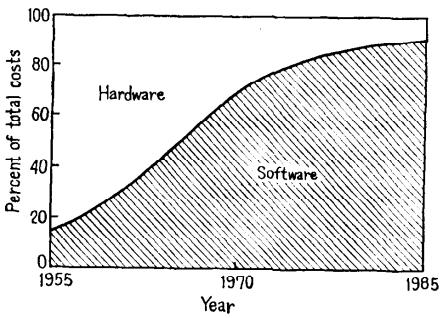

図-1　ハードウェア・ソフトウェア価格比[1]

ここでは，プログラムにトランスファラビリティを持たせるための方法について解説する．

2. 標準化による方法

これには以下に述べるように高級言語の標準化による方向とアセンブラ言語の標準化による方向の2つがある．

2.1 高級言語の標準化

プログラムにトランスファラビリティを持たせようとする場合，すぐに考えつくことはどのようなマシンでも解釈・実行できるような言語でプログラムを書けばよいということである．そこで COBOL とか FORTRAN とか ALGOL といった汎用の高級言語の標準化が行なわれている．これによって，たとえば JIS-3000 レベルとか JIS-7000 レベルとかの FORTRAN 言語でプログラムを記述すれば，そのレベル以上の機能をもつコンパイラを備えたコンピュータ・システムに対してはそのプログラムはトランスファラビリティを持つことができることになる．

しかし，実際にはこれらの汎用の高級言語で記述されているプログラムは決して多いとはいえない．たとえば，システム・プログラムの場合にはそのほとんどがアセンブラ言語レベルで記述されており，高級言語

---

* Program Transferability by Yutaka MATSUSHITA, Haruaki YAMAZAKI and Eiji TANGE (OKI Electric Industry Co., Ltd., Software)

** 沖電気工業(株)ソフトウェア事業部

を用いている例はきわめてわずかの例しかないといえる．また比較的使用頻度が高いと考えられるユーザ・プログラムにおいても，特定業務用に開発された専用言語で記述されたものが少なからず存在し，そういう専用言語に対して標準化をするという試みはなされないであろう．

上述したように標準化によってトランスファラビリティを持たせうるプログラムの範囲が限られている上に，標準化されているはずの汎用高級言語においてもそのコンパイラのメーカの方針等の違いによって，いわゆる方言などが発生している．すなわち，標準化によってトランスファラビリティをもたせようとする方法は十分とはいえない．

2.2 アセンブラ言語の標準化

アセンブラ言語の標準化とは，マシンに従属しているアセンブラ言語をマシンから独立したものとすることである．システム・プログラムの大部分がアセンブラ言語で記述されているという現状では，システム・プログラムにトランスファラビリティを持たせようとするためには，このアセンブラ言語を標準化することが第一に考えられることである．

しかし，アセンブラ言語を標準化しようという機運はほとんどない．ただアセンブラ言語をマシン独立なものとする試みが2,3行われているにすぎない．その開発例として，慶応大学の CAL(Common Assembly Language)[2] とか，Arizona State University の MACS (Meta-Assembler For Computing Systems)[3] などがある．

CAL は，現在の計算機の代表的な命令から構成され，そのうえコンパイラを記述できる程度の機能をもつ言語である．この CAL で記述されたプログラムを実在のアセンブラ言語に変換するという構成をとっている．

それに対して MACS では，MACS 言語で記述されたプログラムを MACS アセンブラによって直接に実在の計算機のマシン言語におとすという構成をとっている．

この MACS アセンブラは FACS (A Fortran Assembler For Computing Systems) アセンブラによってアセンブルされる．すなわち，このシステムでは，MACS アセンブラを実在のマシン上で走らせるために，MACS アセンブラを FACS アセンブラによって各種のマシンの機械語プログラムに変換するという方法によってトランスファラビリティを持たせている．

FACS アセンブラは，FORTRAN 言語で記述されており，その意味で一種の機械独立アセンブラといえる．

この FACS にはマクロ機能があり，これによって対象マシンのハードウェアにないような命令をいくつかの命令シークェンスにおとすことができる．

FACS アセンブラは次のような3つのフェイズがある．

a．初期フェイズ

このフェイズには次に掲げるような3つの入力がある．

ⅰ）対象マシンを定義するデスクリプタ（命令定義，マクロ命令定義，キャラクタ・セット表現等）

ⅱ）コマンド・ニーモニックとラベル（上記のデスクリプタや他のパラメータはラベルで参照するためその対応を与える．）

ⅲ）5種のオプション指定（ソース・リスティング，オブジェクト・リスティング等）

b．パス1

プログラム（MACS アセンブラ）を入力し，シンタックスのチェックなどを行う．

c．パス2

対象マシン定義を用いて対象マシンのオブジェクト・コードに展開する．

FACS では，命令は命令コード・フィールド，タグ・フィールド群，オペランド・フィールド群の3つに分けることができるという認識に立っている．その場合，命令コードは最左端にあり，オペランドはその右側にあり，タグは通常命令コードとオペランドの中間にある．さらに，1命令が複数語にわたることがある．（複数命令が1語になる場合については，FACS では考慮していない．）この認識に立って機械命令特性表 (Instruction Signature) と総称する3種のデスクリプタを考えている．これを以下に述べる．

a．内容語 (Content Word)

これはオペランド，タグの存在/不在を表示するものである．$T_1,\cdots,T_n$ をタグ表示，$A_1,\cdots,A_m$ をオペランド表示とすれば，内容語の仕様の1例として $A_5A_4A_3T_3A_2A_1T_2T_1$ などとすることができる．

b．分節点語 (Break Point World)

これは1命令が複数語になるとき，その語に切られるところを示すデスクリプタである．たとえば分節点語の仕様の例として次のようなものを考えることができ

る．

$$\begin{matrix} \times\times\times & \times\times\times\times\times \\ A_5A_4A_3 & T_3A_2A_1T_2T_1Q \end{matrix}$$

ここで Q は命令コード・フィールドを表わす．

c．ビット位置指示

これは1命令の各構成要素のビット位置を表わすデスクリプタである．これによって各フィールドの長さを知ることができる．

この機械命令特性表以外にも各種の情報デスクリプタが存在している．たとえばコマンド記号表 (Command Mnemonic Table) で，これにはニーモニック，代入すべき機械命令コード，機械命令特性表へのリンク付けが記載されている．

これらのデスクリプタによってプログラムを求めるマシンの機械語プログラムに変換することができる．

機械独立アセンブラ言語を作成する場合，MACS (FACS) におけるように，対象マシンの機械特性を入力することによって，対象マシンの機械語プログラムに変換するという方法では，あらゆるマシンの機械語の特性を統一的に表現することは不可能であり，結局限られた範囲内でしかトランスファラビリティを持たせることができない．

それゆえ，ソフトウェア・エンジニアリングという広い立場からみれば，高級言語の標準化の方法の方がすぐれているといえる．すなわち，システム的な観点からは高級言語を用いることはシステムのプログラム製造の全過程，メインテナンス段階の改善ということにつながる．このことからシステム・プログラムの開発に高級言語が使われるようになてっきている．その典型的なものとして MULTICS における PL/1 の採用がある．しかし，このことに対する反論として高級言語を使用すれば優秀なプログラマがつくるプログラムよりも効率の悪いものとなるということがよくだされる．しかしコンパイラが行うよりもすぐれたプログラムをつくることができるプログラマはきわめてまれであり，性能のよいコンパイラは，平均的なプログラマの作るプログラムと同程度のプログラムを生成できるといわれている．すなわち，システム・プログラム等にもっと高級言語を用いるべきであり，その高級言語が標準化されたものであるならシステム・プログラムにもある程度のトランスファラビリティを実現できたことになる．しかし，数種のマシンしか持たないようなメーカという狭い立場からすると，アセンブラの標準化という方法は極めて有効である．

上述したように言語の標準化によってある程度のトランスファラビリティを実現することができる．しかし，それでも強くそのマシンに従属している部分が存在する．これに対する方法を次に述べる．

3. プログラミング手法による方法

3.1 プログラムのモジュール化

さて，ソース・プログラムの表現形式に標準化を計ってトランスファラビリティを持たせるとしても，たとえばI/Oハンドラのように強くマシンに従属している部分を如何にするかという問題が残る．このことに対しては，マシンに従属する部分を1つ，もしくは複数個にまとめて，他の部分は全くマシンと独立するように作成するということが考えられる．そうすればマシンに従属する部分のみを書きかえれば他のマシンに対してもプログラムは走ることができ，このプログラム全体は一応のトランスファラビリティを実現できたこととなる．

本節では D. L. Parnas[4] の与えた，プログラムをモジュールに分解する際の基準についての例をあげての示唆についてみてみる．KWIC インデックス生成システム*のモジュール化したものとして以下の2種のものがあげられている．

(1) モジュール化1

データを入力してメモリに格納する入力モジュール，その入力データの行中の語を巡回シフトしたものをつくる巡回シフト・モジュール，巡回シフトしたものをアルファベット順に並べるモジュール，それをリスト・アウトするモジュールというように分解する．

(2) モジュール化2

モジュール化1の各モジュールにおけるメモリ・データの操作を共通したものを新たにモジュールとして分離したものである．

これら2種のモジュール化を比べてみよう．大低のプログラマの行うものはモジュール化1の方である．しかし以下のように項目をあげて比較してみるとモジュール化2の方が優っているといえる．

a．変更しやすいか

この点については明らかにモジュール化2の方が優

* KWIC インデックス (Key Wordin Context Index) はたとえば書物の題名の索引の方法で，キーワードとなる語がどの部分にあっても一定の位置にくるようにしたものである．このシステムの入力は行である．その行は複数個の語からなっている．各行の最初の語を行の最後尾におくという巡回シフト操作を繰り返し，それによってつくられたものをアルファベット順に並べてリスト・アウトする．

れている．たとえば，モジュール化1では入力したデータの格納形態が全モジュールで用いられているのに対して，モジュール化2では行メモリ・モジュールに吸収されてしまっており，入力フォーマットの変更，その格納形式の変更に対してはモジュール化2の方が対応しやすくなっている．

b．独立して開発しやすいか

モジュール化1ではモジュール間インターフェイスやテーブル構成がわりあいこみいっており，しかもそのフォーマットが各モジュールの効率を左右するので，モジュール作成中にインターフェイスの確立をするという手順になってしまう．しかしモジュール化2では，モジュール間インターフェイスはある程度抽象的になっている．すなわち，関数名と種々なタイプをもついくつつかのパラメータというインターフェイスとなっており，モジュール作成を独立して進めていくことがわりあい早めに行うことができる．

c．理解しやすいか

モジュール化1においては，たとえば出力モジュールを理解しようとすれば，前のアルファベット順モジュール，巡回シフト・モジュール，入力モジュールをも理解しなければならない．しかしモジュール化2ではそういう必要はないと Parnas は判断した．

すなわち，機能別のモジュール分解という方法では各モジュール間インターフェイスが複雑になってしまう．故にインターフェイスを簡略化する方法の方が良く，すなわち処理の流れを基準にした方が良いといえる．このような良いモジュール分解の仕方がトランスファラビリティの実現に有力な手段を与えてくれることとなる．

3.2 プログラム・ストリング構造[5]

プログラム・モジュールが完全に分離したエンティティとして取り扱えるならば，大きなプログラムをつくるために任意に結合できるようなプログラム・ビルディング・ブロックとして用いることができる．このようにプログラム・ストリング構造は実際に結合しなくともプログラム群を集合的に走らせることができるという基本原則に則っている．

さて，プログラム・モジュールを完全に分離したものとして取り扱うために，プログラムの出力から他のプログラムの入力へのデータの流れの中にバッファ・ファイルを導入する．すなわち，プログラムは入力データ・ストリームをバッファ・ファイルからとりだし，出力データ・ストリームはバッファ・ファイルにおくように作成する．このとき，各プログラムは，入力データのフォーマットや構造，出力データのフォーマットや構造を明確に定義しておきさえすればシステムに準備されている自動的リフォーマッティング・プログラムによってインターフェイスを整合できるので，他のプログラムに関知しなくともプログラミングできることとなる（**図-2** 参照）．

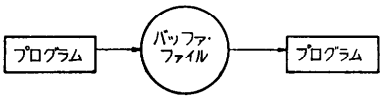

図-2 バッファ・ファイル

このバッファ・ファイルは一般の入出力ファイルと類似しており，そのアクセスは単一方向のみでモニタによりプロテクトがかけられている．バッファ・ファイルの生成は実行時に行われる．バッファ・ファイルに対するアクセス・コマンドとして OPEN コマンド，READ/WRITE コマンド，COMPLETE コマンドを用いることができる．OPEN コマンドにより BFW (Buffer File Control Word) を生成して入力と出力との対応づけを行う．READ/WRITE コマンドによりデータのアクセスを行い，COMPLETE コマンドによってアクセス終了の指示を行う．

またプログラム群を集合的に使用することができるということは，非同期的に走ることができるということを意味し，同期をとるための手段としてもバッファ・ファイルを利用することができる．

このようにしてバッファ・ファイルの導入は，オーバヘッドの増大をまねくが，それでもプログラムのモジュール化を推進する有力な手段である．

3.3 ストラクチャード・プログラミング

ストラクチャード・プログラミングについては他の解説で論ぜられているからここでは詳しく述べない．ただし，ストラクチャード・プログラミングの定義が現在あいまいになっていると思われるのでこのことにふれてみたい．Denning によると SP (Structured Programming) について一般にうけとられているイメージの代表的なものは以下のものである[6),7)]．

a．常識へのたちかえりである．

b．go to 文を使用しないプログラミングである．

c．トップ・ダウン・プログラミングである．

d．種々の大きさと複雑度を持つフローチャートを基本的論理構造の繰り返しとネストで表現されるような標準的なものに変換することを扱う理論である．

e．プログラムが容易に理解でき修正できるようにプログラムを作ったりコーディングしたりするしかたである．

このように種々の考え方があるが，結局プログラミングに一貫性をもたせようとすることにほかならない．

プログラムを構造化することによってそのプログラムは変更，修正しやすくなり，そのことによってトランスファラビリティをもたせることができるようになるといえる．

4. 既存のプログラミングにおける方法

さて，もうすでに作成してしまっているプログラムを他のマシンにのせることができるようにするためにはどのようにしたらよいであろうか．この問題に対して最初考えられたことは，UNCOL (Universal Computer Oriented Language)[8] というものを媒体として高級言語プログラムと計算機を結びつけようということであった（図-3参照）．そのためには次にあげる2種のプログラムが必要である．

a．ジェネレータ

これはコンパイラ言語で記述されているプログラムを UNCOL 言語プログラムに翻訳するものである．なお，これは各コンパイラ言語対応に作成されるべきものである．

b．トランスレータ

これは UNCOL で書かれたプログラムを対象マシンの機械語に変換するもので，各マシンごとに作成されるものである．

また，UNCOL 言語で書いた新規作成のプログラムは，上記のトランスレータが存在するマシン群に対してトランスファラビリティが存在することになる．しかし，この UNCOL はどのような高級言語をも吸収できるような汎用高級言語であるから，その仕様は非常に扱いにくいものとなってしまい，結局後述する PILTER システムのようなアプローチがとられるようになった．

また他の方法として，1つのステートメントを別のマシンの命令シークェンスに対応づけて代入するという方法で 90% 程度の自動変換を得た例[9] とか，実行のシミュレーションを行う方法（たとえば IBM 360 上で IBM 650 シミュレータを動かして IBM 704 をシミュレートし，結局 IBM 704 がシミュレートしている IBM 650 Payroll プログラムを IBM 360 上で走らせることができたという例がある）があるが，効率が悪いとかインプリメントが難しいといった問題点がある．

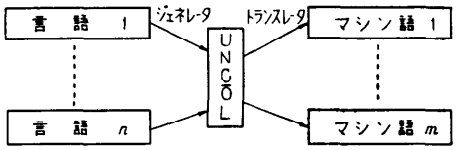

図-3 UNCOL の位置

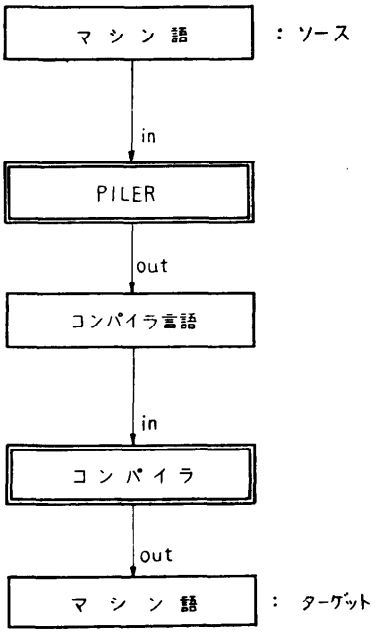

図-4 PILER システムの位置づけ

ここでは，PILER システムを説明しよう[10]．PILER システムでは，コンパイラが，そのハードウェア・ソフトウェア環境を最適な仕方で用いることができるようにプログラムをコンパイルするように設計されているという認識に立っている．すなわち，機械語プログラムから異なるマシンでの高級言語に変換できれば，そのマシンのコンパイラによって最適なプログラムになることが保証されることになる．故にマシン語から高級言語に変換するという逆コンパイラとして PILER を位置づけできる（図-4参照）．

PILER システムは図-5（次頁参照）に示すように3つの部分にわけることができる．すなわち，インタプリタ，アナライザ，コンバータであり，インタプリタはソース・コンピュータのハードウェア・ソフトウェア特性を認識して機械語プログラムを解釈しそれを

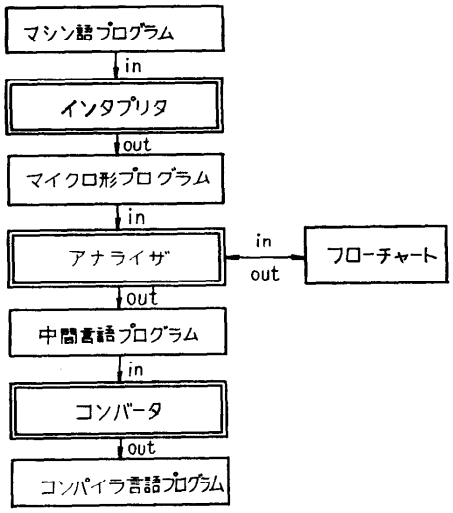

図-5 PILER システムの構成

テーブル化したものを出力する．アナライザはインタプリタの情報をもとにしてプログラムの論理機能を決定し，中間言語プログラムを出力する．この中間言語を用いてコンバータはそれぞれのコンパイラ言語に変換する．

インタプリタはマシン対応に1つずつ存在し，コンバータは各マシンのコンパイラ言語対応に1つずつ必要であり，アナライザは1つあれば十分である．

この PILER における方法が有効であるのは，たとえばあるマシンでの m 個のコンパイラ言語から他のマシンへの n 個のコンパイラ言語への両方向の変換に要するプログラムの数は $2mn$ 個であるのに対し，PILER のように機械語からコンパイラ言語への変換の方法の場合には $2(m+n)$ 個必要であるから作成プログラムの数が少なくてよいからである．

5. 新規設計のプログラムにおける方法

ここでは例を Mobil Programming System[11] にとって説明を加える．まず Mobil Programming System の問題意識はどうであるかというと，たとえば COBOL, FORTRAN, ALGOL といった汎用高級言語は特定のある業務に対して最適であるというわけにはいかない．かといって，特定業務用に開発された言語というものは汎用性がないという理由で普及することがなく，したがって特定のマシンに対してしかそのコンパイラは存在しない．このジレンマを解決するための方法として2つの方法が考えられる．その1つは，どのような業務に対してもぴったりと適合するような汎用言語をつくるという方法である．しかし，これは前述した UNCOL のようなもので総ての業務の和集合に対するものとなってしまい，作成が非常に困難である．ゆえに特定業務に適応した言語のコンパイラにトランスファラビリティを持たせればよいという考えがでてくる．これがもう1つの方法である．この後者の考え方に立って Mobil Programming System が作成されたのである．この Mobil Programming System でとられた抽象マシン・プロセシングのアプローチとは以下のものである．

a．実在のマシンに対して単純な抽象マシンを考え，その抽象マシンが実在のマシン上で実現できるようにする．

b．与えられた特定業務に対してその業務を最適に実行するような言語で記述されたプログラムを a で考えた抽象マシン上で走ることができるようにする．（すなわち，a での抽象マシンのオペレーションを用いたその専用言語のコンパイラを作成する．）

Mobil Programming System はこの抽象マシン・プロセシングのアプローチを基とし，そのインプリメントにマクロ・プロセシングの概念を用いている．

このシステムでは上記の a に相当する抽象マシンとして図-6 に示すような簡単な構成の FLUB マシンを考える．そして，そのオペレーションとしてシフトやブール演算などは含まれないような通常のアセンブラ・ライクのものとしている．

この FLUB マシンをインプリメントする方法は以下のとおりである．すなわち，

a．FLUB のオペレーションを実在マシンのアセンブラコードにおきかえるマクロ・プロセッサ SIMCMP を考える．ゆえにこの SIMCMP は実在のマシンにデペンドするプログラムである．

b．この SIMCMP は実在のマシン上で走ることができるように FORTRAN 言語で記述される．ゆえに SIMCMP は FORTRAN コンパイラを備えてい

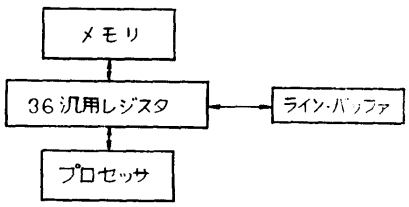

図-6 FLUB マシンの構成

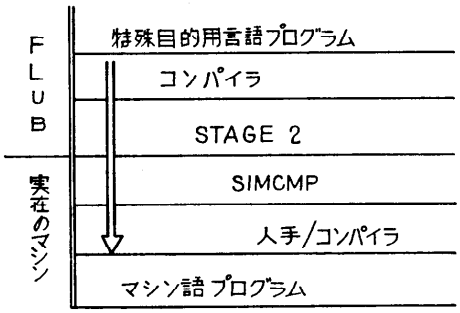

図-7 FLUB マシンと実在のマシン

る実在のマシンにはすぐのせることができ，ゆえにFLUB マシンが実在のマシン上に実現できたことになる．また FORTRAN コンパイラがないような実在のマシンでは，人手で簡単にコード化できるように FLUB マシンのオペレーションを考慮してあるので，やはり，FLUB マシンをその実在のマシン上に実現できることになる．

この FLUB のオペレーションは簡単で数が少ないため，STAGE 2 というマクロ・プロセッサを作っている．この STAGE 2 は FLUB 言語で記述され，ゆえに SIMCMP はこの STAGE 2 をローディングするためのブート・ストラップの位置に立つことになる．そして，結局この STAGE 2 のマクロを用いて上述した専用言語のコンパイラ・プログラムを記述すれば，STAGE 2 によって実在のマシンのアセンブラ・コードにおちることとなり，その専用言語のプログラムを実行できるようになる（**図-7** 参照）．

またこのようにしてコンパイラを作成しておけば，新しいものへの移行の際に必要なものは SIMCMP のみである．

このようにして Mobil Programming System ではプログラムのトランスファラビリティ（モビリティ）を実現しているのである．

4章と5章とで述べたことはまず対象とするプログラムの範ちゅうの差としてみた．すなわち，もうすでにプログラミングされてしまっているプログラムに対しては Mobil Programming System は対処の方法がない．PILER システムのような方法をとるしかない．しかし，いったん FLUB マシンに対してプログラムをつくったとした場合，Mobil Programming System においては実在のマシンに対して SIMCMP を1つ作成すればよい．また PILER システムにおいては FLUB に対するインタプリタが既に存在しているときには，実在のマシンの高級言語コンパイラに対して1つのコンバータを作成すればよく，Mobil Programming System のアプローチの場合と大差はない．結局，4章における PILER のアプローチと5章の Mobil Programming System のアプローチの差は，別の観点から言えば前者は高級言語で記述されたユーザ・プログラムに着目してそれにトランスファラビリティを持たせる方向であるのに対し，後者はコンパイラそのものにトランスファラビリティを持たせようという考え方である．

6. データに対する方法

プログラムのトランスファラビリティをなくす原因の大きなものはデータの取り扱いである．すなわち，データの記憶構造をそのまま扱うようなプログラムではなかなかトランスファラビリティを実現することは難しい．ゆえにプログラムはデータの論理構造を扱うようにすれば，たとえ記憶構造が変化したとしても影響を受けることはなく，データに関してのトランスファラビリティを実現できることとなる．

6.1 汎用データ・ベース・システム

データ・ベース・システムはデータ管理の一般化を計るものであり，データの論理構造と記憶構造とを分離するものである．プログラムの内ではデータの論理構造のみを対象としてデータを操作するコマンド (DML: Data Manupulation Language データ操作言語) を用いればよく，データの論理構造の記述，宣言はデータ記述言語 (DDL: Data Description Language) によって行われ，それをもとにして DBMS (Data-Base Management System) がデータの記憶構造に写像する．すなわち，プログラムはこのようにしてデータからの独立（トランスファラビリティ）を実現するので，論理構造の中を自由に操作してプログラミングしていくこととなる．

この現実的な実現を計ろうとするものが CODASYL の COBOL 機能拡張であり[12]，DDL[13] である．また実用されているデータ・ベース・システムとして DM-1 などがある[14]．

6.2 データ独立プログラミング

データ・ベース・システムにおいてはプログラムは論理データ構造の中をあたかも航海するかの如くつくられる[15]．これをさらにおし進めて，論理データ構造の包含関係を基としてそのデータの属性までをも透明なものとしてプログラムすることが必要である．この

ことはストラクチャード・プログラミングにおいてトップ・ダウンでプログラムを作成していく場合に重要なことであり，とくにシステム記述言語においては本質的なことであろう．このことは一言でいえばプログラミング言語にユニフォーム・リファレント[18]という性質をもたせることであり，アウト・サイド・イン記述のどの段階においてもデータの参照機構を一様化することである．ここでは LIS 言語における例を引用してこのユニフォーム・リファレント (Uniform Referent) を説明する[16]．

タイプ PERSON は図-8 のように定義される．図-8 での定義 (1), (2) の差違は，PERSON の AGE は (1)ではフィールドとして定義されるが，(2)ではタイプ定義の後で与えられるアクションによって計算される．

```
(1)  TYPE PESON=
       PLEX
         AGE: (0..100);
         BIRTH: (0..2000);
       END
(2)  TYPE PERSON=
       PLEX
         AHE: (0..100) ACTION;
         BIRTH: (0..2000);
     PROGRAM PERSON. AGE
     BEGIN
       RETURN DATE-BIRTH;
     END
```

図-8 タイプ PERSON の定義

```
a  FAMJLY: ARRAY (1..60) OF PERSON;
     X: REF FAMILY;
b  FAMILY: DOMAIN OF PERSON;
     X: REF FAMILY;
c  X: PERSON;
```

図-9 ある PERSON, X の定義

またある PERSON, X の定義の仕方には図-9 に示すようなものがある．図-9 の a の定義では，X はタイプ PERSON の配列 FAMILY の 1 要素を示すインデックスである．b では X は明記されていない領域 FAMILY の 1 オブジェクトを示す参照変数であり，c ではタイプが PERSON であるような 1 オブジェクトである．

ここである PFRSON, X の AGE を示すには，LIS では図-8 と図-9 の定義のどの組み合わせでも X. AGE と記述できる．しかるに PASCAL 言語では図-10 に示すように種々の記述が発生する．ゆえに PASCAL 言語はユニフォーム・リファレントの性質を持たないことになる．

```
(1)とa   FAMILY(X). AGE
(1)とb   X↑. AGE
(1)とc   X. AGE
(2)とa   AGE (FAMILY(X))
(2)とb   AGE (X↑)
(2)とc   AGE (X)
```

図-10 ある PERSON, X の参照

ユニフォーム・リファレントの性質をもった言語の他の例として，R. M. Balzer によって提出された PL/1 の拡張形としての Data-less Programming System 言語がある[17]．

また，Douglas T. Ross の AED-0 言語[18]ではユニフォーム・リファレントの性質から生ずる宣言分離 Separable Declaration の機能も備えている．これは，宣言を宣言ファイルに登録しておいてコンパイル時に宣言ファイルから挿入することができることであり，このこととユニフォーム・リファレントの性質によってプログラム本体の書き換えを行わないで種々のバージョンのプログラムを作成することができる．

前述したようにユニフォーム・リファレントによってプログラマはある程度データ構造から開放され，データの包含関係によってプログラムを記述すればよいことになった．この点でデータ・ベースの考えを一歩進めたものとなっているが，入出力のアクセスにはやはりデータ・ベースの力を利用する必要がある．結局ユニフォーム・リファレントの性質を持たせるにはコンパイラは相当の処理が必要となる．

また，ユニフォーム・リファレントではデータの一般的な取り扱いができるが，データの詳細な特質を利用できないためにオーバーヘッドをくうという犠牲を払うことになる．

7. おわりに

以上みてきたように，種々のレベルにおけるトランスファラビリティの実現の試みがなされている．これらの方法を組み合わせて行うことも必要である．また種々のコンピュータが出まわり，いろいろのコンピュータを用いたシステム化が行われてきている現在，このトランスファラビリティを実現することが，膨大化するソフトウェア開発費を節減する強力な手段であり，今まで述べてきた諸例は，トランスファラビリティの実現に対してソフトウェア・エンジニアリング的な試みがなされてきた例となっている．

参考文献

1) Barry W. Boehm: Quantitative Assessment,

Datamation, pp. 49～59 (May 1973)
2) 大駒誠一：共通アセンブリ言語とその処理プログラム，第22回情報科学研究会資料，pp. 1～22 (昭 48, 2)
3) Myron Ammon Calhoun: Machine-Independent Assemblers For Computing Systems, Ph. D. Dissertation, p. 239 (June 1968)
4) D. L. Parnas: On the Criteria to be used In Decomposing Systems Into Modules, p. 25, Carnegie-Mellon University, Pittsburgh (August 1971)
5) F. Morenoff, J. B. McLean: Inter-Program Communication program string structures and buffer files, Prac. SJCC, pp. 175～181 (April 1967)
6) Denning, P. J.: ACM SIGPLAN Notice (Oct. 1973)
7) David Gries: On Structured Programming-A Reply to Smoliar, acm forum in CACM, Vol. 17, No. 11, pp. 655～657 (Nov. 1974)
8) Mock, O., Olsztyn, J., Steel, T., Strong, J., Teitter, A., Wegstein, J.: The Problem of Programming Communications with Changing Machines: A Proposed Solution, CACM, 1, No. 8, pp. 12～18; 1, No. 9, pp. 9～15 (1958)
9) Olsen, T.: Philco/IBM Translation a Ploblem Oriented, Symbolic and Binary Levels, ACM Symp. Reprogramming Problem, Princeton, New Jersey, June 1965
10) Penny Barbe: Techniques for Automatic Program Translation, Software Engineering Vol. 1, pp. 151～165, Academic Press, New York (1970)
11) P. C. Poole, W. M. Waite: Input/Output for a Mobile Programming System, Software Engineering Vol. 1, pp. 167～177, Academic Press, New York (1970)
12) CODASYL Data Base Language Task Group (PLC): The COBOL Data Base Facility, p. 141, Jan., 1973.
13) CODASYL Data Description Language Committee CODASYL Data Descviption Language Journal of Development, 1973.
14) P. J. Dixon, DR. J. Sable: DM-1—A generalized data management system, SJCC, pp. 185～198 (1967)
15) Charles W. Bachman: The Programmer as a Navigator, CACM, Vol. 16, No. 11, pp. 653～658 (Nov. 1973)
16) J. D. Ichbiah, J. P. Rissen. J. C. Heliard: The two level approach to data definition and space management in the LIS System Implementation Language, Proc. Sigplan/Sigops Interface Meeting, Sigplan Notices, Vol. 7 (1973)
17) R. M. Balzer: Dataless Programming, FJCC, pp. 535～544 (1967)
18) Douglas T. Ross: Uniform Referents: An Essential property for a Software Engineering Language, Software Engineering Vol. 1, pp. 91～101, Academic Press, New York (1970)

(昭和50年6月12日受付)