マクロアセンブラ*

西村　恕彦**

1. はじめに

アセンブラというのは定義としてははっきりしていないが，いちおうの概念は世間一般に通用している．たとえば日本では Sip がアセンブラ言語の代表的なものであろう．Sip を作ったときには手軽に使えることがアセンブラの取りえだと考えたそうだ．ところがこのごろはアセンブラでもだんだん手重くなってきたので，このようなものを仮にマクロアセンブラと呼んでまとめて検討してみる．**文献 1, 2, 3, 4** とほぼ同じ態度で取り扱う．

重くなってきた理由はアセンブラ自体の自然の発展のほかに，オペレーティングシステムに組みこむためとか，コンパイラとの関係づけとかいった外部の事情もある．その結果コンパイラのように，昔だとずっと高級でまったく別種だと思われていた言語に近づいて区別がつけにくくなっている．このことは，たとえば 7070 の Cobol-Autocoder や 7090 の Fortran-Map の隔たりを昔の 650 の Fortran-Soap や 704 の Fortran-Sap の隔たりと比べてみればわかる．

現在の整った計算組織のなかで，アセンブラが用いられるのにはおそらく二通りの用途がある．一つはコンパイラの翻訳結果をアセンブラで受けとめるものである．いろいろのコンパイラが共通の下位言語におちるようにしておけば，システム作りの手間がある程度軽減されるはずである[19]．この目的のアセンブラ言語はなるべく単純な外部形式[1]が望まれる．つまり翻訳においては，文字処理や文脈解析といった分析的な段階でやたらに時間を食うものである．だから最初の源言語だけはやむをえないが，アセンブラの段階ではこの種の手間はかけないで済ませたい．たとえば入れ子構造（ブロック構造・式・修飾）のように読むはじからの翻訳のできないものも除外したいわけだ．もちろんアセンブラで十分うまくできることまでコンパイラにやってもらう必要はない．

もう一つの用途は人間の書く源言語としてである．長いジョブの全体をアセンブラで書き下すことはないだろうが，コンパイラ言語だけでは書ききれない部分をサブプログラムとして書くことはあるだろう．そのためには書きやすさのほかに，機械の命令を駆使できること[3,4]，relocatability，サブプログラム間のつなぎ，コンパイラとの共通性などをアセンブラがもっていないといけない．逆にコンパイラのほうにも，アセンブラで書かれたサブプログラムを受け入れる能力がないといけない．

2. マクロ命令

プログラママクロというのは形式上のパラメータを使ってプログラマが命令群の骨組を定義し，それに名前（マクロ命令コード）をつけておく．のちにプログラムの本体の部分で実際のパラメータを伴ってそのマクロ命令コードが利用されると，翻訳ルーチンがさきの定義を参照し，骨組のなかの形式上のパラメータを実際のものにおきかえた命令群を作りだす．つねに同じ骨組で出てくるものを，単純なおきかえによるマクロという．これに反し場合によって異なった命令群の得られるものを条件付翻訳という[15,16]．

条件付翻訳は実際のパラメータの性質を解析しながら進められる．この性質には二つの面がある．第1は参照されかたであって，実際のパラメータがそこに書かれている形式に関するものである．第2は宣言されかたであって，それがどこでどのように定義されたかである．前者を参照情報，後者を宣言情報と呼ぼう．

ここでは IBM 7070 Autocoder の Macro Generator にもとづいて述べる[6~9]．それは内容がよくわかっていること，比較的わずかな機能でやれそうなこと，Cobol 流の条件付翻訳（たとえば Move）に適していること，普通のプログラミング言語の体系とよく調和しそうなことなどによる．ただし，ゼネレータには Algol もどきの記法を用い，オブジェクトには Sip もどきの記法を用いる．

2.1 単純なおきかえ

次のように翻訳されるマクロ ADDI を作れ．

```
ソース……ADDI  X, Y, Z
オブジェクト……XA    X
               A     Y
               T     Z
```

* Macro Assemblers, by Hirohiko Nisimura (Electrotechnical Laboratory, Tokyo)

** 電気試験所電子計算機部

```
generator ADDI; begin
  generate (XA text[1]; A text[2];
    T text [3];) end
```

または

```
generator ADDI; begin generate (M); exit;
  M: begin XA text[1]; A text[2];
  T text[3]; end M end
```

宣言 generator は以下の手続が翻訳ルーチンによって実行されるものであることを示す．この手続の実行上の終点は end または命令 exit によって示す．命令 generate は翻訳結果の作製を示す．そのパラメータは一連の手続またはそれを名前で呼んだものである．

text[i] は参照情報であって，i 番目に書かれている実際のパラメータを示す．

2.2 整数と実数の選択

実際のパラメータの型および個数によっていろいろに翻訳しわけるマクロ ADDIR を作れ．

```
ソース（整数，3個）…ADDIR  I, J, K
オブジェクト……XA      I
                A      J
                T      K
ソース（整数，2個）…ADDIR  L, M
オブジェクト……XA      L
                AT     M   add to store
ソース（整数，3個）…ADDIR  N, P, N
オブジェクト……XA      P
                AT     N
ソース（実数，3個）…ADDIR  A, B, C
オブジェクト……XA      A
                AF     B
                T      C
ソース（実数，2個）…ADDIR  D, E
オブジェクト……XA      D
                AF     E
                T      E
```

```
generator ADDIR; begin
  if real[1]∧real[2]          then go to R;
  if integer[1]∧integer[2]    then go to I;
  ER: generate (N1 thru N3);          exit;
  I: if empty[3]              then go to I2;
    if ¬integer[3]            then go to ER;
    if text[2]=text[3]        then go to I2;
    if text[1]=text[3]        then begin
      text[1]:=text[2];
      text[2]:=text[3];       go to I2   end
  I3: generate (A thru C);            exit;
  I2: generate (A, C2);               exit;
  R: if empty[3]              then begin
       text[3]:=text[2];      go to R3   end
    if ¬real[3]               then go to ER;
  R3: generate (A, BR, C);            exit;
MODELS:
A:  XA text[1];     A text[2]; C: T text[3];
C2: AT text[2];     BR: AF text[2];
N1: NE text[1];     NE text[2];
N3: NE text[3];
end generator ADDIR
```

empty[i] は参照情報で，実際のパラメータが存在していないことを示す．integer[i] と real[i] は宣言情報で，実際のパラメータが宣言された型を示している．

2.3 パラメータ情報

上の簡単な2例でもわかるように，記号処理の時期が済んだあとのマクロゼネレータの動作は，普通のプログラミング言語で容易に記述できる．ただし，そのためには若干のパラメータ情報が参照しやすい形できちんと用意されている必要がある．この情報は発生源によって参照情報と宣言情報に分けられ，また型としては条件値，整数値，文字値がある．各情報は固有の名前がつけられパラメータとは添字で対応する．

これらの情報はゼネレータプログラムで参照され計算や検査に使われる．またモデルの骨組に使われオブジェクトとして出力される．

参照情報
- 条件値　パラメータの省略，記号とリテラルの別，リテラルの型，添字の有無
- 整数値　パラメータの記号の長さ，リテラルの値，パラメータの個数
- 文字値　パラメータそのもの，分離符

宣言情報
- 条件値　無定義記号，型（整数・実数・文字・条件・ラベル），表現
- 整数値　割付番地，金物の語中の位置，長さ，小数位けた数，添字限界
- 文字値　属するレコードの名前

Autocoder ではこれらの情報がパラメーター一つあ

たり 12 語に集約して収められている．ただし割付番地はわからない．それは Autocoder ではマクロ作製時期がすべて終ってから番地割付に移るためである．このことはまた翻訳時にできるだけ仕事を済ませ，実行時に冗長な要素を残さないという Autocoder の思想にも関係がある（Map のやりかたはこれとは違っている）．

命令 generate については作製の場所および頻度の指定がある．普通の開いたサブルーチンは参照された場所に毎回作られるが，閉じたサブルーチンならば別の場所に1回だけ作ればよい．

モデル中に書かれる要素にもいろいろある．実際のパラメータでおきかえるもの，固定のもの（命令コードなど），ゼネレータが参照するだけで出力しないもの（モデルの参照ラベル），演算結果でおきかえるもの（位取のシフト数など），ゼネレータが1回ごとに違う値を作るもの（マクロ内部での局所的な飛越の行先など）がある．

マクロゼネレータの最大の特徴は利用できるパラメータ情報の量が多いことである．このために一つのマクロ命令でカバーできる動作の範囲がきわめて広くなり，また実際のパラメータの記入の誤りを検査し柔軟に対処できる．その反面パラメータ情報の解析・検査の段階で著しいコーディングの手間と翻訳時間とを要する．それはつねに最適なオブジェクトを出力しようとすればなおさらである．

余談ながら，最適さあるいは冗長さに関して翻訳ルーチンの能力が人間のそれと比較されることがある．ところが，たとえば Cobol の Move 命令を開いたルーチンとして書き下すような場合には，マクロゼネレータの出力がたいていのプログラマのそれよりも良くなるというような現象は十分期待できるのである．

2.4 MAP

Map（IBM 7090）のマクロについては割合によく知られているし，また広く採用されている[11~16]．パラメータの単純なおきかえが中心である．参照情報としてパラメータそのものの文字値，宣言情報として割付番地がある．これらによる条件式や番地式によってかなりの条件付翻訳ができるが，本質的にパラメータ情報が少ないことからくる制約は大きい．

等価番地を翻訳時に動的に評価する．命令群を一定回数またはパラメータの個数回作る，省略されたパラメータに対して記号番地を作る，定義を入れ子にするなどの仕様も含まれている．

Map の機能の多くはさきのようなゼネレータで記述できる．たとえば次のゼネレータによって作られるマクロ SUM を考えてみよ[3,15,16]．

```
generator SUM; begin
  if text[1]≠'ACC' then generate
                              (XA text[1];);
  i:=3;
  if separator[2]='(' then
       for i:=2, i+1 while separator[i]≠')'
       do generate (A text[i];);
  if text[i]≠'ACC' then generate
                              (T text[i];);
  end
```

Utmost（Univac Ⅲ）のマクロも Map のそれとほとんど同じである．命令群の作製回数を指定する番地式のなかに条件式を含むことができ，しかもその値が整数の 1，0 として評価される点がおもしろい．

これらのマクロの方式が前述のマクロゼネレータに比して劣っている点を考えよう．まず第1は参照できるパラメータ情報の量が非常に少ないことである．もっともこのことは逆に翻訳ルーチンを手軽なものにするという利点がある（さきに宣言情報として割付番地をあげておいたが，実は非常に強い制限がある．普通の意味の割付番地を全面的に条件付翻訳に用いることは実際上は不可能であろう）．

第2は翻訳用の特別な命令が必要なことである．翻訳時に実行される動作は計算・判断・飛越・繰り返しなど通常のプログラミング言語で記述できるものが多い．にもかかわらず Map のやりかたではいちいち別個の命令を作らねばならない．けっきょくその言語のすべての要素を用いてゼネレータを書き，ただそれが実行される時期だけが通常の場合と違うという Autocoder のやりかたが正当なのではなかろうか．このことは Map の番地式処理に対する**文献4（§11）**の批評と同じ趣旨のものである．

3. 被演算数

3.1 計算式と添字式

アセンブラの被演算数欄に書かれた式は計算の仕方によっていろいろ区別される．そのうち特に重要なものは計算式・添字式・番地式である．それらは次の点で相違している．

（1） 計算時期：翻訳時（またはロード時）か，実行時か．

（2） 記号番地：内容の値をとるか，割付番地の値をとるか．

（3） 計算結果：データ（整数・実数・条件）か，番地（整数）か．

コンパイラや Autocoder では式のなかにデータの名前が書いてあると，その「内容」を参照して計算を「実行時」に行なう．四則やカッコの組み合わさった式をきちんと解析するにはかなりめんどうな処理を要するから，コンパイラとアセンブラの区別をこの点でつけても無理はない．

計算結果が「データ」としてある変数の値を定義するのに用いられるのが計算式である．それとは違って実行時に可変な「番地」またはその変更数として用いられるのが添字式である．

通常の演算の回路と番地演算の回路とが別個にある計算機では，ある制限内の添字式は巧妙なやりかたで扱われる（**文献 4 §9**）．その最も素朴なものが指標レジスタであって，たとえば Autocoder では MOVE A[I+3] TO B というふうに書かれる．

3.2 番地式

機械語の命令においてはデータや行先は所在番地によって指定される．この番地を一般の式の形で表現できるのが Map 系のアセンブラである．式の要素は記号番地・その命令自身の番地・絶対番地・リテラルなどである．また指標レジスタや間接番地を指定することもある．番地式の目的は記号の数を減らす，基本になる名前に関連した前後の番地を参照する，プログラマにとって不確定な値を示す，代入を行なうなどであろう．

Map の番地式は「翻訳時」に記号番地の「割付番地」の値をとって計算する．これをさきの添字式の場合と比較してみよう．まず一般に原式は解析されて同等な機械語の命令群におきかえられる[17]．この命令群が実行時まで残されるのが添字式であり，翻訳時（またはロード時）にただちに実行されてしまい機械語プログラムとして表立ってあらわれないのが番地式である．これが第1の違いである．第2の違いは記号番地によって参照される値である．Autocoder のように記号番地の名前を書いておくと，その内容が参照されるという方式と，Map のように記号番地が式のなかでは割付番地の値をとるという方式とがある．番地の概念を利用した計算機[18,19]においてどちらの方式がよりすっきりしているかは簡単には答えられない．下のような書き方はきわめてまぎらわしい．

CLA A+3　　Map：(A+3) 番地の内容をアキムレータに入れる．

ARITH A+3　　Autocoder：A 番地の内容に3を足してアキムレータに入れる．

Map の例を CLA A[3] と書けば，このまぎらわしさはなくなる．ただし配列や関数の名前はアセンブラでは記号番地におきかえて参照されるから，本来の添字や引数を番地変更と区別する必要がある．それにもっと一般的な式の書き方も問題である．筆者としては割付番地を知るための関数（Fortran の XLOCF）と絶対番地のための配列[19,21,22]とを設けたい気がする．明らかに A≡memory [location(A)]．しかしそれでは以下の書き方がすべて同等なものであるということは十分明白であろうか．そして番地定数（addcon）の書き方はどうすればよいだろうか．

```
CLA  A+3
CLA  A[3]
CLA  location (A)+3
CLA  memory [location(A)+3]
```

Map においては被演算数は所在番地の単一の値によって指定できた．しかし，けた指定のできるシステム（Cobol や Autocoder）においては被演算数はこのような単一の番地によっては参照できない．すなわち少なくとも始めの番地と終りの番地（またはデータの長さ）という二つの値ではじめて定義できる．ことによるとそのほかに型や表現の情報をも内包している．このとき Map 流の番地式を書くことに意味があるだろうか．一つの答は記号番地の解釈の仕方が違うというものである．すなわち，アセンブラは番地を扱い，コンパイラは変数を扱うのだというのである．

3.3 割り付け

アセンブラにおいてはコーディングシート上に書かれた命令がその順序に割り付けられてゆく．プロセッサが番地割付をすることならびにそれを制御する手段をプログラマがもっていることはコンパイラよりはアセンブラ（またはローダ）にむいた仕様である．名前の欄に書かれた記号番地は割付カウンタのそのときの値を実際番地として割り付けられる．一方，所在がはっきりと指定されなくてもよい要素（リテラルやサブルーチン）は空いた番地に適当に割り付けられる．

プログラムをいくつかのサブプログラムの集まりとして書くときには，割付カウンタをある値にセットすること，複数個のカウンタを並用すること，カウンタや番地の値の最大最小を知ることなどが必要である．

サブプログラム内でまったく局所的な名前についてはなんら問題がない．そうでないデータやラベルの取扱い，特に別々に書かれあるいは翻訳されるサブプログラム間のつなぎ（linkage）にはさまざまのやりかたがある．それはおそらく決定的にうまい方式が見つかっていないためであろう．たとえば Fortran の Common や Cobol のデータは，すべてのサブプログラムが共同で一定の場所を参照するという，もっとも単純な解決をとっている．

Map のそれは比較的うまくできている．つまり外部から参照される名前と外部で定義される名前とはそれぞれ宣言され辞書に入れられる．そしてローダがこの辞書を参照しながら割り付けをする．Map には別の面に欠点がある．というのはこの辞書や番地式がすべて通訳ルーチンとして処理されるため[17]，オブジェクトデック（relocatable binary deck）の量は小さいが，ローダにうんと時間を食うのである．

3.4 データの構造

計算機内部におけるデータ配置の指定には，語を単位とするものと，けたを単位とするものとがある．データは1要素ずつばらばらに散在している場合と，いろいろの構造をもっている場合とがある．現在の記憶装置は番地の性質からは一次元の配列とみなせるし[21]，等速呼出の性質からは相互に独立な要素からなるとみなせる．構造には配列・タナ（FILO）やトンネル（FIFO）・リスト・フィールド・ファイルなどがある．構造をもったデータは語単位ならば翻訳・実行に容易であるが，さもないと非常なめんどうが起こる．また，その名前によって構造の全体を参照できる場合と特定の1要素しか参照できない場合とがある．データにつけられた名前の実際番地へのおきかえには，一連の番地の最初・最後・その他いろいろのやりかたがある．Autocoder のデータ名はレコードの名前とレコード中の相対番地とに分解しておきかえられる．レコードの名前はベースアドレスとして指標レジスタにはいり（implicit indexing），相対番地は機械語命令の番地部にはいる．

3.5 データの型

データの見掛け上の取扱い(型)にはいろいろある．また金物では内部表現と外部表現とが区別される．データの型・表現・長さによって所要の機械語命令が異なるのが普通である．この機械語命令の差異を源言語に反映させ，実行命令で局所的に書きわけるシステム（Fortran II，SOS の Intran/Outran）もある．他方データの宣言情報をプロセッサが参照して翻訳しわけるシステム（Algol, Cobol, Autocoder）もある．もちろん後者のほうが使いやすい．

実数を表現するのに浮動小数点によるのと Cobol 流の自動小数点によるのとがある．後者のほうが計数型計算機の性質に調和しているように考えられる．しかし一般の計算式とくに割り算を含む場合の小数位の評価を翻訳時に行なうのはたいへんかもしれない．これを完全に実行時にまかせれば，けっきょく浮動小数点のサブルーチンとほとんど同じになるだろう．

参考文献

* 一般

1) 西村恕彦：アセンブラ，第5回プログラミングシンポジウム報告集，1964-1，S 2-72～90
2) 西村恕彦：Assembler-Compiler I，電子協，1963-4，pp. 78
3) 吉村鉄太郎：Assembler-Compiler II，電子協 1963-4，pp. 12
4) 和田英一：Assembler-Compiler III，電子協，1963-5，pp. 56
5) E.J. Corbató et al.: Advanced Computer Programming -A Case Study of a Classroom Assembly Program, The MIT Press, 1963, pp. 170

* マクロ

6) IBM Systems Reference Library: IBM 7070 Series Programming Systems: Writing Macro Generators, C 28-6363, 1963, pp. 101
7) IBM 7070 Bulletin: Writing Macro Generators, J 28-6053, 1959, pp. 37
8) Operator's Manual: IBM 7070 Series Programming Systems: Compiler Systems, C 28-6249, 1962, pp. 67
9) 2) の第4部マクロゼネレータ pp. 56～78
10) 4) の § 13～17, pp. 25～54
11) IBM Systems Reference Library: IBM 7090/7094 Programming Systems: MAP Language, C 28-6311-1, 1963, pp. 54
12) IBM Systems Reference Library: IBM 7040/7044 Operating System: Macro Assembly Program Language, C 28-6335, 1963, pp. 46
13) Univac III Technical Bulletin: UTMOST Programmer's Guide, 1963-2
14) Control Data 3600 Computer: COMPASS/Reference Manual, Control Data Corp., 1962-11, pp. 52
15) M.D. McIlroy: Macro Instruction Extensions of Compiler Languages, CACM 3-4, 1960-4, pp. 214～220
16) P.M. Sherman 著 関根智明訳：電子計算機

プログラミングとコーディング，竹内書店，1964-9，pp. 554，第 17 章マクロ命令
* 番地
17) 4) の §7～12，pp. 10～25
18) A.A. Grau: A Translator-Oriented Symbolic Programming Language, JACM **9**-4, 1962-10, pp. 480～487
19) T.B. Steel, Jr.: A First Version of UNCOL, Proc. WJCC. **19**, 1961-5, pp. 371～378，情報処理 **2**-p. 286
20) Operating Compatibilily of Sestems Conventions, CACM **4**-6, 1961-6, pp. 266-267，情報処理 **4**-p. 51
21) K.E. Iverson: A Programming Language, John Wiley, 1962, pp. 286
22) 淵一博: ETL Mk 6 のアセンブラ言語，情報処理学会，講演予稿集，1963，pp. 27～28

（昭和 39 年 9 月 16 日受付）