共通アセンブリ言語とその変換プログラム†

大 駒 誠 一††

Abstract

If we could have a common assembly language, it would be quite profitable. For instance, if a compiler is written once in the common language, it can be applicable to any other computers.

Then, the author proposes a new simple language called "Common Assembly Language (CAL)." As CAL is so simple, if an assembler has an ability to define so-called macro instructions, all of CAL instructions can be easily defined as macros. Otherwise CAL must be converted by its conversion perogram called "Language Conversion Program (LCP)" which is able to convert not only CAL to any assembly language but also any context free language to others.

The LCP converts CAL instructions according to "Specification Language (SL)" which specifies how to translate each CAL instructions.

The LCP has successfully converted LCP itself which was carefully written in COBOL. These systems are expected to be used by other applications.

1. はじめに

アセンブリ言語というのは本来機械に密着した言語なので，異なる機種の間では互換性がないのが当然であるが，どの計算機にも共通なアセンブリ言語というのがもしできるとすればいろいろ便利なことが多い．たとえば，FORTRAN, COBOL あるいはその他のシステムプログラムを一度この共通のアセンブリ言語で書いておき，新しい計算機ができたとき，まずこの共通言語を機械語に翻訳するアセンブラを作ってしまえば，新しく FORTRAN や COBOL のコンパイラを作る心要がないわけである．

しかしながら，アセンブリ言語はだいたい，機械語を人間の見やすい記号で書くようにしたものだから，もしアセンブリ言語の JIS 規格などができてしまうとどの計算機の命令構成も似たようなものになってしまう可能性があり面白くない．プログラムの互換性は，FORTRAN とか COBOL といった高いレベルの言語で保証することにして，アセンブリ言語のように低いレベルの言語の標準は進歩を阻害することにでなりかねないので，無いほうがいいと思う．

しかしそれでもなお，どの計算機にも共通で，しかもごく簡単に機械語やアセンブリ言語に変換できる低いレベルの言語は魅力がある．たとえば，いろいろな種類の計算機を対象とするコンパイラコンパイラ[1),2)]のような場合，共通ということでかなりの部分を計算機に独立に考えることができるからである．

そこで，機械語を統一するなどということではなく，いまあるたいていの計算機に共通で，コンパイラがどうにか書ける程度の機能だけをもち，しかも簡単に実在のアセンブリ言語に変換できるような言語を考えてみた．そして，これを共通だがレベルの低い言語という意味で**共通アセンブリ言語 CAL** (Common Assembly Language) と呼ぶことにし，同時に，この CAL で書いたプログラムを実際のアセンブリ言語に変換するプログラム **LCP** (Language Conversion Program) を作成した．この LCP は，CAL の各命令をそれぞれどんな実際の命令に変換するかを記述した**変換指定言語 SL** (Specification Languge) の指定にしたがって変換を行なう．

2. 共通アセンブリ言語 (CAL)

コンパイラを書くための十分な機能を考えたのでは

† A common assembly language and its conversion program, by Seiichi OKOMA (Faculty of Engineering, Keio University)

†† 慶応義塾大学工学部管理工学科

```
Numeric data
    MOVE    np1 ,np2
    ADD     np1 ,np2 ,np3
    SUB     np1 ,np2 ,np3
    MULT    np1 ,np2 ,np3
    DIVIDE  np1 ,np2 ,np3[,np4]
    COMP    np1 ,np2 ,l1 ,l2 ,l3
Character data
    MOVEC   cp1 ,cp2 ,n
    COMPC   cp1 ,cp2 ,n, l1 ,l2 ,l3
    COMPN   cp1 ,l1 ,l2
    COMPA   cp1 ,l1 ,l2
Miscellaneous
    JUMP    l
    INTERR  l1 ,l2
    EXIT
    STOP
Input and output
    READ    u,cp,n,l
    WRITE   u,cp,n
Area and constant definition
    DNA     n     (Define Numeric Area)
    DNC     n     (Define Numeric Constant)
    DCA     n     (Define Character Area)
    DCC     'any string' (Define Character Constant)

  Where n is an integer,
        np is a numeric operand,
        cp is a character operand,
        l is a label and
        u is an input output unit number.
```

Fig. 1 List of CAL instructions

きりがないので，必要最小限の機能を考える．コンパイラというのは原始プログラムの文字列をいろいろな条件にしたがってアセンブリ言語（直接機械語に落さないとすれば）の文字列に置き換えるだけであるから，コンパイラを作るのに，いわゆる数値計算やビット操作の命令がなくても，最低限文字列の移動や比較の命令と簡単な補助の命令があれば済みそうである．

このように考えて作成したのが共通アセンブリ言語（CAL）で，その全部の命令を Fig. 1 に示す．互換性を保つために，文字データと数値データとは別々に定義し，それぞれ別の命令で参照する．また各命令はマクロ命令として簡単に定義できるように，オペランドの変数がどう定義してあるかには無関係に一意にその機能がきまるようになっている．

2.1 データの定義

文字データ これは

ALPHA DCA 100

のように定義し，ALPHA は 100 文字分を占める領域であることを示す．その一文字一文字は ALPHA(3)，ALPHA(N) のように1次元の添字をつけてその何番目の文字というように参照する．添字をつけないときは ALPHA(1) と同じで，その先頭の一文字を指す．また文字定数（リテラル）は

MESSAGE DCC 'ILLEGAL STATEMENT'

のように定義し，17 文字の文字定数が MESSAGE 以後に並んでいることを示す．参照のしかたは前と同じで，MESSAGE(5) はこの5番目の文字 'G' を指す．

数値データ

数値は対象となる計算機でもっともあつかいやすい長さを単位としてとりあつかう．いわゆるワードマシンならば一語に一つの数値を入れることになるだろう．そして CAL でもこの単位を語と呼ぶことにする．とりあつかう数値は整数のみである．主な用途が添字用なので 10^5 の値を入れることができれば十分である．負の値が入ることはほとんどない．

BETA DNA 50

は BETA というところから 50 語数値の入る領域が並んでいることを示す．何番目の語を参照するかはやはり添字を使う．

GAMMA DNC 25

は数値定数の定義で，GAMMA という 1 語に数値 25 が入ることを示す．

2.2 数値データ用命令

四則演算命令は原則として3アドレス形成をしており，数値データ用としては他に転記と比較の命令がある．

四則演算 加算命令は

ADD DATA1(I), DATA2(5), DATA3(K)

のように書く．これは

DATA3(K)＝DATA1(I)＋DATA2(5)

のことである．減算，乗算も同様である．除算は4アドレスにして剰余も求められるようにしてある．

DIVIDE A, B, C, D

は

C＝[A÷B]

D＝A−C×B

のことである．4番目のアドレスを省略すれば，剰余は求めない．剰余は2次元以上の表をあつかうのにしばしば便利なのであえて入れた．

転記

MOVE I, K

は1語の転記で

ADD I, 0, K

と同じである．

比較

COMP ALPHA,BETA,SMALL, EQUAL, BIG

は比較命令で ALPHA と BETA を代数的に比較し

て，前者の方が小さければ SMALL へ，等しければ EQUAL へ，大きければ BIG へコントロールが移る．行き先を省略したら次の命令を指す．たとえば，

COMP X,0,,,POSI

は X が正なら POSI へコントロールへ移り，ゼロか負のときはこの命令の次の命令へコントロールが移る．

2.3 文字データ用命令

転記 文字データの転記は

MOVEC HANEDA(4),NARITA,15

のように書く．これは HANEDA の 4 文字目から 15 文字分を NARITA から始まる場所へ移す．

比較

COMPC NIXON,MAO,1,SMALL,EQUAL,BIG

文字データの比較は上記のように書いて，指定した長さの文字列を文字の大小順序 (collating sequence) に従って比較する．上の場合，NIXON と MAO の先頭の一文字を比較して，その大小によりコントロールを移す．

COMPN CHARACTER,NUMERIC,NOTNUM

COMPA CHARACTER,ALPHA,NOTALPHA

の 2 つの命令は CHARACTER の先頭の一文字が，数字 (0～9) であるか，英字 (A～Z) であるかによって，コントロールの行き先を変える．

この一文字が数字であるか英字であるかを調べる命令は，コンパイラが原始プログラムを読み込んで命令を変数，定数，区切り記号などに分解する段階で非常に便利なものである．

2.4 入出力

入出力も簡単で

READ 5,RECORD,80,EOF

WRITE U,LINE,120

のように書く．入出力の機器番号，文字データ変数，文字数の順で指定する．入力の場合はデータが無くなったときの行き先も書く．

2.5 その他の命令

PERFORM という命令は

PERFORM KARA, MADE

のように書いて，COBOL の PERORM 命令と同様に，KARA から MADE までの命令を実行する．MADE は次に述べる EXIT 命令でなければならない．

MADE EXIT

はいわゆる No-Operation であるが，PERFORM 命令で実行される最後の命令は必ずこの EXIT でなければならない．

JUMP と STOP は文字通り飛越しと停止である．

以上いずれの命令も名札をつけることができる．

2.6 補足

この CAL にもう少し浮動小数点演算の命令その他を追加して，コンパイラから出てくる目的プログラム用の言語としても使えば，完全に計算機に独立なコンパイラができるわけである．しかし計算機個有の特徴を生かすことができないので，コンパイラや実行時の効率は悪くなる．これは効率よりも互換性の方に重点に置いているので止むを得ない．

まだいろいろ不十分な点は多いが，ひとまずこれだけにし，あとは必要に応じて変換指定言語 SL で定義することにした．たとえば，いまのところ数値データを入出力する機能がないが，中間結果を磁気テープなどに文字データと数値データをいっしょに書き出せるようにして，なお互換性を失なわないようにするにはどうすればいいか，まだ知恵がうかばないでいる．

またこの CAL の変数名や定数の書き方，あるいは名札や命令コードを書く位置などはとくに定めていない．これらは SL で定義されてはじめて書き方がきまるわけである．

マクロ命令が定義できるアセンブリ言語なら，この CAL の命令一式をマクロ命令として定義し，システムに登録してしまうと次に述べる LCP を使わなくて済むので好都合である．

2.7 プログラム例

Fig. 2 にこの CAL で書いたプログラムの例をあげる．実はこれは初めから CAL で書いたプログラムではなく，COBOL のプログラムを LCP で手続き部だけ CAL に変換したものである．

```
READCARD READ    5,CARD,80,CLOSING
         MOVE    0,J
         MOVE    0,I
         MOVEC   SPACE,PRINT,80
ADDI     ADD     1,I,I
         COMP    I,81,,,PRINTOUT
         COMPC   CARD(I),SPACE,1,,ADDI
         ADD     1,J,J
         MOVEC   CARD(I),PRINT(J),1
         JUMP    ADDI
PRINTOUT WRITE   6,PRINT,80
         JUMP    READCARD
CLOSING  STOP
```

Fig. 2 An example of CAL program which is converted by LCP from COBOL source program.

3. 変換プログラム (LCP)

CAL で書いてあるプログラムを任意のアセンブリ言語に変換するのがこの**変換プログラム LCP** (Language Conversion Program) である．すなわち，LCP は CAL のそれぞれの命令に対して，どんなアセンブリ言語の命令を対応させるかを記述した SL にしたがって，CAL をその対応のアセンブリ言語に変換するプログラムである．

ただこれ以外にも，一般に一つの命令を一つまたは複数個の命令に置き換える作業，たとえば，あるアセンブリ言語で書いてあるプログラムを同じ機能をする別のアセンブリ言語に変換することなどが考えられ，CAL の変換だけなら不要な機能も持たせてある．

変換の対象となる命令は，名札，記号命令，オペランドなどが区切りで区切られて一つのレコード（たとえば一枚のカード）になっているものとし，これを**原命令**と呼ぶことにする．

LCP の機能は，SL を全部読み込んだ後各原命令について，(1)原命令を SL で指定した区切りで切り出して要素に分解する．以後この分解されたものを**要素**と呼ぶことにする．(2)この要素と SL の比較部を探して，この原命令がどの種類の命令であるかをみつける．(3)みつかった比較部以下にある SL の変換部を通訳的に実行して対応のアセンブリ言語の命令群を作り出す．

この LCP のプログラムは互換性を考えて COBOL で書いてあるので，他の計算機でも容易にこのシステムは使えるはずである．

またこの LCP には，はじめからデバッグの機能が入っていて，制御カード（後述）の指定によりデバッグに必要な情報が3つのレベルで印刷できるようになっている．すなわち，原命令が分解された状況，コントロールの流れ，アセンブリ言語が組立てられていく過程などが必要に応じて逐一印刷できる．Fig. 4 はその一例である．この LCP のプログラムは注記を除いて COBOL で約 800 枚であるが，このうち2割強がデバッグ用に費やされている．

4. 変換言語 (SL)

共通アセンブリ言語 CAL のそれぞれの命令を実在のアセンブリ言語のどんな命令群に置き換えるかを LCP に教えるのが**変換指定言語 SL** (Specification Language) である．まず CAL が区切りとしてどんな文字を使っているかを指定し，続いてこの区切りで切り出された各要素を実際のアセンブリ言語として，どんな形でどんな位置に置くかを指定するのである．各種の機能の指定や選択は制御カードで行なう．

```
+DELIMITER
(),.
```

Fig. 3 A specification of delimiters.

4.1 区切りの指定

区切りは Fig. 3 のように制御カード

+DELIMITER

に続くカードで指定する．Fig. 3 では空白，左括弧，右括弧，コンマ，終止符が区切り文字であることを示す．指定できるのは1桁の区切りだけである．

Fig. 4 (a) はこの区切りにより原命令が要素に分解されたところを示す．左端の数字の列は単に要素の先頭から番号をつけたものである．次の列は各要素の長さ（文字の数）で，その次は要素の先頭の文字位置を示す．たとえば2番目の要素 'OKINAWA' は長さが7文字で，それはカードの11桁目から始まっていることを示している．これらの情報は以下の比較部，変換部で，E(s)（要素），N(s)（長さ），P(s)（位置）で参照できる．Sは要素の番号を示す式（後述）．たとえば，E(4) は '15' という2けたの文字列をさし，N(4)，

```
MOVE  OKINAWA (15) TO JAPAN.
1     4     5      MOVE
2     7     11     OKINAWA
3     1     19     (
4     2     20     15
5     1     22     )
6     2     24     TO
7     5     27     JAPAN
8     1     32     .
9     3     33     ...
```

Fig. 4 (a) Delimited elements without blanks.

```
MOVE  OKINAWA (15) TO JAPAN.
1     4     1
2     4     5      MOVE
3     2     9
4     7     11     OKINAWA
5     1     18
6     1     19     (
7     2     20     15
8     1     22     )
9     1     23
10    2     24     TO
11    1     26
12    5     27     JAPAN
13    1     32     .
14    3     33     ...
```

Fig. 4 (b) Delimited elements with blanks.

```
+DELIMITER
 ()+-*/,.
8,72
16,21
```

Fig. 5 Specifications for a range and positions of null delimiters.

P(4) はそれぞれ4番目の要素の位置と長さ，すなわち2と 20 という数値を指す．

Fig. 4 (b) は，原命令は同じであるが，空白も要素として使いたいときの分解例である．これは

+BLANK

という制御カードで指定する．

原命令の前後に不要な部分があるときは Fig. 5 のように指定する．2枚目のカードは，原命令がカードの8桁目から72桁目の間にあることを示している．これがなければ，

1, 80

と指定したのと同じになる．その次の3枚目のカードは**空の区切り**の指定である．これはよくアセンブリ言語で，名札，命令，オペランドなどの区切りに，空白やコンマなど文字による区切り以外に，カードの桁位置で区切ることがあるからである．Fig. 5 の場合は 16 桁目と 17 桁目の間，および 21 桁目と 22 桁目の間に空の区切りがあるものとみなし，文字列が表面上ここでつながっていても，分離して別々の要素にする．

4.2 変換指定

変換は CAL の命令ごとに比較部と変換部を書いて指定する．LCP は比較部と原命令を分解した要素とを比較していって，一致したものがみつかるとそれ以下の変換部を実行してアセンブリ言語に変換する．

比較部

比較部は1桁目がハイフン (-) で始まり，終止符で終るもので，引用符でかこまれたリテラルと要素とを比較する．以下の例ではすべて Fig. 3 の区切りで原命令を分解するものとし，上の数字は分解されたときの要素の番号を示す．

例1

```
-'MOVE'.
```

これには1番目の要素が 'MOVE' ならマッチする．2番目以後の要素は何であってもよい．

例2

```
-'ADD'  *=3  'TO'.
```

星印は比較する要素の番号を入れておくところで，普通はリテラルがマッチするたびに1ずつ増えるが，これを任意に変更できる．この例では1番目の要素が 'ADD' で，3番目の要素が 'TO' ならばマッチする．したがって，

```
  1   2   3   45
ADD  I  TO  I.
```

はマッチするが，

```
  1   2345   6   78
ADD  A(B)  TO  J.
```

はマッチしない．

例3

```
-M(3)  'MOVE'  M(*+7)  'TO'.
```

M(3) は次の 'MOVE' があるかどうか3番目の要素までは探せということである．1, 2, 3 番目のどれかに 'MOVE' があればひとまずマッチして次を続ける．

次の M(*+7) は前の 'MOVE' でマッチした位置から7つ先の要素までの間に 'TO' があるかどうか探せということである．したがって，

```
  1  2     3          4     567   8   9 10
TENKI. MOVE  ALPHA(K)  TO  L.
```

はマッチするが，

```
  1    2   3   4 567 89   10  11 12
MOVE  A  IN  B (I, J)  TO  C.
```

はマッチしない．'MOVE' と 'TO' の間に8つの要素があるからである．

いつも最後まで調べてマッチしなかったことがわかるのは能率が悪いので，このように比較する位置の上限を指定することにした．

この場合，原命令の何番目の要素がマッチしたのかは B(s) で参照できる．すなわち，B(1) には最初にマッチした要素の番号が入り，B(2) には2番目にマッチした要素の番号が入る．したがって，前者の比較の結果は

```
B(1) =  3
B(2) =  8
```

となる．

変換部

変換の指定は，比較部の終止符の後に続いて書き，どんな情報をどこへ出すかを指定する．例題の右端の数字は説明の便宜のためにつけたもので SL の一部ではない．

例4

```
-'MOVE'  M(*+1)  'TO'                    1
  C=10,  'CLA'  C=15,  E(2)/             2
  C=10,  'STO'  C=15,  E(B(2)+1)/.       3
```

これに

```
  1     2   3   45
MOVE  M  TO  N.
```

という原命令がマッチしたとすると2行目の

C=10

から実行される．このCは次に出力すべき要素の出力位置を示している．すなわち，この場合は次の'CLA'というリテラルが 10 桁目から出力されることを示している．

2行目の終りの方の E(2) は分解した2番目の要素をそのまま出力することを示している．これは前に

C=15

があるので，15 桁目から出力される．2行目最後の斜線 (/) は出力行の終りを示す．ここで実際の出力が行なわれる．

3行目では，10 桁目から 'STO' を出し，15 桁目から B(2)+1 番目の要素を出力することを示している．B(2) は2番目にマッチした要素の番号であるから，この場合 'TO' の番号で 3 である．したがって E(B(2)+1) は E(4) と同じことになり 4 番目の要素 'N' が出力される．終止符は変換部の終りで，再び次の原命令を読む作業に移る．この結果さきの原命令は

```
10      15  けた位置
CLA   M
STO   N
```

と2つのアセンブリ言語の命令に変換される．

例5

```
-'MOVE'  M(＊+2)  'TO'.                    1
  C=10  'CLA'  C=15  E(2)/                 2
  ＊=2,  L(A1)  ＊=＊+2                    3
  C=10,  'STO'  C=15,  E(＊)/              4
  W(E(＊+1), ', ', A2,A1,A2) L(A2).        5
```

2行目までは例4と同じである．3行目の星印 (＊) には適当な番号を記憶したり，変更したりすることができる．ここには通貨記号 ($) も同様に使える．

L(A1) は名札の定義で，他の場所からここへコントロールを移すことができる．括弧の中の 'A1' が名札である．

5行目の W で始まるのは条件テストである．この場合は E(＊+1) の要素が ',' に等しいかどうかの比較をし，等しければ，その後に続く3つの名札のうち中央の A1 にコントロールが移る．小さければ左側，大きければ右側の名札にコントロールが移る．したがってこの場合，'TO' から後一つ置きにコンマ ',' があれば，名札 A1 以下を繰返えすわけである．

```
1     2   3   45  67  89
MOVE  A   TO  B,  C,  D.
```

が原命令なら，

```
CLA  A
STO  B
STO  C
STO  D
```

が出力されることになる．

条件の行き先を省略すると，すぐ次へコントロールが移るので，5行目は

W(E(＊+1), ', ', ,A1,).

と書いても同じである．

例6として CAL から TOSBAC-3400 のアセンブリ言語[3] への変換の実例を Fig. 6 に示す．全部は

```
+DELIMITER
 (),,
+INSTRUCTION
-'ADD' M(＊+4) ',' M(＊+4) '.'.                          1
 L(SB1)        ＊=2, M(E(,+1),'(',AD4,,AD3)              2
 L(AD1) C=2,   'LX', M(E(＊+2), N,AD2,) 'I'              3
 L(AD2) C=17,  F(＊+2), ',1'/                            4
 L(AD3) C=9,   '1A',                                     5
 L(AD4)        M(E(＊),N,AD5,) 'I'                       6
 L(AD5) C=17,  F(＊), M(E(＊+1),'(',AD6,,AD4) '-1,1'     7
 L(AD6)        /                                         8
 L(AD7)        ＊=R(2)+1                                 9
 L(AD8)        M(E(＊+1),F(',' AD9,,AD5)                 10
               M(E(＊+3),E(＊+2),,AD9,) G(AD1,AD3)       11
 L(AD9) C=2,   L(SB2) 'A'                                12
 L(SB3)        G(AD4,AD7)                                13
               ＊=B(3)+1, G(AD8,SB2)                     14
               'TA' G(AD4,AD7).                          15
-'SUB' M(＊+4) ',' M(＊+4) '.'.                          16
               G(SB1,SB2), 'NI', G(SB3).                 17
+ASSEMBLER
＊
      ADD   A,4,C          Source instruction
        LA      A          }
        AI      3          } Converted instructions
        TA      C          }
      SUB   TOKYO(2),OSAKA,NAGOYA(J)
        LXI     2,1
        LA      TOKYO-1,1
        B       OSAKA
        LX      J,1
        TA      NAGOYA-1,1
      ADD   15,MEMORY(K1),STORAGE(＊1)
        LAI     15
        LX      K1,1
        A       MEMORY-1,1
        TA      STORAGE-1,1
```

Fig. 6 An example of SL program and converted result from CAL to TOSBAC-3400 assembly language.

長いので加算と減算だけであるが，CAL の隣り合うオペランドが同じ添字を持っているときは，インデックスレジスターのセットは重複してしないようにしてあるし，TOSBAC-3400 には命令語のオペランドの値をそのままデータとしてあつかえる命令 (Immediate 命令といい，記号命令に 'I' をつける) があるので，CAL のオペランドや添字が定数のときはこれを使うようにしてある．

3行目の

W(E(＊+2), N, AD2,)

は，要素の先頭の一文字が数字かどうか調べるもので，この場合，要素の先頭の文字が数字なら，コント

ロールはすぐこの次へ移り，数字でなければ名札 AD 2 へ移る．11 行目の

G(AD1, AD3)

は，名札 AD1 から AD3 までを実行せよという意味である．これはとても有効で，このために減算の変換指定はわずか1行で済んでいる．

この他に，T(s) というのは名前を変更する機能をもっている．これは要素を表に登録し，

Xdddd

の形に置き換えて出力する．無論すでに登録してある要素は再登録はしない．dddd は 4 桁の数字で，表の登録番地である．Xに4桁の数字をつけた名前はたいていのアセンブリ言語の文法規則にあてはまると思われるので，どのように変換するかの指定はしないことにした．これは，原命令の名前が変換すべき文法規則に合わないときに変換したり，何回も出てくる数字定数を1回の定義だけで済ましたいときなどに使える．たとえば

W(N(*), 8, , ,BIG) E(*) G(Q) L(BIG) T(*) L(Q)

は要素の長さが8文字以下の場合は要素をそのまま出力し，9文字以上だったら名前を置き換えて出力することを示している．

以上で SL のほぼすべての機能を述べた．機能の一覧表を Fig. 7 にあげる．SL は LCP により通訳的に実行されるので，すべて最初の1文字を見れば何をすべきかがわかるようになっている．しかしこのために人間の読み易さは犠牲になってしまった．これはリ

```
L(l)          Label definition
N(s)          Continue the compare until element s
E(s)          Compare or output element s
T(s)          Output element s after conversion
P(s)          Position of element s
N(s)          Length of element s
G(l)          Go to l
G(l1,l2)      Perform l1 through l2
'any string'  Compare or output any string
W(op1,op2,l1,l2,l3)  Jump according to the comparison
W(op,N,l1,l2)        Jump if numeric
W(op,A,l1,l2)        Jump if alphabetic
C=|C+|n       Adjust output position
/             End of output line
.             End of matching or conversion
*=s
$=s           Assign the value s to

  Where  n is a integer,
         l is a label,
         op is E(s), P(s), N(s) or 'any string' and
         s is n, *[±n], $[±n] or D(n)[±n].
```

Fig. 7 List of SL instructions.

テラル以外の空白は無視することを利用して，人間が書くときに工夫することでおぎなわなければならない．比較部や変換部内での区切りはコンマ ',' であるが，空白を除去しても各指定の切れ目がわかるときは省略可能である．

4.3 制御カード

変換すべきプログラムの始まり，デバックのレベルなどは1桁目が＋で始まる制御カードで指定する．詳しい説明は省略し，一覧表を Fig. 8 にあげるにとどめる．

＋END

を除き，いずれもどこに置いても何回置いてもかまわない．いつも最後の指定が有効である．

```
 +DELIMITER   Definition of delimiters
 +INSTRUCTION Beginning of specifications
 +ASSEMBLER   Beginning of the language to be converted
 +END         End of processing
 +PUNCH       Output the converted language
o+NOPUNCH     Print only
 +DEBUG       Debug print a little
 +TRACE       Debug print medium
 +TRACEMORE   Debug print much
o+NODEBUG     Normal process without debug print
 +BLANK       Blank elements are valuable
o+NOBLANK     Blank elements are neglected

      o default
```

Fig. 8 List of control cards.

4.4 問題点

この SL が1桁の区切りだけで原命令を分解しているので，一つの文字がいろいろな意味で使われているときは困難を生ずる．たとえば '.' が小数点と終止符の両方の意味に使われているとき，これを区切りとして指定すると 3.14 という定数は3つに分解されてしまうのでまた組立て直さなければならない．

きた，多くのアセンブリ言語にあるように，オペランド欄の最初の空白以後は注記というのも処理が困難である．しかしながら今後も，'＊＊', '･EQ･' など2桁以上の区切りを指定できるようにしたり，数値の文字表現への変換の機能を加えるなどすこしずつ LCP を育てていきたいと思っている．

5. あとがき

以上のシステムはまだできたばかりで使用例は少ない．実際には，COBOL で書いてある LCP の手続き部だけを次の2通りの方法で CDC 6400 のアセンブリ言語 COMPASS への変換を実験してみただけである．

1. COBOL————→COMPASS
2. COBOL→CAL→COMPASS

この直接 COBOL から COMPASS に変換したものも，CAL を経由したほうも，結果はほとんど同じで，COBOL のときより，実行プログラムの容量が約2割増え，実行時間も2割位遅くなった．原因はいろいろあると思うが一つは変換の際に最適化を考慮に入れていないためである．この SL では，いつも原命令1個ずつしか見ることができないので，その中で複数個のレジスタを使いわける程度のことはできるが，前後をよく見渡して最適化することは不可能である．

なお，この LCP を使って COBOL からいわゆるワンパスで CAL や COMPASS に変換できているが，これは LCP がはじめからこのことを考慮に入れて注意深く書いてあるからで，いつでも COBOL のプログラムがこの LCP で簡単にアセンブリ言語に変換できるわけではない．

この CAL や LCP の応用例はまだこれだけであるが，これで COBOL からアセンブリ言語へ変換できたように，CAL の変換以外に他の用途にも使えると思う．たとえば，Aの計算機のアセンブリ言語からBの計算機のアセンブリ言語への変換は容易にできる．しかし，あまり異質の計算機同士の変換は結果の効率がはなはだしく悪くなるだろう．

CAL の各命令がマクロ命令としてシステムに登録されてしまうと LCP が不要になって一番いいのであるが．

これら，CAL と SL との言語の設計，および変換プログラムを作成するにあたって一番重点を置いたのが前にも述べたように互換性である．そこで，使用する文字もどの計算機にもあるような一般的なものだけを使い，プログラムは JIS 規格があり，しかも文字をあつかっても容易に互換性の得られる COBOL を使って書いた．そして，この規格[4]のうち，中核，表操作および順呼出しのそれぞれ水準1だけの機能を使い，計算機に固有の特長はいっさい使わないように注意深く書いてあるので，まだ実際には試みてみる機会がないのであるが，今使っている CDC 6400 の COBOL 以外でも必らず同じ結果が得られるはずである．

最後に，これらのシステムを作成するにあたって，いろいろ便宜をはかって下さった西ドイツのアーヘン工科大学計算センターの D. Haupt 教授，H. Petersen, H. Gipper のかたがたに感謝します．

参 考 文 献

1) 大駒誠一：コンパイラ・コンパイラ，情報処理，Vol. 11, No. 6, pp. 335～341 (1970).
2) 井上謙蔵：コンパイラ・コンパイラ，産業図書，1970.
3) 浦 昭二：アセンブリ言語，培風館 (1970).
4) 電子計算機プログラム用言語 COBOL，JIS C 6205-1972，日本規格協会.

(昭和47年5月30日受付)
(昭和47年7月28日再受付)