マイクロコンピュータ 36－2
(1985. 6. 18)

CP/M-86 アセンブリ言語へのデータ抽象化機能の導入

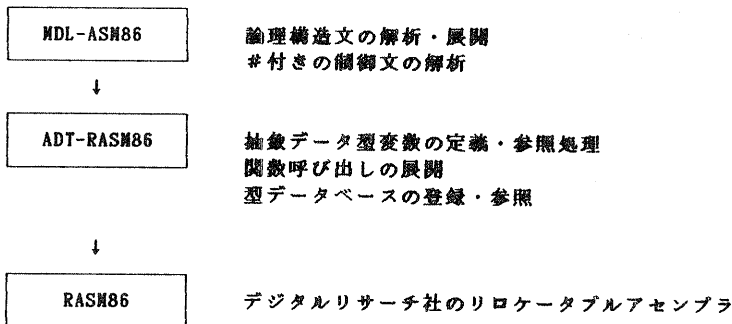
内田智史　　間野浩太郎
青山学院大学理工学部経営工学科

1．はじめに

前稿で報告した通り、我々は MDL-ASM86を作成し、実際のプログラム作成作業に適用して来た。MDL-ASM86 は、市販されているＣ言語コンパイラに比べると、圧倒的に8086の実行速度は速くなるが、大規模なプログラム作成に要する時間は、アセンブリ言語と比較しても多少良くなった程度であった（シンボリックデバッガの導入によるデバッグ効率の向上は著しかったが）。そこで、われわれは、データ抽象化をアセンブリ言語に導入することにより、(a) プログラミングの容易さ、(b) 部品化によるプログラムの再利用、(c) パッケージによる信頼性の向上、(d) タイプチェックによるプログラムの正当性の確認、などのデータ抽象化が持つ利点をアセンブリ言語にも適用しようと考え、ADT-RASM86を作成した。

2．ADT-RASM86の概要

ADT-RASM86システムは、次の構成になっている。



すなわち、論理構造文や#の付く制御文の展開は MDL-ASM86にまかせ、ADT-RASM86は、主に、抽象データ型の定義や宣言文の管理・関数呼び出しの展開・関数の引数や自動変数の処理を行う。また、抽象データ型はデータベース化して管理されるので、定義された抽象データ型のデータベースへの登録、あるいは、参照処理も行う。また、この他に、プログラムの部品化を支援するために、抽象データ型を定義しているパッケージから仕様書を作成する SPEC 、抽象データ型が正しく使用されているかチェックする TYPECHK、型に関する情報を提供する TYPESPEC がある。

具体的な ADT-RASM86 の特徴としては

(1) 抽象データ型の導入

利用者の定義できる型を導入することにより、作られるプログラムの設計思想をより明確なものにする。つまり、通常のアセンブリ言語の持つ型は、ワードやバイトなどのデータの大きさに依存したものであるが、このようにハードウェアに依存した型ではなくて、プログラミングオリエントな形である、ファイル型やテキスト型などのような型を採用すれば、より実際の対象により近いところでプログラミングすることができる。また、マルチウィンドウなどを表現する複数対象物の処理が容易に実現できる。

(2) パッケージ化

共通の抽象データ型変数の対象物に対する操作を１つのパッケージの中に統合する。これにより、情報隠蔽の利点が利用でき、抽象データ型を操作するプログラムの管理が容易になる。

(3) プログラム部品

抽象データ型をプログラム部品とし、それを再利用することによりアセンブリコーディングを簡単に再利用できるようにする。アセンブリ言語の場合、プログラムの新規開発には非常に時間がかかる。すでに開発されていて実際に使用されているプログラムは、かなりのテストを経ていると考えられるので、これを用いない手はない。

(4) タイプチェック

型の導入により、抽象データ型に対しての利用者の不正な操作を容易に発見できるのでプログラムの信頼性が向上する。

3．処理系の概要

処理系は次の４つから成る。なお、これらは CP/M-86 の RASM86 の上で動く。

・ ADT-RASM86 プリアセンブラ
・ SPEC　　抽象データ型に対する操作の一覧表・仕様書の出力
・ TYPECHK　抽象データ型が正しく使用されているかどうかの検査結果の出力
・ TYPESPEC　抽象データ型名に関する情報出力

3.1 ADT-RASM86プリアセンブラ

データ抽象化を円滑に記述するため、アセンブリ言語の中に、抽象データ型の対象物の定義(>typedef 文）とその型情報の型データベースへの登録や、それらの型の変数の宣言(>dcl 文）を組み込んだ。また、抽象データ型に対する操作を行ない易くするために、Ｃ言語風の関数呼び出しを導入した。さらに、モジュールとしての関数の記述を容易にするために、引数・自動変数の定義(>arg 文、>auto 文）が出来るようになっている。図3-1 にtypecmd をADT-RASM86で記述した場合のプログラム例を示す。

```
モジュール名        #module typecmd

宣言文 <<--->>      #dcl

                    :  >include system.db
                    :  include adt.mcr

                    #end dcl

 data   section     #data

                          >dcl FCB       *DefaultFCB=05ch
                          >dcl DMA       *DefaultDMA=80h
                          >dcl textfile InFile
                          >dcl string    SourceLine(200)
                          >dcl iostat    result

                    #end data


 code   section     #code


section <<  0>>     #proc main

        1           :  % result = file$open( DefaultFCB )
        2           :  *if result <> ErrorCode then
        3           :  :  % textfile$create( 'r',InFile,DefaultFCB,DefaultDMA )
        4           :  :  *loop テキスト表示
        5           :  :  :  % result = textfile$read( InFile,SourceLine )
        6           :  : <---*leave テキスト表示 ( result = EOF )
        7           :  :  :  % sysout$write( SourceLine )
        8           :  :  *end-loop テキスト表示
        9           :  *else
       10           :  :  % sysout$writeln("ファイルが見つかりません。");
       11           :  *end-if

                    #end main


 data   section     #data

                          EOF            equ      01ah
                          ErrorCode      equ      0ffh

                    #end data


      プログラムのエラー総数    0
```

図 3-1

3.2 SPEC

各抽象データ型の操作の仕様書を、(a) 概要、(b) 詳細のどちらかの形式で出力する。図3-2 にstring 型の仕様書を示す。

```
string型              仕様書      (詳細)
                               85/06/03  04:30:51

     文字列関係の処理  Ver. 1.0  Rev 001

  操作名                    意味
================================================================
  create                   c形式の文字列を文字型へ変換する
----------------------------------------------------------------
    (1) int string$create(str_adr,MAX,cstr);
        string *str_adr; /* 文字列型へのポインタ */
        const  MAX;      /* str_adrの領域の最大バイト数 */
        char   *cstr;    /* 元の文字列(C形式) */

        戻り値: 0:正常, 1:エラー(最大長を越えた)

        機能:C形式の文字列を文字列型へ変換する
================================================================
  append                   ある文字列の後ろに別の文字列を追加する
----------------------------------------------------------------
  (2)   void string$append(str_dest,str_source)
        string *str_dest; /* この文字列の後ろにstr_sourceの文字列が追加される */
        string *str_source; /* 追加される文字列 */

================================================================
  compare                  2つの文字列の比較
----------------------------------------------------------------
  (3)   int string$compare(str_a,str_b)
        string *str_a,*str_b;

        機能:str_aとstr_bの内容が等しいかどうか検査
             もし等しいなら, 0を返す
             もし等しくないなら,それ以外(0以外)の値を返す
================================================================
  index                    文字列探査
----------------------------------------------------------------
  (4)   int string$index(str_a,str_b,[start])
        string *str_a,*str_b;
        int     start;

        機能:str_aの中にstr_bがあればその位置を返す。なければ0を返す。
             start位置が指定されていれば,そこから探索を開始する。
================================================================
  indexc                   文字検査
----------------------------------------------------------------
        int string$indexc(c,str)
        char c;
        string str;

        機能:  str の中に c があるかどうか検査する。
        戻り値: 0:なし あった場合には、その文字の位置が返る。
================================================================
  copy                     文字列複写
----------------------------------------------------------------
  (5)   void string$copy(str_dest,str_source)
        string *str_dest,*str_source;

        機能:str_sourceをstr_destに複写する
================================================================
  load                     文字列ロード
----------------------------------------------------------------
  (6)   void string$load(str_d,pos,str_s)
        string *str_d,*str_s;
        int     pos;

        機能: str_s を str_s の位置 pos からロードする
```

図 3-2

3.3 TYPECHK

各操作が正しく使用されているか検査する。

3.4 TYPESPEC

型データベースから、各型の属性を収集し一覧表として作成する。

4．ADT-RASM86文法の説明

ADT-RASM86は、MDL-ASM86 に以下の文を追加した言語である。

(a) 型データベース取込文の定義
(b) 抽象データ型対象物の定義
(c) 抽象データ型変数宣言
(d) 引数変数の定義
(e) 自動変数の定義
(f) 変数セグメント属性指定文
(g) 関数呼び出し

4.1 プログラムの構成

プログラムは、モジュール(module)と抽象データ型パッケージ定義(package) に分かれる。モジュールとは単なる外部手続きのことである。

以下に、モジュールと抽象データ型定義の制御文の構造について示す。

```
$module モジュール名
$dcl
    使用するコードマクロや型データベースの引用の定義
$end dcl
$argument
    引数の定義
$end argument
$auto
    自動変数の定義
$end auto
$proc main
      STACKFRAME  自動変数の下図 *2 ←  RASM86のコードマクロ
                                      (スタックフレーム開始命令)
      STACKOUT     ←  RASM86のコードマクロ (スタックフレーム終了命令)
$end  (main)    ←   main を付けると retf 文を生成する。
$proc
    セクションの記述
$end

     :
     :
```

module の制御文

```
$package  抽象データ型パッケージ名
$dcl
      使用するコードマクロや型データベースの引用の定義
$end dcl
$note
  この抽象データ型全般に対する説明
$end note
$object
  抽象データ型名と寸法の定義
  抽象データ型の対象物の構造 (フレーム) の定義
$end object
$data
  使用する他の抽象データ型変数の宣言
$end data
$code
$operation  操作名
$note
  この操作に対する説明
$end note
$argument
  引数の定義
$end argument
$auto
  自動変数の定義
$end auto
$proc main
      STACKFRAME  自動変数の個数 *2
            プログラムの記述
      STACKOUT
$end
$proc セクション名
            セクションの記述
$end

      :
      :
```

package の制御文

```
― コードマクロの構成 ―
  STACKFRAME
       push   bp
       push   es
       mov    bp,sp
       sub    sp,自動変数の個数＊2

  STACKOUT
       mov    sp,bp
       pop    es
       pop    bp
```

4.2 変数の属性

抽象データ型変数には、全て対象物へのポインタがセットされる。対象物の領域がデータセグメント以外に確保される場合には、どのセグメントレジスタ相対になるか指定しなければならない。このように、変数を指定する際に、オフセット値だけでなく、セグメント値も必要な場合に、その変数はセグメント属性を持つという。ただし、抽象データ型変数の領域は必ずデータセグメント内に確保される。

4.3 抽象データ型パッケージ名と抽象データ型名

1つの抽象データ型パッケージに対して、そのパッケージに関連した複数の抽象データ型を定義できる。たとえば、 file 型という抽象データ型パッケージは、その中に　file 型, FCB型, DMA型という抽象データ型を定義している。つまり、抽象データ型パッケージ名とは操作の集合に対して付けられた名前であり、抽象データ型名とは、その中で用いる対象物に付けた名前である。

4.4 ADT-RASM86の文法の概要

以下にこれらの文法の概要を説明する。説明の中で[...] はその中が省略可能であること、{　} ... はその中を繰り返すことをそれぞれ示している。

[1] 型データベース取込文

プリアセンブルで利用する抽象データ型変数の属性が登録されているデータベースを指定する。この文の一般形式は次の通りである。

>include　データベース名

この文は一つのプリアセンブル過程の中で複数指定できる。この過程で、後述する抽象データ型対象物の定義(>typedef 文) があれば、その情報は最初に出て来たデータベースの中に追加あるいは変更される。

[2] 固定寸法の抽象データ型対象物の定義

抽象データ型の対象物は、寸法が常に固定的な対象物と、抽象データ型変数によって寸法が可変な対象物の2種類に分類される。たとえば、DMA 型という抽象データ型を考えてみると、これは CP/M-86では、常に 128バイトに固定されているので、その型として宣言された変数の取り得る寸法も常に 128バイトとなる。しかし、例えば、 string 型という抽象データ型では、その寸法は個々の抽象データ型変数によって異なる。例えば、str1という string 型変数は 256バイト、str2は 45 バイトというようにである。

まず、ここでは、固定寸法の抽象データ型対象物について説明する。

>typedef文により、抽象データ型の名前と全体の寸法を定義する。以下に、対象物の寸法が固定されている場合の >typedef 文の一般形式を示す。

>typedef　抽象データ型名[:セグメント名]　寸法（数値）

セグメント名は、 ds,es,ss,cs　のどれかである。セグメント名を指定すると、この型名を持つ抽象データ型の対象物は、そのセグメント相対となることを示す。このセグメント属性の指定はあくまでも規定値であり、後述する >dcl 文により変更することができる。

寸法には、その抽象データ型の対象物の寸法を記述する。たとえば、CP/M-86 では DMAバッファは 128バイトなので、 DMA型という抽象データ型変数の対象物の領域は 128バイトであることを指示している。この記述により、型データベースにこの情報が追加され、後述する>dcl文により、プログラム中に DMA型の変数が宣言されると、そこにその大きさの領域が確保されるようになる。

（例 1)　>typedef DMA:ds 128

DMA 型を定義する。この対象物はデータセグメント内にあり、寸法は 128バイトであることを示している。

（例 2)　>typedef textfile　5

textfile　型を定義する。この対象物は、通常はセグメントを意識する必要がなく（たいていはデータセグメントにある）、その寸法は 5バイトであることを示している。

[3] 固定された寸法の抽象データ型変数の宣言

>dcl文により、特定の抽象データ型の変数を宣言する。対象物の寸法が型に固定されている場合の>dcl 文の一般形式は、次のとおりである。

>dcl　抽象データ型名　{[*]変数名[:セグメント名] [=数値]} ...

変数名の前の＊は、この変数が対象物のポインタであり、この場合には実際の対象物の割り付けは行なわれないことを意味している。＊がない場合には、実際に対象物が型データベース中に定義された寸法で割り付けられ、その変数には、その対象物のオフセットポインタがセットされる。セグメント名は、型データベース中に登録されている（つまり>typedefで指定された）セグメント属性を変更する場合に指定する。数値は、＊が指定された場合のみに指定でき、その変数に直接オフセットポインタを設定する場合に用いる。なお、セグメント名に ns （ノンセグメント）を指定でき、それを指定した場合には、セグメント属性が解除されることを意味する。

(例 1)　>dcl DMA DMABuffer

DMA 型の変数 DMABufferを宣言している。また、この変数は必ずデータセグメント相対であるとして扱われ、128バイトの領域が確保される。これは次のように展開される。

```
DMA_b  rs  128
DMA    dw  offset DMA_b
```

(例 2)　>dcl DMA a:es,b:ss

DMA 型の変数 aと bを宣言している。a はエクストラセグメント相対、 bはスタックセグメント相対になる。この例で注意したいことは、対象物がそのセグメント内にあることを保証するのはあくまでも利用者の責任であるということである。b は実際には、この>dcl文のあるセグメント内（例えば#data... #end data 内にあればデータセグメント，#code...#end code 内にあればコードセグメント内）に配置される。従ってb を参照するときはスタックセグメントレジスタにはそのセグメント値を与えなければならない。

(例 3)　>dcl DMA *DefaultDMA=80h

DMA 型へのポインタ DefaultDMA を宣言している。この場合には、このポインタには 80hがその初期値として設定されている。これは次のように展開される。

```
DMA    dw  80h
```

[4] 可変寸法の抽象データ型対象物の定義と宣言

型の種類によっては、寸法を固定したくない場合もある。例えば、文字列を表現する string 型の場合には、その長さの指定は、宣言時に行ないたい。この場合には、次の例のように寸法の記述は $1+2 となる。実際の $1 の値は、>dcl文のパラメータで数値として与えられる。２だけ加算されているのは、 string 型の構造が、２バイトの文字数＋文字列の実体の形式をしているからである。

>typedef string:ds $1+2

これに対する>dcl文は次のようになる。

>dcl string str1(20)

この時、str1は string 型の変数であり、データセグメント相対であるとして扱われる。そして、その寸法は 22 として割り付けられる。つまり、>dcl文に於いて各変数の後に寸法をかっこで括って示すことができる。つまり、

>dcl　抽象データ型名　{ 変数名[:セグメント名] [(寸法{[, 寸法]}...)]} ...

となり、目的に応じて複数の寸法を指定できる。 $1 は１番めの寸法を意味し、 $n は n番めの寸法を意味している。

例として、氏名・住所・電話番号からなる抽象データ型 personal を考えてみる。氏名、住所、電話番号は、それぞれ異なる３つの string 型から構成されるとすると

>typedef personal　$1+$2+$3+6+4*3

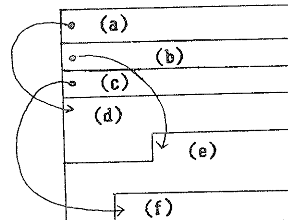
の大きさの領域が必要になる。 $1+$2+$3+6 (=($1+2)+($2+2)+($3+2)) は文字列そのものを入れる変数の大きさである。4*3 は、それらの文字列に対するポインタをセットする領域である。これに対する>dcl文は次のようになる。

>dcl personal table1(10,20,8)

名前用としては 10 文字分、住所用としては 20 文字分、電話番号用としては 8文字分の領域が確保され、全体で 10+20+8+6+4*3バイトの領域が確保される。

ここで注意しなければいけないのは、これは単に領域を確保するだけあって、以下のようなポインタ割り当ては、その抽象データを管理するパッケージの中で行なわなくてはならないということである。

(a)　名前用の文字列へのポインタ
(b)　住所用の文字列へのポインタ
(c)　電話番号用の文字列へのポインタ
(d)　名前用の実際の文字列
(e)　住所用の実際の文字列
(f)　電話番号用の実際の文字列



また、各対象物の内部構造（フレーム）は、各自 equ文で定義しなければならない。

例：　string型の場合

```
StringLength   equ   0
StringBody     equ   2
```

[5] 仮引数の定義

別のモジュールから操作が呼び出されるとスタックフレームが ASM86のコードマクロであるSTACKFRAMEによって形成される。このスタックフレーム上に仮引数や自動変数が展開される。

各モジュールや抽象データ型に対する操作に対する仮引数は次のように記述する。

```
>arg 抽象データ型名 ￥引数名 {[ ￥引数名 ]} ...
```

仮引数は、必ずその先頭に￥記号を付ける。この記号により、仮引数は、各処理ごとに作成される内部名に変換され、さらに、スタックセグメント相対になるように[bp]が付けられる。

(例) >arg sometype ￥sample と宣言されたとき

```
mov ax,￥sample
```

は実際には次のようなアセンブリコードに展開される。

```
mov ax,sample_10[bp]
```

仮引数が string 型のようにセグメントとペアになっている場合には次のように翻訳される。

```
#argument                                  str1_s_2  equ   6
  >arg string str1      －展開→           str1_o_2  equ   8
  >arg string str2                         str2_s_2  equ  10
#end argument                              str2_o_2  equ  12
```

仮引数がセグメント属性を持つ場合、>set文によってそのオフセット値とセグメント値とをレジスタに与えることが出来る。

```
                                           mov si,str2_s_2[bp]
>set ￥str2 si es      －展開→             mov es,si
                                           mov si,str2_o_2[bp]
```

これらの変数名（たとえば、 str2_s_2)の中で、最後に付けられた番号は、そのパッケージの中での操作定義番号である。これにより、同じパッケージの中でも、操作名が異なれば、同じ名前を異なる仮引数として用いることができる。

[6] 自動変数の定義

自動変数は次の一般形式で定義する。自動変数は仮引数と同じように実体は取られず、単にそのポインタが確保されるだけである。自動変数にも￥を先頭につける。

```
>auto 抽象データ型名 ￥変数名
```

この変数は、仮引数と同じようにスタック上に取られ、内部名に変換され[bp]が最後に付けられる。実際には次のようにアセンブリコードに展開される。

```
>auto textfile InFile                     InFile_3  equ -2
>auto string   str2      －展開→          str2_o_3  equ -4
                                          str2_s_3  equ -6
```

自動変数の最大の利点は、自動変数がスタックセグメント相対であるため、データセグメントの内容を変更しても使えるという点にある。

[7] 変数セグメント属性指定文

抽象データ型変数以外の変数に対して、セグメント属性を指定する場合には、次のようにする。

```
$ 変数名：セグメント名
```

ただし、この変数は、別に変数宣言をして領域を確保しなければならない。

(例)

```
$ xyz:ds
  xyz  dw  1000
```

[8] 関数、抽象データ型の操作の呼び出し

他のモジュール、抽象データ型の操作を関数として呼び出すことができる。その一般形式は次の通りである。

関数の場合（一般のモジュールを呼び出す場合）

```
% [ v ] = 関数名( {[実引数1]}... )
```

抽象データ型の操作の場合

```
% [ v ] = 抽象データ型名$操作名( {[実引数1]}... )
```

ここで、vは、変数名かレジスタ名（セグメントレジスタも含む）である。実引数には、変数・定数・”（二重引用符）で囲まれた文字列が記述できる。セグメント属性が指定されている変数の場合、その変数，指定されているセグメントという順序でスタック上に送られる。実引数が >arg 文で指定された仮引数の場合には、オフセットの部分（例えば str_o_3[bp])，セグメントの部分（例えば str_s_3[bp])のようにスタック上に送られる。文字列の場合には、データセグメント相対の string 型の内部変数が生成され、

そのポインタ渡しに変換される。つまり実引数がセグメント属性を持つ場合には、必ずオフセット値、セグメント値の順で渡される。送り出す際には、 ax レジスタに引数の個数を入れ、関数の結果は ax に戻る（と約束する）。結果がダブルワードになる場合には、ax,dx に戻る。
（例）

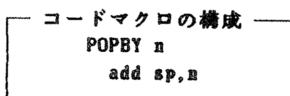
```
>dcl DMA *DefaultDMA=80h
>dcl FCB *DefaultFCB=5ch
>dcl textfile InFile
```

と宣言されているとき

```
% textfile$create('r',InFile,DefaultFCB,DefaultDMA);
```

は次のように展開される。

```
push DefaultDMA
push ds
push DefaultFCB
push ds
push InFile
mov ax,'r'
push ax
mov ax,4
call textfile$create
POPBY 2*6
```



文字列は次のようになる。

```
% i = string$index("This is a pen","is")

push sssss_1
push ds
push sssss_2
push ds
mov ax,2
call string$index
POPBY 2*4
mov i,ax

      dseg
sssss_w_1   dw   13               ←――――― 文字数
            db  'This is a pen'   ←――――― 実際の文字列
sssss_1     dw   offset sssss_w_1
sssss_w_2   dw   2
            db  'is'
sssss_2     dw   offset sssss_w_2
```

なお、RASM86では、「抽象データ型名$操作名」における$は文字としては全く無視されるので、実際には$が抜けて「抽象データ型操作名」として扱われる。

6．考察

高級言語にデータ抽象化機能を導入する場合に比べて、CP/M-86 の RASM86 アセンブリ言語にそれを導入するには、いくつかの問題点があった。本節では、データ抽象化をアセンブリ言語に導入する際に特に注意した点について述べ、次にプログラムの部品化について述べることにする。

6.1 アセンブリ言語にデータ抽象化を適用する際に特に注意した点

[1] 処理系の簡素化

我々の開発環境は、主記憶 384KbytesのPC-9800E，1Mbytes 8 inchフロッピーディスク×2 、10Mbytes固定ディスクである。これらの上に、処理速度と記憶効率の点から実用的な処理系を作成するには、処理系の簡素化は重要な問題である。従って、ADT-RASM86は、MDL-ASM86 と組み合わせて使用する形式を採用し、処理系をコンパクトにした。また、２つの前処理系の導入によりプリアセンブル時間が大幅に増大するので、処理速度を上げるために、主記憶の一部とVRAMをRAM DISKとして用い、処理速度の向上を計った。

[2] 抽象データ型の簡易的な実現

高級言語とは異なり、データ抽象化の最終的な表現形式が、その利用者に見える必要がある。その理由としては、(a) 処理時間がかかってはいけない。アセンブリ言語の最大の特徴である実行効率は犠牲にはできない。(b) オブジェクトサイズが大きくなってはいけない。つまり利用者が常に効率の良いオブジェクトコードを意識して構築できるようにしておかなければならない。(c) アセンブリ言語では、きめの細かい処理が要求されている。たとえば、文字列を扱う string 型という抽象データ型の各操作では、ほとんどの処理は、8086のリピートプリフィックス命令を用いて高速に処理しなければならない。（この場合には、データセグメントの内容を変更させる必要がでてくる。従って、この場合には、データセグメント相対の変数は使えず、スタックセグメント相対の自動変数を用いる）。

[3] メモリ管理

CP/M-86 のメモリ管理機能の低さ（８レベルまでしかエリア確保が出来ない）と実行速度の観点から、処理系の標準的な仕様としてはメモリの動的割り付けは行なわず、全て静的に確保することにした。ケイパビリティも採用しない。しかし、動的対象物を扱うプログラムやエディタのように大きなメモリを扱うプログラムの作成時には、動的にメモリを割り付けたい場合もあろう。特に、扱う対象物が頻繁に移動するような場合である。その場合には、動的割り付けを支援する抽象データ型を特別に用意し、利用することにした。たとえば、その目的のために、 list 型や heap 型を用意することにした。

アセンブリ言語のように、データ領域の全域が可視状態である環境では、部分的な情報隠蔽しか実現できないので、これによるプログラム信頼性向上は過度には期待しないことにした。

6.2 プログラムの部品化

[1] 基本的な抽象データ型の充実

アセンブリ言語の基本レベルでは、小さなプログラムを組むのも非常に面倒である。従って基本的な操作は、組込み機能として備わっていなければならない。まず、CP/M-86 に備わっている基本機能は全て抽象データ型として構成する必要がある。このことにより、以下の利点が得られる。

(a) 移植性

BDOSなどの OS 固有の機能を抽象データ型として吸収することにより，異なる OS （例えば MS-DOS ）への移植性を高めることができる（BDOSの抽象データ型については後述）。

(b) コンパクト性

基本機能を標準化することにより、やたらと部品が多くなることを防ぐことが可能となる。つまり、なるべく基本機能の組み合わせでプログラムを作成するようにする。

(c) CP/M-86 の機能向上

CP/M-86 には、備わっていない多くの機能を用意することにより、CP/M-86 の機能の低さを補う。例えば、文字列を管理する string 型、データベースを管理する database 型、Ｃ言語と同じ入出力（fopen,printf,fprintf等）を実現するＣ型など。

[2] BDOS の抽象データ型

BDOS は、パラメータが原始的で分かりにくく、たとえば、ファイルをオープンするにも FCBを作成するなど面倒な点も多い。これでは、データ抽象化を導入した意味が無くなってしまう。そこで、BDOS（拡張BDOSも含む）に対応するいくつかの抽象データ型が用意されている。

(a) file型　：ファイル操作・ディスク操作
(b) sysout型　：コンソール出力
(c) sysin 型　：コンソール入力
(d) printer 型：プリンタに関する操作
(e) memory型　：メモリ管理、その他
(f) clock 型　：時間・時計・タイマ
(g) key 型　：キーの設定

7．おわりに

ADT-RASM86は、１９８５年３月上旬にプロトタイプ版が完成し、それを用いて３月上旬から４月上旬にかけて、実際にマルチウィンドウ型エディタ（感じとしては日本語ワードスターのマルチウィンドウ版である：WardStar is a trademark of MicroPro International Corporation ）を作成してみた。本報告での多くの文法は、この時の反省に基づいて導入されたものであり、そこで発生した多くの問題点は解決された。紙面の関係からここではいまだに未解決の問題点について述べる。

(1) 抽象データ型の設計：初心者にとって、どのような抽象データ型を作成すれば良いのかの決定は非常に難しい。これは、いわゆるモジュラ分割や Lisp の関数の設計の難しさにも似ている。また、どのような抽象データ型を用意するかは、プログラム部品化の立場からも重要な問題である。前述のエディタの作成では、テキストの管理とリストの記憶管理がごちゃごちゃになってしまった。

(2) プログラム部品の標準化：いわゆる使いやすい汎用的なプログラム部品としての抽象データ型を用意することもまた非常に困難な仕事である。

すなわち、これらの問題点は、何を抽象データ型の対象物として選ぶかという決定である。これらは、今後の使用経験の中で解決してゆこうと思っている。

本報告で述べた文法を持つ ADT-RASM86 は、プロトタイプ版(>typedef,>dcl,>arg,>auto 等の処理がない）を用いておよそ８割が完成している。1985年 6月中旬には全システムが完成する予定である。

なお、プロトタイプ版で完成したエディタは、市販されている類似のマルチウィンドウ型エディタと比較しても遜色がないもので、機能・実行速度の点でははるかに優れている。このようなエディタが比較的短期間（1.2 人月）で完成したのは、データ抽象化の効果であると考えている。今後は、このシステムを用いて、プログラムの部品化の応用の研究を行う予定である。

最後に、処理系作成に協力している卒研生の木村、西岡、余越、加藤君、ならびに間野研究室の方々に感謝します。

［参考文献］

［内田83］ 内田智史：Improving Method of Programming Environment Based on Data Abstraction
1983年度、青山学院大学理工学部経営工学科修士論文（1984.3）

［内田85］ 内田智史・榎田幸・戸浪聡・間野浩太郎：CP/M-86 のアセンブリ言語への構造化プログラミングとモジュラプログラミングの導入
情報処理学会　マイクロコンピュータ研究会資料36-1（1985.6）pp.1-10

［佐渡81］ 佐渡一広・米沢明憲：抽象データ型言語
情報処理学会論文誌　Vol.22 No.6（1981）pp.525-530

[Isner82] John F.Isner: A Fortran Programming Methodology Based on Data Abstraction
CACM Vol.25 No.10（1982）pp.686-697

[Stroustrup82] Bjarne Stroustrup : An Abstract Data Type Facility for the C Language
SIGPLAN NOTICES Vol.17 No.1 JANUARY 1982