プリアセンブリ言語 ADT-RASM 86 とそのデータ抽象化機構†

内田智史†† 間野浩太郎††

一般にアセンブリ言語は，実行効率は良いがプログラム作成効率が低い．本論文では，アセンブリ言語によるプログラミングの生産効率を上げるために，これにデータ抽象化を導入することを提案し，非常にコンパクトな機構でそれが実現可能なことを示す．ところで，データ抽象化をアセンブリ言語に導入すると，高級言語への導入とは違った問題が生じる．それは，データ抽象化を導入すると，a）実行時に多少のオーバヘッドが存在する，b）記憶の使用効率が悪くなる，c）デバッグが困難になる，という点である．前者の2点を解決するために，抽象データ型の形式として複合型パッケージを提案する．複合型パッケージでは，その中で関連した複数の型を定義する．したがって，記憶上の操作を細かな単位で行うことができるので，記憶効率・実行効率が良くなる．c）を解決するために，対象物の先頭部分に対象物の属性を格納したヘッダを付け，実行時チェックをできるようにする．これらの方針のもとにマイクロコンピュータ用のオペレーティングシステムである CP/M-86 のアセンブリ言語である RASM86 へのプリプロセッサとして，ADT-RASM86 を作成した．本論文では，アセンブリ言語という低水準の言語にも，データ抽象化が非常に単純な方式で実現可能であることを，ADT-RASM86 の実現例から示し，そのデータ抽象化機能の実現方法について示す．

1. まえがき

マイクロコンピュータの普及に伴い，生産すべきマイクロコンピュータ用ソフトウェアの量が増大している．マイクロコンピュータ用ソフトウェアの開発におけるアセンブリ言語の重要性は，C言語の採用の増大により低くなって来ているが，それでも，通信を含む制御系や会話型ソフトウェアなどのように実行効率・記憶効率が特に重視される環境では，アセンブリ言語の使用は必須である．しかし，アセンブリ言語は，他の高級言語に比べて，プログラミングの効率が悪いうえに，作られたプログラムの保守性が低く，多品種のソフトウェアの大量生産には向いていない．このことは，実務レベルのプログラミングにおいて大きな問題となっているが，これといった有効な手段のないまま，現場のプログラマが悪戦苦闘しているのが現状である．そこで，我々は，データ抽象化[9]をアセンブリ言語に導入し，データ抽象化に基づくプログラムの部品化により，多品種のプログラムの開発期間の短縮を可能にするプリアセンブリ言語システム ADT-RASM 86[13),14)] とその支援ツール群をマイクロコンピュータ 8086 上の OS である CP/M-86[1)] 上に作成した．

本システムにより，従来アセンブリ言語でないと効率的に記述することができなかったプログラムを，抽象データ型の概念を用いてプログラミングすることが可能になった．特に CPU の特徴を活かした細かな動作を行う抽象データ型を作成できるので，エディタやワードプロセッサ，データベースなどの OA 用ソフトウェア，BIOS や BDOS を扱うシステムプログラムの記述に適している．

ところで，データ抽象化の利点が認識されるにつれて，様々な高級言語にデータ抽象化機能を導入する試みがなされている．たとえば，C言語の Classes[10)] は代表的である．また，マクロ形式で主に PL/I に対してデータ抽象化を実現している処理系もある[4)]．このように高級言語に対してデータ抽象化を適用することは比較的容易であるが，アセンブリ言語へそれを導入するには，高級言語とは異なった考慮が必要となる．別々にコンパイルでき，そのコンパイル単位内に静的な局所変数を持つことができ，そのコンパイル単位へのマルチエントリ機能があれば，抽象データ型を形成することができる[7)]．データ対象物（以下，単に対象物と略す）の記憶域割り付けは，一般には，ケイパビリティベースドアドレシングによる動的割り付けが考えられる．しかし，データ抽象化をアセンブリ言語に導入する際に最も考慮すべき点は，アセンブリ言語の利点である実行速度を低下させないことである．多くの抽象データ型言語では，対象物を実行時に動的に割り付けているので，その寸法を実行時に決定できるという利点がある．しかし，この方式を採用すると実行時に割り付けのオーバヘッドが存在してしまうので，我々は対象物の寸法を固定し，静的に割り付けることを基本とした．この方法は，プログラミングの自由度を多少損ない，記憶効率を低下させるが，実行効率は

† Pre-Assembly Language: ADT-RASM86 and Its Data Abstraction Facility by SATOSHI UCHIDA and KOUTARO MANO (Department of Industrial and Systems Engineering, College of Science and Engineering, Aoyama Gakuin University).

†† 青山学院大学理工学部経営工学科

良くなる．そこで，記憶効率を多少なりとも高め，さらに実行効率を向上させるために，小さな単位の複数の抽象データ型をグループ化して扱う複合型パッケージを導入した．また，マイクロコンピュータという小さな動作環境も考慮しなければならないので，本システムはコンパクトな機構で実現できるように設計する必要がある．

ADT-RASM 86 は，1985年1月にプロトタイプ版の開発が始まり，約2人月で同年3月上旬に完成した．その後，プロトタイプ版を用いて3月上旬から約1.8人月をかけて，マルチウィンドウ型スクリーンエディタを試作し，その使用経験を基に再設計し，1985年5月から約1人月かけて ADT-RASM 86 処理系が完成した．

本論文の目的は，アセンブリ言語の利点を損なわずに，アセンブリ言語に対してもデータ抽象化が非常にコンパクトな機構で十分適用可能であることを，ADT-RASM 86 の実現例から示し，その実現手段を示すことである．また，アセンブリ言語におけるデータ抽象化を利用したソフトウェアの再利用方式とその支援ツールについても述べる．

2. ADT-RASM 86 のシステム概要

ADT-RASM 86 システムは，以下の処理系群から構成される．(1) ADT-RASM 86 データ抽象化プリプロセッサ：データ抽象化機能を CP/M-86 のアセンブリ言語である RASM 86[2] に変換する．(2) ADT-RASM 86 ドキュメンテイタ：ADT-RASM 86 のソースプログラムを整書出力する．(3) SPEC：ADT-RASM 86 のパッケージの仕様書を出力する．ADT-RASM 86 は，我々の研究室内で実用に供され，ADT-RASM 86 処理系群やスクリーンエディタ，コマンドシェルなどのユーティリティが本システムを用いて作成されている．**図1**に，本システムの概要を示す．

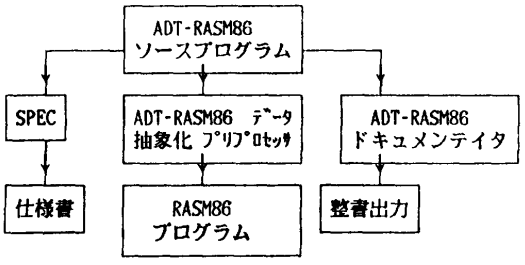

図 1 ADT-RASM 86 システムの概要
Fig. 1 The outline of the ADT-RASM 86 system.

3. ADT-RASM 86 のデータ抽象化機構

3.1 複合型パッケージ機構

3.1.1 複合型パッケージの提案とその実行効率

抽象データ型の対象物へアクセスする場合，カプセル化のためにその抽象データ型に付随する操作でしかデータを操作できないので，冗長性やオーバヘッドが多くなる．カプセル化は，データの安全性を高めるという利点はあるが，高速性が要求されるアセンブリ言語の使用される環境では，こういったオーバヘッドは極力減らさなければならない．そこで，効率の良いデータ抽象化を実現するために，複合型パッケージを提案する．複合型パッケージは，関連し合う複数の型の定義群とそれらの型の対象物の内部構造に対する操作群から成る．つまり，関連し合ういくつかの抽象データ型をグループ化して一つにまとめ，そのグループ内にある抽象データ型はお互いに内部構造を可視にする．このようにすれば，そのグループ内の抽象データ型の対象物間の操作に伴うオーバヘッドが軽減されるので，実行効率の良いプログラムを書くのに有効となる．**図2**に，ADT-RASM 86 の複合型パッケージの概念図を示す．例として，テキストを CRT 画面に表示する処理を考える．CRT 抽象データ型と text 抽象データ型の二つが完全に異なるモジュールとして定義されていると，text 型の対象物からいったんテキスト行を取り出して，中間的なバッファに蓄え，それを CRT 抽象データ型に送らなければならず，いったんバッファにデータを蓄えるという無駄がある．ADT-RASM 86 の方式では，CRTtext 複合型パッケージの中に，CRT 型と text 型の二つの型を定義する（**図3**）．したがって，text 型の対象物を直接 CRT 型に送ることができ，実行効率が良くなる．

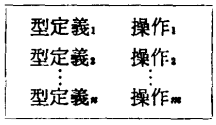

図 2 ADT-RASM 86 の複合型パッケージの概念図
Fig. 2 The conception of the compound type package of ADT-RASM86.

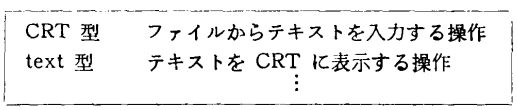

図 3 CRTtext 複合型パッケージ
Fig. 3 CRTtext compound type package.

3.1.2 複合型パッケージと既存の言語との関係

複合型パッケージの概念は，一つの複合型パッケージで複数の型が定義できるという点で，CLU[8] などの抽象データ型のそれとは異なる．しかし，次の条件を満たす高級言語では，複合型パッケージを実現できる．それは，1) 他のプログラムで使用可能なデータ型（レコード，構造体を含む）を定義する機能，2) データ型の内部構成をそのパッケージ以外のプログラムから隠蔽する機能，3) 複数の手続き（操作）を一つのコンパイル単位に含める機能，である．たとえば，private 型あるいは limited private 型を可視部で宣言した Ada のパッケージ[5] では，複合型パッケージを実現できる．また，C言語の struct を用いた場合には，複数の構造体の定義とそれらに対するいくつかの操作を一つのファイル中にまとめることができるので，1) と 3) の条件は満たすが，この方式では，他のファイルでも同様の struct 宣言が必要となるので，構造体の内部構造，すなわちメンバ名がそのファイルの外側で可視になり，2) に違反する．したがって，C言語では，構造体の内部構造が外部で見えても使用しないという条件のもとに，複合型パッケージを実現できる．ADT-RASM 86 は，上記の三条件を満足する複合型パッケージを記述する文法上の機構を持っている．

3.1.3 データ対象領域の共用利用

ところで，複合型パッケージのもう一つの利点として，データ対象領域の共用利用がある．たとえば，ファイルを扱う場合に，ファイル型という単一の型で扱うのではなく，ファイルという複合型パッケージを定義し，その中に，ファイルポインタや名前を管理する FileName 型と，入出力バッファである FileBuffer 型の二つの型を定義する．こうすると，この二つの型は，分割して扱えるので，必要に応じて組み合わせることにより，記憶域の共用利用ができる．この方法は，抽象データ型のモジュール性を低下させるので，作成されたプログラムの信頼性を低くする可能性があるが，アセンブリ言語にデータ抽象化を効率良く導入するには，場合に応じて原始的な記述も可能にする必要がある．

3.2 アセンブリ言語における型の概念

ADT-RASM 86 では，すべての変数は型付けされている．型は，RASM 86 の db, dw などの疑似命令によって定義され，アセンブリ命令の操作対象となるハードウェア固有の型と，ADT-RASM 86 のデータ抽象化機構によって定義される利用者定義の型の二種類に分類される．利用者定義の型を持つ変数は，対象物へのポインタがセットされる．その型が定義されている複合型パッケージの外側では，手続き呼び出しの実引数にのみ変数の記述が許され，他の操作はできないので，その対象物の内部構造に対するアクセスの手掛かりを与えない．その変数は，その型が定義されている複合型パッケージの内部に引数として引き渡されて，初めて，ハードウェアの特性を活かした効率の良い処理が行われる（図 4）．

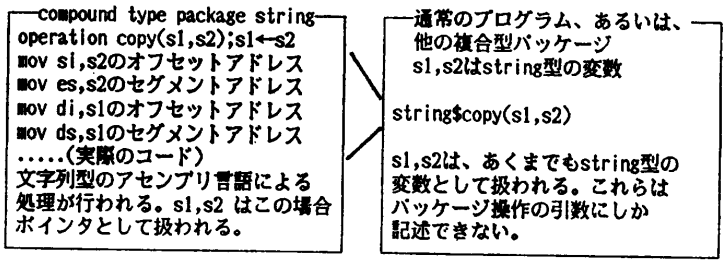

図 4 アセンブリ言語における抽象的な型の概念
Fig. 4 The conception of an abstract type in assembly language.

3.3 型情報管理表に基づいた対象物の管理

ADT-RASM 86 では，利用者定義の型の変数を定義する際に，割り付けるべき領域の寸法を寸法子によって指定する．与えられた寸法子から実際に割り付ける寸法値を計算する（図 5）．型付けされた対象物の割り付けに関する情報は，すべて型情報管理表を通して入手される．型を定義する文により，複合型パッケージ中に型情報管理表への登録が行われる．登録される項目は，型名，寸法子の個数，パラメータ式，その対象物が割り付けられるセグメント（これを配置セグメントと呼ぶ．4.2 節参照），定義されている複合型パッケージ名である．パラメータ式には，割り付ける寸法の計算式が格納されており，たとえば，図 5 の $1 は 1 つ目の寸法子と置換されて，計算がなされることを意味している．

3.4 対象物のヘッダによるエラーの発見

抽象化された変数の不当な扱いの検出は，アセンブリリストを見ても，または，アセンブリ言語用のトレーサで追っても，それらが扱う対象が細かすぎて容易ではない．たとえば，最大 10 文字をセットできる文字列型変数にそれよりも長い文字列データを入れるといったエラーの発見はかなり困難である．ADT-RASM 86 のプロトタイプ版による我々の経験から，

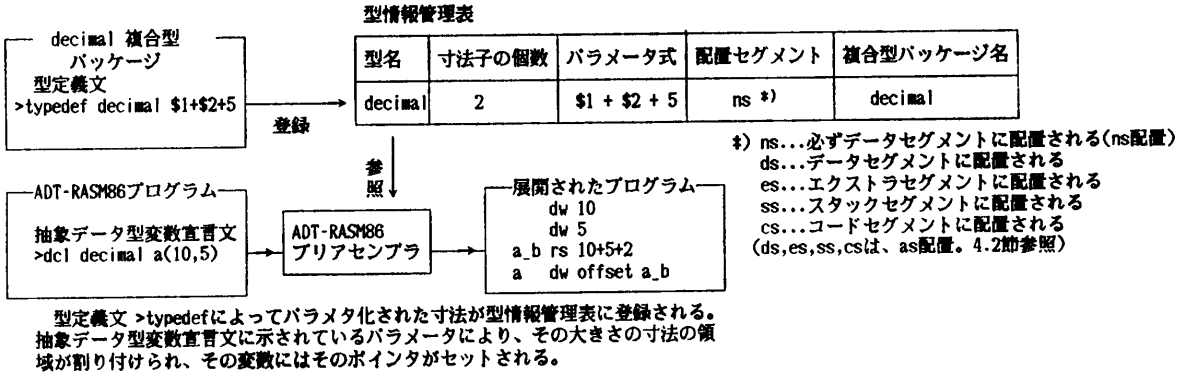

図 5　型情報管理表によるデータ対象物の管理
Fig. 5　The Management of the data object by using the type information management table.

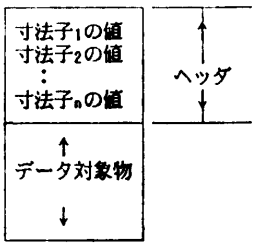

図 6　データ対象物の構造
Fig. 6　The structure of the data object.

この種のエラーの大半は CPU を暴走させてしまい，その修正に多大の時間を要することが分かった．そこで，割り付けられた各対象物のヘッダに，その対象物の寸法をセットし(**図 6**)，このヘッダを用いて，対象物への不当な操作を検出するようにした．また，このヘッダにより割り付けられた対象物の寸法をプログラムで知ることもできる．配列も抽象データ型の特別な形と考え，配列の次元や添字値などを入れたヘッダを用意する．このため，引数に配列を記述する場合には，その配列をアドレスを渡すだけでその次元や寸法の情報が分かるので，呼び出された操作の側で配列の添字のオーバフローを検出することがこのヘッダにより可能となる．

4. ADT-RASM 86 の文法

ADT-RASM 86 は，RASM 86 固有の文，データ抽象化機能を実現するために追加された文および複合型パッケージや一般のモジュールを形成する疑似命令文からなる．

4.1 ADT-RASM 86 の疑似命令

ADT-RASM 86 では，RASM 86 の疑似命令のほかに，# 記号で始まる ADT-RASM 86 の疑似命令を使用する．これは，複合型パッケージや一般的な手続き（これを ADT-RASM 86 では，モジュールと呼ぶ）の枠組みを構成するために用いる．

```
#package　パッケージ名
#dcl
    >include 型情報管理表名
        ⋮
#end dcl
#note*)
    概要の説明
#end note
#object
    型の内部構造の定義
#end object
#operation 操作名
#argument
    仮引数の宣言
#end argument
#auto
    自動変数の宣言
#end auto
#proc main
    メイン処理記述
#end
#proc 副処理手続き名
    副処理記述
#end
        ⋮
#operation 操作名
        ⋮
```

*) このほかにも，
#syntax (構文，使用法)，
#history (開発過程)，
#limitation (制限)，
#example (使用例)
がある．
モジュールを記述するには，
#module 制御文を用いる．

図 7　複合型パッケージを形成する疑似命令
Fig. 7　The directives which form the compound type package.

複合型パッケージは，**図 7** に示す疑似命令で形成する．一つのプリアセンブル単位となるソースファイルの中には，複合型パッケージやモジュールを一つだけ記述する．

4.2 対象物の配置

8086の記憶域は，64 kバイトのセグメントから構成されている．記憶域を有効に使用するには，どのセグメント上にも対象物を配置できるようにする必要がある．しかし，実際には，対象物の大半がデータセグメント上に配置されるので，ADT-RASM 86 では，(1) データセグメント上のみに配置される対象物，(2) 任意のセグメント上に配置される対象物，の二種類の対象物を区別して扱う．前者を ns (non segment) 配置，後者を as (any segment) 配置と呼ぶ．型定義時に，その型の対象物の配置を指定する．as 配置の場合には，型定義時にその対象物が配置されるセグメント（配置セグメント）を指定する．ns 配置の場合には，対象物を指すポインタはオフセットアドレスのみの 16 ビットであるが，as 配置の場合には，セグメントアドレス・オフセットアドレスの32ビットになる．

4.3 データ抽象化のための文

4.3.1 型定義文

ADT-RASM 86 では，多くのデータ型を扱うが，これらの情報はすべて型情報管理表の中にセットされている．型情報管理表を取り込むための文として，>include がある．これは，そこに指定されている型情報管理表をアクセスし，データ型に関する情報を取り込む．型情報管理表に，型の情報を与えるための文として，>typedef がある．たとえば，string 型は，文字列の実体とその寸法を保存する領域が必要であるので，50 文字分の string 型変数は，寸法子の値＋寸法子の長さ＝50＋2＝52 バイト必要となる．これは，次のように宣言する．

```
>typedef string : ds $1+2
```

ここで，$1 は，後述する抽象データ型変数宣言文における1番目の寸法子の値を意味し，:ds は，この型の対象物が as 配置でそれが配置されるセグメント(配置セグメント）は，データセグメントであることを示す．なお，対象物のデータ構造は，EQU 疑似命令を用いたオフセットによって定義する．string 型のデータ構造の定義は，以下のようになる．

```
StringMaxLength     equ (-2)
StringLength        equ 0
StringBody          equ 2
```

他の例を示す．DMA 型は，CP/M-86 では，128 バイトと決まっているので，

```
>typedef DMA : ds 128
```

decimal 型は，整数部と小数部の数値を格納する領域のほかに符号の管理などの情報を格納する領域を5バイト必要とするので，

```
>typedef decimal $1+$2+5
```

となる．この場合，decimal のあとにセグメントの指定が無いので，ns 配置であることを意味する．

4.3.2 抽象データ型変数宣言文

抽象データ型変数を宣言するには，>dcl 文を用いる．最大長が 50 と 105 の string 型変数（それぞれ，s1 と s2) を宣言するには次のようにする．

```
>dcl string s1(50), s2(105)
```

この場合，s1, s2 のためのデータ領域が指定された寸法でデータセグメント上に割り付けられ，s1, s2 には，その領域へのポインタがセットされる．また，CP/M-86 のデフォルト DMA アドレス値は，$80_{(16)}$ である．これは，DMA 型として扱われるが，変数名の頭に星印を付けることで，$80_{(16)}$ を指す DMA 型のポインタ変数を宣言できる．この場合は，ポインタなので対象物は割り付けられず，単にポインタ値を格納する記憶領域だけが割り付けられる．なお，以下の例で 80H は RASM 86 の記法であり，$80_{(16)}$ を意味する．

```
>dcl DMA *DefaultDMA=80H
```

このように，アセンブリ言語での概念も型として扱うことが可能である．

decimal 型（10 進演算型）は，整数部と小数部に寸法を分けて宣言する．整数部の寸法が 40 桁，小数部の寸法が 15 桁の decimal 型の変数 d を宣言するには次のようにする．

```
>dcl decimal d(40, 15)
```

配列を宣言するには，{ } を用いる．

```
>dcl string s(10) {2, 3, 4}
```

なお，次のように変数名の後に，:es と宣言すると，>typedef 文によって定義された配置セグメント上ではなく，エクストラセグメント上に割り付けられる．

```
>dcl string s : es(10)
```

引数は，>argument 文，自動変数は >auto 文で宣言する．

4.3.3 複合型パッケージ操作呼び出し文

対象物の操作は，CLU に似た以下の形式で呼び出す．

```
% 複合型パッケージ名 $ 操作名 (実引数 [, 実引数]…)
```

```
モジュール名     #module submit

宣言文 <<--->>   #dcl
                 :  include adt.mcr
                 :  >include system.db
                 #end dcl

 note   section  #note
                       作成者 : 内田智史
                       連絡先 : 03-307-2888 (内線 268 あるいは 367)
                       作成日 : 1985 年 8 月 6 日
                       公開   : 自由
                 #end note

 usage  section  #usage
                       CP/M-86 の submit コマンドの高速版(Aドライブを見ずにメモリ上で実行)
                       使用法は、CP/M-86 の submit コマンドと同じ
                       引数を指定しないと、バージョン番号を表示する
                 #end usage

history  section #history
                       1985/8/ 6 初版
                       1985/8/13 textfile型の修正に伴ない、修正した
                       1985/8/14 system型の修正に伴ない、Version番号のみ修正した
                       1985/8/15 部品ライブラリ(part.186)の不良に伴ないVersion番号のみ修正した
                 #end history

 data   section  #data
                       >dcl integer  ArgCount,n
                       >dcl string   ArgumentBody(30)
                       >dcl string   TextLine(255)
                       >dcl string   RepNumber(3),RepBody(5)
                       >dcl FCB      *DefaultFCB = 05ch
                       >dcl DMA      *DefaultDMA = 80h
                       >dcl textfile SubmitFile
                       >dcl argument ArgumentBuffer
                 #end data

section <<  0>>  #proc main
           1 *   :  % system$init()
           2 *   :  % argument$backup(ArgumentBuffer)
           3 *   :  % ArgCount = argument$argc()
           4 *   :  *if ArgCount > 0 then
           5 *   :  :  % file$changefcbtype(DefaultFCB,"SUB")
           6 *   :  :  *exec<  1>サブミット実行
           7 *   :  *else
           8 *   :  :  % sysout$writeln("Version 004 1985/8/15")
           9 *   :  *end-if
          10 *   :  % system$exit()
                 #end


section <<  1>>  #proc サブミット実行
          11 *   :  *exec<  2>サブミットファイルのオープン
          12 *   :  *loop サブミット
          13 *   :  :  % textfile$read(SubmitFile,TextLine) ; サブミットファイルから1行入力
          14 *   : <---*leave サブミット ( ax = EOF )
          15 *   :  :  *exec<  3>引数の置換
          16 *   :  :  % string$cuttail(TextLine,2)   ; 改行(CR,LF)記号をテキストから取る
          17 *   :  :  % sysout$write(TextLine)   ; コンソールに表示
          18 *   :  :  % system$exec(TextLine)   ; コマンドの実行
          19 *   :  :  *exec<  4>キーが押されたか検査
          20 *   :  *end-loop サブミット
                 #end
```

図 8 プログラム例: SUBMIT (ドキュメンテイタによる出力例: 部分)

Fig. 8 The part of the example of ADT-RASM86 program: SUBMIT printed by the documentator.

```
    string型            仕様書    (詳細)          86/05/09  11:45:02

  【note】
      文字列関係の処理  Ver. 1.0  Rev 017 (86/3/3 16:12)
  【history】
1985/8/21 Rev 005       number の追加(名前の無い修正は内田)
1985/8/22 Rev 006       setcr  の追加
1985/8/24 Rev 007       backindex の追加
1985/8/27 Rev 008       getexpression の追加
1985/8/28 Rev 009       getexpression のバグを取った
1985/8/30 Rev 010       takepart で範囲の指定に誤りがある場合、空の文字列を返すようにした
1985/8/31 Rev 011       insertchar の追加(西岡)
1985/9/8  Rev 012       sikpblank のバグ (ds 相対以外の時動作しない)を取った(木村)
1985/9/9  Rev 013       indexc のバグ (ds 相対以外の時動作しない) を取った
1985/9/11 Rev 014       gettokens のバグ (Null String を返す場合、非分離記号の文字列を
                        Null String にしていた) を取った (木村)
1986/3/1  Rev 015       skiptoken 追加
1986/3/1  Rev 016       skipword  追加
1986/3/3  Rev 017       appendc 追加

      操作名                意味
-------------------------------------------------------------------------------
    compare                 2つの文字列の比較
-------------------------------------------------------------------------------
    【note】
        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
-------------------------------------------------------------------------------
    【note】
        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。

-------------------------------------------------------------------------------
    backindex               文字列を逆順で探索する
-------------------------------------------------------------------------------
    【note】
        int string$backindex(str1,str2 [,start])
        string str1;    /* 探索対象の文字列 */
        string str2;    /* 探索文字列 */
        int    start;   /* 探索開始位置:この位置から前向きに探索を開始する */

        文字列 str1 の中に、文字列があるか探索し、その位置を返す。なければ0を返す。
        ただし、string$indexとは異なり、文字列の終りから探索を開始する。
        start が指定されている場合には、その位置から逆順に向って探索が開始される。

    【example】
        string$backindex("this is a pen.","is")    は 6
        string$backindex("this is a pen.","is",5)  は 3
        string$backindex("this is a pen.","is",3)  は 3
        string$backindex("this is a pen.","is",2)  は 0
    【remarks】
        start が文字列 str1 のない部分を指している場合、start は無視される。
    【history】
        R001    1985/8/24       初版
```

図 9 仕様書例: string 複合型パーケージ (SPEC による出力例: 部分)

Fig. 9 The part of the example of compound type package specification for users: string package by SPEC.

4.3.4 アドレス変換文

複合型パッケージの内部では，抽象データ型変数の対象物のアドレスを扱うことを認めている．ns 配置の対象物を指している抽象データ型変数は，RASM 86 固有の命令中では，そのまま 16 ビットのポインタとして扱うことができる．しかし，as 配置の対象物に対する抽象データ型変数は，32 ビットのポイントとして扱われる．そこで，>set 文で任意のレジスタにそのセグメント値とオフセット値を取り出し，>reset 文で，その逆の動作をすることができる．たとえば，

>set FileName es bx

は，FileName の対象物の配置されたセグメントの値を es レジスタに，オフセットを bx レジスタにセットする．

4.3.5 デバッグ文

変数のトレースのために >display 文が用意されている．たとえば，d1 を decimal 型の変数であるとすると，

>display "d1=", d1

によって，d1 の内容が指定された文字列 d1= とともに表示される．>display 文は，データ型属性を持つ変数が指定されると，その変数の内容を表示するために，複合型パッケージ名 $display データ型名という操作を呼び出す．上記の例では，decimal 型を定義している decimal という複合型パッケージに display-decimal という操作を用意しておく．この手続きを用意することは複合型パッケージの作成者の責任であり，これがないとリンク時にエラーとなる．

5. ADT-RASM 86 の処理系の概要

ADT-RASM 86 データ抽象化プリプロセッサは，1パス方式である．型情報管理表は，テキストファイルとして構成されている．複合型パッケージ名 $ 操作名は，そのまま外部識別子として使用する．RASM 86 は，識別子の長さを 255 文字まで許しているので，長さに関してはほとんど問題とならない．単一引用符で囲まれた文字は char 型の定数として，二重引用符で囲まれた文字列は string 型の定数として扱われる．char 型と string 型のみが，ADT-RASM 86 システムにあらかじめ組み込まれている型である．なお，ADT-RASM 86 には，*if 文や *loop 文などの構造化プログラミングの機能もあるが，これらは以前に作成した処理系[3],[12] により処理される．ADT-RASM 86 ドキュメンテイタは，ソースプログラムを2パス方式で編集しそのリストを出力する．**図 8** に，CP/M-86 でバッチ処理を行う SUBMIT コマンドを ADT-RASM 86 で作成したプログラムの編集リストを示す．SPEC は，ADT-RASM 86 の複合型パッケージソースプログラムを入力し，その仕様書を編集する．SPEC の出力例を**図 9** に示す．なお，**表 1** に ADT-RASM 86 処理系のオブジェクトサイズを示す．

表 1 ADT-RASM 86 システムのオブジェクトサイズ
Table 1 The object size of ADT-RASM 86 system.

ADT-RASM 86 データ抽象化プリプロセッサ	52 kbytes
ADT-RASM 86 ドキュメンテイタ	32 kbytes
ADT-RASM 86 SPEC	8 kbytes

6. アセンブリ言語におけるデータ抽象化の効用

6.1 システムコールのデータ抽象化によるプログラムの読解性の向上

一般に，8086 用オペレーティングシステムのシステムコールは，その機能番号を特定のレジスタに入れて，INT 命令を用いて呼び出す方式を採用しているので，分かりにくい．我々は，このシステムコールをデータ抽象化を用いて再構成した．たとえば，標準入力型，標準出力型，ファイル型，システム型 (os コマンドの実行など)，時刻型，グラフィックス型，外部デバイス型などである．また，C言語流の入出力を行いたいユーザのために，C複合型パッケージがあり，C$fopen, C$printf などがある．また，初心者にとって扱いにくい INT 命令によるシステムコールを関数呼び出しの形式にすることで，アセンブリ言語の教育が行いやすくなるなどの利点がある．半面，手続き呼び出しのオーバヘッドがあることがこの方式の欠点である．

6.2 ADT-RASM 86 におけるプログラムの再利用

複合型パッケージをソフトウェア部品とみなすことにより，プログラムの再利用を実現できる[11),14),15)]．そのために，ADT-RASM 86 では，複合型パッケージをソフトウェア部品として利用する際に必要な情報を，その複合型パッケージの作成者がソースプログラム中に格納する．その情報には，(1) 概要の説明，(2) 呼び出し法，(3) 開発の歴史，(4) バグ状況，(5) 使用例などがある．これらの情報は，SPEC により編集され，仕様書として出力される．現在の使用経験では，実用上，これらの情報により十分再利用が可能である．string 型や text 型のような標準組込み

表 2 ADT-RASM 86 の主な複合型モデルパッケージ
Table 2 The part of the list of the compound type packages prepared in ADT-RASM 86.

複合型パッケージ	定義されているデータ型	機能
file	FCB, DMA, iostat	ファイルの入出力に関する操作の集合
argument	argument	引数の管理に関する操作の集合
textfile	textfile	1行ごとのファイルの入出力の操作
system		os コマンドの実行などシステム的な操作群*1
sysout		標準出力に関する操作の集合
rs 232 c		rs 232 c コントローラ
mouse		マウスドライバ
avl	avl	avl 木の処理
graphic		グラフィック処理の操作の集合

*1 CP/M-86 のコマンドの実行に関しては文献 6) を基礎として用いた.

関数的な色彩の強い複合型パッケージだけでなく，qed 複合型パッケージ（行エディタである qed の各コマンドを実行する複合型パッケージ），popupmenu 複合型パッケージ（ポップアップメニューによる入出力）などの複合型パッケージも用意され，大きなレベルの部品化もなされている．表 2 に現在作成されているこのほかの主な複合型パッケージの一覧を示す．

7. ADT-RASM 86 のプログラム例とその評価

本章では，ADT-RASM 86 のプログラムを示し，ADT-RASM 86 を，（1）プログラム作成効率，（2）実行効率，（3）記憶効率の観点から評価する．

7.1 プログラム例 1：SUBMIT

図 8 は，CP/M-86 の SUBMIT コマンドを ADT-RASM 86 で記述したプログラム例の一部である．このプログラムには，RASM 86 のアセンブリ言語の文は，ほとんど現れず，複合型パッケージの操作の呼び出しの組合せを中心にしてプログラムが構成されている典型的な例である．ADT-RASM 86 の使用実績が進むにつれ，複合型パッケージの種類が増大するので，作成に要する時間を短縮でき，アセンブリ言語の文の比率が少なくなる．このプログラム（初版）の作成時間は，設計からデバッグ終了まで約 3 人時間であった．我々の経験では，このプログラムは，実行時ルーチンが整備されている場合でも少なく見積って，約 1 人日以上は掛かると思われるし，それがなければ，数人日は掛かると思われるので，期待した効果があると考えている．

7.2 プログラム例 2：grep・wc

7.2.1 実行効率

実行効率を調べるために，Unix の fgrep コマンドの簡易版（一致する文字列を持つ行の表示）と wc コマンドを用いる．ADT-RASM 86 によるプログラム（以下 PA と略す）とハンドコーディングによるアセンブリ言語のプログラム（以下 PR と略す）と 8086 上で動作する C コンパイラ (Computer Innovations 社の OPTIMIZING C86) によるプログラム（以下 PC と略す）の実行時間の比較を表 3 に示す．PR の実行時間を 1 とすると，PA は 1.1〜1.2，PC は 2.7〜2.9 程度となる．PR と PA の差は主に，操作呼び出しのコスト（引数渡し，スタックフレーム形成）である．PA と PC の差は，高級言語処理に必要なオーバヘッドと，C では 8086 のハードウェアの性能を引き出せるようにプログラミングできないからである．それは，文字列の扱いに現れている．ADT-RASM 86 では，string 型のデータ構造を 8086 の列操作命令を適用しやすいように設計し，高速に動作するようにプログラミングしている．これは，複合型パッケージをアセンブリ言語で記述できることの利点である．しかし，PA と PC の差のかなりの部分が C コンパイラの性能によることもまた事実であるが，現実に現存する C コンパイラの比較という実用上の意味から我々の実行効率に関する目標は達成できたと考えている．

表 3 grep と wc の実行時間の比較
Table 3 The comparison of the execution time of grep and wc program.

プログラム名	RASM 86	ADT-RASM 86	C
grep	8.6	8.8	23.7
wc	7.6	8.7	21.8

grep：フロッピーディスク上の C プログラム（約 1,000 ステップ）から printf のある行を検索した場合 (PC-9800E：8 MHz)
wc：約 3,000 文字のファイルのカウント
単位：秒（計測はストップウオッチで行った.）

表 4 gerp と wc のオブジェクトサイズの比較
Table 4 The comparison of the object size of grep and wc program.

プログラム名	セグメント	RASM 86	ADT-RASM 86[1]	ADT-RASM 86[2]	C
grep	コードセグメント	0.319	0.943	7.087	11.967
	データセグメント	0.671	0.959	1.039	64.000[3]
wc	コードセグメント	0.287	0.943	6.463	11.711
	データセグメント	0.287	0.863	0.943	64.000[3]

単位は，KB (キロバイト).
[1] ショートカットした場合.
[2] ショートカットしない場合.
[3] C (OPTIMIZING C86) のデータセグメントの寸法は，RASM 86 や ADT-RASM 86 とは異なり，スタック領域も含むため，64 kbytes の大きさになった.

7.2 記憶効率

(コードセグメントの) 記憶量に関しては，PR を 1 とすると，PA は約 22〜25，PC は約 37〜40 となってしまった (表 4). これは，主にリンカの性能による．すなわち，使用した LINK 86 には，複合型パッケージ内の必要な操作だけをリンクするという機能がないので，ある複合型パッケージの操作を一つでも呼び出すとその複合型パッケージの全操作をリンクしてしまうからである．複合型パッケージ内の必要な部分だけを選び出すようにリンカを改良することはかなり手間のかかる仕事であるので，我々は，ソースプログラムレベルで，呼び出されている操作とその操作で必要になる操作だけを切り出してリンクするショートカットプログラムを作成した．その結果，記憶量は PR の約 3 倍程度に減少した．記憶量がアセンブリ言語の 3 倍になった理由は，(1) データ抽象化の呼び出しオーバヘッド，(2) 部品となる複合型パッケージは汎用的に作られているので，特定の処理には不要な部分が各操作中に残ってしまう，(3) 対象物のヘッダ領域，などのためであると考えられ，本方式ではこれ以上の効率化は困難であると思われる．また，この例では問題とならないが，動的なメモリ割付けで記憶量を低減しているプログラムを，ADT-RASM 86 の静的な抽象データ型変数だけで作成するとさらに記憶量は増加すると思われるので，その場合にはヒープなどを扱う複合型パッケージを作成することが望ましい．

8. むすび

本稿では，データ抽象化がアセンブリ言語にも十分適用可能なことを ADT-RASM 86 の実現例から示した．また，その処理系もかなり小規模に構成できることを示した．アセンブリ言語の高速性・記憶効率の良さという特徴をそれほど犠牲にせずに，プログラミング効率を上げることができたことに価値があると考えている．最近は，マイクロコンピュータ上に様々な種類のソフトウェアを短期間に作成しなければならなくなっている．ADT-RASM 86 の複合型パッケージは，以前に作成した複合型パッケージをソフトウェア部品化するという意味でも有効である．また，制御系プログラム作成などでは，割込みやポート入出力などもデータ抽象化の対象にすることが可能なので，制御対象となるハードウェアを容易に複合型パッケージでモデル化できる．デバッグに関しては，現在では，>display 文によるトレース，あるいは，ヘッダによる実行時チェックなどで対処している．アセンブリ言語では，全アドレス空間が可視なので，情報を完全に隠蔽することはできないが，このヘッダにより対象物の範囲を越えたデータアクセスを未然に防ぐことができる．また，各操作の出入口で適当なチェックを対象物に対し行えば，プログラムミスの多くをデータの意味の正当性の観点から事前に検出できる．たとえば，ファイル型であれば，オープンされていないとか，帳票型であれば，重要事項が記載されていないなどの高度なレベルのエラーチェックが可能となる．これらは，その代償として，記憶量の増加や実行速度の低下をもたらしてしまうので，将来的にはプリアセンブル時のオプション指定で動作の有無を選択できるようにする予定である．しかし，さらにデバッグを効率的にするには，抽象化された型の変数に対するシンボリックデバッガが必要になるであろう．

近年，グラフィックスなどの特殊目的用 CPU が多数開発されているが，このような多種多様な CPU の上に C 言語などの最適化された高級言語コンパイラを構築するコストは少なくない．ADT-RASM 86 で採用した実現方法は，長い識別子を許すリンカが存在するアセンブリ処理システムの上であれば適用可能であ

り，十分安価で汎用性の高いものである．したがって，新しいCPUに対して，有効なプログラミング環境を少ないコストで構築することができる．我々は，アセンブリ言語の持つ原始的な側面と高級言語の持つ高度な側面を無理なく融合できたと考えている．しかし，この方法では，扱う複合型パッケージの個数の増大が将来的には問題となる．そこで，オブジェクト指向のインヘリタンス機能を利用して，扱う複合型パッケージの個数を減らす計画を進めている．本システムの実現例が，様々なアセンブリ言語環境の効率化に役立てば幸いである．

謝辞 最後に，本プロジェクトに協力してくれた，木村英一，西岡淑光，余越誠の諸氏に感謝の意を表したい．

参考文献

1) CP/M-86 System Reference Guide, Digital Research Inc. (1980).
2) Programmer's Utilities Guide for the CP/M-86 Family of Operating Systems, Digital Research Inc. (1983).
3) 榎田 幸，内田智史，佐藤 衛，間野浩太郎：マイクロコンピュータソフトウェア開発のための高級アセンブリ言語とデバッギング支援ツール，第26回情報処理学会全国大会論文集，3N-1 (1983).
4) Fukazawa, Y.: Abstraction Mechanisms Supported by a Macro Processor, *J. Inf. Process.*, Vol. 6, No. 2, pp. 59-65 (1983).
5) Gehani, N.: *Ada, An Advanced Introduction*, Prentice-hall, Englewood Cliffs, NJ (1983).
6) Ida, M.: A Design and Implementation of Personal Lisp Systems Considering IE, *Proc. of the 7th Annual Conf. on Computers and IE*, pp. 5-9 (also in *J. Computers and IE.*, Vol. 9, Suppl., pp. 5-9) (1985).
7) Isner, J. F.: A Fortran Programming Methodology Based on Data Abstraction, *Comm. ACM*, Vol. 25, No. 10, pp. 686-697 (1982).
8) Liskov, B., Snyder, A., Atkinson, R. and Schaffert, G.: Abstraction Mechanism in CLU, *Comm. ACM*, Vol. 20, No. 8, pp. 564-576 (1977).
9) 佐渡一広，米沢明憲：抽象データ型言語，情報処理，Vol. 22, No. 6, pp. 525-530 (1981).
10) Stroustrup, B.: Classes: An Abstract Data Type Facility for the C Language, *Sigplan Notices*, Vol. 17, No. 1, pp. 42-51 (1982).
11) 内田智史，間野浩太郎：データ型パッケージによるソフトウェアの部品化とそれを用いた再利用，第28回情報処理学会全国大会論文集，4J-8 (1984).
12) 内田智史，榎田 幸，戸浪 聡，間野浩太郎：CP/M-86 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).
13) 内田智史，間野浩太郎：CP/M-86 アセンブリ言語へのデータ抽象化機能の導入，情報処理学会マイクロコンピュータ研究会資料，36-2 (1985).
14) 内田智史，間野浩太郎：マイクロコンピュータのアセンブリ言語へのデータ抽象化の導入による再利用環境の構築，情報処理学会 夏のシンポジウム『プログラムの合成，変換，再利用』(1985).
15) 内田智史，間野浩太郎：データ抽象化に基づく事務処理プログラミング方法論に関する研究，経営工学会昭和60年度秋季研究大会予稿集，p. 132 (1985).

(昭和61年5月15日受付)
(昭和62年9月9日採録)

内田 智史（正会員）

昭和33年生．昭和57年青山学院大学理工学部経営工学科卒業．昭和59年同大学大学院修士課程修了．昭和62年同大学大学院博士後期課程単位取得済退学．同年青山学院大学理工学部経営工学科助手，現在に至る．プログラム言語，プログラミング方法論に興味を持つ．日本経営工学会，ソフトウェア科学会，人工知能学会各会員．

間野浩太郎（正会員）

大正10年生．昭和19年9月東京大学理学部物理学科卒業．工学博士．鉄道技術研究所計算センター室長を経て，昭和44年より青山学院大学理工学部経営工学科教授．教育用ソフトウェアとプログラミング支援ツールの研究に従事．ACM，日本経営工学会，日本ソフトウェア科学会，人工知能学会各会員．