ソフトウェア工学 24－2
（１９８２．６．４）

ＰａｓｃａｌによるＰａｓｃａｌ用実行支援系の作成経験

東京工業大学総合情報処理センター
白浜律雄

概要

アセンブラで書かれていたＰａｓｃａｌの実行支援系をＰａｓｃａｌでかきなおした。その経験を報告する。ねらいは、言語処理系全体を一人で片手間に保守改良していける形に持ち込むことにある。実用品としての実行効率も重視している。

保守改良性ををあげるには、実現対象そのものを単純化することおよびその記述（原始プログラムと付随文書）を減少させることが重要である。

対象たる処理系をどう実現するか細部まで一度にはきめられないので、アイディアがでた時に簡単になおせる構造にすることが先決である。高級言語の利用は記述量の減少に有効である。とりわけ対象言語を使用したので、保守者の負担を軽減し、処理系の自由度を確保できた。

仕様内ではＰａｓｃａｌで記述できない処理や実行効率に重大な影響がある処理では、アセンブラももちいたが、使い方には強い制限を課した。

性能は十分であるが、保守性がねらいどおりに達成できたかはまだ作成者の主観的判断しかなされていない。ねらい、その背景、接近法、構成を述べる。

目次

１．　背景
　１．１　発端
　１．２　なぜ一人が片手間に？
２．　接近法
３．　Ｐａｓｃａｌを使用した利点
　３．１　選択した理由
　３．２　高級言語としての利点
　３．３　実行支援系の実行支援？
４．　弱点をいかに補うか？
　４．１　機能不足
　４．２　アセンブラを使う場合の規範
　４．３　大きさの増大にどう対処するか
５．　新版
　５．１　構成
　５．２　機能実現の階層間移動
　５．３　旧版との比較
６．　まとめ

1. 背景

1．1 発端

アセンブラでかかれていたＰａｓｃａｌの実行支援系を、Ｐａｓｃａｌでかきなおした。

従来の処理系（ＡＡＥＣ版と呼ぶ。新しい処理系をＴＩＴ版と呼ぶ）でも、翻訳系はＰａｓｃａｌで記述されていたので、改良によって現在ではかなり小さく見通しの良いものになっている（５３００行）。ところが、実行支援系はアセンブラで書かれ、４４００行もある。処理系の保守に専念しているわけではないので、重大なバグが発見されると取り除くぐらいしかできなかった。内部説明書がなかったことも一つの原因であるが、多少読んで見ると絡み合いがひどいことが判明し、これ自体を改良するのは労力の無駄と判断した。

一方では、処理系改良の必要性はたかまっていた。ＡＡＥＣ版もただ使うだけならおおむね良くできているが、道具を作るための道具として日々使用していれば、不備が目につき、不満は溜る。例えば、モジュール化を促進する機能、漢字の取り扱い等である。ある言語を実際に計算機上で使用していく上では、その言語の設計もさることながら、処理系（特に実行支援系）のでき具合も重大な要因である。良くできたものでも、保守改良をおこたれば陳腐化する例といえよう。

また、翻訳系の洗練、目的コードの改良でも、実行支援系の不可触性が壁になっていた。

1．2 なぜ一人が片手間に？

この処理系全体を一人が片手間に保守改良できる形にもちこもうとしている。いいＰａｓｃａｌの処理系を望む声がいまだにあるからである。国産機メーカはいい処理系を供給していないようである。

一方、大学等においては、Ｐａｓｃａｌ処理系の開発保守に一人が専念する時代はおわっている。ただ実用品として使いたいわけである。

もし一人が片手間に面倒みられれば、特定の者を保守に割り当てるのでなく、したい者が誰でも処理系を個性化したり、新機能の実験台にしたりできるようになる。メーカのお仕着せをただ使うよりいい。

2. 接近法

接近方針は「読んで理解しなければならない量を減らす」という常識的なものである。それには、記述の絶対量を減らすことと、記述される内容を理解しやすくすることが大事である。

高級言語を利用すれば、プログラムの記述絶対量（行数、文字数）がへり、プログラムそのものが説明としての価値をもってくることは良くしられている。

しかし、何語でかこうと複雑な処理は複雑なのであるから、処理内容を単純化することも大事である。翻訳系（目的コード）と実行支援系の機能分担、実行環境の維持・操作方法、目的コードと実行支援系との界面等をすっきりさせねばならない。これらを一度に全部設計することはできないので、まずＡＡＥＣ版における目的コード－実行支援系間界面はそのままにして作成し、次にその後の作業がしやすいように構成をかえ、少しずつ改良していくことにした。

3. Ｐａｓｃａｌを使用した利点

3．1 選択した理由

実際には、作成用高級言語として対象言語であるＰａｓｃａｌを選択した。他に適当な言語がなかったという消極的な理由ではなく、Ｐａｓｃａｌ処理系中に記述の面でも、動作の面でも、他言語を排除するという積極的理由があった。

他言語の使用は、保守者がしらねばならないことをふやすことになる。また、その言語の実行支援系がなんらかの制限をくわえ、Ｐａｓｃａｌ用実行支援系としての自由度・透明度を下げることもありうる。異物の混入は複雑さをますだけである。

3．2 高級言語としての利点

Ｐａｓｃａｌという言語の一般的な利点（データ構造、制御構造、翻訳時検査）や高級言語全般が持つ利点ははそのまま享受している。例えば、実行支援系が参照する目的コード中のオブジェクトを洗い出して、データ型を定義してある。目的コードから渡ってきたパラメータを厳格に検査するのも、実行支援系しか操作しないデータの一貫性を検査するｎも手をぬかずに書いていける。エラーを検出した場合の関連情報提供も利用者がみやすい形式を優先できる。また、スナップ・ショットを気楽にうてるのはデバッグにやくだった。

最大の利点は変更の容易さである。ＯＳ（オペレーティング・システム）の不備を補うために、無理な処理をしていたのをやめたり、自然な方法にあらためたりもした。目的コードから実行支援系への引数界面もアセンブラ向きの方法からＰａｓｃａｌ向きにあらためた。このような変更を翻訳系と同時に行なった。

多少扱いにくかったのは、可変長の文字列である。目的コードからわたされる文字列の長さは予測できないが、Ｐａｓｃａｌには可変長配列を扱う機能が全くないので、事実上無限といえる長さの文字列へのポインタで参照する。

3．3 実行支援系の実行支援？

他の高級言語を使用していたならば、その言語の実行支援系が必要であっただろう。では、対象言語で書いたならばどうか？

目的プログラムの面倒を見る実行支援系をＲＳ０とする。ＲＳ０がＰａｓｃａｌの機能を利用できないのならば、Ｐａｓｃａｌで書く利点が少なくなる。例えば、エラーを検出した後のメッセージ出力をｗｒｉｔｅでかけるかかけないかは大違いである。しかしそのためにさらに実行支援系（ＲＳ１）がいるならば面倒である。ＲＳｉにＲＳｉ＋１という連鎖を無限に続けるわけにはいかないので、どこかであきらめてＰａｓｃａｌの機能をつかえなくするしかない。あるレベルでどの機能が使えるかを意識して書くのは混乱の元になる。メモリ占有量の点でも不利である。

実際には、ＲＳ０の世話はＲＳ０自身が見るようにしてこの問題を回避している。つまりＲＳ０内で実行支援系の助けが必要な機能を使うと、ＲＳ０自身がよびだされるようにした。ＲＳ０はユーザ・プログラムを世話しているのか、自分自身の世話をしているのか区別しない。従って実行支援系内でもＰａｓｃａｌの全機能を利用できる。

4．　弱点をいかに補うか？

4．1　機能不足

Ｐａｓｃａｌを選択すると、Ｐａｓｃａｌではかけない機能（おもにＯＳへのサービス依頼）をどう実現するかが問題になる。

翻訳系への機能追加はしなかった。既に十分大きい翻訳系に実行支援系しか使用しない機能をとりこんで、さらに膨張させるのはさけたかった。もともとＯＳ依存のコードを生成しない翻訳系をそのままにしておきたくもあった。ＯＳの呼び出し列が一様かつ短かいものであれば入れたかもしれないが、そうであれば実行支援系はもともと不要なのである。

そこで、アセンブラをもちいた。言語処理系を保守するものが、アセンブラにうといはずはない。速度や動作記述の直接さ等アセンブラのいい所をいかし、悪い面がでないように制限して使う。アセンブラを徹底的に排除するのが目的ではなく、実用品としての効率をたもったまま、全体の記述量を下げるのが目的なのである。

4．2　アセンブラを使う場合の規範

複雑な処理、頻繁によびだされないので遅くてもいい処理、ていねいで確実な操作が必要な処理等がＰａｓｃａｌで記述されているが、やむをえずアセンブラももちいた。よみにくくならないように、ある機能をアセンブラで実現する場合には以下のような規範にしたがった。

（１）Ｐａｓｃａｌではかけないことに限る。ＯＳ呼び出しまたは非常に高速な処理である。

（２）短かいコードに限る。入ってから出るまでたかだか２０命令程度である。

（３）繰り返しやサブルーチン呼び出しはしない。制御の合流も避ける。誤りが混入しやすくコードを追跡しにくくなるからである。

（４）概念の整理をして、マクロ命令を定義する。（従って、みかけのステップ数は上記よりさらに短かい）。

この制限内であれば、ある言語機能を翻訳系でのコード生成として間接的に記述するよりも、アセンブラで書く方が直接的で良い場合すらある。ＡＡＥＣ版にくらべると、単に行数が１／４になった量の変化より、質の変化が保守性向上に貢献している。

4．3　大きさの増大にどう対処するか

高級言語を使用すると目的コードが増える心配がある。アセンブラで書かれたＡＡＥＣ版の実行支援系は１５ＫＢあった。この大きさでも、結合されてすぐに実行可能な状態でいくつかのプログラムを保存しようとするとディスク容量が無視できなくなる。

Ｐａｓｃａｌの目的コードは再入可能（リエントラント）である。アセンブラ部分は小さいので、これを再入可能に作るのは容易である。それによって、実行支援系全体がメモリ常駐可能になる。ユーザは翻訳結果だけを保存すれば良く、むしろディスク占有量を減らすことができる。

さらに、ユーザの書いた部分もふくめてＰａｓｃａｌプログラム全体を常駐させることもできるので、多数の利用者があるユーティリティ（例えばエディタ）に適用すればシステム性能（レスポンス、スループット）を向上させられる。

5. 新版

5．1 構成

実行中のプログラム階層を簡単に説明する（図1）。矢印は呼び出しをあらわす。

目的コードが実行支援系の助けを必要とする時ＲＳＡ（アセンブラで書かれている）を呼ぶ。ＲＳＡには、約６０の入口があり、分岐表になっている。目的プログラムから実行支援系への入口はここにあるだけである。ＲＳＡはユーザの目的コードからよばれたのかＲＳＰからよばれたのか区別しない。高速実行を要する機能はＲＳＡ内で全ての処理をおえてかえる。他のものはＲＳＰ（Ｐａｓｃａｌで記述）を呼ぶ。ＲＳＡで処理中に異常を検出した場合も、後仕末はＲＳＰにまかせる。目的コード（ＲＳＰ）からの呼び出し列は一様で、ＲＳＰを呼ぶための界面整合はＲＳＡが行なう。

ＲＳＰの大部分は入出力の支援であり、必要に応じてＯＳを呼ぶが、直接にはできないので、ＯＳＣを経由する。ＯＳＣはアセンブラで書かれた機能対応のルーチンの集まりである。ＯＳへのパラメータ設定はＲＳＰが行なうのでＯＳＣはトンネルにすぎない。

ＲＳＡのコードの各々はよんだ手続きの環境内で実行する。レジスタの退避回復は各々が必要最低限（全くしないことも多い）しかしないので、呼び出しの手間は小さく、非常に高速の実行が可能である。

スタックは１つだけで目的プログラムとＲＳＰが共有する。メモリ管理は単純である。

5．2 機能実現の階層間移動

ある機能の実現を、ある階層から他へ移行することが容易な構成になっている。実現手段は３種類ある。

（１）目的コードが実行する（翻訳系が展開する）

（２）アセンブラで書く（ＲＳＡに入れる）

（３）Ｐａｓｃａｌで書く（ＲＳＰに入れる）

ＲＳＡ中のコードの呼び出しの手間は非常に小さいので、展開するのをやめてＲＳＡにうつしても実行速度にあまり影響しない。これは目的コードの縮小、翻訳系の単純化につながる。実行支援系の機能として関連するものをまとめて移行して管理を統合し、見通しを良くした。実行支援系と目的コードに分散していたスタック・ヒープ操作を実行支援系によせたのがその例である。

このＯＳ（ＨＩＴＡＣ　ＶＯＳ３）では、入出力は煩雑なので、基本的にはＰａｓｃａｌで処理する。頻繁によびだされてその手間が問題になる機能（例えば文字の入出力）では、とりあえずＰａｓｃａｌで書いて確認し、それはそのままにして、簡単に処理できる場合だけはＲＳＡで実行してしまうような追加（２０命令）をした。さらにそれもそのままにして、目的コード中に展開することも翻訳系の変更だけでできる。

決定には、保守性、実行効率、目的にプログラムの大きさ等を総合的判断しなければならない。

5．3　旧版との比較

機能が一致しないので、直接比較にはなじまないが、ＴＩＴ版とＡＡＥＣ版の比較を表１に示す。時間はＨＩＴＡＣ　Ｍ２００－Ｈで測定した。

	ＴＩＴ	ＡＡＥＣ	
原始プログラム			
アセンブラ	１０００	４４００	行
Ｐａｓｃａｌ	１３００	０	行
メモリ占有量	２４	１５	ＫＢ
文字のｒｅａｄ	０．５４	０．８３	秒（１２０ＫＢのファイル）

表１　　ＴＩＴ版とＡＡＥＣ版の比較

6．　まとめ

実行支援系の大部分をＰａｓｃａｌで記述して保守性を上げ、一方ではアセンブラも強い制限の下で使って実行効率もあげた混合戦略による実行支援系を紹介した。翻訳系と歩調を併せて変更を重ね、実行環境の操作・維持、目的コードと実行支援系との界面等をすっきりさせ、ことさらに説明しなければならないことを削減した。翻訳系もより単純になった。現時点で処理系が細部まで最適な形状になっているとはいえないが、その方向にすこしずつよせていくことが可能な構成になっている。

従来の全部がアセンブラでかかれていたものより速く、実用品として使える速度を実現できたことは確認されたが、一人が片手間でもできる保守性については、行数が減ったこと以外にまだ作成者の主観的判断しかない。

ハードウェアやＯＳの不備をいかにスマートにおぎなったかをおみせしたわけであるが、大局的には、その問題の根元を改善しなければならないことを注意したい。

最後に：翻訳系の改造を担当された前野助教授（東京工業大学）に感謝します。

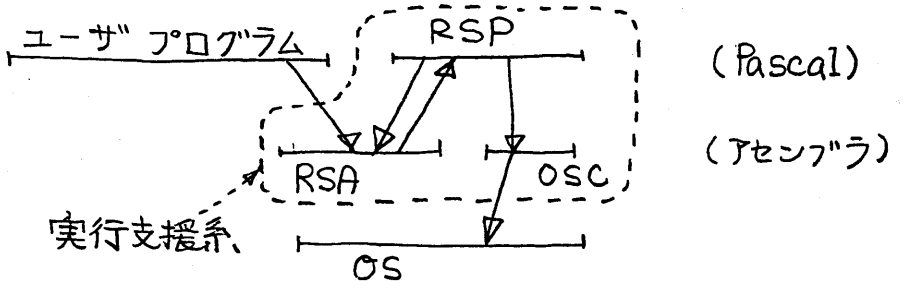

図１　実行中のプログラム階層