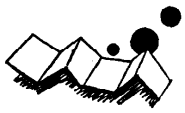
解　説



Ada の処理系†

近　山　　隆††

1. はじめに

Adaには，在来の言語に見られなかった機能が豊富に取り入れられている．しかしながら，その処理系も在来の言語処理系に比して複雑・巨大化し，作成には困難が伴うであろう，とする予想は必ずしも正しくない．Adaの新奇に見える諸機能のうちには，在来の言語の機能を統一・系統化し，概念としてはむしろ簡素になったものが少なくない．また，真に新たな機能に関しても，導入にあたっては常に処理系の作成が念頭に置かれている．このため，文法の記述が不十分または不適当と思われるいくつかの部分を除いては，本質的な処理系作成上の困難は皆無といってよいだろう．

全体を通じて，ひとつの式の中など以外は，単一パスでの翻訳が可能な設計となっている．これは本来は処理系のためより，むしろ読譜者の便宜を計ったものであろう．しかし，単一パスにすべきか否かはともかく，この性質のために処理系の負担が大幅に軽減されているのは事実である．

次章以下に，処理系作成上の技術的問題点と解決の手法のいくつかと，処理系の実例を紹介する．

2. 多義の解決

多義 (overloading) 機能は，従来の言語にもあった機能の統一化である．演算子 "+" が，整数値の被演算子に対しては固定小数点加算，実数値に対しては浮動小数点加算を表わし，どちらを用いるべきかを文脈から判定する手法は，従来から普通に用いられてきた．この特殊性のため，従来の処理系ではこの部分を特別扱いし，ぎこちない記述になっていることが少なくない．

演算子を関数の一種と考え，作譜者定義可能とすることは，ごく自然な拡張である．その演算子に多義を許すなら，一般に関数名，さらには手続き名にも許すのが当然の帰結だろう．一般化した多義機能のため，演算子の適用の処理は，副譜呼び出しとまったく同様に行える．このために処理系はいくぶん大きくはなるが，その構造はかえって簡素に統一できる．

多義な式の翻訳に原理的困難は何もない．式中の多義な識別子・文字リテラル・演算子のすべてについて可能な組み合わせを調べ尽し，可能な解釈が唯一存在すれば，その解釈が正しい解釈である．複数の解釈が成立する場合には，式はあいまいで，算譜の誤りである．

可視 (visible) な識別子の数は，譜包 (package) の機能を十分に活用すれば，あまり多くはなるまい．多義の解決はひとつの式などの中で行われるので，単純な算法を用いてもさほど支障はない．しかし，多義解決の高速な算法の研究もさかんである．

多義機能を効率よく実現するためには，適切な名前表の形式が重要である．識別子等が与えられた際に，即座にそれが指し示す可能性のある実体のリストが得られるような形式が望ましい．これには識別子を鍵とするハッシュ表が最適である．ハッシュ表の内容は，その時点で直接可視 (directly visible) な実体のリストとする．

多義な集成式 (aggregate) の解決は最大の難点である．名前による集成式 (named aggregate) に対して，**図-1** のような困難が生ずる．(1) では，レコード型Rの名は直接可視ではない．しかし，R型の有効範囲ではあるので，R型の集成式を書くことはでき，その中ではR型の要素名は直接可視になる．(1)でSの引数はこのR型の集成式である．一方，(2)のSの引数はA型の集成式である．このように，名前による集成式の中の選択子 (choice) のところでの可視性は，いわば選択子自身によって開かれてしまうのである．

この問題を効率よく解決するためには，すべての有効なレコード型の要素名に対して，直接可視であるかのいかんにかかわらず，それが選択子たり得ることを

† Implementation of Ada by Takashi CHIKAYAMA (Dept. of Math. Eng. and Instr. Phys., Fac. of Eng., Univ. of Tokyo).

†† 東京大学工学部計数工学科

```
declare
   package P is
      type R is
         record
            I : INTEGER;
            X : FLOAT;
         end record;
   end P;
   type A is
      array (1..2) of INTEGER;
   procedure S (F : P.R);
   procedure S (F : A);
      ------------
   I : constant := 1;
   X : constant := 2;
begin
   S ((I => 3, X => 3.0));   -- (1)
   S ((I => 3, X => 3));     -- (2)
end;
```

図-1 集成式内の可視性

知るための別のリストを名前表中に保持すればよい．しかしこの問題については，むしろ文法を改訂して集成式の型が外部の文脈から一意に決められるよう義務づけた方が得策であろう．

3. 例外処理

例外が生じたときには対応する例外処理譜が呼ばれるが，ある時点で呼ぶべき例外処理譜は動的に決定される．図-2 で Q は例外 X を生じるが，この例外を処理するのは，Qが（2）でRから呼ばれたときには(3)，(4)で直接Pから呼ばれたときには（5）になる．

これを実現するには，例外処理譜をもつブロックの開始の際にそこで処理すべき例外名をスタック上に登録し，例外発生時にスタック上を探す方法（図-3 左）と，動的に最も内側の例外処理譜の位置を固定場所に覚えておき，ブロックの出入りの際にこれを更新する方法(図-3 右)が考えられる．これらはいずれもLISPのような動的スコープを持つ言語の変数結合に用いられる手法である．後者は例外発生時の処理速度の点ではるかに前者に優るが，ブロックの出入りの際の手間がやや多い．例外処理が行われる頻度が両者の優劣を決めることになる．

例外はその有効範囲外でも伝播されねばならない．図-4 で P→Q→P→Q の順で呼ばれたとき，（2）で発生した例外 X は（3）で処理され，（4）でふたたび例外 X を生じる．この例外は（1）で宣言されたもので，Pの例外処理譜（5）の X とは異なり，ここでは有効範囲外であるが，この部分も通過してひとつ外側の（3）で処理されることになる．

```
procedure P is
   X, Y : exception;
   procedure Q is
   begin
      raise X;          -- (1)
   end Q;
   procedure R is
   begin
      Q;                -- (2)
   exception
      when X | Y =>     -- (3)
        ---
   end R;
begin
   Q;                   -- (4)
exception
   when X =>            -- (5)
     ---
end P;
```

図-2 例外処理譜の対応

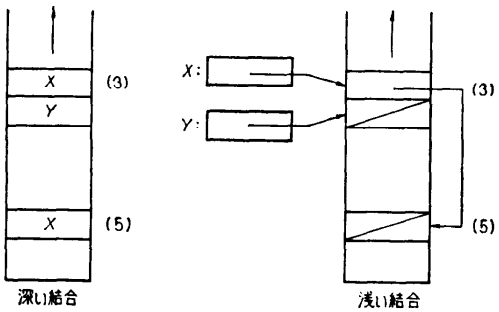


図-3 例外処理の実現
(P⇒R⇒Q と呼んだ時)

このような例を正しく処理するためには，例外名を一意な名前（一連番号等）につけなおす必要がある．譜庫単体 (library unit) を用いる場合，例外の伝播は翻訳単体 (compilation unit) をまたがって行われる．したがって，一意名へのつけなおしは，翻訳時だけでなく，結合編集時にも行う必要がある．

4. 譜庫機能と分割翻訳

譜庫機能を実現するためには，翻訳時には翻訳すべき単体と共に譜庫帖 (library file) を処理系に渡す必要がある（図-5）．

```
procedure P is
   exception X;
   procedure Q is
      exception X;        -- (1)
   begin
      ---
      P;
      ---
      raise X;            -- (2)
      ---
   exception
      when X =>           -- (3)
         ---
         raise X;         -- (4)
   end Q;
begin
   ---
   Q;
   ---
exception
   when X =>              -- (5)
      ---
end P;
```

図-4 例外の伝播

翻訳単体は Ada で書かれたひとつの譜包または副譜の宣言・本体に，参照する標準以外の譜庫単体名を記した文脈指定 (context specification) を付加したものである．譜庫単体を用いるために必要な情報は，すでに譜庫帖に格納されていなければならない．処理系はこの情報を用いて名前表を初期化し，与えられた単体を翻訳する．その結果，リスティング等を出力し，目的譜と今回翻訳した単体に関する情報を譜庫帖に格納する．

分割翻訳を指定された本体控え (body stub) が現われたときは，その文脈での名前表の状況も譜庫帖に格納しておく．後に対応する副体 (subunit) を翻訳するときには，この名前表を初期化に用いることにより，翻訳の行われるべき文脈を再現する．

翻訳・再翻訳の順序には規則がある．ある単体の翻訳の前に，それが用いる譜庫単体の宣言はすべて翻訳済みでなければならない．宣言さえ翻訳済みならば，その本体は後で翻訳してもよい．ある単体の副体は，その単体の翻訳の後でしか翻訳できない．また，一般に本体の翻訳は，対応する宣言より後で行う．

この規則に従う限りは翻訳順序は自由である．実際の順序が規則に合致するかどうかを処理系が確かめるために，翻訳順序・単体の依存関係に関する情報も譜庫帖に格納される．

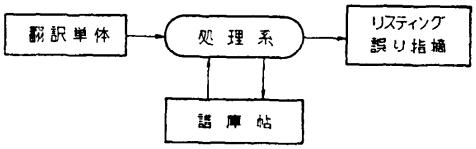


図-5 処理系の入出力

実行にあたっては，実行すべき単体の目的譜を譜庫帖から取り出して実行するわけだが，その前にこの単体が直接・間接に用いる譜庫単体をすべて結合編集する必要がある．この際，再翻訳の順序等が規則通りであるかを検査したり，例外に一意名を与えたりするため，専用の結合編集系が必要である．

処理系は，翻訳機能のほかに，譜庫帖の新規作成，状態の表示，ほかの譜庫帖からの翻訳済み単体の取り入れ等の機能を有すべきである．処理系全体の使い勝手は，これらの機能の優劣で大きく左右されるであろう．

5. 処理系の実例

現在各国で多くの Ada 処理系の作成努力が行われているが，まだ完全な処理系は完成をみていない．ここでは西独の Karlsruhe 大学と東京大学のふたつの処理系の概略を紹介する．

5.1 Karlsruhe 大学の処理系[1]

処理系は Siemens 7.760 上に作成中で，構成の大略は図-6 の通りである．解析部と合成部は完全に独立で，合成部のみの交換で移植が可能になっている．

ここに用いられている中間言語 AIDA (An Intermediate Description of Ada) は，構文木に意味情報を付加した形式で，機械独立である．副産物として，合成部を置き換えた Ada 整書系も作られた．

処理系の作成にあたり，まず処理系記述用に Ada-0 を設計し，その処理系を作った．Ada-0 は Ada 処理系記述に必要な機能のみを残した Ada の部分言語で，Ada-0 の算譜は Ada の文法にも合致する．

最初の処理系は LIS[2]で記述され，AIDA を出力する解析部と，LIS を出力する合成部からなる．この出力を LIS 処理系に入力して機械語を得る．LIS による目的譜は Ada 原譜の識別子をそのまま残したもので，Ada-0 処理系の負担は軽い．Ada-0 処理系は約 200K バイトで毎秒 250 行程度，LIS 処理系は約 1 M バイトで毎秒 25 行程度の大きさ，処理能力である．

次の段階は Ada-0 による Ada 処理系の解析部の

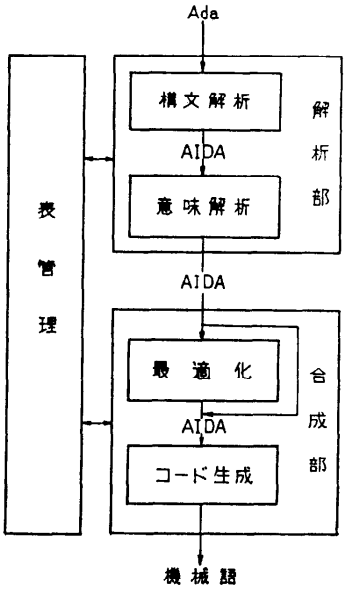


図-6 Karlsruhe 大学の処理系

記述である．構文解析には表駆動の LALR（1）解析を用い，このための表は自動作成された．意味解析部は Ada 自身で記述された Ada の形式定義[3]に基づいて作成された．多義解決には独自の 2 パスの算法を用いている．解析部の速度は毎秒 50 行程度になった．

処理系は当初 Preliminary Ada 用に作成されたが，最終版用に修正を加えて，81 年秋の完成を予定している．

5.2 東京大学の処理系[4]

この処理系は Hitac 8800 の上に作られた Preliminary Ada の処理系である．全体の構成は図-7 に示す通りで，構文解析部のみが Pascal 8000[5]，ほかは HLISP[6]で記述されている．

二種の木構造の中間言語を用いているが，特に中間言語（2）は機械独立で，コード生成部の書き替えによる移植性を考慮したものである．目的コードは HLISP である．この処理系は作成を通じて Preliminary Ada の理解を深め，評価を行うことを目的としており，実用に供する意図はないので，機械語を生成する煩雑さを避けたのである．

処理系の大きさは，Pascal の部分が約 2200 行，HLISP の部分が約 3000 行である．HLISP の部分が小さいのは，LISP の特質によるものでもあるが，マクロを多用したのが大きな理由である．本格的処理系も，この数倍程度の大きさで作成可能と予想される．

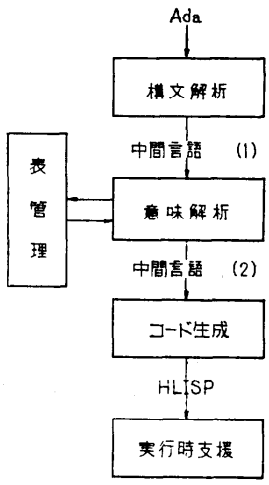


図-7 東京大学の処理系

翻訳速度は毎秒 30〜40 行程度で，主たる計算時間は意味解析に費やされる．多義の解決には単純な算法を用いており，名前表の構成もよく吟味されておらず，しかも HLISP のインタプリタ上での速度なので，この百倍程度までの改善の余地はあるだろう．

最適化はまったく行っておらず，多くの検査コードを実行するので，実行速度は直接に HLISP で記述した場合の 5〜10 分の 1 程度になった．

これらふたつの処理系は，まったく独立に開発されたものであるが，よく似かよった構成をとっている．特に，木構造の中間形式を用いることによって，機械依存部を切り離している点が注目される．Ada では従来の言語に比して解析部の仕事量が多いので，今後の処理系でもこの手法が用いられることになろう．

両者に共通する難点として，誤り指摘のもどかしさがあげられる．式があいまいであるとき，あいまいさは式全体の相互作用により生ずるので，どこがあいまいかを示すことは難しい．可能な解釈の詳細を示す方法は考えられるが，作譜者に理解しやすいかは疑問である．この点は今後の Ada 処理系の最大の研究課題となるだろう．

6. おわりに

Ada 処理系の作成には，原理的な技術的困難は少ない．むしろ課題は，人間とのインタフェースを良くし，使い勝手のよいシステムを作り上げることにある．Ada という言語は，このようなシステムを築き上げるための土台として，従来のどの汎用言語よりも優れて

いると考えられる．Adaに見られる概念の統一・系統化を反映した．簡潔で明快な処理系が期待される．

参 考 文 献

1) Daussmann, M. et al.: The Ada Compiler Development Project - Overview, Universität Karlsruhe (July 1980).
2) LIS Reference Manual, Siemens A. R., München (1978).
3) The Green Language, a Formal Definition, Cii Honeywell Bull, Louviciennes, France (1979).
4) 近山：プログラミング言語 ADA 処理系の試作，第 21 回プログラミング・シンポジウム報告集 (1980).
5) Hikita, T. and Ishihata, K.: PASCAL 8000 Reference Manual, Tech. Rept. 76-02, Dept. of Inf. Sci., Fac. of Sci., Univ. of Tokyo (1976).
6) Kanada, Y.: HLISP and Supplementary H-LISP-REDUCE Manual, Dept. of Inf. Sci., Fac. of Sci., Univ. of Tokyo (1979).

(昭和 55 年 11 月 10 日受付)