解　説

コンパイラ自動生成系による Ada 構文検査系作成†

徳田雄洋†† 吉田裕之†† 井上謙蔵††

1. はじめに

本稿は，Ada 処理系の一部をコンパイラ自動生成系を用いて作成する試みから，若干の話題を選んで報告するものである．もとより Ada 処理系作成上の問題全般について解説を試みることは時期尚早と考えられるので，本稿ではコンパイラ自動生成系を用いて 1979 年 10 月から 1980 年 3 月にかけて行った構文検査系作成実験[19]の範囲から，作業の概要と作業中にある程度詳細に検討した問題点のうち 2 つを選んで報告することとしたい．

本実験で使用したコンパイラ自動生成系は，本稿の第 3 著者の 1977 年の提案[10]に基づいて佐々政孝，徳田淳子他により開発されたもの[14]で，1978 年から使用に供されている．

本実験は 1979 年 10 月にスタートしたため，1980 年 3 月までの作業は Preliminary Ada（本稿では以下 1979 年 6 月版 Ada と呼ぶ）に基づいている．しかしながら，問題点の検討作業は 1980 年 4 月以降も継続して行っており，本稿における技術的検討はすべて 1980 年 7 月版 Ada 用に更新してある．

本稿では，Ada[3]（以下特に断わらない限り 1980 年 7 月版Ada のこと）または 1979 年 6 月版 Ada[4),5)]に関する一般的知識を仮定しているが，これらについてはたとえば文献[11),12),16),18),20)]を参照されたい．なお，コンパイラ作成に関する基本的技法については文献[1),2),9)]を参照されたい．

本稿の構成は次の通りである．第 2 章では自動生成系による作成の概要を紹介する．第 3 章では効率が良いと思われる名前表の構造を検討する．第 4 章では従来の言語より複雑となった演算子判定の問題を検討する．第 5 章ではまとめを述べる．

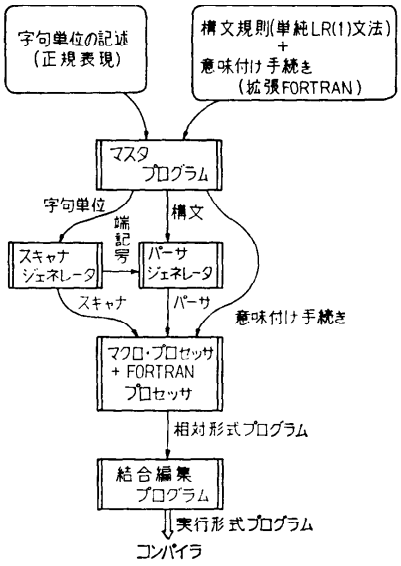


図-1　コンパイラ自動生成系の構成

2. 自動生成系による構文検査系作成の試み

本章では，コンパイラ自動生成系を用いて行った作成実験の概要を示す．

ここで用いたコンパイラ自動生成系は，Algol 68 等の多重パス・コンパイラ作成のために設計されたものであるが，Ada は 1 パスで解析可能となるように設計されているため，この自動生成系を 1 パス用にして使用した．生成系の概要を **図-1** に示す（図-1, 2, 3 については文献[14]を参照されたい）．なお，我々の手元では Yacc システムをはじめとして何種類かのコンパイラ作成システムが利用可能であるが，これらは十分整備されていないので上記システムを選択することとなった．

† An Implementation of ADA Syntax Checker Using a Compiler Generator System by Takehiro TOKUDA, Hiroyuki YOSHIDA and Kenzo INOUE (Department of Information Science, Tokyo Institute of Technology).

†† 東京工業大学理学部情報科学科

```
class
  LET="ABCDEFGHIJKLMNOPQRSTUVWXYZ": ID,
  DIG="0123456789"                : NL,
  LST="<"                         : LT,
    ⋮
table
  RESERVED= 'ABORT', 'ACCEPT',
                ⋮
            'XOR';
    ⋮
symbol
  ID=LET<LET|DIG|'-'LET|'-'DIG>
     in RESERVED out ?STORE_ID!IDENTIFIER,
  LT= '<'<'<'|'='|'>'|/>!,
    ⋮
```

図-2 字句単位記述の例

我々の当面の目標は，Ada の完全なコンパイラを作成することではなく，言語自体の評価のための道具として，静的に可能な各種の検査を実行する構文検査系を試作することであった．また，主な言語の機能のうち，並列処理(task)，分離翻訳 (separate compilation)，総称宣言 (generic declaration) 等を省略せざるをえなかった．

まず語彙解析部分に対する主な作業は，Ada の各字句単位 (lexical unit) を正規表現 (regular expression) で記述すること (**図-2**) と，識別子 (identifier) や数値リテラル (numeric literal) を内部表に格納するための手続き (図-2 中では STORE_ID) を記述することである．1979 年6月版 Ada の場合はこの部分に関して大きな問題はなかったが，1980 年7月版 Ada の場合は，引用符(')を文字リテラル (character literal)，型指定式 (type specification)，属性 (attribute) に用いるようになったため扱いが複雑になっている．

次に構文解析部分に対する主な作業は，Ada の構文規則を単純 LR (1) 文法 (Simple LR (1) grammar) で記述することである (**図-3**)．Ada のような大きな言語の構文規則を制約の強い単純 LR (1) 文法で記述するため，この作業はかなり困難であった．また Ada のいくつかの構文規則は単純 LR (1) 文法では自然に記述できないことも判明した (詳しくは文献[19]を参照されたい)．結局，単純 LR (1) 文法の制約と我々の生成系自体の大きさに関する制約のため，前述のように Ada の構文全体を扱うことは断念することとした．

さらに意味解析部分に対する作業は，各構文規則に対してそれが還元されるときに行う手続き (図-3 中では <*と*> に囲まれた部分) を与えることである．これらの手続きが行う主な処理は，宣言された名前を名前表に登録することと，その名前表を用いて演算子判定，型検査等を行うことである．名前表と演算子判定に関しては第3章と4章で詳述する．

```
variable_declaration→'IDENTIFIER'⟨ID⟩ ':'
                        subtype_indication⟨TYP⟩ ';'
    ⟨* if FOUND_IN_CURRENT_SCOPE(ID) then
          call ERROR ("illegal redeclaration")
       else
          P:=NEW_RECORD (variable);
          call INSERT (P, ID, variable);
          OBJECT_TYPE (P):=TYP
            ⋮
       fi *⟩
        ⋮
subtype-indication⟨TYP⟩ → name⟨NAM⟩
    ⟨* if NAME_CLASS (NAM)=type then
         TYP:=NAM
       else
         call ERROR ("illegal usage of name")
       fi *⟩
        ⋮
return_statement→ 'RETURN' expression⟨EXP⟩ ';'
    ⟨* if NAME_CLASS (CURRENT_UNIT)/=function
       then
          call ERROR ("illegal return statement")
       else
         T:=RETURN_TYPE (CURRENT_UNIT);
         call IDENTIFY (EXP, T)
       fi *⟩
        ⋮
```

図-3 構文とその意味付けの例

作成した処理系に例題を与えた結果を **図-4** に示しておく．

3. 名前表の構造について

本章では，名前表の基本的構造とユース・クローズ (use clause) の処理に関して検討する．

1980 年7月版 Ada からはスコープ (scope) の直接的な制御機能* が除かれた．したがって，名前表の構造は基本的には従来の手法で十分扱えるようになったが，オーバロードに関連し多少の検討が必要である．

まず **図-5** のような構造を持つ名前表において，たとえばBという識別子が表わす名前を検索する場合について考えてみよう．現在のスコープ (プログラムの現地点を囲む最も内側のスコープ，図ではスコープ3) に対するチェインを順々にたどっていくと，最初に

* スコープの外側の名前を見えなくする機能．visibility restriction のこと．なお J.D. Ichbiah 氏によると 1979 年6月版と 1980 年7月版 Ada の相違を詮索する学問を ADAchaeology と呼ぶのだそうである．

```
LINE    SOURCE CARD

  1
  2         --   STANDARDS   --
  3
  4         PRAGMA LIST ( OFF ) ;
247
248         -- END STANDARDS --
249
250
251       RESTRICTED( TEXT_IO )
252             -- VISIBILITY RESTRICTION ( FOR PRELIMINARY ADA )
253       PROCEDURE SAMPLE_OF_ADA IS
254        USE TEXT_IO ;
255         I, J: INTEGER := 0 ;
256         X, Y: FLOAT   := 0.0 ;
257         S   : STRING  ;
*****                   ?                          #3100  => STRING
258             -- NOT DECLARED NAME
259       BEGIN
260         I := J + 1    ;
261         X := Y + 1.0 ;
262         I := J + 1.0 ;
*****                   ?                          #3061
263             -- UNIDENTIFIED EXPRESSION
264         PUT(X, 10, 5)
265       END SAMPLE_OF_ADA ;
*****   ?                                          #5010
266             -- LACK OF ';'
267
268
```

図-4 構文検査の実例（1979年6月版 Ada 用）

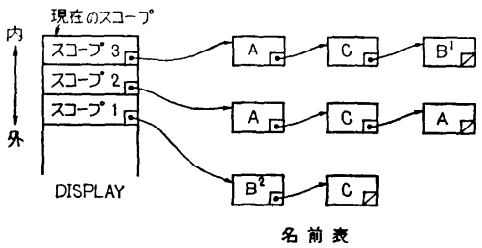


図-5 名前表の構造（その1）

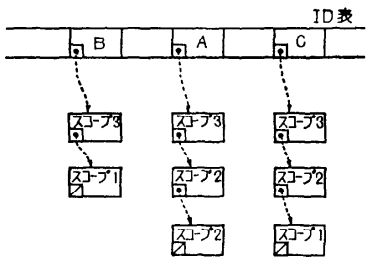


図-6 名前表の構造（その2）

名前 B^1 に到達する．もしもこの名前がオーバロード (overload) しない名前（オブジェクト名，型名など，以下本稿ではこれらを仮に第Ⅰ種の名前と呼ぶことにする．）であれば検索は終了し，識別子Bはこの名前 B^1 のみを表わしていたことになる．一方 B^1 がオーバロードする名前（副プログラム名など．以下これらを第Ⅱ種の名前と呼ぶことにする．）であるときには，さらに検索を続けねばならない．図-5の場合，名前 B^2 が第Ⅰ種であれば検索はそこまでで済み，識別子Bはやはり B^1 のみを表わしていたことになる．しかし，B^2 が第Ⅱ種の名前であるときは B^1 と B^2 がオーバロードし，検索はさらに外側のスコープへと，最悪の場合には最も外側のスコープに至るまで続行する必要がある．こうした事情から，Ada のようにオーバロードを広範囲に許す言語に対する名前表は，表の作成・管理に対する効率よりも検索の効率が重視されると考えられる．

図-6 は，同じ識別子を持つ名前をチェインでつないだ構造の名前表である．宣言された名前は，その識別子に対するチェイン上の最も手前の位置に登録する．各チェインを，手前の位置にある名前ほど内側のスコープに属すように管理すれば，ある識別子が表わす名前は，その識別子に対するチェインの先頭の1個ないしは（オーバロードしていれば）複数個の名前である*．しかし，こういう状態を保つためには閉じたスコープに属す名前を各々のチェインから除去しなければならない．前例の場合は DISPLAY の頂上を示すポインタを変えるだけで暗黙の内にそれらの名前を取り除いたことになるが，図-6 の場合は1つ1つリンクを付け変えねばならない．そこで，検索と管理の手間のトレード・オフを考慮する必要がある．前述したように，Ada は一部の名前のオーバロードを認めたために広範囲に及ぶ検索を余儀なくされており，表管理の効率を多少悪くすることはやむをえないと思われる．

リンクの付け変えを効率よく行うための方法の1つは，図-5 と類似のチェインを導入することである（図-7 以下，図の点線のチェインをインデックス・チェイン，実線をスコープ・チェインと呼ぶ）．スコープが閉じたときに，DISPLAY のポインタを変えるとともに，スコープ・チェインをたどって各インデック

* 実際にはさらに specification の同値性を考慮する必要がある．

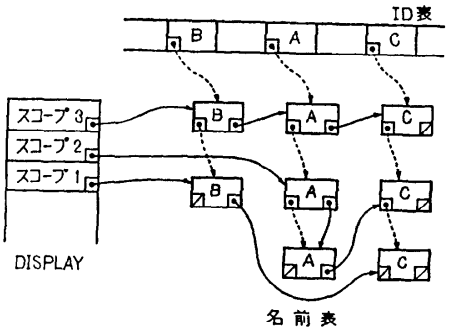


図-7　名前表の構造（その3）

ス・チェインを修正していけばよい．

このスコープ・チェインには別の使い道がある．Ada の名前のうち，選択成分 (selected component) の可視性 (visibility) の扱いは特殊であり，識別子に対してそれが表わす名前が属すべきスコープを点記法によって指定したものとも考えることができる．これを検索する場合にはむしろスコープ・チェインを用いる方が自然であろう．また，パッケージ (package)，記録型定義 (record type definition) などの中で宣言された名前は，これらのスコープが閉じた後，つまりインデックス・チェインから除かれた後で使用されることがある．この場合も，スコープ・チェインを残しておけばこれを使って検索できる．

最後にユース・クローズの処理について考察する．ユース・クローズは，「内側のものが優先する」という入れ子構造の原則に従わず，内側のスコープに属するものも外側のものも対等である．したがって，検索は常に最も外側に置かれたユース・クローズにまで行う必要がある．これを効率良く行うためには，図-6と同様にユース・クローズによって可視 (visible) になる名前を識別子ごとのチェイン（以下ユース・チェインと呼ぶ）でつないでおくのが妥当であろう．したがって，インデックス・チェインと合わせて1つの識別子につき2本のチェインができることになる．ユース・チェイン上に可視になる名前のみを置くことができると非常に都合がよいわけであるが，実現は難しい．あるパッケージがユース・クローズに指定されたときに，そのパッケージに属する名前が可視になるかどうかは，そのときのインデックス・チェインの状態，その名前が第Ⅰ種か第Ⅱ種か，それまでのユース・チェインの状態によって異なる．したがって，ユース・クローズが現われたときにユース・チェインを修正し，そのユース・クローズの属すスコープが閉じたときにこれを元の状態に戻すことは非常にやっかいな作業である．それよりも，ユース・チェインには可視になるかもしれない名前をすべてつないでおくようにする方が現実的であろう．ユース・チェイン上のどの名前が可視かという判定は毎回検索のときに行うことになるが，この判定を効率良く行うことは Ada 処理系の1つの課題となるであろう．

4. 演算子判定の問題について

本章では，演算子* の判定を意味解析段階においてどのような方針で行うべきかについて検討する．

従来のプログラミング言語においても，たとえば加法の演算子 "+" が整数型用のものか実数型用のものか等を判定する問題は生じたが，被演算子の型のみで判定を行うよう設計されているため，解析木を仮に作るとしても葉から根へ上向き (bottom-up) の評価を1回行うだけで判定が可能であった．したがって，特に解析木を作る必要もなく，たとえば各構文規則の還元時にそのとき決定した情報を意味スタックに積むという方法で解決できた．

Ada においては，被演算子の型以外の情報も使って判定を行うことが許されているため，式に対して解析木を作りその木の上で何らかの方針で tree walk をして演算子の判定をする必要がある**．

一見このtree walk はそれ以上新しい情報が伝播しなくなるまで，つまり収束するまで無制限に繰り返さねばならないように思われる．しかしながら，実は定数回のwalk によって収束することを，Ganzinger[6] が示した．Ganzinger の方法は，まず根から葉へ下向き (top-down) に情報を送り次に上向きに情報を送るという一連の操作を2回繰り返すものである（2回めの上向きの評価は実は不要***）．この後，Karlsruhe 大学のグループがまず上向きに次に下向きに評価を行うと収束することを示した[13]．ここでは，この2回の評価で収束する方法の存在を，Karlsruhe の扱い方とは独立のアプローチで示すことができた[17]ので，我々の流儀で示すことにしよう．

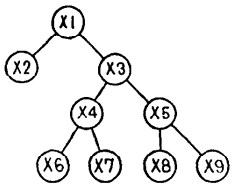
まずこの演算子判定問題で収束解を求める問題を，

---

* ここでいう演算子は広義の演算子のことで，何らかの値を返す構成要素はみな演算子と考えている．

** 演算子の判定問題に関しては文献2）の第2章に古典的説明がある．

*** 1979年6月版 Ada の場合は，type specification と type conversion の問題があるのでさらにもう1組繰り返し合計6回の評価を行う．



(a) 木 T

節の組 (Xi, Xj, Xk)	許される数字の関係 (ti, tj, tk)
(X 1, X 2, X 3)	(2, 1, 1) か (1, 3, 2)
(X 3, X 4, X 5)	(1, 2, 1) か (1, 1, 2) か (2, 1, 3)
(X 4, X 6, X 7)	(1, 1, 2) か (2, 1, 2) か (3, 1, 2)
(X 5, X 8, X 9)	(1, 1, 2) か (2, 1, 2)

(b) 表 R

図-8　演算子判定問題のモデルとなる組合せ問題

生成規則：$Xi ::= Xj\ Xk$
/* 属性 S(・) と I(・) は値として
　集合 {1, 2, 3} の部分集合をとる.*/
意味規則：S (Xi):=Root (i, j, k)
　　I (Xj):=Leaf (i, j, k, j)
　　I (Xk):=Leaf (i, j, k, k)
ただし Root(i, j, k) の定義は次の通りで,
　Root:=φ;
　for each relation (ti, tj, tk) at (Xi, Xj, Xk)
　do if (Xj is a terminal **or** tj is in S(Xj)) **and**
　　(Xk is a terminal **or** tk is in S(Xk))
　　then add ti to Root **fi**
　od
そして Leaf(i, j, k, l) の定義は以下の通りである.
　Leaf:=φ;
　for each relation (ti, tj, tk) at (Xi, Xj, Xk)
　do if ((Xi is the root **and** ti is in S(Xi)) **or**
　　(Xi is not the root **and** ti is in I(Xi)))
　　and (Xj is a terminal **or** tj is in S(Xj))
　　and (Xk is a terminal **or** tk is in S(Xk))
　　then add tl to Leaf **fi**
　od

図-9　各節の解の最大集合を求めるアルゴリズム

次のような設定のもとでの木の各節の解の値の集合で最大なものを求める問題と考えることにする.

〔問題〕 **図-8**（a）の木Tのすべての節に,数字1, 2, 3のいずれか1つを割当てる. ただしその割当て方は**図-8**（b）の表Rの関係に違反してはならない. このとき, 木T全体への数字の割当て方を解と呼ぶことにする. 各節における解の値の集合で濃度が最大となる集合（解の最大集合）を求めよ.

実際に演算子判定問題で判定したいのは, 以下の点である.

ⅰ）各演算子が解釈を1通り持つのか, それとも0通りまたは複数通り持つのかを判定する. 0通りまたは複数通りの解釈を持つ演算子が少なくとも1個存在するプログラムは, 無意味か曖昧となり, エラーとなる.

ⅱ）どの演算子もちょうど1通りの解釈を持つプログラムに対しては, 各演算子の解釈を求める.

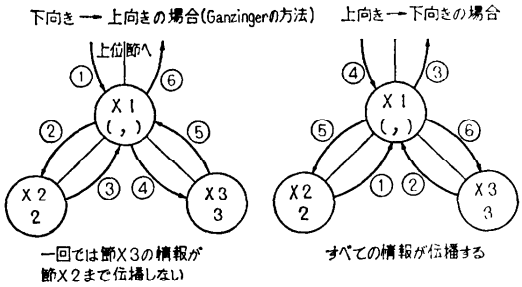


図-10　解析木の評価順序

したがって上記の問題に直すと, 各節の解の最大集合（各演算子の解釈の全部からなる集合）の濃度が1のときは, 各節における解を求め, 少なくとも1つの節の解の最大集合の濃度が0または2以上のときは, エラーを宣告することになる.

〔解法〕 解法を属性文法の属性の計算手順の形で**図-9**に示す. 1回目の上向きの評価で, 属性 S(Xi) の値, すなわち節 Xi を根とする部分木に関する問題の根における解の最大集合が求まる. したがって特に木Tに関する問題の根における解の最大集合も求まる. 次の下向きの1回の評価で, 属性 I(Xi) の値, すなわち木Tに関する問題の（根を除く）すべての節 Xi における解の最大集合が求まることになる.（この問題は幸いなことに解析木の上の問題なのでやさしくなっている. 文献17) に示すように木ではなくグラフ上の問題に一般化すると, この問題には効率の良い解が存在しなくなってしまう.）

Ganzinger の方法が下向きと上向きの2回の評価で収束しない理由は**図-10**に示す通りである. この方法は余分な評価が必要で, したがって効率が悪いように思えるが, 我々の経験から判断すれば実用性はむしろ高いといえる. 図-10において節X2から節X1への上向き情報を言葉で表わせば,「第1欄の型はすべ

ての整数型 (integer type) とその導出型 (derived type) のうちのいずれか1つ」である*. さらに節X1からの上向き情報は「すべての整数型とその導出型のうちのいずれかの型の欄を2つ持つ配列型 (array type) または記録型 (record type)」である*. このような複雑な情報に対しどのような内部表現を選ぶことが現実的であろうか. Ganzinger と Ripken[8] は, 下向き情報を先に評価することと, 型の階層性を利用することで巧妙にこの問題を解決している.

5. おわりに

1980年7月版 Ada は1979年6月版と比べると, 実現上の困難な問題がかなり削除されてしまっている. これは処理系作成には喜ばしいことであろうが, 半面 Ada の異色性 (異常性) が減って, おとなしい言語に近づいたことになるのかもしれない.

コンパイラ自動生成系を用いて Ada を作成する試みは各地で行われているようであるが, これは言語仕様の変更に強いことと, 試験的作成に向いていることによるものと思われる. 言語処理系以外の道具の作成にも, もう少しコンパイラ自動生成系が使われることを望みたい.

本稿が Ada 処理系の実現法をめぐる議論に多少とも参考となれば幸いである.

参考文献

1) Aho, A. V. and Ullman, J. D.: Principles of Compiler Design, Addison-Wesley (1974), (邦訳が培風館から近く刊行される予定).
2) Bauer, F. L. et al.: Compiler Construction, Springer-Verlag (1976).
3) Department of Defense: REFERENCE MANUAL FOR THE Ada PROGRAMMING LANGUAGE (1980).
4) Department of Defense: PRELIMINARY ADA REFERENCE MANUAL, SIGPLAN Notices, Vol. 14, No. 6 Part A (1979).
5) Department of Defense: Rationale for the Design of the ADA programming language, SIGPLAN Notices, Vol. 14, No. 6 Part B (1979).
6) Department of Defense: REQUIREMENTS FOR HIGH ORDER COMPUTER PROGRAMMING LANGUAGES, "STEELMAN" (1978).
7) Department of Defense: REQUIREMENTS FOR Ada PROGRAMMING SUPPORT ENVIRONMENTS, "STONEMAN" (1980).
8) Ganzinger, H. and Ripken, K.: Operator Identification in Ada, Formal Specification, Complexity, and Concrete Implementation, SIGPLAN Notices, Vol. 15, No. 2, pp. 33-42 (1980).
9) Gries, D.: Compiler construction for digital computers, John Wiley & Sons (1971), (邦訳, 牛島和夫訳: コンパイラ作成の技法, 日本コンピュータ協会).
10) Inoue, K. et al.: A generation-method of multiphase-compilers, 18th Proc. IPSJ p. 302 (1977).
11) 筧 捷彦: Ada—米国国防総省新言語, 情報処理, Vol. 21, No. 9, pp. 975-979 (1980).
12) 中田育男: プログラミング言語の歴史と展望, 情報処理, Vol. 21, No. 5 (1980).
13) Persch, G., Winterstein, G. et al.: Overloading in ADA, Bericht. Nr. 23/79, Universität Karlsruhe (1979).
14) Sassa, M., Tokuda, J., Shinogi, T. and Inoue, K.: Design and Implementation of a Multipass-Compiler Generator, Journal of Information Processing, Vol. 3, No. 2, pp. 77-86 (1980).
15) 近山 隆: 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, pp. 137-142 (1980).
16) 徳田雄洋: Ada 実現に関する問題点について, 情報処理, Vol. 21, No. 3, pp. 226-232 (1980).
17) Tokuda, T.: An exercise in transforming Wijngaarden grammars into Knuthian grammars, 京都大学数理解析研講究録 "ソフトウェア科学・工学における数理的方法" (1980).
18) TSINKY: 新しいプログラミング言語 ADA, bit, Vol. 12, No. 3, pp. 51-59 (1980).
19) Yoshida, H. and Inoue, K.: Experimental Implementation of ADA translator using a Compiler Generation System, 21 st Proc. IPSJ, pp. 2 B-10 (1980).
20) 和田英一: Ada の概要, 情報処理, Vol. 22, No. 2 (1981).

(昭和55年11月4日受付)

付記

本稿作成後, 文献 8) に対するコメントが SIGPLAN Notices 誌1980年7月・8月合併号, および1980年9月号にそれぞれ発表されている.

* 正確にはもっと複雑で長い表現となるが, 省略する.