関数型データベースのための視覚的検索言語

永 江 尚 義[†] 有 澤 博[†]

本稿では，関数型データベースのための新しい検索言語を提案する．本稿の議論の基盤となる関数型データモデルでは，構造化オペレータと呼ばれるデータ操作系によってデータベースのデータを階層型のデータへ構造化することができる．これにより，単一のデータベースで様々な複合オブジェクトを提供することが可能となる．しかし，構造化オペレータのプロシージャには手続き的な処理の記述が必要であったり，プロシージャ中で各オペレータ間の上下関係や依存関係を直観的に把握しづらいなどの問題点がある．そこで本稿では，構造化オペレータの持つこれらの問題を解決し，データベース利用者が自分が作り出す階層型データをより直観的に把握しながら，データベースへの検索を記述することを可能にするための視覚的検索言語を提案する．またさらに，視覚的検索言語を等価な構造化オペレータへ変換するアルゴリズムを示す．

The Visual Query Language for the Construction of Hierarchical Data in Functional Databases

HISAYOSHI NAGAE [†] and HIROSHI ARISAWA [†]

This paper presents a new query language in functional databases. Our functional data model can construct hierarchical data from "flat" data by using special database operators. But the database operators request database users to describe procedural query programs. Furthermore, it is hard for users to understand relationships or dependencies between operators in a query program. This paper proposes a new visual query language which can provide users with non-procedural and intuitive description of programs. The visual query language enables database users to understand the structure of the complex objects retrieved, and the relationships between attribute values in such objects. We also show an algorithm which converts a visual query program into database operators.

1. はじめに

データベースシステムの応用分野がCAD，フルテキスト処理，マルチメディア情報処理，知識処理支援などの高度情報処理へ広がるにつれて，文字・数値データを主な対象としていた関係データモデル(relational data model) では扱い切れないようなデータの蓄積と操作が要求されるようになった．例えば，複雑で大容量の部分構造を持つオブジェクト（いわゆる複合オブジェクト）などがこれにあたる．ただし，データベースで複合オブジェクトを管理する場合，データ共有の立場から，単に1つの見方（ビュー）から複雑な構造のデータを管理するだけでは不十分であり，その複合オブジェクトの上下の構造が逆転したものなど，様々な構造のデータを提供できる必要がある．

このような問題を解決する1つの方法として素データがむしろ単純な二項関係で蓄積される関数型データモデル (functional data model)[1)~3)]でデータを表現し，データの構造情報を表現するための意味制約と検索の際にデータを構造化して取り出す機構（これを構造化オペレータと呼んでいる）を付加し，検索の結果として構造化されたデータを作り出す手法が提案されている[4)]．この手法により，データベース中から自由な形の複合オブジェクトを作り出すことが可能となり，オブジェクト指向の最大の欠点であるデータベース設計者によって固定化された構造しか扱えないという問題点を解消している．

ただし，構造化オペレータによってデータベース利用者が必要とする複雑に構造化されたデータ（複合オブジェクト）をデータベースから生成するためには，いくつものオペレータによって構成されるオペレータ列を手続き的に記述する必要がある．しかし，構造化オペレータには一見しただけではデータベース利用者が『自分が作成しようとしている階層型データのどの部分を操作しているのか』をイメージしづらいことや

† 横浜国立大学工学部電子情報工学科
Division of Electrical and Computer Engineering, Faculty of Engineering, Yokohama National University

ひとまとまりの処理を表すオペレータ列を直観的に把握できない，オペレータの上下関係が不明確である等の問題点があった．

本稿では，構造化オペレータの持つこれらの問題を解決し，データベース利用者が自分が作り出す階層型データをより直観的に把握しながら，データベースへの検索を記述することを可能にするための視覚的検索言語を提案する．さらに，視覚的検索言語をそれと等価な構造化オペレータへ変換するアルゴリズムについてもあわせて提案する．

2. 複合オブジェクトの表現

2.1 関数型データモデルと複合オブジェクト

関数型データモデルでは基本的に主体(entity)と関数（function）の概念だけでデータベースが構成される．ここで言う主体とは，データベース化の対象となっている世界を観察した結果，独立した事物・事象やそれに付随する属性値など，1つの『もの』として認識され，データベースに格納される最小単位のことである．オブジェクト指向データベースにおけるオブジェクトの概念と異なり，主体は内部構造を持たず，構造を持った『もの』はすべて最小単位の主体にまで分解し，上位概念を表す主体と部分を表す主体との間の連係（後述の関数）によって表現する．また，ある共通の意味を表す主体の集まりを主体集合（entity set）と呼び，主体集合につけられた名称（すなわち，個々の主体を総称するための名称）を主体型（entity type）と呼ぶ．関数はこの主体間の対応付けを表現する．本稿で述べる関数型データモデルの関数は1つの引数だけをとり*,与えられた個々の主体を主体の集合へ写像する．

図1は，このような関数型データモデルのモデリング手法に基づいてモデル化された課－従業員データベースのインスタンス(instance)を示している．このダイアグラムにおいて，長方形は『主体集合（entity set)』，長方形の脇の文字列は『主体型（entity type)』を表す．また，長方形中の小円は個々の『主体(entity)』を表し，小円の脇の文字列（もしくは，数値）は，主体識別記号または主体の表す数値（文字）情報を示している．長方形（すなわち，主体集合）の間の太い矢印は，矢印で結ばれた2つの主体集合上で定義された

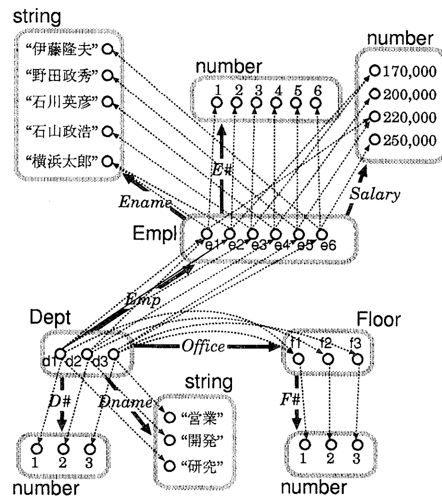


図1　課-従業員データベース(インスタンス)

Fig. 1　A department-employee database (instance).

『関数(function)』を表し，矢印の根元側の長方形が関数の定義域（domain)，矢印の先端側が値域（range）の主体集合となる．また，矢印上の文字列は『関数名(function name)』を表す．さらに，破線の矢印は，個々の主体間の関数による対応を表している．

このように，関数型データモデルではすべてのデータ間の関連が主体（entity）と関数（function）の概念だけの二項関係で表現されているため，現実世界に存在する複雑なデータ構造もすべて二項関係に分解される．このことにより，上下関係のない自由度の高い形でデータを表現し，蓄積することが可能であるが，一方でデータベース利用者が複合オブジェクトという形でデータを利用したい場合には，データベース利用者は各自でデータの二項関係で構成される関数型データベースから階層構造を持ったデータで構成されるいわゆる複合オブジェクトを必要に応じて生成する作業が必要となる．

2.2 主体木による複合オブジェクトの表現

本稿で，関数型データモデルから検索する複合オブジェクトは単に『データを階層的に結び付けたもの』であり，一般的なオブジェクト指向データベースにおける複合オブジェクトとは若干異なるため，以下では『**主体木（entity tree)**』と呼ぶ．

定義1　主体木（entity tree)

主体木は次のように再帰的に定義される．

1. 数値，文字および文字列，non-printable データなどデータベース中のすべての主体をアトム主体木（atomic entity tree）と呼ぶ．
2. BOTTOM（⊥，undefined entity tree）は特殊

---

* 一般に，引数の「n 個組」をまた1つの主体と考え，それぞれの引数からその主体への関数を定義することにより，1引数の関数だけを使って n 引数の関数を表現することができる．したがって，本稿では1引数関数のみを考え，あえて n 引数関数への拡張は行わない．

な主体木である．

3．$e_1, e_2, \cdots, e_n$ がアトム主体木もしくはタプル主体木のとき，$\{e_1, e_2, \cdots, e_n\}$ をセット主体木（set entity tree）と呼ぶ．

4．主体 e の属性が $f_1, f_2, \cdots, f_n (n \geq 0)$ であり，主体木 $e_1, e_2, \cdots, e_n$ がそれぞれ属性値であるとき，$(e)[f_1{:}e_1, f_2{:}e_2, \cdots, f_n{:}e_n]$ をタプル主体木（tuple entity tree）と呼ぶ．

5．主体木は，アトム主体木，セット主体木，タプル主体木，BOTTOM のいずれかである．

いま，課データをオブジェクトの頂点(根，root)とし，その属性として，『課名』，『所属従業員の列』があり，さらにその下位の構造として，各所属従業員ごとに『従業員番号』，『氏名』，『給与』という属性が存在するような階層型のデータがデータベースから検索されたとする．このデータは図2のような主体木として表現される．

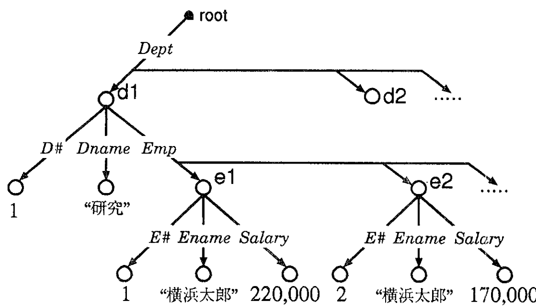


図2 課-従業員データを表す主体木
Fig. 2 An entity tree of a department-employee object.

なおここで，主体木中の異なる位置に同一の主体が重複して存在していてもよい．例えば，d_1 の属性 **Emp** の属性値として e_1 が存在し，同時に d_2 の属性 **Emp** の属性値として e_1 が存在してもよい．これはそれぞれの e_1 が d_1 に所属する e_1 と d_2 に所属する e_1 という別々の意味を表していると考えられるためである．すなわち，同一の主体が主体木中の別の位置に重複して現れているのはそれぞれの主体が「ある構造における別々の意味（役割）」を表しているためであり，主体木中ではこれらの主体は別の主体であるとみなされる．

本稿の関数型データモデルでは，図2のような階層構造のデータを図3のように角括弧（"[ ]"），ブレース（"{ }"）を使用して主体木を記述したものを『**主体木式（entity tree expression）**』と呼ぶ．

また，この主体木中のそれぞれの構造は，**図4**で示される意味を表している．

このように主体木の表現形式は，従来から階層構造のデータを表現するために伝統的に使用されてきた記述法と大きな違いはない．

なお，主体木中のある共通の意味を表すインスタンスは，必ず共通の構造を持つ．例えば，図2の主体木中の各々の従業員を表すタプル主体木はすべて同じ属性を持っている．一般に，オブジェクト指向データベースなどではデータベース中の個々のデータ（オブジェクト）はそれぞれ異なる構造を持つことが許されている．一方，本稿の主体木では，データはすべて構造が同一でなければならないが，このような制約を与えると，一部のデータにだけ特別な属性値が存在するなどデータの構造が必ずしもすべて同一でないようなケ

```
[ Dept:{(d1)[ D#:1, Dname:"研究", Emp:{(e1)[ E#:1, Ename:"横浜太郎", Salary:220,000 ],
                                    (e2)[ E#:2, Ename:"横浜太郎", Salary:170,000 ],
                                    … }],
        (d2)[ D#:2, Dname:"開発", Emp:{(e1)[ E#:1, Ename:"横浜太郎", Salary:220,000 ],
                                    … }],
        … }]
```

図3 課-従業員データを表す主体木式
Fig. 3 An entity tree expression of a department-employee object.

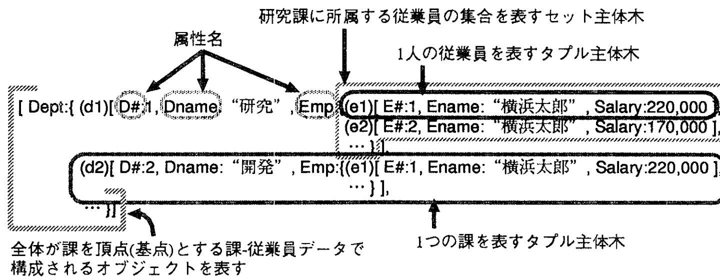


図4 主体木式の構造と意味
Fig. 4 The semantics of components of an entity tree expression.

ースの場合に問題が生じてしまうと思われるかもしれない．本稿の関数型データベースでは，主体に対して無意味な関数を適用した結果は常に未定義主体（⊥）となる．この性質を利用して，構造が異なるデータやある属性が他のデータでは定義されていない場合には，主体木中のすべてのデータに対して強制的に属性値が⊥であるような属性を付加して，結果的にすべてのデータが共通の構造を持つようにすることができる．

2.3 主体木とランク記述子

主体木のインスタンスが共通の構造を持つために，例えばデータベース利用者は『主体木中の属性名“**Dept**”のインスタンス中の属性名“**D #**”の各主体木に対して…という操作を行う』と指定するだけで主体木に対する操作を記述できることになる．本稿では主体木に対して操作を行う際に操作対象のデータを指定する方法として，主体木の属性名を列挙することを採用する．例えば，前述の例は『**/Dept/D #**に対して…』と記述される(図5参照)．このように操作対象の主体木を指定するために属性名を列挙した文字列を『**ランク記述子（rank descriptor)**』と呼ぶ．ここで，スラッシュ（/）はデリミタである．

いま，データベース利用者は1つのランク記述子によって主体木中の複数の主体を一度に指示することができる．例えば，図5の主体木に対して，**/Dept/D #**というランク記述子によって，すべての課（Dept）データの属性**D #**の値を指定することができる．このことは，後述する集合演算の形式で与えられる構造化オペレータによる操作を記述する際に有用となる．

ここで，ランク記述子の書式は2種類存在する．1つは，主体木の根(root)から目的の主体が位置している属性までの属性名を列挙する表記法である．図5の例がこれにあたり，この表記法を『絶対ランク記述子(absolute rank descriptor)』と呼ぶ．

これに対し，主体木中のある注目点から目的の主体が位置している属性までの属性名を列挙した表記法は，『相対ランク記述子（relative rank descriptor)』と呼ばれる（図6参照)．なおここで，1ランク上へ上がる場合には，特別に“..”を属性名とする．

2.4 主体木に対する操作

本節では，データベース利用者が利用しやすい形式に構造化されたデータを生成/操作するための構造化オペレータについて議論する．データベース中において単純な二項関係に細かく分割され，様々な二項関係の中に埋没し，明確に区別されなくなってしまった元々のデータが持っていた階層構造などの構造情報を

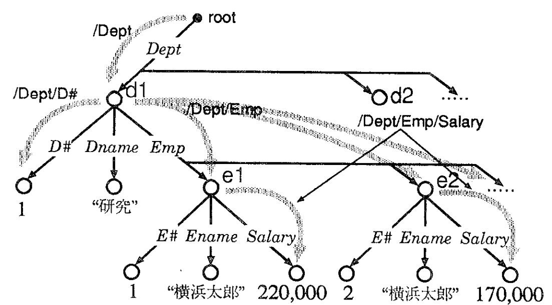


図5 主体木と絶対ランク記述子
Fig. 5 The absolute rank descriptors.

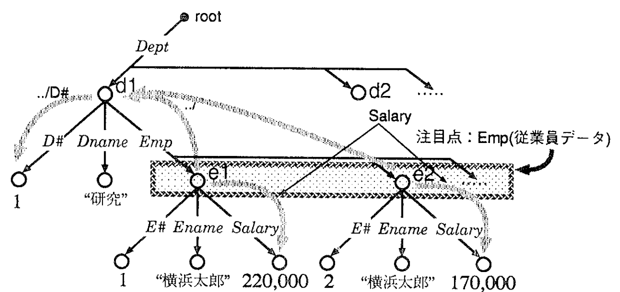


図6 主体木と相対ランク記述子
Fig. 6 The relative rank descriptors.

組み合わせてデータベース利用者の意図する主体木を生成/操作するために，本稿で議論する関数型データモデルでは，以下の5つのオペレータを提供している[4]．

（1） **根の生成（generate）**

generate は指定された主体集合に属するすべての主体の集合を生成する．引数で指定された主体型を属性名とし，その主体型が指す主体集合中に属するすべての主体の集合をその属性値とするようなタプル主体木を生成する．

$generate(E)$:NULL→$[E:\{e_1, e_2, \cdots, e_n\}]$

$e_i \in E$（E は生成する主体型である）

（2） **枝の生成（apply）**

apply はランク記述子で指定された主体に対して与えられた関数を適用し，その結果として得られた主体（もしくは，主体の集合）を主体木内へ埋め込む操作である．

$$apply(R, F, A): e_i \rightarrow \begin{cases} (e_i)[A:f] \\ \quad (\text{単値関数 } F(e_i)=f \text{ のとき}) \\ (e_i)[A:\{f_1, f_2, \cdots, f_n\}] \\ \quad (\text{多値関数 } F(e_i)=\{f_1, f_2, \cdots, f_n\} \text{ のとき}) \end{cases}$$

R は関数を適用する主体の位置を示すランク

記述子，e_i はランク記述子 R によって指定される主体．F は適用する関数名，A は主体木中における属性名である．

（3） **計算値の挿入（extend）**

extend はランク記述子で指定された主体木中の位置へ計算式を計算した結果得られた値を挿入する操作である．なお，第1引数のランク記述子は絶対ランク記述子を使用し，第3引数の計算式中では第1引数で指定された地点からの相対ランク記述子を使用する．

$extend(R, B, M): e_i \rightarrow (e_i)[B: M(e_i)]$

R は計算値を挿入する位置を示すランク記述子，e_i はランク記述子 R によって指定される主体．B は主体木中における属性名，M は計算式である．$M(e_i)$ は計算式 M の返り値を表す．

（4） **枝の削除（project）**

project は主体木中から引数で指定された特定の属性を削除する．

$$project(R/A_j): (e_i)[A_1: a_1, A_2: a_2, \cdots, A_n: a_n] \rightarrow (e_i)[A_1: a_1, \cdots, A_{j-1}: a_{j-1}, A_{j+1}: a_{j+1}, \cdots, A_n: a_n]$$

R/A_j は削除する属性の位置を示すランク記述子である．e_i はランク記述子 R によって指定される主体．a_j は属性 A_j の属性値を表す主体木である．

（5） **データ選択（select）**

select はデータベース利用者が生成した主体木中から利用者が必要としている一部のデータだけを取り出す．条件式の評価結果が真(true)であれば，主体を残し，偽（false）であれば，その主体をBOTTOM（⊥）に置き換える（これは，その主体を主体木中から削除することに対応する)．なお，第1引数のランク記述子は絶対ランク記述子を使用し，第2引数の条件式中では第1引数で指定された地点からの相対ランク記述子を使用する．

$$select(R, C): e_i \rightarrow \begin{cases} e_i & (\text{条件式 } C(e_i) = \text{真のとき}) \\ \perp & (\text{条件式 } C(e_i) = \text{偽のとき}) \end{cases}$$

R は選択の対象となる主体の位置を示すランク記述子，e_i はランク記述子 R によって指定される主体．C はデータ選択の条件式である．

上記の構造化オペレータは，検索処理の過程においてデータベース中に散在しているデータの構造化情報を結び付けることにより，構造化されたデータをデータベース利用者へ提供することを可能にしている．しかも，関数型データベース中の二項関係には本質的な上下関係がないために，データベース利用者は，構造化オペレータを使用して1つのデータベースから容易に様々な構造を持つ構造化データを生成/操作することができる．

ところで，本稿で提案している主体木は従来のデータベースにおけるビュー（view）の概念に相当する．よって，枝の生成/変更/削除はその主体木に対してのみ影響を与える操作で，データベースに格納されているデータに対しては何ら影響を与えない．

[Example]

課データ（**Dept**）を基点とし，その属性として課番号（**D #**）と従業員データ（**Emp**）が並べられ，さらに **Emp** データの属性として従業員番号（**E #**），氏名（**Ename**），給与（**Salary**）が存在するデータを検索することを考える．ただし，**Emp** データのうち，給与が20万円未満の者は検索データから除外する．

この検索を構造化オペレータを使って記述すると以下のようになる．

手続き1 *Query Procedure* 1

1 *generate* (**Dept**)
2 *apply* (**/Dept**, **D #**, **D #**)
3 *apply* (**/Dept**, **Emp**, **Emp**)
4 *apply* (**/Dept/Emp**, **E #**, **E #**)
5 *apply* (**/Dept/Emp**, **Ename**, **Ename**)
6 *apply* (**/Dept/Emp**, **Salary**, **Salary**)
7 *select* (**/Dept/Emp**, **Salary**≧200,000)

手続き1の各行について簡単に述べる．まず，行番号1では，まず主体木の基点となる課データの集合を生成している．次に，課データの要素として，属性 **D #**（課番号データ）と **Emp**（課を構成する従業員データの集合）を生成している（行2,3）．さらに，従業員データの要素として，3つの属性 **E #, Ename, Salary** をデータベースから検索している（行4,5,6）．ここまでの操作によって作られた主体木は，データベース中に存在するすべての課/従業員データによって構成されている．この中から行7の条件検索オペレータによって，データベース利用者が必要としているデータ（今回の場合は，給与が20万円以上の従業員）だけが抽出される．**図7**は，手続き1によって生成される主体木の構造を示したものである．

3．視覚的検索言語

前章の構造化オペレータは，繰り返しの手順を記述する必要が無い等，通常の手続き型プログラミング言語に比べて記述しやすく設計されている．しかし，オペレータ式の順番が検索処理の手順に密接に関連して

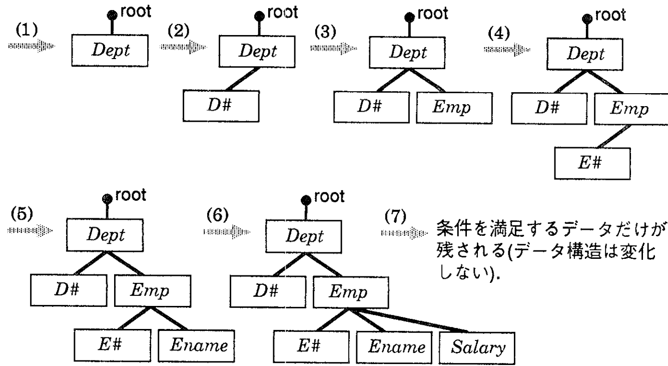


図7　手続き1による主体木の生成
Fig. 7 The generation of an entity tree by procedure 1.

おり，手続き的言語であることには変わりがない．しかも，構造化オペレータは1つのオペレータによって処理される対象が，1つのランク記述子によって指定される主体群に制限されているため，通常，データベース利用者が必要とする複雑に構造化されたデータ(複合オブジェクト)をデータベースから生成するためには，いくつものオペレータによって構成されるオペレータ列（手続き）を記述する必要がある．しかし，構造化オペレータには，検索式全体の中で個々のオペレータを一見しただけでは各オペレータがデータベース利用者が作成しようとしている『階層型データにおける，どの部分を操作しているのか』をイメージしづらいことやひとまとまりの処理を表すオペレータ列を直観的に把握できない，オペレータの上下関係が不明確である等の問題点がある．

本章では，構造化オペレータの持つこれらの問題を解決し，データベース利用者が自分が作り出す階層型データをより直観的に把握しながら，データベースへの検索を記述することを可能にするための視覚的な検索言語を提案する．

3.1　視覚的検索言語の構成

いま，次のような検索処理を考える．

課データ（**Dept**）を基点とし，その属性として課名（**Dname**）とその課に所属する従業員の平均給与（**AVESAL**），従業員データ（**Emp**）が並べられ，さらに **Emp** データの属性として従業員番号（**E #**），氏名（**Ename**）があるようなデータを検索することを考える．ただし，**Emp** データのうち，給与が各自の所属している課の平均給与以上の者だけを検索する．

この検索は，2.4 節の構造化オペレータを用いて，次のように記述される．

手続き 2　*Query Procedure* 2

1　*generate*（**Dept**）
2　*apply*（**/Dept, Dname, Dname**）
3　*apply*（**/Dept, Emp, Emp'**）
4　*apply*（**/Dept/Emp', Salary, Salary**）
5　*extend*（**/Dept, AVESAL,** ave（**Emp'/Salary**））
　ave()は平均値を計算する組込み関数である．
6　*project*（**/Dept/Emp'/Salary**）
7　*project*（**/Dept/Emp'**）
8　*apply*（**/Dept, Emp, Emp**）
9　*apply*（**/Dept/Emp, E #, E #**）
10　*apply*（**/Dept/Emp, Ename, Ename**）
11　*apply*（**/Dept/Emp, Salary, Salary**）
12　*select*（**/Dept/Emp, Salary≧../AVESAL**）
13　*project*（**/Dept/Emp/Salary**）

この検索処理と同等の操作を視覚的検索言語で記述すると，手続き3のようになる．

手続き 3　*Query Procedure* 2

```
[ Dept:[ Dname:,
         AVESAL:@=(ave(Emp'/Salary), [Emp|Emp':@[Salary:@] ]),
         Emp:[ E#:, Ename:, Salary:@]{Salary≧../AVESAL}
       ]
]
```

関数型データベースに対する検索を記述するための視覚的検索言語による検索式は，前章で定義した主体木を記述するための主体木式の書式とほぼ同じである．これは，データベース利用者が自分がデータベースから検索しようとしているデータ（すなわち，複合オブジェクト）の構造を直観的にイメージしながら検索式を記述できるようにするためである．この視覚的検索言語の文法を BNF 表記法にしたがって記述したものを付録 A に示す．また，視覚的検索言語の文法は，主体木に対する操作別にまとめると次のようになる．

（1）　**タプル構造の生成**

主体木中にタプル構造を生成するには，そのタプル構造を持つ主体の位置の部分に角括弧（“[　]”）を書いて表す．

（2） **属性（関数値）の抽出**

タプル構造中に属性値を埋め込むには，タプルを表す角括弧中に，属性名をコロン（“:”）と共に記述する．このとき，データベース中の関数名と主体木中の属性名が同じ場合には，単に“属性名:”と記述する．また，関数名と属性名が異なる場合には，“関数名｜属性名:”のように2つの名称の間にバー（“｜”）を書く．

（3） **データの条件検索**

主体木中に含まれるデータに対する条件検索は，ブレース（“{　}”）に囲まれた論理式を書くことで表現する．具体的には，条件検索の対象となる主体を表しているタプル主体木の右側にブレースで囲まれた条件式を記述する．

ここで，条件式中で参照できる属性は，全体の構造中の属性から，その条件式が指定されているタプル*を含まない**タプルの属性を取り除いたものだけである．また，代入式の右辺に存在するデータをアクセスすることもできない．このデータ参照の制約によって，検索式によって生成される主体木の一意性を保証することができる．

（4） **計算値の挿入**

主体木中にデータベースのデータから計算によって求めることができる計算値（**導出データ**（derived data）と呼ばれることもある）を挿入するには，“$A{:=}(M, T)$”のように記述する．

ここで，A は挿入される計算値の属性名，M は計算値を求める計算式，T は計算に必要なデータをデータベースから検索するための検索式である．また，以下では上記のように計算式 M と検索式 T を括弧で囲んで組みにしたものを代入式と呼ぶ．

検索式 T は，挿入する属性 A の1つ上のランクに対する検索式であり，計算式 M で使用するすべてのデータは，この検索式によって作成される部分主体木中に存在するものでなければならない．また，計算式 M 中で検索式 T のデータを参照する場合には，相対ランク記述子を指定する．

（5） **属性の削除**

視覚的検索言語では，データベース利用者が最終的に必要としていない属性であっても途中の検索過程（条件検索など）で参照される場合には，データベースから一旦抽出し，主体木中へ埋め込んでおく必要がある．このような，検索途中では必要だが，最終的な検索結果として保持する必要がない属性であることを指定するには，属性のコロンの右側にアットマーク（“@”）を書く．

したがって，手続き3の検索式のそれぞれの構造は，図8のような意味を表している．

図8の視覚的検索言語について簡単に説明する．(1)の角括弧は，主体集合 **Dept** の要素である課データを **Dname, AVESAL, Emp** の3つの属性から成るタプルとして検索することを宣言している．(2)は，課データに対して **Dname** という関数を適用し，その結果を属性名 **Dname** の属性値として主体木中へ格納する操作である．

(3)は，**AVESAL** が計算によって得られる導出された属性値であることを意味している．(4)，(6)は，代入式である．(4)は，導出データ（今回の例では，平均給与）を計算するために必要なデータ（従業員の給与データ（**Emp'/Salary**））を検索するための検索式である．ここで，(5)のように，適用する関数名（**Emp**）と主体木中における属性名（**Emp'**）が異なっているが，これは，(1)で指定された属性 **Emp** と名称が重複してしまうことを避けるためである．(6)は，(4)によって検索された従業員の給与データから平均給与を計算する操作を記述している．

(7)は，属性 **Emp** の従業員データに対する検索条件を記述している（ここでは，(3)～(6)で計算した平均給与 **AVESAL** 以上の給与の従業員だけを検索している）．(8)は，従業員の給与データ（**Salary**）が，検索途中（今回の例では，(7)の条件検索の際に使用している）で使用するが，最終的には必要ない属性であることを表す（**Salary** 以外にも，**AVESAL** を計算するためだけに(4)で検索されたデータも同様にして最終的な検索結果からは削除されている）．

ところで，上記の5つの構成要素からなる検索式によって生成される主体木は必ず一意性を持つことが保証されている．すなわち，ある1つの検索式が与えられた時，その検索式によって生成される主体木は論理的な評価がネストの内側から先に行われるために必ずただ1つに定まる．

3.2 視覚的検索言語と構造化オペレータの対応

視覚的検索言語を構成する構成要素は，それぞれ次のような構造化オペレータに対応している．

（1） **主体木の基底の生成**

主体木生成の基底部分の検索式の一般形を［R:［…］］とすると，この操作は次のような *generate* オ

---

* すなわち，条件式の｛　｝のすぐ左側にある“]”と対をなしている各括弧が表すタプルのこと．

** “$[^A X{:}[^B Y{:}]^B]^A$”の時，「タプル“$[^A X{:}]^A$”はタプル“$[^B Y{:}]^B$”を含む」という．

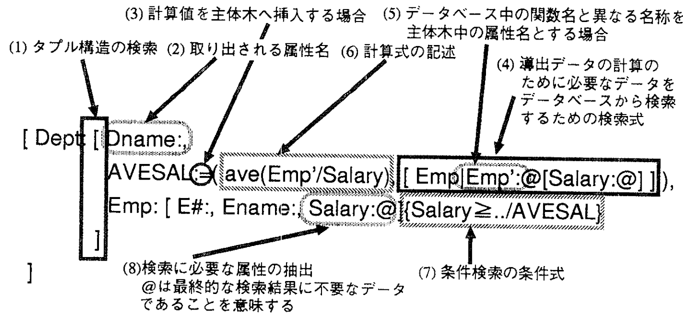


図8 視覚的検索言語の構造
Fig. 8 The semantics of a visual query procedure.

ペレータに対応する.

$[R:[\cdots]]\rightarrow generate(R)$

(2) **属性（関数値）の抽出**

属性（関数値）抽出の検索式の一般形を$[R:[F|A:]]$とすると，この操作は次のような *apply* オペレータに対応する.

$[R:[F|A:]]\rightarrow apply(R, F, A)$

(3) **データの条件検索**

主体木中に含まれるデータに対する条件検索の一般形を，$[R:[\cdots]\{C\}]$とすると，この操作は次のような *select* オペレータに対応する.

$[R:[\cdots]\{C\}]\rightarrow select(R, C)$

(4) **属性の削除**

主体木中の属性の削除操作の一般形を，$[R:[A:@]]$とすると，この操作は次のような *project* オペレータに対応する.

$[R:[A:@]]\rightarrow project(R/A)$

(5) **計算値の挿入**

主体木中にデータベースのデータから計算によって求めることができる計算値を挿入する一般形を，$[R:[A:=(M, T)]]$とすると，この操作は次のような *extend* オペレータに対応する.

$$[R:[A:=(M,T)]] \rightarrow \begin{cases}\text{構造 } T \text{ を生成するオペレータ列}\\ extend(R, A, M)\end{cases}$$

構造 T を生成するオペレータ列は，(2)〜(4)の規則にしたがった構造化オペレータ列となる.

上記の対応づけにしたがった変換アルゴリズム（付録Bを参照）によって視覚的検索言語は，それと等価な代数的な構造化オペレータ列へ一意に変換することができる.

3.3 検索式の変換

本稿で示す変換アルゴリズム（付録B参照）は構造化オペレータ間に次のような変換時における優先順位を設定して，与えられた視覚的検索言語を構造化オペレータの列へ変換する（$A<B$ は A が B より優先順位が高いことを意味する）.

$$generate < apply < (select^1 < extend) < select^2 < project$$

ここで，$select^1$ は *extend* 中で参照される属性値に対する選択操作，$select^2$ はそれ以外の属性値に対する選択操作である.

この順番にしたがったオペレータ列は，まず基底部分となる主体集合を生成（*generate*）してから，後の検索/操作で必要となるデータベース中のデータをすべて主体木中に埋め込み（*apply*），次に，データベースのデータから計算される値を計算し*（*extend*），これらのオペレーションによってすべての属性値を生成した後に，主体木中の条件検索を行い（*select*），最後に不要な属性値を削除する（*project*），という生成手順になるため，最終的に生成される主体木の一意性を保証することができる.

付録Bの変換アルゴリズムによって，検索式がどのような構造化オペレータ列へ変換されるかを具体例を用いて示す.

[Example]

課データ（**Dept**）を基点とし，その属性として課名（**Dname**）とその課に所属する男性従業員の平均給与（**AVESAL**），従業員データ（**Emp**）が並べられ，さらに **Emp** データの属性として従業員番号（**E#**），氏名（**Ename**）があるようなデータを検索することを考える．ただし，**Emp** データのうち，女性であり，かつ給与が各自の所属している課の男性従業員の平均給与以

* *extend* の操作でアクセスする属性値に対する条件検索（$select^1$）だけは，この *extend* の実行前に処理する．さらに，もし計算値を計算するために，別の計算を行わなければならない時には，入れ子状に $select^1$, *extend* の処理を行う.

上の者だけを検索する.

この検索は, 3.1 節の検索式を用いて, 手続き 4 のように記述される.

手続き 4 *Query Procedure* 3

```
[ Dept:[ Dname:,
         AVESAL:@=(ave(Emp'/Salary), [Emp|Emp':@[Salary:@, Sex:@]{Sex="男"}]),
         Emp:[E#:, Ename:, Salary:@, Sex:@]{Sex="女" AND Salary≧../AVESAL}
       ]
]
```

この検索式は, アルゴリズム 1～7 によって, 次のような構造化オペレータへ変換される.

手続き 5 *Query Procedure* 3

1 *generate* (**Dept**)
2 *apply* (**/Dept, Dname, Dname**)
3 *apply* (**/Dept, Emp, Emp'**)
4 *apply* (**/Dept/Emp', Salary, Salary**)
5 *apply* (**/Dept/Emp', Sex, Sex**)
6 *apply* (**/Dept, Emp, Emp**)
7 *apply* (**/Dept/Emp, E #, E #**)
8 *apply* (**/Dept/Emp, Ename, Ename**)
9 *apply* (**/Dept/Emp, Salary, Salary**)
10 *apply* (**/Dept/Emp, Sex, Sex**)
11 *select* (**/Dept/Emp', Sex**= "男")
12 *extend* (**/Dept, AVESAL**, ave (**Emp'/Salary**))
13 *select* (**/Dept/Emp, Sex**= "女" AND **Salary**≧../**AVESAL**)
14 *project* (**/Dept/Emp'/Salary**)
15 *project* (**/Dept/Emp'/Sex**)
16 *project* (**/Dept/Emp'**)
17 *project* (**/Dept/AVESAL**)
18 *project* (**/Dept/Emp/Salary**)
19 *project* (**/Dept/Emp/Sex**)

3.4 視覚的検索言語の書式

本章で提案した視覚的検索言語によって, データベース利用者は, 階層型データの記述式に近い形で検索式を記述することによって, 非手続き的に検索処理を記述でき, さらに階層型データ中の処理対象を明確にし, ひとまとまりの処理 (例えば, 特定の属性に対する様々な処理) をまとめて表記でき, しかも, 検索処理の上下関係等も明示することができる.

しかし, 本稿で提案した視覚的検索言語は, 単にデータベース利用者の検索処理の記述を容易にするためだけのものではない. 構造化オペレータの場合, 個々の検索処理が別々のオペレータに分割されるために各オペレータ間の従属関係や上下関係の検査が複雑になり, 並列実行可能な処理の自動抽出などが難しくなる.

一方, 主体木において, 例えばタプル $[A_1, A_2, \cdots, A_n]$ の各 A_i の間に従属関係が存在しない時, それぞれの属性ごとに並列に処理できることがわかっており, 視覚的検索言語で記述された検索式からであれば, このような主体木の性質による処理の最適化や並列性の抽出をより容易に行える可能性がある.

本稿で提案した視覚的検索言語の特徴の 1 つは,『データベースからユーザが検索しようとするデータを構造化させる手順の記述と検索の際の仮想的なデータの生成の手順の記述とを独立させている』点である. 視覚的検索言語の書式をこのようにしたのは, 1 つには, ある値 A を生成するためにある値 B を参照するが, 一方で, B を生成するためにはまず A を生成しなければならない, というデッドロックのような誤った検索の記述をあらかじめ排除するためである.

さらに, ある属性を生成するために必要なデータの組合せとその計算法をすべて "代入記号(=)" の右辺に "(計算式, 必要な属性)" のようにまとめて書けることから,『実際に自分が検索結果としてどのような構造を生成しようとしているか』ということと『計算値の生成』を完全に切り離して, 考えることができる. もし仮に, 計算値の参照の際に, 自由に属性をアクセスできてしまうと, 検索結果を作り出す主体木の構造の中に, わざわざ離れた位置へ計算にだけ使用される属性を生成しておく必要が出てきてしまう. このような場合, 例えば後でその計算処理が不要になったり, 変更しなければならなくなった場合に, 計算式を削除/変更すると同時に, その計算式中だけで使用されていた属性を検索言語の構造中から見つけ, 同時に削除/変更しなければならない. 一方, 本稿の検索式のように, データベースの検索と仮想的な値の生成の処理の記述が独立であれば, "=" 以降の括弧 "( )" に囲まれた部分だけを削除/変更すればよいことになる.

ところで, 手続き 3 のように, 一度従業員データ (**Emp'**) を生成してから, 一旦削除してまた同じ従業員データ (**Emp**) を生成するのは, 二度手間のように思われるが, 我々は, この問題は検索処理の最適化の問題であると考えている. すなわち, 検索言語レベルでは, 同じデータをわざわざ 2 回生成するように記述していても, 実際の構造化オペレータの処理において 1 回だけ生成すればよい場合には, 検索処理の最適化手法を応用すれば, システムが自動的に判断して 1 回しか実行しない (もしくは, 構造化オペレータを 1 回だけ生成するような形に書き換える) ことは十分可能であると考えている.

また, 1 つの代入式では, 1 つの属性だけしか生成できないために, 同じような属性値をもとに計算を行う場合, 代入式の第 2 引数 (計算に必要なデータの生成

の記述）部分に類似した部分構造を何度も書かなければならないことになる．ただし，このような問題は，マクロなどの手法を導入することによって，検索式の無用な記述の複雑化の解消や検索式の理解しやすさの向上を行うことができると考えられる．

4. おわりに

本論文では，主体と主体間の1引数関数という単純な基本要素だけからなる関数型データモデルをもとで，利用者が階層型データの表現方法に近い形式でデータベース検索を行うことを可能にする視覚的検索言語を提案した．

本検索言語は，構造化オペレータと比較して，データベース利用者が自分が作り出す階層型データをより直観的に把握しながら，データベースへの検索を記述することを可能とする点で構造化オペレータより利用しやすく設計されている．また，3.3節の例題が示すように，従来のデータベース言語では1つの質問式で書けなかったような質問が容易にかけるなど，高い表現力を持っている．

さらに，視覚的検索言語による問合せ手続きを構造化オペレータの手続きへ一意に変換するアルゴリズムについて述べ，本稿で提案した視覚的検索言語を使用しても構造化オペレータを使用した場合と全く同様に自由な階層型データを生成できることをあわせて示した．

なお，本稿の視覚的検索言語は単にユーザへ直観的な問合せ記述手法を提供するだけでなく，ユーザが構造化オペレータを直接記述した場合と比べて，無意味な処理もしくは矛盾を含むようなオペレータ列を生成してしまうことを排除できるという特長を持つ．ここで，本稿で述べたように視覚的検索言語の個々の構成要素が構造化オペレータの各機能とほぼ1対1に対応していることから，逆に健全なオペレータ列を与えればそれと等価な視覚的検索言語が生成できることが期待されるが，この点についての厳密な検証は今後の研究課題である．

参考文献

1) Buneman, P., Frankel, R.E. and Nikihil, R.: An Implementation Technique for Database Query Languages, *ACM Trans. Database Syst.*, Vol. 7, No. 2, pp. 164-186 (1982).

2) Shipman, D.W.: The Functional Data Model and the Data Language, *ACM Trans. Database Syst.*, Vol. 6, No. 1, pp. 140-173 (1981).

3) Batory, D.S., Leung, T.Y. and Wise, T.E.: Implementation Concepts for an Extensible Data Model and Data Language, *ACM Trans. Database Syst.*, Vol. 13, No. 3, pp. 231-262 (1988).

4) 永江尚義, 有澤　博：関数型データベースにおける階層型検索データの生成, 情報処理学会論文誌, Vol. 35, No. 3, pp. 436-443 (1994).

付　　録

A. 視覚的検索言語の文法

視覚的検索言語式をBNF表記法にしたがって記述したものを次に示す．以下では，“< >”は非終端記号を表し，“(a)*”はaの0回以上の繰り返しを表す．また，検索言語中で特別な働きをする文字は，ダブルクォート(”)で括って示す*．

```
<検索式> ::= "[" <主体型>":"<タプル式> "]"
<タプル式> ::= "[" <属性>"," (<属性>)* "]{" <条件式> "}"
<属性> ::= <名称>":" | <名称>":@" |
        <名称>":"<タプル式> | <名称>":@"<タプル式> |
        <名称>":=("<計算式>", ["<属性>", "(<属性>)*"])" |
        <名称>":@=("<計算式>", ["<属性>", "(<属性>)*"])"
<名称> ::= <関数名>"|"<属性名> | <関数名>
<条件式> ::= NULL | <論理式> | <論理式> AND <論理式> |
        <論理式> OR <論理式> | NOT <論理式>
<論理式> ::= <ブール演算子> | "(" <論理式> AND <論理式>")" |
        "(" <論理式> OR <論理式> ")" | "(" NOT <論理式> ")"
<ブール演算子> ::= bool_n(<引数>, <引数>, …, <引数>)
<計算式> ::= func_n(<関数引数>, <関数引数>, …, <関数引数>)
<引数> ::= 定数 | <相対ランク記述子>
<関数引数> ::= <引数> | <計算式>
<絶対ランク記述子> ::= "/"<属性名>("/"<属性名>)*
<相対ランク記述子> ::= ("../")*<属性名>("/"<属性名>)*
<主体型> ::= 文字列
<属性名> ::= 文字列
```

B. 変換アルゴリズム

アルゴリズム1～7は，与えられた検索式を3.3節の手順にしたがって，等価な構造化オペレータへ変換するアルゴリズムである．

アルゴリズムについて説明する前に，以下のアルゴリズム中で使用している記号について簡単に説明を行う．一般に，検索式は次のような一般形で表現される．

$$Q=[F_i|A_i{:}p_iE_i] \quad (i=1, \cdots, N)$$

ただし，Nはタプル構造中の属性の数である．

コロン(“:”)の左側は属性名**である．これに対し，コロン(“:”)の右側は，“@”や“=”という特殊な記号(p_i)とその属性に対する操作(E_i)を表しており，p_iはNULL, “@”, “=”, “@=”のいずれか，E_iはNULL, タプル式，計算式のいずれかである．例えば，図8の**Dname**のように，単にデータベースから関数値を検索する場合には，p_i, E_iはともにNULLとなる．一方，

* $bool_n()$はプログラミング言語で提供されるn引数のブール関数(真/偽のいずれかを返り値として返す関数)を表す．また，$func_n()$はプログラミング言語で提供されるn引数の数値関数を表す．

** 属性名が関数名と同一の場合には，“$F_i|$”は省略される．

AVESALのように計算値を代入する場合には，p_i= “=”, E_i=“(ave(**Emp'/Salary**), [**Emp'**:@[**Salary**:@]])” となる．

なお，本変換アルゴリズムを利用した検索言語変換システムは，UNIX (Sun OS 4.1.3) 上のC言語によってインプリメントされている．

アルゴリズム1 *Transform*1(*Q*)

```
     Input: 検索式
     ▷ 検索式の一般形を Q = [A : E] とおく.
     Output: 構造化オペレータの手続き
 1   begin
 2       output("generate(A)")
 3       if (E = タプル式) then
         ▷ タプル式の一般形を E = [T]{C} とおく.
 4           call Transform2(/A,T)
 5           call Transform4(/A,T)
 6           if (条件式 Cが存在する) then
 7               output("select(/A, C)")
 8           endif
 9           call Transform6(/A,T)
10           call Transform7(/A,T)
11       endif
12   end
```

アルゴリズム2 *Transform*2(*R*, *T*)

```
     Input: ランク記述子,
            タプル式 ([T] = [F_i|A_i : p_iE_i], i = 1〜N)
     Output: 構造化オペレータ (apply)
 1   begin
 2       for i = 1 to N do
 3           if (E_i = NULL) then
 4               call Transform3(R, F_i, A_i)
 5           else if (E_i = タプル式) then
             ▷ タプル式の一般形を E_i = [T_i]{C_i} とおく.
 6               call Transform3(R, F_i, A_i)
 7               call Transform2(R/A_i, T_i)
 8           else if (E_i = 代入式) then
             ▷ すなわち, p_iが "=" もしくは "@=" である.
             ▷ 代入式の一般形を E_i = (M_i, [T_i]) とおく.
 9               call Transform2(R, T_i)
10           endif
11       endfor
12   end
```

アルゴリズム3 *Transform*3(*R*, *F*, *A*)

```
     Input: ランク記述子, 関数名, 属性名
     Output: 構造化オペレータ (apply)
 1   begin
 2       if (主体木中で関数名を変更する) then
         ▷ すなわち, F ≠ A であるとき
 3           output("apply(R, F, A)")
 4       else
 5           output("apply(R, A, A)")
 6       endif
 7   end
```

アルゴリズム4 *Transform*4(*R*, *T*)

```
     Input: ランク記述子,
            タプル式 ([T] = [F_i|A_i : p_iE_i], i = 1〜N)
     Output: 構造化オペレータ (extend, select)
 1   begin
 2       for i = 1 to N do
 3           if (E_i = タプル式) then
             ▷ タプル式の一般形を E_i = [T_i]{C_i} とおく.
 4               call Transform4(R/A_i, T_i)
 5           else if (E_i = 代入式) then
             ▷ すなわち, p_iが "=" もしくは "@=" である.
             ▷ 代入式の一般形を E_i = (M_i, [T_i]) とおく.
 6               call Transform5(R, T_i)
 7               output("extend(R, A_i, M_i)")
 8           endif
 9       endfor
10   end
```

アルゴリズム5 *Transform*5(*R*, *T*)

```
     Input: ランク記述子,
            タプル式 ([T] = [F_i|A_i : p_iE_i], i = 1〜N)
     Output: 構造化オペレータ (extend, select)
 1   begin
 2       for i = 1 to N do
 3           if (E_i = タプル式) then
             ▷ タプル式の一般形を E_i = [T_i]{C_i} とおく.
 4               call Transform5(R/A_i, T_i)
 5               if (条件式 C_iが存在する) then
 6                   output("select(R/A_i, C_i)")
                 endif
 7           else if (E_i = 代入式) then
             ▷ すなわち, p_iが "=" もしくは "@=" である.
             ▷ 代入式の一般形を E_i = (M_i, [T_i]) とおく.
 8               call Transform5(R, T_i)
 9               output("extend(R, A_i, M_i)")
10           endif
11       endfor
12   end
```

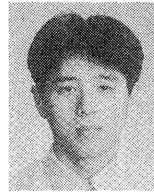
アルゴリズム6 *Transform*6(*R*, *T*)

```
     Input: ランク記述子,
            タプル式 ([T] = [F_i|A_i : p_iE_i], i = 1〜N)
     Output: 構造化オペレータ (select)
 1   begin
 2       for i = 1 to N do
 3           if (E_i = タプル式) then
             ▷ タプル式の一般形を E_i = [T_i]{C_i} とおく.
 4               if (条件式 C_iが存在する) then
 5                   output("select(R/A_i, C_i)")
                 endif
 6               call Transform6(R/A_i, T_i)
 7           endif
 8       endfor
 9   end
```

アルゴリズム 7 *Transform7*(R, T)

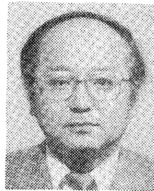
```
    Input: ランク記述子,
           タプル式 ([T] = [F_i|A_i : p_i E_i], i = 1〜N)
    Output: 構造化オペレータ (project)
 1  begin
 2      for i = 1 to N do
 3          if (E_i = タプル式) then
            ▷ タプル式の一般形を E_i = [T_i]{C_i} とおく.
 4              call Transform7(R/A_i, T_i)
 5          else if (E_i = 代入式) then
            ▷ すなわち, p_i が "=" もしくは "@=" である.
            ▷ 代入式の一般形を E_i = (M_i, [T_i]) とおく.
 6              call Transform7(R, T_i)
 7          endif
 8          if (p_i="@" OR p_i="@=") then
 9              output("project(R/A_i)")
10          endif
11      endfor
12  end
```

(平成 6 年 4 月 4 日受付)
(平成 7 年 3 月 13 日採録)



永江　尚義（正会員）

1966 年生. 1989 年横浜国立大学工学部電子情報工学科卒業. 1994 年同大大学院工学研究科博士課程（電子情報工学専攻）修了. 同年（株）東芝入社. 現在, 同社マルチメディア技術研究所に所属. 工学博士. 主として, オブジェクト指向データベース, 知的データベース検索に関する研究に従事.



有澤　　博（正会員）

昭和 23 年生. 昭和 48 年東京大学理学部物理学科卒業. 富士通を経て, 昭和 50 年横浜国立大学工学部, 現在同学部電子情報工学科教授. 工学博士. データベース理論, マルチメディアデータベースシステムを研究テーマとしている. コンパクトエンサイクロペディア情報処理（情報処理学会）編集委員長. 電子情報通信学会会員.