データベースシステム 119−54
（1999. 7. 23）

応用データ自動生成のための変換定義言語

有澤達也† 遠山元道††

関係データベースからの検索結果をLaTeXやHTMLのような応用データに変換するプログラムは，簡単な変換に関しては4GLで記述が可能だが，複雑な変換に関しては汎用プログラミング言語で応用データの種類や用途ごとに作成するため，負担が大きい．本論文では，応用データへの変換プログラムの開発者の負担軽減のために，変換規則の定義言語を提案する．そして一実装として，変換定義言語からSuperSQL処理系のジェネレータを生成するコンパイラについて述べる．

The Translation Definition Language for Automatic Generation of Applicational Data

TATSUYA ARISAWA† and MOTOMICHI TOYAMA††

The easy translation programs from a retrieval result of the relational database into the applicational data, such as LaTeX and HTML, are developed using 4GL. But the hard ones are developed for each kind and each usage of its data using general purpose programming language. And the development makes the heavy burden. In this paper, we propose the translation definition language to express a translation rule in order to reduce the burden of the translation program developers. And we have implemented the prototype of compiler that convert the translation definition language into a generator for the SuperSQL processor.

1. はじめに

情報化社会と呼ばれる現在では，データベースは氾濫する情報の格納場所として多くの場面で利用されている．データベースに格納されたデータは保存や共有を目的としており，その格納された情報を利用するデータベースへの検索が重要となっている．

関係データベースでは，入力するデータを関係に分割することにより，効率良くデータを格納することができる．その反面，関係データベースからの直接の検索結果は二次元のフラットな構造のため，表現力に乏しい．利用者はWWWや表計算ソフト等において，データのセマンティクスに近い直感的な表現形態を望むため，HTMLのような構造表現力のある応用データへと検索結果を変換する必要性がある．

データ変換を行うアプリケーションは，一般には第4世代言語によってエンドユーザにも設計し利用することが可能である．しかし，第4世代言語で定義できる変換は定型的で，少し複雑な変換になると，perlやC++などの汎用プログラミング言語によって作成しなければならない．これらの言語による変換プログラムの開発は非常に負担が大きい．

そこで本論文では，応用データの変換プログラムを記述する開発者の開発および保守の負担軽減のために，階層構造を持ったデータから応用データへの変換規則を表現するための変換定義言語の提案を行う．また，一実装として提案された定義言語からSuperSQL処理系[1)~3)]のジェネレータへと変換を行うコンパイラを作成し，提案した定義言語の有効性を示す．

2. 応用データ

関係データベースでは，検索されたデータは二次元のフラットな表として配置される．このデータは単純にデータのみを並べた形であるため，直接利用者が利用しにくい．そこで，多くの場合，利用者はこの直接の検索結果をより理解しやすい形態へ変換をする作業が必要となる．

この変換されたデータ形態のように，利用者が直接扱う情報をさまざまな規則に従って並べたデータを包

† 慶應義塾大学大学院 理工学研究科 計算機科学専攻
Department of Computer Science, Faculty of Science and Technology, Keio University.
†† 慶應義塾大学 理工学部 情報工学科
科学技術振興事業団 さきがけ研究21「情報と知」領域研究員
Department of Information and Computer Science, Faculty of Science and Technology, Keio University.
PRESTO, JST

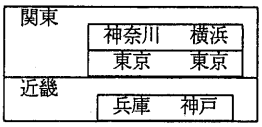

図1　階層型データに対応する LaTeX 応用データの表現例

括的に表す概念のことを，本論文では応用データと位置付ける．応用データには，レイアウトなどの構造情報や表示方法を規定するための装飾情報が，データと共にある規則に従って含まれていて，その規則を理解する処理系を用いることにより直観的にわかりやすい出力結果を得ることができる．

応用データは実に多くのデータ形式が当てはまる．応用データの多種多様な姿は，利用者から起こるニーズに対して，個々に対応した応用データへの変換プログラムを生成することはできる．しかし，このようなプログラムは変換汎用プログラミング言語によって記述しているため，自由な記述によって変換を定義することが可能だが，記述量が多くなり，また応用データの文法規則との対応などは明示的ではない．

3. 階層型データ構造の利用

検索結果から応用データへの変換を行うために，直接装飾情報を付加して作成することも可能である．しかし，関係データベースからの検索結果には結合演算等によって1対多の関連が多数存在するため，フラットな表に表現することによって，検索結果は同じ情報を繰り返し含んでしまう．

そこで，1対多の関連をそのままリスト型にした階層型データ構造で検索結果を表現することを考える．この方法は SuperSQL で利用されている手法で，利用者が検索結果の属性間の関連を明示的に指定することで，リスト構造で検索結果を保持する．

例えば階層型データ

((関東 ((神奈川 横浜) (東京 東京))) (近畿 ((兵庫 神戸))))

に対して，図1の LaTeX の表現は，図2の LaTeX ソースによって実現できる． begin ～ end のブロックに着目し，図2を見てみると，元の階層型データの持っている構造が対応する LaTeX ソースに入れ子状に現れているといえる．つまり，応用データの持っている構造は階層型データの構造と対応づけることができることがわかる．

データ間の位置関係情報は応用データへの変換を行う上で最も重要な情報である．例えば，二つのデータを縦に並べる時と横に並べる時には次の図3の応用データが必要である．(a) と (b)，もしくは (c) と

```
\begin{tabular}[t]{|cc|}
 \hline
 関東 &
  \begin{tabular}[t]{|cc|}
   \hline
    神奈川 & 横浜 \\ \hline
    東京 & 東京 \\ \hline
  \end{tabular}
 \\ \hline
 近畿 &
  \begin{tabular}[t]{|cc|}
   \hline
    兵庫 & 神戸 \\ \hline
  \end{tabular}
 \\ \hline
\end{tabular}
```

図2　階層型データに対応する LaTeX ソース

```
\begin{tabular}[t]{|c|}
 \hline
 神奈川
 \\ \hline
 横浜
 \\ \hline
\end{tabular}
```

(a) LaTeX 縦

```
\begin{tabular}[t]{|c|c|}
 \hline
 神奈川 & 横浜
 \\ \hline
\end{tabular}
```

(b) LaTeX 横

```
<table>
 <td> 神奈川 </td>
 <td> 横浜 </td>
</table>
```

(c) HTML 縦

```
<table>
 <tr>
  <td> 神奈川 </td>
 </tr>
 <tr>
  <td> 横浜 </td>
 </tr>
</table>
```

(d) HTML 横

図3　連結方向による応用データの違い

(d) を比較すると，データの配置方向による相違点を見ることができる．

まず，二つのデータの前後に存在する文字列が異なっていることがわかる．これらは，対を為してレイアウトの骨格になっており，この二つの文字列に囲まれた部分で一つのグループを形成しているのである．つまり，これらの文字列はレイアウト方向毎に前置，後置される部分であるといえる．もう一つ，データ間に存在する文字列が異なっていることがわかる．これはセパレータと考えることができ，レイアウト方向毎に指定される文字列である．これら三つの要素が方向を決定する構成子（コンストラクタ）であると考えられる．

一般的に，SuperSQL ではこれらのレイアウト方向をメディアから抽象し，次元として扱っている．逆に，この次元にメディア個有の情報を与えることで各次元のもつ意味が決定する．例えば，第三次元を LaTeX ではページ変えの意味を持たせる一方，HTML ではハ

```
##class(do-c1)
##cond(c1)
#body
"\\begin{tabular}[]{|"
$repeat("c",$itemnum)
"|}";
$clist("\\hline";,
       "&";,
       "\\\\\\hline";)
"\\end{tabular}";
##end

##class(do-g2)
##cond(g2|g4)
#body
"\\begin{tabular}[]"
"{|c|}";
$glist("\\hline";,
       "\\\\\\hline";,
       "\\\\\\hline";)
"\\end{tabular}";
##end
```

```
##itemclass
#body
IF match("font") THEN
 "{\\" $match[2] " "
 ==PUTITEM
 "}"
ELSE
"\\verb|"
==PUTITEM
"|"
ENDIF
##end

##default(tex)
#head
"\\documentstyle"
"[eclepsf]{article}";
"\\begin{document}";
"\\noindent";
#foot
"\\end{document}";
##end
```

図4　提案する定義言語の記述例

イパーリンクという意味を持たせている．この次元に対する意味付けをメディア抽象と呼ぶ[1]．ここで，LaTeX と HTML のそれぞれ縦方向と横方向の次元に対するメディア抽象による文字列表現をまとめた表を表1に示す．

4. 変換定義言語の設計

前節で述べた，階層型化されたデータから応用データへの変換の規則性に着目して，応用データの構造の抽象化を行い，応用データ生成に特化した変換プログラム定義言語を設計する．本論文で定義の対象とする応用データへの変換プログラムは，関係データベースからの検索結果から生成した階層型データと，データの構造に対応したレイアウトの方向と装飾情報を入力仕様とする．定義言語の文法について，以下で詳細を説明する．また，記述例を図4に示す．

4.1 クラス

応用データには，位置情報を記述するための情報とデータを出力する部分の装飾情報があるということを述べた．レイアウトのブロックがレイアウト情報に対応して入れ子状に配置される．そこで，このブロックごとに一つのクラスを定義する．提案する定義言語では，このクラスを三種類に分類した．この三種類のクラスについて順に述べていく．

4.1.1 固定出力クラス

固定出力クラスでは生成される応用データの最初と最後の部分の定義を行う．このクラスでは，入力データの値にかかわらず生成結果に反映される部分が定義される．例えば，応用データであることを示す文字列や，初期設定や終了設定文字列を記述する．初期設定や終了設定は応用データ毎に必要な設定が異なるため，応用データの種類毎にヘッダとフッタとして一つずつ定義されればよい．

4.1.2 構造記述クラス

構造記述クラスは位置情報の表現に用いるクラスである．階層構造に対応する応用データにおいて位置情報を反映させるには，前置部分と後置部分とセパレータを出力することが必要である．そこで，構造記述クラスをレイアウト方向毎に識別子と共に定義し，三つの情報の記述を行う．

また，同一方向のデータの並べかたとして，連結と反復がある．連結は複数項目を並べるために用いるのに対して，反復は一項目の複数の値を並べる時に用いる．どちらも構造記述クラスとして記述できるが，連結で表示させる場合と反復で表示させる場合では，展開関数の展開の方法が異なっているので，別のクラスとして定義を行う．

4.1.3 値記述クラス

値記述クラスは，階層化されたデータに存在する一データ項目を応用データに変換するために，データ値の装飾文字列を記述するクラスである．データへの装飾に対しては，レイアウト情報に装飾の指定があった時に応用データのどの位置に出力するべきかを，このクラスで条件とともに記述を行う．この値記述クラスは，応用データの種類毎に定まるので，固定出力クラスと同様に応用データの種類ごとに一つずつ定義すれば良い．

4.2 クラス定義内の構成要素

ここでは，それぞれのクラスに記述される内容について述べる．以下に述べる要素を組み合わせて，クラスの定義を記述する．

4.2.1 文字列転記

ダブルクオーテーション（"）によって囲まれた文字列によって，変換時にそのまま転記することを指定する．例えば，「"</table>"」は，応用データに変換される時には常に「</table>」に常に変換される．

また後で述べる展開関数や変数を配置した場合は，その内容が変換プログラム実行時に動的に決定され出力を行う．

4.2.2 展開関数

展開関数は，ダブルクオーテーションで囲まれた文字列とは異なって，そのクラスを参照した時に動的に決定する値を指定するために用いる．提案する定義言

表1　メディア抽象の例

	前置部分	後置部分	セパレータ
LaTeX 縦方向	\begin{tabular}[t]{|c|} \hline	\\ \hline \end{tabular}	\\ \hline
LaTeX 横方向	\begin{tabular}[t]{|c|c|} \hline	\\ \hline \end{tabular}	&
HTML 縦方向	<table><td>	</td></table>	(無し)
HTML 横方向	<table><tr><td>	</td></tr></table>	</td></tr><tr><td>

語では，以下の展開関数を用意した．

- ==putitem

値記述クラスで用いられ，クラスを呼び出した時のデータ値が文字列として与えられる．

- $item

構造記述クラスで用いられ，そのクラスを呼び出した際の階層型データのリストを与える．これは，他の関数の引数として用いるためのものである．

- $clist(*string1*, *string2*, *string3*)
  $glist(*string1*, *string2*, *string3*)

構造記述クラスで用いられ，レイアウト情報に含まれている要素を順に評価し，再帰的に構造記述クラスや値記述クラスを呼び出す．$clist はクラスが連結である時に用い，複数の要素を並べて配置する．$glist はクラスが反復である時に用い，同じ構造を持ったデータのリストを全て連結して配置する．3 つの引数を指定し，それぞれ前置文字列，後置文字列，要素間のセパレータを記述する．

- $itemnum

構造記述クラスで用いられ，クラスが評価された時点で，レイアウト情報に含まれている要素がいくつ存在するかを表す．

- $repeat(*statement*, *integer*)

定義言語の表現 *statement* を *integer* の回数だけ繰り返した文字列に変換する．例えば，LaTeX の表であらかじめ横に並ぶ個数を記述するのに都合が良い．

- $length(*string*)

string の文字列の長さを整数で返す関数．表示する文字数を指定しなければならないとき用いる．

- $newfilename(*prefix*, *ext*), $filename

$newfilename は，新しいファイル名を生成する関数であり，“*prefix* 数字 *ext*” という文字列に変換する．また，$filename は，直前の$newfilename で生成された文字列を指す．

4.2.3　装飾判定ロジック

レイアウト情報には，位置情報だけではなく，装飾のための情報も同時に渡される．装飾判定ロジックは，呼び出したクラスの装飾情報を参照し，その有無にしたがって生成される応用データを変更するための制御構造である．提案する定義言語では，装飾判定ロジックは図 5 の形で記述する．

IF match(*decolation-symbols*)
THEN
装飾が存在した場合の定義
ELSE
装飾が存在しない場合の定義
ENDIF

図5　装飾判定ロジック

decoration-symbols では，装飾のリストを前方一致で指定する．また，ワイルドカードとして “_” 記号を用意する．

装飾が存在した場合，THEN 節の後の記述内で条件に一致した装飾リストの要素を$match 句によって抽出することができる．$match[n] は条件に一致した装飾リストの n 番目の文字列を指す．例えば，$match[2] は条件に一致した装飾リストが「("halign" "left")」であれば「"left"」になる．

4.2.4　変数の定義および参照

変数は文字列の前に「$」を付加したもので，文字列を代入したり，参照して文字列転記や引数に利用することができる．また，数値を代入して四則演算の計算を行うこともできる．

また変数は展開関数$clist や$glist から他のクラスが呼ばれた時に保持している値が呼び出されたクラスでも有効になる．つまり，ブロック全体にかけられた装飾情報をリストの内部へと伝播させることができる．

4.2.5　その他の構成要素

本研究では，今までに述べてきた構成要素以外に以下の要素を利用して記述する．

- ==NEWFILE(*filename*) *statement* ==ENDFILE

応用データの生成先を新しい別ファイルに指定する．*statement* の部分が，ファイル名 *filename* のファイルに生成するように指定するものである．

- ITOA(*integer*)

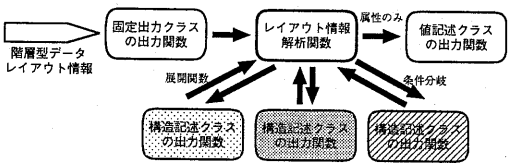

図6　関数の呼び出し関係図

整数 *integer* を文字列に変換する関数である．

4.3　共通部分のクラス化

応用データ生成の変換規則を定義する上で，保守性の高いものを作るためには，同じ定義を繰り返さないことが必要である．そこで，定義の一部を共通化するために共通部分をクラスとして定義し，呼び出す側では，「`CALLF(`*classname*`)`」と記述することで利用する．

4.4　例外関数

これまでの要素で，応用データ変換規則の大部分を記述することができる．しかし，記述が不可能な処理もあり，これに関しては例外関数として扱う．例外関数では，コンパイルして生成される変換プログラムを記述している言語で関数を定義し，定義言語上でその関数を呼び出すために，`CALL` という関数を利用する．

5.　コンパイラの実装

定義言語の記述力を確認するために，UNIX 環境上で lex[7]，yacc[8] 処理系を用いて，定義言語から応用データ生成を行う SuperSQL 処理系のジェネレータ互換の Common Lisp プログラムを生成するコンパイラの実装を行った．

定義言語でのクラスは，それぞれに対して関数を生成し，他にレイアウト情報を解析する関数を生成する．後者の関数では，レイアウトの方向情報を抽出し，構造記述クラスの識別子に一致した時にクラスの関数を呼び出す仕組みになっている．また単独の属性に関しては値記述クラスが呼び出される．これらの関数の依存関係は図 6 のようになる．

また，装飾判定など実装する上で複雑になる一部関数は，Common Lisp 関数群をライブラリとして別に用意し，コンパイラが生成したプログラムに付加することで実装した．これは，共通化により生成プログラムの大きさを抑える．

6.　評価・検討

実装したコンパイラを用いて変換プログラムを生成することで，提案した変換定義言語の有用性についての評価を行う．定義言語の対象とする応用データの種類に関しては，既存の変換プログラムと表現や機能の比較を行うために，SuperSQL 処理系に既にジェネレータが実装されていた “LaTeX” および “HTML” を用いた．そして提案した定義言語と既存のジェネレータとの比較を行い，評価を行った．既存のジェネレータは Common Lisp で記述されているため，この比較は定義言語と汎用のプログラミング言語との比較の一例とみなすことができる．定義言語による仕様の記述は，LaTeX に関しては既存のジェネレータの仕様をほぼ記述したが，HTML に関しては既存のジェネレータから文字と表組みに対する装飾を中心に記述し，いくつかの関数は記述を行っていない．

6.1　記 述 量

作成した LaTeX と HTML の定義内容を用いて，実装したコンパイラによってジェネレータを生成した．その結果，これら二つの応用データの定義記述のサイズと定義記述から生成されたジェネレータのサイズは**表 2** の値になった．

定義言語で記述した仕様は，既存のジェネレータと全く同じものではないため，これらの値を直接比較することはできないが，多機能のジェネレータをわずかな定義文から生成できることがわかる．

表 2　定義言語およびジェネレータのサイズ

		行数	バイト数
LaTeX	提案する変換定義言語	89	2244
	コンパイル後のジェネレータ	431	11528
	既存のジェネレータ	444	16109
HTML	提案する変換定義言語	104	2918
	コンパイル後のジェネレータ	536	13912
	既存のジェネレータ	813	29661

6.2　変換の定義の記述力

定義言語はその変換プログラムの機能を十分に記述できることが必要である．提案した定義言語は，基本的な機能を記述するのに十分な要素を含むよう設計された．しかしながら，全ての機能が直接記述できるわけではない．そこで，変換方法の定義の記述が難しい点について検討を行う．

(1)　複雑な計算処理

例えば，座標の計算等を行うためには，汎用の繰り返しや，算術関数が必要である．しかしながら，算術関数の実装はほとんどなされていない．この部分に関しては，コンパイラが生成する言語でのサポートが必要であると考えられる．

(2)　複雑なファイル操作

例えば，複数の部分で生成されたものを 1 つのファイルにまとめる場合，ファイル名を保存することが必要

である．この際，変数を用いて実装することも可能であるが，複雑になってしまう．

(3) 複数属性の同時処理

関数の実装を行う際には，複数属性を同時に利用する可能性がある．例えば，x 軸と y 軸を決めて点を打つような関数が挙げられる．定義言語では，レイアウト情報内のリスト項目を一つずつ再帰的な処理を行う操作しか定義されていないため機能が十分でないといえる．

しかし，これらの部分は全体からみると非常に少数である．また，例外処理によって間接的に変換方法を定義を行うことが可能である．

6.3 保守・修正

まず，変更場所の読みやすさについて述べる．既存のジェネレータでは，順序はほとんど保存された形式で記述はされているが，文字列部分が他のコマンド内に埋まっており，全体を見ることは困難である．また，複雑な組合せで関数が利用されることで，その解析は容易ではない．それに対して，提案した定義言語ではほとんどが出力文字列で構成されていることに加え，またほとんどが単純なコマンドを単独で用いているため，全体の構成を解析が容易になることが期待される．

次に，応用データの仕様の変更に対する修正の容易さについて述べる．特に，装飾指定の追加の作業に関しては，専用のコマンドが用意されており，提案した定義言語は修正が容易である．逆に，複雑で細かい処理に対しては，定義言語は例外処理に頼らなければならないため，不利であるといえる．

最後に，入力データと変換プログラムとのインタフェイスの変更に対する保守について述べる．既存のジェネレータでは，各応用データ仕様に対応して ad hoc に開発されており，関係データベースからのデータと変換プログラムとのインタフェース部分は仕様に沿ったものをそれぞれの開発者が作成している．この状況下で入力のインタフェイスの仕様変更があると，全てのジェネレータの対応が必要となる．それに対して，提案した定義言語はプログラミング言語に依存していないため，応用データの定義に手を加える必要はなく，コンパイラの変更を行うだけで全ての応用データの定義が同時に新しい仕様に適合できる．極端なことを言えば，プログラム言語の変更さえも，コンパイラの作成のみで対応できる．

7. おわりに

本論文では，応用データの自動生成プログラムを開発する開発者の負担軽減のために，多くの応用データが階層型の構造を持っていることに着目し，階層型データから応用データへの変換規則を表現するための変換定義言語の提案を行った．定義言語の仕様としては，階層型に応じたブロックごとの定義と応用データ生成に特化したコマンド群を挙げた．

この変換定義言語から SuperSQL 処理系のジェネレータへのコンパイラの実装を行い，評価を行ったところ，定義における記述量を小さくすることができることを確認し，定義の可読性が向上することや応用データの仕様の変更に対する定義の変更が容易になることが期待できる．

参考文献

1) Motomichi Toyama: SuperSQL: An Extended SQL for Database Publishing and Presentation, *Preceedings of ACM SIGMOD '98 International Conference on Management of Data*, pp. 584-586 (1998).
2) SuperSQL HOME PAGE, http://www.db.ics.keio.ac.jp/ssql/index.html
3) T. Seto, T. Nagafuji and M. Toyama: Generating HTML Sources with TFE Enhanced SQL, *Proc. ACM Symp. on Applied Computing(SAC '97)*, pp. 96-105 (1997).
4) Aho, Alfred V., Ravi Sethi, and Jeffrey D. Ullman: *Compilers: Principles, Techniques, and Tools*, Addison-Wesley (1986).
5) William M. McKeeman, James J. Horning and David B. Wortman: *A Compiler Generator*, Prentice-Hall, Inc.(1970).
6) Greger Lindén, Henry Tirri and A. Inkeri Verkamo: ALCHEMIST: A General Purpose Transformation Generator, *SOFTWARE - PRACTICE AND EXPERIENCE*, VOL.26, pp. 653-675 (1996).
7) S.C. Johnson: Yacc—Yet Another compiler compiler, tech. rep., Bell Telephone Laboratories (1975).
8) M.E. Lesk and E. Schumidt: Lex—a lexical analyzer generator, tech. rep. Bell Telephone Laboratories (1975).