第 3 章
制御構造変換にもとづく ベクトル化 — 1 リスト処理

要旨

この章では，ベクトル計算機を使用してリスト処理を高速に実行するためのプログラム変換にもとづいた基本戦略を提案する．この戦略は，リストを使用した解探索のベクトル処理やその他のベクトル記号処理のためのプログラムを，ユーザにとって自然なかたちに記述されたプログラムから変換・生成することを可能にするものである．この戦略は，Fortran プログラムのベクトル化技法の拡張とかんがえることができる「くりかえし構造の交換」と「くりかえし構造の 1 重化」というプログラム変換技法にもとづいている．変換結果のプログラムにおいては，複数のリストを要素とするベクトルを使用する．それらに関する操作をベクトル計算機のリスト・ベクトル処理機能や数列生成機能などをもちいて実現する．

上記の戦略は，この章においてはまだ形式化されていないため自動的におこなうことはできないが，これをエイト・クウィーン問題の Prolog プログラムに手動で適用し，ベクトル計算機 S-810 においてスカラ処理の約 9 倍の実行速度をえた．

3.1 はじめに

リスト (linked list) は記号処理においてもっとも重要なデータ構造であり，Lisp や Prolog などの記号処理用言語において欠かせない．したがって，記号処理応用プログラムの高速化および記号処理用言語の高速処理のためには，リスト処理の高速化がもっとも重要な課題である．

リスト処理高速化の方法としては，最適化コンパイラによって逐次計算機用の高速なプログラムを生成するという方法もある．しかし，根本的な高速化のためには，並行処理むきハードウェアの使用が必須である．並行処理むきハードウェアを使用する高速化方法としては，つぎのようなものがかんがえられる．

(1) **実行時の負荷分散にもとづく方法**

複数の処理装置をもつ並列計算機において，並行処理可能な部分処理を実行時にことなるプロセッサに負荷分散し，実行する．各処理装置がまったくことなる処理をおこなうので，MIMD 型並列計算機に適した方法である [Sakai 86, Kanada 87].

(2) **コンパイル時のプログラム変換にもとづく方法**

コンパイル時に並行処理可能な部分をみつけ，ベクトル計算機用または並列計算機用のプログラムを生成し実行する．MIMD 型並列計算機においてもこの方法は利点があるだろうが，逐次処理むきのプログラムをそのままでは実行できない HITAC S-820 や Cray-2 のようなベクトル計算機や，SIMD 型並列計算機に適した方法である．実行時の負荷分散オーバヘッドがない点が (1) より有利だとかんがえられる．

(3) **プログラミング時のアルゴリズム改良にもとづく方法**

プログラマが並行処理むきのプログラムを記述する．どのようなアーキテクチャの計算機にも有効な方法だが，かなりアーキテクチャに依存したプログラミングが要求される．DO 文のかわりに並列くりかえし文をつかうというようなかんたんな改良で目的が達せられるばあいもある．しかし，おおくのばあい，単純なリコーディングなどでは十分な性能はえられず，アルゴリズムの再検討が要求される [Shapiro 84]．たとえば第 2 章でしめした解探索のベクトル処理方法は，逐次処理のプログラムに対してアルゴリズム改良をおこなった結果えられたものとかんがえられるが，両者のプログラムを比較すればわかるように，その対応は単純ではなく，逐次処理のプログラムからベクトル処理のプログラムをみちびくのは容易ではない．

これらの方法はいずれも研究途上にあり，どの応用においてどの方法が成功するかは現在のところわからない．この論文であつかうのは (2) の方法である．この方法は実行時

の負荷分散オーバヘッドがないため (1) より高速に処理できる可能性があり，また (3) におけるほどユーザに負担をかけることがないという利点がある．したがって，有望な方法だということができる．

これまで，ベクトル計算機はほとんど数値計算専用機とかんがえられてきた．また，現在でも Fortran がベクトル計算機で使用できるほとんど唯一の言語である．したがって，ベクトル計算機のためのコンパイル技術あるいはベクトル化技術は，数値計算プログラムにおいて，また Fortran において発展してきた．

ベクトル計算機は，ベクトル処理を高速におこなうことができる専用ハードウェアと，それに関する命令 (ベクトル命令) とをもっている．ベクトル計算機の Fortran コンパイラにおいては，DO ループ内の演算の実行順序を変更して同種の演算をまとめることにより，それを 1 個のベクトル命令で実行するようにする．これによって，それらの演算をベクトル処理専用ハードウェアで実行することを可能にしている．この演算順序の変更をベクトル化とよんでいる．

しかし，Fortran において発展してきたベクトル化技術をそのまま記号処理プログラムおよび記号処理用言語に適用するだけでは，リスト処理プログラムのベクトル化は実現できない．その理由をのべる．

Fortran コンパイラにおいては，プログラム中のループを検出し，そのうちの可能なものをベクトル化する．Fortran プログラムにあらわれるデータ構造は配列であり，ベクトル化に適するかどうかの判定は配列添字の比較をベースにしておこなわれる．一方，記号処理用言語で記述されたリスト処理プログラムにおいては，第 1 に制御構造としては，ループが存在せず，再帰よびだしがつかわれることがおおい．また，第 2 にデータ構造としては，配列はあらわれず，再帰よびだしごとに独立のリストが処理されることがおおい．したがって，配列添字の比較をベースにしたベクトル化法では，まったく対処できない．

しかも，リスト処理のプログラムは，もとのままでは本質的にベクトル処理に適さない構造のプログラムであることがおおい．ベクトル処理に適さないのは，つぎのような理由による．各要素の処理のあいだにデータ依存性があるとベクトル処理はできないが，リスト処理プログラムの最内側ループは，通常，リストの各要素を順に処理していくため，データ依存性がある．また，最内側ループがみじかすぎるためにたとえベクトル化できても加速されないばあいもある[注1]．したがって，あらたなベクトル化法を開発しなければ，ベクトル計算機によるリスト処理を実現することはできない．

3.2 節では，準備として，Fortran コンパイラでつかわれている 3 つの基礎的なベクトル化技術について説明する．

[注1] 3.4 節でしめすエイト・クウィーン問題のプログラムにおける手続き `not_take1` はその例のひとつである．

3.3 節では，プログラム変換にもとづくベクトル・リスト処理の戦略についてのべる．3.4 節では，変換後のプログラムでつかわれるリスト処理基本演算のベクトル処理方法についてのべる．3.5 節では，エイト・クウィーン問題のプログラムにおける変換の適用例についてのべる．3.6 節でそのプログラムに関する実測結果をしめす．関連研究については次章でのべる．

3.2 ベクトル化の基礎

この論文でのべるリスト処理プログラムのベクトル化においても，Fortran におけるベクトル化技術が基礎となる．したがって，この節ではベクトル化の基礎についてかんたんに説明する．

ベクトル化は一種のプログラム変換とみなすことができる [Yasumura 87]．ベクトル化はつぎのようなプログラム変換のくみあわせである．

(1) **ループ分散** (loop distribution)

ベクトル命令は，ロード，加算，ストアなどの 1 個の基本操作だけを，あたえられたベクトルの各要素におこなう微小なループとみなすことができる．DO ループをこのような微小なループに分割するプログラム変換を，ループ分散という [Kuck 81].

(2) **ループ交換** (loop interchange)

ベクトル化に適さないループをベクトル処理可能にするためかまたはよりたかい実行性能をえるためにおこなわれ，またループ 1 重化はおもに実行性能の向上のために Fortran のプログラムのなかには，ループ分散だけではループ内の一部の操作をベクトル命令に変換できないプログラムもある．このようなばあいには，外側ループとのいれかえ (くりかえし方向の変更) をおこなって最内側ループの全体をベクトル化可能にするというプログラム変換がおこなわれることがある．この変換はループ交換とよばれる [Allen 84, Wolfe 86]．ループ交換は，後述するループ 1 重化と同様に，ベクトル長をのばすことを目的としておこなわれることもある．例を**図 3.1** (1) にしめす[注2].

(3) **ループ 1 重化** (loop unrolling)

ベクトル計算機においては，最内側ループのくりかえし回数すなわちベクトル長をのばすことが性能向上につながる．したがって，多重ループを一重ループにする変換がおこなわれる．この変換はループ 1 重化とよばれる [Tsuda 85]．例を図 3.1 (2) にしめす．

これらの変換のうちループ分散はプログラムをベクトル処理可能にするために必須の変換であり，ループ交換およびループ 1 重化は上記のように，ばあいによっておこなわれる．

[注2] かんたんな例にするため，変換前にもベクトル化可能なプログラムをしめしている．

```
   DO 20 I=1,N
     DO 10 J=1,M
       A(I,J)=A(I,J-1)+B(I,J)
10   CONTINUE
20 CONTINUE
```

```
   DO 10 J=1,M
     DO 20 I=1,N
       A(I,J)=A(I,J-1)+B(I,J)
20   CONTINUE
10 CONTINUE
```

(1) **ループ交換**

```
   DO 20 I=1,N
     DO 10 J=1,N
       A(J,I)=B(J,I)+C(J,I)
10   CONTINUE
20 CONTINUE
```

```
   DO 10 IJ=0,N*N-1
     I=IJ/N+1
     J=MOD(IJ,N)+1
     A(J,I)=B(J,I)+C(J,I)
10 CONTINUE
```

(2) **ループ 1 重化**

図 3.1　Fortran におけるループ交換とループ 1 重化

3.3 ベクトル・リスト処理の戦略

この節では，まず，我々が提案するリストのベクトル処理戦略におけるプログラムの処理手順をしめす．つづいて，そこでつかわれる個別のプログラム変換技法について説明する．最後に，この戦略における可変長リストのあつかいについて，とくに説明する．

3.3.1 プログラムの処理手順

このベクトル処理戦略では，およそつぎのような手順でリスト処理をおこなうことをめざす．

(A) プログラミング

プログラマが，記号処理用言語などで自然な (すなわち機械に依存せず過度に並列処理を意識しないですむ) リスト処理プログラムを記述する．この点が，3.1 節でのべたプログラミング時のアルゴリズムの改良にもとづく方法とはおおきくことなる．

(B) プログラム変換

上記のプログラムは，通常そのままではベクトル処理できないので，プログラム変換によってベクトル計算機で処理可能なプログラムに変換する．変換はコンパイラまたはプリプロセッサによっておこなうのがのぞましい．我々が開発した S-810 のための Prolog コンパイラ・プロトタイプにおいては一部のプログラムを自動的に変換することができるが，現在は自動的にプログラム変換できる範囲は非常にかぎられている．したがって，たいていのばあいはプログラマが変換をおこなう必要がある．しかし，ばあいによってはユーザ・オプションというようなかたちでユーザのたすけをかりるにしても，基本的にすべてを自動的に変換することが目標となる．

(C) 実行

変換後のプログラムをベクトル計算機で実行する．ただし，変換後のプログラムが原始プログラムとしてあたえられるばあいは，実行のまえにコンパイルする必要がある．

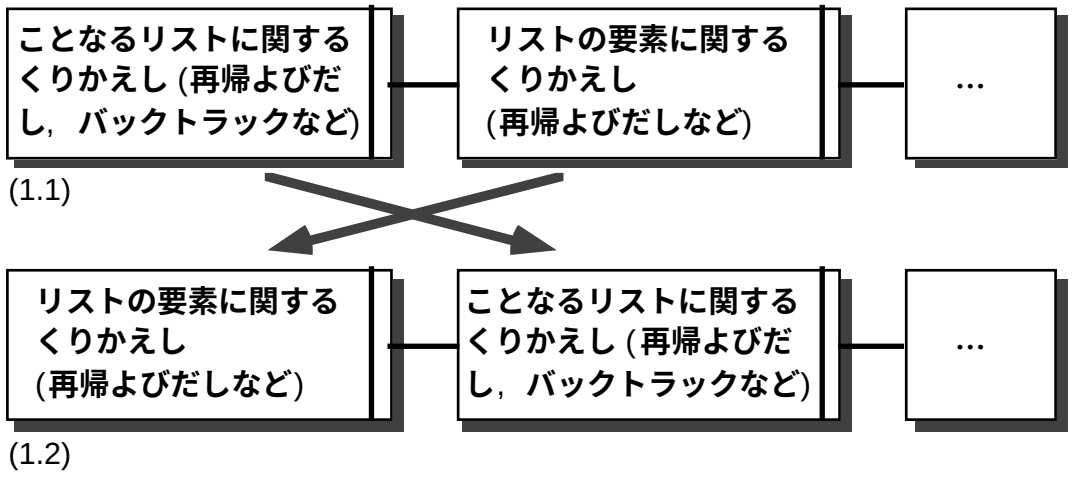
(B) におけるプログラム変換は，3.2 節で説明した Fortran のベクトル化よりこみいっているが，その拡張とかんがえることができる．このプログラム変換はつぎのような 2 種類の変換のくみあわせである．

(1) **くりかえし構造の交換**
(2) **くりかえし構造の 1 重化**

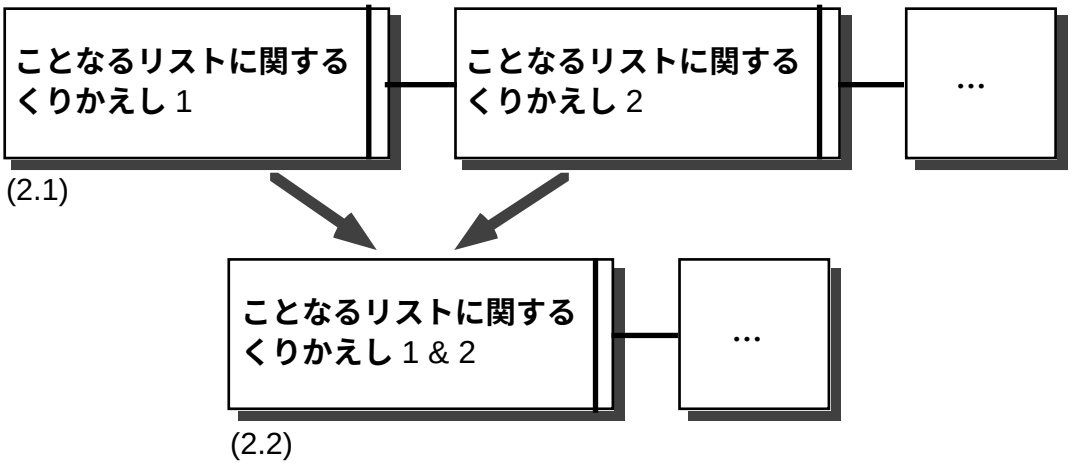
これらについて，つづく 2 つの節で説明する．これらの節では，かんたんのため固定長のリストをあつかう．可変長リストのあつかいについて 3.4 節でのべる．

3.3.2 くりかえし構造の交換

くりかえし構造の交換とは，多重のくりかえし構造における内側のくりかえしと外側のくりかえしとをいれかえることによってベクトル処理できないプログラムをベクトル処理可能にするプログラム変換のことである．**図** 3.2 (1) に例をしめす．くりかえし構造の変換によって，図 3.2 (1.1) に PAD (Problem Analysis Diagram) をつかってしめしたようなプログラムが図 3.2 (1.2) にしめしたようなプログラムに変換される．



(1) **くりかえし構造の交換**



(2) **くりかえし構造の** 1 **重化**

図 3.2　**リスト処理におけるくりかえし構造の交換と** 1 **重化**

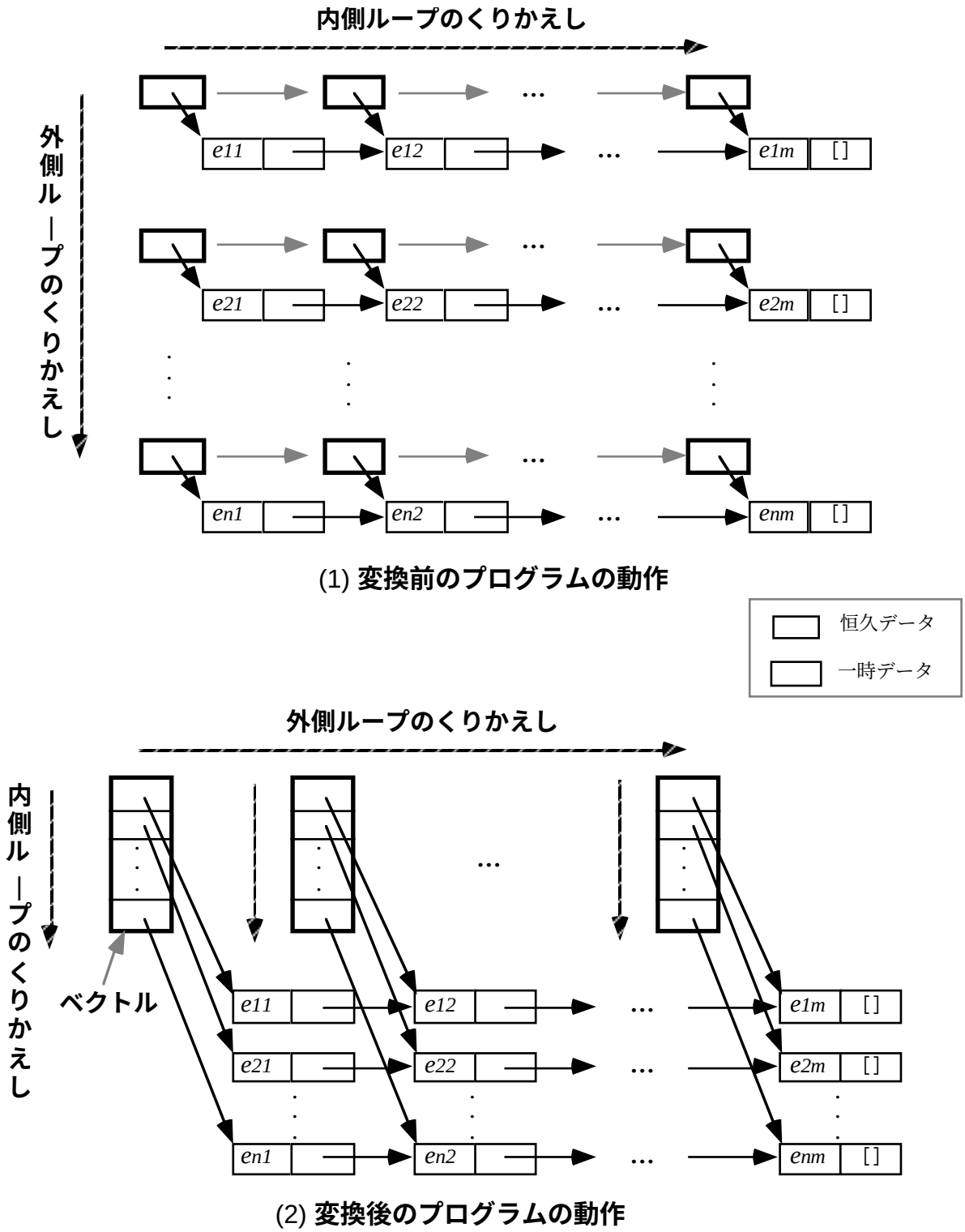


図 3.3　くりかえし構造の変換によるベクトルの生成とくりかえし方向の変化

対象となるくりかえし構造は，ループ，末尾再帰よびだし (tail recursion)，あるいは，自動バックトラックをひきおこす一種のくりかえし構造である．Prolog プログラムに関していえば，再帰よびだしによって内側のくりかえしが形成され，再帰よびだしやバックトラックによって外側のくりかえしが形成されている．

図 3.3 はくりかえし構造の交換前のプログラムの動作と，交換後のプログラムの動作とを対比してしめしている．図 3.3 において，細線であらわしたデータは操作対象である複数のリストであり，太線であらわしたデータは処理過程で一時的につくられるデータである．図 3.3 (1) のように，もとのプログラムの内側のくりかえしはリストの要素に関するくりかえしであって，リスト要素間のデータ依存性のためにベクトル処理できない．しかし，交換の結果，図 3.3 (2) のように，内側のくりかえしがことなる (データ依存性がない) リストに関するくりかえしとなり，ベクトル処理可能になる．

くりかえし構造の変換は，図 3.1 (1) にしめしたループ交換の拡張とかんがえることができる．基本演算のベクトル処理方法については 3.4 節でくわしくのべ，くりかえし構造の交換の例は 3.5 節でしめす．

3.3.3 くりかえし構造の 1 重化

くりかえし構造の 1 重化とは，多重のくりかえし構造を 1 重のくりかえし構造に変換するプログラム変換のことである．図 3.2 (2) に 2 重のくりかえし構造の 1 重化を図示する．くりかえし構造の 1 重化によって，図 3.2 (2.1) にしめしたようなプログラムが図 3.2 (2.2) にしめしたようなプログラムに変換される．

Prolog プログラムに関していえば，再帰よびだしやバックトラックによって多重のくりかえしが形成されるばあいに，くりかえし構造の 1 重化の対象になりうる．この変換によってスカラ処理 (ベクトル長が 1 のベクトル処理) をベクトル処理に変換したり，ベクトル長をのばしてベクトル計算機による実行速度を向上させたりすることができる．

この変換は，図 3.1 (2) にしめした Fortran のベクトル化における多重ループの 1 重化の拡張とかんがえることができる．例は 3.5 節でしめす．

3.3.4 可変長リストのあつかい

3.3.2 節の説明においては，かんたんのためにベクトルの要素である各リストのながさをひとしくした．しかし，リストは一般に可変長であるから，各リストのながさがことなるばあいでも変換後のプログラムがただしく動作するようにしなければならない．

図 3.2 (1.2) の外側のくりかえしにおいて，ながさのことなるリストをあつかうばあい，すべてのベクトル要素について一定回数 n 回の処理をおこなうとすれば，つぎのような不都合が生じる．第 1 に，ながさが n をこえるリストについては処理されない要素がのこる．第 2 に，ながさが n 未満のリストについては空リスト (*nil* または `[]`) を分解しようとして，あやまった処理がおこなわれる．

このような不都合をさけ，ちょうど必要なだけの回数の処理をおこなうようにするためには，ベクトル計算機に用意されている 3 種類の条件制御機構のうちのいずれか，またはこれらをくみあわせてつかえばよい．それらは，マスク演算処理機構，リスト・ベ

クトル処理機構，圧縮・伸長処理機構である [Kamiya 83]．いずれの条件制御機構をおもに使用するかによって，可変長リストをあつかう方法もつぎの 3 種類にわけられる．

(1) **マスク演算方式，**
(2) **インデクス方式，**
(3) **圧縮方式．**

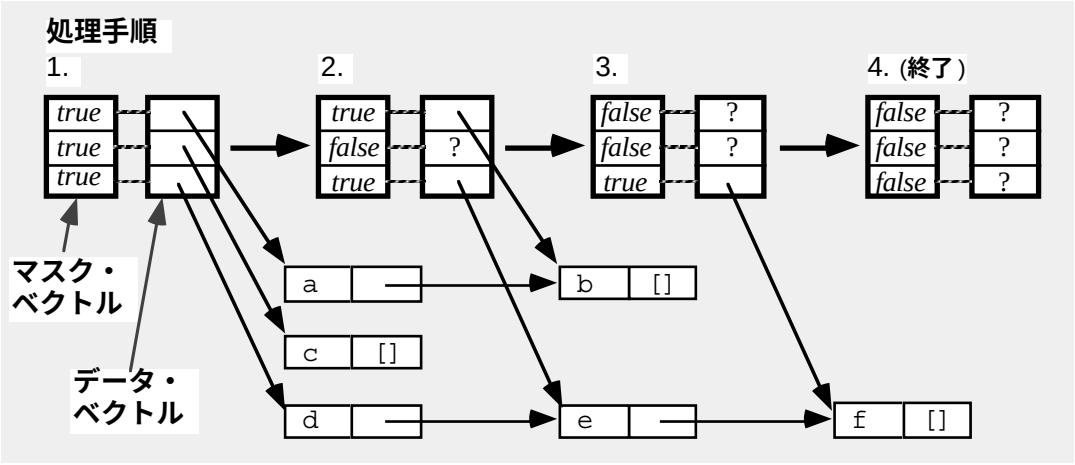
これらの方法を**図** 3.4 をつかって説明する．操作すべきリストへのポインタを要素とするベクトルをデータ・ベクトルとよぶ．可変長リストの処理のばあいにかぎらないが，いずれの方法においてもデータ・ベクトルはリスト処理の過程で一時的につかわれるベクトルであり，恒久的なデータとしてはスカラ処理におけるのとまったくおなじ形式のリストがつかわれる．すなわち，くりかえし構造の変換によって恒久データの形式は変換されることがない．

図 3.4 (1) はマスク演算方式によるリストのアクセス方法をしめす．この方式では各データ・ベクトルの要素が有効かどうかをあらわす論理値をふくむベクトルを使用する．このベクトルをマスク・ベクトルとよぶ．図 3.4 (1) では 3 個のリストを並列に処理するばあいのマスク・ベクトルとデータ・ベクトルの変化をしめしている[注3]．この図においてはデータ・ベクトルは 1 個しかあらわれないが，一般には複数個のデータ・ベクトルが使用され，いずれも同一のマスク・ベクトルの支配をうける．マスク・ベクトルの各要素が，データ・ベクトルの対応する要素 (同一添字要素) が有効かどうか，すなわちその要素に対して処理をおこなうべきかどうかをあらわしている．これに対して，処理のひとつの時点におけるマスク・ベクトルは，基本的にはただ 1 個である．したがって処理中につくられる全データ・ベクトルのベクトル長は唯一のマスク・ベクトルのベクトル長にひとしい．

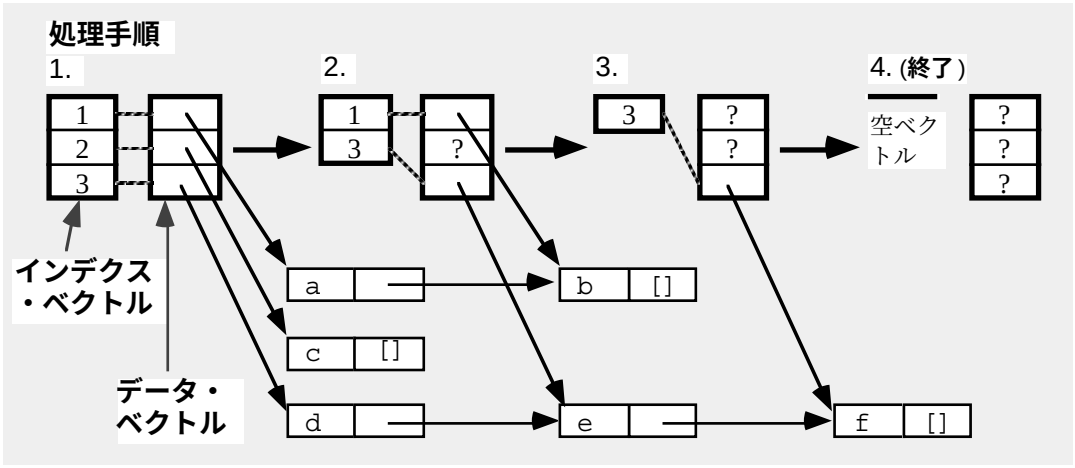
最初はマスク・ベクトルの全要素の値を *true* とする．処理がすすむとデータ・ベクトルの第 *i* 要素に関する処理がリストの末尾に達するので，マスク・ベクトルの第 *i* 要素の値を *false* とする．マスク・ベクトルの対応する要素の値が *true* であるようなデータ・ベクトル要素に関してだけ処理をおこなう．処理がすすむにつれてマスク・ベクトル中には *false* を値とする要素が増加する．マスク・ベクトルの全要素の値が *false* になったとき，処理を終了する．このように処理すべき要素がなくなったときに可変長データのベクトル処理を終了する可変長ループのベクトル処理方法を**残存要素検出法**とよぶ．残存要素検出法およびそれ以外の可変長ループ・ベクトル処理方法については次章でくわし

[注3] 後述する論理型言語のばあいには，破壊的代入ができないために処理の各ステップでつかわれるベクトルはそのたびごとに再生成される．しかし，これらのベクトルはそもそも一時データであるから，最適化処理系においては記憶わりあてをする必要は通常はなく，ベクトル・レジスタ上だけに存在する．そのばあいには，ベクトル計算機のデータフロー的な性質上，ベクトルの再生成はオーバヘッドにはならない．

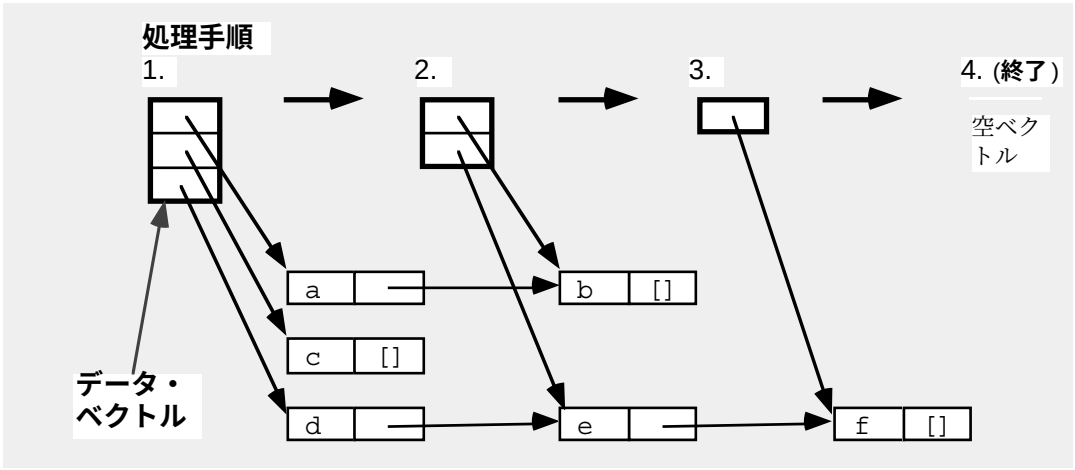
くのべる.



(1) **マスク演算方式**



(2) **インデクス方式**



(3) **圧縮方式**

図 3.4　3 **種類の条件制御方式によるリストの処理方法**

図 3.4 (2) はインデクス方式によるリストのアクセス方法をしめす．この方式では，各データ・ベクトルを直接にアクセスするのではなく，データ・ベクトルの有効要素のインデクス (または変位) をふくむベクトルをつうじてアクセスする．このベクトルをインデクス・ベクトルとよぶ．図 3.4 (2) では，図 3.4 (1) とひとしいリスト処理を例として，インデクス・ベクトルとデータ・ベクトルとの変化をしめしている．マスク演算方式のばあいと同様に，この図においては 1 個のデータ・ベクトルだけが使用されているが，一般には複数個のデータ・ベクトルが，基本的には唯一のインデクス・ベクトルによって支配される．

インデクス方式においては，インデクス・ベクトルから指示される要素だけが処理対象となる．すべてのデータ・ベクトルのベクトル長はひとしく，処理の途中でデータ・ベクトルのベクトル長は変化しないが，インデクス・ベクトルからは処理が終了した要素が除去されていく．したがって，最初は両者のベクトル長はひとしいが，しだいにインデクス・ベクトルのベクトル長は短縮する．そのベクトル長が 0 になったとき，処理を終了する．この方法は，残存要素検出法をインデクス方式に適用したものだということができる．

図 3.4 (3) は圧縮方式によるリストのアクセス方法をしめす．この方式では，データ・ベクトルはつねに有効要素だけをふくむ．そのために，処理が終了した要素が生じるたびに，その要素はデータ・ベクトルから除去されていく．マスク・ベクトルやインデクス・ベクトルのような制御のためのベクトルは基本的に使用されない．そして，すべてのデータ・ベクトルのベクトル長はつねにひとしく，処理がすすむにつれて同期して変化する．このばあいはベクトル長が 0 になったときに処理を終了する．これも残存要素検出法の応用である．

金田 [Kanada 85] でものべているように，これらの方式にはそれぞれつぎのような利点・欠点がある．

(3) 圧縮方式

圧縮方式はマスク・ベクトルやインデクス・ベクトルのような制御のためのベクトルを使用しないため，原理的には他の 2 方式にくらべて単純だということができる．しかし, つぎのような欠点がある. 図 3.4 (3) においては各ステップでただ 1 つのデータ・ベクトルがつかわれているが，一般には複数のデータ・ベクトルがつかわれる．このようなばあい，圧縮方式による実行においては，全データ・ベクトルをいっせいに圧縮しなければならない．なぜなら，そうしないと各ベクトル間での要素の対応がくずれて，対応する要素をみつけることができなくなるからである．同時にあつかうデータの量がおおいばあいには，このベクトル圧縮のオーバヘッドは無視できない．

(2) インデクス方式

インデクス方式においては圧縮方式のようにデータ・ベクトルの数だけのベクトル圧縮をおこなう必要はなく，1 個のインデクス・ベクトルだけを圧縮すればよい．しかし，データ・ベクトルの要素を間接アドレスでアクセスするため，それを処理のためにベクトル・レジスタにロードするのに時間がかかるという欠点がある．また，データ・ベクトルがむだな要素をふくんだまま使用されるため，圧縮方式にくらべて記憶効率がわるいということができる．

(1) マスク演算方式

マスク演算方式においてはベクトルを圧縮する必要はいっさいない．しかし，マスク演算方式においては，処理のために基本的にマスク・ベクトルのベクトル長に比例する時間がかかるため，マスク・ベクトルの要素に *false* がおおくなるとむだな処理がおおくなるという欠点がある．また，インデクス方式と同様にデータ・ベクトルがむだな要素をふくんだまま使用されるため，圧縮方式にくらべて記憶効率がわるいということができる．

このように各方式は一長一短であり，いずれの方式がよいかはばあいによる．

ところで，上記のプログラム変換戦略はプログラム変換の方針をあたえているだけであり，具体的な変換方法はくりかえし構造がループ，再帰よびだし，バックトラックのいずれによって形成されているか，などによってことなり，それぞれ具体的な変換方法をみいだす必要がある．プログラム中に 2 重以上のくりかえし構造があることがこれらのプログラム変換を適用するための必要条件だが，それは十分条件ではないから，それをみたすすべてのプログラムがベクトル化可能な構造に変換可能なわけではない．たとえば，部分的に共有された複数のリストの一部をかきかえる処理などは，プログラム変換できない．**図 3.5** にこの理由でベクトル化できない処理の例をしめす[注4]．しかし，おおくのリスト処理のプログラムは多重のくりかえし構造をもっていて，外側のくりかえしごとにことなるリストを処理するので，この戦略の適用範囲はかなりひろいものとかんがえられる．またこの戦略は，可変長リストだけではなく他の可変長データ構造にも適用することができる．

[注4] このような処理のベクトル化法に関しては第 5 章でのべる．

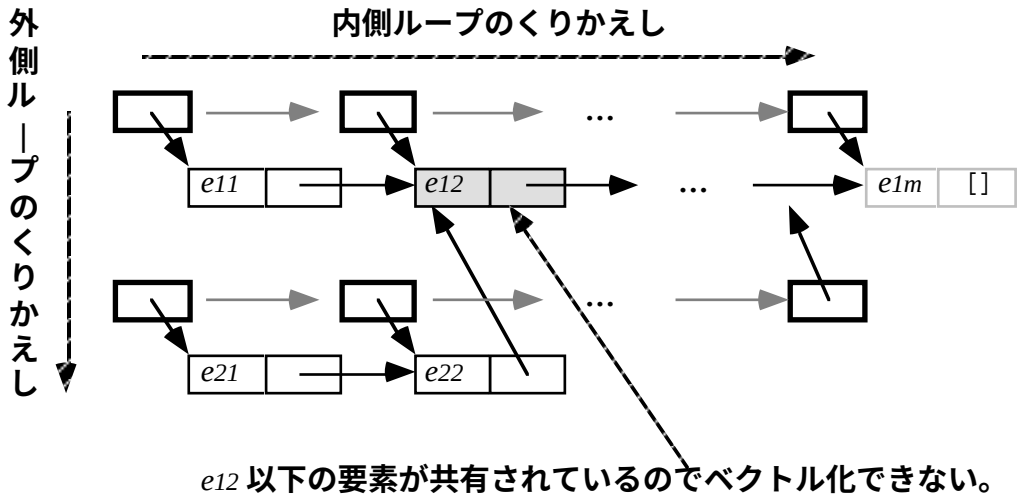


図 3.5　2 重のくりかえしがあるがベクトル化できない処理の例

3.4 リスト処理基本演算のベクトル処理法

3.5 節で，上記の戦略にもとづく具体的なプログラム変換例をしめすが，それにさきだって，変換後のプログラムにおけるリスト処理のベクトル処理方法について説明する．リスト処理における基本的な演算として，データ型の判定，リストの分解，リストの合成があげられる．Lispでいえば null, car, cons などの関数によっておこなわれる演算である．変換後のプログラムではデータ・ベクトルの各要素に対する基本演算をまとめておこなう．すなわち，最内側のくりかえしは複数のリストに関する基本演算のくりかえしになる．このループは，ベクトル計算機 S-810 などにおいてはベクトル処理可能である．

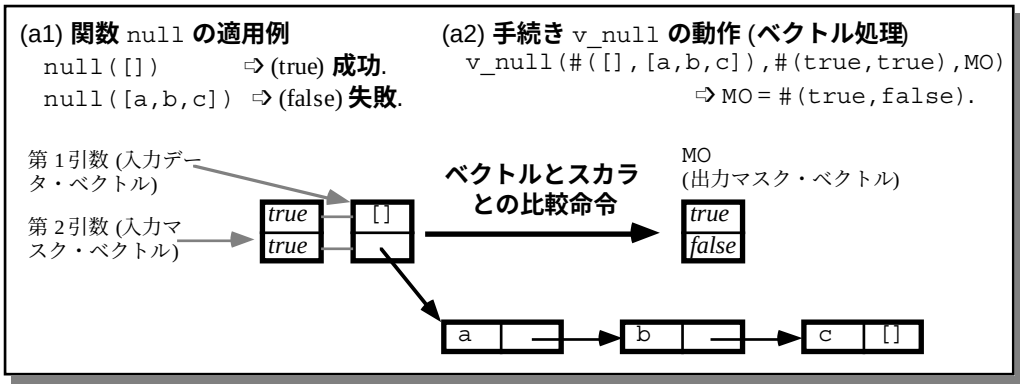
これらの基本演算とそのベクトル処理法について順に説明する．すべての条件制御方式についてのべると煩雑になるので，マスク演算方式についてだけのべる．ほかの条件制御方式も同様に実現可能である．なお，制御構造の変換によらずリストをベクトルにデータ構造変換することによってベクトル処理するばあいのリスト処理基本演算に関しては第 7 章でかんたんにのべる．

3.4.1 データ型判定

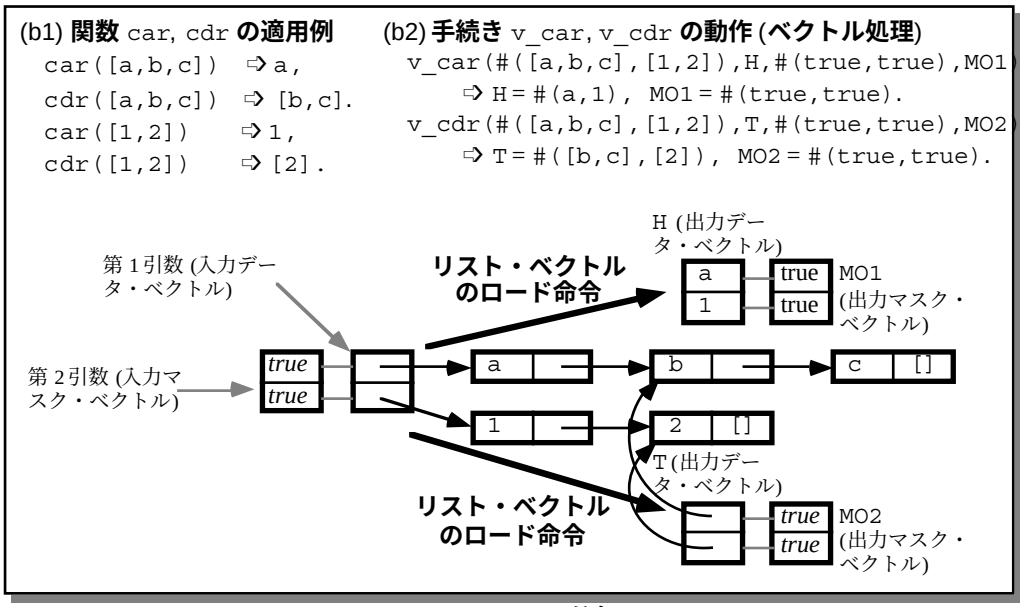
Lisp, Prolog などの記号処理用言語においては，各データは型 (タグ) をともなっている．そして，各データの処理にさきだって，型を判定する必要がある．たとえば，リスト処理においては，通常，空リスト判定 (空リストかどうかの判定) をおこないながらリストをたどる必要がある. 空リスト判定をおこなう Lisp 関数 null を例にとり, **図** 3.6 (a) を使用して，データ型判定の機能とそのベクトル処理の方法を説明する．

関数 null は 1 個の引数をもち，引数が空リストなら *true*，そうでなければ *false* を結果とする．関数 null の適用の例を図 3.6 (a1) にしめす．

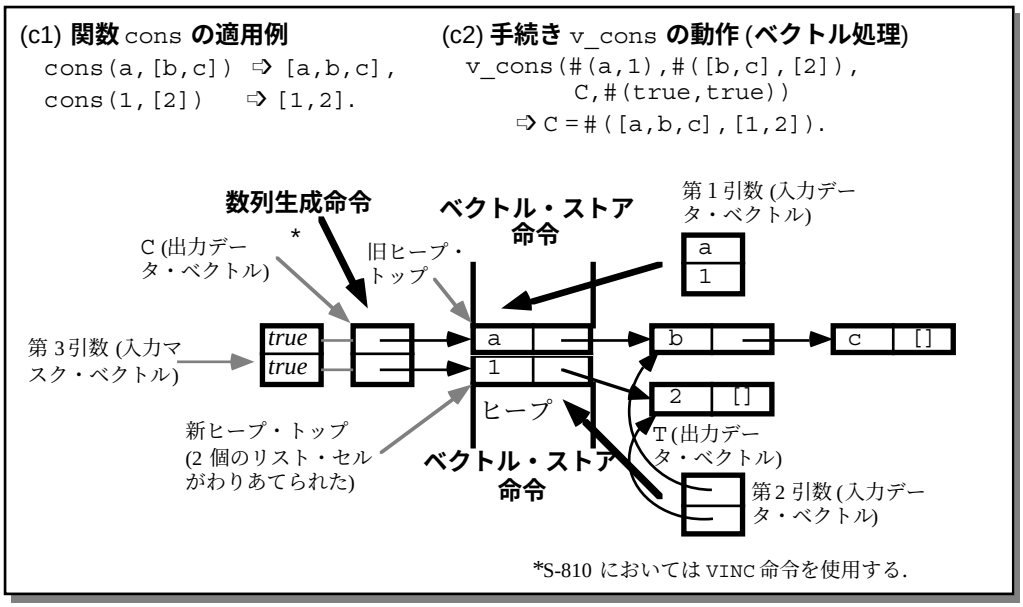
複数のデータに関する空リスト判定をベクトル処理で実現するには，つぎのような機能をもつ手続き v_null を用意しておき，これを使用すればよい．v_null(X,MI,MO) において，引数 X，MI は入力，引数 MO は出力であって，これらは要素数がひとしいベクトルである．以下，データ・ベクトル V の第 i 要素を V[i] であらわす．MI，MO はマスク・ベクトルである．MI[i] が *true* ならば MO[i] に null(X[i]) ($1 \le i \le N$，N は要素数) の値を代入する．MI[i] が *false* ならば X[i] に関する比較はおこなわず，MO[i] も *false* とする．



(a) **空リストかどうかのテスト**



(b) **リストの分解**



(c) **リストの合成**

図 3.6　リスト処理基本演算とそのベクトル処理方法

リスト処理におけるデータ型判定として，ほかには，データがリストかそれ以外の型かを判定する関数 atom などがつかわれる．これらの演算も null と同様にベクトル処理できる．

データ型判定の実行後に，判定結果が *true* のときだけまたは *false* のときだけ実行可能なベクトル演算をデータ・ベクトル X の要素に対して実行するばあいには，データ型判定において出力されるマスク・ベクトルをその演算においてマスク・ベクトルとして入力し使用する．たとえば，3.4.2 節でのべるリストの分解において，データ・ベクトルの要素のなかにリストとそれ以外のものとがまざっているときは，まず各要素がリストであるかどうかの判定をおこなって，その結果が *true* になった要素だけ分解をおこなう必要がある．

なお，ベクトル X の要素がつねにすべて演算対象であれば v_null の引数 MI は不要だが，v_null が条件制御のもとで実行されるばあいには MI が不可欠である．

v_null は，ベクトル計算機のもつベクトルとスカラとの比較命令をつかうことによって，容易にベクトル命令で実現できる．例を図 3.6 (a2) にしめす．

3.4.2 リストの分解

Lisp においてリストの要素をアクセスするには，あたえられたリストを分解する関数，すなわちその頭部をもとめる関数 car と尾部をもとめる関数 cdr とを使用する．Prolog においてはリストはユニフィケーションによって分解されるため，分解の機能は手続きというかたちで陽にはあらわれない．しかし，ユニフィケーションの実装のために関数 car, cdr に相当する機能が必要である．図 3.6 (b)を使用して，これらの機能とそのベクトル処理の方法とを説明する．まず，関数 car, cdr の適用例を図 3.6 (b1) にしめす．

複数のリスト分解をベクトル処理で実現するには，つぎのような機能をもつ手続き v_car, v_cdr を用意しておき，これらを使用すればよい．v_car(X,Y,MI,MO) において，引数 X, MI は入力，引数 Y, MO は出力であって，これらは要素数がひとしいベクトルである．MI, MO はマスク・ベクトルである．MI[*i*] が *true* であってかつ X[*i*] が分解可能ならば Y[*i*] = car(X[*i*]) ($1 \le i \le N$，N は要素数) である．MI[*i*] が *false* または X[*i*] が分解不能ならば X[*i*] に関する分解はおこなわず，MO[*i*] も *false* とする．関数よびだし v_cdr(X,Y,MI,MO) においても同様である．

v_car, v_cdr においては，データ・ベクトルから各要素の頭部または尾部をもとめて，それらを要素とするデータ・ベクトルをつくる．例を図 3.6 (b2) にしめす．より具体的にいえば，つぎのようになる. S-810 などのベクトル計算機は, ベクトル・インデクス (リスト・ベクトルとよばれている) つきのロード・ストア命令をもっている．インデクスつきロード命令の動作を C 言語であらわすと，つぎのようになる．

```
for (i = 1; i <= n; i++) {
   vector1[i] = *(vector2[i] + vectorBase);
}
```

ベクトル化された Fortran プログラムにおいては，通常は vectorBase に相当する命令フィールドに配列の原点 (先頭アドレス)，vector2[i] に相当する命令フィールドにその添字を指定して使用する．しかし，vector2[i] に相当する命令フィールドにリスト・セルの先頭アドレス，vectorBase に相当する命令フィールドにリスト・セルの先頭から頭部または尾部への変位を指定すれば，各リスト・セルの頭部または尾部を要素とするベクトルをもとめることができる．

3.4.3 リストの合成

Lisp においてリストを合成するには，関数 cons をつかう．Prolog においてはリストはユニフィケーションによって合成されるため，手続きというかたちで陽にはあらわれない．しかし，ユニフィケーションの実装のために関数 cons に相当する機能が必要である．図 3.6 (c) を使用して，関数 cons の機能とそのベクトル処理方法を説明する．

関数よびだし cons(X,Y) は，リスト Y の先頭に要素 X をつけくわえたリストを結果とする．関数 cons の適用の例を図 3.6 (c1) にしめす．

複数のリスト合成をベクトル処理で実現するには，つぎのような機能をもつ手続き v_cons をつかえばよい．v_cons(X,Y,C,MI) において引数 X, Y, MI は入力，引数 C は出力であって，これらは要素数がひとしいベクトルである．MI はマスク・ベクトルである．MI[*i*] が *true* ならば C[*i*] = cons(X[*i*],Y[*i*]) ($1 \le i \le N$, N は要素数) である．MI[*i*] が *false* ならば合成をおこなわない (C[*i*] は変化しない)．図 3.6 (c2) に例をしめす．

つぎのようにすれば，v_cons をベクトル命令で実行できる．まず，頭部とするべきデータを要素とするデータ・ベクトル H と，尾部とするべきデータを要素とするデータ・ベクトル T とを入力する．H と T とは要素数がひとしい．そして，H と T の対応する要素の値を頭部および尾部の値とするリスト・セルをつくり，それらをさすポインタを要素とするデータ・ベクトル C をつくる[注5]．そして C を結果とすればよい．リスト・セル長を *Lc* とし，リスト・セルをヒープに連続してわりあてるとすれば，ベクトル C の各要素のアドレス部は，リスト合成の実行前のヒープ・トップ・ポインタの値に 0，*Lc*，2 * *Lc*，… を加算してできた値をふくむ．このような値のベクトルは，ベクトル計算機の数列生成命令 (たとえば S-810, S-820 においては VINC 命令) を使用すれば高速に生成できる．むだのない記憶わりあてのためには[注6]等差でない数列生成またはマスクつき等差数

[注5] v_cons は 1 要素あたり 3 個のポインタを生成することに注意.

[注6] すなわち，「マスクが *false* である要素には記憶をわりあてないためには」.

列生成が高速実行できることがのぞましいが，現状では高速性を優先して等差数列を使用するのがよいであろう[注7]．

なお，MI[*i*] の要素のなかに *true* でないものがあるときには，*true* でない要素に対応するデータ・ベクトル要素に対してはリスト・セルをわりあてないようにするのが場所効率はよい．しかし，そうするとベクトル C のとなりあう要素のアドレスの差分が一定ではなくなるため，ベクトル計算機によってはベクトル化できなくなったり，ベクトル化できてもベクトル C の生成に要する時間が増加したりする．したがって，高速性を優先するばあいには，MI[*i*] が *true* でない要素に対してもリスト・セルをわりあてるほうがよい．

最後に，リストの合成を記号処理における記憶管理という側面からみる．リストの合成は動的記憶わりあてをともなう．したがって，この節ではベクトル処理によって複数のリスト要素の動的記憶わりあてを実行する方法をあたえたことになる．記憶管理におけるもうひとつの重要な操作は記憶の解放である．リスト処理においては記憶の解放のためにガーベジ・コレクションを使用するのが普通である．Appel ら [Appel 89] はベクトル処理によるガーベジ・コレクションの方法をあたえており，これとこの節の記憶わりあて法とをくみあわせることによってベクトル処理による記憶管理法として完結する．

[注7] S-820 をはじめとするいくつかのベクトル計算機においては，等差でない数列生成は等差数列生成よりはるかに低速であるためである．

3.5 エイト・クウィーンの Prologプログラムへの適用

この節では，3.2 節でのべたプログラム変換の戦略を，Prolog で記述されたエイト・クウィーン問題のプログラムに適用した例についてのべる．3.5.1 節ではエイト・クウィーンのプログラムの一部である手続き not_take1 を例としてくりかえし構造の交換についてのべる．3.5.2 節では，エイト・クウィーンのプログラムの一部である手続き select を例としてくりかえし構造の 1 重化についてのべる．Prolog のプログラムをベクトル化するためには，他のベクトル化技法と併用し，しかるべき手順をふむ必要があるが，その手順については第 6 章であらためて説明することにして，この節では上記の手続きのプログラム変換において使用されているくりかえし構造の交換と 1 重化だけに焦点をあてて説明する．なお，この節ではエイト・クウィーンのプログラムを例題とするが，Prolog で記述されたリストを分解する手続き append におけるくりかえし構造の交換と 1 重化に関しては図 6.8 にしめしているので，参照されたい．また次章では，リスト処理への適用例ではないが，より単純な配列の線形検索のプログラムを例題としてくりかえし構造の交換を手続き型言語上でおこなっている．

3.5.1 くりかえし構造の交換

エイト・クウィーン問題をとく Prolog プログラムは，たとえば中島 [Nakashima 83] でしめされている[注8]．つぎにしめすnot_take1 はその一部であり，指定された 1 個のクウィーンがチェス・ボードの対角方向にすでにおかれたクウィーンと衝突するかどうかをしらべる手続きである．not_take1 のすべての引数は入力である．

(1) 原始プログラム

```
not_take1([], Qa, Qs).
not_take1([Q|R], Qa, Qs) :-
    Q =\= Qa, Q =\= Qs,
    Qaa is Qa + 1, Qss is Qs - 1,
not_take1(R, Qaa, Qss).
```

第 1 引数はチェス・ボードをあらわすリストであり，第 2 〜 3 引数はしらべるべき位置 (行番号) をあらわす整数である．

not_take1 はその末尾が再帰よびだしになっていて, 各くりかえしにおいて, チェス・ボードをあらわすリストの要素を 1 個ずつしらべていく．エイト・クウィーンのばあいには，このリストの平均長は 4 以下とみじかい．したがって，この再帰よびだしをそのままループに変換したとしてもベクトル計算機の性能をいかすことはできない [Kanada

[注8] 中島のプログラムの全体は第 6 章でしめす.

88b]．しかし，not_take1 は，その外部で発生する深いバックトラックによって，その全体がくりかえし実行される．これらの実行のあいだにはデータ依存関係がないので，このくりかえしを最内側のくりかえしとするようにプログラム変換すれば，ベクトル処理が可能になる．

変換後の手続き v_not_take1 をしめす．

(2) ベクトル化後のプログラム

```
v_not_take1(_, _, _, MI, MI) :-
    v_finished(MI), !.
v_not_take1(B, Qa, Qs, MI, MO) :-
    v_null(B, MI, MO1),
    v_car(B, Q, MI, M1), v_cdr(B, R, M1, M2),
    'v_=\='(Q, Qa, M2, M3),
    'v_=\='(Q, Qs, M3, M4),
    'vs_+'(Qa, 1, Qaa, M4),
    'vs_-'(Qs, 1, Qss, M4),
    v_not_take1(R, Qaa, Qss, M4, MO2),
    v_end_or(MO1, MO2, MO).
```

v_not_take1 の第 1 〜 3 引数は not_take1 の第 1 〜 3 引数に対応している．not_take1 はバックトラックによってくりかえしよびだされるが，各回における not_take1 の第 1 引数の値を要素とするデータ・ベクトルを v_not_take1 の第 1 引数とする．第 2 〜 3 引数も同様である．エイト・クウィーンのプログラムにおける not_take1 以前に実行される部分は，このインタフェースをみたすようにプログラム変換しなければならない．

変換後のプログラムにおける条件制御の方式として 3 種類があるが，いずれのばあいもプログラムの基本構造はかわらないので，ここではマスク演算方式によるプログラムだけをしめす．v_not_take1 の第 4 〜 5 引数は入力および出力のマスク・ベクトルである．引数であるすべてのベクトルの要素数はひとしい．

(1) から (2) への変換の手順については第 6 章でのべるので，ここでは省略する．(1), (2) の各部分の対応を**表** 3.1 にしめす．手続き v_null，v_car，v_cdr の機能は 3.3 節でのべたとおりである．他の手続きの機能については表 3.1 を参照されたい．

すでにのべたように not_take1 の末尾再帰よびだしはリストの要素に関するくりかえしである．v_not_take1 の再帰よびだしは末尾再帰ではないが，データ・ベクトル B の要素であるリストの要素に関するくりかえしであり，not_take1 の末尾再帰よびだし

と対応している．しかし，v_not_take1 においては，それがよびだす手続き v_null，v_car，v_cdr などの内部にループがあり，これらは not_take1 におけるバックトラックで形成されるくりかえしに対応している．したがって，not_take1 から v_not_take1 への変換においてくりかえし構造を交換しているということができる．

(2) のプログラムにつぎのような入力をあたえたときの実行過程を**図** 3.7 にしめす．ただし，ここで #(e_1, e_2) は e_1，e_2 を要素とするベクトルをあらわす．

第 1 引数:　#([2,4,1], [4,1,3]).
第 2 引数:　#(4,3).
第 3 引数:　#(2,1).
第 4 引数:　#(true,true).

表 3.1　**手続き not_take1 における原始プログラムとベクトル化プログラムとの対応**

原始プログラム	ベクトル化後のプログラム	意味
-*	v_finished(MI)	手続きの実行を終了するかどうか (再帰を停止させるかどうか) を判定する．
[]	v_null(B,MI,MO1)	ベクトル B の要素が空リストかどうかを判定する．
[Q|R]	v_car(B,Q,MI,M1), v_cdr(B,Q,MI,M2)	ベクトルの要素であるリストを頭部と尾部とに分解する．
Q =\= Qa, Q =\= Qs	'v_=\='(Q,Qa,M2,M3), 'v_=\='(Q,Qs,M3,M4)	整数 Q と Qa, Q と Qs とがひとしくないかどうかをしらべる．
Qaa is Qa + 1	'vs_+'(Qs,1,Qaa,M4)	ベクトルの各要素とスカラ・データとの整数加算をする．
Qss is Qs - 1	'vs_-'(Qs,1,Qss,M4)	ベクトルの各要素とスカラ・データとの整数減算をする．
not_take1(R,Qaa,Qss)	v_not_take1(R,Qaa, Qss,M4,MO2)	再帰よびだしをする．
-*	v_end_or(MO1,MO2,MO)	原始プログラム第 1 節に対応する部分と第 2 節に対応する部分から出力されるマスクを合成する．

* 原始プログラムに対応する部分がない．

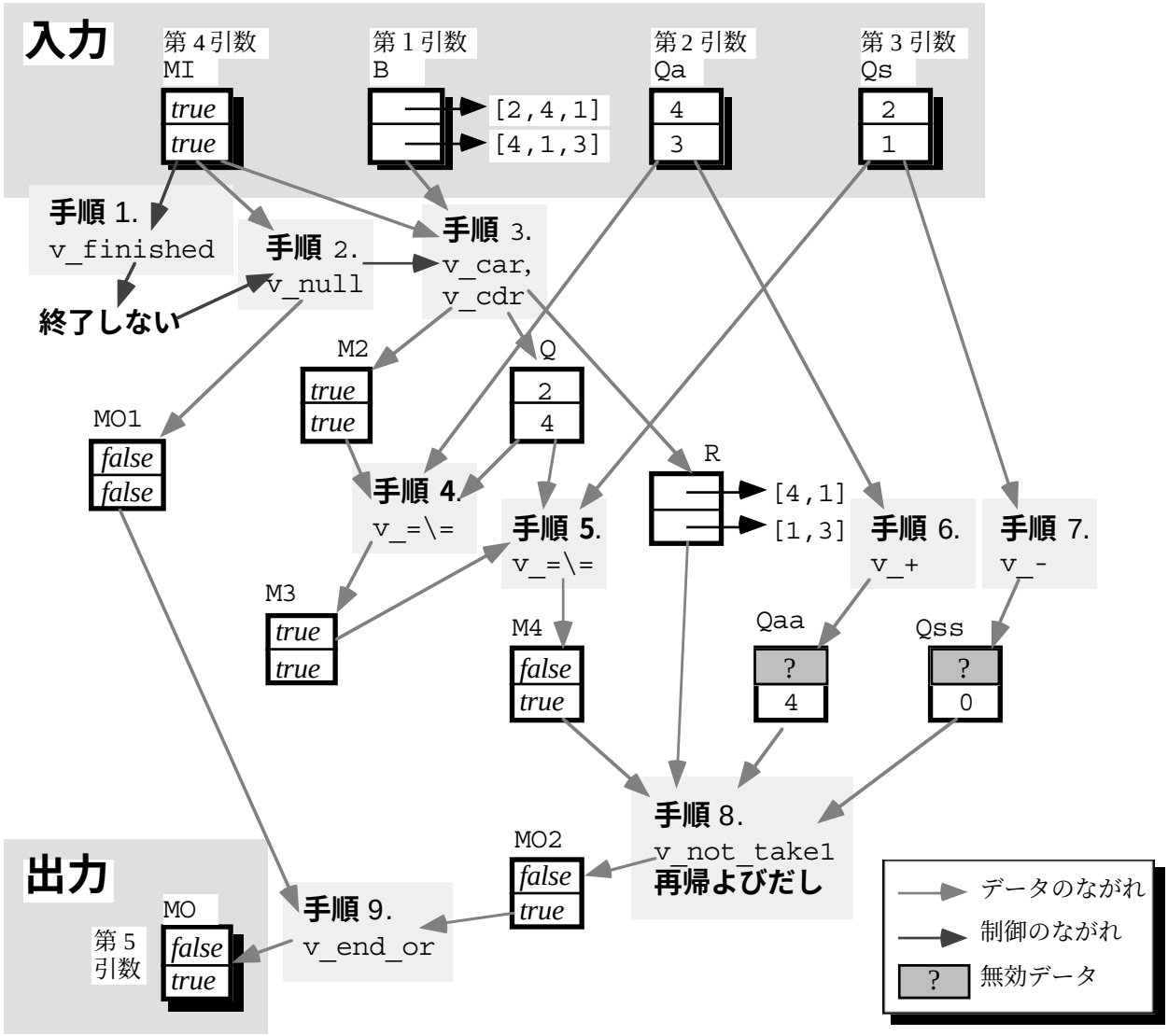


図 3.7　手続き `v_not_take1` の実行例

上記の入力をあたえて `v_not_take1` をよびだすことは，つぎのような 2 つの手続きよびだしを実行することとほぼ等価である．

```
?- not_take1([2,4,1], 4, 2).        …… (3.5.1)
?- not_take1([4,1,3], 3, 1).        …… (3.5.2)
```

(3.5.1) を実行すると失敗し (3.5.2) を実行すると成功する．それに対応して，図 3.7 の実行の結果は `#(false,true)` となる．実行過程については金田 [Kanada 87] でよりくわしくのべている．

3.5.2 くりかえし構造の 1 重化

エイト・クウィーン問題のプログラムにおける手続き `select` は，リストを入力して，そのリストから

1 要素をえらびだした結果と，そのリストからその要素をのぞいたリストとを出力する手続きである．入力されるリストのながさを n とすれば，解の数も n 個だけある．手続き select またはそれと等価な手続き (しばしば delete という名称をあたえられる) は Prolog プログラムにおいてしばしばつかわれるが，エイト・クウィーン問題のプログラムにおいても，クウィーンのリストから 1 個のクウィーンをえらぶのにつかわれる．

select のプログラムをしめす．

(1) 原始プログラム

```
select([A|L], A, L).
select([A|L], X, [A|L1]) :- select(L, X, L1).
```

第 1 引数が入力のリスト，第 2 引数がえらばれた要素，第 3 引数がのこりの要素のリストである．

select にはその末尾に再帰よびだしがあって，その各くりかえしにおいて第 1 節から 1 個ずつ解を出力し，select のよびだしもとに復帰する．ただし，select の実行後にバックトラックが生じなければ第 2 節は起動されず，したがって再帰よびだしも起動されない．

すべての解が出力されるまでバックトラックがくりかえされることを前提とすれば，つぎのようなプログラム変換が可能になる．すなわち，1 個の解がもとめられるごとに解を出力するかわりに，ベクトルを用意してそれに解を蓄積していく．すべての解がもとめられたところで，そのベクトルを出力する．この変換後のプログラムの機能は，Prolog の手続き bagof の機能ににている．このようにことなる解を要素とするベクトルをつかうインタフェースは，3.5.1 節の手続き v_not_take1 のそれと一致している．

変換後の手続き v_select のプログラムをしめす．

(2) ベクトル化後のプログラム

```
v_select(AL, X, Y, MI, BI, BO) :-
    v_select_1(AL, X1L, Y1L, MI, ML),
    v_merge([X1L, Y1L], [X, Y], [BI], [BO], ML).

v_select_1(_, [], [], MI, []) :-
    v_finished(MI), !.
v_select_1(AL, [A'|X1L], [L'|Y1L], MI, [M1'|ML]) :-
    v_car(AL, A', MI, M0'),
    v_cdr(AL, L', M0', M1'),
    v_car(AL, A, MI, M0),
```

```
v_cdr(AL, L, M0, M1),
v_select_1(L, X1L, L1L, M1, ML1),
mapcar(v_cons(A), L1L, Y1L, ML1, ML).
```

(1) から (2) への変換の手順は省略するが，その対応関係を**表** 3.2 にしめす．

v_select の第 1 〜 3 引数は select の第 1 〜 3 引数に対応している．v_select の入力時にも，出力時と同様のベクトル・インタフェースをとる．すなわち，v_select の実行開始以前に複数の解をもつ手続きが実行されていれば，それらを要素とするベクトルが第 1 引数として入力される．入力する解が 1 個のばあいも，唯一の要素をもつベクトルが第 1 引数として入力される．

表 3.2　**手続き** `select` **における，原始プログラムとベクトル化後のプログラムとの対応**

原始プログラム	ベクトル化後のプログラム	意味
-*	v_select_1(_,[],[],MI,[])	空のマルチ・ベクトルを生成する．
-*	v_finished(MI)	手続きの実行を終了するかどうか (再帰を停止させるかどうか) を判定する．
-*	v_select1(AL, [A'|X1L],[L'|Y1L], MI,[M1'|ML])	各マルチ・ベクトルに要素を追加する．
[A'|L'] *1	v_car(AL,A',MI,M0'), v_cdr(AL,L',M0',M1')	ベクトルの要素であるリストを頭部と尾部とに分解する．
[A|L] *2	v_car(AL,A,MI,M0), v_cdr(AL,L,M0,M1)	ベクトルの要素であるリストを頭部と尾部とに分解する．
select(L,X,L,1)	v_select_1(AL,X1L,L1L,M1,ML1)	再帰よびだしをする．
[A|L1]	mapcar(v_cons(A), L1L,Y1L,ML1,ML)	マルチ・ベクトル L1L の各部分ベクトルの要素に関してリストを合成する．
-*	v_merge([X1L,Y1L], [X,Y],[BI],[BO],ML)	マルチ・ベクトル X1L, Y1L をそれぞれ 1 個のベクトル X, Y に併合し，ベクトル BI の要素と対応づけて BO を生成する．

* 原始プログラムに対応する部分がない．

*1 原始プログラム第 1 節に対応している．

*2 原始プログラム第 2 節に対応している．

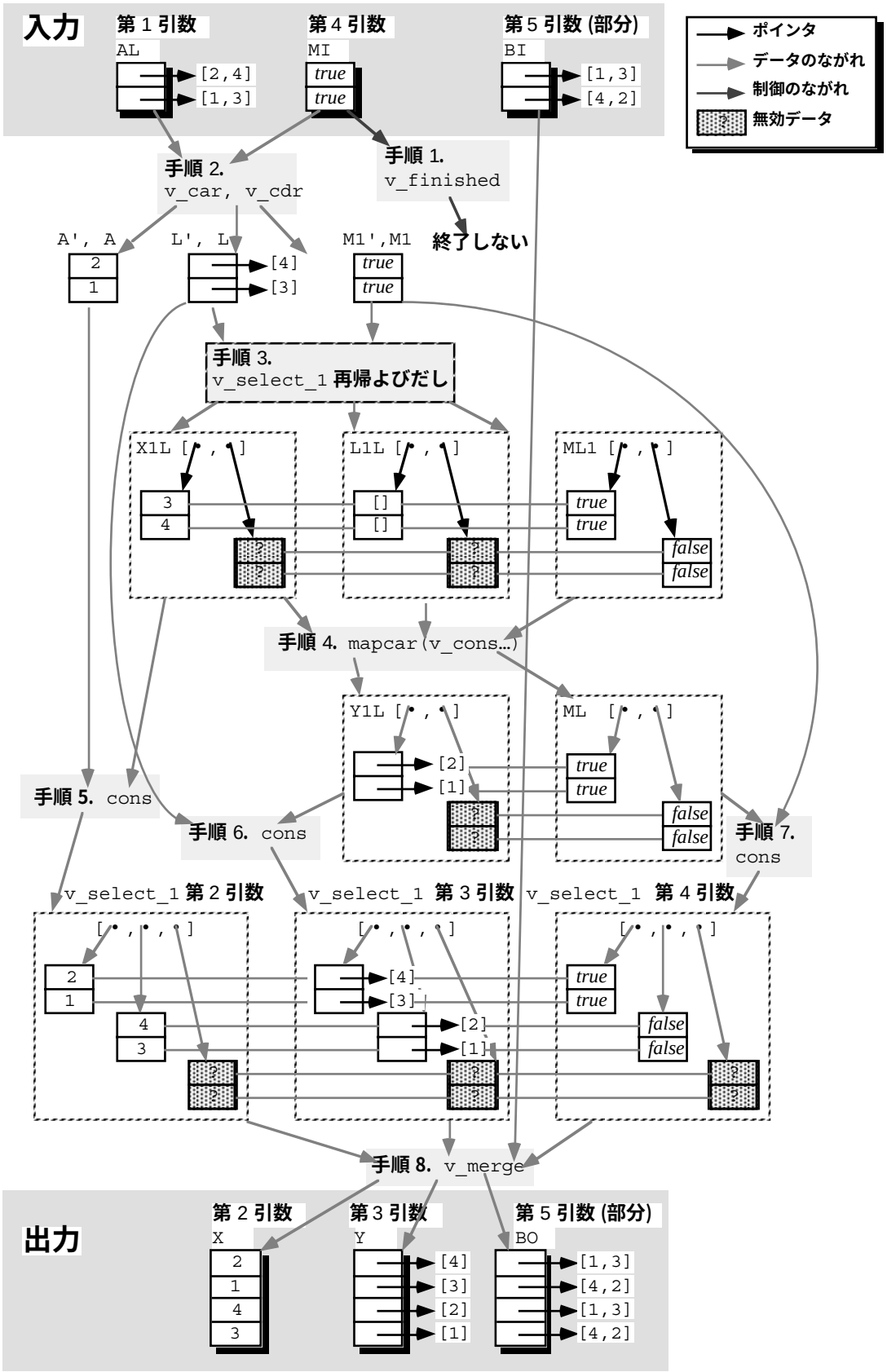


図 3.8　手続き v_select, v_selec1_1 の実行例

v_select の第 4 引数は入力マスク・ベクトルである．すなわち，第 1 引数の各要素の有効性が第 4 引数によってしめされる．しかし，3.4 節の各手続きや手続き v_not_take1 とはちがって，第 1，第 4 引数と第 2 〜 3 引数の要素数はひとしくなく，要素は対応しない．出力である第 2 〜 3 引数の要素はすべて有効である．したがって，マスク・ベクトルは出力しない．

v_select の各出力ベクトルの要素は，各入力ベクトルの要素とは対応がくずれるため，対応をとるべきベクトルを第 5 引数として入力し，その要素を複写・対応づけしてえられたベクトル BO を第 6 引数として出力する．したがって，BO の要素数は第 2 〜 3 引数の要素数とひとしい．エイト・クウィーンのプログラムにおいては要素対応をとるべきベクトルが 1 個だけなので対応づけのための引数は 2 個だけだが，プログラムによっては 4 個以上の引数が必要になる．手続き v_merge は，表 3.2 において説明しているように，複数のベクトルを併合してながいベクトルを生成する手続きである．手続き v_merge はこれらの対応づけのための引数をリストとして入出力するため，対応のための引数が任意の個数のばあいに使用することができる．なお，手続き v_merge の機能に関しては第 6 章でくわしく説明する．

v_select の実行においては，まず v_select_1 をよびだす．v_select_1 の再帰よびだしごとに，その第 2，3，5 仮引数において，もとめた解ベクトルをリストの要素として蓄積する．v_select_1 からの復帰後に，v_merge においてこれらのリストの要素をそれぞれ 1 つのベクトル長のながいベクトルに格納しなおし，出力する．
v_select_1 の定義にあらわれる mapcar(v_cons(A),L1L,Y1L,ML1,ML) は，A を第 1 引数とし，リスト L1L，Y1L，ML1，ML のそれぞれの各要素であるベクトルを第 2 〜 5 引数として手続き v_cons をくりかえし実行する．

すでにのべたように select は再帰よびだし 1 回ごとに解を出力するので，select のよびだし以降に実行されるプログラム部分はくりかえし実行される．したがって，not_take1 と同様の方法でベクトル化すれば，ベクトル化された select のよびだし以降に実行される部分はやはりくりかえし実行されることになる．それに対して v_select においては，もとめた解を蓄積することによってこのくりかえしをなくしている．したがって，v_select のよびだし以降に実行される部分のくりかえしを 1 重化しているということができる．

(2) のプログラムにつぎのような入力をあたえたときの実行過程を**図** 3.8 にしめす．

第 1 引数： #([2,4],[1,3]).
第 4 引数： #(true,true).
第 5 引数： #([1,3],[4,2]).

上記の入力をあたえて v_select をよびだせば，つぎのような 2 つの手続きよびだしを実行することとほぼ等価である．

```
?- select([2,4],X,Y).
```
…… (3.5.3)

```
?- select([1,3],X,Y).
```
…… (3.5.4)

(3.5.3) を実行するとつぎのような解がえられる．

```
X = 2, Y = [4].
```
…… (3.5.5)

```
X = 4, Y = [2].
```
…… (3.5.6)

また，(3.5.4) を実行するとつぎのような解がえられる．

```
X = 1, Y = [3].
```
…… (3.5.7)

```
X = 3, Y = [1].
```
…… (3.5.8)

図 3.7 においては，(3.5.5) 〜 (3.5.8) をベクトルに格納した形の解がえられているので，ただしくベクトル化されていることがわかる．

3.6 評価

3.4 節でしめした方法にしたがってエイト・クウィーン問題の Prolog プログラムを手動でベクトル化し，さらに手動で Fortran と Pascal とによるプログラムに変換したあとコンパイルした．Fortran 部分はベクトル・コンパイラでコンパイルした．Fortran コンパイラのベクトル化オプションをかえることによってベクトル処理用のコードとスカラ処理用のコードとを生成して，両者をベクトル計算機 S-810 で実行した．

実行時間を測定した結果を**表** 3.3 にしめす．3.4 節でしめしたマスク演算方式だけではなく，3 つの条件制御方式のそれぞれをおもに使用するプログラムをコーディングして測定した[注9]．プログラム全体としては，ベクトル処理速度はスカラ処理速度の 8 〜 9 倍となっている．この結果のよりくわしい分析は第 6 章でしめす．

表 3.3　ベクトル計算機 S-810 むきにハンド・コンパイルした エイト・クウィーン問題のプログラムの実行性能

プログラムの版 (主要な条件制御方式)	S-810 **ベクトル処理時間** (ms)	S-810 **スカラ処理時間** (ms)	**加速率**
マスク演算版	18	167	9.3
インデクス版	18	140	7.8
圧縮版	19	160	8.4

エイト・クウィーン問題のプログラムの各部分ごとの実行時間を測定することによって，リスト処理基本演算の加速率すなわちスカラ処理時間とベクトル処理時間の比をもとめ，**表** 3.4 にしめした．`v_car`, `v_cdr` および `v_null` の加速率と，`v_cons` の加速率とをしめしている．前 3 者のそれぞれの時間を個別にしめしていないのは，測定したプログラムにおいては，高速化のためにこれらが 1 個のループのなかに共存していて，個別に測定することが困難だったためである．

前 3 者の加速率は 10 倍をこえていて，満足すべき加速率だとかんがえられる．しかし，`v_cons` は 3 倍程度であり，十分な加速率とはいえない．加速率がひくいおもな原因は，`v_cons` が 1 要素あたり 3 つのポインタのストアを必要とするのに対して，S-810 の主記憶のストア・スループットが十分でないことだとかんがえられる．これは，現在のスーパコンピュータが数値計算むきに最適化されているために，とくに S-810 においてはロード・スループットにくらべてストア・スループットがひくいことの結果だとかんがえられる．

[注9] ただし，実際には各プログラムのなかで複数の方式をまぜて使用している．これは，1 つの方式だけでは実現不可能な部分があるなどの理由による．

表 3.4　エイト・クウィーン問題のプログラムにおける基本演算種別ごとの加速率

プログラムの版 (主要な条件制御方式)	`v_car`, `v_cdr`, `v_null` における加速率	`v_cons` における加速率
マスク演算版	12.1	3.1
インデクス版	13.6	3.3
圧縮版	13.7	3.1

3.7 まとめ

ベクトル計算機を使用し，プログラム変換にもとづいてリスト処理を高速に実行するための「くりかえし構造の交換」および「くりかえし構造の 1 重化」という戦略を提案した．この方式をエイト・クウィーンの Prolog プログラムに適用して，ベクトル計算機 S-810 において逐次処理の約 9 倍の実行速度をえた．かぎられた例題に適用しただけではあるが，この結果はこれらのプログラム変換方式の有効性をしめすものとかんがえられる．したがって，これらの方法に関するより具体的な研究をおこなうとともに，論理型言語プログラムの自動ベクトル化などへの応用を検討する必要があるとかんがえられる．そこで，次章ではさらにこの章でしめした残存要素検出法とならぶくりかえし構造の交換を具体化するもうひとつの方法をしめし，実測結果にもとづいてこれらを比較する．また，これらの方法の論理型言語プログラムのベクトル化への応用に関しては 3.5 節でその概要をしめしたが，それについては第 6 章でくわしくのべる．