(2)

自動ベクトル化Pascalコンパイラに関する研究

國枝　義敏

1990年11月

自動ベクトル化Pascalコンパイラに関する研究

國枝 義敏

内容梗概

電子計算機に対する社会の希求は、今日においても益々さらなる高速処理を指向する状況にある。この際限のない高速処理の要求に応じるため、汎用計算機に比べ飛躍的に高速な処理能力を有するスーパーコンピュータと称せられる電子計算機が研究開発されている。そのスーパーコンピュータの中で、ベクトル計算機は、現時点において実用化されている数少ない例といえる。ベクトル計算機は、アメリカ合衆国、日本、ヨーロッパ各国で様々な大学、民間企業等の計算機センター、研究施設に導入され、数値シミュレーションをはじめ種々の分野で、その高速性能を発揮し実績を挙げつつある。

ベクトル計算機は、パイプライン方式の演算器を有し、複数個のデータに同一の演算や処理を施し高速処理を達成している。従って、ベクトル計算機の有するハードウェア性能を引き出すためには、このパイプライン演算器を有効に動作させる必要がある。すなわち、あるプログラムをベクトル計算機で高速に実行しようとした際、そのプログラムが行っている種々の演算の中から、パイプライン演算器でベクトル処理できるような部分、つまりベクトル実行可能部分を抽出し、実際に演算器に処理すべき一連のデータを与える命令に書き換える一種のプログラム変換操作、つまりベクトル化処理が必要となる。このベクトル化は、具体的には1)システム標準手続/関数呼び出しによる方式、2)プログラミング言語の構文規則を拡張する方式、3)自動ベクトル化コンパイラによる方式に大別できる。このうち、前2者ではプログラム作成者が、ベクトル実行可能部分を抽出し適切な記述を行う必要があるのに対し、第3の自動ベクトル化では、プログラム作成者の負担ははるかに小さいものとなる。また、従来蓄積されてきたプログラムについても記述を変更することなくベクトル計算機によって効率良く処理することができる。これらの点を考慮し、近年のベクトル計算機では自動ベクトル化方式が採用されることが一般的である。逆

に、ベクトル計算機が今日このように普及しつつある理由として、高速性能もさることながら、優れた自動ベクトル化機能を持つことにより、親しみやすいユーザインタフェースを用意した点を見逃すことはできない。

自動ベクトル化コンパイラは、ベクトル計算機をユーザに使いやすいものとした。しかし、現在、ベクトル計算機の利用者には、パイプライン演算器を直接操作する機械語レベルでのプログラミングは許されていない。高級言語についても、ベクトル計算機メーカが提供する自動ベクトル化コンパイラがサポートするプログラミング言語でしか利用できない現状である。それゆえに、利用者はその制限されたユーザインタフェースを通してのみベクトル計算機を使用でき、利用者のプログラムがどの程度ベクトル化されるか、ひいてはどの程度高速実行され得るかは、すべてコンパイラの持つ自動ベクトル化性能の如何によることになる。ベクトル実行される部分がプログラム全体に占める割合、すなわちベクトル化率が高いプログラムはパイプライン演算器を有効に利用することができ、一般的に高速処理が期待できる。従って、このベクトル化率は自動ベクトル化コンパイラの自動ベクトル化機能の重要な一指標とされている。その他にも、自動ベクトル化コンパイラがベクトル計算機のハードウェア性能を十分引き出すためには、1度にまとめてパイプライン演算器で処理するデータの数、すなわちベクトル長、そして並列実行可能なパイプライン演算器群のスケジューリング、パイプライン演算器起動のためのオーバヘッド、ならびに主記憶アクセス方式等までも意識する必要がある。

本論文では、A)ベクトル計算機を現状よりさらにユーザに使いやすいものとし、かつB)高機能な自動ベクトル化を可能とすることを目的に開発された自動ベクトル化 Pascal コンパイラ V-Pascal Version 1 について、主にその自動ベクトル化機能に焦点をあて論じる。自動ベクトル化 Pascal コンパイラ V-Pascal Version 1 の大きな特徴は1) 言語 Pascal を対象とすること、2) 多重ループ全体をベクトル化対象とし強力にベクトル化を行えること、3) 種々のベクトル計算機をターゲットマシンとできることを考慮し設計されていることである。このうち第2点を詳述すると、2-1) ベクトル化可能性の判定に必要となる各種の依存関係解析機能が、多重ループ全体を対象とし、かつ厳密に解析できるものであること、2-2) 同解析機能により判定した依存関係から細かな演算レベルでのベクトル化可能部分を抽出する強力な部分ベクトル化機能を有すること、ならびに2-3) ベクトル化可能な多重ループ内の処理を、等価な1重ループ内での処理として長大なベクトル長でベクトル実行させる目的コード

に翻訳する機能を有することである。従来のメーカ提供の自動ベクトル化コンパイラは、多くの場合多重ループ全体でなく最内側ループのベクトル化にしか対応できていなかった。そのため、見過ごされてきた最内側ループ以外のループ中に存在する部分に関して、前記の機能を合わせ持つことにより初めてベクトル化可能となった。すなわち、そのような多重ループに関しては従来に比べ、ベクトル化率が大幅に向上する。しかも、一重化機能により長大なベクトル長でベクトル実行されるので、最内側ループだけのベクトル長でベクトル実行される場合より、さらに有効にパイプライン演算器を使用でき、高速処理される場合があることを初めて実証した。具体的には、一重化してベクトル実行可能な数種の2重ループに関して性能評価を行った。その結果、V-Pascal コンパイラが生成した目的コードは、従来のメーカ提供の自動ベクトル化 FORTRAN コンパイラが生成した目的コードと比べ、3倍から4倍高速に実行されることが実測された。同時に、この一重化は間接参照型の主記憶アクセスを頻繁に行うため、この種のアクセスを高速処理できるアーキテクチャが重要であることも実証された。これは、今後のベクトル計算機を設計する際に、重要な一指針となる知見であると考えられる。

自動ベクトル化 Pascal コンパイラに関する研究

國枝 義敏

目　次

内容梗概 ..... (1)
目　次 ..... (4)
図表目次 ..... (9)

第1章　緒 論 1
1.1 ベクトル計算機の発達と普及 1
1.2 V-Pascal コンパイラの開発 6

第2章　ベクトル計算機のアーキテクチャ 8
2.1 緒 言 8
2.2 HITAC S-820 シリーズ 9
2.3 FACOM VP シリーズ 13
2.4 NEC SX シリーズ 17
2.5 結 語 20

第3章　自動ベクトル化 21
3.1 緒 言 21
3.2 ベクトル化と依存関係 23
3.2.1 ベクトル化による実行順序の変更 24
3.2.2 データ参照関係とデータ依存 26

3.2.3 制御依存 31
3.2.4 依存グラフ 31
3.3 自動ベクトル化技術の現状 32
3.3.1 ベクトルデータ 33
3.3.2 自動ベクトル化の対象となる演算 35
3.3.3 自動ベクトル化の対象ループ 36
3.3.4 飛び出しを含む DO ループのベクトル化 37
3.3.5 多重 DO ループの自動ベクトル化 38
3.3.6 IF 文のベクトル化手法 39
3.4 結 語 44

第4章　V-Pascal コンパイラの構成 46
4.1 緒 言 46
4.2 フェーズ構成とモジュール構成 47
4.3 主要表およびそのデータ構造 53
4.3.1 識別子表、型表、定数表 53
4.3.2 中間コード 62
4.3.3 コントロールフロー・グラフ 81
4.3.4 プライマリ・セット表 87
4.3.5 D 行列 89
4.4 構文/意味解析 96
4.5 Alias 解析 96
4.6 コントロールフロー解析 100
4.6.1 支配関係解析 101
4.6.2 ループ検出 103
4.7 大域的データフロー解析 108
4.8 最適化 115
4.9 結 語 119

第5章　依存関係解析 120
5.1 緒 言 120

5.2 データ依存関係解析 121
5.2.1 単純変数のデータ依存関係解析 121
5.2.2 構造のある型の変数のデータ依存関係解析 124
5.3 配列要素のデータ依存関係解析 125
5.3.1 添字式に着目したDiophantus不定方程式と繰り返し空間 126
5.3.2 C行列と基本ベクトル 128
5.3.3 数式処理法 130
5.3.4 Sort-Merge法 137
5.3.5 数式処理法とSort-Merge法を組み合せた複合アルゴリズム 140
5.3.6 整数解からの依存関係の判定 143
5.3.7 自己依存関係 147
5.4 制御依存関係解析 149
5.4.1 制御関係解析(I):条件分岐から生ずる制御依存解析 151
5.4.2 制御関係解析(II):直属の条件分岐の解析 155
5.4.3 プライマリ・セット表の作成 160
5.4.4 制御関係解析(III):2出現間の制御の流れによる先行性解析 163
5.5 結　語 167

第6章　多重ループのベクトル化 172
6.1 緒　言 172
6.2 if-then-else構造への正規化 173
6.3 各種依存関係解析とD行列への登録 178
6.4 依存関係によるベクトル化可否判定 179
6.5 最適ベクトル化手法選択 184
6.5.1 配列化によるベクトル化 185
6.5.2 ループ選択によるベクトル化 185
6.5.3 ベクトル化不可能な中間コードの判定 186

6.5.4 ループ構造および制御構造の再構成 187
6.5.5 ループ間にまたがる参照関係をもつ中間項の配列化 189
6.5.6 中間項の受渡しを考慮した最適ベクトル化ループの決定 197
6.6 部分ベクトル化 200
6.7 結　語 209

第7章　目的コード生成 211
7.1 緒　言 211
7.2 実行環境の構造 214
7.2.1 データ領域 214
7.2.2 コード領域 216
7.2.3 ファイル管理 220
7.2.4 ユーザプログラムの目的コードの構造 220
7.3 ベクトル処理命令生成 229
7.3.1 ベクトル・ユニットのレジスタ割り付け 229
7.3.2 主記憶参照の逐次化 235
7.3.3 ベクトル・ユニット起動のためのセットアップ 237
7.4 スカラ処理命令生成 248
7.5 結　語 252

第8章　性能評価 253
8.1 緒　言 253
8.2 多重ループのベクトル化 257
8.3 if文を含むループのベクトル化 261
8.4 一重化の効果 263
8.5 結　語 265

第9章　結　論 267

謝　　辞 269

参考文献 271

論文一覧 276

付録 Pascal 構文図 279

図表目次

第1章

図1.1 ベクトル計算機の発達経過 3

表1.1 ベクトル計算機の世代分類 2

第2章

図2.1 HITAC S-820/80 の構成 10
図2.2 FACOM VP シリーズ E モデルの構成 14
図2.3 NEC SX-2 の構成 18

第3章

図3.1 パイプライン演算器の動作概念図 21
図3.2 パイプライン制御、パイプライン演算器、
マルチプロセッサの比較(概念図) 22
図3.3 DO ループのベクトル化による実行順序の変更 24
図3.4 ベクトル化のソースレベルでの表現 25
図3.5 単純変数の配列化 34
図3.6 飛び出しを含むループのベクトル化の例 37
図3.7 ベクトル化に最適なDO ループの選択の例1 38
図3.8 ベクトル化に最適なDO ループの選択の例2 38
図3.9 多重ループの一重化ベクトル実行の例1 39
図3.10 多重ループの一重化ベクトル実行の例2 39
図3.11 IF 文を含むループのベクトル化の例 40
図3.12 IF 文のベクトル化手法 41
図3.13 収集型のループのベクトル化の例 42
図3.14 拡散型のループのベクトル化の例 42
図3.15 ループ不変の IF 文のベクトル化の例 43
図3.16 ループ展開によるベクトル化の例 43

表3.1 例1で2出現が参照する要素の添字番号 26
表3.2 例1で参照が衝突している要素の実行順序比較 26
表3.3 例2で2出現が参照する要素の添字番号 27
表3.4 例2で参照が衝突している要素の実行順序比較 27
表3.5 定義/引用の順序によるデータ依存の分類 28
表3.6 ループの回転に着目したデータ依存の分類 28

第4章

図4.1 V-Pascal のフェーズ構成/概略モジュール構成 48
図4.2 データの流れを中心としたフェーズ2拡大図 49
図4.3 データの流れを中心としたフェーズ3拡大図 49
図4.4 データの流れを中心としたフェーズ4拡大図 50
図4.5 二分探索木を形成する識別子表 54
図4.6 レコード型とwith 文の例 54
図4.7 レコード型の内部表現 ..... 59~60
図4.8 多次元配列の内部表現 61
図4.9 V-Pascal の中間コードの種別 64
図4.10 手続/関数宣言ノードによる宣言の入れ子の木構造の表現 ... 69
図4.11 単純なソース・プログラムの中間コード表現 70
図4.12 ロングジャンプの例 72
図4.13 ループヒギン・ノードによる
for ループの入れ子の木構造の表現 73
図4.14 基本的な for ループの中間コード表現 74
図4.15 標準的なベクトル型ノードリンク用ノード間の関係 76
図4.16 変数記述子による表現 78
図4.17 ベクトルロード/ストア・ノードによる主記憶参照 80
図4.18 コントロールフロー・グラフとガード式
およびプライマリ・セットの例 88
図4.19 依存グラフとその隣接行列表現の例 89
図4.20 例による D 行列を構成する各セルのリンク構造 93
図4.21 D 行列操作後の D 行列 94

図4.22 言語 Pascal における別名の例 98
図4.23 簡単なコントロールフロー・グラフの例 102
図4.24 部分的に重なりのあるループを持つ
コントロールフロー・グラフの例 104
図4.25 到達する定義の例 109
図4.26 データフロー解析のための定義出現表 113
図4.27 定数の畳み込みの例 115
図4.28 複写の伝播の例 116
図4.29 共通部分式の結果の再利用の例 117

表4.1 識別子表のノードの共通フィールド 55
表4.2 識別子表のノードの可変フィールド 56
表4.3 型表のノードの共通フィールド 57
表4.4 型表のノードの可変フィールド 58
表4.5 中間コードのノードの共通フィールド 63
表4.6 中間コードの種類と意味 ..... 65~68
表4.7 変数記述子の持つフィールド 77
表4.8 変数記述限定子の持つ可変フィールド 77
表4.9 ベクトルロード/ストア・ノードおよび
ベクトル演算ノードに共通フィールド 79
表4.10 コントロールフロー・グラフの頂点の種別 82
表4.11 コントロールフロー・グラフの頂点の共通フィールド 84
表4.12 コントロールフロー・グラフの頂点の可変フィールド .. 85~86
表4.13 D 行列の行列構成セルの持つフィールド 91
表4.14 D 行列の非零要素セルの持つフィールド 91
表4.15 D 行列の非零要素セルに登録されるデータ依存の細分類 95
表4.16 図4.23を例とした支配関係および逆支配関係 101
表4.17 コントロールフロー・グラフの有向辺の類別 105
表4.18 定義を要素とするデータフロー集合 110
表4.19 変数を要素とするデータフロー集合 110

第5章

図5.1 単純変数の配列化による依存の消滅 123
図5.2 簡単な配列参照の2出現とそれから導出される
Diophantus 方程式の例 128
図5.3 図5.2 (a) の2出現のための C 行列 129
図5.4 図5.2 (a) を例とする正規化 132
図5.5 正規化とループ制御変数の最大値/最小値の一般的な関係 135
図5.6 解の存在範囲の絞り込み 136
図5.7 図5.2 の2出現を例とする Sort-Merge 法の処理 139
図5.8 例4 および例5 の配列 A の2出現のための C 行列の比較 147
図5.9 図4.23 の例における第1の仮想的な定義による
制御依存解析 152
図5.10 if-then-else 構造でないコントロール・フロー・グラフの
制御依存解析 153
図5.11 図5.9 の第2の仮想的な定義による直属の条件分岐解析 157
図5.12 図5.10 の第2の仮想的な定義による直属の条件分岐解析 158
図5.13 図5.12 のコントロールフロー・グラフの
プライマリ・セット表 (部分) 162
図5.14 複数の入口を持つループの
コントロール・フロー・グラフの先行性解析例 164
図5.15 Banerjee のアルゴリズムでは
厳密な依存解析が行えない例 169

表5.1 単純変数のデータ依存 122

第6章

図6.1 if 文を含むループのベクトル化の
ソースプログラムによる表現 174
図6.2 網目構造を含むループのコントロールフロー・グラフの例 .. 176
図6.3 網目構造の if-then-else 構造へ変換の例 177

図6.4 ベクトル化可能性判定のための依存関係による
半順序づけ 180
図6.5 ループ選択をともなう部分ベクトル化による
ループ構造の変形の概念図 188
図6.6 一般的なループ間にまたがる中間項の受渡しのループ構造 . 190
図6.7 ループ間にまたがる中間項の受渡しのある
ループ構造の一例 192
図6.8 1次元配列による中間項の配列化 193
図6.9 多次元配列による中間項の配列化 193
図6.10 ソフトウェア・ループ制御を利用した中間項の配列化 194
図6.11 ソフトウェア・ループ制御によるループ繰り返しの変化 ... 195
図6.12 部分ベクトル化処理適用例 ..... 203~205

表6.1 中間項の配列化の3方式 197

第7章

図7.1 実行時のデータ領域の構造 215
図7.2 実行時のモジュール構成 217
図7.3 ユーザプログラムの目的コードの構造 222
図7.4 ユーザ手続/関数呼び出しの制御の流れ 224
図7.5 システムコンスタント・エリアのコード 225
図7.6 標準手続/関数呼び出しの制御の流れ 227
図7.7 標準手続/関数呼び出しに使用される
サブルーチン RUNTIME# 228
図7.8 ベクトルロード/ストア命令による主記憶参照 233
図7.9 セットアップ用データ格納領域 239
図7.10 1回の EXVP 命令による
ベクトル・ユニット起動のスケルトン 241
図7.11 複数回の EXVP 命令による
ベクトル・ユニット起動のスケルトン 243
図7.12 複数回の EXVP 命令による
ベクトル・ユニット起動のタイミング・チャート 246

図7.13 簡単なforループの場合の生成される目的コード .... 249~250

表7.1 実行時ライブラリの親モジュールの機能概略 217
表7.2 標準手続/関数用サブルーチン群のモジュールの機能 .. 218~219
表7.3 テキスト型ファイルのファイルブロックの
構成フィールド 221
表7.4 ノンテキスト型ファイルのファイルブロックの
構成フィールド 221
表7.5 ベクトル・ユニットのレジスタの種別 232
表7.6 スカラ・ユニットの汎用レジスタの役割分担 248

第8章
図8.1 FORTRANの時間測定用ソース・プログラム 254
図8.2 Pascalの時間測定用ソース・プログラム 255
図8.3 Pascalの時間測定用ソース・プログラムの処理の流れ 256
図8.4 FFTバタフライ演算を行う評価用プログラム ..... 258~259
図8.5 行列の転置を行う評価用プログラム 260
図8.6 if文を含む行列の転置を行う評価用プログラム 262
図8.7 if文を含む連続アクセスの評価用プログラム 264
図8.8 if文を含む間接アクセスの評価用プログラム 264

表8.1 64点FFTバタフライ演算の実行時間 259
表8.2 行列の転置を行う評価用プログラムの実行時間 260
表8.3 図8.6の評価用プログラムの実行時間 262
表8.4 図8.7の評価用プログラムの実行時間 264
表8.5 図8.8の評価用プログラムの実行時間 264

第1章
緒論

1.1 ベクトル計算機の発達と普及

電子計算機はその誕生以来、より大規模な計算をより高速に処理できることを目指し発達してきた。汎用計算機の処理能力が限界に近づきつつあるといわれる今日においても、さらなる高速処理が要請される状況に変化はなく、むしろますますその度合いを強める傾向にある。この際限のない高速処理の要求に応じるため、汎用計算機に比べ飛躍的に高速な処理能力を有するスーパーコンピュータと称せられる電子計算機が研究開発されている。スーパーコンピュータのアーキテクチャについては、様々なアプローチがなされており、実現されている方式も種々ある。しかし、現時点において実用化され、かつ台数の点からみて普及しているといえる方式は多くなく、ベクトル計算機はその数少ない例といえる。ベクトル計算機は、アメリカ合衆国、日本、ヨーロッパはじめ世界各国で様々な大学、民間企業等の計算機センター、研究施設に導入され、数値シミュレーションをはじめ分野を問わずその高速性能を発揮し実績を挙げつつある。

ベクトル計算機のアーキテクチャについては、第2章で詳述することとし、まずベクトル計算機の発達の歴史に触れておく。

スーパーコンピュータについては、その処理速度、ならびにハードウェアの特色の観点から、世代分類がなされている[1][2][3]。細かな差異も考慮するといくつかの分類のパターンがあるが、それらの差異は視点の多様性を意味するにすぎない。表1.1に文献[1]の分類から通常ベクトル計算機とされる計算機を抽出してまとめる。図1.1はこれらのベクトル計算機の稼動年と理論最大性能(種々の並列アーキテクチャが理想的に同時に機能した際の処理性能)の関係をおおまかに示したものである。同図中には比較のため汎用計算機のデータも示している。これらの図表にはあげていないが、1988年末の新聞発表によれば富士通は単体のプロセッサとしての最大性能が5GFLOPSのVP-2000シリーズを、日本電気は同じく単体のプロセッサとしての最大性能が5.5GFLOPSのSX-3を開発し、1990年頃より出荷を開始する。このよう

表1.1 ベクトル計算機の世代分類

世代	例		
	計算機 (メーカ名、モデル名等)	理論最大性能 (MFLOPS)	備 考
第1世代	Control Data (CDC) 社 STAR-100	50	
	Texas Instruments (TI) 社 ASC	30	
	ILLIAC Ⅳ	50	
第2世代	Cray Research 社 CRAY-1	160	
	Control Data (CDC) 社 CYBER 205	400 800	64ビット 32ビット
第3世代	Cray Research 社 CRAY X-MP	420 936	2CPU 4CPU
	Cray Research 社 CRAY Y-MP	2666	8CPU
	Cray Research 社 CRAY-2	1952	4CPU
	ETA Systems 社 ETA-10	5000 10000	64ビット、8CPU 32ビット、8CPU
	日立製作所 S-810/20	857	
	日立製作所 S-820/80	2000	
	富士通 VP-200	533	
	富士通 VP-400	1143	
	富士通 VP-200E	857	
	富士通 VP-400E	1714	
	日本電気 SX-1	533	
	日本電気 SX-2	1300	
	IBM 社 IBM 3090VF	116 698	1CPU 6CPU

に、現在のベクトル計算機の最大性能は数 GFLOPS に達している。

ベクトル計算機は、**パイプライン方式**の演算器(第2章、第3章で詳述)を使用する**ベクトル命令**1命令で多数のデータに同一の演算、処理を施すことにより、これらの図表に挙げた高速処理を達成している。ただし、その高速性はパイプライン演算器で処理される(**ベクトル処理**あるいは**ベクトル実行**される)場合、すなわち**ベクトル実行可能**な場合についてのみである。ベクトル実行不可能な場合については、汎用計算機同様に通常の演算器を使用するスカラ命令1命令で1データずつ処理する(これをベクトル処理、ベクトル実行と対比させる意味で「**スカラ処理**」、「**スカラ実**

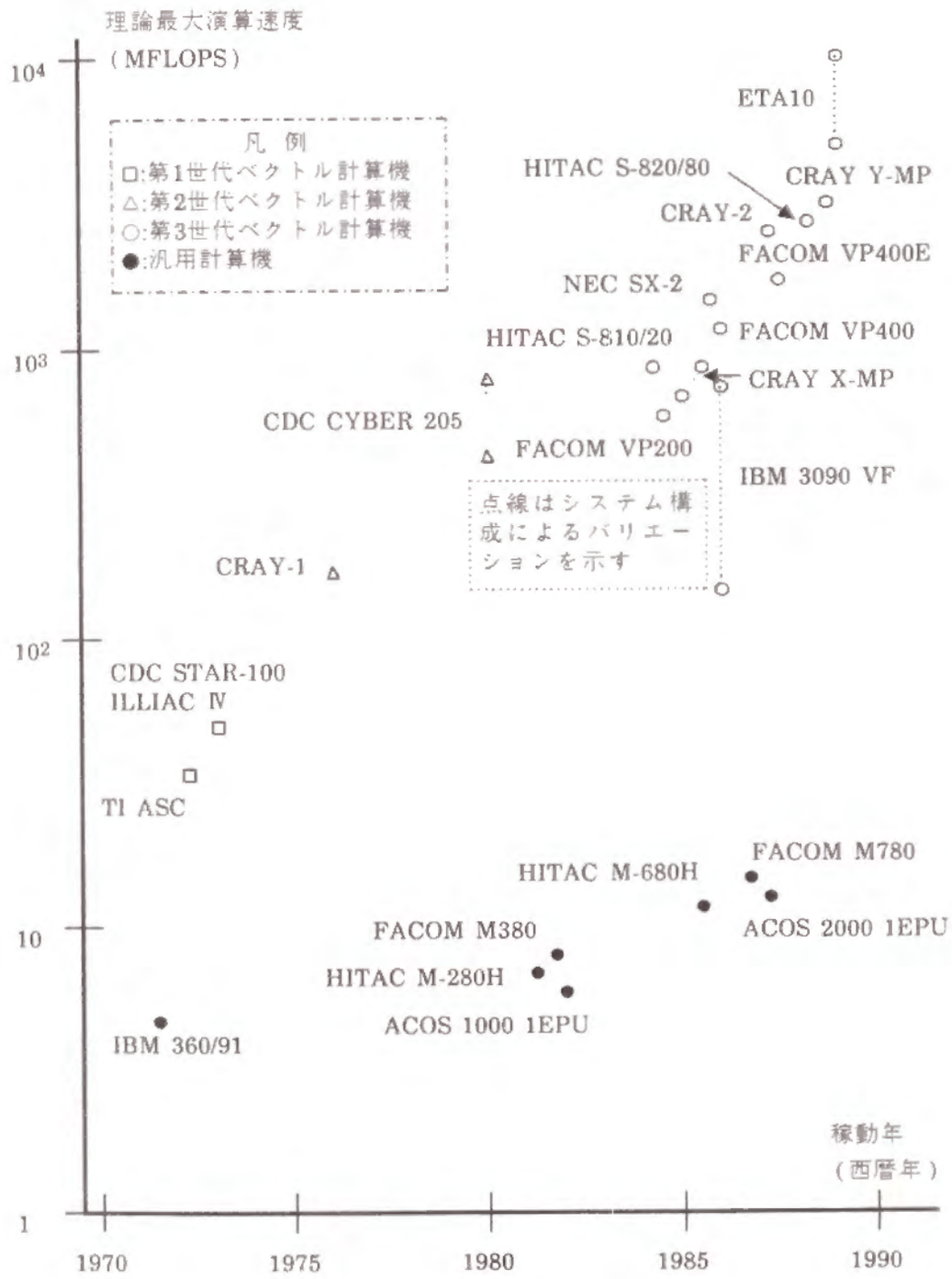


図1.1 ベクトル計算機の発達経過(文献[1]より引用)

行」と呼ぶことが一般的である。また、ベクトル処理される一連のデータをベクトル・データと呼ぶのに対し、スカラ処理されるデータをスカラ・データと呼ぶ)。この後者のスカラ実行の処理性能は、図1.1に参考のため挙げた最新の汎用計算機の処理性能である数十MFLOPS程度でしかないのに対し、ベクトル実行時の(単一のパイプライン演算器の)処理性能は数百MFLOPS以上と桁違いに速い(図1.1、表1.1に示したベクトル計算機の最大性能の数値は第2章で後述するとおり、必ずしも単一のパイプライン演算器の性能をそのまま表していないことに注意が必要である)。従って、ベクトル計算機の有する潜在的な高速性能を発揮させるためには、このパイプライン演算器を可能なかぎり有効に動作させる必要がある。すなわち、あるプログラムをベクトル計算機で高速に実行しようとした際、そのプログラムが行っている種々の演算の中から、ベクトル実行可能な部分を抽出することが重要となる。この一種のプログラム変換操作はベクトル化と呼ばれる。このベクトル化は、だれがどのように行うかの観点から以下のように大別できる。

1) ベクトル実行されるシステム標準手続/関数呼び出しによる方式

2) ベクトル処理を明示的に表記できるようプログラミング言語の構文規則を拡張する方式

3) 自動ベクトル化コンパイラによる方式

このうち、前2者ではプログラム作成者が、ベクトル実行可能性を判定し可能部分を抽出し適切に記述する必要があるのに対し、第3の自動ベクトル化では、プログラム作成者は慣れ親しんだ従来どおりの高級言語で、ベクトル計算機のアーキテクチャをほとんど意識することなく記述できる。しかも、ベクトル実行可能性の判定も含め、使用するベクトル計算機のアーキテクチャに最適と考えられるベクトル化をすべてコンパイラに委ねることができる。それゆえに、プログラム作成者の負担ははるかに小さいものとなる。また、従来から蓄積されてきたプログラムについては、プログラムの利用者は、プログラムの作成者と別人である場合が多く、通常その処理内容を詳細に調査しなければベクトル実行可能性の判定ができない。そのために、前2者の方式での人手によるプログラムの書換えを含むベクトル化は、プログラムの利用者にとって非常に困難かつ手間のかかる作業となる。それに対し、自動ベクトル化方式ではそのままベクトル計算機で処理することができる。これらの点を考慮し、近年のベクトル計算機では自動ベクトル化方式が採用されることが一般的である。ベクトル計算機が今日このように普及しつつある理由として、高速性能

もさることながら、優れた自動ベクトル化機能を持つことにより、親しみやすいユーザインタフェースを用意した点を見逃すことはできない。

このように自動ベクトル化コンパイラは、ベクトル計算機をユーザに使いやすいものとした。しかし逆に、現在ベクトル計算機の利用者は、ベクトル計算機メーカが提供する自動ベクトル化コンパイラがサポートするプログラミング言語でしかベクトル計算機を利用できない。しかも、現時点ではCRAY社をはじめ一部のメーカがFORTRANおよびそのほかの言語(Pascal、C等)を合わせた2ないし3種の言語について自動ベクトル化コンパイラを用意しているにすぎず、その他の多くのメーカは、FORTRANの自動ベクトル化コンパイラしか用意していない。さらに、現在ベクトル計算機の利用者には、パイプライン演算器を直接操作する機械語レベルでのプログラミングは許されていない。すなわち、使用するベクトル計算機のアーキテクチャを熟知した上で、そのアーキテクチャに最適なプログラムの記述を自分で行いたい利用者に、その道を閉ざしている。つまり、利用者はその数限られた既存の高級言語を通してのみベクトル計算機を使用する以外になく、ベクトル化についてもほとんどすべてコンパイラに委ねる以外にすべはない。利用者のプログラムがどの程度ベクトル化されるか、ひいてはどの程度高速実行され得るかは、すべてコンパイラの持つ自動ベクトル化に関する性能の如何によることになる。

ベクトル化率は自動ベクトル化コンパイラの自動ベクトル化機能の重要な一指標とされている。概念的にはベクトル化率とは、ベクトル実行される部分がプログラム全体に占める割合をいう。実際には、ベクトル化率はプログラムの中のベクトル実行可能部分をスカラ実行した実行時間が、プログラム全体をスカラ実行した際の実行時間に占める実行時間の比率で求めたり、ベクトル実行可能部分のスカラ命令に展開したときのステップ数が、プログラム全体のスカラ命令のステップ数に占める比率で代用したりする。いずれにせよ、ベクトル化率が高いプログラムはパイプライン演算器を有効に利用することができ、一般的に高速処理が期待できる。従って、自動ベクトル化コンパイラはまず与えられた任意のソースプログラムに対して、ベクトル化率の高い目的コードを生成できることが重要となる。そのほかにも、ベクトル計算機のハードウェア性能を十分引き出すためには、自動ベクトル化コンパイラは1度にまとめてパイプライン演算器で処理するデータの数、すなわちベクトル長、そして並列実行可能なパイプライン演算器群のスケジューリング、パイ

プライン演算器起動のためのオーバヘッド、ならびに主記憶アクセス方式等ベクトル計算機のアーキテクチャの詳細な部分までも意識する必要がある。

1.2 V-Pascal コンパイラの開発

本論文では、1.1節で述べた自動ベクトル化技術の重要性を踏まえ、A) ベクトル計算機を現状よりさらにユーザに使いやすいものとし、かつB) 高機能な自動ベクトル化を可能とすることを目的に開発された自動ベクトル化 Pascal コンパイラ V-Pascal Version 1 について、主にその自動ベクトル化機能に焦点をあて論じる。自動ベクトル化 Pascal コンパイラ V-Pascal Version 1 の大きな特徴は以下のとおりである。

1) 言語 Pascal をソース言語とすること。
2) 多重ループ全体をベクトル化対象とし強力にベクトル化を行えること。
3) 種々のベクトル計算機をターゲットマシンとして自動ベクトル化ができることを考慮し設計されていること。

第1点は、V-Pascal 開発の主目的の一つであるベクトル計算機利用者層の拡大を目指し、ベクトル計算機の利用者の選択肢を増やし、FORTRAN ユーザに限らずベクトル計算機を利用できることを目的としている。そのための第1歩としてまず、Pascal を選択した。Pascal は、豊富な型を表記でき、ブロック構造を有し、再帰的手続も記述できる等、最新のプログラミング言語にも受け継がれたり、影響を及ぼしている言語機能を数多く持っている。従って、第4章でも述べるが、V-Pascal コンパイラは次の1歩として他の手続型言語をソースとするコンパイラに変更することが容易であるように自然に設計されていると考える。第3点も、同じくベクトル計算機利用者層の拡大という目的でリターゲッタブルな設計を目指している。

次に第2点目の特長について詳述する。

2-1) ベクトル化可能性の判定に必要となる各種の依存関係解析機能が、多重ループ全体を対象とし、かつ厳密に解析できるものであること。
2-2) 同解析機能により判定した依存関係から細かな演算レベルでのベクトル化可能部分を抽出する強力な部分ベクトル化機能を有すること。
2-3) ベクトル化可能な多重ループ内の処理を、等価な1重ループ内での処理として長大なベクトル長でベクトル実行させる目的コードに翻訳する機能(一重化機能)を有すること。

従来のメーカ提供の自動ベクトル化コンパイラは、多くの場合多重ループ全体でなく最内側ループのベクトル化にしか対応できていなかった。そのため見過ごされてきた最内側ループ以外のループ中に存在する部分については、これらの機能を合わせ持つことにより、初めてベクトル化可能となった。すなわち、多重ループに関しては従来に比べ、ベクトル化率が大幅に向上する。しかも、一重化機能により長大なベクトル長でベクトル実行されるので、最内側ループだけのベクトル長でベクトル実行される場合より、さらに有効にパイプライン演算器を使用でき、数倍高速処理される場合があることを初めて実証した。このようにV-Pascalコンパイラには、メーカ提供のコンパイラにない独自の強力な自動ベクトル化機能がインプリメントされている。

第2章
ベクトル計算機のアーキテクチャ

2.1 緒 言

第1章においてベクトル計算機がどのように発達してきたかを概観した。そこで、表1.1で第3世代に分類される現在稼働中のベクトル計算機の中から、アーキテクチャが公開されている代表的なものについて概説する。まず、以下で述べるベクトル計算機の主な共通点を列挙する。

1) ベクトルレジスタを有していること。

2) 並列パイプライン構成をとっていること。

3) スカラ処理ユニットとベクトル処理ユニットを有していること。

第1点目のベクトルレジスタは、主記憶とパイプライン演算器との中間に位置する。パイプライン演算器で処理されるデータは、主記憶からいったんベクトルレジスタにロードされた後、パイプライン演算器へ送られる。演算結果は、やはりベクトルレジスタに書き戻され、その後主記憶にストアされる。このように、ベクトルレジスタを介してパイプライン演算器とデータのやりとりを行うベクトル計算機は、最初にベクトルレジスタを装備したCRAY-1の名をとり、「CRAY型」と呼ばれることがある。これに対し、初期のベクトル計算機では、主記憶とパイプライン演算器間で直接データをやりとりする形態のベクトル計算機もあった。しかし、パイプライン演算器の性能が上がるにつれ、データをフェッチする主記憶アクセスの時間が問題となることから、現在ではベクトルレジスタを装備することが主流となっている。

第2点目の並列パイプラインとは、並列に動作し得る同種あるいは異種のパイプライン演算器を複数装備しているアーキテクチャをいう。並列パイプライン方式を採用することにより、非常に高速なベクトル計算機を実現できる。この場合の最大処理性能(ピーク性能)は、並列に動作し得るパイプライン演算器それぞれの処理性能の総和で表される。しかし、実際にはベクトル化率の高いプログラムでも、その実行の最初から最後までの全過程において、並列パイプラインをすべて動作させることはほとんどない。また、ベクトル計算機の性能は、ベクトル長等によっても大きく左右される。そのため、最大処理性能はあくまでも目安にしかならないことも多い。その結果、ベクトル計算機の実効的な性能を測定することが研究され、いくつかの代表的なベンチマークスーツが開発された。Lawrence Livermore National Laboratoryで作成されたリバモアループは、その例である。

第3点目は、第1章で述べたとおりベクトル計算機が、ベクトル化できなかった部分をスカラ実行することから、当然スカラ処理を行う部分が必要となることを意味する。スカラ処理ユニットとベクトル処理ユニットとが、どの程度独立した構成となるかは、各マシンにより微妙に異なる。この観点から、ベクトル計算機は異種のプロセッサを結合したマルチプロセッサ構成を自然に採っているとみなすこともできる。さらに、ベクトル計算機を効率良く稼働させるため、通常の大型汎用機をフロントエンドにした疎結合マルチプロセッサ方式を採用することも多い。この時バックエンドのベクトル計算機は、フロントエンドの計算機から依頼されたジョブを高速に処理することに専念する。

以降の各マシンのアーキテクチャの解説においては、それぞれのメーカの用語を極力踏襲する。

2.2 HITAC S-820シリーズ

まず初めに、V-Pascalのターゲットマシンである HITAC S-820シリーズについて解説する[4],[5],[6]。図2.1にその最上位モデル S-820/80のアーキテクチャの概略を示す。この下位モデルである S-820/60では、ベクトル・レジスタ、ベクトル・マスク・レジスタの語数(保持できるデータの要素数)が半数となる。また、加算/論理演算パイプライン、乗算パイプライン、加算パイプライン、ベクトル・ロード・パイプライン、ベクトル・ロード/ストア・パイプラインの本数が半数となる。さらに、接続できる主記憶装置、拡張記憶装置の実装最大容量も半減する。

(1) 命令プロセッサ

S-820は、スカラ処理を行うスカラ・プロセッサとベクトル処理を行うベクトル・プロセッサを持つ。これらのプロセッサは、完全に独立に動作可能な設計となっている。ただし、ベクトル・プロセッサの起動はスカラ・プロセッサによる。具体的には、ベクトル・プロセッサの起動のためのスカラ命令EXVP(Execute Vector Processing)

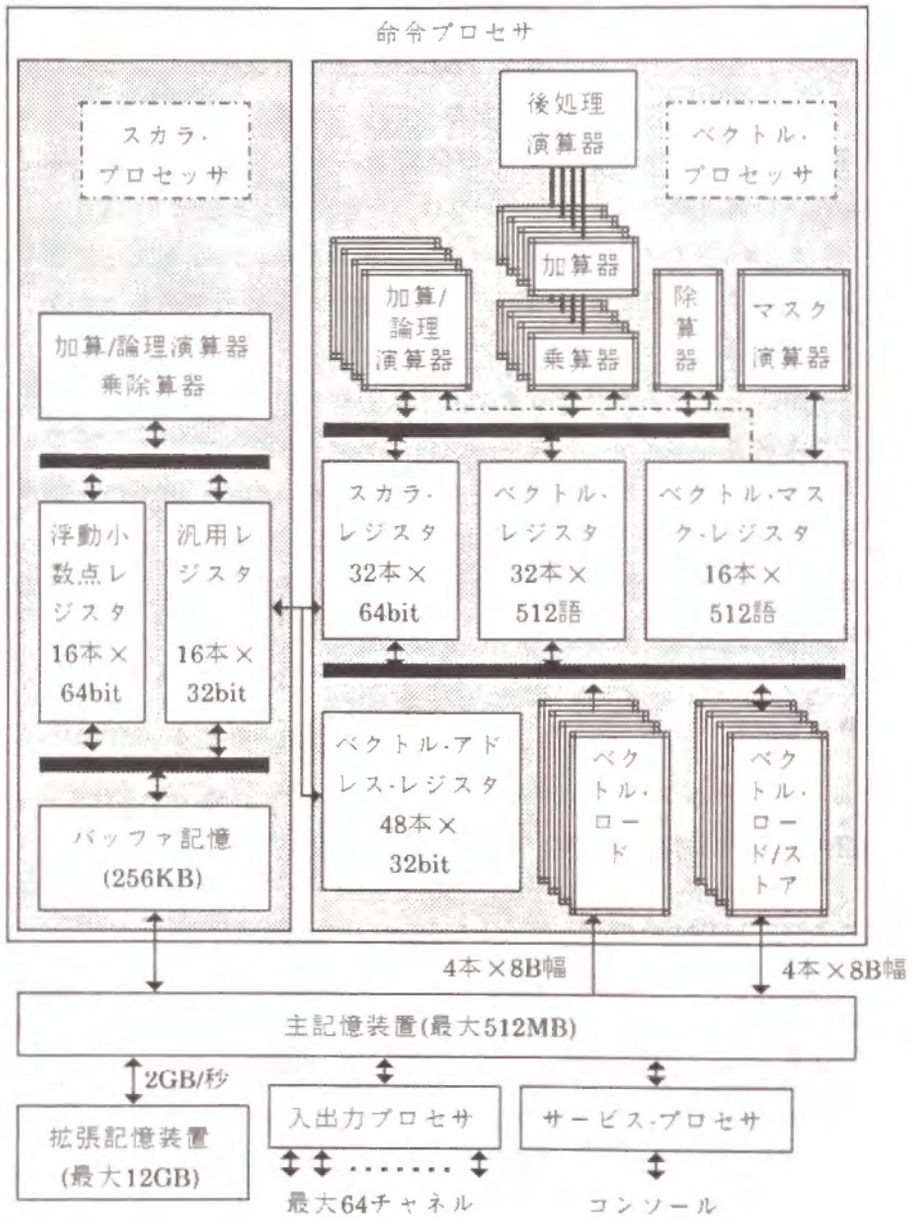


図2.1 HITAC S-820/80の構成(文献[4]より引用)

が用意されている。スカラ・プロセッサは、この命令によりベクトル長(を保持する汎用レジスタの番号)を指定するとともに、ベクトル・プロセッサに委託する一連のベクトル命令列(の先頭アドレス)を指定する。ベクトル・プロセッサが起動された後はベクトル・プロセッサに委託されたすべての全処理が終了するまで、例外的な割り込み機能を有する数種のベクトル命令を除き、ベクトル・プロセッサとスカラ・プロセッサ間では互いに通信することはできない。

(2) スカラ・プロセッサ

S-820のスカラ・プロセッサは、アーキテクチャ上からは上記のEXVP命令のようなベクトル処理に関連する部分を除き、ほとんどHITAC M680等の汎用大型機と同じとみなしてよい。実際、このスカラ・プロセッサが解釈実行する命令体系は、それらの汎用大型機と互換性を持っている。

(3) ベクトル・プロセッサ

S-820は、90種類のベクトル命令を解釈実行する、ベクトル・レジスタを持ったCRAY型ベクトル計算機である。ベクトル・レジスタを持つベクトル計算機で、ベクトル・レジスタが保持できる要素数を越えたベクトル長のデータを処理するには、ループ制御と呼ばれる処理が必要となる。これは、長いベクトル長のデータをベクトル・レジスタの要素数ずつ細かく分けた上で、その分割したデータ列ごとにベクトル命令列を適用し(ループ区分処理[5])、全データが終了するまで繰り返すものである。S-820のベクトル・プロセッサの大きな特徴は、ハードウェアでこのループ制御を行う機構を有していることである。これは、ハードウェアループ制御と呼ばれる。

(4) パイプライン演算器

S-820のベクトル・プロセッサは、図2.1に示す各種のパイプライン演算器を有する。このうち、加算/論理演算器、乗算器、加算器、ベクトル・ロード、ベクトル・ロード/ストアは4要素並列の並列パイプラインである。すなわち、これらのパイプライン演算器は、実際には同種のパイプライン演算器を4本束ねたような構造である。n個のデータの加算を例にとると以下のように4個ずつデータが並列に処理される。

第1番目の加算パイプライン演算器: $A_{4i-3}+B_{4i-3}$

第2番目の加算パイプライン演算器: $A_{4i-2}+B_{4i-2}$

第3番目の加算パイプライン演算器: $A_{4i-1}+B_{4i-1}$

第4番目の加算パイプライン演算器: $A_{4i}+B_{4i}$ ($i = 1,2,\cdots,\lceil n/4 \rceil$; A_j, B_j : データ)

さらに、ベクトル・プロセッサ内の各演算器は独立に動作できる。すなわち、レジスタの競合がない機械語命令は次々と実行され、例えば、加算/論理演算器と除算器が、並列に動作している状態等がみられることとなる。レジスタ競合がある場合でも、ハードウェアで自動的にチェイニングされる[5]。チェイニングとは、例えば、あるベクトル・レジスタに、主記憶から処理したいデータをロードし、演算した結果をベクトル・レジスタを介して、主記憶へストアする場合、これらの1)ロード、2)演算、3)ストアの3種のパイプライン演算器が、それぞれ先の処理が全要素に対して終了するのを待つのではなく、流れ作業のように先の処理が終了した要素について処理を開始することをいう。この結果、理論的には1要素分だけ遅れはするが、レジスタ競合がある場合にも、レジスタ競合している複数の演算器が、実質的に並列に動作することとなる。チェイニングの機構は、CRAY-1で初めて導入された[1]。

図2.1で乗算器の後ろにあり、直接ベクトル・レジスタと結ばれていない加算器は、内積計算で二つのベクトルの積をとり、その結果の総和を求める場合のように乗算の結果をそのまま加算するときに使用される。さらに、その背後にある後処理演算器は、内積演算、総和演算等で4要素並列に四つの部分和を求めた後、それらの総和を求めるときに使用される。

(5) ベクトル・プロセッサ内レジスタ

図2.1のベクトル・レジスタは、上記のようにパイプライン演算器の入出力(ベクトル・)データを保持する。スカラ・レジスタは同じくスカラ・データを保持する。スカラ・レジスタとスカラ・プロセッサの汎用レジスタとは、互いにデータをやりとりできる。例えば、先の内積演算の結果は、スカラ・レジスタを介して汎用レジスタに取り込み、以後の処理に利用できる。

ベクトル・マスク・レジスタは、第3章で詳述するif文のベクトル化に必要なマスク付き演算を制御するベクトル・マスク・データ(0、1のビットベクトル)を保持する。マスク付き演算では、0番ベクトル・マスク・レジスタのマスクが0あるいは1の場合にのみ選択的に演算を行う。

ベクトル・アドレス・レジスタは、ベクトル・レジスタ、スカラ・レジスタに主記憶の内容をロードしたり、ベクトル・レジスタ、スカラ・レジスタの内容を主記憶へストアしたりする際、アクセスするアドレス情報等を保持する。このレジスタへは、スカラ・プロセッサの汎用レジスタから値を取り込むことができる。ベクトル・アドレス・レジスタは、1)ベクトル・ベース・レジスタ(VBR:16本)、2)狭義のベクトル・アドレス・レジスタ(VAR:16本)、3)ベクトル増分レジスタ(VIR:16本)から構成される。VBR、VARは、それぞれIBM360/370アーキテクチャのベース・レジスタ、インデックス・レジスタの役割を果たし、両者の内容の和のアドレスが実効アドレスとなる。VIRは、ベクトル・データが主記憶上で等間隔に並んでいる場合、その間隔を指定するものである。VARの内容は、上述のループ制御により、自動的に更新される。

ハードウェアループ制御のため、ベクトル長レジスタ(VLR)、ベクトル長実行レジスタ(VER)が用意されている。これらは、ベクトル実行開始時にそれぞれ、ベクトル長あるいは0に初期化され、ループ区分処理された要素数だけ、減ぜられたり、あるいは、増やされたりする。ベクトル実行開始時にVLRにベクトル長をセットする以外には、機械語レベルでもこれらのレジスタを操作することはできない。

(6) 拡張記憶

ベクトル計算機の処理能力が向上するのにあわせ、より多くのデータを記憶でき、かつ高速にアクセスできる記憶装置が必要となった。S-820は、S-810と同じく拡張記憶を搭載することにより、この要求に答えようとしている。拡張記憶は、主記憶より巨大であるが、アクセス時間は大きくなっている。拡張記憶とレジスタ間で直接データをやりとりできない。これは、拡張記憶がI/Oネックを避ける目的で導入された、高速な作業ファイルとしての利用を前提としているからであると思われる。しかし視点を変え、主記憶に収容できないような巨大な配列データを拡張記憶上に保持し、スカラ命令の一種である主記憶との転送命令(同期、非同期の2種)を使って、部分ごとに主記憶との間でデータ転送を行い、処理を進めるような高度な利用も可能である。

2.3 FACOM VP シリーズ

次にFACOM VPシリーズについて解説する[7]~[10]。ただし、1.1節で触れた最新のFACOM VP-2000シリーズについては、現時点で詳細なアーキテクチャが公開されていないため省く。FACOM VPシリーズは多くのモデルを持つ。その中で、図2.2はFACOM VPシリーズEモデルの、主としてその最上位モデルVP-400Eのアーキテクチャの概略を示している。この下位モデルであるVP-200E、VP-100Eでは、ベクトル・レジスタ、マスク・レジスタの総容量が順に半減する。また、各ベクトル・パイプラインの並列の本数が半数となる。さらにVP-100Eでは、接続できる記憶装置

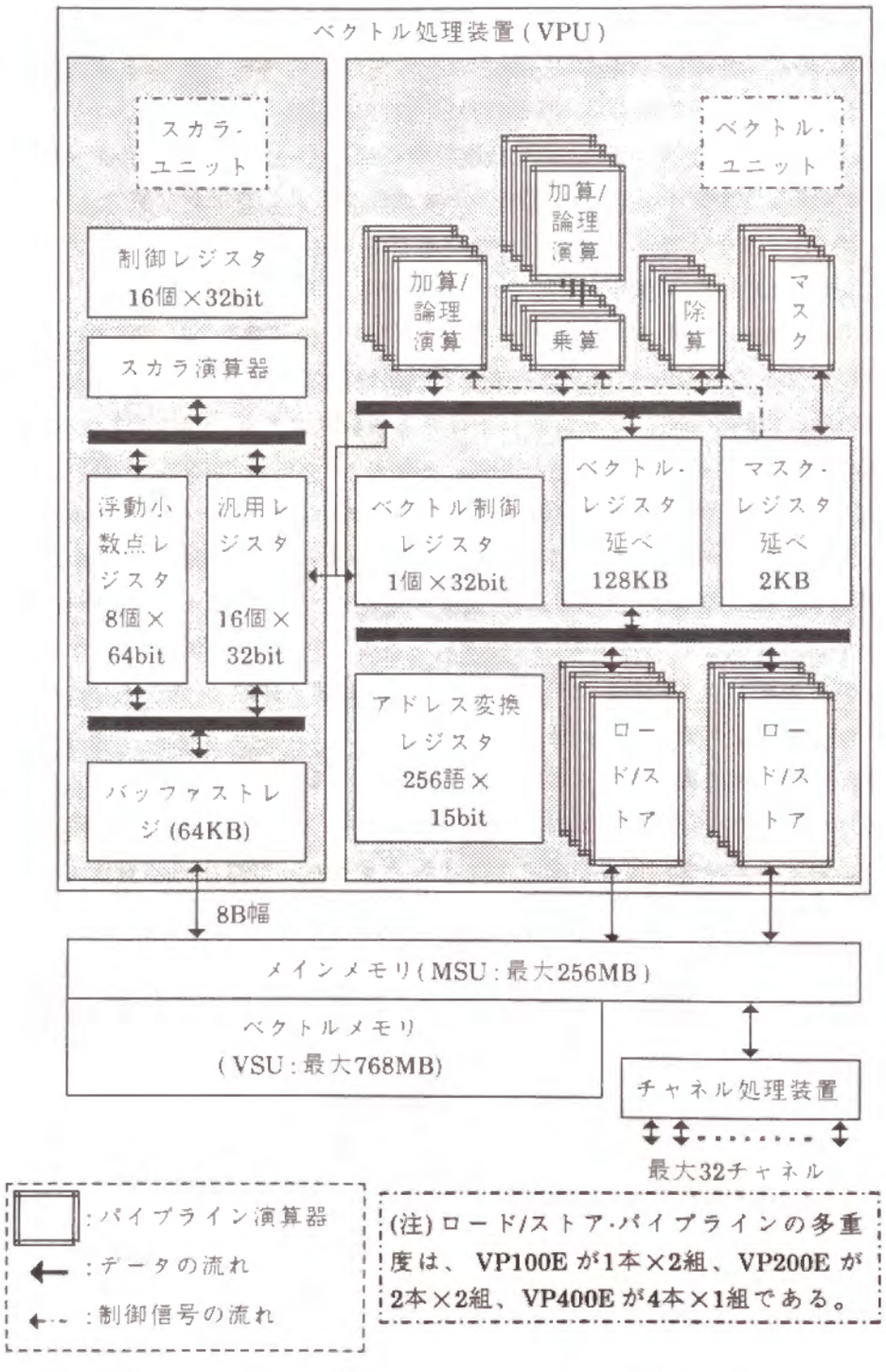


図2.2 FACOM VPシリーズEモデルの構成[7],[8],[9]

の容量も半減する[8]。

(1) ベクトル処理装置(VPU)

VPシリーズはS-820同様、スカラ処理を行うスカラ・ユニットとベクトル処理を行うベクトル・ユニットを持つ。これらのプロセッサは、S-820同様完全に独立に動作可能な設計となっている。ただし、S-820と異なり、スカラ命令とベクトル命令とは混在してよく、スカラ・ユニットが全命令をフェッチ/デコードする。もし、ベクトル命令であったなら、ベクトル・ユニット(のベクトル命令制御部)に送られ、ベクトル・ユニットで実行される。ベクトル・ユニットへのベクトル命令転送後、スカラ・ユニットは独自に動作できる。すなわち、例えば次の命令がスカラ命令であったなら、その実行を行えるし、またもし次の命令がベクトル命令であったなら、ベクトル・ユニットが起動中でも、ベクトル・ユニットへ転送することができる。つまり、スカラ・ユニットは、命令ごとにベクトル・ユニットに処理を依頼する形態である。従ってS-820と比較すると、スカラ・ユニットとベクトル・ユニット間の連携が密であるといえる。

(2) スカラ・ユニット

VPシリーズのスカラ・ユニットも、アーキテクチャ上からは上記のようなベクトル処理に関連する部分を除き、FACOM Mシリーズのような汎用大型機と同じとみなしてよい。実際、解釈実行する命令体系は、それらの汎用大型機と互換性を持っている。

(3) ベクトル・ユニット

VPシリーズEモデルは、98種類のベクトル命令を解釈実行する。これも、ベクトル・レジスタを持ったCRAY型ベクトル計算機である。ただし、S-820と異なり、ハードウェアでループ制御を行う機構を持っていない。従って、ループ制御は機械語レベルでソフトウェアで実現する必要がある(ソフトウェアループ制御)。

(4) パイプライン演算器

VP-400Eのパイプライン演算器は、図2.2に示したとおりすべて4要素並列の並列パイプラインである。さらに、レジスタの競合がない場合これらのパイプライン演算器のうち、ロード/ストア・パイプライン、2種類の任意の演算パイプライン、マスク・パイプラインが並列に動作できる。また、ハードウェアでのチェイニングも行われる。なお図2.2でロード/ストア・パイプラインは2組描いているが、2組装備されているのはVP-200E、VP-100Eだけであり、VP-400Eには1組しかない。

乗算パイプラインの後ろにあり、直接ベクトル･レジスタと結ばれていない加算/論理演算パイプラインは、S-820同様内積計算で二つのベクトルの積をとり、その結果の総和を求める場合のように乗算の結果をそのまま加算するときに使用される。

(5) ベクトル･ユニット内レジスタ

ベクトル･レジスタは、S-820同様パイプライン演算器の入出力(ベクトル･)データを保持する。S-820のスカラ･レジスタ、(広義の)ベクトル･アドレス･レジスタ群に相当するものは無く、スカラ･ユニットの汎用レジスタがこれらの機能を果たす。

マスク･レジスタは、S-820同様第3章で詳述するif文のベクトル化に必要なマスク付き演算を制御するベクトル･マスク･データ(0、1のビットベクトル)を保持する。マスク･レジスタとベクトル･レジスタは、ともに可変構造を特徴とする。すなわち、例えばS-820のこれらのレジスタは、モデルで固定の要素数のベクトル･データを保持する固定した本数しか使用できなかったのに対し、総容量の制限内で保持できる要素数と使用できる本数が機械語レベルで切り換え可能である。例えば、VP-400Eでは、64要素(ベクトル･レジスタの1要素は64ビット、マスク･レジスタ1要素は1ビット)のレジスタ256本が基本で、隣接するレジスタを連結して用いることにより、128要素×128本、256要素×64本、512要素×32本、1024要素×16本、2048要素×8本のいずれかの構成を選択できる。

ベクトル制御レジスタは、上記の可変レジスタ構成にも関係するベクトル長、プログラム割込みコード、状態制御の情報を保持する。ベクトル制御命令で操作する。

アドレス変換レジスタは、名前が示すとおりアドレス変換のために使用される。

(6) 記憶装置

VPシリーズEモデルは、通常の主記憶装置であるメイン･メモリ(MSU)とは別にベクトル･メモリ(VSU)を持つ。この拡張された記憶装置は、ベクトル計算機のメモリの不足を緩和する目的で導入された。ベクトル･メモリ(VSU)は、S-820の拡張記憶程大きくはないが、アクセス時間はメイン･メモリ(MSU)と全く同じである。すなわち、メイン･メモリがそのまま広げられた形である。ただし、ベクトル･メモリ(VSU)へのアクセスはベクトル･ユニットからのみ可能である。従って、ベクトル命令でのみ参照される巨大配列を割り付ける領域として、ならびにベクトル･レジスタの待避領域としての使用が考えられる。

2.4 NEC SX シリーズ

次にNEC SXシリーズについて解説する[11],[12]。ただし、1.1節で触れた最新のNEC SX-3シリーズについては、現時点で詳細なアーキテクチャが公開されていないため省く。図2.3はNEC SX-2シリーズのアーキテクチャの概略を示している。この下位モデルであるSX-1、SX-1Eでは、ベクトル･レジスタ、ベクトル･マスク･レジスタの総容量が順に半減する。また、各ベクトル･パイプラインの並列の本数も順に半数となる。さらにSX-1Eでは、接続できる演算プロセッサ･メモリの最大容量も半減する[12]。

(1) 演算プロセッサ(AP)と制御プロセッサ(CP)

SXシリーズでは、中央処理装置に相当する科学演算処理装置(SPU)が演算プロセッサ(AP)と制御プロセッサ(CP)と呼ばれるまったく独立した2プロセッサからなる。制御プロセッサ(CP)は汎用機と同等の機能を有し、オペレーティング･システム的な機能(ジョブの入出力制御、資源管理、ファイル処理等)、FORTRANプログラムのコンパイル等を実行できる。すなわち、2.1節で述べた汎用機をフロント･エンドに持つ疎結合マルチ･プロセッサ構成を採ることをスタンド･アロンで実現するため、フロント･エンドの汎用機をそっくりそのまま包含し、演算プロセッサ･メモリを共有する密結合マルチ･プロセッサ構成となっている。従って、先のような使用だけでなくユーザプログラムの実行も可能である。ユーザジョブをタスク分割し演算プロセッサ(AP)と同時並行動作も可能である。

演算プロセッサ(AP)はS-820、VP同様、スカラ処理を行うスカラ･ユニットとベクトル処理を行うベクトル･ユニットに分かれる。これらのプロセッサは、やはり完全に独立に動作可能な設計となっている。ベクトル･ユニットの起動に関してはVPと同じである。すなわち、スカラ命令とベクトル命令とは混在してよく、スカラ･ユニットが全命令をフェッチ/デコードする。もし、ベクトル命令であったなら、ベクトル･ユニットに送られ、ベクトル･ユニットで実行される。ベクトル･ユニットへのベクトル命令転送後、スカラ･ユニットは独自に動作できる。

(2) スカラ･ユニット

SXシリーズのスカラ･ユニットは、S-820、VPと異なり、アーキテクチャ上からは汎用大型機とは別に設計された新しいRISCアーキテクチャに近いものである。ま

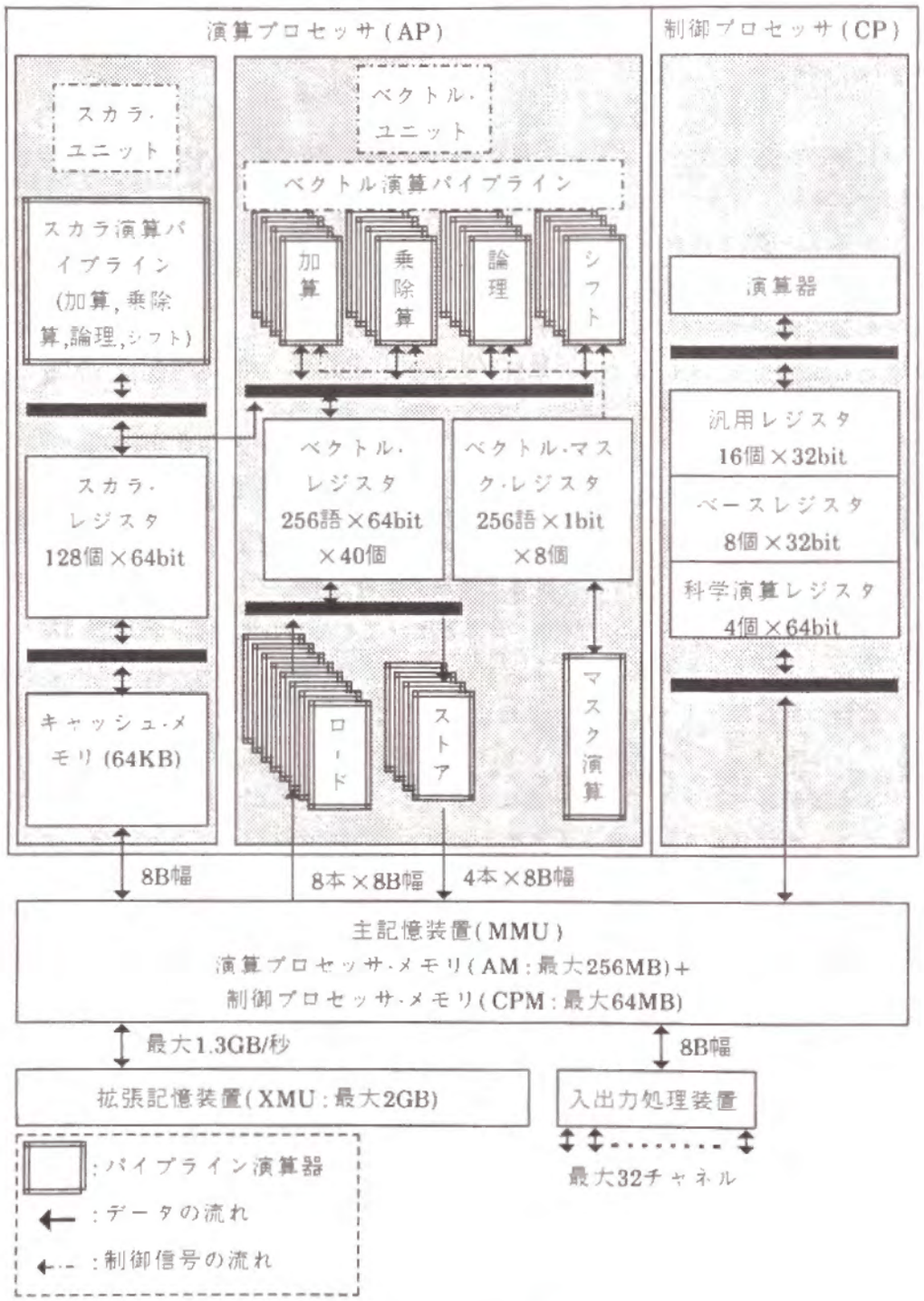


図2.3 NEC SX-2の構成[11]

た、汎用レジスタの数は128個と非常に多い。スカラ命令についても、演算器はパイプライン化されており、命令フェッチ(instruction fetch)、デコード(decode)、アドレス計算(address calculation)、オペランド・フェッチ(operand fetch)のみならず、実行(execution)フェーズも連続的に実行される。ただし、これは先行命令と使用する演算器およびレジスタが異なる場合についてである[11]。そのため、上述のとおり多数のレジスタを持たせ、レジスタ競合が生じにくいアーキテクチャとなっている。

(3) ベクトル・ユニット

SXシリーズも、ベクトル・レジスタを持ったCRAY型ベクトル計算機である。やはり、S-820が有するようなハードウェアでループ制御を行う機構を持っていない。従って、ループ制御は機械語レベルでソフトウェアで実現する必要がある。

(4) パイプライン演算器

SX-2のベクトル演算パイプラインは、図2.3に示したとおりすべて4要素並列の並列パイプラインである。さらに、レジスタの競合がない場合これらの演算パイプラインは並列に動作できる。また、ハードウェアでのチェイニングも行われる。ロードパイプラインは8要素並列、ストア・パイプラインは4要素並列である。

(5) ベクトル・ユニット内レジスタ

ベクトル・レジスタは、S-820同様パイプライン演算器の入出力(ベクトル・)データを保持する。SXシリーズにおいてもS-820のスカラ・レジスタ、(広義の)ベクトル・アドレス・レジスタ群に相当するものは無く、スカラ・ユニットの汎用レジスタがこれらの機能を果たす。

マスク・レジスタは、S-820同様第3章で詳述するif文のベクトル化に必要なマスク付き演算を制御するベクトル・マスク・データ(0、1のビットベクトル)を保持する。

(6) 記憶装置

SXシリーズは、主記憶装置(MMU)とS-820の拡張記憶に相当する拡張記憶装置(XMU)を持つ。拡張記憶装置(XMU)は、複数個の仮想ディスク・ボリュームとして管理され、高速アクセス・ファイルとして使用される。従って、FORTRANの実行時入出力ルーチンにより、主記憶とデータ転送を行う。この点でもS-820の拡張記憶に近い。この拡張記憶装置の導入目的は、ベクトル計算機の処理性能に見合った高速な入出力を達成することにある。主記憶装置(MMU)は、演算プロセッサ・メモリ(AM)と制御プロセッサ・メモリ(CPM)とに分かれる。制御プロセッサ・メモリ

(CPM)が制御プロセッサ専用であり、前述のとおりオペレーティング・システムの機能の大部分がこのメモリ上で動作するのに対し、演算プロセッサ・メモリ(AM)は演算プロセッサと制御プロセッサとで共有され、純粋にユーザプログラムのために使用される。

2.5 結 語

最新のベクトル計算機のアーキテクチャを概観するため、国産3社のベクトル計算機のアーキテクチャについて簡単に解説した。これらの解説においては、メーカから公開されている資料に頼らざるを得ないため、細部に立ち入ることができない。ところが、ベクトル計算機を有効に機能させるためには、そのアーキテクチャの細部こそが重要となる。例えばV-Pascalコンパイラの大きな特徴である間接参照命令による多重ループの一重化では、間接参照命令の性能が大きく影響を受ける。この性能は、ロード/ストア・パイプラインの本数あるいは並列にロード/ストアできる要素数といったカタログ上に現れる数値よりも、ロード/ストア・パイプラインがどの程度豊富なアドレッシング機構を有しているかに関係する。具体的には、S-820シリーズでは十分なアドレッシング機構を有しているため、直接参照命令と同等のアクセス・スピードである。これに対し、SXシリーズでは直接参照命令では、ロード/ストア・パイプラインが並列に8あるいは4要素にアクセスできるが、間接参照命令ではともに1要素にしかアクセスできない。すなわち、直接参照命令に比べ8あるいは4倍遅いことになる。VPシリーズでも同様で、例えばVP-400(E)、VP-200(E)の間接参照命令の処理速度は、直接参照命令の処理速度に比べ約3倍程度遅い。この例から明らかなように、ベクトル計算機のカタログ上に現れる最大処理性能は、(演算用の)並列パイプラインの1本の性能と本数とで決まるが、実効的な性能は、こうしたカタログ上に現れてこないアーキテクチャにも大きく依存する。そして、何よりもベクトル計算機の実効的な性能を最終的に決定するものは、1.2節で触れたとおりソフトウェアである自動ベクトル化コンパイラの能力である。すなわち、ベクトル計算機の性能を真に引き出させるには、コンパイラが上記のようなカタログ上に現れてこないアーキテクチャの細部までを意識し、それに応じて適切な目的コードを生成する必要がある。

第3章
自動ベクトル化

3.1 緒 言

第1章で述べたとおりスーパーコンピュータの一種であるベクトル計算機は、パイプライン方式の演算器(パイプライン演算器、演算パイプライン等と呼ばれる)を持つベクトルプロセッサにより、図3.1に示すような流れ作業で、複数個のデータに同一の演算、処理を施し高速処理を達成している。

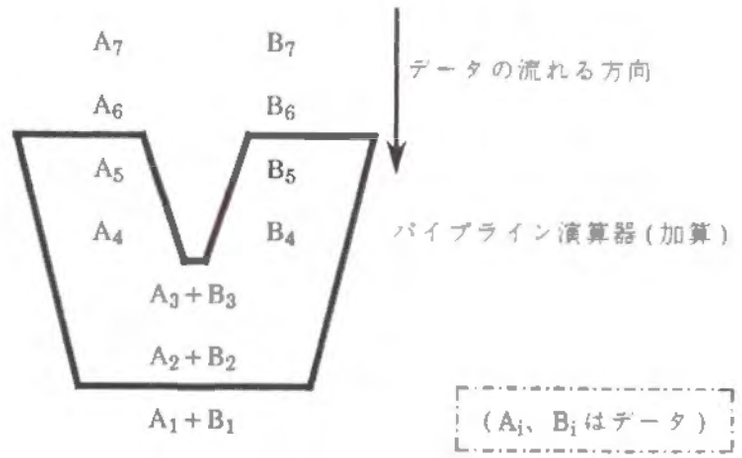
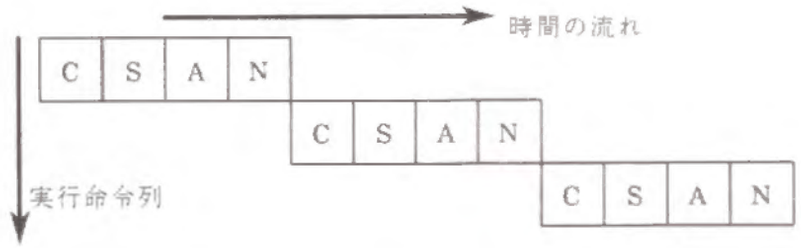


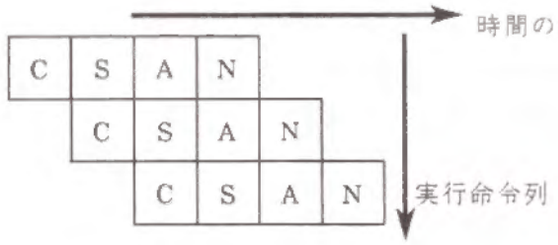
図3.1　パイプライン演算器の動作概念図

従来の異種あるいは同種の命令列に対するパイプライン制御では、各命令を実際の命令実行のフェーズおよびその準備段階の命令読み出し、命令デコード、アドレス計算、アドレス変換、オペランド読み出し等のフェーズに分けて、それらを時間的にオーバーラップさせた。それと同様に、パイプライン演算器は、実際の命令実行のフェーズ自体を複数のステージ(stage)と呼ばれる小さな作業に分割し、各ス

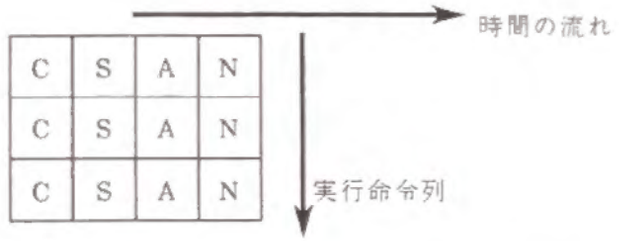
テージの実行までもオーバーラップさせ高速化していると見ることができる(図3.2参照)。この1ステージをデータが通過するのに必要な時間を(パイプラインの)ピッチと呼ぶ。このピッチは通常マシン・サイクルに等しく設計されることが多い。



(a)従来の命令列に対するパイプライン制御(先行制御)の場合、実際の実行はオーバーラップしていない。



(b)パイプライン演算器では各ステージ(注)が同時に動作し実行も一部オーバーラップする。



(c)マルチプロセッサによる並列処理では各実行が完全にオーバーラップする。

(注) C(Compare):指数部の比較、S(Shift):仮数部の桁合せ、A(Add):仮数部の加算、N(Normalize):加算後の正規化の4ステージに分割したものと仮定する。

図3.2 パイプライン制御、パイプライン演算器、マルチプロセッサの比較(概念図)

従って、高速なパイプライン演算器を設計するには、いかにマシン・サイクルを短くできるかに大きく依存する。このピッチに対し、最初のデータがパイプライン演算器に到達するまでの時間を**演算開始時間**あるいは**スタートアップ時間**と呼ぶ。この演算開始時間は、パイプライン演算器を動作させるために必要なオーバーヘッドの時間とみなせる。

このようなハードウェアの構造から、ベクトル計算機の高速性能を引き出すためには、ベクトル処理されるデータ(ベクトルデータ)の量を可能な限り多くすることが、最も重要となる。しかも、その場合パイプライン演算器が同一の演算をまとめて施すので、一連の同一演算を抽出しなければならない。さらに、パイプライン演算器で1度に処理されるデータ数(ベクトル長)が大きい程、スカラ実行する場合と比べて加速効果は大きくなる。逆に、ベクトル長が短いとベクトルプロセッサを起動するオーバーヘッドのため、スカラ実行するより遅くなることがある。このスカラ実行した場合と同じ速度となるベクトル長を**限界ベクトル長**あるいは**交叉ベクトル長**という。従って、ベクトル長が限界ベクトル長以上であれば、ベクトル実行する方が高速であることになる。第2章で挙げたベクトル計算機の場合、この限界ベクトル長は、3から10程度である[1]。

3.2 ベクトル化と依存関係

1.2節で述べたとおり、ベクトル化とは与えられたプログラムからベクトル実行可能部分を抽出する作業である。これは、ある種の並列性検出を行うプログラム変換と見ることもできる。ベクトル化というプログラム変換を行う場合、与えられた元のプログラムの意味を変えないことを保証するために、ベクトル実行しても種々の処理の順序が入れ替わらないかを解析する必要がある。この順序関係(先行関係)の解析を**依存関係解析**と呼ぶ。この依存関係には、データ参照に起因する順序関係と制御の流れに起因する順序関係とがある。前者は、**データ依存関係**あるいは単に**データ依存**(data dependence)[13],[14]と呼ばれ、後者は**制御依存関係**あるいは単に**制御依存**(control dependence)[15]と呼ばれる。

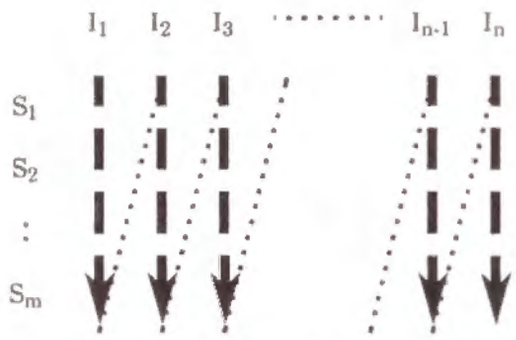
ところで、ベクトル化の対象となるような条件を満たす演算群は、ループ、それも繰り返し回数の多いループ内に検出できる。そこで、1.2節で述べた自動ベクトル化の場合にも、通常ベクトル長が簡単に求められる、すなわち繰り返し回数の明ら

かなFORTRANのDOループの形式のループ(以後単にDOループと記す)、Pascalのforループの形式のループ(以後単にforループと記す)を通常対象としている。以下で使用するプログラム例はFORTRANあるいはPascalで記述するものとする。

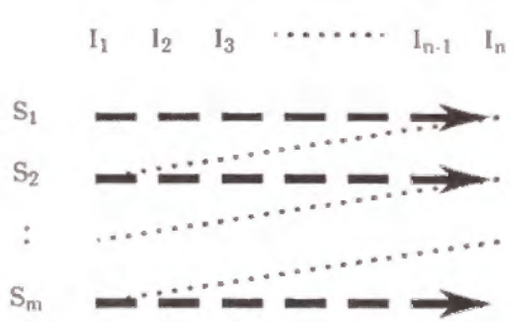
3.2.1 ベクトル化による実行順序の変更

ベクトル化においては、同一の演算をまとめるため、演算の実行順序を変更することが多く見られる。例えば、DOループ内に複数の文が存在する場合、ベクトル化すると図3.3に示すようにスカラ実行時と比べ実行順序が異なる。

このような演算の実行順序の変更は、1文中の複数の演算についても生じる。すなわち、それらの演算がスカラ実行の実行順序でOP_1、OP_2、‥、OP_mであったとす



(a)ベクトル化前のスカラ実行の実行順序



(b)ベクトル化後の(2.2節で述べたループ制御が不要の単純な場合の)ベクトル実行の実行順序

(S_i:i番目の文、I_j:j番目の繰り返し)

図3.3 DOループのベクトル化による実行順序の変更

ると、図3.3においてS_iをOP_iに置き換えて考えればよい。また、通常のマルチ・パス構成のコンパイラが持つ三つ組、四つ組等の内部表現では、こうした中間コード単位に置き換えて考えればよい。いずれにしても、ベクトル化するとスカラ実行時と比べ実行順序が異なるといえる。

```
      DO  10  I=1,N
              S1
              S2
              :
              Sm
10    CONTINUE
```

(S_i:図3.3に同じ)

(a)スカラ実行の実行順序を表現

```
      DO  1   I=1,N
              S1
1     CONTINUE
      DO  2   I=1,N
              S2
2     CONTINUE
              :
      DO  m   I=1,N
              Sm
m     CONTINUE
```

(a)では単一のループで構成されていたが、(b)ではm個のループに分割されている(ループ分割)。

(b)ベクトル実行の実行順序を表現

図3.4 ベクトル化のソースレベルでの表現

図3.3の実行順序の変更をソースプログラムレベルで記述したものが図3.4である。同図中にも記したとおり、ベクトル化はソースレベルで考えれば、文単位にループを分割することに相当するといえる。すなわち、ループ構造がm個の文すべてに複製されたとみなせる。

以上、1重ループのベクトル化について図示ならびに解説してきたが、これらのループが多重ループであっても、メーカ提供の自動ベクトル化コンパイラでは、通常最内側ループのみをベクトル化対象としており、最内側ループのみの実行順序がこれらの図のとおり変更される。それに対し、V-Pascalは多重ループ全体をベクトル化対象とし、適切であると判断すれば、多重ループを1重ループ化しベクトル化する。図3.4でいえば、元の多重ループ構造すべてを複製することになる。多重ループのベクトル化については3.3節で詳述する。

3.2.2 データ参照関係とデータ依存

先にも述べたとおりベクトル化においては、そのプログラム変換により意味を変えないためには、スカラ実行時のデータ参照の順序とベクトル実行時のデータ参照の順序とが変わらないことが必要となる。例で考えることにする。

[例1]ベクトル化可能(データ参照関係適)

```
      DO  10  I=1,3
          A(I+1)=·····          (S1)
          ·····=····A(I)··      (S2)
10    CONTINUE
```

この例1でS_1の左辺に出現するA(I+1)、S_2の右辺に出現するA(I)について、制御変数Iの変化に伴い配列Aの何番目の要素が参照されるかを表3.1に示す。なお以下では、変数が代入文の左辺に出現する時等、値の変更を伴う参照を「定義」、代入文の右辺に出現する時等、値が変更されない参照を「引用」と呼び区別する。

表3.1の参照パターンから、A(2)、A(3)について定義と引用が衝突(同一配列要素あるいは同一変数を参照)していることがわかる。そこで、これらに着目し表3.1の参照パターンに前記の図3.3のスカラ実行、ベクトル実行それぞれの実行順序を重ね合わせてみると、A(2)、A(3)のどちらも表3.2に示すとおりベクトル実行しても定義と引用の順序関係(データ参照関係)は変わらないことがわかる。このような場合を**データ参照関係適**と言い、このままベクトル化が可能であることを意味する。

表3.1 例1で2出現が参照する要素の添字番号

Iの値	1 (I_1)	2 (I_2)	3 (I_3)
S_1の定義	②	③	4
S_2の引用	1	②	③

(S_i、I_i は図3.3と同じ記法)

表3.2 例1で参照が衝突している要素の実行順序比較

	スカラ実行	ベクトル実行
A(2)	定義 → 引用 (S_1,I_1) (S_2,I_2)	定義 → 引用 (S_1,I_1) (S_2,I_2)
A(3)	定義 → 引用 (S_1,I_2) (S_2,I_3)	定義 → 引用 (S_1,I_2) (S_2,I_3)

(S_i、I_i は図3.3と同じ記法)

[例2]ベクトル化不可能(データ参照関係不適)

```
      DO  10     I=1,3
          ·····     =····A(I)··     (S1)
          A(I+1)=·····               (S2)
10    CONTINUE
```

次の例2では、このデータ参照関係がベクトル化により変わってしまう(表3.3、表3.4参照)。このような場合を**データ参照関係不適**と言い、少なくともこのままではベクトル化が不可能であることを意味する。

要するに、データ参照関係のベクトル化に対する適/不適は、上述のようなデータ参照(の順序)関係のチェックを行うことにより判定できる。ところが、不適な場合

表3.3 例2で2出現が参照する要素の添字番号

Iの値	1 (I_1)	2 (I_2)	3 (I_3)
S_1の引用	1	②	③
S_2の定義	②	③	4

(S_i、I_i は図3.3と同じ記法)

表3.4 例2で参照が衝突している要素の実行順序比較

	スカラ実行	ベクトル実行
A(2)	定義 → 引用 (S_2,I_1) (S_1,I_2)	引用 → 定義 (S_1,I_2) (S_2,I_1)
A(3)	定義 → 引用 (S_2,I_2) (S_1,I_3)	引用 → 定義 (S_1,I_3) (S_2,I_2)

(S_i、I_i は図3.3と同じ記法)

でも、先の例1および例2からわかるように、例2の2文の順序を入れ替え例1のパターンに変更しても、配列A以外のデータ参照関係が変わらなければ、例1のパターンでベクトル化できる。現在の自動ベクトル化では、このような入れ換えによるさらに大幅な演算順序の変更も行いベクトル化を促進させるよう工夫されている。従って、データ参照関係の解析は、ベクトル化適/不適の判定よりむしろこのようなスカラ実行時の保存されるべき順序関係であるデータ依存(の存在とその向き)を調査することになる。先の例1では、文S_1→文S_2の向きのデータ依存が存在する。それに対し例2では、文S_2→文S_1の向きのデータ依存が存在する。従って自然な考え方で、この順序関係を保存するには、例2については2文を入れ換えればよいと判定できる。

データ依存を調べるためには、理論的には調査するループ内に出現するすべての同一配列および単純な変数の2出現のあらゆる対についてチェックしなければならない。ただし、2出現が引用-引用の組み合わせの場合には、たとえデータ参照の順序が変わったとしても、プログラムの意味は変わらないので、チェックする必要はない。

表3.5 定義/引用の順序によるデータ依存の分類

用語	意味
フロー依存(flow-dependence)	定義 → 引用
逆依存(anti-dependence)	引用 → 定義
出力依存(output-dependence)	定義 → 定義
入力依存(input-dependence)	同一ファイルから入出力を行う2文間

表3.6 ループの回転に着目したデータ依存の分類

用語	意味
ループ独立依存 (loop-independent dependence)	ループの同一の繰り返しにおいて参照が衝突 (衝突時の2出現の制御変数の値が同一)
ループ運搬依存 (loop-carried dependence)	ループの異なる繰り返しにおいて参照が衝突 (衝突時の2出現の制御変数の値が異なる)

例4によく似た次の例6では、制御変数Iのループおよび制御変数Jのループの入れ

[例6]多重ループ内のデータ依存(その3)

```
      DO  10  J=1,3
          DO 10    I=1,3
             B(J,I)=····A(J,I)··      (S1)
             A(J,I+1)=····B(J,I)··    (S2)
10        CONTINUE
```

子構造が例4と逆転している。この場合にも例4とまったく同じ議論が成り立ち、制御変数Iのループをスカラ実行すれば、制御変数Jのループでベクトル化できる。すなわち、例6ではまずループの入れ子構造を逆転(交換)して例4のようなプログラムとした後、ループ選択機能によりベクトル化すればよい。このようなループ交換を含むループ選択機能のためには、各ループについてそれが最内側にあるとして、ループ独立依存かループ運搬依存かを調べることが必要である。すなわち、多重ループにおけるデータ依存解析では、最初に述べたとおり、各制御変数のループごとに独立にループ独立依存かループ運搬依存かを判定するのである。

このように、これらの詳細な情報(どの出現からどの出現へデータ依存があるのか、そしてその依存の種類は何か)を用い、3.3節で述べる配列化等を含む詳細なベクトル化を行うことが可能となる。

3.2.3 制御依存

制御依存は、条件分岐により制御の流れが切り替わるところで発生する。条件分岐の条件節の実行が終了し、真偽値が確定した後でなければ、条件分岐の真側のパスも偽側のパスも実行できない。言い換えれば、条件分岐の条件節を構成する文(中間コード)は、その条件分岐の影響を受ける真側のパスならびに偽側のパス上の全文(全中間コード)に対して先行しなければならない。この先行関係を制御依存と呼んでいる[15]。例えば、if-then-else構造の場合は、ifの条件節からthen節ならびにelse節への制御依存がある[19]。

3.2.4 依存グラフ

3.3節で述べる自動ベクトル化では、データ依存および制御依存の抽出、さらに、ベクトル化可能性の判定を計算機で行わせる必要がある。その場合、求められた

データ依存ならびに制御依存は、計算機内部では依存グラフ(dependence graph)の形にまとめられることが多い[13],[14]。この依存グラフは、依存を解析する際の単位となる文あるいは中間コードを節(ノード:node)とし、依存の存在を有向辺(arc)とする有向グラフである。しかも、各有向辺は表3.5、表3.6のように分類されている(付加的な情報を保持している)[13],[14]。V-Pascalコンパイラでは、この有向グラフを4.3.5項で述べるD行列として構成している。

3.3 自動ベクトル化技術の現状

第2章で解説した国産3社のベクトル計算機のために、各社が用意しているFORTRANコンパイラは、第1章で述べたとおり、自動ベクトル化機能を有している。発表当初のコンパイラの自動ベクトル化機能は、現在の水準から比べるとかなり低いものであった。例えば、初期においては対象ループ中に1ヵ所でもベクトル化を阻む要因が存在した場合には、ベクトル化を完全に諦めそのループ全体をスカラ実行してしまっていた。それに対し、現在ではその種のベクトル化を阻む要因に関係する文(演算)を切り離し、少しでもベクトル実行できる部分を抽出する部分ベクトル化機能が強力になりつつある。

この節では、世界最高水準にあると言われている日本のコンパイラの自動ベクトル化機能、特に自動ベクトル化の対象について、簡単に述べる。なお、この記述および例は、参考文献[20]～[22]に示したマニュアル類より抜粋、要約したもので、特に明記しない限り国産3社のコンパイラがともに有する機能である。特に自動ベクトル化の対象に話題を絞った理由は、これらの文献では、自動ベクトル化の対象に関しては、具体的な記述が多いが、自動ベクトル化機能がコンパイラ内部で如何に実現されているかについては、ほとんど明らかにされていないからである。

また、自動ベクトル化機能でコンパイラがベクトル化可能性を判定できない部分、ベクトル化しても遅くなる部分等に対しては、プログラマがベクトル化に関する指示を行えるコンパイラオプション(ベクトル化制御、ベクトル化指示、最適化制御等と呼ばれる)が、どのコンパイラにも用意されていて、さらに細かなベクトル化を人間が補佐できるようになっているが、本論文では省略する。

データ依存は、さらに表3.5、表3.6に示す詳細な分類ができる[13],[14],[16],[17]。ただし、文献[18]をはじめ、我々の研究グループでは、フロー依存、逆依存、出力依存をそれぞれその意味のとおり「定義-引用」、「引用-定義」、「定義-定義」と呼んでおり、以下でもこちらの呼称を用いる。文献[16]では、表3.6のループ運搬依存をさらにループ2回(運搬)依存およびループ多回(運搬)依存に分類している。ループ2回運搬依存は、ループのある繰り返しにおいて参照された配列要素あるいは変数がそのループの次の繰り返しにおいて再度参照される場合をいう。すなわち、衝突時の2出現の制御変数の値がちょうどそのループのステップ値(増分値)だけ異なる場合である。それに対しループ多回運搬依存は、2回分以上の異なる回転において参照の衝突が生じることをいう。しかも、この表3.6の分類は、配列の場合には各要素ごとに、そして多重ループに関しては各ループごとに独立した関係として定義される。例えば次の例3では、制御変数Jのループについてはループ独立依存型でかつ、制御変数Iのループについてはループ2回運搬依存型の文S_2→文S_1の(定義-引用)型の依存が存在する。

[例3]多重ループ内のデータ依存(その1)

```
      DO  10  I=1,3
          DO  10    J=1,3
              ·····     =····A(J,I)··      (S1)
              A(J,I+1)=··········          (S2)
   10     CONTINUE
```

ループ独立依存かループ運搬依存かを区別することは、ベクトル化を促進させる機能のひとつであるループ選択機能を適用する際に必要となる。ループ選択機能は、多重ループにおいて外側の何重ループかをスカラ実行すれば、その内側の残りのループ群ではベクトル化可能であると判定する機能である。例えば次の例4では、

[例4]多重ループ内のデータ依存(その2)

```
      DO  10  I=1,3
          DO  10    J=1,3
              B(J,I)=····A(J,I)··      (S1)
              A(J,I+1)=····B(J,I)··    (S2)
   10     CONTINUE
```

配列Aについては、先に例3として述べた文S_2→文S_1の依存が存在する。配列Bに

ついては、制御変数Iのループおよび制御変数Jのループともにループ独立依存型の文S_1→文S_2の依存が存在する。これらの二つの依存により、文S_1および文S_2のどちらの文を先に実行してもベクトル化できない不適な参照関係が生じる。すなわち、この場合には文の入れ換えだけではベクトル化不能である。そこで、制御変数Iのループをスカラ実行したとする。例5に等価なプログラムを示す。この例5からわか

[例5]例4と等価なプログラム

```
      DO  11    J=1,3
          B(J,1)=····A(J,1)··     (S1-I1)
          A(J,2)=····B(J,1)··     (S2-I1)
11    CONTINUE
      DO  12    J=1,3
          B(J,2)=····A(J,2)··     (S1-I2)
          A(J,3)=····B(J,2)··     (S2-I2)
12    CONTINUE
      DO  13    J=1,3
          B(J,3)=····A(J,3)··     (S1-I3)
          A(J,4)=····B(J,3)··     (S2-I3)
13    CONTINUE
```

るとおり、配列Aに関するループの回転に伴う、すなわち次の回転に運搬されるデータの流れは自然に保存されている。そして、制御変数Jの各ループ内ではもはや、配列Aについてデータ依存はない。従って、制御変数Jの各ループについては、制御変数Jでこのままベクトル化できると判定できる。例4に戻っていえば、制御変数Iのループをスカラ実行すれば、制御変数Jのループでベクトル化できることがわかる。この例からわかるとおり、一般にループ運搬依存は、それを引き起こすループをスカラ実行すれば、自然に保存されるので無視してよい。ループ選択機能はループ運搬依存のこの性質を利用するので、ループ独立依存かループ運搬依存かを区別できることが重要となる。なお、ループ独立依存については、例5の配列Bからわかるとおり、制御変数Iのループをスカラ実行しても無視することはできない。同様に、例4で制御変数Iのループと制御変数Jのループとを入れ替えた上で、制御変数Jのループをスカラ実行したとすると、配列Aの依存は制御変数Jについてはループ独立型であるため、配列Aの依存および配列Bの依存ともに無視できず、ベクトル化できなくなる。

3.3.1 ベクトルデータ

第2章で述べた国産のベクトル計算機の有するパイプライン演算器はいずれも、固定小数点演算の場合32ビット、論理演算および浮動小数点演算の場合64ビットのデータ長を主として扱う。FORTRANの型で言えば、4バイトの論理型、4バイトの整数型、4バイトおよび8バイトの実数型、8バイトおよび16バイトの複素数型をベクトルデータとして処理できる。ただし、4バイトの実数型、8バイトの複素数型はプロセッサ内部では64ビットのデータ長として処理される。そして、その他の文字型、1バイトの論理型、2バイトの整数型、4倍精度の実数型および複素数型は、ベクトル化の対象外となる。しかし、日立のFORTRAN77/HAP(23-00)コンパイラをはじめとし、1バイトの論理型、2バイトの整数型等は、使用法に制約があるが徐々にベクトル化対象とされつつあり、ベクトル化対象となる型は今後さらに増加するものと予想される。

ベクトルデータの形式は主記憶上での配置とアクセスの仕方により次の3タイプに分類される。

(A) 連続ベクトル:参照される全データが主記憶上で隣接し、連続してアクセスされる。

[例]A(I),(I=1,2,3,··,n)

(B) 等間隔ベクトル:参照される全データが一定間隔で並び、一定方向に順にアクセスされる。

[例]A(2*I+1),(I=1,2,3,··,n)

A(10−2*I),(I=1,2,3,··,n)

(C) 間接参照ベクトル(リストベクトル、間接指標ベクトル):順にアクセスされる各データの位置が任意の場合。

[例]A(INDX(I)),(I=1,2,3,··,n;INDXはAの添字範囲内の値をとる)

現在の国産のベクトル計算機はすべて、メモリ-メモリ型でなくレジスタ-レジスタ型のベクトル演算命令を有する。さらに、ベクトルデータは必ず主記憶からベクトルレジスタ上に1度ロードされ、演算終了後結果がベクトルレジスタから主記憶へストアされる。従って、演算パイプラインが高速にデータを処理するようになると、今度はこのロード/ストア用のパイプラインの性能が重要となる。その処理速度は一

般に先の(A)、(B)、(C)の順に遅くなる。これは、主記憶アクセスがネックとなることを回避するために、各社とも主記憶を複数のバンクに分け、インターリーブしているが、(A)に比べ(B)、(C)でバンクの衝突(バンク競合あるいはバンクコンフリクト)が生じやすくインターリーブ機構がうまく機能しなくなること等、ハードウェアからの制約が原因となっているようである。なお、同じ連続アクセスでも逆方向、例えば$A(n-I+1),(I=1,2,3,\cdots,n)$のような場合は、(B)と同じように処理され、順方向の場合より遅くなる傾向にある。

ベクトルデータとスカラデータの演算を行うベクトル命令([例]$A_i \leftarrow S \times B_i$;以下Sはスカラデータ、単純変数を示す)も存在する。単純変数は、自動ベクトル化の際引用のみの場合を除き、**配列化**(scalar expansionあるいは単にexpansion)される。これを、ソースレベルで図3.5に表現した。同図中の(*)印の文は最終値保証と呼ばれ、単純変数Tがそれ以後のソーステキストで引用されている場合に、スカラ実行時のループ終了時のTの値を持たせ、意味を変えないために必要となる。同図(b)では、Tがスカラ実行時に先に引用され、TAに対する初期値設定の文も必要となる。

```
      DO 10  I=1,n
        T = ·····        (S1)
        ·· = ·· T ··     (S2)
10    CONTINUE
```

↓ ベクトル化

```
      DO 10  I=1,n
        TA(I)= ·····     (S1)
10    CONTINUE
      DO 11  I=1,n
        ····· = ·· TA(I) ··  (S2)
11    CONTINUE
      T = TA(n)          (*)
```

(a) 最終値保証(*)が必要な配列化

```
      DO 10  I=1,n
        ·· = ·· T ··     (S1)
        T = ·····        (S2)
10    CONTINUE
```

↓ ベクトル化

```
      TA(0) = T          (†)
      DO 10  I=1,n
        TA(I)= ·····     (S2)
10    CONTINUE
      DO 11  I=1,n
        ····· = ·· TA(I-1) ··  (S1)
11    CONTINUE
      T = TA(n)          (*)
```

(b) 初期値設定(†)も必要な配列化

(注)ともに、変数TAは変数Tを配列化した一時的な作業領域

図3.5 単純変数の配列化

3.3.2 自動ベクトル化の対象となる演算

第2章で述べた各種演算パイプラインにより、加減乗除、べき乗計算、各種論理演算、比較、型変換等ほぼすべての処理のベクトル実行が可能である。ただし、整数除算は倍精度の浮動小数点除算に変換される[10]。さらに、sin、cos等各種組み込み関数、(ソーステキストにインライン展開される)ユーザ定義の文関数および内部関数、ならびに数値計算ライブラリもベクトル化されつつある。

次のような、(ベクトル)**マクロ演算**と呼ばれる特殊な処理のベクトル化も可能になってきている。これらのマクロ演算は本来データ参照関係不適な、回帰的な演算でベクトル実行になじまない。それを特別なベクトル命令を用意することでベクトル化している。なお、必ずしもすべてのベクトル計算機に以下の全命令が用意されているわけではない。

(1) 内積型演算($S = S \pm A_i \times B_i$)

(2) 総和型演算($S = S \pm A_i$)

(3) 累積型演算($S = S \times A_i$)

(4) 最大、最小値とその添字番号の探索

(5) 漸化式、1次巡回型演算(あるいは回帰型演算)

$$X_i = A_i \pm X_{i-1} \times B_i$$

$$X_i = (A_i \pm X_{i-1}) \times B_i$$

なお、(1)から(3)はパイプラインの有効利用のためベクトルデータに対する演算順序が変更される。(5)については、その演算順序の変更が不可能なので、処理速度はあまり向上しない。

上記のマクロ演算のベクトル化の場合には、単なるベクトル化と異なり、ある一定の条件を満たす演算のパターンを探索することが必要となる。例えば、(4)の場合次の例のようなソーステキストのパターンから抽出される。従って、他の変数、文等がループ内に混在し、他の処理をまとめて行うようなソーステキストからの抽出は一般に困難で、現在のメーカ提供のコンパイラには判定できない。従って、このようなマクロ演算が自動ベクトル化されるためには、種々の制約が付けられ、結局人間が単純な形のソーステキストにするよう指示されていることが多い。V-Pascal Version 1.5では、先述の依存関係をまとめたD行列から的確にマクロ演算を抽出できる。これについては、別論文としてまとめる予定である。

[例]最大値探索を行うソースプログラム

```
      VMAX = 0.0
      DO  10  I = 1,N
          IF  (VMAX .LT. PM2(I))   THEN
              VMAX = PM2(I)
              MAXIND = I
          ENDIF
10    CONTINUE
```

3.3.3 自動ベクトル化の対象ループ

3.3.1項、3.3.2項において、どのような演算およびデータが自動ベクトル化の対象となるかについて述べた。これらの演算を含む文として自動ベクトル化の対象となり得るものは、代入文、各種IF文である。制御の流れを変えるGO TO文、各種IF文、ループを形成するDO文等については、次のような制約がある。

現在の自動ベクトル化の対象となるループの条件。

(1) DOループであること。GO TO文、各種IF文で形成されるループは除外される。

(2) ループ内にループ外への飛び出しを含まないこと(3.3.4項で詳述する)。

(3) ループ内に逆方向分岐、すなわちGO TO文による逆戻りのジャンプを含まないこと。

(4) 一般にはDOループが多重にネストした場合、最内側ループのみが自動ベクトル化の対象となること(3.3.5項で詳述する)。

つまり、自動ベクトル化の対象となるループは、最内側DOループで、その中の制御の流れが、ループを形成せず、3.3.4項で詳述する単純な場合を除き飛び出しを含まないif-then-else型のような選択構造である場合に限られると考えてよい。この種のIF文のベクトル化手法については、3.3.6項で説明する。

もちろん、3.3.2項で述べた関数、それに入出力文等を含めた各種の手続き呼び出しも、出現してよい。これらは、部分ベクトル化機能により、3.3.2項で述べたようなベクトル化可能なものだけが、自動的に抽出される。FORTRAN77/SXコンパイラでは、外部関数の呼び出しも、プログラマが指示すれば、特殊な実行時の中継ルーチンにより疑似的にベクトル化される[22]。

3.3.4 飛び出しを含むDOループのベクトル化

飛び出しを含むDOループは、一般にはベクトル化できないが、現在次の条件を満たす場合に限り、そのDOループ全体がベクトル化される。これは、3.3.6項で述べるIF文のベクトル化手法と同様に、実行時に飛び出し条件をマスクに変換し、そのマスクを使いベクトル実行すると思われる。

[飛び出しの条件][21][22]

(1) 飛び出し点がただ1ヵ所だけであること。

(2) 飛び出しの条件の評価が、他のIF文等により制御を受けないこと。

(3) 飛び出し点以前で、配列要素へ値の定義がないこと。

(4) 飛び出しの条件がループの繰り返しにより変化すること。

```
ⓥ    DO  10  I = 1,N  {ベクトル化}
          X = A(I)*B(I)
          IF  (X.EQ.0.0)  GO  TO  99
          AV(I) = SQRT(ABS(X))
10    CONTINUE
99    CONTINUE
```

(以下ⓥはDOループがベクトル化された印)

図3.6 飛び出しを含むループのベクトル化の例

上記のうち(3)は、単純変数への定義は許されることを意味している。すなわち、図3.6のような飛び出しの条件に係わる、飛び出し点以前での定義は、実行時に飛び出し条件をマスクに変換する際に、本来実行されないかもしれない繰り返しまでも含めて、全繰り返しについて値の定義を行う必要がある。従って、もし配列への定義であると、変えてはならない要素までも、定義してしまうためベクトル化できない。しかし、もし単純変数への定義ならば、図3.5のように配列化されるので、飛び出した時点での最終値保証により意味が変わることはないからである。

このように、飛び出しを含むDOループも徐々に自動ベクトル化の方向には進んでいるが、部分ベクトル化もできないのが現状である。むしろ、自動ベクトル化できるのは、先のマクロ演算と同様、ある種のパターンだけであると言える。

3.3.5 多重DOループの自動ベクトル化

現在では、多重DOループの自動ベクトル化も進んでいる。

まず、第1に最内側のDOループにのみ各種の実行文が存在する、いわゆるタイトな多重DOループの場合にベクトル化に最適なDOループの制御変数が選択される。図3.7の場合、3.3.1項で述べたとおり連続ベクトルが最も高速であり、ベクトル長もより長くとれることから、最内側ではない制御変数Iでベクトル化される。また、図3.8の場合、最内側の制御変数Jではデータ参照関係からベクトル化できないので、制御変数Iでベクトル化されている。

```
Ⓥ    DO  10 I = 1,24
         DO 10   J = 1,24
            DO 10   K = 1,9
              QS(I,J) = QS(I,J) + QL(K,I)*QL(K,J)*S
10    CONTINUE
```

図3.7 ベクトル化に最適なDOループの選択の例1

```
Ⓥ    DO  10 I = 1,N
         DO 10   J = 2,N
            Z(I,J) = P(I,J-1)*S(I,J)
            P(I,J) = Z(I,J)/QL(J)
10    CONTINUE
```

図3.8 ベクトル化に最適なDOループの選択の例2

ただし、この選択が行えるのにも、ループ内に飛び出し、ベクトル化対象外の要素、マクロ演算を含まないという制約が付く。従って、DOループの選択では、部分ベクトル化、マクロ演算および飛び出しの処理を全く行わないことになる。

第2に、図3.9、図3.10のようにタイトな多重DOループで各配列を連続した複数次元全域に渡って、次元の高い方から順に全点総なめ形参照する時に限り一重化し、その全実行をベクトル化する。

この多重ループの一重化にも、先の最適なDOループの選択と同じ制約条件のほかに、DOループ内に一重化の対象となり得る制御変数が、配列の添字式以外に現れないこと、DOループ実行後のそれらの値を引用しないことという条件がさらに付加される。これは、一重化した制御変数に関しては、DOループ実行により、全


```
      DIMENSION  U(N,N),X(N,N)
Ⓥ     DO  10 I = 1,N
Ⓥ        DO 10   J = 1,N
            U(I,J) = X(I,J)*S
            IF(U(I,J).LT.0.0) THEN
              U(I,J) = 0.0
            ENDIF
10    CONTINUE
```

図3.9 多重ループの一重化ベクトル実行の例1

```
      DIMENSION  W(10,10,10)
      DIMENSION  E(10,10)
Ⓥ     DO  10 I = 1,10
         DO 10   J = 1,5
Ⓥ           DO 10   K = 1,10
              E(I,K) = W(J,I,K)*S
              W(J,I,K) = E(I,K) + 1.0
10    CONTINUE
```

図3.10 多重ループの一重化ベクトル実行の例2

く値が定義されないか、誤った定義がなされることを意味する。

富士通FORTRAN77/VP(V10L20)、日立FORTRAN77/HAP(23-00)両コンパイラでは、タイトでない多重DOループの自動ベクトル化、すなわち最内側以外のネストレベルに存在する実行文の自動ベクトル化も行われつつある。しかし、この場合にも、上記と同様の制約(詳細略)がある上に、より浅いネストレベルに存在する実行文が自動ベクトル化されるのは、その着目した部分と外の部分とのデータ参照関係が適の場合に限られる。すなわち、3.2節で述べたような大幅な演算順序の変更により、データ参照関係不適を解消する所までは到達していないのが現状である。

3.3.6 IF文のベクトル化手法

if-then-else型の制御構造のベクトル化の基本的な考え方は、各文ともすべての繰り返しについて実行するが、その際にマスクをかけることにより、選択的な実行を疑似的に行うものである。その具体的な実現方式は、現在次の3方式である。

(1)マスク機能使用方式

(2) データ編集(収集-拡散、収集-分散、圧縮-伸長等各社で呼び方が異なる)機能使用方式

(3) 間接参照(間接指標、リストベクトル、収集-拡散等と呼ばれる)型命令使用方式

```
ⓥ      DO  10  I = 1,5
            IF  (A(I).GT.0.0)  THEN
              B(I) = C(I)
            ENDIF
10      CONTINUE
```

図3.11 IF文を含むループのベクトル化の例

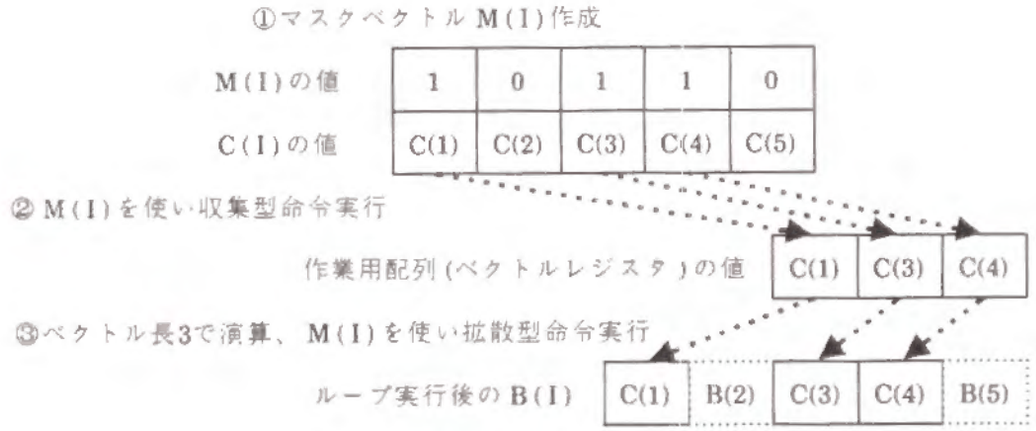
(1)から(3)の方式を図3.11の例に適用し、図3.12に比較した。IF文の条件節からマスクベクトルM(I)作成までは、どの方式も共通である。

図3.12からわかるとおり、(b)、(c)の2種は類似している。これらと(a)との相違点は、実行されるベクトル長である。すなわち、(a)ではTHEN節も(もしあればELSE節も同様に)全繰り返しについて(従ってこの例の場合ベクトル長5で)ロード/ストアおよび各種演算が実行される。これに対し、(b)、(c)ではTHEN節、ELSE節はそれぞれ各々の条件が満たされた要素についてのみ処理が行われる。すなわち、この例の場合のTHEN節はベクトル長3で実行される。

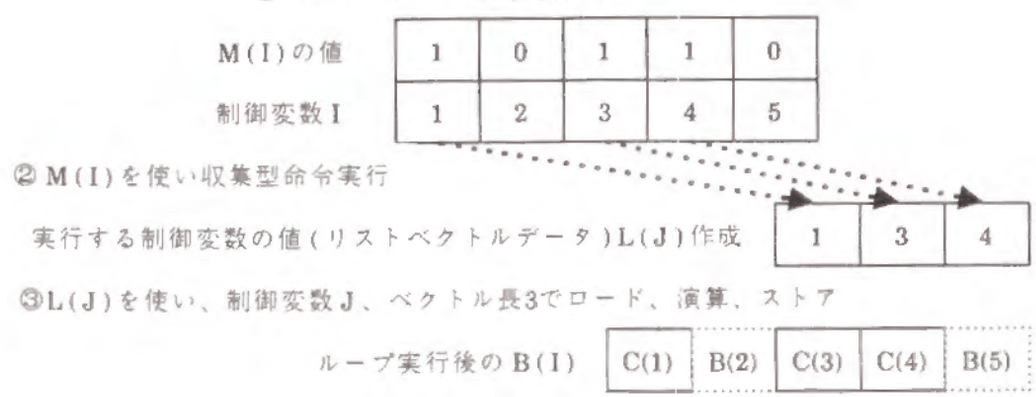
(b)と(c)のとの相違点は、データ編集機能と間接参照型命令とのロード/ストア処理の違いにある。詳述すると、(b)のデータ編集機能ではロード/ストアは全繰り返しについて(すなわちこの例の場合ベクトル長5で)参照される全要素に対して試みられ、マスクが1の要素についてのみ実際にベクトルレジスタと主記憶との転送が生じる。一方、(c)の間接参照型命令では、それに必要なリストベクトルデータを生成する際には(この例の場合ベクトル長5で)制御変数Iの配列化データをすべて走査する。しかし、その後はロード/ストアも含めて、全処理が選択された要素についてのみ行われる。なお、(b)においてもロード/ストア以外の各種演算は、選択された要素についてのみ行われる。

従って、各方式の速度向上については、選択的に実行されるTHEN節、ELSE節それぞれのベクトル長が十分長いこと、従ってもとの全繰り返しのベクトル長も十分長いことが必要であると言える。理論的にはマスク生成はIF-THEN-ELSE型の制御構造が何重にネストしていても可能であるので、各方式ともネストレベルの制限はない。ただし、FORTRAN77/SXコンパイラ(Rev.24)ではネストが3重を越える

(a)マスク機能によるベクトル実行



(b)データ編集機能によるベクトル実行



(c)間接参照によるベクトル実行

図3.12 IF文のベクトル化手法

と、たとえベクトル化できたとしても各選択実行されるベクトル長が短くなっていると予想されるため、自動的にスカラ実行するようである[22]。(a)の場合には先述したように、すべての処理においてマスク演算を行うので、全繰り返し回数に比べて選択実行される繰り返し回数が少ない(選択条件を満たす割合が小さい)と処理効率が劣化するので、方式(b)、(c)を採用することが望ましい。逆に、選択条件を満たす割合が大きいと(b)、(c)のデータ編集用命令や間接参照型命令のオーバーヘッドが目立つことになる。(b)と(c)の処理速度ではデータ編集用命令と間接参照型命令のハードウェア性能が決定的要因となることが予想される。

これら3方式の選択は、コンパイラが自動的に行う。ただし、実行効率の最良の方式が正しく選択されるためには、上述のIF条件節の満足される割合(IF条件節の**真率**と呼ばれる)、ベクトル長に関係する変数群の実行時の値等をプログラマが把握していて、ソーステキスト中のコンパイラへの指示行でそれらの情報を与える必要がある。

データ編集機能を使って、図3.13、図3.14のようなDOループもベクトル化される。これらのベクトル化も、マクロ演算と同じくある種のパターンを抽出しなければ

```
      J = 0
Ⓥ     DO  10  I = 1 , N
          IF  ( A ( I ) .GE. 0.0 )  THEN
              J = J + 1
              B ( J ) = A ( I )
          ENDIF
10    CONTINUE
```

図3.13 収集型のループのベクトル化の例

```
      J = 0
Ⓥ     DO  10  I = 1 , N
          IF  ( A ( I ) .GE. 0.0 )  THEN
              J = J + 1
              A ( I ) = B ( J )
          ENDIF
10    CONTINUE
```

図3.14 拡散型のループのベクトル化の例

ばならない。特に、上例の配列参照の添字式中に現れる単純変数Jのように、補助的な制御変数の役割を果たし、ループの繰り返しにより値が変化していく変数(**帰納変数**、**インダクション変数**: induction variable)を見出すことが必須となる。

FORTRAN77/SXコンパイラでは、ループの繰り返しにより条件節が変化しない

```
          DO  10  I = 1 , N
              IF  ( J .GE. 0.0 )  THEN
                  A ( I ) = A ( I ) + 1
              ENDIF
10        CONTINUE
              (a)ベクトル化前

          IF  ( J .GE. 0.0 )  THEN
Ⓥ             DO 10     I = 1 , N
                  A ( I ) = A ( I ) + 1
10            CONTINUE
          ENDIF
              (b)ベクトル化後
```

図3.15 ループ不変のIF文のベクトル化の例

```
          DO  10  I = 1 , N
              IF  ( I .EQ. 1 ) THEN
                  A ( 1 ) = 0.0
              ELSE
                  A ( I ) = A ( I - 1 ) + 1
              ENDIF
10        CONTINUE
              (a)ベクトル化前

          A ( 1 ) = 0.0
Ⓥ         DO  10  I = 2 , N
                  A ( I ) = A ( I - 1 ) + 1
10        CONTINUE
              (b)ベクトル化後
```

図3.16 ループ展開によるベクトル化の例

IF文を含む場合、IF文をループ外に移動させベクトル化する[22]。その様子をソースレベルで図3.15に示す。また、DOループ内のIF文が制御変数そのものを条件とし、繰り返しの最初あるいは最後のみを例外的に処理している場合には、DOループを展開して、それらの例外処理を除いた繰り返しのみをベクトル化する。その様子をソースレベルで図3.16に示す。もちろん、この2種の変形によりIF文を含まないDOループが得られ、変形前の状態をそのままベクトル化するよりも、マスク生成等のオーバーヘッドが無く、当然効率が改善されると思われる。なお、これらの変形が行われる条件についての詳細は、不明である。

3.4 結 語

3.3節で述べたとおり、自動ベクトル化技術は進歩しつつあるが、未だベクトル化できない部分も多い。その中でも多重ループ全体を対象としたベクトル化技術が、脆弱であると言わざるを得ない。なぜなら、多重ループは実行時のコストが非常に高いにもかかわらず、世界最高水準にあるといわれる日本の各メーカの自動ベクトル化コンパイラをもってしても、制約が強すぎるため十分なベクトル化が行えないからである。例えば、3.3.5項で述べたとおり、富士通のFORTRAN77/VP (V10L20) コンパイラの場合にはデータ参照関係が適でないとループ分割できない。日立のFORTRAN77/HAP (23-00) コンパイラの場合、我々の研究グループが開発した一重化機能も取り込んでいる[23]が、3重ループまでの多重ループしかベクトル化の対象としていない。これらの制限は、3.2節で述べた各種依存解析能力が多重ループに対して十分ではないことが主な原因であると考えられる。

それに対し、第4章で述べるとおり、V-Pascalコンパイラは1985年の設計開始時点から、この多重ループのベクトル化に関する制限を取り除くことが第1の目標であった。そのため、①ループの深さならびに入れ子構造になんらの制約も持たない強力な、しかもデータ参照の衝突の有無を完全に厳密に判定する依存解析能力を有している。そして、②ループの深さならびに入れ子構造になんらの制約も持たないループ分割機能およびループの入れ子構造を越える演算レベルでの実行順序の変更機能を含む強力な部分ベクトル化機能を有している。さらに、③先述のとおり日立製コンパイラにも採用された、長大なベクトル長でベクトル実行させる一重化機能も有

している。これらの機能により、V-Pascalコンパイラは多重ループのベクトル化に関し、一つのブレーク・スルーの役割を果たした。

第4章
V-Pascal コンパイラの構成

4.1 緒 言

自動ベクトル化 Pascal コンパイラ V-Pascal Version 1 の開発の主目的は次の2点である。

A) ベクトル計算機を現状よりさらにユーザに使いやすいものとすること。

B) 高機能な自動ベクトル化を可能とすること。

そして、第1章で既述したが、その大きな特徴は以下のとおりである。

1) 言語 Pascal をソース言語とすること。

2) 多重ループ全体をベクトル化対象とし強力にベクトル化を行えること。

3) 種々のベクトル計算機をターゲットマシンとして自動ベクトル化ができることを考慮し設計されていること。

第1点については、第1章で触れた。第3点については、次節以降で述べる。

最も重要な第2点目の特徴について具体的に述べる。

2-1) ベクトル化可能性の判定に必要となる各種の依存関係解析機能が、多重ループ全体を対象とし、かつ厳密に解析できること。

2-2) 同解析機能により判定した依存関係から細かな演算レベルでのベクトル化可能部分を抽出する強力な部分ベクトル化機能を有すること。

2-3) ベクトル化可能な多重ループ内の処理を、等価な1重ループ内での処理として長大なベクトル長でベクトル実行させる目的コードに翻訳する機能(一重化機能)を有すること。

従来のメーカ提供の自動ベクトル化コンパイラは、3.3節で解説したとおり多くの場合多重ループ全体でなく最内側ループのベクトル化にしか対応できていなかった。そのため見過ごされてきた最内側ループ以外のループ中に存在する部分については、これらの機能を合わせ持つことにより、初めてベクトル化可能となった。しかも、一重化機能により長大なベクトル長でベクトル実行されるので、最内側ループだけのベクトル長でベクトル実行される場合より、さらに有効にパイプライン演

算器を使用できる。つまり、これらのV-Pascalコンパイラの自動ベクトル化機能は、従来にない独自の強力なものといえる。この一重化機能は、まずプリプロセッサ内にメーカに先駆けて実現され、その有効性が検証された[18],[24],[25],[26]。その経験を基に、本格的なコンパイラV-Pascalの開発が開始された。

第4章では、自動ベクトル化 Pascal コンパイラ V-Pascal Version 1[27]のフェーズ構成、モジュール構成、処理手順、内部表現の概要を述べるとともに、簡単な例を挙げ処理の経過により各内部表現がどのように更新されるのかを解説する。V-Pascalコンパイラの特徴である機能については章を改め詳述する。Version 2[28]の制作が現在進行中であり、while型ループのベクトル化等、より強力なベクトル化機能が実現されつつあるが、本論文では Version 1 の機能についてのみ述べる。

4.2 フェーズ構成ならびにモジュール構成

V-Pascal Version 1 は以下の5フェーズからなる。

1) フェーズ1:構文/意味解析フェーズ

2) フェーズ2:ベクトル化前処理フェーズ

3) フェーズ3:部分ベクトル化処理フェーズ

4) フェーズ4:ベクトル化後処理フェーズ

5) フェーズ5:目的コード生成フェーズ

これらの概略モジュール構成を図4.1に示す。図中には、モジュール間で受け渡される主要データについても記した。このうち、下線を施したデータは、その直前の処理で新たに付加されたことを示している。これらについては、4.3節で解説する。なお、データフロー解析部は、最適化処理開始直前に必ず1回呼び出され、かつ最適化処理中に必要に応じて複数回呼び出される子モジュールである。また、網目構造除去処理部では、制御の流れが変化した際にのみコントロールフロー解析部を再度呼び出す。図4.2から図4.4には、それぞれフェーズ2からフェーズ4のモジュール構成を拡大し、各モジュール間で受け渡されるデータの流れを中心にまとめた。本論文の主題であるベクトル化に関連するフェーズ3およびフェーズ5に含まれる処理モジュールについては、章を改め第5章、第6章、第7章で述べ、その他のモジュールについては、4.4節以降で簡単に解説するにとどめる。

図4.1から図4.4を基に V-Pascal Version 1 の処理の流れを概説する。

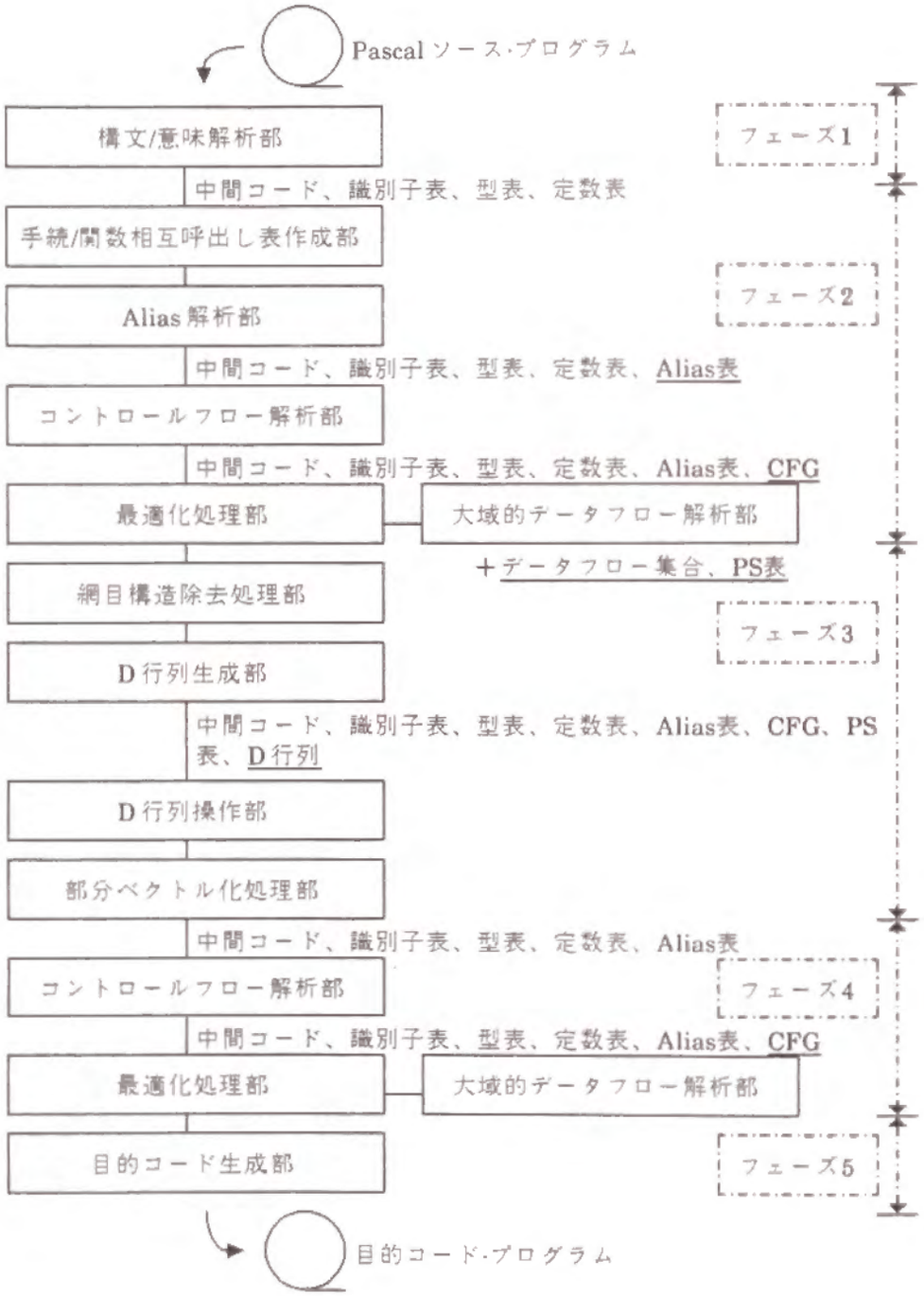


図4.1 V-Pascal のフェーズ構成/概略モジュール構成

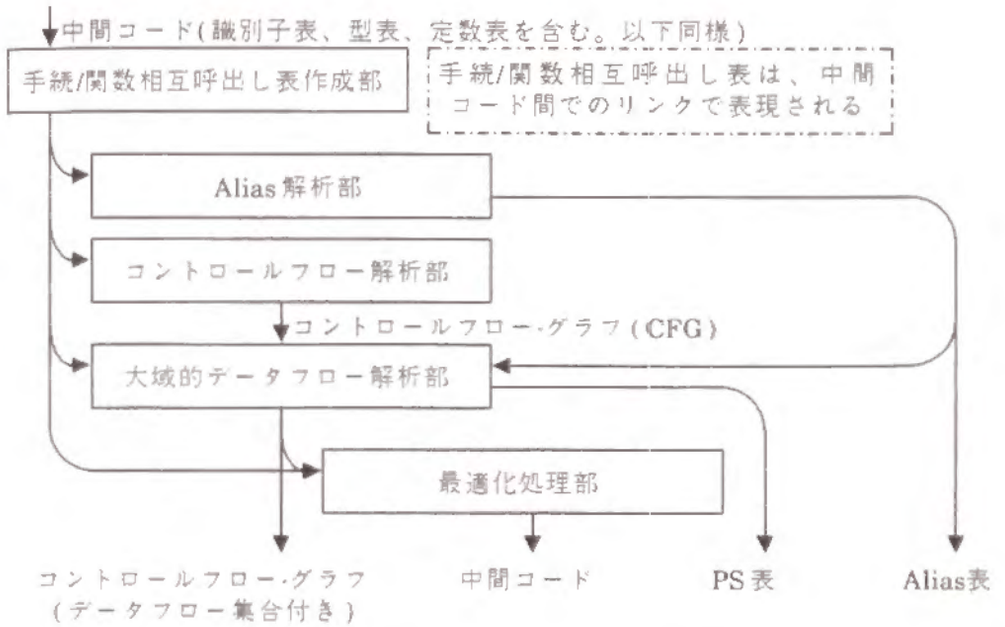


図4.2 データの流れを中心としたフェーズ2拡大図

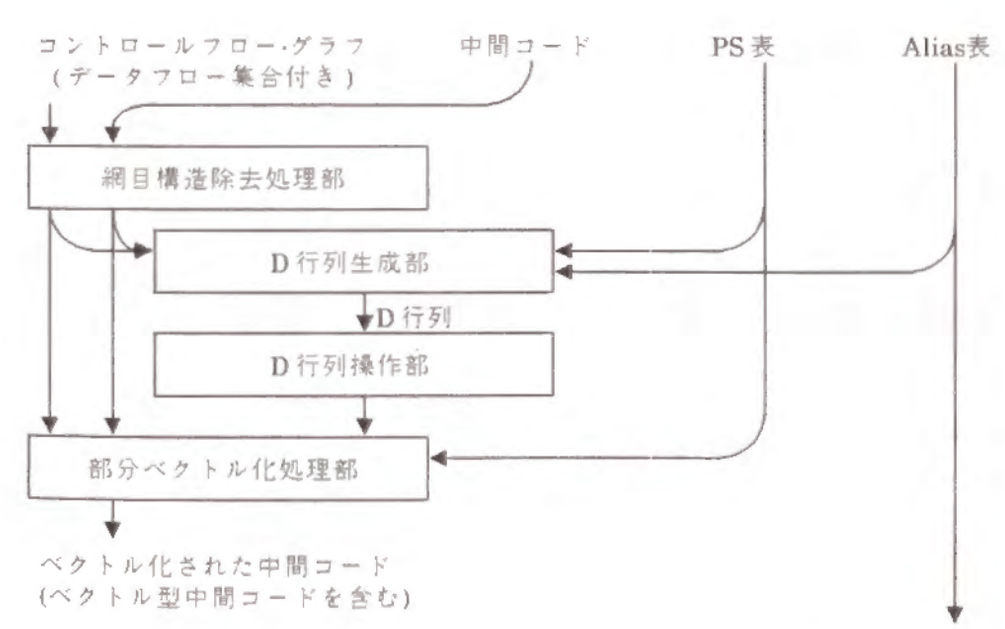


図4.3 データの流れを中心としたフェーズ3拡大図

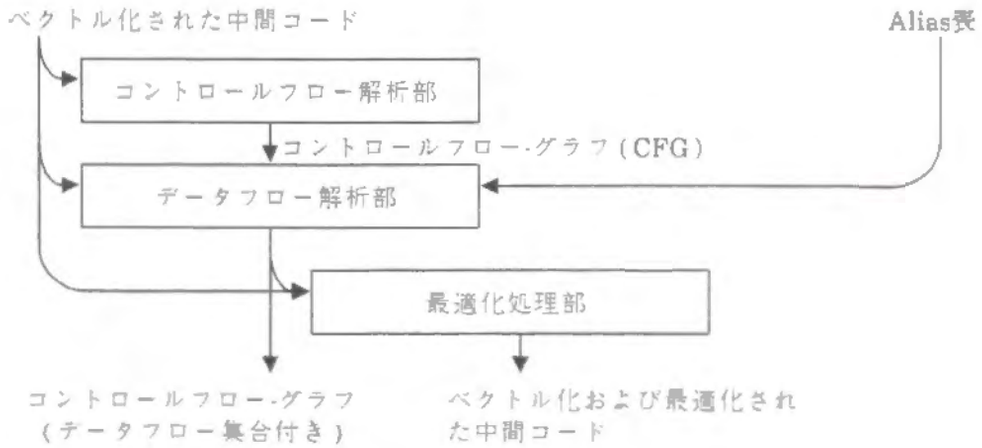


図4.4 データの流れを中心としたフェーズ4拡大図

(1) フェーズ1：構文/意味解析フェーズ

フェーズ1：構文/意味解析フェーズでの処理は、図4.1からわかるとおり、与えられたユーザ・プログラムを読み込み、構文/意味解析を行いながら、内部表現である中間コード（4.3.2項で詳しく述べる）表現に翻訳する。フェーズ2以後のすべての処理は、この中間コード（表現）を操作する。この翻訳と同時に、コンパイル・リストならびに、もしあれば構文/意味上のエラーに関するメッセージを出力する。この種のエラーが発見された場合には、フェーズ2以降の処理はまったく行われない。なお、中間コードは、Pascalの豊富なデータ構造に対するアクセスならびに豊富な制御構造（付録Pascal構文図参照）を記述できるよう設計されているため、Pascalに類似の手続型言語であれば、このフェーズ1を書き換えることにより容易にターゲット言語を変えることができる。

(2) フェーズ2：ベクトル化前処理フェーズ

フェーズ2（図4.2参照）では、手続/関数相互呼出し表作成部が全中間コードを走査し、手続/関数相互呼出し表を作成する。この表は次のAlias解析部（4.5節参照）で手続/関数呼出しの仮引数と実引数の結合関係により生じる**別名関係**（以下Alias）を求める処理に利用される。Aliasは大域的データフロー解析およびデータ依存解析において必要となる。この後の処理はすべてユーザ定義の手続/関数（メイン・プログラムを含む）を単位として行う。

コントロールフロー解析部（4.6節参照）は、中間コードを走査し制御の流れを抽出したコントロールフロー・グラフ（CFG：4.3.3項参照）を生成する。このコントロールフロー・グラフは、ユーザプログラム中に存在する全ループのネスト構造も含め、制御構造を容易に走査でき、次の大域的データフロー解析部を初め以後の処理で頻繁に参照される。コントロールフロー解析部では、抽出したコントロールフロー・グラフのループを検出（4.6.2項参照）するため、コントロールフロー・グラフの支配関係解析（4.6.1項参照）を行っている。

大域的データフロー解析部（4.7節参照）では、ユーザプログラム中で使用される変数（種々の構造の型を有するもの、単純変数、フィールド等すべてを含む）ならびにコンパイラが使用する一時変数（中間コードが定義する中間項を含む）のデータの流れ（参照関係）を抽出し、データフロー集合としてコントロールフロー・グラフ内に記録する。この情報を基に次の最適化、フェーズ3のデータ依存解析およびフェーズ5の目的コード生成部におけるレジスタ割り付けが行われる。同時に、この大域的データフロー解析部では、コントロールフロー・グラフの各頂点のプライマリ・セット（PS：4.3.4項参照）への分割も行う。プライマリ・セットはif文のベクトル化において重要な役割を果たす。プライマリ・セットへの分割については5.4節の制御依存関係解析とあわせて述べる。

最適化処理部では、ベクトル化を促進すると考えられる最適化（4.8節参照）を中間コードに施す。最適化処理はベクトル化とならんで重要である。

(3) フェーズ3：部分ベクトル化処理フェーズ

フェーズ3（図4.3参照）では、まず網目構造除去処理部がベクトル化対象多重ループ内のif文について、網目構造の探索を行い、もしあれば通常の**if-then-else構造**に変換する。これは、if文を含む多重ループのベクトル化を容易にする目的を持つ前処理である。詳細は6.2節でif文のベクトル化にからめて論じる。

D行列生成部（6.3節参照）は、ベクトル化対象多重ループについてデータ依存解析を行うデータ依存関係解析部（5.2節、5.3節参照）、ならびに制御依存解析を行う制御依存関係解析部（5.4節参照）からなる。すなわち、V-Pascalでは、ベクトル化に欠かすことができない依存関係（3.2節で既述）解析をここでまとめて行う。これらの解析結果である中間コード間の種々の依存関係はD行列（4.3.5項参照）にまとめられ、次のD行列操作部に引き渡される。このD行列は依存グラフ（3.2.4項参照）である。すべての依存関係を集約して記録するD行列は、V-Pascalのベクトル化処理の

中心的役割を果たす。

D行列操作部は、こうして得られた各種依存関係を保持できるベクトル実行順序を求めるため、D行列の行および列の並べ換え(順序交換)を行う。依存が閉路を形成し、どのように並べ換えてもベクトル化に不適な依存関係が存在する場合には、その閉路を形成する強連結部分を縮退させ切り離す(6.4節参照)。この一連の操作は、各種依存関係を保持する中間コードの実行順序の変更(3.2.2項で既述)を施すことに相当する。このD行列操作部では、この一連の操作により得られた新しい実行順序を示すD行列を基に、配列化、ベクトル化すべきループの選択等さらに高度なベクトル化技法(6.5節参照)を適用する。

次に部分ベクトル化処理部(6.6節参照)では、このD行列の示す順序に中間コードを並べ換え、同時に縮退せずベクトル化可能と判定された中間コードはベクトル命令に対応するベクトル型の中間コードに置換する。

(4) フェーズ4:ベクトル化後処理フェーズ

上記のフェーズ3のベクトル化処理の結果、与えられたユーザ・プログラムの制御の流れまで大きく変化する。そのため、フェーズ4(図4.4参照)では、コントロールフロー解析ならびに大域的データフロー解析をやり直した後、最適化処理を行う。フェーズ2の最適化処理と異なり、ここではベクトル化処理で新たに付加されたベクトル型の中間コードを中心に最適化する。

(5) フェーズ5:目的コード生成フェーズ

フェーズ5では、フェーズ3のベクトル化処理後のD行列およびフェーズ4の大域的データフロー解析結果を使い、HITAC S-820/80をターゲット・マシンとし、ベクトル・プロセサ/スカラ・プロセサ用にレジスタ割り付けを行い、目的コードを生成する(ファイルに書き出す)。中間コードができるだけ機械依存でないように設計されているため、このフェーズ5を書き換えることによりターゲット・マシンを変えることができる。すなわち、V-Pascalのフェーズ5は、移植性を考慮し設計されている。現在、この移植性を検証する目的で、NEC SXシリーズの目的コードに変換するバージョンの製作が進行中である。

4.3 主要表およびそのデータ構造

V-Pascal Version 1の実現方式を述べるため、まず図4.1中に示した各種の表等について、そのデータ構造を概説する。

4.3.1 識別子表、型表、定数表

V-Pascal Version 1のフェーズ1構文/意味解析部は、Pascal-P4処理系ならびにPascal 8000処理系を参考にして作成されている[29]~[35]。ただし、これらはいずれも1パス・コンパイラである。すなわち、Pascal-P4処理系は構文/意味解析を行いながらP4コードと呼ばれる仮想スタック計算機用の目的コードを直接生成する。Pascal 8000処理系は、構文/意味解析を行いながらIBM360/370系(その他にも各種のターゲット・マシンに対応する複数のバージョンが存在するようである)の機械語命令を直接生成する。これに対し、V-Pascal Version 1は複数パス構成のコンパイラであり、フェーズ1構文/意味解析部は、独自に設計された中間コードに翻訳する点で機能が異なる。しかし、構文/意味解析に関しては同様に行えるので、そのために必要となる識別子表、型表、定数表のデータ構造は、ほぼこれらのものを踏襲している。これらの表は、本来は構文/意味解析のためにのみ使用されているが、V-Pascal Version 1では後述の中間コード中の各フィールドから参照され、図4.1中に示したとおりフェーズ1以降の全フェーズに順次受け渡されていく。

(1) 識別子表

識別子表は、構文/意味解析時において識別子名で検索されることを考慮し、図4.5に示すような識別子名を保持するNAMEフィールドが辞書式順序をなす二分探索木(binary search tree)の論理構造を持つ。物理的には、表の1エントリは二分探索木の1ノード(節)で表わされ、各ノードのLLINK、RLINK両フィールドのポインタによるリンクで二分探索木を構成する。この二分探索木は、Pascalの意味規則の中のスコープ・ルールを解釈しやすくするため、ユーザ・プログラムのメイン・ルーチンならびにユーザ定義の手続/関数(ブロック)ごとに作成される。そして、構文/意味解析時にはこれらのブロックごとに作成された二分探索木の根へのポインタ群を、その手続/関数の宣言されたレベル(ブロックレベル)の値を使い適切に管理することにより、ディスプレイ(厳密にはディスプレイベクタ)[36]を実現している。このディスプ

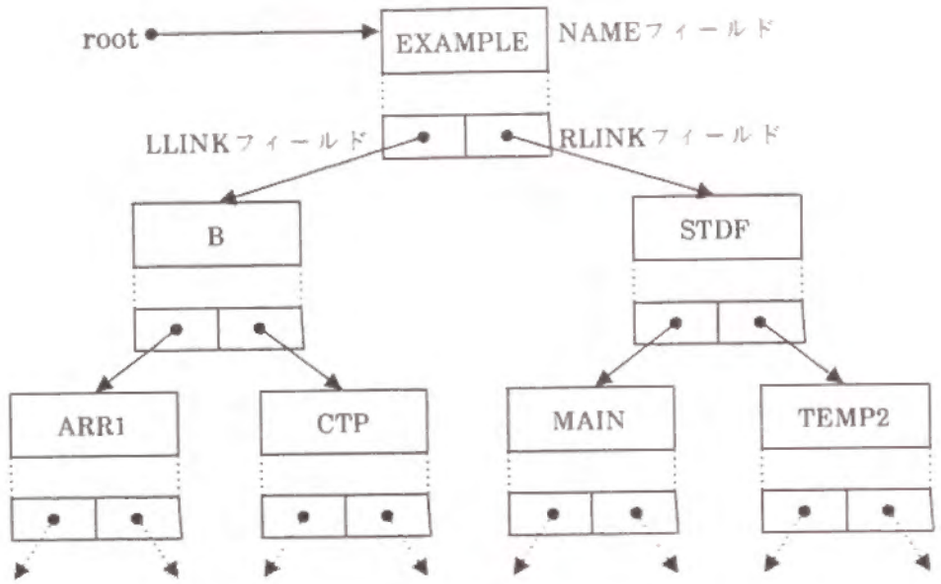


図4.5 二分探索木を形成する識別子表

レイのブロックレベル0のポインタは、Pascalの種々の標準名を保持する二分探索木の根を指し示す。ブロックごとに作成されるだけでなく、図4.6に示すようなPascalのwith文(付録 Pascal構文図のStatement参照)内に現れるフィールド名の検索も効率的に行えるよう、あるレコードのフィールド名(図4.6ではb、c、a)については、独立した二分探索木が構成され、当該レコードの型表内に保持される(後の図4.7参照)。このフィールド名のみの二分探索木は、図4.6に示すようなwith文を解析する間通常のブロックの二分探索木同様にディスプレイのもっとも深いレベルに

```
program example2;
    var
        rec1 : record
                 b : char;
                 c : real;
                 a : integer;
               end;
    begin
        rec1.a := 1;  {with文外でのフィールドへのアクセス}
        with rec1 do
            c := 0.1;  {with文内でのフィールドへのアクセス}
    end.
```

図4.6 レコード型とwith文の例

リンクされ、フィールド名のみの検索を可能とする。フェーズ1終了後は、中間コードの手続/関数宣言ノード中のDISPフィールドが、その手続/関数内で宣言された識別子を保持するため、該当する二分探索木の根を指し示す(後の4.3.2項、図411参照)。

識別子表の1エントリに相当する、二分探索木のノードはPascalの可変レコード(IDENTREC型)で定義され、種々の情報を蓄える。表4.1は、各種ノードに共通のフィールドとそれらが保持する情報についてまとめたものである。表4.2は、タグ-フィールド値により変化する可変フィールドについてまとめたものである。可変

表4.1 識別子表のノードの共通フィールド

フィールド名	意味	主要使用部分
NAME	識別子名(先頭8文字まで)	フェーズ1
LLINK	二分探索木の左の部分木を指す(図4.5参照)	フェーズ1
RLINK	二分探索木の右の部分木を指す(図4.5参照)	フェーズ1
IDTYPE	識別子の型を示す(型表の1エントリを指す)	全フェーズ
NEXT	列挙型の個々の名前、仮引数名のエントリ、フィールド名および変数名で同一の型と列挙して宣言されたもの(図4.7参照)等それぞれを宣言順につなぐ(その他一時的に使用[29])	フェーズ1
PFDCL	宣言された手続/関数に対応する中間コードの手続/関数宣言ノードを示す	全フェーズ
ALIASNUM	Alias集合の識別番号	データフロー解析部、データ依存解析部
DFLWK	データフロー集合における識別番号	同上,フェーズ5
CUREFREF	この識別子を参照する中間コードのリスト(の先頭を指す)	最適化処理部
LSTEFREF	制御の流れの各パスにおいてこの識別子を参照する最後の中間コードのリスト(の先頭を指す)	最適化処理部
KLASS	タグ-フィールド(表4.2参照)	全フェーズ

表4.2 識別子表のノードの可変フィールド

タグ·フィールドKLASSの値	サブフィールド		
	フィールド名	意味	主要参照部分
TYPES(型名)	−		−
KONST (定数名)	VALUES	定数の値(を保持する定数表の1エントリを指す)	フェーズ1
VARS (変数名)	VLEV	宣言されたブロック·レベル	フェーズ5
	VADDR	局所領域内相対番地	フェーズ5
	PARADDR	同上(番地呼びの仮引数の場合)	フェーズ5
	VKIND	直接アクセス/間接アクセス	フェーズ5
	ASCLIST	(間接アクセスの仮引数の場合)対応するAlias集合	Alias 集合生成部
	POINTED	(同上)ポインタで参照されるか否か	Alias 集合生成部
FIELD (フィールド名)	FLDADDR	レコード内相対番地	フェーズ5
PROC (手続名) ならびに FUNC (関数名)	PFDECKIND	Pascal標準手続/関数か、ユーザ定義手続/関数か	フェーズ1, フェーズ5
	KEY	標準手続/関数の識別番号	全フェーズ
	PFLEV	ユーザ定義手続/関数の宣言のレベル	フェーズ5
	PARAMS	識別子表の仮引数名のエントリのリスト(の先頭)	フェーズ1
	PRFNDCLNODE	該当する中間コードの手続/関数宣言ノードを示す	全フェーズ
	PFKIND	仮引数名としての宣言か、実際の宣言か	フェーズ1, フェーズ5
	PFCNT	ユーザ定義手続/関数の番地表のエントリ番号	フェーズ5
	LCSAVE	当該ユーザ定義手続/関数のための局所領域のサイズ	フェーズ5
	PFADDR	仮引数として宣言された場合、実引数の手続/関数の番地を記憶する領域の局所領域内相対番地	フェーズ5
	PASCLIST	当該ユーザ定義手続/関数のため生じるAlias集合	Alias 集合生成部

フィールドのうち、いくつかはさらに可変フィールドを含むものもあるが、その場合については当該フィールドが使用される場合を合わせて記述するにとどめる。識別子表、型表のフェーズ1関連のフィールドについては、文献[29]に詳しい。

(2)型表

型表は、識別子表と違いそれ自体が検索の対象となることはない。従って、概念的に「表」としてとらえるが物理的なデータ構造としては、識別子表のように各エントリが相互にリンクされ、まとまって表を形成しているわけではない。Pascalの種々の型を表せるよう設計された各エントリは、識別子表の物理構造の1ノードと同様に可変レコード(STRUCTREC型)を用いて宣言されており、識別子表および中間コード等の型を示すフィールドからポインタで指されているだけである。ただし、V-Pascalでは Pascalの標準的ないくつかの型については直接参照できるように、それらのエントリを指し示すポインタ群を便宜的に配列にまとめている。

表4.3、表4.4に型表の保持する情報をまとめる。表4.4では、表4.2と同様に下部構造としてさらに可変フィールドを含む場合には、そのフィールドが使用される場合を合わせて記述するにとどめる。表4.3、表4.4からわかるとおり、型表が保持する情報は Pascalの種々の構造がある型(付録 Pascal構文図の Type参照)をも表現でき、型のために必要な領域のバイト数等型の属性も含んでいる。構造がある型の例として図4.7に可変レコードの内部での表現を図示する。この図では型表の各フィールドの意味を中心に述べるため、表4.1、表4.2に挙げた識別子表のフィールドについては、型表との関連で必要となるもののみを記した。図4.8には多次元配列の例を示す。先述のとおり型表は、物理的には表を形成していないことが、これらの図からわかる。型表と識別子表とは相互に補完しあい、構造のある型を表現する。ただ

表4.3 型表のノードの共通フィールド

フィールド名	意味	主要使用部分
FTYPE	ファイル自身またはファイルを含むか否か	フェーズ1
SIZE	この型のために必要な領域のバイト数および境界調整の種別	フェーズ1, フェーズ5
IDPTR	この型の識別子(識別表の1エントリを指す)	データフロー解析部、データ依存解析部
FORM	タグ·フィールド(表4.4参照)、STRUCTFORM型	全フェーズ

し、このような複雑な関連を示すフィールドの中には、表4.3、表4.4に記したとおり、フェーズ1構文/意味解析部でのみ使用され、以後は使用されないものもある。すなわち、中間コード表現では必要な型の情報は、直接この型表の各エントリを指

表4.4 型表のノードの可変フィールド

タグ-フィールド FORMの値	サブフィールド		
	フィールド名	意味	主要使用部分
SCALAR 列挙型、標準型	SCALKIND	列挙型(BOOLEANを含む)か、標準型(INTEGER,REAL,CHAR)か	全フェーズ
	FCONST	列挙型の名前のリストの先頭	フェーズ1
PACKDTYPE 詰め込み型	BASETYPE	詰め込まれる元の型を示す	全フェーズ
SUBRANGE 部分範囲型	RANGETYPE	範囲を限定する元の型	全フェーズ
	MIN	範囲の最小値	全フェーズ
	MAX	範囲の最大値	全フェーズ
POINTER ポインタ型	ELTYPE	指し示される型	全フェーズ
POWER 集合型	PCKDSET	詰め込まれるか否か	フェーズ1
	ELSET	集合要素の型	フェーズ1,5
ARRAYS 配列型	AELTYPE	配列要素の型	全フェーズ
	INXTYPE	配列添字の型	全フェーズ
	AELLENG	配列1要素のサイズ(単位バイト)	フェーズ1,3,5
RECORDS レコード型	FIELDS	全フィールド名の二分探索木の根	フェーズ1
	FSTFLD	固定フィールドの先頭	フェーズ1
	RECVAR	可変レコードの場合タグ-フィールドを示す	フェーズ1
FILES ファイル型	TEXTFILE	通常のテキスト型か否か	フェーズ1,5
	FILTYPE	ファイル要素の型	フェーズ1,5
TAGFIELD タグ-フィールド	TGFLDP	タグ-フィールド名を指す	フェーズ1
	FSTVAR	可変フィールドのリストの先頭を指す	フェーズ1
VARIANT 可変フィールド	FSTVARFLD	当該フィールド名を指す	フェーズ1
	NXTVAR	次の可変フィールドを指す	フェーズ1
	SUBVAR	ネストした可変レコードの場合タグ-フィールドを示す	フェーズ1
	VARVAL	当該フィールドが存在するタグ値	フェーズ1

し示すポインタを用いて得ることができるため、型表内でのあるいは型表と識別子表間での関連の多くは、不要となる。しかし、不要情報をわざわざ除去する操作は効率を考慮し行っていない。

(3)定数表

定数表の1エントリも、整数型(INTEGER)、実数型(REAL)、集合型(SET)、文字列型(文字:CHAR型の詰め込み型1次元配列)それぞれの型の定数を記憶できるように、これらの型のフィールドを併せ持つ可変レコード(VALU型)として定義されている。なお、文字列型については4文字ごとに分割し、その4バイトのセルをリニア-リストの形態でリンクして記憶するため、そのリストの先頭を指示するポインタがフィールドに保持される。この4文字ごとの分割の際、空きが生じる場合には空白を右に詰める。このように、場合によっては文字列の長さが4バイト単位に切上げられてしまう。そこで、本来の長さの情報を保持するため、CHAR型の詰め込み型1次元配列の型を型表に登録し、そのエントリへのポインタをも別フィールドに保持している。すなわち、文字列型定数の場合には2フィールドが用意されている(CSTTAILREC型)。

定数表の物理構造は、上記のエントリをユーザ定義の手続/関数(メイン-プログラムもその一つ)ごとにリンクしてまとめたリニア-リストの形態を採用している。これは、定数表も識別子表と異なり、あるフィールドの値で検索されることがないためである。ただし、同一の手続/関数(メイン-プログラム)のブロック内で同一の型

レコード型の宣言の例

```
type
    r = record
            i,il : integer;
            case b:integer of
                1,2:(j:integer);
                3:(k:integer;
                    case c:boolean of
                        true:(a:real);
                        false:(j2:integer);
                    )
        end;
```

図4.7 レコード型の内部表現(つづく)

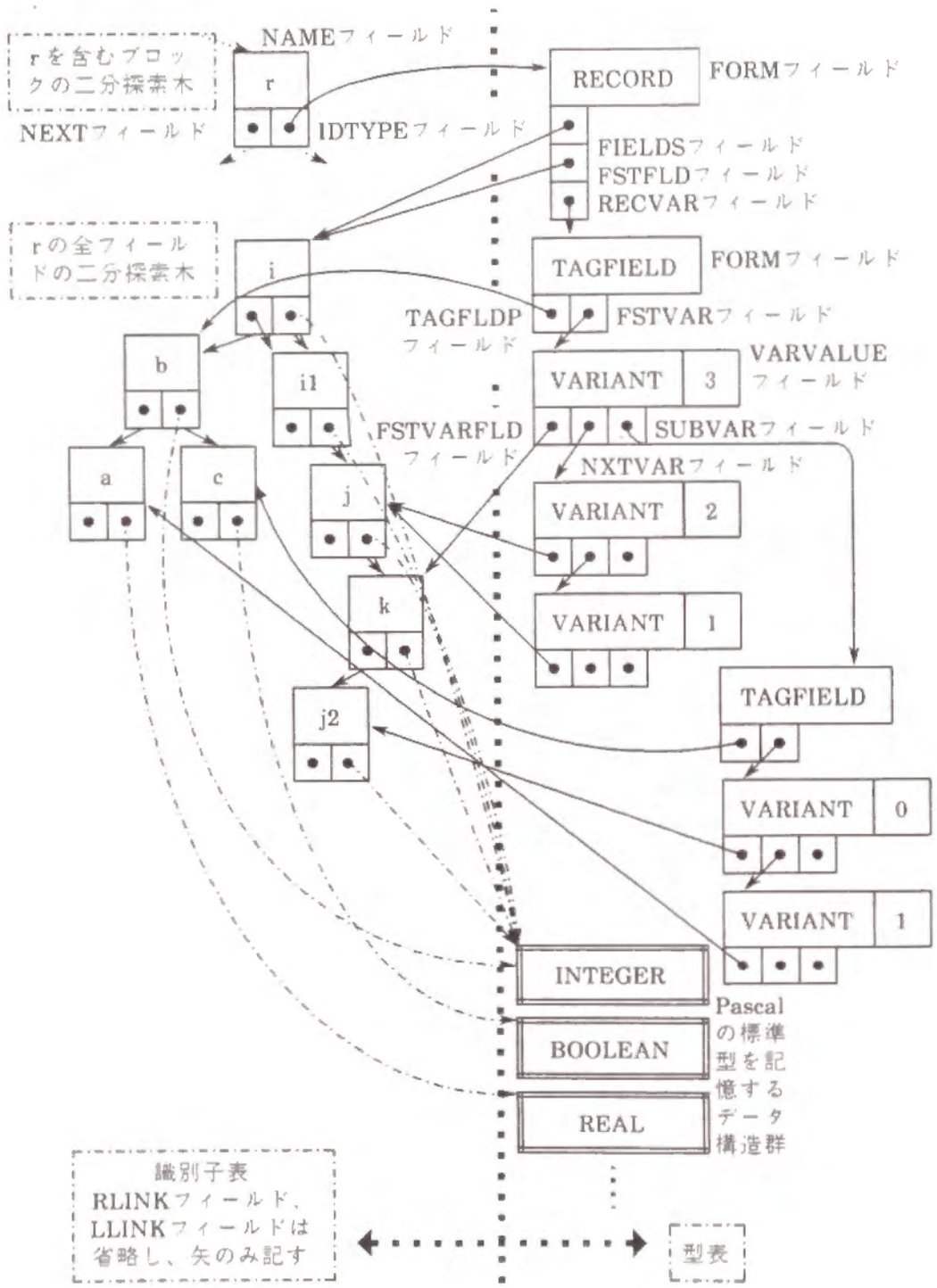


図4.7 レコード型の内部表現(つづき)

同じ構造の多次元配列の宣言の例

```
type
    a1 = array[1..8] of array[1..5] of real;
    a2 = array[1..8,1..5] of real;
```

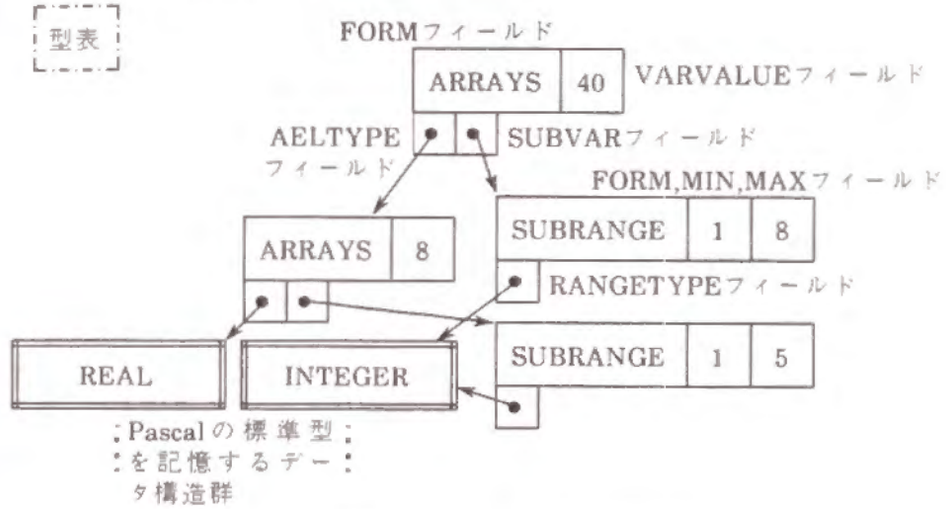


図4.8 多次元配列の内部表現

を持ち、かつ同一の値である定数が複数回出現する場合には、重複して記憶することを避けるため、以前に同一の型でかつ同一の値である定数が既に定数表に登録されていないか走査する。ただし、この走査はあるプログラムにおいて定数の出現頻度が識別子の出現頻度に比較してはるかに少ないことが予想されることから、効率を悪化させることは少ないと考えられる。フェーズ1終了後は、中間コード表現では必要な定数の値は、直接この定数表の各エントリを指し示すポインタを用いて得ることができるため、定数表が走査されることはない。フェーズ5目的コード生成部では、目的コードの定数領域(7.2.4項参照)に各定数の値を順次作成するが、その走査の場合にはリニア・リストが効率的で適切である。このとき、上記のとおり登録時点で同一の型でかつ同一の値である定数が重複していないことが保証されているため、単純に各定数を生成するだけでよい。このように、フェーズ5目的コード生成部まで、このリニア・リスト構造を保持する必要があり、フェーズ1終了後は、中間コードの手続/関数宣言ノード中のCONSTTOPフィールドが、その手続/関数内で宣言された定数表を保持するため、このリニア・リストの先頭を指し示す(後の4.3.2項、図4.11参照)。

4.3.2 中間コード

いままでにも述べてきたとおり、V-Pascal Version 1 のフェーズ1構文/意味解析部は、与えられたユーザ・プログラムを中間コード表現に翻訳する。以降の処理は、この中間コードを走査し進められる。それゆえに、中間コードは重要なデータ構造である。

V-Pascal Version 1 の中間コードは、以下の点を考慮して設計されている。

(1) Pascal で記述された任意のユーザ・プログラムの意味を正確に表現できること。

(2) 通常のスカラ処理だけでなく、ベクトル処理の記述も容易であること。

(3) 最適化処理に伴う中間コードレベルの挿入、削除、移動等の操作が容易にできること。

(4) ベクトル化処理に伴う各種依存解析が容易にできること。

(5) 現存の各種ベクトル計算機にも移植できるように、機械依存部をできるだけ小さくできること。

具体的には、上記第1点のためにはまず豊富なデータ構造の記述に対応するため、4.3.1 項で述べた型表を用い、後述の変数記述子を用意した。この変数記述子は、第4点をも考慮し配列参照の添字式を容易に抽出できる構造となっている。豊富な制御構造の記述に対応するため、後述のような種々の制御の流れに関する中間コードを用意した。

第2点のためには、スカラ処理を記述する中間コードと別にベクトル処理を記述する中間コードも用意した。このベクトル処理記述用の中間コードは、第5点をも考慮し現存の各種ベクトル計算機が有する各種ベクトル命令をできるだけ包含する方針で設計されている。これらの処理を記述する中間コードは、基本的に四つ組(quadruple)[36]の形態を採用し、演算子の種類、第1被演算子、第2被演算子、演算結果のためのフィールドを有する。この中間コードの演算結果を**中間項**と以後呼ぶ。これは、作業領域上の一時変数と解釈できる。実際には、目的コード生成部において、レジスタを割り付けられる対象となる。この中間項には、一意に識別番号が与えられる。本論文中では中間項の識別番号には先頭に「@」を付記し、中間コード自体の識別番号の先頭には「#」を付記することにより明確に区別することにする。中間項は、上述のとおり一時変数と解釈できるが、1中間コードによってのみその値

が定義される。すなわち、通常の変数と異なり、複数回再定義されることはない。つまり、中間項(の識別番号)とそれを定義する中間コード(の識別番号)とは、1対1に対応する。従って、V-Pascal Version 1 では、この中間項(の識別番号)→中間コード(の識別番号)の対応関係をハッシュ表に登録し、高速に検索できるようにした。この種の検索は、最適化はじめ様々な処理において必要となる。

第3点のためには、全中間コードは物理的には、ユーザ定義の手続/関数(メイン・プログラムもその一つ)ごとに、1中間コードを1ノードとし、後述の手続/関数宣言ノードを先頭および最後尾とした双方向のサーキュラ・リスト構造を採用している(後の図4.11参照)。各ノードは、識別子表、型表等と同様に中間コードの下記の種別ごとに必要な情報を保持する可変レコード(INTERCODE型)で実現されている。中間コードの全種共通のフィールドとその意味を表4.5にまとめる。表4.5のLASTLINK、NEXTLINKの両フィールドにより前記の双方向のサーキュラ・リスト構造を形成する。

中間コードの種類は大別すると図4.9に示すとおり、まず制御の流れを記述するためのノード群と各種の処理を記述するためのノード群とに分類できる。表4.5に挙げたタグ・フィールドのOPフィールドの値が示す中間コードの全種類を表4.6にまとめた。この表4.6の種別の欄には図4.9の分類記号を併記し関係を明示した。なお図4.9で、(a-6) EXIT ノードは表4.6内に記したとおり、Pascal 8000 において標準Pascal から拡張された loop 文[31]~[35]内からの飛び出しを表現するために、用意したものである。このノードは、フェーズ1構文/意味解析部が exit 文を走査した時点で

表4.5 中間コードのノードの共通フィールド

フィールド名	意味
IDN	中間コードの識別番号
NUMBER	元のソース・プログラムの行番号
SNUMBER	元のソース・プログラムの同一行内の文番号
GRAPHP	対応するコントロールフロー・グラフのバーテクスを指す。
DMRXPT	対応するD行列の行列ノードを指す。
DFLWK	定義された中間項に関するデータフロー解析用の定義/引用表を指す。
LASTLINK	前の中間コードのノードを指す。
NEXTLINK	次の中間コードのノードを指す。
OP	中間コードの種別(OPRTNS型)を示すタグ・フィールド。

(a) 制御の流れの記述ノード
- (a-1) 手続/関数宣言ノード
  (ヘッダ、PROCFUNCDCL型ノードとも呼ぶ)
- (a-2) 手続/関数呼び出しノード
- (a-3) 分岐型ノード(プレディケート・ノード)
- (a-4) 合流型ノード(コレクティング・ノード)
- (a-5) GOTOノード
- (a-6) EXITノード
- (a-7) ロングジャンプエントリ・ノード
- (a-8) for ループ記述ノード
  - (a-8-1) ループビギン・ノード
  - (a-8-2) ループエンド・ノード
- (a-9) ベクトル型中間コードノードリンク用ノード
  - (a-9-1) EXVPノード
  - (a-9-2) TVPノード
  - (a-9-3) セットアップ・ノード

(b) 処理記述用演算ノード(プロセス・ノード)
- (b-1) スカラ処理記述ノード(スカラ型ノード)
  - (b-1-1) NOPノード
  - (b-1-2) ロード/ストア/ロードアドレス・ノード群
  - (b-1-3) スカラ単項演算ノード群
  - (b-1-4) スカラ2項演算ノード群
- (b-2) ベクトル処理記述ノード(ベクトル型ノード)
  - (b-2-1) ベクトルロード/ストアノード群
    直接アクセス型(3.3.1 項参照)
    連続アクセス(等間隔アクセスを含む)
    収集・拡散(収集・分散、圧縮・伸長)
    間接アクセス型(3.3.1 項参照)
  - (b-2-2) ベクトル単項演算ノード群
  - (b-2-3) ベクトル2項演算ノード群
  - (b-2-4) VENDノード
  - (b-2-5) VWACノード

図4.9 V-Pascal の中間コードの種別

生成するが、それを含む loop 文の最後まで解析が終了すると、飛び先が確定するため取り除かれる。もちろん、その際必要に応じて関係する前後の中間コードの LASTLINK、NEXTLINK の両フィールドがつなぎかえられ、(a-4) 合流ノードが付加される。従って、(a-6) EXIT ノードはフェーズ 1 以降はまったく存在しなくなる。同様に、goto 文を走査した時点で生成される (a-5) GOTO ノードは、飛び先が確定した後はロングジャンプの場合(後述のロングジャンプエントリ・ノードの解説参

表4.6 中間コードの種類と意味(つづく)

種別 [( )内は図4.9 の分類記号]	細分類	
	タグ・フィールド OP の値	意味
(a-1) 手続/関数宣言	PROCFUNCDCL	ソース・プログラムの手続/関数宣言に対応
(a-2) 手続/関数呼び出し	PROCCALL, FUNCCALL	手続/関数呼び出しを表現
(a-3) 分岐	PREDICATE	条件分岐を表現
(a-4) 合流	COLLECTING	制御の流れの合流点を表現
(a-5) GOTO	GOTOX	goto 文による無条件ジャンプを表現
(a-6) EXIT	EXITX	Pascal 8000 の exit 文によるループ外への飛び出しを表現
(a-7) ロングジャンプエントリ	LJMPENTRY	goto 文による手続/関数外への飛び出しの入口点を表現
(a-8-1) ループビギン	LOOPBEGIN	for ループの始まりを表現
(a-8-2) ループエンド	LOOPEND	for ループの終わりを表現
(a-9-1) EXVP	EXVP	ベクトル処理の開始を表現
(a-9-2) TVP	TVP	ベクトル処理の終了を表現
(a-9-3) セットアップ	SETUP	ベクトル処理のためのセットアップ開始を表現
(b-1-1) NOP	NOP	No operation
(b-1-2) ロード/ストア/ロードアドレス	LOADS	スカラ実行による値のロード
	ST	スカラ実行による値のストア
	LOADADRS	スカラ実行による番値のロード
(b-1-3) スカラ単項演算	ABSS	絶対値算出
	NOTS	論理否定
	COMP	符号反転
	MOV	値のコピー

照)を除き、LASTLINK、NEXTLINKの両フィールドによる制御の流れの表記に置換されるので、存在しなくなる。一方、(b-2)のベクトル型ノード群およびそれに関連した(a-2)のベクトル型ノードリンク用ノード群は、当然フェーズ3の部分ベクトル化処理部によってのみ生成されるので、それ以前には存在し得ない。特に、(b-2-

表4.6 中間コードの種類と意味(つづく)

種別 [()内は図4.9の分類記号]	細分類	
	タグ・フィールド OPの値	意味
(b-1-4) スカラ2項演算	ADD	加算
	SUB	減算
	MULT	乗算
	IDIVS	整数除算(余り切り捨て)
	RDIVS	実数除算
	IMODS	整数剰余
	EXPO	べき乗
	PINS	集合要素かどうかの論理演算
	MKSET	集合の生成
	SLL	左論理シフト
	SLA	左算術シフト
	SRL	右論理シフト
	SRA	右算術シフト
	EQS	等しいかどうかの論理演算
	NES	等しくないかどうかの論理演算
	GTS	大きいかどうかの論理演算
	GES	以上かどうかの論理演算
	LTS	小さいかどうかの論理演算
	LES	以下かどうかの論理演算
	ANDS	論理積
	ORS	論理和
	XORS	排他的論理和
(b-2-1)連続アクセスロード/ストア	VL	連続(等間隔)ベクトル・データのロード
	VST	連続(等間隔)ベクトル・データのストア
(b-2-1)収集/拡散	VLE	収集型のベクトル・データのロード
	VSTC	拡散型のベクトル・データのストア
(b-2-1)間接アクセスロード/ストア	VLI	間接参照によるベクトル・データのロード
	VSTI	間接参照によるベクトル・データのストア

4) VENDノード、(b-2-5) VWACノードは、フェーズ5目的コード生成部が生成する一時的な中間コードである。

(b-2-1)の主記憶とベクトル・レジスタ間の種々のロード/ストアを表現する中間コード群は、先に記したとおり現存の各種ベクトル計算機が有する各種ベクトル命令をできるだけ包含する方針で設計されていることの一例である。これらの中間コード群の表す機能は、対応する各種ベクトル命令に同じである。(b-2-2)ベクトル単項演算、(b-2-3)ベクトル2項演算の各種演算についても同様である。

(a-8)のforループ記述ノードは、for文を走査した時点で生成される。これは、V-Pascal Version 1のベクトル化対象ループが多重にネストしたfor文であるため、特に検索を容易にするため設計した。詳細は後述する。Pascalのその他のループを表すrepeat文、while文については、if文およびgoto文で形成されるループと同様に(a-3)分岐ノードおよび(a-4)合流ノードを組み合せて等価なループ構造を表現する。この両ノードを組み合せることにより、if-then-else構造はもちろん、飛び出しのあるループ(forループの場合にはforループ記述ノードをも組み合せることにより)等Pascalが持つ豊富な制御構造の任意のものを表現することが可能となる。

表4.6 中間コードの種類と意味(つづく)

種別 [()内は図4.9の分類記号]	細分類	
	タグ・フィールド OPの値	意味
(b-2-2) ベクトル単項演算	VNOT	論理否定
	VABS	絶対値
	VCOMP	符号反転
	VSM	ベクトル・データの合計
	VMAX	ベクトル・データの最大値(と要素番号)
	VMIN	ベクトル・データの最小値(と要素番号)
	VCLZ	先頭から連続する0の計数
	VCTZ	後尾から連続する0の計数
	VCO	ベクトル・データ内の1の計数
	VMOV	ベクトル・データのコピー
	VMCOMP	マスク・データのビット反転
	VMCLZ	マスクの先頭から連続する0の計数
	VMCTZ	マスクの後尾から連続する0の計数
	VMCO	マスク・データ内の1の計数
	VMMOV	マスク・データのコピー

(a-1)手続/関数宣言ノードは、図4.10に示すとおり元のソース・プログラムの手続/関数宣言の入れ子の木構造をBLLINK(長子)、BRLINK(弟)、BLBLINK(親)、BRBLINK(兄)の4フィールドを使い表現する。なお、図4.10の中にも示したとおり、手続/関数宣言の入れ子の木構造の根であるユーザメイン・プログラムは、ポインタ変数IPRGRMDCLが常に指し示しており、木としてのなぞりを可能としている。ポインタ変数ICUPFDCLは、前述のとおりフェーズ2以降でV-Pascalコンパイラが種々の処理を手続/関数を単位として行う際に、現在処理中の手続/関数を指す。こ

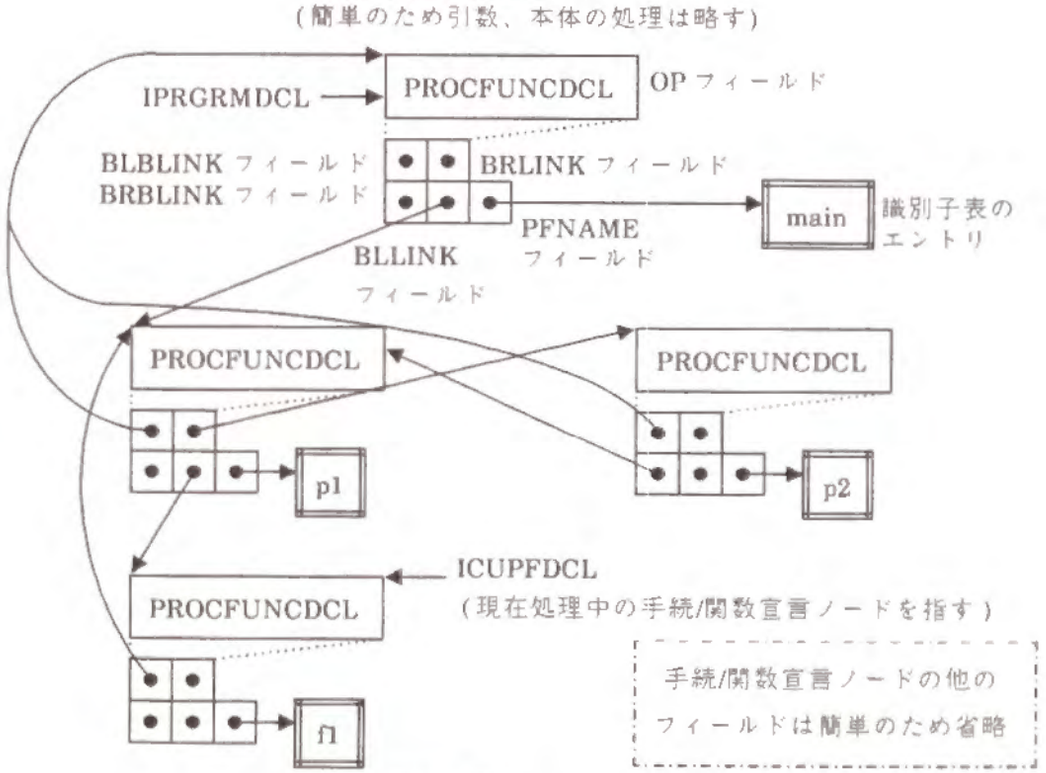
表4.6 中間コードの種類と意味(つづき)

種別 [( )内は図4.9 の分類記号]	細分類	
	タグ・フィールド OPの値	意味
(b-2-3) ベクトル2項演算	VADD	加算
	VSUB	減算
	VMUL	乗算
	VDIV	除算
	VSLL	左論理シフト
	VSLA	左算術シフト
	VSRL	右論理シフト
	VSRA	右算術シフト
	VCE	等しいかどうかの論理演算
	VCNE	等しくないかどうかの論理演算
	VCGT	大きいかどうかの論理演算
	VCGE	以上かどうかの論理演算
	VCLT	小さいかどうかの論理演算
	VCLE	以下かどうかの論理演算
	VAND	論理積
	VOR	論理和
	VXOR	排他的論理和
	VMAND	マスク・データの論理積
	VMOR	マスク・データの論理和
	VMXOR	マスク・データの排他的論理和
	VITR	等差数列の生成
	VMG	マスク制御による2ベクトルのマージ
	VIP	2ベクトル・データの内積
(b-2-4) VEND	VEND	一連のベクトル型中間コードの終了
(b-2-5) VWAC	VWAC	メモリ参照の待ち

の手続/関数を単位とする処理のため、(a-1)手続/関数宣言ノードは、図4.11に示すとおり手続/関数ごとにまとめられた中間コード列、識別子表、型表、定数表を保持する役割も果たす。すなわち、ユーザ・プログラムは一連の中間コード列に翻訳される

```
program main;
    procedure p1;
        function f1:integer;
            begin {f1}
            end {f1};
        begin {p1}
        end {p1};
    procedure p2;
        begin {p2}
        end {p2};
    begin {main}
    end {main}.
```

(a) 手続/関数宣言の入れ子の例
(簡単のため引数、本体の処理は略す)



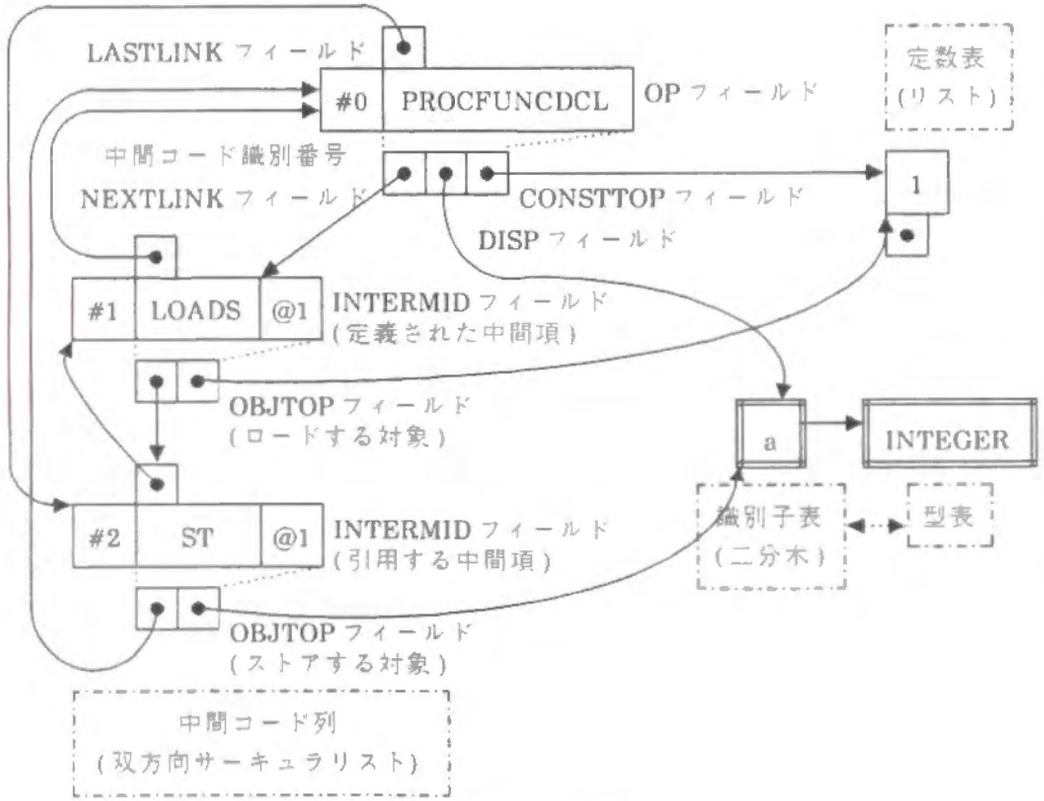
(b) 手続/関数宣言ノードによる表現

図4.10 手続/関数宣言ノードによる宣言の入れ子の木構造の表現

のではなく、ユーザメイン・プログラムを含め手続/関数ごとに独立した双方向のサーキュラ・リストとしている。また、この図では省略したがフェーズ2の当該手続/関数に対するAlias解析(4.5節参照)の結果であるAlias表を保持するため、ALIASLISTフィールドを持つ。同様に、当該手続/関数内で定義/引用される大域変数のリニア・リストを、それぞれDEFLISTフィールドおよびREFLISTフィールドにまとめている。これらのリニア・リストは具体的には後述の変数記述子を持たせたセル

```
program example4;
    var a: integer;
    begin
        a := 1;
    end.
```

(a) ソース・プログラムの例



(b) 中間コード表現

図4.11 単純なソース・プログラムの中間コード表現

(VARLIST型)で構成される。LJUMPLISTフィールド、LJMPENTLISTフィールドは、それぞれ当該手続/関数内でロングジャンプし飛び出すノードおよび子孫の手続/関数内からロングジャンプし飛び込んでくるノードのリストである。さらに(a-1)手続/関数宣言ノードは、CALLLISTフィールド、CALLEDLISTフィールドを持つ。前者が当該手続/関数内に存在する(a-2)手続/関数呼び出しノード(を指し示すポインタ)のリストの先頭を指す。後者は、当該手続/関数を呼び出している(a-2)手続/関数呼び出しノードのリストの先頭を指す。後者のリストは、(a-2)手続/関数呼び出しノードが持つNEXTCALLEDフィールドが、順次次の手続/関数呼び出しノードを指し示すことにより形成される。この二つのリストは、図4.1に示したフェーズ2の手続/関数相互呼出し表作成部により生成される。すなわち、手続/関数相互呼出し表は、物理的にはこれらのリストで表現されている。また、(a-1)手続/関数宣言ノードは、LPLISTフィールドにより、ベクトル化対象多重forループの検索/処理を容易にしている。同フィールドは後述のベクトル化対象多重forループの入れ子構造の最外側の最初のループを指すものである。

(a-2)手続/関数呼び出しノードは、呼び出す手続/関数を示す識別子表のエントリを指すIDPTRフィールド、ならびに実引数の並びを忠実に表現するリストを保持するためのARGLISTフィールドを持つ。この実引数の並びリストでは、値呼びの引数の場合には、引用する中間項番号を保持し、番地呼びの引数の場合には、ロード/ストアノードと同様の変数記述子を保持する。NEXTCALLEDフィールドについては先に触れたとおりである。また、関数呼び出しのノードでは関数の結果の中間項番号を保持するINTERMIDフィールドならびに、その中間項の型を示すため型表のエントリを指すTYPPEフィールドも持つ。

(a-3)分岐ノードならびに(a-4)合流ノードは、それぞれ単に制御の流れの分流/合流点を示すだけであり、中間コードの全ノードが有するNEXTLINK、LASTLINK両フィールドのほかに、前者はNXTLINK2フィールドを持ち、後者はLSTLINK2フィールドを持つ。また、(a-3)分岐ノードは分岐する条件式の結果の中間項番号を保持するCONDフィールドをも有する。同フィールドが示す中間項が偽の場合には、NEXTLINKフィールドの指し示す中間コードへ制御が移り、真の場合には、NXTLINK2フィールドの指し示す中間コードへ制御が移る。

(a-7)ロングジャンプエントリ・ノードは、先述のとおり図4.12に示すような自分の先祖のブロックで定義されたラベルで、自分の先祖の手続/関数本体内部の飛び先へ

ジャンプする場合に、その飛び先に付けられる。実行時のロングジャンプは、図4.12の例では関数f1からリターンし、かつ飛び先であるmain内での環境(スタックトップ等)を復旧する必要があり、通常のジャンプとは目的コードにおいても大きく異なる。

(a-8-1)ループビギン・ノードは、元のソース・プログラムのforループの入れ子の木構造を手続/関数宣言ノードと同様にLLINK(長子)、RLINK(弟)、LBACKLINK(親)、RBACKLINK(兄)の4フィールドを使い表現する。ただし、forループの場合の入れ子の論理的な構造は、厳密には木構造ではなく林構造である。すなわち、図4.13に示すとおり手続/関数本体内部ではネストの深さレベル1の最外側forループが複数個並列して並んでいることもある。このような林構造を表現するため、図4.13に示したレベル0のダミーのforループを考えることにより、木構造にまとめて表現している。こうすることにより、ベクトル化に伴い必要となるforループの探索ならびにforループを対象とした解析/複製/除去等各種の操作が非常に容易となる。レベル0のダミーを設けたことにより、特にレベル1のforループの除去がより深いレベルの場合と同様に統一的に処理できる。なお、図4.13中に記した

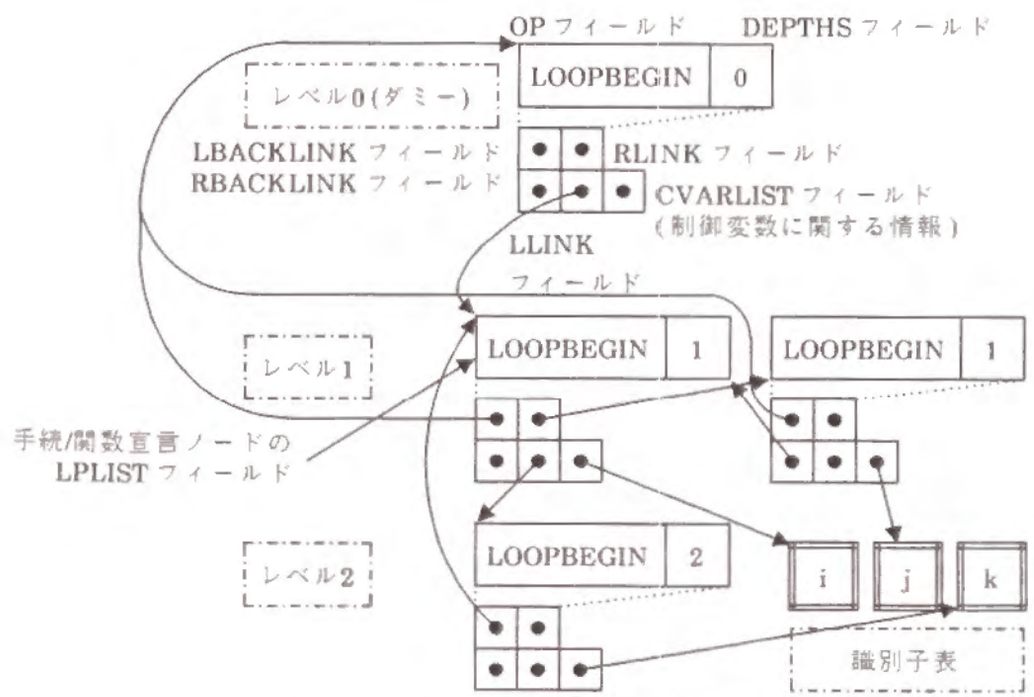
```
program main;
  label 999;{先祖のブロックで定義されたラベル}
    ⋮
  procedure p1;
      ⋮
    function f1:integer;
        ⋮
      begin {f1}
          ⋮
        goto 999;{ロングジャンプ}
          ⋮
      end {f1};

    begin {p1}
      ⋮
    end {p1};

  begin {main}
    ⋮
  999:{先祖の本体内部の飛び先:ロングジャンプエントリ}
    ⋮
  end {main}.
```

図4.12 ロングジャンプの例

CVARLISTフィールドは、図中では単に当該forループの制御変数の識別子表のエントリを指し示すように簡略化しているが、実際には制御変数の上下限式、増分値に関する情報も合わせて保持している。この制御変数の増分値は、標準のPascalでは1または−1(downtoの場合)しか取り得ないが、将来for文で記述されていないfor

```
program forexample;
  var i,j,k:integer;
  begin {main}
    for i:=1 to 10 do begin {for i}
      for k:=1 to 10 do begin {for k}
      end {for k};
    end {for i};
    for j:=1 to 10 do begin {for j}
    end {for j};
  end {main}.
```

(a) forループの入れ子の例
(簡単のためループ内部、その他の処理は略す)



(b) ループビギン・ノードによる表現

図4.13 ループビギン・ノードによるforループの入れ子の木構造の表現

型のループをベクトル化できるように任意の値を保持できる。さらに、将来制御変数と同様の働きを持つ帰納変数(induction variable)[36]についても制御変数と同等に扱えるよう、制御変数/帰納変数の情報を複数個リストにして記録できる。また、同図中 DEPTHS フィールドは、当該for ループのネストの深さレベルを記録している。図4.14には、(a-8) for ループ記述ノードのループビギン・ノードおよびループエンド・ノードの連係により for ループがどのように表現されるかを示した。この図からわかるとおり、手続/関数ごとにまとめられた中間コード列を示す双方向のサーキュラ・リストの基本構造とは別に、両者は互いにリンクされている。こうすることにより、先にも述べた for ループを対象とした各種の操作が非常に容易となる。

ベクトル型ノードリンク用ノードの機能は、ベクトル処理を記述したベクトル型中間コード列およびその実行のために必要となるセットアップのためのスカラ型中

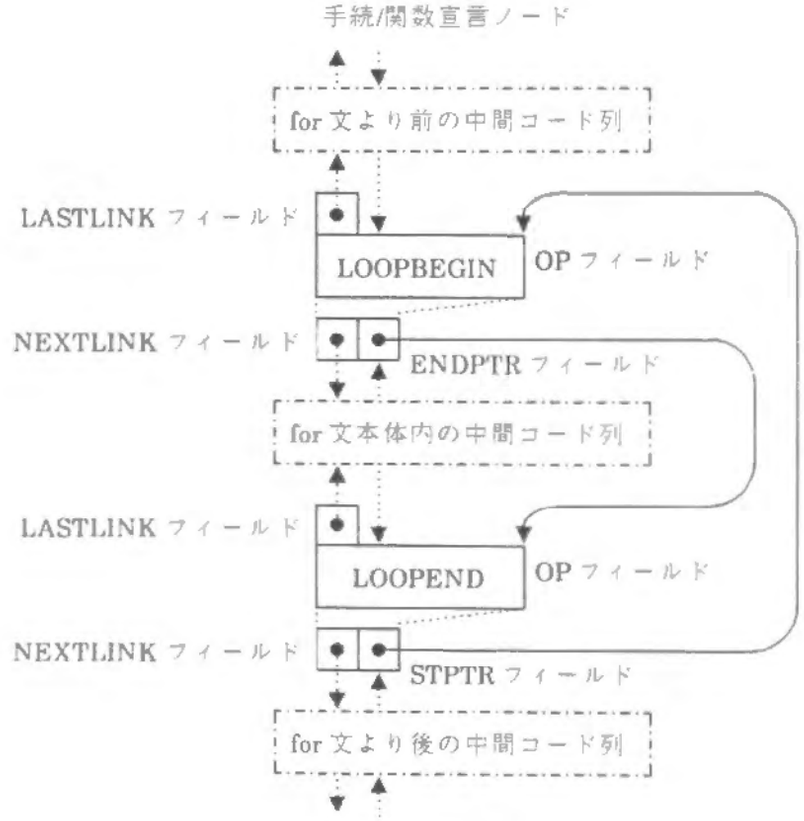


図4.14 基本的な for ループの中間コード表現

間コード列を容易に識別可能な形式で、その他のスカラ型中間コード列とリンクすることである。すなわち、図4.15に示すとおり、(a-9-1) EXVP ノードと (a-9-2) TVP ノードとの関係は上記のループビギン・ノードとループエンド・ノードとの関係と同じく、両ノード間の中間コードを認識/抽出しやすいようにリンクされている。(a-9-1) EXVP ノードと (a-9-3) SETUP ノードとの関係についても同じである。また、(a-9-1) EXVP ノードと (a-9-2) TVP ノードとの関係は、分岐ノードと合流ノードとの関係とも類似点がある。EXVP ノードでは、並列実行されるスカラ型中間コード列への制御の流れと並列実行されるベクトル型中間コード列への制御の流れとが生じ、それら2つの制御の流れが TVP ノードにおいて合流する。従って、この両中間コード列とのリンクも分岐ノードと合流ノードと真側/偽側の中間コード列とのリンクと同様とした。すなわち、図4.15から明らかなように、双方向のサーキュラ・リスト構造を保ちつつ EXVP ノードにおいて二つに枝わかれし、TVP ノードにおいて再度1本にまとめられる。もちろん、分岐ノードの場合と異なり、EXVP ノードにおいて並列実行されるため両方の制御の流れが常に生じ、TVP ノードにおいては同期がとられることとなる。こうして、第2章で述べたベクトル計算機のアーキテクチャが有するスカラ処理ユニットとベクトル処理ユニットとの並列実行を中間コード上で表現できる。なお、実行効率を考慮しセットアップのためのスカラ型中間コード列がその他のスカラ型中間コード中に埋め込まれることが多く、その場合には (a-9-3) SETUP ノードは結合されない。

(b-2-1) NOP ノードは、後述のコントロールフロー・グラフの1頂点を確保するために使用されるだけである。

(b-2-2) スカラ実行のロード(ここでの説明ではロードアドレスも含むものとする)/ストア・ノードは、図4.11に単純化して記述したとおり、変数等の値をロードしてきた結果の中間項番号あるいは変数等に値をストアする中間項番号を記録する INTERMID フィールドならびに、その中間項の型を示すため型表のエントリを指す TYPPE フィールドを持つ。そしてさらに、ロード/ストアの対象となる Pascal の変数(付録構文図の Variable 参照)は特に構造のある型の場合に種々の限定子を有する複雑な記述が可能である。そのような場合にも対処できるように変数記述子(我々のグループでは、Top-sub-descriptor と称している)を用意した。ロード/ストア・ノードは、この変数記述子を OBJTOP フィールドとして持つ。表4.7は、変数記述子はどのようなフィールドを持ち、各フィールドがどのような情報を保持しているかを示し

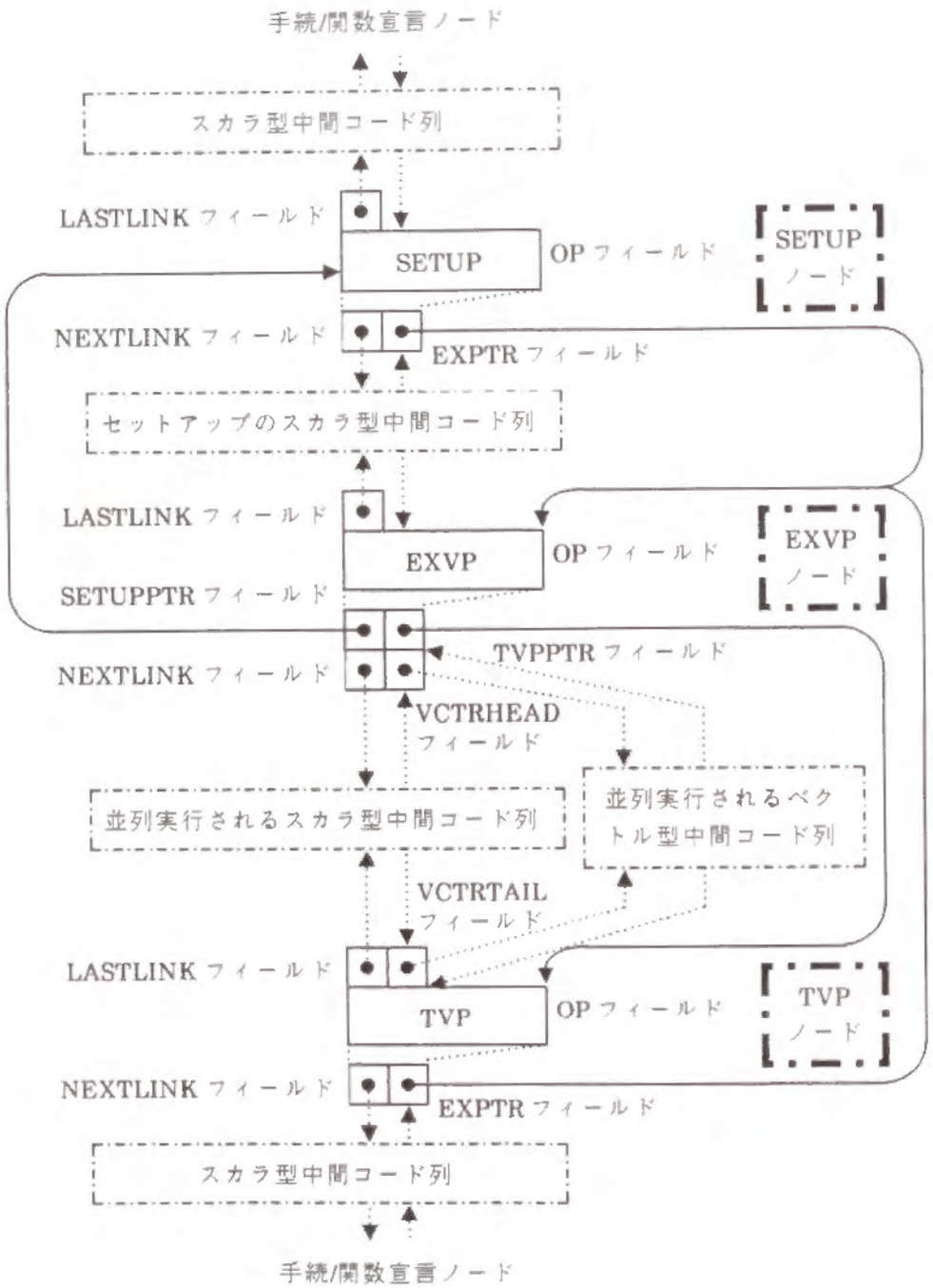


図4.15 標準的なベクトル型ノードリンク用ノード間の関係

ている。特に、重要なPascalの種々の構造のある型の変数を記述できるSTRUCTS(構造型変数)の場合には、表4.8に示した変数記述限定子(我々のグループでは、Sub-descriptorと称している)を用いている。変数記述限定子は、表4.8の可変フィールドのほかに、(1)TYPPEフィールド、(2)NEXTフィールドを持つ。前者は、当該変数記述限定子により限定された型を示す型表内のエントリを指す。後者はもしあれば、さらに続く次のより細かな限定を行う変数記述限定子を指す。なお、表4.8のタグ-フィールドSTRUCTFORMは、このTYPPEフィールドが指し示す型表内のエントリ中に存在する。ポインタ変数により指し示されるレコード等の記述は、表4.8の可変フィールドのない変数記述限定子が1段追加される。図4.16に変数記述子を用いた記述例を示す。この図からもわかるとおり変数記述子による表現において、配列の添字式の結果を表す中間項番号が明示されるため、添字式の演算木を容易にたどることが可能である。これは、配列要素間のデータ依存を解析する際に利点となる。この変数記述子は、スカラ型およびベクトル型の各種ロード/ストア系ノード中はもちろん、番地呼び[36],[37]の実引数を表現する際にも用いられる。

四つ組表現では、通常このような変数の記述は各演算子に対する被演算子として、中間項に相当する一時変数と同様に記述することが多い[36]。それに対し、ここに定義した中間コードは、上記のとおりロード/ストア用の特別な中間コードを用意した。これにより、(1)ベクトル化の際必要となるデータ依存の解析において、変数

表4.7 変数記述子の持つフィールド

タグ-フィールド STRUCTの値	可変フィールド	
	フィールド名	意味
CONSTS(定数)	CONSTP	定数表内のエントリを指す
NONSTRUCT (単純変数)	OBJVAR	識別子表内の変数名のエントリを指す
STRUCTS (構造型変数)	OBJVAR	識別子表内の変数名のエントリを指す
	NEXTDISCP	構造のある変数の変数記述限定子を指す

表4.8 変数記述限定子の持つ可変フィールド

タグ-フィールド STRUCTFORMの値	可変フィールド	
	フィールド名	意味
ARRAYS(配列)	IND	配列添字式を示す中間項番号
RECORDS (レコード)	IDPTR	識別子表内のフィールド名のエントリを指す

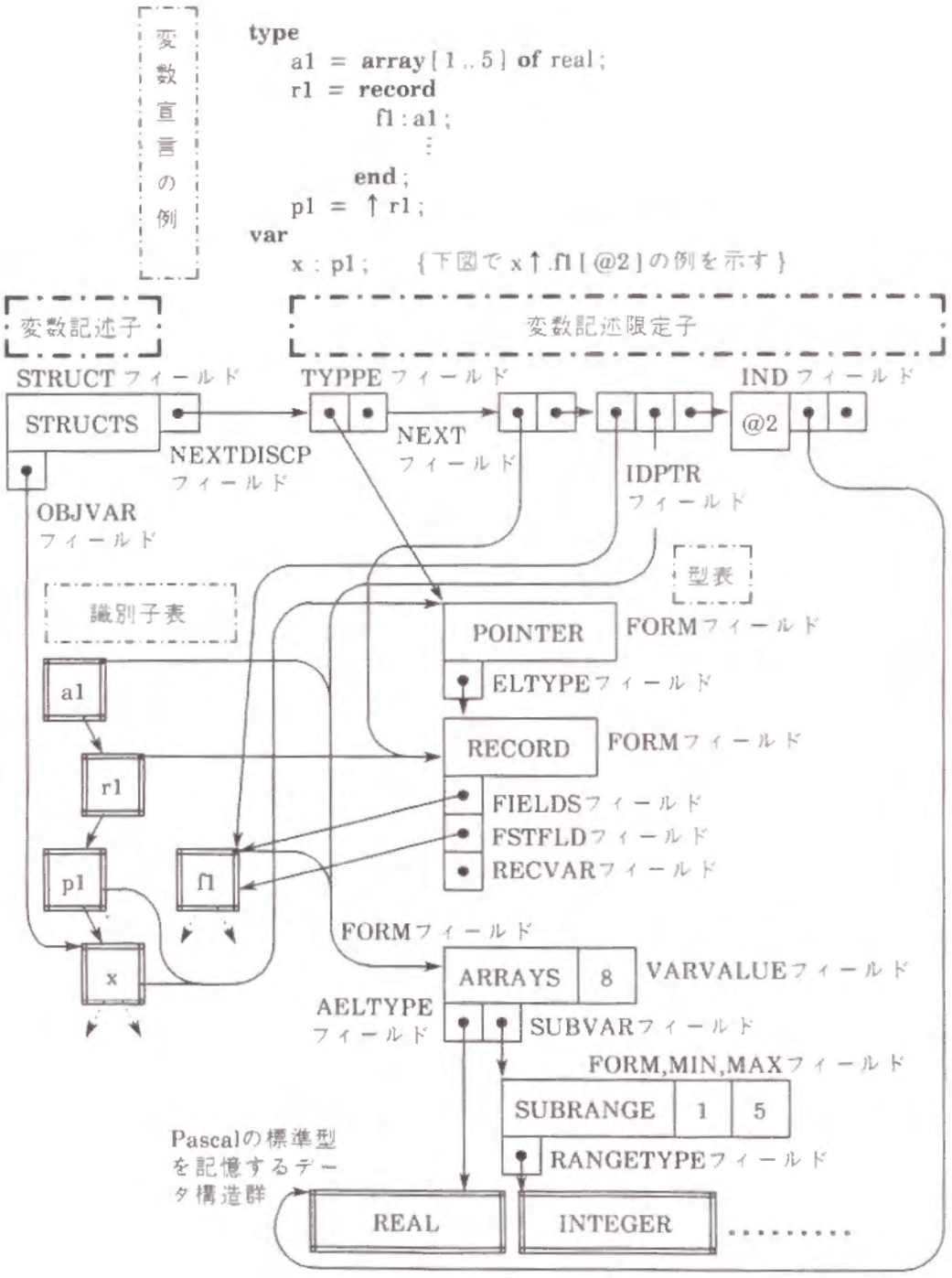


図4.16 変数記述子による表現

の参照を抽出する処理ではこのロード/ストア・ノードを探すだけとなり簡略化される。また、(2)先述のようなPascalの豊富なデータ構造を反映した複雑な表記に対しても、一般の演算ノードには無関係でロード/ストア・ノードの特に変数記述子のみで対応できる。つまり、ソース言語に依存する部分が局所化されており、その意味で抽象度が高い。

(b-1-3)スカラ単項演算ノード、(b-1-4)スカラ2項演算ノードは、表4.6にあげた演算種別を表すOPフィールドおよび演算結果を表すINTERMIDフィールドならびに、その中間項の型を示すため型表のエントリを指すTYPPEフィールドのほかに、第1被演算子の中間項番号を示すTERM1フィールドならびに後者の場合には第2被演算子の中間項番号を示すTERM2フィールドを持つ。これらのフィールドにより四つ組表現をとる。

(b-2-1)ベクトルロード/ストア・ノードは、表4.9に挙げた共通フィールドのほかに、スカラロード/ストア・ノードと同様参照する変数等を表記するため、変数記述子のVOBJTOPフィールドを持つ。また、ベクトルロード/ストア・ノードには第3章で詳述した種々の主記憶参照ベクトル命令に対応し、等間隔型、収集/拡散型、間接参照型を用意した。ただし、煩雑さを避けるためこれらの細かな種別間で保持する情報の差異を可能な限り少なくした。すなわち、間接参照型ベクトル命令には通常ない第1要素のオフセットを示すフィールドを他と共通に持たせた。図4.17に参照するベクトル・データと各フィールドの関係を示した。

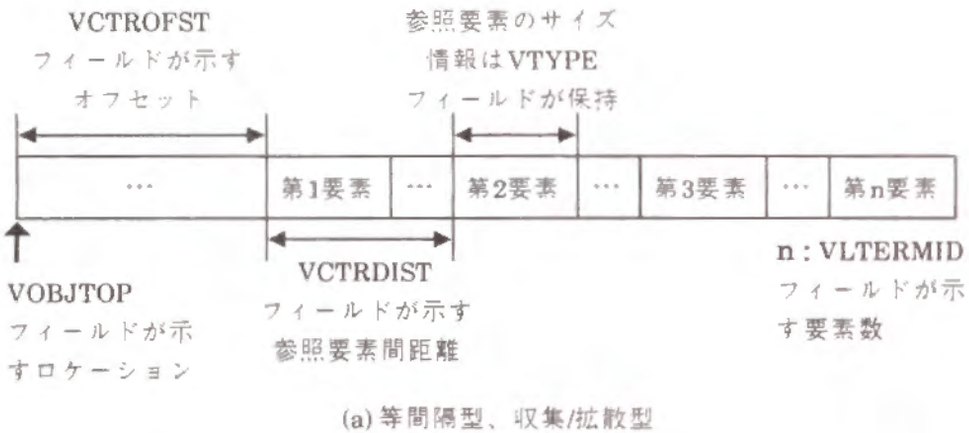
(b-2-2)ベクトル単項演算ノード、(b-2-3)ベクトル2項演算ノードは、表4.9の共通フィールドのほかに、スカラのそれと同じく第1被演算子の中間項番号を示すVTERM1フィールドならびに後者の場合には第2被演算子の中間項番号を示す

表4.9 ベクトルロード/ストア・ノードおよびベクトル演算ノードに共通フィールド

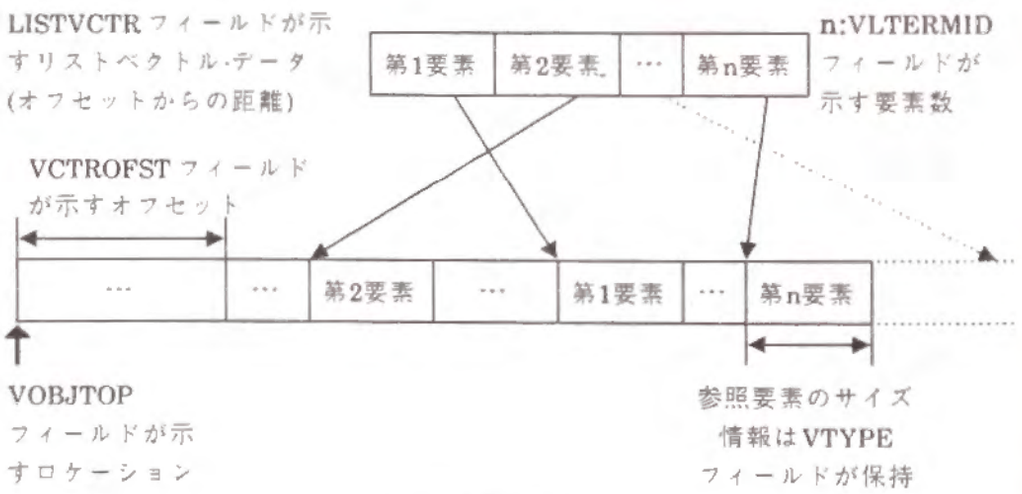
フィールド名	フィールド名
VLTERMID	ベクトル長を示す中間項番号
VTERMID	演算結果あるいは変数等の値をロードしてきた結果の中間項番号あるいは変数等に値をストアする中間項番号
VTYPE	上記の中間項の型を示す型表のエントリを指す
MASKPN	マスク制御を受けるか否か、ならびに受ける場合には正マスク制御(マスクが1の時のみ実行)か補マスク制御(マスクが0の時のみ実行)か
MASKID	マスク制御を受ける場合マスクを示す中間項番号

VTERM2フィールドを持つ。また、ベクトル･データの最大値/最小値を探索するVMAX/VMIN中間コードには、対応するベクトル命令が最大値/最小値の要素番号をも返すことに合わせるため、INDEXIDフィールドを用意した。

ベクトル処理記述ノードはすべて、表4.9に示すとおりマスクを持つことができる。実際のベクトル計算機では、これらの中間コードにおいて対応するベクトル命令がマスクを使えない場合もあるが、移植性を考慮し、このようにすべてのベクトル処理記述ノードにマスクを持つことができるようにした。



(a) 等間隔型、収集/拡散型



(b) 間接参照型

図4.17 ベクトルロード/ストア･ノードによる主記憶参照

4.3.3 コントロールフロー･グラフ

種々の最適化のために必要となる制御の流れに関するコントロールフロー解析[36]~[41](V-Pascalのコントロールフロー解析については4.6節で述べる)では、通常解析のコストを軽減するため、直接中間コードを走査するのではなく、中間コード列を直列形ブロック(straight-line block)あるいは基本ブロック(basic block)と呼ばれる[36],[37]ブロックに分割し、そのブロックを頂点としたコントロールフロー･グラフ(control flow graph あるいは単に flow graph と呼ばれる)[36],[37]を作成し、その上で解析をすすめる。

これに対し、V-Pascalのためのコントロールフロー･グラフは、次の各点を考慮して新たに設計された。

(A) 中間コード表現をとるユーザ･プログラムの制御の流れを正確に表現できること。

(B) 中間コードのベクトル処理の記述にも対応できること。

(C) 中間コードの挿入、削除、移動等の操作にともない、グラフの対応する頂点に対する同種の操作が容易にできること。

(D) ベクトル化処理にともなう制御の流れに起因した依存関係解析が厳密かつ容易に行えること。

(E) データフロー解析の単位として適当な粒度の中間コード群をひとまとめとすること。ならびに、同解析結果を容易に検索できるデータ構造で保持すること。

中間コードは、元のプログラムの意味を忠実に表現することが第一義であったのに対し、コントロールフロー･グラフは、まず上記(A)のとおり制御の流れを解析するために設計されている。すなわち、4.3.2項で述べた中間コード表現から制御の流れを抽出し、有向グラフで表したものである。従って、表4.10に示すとおり中間コードのうち、(a)制御の流れの記述ノードについては、(a-9)ベクトル型中間コードノードリンク用ノードを除き1対1に対応するコントロールフロー･グラフの頂点(以下バーテクスと呼び、中間コードの頂点「ノード」と区別する)を用意した。有向グラフの有向辺はポインタによるリンクで表現する。一連の(a-9)ベクトル型中間コードノードリンク用ノードならびに制御の流れを乱さない(b)処理記述用演算ノードについては、プロセス･バーテクスが対応する。すなわち、一連の(b-1)スカラ処理記述

ノードについては、1プロセス･バーテクスに集約して多対1で対応づける。こうすることにより、(b-1) スカラ処理記述ノードが制御の流れを乱さないため、制御の流れを簡便にしかも忠実に表現できる。この考え方は、上記(E)の設計目標を考慮した上で、通常のコントロールフロー･グラフの頂点として採用される基本ブロックにならったものである。上記(B)の設計目標に関しては、制御の流れを乱さない (b-2) ベクトル処理記述ノードは、図4.14 に示したように同一のベクトル長を持つベクトル処理ごとに、(a-9) ベクトル型中間コードノードリンク用ノードにより、スカラ処理記述と分流した形でリンクされるので、(a-9-1) EXVPノードとともに1プロセス･バーテクスに集約して多対1で対応づける。この粒度についても上記(E)の設計目標も考慮されている。この対応づけでは、コントロールフロー･グラフ上はベクトル処理記述側を先に実行する意味となるが、(a-9-1) EXVPノードで分流したスカラ処理記述とベクトル処理記述とは本来同時実行可能であるため、コントロールフロー･グラフを走査するデータフロー解析等後の処理において誤った結果を導くことはない。逆に、このように (a-9-1) EXVPノードとともにまとめることによりバーテクス数が少なくなり、その結果EXVPノードと分離した場合と比較してコントロールフロー･グラフを走査するコストが少なくてすむ利点を生む。(a-9-1) EXVPノードで分流した

表4.10 コントロールフロー･グラフの頂点の種別

頂点名称	タグ･フィールドGVTX の値	対応する中間コードの頂点
(G1) 手続/関数宣言バーテクス (ヘッダ･バーテクスとも呼ぶ)	HEADER	(a-1) 手続/関数宣言ノード
(G2) 手続/関数呼び出しバーテクス	GPRCALL	(a-2) 手続/関数呼び出しノード
(G3) 分岐型バーテクス	GPRED	(a-3) 分岐型ノード
(G4) 合流型バーテクス	GCOLLECT	(a-4) 合流型ノード
(G5) GOTO バーテクス	GGOTO	(a-5) GOTOノード
(G6) ロングジャンプエントリ･バーテクス	GLJE	(a-7) ロングジャンプエントリ･ノード
(G7) ループビギン･バーテクス	GLPBEGIN	(a-8-1) ループビギン･ノード
(G8) ループエンド･バーテクス	GLPEND	(a-8-2) ループエンド･ノード
(G9) プロセス･バーテクス	GPROCESS	(a-9) ベクトル型中間コードノードリンク用ノードおよび(b) 処理記述用演算ノード
(G10) ループ情報記述バーテクス	GLOOP	–

スカラ処理記述は、ベクトル処理記述側に続く1プロセス･バーテクスに対応づける。(a-9-3) セットアップ･ノードは、単にセットアップ処理を記述している部分を識別するだけのものであるので、通常の (b) 処理記述用演算ノードと同様にまとめてしまう。すなわち、(a-9-3) セットアップ･ノードのために、その前後の (b-1) スカラ処理記述ノード群が別個のコントロールフロー･グラフの頂点に対応づけられることはない。それに対し、(a-9-2) TVPノードでは、分流したスカラ処理記述とベクトル処理記述とが、再度合流するため独立した1プロセス･バーテクスに対応づける。それゆえに、その前後の (b-1) スカラ処理記述ノード群は別々のプロセス･バーテクスに対応づけられる。

コントロールフロー･グラフの各頂点は可変レコード (GRPH 型) で定義した。表4.11 に各頂点の共通のフィールドを、表4.12には同じく各頂点の可変フィールドを示した。表4.11 に示すとおり、IDN フィールドは、コントロールフロー･グラフの各頂点ごとに一意に付けられた識別番号を保持する。そこで、以降の図中ならびにその解説等においては、この識別番号を用い各バーテクスを区別するものとする。その際、中間コードの頂点の識別番号には頭に「%」記号を付記したのと同様に、「G%」を頭に付け表記する。

なお、4.3.2 で述べたとおり、(a-6) EXITノードはフェーズ1で一時的に利用されるだけであるので、対応するコントロールフロー･グラフの頂点は、用意していない。逆に、(G10) ループ情報記述バーテクスは、コントロールフロー･グラフを形成する頂点ではなく、制御の流れのうち for ループおよびそれ以外のすべてのループに関して、そのネスト構造、ループ入口点/出口点等種々の情報を保持する目的で用意され、コントロールフロー･グラフを補完するものである。それゆえに、対応する中間コードは存在しない。このループ情報記述バーテクスは、そのネスト構造の木を忠実に表現した物理的なデータ構造を有するループに関する表を形成するエントリと見ることができる。ここでの物理的なデータ構造は、中間コードの(a-8-1) ループビギン･ノードが for ループに関するネスト構造の木を表現していた場合と同様に双方向のリンク構造、すなわち、親子兄弟を指すリンクによりネスト構造の木を表現する。

従来のコントロールフロー･グラフの頂点は、基本ブロックを用いている。基本ブロックはその入口に複数の制御の流れが流入することならびに、その出口から複数の制御の流れが流出することを許しているのに対し、ここで解説する V-Pascal が持

つコントロールフロー・グラフでは、表4.10に示すとおり(G3)分岐型バーテクスおよび(G4)合流型バーテクスにより、制御の流れの分流点/合流点を明確に分離している。こうすることにより、第5章で述べる制御の流れによる依存関係解析が厳密に細

表4.11 コントロールフロー・グラフの頂点の共通フィールド

フィールド名	意味
IDN	当該頂点の識別番号
LASTLINK	制御の流れで先行の頂点を指す
NEXTLINK	制御の流れで後続の頂点を指す
PSLP	属するプライマリ・セット(4.3.4項、5.4節参照)を指す
ILPP	属する(10)ループ情報記述バーテクスを指す(4.6.2項参照)。なければnil
REVIDOM	直接逆支配頂点(4.6.1項参照)を指す
IDOM	直接支配頂点(4.6.1項参照)を指す
ECLSF1	支配関係解析(4.6.1項参照)の際の当該頂点のLASTLINK側の枝の種別
DFONUM	深さ優先の順序付け(4.7節参照、以下同様)による順序番号
DFONEXT	深さ優先の順序付けで後続の頂点を指す
DFOLAST	深さ優先の順序付けで先行の頂点を指す
VIRTDEFTOP	当該頂点に与えられた仮想的な定義(5.4節参照)のリスト
GEN	当該頂点で発生した定義を示すデータフロー集合(4.7節参照、以下同様)
KILL	当該頂点で無効となる定義を示すデータフロー集合
REACH	当該頂点に到達する定義を示すデータフロー集合
OUT	当該頂点から流れでる定義を示すデータフロー集合
DEFI	当該頂点で定義が引用に先行する変数(中間項も一時変数として含め統一的に扱う。以下同じ)を示すデータフロー集合
USE	当該頂点で引用が定義に先行する変数を示すデータフロー集合
BIN	当該頂点の入口で保持していると思われる値を以後で引用する可能性がある変数の集合を示すデータフロー集合
BOUT	当該頂点の出口で保持していると思われる値を以後で引用する可能性がある変数の集合を示すデータフロー集合
CMPLLIST	当該頂点の入口で配列化(3.3.1項、図3.5参照、以下同様)可能な名前のリスト
UCMPLLIST	当該頂点の入口で配列化不可能な名前のリスト
GVTX	タグ・フィールド

表4.12 コントロールフロー・グラフの頂点の可変フィールド(つづく)

タグ・フィールドGVTXの値	可変フィールド	
	フィールド名	意味
HEADER	INTERP	対応する中間コード
	DFOMAXNUM	深さ優先の順序付けの順序番号の最大値
	PREDVIRTDEF	制御関係解析(I)の仮想的な定義の集合
	PRED2VIRTDEF	制御関係解析(II)の仮想的な定義の集合
	PROCVIRTDEF	制御関係解析(III)の仮想的な定義の集合
	LASTLSTP	制御の流れで先行の複数個の頂点のリスト
	LJMPLSTP	当該手続/関数内のロングジャンプエントリ・バーテクスのみをリンクしたリスト
GLPBEGIN	INTERP	対応する中間コード
	PDEFLT	真側の第1の仮想的な定義のデータフロー集合の要素番号
	PDEFLF	偽側の第1の仮想的な定義のデータフロー集合の要素番号
	PDEFLT2	真側の第2の仮想的な定義のデータフロー集合の要素番号
	PDEFLF2	偽側の第2の仮想的な定義のデータフロー集合の要素番号
	LPPSLS	当該頂点の分岐が関係するプライマリ・セット
	DMYPRDCLC	解体した分岐型頂点、合流型頂点をさす
	LOOPP	当該頂点により形成されるループのループ情報記述バーテクス
GPRED	INTERP	対応する中間コード
	PDEFLT	真側の第1の仮想的な定義のデータフロー集合の要素番号
	PDEFLF	偽側の第1の仮想的な定義のデータフロー集合の要素番号
	PDEFLT2	真側の第2の仮想的な定義のデータフロー集合の要素番号
	PDEFLF2	偽側の第2の仮想的な定義のデータフロー集合の要素番号
	NXTLINK2	制御の流れで真側の後続の頂点
	PPSLS	当該頂点の分岐が関係するプライマリ・セット
	SAMECLCT	if-then-else構造で対応する合流型頂点

かく行える利点を持つ。すなわち、上記(D)の設計目標を満足する。

設計目標(C)のためには、中間コードとまったく同様に基本的に各頂点に自分の前後を指し示す双方向のリンクを用意(表4.11参照)し、(G1)手続/関数宣言バーテクス(ヘッダ・バーテクス)を先頭かつ最後尾とする制御の流れの全パスに沿った双方向

表4.12 コントロールフロー・グラフの頂点の可変フィールド(つづき)

タグ・フィールドGVTXの値	可変フィールド	
	フィールド名	意味
GCOLLECT	INTERP	対応する中間コード
	LSTLINK2	制御の流れで第2の先行の頂点
	SAMEPRED	if-then-else構造で対応する分流型頂点
	ECLSF2	LSTLINK2側の枝の支配関係解析時の種別
GPRCALL	INTERP	対応する中間コード
	JUMPLSTP	当該頂点から飛び出す先のロングジャンプエントリ・バーテクスのみをリンクしたリスト
GLJE	INTERP	対応する中間コード
	LJUMPLSTP	当該頂点へ飛び込むロングジャンプ・バーテクスのみをリンクしたリスト
GLPEND	INTERP	対応する中間コード
GGOTO	INTERP	対応する中間コード
GPROCESS	PRCSLSTP	対応する中間コードのリスト
GLOOP	DEPTH	ループ(4.6.2項参照)のネストレベル
	LLINK	ループのネスト構造の木の子
	RLINK	ループのネスト構造の木の弟
	BLLINK	ループのネスト構造の木の親
	BRLINK	ループのネスト構造の木の兄
	ICONTENT	直接ループを形成するバーテクスのリスト
	CONTENT	ネストした子孫のループを含めた各ループを形成するバーテクスのリスト
	LASTS	制御の流れで当該ループに入る直前のバーテクスのリスト
	NEXTS	制御の流れで当該ループから出た直後のバーテクスのリスト
	LPEXITS	当該ループの出口のバーテクスのリスト
	FORFLG	forループか否か
	LPBGNN	forループの場合ループビギンバーテクス

のサーキュラ・リストを形成する。4.3.2項で既述したとおり、中間コード列が手続/関数ごとにまとめられた双方向のサーキュラ・リストで表現されているので、そこから制御の流れを抽出したコントロールフロー・グラフにおいても、手続/関数ごとにまとめられた相似のデータ構造を持たせた。

4.3.4 プライマリ・セット表

プライマリ・セットとV-Pascal開発プロジェクトチームで呼んでいる概念は、同一のガード式(guard expression)[13]を有するコントロールフロー・グラフの頂点の集合を指す。ここで、ガード式とは、制御の流れのパスに着目し、パスの分岐点に相当する条件分岐の条件式を1ブール変数で表すことにより、各パスの実行条件をそれらのブール変数を用いたブール式で表現したものである(図4.18参照)。それゆえに、同一のガード式を持つコントロールフロー・グラフの頂点の集合であるプライマリ・セットとは、コントロールフロー・グラフの頂点の中で同一の実行条件を有するものをひとまとめにした概念である。すなわち、コントロールフロー・グラフの上部構造を表す。従って、V-Pascalコンパイラが保持するユーザ・プログラムの表現は次の3階層を成す。

Layer-1) 中間コード表現
Layer-2) コントロールフロー・グラフ表現
Layer-3) プライマリ・セット表

コントロールフロー・グラフが中間コード表現から制御の流れを抽出したものであったのに対し、プライマリ・セット表はコントロールフロー・グラフ表現から、同一のガード式の頂点をひとまとめにしている。この「同一のガード式」すなわち「同一の実行条件」とは、ベクトル化できた場合にマスク(3.3.6項で既述)が同一であることを意味する。このように、プライマリ・セットの概念は、if文等の条件分岐を含むループのベクトル化において有用な概念である。

プライマリ・セット表の物理的なデータ構造は、各エントリの単純なリニアリストで実現している。これは、プライマリ・セット表全体に渡る検索はなく、コントロールフロー・グラフのある頂点がどのプライマリ・セットに属するか(表4.11 PSLPフィールドによる対応づけ)、そしてそのプライマリ・セットのガード式を取り出すという目的のための1エントリに対するアクセスだけしか生じないためである。プライマリ・セット表の各エントリ(PSTLIST型)は以下のフィールドを持つ。

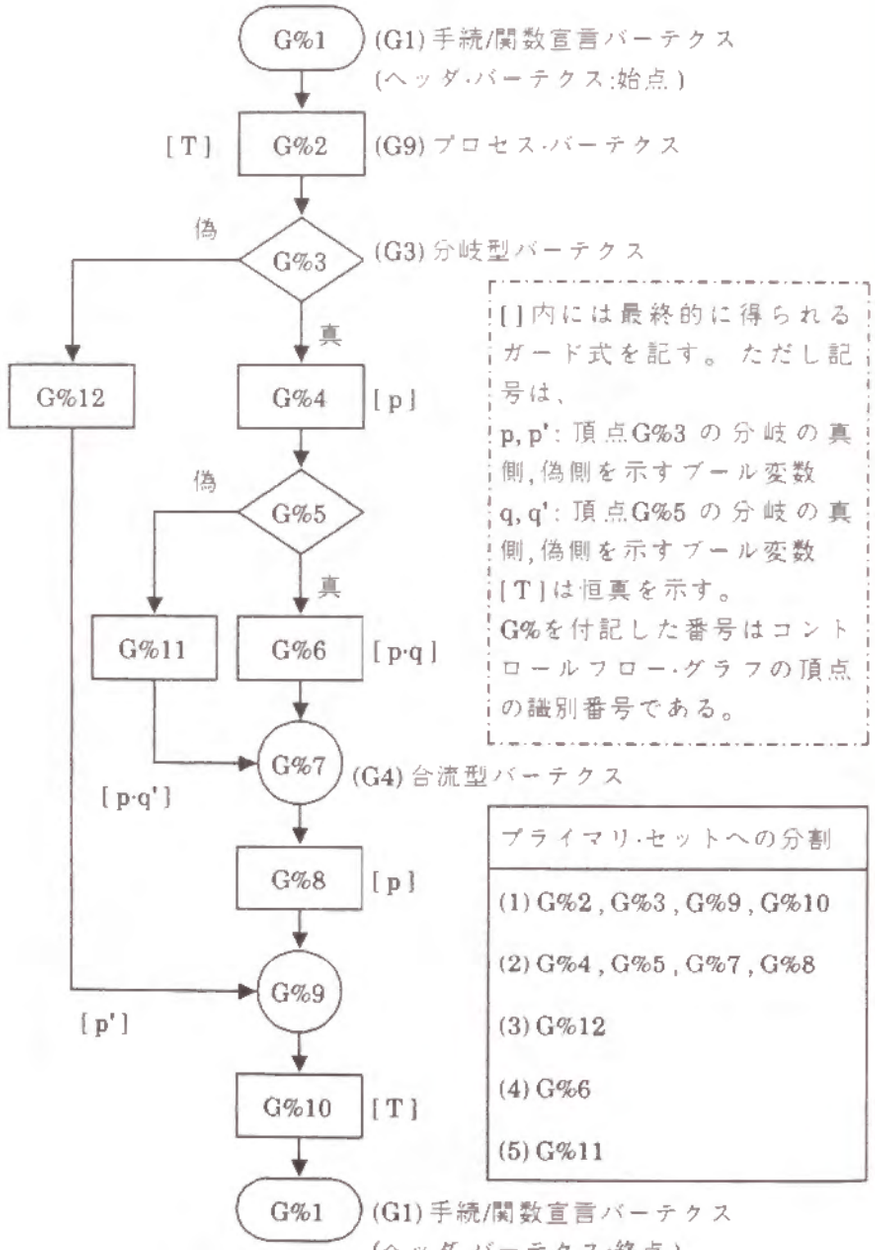


図4.18 コントロールフロー・グラフとガード式およびプライマリ・セットの例

(ア) PSTP: 当該プライマリ・セットに属するコントロールフロー・グラフの頂点を指すポインタのリストの先頭

(イ) INFLPRED: 当該プライマリ・セットに直接影響を及ぼす直近の(G3)分岐型バーテクスを指すポインタおよび真側/偽側の区別

(ウ) BELGLOOP: 当該プライマリ・セットが属するループの(G10)ループ情報記述バーテクス

この(イ)の情報からコントロールフロー・グラフのある頂点のガード式を求めることができる。これを得るアルゴリズムは、直接影響を及ぼす直近の(G3)分岐型バーテクスを求める方式とともに、5.4節において述べる。

4.3.5 D行列

D行列(Dependence Matrices)は、3.2.4項で述べた依存グラフを隣接行列の形式で保持するものである。図4.19に簡単な依存グラフの例とそれと等価な隣接行列を示す。この図からわかるとおり隣接行列では、依存グラフの任意の2頂点間の有向辺を1要素に記憶する。通常隣接行列の各要素には、当該2頂点間に何らかの依存があ

(a) 中間コードのノードを単位とした簡単な依存グラフの例

列構成子の並び
(有向辺の終点側)

行構成子の並び (有向辺の始点側)	%1	%2	%3
%1	0	1	0
%2	1	0	0
%3	0	1	0

(b) 上記依存グラフの隣接行列表現

図4.19 依存グラフとその隣接行列表現の例

り、有向辺が存在するとき"1"を、逆に何の依存も存在せず有向辺が存在しないときには"0"を保持する。そして、隣接行列でも依存グラフの場合にはその性質上、隣接行列の全要素が"1"となる(自分自身を含めすべての2頂点間に有向辺が存在する)ことは少ない。むしろ、大部分は"0"が占める疎行列となることが多い。従って、D行列のデータ構造はこの点を考慮して設計される必要がある。

さらに、V-Pascalでは、演算レベルでの細かなかつ徹底的な部分ベクトル化(ベクトル化を促進させる実行順序の入れ換えを含む)を行うため、中間コードを単位とした依存関係(3.2節参照)を解析する。その解析結果の各種依存を登録するD行列は、当然中間コードを単位としたすなわち中間コードと1対1に対応する頂点からなる依存グラフ(図4.19(a)参照)を表現できる必要がある。D行列のデータ構造の設計方針を以下にまとめる。

(A) 中間コードを単位とした依存グラフを、非零要素の少ない疎行列である隣接行列として適切に表現できること。

(B) 表現した依存グラフの有向辺を矢の向きにも、またその逆方向にも容易にたどれる物理構造とすること。

(C) ベクトル化を促進させる実行順序の入れ換えに相当する行交換ならびに列交換が容易にできること。

(D) 表現した依存グラフの有向辺が輪を形成する強連結部(以下縮退部と呼ぶ)を部分小行列として切り放せ、階層的に表現できること。

(E) その縮退部を表す行(列)構成子は縮退部でない通常の行(列)構成子とまったく同等に扱え、上記(C)の行交換ならびに列交換が容易にできること。

(F) 3.2節で述べた各種依存関係の類別情報を各有向辺ごとに保持できること。

上記(A)より、まず隣接行列としてのD行列の行および列は、1対1に対応する中間コードのノードを指すポインタを持つセルの双方向リニア・リストで形成する。表現したい依存グラフの頂点数に上限を定められないためリスト表現を用いている。しかも、前記(C)の要件を満たすため双方向とした(後の図4.20参照)。ただし図4.19からわかるとおり、行構成子および列構成子は、隣接行列では単に行構成子が有向辺の始点に対応し、列構成子が有向辺の終点に対応する違いだけであり、概念的にはともに依存グラフの1頂点を表す。従って、D行列の行構成子および列構成子は、物理的には融合し単一の行列構成セルとして実現している。表4.13に行列構成セルの

持つフィールドを示す。LASTLINKフィールドとNEXTLINKフィールドとにより双方向リストが形成される。この表からわかるとおり、行列構成セルは前記(C)および(D)を満たすために、縮退部をその他の部分と同等に扱えるようNKINDフィール

表4.13 D行列の行列構成セルの持つフィールド

フィールド名	意味
IDN	当該行列構成セルの識別番号
LASTLINK	前の行列構成セルを指す。なければNIL
NEXTLINK	次の行列構成セルを指す。なければNIL
ROWLINK	当該行の最初の非零要素セルを指す。なければNIL
COLLINK	当該列の最初の非零要素セルを指す。なければNIL
CRROWLNK	当該行の最初の制御依存非零要素セルを指す
CRCOLLNK	当該列の最初の制御依存非零要素セルを指す
NKIND	縮退部(REDUCED)か、あるループの先頭(LOOPTOP)か、それ以外(SINGLE)かの区別(NKINDVAR型)
REDLINK	縮退部のとき縮退部を構成する部分小行列の最初の行列構成セルを指す
INTERPT	縮退部でないとき対応する中間コードノードを指す
REDFLAG	なんらかの縮退部中に含まれるか否か
SLOOPLVL	スカラ実行すべき最深のループレベル
CONTLINK	ループの先頭(ループビギン・ノードに対応)するとき、そのループ内の最初の行列構成セルを指す

表4.14 D行列の非零要素セルの持つフィールド

フィールド名	意味
ELMNUM	当該非零要素セルの識別番号
ROWPT	行構成子としての行列構成セルを指す
COLPT	列構成子としての行列構成セルを指す
ROWLINK	同一行の次の非零要素セルを指す。なければNIL
COLLINK	同一列の次の非零要素セルを指す。なければNIL
RLSTLNK	同一行の前の非零要素セルを指す。なければNIL
CLSTLNK	同一列の前の非零要素セルを指す。なければNIL
CRROWLNK	同一行の次の制御依存非零要素セルを指す。なければNIL
CRCOLLNK	同一列の次の制御依存非零要素セルを指す。なければNIL
CRFLG	制御依存かどうかを示すフラグ
LOOPS	各ループレベルごとのデータ依存の種別

ドをタグ・フィールドとする可変レコード(DMRXRCNODE型)で実現している。縮退部となった部分小行列は、縮退部を表す行列構成セルのREDLINKフィールドにリンクされる(後の図4.21参照)。

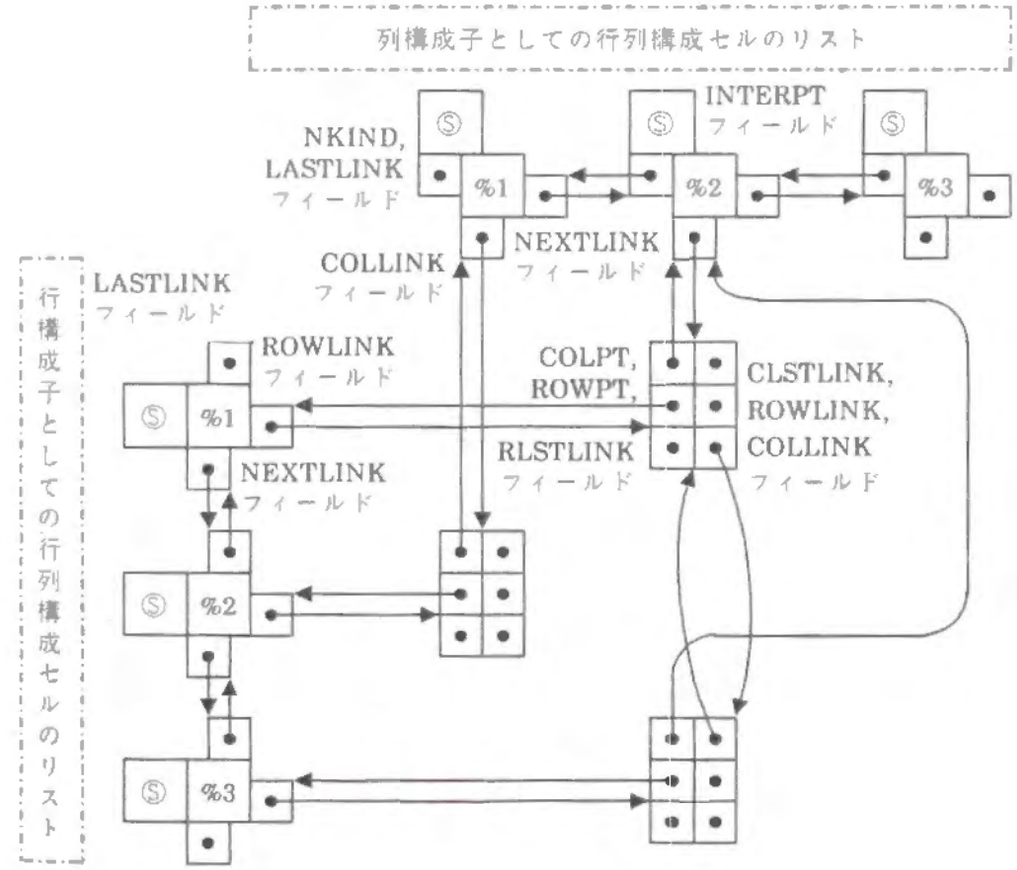
なお、V-Pascalの現バージョンでは、ベクトル化対象多重forループごとに、D行列を作成する。従って、あるD行列の行列構成セルは、当該ベクトル化対象多重forループ内の全中間コードノードに対して用意する。

行列要素に関しては前記(A)を考慮し、D行列では通常疎行列に用いられるデータ構造を採用した。すなわち、有向辺が存在する非零要素のみを、行方向ならびに列方向にリンクする。このリンクも、前記(B)ならびに(C)の要件を満たすため双方向リニア・リストで形成する。非零要素セル(DMRXELMNODE型)の持つフィールドを表4.14にまとめる。非零要素セル間の行方向および列方向の双方向リニア・リストは、それぞれROWLINKフィールドとRLSTLINKフィールドおよびCOLLINKフィールドとRLSTLINKフィールドにより形成する。

以上のリンク構造をまとめて例示するため、図4.20に図4.19の依存グラフを表現したD行列を示す。また、この例に部分ベクトル化を行うD行列操作(依存グラフで実行順序と逆向きのベクトル化に不適な依存を検出し、可能ならば実行順序を入れ換えベクトル化に適とさせ、不可能ならば強連結成分を抽出し1縮退部としてまとめる操作をいう。第6章で詳述)を施した後のD行列を図4.21に示す。図4.21から縮退部を部分小行列として切りわけて表現できていることがわかる。そして、1縮退部を示す行列構成セルは、通常の行/列を示す行列構成セルと区別なく、まったく同様に行/列交換の対象とできることがわかる。すなわち、前記(E)の要件を満たしている。もちろん、縮退部を示す行列構成セルに関し行/列交換を行っても、リンク先の実際の縮退部を表す部分小行列に対しては、何の操作も不要である。

非零要素には、前記(F)の要件を満たすため表4.14に示すCRFLGフィールドおよびLOOPSフィールドにより、依存関係の類別情報を保持できる構造とした。制御依存の場合には、CRFLGフィールドが真となり、データ依存の場合には、LOOPSフィールドが各ループレベルごとにどのような種類のデータ依存が存在するかを表現する。先述のとおり現在のV-Pascalでは、4.3.2項で述べた形式の中間コードを単位とする依存グラフを表現するので、制御依存は始点が分岐型ノードあるいはループビギン・ノードであるのに対し、データ依存は始点は(終点も)スカラ処理記述ノードあるいはループビギン・ノードである。従って、ループビギン・ノードに関する場合

を除き、D行列のある非零要素セルが、すなわち依存グラフの1有向辺が、制御依存とデータ依存ともに併せ持つことはない。つまり、ループビギン・ノードを除き、制御依存とデータ依存両者の存在は互いに排他的である。また、制御依存関係は始点がある一つの分岐型ノードあるいはループビギン・ノードに対応することから、依存グラフの1有向辺が、複数の制御依存関係を表すことはない。同様に、ある中間コードが同一の変数を複数回定義または引用し得ないことから、D行列のある非零要素セルは、その始点および終点に対応する1対のデータ参照に起因するデータ依存のみ



(注)INTERPTフィールドは、実際には対応する中間コードノードを指すポインタであるが、ここでは中間コードの識別番号を記して表記。
NKINDフィールドのⓈは"SINGLE"の意味を表記。

図4.20 例によるD行列を構成する各セルのリンク構造
(解説のため行列構成セルは行構成子/列構成子として別に重複して記述)

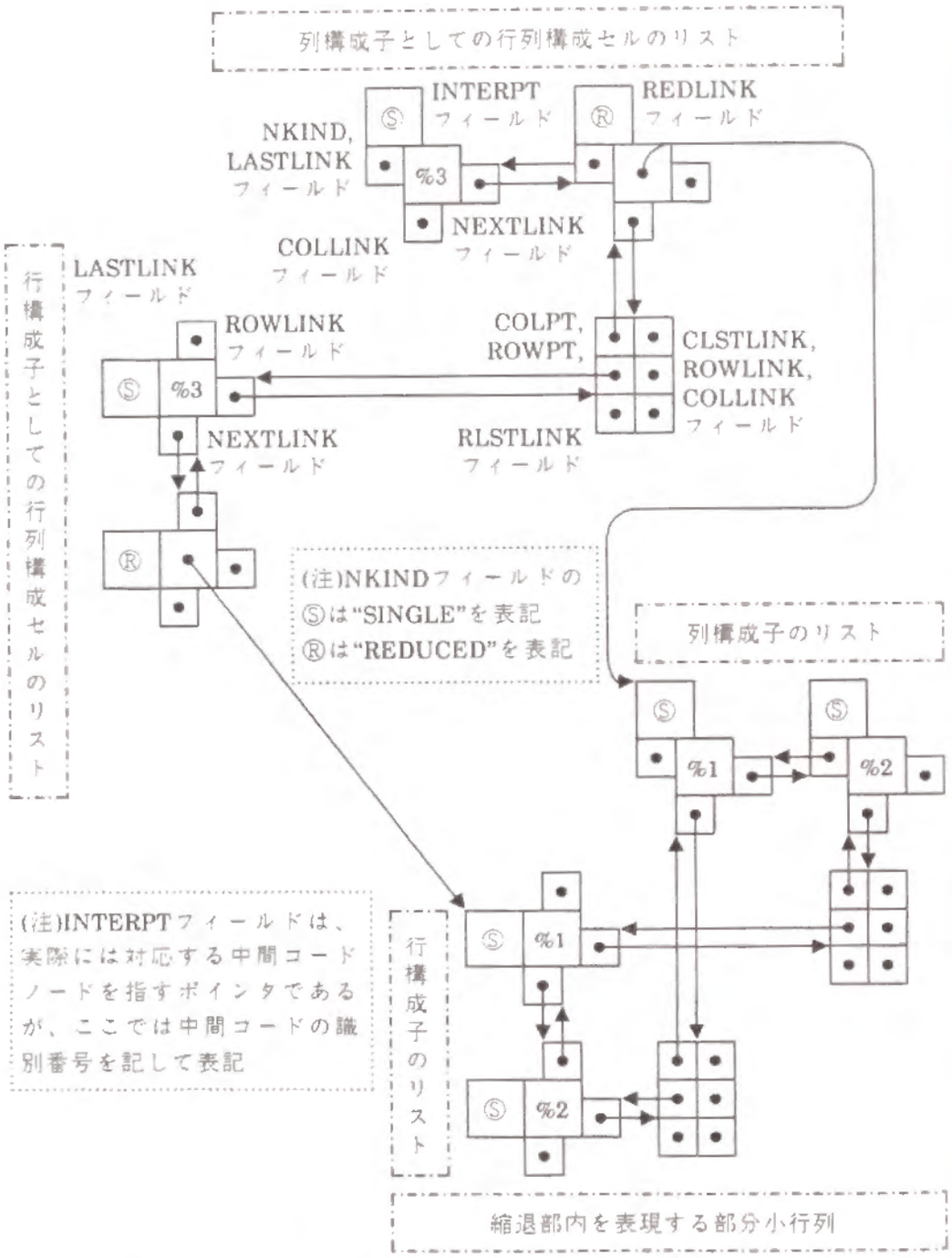


図4.21 D行列操作後のD行列

(解説のため行列構成セルは行構成子/列構成子として重複して記述)

を表現する。ただし、1対のデータ参照に起因するデータ依存でも、ループ選択機能(3.2.2項で既述)のためには、ベクトル化対象多重ループの各ループごとに独立に依存の種類を表4.15の細分類として調べなければならない。そして、各ループごとにこれらのうちの複数の種類が存在し得る。従って、このLOOPSフィールドは表4.15に示すデータ依存の細分類の集合を要素とするループレベル数の1次元配列として実現した。同表において、ダミーの依存とは配列化により消去される依存(詳細は5.2節参照)を意味する。引用-定義の場合についてのみループ独立型ダミー依存があり得る。その他のループ運搬型ダミー依存では、2回運搬依存、多回運搬依存を区別する必要はない。2回運搬依存、多回運搬依存を区別する目的は同表からわかるとおり、定義-引用の場合のみである。これは、定義-引用のループ2回運搬依存は、その性質からループの繰り返しに伴う回帰的な演算を形成するのに必須の依存である。従って、このタイプの依存は3.3.2項で述べた回帰的な演算をベクトル実行する数種のベクトルマクロ演算の抽出のキーとなるからである。いいかえれば、D行列中のtrue1型の依存を探索することにより、ベクトルマクロ演算に変換できる部分を容易に見つけ出すことができる。また、outmとroutmとの違いは、outmが当該有向辺の向きの依存を引き起こすことを示すのに対し、routmは有向辺と逆向きの依存を引き起こすことを示す。これらの区別は、3.2.2項で述べたループ交換を含むループ選択機能で必要となる。しかし、配列化されるダミーの依存についてはこれらの区別は必要ない。詳細は、5.2節および5.3節において解析アルゴリズムとともに述べる。

表4.15 D行列の非零要素セルに登録されるデータ依存の細分類

		定義-引用 (フロー依存)	引用-定義 (逆依存)	定義-定義 (出力依存)
ループ独立		true0	anti0(d)	out0
ループ運搬	2回運搬	true1	antim	(r)outm
	多回運搬	truem		
	ダミー	truemd	antimd	outmd

(注)類別の詳細は3.2節参照。ただし、ダミーは配列化により消去される依存。

4.4 構文/意味解析

V-Pascal Version 1 の構文/意味解析部は、先述のとおり Pascal-P4 処理系ならびに Pascal 8000 処理系[31]~[35]を参考にして作成されている。その解析アルゴリズムは、Pascal の構文に基づく**再帰降下型構文解析法**(recursive descent parsing)[36]によっている。通常再帰降下型構文解析法では、**構文木**(syntactic tree)を生成することなく**非終端記号**(non-terminal symbol)を解析する手続を**構文規則**(syntax)に従って順次呼び出すことにより解析を行う。これに対し、V-Pascal Version 1 の構文/意味解析部は、与えられたソース-プログラムを等価な中間コード列に翻訳することを目的とする。それゆえに、非終端記号を解析するモジュールは、解析と同時にそれぞれが解析した部分ごとに等価な中間コード列を生成し、それまでに作成された中間コード列に順につなぐ。それゆえ、各非終端記号解析モジュールは関数とし、必ずそれまでに作成された中間コード列の最後のノードを指すポインタ値を返すこととした。解析と同時に目的コードを生成する1パスのコンパイラとは、この点で異なる。

Pascal の構文/意味は、1パスのコンパイラで目的コードを生成しやすいように定義されている。例えば、識別子は必ず宣言した後利用する。それゆえに、付録の構文図のブロック(Block)を解析する場合、ラベル宣言、定数宣言、型宣言、変数宣言、手続宣言、関数宣言の各宣言部をまず解析した時点で、当該ブロックの実行部(ボディ部)を解析するのに必要な識別子表、型表、定数表等のテーブル群が完成する。V-Pascal Version 1 の構文/意味解析部も、この性質を利用し中間コード列に翻訳するのに、1度ソース-プログラムを走査するだけである。

4.5 Alias 解析

ソース-プログラム中に現れる異なる変数名等が同一の記憶番地を割り付けられるときそれらの複数個の変数名等が互いに**別名**(alias)であるという[36]。この互いに別名である複数個の変数を集合(alias 集合)として求めるのが Alias 解析である。

この別名は、変数の参照関係を基に行う各種最適化ならびにベクトル化に必須の変数の参照関係により生ずるデータ依存解析に大きく関係する。例えば、実際には互いに別名である複数個の変数(X と Y とする)を、別名と解析できなかった場合を

考える。変数 X と変数 Y は実際には同じ記憶番地が割り付けられているので、変数 X にある値を代入した直後に、変数 Y に別の値を代入しても別名と解析できていないので、先の代入を無用な定義として最適化(4.8節で詳述)できない。また、このような場合両変数への代入間ではデータ参照関係により生ずる(出力)依存が存在するのに検出されない。最悪の場合には誤ったベクトル化を行う可能性が生ずる。逆に実際にはまったく別の変数(V と W とする)を、誤って互いに別名と解析した場合を考える。この場合には先と同様に両変数への代入が連続したとき、誤って先の代入が無用と判定され削除される。つまり、誤って最適化されることがある。それに対し、ベクトル化に関しては、不必要なデータ依存が登録され、最悪の場合にはベクトル化が阻害されるかもしれないが、意味を誤ることはない。従って、V-Pascal Version 1 の Alias 解析においては、解析時間を考慮しできるだけ実際の別名を解析し、解析が不可能あるいは困難な場合には疑わしいものは別名とする方針で解析するアルゴリズムとした。すなわち、疑わしいものも含めた大きめの alias 集合を求めている。それゆえ、最適化においては上記の誤った最適化を行うことのないよう真に別名であると確定でないときには、無用な定義の削除を行わないよう排除させることとした。

Pascal では以下の場合において別名関係が生じる。それぞれの場合の例を図4.22に示す。

(A) 番地呼びの引数の受渡しによる別名

(B) ポインタ変数が指す領域の別名

(C) 可変レコード中のフィールド名どうしの別名

(D) 構造を持つ型の変数の一括参照とその中の一部分の参照

図4.22の(A)では、番地呼びの実引数と仮引数間での別名の例を示したが、2個の番地呼びの仮引数(P と Q とする)に同一の実引数を引き渡す場合等では、2個の仮引数(P と Q)どうしが別名となる。

図4.22の(B)では、p1↑ と p2↑ とが別名である。p1 と p2 とは別名ではない。

図4.22の(C)で vr1.i と vr1.r とが別名である。これは、記憶領域の浪費を避ける目的から通常の処理系では、可変レコード中の同時には存在しないフィールドについては同一の記憶領域を重複して割り付けるため生じる。

Pascal では構造を持つ型の変数の一括参照(図4.22の vr1)が可能である。この一括参照は、部分的な参照すべてを包括した意味となり、部分的な参照それぞれと一

種の別名となる。これが、(D)の別名である。

V-Pascal Version 1 のAlias解析は先に述べた疑わしいものも含めた大きめの alias 集合を求める基本方針にのっとり、以下のとおりのアルゴリズムとした。

[Alias解析アルゴリズム]

1) 変数記述(付録 Pascal 構文図の Variable 参照)中に1ヵ所以上ポインタが指す領域(↑)が現れる場合には、その変数記述子ならびに変数記述限定子の最後で指し示される型の領域を参照するとみなす。

```
program    ExampleOfAlias(output);
  var G1:integer;
        ⋮
  procedure sub1(var P1:integer);
    begin{sub1}
        ⋮
    end{sub1};

  begin{main}
        ⋮
    sub1(G1);    {変数 G1 と P1 とが alias}
        ⋮
end{main}.
```

(A) 番地呼びの引数の受渡しによる別名

```
var p1,p2: ↑integer;
        ⋮
begin
        ⋮
  p1:=p2; {変数 p1↑ と p2↑ とが alias}
        ⋮
end
```

(B) ポインタ変数が指す領域の別名

```
var vr1:   record
             case tf:integer of
               0:(i:integer);
               1:(r:real);
           end; {変数 vr1 と vr1.i と vr1.r とが alias}
```

(C) 可変レコード中のフィールド名どうしおよび
(D) 構造を持つ型の変数の一括参照とその中の一部分の参照

図4.22 言語 Pascal における別名の例

2) 構造を持つ型の変数(領域)の一部分の参照(レコード型のフィールドの参照および配列型の要素の参照)は、その変数(領域)の一括参照とみなす。
3) 番地呼びの引数の受渡しによる別名は、別に求めた Alias 表(4.2 節参照)から判定する。 □

上記のアルゴリズムの1)では、同一の型の領域を指すポインタを含む変数記述はすべて別名となる。しかし、厳密には図4.22の(B)に示したようにポインタ間での代入がなければ、同一の領域を指すことはなく別名とはならない。現時点では解析を簡略化するため、このような代入があることを確かめる過程を省略している。また、上記のアルゴリズムの2)では、あるレコード型の別個のフィールドの参照どうしをも別名としてしまう。

次に番地呼びの引数の受渡しによる alias 集合を Alias 表にまとめるアルゴリズムについて簡単に述べる。

[番地呼びの引数の受渡しによる Alias 解析アルゴリズム]

1) 手続/関数相互呼出し表作成部(図 4.1 参照)が、フェーズ1で作成された中間コード列をすべて走査し、手続/関数呼び出しノードを抽出し手続/関数相互呼出し表を作成する(4.3.2項の手続/関数宣言ノードの解説参照)。
2) 作成された手続/関数相互呼出し表をたどり、番地呼びの仮引数(Yとする)とそれに引き渡される実引数(Xとする)とのすべての組(X, Y)を alias 集合として Alias 表に登録する。
3) 上で登録された alias の組の推移的閉包をとる。すなわち、(X, Y)ならびに(Y, Z)が登録されているとき、(X, Z)も別名とみなす。このとき、反射律も成り立つものとする。つまり、(X, Y, Z)を alias 集合として Alias 表に登録する。 □

上記のアルゴリズムは基本的には文献[36]および[37]によるものである。ただし、これらの文献のアルゴリズムは、上記の3)のように反射律まで仮定することはない。従って、上記のアルゴリズムでは、別名でない組み合せも別名と解析することがある。例えば、(X, Y)ならびに(X, Z)が別名であるとき、(Y, Z)の関係がその例である。この場合、上記の3)で、(X, Y, Z)を alias 集合として Alias 表に登録するため、(Y, Z)も別名とみなされるのである。しかし、このように反射律まで仮定す

ることにより、変数Xが2つのalias集合にまたがって存在することがなくなり、後のデータフロー解析の処理が簡略化される。alias集合として処理することは具体的には次のように実現している。各alias集合に対して固有の(処理系が使用する)識別子を与え、4.3.2項で述べた変数記述子に対する参照は、(a)もしその変数記述子がなんらかのalias集合に属しているなら、当該alias集合に与えられた先の識別子が参照されたとみなし、(b)そうでなければその変数記述子が示す本来の識別子が参照されたとしデータフロー解析等を行う。

文献[42]および[43]では、より厳密なAlias解析アルゴリズムについて紹介されている。V-Pascal Version 1.5では、より精密なAlias解析法が実現されている。

4.6 コントロールフロー解析

V-Pascal Version 1のコントロールフロー解析は、以下の処理を行う。

(1)中間コードから4.3.3項で述べた形式のコントロールフロー･グラフを作成する。

(2)コントロールフロー･グラフの各頂点間の**支配関係**、すなわち各頂点の**支配頂点**(dominator)[40]、**直接支配頂点**(immediate dominator)[40]を求める。

(3)各有向辺の向きを逆にした逆有向グラフ(対称グラフともいう)を考える。元の分流型/合流型頂点はそれぞれ逆有向グラフの合流型/分流型頂点となる。この逆有向グラフにおいて、同じく各頂点間の支配関係、支配頂点、直接支配頂点を求める。以下、これらをそれぞれ**逆支配関係**、**逆支配頂点**、**直接逆支配頂点**と呼ぶことにする。また上記(2)およびこの(3)の解析を合わせ広く**支配関係解析**と呼ぶことにする。

(4)コントロールフロー･グラフからループを検出する。

(1)の目的は4.3.3項で述べたとおりである。支配関係解析は5.3節で述べるV-Pascal Version 1の制御依存解析のために必要となる。上記の定義のうち、直接逆支配頂点の定義は文献[40]にはなく、逆支配頂点を求める際の副次的な概念となっている。これに対し、5.3節制御依存解析で述べる手法では、直接支配頂点および直接逆支配頂点が重要な概念となる。(4)のループ検出は、いうまでもなくベクトル化対象ループの検出等に関係する重要な処理である。以下、順次各処理について概説する。

4.6.1 支配関係解析

支配頂点、逆支配頂点ならびに直接支配頂点の定義[40]は以下のとおり。

[定義]

コントロールフロー･グラフ上の頂点vと頂点wにおいて、グラフの根である始点から頂点wに達するどの経路(path)も必ず頂点vを通るとき、頂点vは頂点wを**支配する**、あるいは頂点vは頂点wの**支配頂点**であると定義する。同様に、コントロールフロー･グラフの逆有向グラフにおいて、頂点vが頂点wを支配するとき、頂点vは頂点wを**逆支配する**、あるいは頂点vは頂点wの**逆支配頂点**であると定義する。さらに、頂点vの支配頂点の集合Dの要素である頂点d(頂点vを除く)が、頂点v、頂点dを除く集合Dの他のすべての要素の頂点に支配されるとき、頂点dは頂点vの**直接支配頂点**という。直観的には、頂点vの直接支配頂点とは、頂点vの支配頂点の中で最も頂点vに近いものである。 □

表4.16 図4.23を例とした支配関係および逆支配関係(本表中ではG%は省略)

頂点	支配関係		逆支配関係	
	支配頂点	直接支配頂点	逆支配頂点	逆直接支配頂点
2	1(始点)	1(始点)	3,4,5,6,7,8,1(終点)	3
3	1(始点),2	2	4,5,6,7,8,1(終点)	4
4	1(始点),2,3	3	5,6,7,8,1(終点)	5
5	1(始点),2,3,4	4	6,7,8,1(終点)	6
6	1(始点),2,3,4,5	5	7,8,1(終点)	7
7	1(始点),2,3,4,5,6	6	8,1(終点)	8
8	1(始点),2,3,4,5,6,7	7	1(終点)	1(終点)
9	1(始点),2,3,4,5,6,7	7	3,4,5,6,7,8,11,12,1(終点)	12
10	1(始点),2,3,4,5	5	3,4,5,6,7,8,11,12,1(終点)	12
11	1(始点),2,3,4,5,12	12	3,4,5,6,7,8,1(終点)	3
12	1(始点),2,3,4,5	5	3,4,5,6,7,8,11,1(終点)	11

表4.16は図4.23のコントロールフロー・グラフについて行なった支配関係解析例である。

直接支配頂点から支配頂点(ならびに直接逆支配頂点から逆支配頂点)を求めるため、V-Pascalでは文献[41]のアルゴリズムを用いている。以下にその概略アルゴリズムを示す。詳細は本論に関係しないので省略する。

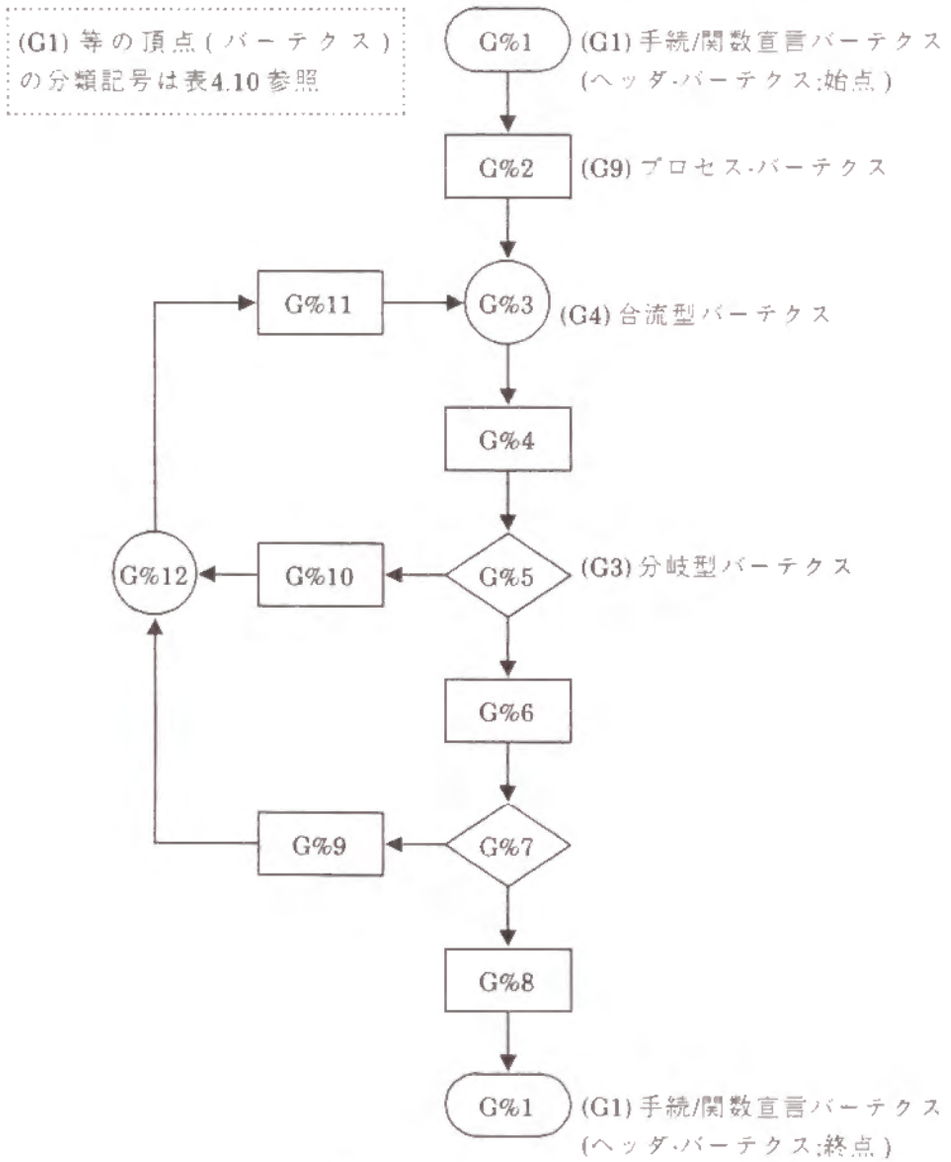


図4.23 簡単なコントロールフロー・グラフの例

[支配頂点を求めるアルゴリズム][41]

(1) コントロールフロー・グラフを深さ優先でなぞり各頂点に**深さ優先の順序づけ** (depth first ordering) を行う。同時になぞった有向辺に印をつけ、部分グラフである**深さ優先探索木** (depth first spanning tree) を得る (4.6.2項で詳述)。

(2) こうして得られた深さ優先探索木を親子関係の木として見たとき、各頂点の**半支配頂点** (semi-dominator) を求める。

(3) 半支配頂点から直接支配頂点を求める。

(4) 直接支配頂点の推移的閉包を求め支配頂点とする。具体的には、ある頂点の直接支配頂点から当該頂点への有向辺をつけた**支配関係の木** (dominator tree) を作成し、その木をたどることにより簡単に求められる。 □

なお、ここで半支配頂点とは、厳密な定義は文献[40]および[41]にゆずるが、直観的にはある頂点の先祖の中で最も若い深さ優先の順序づけ番号を持つ頂点である。この半支配頂点を求める高速な手法が文献[40]および[41]に示されており、その手法がV-Pascal中にも実現されている。

4.6.2 ループ検出

コントロールフロー・グラフを走査し、以下の性質を持つ頂点群をループとして検出する[36],[37]。

(1) すべての頂点が強連結である。

(2) 1個以上の入口がある。すなわち、当該強連結成分に到達するなんらかの経路が存在する。

唯一の入口頂点を持つループを**自然ループ** (natural loop) あるいは**単一入口ループ** (single entry loop) と呼び、複数の入口頂点を持つループを**複数入口ループ** (multi-entry loop) と呼ぶ[36],[37]。さらに、自然ループはV-Pascal Version 1のベクトル化対象ループとそれ以外に分類される。現在のベクトル化対象ループは内部に包含するループはすべてfor文によるループであり、ループ外への飛び出しおよびループ内への飛び込みを持たないfor文によるループである。

また、ループのうち、他のループを含まないループを**内部ループ**と呼ぶ[36],[37]。ループ間の包含関係の一般的な定義は次のとおりである。

[定義]

ループ L_1 に含まれる全頂点がループ L_2 に含まれるとき、ループ L_2 がループ L_1 を含むといい、$L_1 \in L_2$ と表記する。$L_1 \in L_2$ かつ $L_2 \in L_1$ ならば $L_1 = L_2$ である。 □

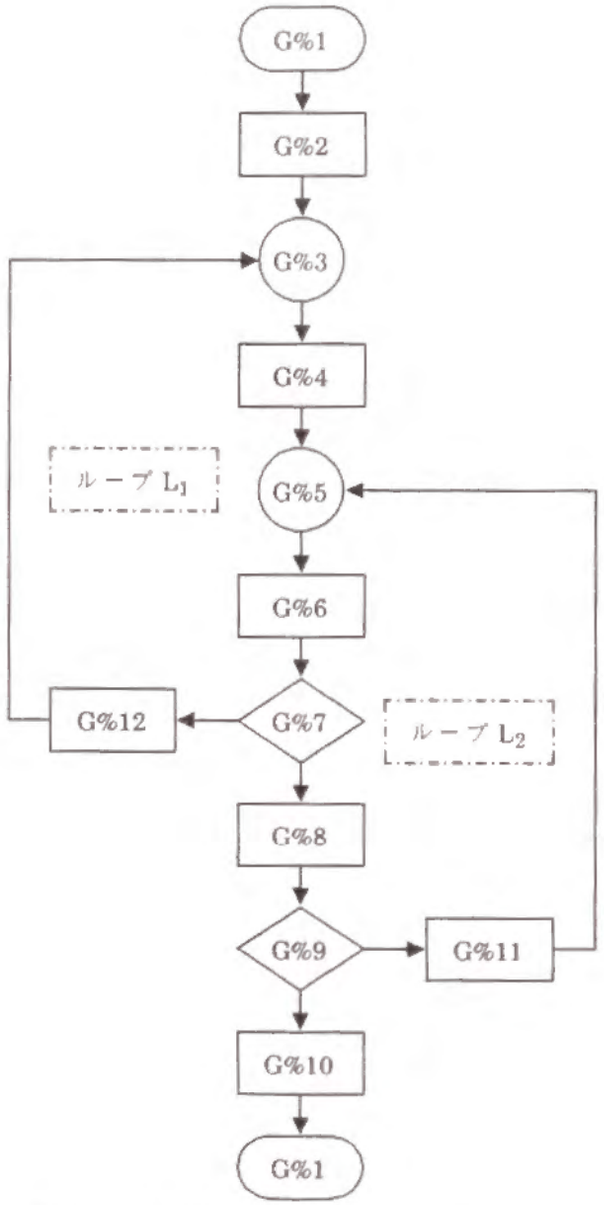


図4.24 部分的に重なりのあるループを持つコントロールフロー・グラフの例

この一般的な定義によれば、図4.24のような部分的な重なりを持つループ間には包含関係がない。しかし、このような場合にも包含関係があるものと定義する方が都合がよいことがある。例えば、ループ内不変式のループ外への移動の最適化では、ループ L_2 に含まれるループ内不変式をループ L_2 の入口頂点の直前(同図 G%4)へ移動させるが、そこはループ L_1 内であるため、再度移動できる可能性が生じる。すなわち、このような場合ループ L_2 は、あたかもループ L_1 に包含されているとみなせる。

従って、V-Pascal では部分的に重なりのあるループ間では、深さ優先探索木において浅いレベルの入口頂点を持つループが、深いレベルの入口頂点を持つループを含むと定義し、一般的な包含関係と同様に扱うこととした。

以上のループおよびその包含関係の定義に基づきループを検出し、包含関係をも合わせて登録するアルゴリズムについて述べる。検出された1ループについては、コントロールフロー・グラフの(10)ループ情報記述バーテクス(表4.10参照)を用意し、その各フィールドに上記の包含関係(表4.12 LLINK フィールド等)をはじめとして、当該ループに関する種々の情報を記憶させることとした。このアルゴリズムは、文献[40]に示されたコントロールフロー・グラフの有向辺の類別(表4.17の4種)情報をもとにループを検出し、包含関係と同時に登録するものである。このコントロールフロー・グラフの有向辺の類別情報は、4.6.1 項で述べた支配関係解析のために導入されたものであるが、ここではループ検出にも利用できることを示す。

表4.17 コントロールフロー・グラフの有向辺の類別[40]

種類	意味
Tree	深さ優先探索木を構成する辺。
Frond	深さ優先探索木において子孫である頂点から先祖である頂点へ向かう辺。Frond の長さはその2頂点間の深さ優先探索木における深さレベルの差である。
Reverse frond	深さ優先探索木において先祖である頂点から子孫である頂点へ向かう辺。Tree でないことからこの場合、2頂点間の深さ優先探索木における深さレベルの差は2以上となる。
Crosslink	深さ優先探索木において互いに先祖/子孫の関係にない2頂点間を結ぶ辺。すなわち、深さ優先探索木において異なる枝を結ぶ辺。

この表4.17の有向辺の類別は、上記の定義から明らかなとおり深さ優先探索木生成と同時に行うのが自然であり効率的である。V-Pascal では次の再帰的なアルゴリズムをそのままインプリメントしている。この再帰的なアルゴリズムの最初の呼び出しは、Frond リストを空に初期化した後、最初になぞる頂点としてコントロールフロー・グラフの始点である現在解析中の手続/関数宣言バーテクス(ヘッダ・バーテクス)を与えることにより行う。

[有向辺の類別ならびに深さ優先探索木生成アルゴリズム]

(1) 与えられた頂点(vとする)の完全に探索済みであることを示すフラグ Tr(v) を偽とする。頂点 v から出る有向辺を一つ選ぶ。その有向辺の終点の頂点を w とする。

(2) 頂点 w がまだ深さ優先順序づけの番号(以下 Dfo(w) と表記)を付けられていなければ

(2-1) 頂点 w に深さ優先順序づけの番号および深さ優先探索木の深さレベルを付ける。有向辺 v → w を Tree と分類する。

頂点 w が既に深さ優先順序づけの番号を付けられており、かつその Dfo(w) について Dfo(w) > Dfo(v) のとき

(2-2) 有向辺 v → w を Reverse frond と分類する。

頂点 w が既に深さ優先順序づけの番号を付けられており、かつその Dfo(w) について Dfo(w) < Dfo(v) のとき

(2-3) フラグ Tr(w) が真であれば有向辺 v → w を Crosslink と分類する。

(2-4) フラグ Tr(w) が偽であれば有向辺 v → w を Frond と分類し、Frond リストに加える。

(3) 頂点 w に対してこのアルゴリズムを再帰的に適用する。

(4) もし頂点 v から出るまだたどっていない有向辺があれば、一つ選ぶ。その有向辺の終点の頂点を w とする。ステップ(2)へ戻る。

(5) 完全に探索済みであることを示すフラグ Tr(v) を真とする。 □

この有向辺の類別情報はコントロールフロー・グラフの ECLSF1 フィールド(表4.11参照)および2本の有向辺が入る(G4)合流型バーテクスでは、その ECLSF2 フィールド(表4.12参照)に記録される。

上記の有向辺類別アルゴリズムを用いるとループ検出アルゴリズムは、次のように実現できる。

[ループ検出アルゴリズム]

(1) 上記の有向辺類別を行う。生成された Frond リストに登録された各 Frond 型有向辺について、Frond の長さの短い順に以下の処理を行う。

(2) 選ばれた Frond 型有向辺の始点を頂点 v、終点を頂点 w とする。この有向辺 v → w によるループをまだ検出していなければ、頂点 v および頂点 w を新しいループ L を構成する頂点として登録し、以下の処理を新たな頂点がループ L の構成要素に追加されなくなるまで繰り返す。

(3) ループ L に含まれる頂点 y に入る各有向辺 x → y について、

有向辺 x → y が Tree あるいは Reverse frond であるとき

(3-1)「頂点 x の深さ優先探索木の深さレベル > 頂点 w の深さ優先探索木の深さレベル」ならば、頂点 x をループ L の構成要素に追加する。そしてさらに、頂点 x が既に他のループ L' に含まれている場合には、ループ L ⊃ ループ L' の包含関係をリンクする。

有向辺 x → y が Frond であるとき

(3-2) 有向辺 x → y によるループ(L' とする)を再帰的に検出する(ステップ(2)に戻る)。ループ L ⊃ ループ L' の包含関係をリンクし、ループ L' に含まれる全頂点をループ L の構成要素に追加する。

有向辺 x → y が Crosslink であるとき

(3-3) 頂点 w が頂点 x の先祖でなければ、頂点 y をループ L の入口頂点のリストに加える。

頂点 w が頂点 x の先祖であれば、頂点 w から頂点 x への経路上に別の Frond 型有向辺が存在するか調べる。存在する各 Frond 型有向辺によるループ(L' とする)を再帰的に検出する(ステップ(2)に戻る)。ループ L ⊃ ループ L' の包含関係をリンクする。頂点 x をループ L の構成要素に追加する。 □

なお、現在の V-Pascal では、上記の(3-2)の場合は図4.24の部分的な重なりを持つループであるので、もしループ L' が for 文によるループであれば、その開始/終了を示すループビギン・バーテクス/ループエンド・バーテクスを除去し、分岐型バーテクス

/合流型バーテクスによるループ表現に置き換えている。同時に中間コード上でも同様の処理を行う。これは、ループL'がループLを形成する飛び出しを含むことから、明らかに現バージョン(Version 1)のベクトル化対象ループからはずせるためである。

上記の(3-3)から、複数入口ループをも検出できるアルゴリズムとなっている。文献[44]では、複数入口ループを単一入口ループに解体し、単一入口ループとして検出する手法が述べられている。ただし、この手法はアセンブリ言語で記述されたプログラムを解析対象としており、ここで述べたアルゴリズムが中間表現を対象とする点で異なる。ループ不変式のループ外への移動等の最適化を考慮すると、単一入口ループへ解体することが望ましい。従って、中間表現において文献[44]と同様の機能を実現するアルゴリズムを開発する必要がある。

4.7 大域的データフロー解析

通常種々の最適化のために必要となる変数のデータの流れに関する解析が、大域的データフロー解析(global data flow analysis)である。例えば、図4.25において、変数Xに対する値の代入(定義:definition)が図中の3か所(①、②、③)のみであったとする。そして、変数Xに対しaliasとなる変数がないものとする。もし、図中のG%10の頂点で変数Xに対する値の引用があれば、定数の畳み込みの最適化により、その引用を定数値2に置き換えることができる。同様にG%9の頂点で変数Xに対する値の引用があれば、その引用を定数値3に置き換えることができる。すなわち、定数の畳み込みの最適化では通常ある変数に対する定義が複数生じ得るため、データの流れを解析しプログラム内のどの部分ではどの定義が有効かを正しく判断できなければならない。これは、共通部分式の削除等他の多くの最適化についても必要な解析である。

V-Pascal Version 1では文献[45]の考え方に基づき以下で述べるデータフロー集合と呼ばれる集合を使って、手続き(および関数)ごとにそれら全域にわたる大域的データフロー解析(以下単にデータフロー解析と記す)を行っている。これらの集合はコントロールフロー・グラフの各頂点ごとに定義される(4.3.3項の表4.11参照)。データフロー解析の単位は細かいほど、解析結果も詳細となる。しかし、解析のコストを考慮し通常コントロールフロー・グラフの頂点を単位とするのが通例である。従って、コントロールフロー・グラフのある頂点内において、複数回ある変数が定義

される場合には、当該頂点内の中間コードを単位とした詳細な解析がさらに必要となる。ただし、このコントロールフロー・グラフの1頂点内の解析のコストは、中間コードの数が十分少なく、また、頂点内での制御の流れが単純に1本であることから、1手続/関数全体を中間コードを単位として解析する場合と比較すると無視できるほど小さくすむ。データフロー解析の種々のアルゴリズムならびにそのコストに関

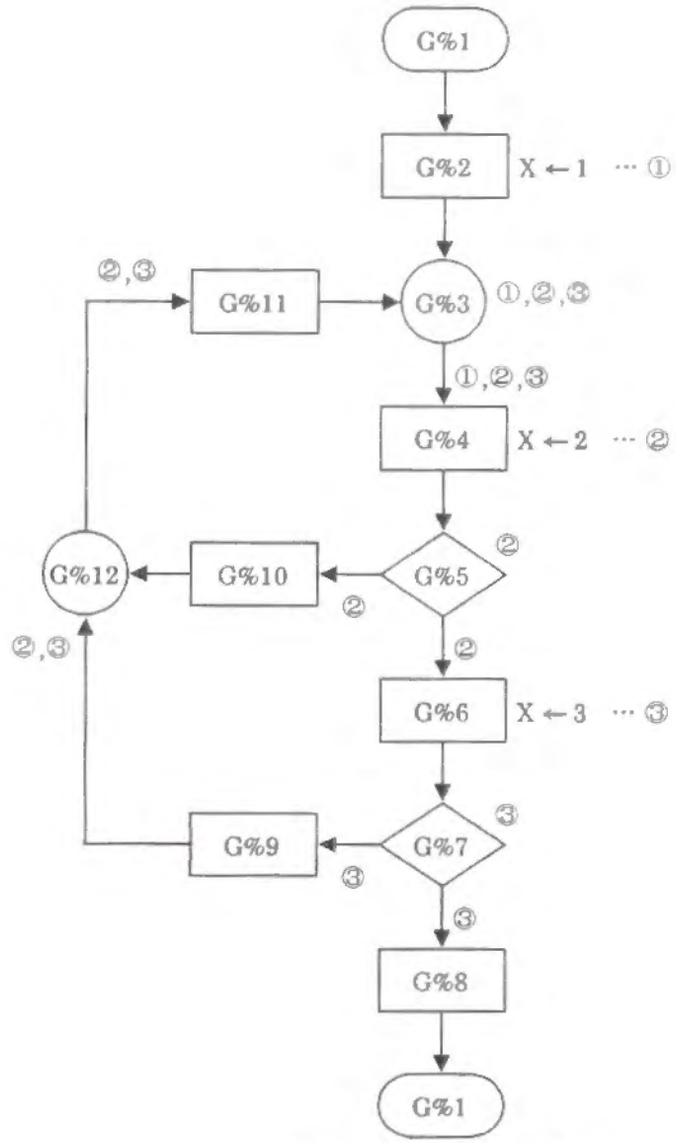


図4.25 到達する定義の例

する論文は文献[45]のほかにも数多くある。データフロー解析の詳細についてはそちらを参照のこと。

表4.18は定義を要素とするデータフロー集合4種、表4.19は変数を要素とするデータフロー集合4種である。表4.18の定義に関する集合、特にin集合を調べることにより、あるコントロールフロー・グラフの頂点に到達する(図4.25参照)、すなわち有効な定義を簡単に判定でき、先述の定数の畳み込み、共通部分式の削除等の最適化が容易に行える。また、表4.19 変数に関する集合のbin集合はいわゆる生きている変数(live variable)[36],[37]を示すものであり、生変数解析(live variable analysis)

表4.18 定義を要素とするデータフロー集合[45]

種類	意味
gen[S]	ある頂点Sで定義され、その頂点Sの最後に到達する定義の集合。すなわち、頂点Sの中で代入文等により、ある変数が一般には複数回定義され得るが、その中で最後の定義だけが、頂点Sの最後に到達する。これは、その最後に定義された値だけが、頂点S以降で有効であることに着目している。
kill[S]	gen[S]の要素である各定義と同じ変数を定義する定義の、プログラム全域にわたる集合から、gen[S]を除いた集合。これは、頂点Sで新しく変数が定義されるため、他の定義が無効となることを示す。
in[S]	ある頂点Sの入口に到達する定義の集合。reach[S](到達定義[36],[37])とも呼ばれる。これは、頂点Sが実行されるとき、有効な定義を示す。
out[S]	ある頂点Sの出口に到達する定義の集合。これは、頂点Sの実行後、有効な定義を示す。

表4.19 変数を要素とするデータフロー集合[45]

種類	意味
use[S]	ある頂点S内で引用が定義に先行する変数(中間項も一時変数として含め統一的に処理している。以下同じ)の集合。
def[S]	ある頂点S内で定義が引用に先行する変数の集合。
bin[S]	ある頂点Sの入口で保持していると思われる値を以後で引用する可能性がある変数の集合。live[S]とも呼ぶ。
bout[S]	ある頂点Sの出口で保持していると思われる値を以後で引用する可能性がある変数の集合。

[36],[37]の結果にほかならない。この情報はレジスタ割り付け時等に利用される。すなわち、生きていなくなった変数に割り付けられたレジスタを、生きていなくなった時点で解放することにより、適切なレジスタ割り付けが可能となる。

これらの集合は以下のデータフロー方程式(data-flow equations)[36],[37]を解くことにより求められる。

$$out[v] = (in[v] - kill[v]) \cup gen[v] \quad (1)$$

$$in[v] = \cup out[P] \quad (P:頂点vの直前の頂点) \quad (2)$$

$$bin[v] = (bout[v] - def[v]) \cup use[v] \quad (3)$$

$$bout[v] = \cup bin[P] \quad (P:頂点vの直後の頂点) \quad (4)$$

具体的には、次のように求める。

[データフロー解析概略アルゴリズム]

(1) コントロールフロー・グラフ中(G7)ループビギン・バーテクスおよび(G8)ループエンド・バーテクスで記述された全forループを(G3)分岐型バーテクスおよび(G4)合流型バーテクスで表現しなおす。

(2) コントロールフロー・グラフを深さ優先でなぞり、各頂点に深さ優先の順序づけ(depth first ordering)を行う。

(3) 各頂点ごとに、gen集合、kill集合、def集合、use集合を求める。他の集合を空集合とする(実際には、早く収束するよう、例えば、out集合にはgen集合をコピーしている)。

(4) 上記のデータフロー方程式の式(1)、式(2)については深さ優先順で、式(3)、式(4)についてはその逆順で、各頂点のどの集合にも変化が無くなるまで、繰り返しin集合、out集合、bin集合、bout集合を方程式のとおり計算する。これは、各集合が漸近的に方程式の解に近づくことを利用している。

(5) ステップ(1)において(G3)分岐型バーテクスおよび(G4)合流型バーテクスで表現しなおした全forループを再度(G7)ループビギン・バーテクスおよび(G8)ループエンド・バーテクスによる表現に戻す。

(6) 同時に(G3)分岐型バーテクスおよび(G4)合流型バーテクスそれぞれについて求められた各データフロー集合から、(G7)ループビギン・バーテクスおよび(G8)ループエンド・バーテクスの各データフロー集合を計算する。 □

上記のアルゴリズムのステップ(1)は、(G7)ループビギン・バーテクスおよび(G8)

ループエンド･バーテクスで記述された全 for ループを一般の (G3) 分岐型バーテクスおよび (G4) 合流型バーテクスによるループ表現に置き換え、上記ステップ(2) 以降統一的に処理する目的から行う。ただし、(G7) ループビギン･バーテクスおよび (G8) ループエンド･バーテクスは取り除かれず、ステップ(2) 以降では単なる (G9) プロセス･バーテクスとみなして処理される。データフロー解析が終了した時点で、すなわち上記ステップ(5) で元に復元される。このステップ(5) のとき、付加されていた (G3) 分岐型バーテクスおよび (G4) 合流型バーテクスは、コントロールフロー･グラフからは除去されるが、最適化等データフロー解析部が繰り返し呼び出されることを考慮し、再利用できるように (G7) ループビギン･バーテクスの DMYPRDCLC フィールドにリンクしておく。従って、このステップ(1) の処理において (G3) 分岐型バーテクスおよび (G4) 合流型バーテクスが新たに確保されるのは、ベクトル化等で元のプログラムの制御構造が大きく変化し、コントロールフロー解析部が呼び出され (図4.1 参照)、コントロールフロー･グラフが新たに作成された直後のデータフロー解析時のみである。なお、このステップ(1) は、現在処理中の中間コードの (a-1) 手続/関数宣言ノードの LPLIST フィールド (4.3.2 項参照) から、(a-8-1) ループビギン･ノードが表す for ループの入れ子の木構造の表現 (図4.13 参照) をたどることにより、簡単に実行できる。

ステップ(2) において、コントロールフロー･グラフを深さ優先でたどる際に、ステップ(4) のために、各頂点についてその順序番号、直前の頂点へのリンク、直後の頂点へのリンクの情報を、それぞれ DFONUM フィールド、DFOLAST フィールド、DFONEXT フィールド (表4.11 参照) に記録しておく。また、その順序番号の最大値を (G1) 手続/関数宣言バーテクスの DFOMAXNUM フィールド (表4.12 参照) に記録する。

また、上記のアルゴリズムのステップ(3) において、コントロールフロー･グラフの各頂点ごとに、gen集合、kill集合、def集合、use集合を求める際、次の図4.26に示す定義の出現に関する表を参照する。この表はステップ(3) において、現在解析中の手続/関数 (図4.26では手続 q) を対象に作られる。この表は図4.26からわかるとおり、

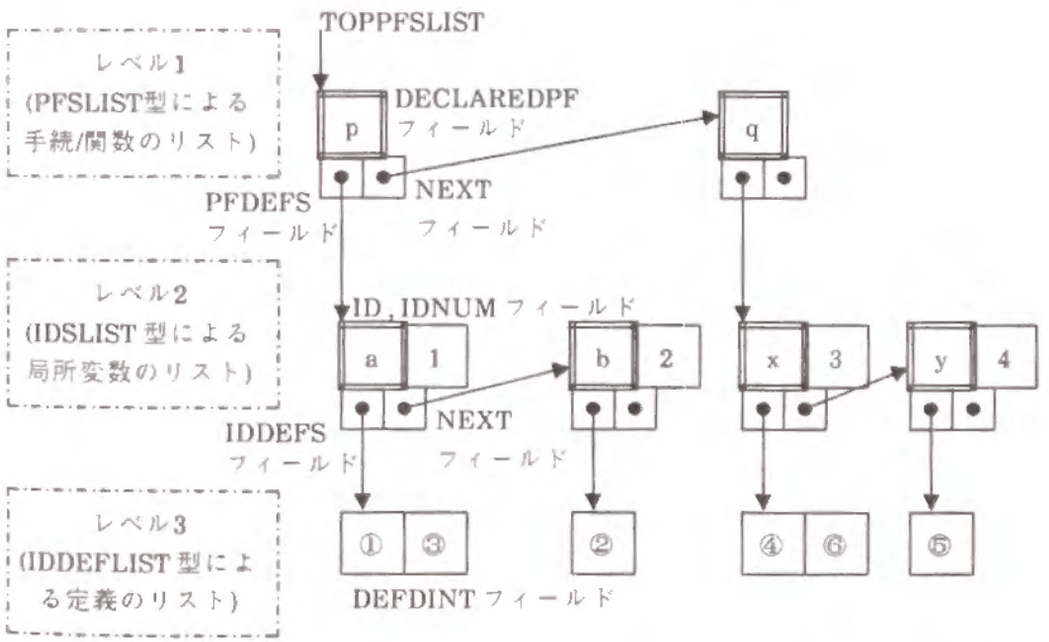
(レベル1) 変数が宣言された手続/関数による分類

(レベル2) 各変数による分類

(レベル3) 各変数の定義の集合


```
program p;
  var a,b:integer;

  procedure q(x:integer);
    var y:integer;
    begin {q}
      a:=1;{①}   b:=2;{②}   a:=3;{③}
      x:=4;{④}   y:=5;{⑤}   x:=6;{⑥}
    end {q};

  begin {main}
  end {main}.
```

(a) ソースプログラム (①②等は先と同様に各定義の識別番号)



(注)レベル1 PFSLIST型の DECLAREDPF フィールドは、中間コードの手続/関数宣言ノードを指すポインタであり、レベル2 IDSLIST 型の ID フィールドは、識別子表のエントリを指すポインタである。また、レベル3 IDDEFLIST 型の DEFDINT フィールドは配列のリストで実現している。

(b) 手続 q の定義出現表

図4.26 データフロー解析のための定義出現表

の3階層のリスト構造を持ち、ある変数に対する定義の出現をまとめている。そのため、最適化において、ある変数に対する定義を調査する際等にも利用される。これは、Pascalではスコープルールにより大域的な(より外側の手続/関数で宣言された)変数をも参照できるためである。なお、4.5節で述べたAlias解析により得られたAlias集合に対してはAlias集合ごとにシステム変数名を一意に与え、あるAlias集合に含まれる変数群に対する定義は、すべて該当するシステム変数への定義としてまとめられる。すなわち、あるAlias集合に含まれる変数群はすべて該当する1システム変数に多対1の関係で写像され対応づけられる。そしてデータフロー解析では、その1システム変数とみなして処理するのである。このようにAlias関係をAlias集合としてまとめてしまう方式では、Aliasペアを個別に扱えないので処理の粒度があらい。そのため、解析の精度はややおちるが、解析のコストは圧縮できる。

同図中のレベル3 IDDEFLIST型のDEFEDINTフィールドは、実際には定義の識別番号(これがそのまま先述の表4.18の定義に関する集合内の要素番号となる)を示すNUMBERフィールドおよびその定義が行われる中間コードを指すINTERPフィールドからなるDEFSREC型レコードを要素とするDEFARY型配列で実現されている。なお、図中にも記したとおり、この配列は固定長の配列であるためIDDEFLIST型をリストとし、最後のセルの有効成分の数を示すためIDDEFLIST型内にCRNTINXフィールドを用意した。また、表4.19の変数に関する集合内の要素番号をレベル2 IDSLIST型のIDNUMフィールドが保持する。これらの集合内の要素番号を基に上記のアルゴリズムのステップ(3)が実行される。

ある識別子表のエントリから、このレベル2 IDSLIST型の対応する変数のエントリをたどれるように、識別子表のDFLWKフィールド(表4.1参照)にポインタが格納される。同じく、レベル2 IDSLIST型の中間項のエントリをたどれるように、その中間項を定義する中間コードのDFLWKフィールド(表4.5参照)にもポインタが格納される。

データフロー解析結果を保持するため、コントロールフロー・グラフの各頂点ごとに各データフロー集合を用意する(表4.11参照)。これらの集合は、Pascalの集合型で宣言されている。すなわち、集合要素を1ビットに対応させ、当該要素が含まれるか否かを"1"、"0"で表現する。集合要素数に等しい(ただし、物理的には64バイト単位にまるめられる)長さのビットベクトルである。従って、各種集合演算は、実際には集合全体の領域に対する論理演算の組み合せで効率よく実行される。例えば、積

集合、和集合はそれぞれ単に論理積、論理和を適用するだけである。先のデータフロー方程式の集合演算も実際にはすべて論理演算数命令で実行される。

4.8 最適化

V-Pascal Version 1では、現在知られる種々の最適化手法[36],[37]のうち特にベクトル化の観点から重要なものを選んで下記のみを実現した。これらの最適化は、すべて手続/関数単位に中間コード上で行われる(同時にコントロールフロー・グラフも書き換えられる場合もある)。

(1) 定数の畳み込み(constant folding)あるいは定数の伝播(constant propagation)
(2) 複写の伝播(copy propagation)
(3) 共通部分式(common subexpression)の結果の再利用
(4) 無用命令の削除(dead code elimination)
(5) ループ内不変式(loop-invariant)のループ外への移動(code motion)

なお、制御の流れが到達できないコード(unreachable code)[36],[37]群は、フェーズ1において中間コードに変換する際除去している。これは、できるだけ早期に除去することにより以降の種々の解析等のコストを軽減するためである。

上記(1)の適用例を図4.27に示す。図中Ⓐを調査するために4.7節で述べたデータフロー集合を利用する。すなわち、定義②の位置に定義①が到達し、かつ変数Aのそれ以外の定義が到達しないことで判定する。

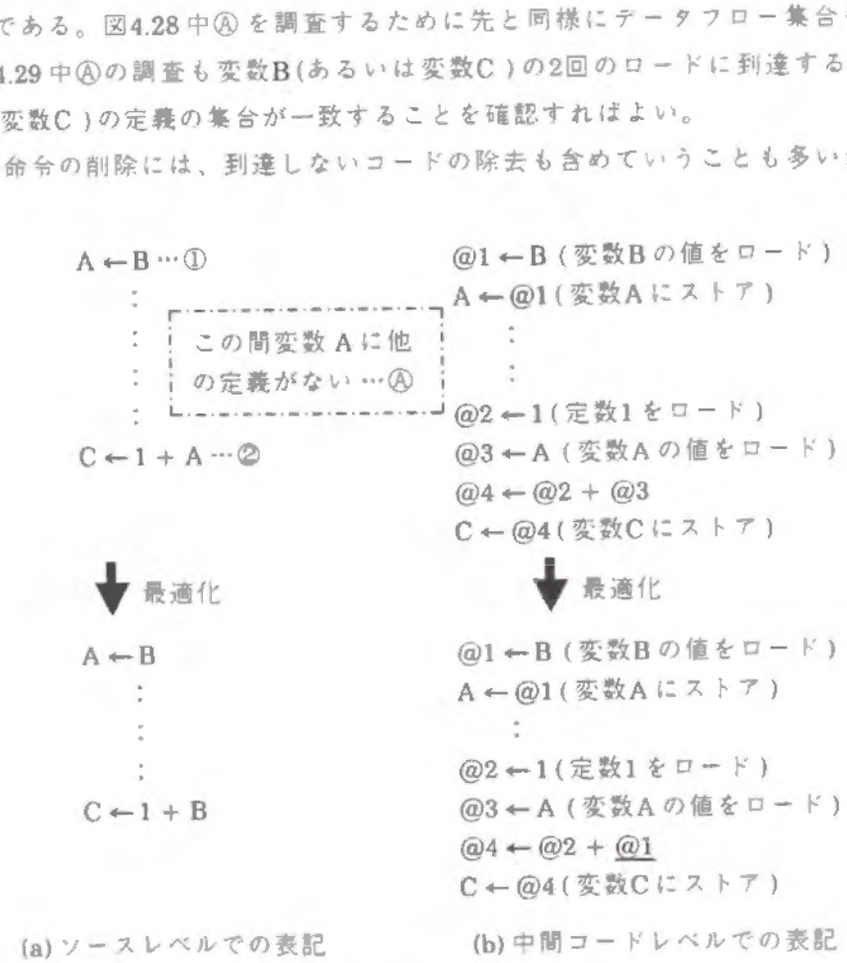
```
A ← 1 + 1 …①
    :          (この間変数Aに他の定義がない …Ⓐ)
B ← 1 + A …②
    ↓ 最適化
A ← 2
    :
B ← 3
```

図4.27 定数の畳み込みの例

さらに、V-Pascal Version 1 では通常のスカラ定数の畳み込み機能と合わせて、ベクトル定数の畳み込み機能を実現している。すなわち、単一の定数だけでなく、定数のみの並びのベクトルに関する四則演算は、コンパイル時に行われ演算結果のベクトル定数に置き換えられる。この機能は、特に第6章で述べる V-Pascal の大きな特長である多重ループの一重化ベクトル実行の際に頻繁に出現する制御変数を配列化したリストベクトル(これは、5.3.2 項で述べる基本ベクトルであり、ベクトル定数となる)を使った配列参照の添字計算が、すべてコンパイル時に行われ高速実行に大きく寄与している。

上記(2)複写の伝播および(3)共通部分式の結果の再利用の例をそれぞれ図4.28および図4.29に示す。なお、図中「@」は、4.3.2でも述べたとおり中間項であることを示す記号である。図4.28中Ⓐを調査するために先と同様にデータフロー集合を利用する。図4.29中Ⓐの調査も変数B(あるいは変数C)の2回のロードに到達する変数B(あるいは変数C)の定義の集合が一致することを確認すればよい。

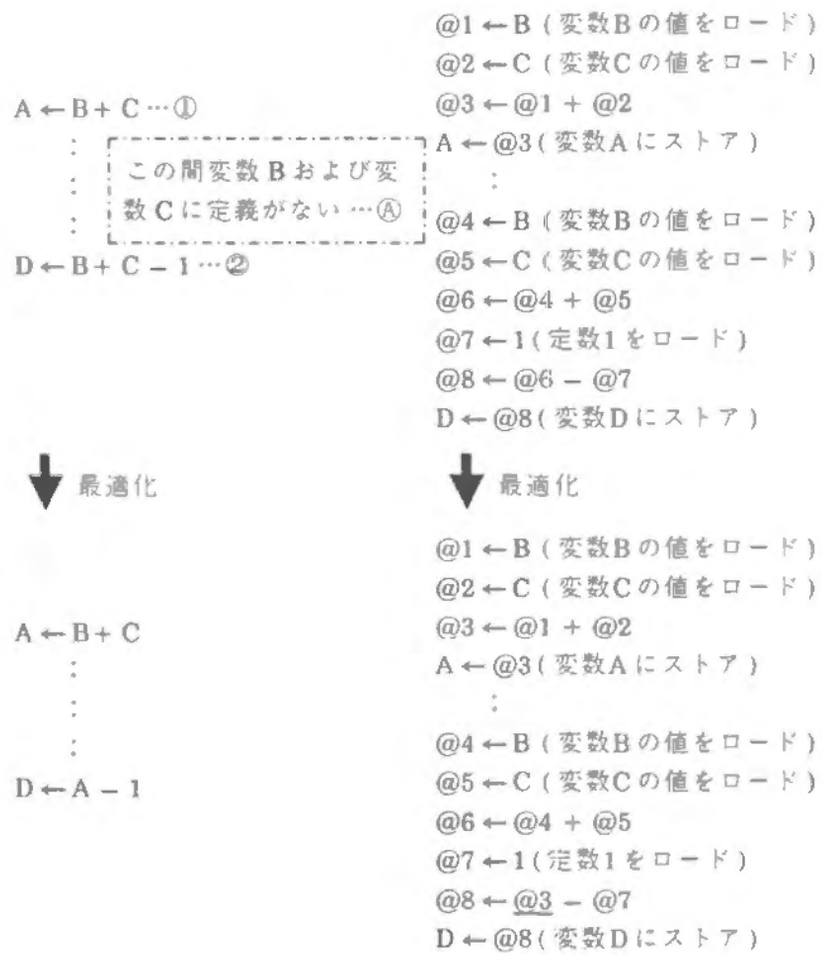
(4)無用命令の削除には、到達しないコードの除去も含めていうことも多いが、先



(a)ソースレベルでの表記　(b)中間コードレベルでの表記

図4.28 複写の伝播の例

述のとおり既に到達しないコード群は除去されている状態での最適化であるため、ここでは含めないものとする。従って、無用命令としては後で引用されることのない変数(一時変数とみなせる中間項を含む)への定義を行うコードの除去を実現した。これは、先のデータフロー集合の bout 集合を調べ、あるコントロールフロー・グラフの頂点内で出現した変数への定義のうち、bout 集合として流れ出さないものを除去するだけである。これにより、例えば連続する同一変数への値の定義を行う複数のコード(例えば、X←1 ; X←3)は、最後の定義のみを残し他はすべて除去され



(a)ソースレベルでの表記　(b)中間コードレベルでの表記

図4.29 共通部分式の結果の再利用の例

る。同様に中間項への定義でも、例えば図4.28(b)の複写の伝播を行った後の中間項@3への定義(@3←A)が除去される。また、図4.28(b)の共通部分式の結果の再利用後の@6への定義(@6←@4＋@5)が除去される。その結果、@4への定義(@4←B)および@5への定義(@5←C)が除去される。すなわち、共通部分式は1度計算されるだけで、重複する他のコードはすべて除去される。このように、この無用命令の削除は、ユーザが記述したソースレベルでは適用されることはないように思われるが、中間コードレベルでは他の最適化により無用命令が多数現れることとなるため、重要な処理である。

(5)ループ不変式のループ外への移動は、次のようなコードについて行われる。ループ内に存在するコードのうち、引用する変数(先と同じく中間項を含む)に到達するその変数への定義がすべて当該ループ外にある場合には、そのコード(計算)は当該ループ外で実行しても意味が変わらない。厳密にいうと、ループ外に存在し、かつそのループの支配頂点であり、かつ引用する全変数に関して同一の到達定義を持つコントロールフロー･グラフの頂点内で実行しても意味が変わらない。現在の実現アルゴリズムでは、この厳密な調査を省略するため、対象ループを自然ループに限定し、当該ループの入口頂点の直前のプロセス･バーテクス中の中間コード列の最後尾に順次移動させる。直前がプロセス･バーテクスでない場合には、プロセス･バーテクスを生成する。ループ不変式か否かの判定は、やはりデータフロー集合の到達定義の関係するものを各個に調査することにより簡単に行える。

これらの最適化により最適化本来の目的である演算量/コード量の縮小が可能となることはいうまでもない。ベクトル化の観点からは、例にも挙げたとおり演算もさることながら無用なロード/ストアが減少することも重要である。なぜなら、無用なロード/ストアが存在することは、そのロード/ストアされる変数へのすべてのロード/ストアとの組み合せ(ロードとロードを除く)について無用なデータ参照関係解析(第5章で詳述)を行うこととなるからである。この解析のコストは一般には高価である。さらに、その結果本来無用なデータ参照関係による依存が生じる可能性があり、その無用な依存がベクトル化を阻害することすらある。また、幸いにもベクトル化されたとしても、ベクトルロード/ベクトルストアは通常四則演算等のベクトル命令と比較すると遅いため、ベクトル計算機による加速率を引き下げることとなる。従って、V-Pascalではこのベクトル化の観点を最重視しまず上記の最適化処理のみを実現している。

4.9 結 語

V-Pascal Version 1のフェーズ構成/モジュール構成、主要な表等のデータ構造を概説した。これらのデータ構造は、識別子表、型表を除き、すべて自動ベクトル化のためにみなおし新規に設計されている。フェーズ構成では、機械依存の処理を極力最終フェーズに閉じ込めるよう留意した。そして、それを容易とする中間コードを設計した。この機械独立な方針でのコンパイラ/中間コードに関する設計論的な問題を扱った論文は数多い。しかし、ベクトル計算機に関するものはその中に見付けられなかった。文献[46]は、並列化ならびに最適化を考慮した演算木表現の中間コードについて論じている。これに対し、V-Pascalの中間コードは、種々の最適化ならびに自動ベクトル化を容易とする設計方針をとり、特に配列のデータ参照関係解析(データ依存)のために添字式の演算木を容易になぞれるようにした。コントロールフロー･グラフは制御依存の解析を容易とするよう細かく頂点を分類した。

D行列に代表される自動ベクトル化のために初めて必要となる重要なデータ構造は、より適切な形態およびそれに合わせたアルゴリズムを今後さらに研究する必要がある。

第5章
依存関係解析

5.1 緒 言

3.2節で既述したとおり、ベクトル化においては元のプログラムが持つ(スカラ実行時の)意味を保存しなければならない。このスカラ実行時の保存すべき各種実行順序を解析するのが依存関係解析である。ゆえに、もしこの解析が不正確で、誤って存在し得ない依存を検出すると、その結果本来ベクトル化可能な部分をスカラ実行してしまう可能性がある。逆に何らかの依存を見落としたりすることになれば、最悪の場合にはベクトル化不可能な部分を誤ってベクトル実行し、元のプログラムが持つ意味が保存されず誤った実行結果をだすこととなる。従って、通常依存関係解析ではもし依存の有無が確定できなかった場合には、フェイル・セーフの考え方で依存が存在するものとして扱う。なぜなら、依存の存在を仮定した場合には、上述のとおり最悪の場合でもスカラ実行されるだけで、決して意味を誤ることはないからである。しかし、元のプログラムの有する並列性を可能な限り検出し高速実行させるというベクトル化の観点からは、不要な依存は極力排除すべきであることはいうまでもない。なぜなら、依存が存在すると単純なベクトル化のみならず、ベクトル実行に最適なループを選択する、あるいはベクトル命令の実行順序を入れ換えて複数のパイプライン演算器を並列に走らせる等のさらに進んだ高速化の手法を適用する機会までも減少させるからである。つまり、依存関係解析は実行結果に関して正しいだけでなく、ベクトル化の観点から厳密であることが望まれる。

本章では3.2節で既述した2種類の依存関係

(1)データ依存関係、ならびに

(2)制御依存関係

それぞれを解析するアルゴリズムについて述べる。これらのアルゴリズムはいずれもV-Pascalのために独自に、かつ新規に考案されたものであり、多重ループ全体にわたる解析を可能とする特長をもつ。これらのアルゴリズムは、V-Pascal Version 1の依存関係解析部に実現されている。これらの解析部で存在すると判定された中間

コードの各種ノード間の依存は、依存の向き(図4.19参照)およびその種別(表4.14参照)とともにすべて4.3.5項で述べたD行列に登録しまとめられる。

5.2 データ依存関係解析

データ依存関係解析は、3.2.2項で触れたとおりベクトル化対象多重ループ内に存在する同一の(厳密には4.5節で述べたAlias関係にあるものも含む。以下同様)変数の参照間での順序関係を解析する。もちろん、大域的/局所的をとわない。ただし、参照の対(以下2出現と呼ぶ)が(引用-引用)の場合には参照の順序が保存されなくてもプログラムの意味は変わらないので、解析する必要はない。すなわち、保存すべきデータ依存関係は存在しない。従って、以下の解説ならびにアルゴリズムでは、この場合は除外されているものとする。またデータ依存関係は、論理的には2出現間の関係であるが、現在のV-Pascalでは実際には2出現それぞれを含む中間コードの2頂点間の関係として抽出している。従って以下では、混乱が生じないかぎり「2出現それぞれを含む中間コードの2頂点」のことを単純に「2出現」とのみ記す。なお、ここで言うベクトル化対象多重ループとはFORTRANのDOループ、Pascalのforループのように制御変数およびその初期値・最終値・増分値がソース・テキストから容易に抽出できるループが任意の親子兄弟の関係で多重にネストしているループ群を指すものとする。そのようなループであれば、以下のデータ依存関係解析アルゴリズムは対象言語によらない。

5.2.1 単純変数のデータ依存関係解析

単純変数(一時変数としての中間項を含む)に関するデータ依存関係解析は、次のようにコントロールフロー解析およびデータフロー解析の結果から簡単にデータ依存の有無を判定できる。

[単純変数の2出現間のデータ依存関係解析アルゴリズム]

(1) コントロールフロー・グラフにおいて2出現がともに同一の頂点(プロセス・バーテクス)に含まれる(表4.5 GRAPHPフィールド参照)とき、当該頂点の実行において先行される出現(以下出現1)から後続の出現(以下出現2)へループ独立依存(表3.6参照)が存在する。さらに、2出現がベクトル化対象多重ループ内に存在することから、ループの回転にともなう出現2から出現1への逆向きのループ運搬依存(表3.6参照)が存在する。また、厳密には出現1から出現2へのルー

プ運搬依存も存在する。以上をまとめるとD行列に登録される依存の種別(表4.15参照)は、2出現が定義か引用かで表5.1のとおりとなる。この場合にはこれらの依存は、2出現が共通に属する多重ループの全ループについてそれぞれ存在する。

(2) コントロールフロー・グラフにおいて2出現が異なる頂点(VおよびWとする)に属するとき、

(2-1) 5.4節で述べる制御依存関係解析結果を用い、ループが存在しないとみなしたときの頂点Vと頂点Wとの先行関係を得る。

(2-2-a) 先行関係がある場合は、ループが存在しないとき先に実行される側を出現1、後から実行される側を出現2とし、(1)と同様に2出現が共通に属する多重ループの全ループについて表5.1のとおり登録する。

(2-2-b) 先行関係がない場合(例えば頂点Vと頂点Wとが同一の条件分岐の真側のパスと偽側のパスとに別れて存在するような場合である)には、2出現が共通に属する多重ループにおいてループ独立依存が生じ得ない。そしてこのとき、2出現のどちらを出現1(出現2)とみなしてもよい。次のア)あるいはイ)のように実際にデータ参照関係があるか否かをデータフロー集合から求める。もしあれば、上述のとおりそれはループ運搬依存しかあり得ないので、表5.1の該当するループ運搬依存のみを登録する。

ア)(定義-定義)型あるいは(定義-引用)型の場合、後者の属するコントロールフロー・グラフの頂点のin集合をみて、前者の定義が実際に到達していることを確認する。

イ)(引用-定義)型の場合、前者の属するコントロールフロー・グラフの頂点のin集合をみて、その頂点に到達する当該変数の定義が後者の属するコントロールフロー・グラフの頂点にも到達していることを確認する。すなわち、後者は前者が引用すべき値を現に再定義していることを確認する。 □

表5.1 単純変数のデータ依存

出現1	出現2	登録される種別
定義	定義	出現1→出現2:out0(ループ独立依存で定義-定義)注)、outmd(ループ運搬依存で定義-定義)
		出現2→出現1:outmd(ループ運搬依存で定義-定義)
定義	引用	出現1→出現2:true0(ループ独立依存で定義-引用)、truemd(ループ運搬依存で定義-引用)
		出現2→出現1:antimd(ループ運搬依存で引用-定義)
引用	定義	出現1→出現2:anti0d(ループ独立依存で引用-定義)、antimd(ループ運搬依存で引用-定義)
		出現2→出現1:truemdと必要ならばtrue1(ループ運搬依存で定義-引用)

注)最適化により単純変数の場合には存在し得ない。

なお、表5.1中のループ運搬依存は実際にデータが受け渡されるtrue1を除き、すべてダミー型(表4.15参照)である。また、ループ独立依存もデータが受け渡される

```
for i;= 1 to 3 do
   begin
      t:= a[i] + b[i];
      c[i]:= t;
   end ;
```

(a) 配列化前のプログラム

```
for i;= 1 to 3 do
   begin
      ta[i] := a[i] + b[i];
      c[i]:= ta[i] ;
   end ;
```

(b) 配列化後の等価なプログラム

(注) 単純変数tをta[i]と配列化。簡単のため図3.5に示した最終値保証は省略した。変数tの2出現間にはループ独立およびループ運搬依存が存在するが、ta[i]の2出現間にはループ運搬依存は存在せずループ独立依存のみ存在。

```
for i;= 1 to 3 do
   begin
      c[i]:= t;
      t:= a[i] + b[i];
   end ;
```

(c) 配列化前のプログラム

```
for i;= 1 to 3 do
   begin
      c[i]:= ta[i] ;
      ta[i+1] := a[i] + b[i];
   end ;
```

(d) 配列化後の等価なプログラム

(注) 単純変数tをta[i]とta[i+1]と配列化。簡単のため最終値保証等の補正は省略。単純変数tの2出現間にはループ独立、ループ2回およびループ多回運搬依存が存在するが、配列化後変数taの2出現間にはループ2回運搬依存のみが存在。

図5.1 単純変数の配列化による依存の消滅

true0を除き、ダミー型(表4.15参照)である。これは、単純変数の場合のこれらの依存は、厳密には存在するが実際には無視してよいためである。なぜなら、例えば図5.1(a)では、ループの1回目の繰り返しにおいて定義された値は、その1回目の繰り返しにおいて引用されるだけである。2回目以降の繰り返しに運搬されることは実際にはない。つまり、図5.1(a)のプログラムの意味は同図(b)のように単純変数を配列化したプログラムの意味と等価であるからである。従って逆に、このダミー型の依存は表4.15の注に記したとおり、配列化すれば消去できる依存であるとみることもできる。同様に考えると、図5.1(c)および(d)に示すとおり、引用-定義の場合のループ独立依存も配列化すれば無視してよいダミーの依存である。ただし、この例からわかるとおり、この場合には定義-引用のループ2回運搬依存は配列化しても消去されない。従って、表5.1中に示すとおりこの場合にはtruemdとともにtrue1をも登録する必要がある。すなわち、この例の場合には運搬依存では、ループ多回運搬依存のみが配列化すれば無視してよいダミーの依存である。

単純変数の場合には、最適化により通常上記アルゴリズム(1)の場合は生じない。例えば、図5.1(a)のプログラム例では変数tの引用は除去され、1番目の代入文の右辺の演算結果の中間項を直接参照するように置き換えられる。すなわち、第2の出現は除去されてしまう。しかし、上記アルゴリズム(1)は特にそういう一時変数としての中間項の定義-引用間での依存種別判定および次に述べる構造のある型の変数のデータ依存解析をも統一的に処理するために必要である。ただし中間項の場合には、ベクトルレジスタが割り付けられ、実行時にその上に各繰り返しごとの中間結果が展開されるため、あたかも自動的に配列化されるとみなせる。従って、中間項の場合には先に述べたダミーのループ運搬依存は、すべて登録する必要がない。

5.2.2 構造のある型の変数のデータ依存関係解析

Pascalでは、その変数(付録 Pascal構文図のVariable参照)はその型に応じてレコード型全体、レコード型の1フィールド、配列全体、配列要素等種々の形態のデータ参照が可能である。ところが、ベクトル化対象多重ループの各制御変数(厳密にはベクトル化対象多重ループ内で定義される帰納変数[36],[37]等の変数を含む。以下同様)からなる添字式を有する配列要素の参照以外はすべて、ループの繰り返しに伴い参照される番地が変化することがない。従って、自動ベクトル化のためのデータ依存関係解析の観点からは、上記のようにループの繰り返しに伴い参照される番地が変化しない場合には、5.2.1項で述べた単純変数とまったく同等にみなして、同じアルゴリズムで解析できる。これに対し、多重ループの各制御変数からなる添字式を有する配列要素の参照間でのデータ依存関係解析では、5.2.1項で述べた単純変数の場合の解析アルゴリズムにおいて、さらに2出現の添字式を調査する必要がある。そして、2出現の添字式から、ループの繰り返しに伴い実際に同一の番地、すなわち同一の要素が参照されるか否かを確認する。例えば、同一の要素が参照されることがないと判定できれば、データ依存は存在しない。逆に同一の要素が参照される場合には、そのときの制御変数の値の大小関係により3.2.2項で既述したとおり各ループごとに、ループ独立依存のみが生じるのか、ループ運搬依存のみが生じるのか、あるいはその両方が生じるのかをも合わせて調査する必要がある。その結果、表5.1の該当する依存の中から実際に存在すると判定された依存のみを登録することとなる。ただし、この場合配列をさらに配列化することは通常原理的には可能であるが、配列化する(元の配列を対象ループの総繰り返し回数分保持する)のに必要な領域、および最終値保証等の補正(一般には各繰り返しごとに元の配列全要素の値の複写が必要)のための処理時間を考慮し配列化は断念するものとする。これは、配列のみならず構造のある型の変数一般にいえることである。このように配列化を行わない場合には、表5.1のダミー型truemd、anti0d、antimd、outmdは、それぞれ(true1あるいはtruem)、anti0、antim、outmに読み替え登録するものとする。なお、true1とtruemを区別する目的は、3.3.2で述べた種々の回帰型の演算をベクトル実行するベクトルマクロ命令を検出することにある。この検出およびベクトルマクロ命令の生成機能は、現バージョンの機能拡張として実現されている。

5.3 配列要素のデータ依存関係解析

ここでは、上記の多重ループの各制御変数からなる添字式を有する配列要素の2出現間のデータ依存関係解析について詳述する。V-Pascal Version 1では、2種類の性質の異なる解析方法を組み合わせることにより、十分高速でかつ既存の手法[13]と比較してより多くの場合に厳密な解析(必要十分な解析)が可能となった。これら2種類の解析方法を数式処理法およびSort-Merge法と呼称している[26]。また、これまでに実現された手法[16],[17]が多重ループの中のループを個別に調べる解析法であるのに対し、本節で述べる方式は多重ループ全体を一重化しベクトル実行させる目的のため、多重ループすべての繰り返し実行全体にわたる解析を1度に行えることも特徴となっている。

配列要素のデータ依存関係解析の処理手順の概略は次のとおりである。まず(1)ベクトル化対象多重ループ内の同一配列の2出現について、当該多重ループの全繰り返し実行全体に渡る各要素の参照の衝突の有無の調査を行う。具体的には、着目している2出現の配列参照の添字式からDiophantus不定方程式を導出し、これを解く。すなわち、方程式の解を求める。(2)その結果、もし存在すれば得られた解から、C行列(Collision Matrix)の概念を使って、当該2出現の依存関係、つまり依存の向きを求める。以下、5.3.1項、5.3.2項で問題を定式化し、その後解法を与える。

5.3.1 添字式に着目したDiophantus不定方程式と繰り返し空間

着目しているk次元配列の2出現に関して同一要素が参照される可能性を探るため、以下の形式の連立方程式をたてる。

$$f_1 = f'_1,$$
$$f_2 = f'_2,$$
$$\vdots$$
$$f_k = f'_k$$

(k:着目している配列の有する次元数)。

ここに、f_i($1 \leqq i \leqq k$)は2出現のうちスカラ実行時に先に実行される側(5.2.1で述べた単純変数の解析アルゴリズムで出現1としたものに相当するので、以下でも出現1と呼ぶ)の第i次元目の添字式である。この出現1が属する多重ループ(ネストの深さpとする)の各制御変数を最外側ループから順にI_1、I_2、…、I_pと名付けるとすると、添字式f_iはI_1、I_2、…、I_pの関数とみなせる。同様にf'_iはスカラ実行時に後に実行される側(先と同様以後出現2と呼ぶ)の第i次元目の添字式である。f'_iは出現2が属するネストの深さqまでのループの各制御変数I'_1、I'_2、…、I'_qの関数とみなせる。I_1、I_2、…、I_pとI'_1、I'_2、…、I'_qのうち最初のr個ずつ、I_1、I_2、…、I_rとI'_1、I'_2、…、I'_r($1 \leqq r \leqq p$、q)は両者共通のループ(**共通親ループ**と呼ぶこととする)の制御変数であるが、上記の方程式中では別々の値を取る異なる独立変数として取り扱う。これは、ベクトル化が対象多重ループを分割し、2出現を別々の多重ループとして実行することに相当する(図3.4参照)ので、独立した動きの別個の制御変数とみなして解析する必要があるためである。なお、上記の出現1と出現2の区別、すなわち、2出現のうちどちらがスカラ実行時に先に実行されるかの判定は、制御の流れを詳細に解析してはじめて判定できる(5.4節参照)。ここでは、本データ依存関係解析処理以前にその種の解析が終了しているものとする。

配列の各要素ごとの参照の衝突の様子の解析は、上記の2出現の添字式から得られた(I_1、I_2、…、I_p;I'_1、I'_2、…、I'_q)に関する(p+q)元のk個の等式からなるDiophantus不定方程式(以下、単に不定方程式あるいは混乱のない限り方程式と略記する)が、どのような整数解を持つかを調べることに帰着される。これらを通常の(p+q)元連立方程式とみなすと、1)複数の解が存在する場合(不定)、2)解が一意に求められる場合、3)解が存在し得ない場合(不能)に場合わけできる。ただし、この場合の解は一般には実数解である。それに対し今の不定方程式の場合には、整数解に絞られるが、同様に1)解が存在しない場合と2)存在し得る場合がある。ただし、その整数解に対する制約条件として、本来の制御変数としての値域内すなわち取り得る値の組み合せの集合に入っていることが必要である。この制約条件は各制御変数が、対応するループの初期値以上かつ最終値以下であるという不等式で表現できる。すなわち、

$$Init_1 \leqq I_1 \leqq Final_1 \text{、} Init'_1 \leqq I'_1 \leqq Final'_1$$
$$Init_2 \leqq I_2 \leqq Final_2 \text{、} Init'_2 \leqq I'_2 \leqq Final'_2$$
$$\vdots$$
$$Init_r \leqq I_r \leqq Final_r \text{、} Init'_r \leqq I'_r \leqq Final'_r$$
$$Init_{r+1} \leqq I_{r+1} \leqq Final_{r+1} \text{、}$$
$$\vdots$$
$$Init_p \leqq I_p \leqq Final_p \text{、}$$
$$Init'_{r+1} \leqq I'_{r+1} \leqq Final'_{r+1} \text{、}$$
$$\vdots$$
$$Init'_q \leqq I'_q \leqq Final'_q$$

の(p+q)個の不等式である。ここに、$Init_i$、$Init'_i$、$Final_i$、$Final'_i$は2出現が属するネストレベルi番目のループの初期値、最終値である。これらは、一般には自分より外側のループ制御変数を含む式である。そしてさらに、解は一般にはこれらの各区間に含まれる整数値かつ、対応するループの増分値(ステップ値)おきの離散的な値でなければならない。ところが、このループの増分値はどのような整定数でも増分値が1の等価なループに変換(正規化)できるので、単純に区間内の整数値という条件としても一般性を失わない。従って、以下では特にことわらない限りループの増分値はすべて1であると仮定する。このように、データ依存関係解析では(p+q)元のDiophantus不定方程式を(p+q)次元の格子空間全域にわたり解を求めるのではな

く、上記の不等式で限定された部分(格子)空間(繰り返し空間、iteration space と呼ばれる[14])内で解を求める。

図5.2に簡単な例に基づき抽出される不定方程式および各制御変数の取り得る値の範囲を示す。

```
for i := 1 to 3 do begin
    for j := i+1 to 3 do begin
        a[i,i] := … ; { 出現1 }
        for k := 1 to 3 do begin
            … := a[i,k]; { 出現2 }
        end;
    end;
end;
```

(a) 配列の2出現の例

i = i' (1次元目),
i = k' (2次元目) (ただし、1 ≦ i,i',k' ≦ 3)

(b) (a)の例の Diophantus 不定方程式

図5.2 簡単な配列参照の2出現とそれから導出される Diophantus 方程式の例

5.3.2 C行列と基本ベクトル

前記の不定方程式より得られた整数解をC行列と呼ぶ行列にまとめる。このC行列の概念を用いると、解の存在とそれが意味する配列要素の参照の順序関係を把握することが容易になる[18],[26]。

C行列は I_1、I_2、…、I_p の取る値の組み合せをスカラ実行時の繰り返し実行順に縦軸方向にとり、同じく I'_1、I'_2、…、I'_q の取る値の組み合せを繰り返し実行順に横軸方向にとり、衝突の生じる、すなわち解の存在する (I_1、I_2、…、I_p; I'_1、I'_2、…、I'_q) の組み合せとなる要素についてのみ"1"を、その他の要素には"0"を持つものとする。つまり、C行列は5.3.1項で述べた (p+q) 次元の繰り返し空間を繰り返し実行順に2次元にたたみこんだものとみなせる。

なお、このように制御変数の取り得る値を繰り返し実行順に並べた、すなわち制御変数自身を配列化した値の並びを以下では**基本ベクトル**(次の図5.3参照)と呼ぶことにする。

このC行列において、

a) $I_1 < I'_1$ および
$I_1 = I'_1$、$I_2 < I'_2$ および
$I_1 = I'_1$、$I_2 = I'_2$、$I_3 < I'_3$ および
⋮
$I_1 = I'_1$、$I_2 = I'_2$、…、$I_{r-1} = I'_{r-1}$、$I_r < I'_r$ (rは5.3.1項と同じ)

に相当する部分を**上三角(部)**と呼ぶ。また、

b) $I_1 > I'_1$ および
$I_1 = I'_1$、$I_2 > I'_2$ および
$I_1 = I'_1$、$I_2 = I'_2$、$I_3 > I'_3$ および
⋮
$I_1 = I'_1$、$I_2 = I'_2$、…、$I_{r-1} = I'_{r-1}$、$I_r > I'_r$

に相当する部分を**下三角(部)**と呼ぶ。また、

c) $I_1 = I'_1$、$I_2 = I'_2$、…、$I_{r-1} = I'_{r-1}$、$I_r = I'_r$

に相当する部分を**対角(部)**あるいは**境界(部)**と呼ぶ。なお、この分類は制御変数 I_1、I_2、…、I_r (I'_1、I'_2、…、I'_r) の増分値が正である場合であり、負の増分値を持つ制御変数が存在する場合には当該制御変数に関する不等号の向きを逆転させて定義するものとする。C行列の簡単な例を図5.3に示す。

図5.3 図5.2(a)の2出現のためのC行列

以上の分類は、着目している2出現に共通の制御変数の大小関係から得られるものである。この分類で、a)の領域に存在する"1"はスカラ実行した際、出現1で参照されるある配列要素を、各ループが順順に繰り返され時間的に後から出現2ででも参照することを意味する。逆に、b)の領域に存在する"1"はスカラ実行した際、出現2で

参照されるある配列要素を、時間的に後から出現1ででも参照することを意味する。c)の領域に存在する"1"はスカラ実行した際、共通親ループのある繰り返しにおいて同時に、出現1と出現2両方で同一の配列要素を参照することを意味する。

従って、C行列を走査し、ある配列要素の参照の衝突を意味する"1"が、上三角部a)の領域に有れば、その衝突から生じる依存関係は出現1→出現2の向きであると判定できる。逆に、下三角部b)領域に有れば、出現2→出現1であると判定できる。対角部c)領域に有れば、ループの繰り返しとしては同時であるが、各繰り返しごとにみれば出現1が先に実行されると仮定したことから、その衝突から生じる依存関係は出現1→出現2であると判定できる。複数の"1"がある場合には、個々の"1"ごとに独立に判定した結果を組み合わせる。例えば、a) b) c)のうち複数の領域にまたがって"1"が存在する場合には、a) c)では依存関係は出現1→出現2であるが、a) b)、b) c)では双方向の依存があると判定できる。逆に、解がどの領域にも存在しない場合には、当該2出現に関しては、参照の衝突がなく依存関係はないと判定できる。

こうしてC行列を使い、ある配列要素の参照の衝突を引き起こす制御変数の値の組から、その参照の衝突により生じる依存の向きを調べることができる。

5.3.3 数式処理法

上で述べたとおり、データ依存関係解析はC行列の概念を用いれば、2出現の添字式から得られる不定方程式の解がC行列の上三角、下三角、対角のどの部位に存在するかを調査することに帰着される。一方これらの部位に関する場合分けは、共通親ループの制御変数のペアI_i、I'_i $(1 \leqq i \leqq r)$の大小関係による。従って、5.3.1項で既述の元の不定方程式および本来のループ制御変数として取り得る値の範囲を表す不等式を使い、これらの制御変数のペアのみに関する条件式(以後**ペア関係式**と呼ぶ)を抽出し、依存関係(依存の向き)ならびに依存の種別を判別すればよい。これが、**数式処理法**(symbolic manipulation)の概要である。この数式処理法は、5.3.1項で述べた不定方程式の繰り返し空間における解探索アルゴリズムとみることができる。

なお、この数式処理法のアルゴリズムを**V-Pascal**コンパイラにインプリメントする際、以下の制約を付加した。1)多重ループの各制御変数の初期値、最終値が、より外側の制御変数の線形式で表され(最外側の場合には定数値である)、増分値は定数であること。2)同じく2出現の各添字式が制御変数の線形式で表されること。この制約条件は、インプリメントに要する期間を短縮するために、式を記憶するデータ構造を単純なテーブルとし、変数の消去等の式の操作を容易にする目的で付加された。

た。従って、複雑な式を記憶できるデータ構造ならびに強力な式操作モジュールを作成すれば、非線形の項を新たに1変数に置換するなどして、アルゴリズムとしては上記の制約を緩和し、より広いクラスのループ、配列添字式に対応できる。

[数式処理法の概略アルゴリズム]

(1) 得られた不定方程式について、最外側のループ制御変数から内側へ順に、制御変数の増分値が正で初期値が0となるよう変数変換を行う(図5.4参照)。この処理を**正規化**と呼んでいる。ただし、この図5.4では説明のためソースプログラム上で正規化の処理を行っている。

(2) 不定方程式の各等式について、変化が無くなるまで次の2ステップを繰り返す。

(2-1) 制御変数を含まない等式(恒等式)が得られた場合には、両辺の無矛盾性を検査し矛盾があれば(例えば、1=2のように)、解無しすなわち依存無しと判定する(終了)。

(2-2) X = 1のように、制御変数を決定する式が得られた場合には、他の等式中の当該変数を決定された定数値に置き換える。

(3) 得られた不定方程式の各等式について、後で述べる最大公約数に関する検査(GCDテストと呼ばれる[13])を行い無矛盾性を調査する。もし、解が存在し得ないと判定されたなら終了。

(4) こうして得られた不定方程式の各等式を通常の多次元配列の添字計算の手法で**1次元化**(array linearization)し、1個の等式にまとめる。例えば、図5.2の配列aが、a[2..9, 1..10]のように宣言されていたとすると、正規化された図5.4(c)の各次元ごとに得られた

$$I + 1 = I' + 1 \text{ (1次元目) および } I + 1 = K' + 1 \text{ (2次元目)}$$

すなわち、

$$I = I' \text{ (1次元目) および } I = K' \text{ (2次元目)}$$

の2式から、

$$(I - 2) \times 10 + (I - 1) = (I' - 2) \times 10 + (K' - 1)$$

を得る。この1次元化の段階で、情報が失われることはない。

(5) 最外側のループ制御変数から内側へ順に、元のループの初期値、最終値の式から、取り得る値の最大値および最小値を定数値として求める(例えば、図5.4の正規化後の制御変数では、$0 \leqq I, K \leqq 2$、$0 \leqq J \leqq 1$となる)。あるループ制御変数に関して処理する際、当該ループの初期値、最終値の式中に現れるより外側

```
for I := 0 to 2 do begin
    for j := I+2 to 3 do begin
        a[I+1,I+1] := … ;{ 出現1 }
        for k := 1 to 3 do begin
            … := a[I+1,k];{ 出現2 }
        end;
    end;
end;
```

(a) i←I+1の変数変換を行った結果

```
for I := 0 to 2 do begin
    for J := 0 to 1-I do begin
        a[I+1,I+1] := … ;{ 出現1 }
        for k := 1 to 3 do begin
            … := a[I+1,k];{ 出現2 }
        end;
    end;
end;
```

(b) さらにj←J+I+2の変数変換を行った結果

```
for I := 0 to 2 do begin
    for J := 0 to 1-I do begin
        a[I+1,I+1] := … ;{ 出現1 }
        for K := 0 to 2 do begin
            … := a[I+1,K+1];{ 出現2 }
        end;
    end;
end;
```

(c) さらにk←K+1の変数変換を行った結果(正規化完了)

図5.4 図5.2(a)を例とする正規化

の制御変数を、この(5)で先に求めた定数値としての最大値/最小値を使い消去する。ただしもし、(2-2)である定数値として値が求められている場合には、このとき得られた最大値と最小値の区間内に、当該定数値が入っていることを確認したうえで、最大値および最小値を当該定数値とする。入っていなければ解なし(終了)。

(6) 1次元化後の方程式から、最外側のループ制御変数から内側へ順に、次の(7)から(9)を繰り返し、I_i-I'_i平面($1 \leqq i \leqq r$)上での解の存在領域を確定する。

(7) 着目しているループ制御変数以外の制御変数を消去し、I_iとI'_iのみのペア関係式$c \leqq aI_i + bI'_i \leqq d$(a、b、c、d:整数)を導出する。この消去は、解空間での解の存在する部分を、I_i-I'_i平面に射影するとみることができる。その他の制御変数を消去する際には、(2-2)で得られた等式および本来の制御変数としての初期値、最終値による不等式を用いる。そのため、初期値および最終値が、より外側の制御変数の式であることを考慮し、最内側ループの制御変数から外側の制御変数へと順に消去する。I_iとI'_iとの一方のみを持つ等式が得られた場合には、aあるいはbが0であるとして以下の処理を行う。

(8) I_iとI'_iとの係数の最大公約数の関係を使い上記(6)で得られたペア関係式の取り得る値の範囲をさらに限定する。つまり、ペア関係式

$$c \leqq aI_i + bI'_i \leqq d$$

の下限値c(上限値d)を[aおよびbの最大公約数]×Sの倍数で、しかもこのc値以上の最小のもの(d値以下の最大のもの)となるように置き換える。Sは当該ループ制御変数の増分値である。たとえば、もともと

$$1 \leqq 2I + 2I' \leqq 5、S=1$$

として得られたならば、これを

$$2 \leqq 2I + 2I' \leqq 4$$

と改める。

(9) そして、同一の制御変数に関する複数のペア関係式が存在する場合には、それらの示す範囲の積集合をとる。その結果取り得る値が存在しなければ解無し、すなわち依存無しと判定する。 □

上記のアルゴリズムにおいて、(1)で述べたようなこの種の正規化では、より徹底して増分値を1とする変数変換[17]もあるが、本アルゴリズムでは、(8)の最大公約数

による検査の際に同時に増分値に関する制約条件を考慮し組み込んでいる。ただし、V-Pascalの場合にはベクトル化対象ループがPascalのforループであることから、自動的に増分値は1に正規化される。

この(3)の係数の最大公約数による検査および(8)の手順では、一般によく知られている次の定理を利用している。

[定理]

変数x、yに関する方程式ax+by=c(a、b、c:整数)が整数解を持つ必要十分条件は、cがaとbの最大公約数で割り切れることである。 □

上記の定理に、正規化後の制御変数I(およびI')のループは増分値がS(整定数)、初期値が0であることを考え合わせると次の系が成り立つ。

[系]GCDテスト

ペア関係式aI+bI'=cが整数解を持つための必要十分条件は、cが[a、bの最大公約数]×Sで割り切れることである。 □

従って、(3)では上記の系がそのまま使用できる。また、上記(7)で得られた一般に不等式の形のペア関係式についても、(8)に示したとおり、その上下限値cおよびdが[a、bの最大公約数]×Sの倍数となるように範囲を限定できる。

(5)で求める制御変数の最大値および最小値は、一般的にはループ内不変式となるが、先述のインプリメント上のループの初期値、最終値に関する制約条件を付加すると定数値となる。さらに線形式(1次式)であることも仮定しているため、図5.5に示すとおり、一般に(1)の正規化により最小値と最大値との範囲は狭められ、それ以降の処理でより厳密な解の存在範囲の絞り込みが可能となる。

(9)の複数のペア関係式による解の存在範囲の絞り込みは、図5.6に示すようなI_i-I'_i平面上で、(5)で得られた不等式が示す領域(繰り返し空間のI_i-I'_i平面への射影)ならびに各ペア関係式が示す領域を重ね合わせ、これらすべての領域に含まれる部分が存在するか否かを調査することにより行われる。これは、具体的には各ペア関係式の係数の値によって場合わけすることにより実現している。もし、すべての関係式を満足する領域が存在したならば、その領域が図5.6に示すように直線$I_i = I'_i$をまたぐのか、それとも$I_i > I'_i$の領域にのみ存在するのか、$I_i < I'_i$の領域にのみ存在するのかをさらに調べることにより、5.3.2項で述べたC行列上での解の存在部位を

確定し、依存の向きを判定できる。ただし、(5)および(7)で消去処理を行う際、不等式を利用していること、また、解を整数解に限定していないことにより、このアルゴリズムは必要条件でしか解いていない。すなわち、このアルゴリズムで解析された解の存在領域は、実際には整数解が存在する可能性のある領域を求めているにすぎないことに注意が必要である。

(4)の1次元化によっても、論理的にはk個の独立な等式(rankがkと言い換える

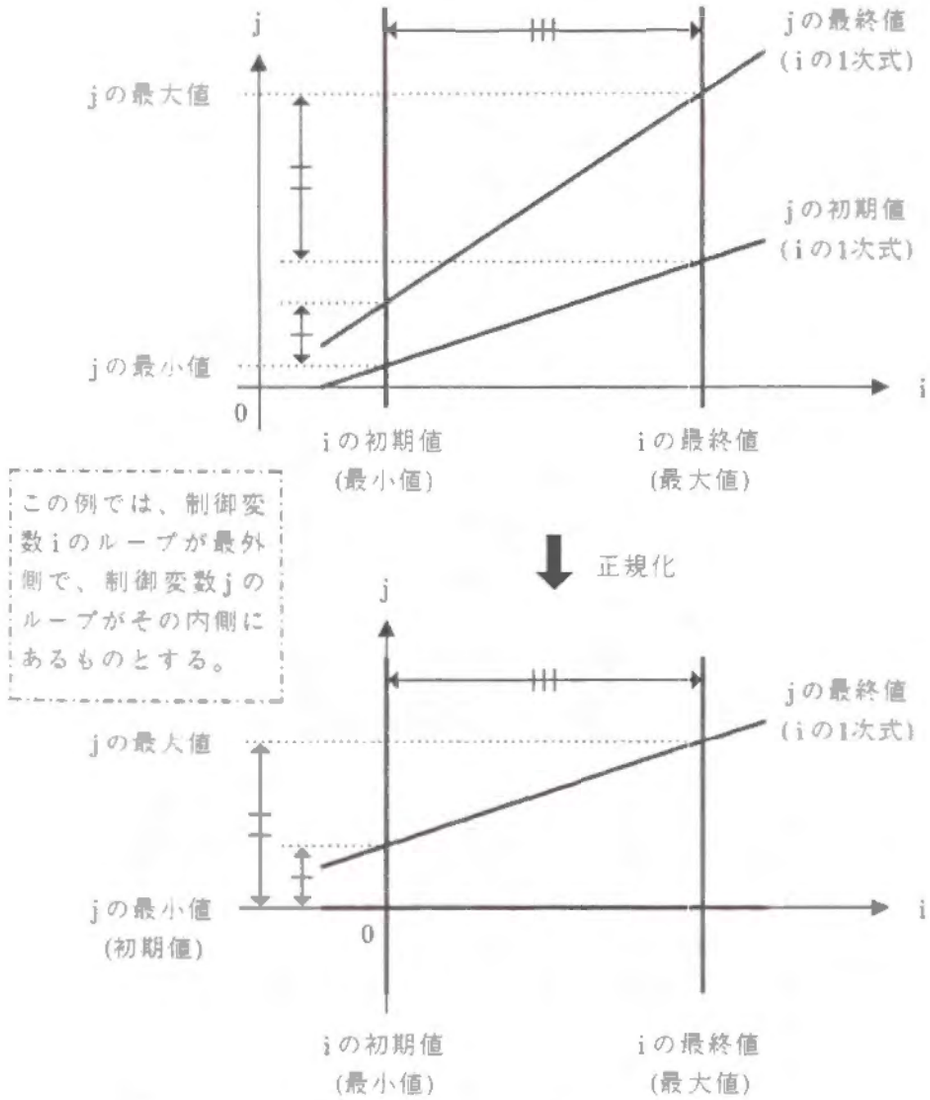


図5.5 正規化とループ制御変数の最大値/最小値の一般的な関係

ことができる)から、ただ一つの等式を導きだし、以降の処理には他の独立な(k−1)個の等式を使用しないことから、必要条件でしか解いていない。しかし、実際には配列は1次元化され記憶領域が割り付けられるため、この定式化は、あながち誤りではない。図5.2の配列aが、a[2..9,1..10]と宣言されていたとすると、a[2,11]のような宣言時の寸法を越える要素の参照は、実行時のエラーとなる処理系もある(特にPascalではこのような実行時の添字検査を行う処理系が多い)が、何の検査もせず単純に添字の計算を1次元化し、a[3,1]の参照と同じロケーションをアクセスする処理系もある(通常のFORTRANの処理系は処理時間を優先させるため添字検査を行わないのが一般的である)。この後者の場合には、むしろこのアルゴリズムで行う1次元化は自然であるといえる。現在は、この点を考慮した上で、アルゴリズムが単純となり時間のコストが少なくなることから1次元化した方程式を処理している。

なお、これらの式変形処理を容易にするため、予め不定方程式の右辺の項はすべ

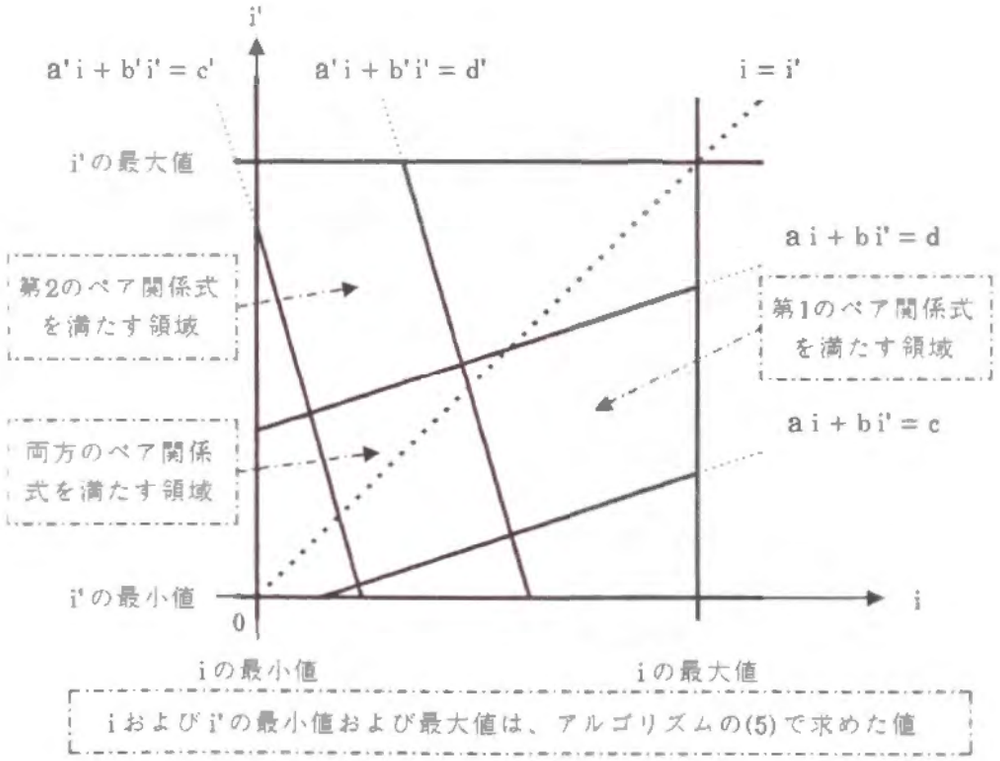


図5.6 解の存在範囲の絞り込み

て左辺に移項し、左辺=0の形にして扱うようプログラムしている。

以上述べてきた数式処理法の特徴を要約する。

[利点]

1)ループ繰り返し回数に依存しない高速な処理が可能である。

2)扱えるループの初期値、最終値および添字式に関する制約が少ない。

[欠点]

変数の消去等不等式による範囲の限定(式変形)を行うため、一般には必要十分には解を求められない場合がある。すなわち、厳密に解の存在を判定できず実際には存在しなくても依存関係があると判定せざるを得ないことがありうる。従って、完全に厳密なベクトル化ができないことがある。

この厳密さを欠くという問題点を克服するため、次のSort-Merge法をも開発し併用している。

5.3.4 Sort-Merge 法

この手法は、コンパイル時にC行列が表す格子点空間を、すなわち各制御変数の取る値の組み合せを確定させるため、次の条件を満足する多重ループについてのみ適用できる。

1)多重ループの各制御変数の初期値、最終値、増分値が、より外側の制御変数のみの式で表される(最外側の場合には定数値である)こと。

2)不定方程式が(従って、2出現の各添字式が)制御変数のみの関数であること。

そして、この手法をインプリメントする際にも、数式処理法の場合と同じ目的で同じ制約(5.3.3項参照)を付加した。

[Sort-Merge法の概略アルゴリズム]

(1) 2出現それぞれの属する多重ループ全体に渡る各制御変数の取り得る値の組み合せ(図5.3の基本ベクトル)を求める。

(2) 上で求めた基本ベクトルをもとに、2出現の添字式を計算し、各繰り返しごとに何番目の要素が参照されるかを求める。多次元配列の場合には通常のアドレス計算同様1次元化した要素番号を求める。

(3) 上で求めた要素番号および(1)で求めた各制御変数の値の組を組み合わせて1レコードとし2出現それぞれに対しレコード列を作る。その二つのレコード列について各々要素番号をキーとして基底法(Radix Sort)により昇順にソートする。

(4) ソートされた両レコード列を先頭から順に要素番号をキーとしてマージ・ソートの手法により同一要素番号を両方が参照するかどうか調べる。

(5) もし、同一要素を参照する組み合せが存在した場合には、両レコード中に記されている当該要素番号を参照する時の各制御変数の値の組が元の方程式の解である。 □

この後、解となるレコード中に保持されている各制御変数の値を比較することにより、C行列のどの部分に相当するか、そしてどういう依存関係が存在するかを判定する(5.3.6項で詳述する)。そのとき、一般には2出現各々が同一要素を複数回参照し得る。従って同一キーのレコード複数個ずつを突き合わせることとなる。そこで、その場合には、その複数個のレコードについて、両レコード列ごとに2出現に共通のループの各制御変数の最大値および最小値を探索する。各制御変数について得られた[最小値、最大値]の形式の2閉区間の大小関係からC行列の部位を判定し、依存関係を決定する。以上の処理の流れを簡単に図5.7に示す。

なお、(2)で1次元化した要素番号を求めることは、Pascal等では、コンパイル時に配列の各次元の寸法が確定するため、容易である。もし、寸法が確定しない言語についてこの手法を適用する場合には、多次元配列の各次元ごとの要素番号をあたかも多重キーとみなし、文献[47]に示されているマルティプルキージョインの際の順序保存ハッシュ関数を用いることにより、疑似的に1次元化可能である。コンパイラ中で実現したアルゴリズムでは、(1)および(2)を同時に求めている。また、(3)のソーティングの手法に基底法を選択した理由は高速であり、かつこの解析を行うプログラム自体をベクトル実行させた場合にも、ベクトル計算機になじみ易いからである[48],[49]。

Sort-Merge法の特徴をまとめる。

[利点]

(1) 通常C行列の各要素全部について解探索しようとするとC行列のサイズ(配列参照の2出現それぞれの属する多重ループ全体の繰り返し回数、つまりそれぞれの基本ベクトルの長さの積)に比例した計算量がかかるのに対し、この手法ではC行列の両辺

の長さの和(それぞれの基本ベクトルの長さの和)に比例した計算量ですむ。

(2) 衝突のチェックを必要十分に行うため、厳密な依存関係が求められ、ひいては完全に厳密なベクトル化判定が行える。

(3) 対象多重ループの初期値および最終値、ならびに配列参照の添字式が非線形であっても、複雑な式であっても、統一的に処理できる。

[欠点]

(1) 対象多重ループの初期値および最終値、ならびに配列参照の添字式がコンパイル時に計算できなければならないという制約がある。

	出現1側	出現2側
基本ベクトル	i: 1 1 2 j: 2 3 3	i': 1 1 1 1 1 1 2 2 2 j': 2 2 2 3 3 3 3 3 3 k': 1 2 3 1 2 3 1 2 3
参照要素番号	1 1 5	1 2 3 1 2 3 4 5 6

(a) 基本ベクトル、参照要素番号を求める(a[1..3,1..3]と宣言されたとする)

	出現1側	出現2側
基本ベクトル	i: 1 1 2 j: 2 3 3	i': 1 1 1 1 1 1 2 2 2 j': 2 3 2 3 2 3 3 3 3 k': 1 1 2 2 3 3 1 2 3
参照要素番号	1 1 5	1 1 2 2 3 3 4 5 6

(b) 参照要素番号をキーにソート

	要素番号1のとき	要素番号5のとき
衝突時の制御変数の区間	$1 \leqq i, i' \leqq 1$ $2 \leqq j, j' \leqq 3$	$2 \leqq i, i' \leqq 2$ $3 \leqq j, j' \leqq 3$
制御変数の大小関係	$i = i'$ かつ $j = j', j < j', j > j'$	$i = i'$ かつ $j = j'$
依存関係	出現1 ⇆ 出現2	出現1 → 出現2

(c) 衝突の状況調査および判定される依存関係

図5.7 図5.2の2出現を例とするSort-Merge法の処理

(2)解析対象の配列参照の属する多重ループの繰り返し回数が多い場合計算量が大きくなる。場合によっては、領域の制限から適用できなくなることがある。

5.3.5 数式処理法と Sort-Merge 法を組み合せた複合アルゴリズム

数式処理法ならびに Sort-Merge 法それぞれの欠点を克服するため、両者を組み合せた複合アルゴリズム(combined algorithm)を開発した。このアルゴリズムにより現在の V-Pascal では、対象に合わせて両手法の長所が生かされ、短所が補完されるように組み合せて解析が進められる。

[複合アルゴリズムの概略]

(1) 【数式処理法の前処理適用】与えられた不定方程式および繰り返し空間に対して、5.3.3 項の数式処理法の処理(1)から(5)をまず適用する。

(2) 最外側のループ制御変数から内側へ順に、次の(3)を繰り返し、解をその順序で再帰的に確定する。すなわちまず、iを1として(3)へ。

(3) 【再帰的解探索】この時点で再帰的に解くべき子問題では、深さ i−1 番目までのループの制御変数が既にある値に確定した状態にある。すなわち、この時点で与えられた1次元化後の方程式中にはもはや深さ i−1 番目までのループの制御変数は含まれない。そして、解探索の対象領域は、i=1 の場合のみ繰り返し空間(C行列)全体で、i>1 の場合には深さ i−1 番目までのループの制御変数がそれぞれの値に確定した部分空間(部分C行列)である。この部分C行列(以下 C_i^* と呼ぶ)の縦軸/横軸の両基本ベクトル(以下 bv_i と呼ぶ)は、深さ i 番目以上(すなわち、より深いループ)のループ制御変数に関するものである。

この条件下で深さ i 番目以上のループの制御変数の解を以下の手順で求める。

(3-1) 【解領域限定】現時点の1次元化後の方程式から、数式処理法の処理(7)から(9)を適用し、I_i-I'_i 平面($1 \leqq i \leqq r$)上での解の存在(可能)領域を限定する。同時に同領域の I_i および I'_i の最大値/最小値を求める。

(3-2) その際解なしの場合、その時点でリターン。さもなければ次へ。

(3-3) 【基本ベクトル限定】(3)で述べた基本ベクトル bv_i の両端部に、その I_i (I'_i)として(3-1)で得られた I_i (I'_i)の最大値/最小値で示される閉区間外の値をとる要素があればすべて取り除く。こうして限定された縦軸側と横軸側の両基本ベクトルの長さの和を Lbv_i^* とする。

(3-4) 【解法選択】次の条件を、この順序で調べ解法を選択する。

(条件a) 【再帰的探索終了?】i = r の場合には、(3-6)へ。さもなければ次へ。

(条件b) 【依存の種類確定済み?】上記(1)により深さ i+1 番目以上のループ制御変数に関するペア関係式が得られており、かつそのペア関係式から深さ i+1 番目以上のループ制御変数に関する依存の種類($I_k < I'_k$、$I_k = I'_k$、$I_k > I'_k$ のいずれであるのか、そしてさらに関連の2出現が定義-引用で、かつペア関係式が不等式の場合には2回運搬か多回運搬か、すなわち $I_k = I'_k \pm Inc_k$、$I_k < I'_k + Inc_k$、$I_k + Inc_k > I'_k$ のいずれであるのか;ただし Inc_k は深さ k 番目のループ制御変数の増分値、$i+1 \leqq k \leqq r$)が確定しているか。yes ならば(3-5)へ。no ならば次へ。

(条件c) 【計算量予測】(3-1)で得られた領域内の(I_i, I'_i)の解候補の組の個数(Sとする)を、I_i-I'_i 平面における同領域の面積で近似し求める。次に、(3)の記述同様に、今の状態から深さ i 番目のループ制御変数をある値に固定したときの部分C行列 $C_{(i+1)}^*$ を考える。複数個存在し得る各部分C行列 $C_{(i+1)}^*$ の縦軸/横軸の両基本ベクトルの長さの和を求め、それらを平均(平均値を $Lbv_{(i+1)}$ とする)する。そのとき条件

$$S \times Lbv_{(i+1)} < Lbv_i^*$$

が成立するか。yes ならば次へ。no ならば(3-6)へ。

(3-5) 【子問題設定】(3-1)で得られた領域内の(I_i, I'_i)の解候補の組を実際に求める。その組すべてについて、与えられた1次元化後の方程式に、I_i および I'_i それぞれの値を代入し、i を i+1 として再帰的に(3)を実行する。ただしこの再帰的な解探索で、もし解がみつからなければ順次 I_i および I'_i に次の候補の値の組を代入し探索を続ける。もし解がみつかれば、そして(3-4)の条件bで、深さ i+1 番目以上のループ制御変数に関する依存の種類が確定できていれば、解がみつかった値の組と I_i および I'_i に関する大小関係(依存の種類)が同じである未探索の解の候補の(I_i, I'_i)の値の組に関する探索を省略する。

(3-6) 【Sort-Merge 法適用】与えられた1次元化後の方程式に対して、(3-3)で限定した基本ベクトルを用いて Sort-Merge 法を適用し、現在再帰的に解いている子問題に対する深さ i 番目以上のループ制御変数の解と依存関係をすべて求めリターン。 □

このように、上記の複合アルゴリズムでは、数式処理法の時間コストが対象ルー

プの繰り返し回数によらず小さいことから、まず数式処理法を適用し解の存在(可能)領域を求めている。次に、その結果から(3-5)で再帰的に次の深さのループの制御変数に対して解を探索すべきか、(3-6)でSort-Merge法を適用すべきかを、処理コストを考慮し(3-4)において自動的に判定する。なお、この複合アルゴリズムを実現した際考慮している処理コストとは、現時点では数式処理法の処理のための時間コスト、(3-5)で解候補の組を実際に求めるための時間コスト、ならびにSort-Merge法の時間コスト/領域コストである。こうして最終的に確定された解は、すべて厳密な整数解である。すなわち、必要十分に不定方程式を解いたことになる。再帰的に次の深さのループの制御変数に対して解を探索するこの方法は、不定方程式に解の候補を代入し、**子問題**を解くことに相当する。

上記の(3-4)解法選択の条件bは通常のプログラムでは、多くの場合満足されると考えてよい。そして、その場合には、それ以前の種々のチェックをくぐりぬけてきた方程式にはほとんどの場合解が実際に存在するので、(3-5)において多数の解の候補について子問題を解くことを省略できる。

また、上記の(3-4)解法選択の条件cにおいては、計算量を概算する。そのために必要な計算はできるだけ簡略化している。まず、(3-1)で得られた領域内の(I_i, I'_i)の解候補の組の個数Sを求めるとき、例えば先の図5.4のIおよびI'の場合では、1次元化後の方程式(および繰り返し空間)

$$11\cdot I - 10\cdot I' - K' = 0 \quad (0 \leqq I,\ I',\ K' \leqq 2)$$

からK'を消去し、IおよびI'のペア関係式(および繰り返し空間)

$$0 \leqq 11\cdot I - 10\cdot I' \leqq 2 \quad (0 \leqq I,\ I' \leqq 2)$$

を得る。この不等式が示す領域内の解候補の組の個数Sは、この領域内の各格子点を含む面積1の正方形すべての面積の総和である。その階段状の境界を持つ領域の面積を、上記の不等式が示す領域を上下左右に1/2ずつ押し広げた領域の面積とみなす。つまり、不等式

$$-0.5 \leqq 11\cdot I - 10\cdot I' \leqq 2.5 \quad (-0.5 \leqq I,\ I' \leqq 2.5)$$、あるいは

$$0 \leqq 11\cdot I - 10\cdot I' \leqq 3 \quad (0 \leqq I,\ I' \leqq 3)$$

が示す領域の面積をSと近似する。しかも、I'の最大値/最小値の条件も無視し、平行四辺形とした上で、求めた面積の小数点以下を切り捨て、S＝3を得る。これは今の場合、実際の解の候補$(I, I') = (0,0), (1,1), (2,2)$の3とおりと合致する。次に、$Lbv_{(i+1)}$、すなわち、深さ1～i番目のループ制御変数の値を確定させたときの

部分C行列の両辺の長さの和の平均を求める。今の例では、図5.3からわかるとおり、C行列のサイズは全体では3×9である。制御変数IおよびI'(図5.3では正規化前のiおよびi'に対応)確定後の部分C行列C_2*のサイズは、$(i, i') = (1, 1)$のとき2×6、$(i, i') = (1, 2)$のとき2×3、$(i, i') = (2, 1)$のとき1×6、$(i, i') = (2, 2)$のとき1×3となる(Sort-Merge法のコストは、この縦横の長さの和に比例する)。すなわちアルゴリズム中の$Lbv_1 = 12$、$Lbv_2 = 6$となる。しかし、この計算においても、このようにすべての(今の場合4個の)部分C行列のサイズを計算するのではなく、簡略化し両基本ベクトルの端点の2個の部分C行列のみについて、すなわち、今の場合$(i, i') = (1, 1), (2, 2)$の場合のみから平均値$Lbv_2 = 6$を計算している。

この複合アルゴリズムでは、対象多重ループおよび添字式に関する制約を緩和するよう容易に拡張できる。すなわち、ループ制御変数以外の変数が含まれる場合には、元の数式処理法に相当する部分のみを適用する。その際には、深さi番目のループの制御変数の解を求めるとき、与えられた1次元化後の方程式中に深さi－1番目までのループの制御変数が消去できずに含まれていることもある。当然この場合には、必ずしも厳密な解析とはならない。また、非線形式の場合には、その非線形の項をあたかもループ制御変数以外の別の変数とみなすことにより、上記同様に数式処理法に相当する部分を適用し、その後当該非線形項がコンパイル時に確定する式であれば、当該非線形項も含めSort-Merge法を適用すればよい。Banerjeeが提案したアルゴリズム[13]あるいはそれと同様の従来の方式では線形の多項式の添字式を対象とするため、このような非線形式に関する厳密な解析は諦められていたのに対し、今回の方式にして初めて非線形式への拡張が可能となる。つまり、今回の複合アルゴリズムによれば、より一般的な添字式のクラスについてもデータ参照関係解析が行え、強力な自動ベクトル化技術となる。

この複合アルゴリズムにより、高速かつ必要十分に依存関係を求めることができる。その結果、従来の方式では十分条件で判定するため依存関係があいまいにしか求められず最深部のループのみしかベクトル化できなかった多数の多重ループについて、対象多重ループ全体をベクトル化できるクラスをさらに大きく拡大することができ、ベクトル化率を向上させることができる。

5.3.6 解による依存関係の判定

こうして得られた解のI_iおよびI'_i $(1 \leqq i \leqq r)$の大小関係から、5.3.2項で述べたC行列の概念を利用して、依存の向きを判定し、依存の類別を行う。なお、実際には

比較する各制御変数の値は、あるひとつの値とは限らず5.3.4項のSort-Merge法の解説で述べたとおり、一般には幅を持つ区間で表現される。従って、下記のアルゴリズムにおいて大小比較は、このような区間演算であるものとする。区間に重なりがあれば、$I_j < I'_j$、$I_j = I'_j$、$I_j > I'_j$ のすべての場合があるものとする。

[依存関係の判定アルゴリズム]

(1) 最外側のループの制御変数から内側のループの制御変数へ順に $I_j = I'_j$ 以外の大小関係が生じるものを探索する。あれば(2)へ。なければ、この解のC行列上での存在域は対角部であるので、5.2節で述べた単純変数の場合と同様に、2出現間の先行関係を求め、その順序を依存の向きとし、D行列(4.3.5項参照)に登録する(D行列の非例要素を付け加える)。依存の類別も5.2節同様に行うが、今の場合には、深さ1～rまでのすべてのループにおいてループ独立型であるので、表5.1でループ独立であるもののみをあわせて登録する。ただしこのとき、先述したとおり表5.1のダミー型をそうでないものに読み替えて登録する。

(2) 深さjのループで、$I_j = I'_j$ 以外の大小関係が生じたとする。

$I_j < I'_j$ ならば依存の向きは出現1→出現2である。逆ならば、出現2→出現1である。この向きで、D行列に登録する(D行列の非例要素を付け加える)。両方の大小関係があれば独立して両方ともに登録する。依存の類別は、それぞれの場合表5.1で先に判定した出現1→出現2あるいは出現2→出現1のもののみでよい。出現1および出現2が定義か引用かで、(定義-引用)、(引用-定義)、(定義-定義)は決定される。また、深さ1～(j−1)までのループにおいてループ独立型で、深さjのループでループ運搬型とする。(定義-引用)の場合には、さらに I_j および I'_j の差がちょうど当該ループの増分値に等しいかどうかも調べ、2回運搬依存かどうかも調べる。

(3) 深さj+1～rまでのすべてのループについて、さらに以下のようにループ独立型あるいはループ運搬型とする。

(3-1) $I_k = I'_k$ $(j+1 \leq k \leq r)$ については、(定義-引用)、(引用-定義)、(定義-定義)の区別は(2)と同じとし、ループ独立型として登録する。

(3-2) (2)と同じ向きの不等号が成り立つ場合には、(定義-引用)、(引用-定義)、(定義-定義)の区別は(2)と同じとし、ループ運搬型として登録する。(定義-引用)の場合には、(2)と同様2回運搬依存かどうかも調べる。

(3-3) (2)と逆向きの不等号が成り立つ場合には、逆向きの運搬依存であることを示すため(2)で(定義-引用)あるいは(引用-定義)であるときには、それぞれ逆の(引用-定義)型運搬依存あるいは(定義-引用)型運搬依存として登録する。(定義-引用)の場合には、(2)と同様2回運搬依存かどうかも調べる。(2)で(定義-定義)であった場合には、表4.15に示したoutmに対してroutm特別に用意された種別を登録する。 □

依存の向きは、3.2.2項で述べたとおり、自動ベクトル化を行う際には、必ず調べる必要がある。依存の類別は、同じく3.2.2項で述べたとおり、ループ選択等の高度なベクトル化のために必要となる。すなわち、3.2.2項で述べたとおり、ループ運搬型の依存は、それを引き起こすループをスカラ実行すれば無視できる。次の例1で

[例1]多重ループ内のデータ依存(その1)

```
      DO  10  I=1,3
        DO 10    J=1,3
          ····=··A(J,I)··        (S1)
          A(J,I+1)=····          (S2)
10    CONTINUE
```

は、配列Aについては(定義-引用)型で文S2→文S1の向きの、制御変数Iではループ運搬型、制御変数Jではループ独立型の依存が生じている。従って、制御変数Iのループをスカラ実行すれば、この配列Aの依存は無視できる(3.2.2項の例4および例5参照)。同様に次の例2では、配列Aの依存については例1と異なるのは制御変

[例2]多重ループ内のデータ依存(その2)

```
      DO  10  I=1,3
        DO 10    J=1,3
          ····=··A(J,I)··        (S1)
          A(J+1,I+1)=····        (S2)
10    CONTINUE
```

数Iおよび制御変数Jともにループ運搬型となる点である。このような場合には、制御変数Iのループおよび制御変数Jのループともにスカラ実行すれば、この配列Aの依存は無視できる。つまり、複数のループで運搬される依存は、それらのループす

べてをスカラ実行してはじめて無視できる。また、次の例3では制御変数Iのループ

[例3]多重ループ内のデータ依存(その3)

```
      DO  10  J=1,3
          DO  10    I=1,3
              ····=··A(J,I)··        (S1)
              A(J,I+1)=····          (S2)
10        CONTINUE
```

および制御変数Jのループの入れ子構造が例1と逆転している。この場合3.2.2項で述べたとおり、ループの入れ子構造を逆転(交換)して例1のようなプログラムとした後、制御変数Iのループをスカラ実行すれば、制御変数Jのループでベクトル化できる。これに対し、次の例4では、参照が衝突する際、制御変数Iおよび制御変数Jの

[例4]多重ループ内のデータ依存(その4)

```
      DO  10  J=1,2
          DO  10    I=1,2
              ····=··A(J,I)··        (S1)
              A(J-1,I+1)=····        (S2)
10        CONTINUE
```

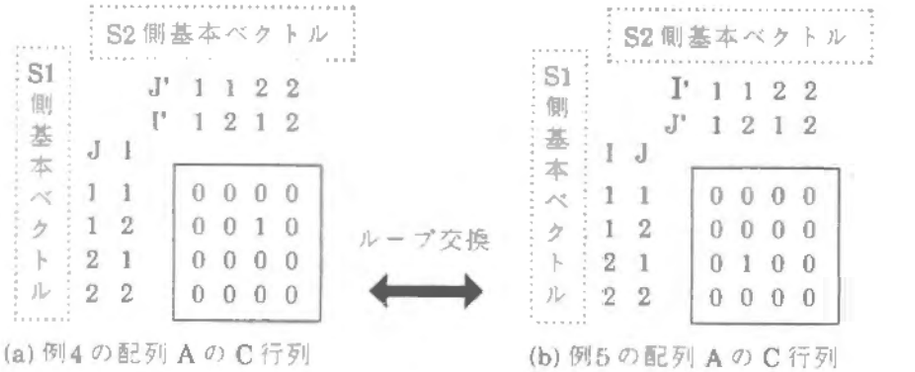
ベア関係式で不等号の向きが逆である。すなわち、上記のアルゴリズムで(3-3)が生じる例である。このような場合には制御変数Iのループおよび制御変数Jのループの入れ子構造を逆転させることはできない。なぜならもし逆転させた場合には例5のよ

[例5]多重ループ内のデータ依存(その5)

```
      DO  10  I=1,2
          DO  10    J=1,2
              ····=··A(J,I)··        (S1)
              A(J-1,I+1)=····        (S2)
10        CONTINUE
```

うになるが、例4では上記のアルゴリズム(2)から依存の向きは文S1→文S2の向きであるのに対し、例5では依存の向きは文S2→文S1の向きで、逆転してしまうからである。この現象はC行列上で考えると、制御変数Iのループおよび制御変数Jのループの入れ子構造を逆転したことにより、基本ベクトルの要素の順序が入れ替わ

り、そのためC行列の行および列交換が引き起こされ、上三角と下三角の要素が入れ替わってしまったとみることができる(図5.8参照)。従って、この例4のような制御変数によってベア関係式で不等号の向きが異なる依存は、ループ交換を伴うベクトル化ループの選択を行う際に、逆向きの不等号の成り立つベア関係式となる制御変数(例4のI)を持つループを他の不等号の成り立つベア関係式となる制御変数を持つループすべてを飛び越して最外側に出すことはできない。このアルゴリズムの(3-3)の情報を使えば、このようなループ交換を伴うベクトル化ループの選択をも正しく処理することができる。



(a) 例4の配列Aの C行列　　(b) 例5の配列Aの C行列

(注) これらのC行列では、対角要素そのものが対角部となる。

図5.8 例4および例5の配列Aの2出現のためのC行列の比較

5.3.7 自己依存関係

配列化を行わない場合、次の例6のような1定義について、**自己依存**(self-

[例6]多重ループ内のデータ依存(自己依存)

```
      DO  10  J=1,3
          DO  10    I=1,3
              A(I)=····              (S1)
10        CONTINUE
```

dependence)をも調査しておく必要がある[14]。今まで述べてきたデータ依存はすべて2出現間でのループの繰り返しに伴うデータ参照関係を保存する目的であったのに対し、この自己依存はループのある繰り返しにおける定義が、その定義参照のそれ以前の繰り返しにおける定義とデータ参照関係があるか否かを調べるものである。

その意味で、定義-定義型のデータ依存関係解析の特殊な場合であるといえる。ただし、当該定義が属する多重ループすべてについてループ独立型である自己依存(先の例6A(I)ではなくA(I,J)のような場合)の場合は、同一の配列要素が複数回定義されることはないので、保存すべきデータ参照関係はない。つまり、この自己依存についてはループ運搬依存のみが問題となる。先の例6に戻れば、制御変数Jのループの繰り返しにより、同一の配列要素が複数回定義される、すなわち、制御変数Jのループにより生じるループ運搬依存が問題となる。

従って自己依存の解析はまず、例えば先の例6でいえば、文S1があたかももうひとつ存在するかのように考えて、A(I)への二つの定義について、今まで述べてきた添字解析等のアルゴリズムをまったく通常の2出現同様に適用する。存在すると判定された種々の型の依存からループ独立型のものをすべて無視し取り除く。その上でもし、何らかの型の依存があればそれをD行列に、出現1から出現1への向きの、すなわちある中間コードからそれ自身への依存として登録する。すなわち、この自己依存は、D行列の対角要素に非零要素として登録されることとなる。もちろん実際には、この登録もD行列登録用の手続が異なる中間コード間の依存の登録と同様に、統合的に処理している。

先に述べたとおりループ運搬依存は、従ってそれのみからなる自己依存は、その運搬を起こしているループをスカラ実行する場合には無視できる。例えば、先の例6では制御変数Jのループがスカラ実行されれば、制御変数Iのループがスカラ実行されてもベクトル実行されても、J＝1のとき定義されたA(1)からA(3)が、J＝2そしてJ＝3のときと順に定義し直され元のプログラムのこの定義に関するデータ参照関係は保存される。しかし、V-Pascalのように多重ループ全体を間接参照を用いて一重化しようとすると、自己依存は無視できない。言い換えれば、一般にはある定義について自己依存が存在すると、すくなくともその定義についてはその定義が属する多重ループすべてについてベクトル化することはできない。ところが、V-Pascal Version 1のターゲット・マシンであるHITAC S-820(ならびにS-810)には、このような参照順序を保存する間接参照ベクトル・ストア命令が用意されているため、自己依存が存在しても多重ループすべてについてベクトル化が可能となっている。しかし、この特別な間接参照命令を利用すべきか否かの判定は自己依存の有無で行うので、結局ターゲット・マシンにかかわらず自己依存関係解析は欠かすことができない。

5.4 制御関係解析

3.2節で述べたとおり自動ベクトル化技術では、5.2節および5.3節で述べたデータ依存関係解析と、制御依存を調べる解析が2本の重要な柱となっている。以下では、後者の制御関係解析のための新しい方法を提案するものであって、次の三つの事項を対象としている。

(I)条件分岐での制御の流れによる先行関係から生ずる、中間コード群間の制御依存。例えばif条件節の実行は、対応するthen節およびelse節の実行より必ず先行しなければならないという制約から生ずる依存である。

(II)ネストしたif文の各レベルのthen節およびelse節に対応してマスクベクトルを生成する際、どの条件分岐により直接制御を受けるかの判定。これを**直属の条件分岐の解析**と呼ぶ。

(III)前記のデータ依存関係解析において調査の対象となる同一変数の2出現の間に生ずる制御依存。

なお、これらの解析においては、Pascalのfor文、while-do文およびrepeat-until文は、中間コードから抽出したコントロールフロー・グラフ(4.3.3項参照)ではif-gotoと同様に表現されるので、これらの制御構造から生じる制御関係も、当然通常のif文から生じる制御関係と区別なく調べられる。また上の項目(III)は、5.2節および5.3節で述べたデータ依存関係解析を補完するものである。すなわち、データ依存関係解析において対象となる同一変数の2出現間のスカラ実行時の先行関係を調査するものである。

3.3.6項で述べたとおり実機上では、if文は次の3種類の方式でベクトル実行される。

a)マスク付きベクトル命令使用

b)ベクトルデータ編集(収集-拡散、収集-分散、圧縮-伸長等各社で命令の呼称は異なる)命令使用

c)間接参照(間接指標、リストベクトル等各社で命令の呼称は異なる)命令使用

以上3方式いずれの場合にも、まずマスクベクトルを生成しなければならない。すなわちifの条件節が、then節およびelse節に先行して、ループの各繰り返しについて実行されなければならないことを意味する。これが、(Ⅰ)として挙げた制御依存である。これについては5.4.1項で詳述する。if文による条件分岐がループ外に飛び出すような場合、さらにこの解析は複雑となる。しかもこの解析はベクトル化可能性の判定に大きく影響する。従って、if文のベクトル化可能性の判定には、この制御依存の解析が必須となる。また、マスクベクトルを生成するには、ある処理がネストしたif文の中で、どのif文のthen節あるいはelse節に含まれるかを決定できなければならない。これは、(Ⅱ)として示した制御関係解析である。これについては5.4.2項で詳述する。そして、この情報を用いて、コントロールフロー・グラフの各頂点は、4.3.4項で述べた同一の条件分岐の影響を真偽同じように受ける頂点の集合であるプライマリ・セットの集合へまとめられる。つまり、4.3.4項で述べたプライマリ・セット表を作成する。以下(Ⅰ)および(Ⅱ)をまとめ**条件分岐の影響解析**と総称する。以上述べてきたとおり、この条件分岐の影響解析はデータ参照関係に起因する依存関係とともに、自動ベクトル化コンパイラに必須の技術である。

以下ではV-Pascalコンパイラに組み込まれている上記の3種類の制御関係解析(Ⅰ)、(Ⅱ)、(Ⅲ)を、通常の最適化コンパイラが行う大域的データフロー解析(4.7節参照)により同時に解析する新しい手法について述べる。

4.7節で述べたデータフロー集合のうちの表4.18の4種は、ある定義が制御の流れ(コントロールフロー・グラフの有向辺)に沿って、どの頂点に到達するか、どの頂点には到達しないかを調査するためのものである。大域的データフロー解析のこの特徴に着目し、特殊な定義を仮想的に発生あるいは消滅させることにより、これら3種類の制御関係解析(Ⅰ)、(Ⅱ)、(Ⅲ)を行うことが本手法の新しい特色である。以下では、これらのアルゴリズムを説明のため個別の解析として記述するが、実際には4.7節で既述した大域的データフロー解析の際、すべて同時に実行している。なお、これらのアルゴリズム記述では、すべての頂点の各データフロー集合が、この大域的データフロー解析手法にのっとり、適切に初期化されていることを仮定している。また、当然大域的データフロー解析にも必要な4.6節で述べたコントロールフロー解析が終了しているものとする。以下のアルゴリズムにおいては、その中でも特に4.6.1項で述べた支配関係解析の結果を使用する。

以下では「発生させる」とはこのデータフロー集合のgen集合に要素として含めることをいい、同様に「消滅させる」とはkill集合に含めることを意味するものとする。

5.4.1 制御関係解析(Ⅰ):条件分岐から生ずる制御依存解析

まず、if文のベクトル化に必要な制御の流れの分岐により生ずる依存の解析方法について述べる。この依存は、あるif文の条件節の実行は、その制御を受ける部分すべてに対して、先行される必要があることから生ずる。従って、制御の流れを抽出したコントロールフロー・グラフにおいて、ある条件節の制御を受ける頂点をすべて選び出せば、必然的にそれらの各頂点に対応する中間コード群へ、条件節の頂点に対応する中間コードからの依存が存在すると解析できる。すなわち、コントロールフロー・グラフ上での、各条件分岐の制御を受ける範囲の解析である。なお、先に述べたとおり、ここでの手法はif条件節に限らず、広く一般の条件分岐すべてを含めて統一的に解析できる。

例えば、図5.9ではあるコントロールフロー・グラフについて、分岐型バーテクスDの制御を受ける頂点群を求めるため、仮想的な定義ⓓおよびⓓ'を頂点Dの真側および偽側それぞれに対応させ用意する。それぞれの定義を頂点Dの真側および偽側それぞれの後続の頂点Iおよび頂点Eのgen集合に含める。すなわち、これらの頂点で発生させる。同時に、両者ともに頂点Dの直接逆支配頂点Eのkill集合に含める。すなわち、この頂点で消滅させる。次にデータフロー方程式を解く。その結果、これらの定義が到達する頂点(同図中ⓓと付記した頂点群)、すなわち、これらの定義をin集合の要素に持つ頂点はすべて、かつ、それらの頂点のみが頂点Dの制御を受けると判定できる。厳密な手順は以下のアルゴリズムに譲るが、同図の頂点I、頂点E等仮想的な定義をgen集合、kill集合に含めるべき頂点は特別に処理する。同図では同様に、頂点Fの制御依存解析のために仮想的な定義ⓕおよびⓕ'も使用している。さらに、異なるコントロールフロー・グラフでの解析例を図5.10に示す。

[条件分岐から生ずる制御依存解析アルゴリズム]

(1) 各分岐型バーテクスにユニークな識別子を持つフラグ変数を真側出口に1個、偽側出口に1個与え、各分岐型バーテクスからでる真側および偽側の有向辺それぞれに連なる次の頂点において仮想的に当該真側用あるいは偽側用フラグ変数に1回だけ定義が発生したものとし、同頂点のgen集合に含める。次の5.4.2項の制御関係解析(Ⅱ)のために用いる仮想的な定義と区別するため、仮想的に発生させるこの定義を**第1の仮想的な定義**と呼ぶ。

(2) 分岐型バーテクスPの直接逆支配頂点(4.6.1項参照)において、(1)で用意した当該分岐型バーテクスに対応するフラグ変数が仮想的に再定義しなおされたものとし、(1)でgen集合に含めた仮想的な定義を、この直接逆支配頂点のkill集合に含める(消滅させる)。もちろん、この再定義自身は、以後の解析に不用であり、この直接逆支配頂点のgen集合に含める必要はない。なお、図5.9の頂点

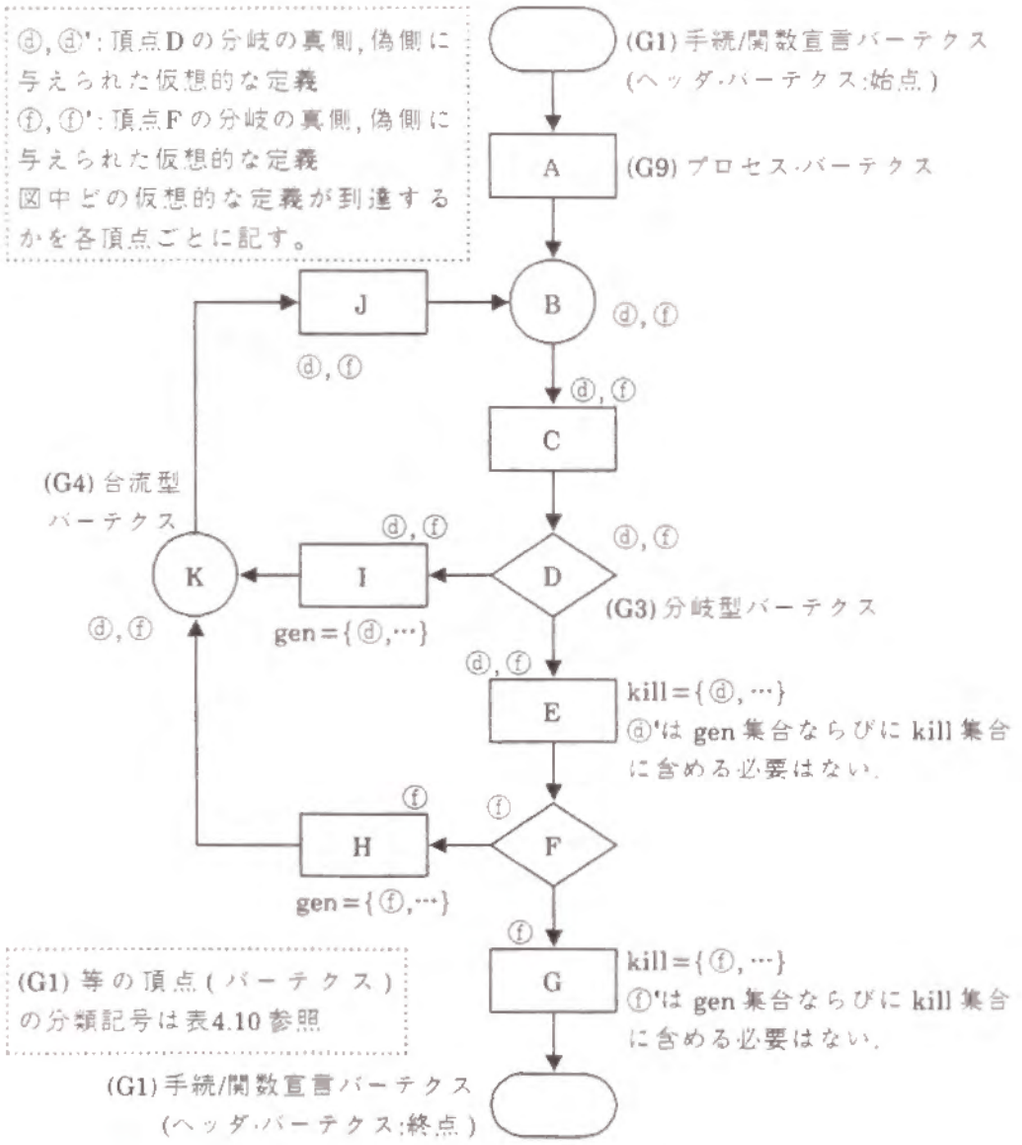


図5.9 図4.23の例における第1の仮想的な定義による制御依存解析

E、頂点Gのように定義をgen集合に含むべき分岐型バーテクスの次の頂点が、当該分岐型バーテクスの直接逆支配頂点となることがある。この場合には、同図のⓓ'およびⓕ'で例示したとおり、むだな処理を避けるため、仮想的定義をgen集合に含めない。従って、kill集合にも含める必要はない。

(3) 他の定義を合わせデータフロー方程式を解く。

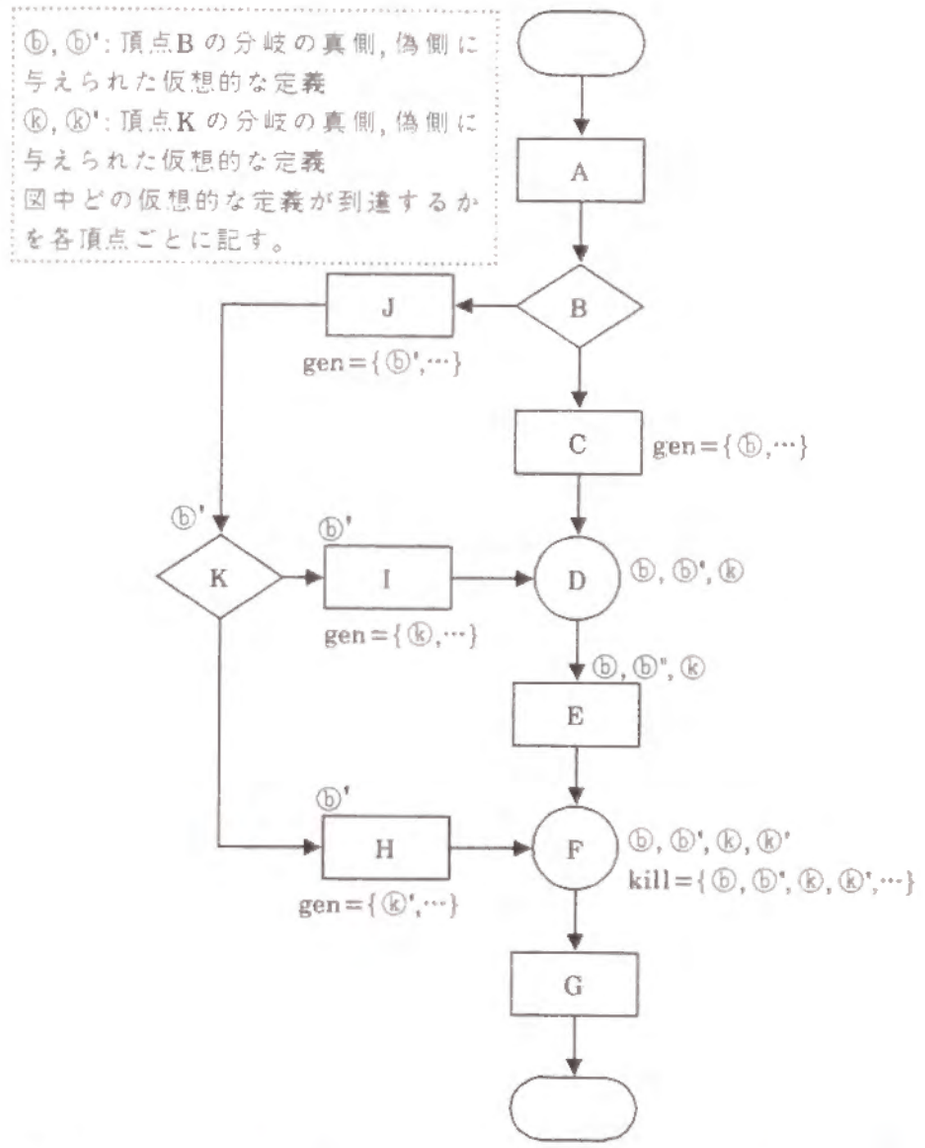


図5.10 if-then-else構造でないコントロール・フロー・グラフの制御依存解析

(4) 各データフロー集合が確定した後、仮想的な定義についてのみ、各頂点のin集合を集合(in ∪ gen − kill)で置き換える。すなわち、仮想的な定義を発生させた頂点へも依存があることを統一的に処理するためgen集合を含めて(in ∪ gen)とする。一方、各分岐型バーテクスの直接逆支配頂点自身にも、仮想的な定義が到達する。ところが、ある分岐型バーテクスの直接逆支配頂点は、その条件分岐により複数に分かれた制御の流れが再度初めて1本にまとまる頂点である。それゆえ、直接逆支配頂点自身は条件分岐の影響をもはや受けないので、依存なしとして処理する。そのため、kill集合の要素を取り除き、当初のin集合にかえて(in ∪ gen − kill)とするのである。

(5) 上記(4)の置き換え後、単純に各頂点のin集合を調査し、仮想的な定義が到達する頂点群のみが、その定義に対応する分岐型バーテクスの制御を受けると判定できる。図5.9の頂点D、頂点Fのようにループに含まれる分岐型バーテクスにも、それぞれの自身の仮想的な定義が到達し、正しく判定できる。 □

仮想的なフラグ変数に対する最初の定義は、当該条件分岐の制御を受ける全頂点に到達する。しかも、制御を受けない頂点には到達しない。図5.9の頂点E等の逆支配頂点自身に対する例外的な処理は上記(4)の置き換えで簡単に一般の場合に含めてしまうことができる。このようにして、フラグ変数の定義が到達するかどうかを調べることにより、制御の流れによる依存関係をデータフロー解析と同時に統一的に処理し求めることが可能となる。

なお、(1)では真側、偽側2種類の仮想的な定義を与えたが、単に制御依存を求めるだけであれば次の(1')のように簡単化できる。

(1') 各分岐型バーテクスにユニークな識別子を持つフラグ変数1個を与え、各分岐型バーテクスにおいて仮想的にそれぞれの頂点に対応するフラグ変数に定義が発生したものとし、gen集合に含める。

こうすると、仮想的定義をgen集合に含む頂点への制御依存はないので、(4)の処理ではin集合にかえて(in − kill)とするだけでよい。しかし、インプリメントにあたっては先のアルゴリズムを採用した。先のアルゴリズムは次の制御関係解析(II)の処理と共用できる部分が多く、インプリメントが容易になるためである。また、その結果得られるより豊富な情報を基に制御関係解析(II)の結果の検証をも行っている。

この条件分岐ごとにその真側/偽側にそれぞれ用意する第1の仮想的な定義のデータフロー集合における要素番号は、真側/偽側それぞれ(G3)分岐型バーテクスおよび(G7)ループビギン・バーテクスのPDEFLTフィールドならびにPDEFLFフィールド(表4.12参照)に登録される。また、この制御関係解析(I)のために用意された第1の仮想的な定義のデータフロー集合における要素番号すべてを、(G1)手続/関数宣言バーテクスのPREDVIRTDEFフィールド(表4.12参照)に集合として記録している。

ここで提案した手法によれば、if文によりループ外へ制御が飛び出す場合も含め、どのように複雑な制御構造を有するプログラムを与えられても正しく、しかも容易に解析できる。

5.4.2 制御関係解析(II):直属の条件分岐の解析

先に述べたとおり、if文のベクトル化では、ある処理がどういうマスクベクトルの制御下に実行されるのかを決定できなければならない。必要となるマスクベクトルは個々の中間コードごとに定義されるが、効率を考慮しコントロールフロー・グラフの頂点を単位に解析すれば十分である。これは、先の制御関係解析(I)とは逆の対応の調査である。一般にif文がネストすることから、マスクベクトルは制御を受けるif条件節それぞれの結果をブール変数としたブール式で表現できる。このブール式による表現を文献[15]ではガード(guard)式と呼んでいる。ところが、複雑に制御の流れが絡み合う任意のプログラムにおいて、ある頂点が制御を受けるこのブール式を求めることは容易ではない。5.4.1項で用いた仮想的な定義からの情報を使っても、特に例えば図5.10のような単純なif-then-else構造以外では、条件分岐の影響範囲のネスト構造とそれに付随するブール式を簡単には決定できない。図5.9のように条件分岐によりループを形成する場合には、さらに問題が複雑となる。

従って、次に条件分岐の影響範囲のネスト構造を調査するため、コントロールフロー・グラフの各頂点において、直接制御を受ける直属の条件分岐のみをまず求め、その直属の条件分岐自身が同じく直接制御を受ける条件分岐を順にたどり、所期の目的のブール式を得る手法について述べる。この手法は、前節同様本来の目的であるifによる分岐に限らず、広く一般の条件分岐を対象とする。具体的には、次に示すとおり、5.4.1項の仮想的な定義に似た第2の仮想的な定義を、各条件分岐ごとに用意して、それを用いて解析する。この第2の仮想的な定義は、5.4.1項で解析に使

用した定義とは全く独立である。すなわち、5.4.1項で各分岐型バーテクスごとに真側、偽側に定義を流す2個のフラグ変数を用意したのと同様に、それらとは別にさらに2個のフラグ変数を与えるのである。これらを使い、マスク生成のための情報であるブール式を得る際に真側、偽側を区別することが可能となる。第1の仮想的な定義同様に発生および消滅させるこの第2の仮想的な定義が、第1の仮想的な定義と異なる点は、以下に示すとおり何度も消滅ならびに再発生を繰り返すことである。解析例図5.11で解説する。図中の下線部が第1の仮想的な定義と異なる処理である。すなわち、仮想的な定義ⓓ、ⓕが分岐型バーテクスDに到達した瞬間、同頂点にてこれらの定義をkill集合に含めて消滅させ、同時に、その分岐型バーテクスDの直接逆支配頂点Eにて再度gen集合に含める。分岐型バーテクスFでは到達した仮想的な定義ⓕをkill集合に含め消滅させる。こうすることにより、多重にネストした条件分岐により複雑に影響を受ける場合でも、ある頂点に到達する第2の仮想的な定義は、その多重にネストした条件分岐の最内側の直属の条件分岐に対応する定義のみとなる。図5.12では、図5.10のコントロールフロー･グラフにおいて、第2の仮想的な定義がどのように処理されるかを示す。

[第2の仮想的な定義に関する処理アルゴリズム]

(1) 各分岐型バーテクスにユニークな識別子を持つフラグ変数を真側出口に1個、偽側出口に1個与え、各分岐型バーテクスからでる真側および偽側の有向辺それぞれに連なる次の頂点において仮想的に当該真側用あるいは偽側用フラグ変数に定義が1回だけ発生したものとし、同頂点のgen集合に含める。ただし、5.4.1項の場合同様、発生させる頂点と次のステップ(2)でkill集合に含める頂点が重なる場合には、gen集合ならびにkill集合に含めない。

(2) 各分岐型バーテクスの直接逆支配頂点において、(1)でgen集合に含めた仮想的な定義をkill集合に含める。

(3) 大域的データフロー解析を行い、仮想的な定義を流す。その際、5.4.1項の第1の仮想的な定義と異なり、もし自分自身も含め、ある分岐型バーテクスPに到達した定義は、1度その分岐型バーテクスPでkill集合に含めて消滅させ、分岐型バーテクスPの直接逆支配頂点においてgen集合に含めて再度発生させる。ただし、上述の1)のステップ同様、その再発生させる頂点がステップ2)に記した本来消滅させるべき自分自身の直接逆支配頂点の場合にはgen集合に含めない。

(4) 各データフロー集合が確定した後、この仮想的な定義を分岐型バーテクスPのkill集合から取り除く。これは、頂点Pには、定義が到達したものとして以後の処理を統一的に行うためである。

(5) その後、第1の仮想的な定義同様、仮想的な定義についてのみ、各頂点のin集合を集合(in ∪ gen − kill)と置き換える。 □

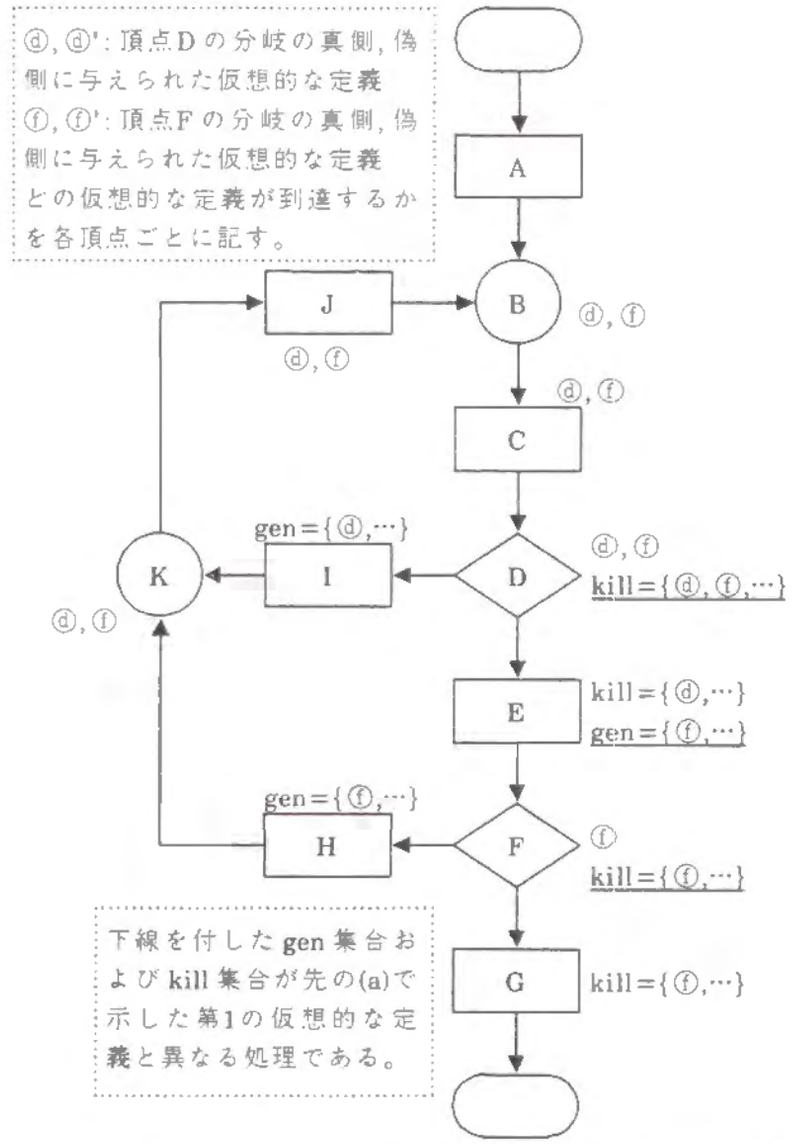


図5.11 図5.9の第2の仮想的な定義による直属の条件分岐解析

ある分岐型バーテクスPのための第2の仮想的な定義は、上述のとおり、ある分岐型バーテクスP'に到達するとそこで消滅し、その分岐型バーテクスP'の直接逆支配頂点で再度発生する。すなわち、より深くネストした分岐型バーテクスP'の影響範囲内の頂点群にはこの定義は到達しない。そして、分岐型バーテクスPが直接影響

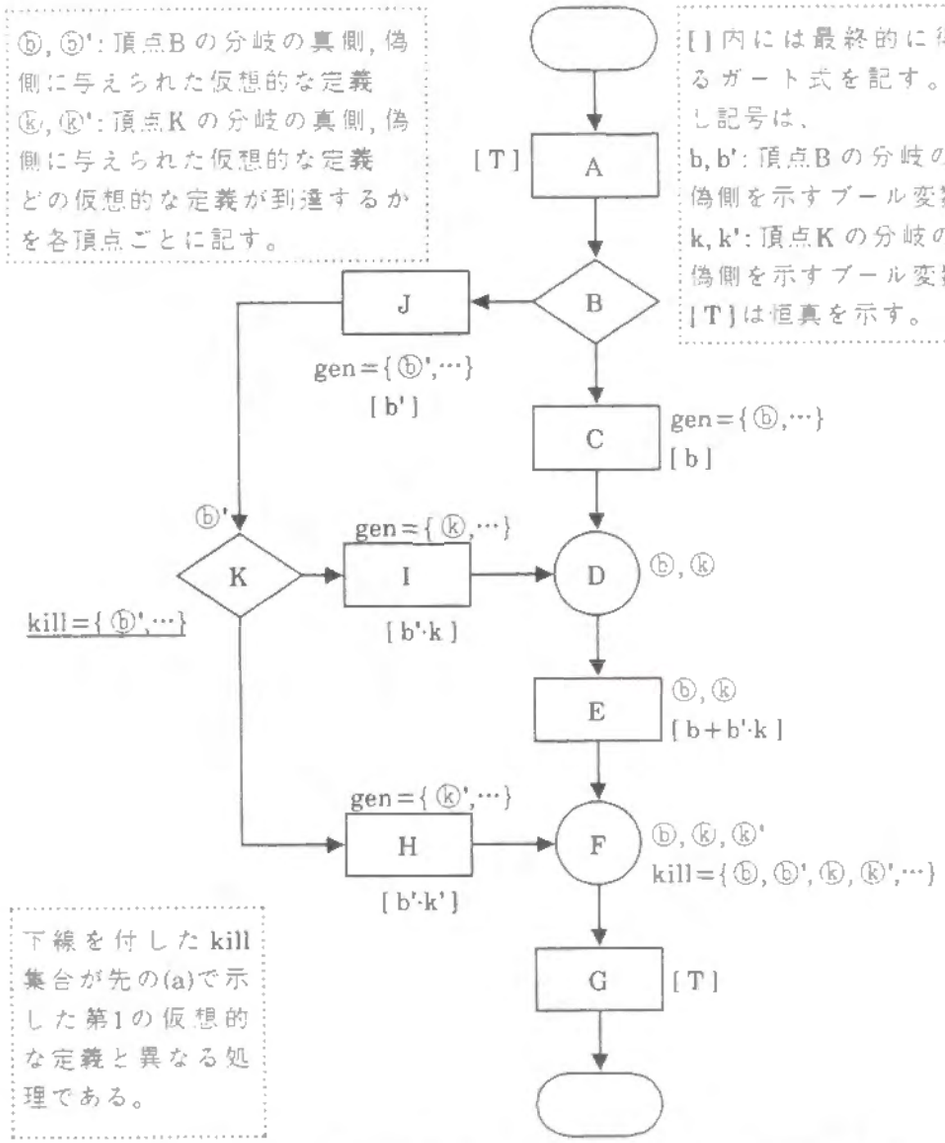


図5.12　図5.10の第2の仮想的な定義による直属の条件分岐解析

する頂点群にのみ到達する。この性質により、各頂点ごとに先に述べた直属の条件分岐を簡単に決定できる。具体的に制御を受けるブール式を完全に求めるためには、以下の手順で行う。このアルゴリズムを述べる前に図5.12を用い簡単に解説する。頂点Eについて考える。頂点Eには第2の仮想的な定義ⓑおよびⓚが到達する。すなわち、頂点Eの実行を直接制御するのは、分岐型バーテクスBおよびKであり、しかも、頂点Eはこれらの両頂点それぞれの真の側の経路に存在することがわかる。ここで、二つの分岐型バーテクスの制御を受けるということは、両者がどちらも直属であることから、両者がネスト関係にはなく、両者から独立に制御を受けることを意味する。すなわち、(分岐型バーテクスBが真)OR(分岐型バーテクスKが真)が真の時実行されると解析できる。一方、分岐型バーテクスKには仮想的な定義ⓑ'が到達する。このことから、頂点Kは分岐型バーテクスBが偽の時実行されることがわかる。これらを合わせると頂点Eのガード式は図中の[ ]内に示すブール式で表されると解析できる。

[直属の条件分岐およびガード式解析のためのアルゴリズム]

(1) 第2の仮想的な定義のうち、到達する定義に対応するブール変数すべてのORをとる。もし、仮想的な定義が一つも到達しない場合には、プログラムの実行時において必ず1度は実行される部分であり、T(恒真)とする。この直属の条件分岐に対応するブール式を以下では単に**直属ブール式**と呼ぶことにする。

(2) ステップ(1)で1個以上の条件分岐の影響を受ける場合には、次のように順にそれぞれの条件分岐自身のステップ(1)で求めた直属ブール式をANDで結合する。例えば、ある頂点Vの直属ブール式中に、ブール変数pが現れ、それは分岐型バーテクスPがに対応するとすると、ブール変数pを(分岐型バーテクスPの直属ブール式)·pに置き換える。ただし、この結合の際、頂点Vが分岐型でループを形成し自分自身の影響を受ける場合(図5.11で頂点Dおよび頂点Fがその例)、つまりV＝Pの場合には、無限にAND結合を続ける、すなわち無限に展開することになる。これを避けるため、このような場合には再帰的なブール表現となることを示す特別な印を付けるだけとする。

(3) ステップ(2)を再帰的に結合すべき変数がなくなるまで繰り返す。　□

実際にコンパイラ中にインプリメントされた手法は、各頂点ごとに上記の手法を繰り返すのではなく、ステップ(1)の直属ブール式解析の処理のみを行っている。ス

テップ(2)以降のガード式解析処理は部分ベクトル化を行うモジュールで、必要になった頂点のみに対して行えばよく、その場合にはループはベクトル化されるため上述のような無限展開のような状況は生じない。このループを形成する場合に、分岐型バーテクスがそれ自身のブール変数をガード式中に有することは、前回のループ実行時の自分自身の条件分岐の方向に、自分自身の実行が依存しているという当然の事実を示すものであり、正しい解析結果である。厳密にはあるループ実行時の自分自身の条件分岐に対応するブール変数 p_i とその前回のループ実行時の自分自身の条件分岐に対応するブール変数 p_{i-1} とは区別すべきものであり、ブール変数 p_i がブール変数 p_{i-1} の関数となっているみることができる。

ステップ(1)の直属ブール式解析の処理でコントロールフロー･グラフの各頂点は、直属ブール式により分類できる。すなわち、この情報を用いて、4.3.4項で述べた同一のガード式を有する頂点の集合であるプライマリ･セットの集合へまとめられるわけである。つまり、4.3.4項で述べたプライマリ･セット表をステップ(1)の直属ブール式解析の処理の後作成する。具体的には次の5.4.3項で述べる。

この条件分岐ごとにその真側/偽側にそれぞれ用意する第2の仮想的な定義のデータフロー集合における要素番号は、真側/偽側それぞれ(G3)分岐型バーテクスおよび(G7)ループビギン･バーテクスのPDEFLT2フィールドならびにPDEFLF2フィールド(表4.12参照)に登録される。また、この制御関係解析(II)のために用意された第2の仮想的な定義のデータフロー集合における要素番号すべてを、(G1)手続/関数宣言バーテクスのPRED2VIRTDEFフィールド(表4.12参照)に集合として記録している。

このように、この場合にも仮想的な定義を用いることにより、複雑に条件分岐がネストしたプログラムについても容易に解析できる。

5.4.3 プライマリ･セット表の作成

先の5.4.2項で述べた直属ブール式から、プライマリ･セットへ分解することができる。その理由は以下のとおりである。まず、プライマリ･セットは、4.3.4項で述べたとおり同一のガード式を有するコントロールフロー･グラフの頂点の集合であった。そこで、もし異なる直属ブール式を有する2頂点が同一のガード式を取り得たと仮定する。ところが、直属ブール式はブール変数のOR結合しか発生しない、しかもその結合の順序は仮想的な定義のデータフロー集合の要素番号順であるため一意的に定まる。従って、この仮定を満たすのは、5.4.2項で述べたガード式を求めるアルゴ

リズムにおいて、ステップ(2)以降のAND結合の順序が交換している場合である。AND結合の順序が交換するということは、条件分岐のネスト関係がどこかで逆転することを意味する。このようなことは、ループを形成する条件分岐と当該ループ内に存在する条件分岐とで疑似的に生じるだけである。すなわち、実際には生じない。なぜなら、ループを形成する条件分岐では5.4.2項で述べたとおり、厳密にはあるループ実行時のループを形成する条件分岐に対応するブール変数 p_i とその前回のループ実行時の自分自身の条件分岐に対応するブール変数 p_{i-1} とは区別すべきものである。しかし、実際には5.4.2項で述べたとおりループを形成する条件分岐に対応するAND結合は展開されないため、このような区別は不要であり、かつまた、AND結合の順序が交換するということは生じない。結局先の仮定を満たす場合はなく、従って、直属ブール式が同一の場合にのみ同一のガード式となりうるといえる。このとから、プライマリ･セットは、同一の直属ブール式を有するコントロールフロー･グラフの頂点の集合と言い換えることができるからである。

以上の考察から、5.4.2項で述べた直属ブール式をコントロールフロー･グラフの各頂点について求めた時点で、順次おのおのの頂点を走査しつつ同一の直属ブール式を有するかどうかで振り分け、プライマリ･セットに分割する。すなわち、

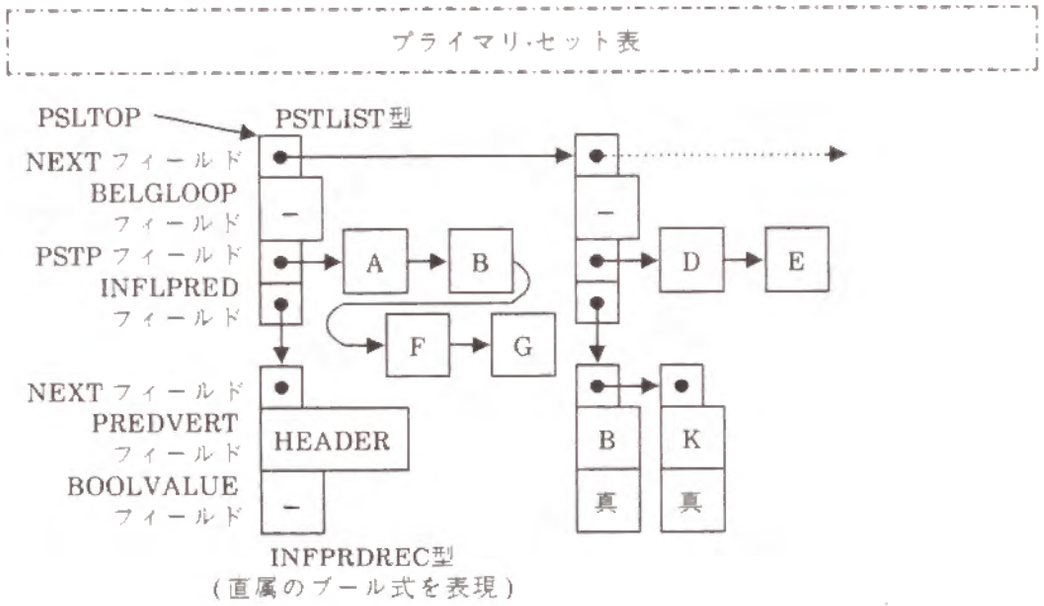
(A) 新たな直属ブール式を持つ頂点が出現した場合には、4.3.4項で述べたプライマリ･セット表に新しいエントリ(PSTLIST型)を追加し、そのエントリのPSTPフィールドに、当該頂点を登録する。

(B) そうでなく、既に同一の直属ブール式を有する頂点群のエントリがプライマリ･セット表に存在すれば、単にそのエントリのPSTPフィールドのリニア･リストに、当該頂点を追加する。

これと同時に処理したコントロールフロー･グラフの頂点に到達する5.4.1項で述べた第1の仮想的な定義を調べることにより、この頂点に制御依存関係を持つ条件分岐を調べ、その条件分岐に相当する(G3)分岐型バーテクスあるいは(G7)ループビギン･バーテクスにこの制御依存関係を登録する。この場合、プライマリ･セットは同一のガード式を有するコントロールフロー･グラフの頂点の集合であることから、ある頂点に制御依存関係を持つ条件分岐は同様に当該頂点の属するプライマリ･セットに含まれる全頂点に制御依存関係を持つ。ゆえに、この制御依存関係を登録にあたっては、コントロールフロー･グラフの頂点を単位とするのではなくプライマリ･セット

を単位とすれば十分である。従って、(G3)分岐型バーテクスではPPSLSフィールド(表4.12参照)に、(G7)ループヒギン・バーテクスではLPPSLSフィールド(表4.12参照)にこのプライマリ・セットを単位とした制御依存関係を登録する。なお、この登録の際に5.4.1項で述べたとおり、第1の仮想的な定義でも条件分岐の真側/偽側を区別できることを使い、5.4.2項で述べたアルゴリズムで得られたブール式に誤りがないか検査している。そして、プライマリ・セットごとに到達する第1の仮想的な定義の集合をINFLALLPフィールドに記録している。

なお、上記(A)でプライマリ・セット表に新しいエントリを追加する際には、当該エントリのINFLPREDフィールドにそのプライマリ・セットの直属ブール式の情報を登録する。具体的には、同フィールドは直属ブール式が条件分岐に対応するブー



(注)PSTPフィールドが形成するリニア・リストは実際にはコントロールフロー・グラフの頂点をさすポインタのリストであるが、説明のため頂点の名称を記した。そして、リンク・フィールドは省略し矢のみを記した。同じくPREDVERTフィールドも実際には頂点をさすポインタである。なお、HEADERは(G1)手続/関数宣言バーテクスの意味である。

INFPRDREC型は実際には、各ブール変数に相当する第2の仮想的な定義の番号を保持するPVDEFNUMフィールドも持つが省略した。

図5.13 図5.12のコントロールフロー・グラフのプライマリ・セット表(部分)

ル変数のOR結合であることから、ブール変数に相当する情報を持たせたセル(INFPREDREC型)のリニア・リストでOR結合を表現し、そのリストの先頭をさすこととした。すなわち、論理的なOR結合を物理的にはリニア・リストで表現している。図5.13に例として図5.12のコントロールフロー・グラフのプライマリ・セット表の一部を図示する。この図において、プライマリ・セット表の最初のエントリでは、頂点A、頂点B、頂点F、頂点Gが同一のプライマリ・セットであり、PREDVERTフィールドが特別にヘッダ・バーテクスを指す(このとき、BOOLVALUEフィールドは意味を持たない)ことにより直属のブール式は恒真であることを表現している。2番目のエントリでは、頂点D、頂点Eが同一のプライマリ・セットであり、直属のブール式は(分岐バーテクスBが真)OR(分岐バーテクスKが真)であることを表現している。なお、この例ではループが存在しないため、BELGLOOPフィールドはすべてNILとなっているが、このフィールドは当該プライマリ・セットに含まれるコントロールフロー・グラフの頂点が属するループに対応する(G10)ループ情報記述バーテクスを指す。

5.4.4 制御関係解析(III):2出現間の制御の流れによる先行性解析

5.2節および5.3節で述べたとおり、自動ベクトル化技術の中核をなすデータ参照関係解析では、同一変数の2出現ごとのデータ参照関係から生じる依存関係を解析する。その際、制御の流れの解析も必要となる。例えば、あるif文のthen節とelse節に2出現が別れているような場合には、ループのある同一の繰り返しにおいては参照の衝突が生じ得ない。また、あるループ内に存在する2出現について、ループのある同一の繰り返しにおいて参照の衝突が生じた場合には、そのループ本体のみに着目して、どちらの出現が先に実行されるか、すなわちどちらの出現に制御が先に来るかによって、データ参照関係からくる依存の向きが決定される。このように、与えられた2出現に対する制御の流れによる先行性、すなわち依存関係があるのか、ないのか、また、ある場合にはどちらが先に実行されるのかという解析も欠かすことができない。詳細な部分ベクトル化を行うためには、中間コードレベルで、2出現間のデータ参照関係解析を行う必要がある。従って、2出現間の先行関係解析においても、中間コード間の先行性を解析できる必要がある。しかし、効率を考えるとコントロールフロー・グラフの頂点間で解析できれば十分である。なぜなら、複数の中間コードに対応するのは、コントロールフロー・グラフの(G9)プロセス・バーテクスだけであるが、同一の(G9)プロセス・バーテクス(あるいは基本ブロック)に属する中

間コード間の先行性を解析するには、単に当該頂点の先頭の中間コードから順にたどり、どちらの中間コードが先に現れるか調べるだけでよいからである。

ここでは、やはり仮想的な定義を用いることにより、データフロー集合から簡単にコントロールフロー・グラフの頂点間で先行関係を解析する手法について述べる。まず例として、図5.14で頂点Eと頂点Iについて考える。これらの頂点は図中に示す

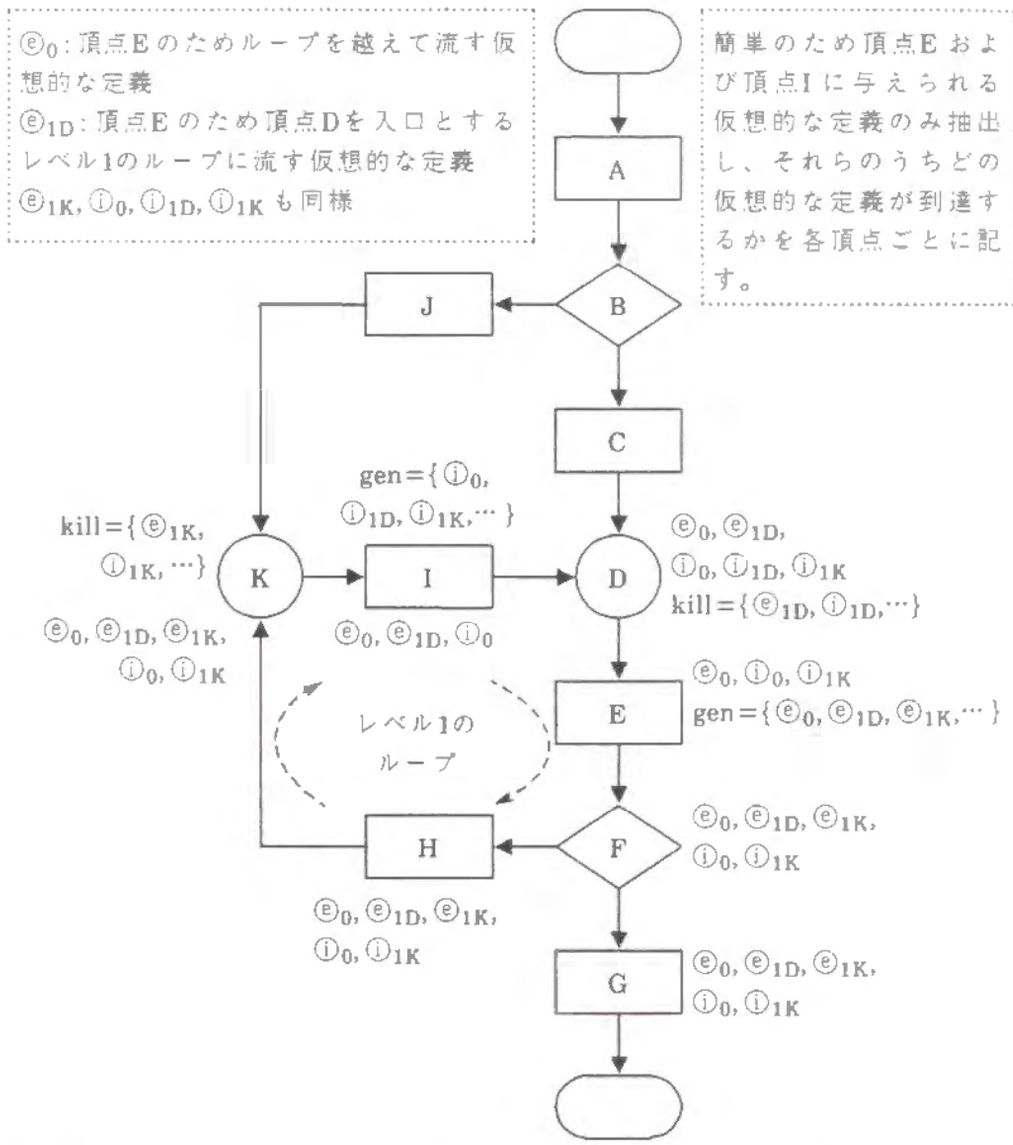


図5.14　複数の入口を持つループのコントロール・フロー・グラフの先行性解析例

とおりネストレベル1のループに含まれている。しかも、そのループは頂点Dおよび頂点Kを入口に持つマルチエントリ・ループとなっている。そのために、ループがどの入口から開始されるかにより、先行関係が逆転する可能性を有する。現に、頂点Dからループが開始された場合、頂点Eが頂点Iに先行し、頂点Kから開始された場合には、逆転する。このような詳細な先行関係の解析を可能とするため、ループ内の各頂点（厳密には所期の目的からデータ参照を起こす(G2)手続/関数呼び出しバーテクス、(G7)ループビギン・バーテクス、(G9)プロセス・バーテクスについてのみでよい）でループの入口点数と同数の仮想的な定義を独立に発生させ、これら独立な定義をそれぞれループの各入口点でkill集合に含めることにより消滅させる。例えば図中頂点Eについては、入口点D、Kそれぞれで消滅させる仮想的な定義ⓔ1D、ⓔ1Kを流している。これらの仮想的な定義を用いて、頂点Eにⓘ1Kが到達することから、頂点Iが頂点Eに先行することを、そして頂点Iにⓔ1Dが到達することから、頂点Eが頂点Iに先行することを判定できる。なぜなら、これらのループ入口点で消滅させる（kill集合に含める）仮想的な定義は、ループの繰り返しによって何度もループ内の各頂点に到達することはなく、ループの繰り返しがないとしたときの、ループ内の各頂点の先行関係にのみ依存して到達するからである。これらの定義がループを越えて流れ出すことはないため、ループ外の頂点との先行関係解析時には、ループ入口点で消滅させない仮想的な定義を別個に流す。例えば図5.14の頂点Eでは定義ⓔ0がそれに相当する。

[2出現間の先行関係解析アルゴリズム]

(1) あるコントロールフロー・グラフの頂点Vについて、仮想的な定義をgen集合に含める。この定義は消滅させず、終点まで流す。

(2) その頂点Vがある多重ループ（特別な場合として1重の場合を含む）内に存在するとき、各ループごとに当該ループの入口の数だけ仮想的な定義を発生させ、同頂点のgen集合に含める。各ループの入口の数だけ発生させた定義は、各々当該ループの異なる入口でkill集合に含める。

(3) 大域的データフロー解析を行う。

(4) 調査したい2出現がともにある同一の(G9)プロセス・バーテクスVに存在する場合には、最悪の場合でもその頂点Vに属する全中間コード列を1度走査するだけで両者の先行関係は決定できる。

(5) ステップ(4)以外の一般の場合、2出現それぞれが属する頂点VおよびW間の先行関係を調べる。この場合、さらに次の(6)あるいは(7)の2とおりに場合分けする。

(6) 頂点VおよびWが共通のループに属さないとき、ステップ(1)で発生させた両頂点の定義が相手の頂点に到達しているかどうか調べる。頂点Vで発生させた定義が頂点Wに到達すれば、頂点Vは頂点Wに先行すると判定できる。逆の場合は、逆の先行関係にあり、どちらの定義も相手に到達しなければ、両者の間に先行関係(制御依存関係)がないと判定できる。

(7) 頂点VおよびWが共通のループLに属するとき、ループLについてループLの入口の個数だけ発生させた両頂点の定義それぞれについて、ステップ(6)と同様の判定を独立に行う。図5.14に例示したように複数の入口を持つループ内にあり、どの入口からループが開始されるかにより、先行関係が逆転する場合でも、正しく両方向の先行関係を調べることができる。もちろん逆転しない場合でも、正しく調査できる。これは、ステップ(2)で発生させたループごとの仮想的な定義により、当該ループによる繰り返しがなかった場合、すなわち仮にループを無くした場合の制御の流れが、そのまま判定できるからである。□

この制御の流れからくる先行関係は、一般には先述の支配関係からだけでは、正しい判定が困難である。例えば、図5.14の頂点Cは頂点Eを支配しないが、これら2頂点間でも先行関係はある。これに対し、ここで提案した方法は、調査のコストが少なく、かつ正しく調べられるという特徴を持つ。なお、ステップ(2)で用意した定義は、一部ループ外にも流れ出すが、他の解析に影響を及ぼすことはないので、わざわざループの全出口の次の頂点においてkill集合に含め消滅させる必要はない。

なお、この制御関係解析(Ⅲ)のために用意された仮想的な定義のデータフロー集合における要素番号すべてを、(G1)手続/関数宣言バーテクスのPROCVIRTDEFフィールド(表4.12参照)に集合として記録している。また、コントロールフロー・グラフの各頂点ごとに用意されたこの制御関係解析(Ⅲ)のための仮想的な定義は、当該頂点のVIRTDEFTOPフィールド(表4.11参照)に以下のフィールドを持つセル(VIRTDEFPROC型)のリストとして記録される。

(ア) BELONGLOOPフィールド:どのループについて用意されたかを、(G10)ループ情報記述バーテクスを指すことにより区別するためのフィールド。前記ステップ(1)で用意するものについてはNILとなる。

(イ) ENTRYVERフィールド:当該ループのどの入口でkillされるかを、入口となるコントロールフロー・グラフの頂点を指すことにより区別するためのフィールド。

(ウ) VDEFNUMフィールド:当該定義のデータフロー集合における要素番号

(エ) NEXTフィールド:次のセルを指す。

5.5 結 語

データ依存関係解析では、文献[13]に挙げたBanerjeeのアルゴリズムが著名である。この方式の中核をなすBanerjeeテストは、簡単に述べると凸多面体となる繰り返し空間の端点において、1次元化した上で右辺の項をすべて左辺に移項したDiophantus方程式の左辺式が正負いずれの値となるかを調べることにより、中間値の定理から正負両方をとる場合について、解がある、すなわち依存があると判定するものである。このとき、繰り返し空間が凸多面体となることを保証するため、ここで実現のために設けたループの初期値、最終値がより外側のループの制御変数のみの線形式であり、増分値は定数であるという制約をやはり課している。ただし、Banerjeeテストはこの制約を利用することにより、すべての端点についてDiophantus方程式の左辺式を計算する必要がなく、より簡単にBanerjeeの不等式が満たされるか否かを調べるだけでそれと等価な調査が行えることを基礎とするものである。つまり、Banerjeeテストは凸多面体となる繰り返し空間をDiophantus方程式が表す超平面が横切れば、解があるとする。すなわち、実数解の存在可能性の調査を行うものであって、決して整数解の有無を調べるものではない(1重ループを除く)。

すでに述べた数式処理法、Sort-Merge法ならびに複合アルゴリズムの3者それぞれとBanerjeeテストとを比較する。まず、Banerjeeテストと数式処理法とはともに必要十分な解析ではない点は同じであるが、Banerjeeテストの方が単純であり解析コストの点で勝る。Sort-Merge法は、対象ループの総繰り返し回数に比例した解析コストがかかるのに対し、Banerjeeテストの解析コストは、対象ループの総繰り返し回数には依存せず、対象ループのネストの深さに比例した解析コストがかかる。従って、通常ベクトル化対象とするループにおいては、Sort-Merge法に比べ

Banerjeeテストの方がやはり解析コストの点で勝る。ところが、複合アルゴリズムでは、十分小さな子問題に分割してからSort-Merge法を適用するため、領域コストが禁止的に大きくなることはない。しかも、Sort-Merge法は厳密に整数解による依存解析が可能であることから、Sort-Merge法ならびに複合アルゴリズムの解析能力は完全にBanerjeeのアルゴリズムより勝るといえる。なお文献[13]では、Banerjeeテストは1重ループについては厳密な解析が可能であることを示しているが、多重ループについては拡張されていない。そのため例えば図5.15の2例について、Banerjeeのアルゴリズムは厳密な解析ができない。同図(a)の2出現に関する依存解析をBanerjeeのアルゴリズム[13]で解析すると、同図(c)の1次元化した方程式の左辺式をfとおいて、fの繰り返し空間における最大値f_{max}および最小値f_{min}がそれぞれ、210と-161と求められる。その結果これらの最大値f_{max}/最小値f_{min}が同図(c)の1次元化した方程式の右辺の値をはさみ、Banerjeeの不等式

$$-161 = f_{min} \leqq -11 \leqq f_{max} = 210$$

が成り立つ。従って、同図(c)の1次元化した方程式の解は存在し得ると誤って判定され、そのため依存も存在するとみなすこととなる。これに対し、複合アルゴリズムを適用すると、同図(c)の1次元化した方程式の整数解は存在しないことが厳密に判定できる。この場合数式処理法をまず適用した段階では、iとi'に関するペア関係式として

$$i + i' \geqq 10 \text{ かつ } i - i' \leqq 2$$

が得られる。そして、Sort-Merge法を併用してはじめて解が存在しないことが判明するのである。

さらに、図5.15(d)の例を考える。これは、同図(a)の2出現の2次元目の添字式を交換したものである。この場合同図(e)のDiophantus方程式から、

$$i = 4, j' = 5$$

と解が一意に求められる。ところが、この解は繰り返し空間内($5 \leqq i \leqq 10$であることが必要)にない。それにもかかわらず、同図(f)の1次元化した方程式をBanerjeeのアルゴリズムで解析すると先と同様に、この場合にもBanerjeeの不等式

$$-191 = f_{min} \leqq 29 \leqq f_{max} = 191$$

が満たされるので解が存在し得ると誤って判定される。それに対し、複合アルゴリズムを適用すると、同図(f)の1次元化した方程式に数式処理法を適用しただけでは、iとi'に関するペア関係式は


```
for i := 1 to 10 do begin
    for j := 10-i to i do begin
        a[i+1,j+2] := … ; { 出現1 }
        … := a[j,11-i]; { 出現2 }
    end;
end;
```

(a) 配列の2出現の例1(a[0..19,0..19]と宣言されているものとする)

$$i+1 = j' \text{ (1次元目)},$$
$$j = 9-i' \text{ (2次元目)}$$
$$1 \leqq i, i' \leqq 10$$
$$10-i \leqq j \leqq i$$
$$10-i' \leqq j' \leqq i'$$

(b) (a)の例のDiophantus不定方程式と繰り返し空間の定義域

$$20 \cdot i + i' + j - 20 \cdot j' = -11$$

(c) (a)の例の一次元化したDiophantus不定方程式

```
for i := 1 to 10 do begin
    for j := 10-i to i do begin
        a[i+1,11-i] := … ; { 出現1 }
        … := a[j,j+2]; { 出現2 }
    end;
end;
```

(d) 配列の2出現の例2(a[0..19,0..19]と宣言されているものとする)

$$i+1 = j' \text{ (1次元目)},$$
$$9-i = j' \text{ (2次元目)}$$
$$1 \leqq i, i' \leqq 10$$
$$10-i \leqq j \leqq i$$
$$10-i' \leqq j' \leqq i'$$

(e) (d)の例のDiophantus不定方程式と繰り返し空間の定義域

$$-19 \cdot i + 21 \cdot j' = 29$$

(f) (d)の例の一次元化したDiophantus不定方程式

図5.15 Banerjeeのアルゴリズムでは厳密な依存解析が行えない例

$$19\cdot i + 21\cdot i' \geqq 181 \text{ かつ } 19\cdot i - 21\cdot i' \leqq -29$$

となり解が存在し得るが、次にSort-Merge法を適用することにより、整数解無しと正しく判定できる。

このように5.3.5項の複合アルゴリズムは必ず厳密な解析が可能である。従って、多重ループ全体にわたるデータ依存関係解析では、ベクトル化による実行時の高速化を考慮すると、コンパイル時の解析コストが大きくなっても厳密な解析が可能である5.3.5項の本方式は優れていると考える。また、Sort-Merge法では実際に整数解を求めるので、ループ独立依存とループ運搬依存、あるいはループ2回運搬依存とループ多回運搬依存の類別は簡単に行える。これに対しBanerjeeのアルゴリズムでは、ループ独立依存とループ運搬依存の類別を行うためには、多重ループの各制御変数ごとに、例えばi＝i'の場合、i＜i'の場合、i＞i'の場合に場合分けし繰り返し空間を分割して、それぞれの場合について別個に調べなければならない。ループ2回運搬依存とループ多回運搬依存の類別は、i＝i'+cの場合、i+c＝i'の場合（ここにcは当該ループの増分値）について、やはり問題を設定しなおし調査しなおすこととなる。しかも、これらの調査もやはり厳密ではないことに注意が必要である。それに対し、5.3.5項の複合アルゴリズムは厳密な依存判定と合わせて、最終的にSort-Merge法を適用するため、5.3.6項で述べたとおり実際に存在する依存の類別を1度にまとめて調査できる。

従って、V-Pascalのために新たに開発された5.3.5項の複合アルゴリズムは多重ループにわたる厳密かつ詳細なデータ依存関係解析を行う、新しい有効な方式であると結論できる。

次に制御依存関係解析については、文献[15]では、制御依存をフラグ変数のデータ参照関係からくる依存ととらえ直すことにより、5.4.1項および5.4.2項と類似の解析を行っている。ただし、変換アルゴリズムが、ここで述べた方法に比して複雑である。また、文献[15]では扱う対象がプリプロセッサであるためソーステキストレベルで変換を考えており、その方法を中間コードレベルでの詳細な解析に適用したとすると、解析のコストがさらに大きくなると予測される。これに対し、ここで提案した解析手法は、中間コードあるいはコントロールフロー･グラフを解析対象としたコンパイラに適した方式である。さらに、通常の最適化の際に使用される大域的データフロー解析をそのまま適用するだけで、大域的データフロー解析と同時に種々の制御関係解析を行うことができる。中核となる大域的データフロー解析部は

4.7節で述べたとおり、文献[45]の集合演算で解析するアルゴリズムをそのままPascalで記述しインプリメントしている。Pascalの集合演算は集合を構成する全要素のビットベクトルに対する論理演算で実現されている。従って、この手法を用いるために必要となる仮想的な定義の数（[条件分岐の数]×4+Σ{[コントロールフロー･グラフの5.4.3項の解析対象となる頂点のループの深さ]×[当該ループの入口点数]+1}；ここでΣは5.4.3項の解析対象となる頂点すべてに対する総和である）は通常発生する定義の数に比較してほとんど無視できることを考え合わせると、今回提案した解析手法の実行時のコストは通常の大域的データフロー解析にほぼ完全に埋没すると考える。

従って、V-Pascalに実現されている制御関係解析のためのこの新しいアルゴリズムは、一般の自動ベクトル化コンパイラについても有効な手法であるといえる。

第6章
多重ループのベクトル化

6.1 緒 言

第3章で既に述べたとおり、現状の自動ベクトル化は多重ループについては機能が十分ではない。これに対しV-Pascalコンパイラは、設計当初から多重ループ全体にわたる強力な自動ベクトル化機能を実現すべく開発されてきた。そのため第5章で述べた多重ループ全体にわたる各種依存解析手法を新たに独自に開発/実現した。これらの依存解析により、保存すべき依存は、すべて依存グラフを表現したD行列にまとめられる。ここでは、そのD行列を使いどのように多重ループ全体をベクトル化していくかについて述べる。もちろん、ここで述べる多重ループ全体にわたる自動ベクトル化技術も津田研究室において新たに独自に開発/実現したものである。

V-Pascalコンパイラの自動ベクトル化の概略は以下のとおりである。

(1) ベクトル化対象多重ループの抽出
(2) if-then-else構造への正規化
(3) 行列構成セルによるD行列の枠組み作成
(4) 各種依存解析ならびに検出された依存のD行列への登録
(5) D行列の行および列交換、ならびに縮退操作
(6) 最適ベクトル化のループ選択
(7) 部分ベクトル化：中間コードの(b-2)ベクトル処理記述ノードの追加(含む置換)

このうち、(1)は与えられた中間コード表現のプログラムを走査し、次のような多重forループを取り出す。

(ア) for文によるループのみからなること。すなわち、中間コードの(a-8)forループ記述ノード(4.3.2項参照)による多重ループであること。
(イ) 各forループの出口は1点であること。すなわち、forループの終了条件のみにより、ループが終了すること。ループ外への飛び出しはあってはならない。

このうち、(ア)の条件はまず、現在処理中の手続/関数の中間コードの(a-1)手続/関数宣言ノードのLPLISTフィールド(4.3.2項参照)から、(a-8-1)ループビギン-ノードによるforループの入れ子の木構造(図4.13参照)を表すリンクをたどることにより簡単に最外側にあるforループを探索することができる。各最外側にあるforループごとに、対応するコントロールフロー-グラフのILPPフィールド(4.3.3項表4.11参照)から(G10)ループ情報記述バーテクス(4.3.3項表4.10参照)をたどり、このループ情報記述バーテクスが持つループ(forループ以外のループ構造も含む)の入れ子の木構造を表すリンク(表4.12参照)をたどり、同じく同バーテクスが持つFORFLGフィールド(同上)を調べることにより、forループ以外のループかどうかを容易に調べられる。また、同時に(イ)の条件についても、同じく同バーテクスが持つLPEXITSフィールド(同上)を調べることにより簡単に判定できる。

このように、V-Pascal Version 1では飛び出しのない多重forループをベクトル化対象としているが、Version 2ではより一般のループ構造をも対象とする研究が現在進行中である。

上記の(3)は、(1)で抽出された多重forループに含まれる中間コードに1対1に対応する、4.3.5項で述べたD行列の行列構成セルを作成する。そして、対応する中間コードとのリンク、ならびに元の中間コードの順序にあわせ隣接する行列構成セルを示すリンクを適切につなぐ。このとき、(2)で対象多重ループ内のif文はすべてif-then-else構造に正規化されているので、中間コードのある(a-3)分岐型ノードを処理したら、まずthen節側(NXTLINK2フィールド、4.3.2項参照)をたどって処理し、次にelse節側(NEXTLINKフィールド、同上)をたどって処理する。これを再帰的に繰り返すこととした。

以下上記の(2)および(4)から(7)について節をあらため順に説明する。

6.2 if-then-else構造への正規化

3.3.6項で述べたとおりベクトル計算機では、if文はマスクベクトルを用いてベクトル実行される。そして、そのif文を含むループのベクトル化も、if文を含まないループのベクトル化と同様に、各演算ごとにループ分割されることとなる。すなわち、if文を含まないループについて3.2.1項の図3.4と同様の処理である。このif文を含むループのベクトル化を図3.4と同じくソースプログラムで表現すると図6.1の

ようになる。なお、この図ではソースプログラムで表現するため、文単位でループ分割されているが、実際には中間コード単位でループ分割される。この図からわかるとおり、ベクトル化前に then 節あるいは else 節に含まれていた文は、ベクトル化にともなうループ分割後も元のプログラムの意味を変えないよう if の条件分岐の制御を受けるようにする必要がある。すなわち、第3章で述べたループ分割は、単純に元のプログラムのループ構造を複製するものであったのに対し、一般にネストした if 文を含むループのベクトル化では、そのネストした if 文の条件分岐の制御構造をも複製するのである。もちろん、図6.1にも記したとおり、各文がベクトル命令に変換されるならば、if 文の条件分岐の制御(図中の p[i] あるいは not p[i])はマスク機能による制御で実現されるため、if 文が明示的に複製されるわけではない(図中で

```
for i:= 1 to n do begin
    {文1};
    if p then begin
        {文2};
        {文3};
    end
    else begin
        {文4};
        {文5};
    end;
    {文6};
end;
```

(a) ベクトル化前

制御依存/データ依存ともにこのままの順序でベクトル化可能であったとする。

ベクトル化

```
for i:= 1 to n do begin
    {文1};
end;
for i:= 1 to n do begin
    if p[i] then begin
        {文2};
    end;
end;
for i:= 1 to n do begin
    if p[i] then begin
        {文3};
    end;
end;
for i:= 1 to n do begin
    if not p[i] then begin
        {文4};
    end;
end;
for i:= 1 to n do begin
    if not p[i] then begin
        {文5};
    end;
end;
for i:= 1 to n do begin
    {文6};
end;
```

(b) ベクトル化後

ベクトル化後は各文ごとにループ分割され独立したループとなる。なお、if 文の条件節である単純変数 p は、配列化され p[i] となっている。この p[i] はマスクベクトルそのものと考えてよい。

図6.1 if 文を含むループのベクトル化のソースプログラムによる表現

はあくまで解説のため明示的に表記した)。言い換えれば、if 文が明示的に複製されるのは、部分ベクトル化によりループ分割された、スカラ実行せざるを得ない文群についてのみである。

この if 文の条件分岐の制御構造の複製処理は図6.1のような単純な if-then-else 構造では比較的容易である。なぜなら、スカラ実行せざるを得ないと判定された各中間コードは、それが4.3.4項で述べたプライマリ・セットのどれに属するかを調べ、所属するプライマリ・セットが直接影響を受ける条件分岐およびその真側のパスにあるのかあるいは偽側のパスにあるのかが判明するので、その情報から必要に応じて(a-3)分岐型ノードおよび(a-4)合流型ノードを付加すればよい(詳細は6.5節参照)。ネストした if-then-else 構造についても、数は増えるが同様に(a-3)分岐型ノードおよび(a-4)合流型ノードを付加すればよい。ところが、図6.2に示すような飛び出しのあるループではないが、ネストした if-then-else 構造を含むループでもない場合がある。同図で G%2 の識別番号を持つ(G7)ループビギン・バーテクスと G%8 の識別番号を持つ(G8)ループエンド・バーテクスとではさまれた部分がその例である。この例では頂点 G%9 が(G3)分岐型バーテクス G%3 および G%5 のどちらかが真のとき、実行されるため、完全な if-then-else 構造ではなくなっている。以下このような構造をそのコントロールフローの形から網目構造と呼ぶことにする。網目構造を含むループをベクトル化しようとしたとき、いま先の頂点 G%9 に属するある中間コードがスカラ実行しなければならないと仮定する。このときにも、通常の if-then-else 構造の場合と同様に if 文の条件分岐の制御構造の複製処理が必要になるが、スカラ実行しなければならない中間コードで他の頂点に属するものがあると、この処理は複雑になる。例えば頂点 G%10 に属するある中間コードもなんらかの依存関係により同時にスカラ実行しなければならない場合には、頂点 G%5 の真側のパスは当然複製されているが、同時にスカラ実行しなければならない頂点 G%11 に属する中間コードがまったくなかったとしても、頂点 G%3 の真側のパスも複製しなければならない。頂点 G%10 と頂点 G%11 との関係が逆の場合にも同様のことがいえる。

そこで、現在の V-Pascal ではこの if 文の条件分岐の制御構造の複製処理を少しでも単純化できるように、網目構造をネストした if-then-else 構造にあらかじめ変換しておくことにした。すなわち、図6.2の頂点 G%9 に含まれる中間コード列の複製を作り、分岐型バーテクス G%3 および G%5 のそれぞれの真側のパスに独立させて置くのである(図6.3参照)。このように、中間コード列を複製することは結果的には

生成される目的コードが大きくなる点を除き問題はない。なぜなら、複製された中間コード列と元の中間コード列とは、同時に実行されることはなく、他の部分の中間コード列も含め、中間コードの実行順序あるいは実行(繰り返し)回数が変換前と変わることはない。これは、複製された中間コード列が副作用のある手続/関数呼出しを含んでいたとしても同じことがいえる。つまり、この変換はプログラムの意味

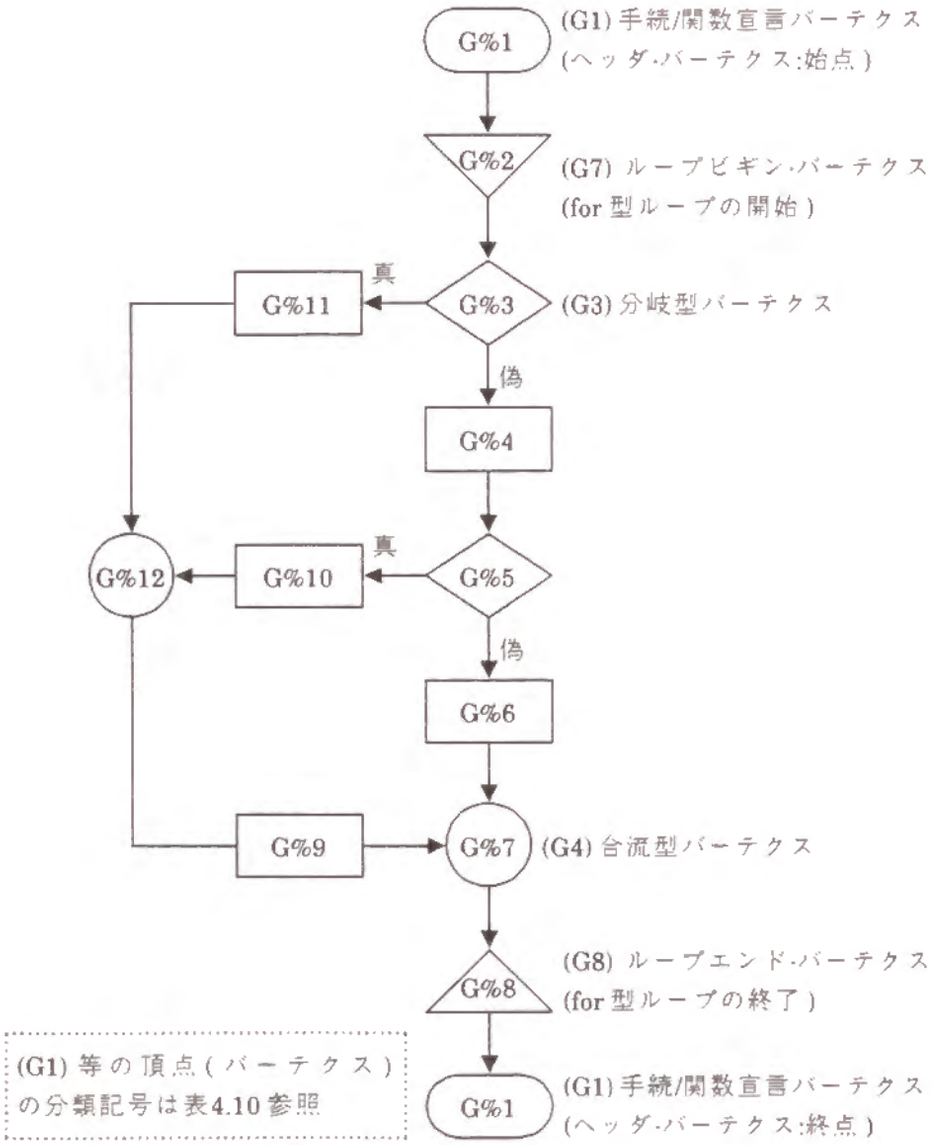


図6.2 網目構造を含むループのコントロールフロー-グラフの例

を変えるものではない。実行時間については、もし複製された中間コード列と元の中間コード列とがすべてスカラ実行されるなら、変換前後でほとんど差はない。ところが、もともと同一の中間コードが複製され、それらが別々にベクトル化された場合には、本来なら変換前のガード式(先の図6.2の場合[G%3が真]or[G%5が真])で1度実行されるだけであるものが、別々に(先の図6.2の場合[G%3が真]のガード

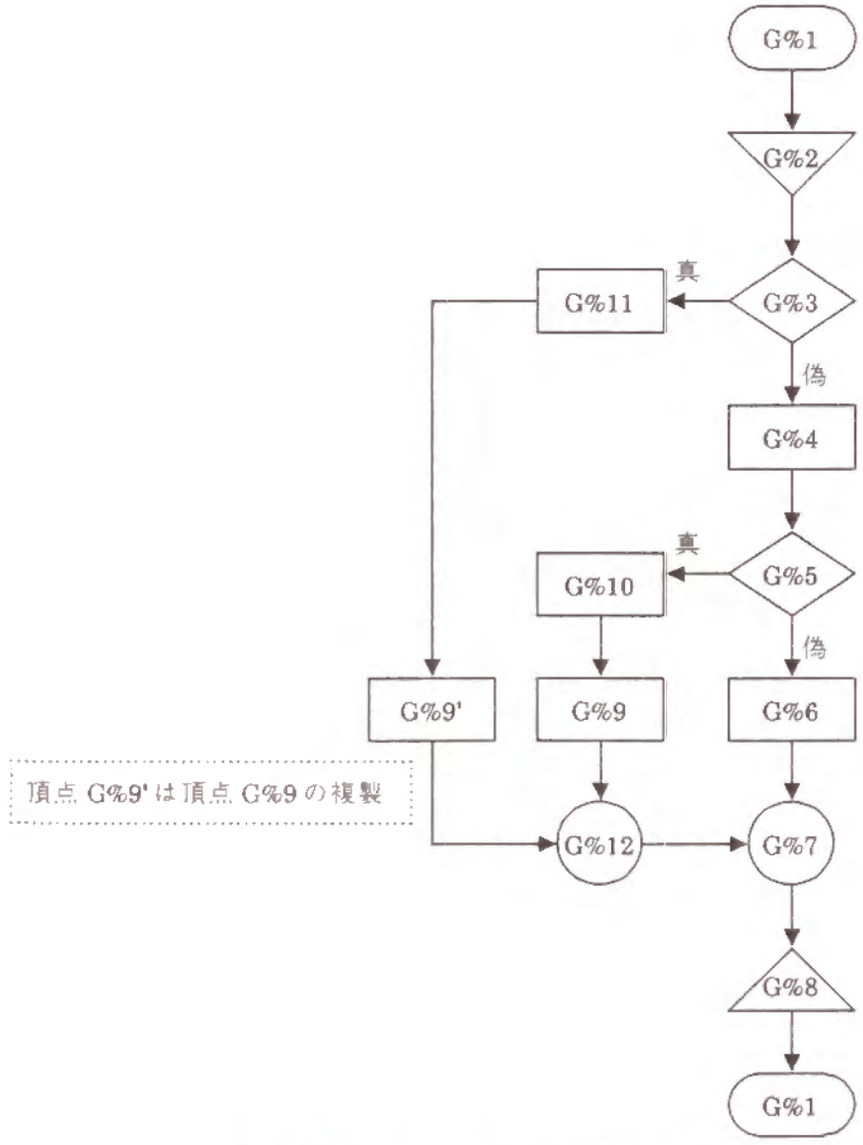


図6.3 網目構造の if-then-else 構造へ変換の例

式で1度、[G%5が真]のガード式で1度)計2度実行されるため、変換後の方が遅くなる。しかし、このような場合には、ベクトル化後中間コードをマージして、もとのガード式をもつ1中間コードに再構成すればよい。

6.3 各種依存関係解析とD行列への登録

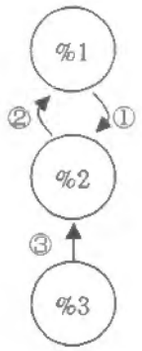
制御依存関係の登録は以下のように行う。既に作成されているD行列の行列構成セル(4.3.5項表4.13参照)を先頭から最後まで順にたどりながら、そのINTERPTフィールドにより各行列構成セルに1対1に対応する(ベクトル化対象ループ内の)中間コードを順に調べる。もし、その中間コードが(a-3)分岐型ノードあるいは(a-8-1)ループビギン・ノードであれば、そのGRAPHPフィールド(4.3.2項表4.5参照)により、それぞれのコントロールフロー・グラフの頂点(G3)分岐型バーテクスあるいは(G7)ループビギン・バーテクスを得る。これらの頂点には、5.4.3項で述べたとおりこれらの頂点に制御依存するプライマリ・セットがPPSLSフィールド、LPPSLSフィールド(4.3.3項表4.12参照)にそれぞれ記録されている。この情報から、制御依存するプライマリ・セットに含まれるコントロールフロー・グラフの各頂点に属する全中間コードが、当該(a-3)分岐型ノードあるいは(a-8-1)ループビギン・ノードに制御依存するものとしてD行列に登録する。D行列の非零要素セルのCRFLGフィールドを真とすることで制御依存であることを示す(4.3.5項および表4.14参照)。

データ依存関係解析は、やはりベクトル化対象ループ内のコントロールフロー・グラフの各頂点ごとに処理をすすめる。そして、中間項ならびに変数を参照し得る(G2)手続/関数呼び出しバーテクス、(G3)分岐型バーテクス、(G7)ループビギン・バーテクス、(G9)プロセス・バーテクスについてのみ対応する中間コートをたどり、データフロー集合を用いて参照関係を生じる2出現を抽出する。抽出された中間項ならびに変数に関する2出現に対し適用されるデータ依存関係解析アルゴリズムは5.2節および5.3節で述べたとおりである。5.3節で述べた配列添字に基づく配列要素間のデータ依存関係解析アルゴリズムの現在の制約を越える添字式、ループ上下限式の場合には、5.1節で述べたとおり2出現に共通の全ループにおいてデータ依存があるものとみなす。なお、(G2)手続/関数呼び出しバーテクスについては、そのINTERPフィールド(4.3.3項表4.12参照)により、対応する中間コードを得たあと、それが呼び出す(a-1)手続/関数宣言ノードのDEFLISTフィールドおよびREFLISTフィールド(4.3.2項参照)をみることにより、簡単に当該手続/関数呼び出しにより参照される大域変数がたどれる。その結果からデータ参照関係のありうる2出現を抽出し、先と同様にデータ依存関係解析アルゴリズムを適用する。

また、(G2)手続/関数呼び出しバーテクスの特殊な場合である標準手続/関数の中のファイルI/Oに関係するものについては、同一のファイルに関するこれらの標準手続/関数のスカラ実行時の順序関係を保存する必要がある。それゆえに、あるファイルに関係するこれらの入出力関連の標準手続/関数は、先行するすべての同一ファイルに関する入出力関連の標準手続/関数に依存があるとして、D行列に登録している。依存の種別は、種々のベクトル化手法を用いても無視できないように、どのループにも属さない0番目のループの定義-定義型とした。

6.4 依存関係によるベクトル化可否判定

V-PascalではD行列にまとめられた依存関係を使い、ベクトル化可否判定を行う。D行列は4.3.5項で述べたとおり、依存グラフを隣接行列の形で表現したものである。例えば、図6.4(a)に示した依存グラフの例でスカラ実行時の順序そのまま%1、%2、%3とベクトル実行させようとすると、%2→%1(同図中②の有向辺)、%3→%2(同図中③の有向辺)の依存の向きが逆であるため、意味が変わってしまう。これらの依存がデータ依存であれば、3.2.2項で述べたデータ参照関係不適の状態が生じていることにほかならない。このことを図6.4(a)の依存グラフを表現したD行列である同図(b)で考えてみる。この図からわかるとおり、依存の向きが逆であることは、D行列上では依存の存在を示す非零要素が下三角部にあることと等しい。そこで、これらの依存の向きが逆転しないように、ベクトル実行させるときの中間コード、すなわち今の依存グラフの頂点の順序を並べ換えてみる(同図(c)参照)。当然%2→%1(同図中②の有向辺)、%3→%2(同図中③の有向辺)の依存の向きは、頂点の順序と一致している。しかし、今度は%1→%2(同図中①の有向辺)の依存の向きが逆転してしまっている。先と同様に同図(d)のD行列で考えてみる。依存グラフの頂点の順序を交換したことは、D行列上では行および列を並べ換えたことに相当する。この行/列交換により、②および③の非零要素は確かに上三角部に移動した。その一方で、①の非零要素が逆に下三角部に移動してきている。すなわち、この簡単な例では%2→%1(同図中②の有向辺)および%1→%2(同図中①の有



(a) 中間コードのノードを単位とした依存グラフの例

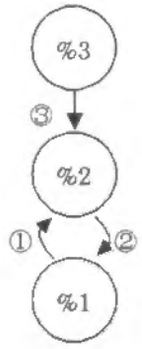
(注) D行列の非零要素は実際には4.3.5項で述べた種々の情報を持つが、省略し(a)との対応のみを記した。また0の要素は実際には存在しない。

列構成子の並び
(有向辺の終点側)

行構成子の並び
(有向辺の始点側)

	%1	%2	%3
%1	0	①	0
%2	②	0	0
%3	0	③	0

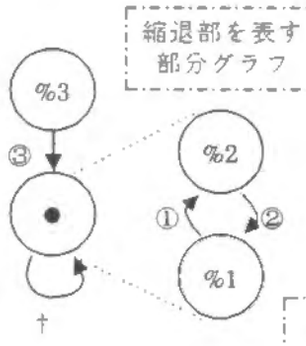
(b) (a) の依存グラフの D行列表現



(c) 順序交換後の依存グラフ

	%3	%2	%1
%3	0	③	0
%2	0	0	②
%1	0	①	0

(d) (c) の依存グラフの D行列表現



(f) 縮退化処理後の依存グラフ

	%3	
%3	0	③
0	†	

縮退部を表す
部分小行列

縮退部を指す
行列構成セル
(4.3.5項参照)

	%2	%1
%2	0	②
%1	①	0

(注) 縮退部の「†」は自己依存(5.3.7項参照)とは別

(e) 縮退化処理後の D行列表現

図6.4 ベクトル化可能性判定のための依存関係による半順序づけ

向辺)の依存が閉路を形成し、そのために依存グラフの頂点%1および%2はどのように並べ換えても、依存グラフの有向辺の向きに並べることができない例となっている。つまり、頂点%1および%2はこの依存グラフの強連結成分(一般の有向グラフにおいてはn-ブロックと呼ばれる)となっている。そこで、このような依存が閉路を成す強連結成分は、図6.4(e)のD行列に示したとおり強連結成分に対応する行および列を縮退させて部分小行列として切り分けることにより、その他の行/列(依存グラフでの頂点)の交換を引続き行う。このD行列縮退処理は依存グラフでは図6.4(f)に示すとおり強連結成分を1頂点とみなすことに等しい。このように縮退させ切り分けられた強連結成分そのもの、あるいはそれらの頂点からなる部分グラフあるいは部分小行列のことを**縮退部**と呼ぶことにする。

こうして、D行列の下三角部にある非零要素を上三角部に移動させるようにD行列の行/列交換を行いつつ、発見された強連結成分を縮退させることにより、あらゆる依存関係を保持したままベクトル化可能な中間コードの実行順序が極めて自然に求められる。縮退部として切り離された部分小行列の下三角部には、まだ依存が残されたままである(同図(e)参照)ので、すくなくともこのままではベクトル化不可能であることを判定できる。すなわち、縮退処理を含む以上のD行列の行/列交換を中間コード単位、すなわち演算単位の細かな依存関係を登録したD行列に施すことにより、第3章で述べたベクトル化を促進する演算の実行順序の変更ならびに演算単位の部分ベクトル化(スカラ実行部とベクトル実行部の切りわけ)が自然に同時に遂行できる。このD行列の行/列交換は、依存グラフの各頂点を有向辺の向きに半順序関係で並べる(有向グラフの**半順序付け**、**partial ordering**)ことに相当する。

以下、この縮退処理を含む以上のD行列の行/列交換(混乱の生じない限り以下では単にD行列の行/列交換とのみ記す)のための、V-Pascalに実現されているアルゴリズムを示す。

{D行列の行/列交換アルゴリズム}

(1) スタックを用意し空に初期化する。

(2) kをD行列の最初の行列構成セル(4.3.5項参照)とし、スタックにkを積む。

(3) この時点でD行列の大きさがn×nであるとする。D行列において第k列の下三角部に非零要素があるか調べる。もしあればステップ(4)へ。なければステップ(10)へ。

(4) D(j,k)要素が非零要素であったとする(k+1≦j≦n)。jが既にスタックに積まれているか調べる。積まれていればステップ(5)へ。なければステップ(8)へ。

(5) スタック上でjからkまでの行および列を縮退させる。すなわち、jからkまでの行列構成セルは縮退部として部分小行列として切り離す。もとのD行列のD(x,y)要素(x,y∈{スタック上でjからkまでの行列構成セル})を部分小行列に移動する。もとのD行列の除去されたjからkまでの行列構成セルのかわりに縮退部を指す行列構成セルrを新たに用意してつなぐ。もとのD行列の除去されたjからkまでの各行および各列の非零要素を1とみなし零要素との論理和をとる形でコピーする。つまり、非零要素のみを集める形で1行/1列にマージしつつコピーし、行列構成セルrにつなぐ。

(6) 縮退させたjからkまでの行列構成セルをスタックから取り除く。

(7) kをステップ(5)で新たに用意した縮退部を指す行列構成セルrとし、それをスタックに積んだ上でステップ(3)へ戻る。

(8) jをスタックに積む。

(9) jおよびkの行/列を交換し、kをjとし、ステップ(3)へ戻る。

(10) この時点でkがD行列において最後の行列構成セルであれば終了。そうでなければ、kをその次の行列構成セルとし、スタックを空にした上でステップ(3)へ戻る。 □

なお、上記のアルゴリズムでkならびにスタックの各要素は実際には行列構成セルへのポインタとして実現されている。

D行列がデータ参照関係をも表現していること、ならびに上記のアルゴリズムでD行列の行/列を依存グラフの有向辺の向きに半順序関係で1列に並べる際通常順序関係のない行/列間では順序づけに自由度があることに着目すると、さらにD行列の行/列を次のように並べ換えることが考えられる。この再並べ換えは、中間項が生きている(live)[36],[37]期間を短縮させるために行う。中間項が生きている期間を短縮できれば、レジスタが専有される期間を短縮できる上に、6.5節で述べる配列化される中間項を減少させる可能性がある。また、上記の行/列交換アルゴリズムを適用すると、決定された新たな実行順序の最初にベクトル・ロードが、最後にベクトル・ストアが集中する傾向を示す。それに対し、この再並べ換えは、集中を可能な限り解消する作用も有する。その結果、ロード/ストア・パイプラインと各種演算パイプラインとの並列実行、チェイニング等によるハードウェアの高速性を効果的に発揮させる可能性が増大する。

[中間項の生きている期間の短縮のための再並べ換えアルゴリズム]

(1) D行列の行列構成セルを先頭から順に走査し、対応する中間コードがなんらかの中間項(t_xとする)を定義しているもの(行列構成セルxとする)すべてについて以下のステップ(2)から(6)を行う。

(2) D行列の最後の行列構成セルの対応する中間コードの属するコントロールフロー・グラフのデータフロー集合から、中間項t_xが当該ベクトル化対象ループ終了後も生きているかどうか、すなわちループ終了後も参照されるかどうかを調べる。もし、そうならステップ(1)へ戻り次の中間項の処理を行う。

(3) 行列構成セルxに関してD行列の第x行を後ろから順に走査し、データ依存のあるすなわち最後に中間項t_xを引用する行列構成セルを見つける。それを行列構成セルyとする。

(4) さらに第x行を後ろから順に行列構成セルxまで走査しながら、データ依存のない行列構成セル(zとする)すべてについてステップ(5)および(6)を行う。

(5) D行列の第z行のD(z,z+1)要素からD(z,y)要素までに非零要素があるか調べる。あればステップ(3)へ戻り次の候補の探索を行う。

(6) D行列の第z行および第z列を第y行および第y列の直後に移動する。

(7) D行列の行列構成セルを先頭から順に走査し、対応する中間コードがなんらかの中間項(t_xとする)を定義しているもの(行列構成セルxとする)すべてについて以下のステップ(8)から(10)を行う。

(8) 第x行を前から順に行列構成セルy(ステップ(3)と同じ最後に中間項t_xを引用する行列構成セル)まで走査しながら、データ依存のない行列構成セル(zとする)すべてについて以下のステップ(9)および(10)を行う。

(9) D行列の第z列のD(x,z)要素からD(z−1,z)要素までに非零要素があるか調べる。あればステップ(8)へ戻り次の候補の探索を行う。

(10) D行列の第z行および第z列を第x行および第x列の直前に移動する。 □

上記のステップ(9)および(10)によりデータ参照関係のない中間コード(それに対応する行列構成セル)を中間項の生きている期間外へ移動させる。これにより中間項

が生きている期間を短縮させるのである。そのとき、ステップ(5)および(9)で、上三角部に存在する依存が下三角部に移動しないことを確認する。もしそのようなことがある場合には、その行列構成セルを移動することを断念している。

また、ステップ(7)から(10)は次の(7')から(12')のように中間項を引用する側から処理することもできる。

(7') D行列の行列構成セルを末尾から順に走査し、対応する中間コードがなんらかの中間項(t_xとする)を引用しているもの(行列構成セルxとする)すべてについて以下のステップ(8')から(12')を行う。

(8') 中間項t_xが当該ベクトル化対象ループ外で定義されているかどうか、すなわちループ不変式かどうかを調べる。もし、そうならステップ(7')へ戻り次の中間項の処理を行う。

(9') 中間項t_xを定義する行列構成セルをyとする。

(10') 第x列を前から順に行列構成セルxまで走査しながら、データ依存のない行列構成セル(zとする)すべてについて以下のステップ(11')および(12')を行う。

(11') D行列の第z列のD(y,z)要素からD(z−1,z)要素までに非零要素があるか調べる。あればステップ(10')へ戻り次の候補の探索を行う。

(12') D行列の第z行および第z列を第y行および第y列の直前に移動する。 □

6.5 最適ベクトル化手法選択

6.4節で各種依存関係に基づく中間コード単位のベクトル化可否判定が行われた。次にさらに高度なベクトル化技法をも適用し、さらなるベクトル化を試みた後、依存関係以外のベクトル化を阻害する要因を加味し、最終的に中間コード単位のベクトル化可否判定を行うとともに、最適なベクトル化ループを決定する。具体的には以下の概略アルゴリズムとなる。

(1) 配列化およびループ選択によるD行列の縮退部内の中間コード群のさらなるベクトル化
(2) 本質的にベクトル化不可能な中間コードの判定
(3) 中間項の受渡しを考慮した最適なベクトル化ループの決定ならびに配列化すべき中間項の判定
(4) ベクトル化可能な多重ループに関する最適な一重化ループ群の選出

以下順に説明する。

6.5.1 配列化によるベクトル化

5.2節で述べたとおり単純変数に関するデータ依存の中には、配列化することにより無視してよいものがある(5.2節でダミー型と呼んでいるものである)。従って、D行列の各縮退部について、もしダミー型のみが登録された非零要素があれば、6.5.5項の中間項の配列化同様、配列化に必要な領域が確保できるか、そして5.2節で述べた配列化の実行時のオーバーヘッドを考慮してもなおベクトル実行すべきかどうか等の条件を調査する。もし配列化可能であれば対応するダミー型のデータ依存を無視して再度縮退部のみを先の6.4節で述べた並べ換えアルゴリズムで行/列の交換を行ってみる。このとき、ダミー型のみが登録された非零要素を持つ行列構成セルには配列化が必要となることを記録しておく。配列化に必要な具体的な処理は6.5.5項で述べる中間項の配列化と同じである。

6.5.2 ループ選択によるベクトル化

5.3.6項で述べたとおり、配列変数に関するデータ依存の中のループ運搬依存は、ループ運搬しているループすべてをスカラ実行することにより無視できる。この点に着目しD行列の各縮退部について、次のアルゴリズムでループ選択によるベクトル化を試みる。ただし、この現在実現されているループ選択は、5.3.6項で触れたループ交換を含むものではない。

[ループ選択によるベクトル化アルゴリズム]

(1) nを当該ベクトル化対象多重ループの最内側ループのネストの深さとする。

(2) n＝0ならば終了。

(3) kを現在処理中の縮退部の最初の行列構成セルとする。

(4) jをkの次の行列構成セルとする。

(5) kがjに制御依存しているかあるいは深さn以上のネストのループでデータ依存しているか調べる。すなわち、当該縮退部のD(j,k)要素が非零要素であり、かつそのCRFLGフィールド(4.3.5項表4.14参照)が真かまたはLOOPSフィー

ルド(同上)の第n番目以降の集合要素が空でないか調べる。もしそうならば次のステップ(6)へ、さもなければステップ(10)へ。

(6) 行列構成セルk、j間の全行列構成セルを深さn未満のネストのループによるループ運搬依存を除くその他のすべての依存の方向が変わらないようにして、すなわち上三角部/下三角部間での移動がおきないようにしてk、j間から追い出す。移動がおきるものは、そのまま残す。

(7) 行列構成セルk、j間に行列構成セルが残っているか、あるいは行列構成セルjがkに制御依存しているかあるいは深さn以上のネストのループでデータ依存しているか調べる。もしそうなら、次のステップ(8)へ、さもなければステップ(9)へ。

(8) 行列構成セルk、jならびにk、j間のループ選択レベルがまだ記録されていない行列構成セルについてループ選択レベルnをそのSLOOPLVLフィールド(4.3.5項表4.13参照)に記録する。ステップ(10)へ。

(9) 行列構成セルkとj(第k行および第k列と第j行および第j列と)を交換し次へ。

(10) jを現在のjの次の行列構成セルとする。もし次の行列構成セルがあればステップ(5)へ戻る。なければ次へ。

(11) kを現在のkの次の行列構成セルとする。もし次の行列構成セルがあればステップ(4)へ戻る。なければ次へ。

(12) n = n − 1とし、ステップ(2)へ戻る。 □

上記のステップ(8)で使用するSLOOPLVLフィールドの意味は、表4.13に記したとおりスカラ実行すべき最深のループのネストの深さである。すなわち、ステップ(7)において深さn未満のループ運搬依存を無視してもなお依存が閉路をなす部分を検出し、その部分は深さn(あるいはそれよりさらに内側)のネストのループまでスカラ実行せざるを得ないと判定するものである。

6.5.3 ベクトル化不可能な中間コードの判定

今までに述べてきた依存関係からではなく本質的にベクトル化不可能な処理が以下のものである。

(1) 手続/関数呼び出し

(2) 整数/実数/論理型以外のベクトル・データに対するロード/ストアおよび各種演算

(3) 整数型のベクトル・データに対する剰余演算、整数/実数型のべき乗演算

このうち、(1)については sin、cos のような副作用のない算術関数を除きベクトル計算機にとって本質的に適合しない。ただし、実引数の計算等は別の中間コードで表されるので(1)には当然含まれない。FORTRAN では sin、cos のような算術関数にベクトル・データが実引数として引き渡される場合に、特別に用意された実行時ライブラリにより、これらの関数本体の演算もベクトル実行できるように工夫されているようであるが、V-Pascal Version 1 ではまったくベクトル化しない。

(2)は厳密には4バイトの整数/論理型か、8バイトの実数型しかベクトル・データとしていない。FORTRAN と異なり、Pascal ではプログラマは4バイトの実数型は使用できない。また、現時点では Pascal の文字/文字列/集合型のベクトル化には対応できていない。同様に Pascal の詰め込み配列(packed array)は各要素が4バイト未満となるため、ベクトル化しない。逆に Pascal で記述できるレコード型の一括代入/比較は、レコードの大きさが4バイトあるいは8バイトを越えるため、やはりベクトル化しない。4バイトあるいは8バイトのベクトル・データしか現在のベクトル計算機では、ロード/ストアできないためである。

上記の(2)あるいは(3)には現在のベクトル計算機の複数のベクトル命令、従って複数のベクトル型中間コードに変換することによりベクトル化できるものもある。それらのベクトル化は、今後の課題であるといえる。

従って、D行列の行列構成セルを先頭から末尾まで順にたどり、これらに該当する中間コードについては、6.5.2項で述べた SLOOPLVL フィールドに現在ベクトル化処理を行っている多重ループの最深のネストの深さを記録することにより、すべてのループについてスカラ実行させることを指示する。

6.5.4 ループ構造および制御構造の再構成

ベクトル化は3.2節の図3.4で述べたとおり、ベクトル化の単位ごとに、今の場合中間コードごとにループを分割することと見ることができる。従って、ある中間項を定義する中間コードとその中間項を引用する中間コードとが、ベクトル化前には同一のループに属していた(ループ不変式の場合には定義側中間コードがより外側のループに出ていることもある)のに、ベクトル化後にはまったく別のループに属することが起こり得る。なお、図3.4は簡単のためすべての中間コードがベクトル化されたときのようすを示している。しかし、一般には今までに述べてきた(部分)ベクトル化により一部の中間コード(群)については、それらがベクトル化前に属していた

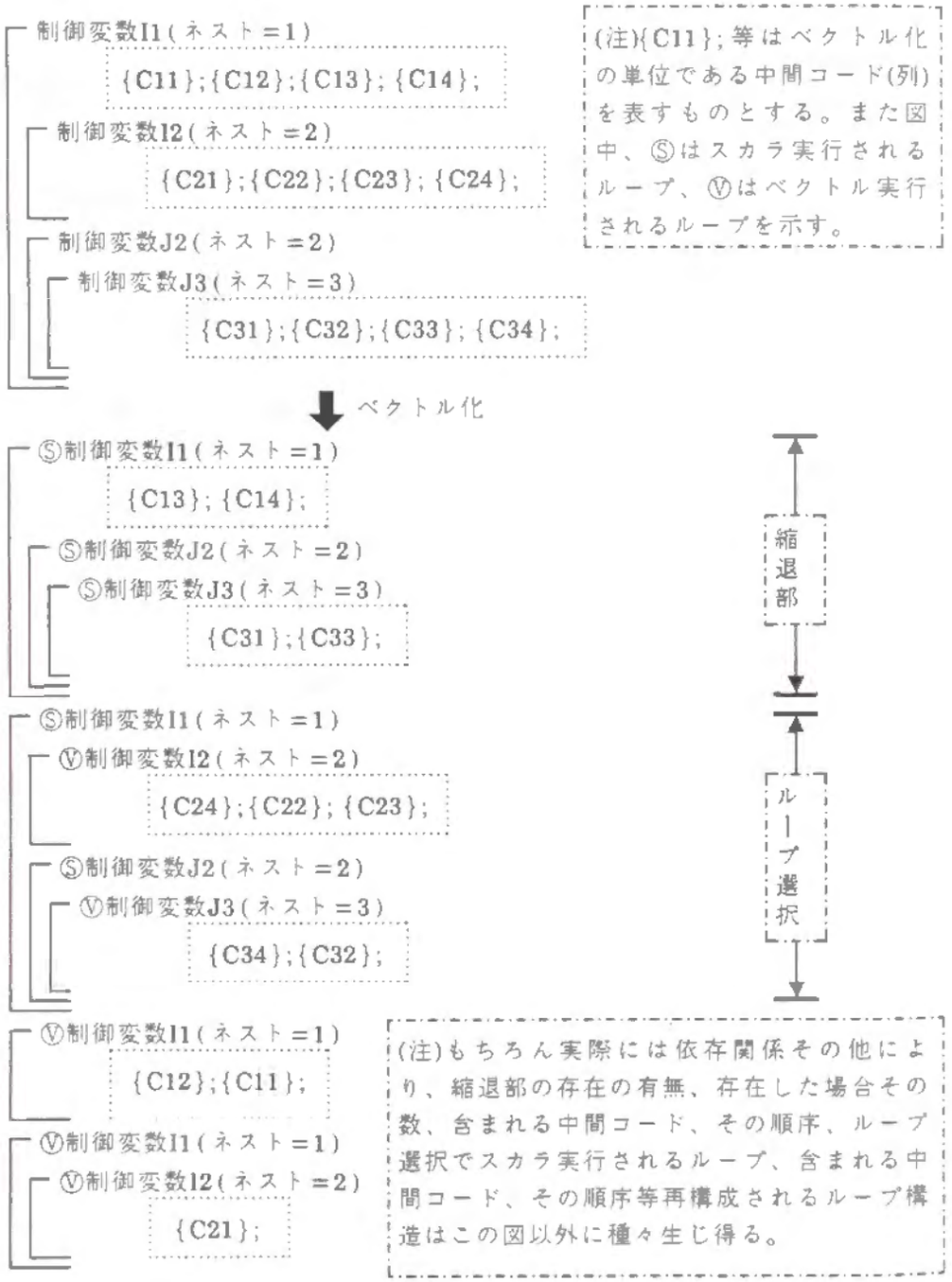


図6.5 ループ選択をともなう部分ベクトル化によるループ構造の変形の概念図

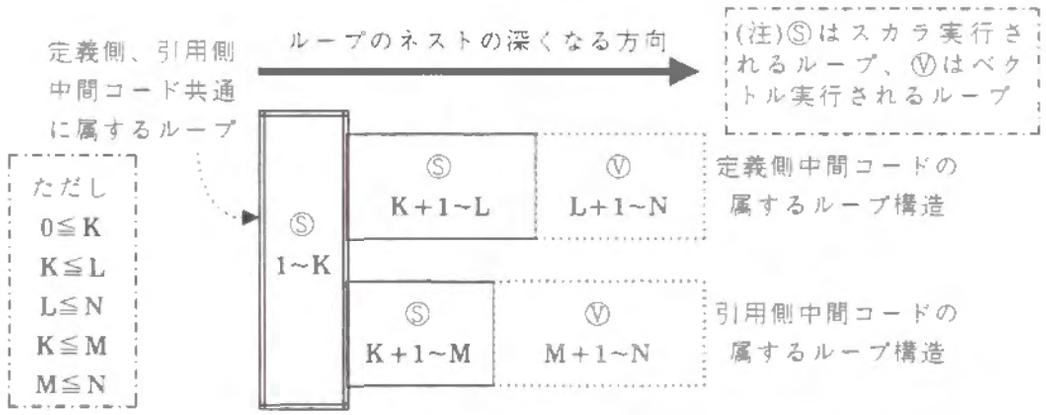
多重ループのうちいくつか(あるいはすべて)のループをスカラ実行せざるを得ないと判定されていると考えられる。このとき、依存関係からスカラ実行と判定されたループを持つ中間コードについては、6.5.3項で述べたループ選択を適用した後も、同一の縮退部に属する中間コード群がD行列において必ず前後に隣接して並べられている。当然、これらの同一の縮退部に属する中間コード群に対して共通にスカラ実行と判定されたループ構造を共有する形としなければならない。つまり、それぞれの中間コードが独立にスカラ実行したのでは、図3.4ですべてのループを分割すること、すなわちすべてのループでベクトル化することと変わらず依存関係が保存されないからである。具体的には、縮退部の下三角部に存在する依存が保存されない。また、ループ選択により無視されたループ運搬依存は、共有するスカラ実行ループによりデータ参照が運搬される(3.2.2項の例5参照)のであって、分割されたスカラ実行ループでは運搬されない。従って、ループ選択をも適用する部分ベクトル化では、図6.5に示すようなループ構造の変形が生じる。もちろん、6.5.3項で述べた依存関係以外の要因でスカラ実行される場合にも、ループの繰り返し判定の回数を減少させるため、共通のスカラ実行ループは極力まとめるべきである。つまり、ループ選択をも適用するD行列並べ換え後の隣接する中間コードにおいて、並べ換え前に共通であったスカラ実行ループは、共有する形でループ構造を再構成するのである。現在はループ交換を行わないためベクトル化後のループのネスト構造の親子関係は、ベクトル化前のそれと変化せず共有できる可能性は高いといえる。なお、条件分岐による制御構造に関しても同様のことがいえ、並べ換え前に共通であったスカラ実行される条件分岐は、共有する形で再構成する。以上ここでは概念的な解説にとどめ、具体的な方法は6.6節で述べる。その6.6節で述べるが、実際に中間コード表現でこれらの制御構造の再構築を行うのは、本節で述べる種々の要因から各中間コードごとにベクトル化するループを完全に決定した後である。

6.5.5 ループ間にまたがる参照関係をもつ中間項の配列化

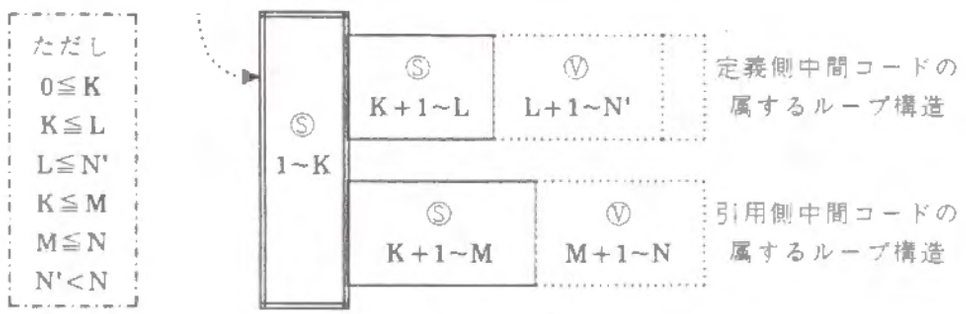
6.5.4項で述べたとおり、ベクトル化前には同一のループに属していたある中間項を定義する中間コードとその中間項を引用する中間コードとが分離され、ベクトル化後にはスカラ/ベクトルの違いも併せ、様々なループ間にまたがる参照関係を引き起こし得る。先の図6.5から一般的には次の図6.6のようなループ間での定義-引用関係が生じるものと考えられる。この図6.6を基にどのような場合にどのように中間項を配列化する必要があるのかを具体的に述べる。

(1)定義側中間コードと引用側中間コードとが、ともにスカラ実行される(図6.6でⓋの印のループがない)場合。

(1-1)定義側中間コードと引用側中間コードとがともに共通のスカラ実行ループに属する場合(図6.6(a)でK=N)には、配列化する必要はない。通常のスカラ実行の場合とまったく同様に、最内側の深さKまでのループの繰り返しごとに1回定義される1回分の中間項を、スカラ-データを保持するレジスタまたは一時領域に保存しておくだけでよい。図6.6(b)では(N'<K=N)、深さN'までのループの繰り返しごとに1回中間項が定義されるだけでその他は同じ。1回分の中間項を保存しておくだけでよい。深さN'+1~Nのループのすべての繰り返しにおいて、その1回分の中間項が引用される。



(a)ベクトル化前定義側、引用側中間コードが同一のループに属する場合



(b)ベクトル化前定義側中間コードがより外側のループに属する場合

図6.6 一般的なループ間にまたがる中間項の受渡しのループ構造

(1-2)定義側中間コードと引用側中間コードとが互いに異なるスカラ実行ループに属する場合(図6.6(a)でL=M=N)には、深さK+1~L(=M=N)のループの繰り返し回数分定義される中間項をその繰り返し順に保存する必要がある。すなわち、配列化する。保存された各繰り返しごとの中間項は、引用側の同じく深さK+1~Lのループの繰り返しにより、その繰り返し順に引用される。図6.6(b)でも(L=N'<M=N)やはり、深さK+1~L(=N')のループの繰り返し回数分定義される中間項をその繰り返し順に保存する必要がある。定義側のループの繰り返し回数1回につき定義される中間項は、引用側の深さL+1~Nのループの繰り返し回数分引用される点を除き、図6.6(a)と同様である。

(2)定義側中間コードと引用側中間コードとが、ともにベクトル実行される場合。

(2-1)図6.6(a)でL=M=Kで、すなわち異なるスカラ実行ループがなく、定義側中間コードと引用側中間コードとが同一ベクトル長の一連のベクトル実行として処理される(すなわち、途中に異なるループ構造に属する中間コードがない)場合、配列化する必要はない。論理的には深さK+1~Nのループの繰り返し回数分定義される中間項すべてが、その繰り返し順にベクトル・レジスタ上に展開して保持され、引用側はそのベクトル・レジスタ上の値を順に引用する。

(2-2)その他の場合はすべて、深さK+1~N(あるいはN')のループの繰り返し回数分定義される中間項をその繰り返し順に保存する必要がある。すなわち、配列化する。

(3)定義側中間コードと引用側中間コードとが、それぞれスカラ実行されるものとベクトル実行されるものに分かれる場合、深さK+1~N(あるいはN')のループの繰り返し回数分定義される中間項をその繰り返し順に保存する必要がある。すなわち、配列化する。

従って、以上を要約すると、定義側中間コードと引用側中間コードとが、スカラ実行/ベクトル実行を問わず別個に独立なループ構造に含まれる場合には、その共通でないループ(図6.6(a)で深さK+1~Nあるいは同図(b)では深さK+1~N'のループ)の繰り返し回数分定義される中間項をその繰り返し順に保存する必要がある。例外的に、上記の(2-1)の場合には、配列化が不要である。

定義される中間項を配列化し、繰り返し順に保存する領域の大きさをどのように設定するかは、次の3種類が考えられる。ここでは、領域の大きさの単位を中間項の型に応じた1回に定義される中間項を記憶するのに要する領域(セル)のサイズを単

位として数えるものとし、何個分のセルが必要となるかで議論をすすめるものとする。

(a)共通でないループの総繰り返し回数個分のセルを用意し、共通でないループを1次元化し、1次元配列として領域を取る。

(b)(a)同様共通でないループの各繰り返し回数の積の個数分のセルを用意し、共通でないループごとに個別に1次元を与えた多次元配列として領域を確保する。

(c)ある一定の個数分だけセルを確保する。

```
for i := 1 to 5 do begin {l1}
    for j := 1 to 3 do begin {l2-1}
        for k := j+1 to 3 do begin {l3-1}
            {中間コード群c1}
        end {l3-1};
    end {l2-1};
            {中間コード群c2}
    for j := 1 to 3 do begin {l2-2}
        for k := j+1 to 3 do begin {l3-2}
            {中間コード群c3}
        end {l3-2};
    end {l2-2};
end {l1};
```

図6.7 ループ間にまたがる中間項の受渡しのあるループ構造の一例

以下図6.7の例を用いて具体的にソースプログラムに近い表記を用い解説する。

この例で、最初の3重ループに属する中間コード群c1から、最後の3重ループに属する中間コード群c3へある中間項が受け渡されるものとする。共通でないループは、3重ループの内側の2重ループである。従って、上記の(a)の考え方をとると、図6.8の1次元配列Tのように配列化することとなる。このとき、共通でないループ群を1次元化し処理するために変数tを用いている。この変数tは、共通でないループのみに関する最内側での繰り返し回数のカウンタとみることもできる。従って、上記の(a)の考え方では、配列化する領域には、共通でないループのみに関する最内側での総繰り返し回数分(今の場合には3)だけの大きさを取ることになる。つまり、論理的に配列化するのに必要十分なサイズといえる。


```
for i := 1 to 5 do begin {l1}
    t := 0;
    for j := 1 to 3 do begin {l2-1}
        for k := j+1 to 3 do begin {l3-1}
            t := t+1;
            T[t] := … ;{中間項定義}
        end {l3-1};
    end {l2-1};
            {中間コード群c2}
    t := 0;
    for j := 1 to 3 do begin {l2-2}
        for k := j+1 to 3 do begin {l3-2}
            t := t+1;
            … := f(T[t]);{中間項引用}
        end {l3-2};
    end {l2-2};
end {l1};
```

図6.8 1次元配列による中間項の配列化

```
for i := 1 to 5 do begin {l1}
    for j := 1 to 3 do begin {l2-1}
        for k := j+1 to 3 do begin {l3-1}
            T[j,k] := … ;{中間項定義}
        end {l3-1};
    end {l2-1};
            {中間コード群c2}
    for j := 1 to 3 do begin {l2-2}
        for k := j+1 to 3 do begin {l3-2}
            … := f(T[j,k]);{中間項引用}
        end {l3-2};
    end {l2-2};
end {l1};
```

図6.9 多次元配列による中間項の配列化

これに対し、上記の(b)の考え方をとると、図6.9の2次元配列Tのように配列化することとなる。このときには、先とちがい補助的な変数tは必要ない。ところが、この例のように共通でないループごとに、独立した次元を対応させるため、それぞれの次元の寸法は、対応するループの繰り返し回数の最大値分に等しくする必要がある。今の場合には3(1次元目)×2(2次元目)となり、明らかに先の方式に比べむだな領域が存在する。

最後に上記の(c)の考え方をとると、図6.10の1次元配列Tのように配列化することとなる。なお、この例では説明を簡単にするため、図6.7とちがい共通でないループが多重ループ構造でないようにしたが、多重ループであってもそれらのループを図6.8のように1次元化し処理すればまったく同様に扱える。また実際には、制御変

```
for i := 1 to 5 do begin {l1}
    for j := 1 to x do begin {l2-1}
            {中間項定義}
    end {l2-1};
    for j := 1 to x do begin {l2-2}
            {中間項引用}
    end {l2-2};
end {l1};
```

(a)配列化前のループ構造

```
for i := 1 to 5 do begin {l1}
    for t1 := 0 to ⌈x/VL⌉−1 do begin {新たに付加されたループ}
        for t2 := 1 to VL do begin {l2-1'}
            j := t1 × VL + t2;
            T[t2] := … ;{中間項定義}
        end {l2-1'};
        for t2 := 1 to VL do begin {l2-2'}
            … := f(T[t2]);{中間項引用}
        end {l2-2'};
    end {新たに付加されたループ};
end {l1};
```

(b)配列化後

図6.10 ソフトウェア・ループ制御を利用した中間項の配列化

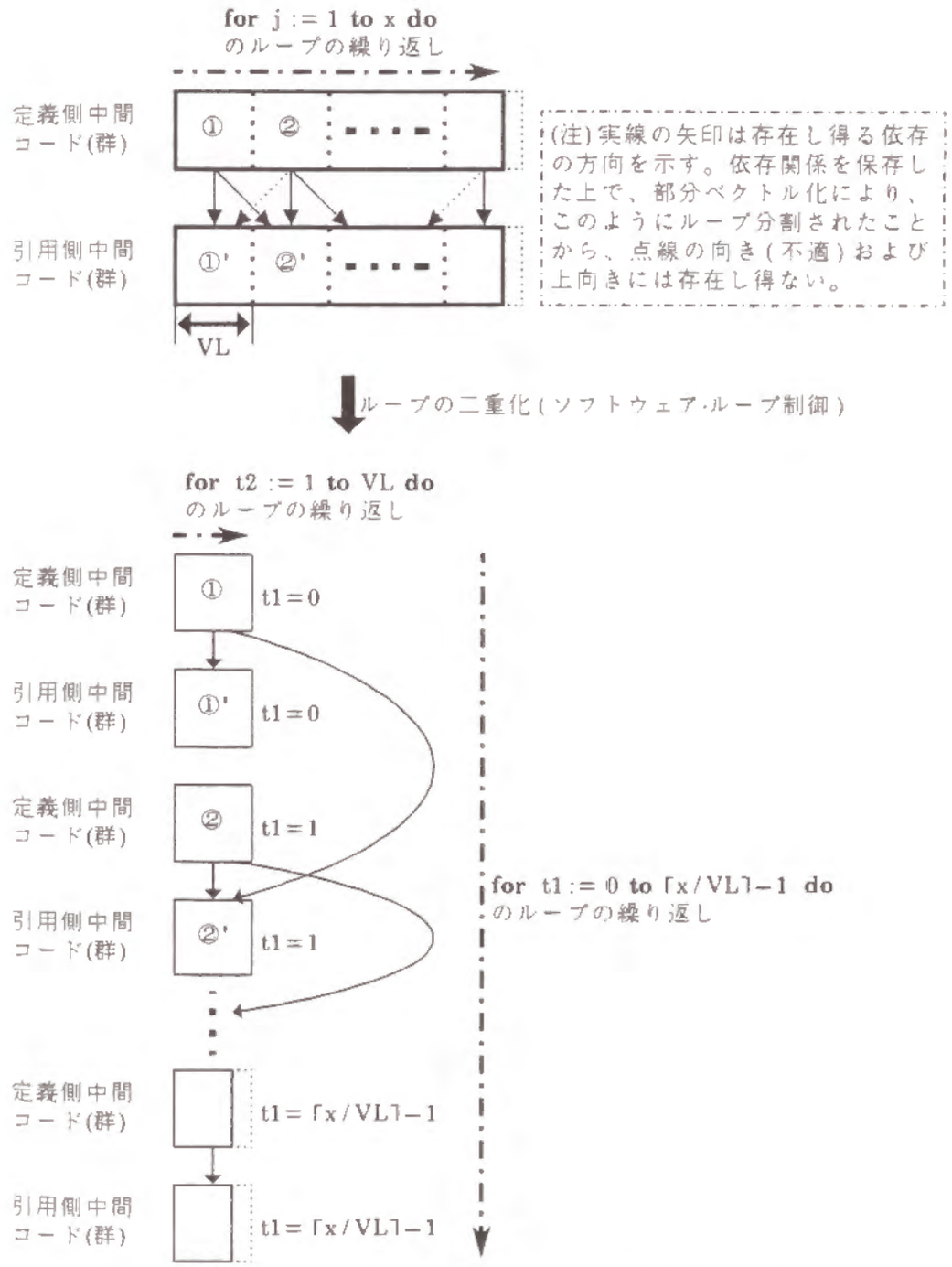


図6.11 ソフトウェア・ループ制御によるループ繰り返しの変化

数t1の最後の繰り返しについては、内側の制御変数t2のループの繰り返し回数は、もとの繰り返し回数(x)からそれまでの繰り返しで実行された分を引いた残りとなるように特別な処理となるが、この図では簡単のため省略した。この図6.10からわかるとおり、同図(a)の配列化前のループ構造の制御変数jの内側のループを、ある繰り返し回数(図中のVL)ごとに切りわけ、同図(b)の配列化後のループ構造の制御変数t1およびt2の2重ループとすることにより、共通でないループは同図(b)の制御変数t2のループのみとなっている。従って、配列化する領域は制御変数t2の繰り返し回数(図中のVL)分だけですむ。このVLは理論的には任意の定数値とできるが、ベクトル計算機の場合にはベクトル・レジスタ長とすると効率的である。こうしたとき、2.2節で述べたループ制御をソフトウェアで行っているとみることができる。なお、このようにあるループを定数回数ごとに切りわけ、2重ループとする手法は一般にはストリップ・マイニング(strip-mining)法と呼ばれる[14]。もちろん、このようなループ構造の変形は、図6.11に示すとおり依存関係を保存するので、意味を変えることはない。ただし、図6.7の中間コードc2ように定義側中間コードが属するループ構造と引用側中間コードが属するループ構造との間に異なるループ構造に属する中間コードが存在する場合には、共通でないループを2重ループに変形させても、図6.10(b)の場合と異なり、共通でないループは制御変数t2のループのみとはならないので、この方式は適用できない。

なお、これら3種類の方式すべてについていえることであるが、もし、ベクトル化前に定義側の中間コードがループ不変式で、引用側の中間コードが属するループよりも浅いループに属している場合にも、これらの図で引用側の中間コードだけにさらに何重かのループが付加されているとみなせばよく、まったく同様の議論が成り立つ。

以上考察してきたこれらの3種類の方式の特徴を表6.1にまとめる。これらの手法の長所/短所を勘案し適用する順位を決定する。このときまず考慮しなければならないのが、実行時の処理時間のオーバーヘッドである。当然方式(c)が、この点で最優先されることになる。配列化のための領域を動的に確保/解放する方式(a)および(b)では、このオーバーヘッドが大きいだけでなく、ベクトル化すべきループの繰り返し回数が通常大きく、そのために配列化のための領域の大きさが禁止的に巨大であることが考えられることからも、極力採用すべきではない。方式(a)と(b)との比較では、スカラ実行されるときにもベクトル実行されるときにも添字計算が簡略な方

表6.1 中間項の配列化の3方式

方式	領域の確保/解放	領域の大きさ	適用条件
(a)	通常動的 (配列化するループの開始前/終了後)	配列化するのに必要十分	配列化するループの開始前に領域の大きさを見積もる計算ができること。かつ、その大きさが十分小さいこと。
(b)	通常動的 (配列化するループの開始前/終了後)	(a)に等しいか(a)より大きい	配列化するループの開始前に領域の大きさを見積もる計算ができること。かつ、その大きさが十分小さいこと。
(c)	常に静的 (通常の作業領域と同様)	常に一定	配列化する定義-引用それぞれが属するループ間に他のループ構造が存在しないこと。

式(b)を優先すべきである。従って、配列化の方式の適用可能性を検討する順位は(c)、(b)、(a)の順となる。なお、方式(a)および(b)で配列化のための領域の確保/解放は、その領域の大きさが見積もれるループの中で、最外側のループで1回だけ行うようにすべきであることはいうまでもない。

このように、ベクトル化すると配列化しなければならない中間項が生じることから、6.4節で述べた中間項の生きている期間を短縮させる処理は、それにより配列化しなければならない中間項の数を減少させ得ること、ならびに上記の方式(c)を適用できる機会を増大させることから非常に重要となる。

6.5.6 中間項の受渡しを考慮した最適ベクトル化ループの決定

6.5.5項で考察した中間項の配列化を考慮し、与えられたあるD行列内に存在する中間項の定義-引用について次のアルゴリズムによりベクトル化すべきループを決定していく。ただし、このときD行列に登録されている中間項の定義-引用関係のうち、ベクトル化することにより引用されなくなる添字式(詳細は次の6.6節で述べる)については考慮する必要がないので、このアルゴリズムの適用前にD行列の該当する非零要素にあらかじめ印を付けておくことにより、対象からはずすものとする。

[中間項の受渡しを考慮したベクトル化ループ判定アルゴリズム]

(1) D行列の行列構成セルを先頭から順に走査し、対応する中間コードがなんらかの中間項(t_xとする)を定義しているもの(行列構成セルxとする)すべてについて以下のステップ(2)から(14)を行う。

(2) 行列構成セルxに関してD行列の第x行を後ろから順に走査し、データ依存のあるすなわちD行列に登録されている中間コードのうち最後に中間項t_xを引用する行列構成セルを見つける。それを行列構成セルyとする。

(3) これまでの処理で行列構成セルxから行列構成セルyまでの各行列構成セルそれぞれに対して与えられている(SLOOPLVLフィールド、表4.13参照)スカラ実行すべき最深のループ情報から、図6.6に示したようなループ構造を抽出する。

(4) 第x行のうち行列構成セルxから行列構成セルyまで走査しながら、データ依存のある行列構成セル(zとする)すべてについて以下のステップ(5)から(14)を行う。すべての定義-引用関係が処理されたなら、行列構成セルxの次のなんらかの中間項を定義しているものを探し、それを新たに行列構成セルxとし、ステップ(2)に戻る。D行列の最後まで処理したならば、スカラ実行すべきループに変化が生じている限りステップ(1)に戻り、処理を繰り返す。変化が無くなれば終了。

(5) 行列構成セルxとそれが定義する中間項t_xを引用する行列構成セルzとの定義-引用関係において、配列化すべき異なるループ構造がなければ、ステップ(4)に戻り中間項t_xを引用する次の行列構成セルを処理する。

(6) 行列構成セルxが属するループ構造と行列構成セルzが属するループ構造との間に図6.7に示したような別のループ構造に属する行列構成セル(中間コード)が存在する場合、その別のループ構造おのおのについて、それ自身and/or行列構成セルxが属するループ構造and/or行列構成セルzが属するループ構造のスカラ実行すべき最深のループを深くすることにより、行列構成セルxが属するループ構造あるいは行列構成セルzが属するループ構造と同じループ構造となるよう調整する。

(7) 調整が成功し別のループ構造が存在しなくなった場合(当初より存在しない場合を含む)、次のステップ(8)へ、存在する場合にはステップ(14)へ。

(8) 行列構成セルxが属するループ構造と行列構成セルzが属するループ構造とで、配列化すべき異なるループ構造がなくなるよう、これらのループ構造のスカラ実行すべき最深のループを深くすることにより調整を試みる。ただし、ベクトル化可能であったループがまったくなくなり、すべてスカラ実行しなければならなくなる場合には調整をあきらめる。調整が成功し配列化すべき異なるループ構造がなければ、ステップ(4)に戻り中間項t_xを引用する次の行列構成セルを処理する。さもなければ、次のステップ(9)へ。

(9) 行列構成セルxが属するループ構造と行列構成セルzが属するループ構造とで、配列化すべき異なるループ構造が1ループとなるよう、これらのループ構造のスカラ実行すべき最深のループを深くすることにより調整する。

(10) 先のステップ(9)で配列化することが決定されたループに、6.5.5項で述べた配列化の方式(c)を適用する。

(11) まず、当該ループが既に二重化されていない場合には、当該ループを二重化する。すなわち、新たにシステム変数を用意し、それを制御変数とする新しいループビギン中間コード/ループエンド中間コードを用意する。ただし、これらの中間コードの追加ならびにそれに伴う中間コード表現でのループ構造の変形は次の6.6節で述べる部分ベクトル化の際に同時に行う。

(12) 次に配列化のための領域を(AVAILリストで管理されている)システム変数として確保する。先のステップ(11)で用意したシステム変数である制御変数を添字として、各繰り返しにおける中間項t_xの値を順に配列化のための領域に待避させるストアの中間コード(および対応する行列構成セル)を行列構成セルxの直後に追加する。同時に同じくシステム変数である制御変数を添字として、同領域から中間項t_xの値を復旧させるロードの中間コード(および対応する行列構成セル)を行列構成セルzの直前に追加する。なお、このステップでの処理は先に別の定義-引用関係において既に中間項t_xが待避/復旧され、重複する場合には省略する。

(13) 行列構成セルzに対応する中間コードが、先のステップ(12)で新たに復旧される中間項t_x'を引用するように、引用する中間項の識別番号を書き換える。ステップ(4)に戻り中間項t_xを引用する次の行列構成セルを処理する。

(14) 行列構成セルxが属するループ構造、行列構成セルzが属するループ構造、ならびにそれらの間の別のループ構造すべてについて、スカラ実行すべき最深のループを深くする。こうして、スカラ実行するループが増えたことにより、通常無視できる依存関係も増えるので、別のループ構造に属する中間コード(対応する行列構成セル)を、行列構成セルxが属するループ構造の前あるいは行列構成セルzが属するループ構造の後ろに可能ならば移動させる。そして、別の

ループ構造がすべて移動できるまでスカラ実行すべき最深のループを深くする。ステップ(9)へ。 □

あるベクトル化対象多重ループに対応するD行列の処理が終了したら、そのD行列の中で配列化のため確保されたシステム変数を、その型ごとにAVAILリストにつないでおくことにより、再利用を図る。

上記のアルゴリズムは、6.5.5項での考察に基づき、配列化の方式の(a)および(b)をまったく適用しないものとしている。しかし、ステップ(7)ならびに(14)でスカラ実行するループを増やすことにより、配列化の方式(c)を適用するようにしている部分では、もし可能ならば配列化の方式の(a)および(b)をも併用することにより、スカラ実行するループを増やす必要がなくなる場合が考えられる。

なお、配列化される中間項が、なんらかの条件分岐の条件節の演算結果の場合には、真/偽値をそれぞれ整数の1/0に変換して待避させ、復旧する場合には逆変換する必要がある。しかも、ベクトル実行される場合には真/偽値はマスク・ベクトルとして保持されていることに注意が必要である。すなわち、スカラ実行での待避では、真のとき1を、偽のとき0をストアするif-then-else構造とするのに対し、ベクトル実行での待避では、マスク・ベクトルが真のとき1を、偽のとき0をストアする二つのベクトル・ストアで表現する。スカラ実行での復旧では、1/0の整数値をロードし、整数値1と等しいか比較する中間コードも必要である。ベクトル実行での復旧では、1/0の整数ベクトルをベクトル・ロードし、整数値1と等しいか比較するベクトル比較の中間コードを付加するのである。

6.6 部分ベクトル化

6.5節までにおいて、部分ベクトル化のための、新しい中間コードの実行順序とそれぞれのスカラ/ベクトル実行すべきループが決定された。この後は、これらのD行列の情報をもとに、実際に中間コード表現で部分ベクトル化を行う。具体的には以下の処理を行う。

[部分ベクトル化の概略アルゴリズム]

(1) ベクトル化された等間隔型参照のロード/ストア中間コード(図4.17参照)に対して、第1番目の参照要素のオフセットおよび参照要素間距離を求める。

(2) 新たな中間コードの実行順序とそれぞれのスカラ/ベクトル実行すべきループに合わせた6.5.4項で述べたループならびに条件分岐の制御構造の再構成を実際に中間コードで表現する。

(3) 同時に、ベクトル実行すべきループについては、もし多重ループでベクトル化可能ならば一重化すべきか否かを、ループ長等の情報を用いて判定する。なお、一重化するかしないかをユーザが指示できるコンパイラ・オプションも用意している。ベクトル処理記述ノードをリンクするために、ベクトル実行するループ(群)ごとにEXVPノード/TVPノードを用意し、4.3.2項の図4.15の構造を形成させる。一重化すべきループであれば5.3節で述べたそれらのループの制御変数の基本ベクトルを生成し、その長さをベクトル長とするためロードする中間コードをEXVPノードの前につなぐ。一重化しないループでは、ベクトル長となる最内側のループ長を計算する中間コードをEXVPノードの前につなぐ。

(4) ベクトル実行するスカラ処理記述ノードの中間コードについては、今までのスカラ実行での演算に相当するベクトル実行での演算を示すベクトル処理記述ノードに置き換える。このとき、(3)で用意したベクトル長を示す中間項番号を記録する。また、もしなんらかの条件分岐の影響を受けている場合には、その条件分岐が参照する条件節の演算結果の中間項番号(もし配列化されていて、別に復旧される場合には、その新しい復旧された中間項番号)をマスク・ベクトルとして引用する。条件分岐の真側/偽側いずれのパスにあるかの情報は、マスク・ベクトルが真の時実行するか、偽の時実行するかを示すMASKPNフィールド(4.3.2項表4.9参照)に残す。

(5) その他の各種の制御の流れを表す中間コードについては、(2)で再構成するのに流用できるものを除き除去する。 □

このうち、(2)から(5)の処理は、現在のバージョンでは1度D行列の行列構成セルを先頭から末尾まで走査し、その順序で対応する中間コードをつないでいきつつ、同時に行っている。

なお、上記の(2)において、隣接する中間コードが同一のループおよび条件分岐の制御構造に属するか否かを調査する必要がある。その調査は、まずそれらが同一のプライマリ・セット(4.3.4項参照)に属するか否かを調べることにより簡単に行うこ

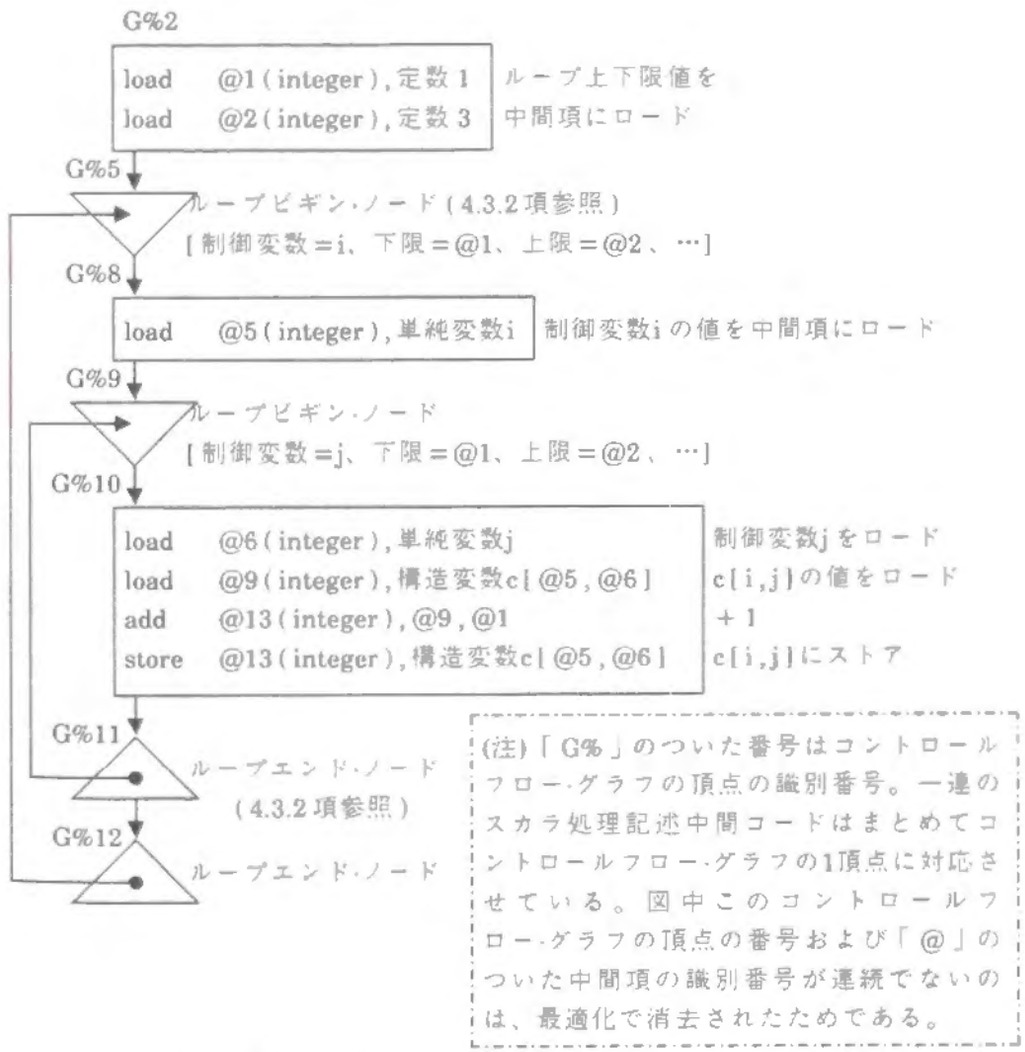
とができる。さらに、同一のループ構造に属するか否か調べる際には、それらが属するコントロールフロー・グラフの頂点が含まれるループ情報を調べることにより、やはり容易に解析できる。また、上記の(4)において、ある中間コードがなんらかの条件分岐の影響を受けているか否か、そして受けている場合には真側/偽側いずれのパスにあるかを調査する場合にも、当該中間コードが属するプライマリ・セットの直属の条件分岐(5.4.2項参照)に関する解析結果を用い、簡単に判定することができる。つまり、D行列をはじめ、V-Pascalがこれまでに行ってきた種々の解析が有効に利用され、複雑な部分ベクトル化機能が簡潔/容易に実現されているといえる。

図6.12は簡単なソース・プログラムについてV-Pascal Version 1でコンパイルした際の中間コード表現を示している。同図(c)は、ソース・プログラム(同図(a))を最内側のループ(制御変数j)でベクトル化した(厳密にはフェーズ4の最適化も終了した)後の中間コード表現である。同図(d)は、別に2重ループ(制御変数iおよびj)を一重化させてベクトル化した(同上)後の中間コード表現である。元のソース・プログラムにベクトル化を阻害する依存がないので、同図(c)および(d)両方ともに、ベクトル化前にループ中にある(同図(b)のG%8およびG%10内の)スカラ処理記述中間コードすべてを、そのままの順序でベクトル化できる。

この図6.12(c)で、最内側のループ(制御変数j)でベクトル化する場合には、上記のアルゴリズムの(3)で、ベクトル長を計算する中間コードが生成され、EXVPノードの直前につながれる。今の場合には、ベクトル長は定数値3であることから、それをロードする中間コードが生成され、つながれたが、フェーズ4の最適化(共通部分式の結果の削除)により除去され、G%2の定数値3をロードする中間コードの中間項@2をベクトル長にも使用している。そして、ベクトル化されるループに含まれる(図6.12(b)のG%10内の)スカラ処理記述中間コードすべてを、上記のアルゴリズムの(4)で、その順にベクトル処理記述ノードに置き換える。まず、最初はループの制御変数jのロードからである。ベクトル化されるループの制御変数jのロードは、制御変数がループの繰り返しに伴い順にとる値を配列化し、ベクトル・データ(今の場合(1,2,3))として生成する(VITR)中間コード(4.3.2項、表4.6参照)に置き換える。ところが、制御変数jの値(中間項@6)は配列cの添字計算に用いられるだけである。そして、制御変数jのループでベクトル化すると、制御変数jの値の変化は配列cの参照要素間距離(図6.12(c)のvl、vstのdist)に置き換えられる。また、制御変数jの初期値(今の場合1)の情報を合わせてオフセット(同じく同図(c)のvl、vst

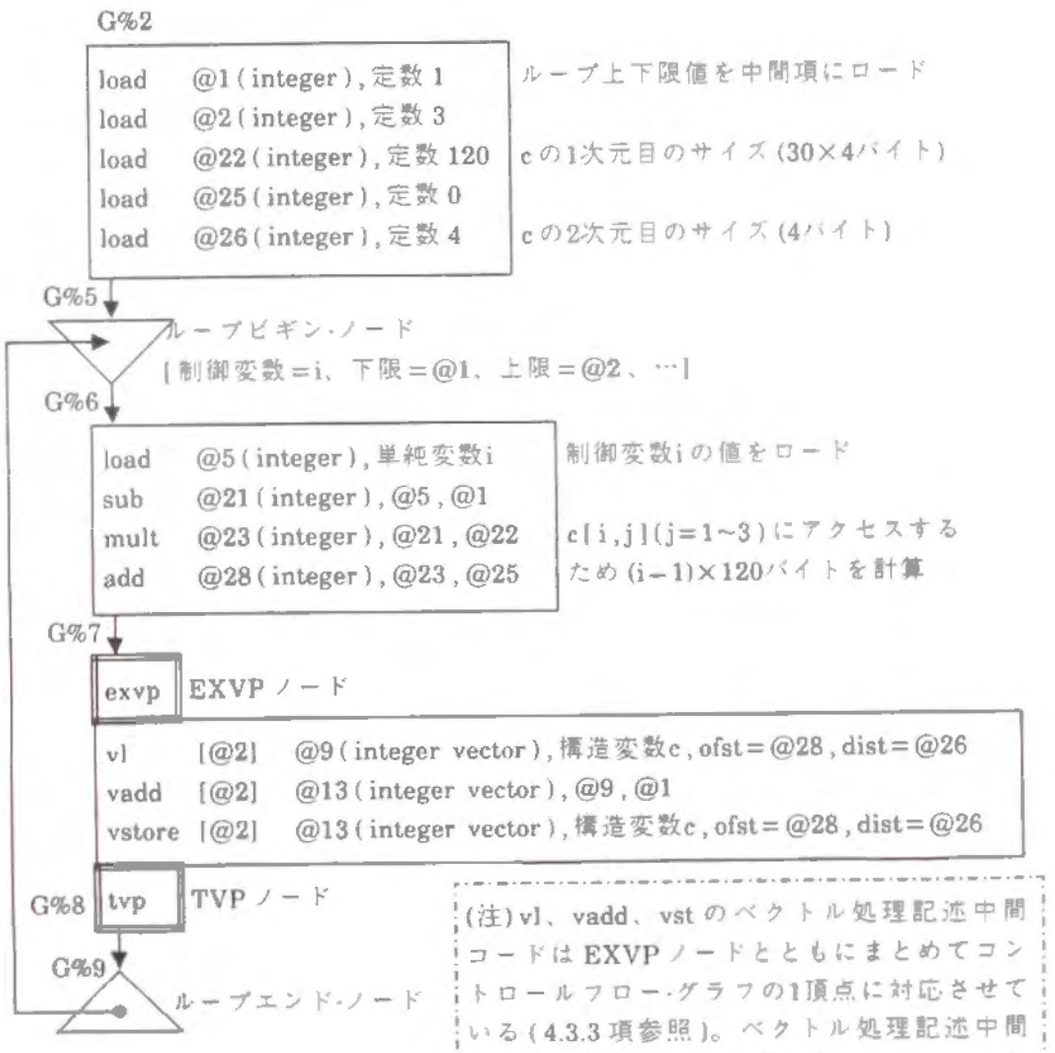

```
for i:= 1 to 3 do
    for j:= 1 to 3 do
        c[i,j]:= c[i,j] + 1;
```

(a) ソース・プログラム(配列cの型は **array**[1..30,1..30] **of** integer)



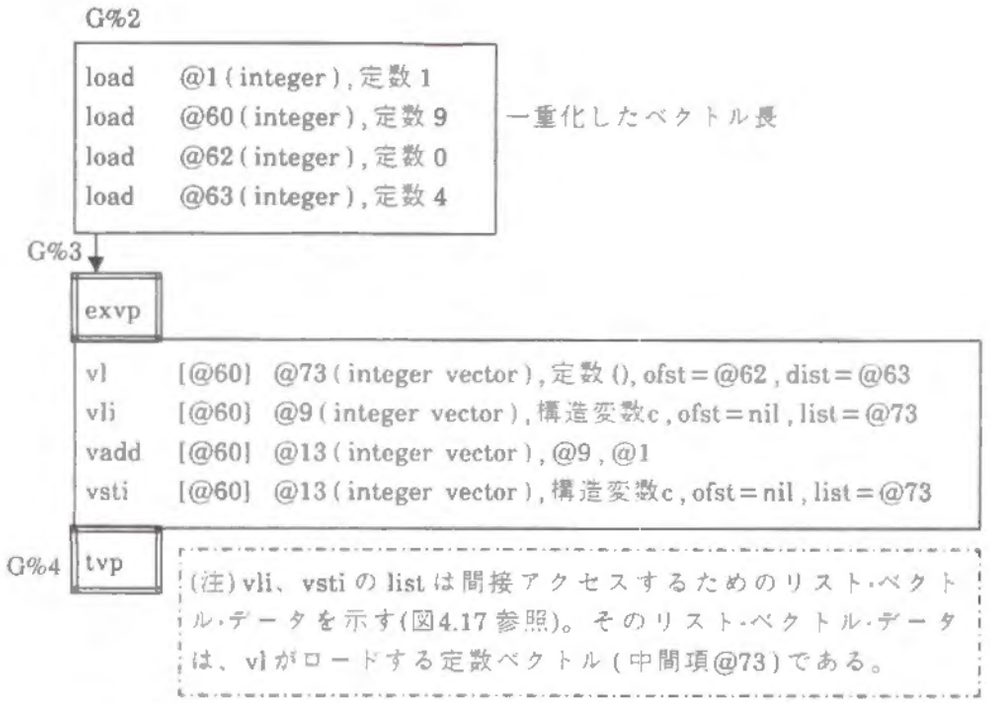
(b) 部分ベクトル化処理(フェーズ3)直前の中間コード表現

図6.12 部分ベクトル化処理適用例(つづく)



(c) 最内側ループのみの部分ベクトル化処理終了後の中間コード表現

図6.12 部分ベクトル化処理適用例(つづく)



(d) 一重化させる部分ベクトル化処理終了後の中間コード表現

使用されるリスト・ベクトル・データ = (0,1,2,30,31,32,60,61,62)
= 30 × {(制御変数iの基本ベクトル) − (1,1,…)}
+ {(制御変数jの基本ベクトル) − (1,1,…)} (†)
ここに(制御変数iの基本ベクトル) = (1,1,1,2,2,2,3,3,3)
(制御変数jの基本ベクトル) = (1,2,3,1,2,3,1,2,3)
なお、制御変数の基本ベクトルに関しては5.3.2項参照。

(e) (d)で使用されるリスト・ベクトル・データ

```
for t:= 1 to 9 do
    c[i¥[t],j¥[t]]:= c[i¥[t],j¥[t]] + 1;
```

{ただし、1次元配列i¥およびj¥はそれぞれ上記(e)の制御変数iの基本ベクトル、制御変数jの基本ベクトルを値として持つものとする}

(f) (d)をソース・レベルで表現

図6.12 部分ベクトル化処理適用例(つづき)

のofst)の計算がなされている。これらの処理は上記のアルゴリズムの(1)ですでに行われている。つまり、ベクトル化されると、もはや制御変数jの値は参照されない。従って、フェーズ4の最適化(無用命令の削除)により、この置き換えられた(VITR)中間コードは除去されている。図6.12(b)のG%10内の次の配列c[i,j]のロードは、先に述べたとおりアルゴリズムの(1)で作られているオフセットおよび参照要素間距離を示す中間項(今の場合、それぞれ@28、@26)を引用したベクトル-ロード中間コードに置き換えている。制御変数jのループでベクトル化するとき、配列c[i,j]のアクセスのためのオフセットは、通常の1次元化を行う配列の添字計算と同様に、{i－(配列cの宣言の1次元目の下限値;1)}×(配列cの宣言の1次元目のサイズ;120)＋{(制御変数jの初期値;1)－(配列cの宣言の2次元目の下限値;1)}×(配列cの宣言の2次元目のサイズ;4)として計算できる。この計算を行うのが図6.12(c)のG%6内の中間コード群である。このようにオフセットを計算することにより、参照するベクトル-データの先頭は、配列cの先頭に等しくなる。すなわち、図4.17に示したVOBJTOPフィールドは配列cとするだけでよい。図6.12(b)のG%10内の加算(add)の中間コードは、図6.12(c)のG%7のベクトル加算(vadd)の中間コードに置き換えられていることがわかる。すなわちこの場合には、オペレーションと中間項(@13)の型を、それぞれadd→vadd、integer→integer vectorと変更するだけである。図6.12(b)のG%10内の最後の配列c[i,j]へのストアは、先のロードとまったく同様にベクトル-ストアに置き換えられる。この時、このストアのために、上記のアルゴリズムの(1)によりオフセットおよび参照要素間距離を計算する中間コード群が、先のロードとは別に作られている。その結果、アルゴリズムの(4)でベクトル化する際には、その別に用意されたオフセットおよび参照要素間距離を使用することとなる。ところが、このストアのオフセットおよび参照要素間距離は、先のロードのオフセットおよび参照要素間距離と同じ計算であるので、フェーズ4の最適化(共通部分式の結果の削除)により除去され、ロードのオフセットおよび参照要素間距離を示す中間項(今の場合、それぞれ@28、@26)を共用する形となる(図6.12(c))。

制御変数iおよびjの2重ループでベクトル化する場合(図6.12(d))には、まず上記のアルゴリズムの(3)で、ベクトル長を計算する中間コードが生成され、EXVPノードの直前につながれる。今の場合には、ベクトル長が定数値9であることから、それをロードする中間コードが生成され、つながれる(図6.12(d)のG%2内)。そして、先の最内側ループのベクトル化の場合と同様に、ベクトル化されるループに含まれ

る(図6.12(b)のG%8およびG%10内の)スカラ処理記述中間コードすべてを、上記のアルゴリズムの(4)で、その順にベクトル処理記述ノードに置き換える。まず、G%8内の制御変数iのロードである。このロードが定義する中間項(@5)も、先述の制御変数jのロードの中間項(@6)同様、図6.12(b)のG%10内の配列c[i,j]のアクセスのための添字式中で引用されるだけである。すなわち、この制御変数iのロードは、本来配列c[i,j]のアクセスの中間コードとともにG%10内に存在したものが、フェーズ4の最適化(ループ不変式のループ外への移動)により、G%8内に移動されたものである。従って、この制御変数iのロードを2重ループでベクトル化する場合には、図6.12(e)に示す制御変数jの基本ベクトル(コンパイル時に計算できる)を定数値ベクトルとしてロードするベクトル-ロードの中間コードに置き換える。この時、定数値ベクトルは各要素が連続しているため、オフセットおよび参照要素間距離は、それぞれ定数0、4の値を持つ中間項を示す。同様に図6.12(b)のG%10内の制御変数jのロードの中間コードは、制御変数jの基本ベクトルをロードするベクトル-ロードの中間コードに置き換える。ところが、これらの制御変数iおよびjの基本ベクトルをロードするベクトル-ロードの中間コードが定義する中間項は、先に述べたとおり配列c[i,j]のアクセスのための添字式中で引用されるだけである。そして、その添字式の計算は、スカラ実行される場合とまったく同様であり、図6.12(e)の(†)の式で計算される。しかも、制御変数iおよびjの基本ベクトルがともに定数値ベクトルであるため、フェーズ4の最適化(定数の畳み込み)によりコンパイル時に計算され、図6.12(e)に示した1本のリスト-ベクトル-データにまとめられる。つまり、制御変数iおよびjの基本ベクトルをロードするベクトル-ロードの中間コード2ノードは、図6.12(d)のG%3内の最初のベクトル-ロードの中間コード1ノードにおきかえられている。図6.12(b)のG%10内の配列c[i,j]のロード/ストアの中間コードは、このリスト-ベクトル-データを使用した間接参照型のベクトル-ロード/ベクトル-ストアの中間コード(図6.12(d)のG%3内のvli、vsti)に置き換えられる。これらの間接参照型のベクトル-ロード/ベクトル-ストアの中間コードのLISTVCTRフィールド(図6.12(d)では単にlistと記した)は、アルゴリズムの(4)で処理した際には、上述のとおり制御変数iおよびjの基本ベクトルから添字式(図6.12(e)の(†)の式)の値を計算させる一連のベクトル処理記述中間コードの計算の最終結果を表す中間項の番号が書き込まれている。それが、上述のとおりフェーズ4の最適化(定数の畳み込み)によりまとめられた1本のリスト-ベクトル-データをロードしてきた中間項の番号(図6.12

(d)では@73)に最適化処理部が書き換えたのである。図6.12(b)のG%10内の加算(add)の中間コードに対する処理は、先の最内側ループのベクトル化の場合とまったく同様である。図6.12(f)は、この一連の中間コード上での一重化処理を、ソース・コードで表現したものである。

以上述べてきたとおり、上記のアルゴリズムの(1)では、連続アクセスを含む等間隔アクセスとなるか否か調べるため(そして連続アクセスを含む等間隔アクセスの場合には上記のオフセットおよび参照要素間距離を計算する中間コード群を生成するため)、ベクトル実行されるロード/ストアの中間コードが引用する添字式がaI+b(a、b定数、厳密にはループ不変式)と、ベクトル実行される(最内側の)ループ制御変数Iに関する線形の1次式となっていることを調査する必要がある。多次元配列に対するロード/ストアでは、添字式が次元数分存在し、それらすべてが、前記の形のループ制御変数に関する線形の1次式でなければならない。その場合には、次元数分存在する添字式を1次元化し、1添字式としてまとめてから、調査することにより、1次元の変数と同様に扱うことができる。その調査のアルゴリズムは以下のとおりである。

[添字式がループ制御変数に関する線形の1次式を求めるアルゴリズム]

(1) 与えられた添字式の演算結果を示す中間項をt_xとする。以下のステップ(2)から(7)を再帰的に繰り返す。

(2) 中間項t_xを定義している中間コードxの演算の種類により、以下のステップ(3)から(7)の場合わけを行う。

(3) 中間項t_xがループ不変式の場合、すなわち中間コードxがベクトル実行されるループ外に存在する場合、a=0、b=中間項t_xの識別番号としてリターン。

(4) 中間コードxがロードの場合、もしベクトル実行されるループ制御変数ならば、a=1、b=0としてリターン。その他の変数の場合には、線形式ではないとしてリターン。

(5) 中間コードxが加減算のとき、第1被演算子をt_xとして、再帰的にステップ(2)に戻りその結果を(aI+b)とする。同様に第2被演算子をt_xとして、再帰的にステップ(2)に戻りその結果を(a'I+b')とする。いずれか一方でも線形式でなかった場合には、この結果も線形式ではないとしてリターン。さもなければ、得られた(aI+b)と(a'I+b')とから、この結果の線形式の係数を求めリターン。

(6) 中間コードxが乗算のとき、第1被演算子をt_xとして、再帰的にステップ(2)に戻りその結果を(aI+b)とする。同様に第2被演算子をt_xとして、再帰的にステップ(2)に戻りその結果を(a'I+b')とする。いずれか一方でも線形式でなかった場合ならびにaおよびa'がともに非零の場合には、この結果も線形式ではないとしてリターン。さもなければ、(aI+b)と(a'I+b')とから、この結果の線形式の係数を求めリターン。

(7) その他の演算、関数呼び出しの場合にはこの結果は線形式ではないとしてリターン。 □

6.7 結 語

多重ループのベクトル化は、第5章で述べた各種依存関係解析が非常に複雑となること、ならびに、ループ選択等種々の高度なベクトル化手法を組み合せることができ、採用し得る戦略に関して選択の自由度が高いことの2点において最内側のみのループのベクトル化とは大きく異なる。この点を考慮した上で、V-Pascalでは多重ループのベクトル化を設計開始時点から目標とし、第5章で述べた高機能の各種依存関係解析アルゴリズムを採用し、その解析結果をD行列としてまとめることにより、中間コードを単位とする順序交換および部分ベクトル化、配列化、ループ選択までも、D行列に対する一連の操作として実現している。また、プライマリ・セットおよびその直属の条件分岐の解析、ならびにコントロールフロー・グラフおよびその頂点の属するループの解析等V-Pascalがこれまでに行ってきた種々の解析が、この部分ベクトル化処理部において集約され有効にかつ適切に利用されることにより、複雑な部分ベクトル化機能が、簡潔に記述され実現されているといえる。

なお、ここで述べた部分ベクトル化機能は、ソース言語によらないことはいうまでもないが、一重化すべきか否かの判定を除き、ターゲット・マシンにも依存していない一般的な技術となっている。従って、他のソース言語あるいは他のターゲット・マシンを対象とする自動ベクトル化コンパイラにもそのまま実現可能であると考える。

第5章で述べた各種依存関係解析は、ループ交換を含むループ選択に対応できる解析結果を得ることができるが、ここで述べた現在の部分ベクトル化では、そこまでのベクトル化を行っていない。これは、その他さらにベクトル化を促進させる手法

を実現することと合わせ今後の課題である。特に多重ループのベクトル化では、複数のループ構造が存在する上に、実行順序の交換を含む部分ベクトル化により、さらに複雑なループ構造が出現する。それゆえに、6.5.6項で述べた配列化を考慮した中間項の受渡しにおいてさらに工夫することも重要であると考える。例えば1中間コードを複製し、一方をスカラ実行し、他方をベクトル実行させることにより、配列化を避けたベクトル実行も可能となる。単純な場合には、配列化のオーバヘッドと比較して、この複製による再計算の方が高速である。このような考察もさらに深く行う必要がある。ただし、このような実行時のオーバヘッドを勘案した戦略の選択においては、ターゲット・マシンに依存せざるを得ず、機械依存部を圧縮するという観点から注意が必要であると思われる。

第7章
目的コード生成

7.1 緒 言

V-Pascal Version 1では、ユーザ定義の手続/関数ごとに各種解析、ベクトル化、最適化を行っており、目的コード生成も同様である。目的コード生成部は、その性格上ターゲット・マシンに依存した記述となるが、V-Pascal Version 1ではできる限り依存した記述をまとめ、処理を明確に切りわける意味で、以下のように複数回中間コードを走査する実現方法を採用した。

(1) ユーザ定義の手続/関数の本体部開始用目的コード生成部
(2) ベクトル・ユニットの各レジスタ割り付け部
(3) 中間項の配列化処理部
(4) ベクトル・ユニットのセットアップ用中間コード生成部
(5) 中間項の参照関係調査部：トレース1
(6) ユーザ定義の手続/関数の本体部の目的コード生成部：トレース2
(7) ユーザ定義の手続/関数の本体部終了用目的コード生成部

なお、すべてのユーザ定義の手続/関数について目的コード生成が終了した後、リンカ(Version 1のターゲット・マシンであるHITAC S-820の場合は、スカラ処理部がIBM 360/370アーキテクチャであるので、厳密にはリンケージ・エディタ)に対する指示のための各種カードを出力し、翻訳を終了する。

上記の(1)および(7)は、7.2節で述べるPascalの実行時ライブラリ(Runtime Library)[31]~[35]とのインタフェースとなる目的コードを生成するもので、定型的なスケルトンとなっている。それゆえに、これらの処理では中間コードは実質上まったく走査しない。上記の(1)では、終了時ダンプ(Postmortem Dump)[31]~[35]情報の所在を示す目的コード、当該手続/関数の本体部で必要となるファイルをオープンするための(実行時ルーチン呼び出しの)目的コードも生成する。(7)では、実行時ライブラリを経由して当該手続/関数の本体部からリターンする目的コード、当該手続

/関数の本体部内のベクトル処理記述の中間コードから得られるベクトル命令列の目的コード、当該手続/関数の本体部内で使用されるスカラおよびベクトル定数データの目的コード、終了時ダンプ情報の目的コードを生成する。

上記の(2)から(4)は、当該手続/関数の本体部内のベクトル処理記述の中間コードのみを走査する、ベクトル命令生成のために必要な処理である。第2章で述べたとおり、ベクトル計算機は必然的にベクトル処理部とともにスカラ処理部を合わせて持つアーキテクチャとなる。しかも、これらの処理ユニットは高速化のため並列実行できるよう独立した構造となる。すると、ベクトル処理に関する目的コード生成は、スカラ処理との連携を除き、ほとんど独立に考えることが一般に可能となる。特に、V-Pascal Version 1のターゲット・マシンであるHITAC S-820シリーズでは両者間の独立性が高く、このようにベクトル命令生成に必要となる処理を完全に切り離すことができる。

これら(2)から(4)のベクトル命令生成に必要な処理のうち、先の(2)のベクトル・ユニットのレジスタの割り付けを行ったとき、レジスタが不足する場合には本来1回のベクトル・ユニットの起動で処理できる一連のベクトル処理記述の中間コード列を複数回の起動で処理させなければならない。これは、ベクトル実行されるループを分割する処理に相当する。従って、そのように複数回のベクトル・ユニットの起動に分断された一連のベクトル処理記述の中間コード列内で、中間項の定義と引用が別の処理単位に分離することがある。その場合、ベクトル・レジスタ上に保持されている中間項を主記憶に待避/復旧する操作が必要となる。これが上記の(3)で行う中間項の配列化であり、それゆえに6.5節で述べた通常の中間項の配列化とは別に処理している。しかし、配列化の方式については、まったく同様にでき、6.5.5項で述べた方式(c)を適用する。すなわち、本来の論理的なベクトル長を、物理的なベクトル・レジスタ長に切り分けてその長さでベクトル処理させる。つまり、本来の論理的なベクトル長を2重ループ化し、内側のベクトル・レジスタ長のループをベクトル実行する。言い換えれば、2.2節で述べたループ区分処理をソフトウェアで制御する。しかも、こうすることにより、V-Pascal Version 1のターゲット・マシンであるHITAC S-820シリーズについては、連続するベクトル・ユニットの起動間では、ベクトル・ユニットのベクトル・レジスタ、スカラ・レジスタ、マスク・レジスタ(2.2節参照)の値が自動的に引き継がれるので、中間項を主記憶に待避/復旧する操作は不要となる。従って、この場合には単に本来の論理的なベクトル長を2重ループ化し、ソフトウェア・ループ制御を実現する中間コードとするだけでよい(詳細は7.3節で述べる)。

上記の(5)および(6)のみが、すべての中間コード列を走査する。それゆえに、これらの処理をそれぞれトレース1およびトレース2と呼称している。前者の(5)のトレース1と呼んでいる中間項の参照関係調査部では、トレース2の実質上の目的コード生成過程とまったく同じ順序で中間コードのなぞりを行い、中間項の引用関係の調査を行う。具体的には、次の2種類の引用関係をまとめる。

(5-1) 各中間項を最後に引用する中間コードを調べる。

(5-2) 引用が定義よりも先に出現する中間項をリスト・アップする。

この(5-1)の調査は、レジスタ割り付けのために行う。各中間項はできるだけレジスタに置くことが望ましい。数少ないレジスタを有効に使用するためには、各レジスタに保持されている中間項が、もはや引用されることがなくなったときには即刻、そのレジスタを解放する(その中間項の値の保持を止める)ように管理しなければならない。ある中間項がもはや引用されることがないかどうかは、データフロー解析の結果から簡単に判定できる(4.7節参照)。しかし、このようなレジスタ管理の観点からすると、コントロールフロー・グラフの頂点というデータフロー解析の単位はあらすぎて、そのまま利用すると無用に中間項の値を保持することとなる。なぜなら、ある中間項について、データフロー解析の結果から、それがもはや引用されないと確定できるのは、その中間コードが属するコントロールフロー・グラフの1頂点に含まれる複数の中間コードすべての処理がすんだ時点である。従って、ある中間コード以降で引用されない中間項は、その中間コード以降の、その中間コードが属するコントロールフロー・グラフの1頂点に含まれるすべての中間コードの処理の間、無用にレジスタ上に保持されるからである。これを避けるため、各中間項について、それを最後に引用する中間コードを調べ、その(中間項→中間コード)の対応関係(写像)を表にまとめることとした。この表の実現に際しては物理的には単純なリニア・リスト(その先頭は変数LASTREFHEADが指す)のデータ構造を採用している。なお、この表を検索するための関数IILASTREFを用意した。

(5-2)では、制御の流れでは起こり得ないが、この目的コード生成部で採用している中間コードのなぞりで、引用が定義よりも先に出現する中間項を調査し、それらの中間項に、待避のための領域として主記憶上に一時領域を割り付けておく。この処理は、次のような理由から必要となる。トレース2において各中間項はそれが定義

される時点でレジスタを割り付けられるため、引用が定義よりも先に出現する中間項については、その引用される時点ではレジスタは割り付けられておらず、目的コード生成ができなくなるからである。従って、このような場合、あらかじめトレース1で割り付けられていた、主記憶上の一時領域を参照する目的コードを生成することとした。当然、当該中間項を定義する中間コードから対応する目的コードを生成する際には、対応する一時領域に値を待避する目的コードも付加しておく。トレース1では、引用が定義よりも先に出現する中間項とそれに割り付けた一時領域の番地の表を作成する。この表も先と同様物理的には単純なリニア・リスト(その先頭は変数ITUNDEFHEADが指す)のデータ構造を採用している。なお、ここで使用する主記憶上に割り付けられる一時領域をはじめ、レジスタ待避等のため目的コード生成の際に確保される一時領域はすべて、6.5節で述べた配列化のために確保される一時領域等他のフェーズで使用される一時領域と合わせて、AVAILリストで管理し再利用を図っている。そのため、ここでの一時領域については、対応する中間項がもはや引用されることがなくなった時点で、AVAILリストに戻される。

7.2 実行環境の構造

Pascalの処理系は一般に実行時には、記憶領域を(a)データ領域と(b)コード領域とに分け使用する。V-Pascalの**実行環境**(runtime environment)[37]は、Pascal 8000[31]~[35]とまったく同じに設計されている。以下順にこれらについて解説する。

7.2.1 データ領域

実行環境のうち、(a)のデータ領域としては、

(a-1) スタック領域

(a-2) ヒープ(heap)領域[31]~[37]

の2種類が割り付けられる作業領域を用意している。この(a-2)ヒープ領域とは、Pascalの標準手続newによりユーザ・プログラム実行中に動的に確保される領域を割り付ける部分をいう。これらの領域は独立して管理されるが、物理的には図7.1に示すように一つの大きな作業領域として後述の実行時ライブラリにより確保される。この図からわかるとおり、スタック領域は、Algolに代表されるブロック構造を有する言語のブロックの実行記録(あるいは活動記録、activation record)[36],[37]を保持

するために利用される。具体的には、Pascalのユーザ手続/関数が呼び出されたとき、そのブロックのための実行記録がスタック上に積まれ(push down)、その手続/関数からリターンするとき、その実行記録がスタックから取り除かれる(pop up)。

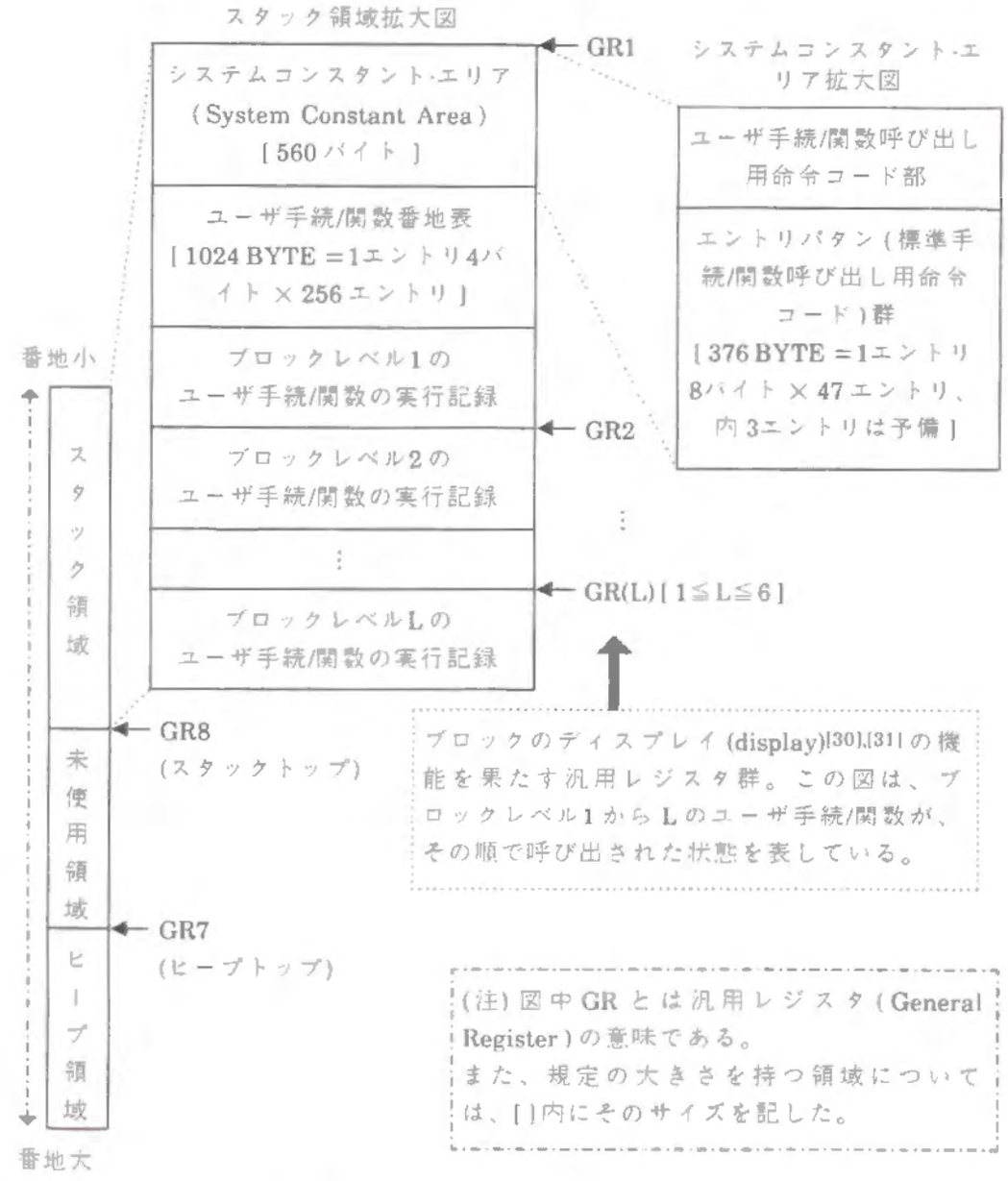


図7.1 実行時のデータ領域の構造[31]~[35]

従って、再帰的に呼び出される手続/関数の場合には、各呼び出しごとに実行記録がスタック上に複数存在することとなる。そして、それぞれのブロック(手続/関数)について、実行時のある時点で1個以上存在する実行記録の中から有効な実行記録を指すディスプレイ(display)[36],[37]ポインタとして、当該ブロックの静的な(宣言時の)ネストレベルに等しい番号の汎用レジスタ(General Register、GRと略記する)を固定的に使用している。現在、実行記録は次の領域からなる。

(1) 呼び出し側の汎用レジスタ待避領域(64バイト)

(2) 関数の場合の返却値(return value)のための領域(8バイト)

(3) 当該手続/関数の仮引数用の領域:番地呼び(call by reference)[36],[37]の変数引数は1個につき4バイト、値呼び(call by value)[36],[37]の変数引数はその型に応じたサイズ、手続/関数引数は1個につきそれぞれ64バイト/72バイト[31]~[35]

(4) 当該手続/関数の局所変数用の領域:各変数それぞれの型に応じたサイズ

(5) 当該手続/関数の実行時の一時領域(第6章で述べたベクトル化、配列化等のために使用される):それぞれの目的に応じたサイズ

なお、図7.1に示したとおり、ネストレベル1のユーザ・プログラムのメイン・ルーチンの場合、ディスプレイ・ポインタである汎用レジスタ1(GR1)は、メイン・ルーチンの実行記録とともにシステムコンスタント・エリア(System Constant Area)およびユーザ手続/関数番地表をも合わせて指示する。これらの領域は、後述するとおりユーザ・プログラムの実行中常に参照できる必要性から、図示したとおりスタックの先頭に配置されている。

7.2.2 コード領域

他方の(b)コード領域としては(b-1)コンパイラが生成するユーザ・プログラムの目的コードと、(b-2)実行時ライブラリ(Runtime Library)とからなる。前者のユーザ・プログラムの目的コードは、実行環境である後者の実行時ライブラリとリンクされ実行される。なお図7.1に示したとおり、厳密には先のスタック領域の先頭にあるシステムコンスタント・エリアも命令コードの系列である。しかし、これらの命令コード列は実行時ライブラリとともにアセンブルされ、実行時ライブラリにより確保される作業領域(スタック領域)の先頭に、実行時ライブラリにより複写されるものであるので、コード領域とはみなさないものとする。図7.2に実行時のモジュール

構成を示す。図中に示した実行時ライブラリのユーザ・プログラムの目的コードから見て、呼び出し関係で親モジュールの機能の概略を表7.1にまとめる。また、表7.2に標準手続/関数用サブルーチン群の各モジュールの機能の概略をまとめる。なお、

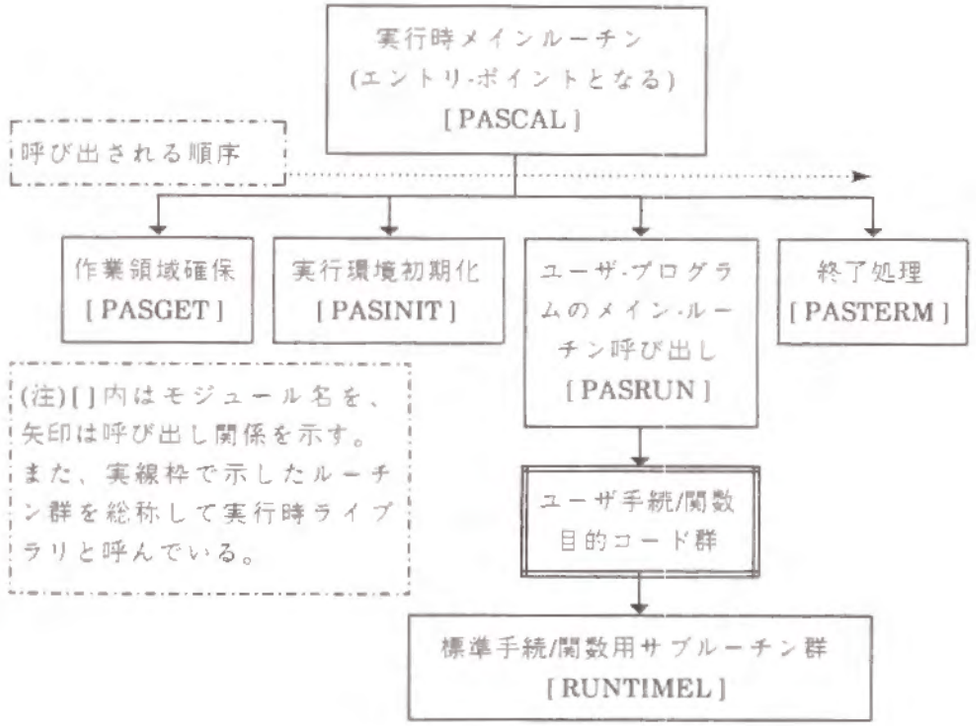


図7.2 実行時のモジュール構成[31]~[35]

表7.1 実行時ライブラリの親モジュールの機能概略[31]~[35]

モジュール名	機能
PASCAL	実行時のメインルーチンであり、エントリポイントとなる。
PASGET	図7.1に示した作業領域を確保する。
PASINIT	作業領域の初期化、ユーザ手続/関数番地表の記入、タイマのセット、標準出力ファイルのオープン。
PASRUN	作業領域の先頭にシステムコンスタント・エリアのコードを複写、ユーザ・プログラムのメイン・ルーチン呼び出し。
PASTERM	クローズされていないファイルのクローズ、異常終了の場合に終了時ダンプ(postmortem dump)の出力等。

表7.2 標準手続/関数用サブルーチン群のモジュールの機能 (つづく)[31]~[35]

エントリ名	機能
RUNTIME#	全標準手続/関数用実行時ルーチン群の入口。レジスタ待避、もしあれば、実引数のレジスタへのロードを行い、以下の各標準手続/関数用サブルーチンへジャンプ。
PSIN	標準関数 sin 計算。
PCOS	標準関数 cos 計算。
PEXP	標準関数 exp (e の x 乗) 計算。
PSQRT	標準関数 sqrt (平方根) 計算。
PLN	標準関数 ln (自然対数) 計算。
PARCTAN	標準関数 arctan 計算。
OPENEXT	外部ファイルのオープン用標準手続。
CLOSEEXT	外部ファイルのクローズ用標準手続。
GET	標準手続 get (ファイルの1レコード読み込み) の処理。
PUT	標準手続 put (ファイルの1レコード書き込み) の処理。
RESET	標準手続 reset (入力ファイルのオープン) の処理。
REWRITE	標準手続 rewrite (出力ファイルのオープン) の処理。
WRITBOOL	テキスト型ファイルへの Boolean 型の値の出力。
WRITEINT	テキスト型ファイルへの整数型の値の出力。
WRTREALE	テキスト型ファイルへの実数型の値の指数付き浮動小数点形式の書式での出力。
WRTREALF	テキスト型ファイルへの実数型の値の固定小数点形式の書式での出力。
WRITCHAR	テキスト型ファイルへの文字型の値の出力。
WRITSTRG	テキスト型ファイルへの文字列型の値の出力。
READINT	テキスト型ファイルからの整数型の値の入力。
READREAL	テキスト型ファイルからの実数型の値の入力。
READCHAR	テキスト型ファイルからの文字型の値の入力。
READLN	入力テキスト型ファイルの1行改行。
OPNINPUT	標準入力テキスト型ファイルのオープン。

この表のうち最初のモジュール RUNTIME# を除き、各モジュールは先の図7.1 に示したシステムコンスタント-エリア内のエントリパタン (標準手続/関数呼び出し用命令コード) 群中に登録されている。この表に記載された順序は、現在の対応するエントリパタンの順に合わせている。

表7.2 標準手続/関数用サブルーチン群のモジュールの機能 (つづき)[31]~[35]

エントリ名	機能
GETTEXT	テキスト型ファイルに対する標準手続 get の処理。
PUTTEXT	テキスト型ファイルに対する標準手続 put の処理。
RESETTXT	テキスト型ファイルに対する標準手続 reset の処理。
REWRTTXT	テキスト型ファイルに対する標準手続 rewrite の処理。
ERROR1	未定義の case ラベルへのジャンプによる異常終了。
ERROR2	部分範囲型変数等への範囲を越えた参照による異常終了。
ERROR3	ポインタ変数による作業領域外への参照による異常終了。
ERROR4	スタックあふれによる異常終了。
ASSLONG	256 バイト以上の変数等領域の複写。
RELLONG	256 バイト以上の変数等領域の比較。
CLOCKF	標準手続 clock (CPU 時間測定) の処理。
TIME	標準手続 time (現在時間の読み出し) の処理。
DATE	標準手続 date (実行された時点の日付の読み出し) の処理。
INTPOWER	x (整数型の値) の y (整数型の値) 乗の計算。
HALT	標準手続 halt (ユーザ-プログラムの停止) の処理。
PAGE	出力テキスト型ファイルの改ページ。
WRITELN	出力テキスト型ファイルの1行改行。
MESSAGE	コンソール画面への1行出力。
LONGJUMP	ロングジャンプ (4.3.2 項、図4.11 参照) の処理。
READSTRG	テキスト型ファイルからの文字列型の値の入力。
TLIMIT	指定された CPU 実行時間が終了した際の処理。

7.2.3 ファイル管理

7.2.2項の表7.2でテキスト型ファイルとは、Pascalで file of char あるいは text と型宣言される文字コードのみからなるファイルをいう。つまり、ユーザ可読ファイルである。これに対しノンテキスト型ファイルとは、文字以外の型を含むファイルをいう。従って、一般にはバイナリファイルとなる。具体的な入出力標準手続/関数としては、ノンテキスト型ファイルについては、get、put、reset、rewrite、eofのみが使用できる。テキスト型ファイルでは、これらに加え、read(ln)、write(ln)、eolnも使用できる。すなわち、ノンテキスト型ファイルは既定のフィールドからなるレコード単位の入出力であり、テキスト型ファイルでは1レコードは1行の概念に対応し、各行は既定されない任意の文字列で構成されるストリーム型の入出力であるといえる。ただし、このテキスト型ファイルにおいても、実際に入出力を行う際には効率を考慮しレコード単位の入出力を行う。そのために、これらの入出力を行う標準手続/関数用実行時ルーチン内では、実際に入出力を行うレコードをバッファ(以下ファイルバッファ)で管理している。このファイルバッファを管理するためのデータ構造のファイルブロックの構成フィールドを表7.3に示す。ノンテキスト型ファイルについても表7.4のようなファイルブロックが用意される。これらのファイルブロックは、そのファイルを宣言したユーザ手続/関数内の局所変数と同様に、当該ユーザ手続/関数内のブロックの実行記録の領域に割り付けられ使用される。

7.2.4 ユーザプログラムの目的コードの構造

7.2.2項で触れたとおり、ユーザプログラムの目的コードは個々の手続/関数の本体部ごとの独立した目的コードから形成される。図7.3にその様子を示す。同図中 Extended Address Field(以後 E.A. Field と略記する)は、同図の注釈に記したとおり、当該手続/関数本体の目的コードの長さが、1ベースレジスタのみでは不足する(4096バイト以上)ときのみ、拡張ベースレジスタを用意することを示すフィールドである。ただし、ここでいう目的コードの長さには、ベクトル定数データ領域は含まない。すなわち、ベクトル定数データへのアクセスは、各ベクトル定数データの先頭を指すベクトル定数データ番地表を使用した間接的なアクセスとしている。この理由は、一重化に必要な基本ベクトル等でベクトル定数データの総量が大きくなっても、そのためにそれら全体を含むユーザプログラムの目的コード用に拡張ベースレジスタを多数用意することは、レジスタ割り付けの観点から望ましくない

表7.3 テキスト型ファイルのファイルブロックの構成フィールド[31]~[35]

フィールド名	意味/機能
BLKDFCB	ファイルがオープンされる前、標準入力ファイル INPUT のときには 0、それ以外のファイルのときには論理レコード長。オープンされた後は DCB (Data Control Block) の先頭番地。
BLKDDN	ファイルがオープンされる前のみ使用、その後は以下のフィールドとして使用される。外部ファイル(ユーザプログラム中で、プログラム名に続けて宣言されたもの)のとき、そのファイルの名前(DDNAME となるので8文字以内)、それ以外のファイルのときには 0。
BLKLRECL	ファイルの論理レコード長。
BLKSTAT	ファイルの状態をビットパターンで保持。
BLKPC	ファイルバッファ中の最後に入出力を行った次の番地を指すカレントポインタ。
BLKPE	ファイルバッファの最後の番地を指すエンドポインタ。
BLKPS	ファイルバッファの先頭の番地を指すスタートポインタ。

表7.4 ノンテキスト型ファイルのファイルブロックの構成フィールド[31]~[35]

フィールド名	意味/機能
BLKDFCB	ファイルがオープンされる前、標準入力ファイル INPUT のときには 0、それ以外のファイルのときには論理レコード長。オープンされた後は DCB (Data Control Block) の先頭番地。
BLKDDN	ファイルがオープンされる前のみ使用、その後は以下のフィールドとして使用される。外部ファイル(ユーザプログラム中で、プログラム名に続けて宣言されたもの)のとき、そのファイルの名前(DDNAME となるので8文字以内)、それ以外のファイルのときには 0。
BLKLRECL	ファイルの論理レコード長。
BLKSTAT	ファイルの状態をビットパターンで保持。
BLKNTB	1レコードを格納するためのファイルバッファ。

ためである。拡張ベースレジスタを多数用意するために、使用可能なレジスタ数を減少させ、ひいてはレジスタ待避/復旧を頻繁に行う実行効率の悪い目的コードを生成せざるを得なくなることを考慮すると、ベクトル定数データへの間接的なアクセスのオーバーヘッドは、無視できる程度に小さい。以上の考察から、現バージョンではベクトル定数データ番地表を用意する図7.3の構造としている。そのため、リニ

図7.3 ユーザプログラムの目的コードの構造

ア・リストの定数表(4.3.1項参照)を1度走査し、まずスカラ定数を生成する。同時に生成した定数は、定数表から除く。再度定数表のリストを走査し、ベクトル定数データ番地表を生成する。最後にベクトル定数データを生成する。なお、このベクトル定数データ番地表は、現在当該手続/関数本体の目的コードの先頭からの相対番地(ディスプレースメント)を格納している。相対番地であるが、IBM 360/370 アーキテクチャのインデックス・レジスタの機構を利用することにより、絶対番地が格納されている場合と同数の命令でベクトル定数データへ間接アクセスできる。

図7.3に示したとおり、ユーザプログラムの個々の手続/関数の本体部ごとに独立した目的コードを相互に呼び出し可能とするため、スタック領域のユーザ手続/関数番地表(図7.1も参照)を使用する。そのユーザ手続/関数呼び出しの制御の流れを図7.4に示す。また、このとき利用されるシステムコンスタント・エリアのコードを図7.5に示す。図7.4からわかるとおり、ユーザ手続/関数呼び出しの目的コードでは、呼び出す手続/関数を識別する情報として、ユーザ手続/関数番地表中の呼び出す手続/関数の先頭番地(絶対番地)を保持するエントリまでの相対番地(ディスプレースメント) d_i が出力される。このユーザ手続/関数番地表のエントリ順は、ソースプログラムにおいてユーザ手続/関数が宣言されている順序に一致させているため、相対番地 d_i は、コンパイル時に呼び出したい手続/関数が何番目に宣言されたものか(表4.2識別子表のPFCNTフィールド参照)を調べるだけで簡単に求めることができる。そしてユーザ手続/関数番地表の各エントリの番地情報は、以下のように生成される。まず、あるユーザ手続/関数の目的コードがすべて生成された時点で、次のユーザ手続/関数の目的コードの先頭番地が確定するので、その番地情報を記録しておく。この番地情報は、最初のユーザ手続/関数、すなわちをユーザメインルーチンの目的コードの先頭番地を0とした相対番地である。次に7.1節で触れたとおり、すべてのユーザ手続/関数の目的コードの生成が完了した後、これらの各ユーザ手続/関数の目的コードの先頭番地に関する情報を、リンカーが絶対番地にリロケートできるような形式で出力する。そして実行時に、リンカーにより絶対番地にリロケートされた番地情報を、表7.1にあげた実行時ライブラリの親モジュールの一つであるPASINITが、同じく実行時に確保されたスタック領域(作業領域)のユーザ手続/関数番地表のための領域に記入する。こうすることにより、ユーザ手続/関数呼び出しの目的コードの生成が簡単化され、しかも呼び出したいユーザ手続/関数の目的コードの先頭番地が確定していない場合(例えば特に1パスでコンパイルする際、FORWARD宣言に

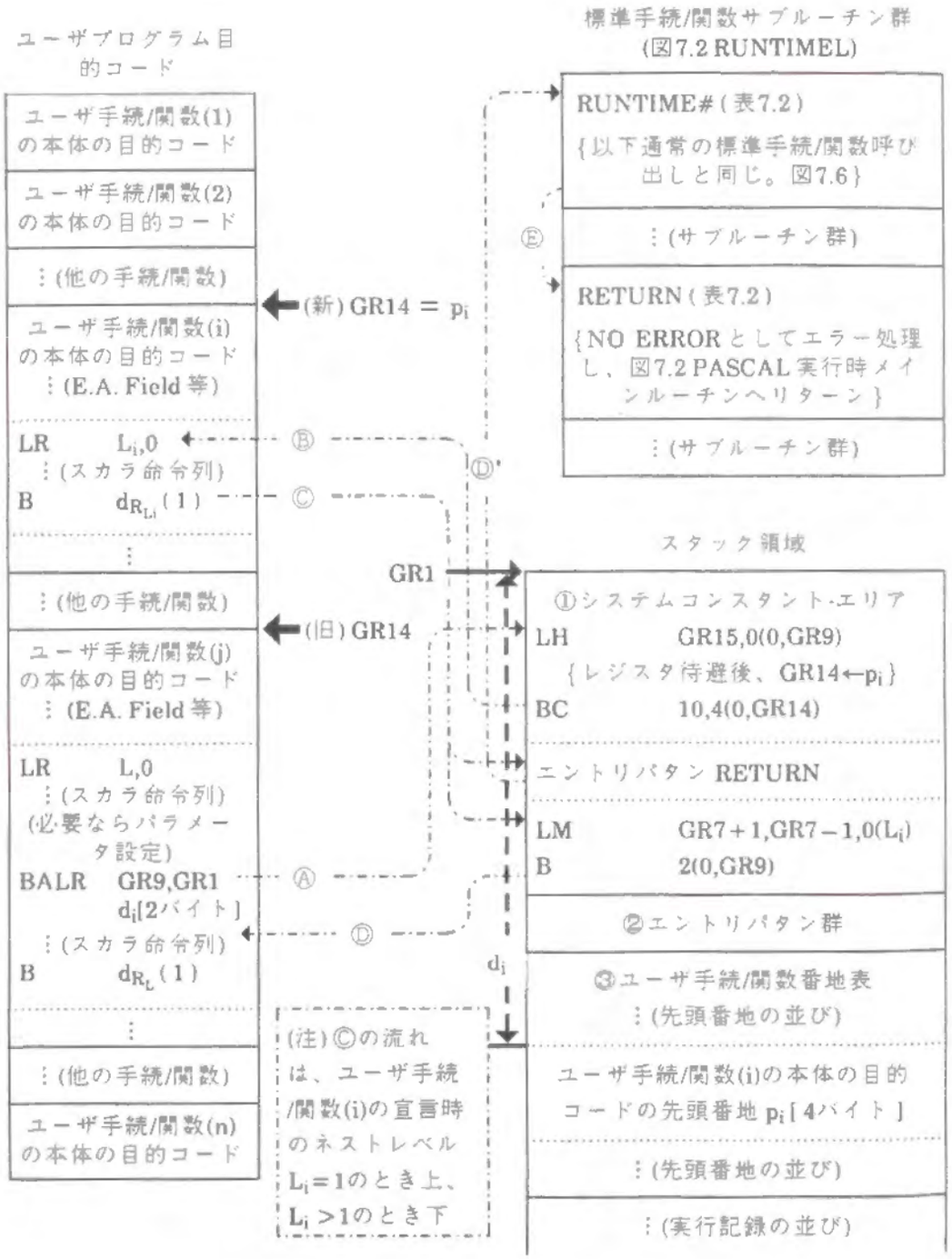


図7.4 ユーザ手続/関数呼び出しの制御の流れ

(注) この図では汎用レジスタ GR_i を R_i と略記
SYSCON はシステムコンスタント・エリアの長さの定数値

```
STACK   DS      0D
        LH      R15,0(0,R9)         図7.4の相対番地 d_i 取得
        STM     R7+1,R7-1,0(R8)     スタックトップへレジスタ待避
        L       R14,SYSCON+*-*(R15,R1)
                                    R14に呼び出す手続きの先頭番地
        LR      R0,R8               スタックトップ待避
        A       R8,0(0,R14)         呼び出す手続きの実行記録の領域をスタック上に確保してみて、オーバーフローしない(E.A. FIELDも0)なら呼び出す手続きへ。
        CLR     R7,R8               E.A. FIELDを0とし、再度オーバーフローチェックし、オーバーフローならエラー処理部へジャンプ。
        BC      10,4(0,R14)
        LA      R8,0(0,R8)
        CLR     R7,R8
        BC      4,STK3
STK1    SR      R15,R15
        IC      R15,0(0,R14)        E.A. FIELD取り出し
        LA      R6,14               LR R(L),R0を修飾し、LR R(L),R14として実行。R(L)にも呼び出す手続きの先頭番地。
        EX      R6,4(0,R14)
        BALR    R9,0
        B       *(R15)              E.A. FIELD(4*L)分だけジャンプ
        LA      R2,4082(R6,R1)      R2=(R1)+4096
        LA      R3,4082(R6,R2)      R3=(R2)+4096
        LA      R4,4082(R6,R3)      R4=(R3)+4096
        LA      R5,4082(R6,R4)      R5=(R4)+4096
STK2    LA      R6,4082(R6,R5)      R6=(R5)+4096
        B       4(0,R14)            呼び出す手続きへジャンプ
                                    R(L+1)~R6の必要な分だけ拡張ベースレジスタをセット。
        ⋮(オーバーフローの処理:エラー処理部へジャンプ)
        ⋮(エントリパターン RETURN, 図7.4参照)
        LM      R7+1,R7-1,0(R2)     レジスタ復旧
        B       2(0,R9)             レベル2の手続/関数から復帰
        LM      R7+1,R7-1,0(R3)     レジスタ復旧
        B       2(0,R9)             レベル3の手続/関数から復帰
        LM      R7+1,R7-1,0(R4)     レジスタ復旧
        B       2(0,R9)             レベル4の手続/関数から復帰
        LM      R7+1,R7-1,0(R5)     レジスタ復旧
        B       2(0,R9)             レベル5の手続/関数から復帰
        LM      R7+1,R7-1,0(R6)     レジスタ復旧
        B       2(0,R9)             レベル6の手続/関数から復帰
        ⋮(定数領域)                 (これらの復帰は図7.4参照)
(以下エントリパターン群, 図7.1参照)
```

図7.5 システムコンスタント・エリア(図7.1参照)のコード

より、手続/関数の名前だけが宣言され、その本体がコンパイルされていない場合)にも呼び出しの目的コードの生成が可能となる。

図7.4からわかるとおり、ユーザ手続/関数呼び出しでは、必ずシステムコンスタント・エリアを経由するようにすることにより、図7.5に記した①レジスタ待避/復旧、②呼び出す手続/関数のための実行記録のスタック上での確保/解放(解放はレジスタ復旧により、常にスタックトップを指すGR8の値が、呼び出し前の状態に戻ることにより行われる)、③確保の際のスタック・オーバーフローのチェック、④呼び出す手続/関数のE.A. Fieldに何らかの値が設定され当該手続/関数の目的コードのために拡張ベースレジスタが必要となる場合には必要な数だけレジスタに値を設定するという定形的な処理をサブルーチン化しまとめている。なお、この図7.5から明らかなとおり、現在は拡張ベースレジスタとしてはGR(L+1)からGR6までが使用できる。ここに、Lは当該手続/関数の静的な宣言のネストの深さである。この静的な宣言のネストの深さは6までとし、(6−L+1)×4096バイトまでの目的コードのサイズが許される。ただし、7.2.4項で述べたとおりベクトル定数データは含まないサイズである。

以上述べてきたとおりユーザ手続/関数呼び出しでは、システムコンスタント・エリアならびにユーザ手続/関数番地表を必ずアクセスする。しかも、ユーザ手続/関数呼び出しは、どのユーザ手続/関数内からも可能とする必要がある。従って、システムコンスタント・エリアならびにユーザ手続/関数番地表は、スタック領域の先頭部分に配置され、常にGR1がスタック領域の先頭を指し示すことにより、実行時に常にアクセス可能となっている。ただし、現バージョンでは簡単のためユーザ手続/関数番地表にエントリされるユーザ手続/関数の数は256個までとしている。

図7.6は標準手続/関数呼び出しの制御の流れを示したものである。また、図7.6にも記したこのとき利用されるサブルーチンRUNTIME#のコードの詳細を図7.7に示す。図7.6中のエントリパターンのコードで、STKRTは標準手続/関数サブルーチン群(のRUNTIME#)の先頭番地(絶対番地)を保持する番地定数のラベルである。従って、このエントリパターンのコードで、スタック領域の先頭を指していたGR1は、一時的に標準手続/関数サブルーチン群の共通の入口であるRUNTIME#への分岐に使用され、GR1には呼び出したい標準手続/関数サブルーチンのコードのRUNTIME#からの相対番地の情報が保存される。そして、標準手続/関数のサブルーチンからリターンする際、図7.7のRUNTIME2のラベルを有するコードによ

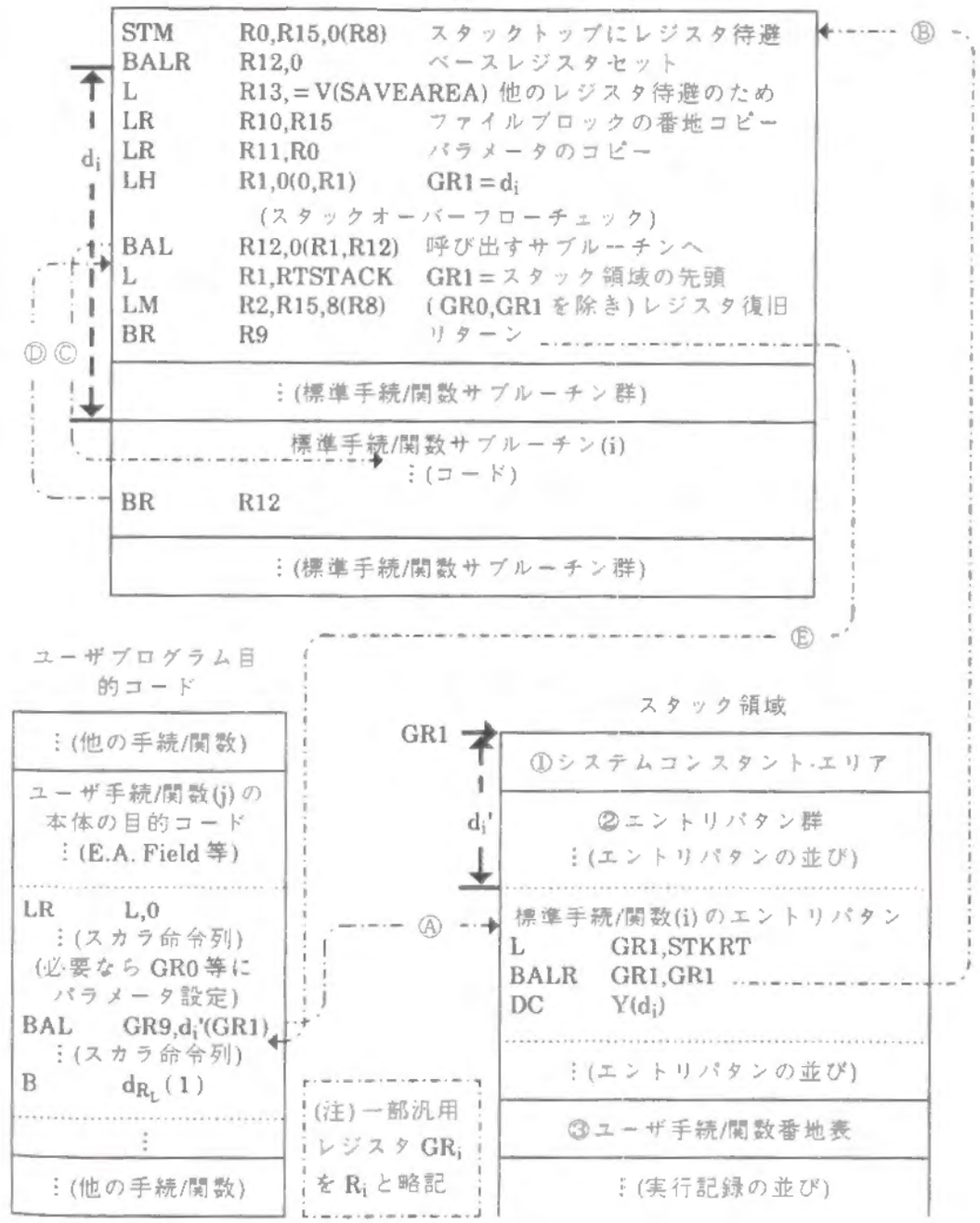


図7.6 標準手続/関数呼び出しの制御の流れ

り、GR1は呼出し前のスタック領域の先頭を指す。すなわち、この部分のコードに現れるRTSTACKのラベルを持つ番地定数領域には、実行時の最初にスタック領域(作業領域)が確保された時点で、その先頭番地(絶対番地)の情報が保存される。

図7.6および図7.7からわかるとおり、ファイル入出力関連の標準手続/関数呼出しの目的コードとしては、処理対象となるファイルのファイルブロック(表7.3および表7.4参照)をGR15が指し示すようにして、ファイルブロックを実パラメータとして引き渡す。また、同様に標準手続/関数呼出しにおけるその他の実引数(値そのもの、あるいは値を保持するレジスタ番号の場合もある)は、すべてGR0を通じて受け渡される。

図7.6からわかるとおり、標準手続/関数サブルーチン群の共通の入口であるサブルーチンRUNTIME#は、ユーザ手続/関数呼び出しの際のシステムコンスタント・エリア部のコードと同様の機能を果たす。すなわち、レジスタ待避/復旧、汎用レジスタ待避領域のスタック上での確保/解放、確保の際のスタック・オーバーフローのチェックという定形的な処理をサブルーチン化しまとめている。

```
RUNTIME#   CSECT
           STM   R0,R15,0(R8)          スタックトップにレジスタ待避
           BALR  R12,0                 ベースレジスタセット
           L     R13,=V(SAVEAREA)      他のレジスタ待避のため
           LR    R10,R15               ファイルブロックの番地コピー
           LR    R11,R0                パラメータのコピー
           LH    R1,0(0,R1)            GR1=d_i
           LA    R6,4*16(0,R8)         新スタックトップ
           CLR   R7,R6                 スタックオーバーフロー?
           BC    10,RUNTIME1           そうでなければジャンプ
           LA    R1,ERROR4-RUNTIME#    オーバーフローエラー
RUNTIME1   BAL   R12,0(R1,R12)         呼び出すサブルーチンへ
RUNTIME2   L     R1,RTSTACK            GR1=スタックの先頭
           LM    R2,R15,8(R8)          (GR0,GR1を除き)レジスタ復旧
RUNTIME3   BR    R9                    リターン
```

(注) この図では汎用レジスタGRiは一部Riと略記。
SAVEAREAは浮動小数点レジスタ等を待避する領域。待避が必要なサブルーチンが個別に待避するコードを有する。

図7.7 標準手続/関数呼び出しに使用されるサブルーチンRUNTIME#

7.3 ベクトル処理命令生成

ベクトル処理命令の生成は、以下の手順で行われる。

(1) ベクトル・ユニットの各レジスタ割り付け
(2) 主記憶参照の逐次化用中間コード生成
(3) 中間項の配列化
(4) ベクトル・ユニットのセットアップ用中間コード生成
(5) ベクトル命令の目的コード・ファイルへの書き出し

このうち、(3)は7.1節において述べたとおり、(1)のベクトル・ユニットの各レジスタ割り付けを行った際にレジスタが不足し、一連のベクトル型中間コード列が複数回のベクトル・ユニットの起動(EXVP命令による)に分割せざるを得なかった場合に、ソフトウェアによるループ制御を行うべく、論理的なベクトル・ループを二重化する中間コード表現に変形させる処理を行う。(5)は機械的にバイナリ表現でファイルへの書き出す処理のみである。従って、以下では(1)、(2)、(4)について述べる。

7.3.1 ベクトル・ユニットのレジスタ割り付け

ベクトル・ユニットのレジスタ割り付け(register assignment)[36],[37]は、図4.15に示したEXVPノードとTVPノードとでリンクされた一連のベクトル型中間コード列ごとに行う。その理由は以下のとおりである。まず、他のスカラ型中間コードはベクトル・ユニットで処理されないので考慮する必要がないことはいうまでもない。また、現在のターゲット・マシンであるHITAC S-820では、同一のベクトル長のベクトル命令列をまとめて1度にベクトル・ユニットで処理する。そして、ベクトル・ユニット起動後はベクトル長を保持するレジスタ(2.2節VLR)をはじめとするベクトル・ユニットのすべてのレジスタの内容を変更できないアーキテクチャとなっている。しかも、ベクトル・ユニットの処理が終了した後も、スカラ・レジスタを除き、レジスタの内容を参照することができない[6]。つまり、ベクトル・ユニットを起動するということは、現在のターゲット・マシンであるHITAC S-820では、ほとんど別の計算機に処理を依頼することに等しい。そのために、第6章で述べたベクトル化では、同一のループでベクトル化されるベクトル型中間コードのみをEXVPノードとTVPノード

とでリンクすることとした。そして、異なるループ構造間で受け渡される中間項(スカラ・データを除く)は、すべて配列化しメモリを経由して受け渡すよう中間コードで表現している。従って、EXVPノードとTVPノードとでリンクされた一連のベクトル型中間コード列は、個々に上記のように独立したベクトル・ユニット起動に相当するものであるので、それぞれ他とは独立に(重複して)ベクトル・ユニットのレジスタを割り付けてよい。

以上述べてきたとおり、ベクトル・ユニットのレジスタ割り付けは、EXVPノードとTVPノードとでリンクされた一連のベクトル型中間コード列ごとに行うので、いわゆる大域的なレジスタ割当て(global register allocation)[36][37]は考慮する必要はない。すなわち、局所的なレジスタ割当てを考えるだけでよい。EXVPノードとTVPノードとでリンクされた一連のベクトル型中間コード列は、制御の流れが1本の一種の基本ブロックであり、複数の演算木から構成されるだけである。しかも、現在のターゲット・マシンであるHITAC S-820では、ベクトル演算はすべてベクトル・レジスタ/スカラ・レジスタ間でのみ行える。すなわち、ベクトル型中間コードが定義/引用する全中間項は、レジスタ上に置いておかなければならない。つまり、どの中間項をレジスタ上に置くかを選択するレジスタ割当ての問題は、今の場合には考える必要がない。

どの中間項をどのレジスタ上に置くかを決定するレジスタ割り付けには、

(A) 並列処理優先割り付け

(B) レジスタ優先割り付け

の2種類の戦略がある。

前者は、その名のとおり演算木の並列実行可能な節(1演算に相当)に、使用可能な、すなわち引用される中間項を保持していないレジスタを順次割り付けるものである。そのため、当然演算木の並列実行可能性は高くなるが、有限個のレジスタでは、レジスタ数が不足し割り付けられなくなる可能性が高い。しかも、レジスタが不足したため7.3節の始めに述べたように当該ベクトル・ユニットの起動を複数回に分割したとすると、そのとき演算木の葉に近い部分の演算ばかりがレジスタを有し実行されているので、それらの部分木の根の演算の結果すべてを配列化し、引き渡すこととなる。つまり、直観的には(A)並列処理優先割り付けは、複数の演算木に対し、葉から幅優先で割り付けることになるので、部分的にしか割り付けられなかった場合には、多くの枝が刈り取られる。刈り取られた部分木の根の演算の結果を配

列化する必要があるので、多くの中間項が配列化されることになる。

これに対し後者の(B)レジスタ優先割り付けは、ある部分木ごとにレジスタを順次割り付けようとするもので、複数の演算木に対し、深さ優先で割り付けることになる。その結果一時に保持していなければならない中間項の数が、すなわち必要となるレジスタ数が最少となる。従って、たとえレジスタが不足したため当該ベクトル・ユニットの起動を複数回に分割するとしても、そのとき配列化される中間項の数は最少ですむ。

一般にレジスタ割当てならびにレジスタ割り付けを最適に行うことは、NP完全であることが知られている[37]。そこで、このベクトル・ユニットのレジスタ割り付けについては、最適なアルゴリズムを実現することはあきらめ、以下のように簡単に処理することとした。

[ベクトル・ユニットのレジスタ割り付け概略アルゴリズム]

(1) 与えられた一連のベクトル型中間コード列を走査し、各中間コードごとに新たに必要となるベクトル・ユニットのレジスタの種別(表7.5参照)の集合を求める。

(2) レジスタ割り付けが成功した中間コードのリストLを空とし、以下のレジスタ割り付けアルゴリズムを再帰的に適用する。

(3) 演算木の林の中で、まず上記(A)の並列処理優先割り付けの考え方で、レジスタがまだ割り付けられていない最初の中間コードを探索する。なければ終了。

(4) 見つかった中間コードC1に新たに必要となる(複数個の)レジスタを順次割り付けてみる。

(5) すべて割り付けられたなら、中間コードC1をリストLの末尾に加える。中間コードC1が引用する中間項すべてについて、リストL中の中間コードからのみ引用され、もはや引用されることがないか調べる。もしそうなら、その中間項を保持するスカラ・レジスタ/ベクトル・レジスタを解放する。再帰的に(3)へ戻る。

(6) レジスタが不足した場合には、当該中間コードに割り付けられたレジスタを解放し、割り付けが失敗したとして最初の中間コードの割り付けの状態までリターンする。その際各中間コードに割り付けられたレジスタを解放するとともに、リストLから各中間コードを取り除いていく。

表7.5 ベクトル・ユニットのレジスタの種別

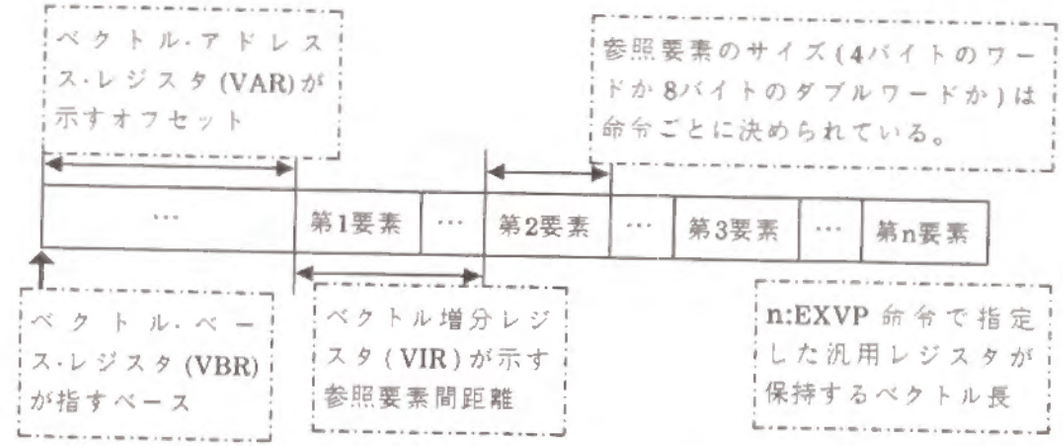
種別	意味/機能
SRTYPE	スカラ・レジスタ(スカラ・データを保持)1個。
VRTYPE	ベクトル・レジスタ(ベクトル・データを保持)1個。
VMRTYPE	ベクトル・マスク・レジスタ(マスク・ベクトルを保持)1個。
SRSRTYPE	連続する番号を持つ2個のスカラ・レジスタ1組。例えば最大値を検索するベクトル命令で最大値とそのインデックス値を保持する場合のレジスタの指定。
VRVRTYPE	連続する番号を持つ2個のベクトル・レジスタ1組。例えばS-810のベクトル除算において機械語では複数命令に展開され、補正値と結果を保持する場合のレジスタの指定。
SRVRTYPE	同一番号を持つスカラ・レジスタとベクトル・レジスタ各1。例えば回帰的なベクトルマクロ命令で初期値と展開された結果のベクトルを保持する場合のレジスタの指定。
VBRTYPE	単一のベクトル・ベース・レジスタ(2.2節および図7.8参照)1個。
VARTYPE	単一のベクトル・アドレス・レジスタ(同上)1個。
VIRTYPE	単一のベクトル増分レジスタ(同上)1個。
VARVIRTYPE	同一番号を持つベクトル・アドレス・レジスタとベクトル増分レジスタ(同上)各1。

(7) 再帰的に呼び出した最初の親側は、上記(B)のレジスタ優先割り付けの考え方で、レジスタがまだ割り付けられていない最初の中間コードを探索する。なければ終了。その中間コードC2が次に割り付けられるか調べる。すなわち、当該中間コードC2が引用する中間項すべてについて、それぞれの中間項を定義する中間コードが既にレジスタを割り付けられリストL中に存在するかどうかを調べる。もし、中間コードC2が次に割り付けられれば、次へそうでなければ再帰的に当該演算木の中間コードC2を根とする部分木の兄の方向へ、次に割り付けられる最初の中間コードを探索する。見つかった中間コードを新たに中間コードC2とする。

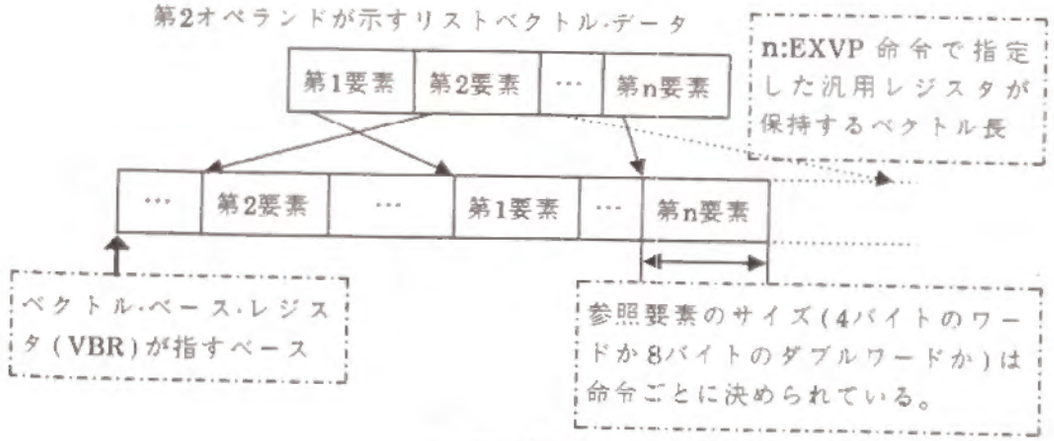
(8) 見つかった中間コードC2に新たに必要となる(複数個の)レジスタを順次割り付けてみる。

(9) すべて割り付けられたなら、中間コードC2をリストLの末尾に加える。中間コードC2が引用する中間項すべてについて、リストL中の中間コードからのみ引用され、もはや引用されることがないか調べる。もしそうなら、その中間項を保持するスカラ・レジスタ/ベクトル・レジスタを解放する。再帰的に(7)へ戻る。

(10) レジスタが不足した場合には、当該中間コードに割り付けられたレジスタを解放し、割り付けが失敗したとして再帰呼び出しの最初までリターンする。



(a) 等間隔型、収集/拡散型



(b) 間接参照型

図7.8 ベクトルロード/ストア命令による主記憶参照[6]

(11) 複数回のベクトル･ユニットの起動となることを示すため、リストLの末尾にVEND中間コードを付加し、当該ベクトル･ユニットの起動において実行できるベクトル型中間コードの終わりであることを記録する。全レジスタを解放し(3)に戻る。 □

この最初のステップ(1)において、表7.5の類別を行っている。この種別のなかでベクトルロード/ストア命令による主記憶参照におけるベクトル･ユニットの各種のレジスタの果たす機能[6]を図7.8に示す。同図から、図中のレジスタ群と緊密な対応関係を持つように、図4.17に示すベクトルロード/ストア中間コードに各フィールド群を持たせた設計となっていることがわかる。なお、HITAC S-820では、図7.8(a)の等間隔型および収集/拡散型のベクトルロード/ストア命令では、そのオペランドでベクトル･アドレス･レジスタおよびベクトル増分レジスタは同一の番号を有するものどうしを組として指示する。この組み合せたレジスタ割り付けを示すのが、表7.5のVARVIRTYPEである。図7.8に示すベクトルロード/ストア命令に必要な各種レジスタも、可能な限り複数の命令群で共用するように割り付けている。例えば、同図中のベクトル･ベース･レジスタは、同一の配列を参照するベクトルロード/ストア命令で共用できる。すなわち、ベクトルロード/ストア中間コードのVCTROBJTOPフィールド(図4.17参照)が同じである複数の中間コード間で共用できる。ベクトル･アドレス･レジスタおよびベクトル増分レジスタは、ベクトルロード/ストア中間コードのVCTROFSTフィールドおよびVCTRDISTフィールド(ともに図4.17参照)がそれぞれ共通の中間項を引用する複数の中間コード間で共用できる。従ってこれらのレジスタに関しては、新たにベクトルロード/ストア中間コードに対しレジスタ割り付けを行う際、既にレジスタ割り付けが完了しているベクトルロード/ストア中間コードの中で上記の条件を満たし共用可能なものをまず探索することとしている。これに対し、これら以外のベクトル･レジスタ、スカラ･レジスタ、マスク･レジスタについては、中間項番号ごとに独立に割り付ける。

上記割り付けアルゴリズムの最後のステップ(11)が前記(B)のレジスタ優先割り付けの考え方でもレジスタが不足する場合の処理である。7.3節の冒頭で述べたとおり、当該ベクトル型中間コード列は複数回のベクトル･ユニットの起動で実行され、そのためソフトウェアによるループ制御が行われる。このとき上記のアルゴリズムのステップ(11)に記したとおりVEND中間コードがリストLに付加される。また、

与えられたベクトル型中間コードが並んでいる順序と、レジスタが割り付けられリストLに付加される順序とは一般には異なる。当然リストL(以下レジスタ割り付け順リストと記す)に示される順序でベクトル命令を生成する必要がある。従って、以降の処理では中間コード表現の本来のリニアリストではなく、上記のアルゴリズムで作られた当該ベクトル型中間コードのレジスタ割り付け順リストLにより、ベクトル型中間コード列を走査する。このレジスタ割り付け順リストLを構成するノードには、対応する中間コードへのポインタのフィールドとともに割り付けられたレジスタ番号をも合わせて記録しておく。

なお、上記のアルゴリズムにおいては、まず与えられたベクトル型中間コード列中のすべての中間項の定義/引用関係から演算木の林を物理的に(少なくとも論理的に)構築するのが効率的である。この演算木の林は、厳密には閉路のない有向グラフ(Directed Acyclic Graph、通常DAGと略記される)[36],[37]に属するいわゆる計算DAG(Computation DAG)[36],[37]である。計算DAGは、葉に変数等の識別子を持ち、その他の節に種々の演算子を持ち、各葉が示す変数等の値および各節が示す演算子の演算結果のデータの流れを有向辺とするDAGである。ところが、D行列が表現するデータ依存関係がこの計算DAGの有向辺そのものであり、ベクトル型中間コードに対応するD行列の行列構成セルがこの計算DAGの葉および節に相当する点に着目すれば、与えられたベクトル型中間コード列に対応するD行列の部分行列を計算DAGと見なせる。従って、V-Pascalの現バージョンでは、ここでもD行列を利用することにより、計算DAGを構築することなく、効率的に上記のアルゴリズムを実現することができている。

7.3.2 主記憶参照の逐次化

現在のベクトル計算機では、第2章で述べたとおり通常並列実行可能な演算パイプラインを複数本有するアーキテクチャを採用している。そのため、必ずしも生成されたベクトル命令が生成された順序で実行されるとは限らない。V-Pascalの現バージョンのターゲット･マシンであるHITAC S-820(S-810も同様)では、同一のレジスタに対しアクセスするベクトル命令間の実行順序は、それらの命令が目的コード中で並んでいる順序関係を保存する。しかし、ある時点で実行中のベクトル命令とレジスタが競合しない、すなわち同一のレジスタに対しアクセスしないベクトル命令は、そのベクトル命令が使用する演算パイプラインで空いているものがあれば、並列に実行される。しかも、2.2節で述べたとおりチェイニング機構をも有しているた

め、レジスタが競合する場合にも先のベクトル命令の実行と後のベクトル命令の実行とが一部オーバーラップする[6]。

このような種々の並列機構のため、同一の主記憶の番地に対し参照を行うベクトルロード/ストア命令間で実行を逐次化する必要がある場合が生じる。例えば、配列Aの一部に何らかの演算結果を保存するベクトルストア命令と、その一部書き換えられた後の配列Aの値を読み出してくるベクトルロード命令とでは、これら2命令間でレジスタが競合しない場合には、必ず先のベクトルストア命令の実行が完全に終了するのを待って、次のベクトルロード命令が実行されねばならない。このような主記憶参照の逐次化を行うため、HITAC S-820(S-810も同様)では、VWAC(Wait Until MS Access Complete)命令が用意されている[6]。この命令は、この命令が発効された時点で終了していないすべてのベクトルロード/ストア命令について、それらの実行が完了するのを待つ。

従って、7.3.1項のアルゴリズムで作られたレジスタ割り付けの順序を示すリストLの必要な位置にVWAC命令に対応するVWAC中間コードを挿入する。挿入すべき位置は以下のアルゴリズムにより求めることができる。

[主記憶参照の逐次化のための中間コード付加アルゴリズム]

(1) リストL中の各ベクトル中間コードに対し、そのリストLが示す順序で番号づけを行う。

(2) 同一の(Aliasであるものも含む)変数あるいは配列要素を参照するベクトルロード/ストア中間コードのすべてのペア(データ依存解析の場合と同様、引用-引用では順序関係を保存する必要がないのベクトルロードどうしは除く)について、次の(3)および(4)の処理を行う。

(3) ペア間になんらかのレジスタ競合が生じうるかどうか調べる。すなわち、先のベクトルロード/ストア中間コードが使用するレジスタと同一のレジスタを使用する中間コードが、ペア間(後のベクトルロード/ストア中間コードを含む。以下同じ)に存在するか否かを調べる。あれば、その見つかった中間コードxとレジスタ競合している中間コードが、中間コードx以降のペア間に存在するか否かを調べる。これを再帰的に繰り返し、レジスタ競合する中間コードのチェインを作ったとき、最終的にペアの後のベクトルロード/ストア中間コードにまで達することができれば、当該ペアはレジスタ競合する。

(4) レジスタ競合しなかった、すなわち逐次化を必要とするペアについてのみ、上記(1)でそれぞれのベクトルロード/ストア中間コードにつけられている順序番号を開区間(i,j)の形にして開区間の集合Sに登録する。

(5) こうして得られた開区間の集合Sをiを主キー、jを2次キーとしてソートし順序集合とする。このときもし、(i,j)ともに等しい要素が複数個あれば、1個を除きすべて集合Sから除去しておく。

(6) 集合Sの最初の要素(i_1, j_1)のj_1の番号を持つベクトルロード/ストア中間コードの直前にVWAC中間コードを挿入する。

(7) 集合Sを先頭から走査し、j_1-1を含む開区間をすべて取り除く。もし、集合Sが空となれば終了。そうでなければ、(6)に戻る。 □

このアルゴリズムのステップ(2)において、同一の変数あるいは配列要素を参照するベクトルロード/ストア中間コードのペアは、D行列から簡単に求めることができる。すなわち、リストLに含まれるベクトル型中間コード列に対応するD行列の部分行列を形成する行列構成セルを先頭から順に走査し、ベクトルロード/ストア中間コードに対応するのであれば、その部分行列の行に含まれるデータ依存を表す非零要素を検索する。データ依存を表す非零要素があれば、それが属する列の行列構成子に対応する中間コードがベクトルロード/ストア中間コードかどうか調べる。そうであれば、ペアが発見されたことになる。そうでない場合は中間項の授受による依存であるので、ペアとしない。ベクトルロード/ストア中間コード以外のベクトル型中間コードでは主記憶参照は起こり得ないので調査する必要はない。

7.3.3 ベクトル・ユニットの起動のためのセットアップ

V-Pascalの現バージョンのターゲット・マシンであるHITAC S-820(S-810も同様)では、7.3.1項で述べたアルゴリズムで割り付けられたベクトル・ユニットのレジスタの一部は、当該ベクトル命令列をEXVP命令で起動する前に、スカラ・ユニットから初期化(セットアップと呼ばれる)しておく必要がある。セットアップが必要なレジスタは、具体的には次の4種類である(これらのレジスタの機能、数等については2.2節を、後の3種については図7.8をも参照)[6]。

(1) スカラ・レジスタ:スカラ・データ保持。SR0~SR31の32本。

(2) ベクトル・ベース・レジスタ:VBR0~VBR30の16本(偶数番号のみ)。

(3) ベクトル・アドレス・レジスタ:VAR0~VAR30の16本(偶数番号のみ)。

(4) ベクトル増分レジスタ : VIR0~VIR30 の16本 (偶数番号のみ)。

ただし、(1) のスカラ-レジスタに関しては、ベクトル-ユニット側で計算された値を保持するために割り付けられているものもある。例えば、最大値/最小値探索の結果等がその例である。もちろん、このようにスカラ-ユニットからベクトル-ユニットへデータを受け渡す必要がない場合には、そのスカラ-レジスタをセットアップする必要はない。つまり、(1) のスカラ-レジスタに関しては、スカラ型中間コードで定義された (スカラ-データの) 中間項をベクトル型中間コードで引用する場合にのみセットアップする。それ以外の (2) から (4) のレジスタに関しては、必ずセットアップする必要がある。

HITAC S-820 (S-810 も同様) では、このセットアップを行うスカラ命令を数種持っている[6]。これらは、(A) 主記憶上のデータをロードする形式、(B) スカラ-ユニットの汎用レジスタ上のデータをロードする形式の2種類に大別できる。ただし、上記の4種類のレジスタのうち、これら (A) および (B) 両方式それぞれの命令をともに備えているのは、(1)、(3)、(4) のレジスタのみである。(2) のベクトル-ベース-レジスタに関しては、(A) の形式の命令しか有していない。そこで、V-Pascal の現バージョンでは、セットアップの処理をレジスタの種類によらず統一的に行うため、上記の4種類すべてのレジスタに対して、(A) の形式の命令でセットアップを行っている。しかも、この (A) の形式の命令は、連続する番号を持つ同種の複数のレジスタ群をまとめて1命令でセットアップすることができる。これに対し、(B) の形式の1命令は、常に1レジスタしかセットアップすることができない。従って、(B) の形式の命令はレジスタ-レジスタの転送であるため1命令の処理時間は (A) の形式の命令よりも少ないが、同種の多くのレジスタをセットアップする場合には (A) の形式の命令でセットアップする方が処理時間のオーバーヘッドが少ないと予想される。しかも、7.3.1 項で述べたとおりハードウェアの持つ各種の並列処理機構を有効に機能させるためには、レジスタ競合が少なくなるように、従って通常多くのレジスタが割り付けられると考えられる。以上の理由からも、V-Pascal の現バージョンでは、(A) の形式の命令でセットアップを行っている (図7.9 参照)。

以上述べたとおり、主記憶を介しスカラ-ユニットからベクトル-ユニットへデータを受け渡すため、図7.9 に示す領域を当該ベクトル型中間コード列を有する手続/関数の実行記録 (7.2.1 項、図7.1 参照) の領域に一時領域として確保する。このセットアップ用領域は、当該ベクトル型中間コード列が複数回の EXVP 命令で起動される場合には、それぞれのセットアップ用に各1個確保する。そして、セットアップ用領域は当該手続/関数本体内の他のベクトル型中間コード列のセットアップにおいて再利用する。

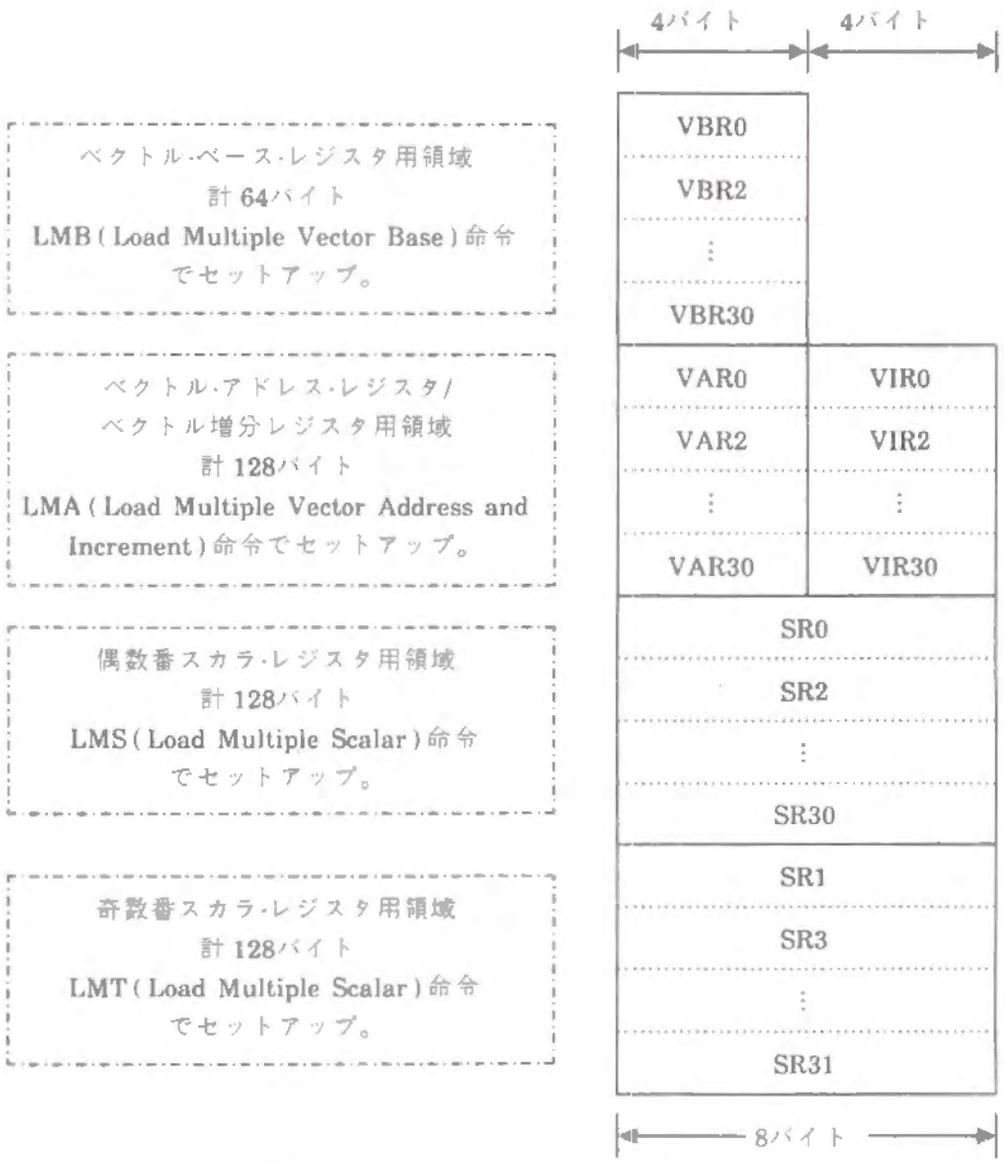


図7.9 セットアップ用データ格納領域

図7.9に示すセットアップ用領域は、現在セットアップされる可能性のあるすべてのレジスタ分用意している。これは、上述したとおり複数回再利用されるそれぞれの場合にセットアップすべきレジスタを調査するコンパイル時のコストが大きいのに対し、それにより削減できる可能性のある領域のサイズは、せいぜい数十バイト程度しかないと見込まれることによる。さらに、このようにレジスタ最大数分を確保することにより、すべてのレジスタのそれぞれの相対番地が常に一定となり、目的コード生成時のアドレス計算も簡単化される。

図7.10にレジスタが不足せず与えられたベクトル型中間コード列を1回のベクトル・ユニットの起動で実行できる場合の目的コードのパターン(スケルトン; skeleton)を示す。この場合には2.2節において述べたとおり、ハードウェアによるループ制御となるため、論理的なベクトル長そのものを汎用レジスタ上にセットし、その汎用レジスタ番号をオペランドとしてEXVP命令を起動する。つまり、ハードウェア・ループ制御ため、生成される目的コードも単純なものとなっている。また、当然セットアップ用領域も1個使用されるだけで、セットアップ用領域へのデータの格納および同領域からのセットアップ用命令(図7.9参照)も1ヵ所にまとめられ1度実行されるだけである。なお、このセットアップ用領域へのデータの格納のための目的コード群では、スカラ・ユニットの特に汎用レジスタを有効に利用する目的から、必要なデータ(中間項)が定義された直後で、直ちにセットアップ用領域の該当するフィールドに格納し、当該中間項がもはや参照されないなら直ちに当該中間項を保持するスカラ・ユニットのレジスタを解放するようにしている。一方ベクトル長を示す中間項は、EXVP命令を実行する時のみレジスタ上に置く。そのため、もしEXVP命令を生成する時点で当該中間項が一時領域に待避されている場合には、先にレジスタを確保しレジスタ上にロードしてくる。EXVP命令生成後、当該中間項がもはや参照されないならば、当該中間項を保持するレジスタを解放する。

セットアップ用領域としてはレジスタ最大数分を確保するが、実際にロードすべきレジスタは一部である場合もある。その場合にはもちろん、図7.10のスケルトンにおいてセットアップ用命令(図7.9参照)1命令で同種の複数のベクトル・ユニットのレジスタに値をロードする際に、レジスタ番号を指定することにより、必要なレジスタのみをセットアップしている。

図7.10中に記したとおり、EXVP命令/TVP命令はそれぞれベクトル・ユニットを起動させることができたか/ベクトル・ユニットの処理が完了したかをテストしその結

果をコンディション・コードとして返す機能を有している[6]。この機能を利用し、結果のコンディション・コードにより、所望の結果となるまでそれぞれの命令を繰り返し、待ち合わせる目的コードを生成している。なお、図7.10からわかるとおり、EXVP命令(とそれへのジャンプ命令)およびTVP命令(とそれへのジャンプ命令)とではさまれたスカラ命令列が、ベクトル・ユニットでの処理と完全に並列に実行される。一方EXVP命令で起動されるベクトル命令列は、図7.10に示すとおりスカラ

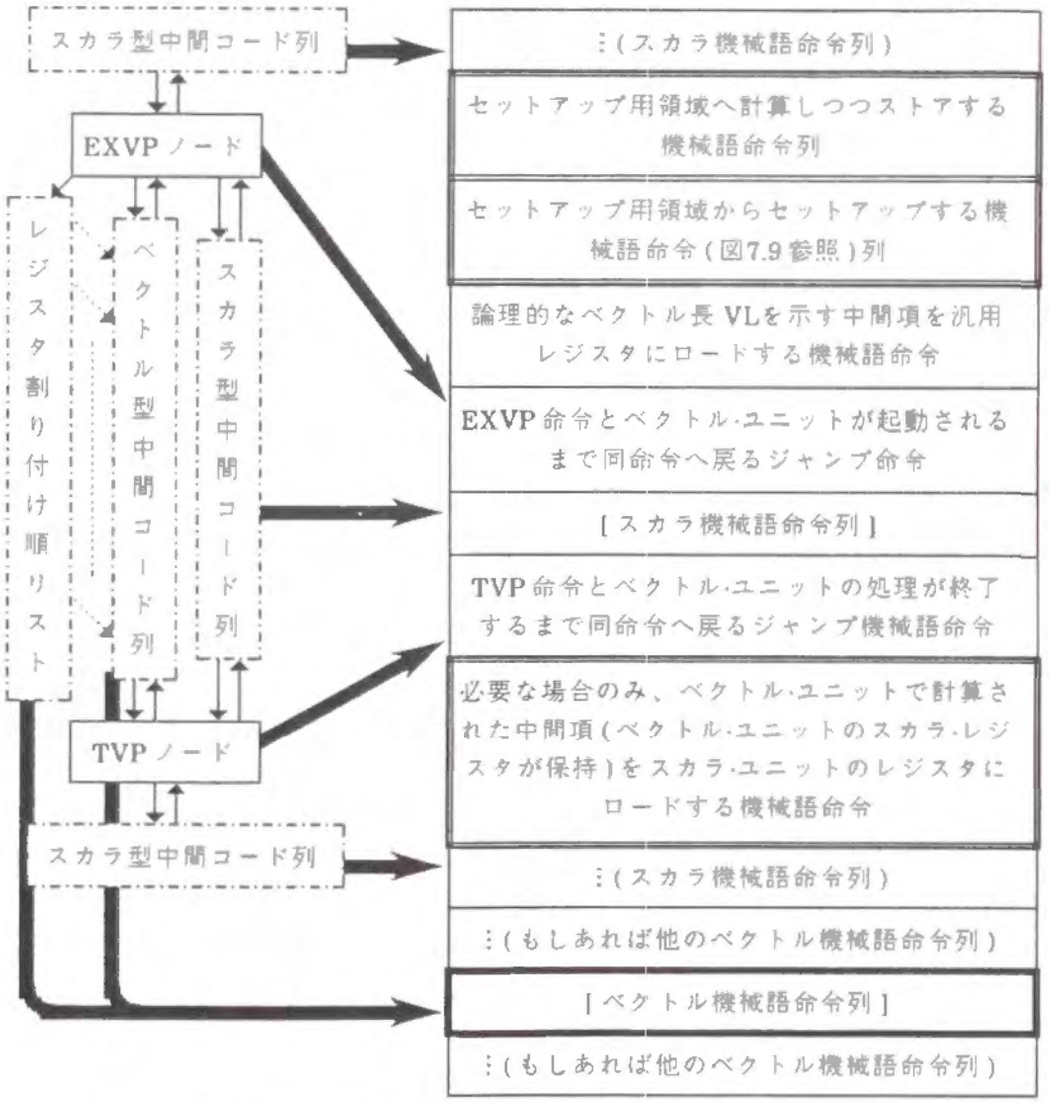


図7.10 1回のEXVP命令によるベクトル・ユニット起動のスケルトン

命令列をすべて生成した後(図7.3参照)、再度各EXVPノードをたどりながら生成される。

セットアップは、スカラ・ユニットからベクトル・ユニットへのデータの受渡しであった。それに対し、逆にベクトル・ユニットからスカラ・ユニットへのデータの受渡しも必要となる。ところが、一般にはその向きに受け渡されるデータは、ベクトル・ユニットで計算されたベクトル・データである。ベクトル・データの場合には、演算結果の中間項を保持するベクトル・レジスタをスカラ・ユニットから参照できないので、主記憶に書きもどすことになる。この処理は第6章で述べた中間項の配列化でありベクトル化処理部がこの書きもどす処理等を既に中間コードで表現してあるので、目的コード生成部では特別に処理する必要はない。マスク・データの場合も同様で、ベクトル・ユニットからスカラ・ユニットへのデータの受渡しは主記憶を介して行うように既に中間コードで表現してある。それに対し、スカラ・データの場合には、このようなベクトル・ユニットからスカラ・ユニットへのデータの受渡しを主記憶を介して行うようには表現されていない。なぜなら、スカラ・データの場合には、通常直接ベクトル・ユニットのスカラ・レジスタから、スカラ・ユニットのレジスタへロードする命令が用意されているためである。V-Pascalのターゲット・マシンである HITAC S-820でも、この種の命令が用意されている。そこで、この種の命令を使って必要なデータをスカラ・ユニットのレジスタへロードすることとした(図7.10参照)。

図7.11はレジスタが不足し与えられたベクトル型中間コード列をk回のベクトル・ユニットの起動で実行させる場合の目的コードのパターン(スケルトン)である。ベクトル型中間コード列がk個に分割されたため、先に述べたとおりソフトウェアによるループ制御[6]を行っている。すなわち、図7.11からわかるとおり、論理的なベクトル長を物理的なベクトル・レジスタの要素数ごとに切りわけ、(L+1)回目的コードで明示的にループ区分処理[6]させている。L回のループ区分処理のベクトル長は、物理的なベクトル・レジスタの要素数分に等しい。目的コードとしてはこの部分は、図7.11の後半部分で実行される。残りの1回のループ区分処理のベクトル長は、論理的なベクトル長を物理的なベクトル・レジスタの要素数で割った余りである。目的コードとしてはこの部分は、図7.11の前半部分で実行される。つまり、(L+1)回のループ区分処理すべてを同一の目的コードを繰り返し処理するのではなく、初回のループ区分処理のみを別の目的コードとして実行させ、その後L回のループ区分処理を実行させる。このように、初回のループ区分処理のみを別の目的コードとして分

(注)・与えられたベクトル命令列がk個に分割されたとする。
・セットアップ部は、主記憶のセットアップ用領域への値のストアならびに同領域からセットアップする命令列からなる(図7.10参照)。
・ベクトル命令列の終了待ち(TVP命令とそれに戻るジャンプ命令)/計算された中間項待避の目的コード群は必要な場合にのみ生成される(図7.10参照)。ただし、ベクトル命令列jの終了待ちの直前にベクトル命令列jのセットアップ用領域の値の更新を行う目的コード群を必ず生成する。2回目からのセットアップ(図中二重線内)はロードのみ。

L=L(論理的なベクトル長VL)/(ベクトル・レジスタ長)」 M=(論理的なベクトル長VL) mod (ベクトル・レジスタ長)
M(最初のベクトル長)を汎用レジスタにロードする命令
分割されたベクトル命令列1のセットアップ
分割されたベクトル命令列1の起動
[スカラ命令列](EXVP/TVP両中間コード間)(†)
分割されたベクトル命令列2のセットアップ
ベクトル命令列1の終了待ち/計算された中間項待避
分割されたベクトル命令列2の起動
分割されたベクトル命令列3のセットアップ
ベクトル命令列2の終了待ち/計算された中間項待避
⋮
分割されたベクトル命令列kのセットアップ
ベクトル命令列(k−1)の終了待ち/計算された中間項待避
分割されたベクトル命令列kの起動
ループ制御回数Lおよび(ベクトル・レジスタ長:2回目以降のベクトル長)を汎用レジスタにロード
ベクトル命令列kの終了待ち/計算された中間項待避
(†)を除く上記と同じベクトル命令列1~kのセットアップ/起動/終了待ち/計算された中間項待避を順に行う命令列
Lを1減じ、0になるまでループさせる命令
ベクトル命令列1~kで計算され、待避された中間項を必要に応じ汎用レジスタにロードする命令

繰り返し回数Lのループ

図7.11 複数回のEXVP命令によるベクトル・ユニット起動のスケルトン

けた理由は、次の2点である。

(1) 1回だけ異なるベクトル長のループ区分処理を行う必要があること。同じく1回だけベクトル・ユニットと本来並列に実行されるスカラ命令列(図7.10参照、図7.11では(†)で示した部分)の処理を行う必要があること。

(2) 2回目以降のループ区分処理に必要なセットアップのデータは、1回目のセットアップのデータの多くをそのまま利用できること。

上記の第2点を実現するためには、k個に分割されたベクトル型中間コード列に対し各1個のセットアップ用領域を独立に用意しておく必要がある。すなわち、セットアップ用領域をk個に分割されたベクトル型中間コード列間で共用できない。

1回のループ区分処理を行うi番目(i=1~k)のベクトル・ユニットの起動は、以下の手順で実行される。

(a) セットアップ

(a-1) セットアップ用領域iへスカラ・ユニットのレジスタからデータをストア

(a-2) セットアップ用領域iからベクトル・ユニットのレジスタへデータをロード

(b) EXVP命令によるベクトル・ユニットのi回目の起動

(c) TVP命令によるi回目のベクトル・ユニットの終了待ち

(d) ベクトル・ユニット内の必要となる一部のスカラ・レジスタの待避

(d)は、図7.10でベクトル・ユニットのレジスタ上にあるスカラ・データの計算結果(中間項)をその後のスカラ処理で参照する場合に、スカラ・ユニットのレジスタへ取り込む命令に相当する。このループ区分処理を行う場合には、次のループ区分処理にデータを引き継ぐため、i+1番目以降のベクトル・ユニットの起動で破壊されてはならないデータのみを主記憶のセットアップ用領域に書きもどす。このように次のループ区分処理に引き継がれるデータは、3.3.2項で述べた回帰型のベクトル・マクロ命令(最大値/最小値探索等)における、当該ループ区分処理までの中間結果である。これらの中間結果は、最後のループ区分処理が終了した後、図7.10と同様最終結果をスカラ・ユニットのレジスタへ取り込む(図7.11のスケルトンの最後の部分)。ただし、次のループ区分処理に引き継ぐべきデータがない場合には、i<kのときの(c)、(d)および図7.11の最後の部分は省略できる。

上記の(b)および(c)は、それぞれ図7.10に示すEXVP命令/TVP命令とそれが設定

したコンディション・コードに応じそれぞれの命令へ条件分岐する命令からなる。

(a-1)は、先述のとおり図7.11の前半部の初回のループ区分処理では、ベクトル・ユニットのスカラ・レジスタ、ベクトル・ベース・レジスタ、ベクトル・アドレス・レジスタ、ベクトル増分レジスタ全種について必要なデータをストアする。それに対し、図7.11の後半部の2回目以降のループ区分処理では、セットアップする各ベクトル・アドレス・レジスタのためのデータを、(前回のループ区分処理のベクトル長)×(当該ベクトル・アドレス・レジスタと同一番号のベクトル増分レジスタに与えられる参照要素の間隔)だけ増して(あるいは減じて)おくだけでよい。(a-2)は各ループ区分処理共通に、図7.10と同様セットアップするレジスタ番号を指定したセットアップ用の命令列で構成される。

先にも触れたとおり、EXVP命令によりベクトル・ユニットを起動した後、スカラ・ユニットはベクトル・ユニットと並列に処理を行うことができる。従って、上記の(a)から(d)のうち、(b)と(c)の間で次のi+1番目のベクトル・ユニットの起動のための(a)のセットアップを行っている(図7.11および図7.12参照)。また、同じく並列にi番目のベクトル・ユニットの起動の次のループ区分処理のため、上述のセットアップ用領域jのセットアップすべきベクトル・アドレス・レジスタのためのデータの更新を行っている(図7.12の(a-1'))。そのため、図7.11の注にも記したとおり、図7.12にも示す2回目以降のループ区分処理では、(a)のセットアップとしては、(a-2)のセットアップ用領域からの値のロードをするだけで、ベクトル・ユニットの起動を行える。その後、やはり次のループ区分処理のため、図7.12の(a-1')を先に実行している。なお、図7.12からわかるとおり、i番目のベクトル・ユニットの起動中に、i+1番目のベクトル・ユニットの起動のための(a)のセットアップを行っている。HITAC S-820のアーキテクチャでは、スカラ・ユニットからセットアップされるベクトル・ユニットのレジスタ群は、実際には2組用意されている。一方はセットアップ用命令で値が書き換えられる、すなわちスカラ・ユニットからアクセスされる(便宜的にこちらを表側のレジスタと呼ぶことにする)のに対し、他方は実際にベクトル・ユニットがその実行中にアクセスする(便宜的にこちらを裏側のレジスタと呼ぶことにする)ものである。そして、表側のレジスタから裏側のレジスタへは、EXVP命令を実行すると全データがコピーされる[6]。このような機構により、i+1番目のベクトル・ユニットの起動のための(a)のセットアップを行っても、それは表側のレジスタの値を変更するだけであるので、裏側のレジスタをアクセスしているi番目のベクトル・ユ

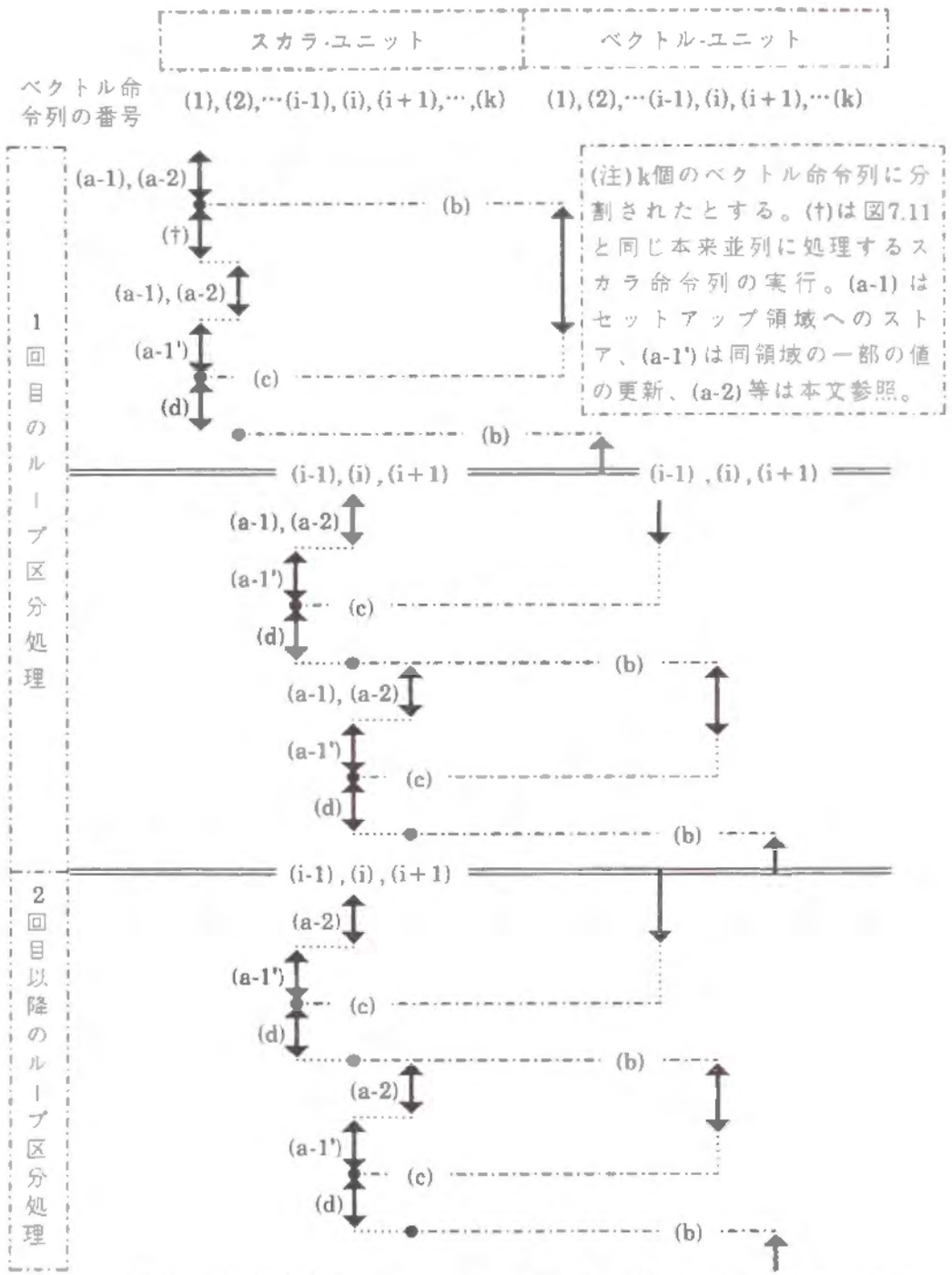


図7.12 複数回の EXVP 命令によるベクトル-ユニット起動のタイミング-チャート

ニットの処理にはなんら影響しない。

このように、セットアップを行わせるために現バージョンでは次の処理を行っている。

[セットアップ用の目的コード生成の概略アルゴリズム]

(1) まず、中間コードをたどり、各 EXVP ノードにリンクされているベクトル型中間コード列ごとにレジスタ割り付け (7.3.1 項参照) を行う。

(2) 与えられたベクトル型中間コード列においてセットアップが必要となるレジスタの種別 (前記の4種のうちのどれであるか) と番号を表にまとめる。同時に、セットアップと逆にベクトル型中間コード列において定義され、後のスカラ型中間コード列において引用される中間項も別の表にまとめる。

(3) その表に基づき、分割されたi番目のベクトル型中間コード列のセットアップに必要な中間項を実行記録のセットアップ用領域へストアする中間コード群を、仮に当該ベクトル型中間コード列の直前に挿入しておく。

(4) トレース2において、実際に目的コードを生成する際、EXVP ノードに対する処理として、以下を行う。

(4-1) リンクされているベクトル型中間コード列が分割されている個数分セットアップ用領域を一時領域として確保。

(4-2) 図7.10 あるいは図7.11 に示したスケルトンどうりに、(3) で作ったストア中間コード群からセットアップ用領域へのストア命令を生成。そのとき可能ならば、中間項を定義する命令にできるだけ近い位置でストア命令を生成する。

(4-3) (2) で作った表を再度参照し、前記の4種のレジスタの種別ごとに、セットアップが必要となるレジスタ番号の最小値/最大値を求め、図7.9 中に示す命令を順次生成する。

(4-4) EXVP 命令を生成する。

(4-5) 最初のみ、図7.11 (†) の目的コードを生成する。

(4-6) 分割されたベクトル型中間コード列の場合には、図7.12 (a-1') の目的コードを生成する。

(4-7) (2) で作った別表を参照し、必要ならば先述の (c) および (d) の目的コードを生成する。1回の起動で実行されるベクトル型中間コード列の場合に

は、直接スカラ･ユニットのレジスタへデータを転送する命令とする(図7.10参照)。

(5) 分割されたベクトル型中間コード列の場合には、図7.11の後半部のように、さらに2回目以降のループ区分処理を行うよう目的コードを生成する。 □

7.4 スカラ処理命令生成

スカラ処理命令の生成に関しては、7.2節で既述したPascal 8000処理系の実行環境に合わせるため、同処理系と同じスケルトン[31]~[35]で目的コードを生成する。従って、ユーザ手続/関数の目的コードを実行中のスカラ･ユニットの汎用レジスタの

表7.6 スカラ･ユニットの汎用レジスタの役割分担

番号	機能
0	(1) 標準手続/関数呼び出しの際の入出力パラメータの保持(図7.7参照)。 (2) スカラ実行されるforループの制御変数と上限値の比較の際に、制御変数の値を保持、また制御変数の値の増減にも使用(図7.13参照)。
1~6	(1) 番号と同一レベルのブロックのディスプレイ(図7.1参照)。 (2) ネスト･レベルLのユーザ手続/関数の目的コードの拡張ベース･レジスタとして(L+1)~6番を使用(図7.5参照)。
7	ヒープ･トップを指す(図7.1参照)。
8	スタック･トップを指す(図7.1参照)。
9	(1) 手続/関数呼び出しの際の復帰番地記憶(図7.4および図7.6参照)。 (2) 実行記録(図7.1参照)の4096バイトを越える部分を参照する際の拡張ベース･レジスタ(図7.13参照)。
10~13	(1) スカラ･データの中間項の保持。 (2) スカラ実行される配列参照の番地計算。
14	ユーザ手続/関数の目的コードのベース･レジスタ(図7.4および図7.13参照)。
15	(1) ユーザ手続/関数呼び出しの際ユーザ手続/関数番地表のエントリの相対番地を保持(図7.5参照)。 (2) 入出力関係の標準手続/関数呼び出しの際、対象となるファイルのファイル･ブロック(表7.3および表7.4参照)の番地を保持(図7.7および図7.13参照)。

機能も同処理系と同じように表7.6に示す固定的な役割分担とした。同表の汎用レジスタの機能を示す具体例を図7.13に示す。

表7.6からわかるとおり、IBM 360/370アーキテクチャの汎用レジスタ16本のうち、中間項を保持し実際に各種の演算を行うためのレジスタとしては、浮動小数点演算用に4本、固定小数点演算および番地計算用にGR10からGR13の4本しかない。しかも、固定小数点の乗除算では、偶数番レジスタと奇数番レジスタとを組み合せて64ビットのレジスタとして使用する。その場合には、(GR10-GR11)あるいは(GR12-GR13)の2組しか割り付けられない。このように、現在採用している固定的な役割分担のもとでは、演算用のレジスタが極度に不足するため、スカラ･ユニットの

```
PROGRAM CODETEST(INPUT,OUTPUT);
 VAR I:INTEGER;
BEGIN
 FOR I := 0 TO 3 DO;
END.
```

(a) ソース･プログラム例1

```
相対番地,コード,  ニーモニック
00    00000A48                    E.A. Fieldと実行記録サイズ(図7.3参照)
04    1810      LR    1,0         実行記録のディスプレイ
06    47000048  BC    0,72(0,0)   72=コードサイズ
0A    41F01A30  LA    15,2608(0,1) INPUTファイルのファイル･ブロック
0E    45901150  BAL   9,336(0,1)  OPENINPUT(表7.2参照)呼び出し
12    17AA      XR    10,10       GR10=0(ループの初期値)
14    50A01A44  ST    10,2628(0,1) Iにストア
18    58001A44  L     0,2628(0,1) Iの値GR0へ
1C    5900E044  C     0,68(0,14)  ループの上限値と比較
20    4720E034  BC    2,52(0,14)  大きければジャンプ
24    58001A44  L     0,2628(0,1) Iの値GR0へ
28    5A00E040  A     0,64(0,14)  増分加算
2C    50001A44  ST    0,2628(0,1) Iにストア
30    47F0E01C  BC    15,28(0,14) 繰り返す
34    41001A30  LA    0,2608(0,1) INPUTファイルのブロック
38    459010D8  BAL   9,216(0,1)  CLOSEEXT(表7.2参照)呼び出し
3C    47F01058  BC    15,88(0,1)  リターン(図7.4参照)
40    00000001                    (ループの増分値:リテラル)
44    00000003                    (ループの上限値:リテラル)
```

(b) (a)の目的コード

図7.13 簡単なforループの場合の生成される目的コード(つづく)

レジスタについても、大域的なレジスタ割当ては行っていない。従って、レジスタ割り付けは、単に未使用のレジスタの探索を行い、もしあれば割り付け、未使用のレジスタがなければ、使用中のレジスタの内容を一時領域に待避させ、そのレジス

```
PROGRAM  CODETEST(INPUT,OUTPUT);
 VAR N:ARRAY(.1..500.)OF INTEGER;{十分大きなスペース}
     I:INTEGER;
BEGIN
 FOR I := 0 TO 3 DO;
END.
```

(c) ソース・プログラム例2

```
相対番地,コード,  ニーモニック
00    00001218                    E.A. Fieldと実行記録サイズ(図7.3参照)
04    1810        LR    1,0       実行記録のディスプレイ
06    47000060    BC    0,96(0,0) 96=コードサイズ
0A    41F01A30    LA    15,2608(0,1) INPUTファイルのファイル・ブロック
0E    45901150    BAL   9,336(0,1) OPENINPUT(表7.2参照)呼び出し
12    17AA        XR    10,10     GR10=0(ループの初期値)
14    5890E05C    L     9,92(0,14) GR9=X'1000'
18    1A91        AR    9,1       GR9はGR1の拡張ベースレジスタ
1A    50A09214    ST    10,532(0,9) Iにストア(GR1はベースとして届かない)
1E    5890E05C    L     9,92(0,14) GR9=X'1000'
22    1A91        AR    9,1       GR9はGR1の拡張ベースレジスタ
24    58009214    L     0,532(0,9) Iの値ロード
28    5900E058    C     0,88(0,14) 上限値と比較
2C    4720E046    BC    2,70(0,14) 大きければジャンプ
30    5890E05C    L     9,92(0,14) GR9=X'1000'
34    1A91        AR    9,1       GR9を拡張ベースレジスタに
36    58009214    L     0,532(0,9) Iの値GR0へ
3A    5A00E054    A     0,84(0,14) 増分加算
3E    50009214    ST    0,532(0,9) Iにストア
42    47F0E028    BC    15,40(0,14) 繰り返す
46    41001A30    LA    0,2608(0,1) INPUTファイルのブロック
4A    459010D8    BAL   9,216(0,1) CLOSEEXT(表7.2参照)呼び出し
4E    47F01058    BC    15,88(0,1)  リターン(図7.4参照)
52    0000                          (境界調整)
54    00000001                      (ループの増分値:リテラル)
58    00000003                      (ループの上限値:リテラル)
5C    00001000    (局所領域ベースレジスタ拡張用ディスプレースメント)
```

(d) (c) の目的コード

図7.13 簡単な for ループの場合の生成される目的コード(つづき)

タを新たにわりつけることとした。もちろん、中間コードが目的コードに変換された時点で、もはや参照されることのない中間項を保持している、すなわち解放できるレジスタがないか調査し、あれば直ちに解放することにより、むだな一時領域への待避がないようにしている。もはや参照されることのない中間項を保持している一時領域についても同様である。特にループ不変式である中間項を保持している場合は、当該ループ終了後ただちに解放する。ある中間項がもはや参照されることがないことの調査は、データフロー解析の結果ならびに7.1節で述べた中間項を最後に引用する中間コードの表(物理構造は変数 LASTREFHEAD の指すリニア・リスト)から容易に判定できる。

上記のレジスタ割り付けを行うため、各レジスタごとに現在使用中か否か、使用中の場合には、どの中間項を保持しているかの情報(**レジスタ記述子**、**register descriptor** と呼ばれることがある[36],[37])を持つレジスタ管理表を用いている。また、各中間項ごとに、現在それを何番レジスタが保持しているか/何番地の一時領域に待避されているのかという情報(**番地記述子**、**address descriptor** と呼ばれるものに相当する[36],[37])を持つ中間項管理表も使用している。これらの表は、あらたにレジスタが割り付けられたとき、レジスタが解放されたとき、中間項が一時領域に待避されたとき適切に更新される。なお、スカラ・ユニットのレジスタ割り付けにおけるレジスタ種別は、1汎用レジスタ、1組の偶数番汎用レジスタと奇数汎用番レジスタ、浮動小数点レジスタの3種類だけである。

現在の目的コード生成部では、**覗穴的最適化**(**peephole optimization**)[36],[37]に分類される以下の最適化を目的コード生成時に行っている。

(1) 代数的単純化:例えば X+0 は、加算を行わない等。

(2) 機械命令の特徴を利用

(2-1) インデックス・レジスタの利用:例えば配列参照の際、添字計算の加算を1回省ける。

(2-2) 第2オペランドに主記憶を参照できる命令の使用:第2オペランドの値をロードする命令を省ける。第1オペランドが主記憶を参照し、可換な演算の場合には両オペランドを交換した上で主記憶参照命令を使用。

(2-3) **演算の強さの軽減**(**reduction in strength, strength reduction**)[36],[37]:特に2のべき乗をかけるとき、シフト命令で実現している。

7.5 結 語

V-Pascal Version 1では、ベクトル命令の生成に際してもD行列を有効に利用している。このように、自動ベクトル化コンパイラにとってD行列は、重要な概念である。また、スカラ命令の生成におけるレジスタ割り付け、ならびにベクトル命令の生成におけるセットアップ等のデータ転送の必要性の判定には、データフロー解析の結果が利用されている。自動ベクトル化コンパイラにとって、データフロー解析の結果も通常の最適化を行うコンパイラ以上に重要性を増す。

V-Pascal Version 1の開発目的は、ベクトル化に主眼が置かれているため、最適化も限定した機能のみが実現されている。同様の理由で、スカラ・ユニットのレジスタ割り付けに関しても十分高性能なアルゴリズムが実現されているとはいえない。今後の改良では、特にスカラ処理の高速化にも力を注ぐ必要があると考えられる。なぜなら、十分ベクトル化され、ベクトル処理される部分の処理時間が短縮されると、残りのスカラ処理される部分の処理時間が相対的に増大するためである。

第8章
性能評価

8.1 緒 言

V-Pascalコンパイラの開発の第1の目標であるユーザプログラムの自動ベクトル化機能の能力を検証/評価するため、いくつかのPascalプログラムに対して生成された目的コード・プログラムを東京大学大型計算機センターのHITAC S-820/80で実行し、処理時間を測定した。また比較のため、同じ処理を記述したFORTRANプログラムをFORTRAN77/HAPと呼ばれるメーカ提供の自動ベクトル化コンパイラでコンパイルし、生成された目的コード・プログラムを同じくHITAC S-820/80で実行し、処理時間を測定した。第8章に示した測定結果は、すべて1988年7月から1989年2月にかけて行ったものである。後者のHITAC S-820/80上のFORTRANソース・プログラムについては、ソース・プログラムの任意の部分を実行するのに要したスカラ・ユニットの処理時間(以後SPUと記す)およびベクトル・ユニットの処理時間(以後VPUと記す)をそれぞれ独立に測定することができる。この測定は具体的には、スカラ・ユニット/ベクトル・ユニットそれぞれのそれまでの処理時間を計測する標準手続を、ソース・プログラムの測定したい部分の前後で呼び出すことにより簡単にできる。従って、実際のFORTRANで記述された時間測定用のソース・プログラムは、すべて図8.1に示す形態とした。ただし、これらの標準手続はFORTRAN/HAPコンパイラ固有のものである。この図に示すとおり、通常測定したい部分の処理時間の計測の誤差を相対的に減少させる目的から、測定したい部分を適当な回数分(図中M)だけ繰り返し実行させる(図中文番号555のループ)。そして、測定したい部分を繰り返し実行する際に、コンパイラの無用な最適化、例えば測定したい部分の(一部の)処理をループ不変式として同図中文番号555のループ外へ移動すること等、を極力抑えるために、測定したい部分のみを別のサブルーチン(同図中SUB1)とした。そのため、このままでは測定したい部分のみならず、それを含むサブルーチンを呼び出す処理、そのサブルーチンからリターンする処理、および同図中文番号555のループ制御変数に対するループの回転に伴う処理をも含めて時間を測定することと

なる。それゆえに、まず(A)図8.1に示す形態で時間を測定し、次に(B)図8.1から測定したい部分のみを取り除いて時間を測定する。{(A) – (B)}/Mを純粋な処理時間とする。なお、Pascalとの比較のためFORTRANで記述された図8.1の形式の時間測定用のソース・プログラムをメーカ提供の自動ベクトル化FORTRANコンパイラでコンパイル実行させる際には、最も強力な最適化/ベクトル化をコンパイル・オプション(OPT(4)、SOPT、HAP)で指定した。

次にPascalで記述された時間測定用のソース・プログラムの時間計測方法について述べる。V-Pascalの入力であるPascalソース・プログラム上で直接上記のFORTRAN同様SPUおよびVPUを簡単に測定できる機能を実現させることは、これらの時間計測が実際にはOSの特別なスーパーバイザ・コール(SVC)で実現されており、しかもそのスーパーバイザ・コールの仕様が公開されていないため非常に困難である。それゆえに、今回の性能評価のためPascalプログラムの実行時間測定は、すべて図8.2に示す形のプログラムで次のようにして行った。まず、測定したい部分をPascalの手続(図8.2(a)のSub1)として記述する。これは、FORTRANの場合と同様コンパイラの無用な最適化を抑えるためである。そして、その手続をPascalメイン・ルーチン(図8.2(a)のTest)の本体で適当な回数分(図中M)だけ繰り返し実行させる(図中のforループ)。このループの目的もFORTRANの場合の図8.1の文番号555のループ

```
      PROGRAM MAIN
      REAL*8 SPU,VPU
C     VPUおよびSPUそれぞれの計測のためのタイマの起動
      CALL VCLOCK
      CALL XCLOCK
      DO 555 I=1,M (測定時間を計測に適する程度に大きくするループ)
         CALL SUB1
  555 CONTINUE
C     VPUおよびSPUそれぞれのタイマの計測
      CALL XCLOCK(SPU)
      CALL VCLOCK(VPU)
      PRINT *,' SPU=',SPU,' VPU =',VPU
      STOP
      END
C
      SUBROUTINE SUB1
      (ループ等の実際に測定したい部分)
      RETURN
      END
```

図8.1 FORTRANの時間測定用ソース・プログラム

と同じである。そして、さらに実際に時間を測定させるために、このPascalプログラム全体(図8.2(a))をFORTRANのメイン・ルーチンから子モジュールとして呼出し、その前後で図8.1と同様にSPUおよびVPUを測定する(図8.2(b))。すなわち、この場合図8.3に示す処理の流れとなる。図8.3からわかるとおり、このPascalで記述されたソース・プログラムの時間計測でも、本来測定したい部分以外の処理にかかった時間も含めて測定される。そこで、FORTRANの場合と同じくまず(A)図8.2に示す形態で時間を測定し、次に(B)図8.2から測定したい部分のみを取り除いて時間を測定する。{(A) – (B)}/Mを純粋な処理時間とする。こうすることにより、図8.3に示した本来測定したい部分以外の処理にかかった時間をすべて取り除くことが

```
program Test(input,output);
  var i:integer;

  procedure Sub1;
    begin {Sub1}
      {ループ等の実際に測定したい部分}
    end {Sub1};

  begin {Test}
    for i:=1 to M do{測定時間を計測に適する程度に大きくするループ}
      Sub1;
  end {Test}.
```

(a) Pascalの時間測定用ソース・プログラム

```
      PROGRAM MAIN
      REAL*8 SPU,VPU
C     VPUおよびSPUそれぞれの計測のためのタイマの起動
      CALL VCLOCK
      CALL XCLOCK
C     Pascal実行時メイン・ルーチン呼び出し
      CALL PASCAL
C     VPUおよびSPUそれぞれのタイマの計測
      CALL XCLOCK(SPU)
      CALL VCLOCK(VPU)
      PRINT *,' SPU=',SPU,' VPU =',VPU
      STOP
      END
```

(b) Pascalの時間測定用プログラムを呼び出すFORTRANソース・プログラム

図8.2 Pascalの時間測定用ソース・プログラム

できる。

なお、FORTRANの場合もPascalの場合も測定されたSPUにはVPUの時間が含まれていることに注意が必要である。なぜなら、HITAC S-820(S-810も同様)の場合、スカラ・ユニットがベクトル・ユニットを起動し、ベクトル・ユニットの処理中スカラ・ユニットは並列に別の処理をするか、あるいはTVP命令でベクトル・ユニットの終了をビジーウエイト(busy wait)するかである。つまり、ベクトル・ユニットの処理中も入出力を行わない限り、常にスカラ・ユニットのCPUが起動している状態にある。そして、ここで述べる時間測定を行ったプログラムでは実際並列に入出力を行わないからである。つまり、このSPUは、ベクトル計算機でない通常の汎用計算機におけるCPU時間に相当する。当然ベクトル化率の高いプログラムでは、SPUとVPUは近い値となる。しかし、ベクトル化率が高くベクトル化できたとしても、ス

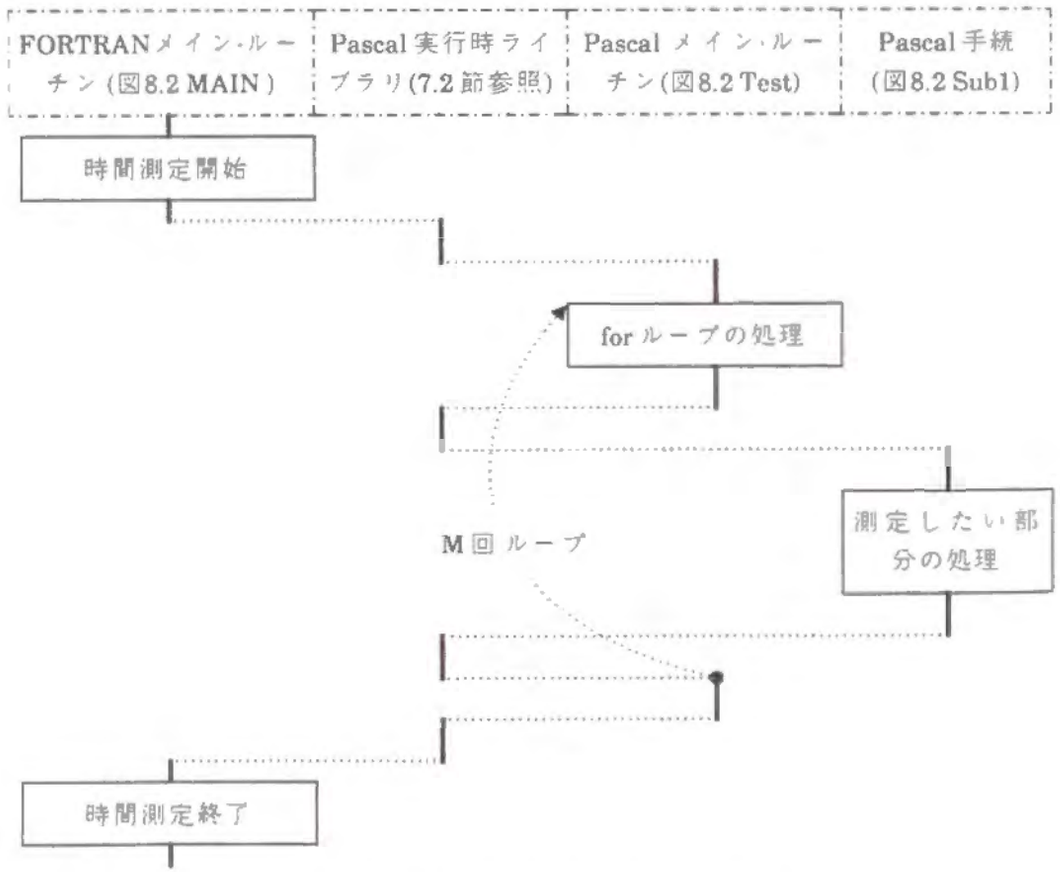


図8.3 Pascalの時間測定用ソース・プログラムの処理の流れ

カラ実行したときのSPUより、時間がかかることは望ましくない。これらのことから、以下では単にSPUの時間で性能評価を行う。また、時間計測を行ったプログラム例は以下では図8.1あるいは図8.2に示す時間計測プログラム全体ではなく、測定したい部分のみをプログラムとして示すことにする。

なお、V-Pascalコンパイラ自身が実行される計算機とFORTRAN77/HAPコンパイラが実行される計算機とが異なる。従って、V-Pascalコンパイラのコンパイル時間に関しては、厳密な評価を行えないため以下では言及を避ける。

8.2 多重ループのベクトル化

図8.4は、FFTバタフライ演算を64点のノードに対して施すPascalプログラムである。このプログラムは、通常2のべき乗個のノードに対するバタフライ演算は、各ノードのデータを保持する配列の2のべき乗個離れた要素どうしを演算するため、その配列参照の際主記憶のバンク・コンフリクトを生じる可能性が高い。このプログラムは、使用されない要素を適切に配置することにより、バンク・コンフリクトを避けるように、すなわちベクトル処理向けに作られている[50]。ベクトル計算機による外部多次元FFTに関しては文献[49]に詳しい。この図8.4のプログラムをFORTRANに書き換えたプログラムも用意し、両者の実行時間を8.1節で述べた方式で測定した結果を表8.1に示す。なお、FORTRANとPascalとでは、多次元配列を主記憶上に1次元化して割り付ける添字計算が行優先/列優先と異なるため、FORTRANプログラムの2次元配列の参照の添字式はPascalプログラムと1次元目/2次元目を入れ換えたものとした。以下で述べるすべての評価用プログラムについて、このように添字式を入れ換えて比較している。厳密な比較を行うには、3次元以上の配列についても同様に添字式の並びの順序を交換しておく必要がある。しかし、今回測定した評価用のプログラムに関しては2次元までの配列しか出現しない。

表8.1において、V実行の欄はベクトル実行した際のSPUを示し、S実行の欄は強制的に評価用のプログラムをスカラ実行した際のSPUを示している。この図8.4の64点FFTバタフライ演算を行う評価用プログラムは、本来3重の最外側ループを展開したため、2重ループが6個並んだ形となっている。しかも、いずれの2重ループも外側および内側のループともに繰り返し回数が多くない。このプログラムをメーカ提供のFORTRANコンパイラでコンパイルさせると、各2重ループの大きい方の繰

り返し回数を有するループが選択され、そのループでベクトル実行される。しかし、それでも繰り返し回数が単純平均19弱と多くないため、表8.1のFORTRANのS/V比は大きくない。すなわち、ベクトル長が短いためベクトル実行した際の加速率が低い。これに対し、V-Pascalでコンパイルした場合には各2重ループすべてについて、2重ループ全体を一重化し32のベクトル長でベクトル実行される。そのため、表8.1のPascalのS/V比は大きくなっている。ただし、S/V比すなわち加速率には、S実行とV実行両方のSPUが関係することに注意が必要である。今の場合、V-

```
FOR I:= 1 TO 32 DO {配列データおよび演算はすべて8バイトの実数型}
   FOR K:= 1 TO 1 DO BEGIN
      VR(.K,I.):= WR(.32*K-31.)*FR(.I+K+31.) -
                  WI(.32*K-31.)*FI(.I+K+31.);
      VI(.K,I.):= WR(.32*K-31.)*FI(.I+K+31.) +
                  WI(.32*K-31.)*FR(.I+K+31.);
      GR(.2*I+K-2.):= FR(.I+K-1.) + VR(.K,I.);
      GI(.2*I+K-2.):= FI(.I+K-1.) + VI(.K,I.);
      GR(.2*I+K-1.):= FR(.I+K-1.) - VR(.K,I.);
      GI(.2*I+K-1.):= FI(.I+K-1.) - VI(.K,I.);
   END;
FOR I:= 1 TO 16 DO
   FOR K:= 1 TO 2 DO BEGIN
      VR(.K,I.):= WR(.16*K-16+1.)*GR(.2*I+K+30.) -
                  WI(.16*K-16+1.)*GI(.2*I+K+30.);
      VI(.K,I.):= WR(.16*K-16+1.)*GI(.2*I+K+30.) +
                  WI(.16*K-16+1.)*GR(.2*I+K+30.);
      FR(.5*I+K-5.):= GR(.2*I+K-2.) + VR(.K,I.);
      FI(.5*I+K-5.):= GI(.2*I+K-2.) + VI(.K,I.);
      FR(.5*I+K-3.):= GR(.2*I+K-2.) - VR(.K,I.);
      FI(.5*I+K-3.):= GI(.2*I+K-2.) - VI(.K,I.);
   END;
FOR I:= 1 TO 8 DO
   FOR K:= 1 TO 4 DO BEGIN
      VR(.K,I.):= WR(.8*K-8+1.)*FR(.5*I+K+35.) -
                  WI(.8*K-8+1.)*FI(.5*I+K+35.);
      VI(.K,I.):= WI(.8*K-8+1.)*FR(.5*I+K+35.) +
                  WR(.8*K-8+1.)*FI(.5*I+K+35.);
      GR(.9*I+K-9.):= FR(.5*I+K-5.) + VR(.K,I.);
      GI(.9*I+K-9.):= FI(.5*I+K-5.) + VI(.K,I.);
      GR(.9*I+K-5.):= FR(.5*I+K-5.) - VR(.K,I.);
      GI(.9*I+K-5.):= FI(.5*I+K-5.) - VI(.K,I.);
   END;
```

図8.4 FFTバタフライ演算を行う評価用プログラム(つづく)


```
FOR I:= 1 TO 4 DO
   FOR K:= 1 TO 8 DO BEGIN
      VR(.K,I.):= WR(.4*K-4+1.)*GR(.9*I+K+27.) -
                  WI(.4*K-4+1.)*GI(.9*I+K+27.);
      VI(.K,I.):= WI(.4*K-4+1.)*GR(.9*I+K+27.) +
                  WR(.4*K-4+1.)*GI(.9*I+K+27.);
      FR(.17*I+K-17.):= GR(.9*I+K-9.) + VR(.K,I.);
      FI(.17*I+K-17.):= GI(.9*I+K-9.) + VI(.K,I.);
      FR(.17*I+K-9.):= GR(.9*I+K-9.) - VR(.K,I.);
      FI(.17*I+K-9.):= GI(.9*I+K-9.) - VI(.K,I.);
   END;
FOR I:= 1 TO 2 DO
   FOR K:= 1 TO 16 DO BEGIN
      VR(.K,I.):= WR(.2*K-2+1.)*FR(.17*I+K+17.) -
                  WI(.2*K-2+1.)*FI(.17*I+K+17.);
      VI(.K,I.):= WI(.2*K-2+1.)*FR(.17*I+K+17.) +
                  WR(.2*K-2+1.)*FI(.17*I+K+17.);
      GR(.33*I+K-33.):= FR(.17*I+K-17.) + VR(.K,I.);
      GI(.33*I+K-33.):= FI(.17*I+K-17.) + VI(.K,I.);
      GR(.33*I+K-17.):= FR(.17*I+K-17.) - VR(.K,I.);
      GI(.33*I+K-17.):= FI(.17*I+K-17.) - VI(.K,I.);
   END;
FOR I:= 1 TO 1 DO
   FOR K:= 1 TO 32 DO BEGIN
      VR(.K,I.):= WR(.K.)*GR(.33*I+K.) -
                  WI(.K.)*GI(.33*I+K.);
      FR(.65*I+K-65.):= GR(.33*I+K-33.) + VR(.K,I.);
      FI(.65*I+K-65.):= GI(.33*I+K-33.) + VI(.K,I.);
      FR(.65*I+K-33.):= GR(.33*I+K-33.) - VR(.K,I.);
      FI(.65*I+K-33.):= GI(.33*I+K-33.) - VI(.K,I.);
   END;
```

図8.4 FFTバタフライ演算を行う評価用プログラム(つづき)

表8.1 64点FFTバタフライ演算の実行時間(時間の単位はμsec)

Pascal			FORTRAN		
V実行	S実行	S/V比	V実行	S実行	S/V比
23.48	891.4	38.0	81.02	163.2	2.01

V実行は、ベクトル実行したときのSPU
S実行は、強制的にスカラ実行したときのSPU
S/V比は、両者の実行時間の比(加速率)

Pascalが生成するスカラ実行の目的コードが良くないため、表8.1のPascalのS実行のSPUが、FORTRANのそれと比べ大きく、その分S/V比が大きくなっているのである。そこで、V-PascalとFORTRANコンパイラ両者が生成したベクトルの目的コード(この例ではループ内のすべての処理がベクトル化されるため、スカラの目的コードの実行時間はほとんど無視できる)に関する性能をV実行のSPUで比較する。すると、V-Pascalでコンパイルした場合一重化実行の効果により、表8.1のPascalのV実行が、FORTRANのそれと比べ4倍弱高速に実行されていることが実証された。

次に、図8.5に挙げた行列の転置を行う評価用プログラムの実行結果を表8.2に示す。この表で、NMAXの欄は図8.5中のループの上限値を変化させた値を示している。この値は、処理する行列(2次元配列)の1辺の大きさである。また、先の例で述べたとおり、V-Pascalのベクトル化の能力を検証するのに最適である、Pascalの

```
FOR I:= 1 TO NMAX DO BEGIN{配列データはすべて4バイトの整数型}
    FOR J:= 1 TO NMAX DO BEGIN
        A(.I,J.):= 1;
    END;
    FOR J:= 1 TO I-1 DO BEGIN
        A(.I,J.):= 0;
    END;
    A(.I,I.):= 2;{対角要素初期化}
    A(.I,I+1.):= 3;
    FOR J:= 1 TO NMAX DO BEGIN
        A(.I,J.):= A(.J,I.);
    END;
END;
```

図8.5 行列の転置を行う評価用プログラム

表8.2 行列の転置を行う評価用プログラムの実行時間(時間の単位はμsec)

NMAX	Pascal V実行	Pascal S実行	Pascal S/V比	FORTRAN V実行	FORTRAN S実行	FORTRAN S/V比	$V_{FORTRAN}/V_{Pascal}$
20	10.46	270.4	25.9	38.17	70.39	1.84	3.7
50	31.44	1579.	50.2	100.9	375.9	3.73	3.2

V実行は、ベクトル実行したときのSPU
S実行は、強制的にスカラ実行したときのSPU
S/V比は、両者の実行時間の比

V実行とFORTRANのV実行との比を最後の欄につけた。この値は、PascalのV実行が、FORTRANのそれと比べ何倍速いかを示すものである。その他の欄は、表8.1と同じである。この図8.5の評価用プログラムは、行列の上三角/下三角/対角要素等を初期化し、最後に転置を行うものである。この評価用プログラムでは、FORTRANコンパイラが最内側のループのみでベクトル化(ベクトル長の単純平均NMAX/2程度)し、対角要素の初期化部はベクトル化できなかった。これに対し、V-Pascalでコンパイルした場合2重ループ内の文はすべて一重化しベクトル実行され、対角要素の初期化部等の1重ループ内の文もすべてベクトル実行される。その結果、V-Pascalでコンパイルした場合、FORTRANのベクトル実行より3倍以上高速に実行されることが実証された。

このように、V-Pascalが持つ一重化機能は、特に多重ループの中のいずれのループも繰り返し回数が多くない場合に有効であるといえる。また、図8.5の例の場合のように多重ループ内に本来ベクトル化可能であるのに、従来のコンパイラではそれを検出できずベクトル化できない部分が存在する場合には、いうまでもなくその点でもV-Pascalの能力の優位性が実証されたと考える。

8.3 if文を含むループのベクトル化

先に、図8.5に挙げた行列の転置を行う評価用プログラムにif文を追加した図8.6の評価用プログラムの実行結果を表8.3に示す。この表の各欄は、表8.2と同じである。この図8.6の評価用プログラムは、図8.5の評価用プログラム同様行列の上三角/下三角/対角要素等を初期化し、最後に転置を行うものである。図8.5の評価用プログラムと異なる点は、最初の上三角の初期化がif文の制御を受け配列A自身の値により実行されるか、実行されないかが依存している。この評価用プログラムの実行時間測定に際しては、図8.2(a)のPascalメイン・ルーチンTestにおいて、まず配列Aのすべての偶数列を0に、すべての奇数列を1にあらかじめ初期化した(この初期化に要する時間は8.1節で述べた測定方法を採用すると、表8.3の時間には含まれない)。そして表8.3に示すとおりNMAXの値は偶数であるので、この図8.6のif文の真率(真となる割合)は、50%である。つまり、この図8.6のif文の制御を受ける文は、スカラ実行ではif文の実行回数の半分の回数だけ実行される。ベクトル実行では、3.3.6項で述べた種々のif文のベクトル化の中で、HITAC S-820ではマスク付き命令

```
FOR I:= 1 TO NMAX DO BEGIN{配列データはすべて4バイトの整数型}
   FOR J:= I TO NMAX DO BEGIN
      IF A(.I,J.) > 0 THEN
         A(.I,J.):= -1;{対角要素初期化}
   END;
   FOR J:= 1 TO I-1 DO BEGIN
      A(.I,J.):= 0;
   END;
   A(.I,I.):= 2;{上三角要素初期化}
   A(.I,I+1.):= 3;
   FOR J:= I TO NMAX DO BEGIN
      A(.I,J.):= A(.J,I.);
   END;
END;
```

図8.6 if文を含む行列の転置を行う評価用プログラム

表8.3 図8.6の評価用プログラムの実行時間(時間の単位はμsec)

	Pascal			FORTRAN			$V_{FORTRAN}$
NMAX	V実行	S実行	S/V比	V実行	S実行	S/V比	$/V_{Pascal}$
20	10.69	291.5	27.3	44.01	80.08	1.82	4.1
50	33.58	1666.	49.6	117.1	441.7	3.77	3.5

V実行は、ベクトル実行したときのSPU
S実行は、強制的にスカラ実行したときのSPU
S/V比は、両者の実行時間の比

で実行するのが効率的である。従って、ベクトル実行の場合には、スカラ実行の実行回数に相当するベクトル長は、if文のそれと同じである。

この図8.6の評価用プログラムでも、FORTRANコンパイラが最内側のループのみでベクトル化(ベクトル長の単純平均NMAX/2程度)し、対角要素の初期化部はベクトル化できなかった。これに対し、V-Pascalでコンパイルした場合2重ループ内の文はすべて一重化しベクトル実行され、対角要素の初期化部等の1重ループ内の文もすべてベクトル実行される。その結果、V-Pascalでコンパイルした場合、FORTRANのベクトル実行より4倍程度高速に実行されることが実証された。

8.4 一重化の効果

図8.7の評価用プログラムの実行時間を表8.4に示す。この表ではSPUとVPUの値を示した。また、一重化の効果を検証するためにPascalで記述されたプログラムについては、V-Pascalコンパイラで一重化しベクトル実行させた場合、最内側のループのみをベクトル実行させた場合(コンパイル・オプションで指定できる)、それぞれについて比較した。そして、if文の真率も変化させて比較した。この結果からも、V-Pascalコンパイラの一重化機能により、FORTRANコンパイラが生成する目的コードよりも2倍程度実行速度が速いことがわかる。このPascalで記述されたプログラムを一重化しベクトル実行させた場合には、SPUとVPUがほとんど一致していることから、ベクトル化率が非常に高いといえる。実際、このプログラム例ではループ中の処理はすべてベクトル化されるので、スカラ・ユニットでの処理は1回のベクトル・ユニットの起動に必要なセットアップのみである。すなわち、この場合には1回のセットアップにほぼ1μsec要すると思われる。それに対し、Pascalで記述されたプログラムを最内側のループのみをベクトル実行させた場合には、やはりループ中の処理はすべてベクトル化されるが、セットアップが外側のループの繰り返し回数(今の場合30)必要となるため、スカラ・ユニットでの処理がその分増大しSPUとVPUとの差が24μsec程度とひらいている。それに対し、同様の処理を行っていると思われるFORTRANコンパイラが生成する目的コードでは、SPUとVPUとの差がほとんどない。これは、スカラ・ユニットでのセットアップの処理をベクトル・ユニットの処理と並列に実行させているためであると考えられる。また、Pascalで記述されたプログラムを最内側のループのみをベクトル実行させた場合のVPUが、FORTRANで記述されたプログラムをベクトル実行させた場合のVPUと比較し大きいのは、使用するベクトル命令、ベクトル・レジスタの組み合せ等アーキテクチャの細部まで見渡した効率的な目的コード生成が行われていないためであると考えられる。

この表8.4から、if文の真率はすくなくとも今の場合のようにマスク付き命令で処理する場合には、ほとんど実行時間に影響しないことも判明した。

図8.8は図8.7が主記憶参照はすべて連続アクセスのベクトルロード/ストア命令となる(3.3.1項参照)のに対し、図8.8は主記憶参照はすべて間接アクセスのベクトル

```
FOR I:= 1 TO 30 DO{配列データはすべて4バイトの整数型}
    FOR J:= 1 TO 30 DO
        IF B(.I,J.) > 0 THEN
            A(.I,J.):= A(.I,J.) + B(.I,J.) ;
```

図8.7 if文を含む連続アクセスの評価用プログラム

表8.4 図8.7の評価用プログラムの実行時間(時間の単位はμsec)

if文の真率	Pascal				FORTRAN (最内側のみV)	
	一重化V		最内側のみV			
	SPU	VPU	SPU	VPU	SPU	VPU
80	9.69	8.73	51.9	27.7	18.0	17.6
50	8.64	7.65	51.4	27.6	17.9	17.6
20	7.23	6.52	50.9	27.5	17.9	17.6

一重化Vとは一重化し多重ループ全体をベクトル実行した場合
最内側のみVとは多重ループの最内側ループのみをベクトル実行した場合

```
FOR I:= 1 TO 30 DO{配列データはすべて4バイトの整数型}
    FOR J:= 1 TO 30 DO
        IF C(.I,J.) > 0 THEN
            A(.I*J.):= B(.I*J.) * 2;
```

図8.8 if文を含む間接アクセスの評価用プログラム

表8.5 図8.8の評価用プログラムの実行時間(時間の単位はμsec)

if文の真率	Pascal				FORTRAN (最内側のみV)	
	一重化V		最内側のみV			
	SPU	VPU	SPU	VPU	SPU	VPU
80	7.94	6.90	63.5	29.0	14.1	13.6
50	7.31	6.35	62.6	28.3	14.0	13.5
20	6.88	5.88	61.1	27.2	14.0	13.5

一重化Vとは一重化し多重ループ全体をベクトル実行した場合
最内側のみVとは多重ループの最内側ループのみをベクトル実行した場合

ロード/ストア命令となる(3.3.1項参照)。この場合の実行時間を表8.5に示す。この結果からも、V-Pascalコンパイラの一重化機能により、FORTRANコンパイラが生成する目的コードよりも2倍程度実行速度が速いことがわかる。その他の点についても先の図8.7、表8.4の場合とまったく同様の結果となっている。従って、一重化の効果は主記憶参照の方式には関係しないといえる。

8.5 結 語

以上述べてきたとおり、V-Pascalが持つ多重ループ全体にわたる強力なベクトル化機能により、最内側ループ以外のループに属する文も可能ならばすべてベクトル化され、図8.5、図8.6の場合のように現在のFORTRANコンパイラではベクトル化できなかった文をもベクトル化できるようになった。さらに、V-Pascalが持つ一重化機能により、最内側ループの繰り返し回数が少ない場合(S-820において、どのような場合に一重化が効果的であるかは、文献[23]に詳しい)、最内側ループのみをベクトル実行するよりも数倍高速な処理が可能となった。これらのV-Pascalの特長的な機能は、第5章および第6章で述べたとおりソース言語に依存しない。それゆえ、現在の自動ベクトル化FORTRANコンパイラに組み込むことも可能である。その場合には、現在のFORTRANコンパイラの強力な最適化機能ならびに秀逸な目的コード生成能力を併せ持つこととなり、さらに優れた自動ベクトル化コンパイラとなる。もちろん、この一重化機能の効果はターゲット・マシンの設計思想、特に一重化に多用される間接アクセス型のベクトルロード/ストア命令の性能をどの程度重視するかにより大きく変化する。V-Pascalが現在ターゲット・マシンとして選んだHITAC S-820(S-810も同様)シリーズの場合、この間接アクセス型のベクトルロード/ストア命令の実行速度は、連続アクセス型/等間隔アクセス型のベクトルロード/ストア命令の実行速度とほぼ同じであるため、一重化に伴うオーバーヘッドは最小限にとどまっていると考えられる。それに対し、V-Pascalコンパイラの前身ともいえるFORTRAN用の一重化を行うベクトル化プリプロセッサ[24]~[26]の開発当初に行った予備実験では、当時の日立以外の国産メーカのベクトル計算機の間接アクセス型のベクトルロード/ストア命令の性能は、連続アクセス型/等間隔アクセス型のベクトルロード/ストア命令の実行速度と比べ3倍から9倍遅かった(2.5節参照)。このような場合には、一重化に伴うオーバーヘッドが大きく、一重化の効果が生じるケースが

少なくなる。そして、現在一重化は基本ベクトル(5.3節参照)をコンパイル時に生成できる場合にのみ行っている。一重化機能の適用範囲をさらに広げるためには、基本ベクトルを実行時に生成することも考える必要がある。ところが、その場合には基本ベクトル生成の時間も一重化に伴うオーバーヘッドの大きな因子となることに注意が必要である。

ベクトル計算機は、第2章で述べたとおり種々の並列処理機構を有している。そのため、処理性能の尺度をなにに置くかということから、様々な議論がなされている[51]。ここでは、単純にSPUで比較しているが、評価用のプログラム・ライブラリ(いわゆるベンチマーク・スーツ)と合わせ基準/標準化が望まれる。

第9章
結論

以上述べてきたとおり、自動ベクトル化Pascalコンパイラ V-Pascalの主目的である強力なベクトル化機能は、

(1) ベクトル化可能性の判定に必要となる各種の依存関係解析機能を、多重ループ全体を対象とし、かつ厳密に解析できるものとすること、

(2) 同解析機能により判定した依存関係から細かな演算レベルでのベクトル化可能部分を抽出する強力な部分ベクトル化機能を持たせること、

(3) ベクトル化可能な多重ループ内の処理を、等価な1重ループ内での処理として長大なベクトル長でベクトル実行させる目的コードに翻訳すること

により実現されている。従来のメーカ提供の自動ベクトル化コンパイラは、多くの場合多重ループ全体でなく最内側ループのベクトル化にしか対応できていなかった。そのため、見過ごされてきた最内側ループ以外のループ中に存在する部分が、これらの機能を合わせ持つことにより、初めてベクトル化可能となった。すなわち、このような多重ループに関しては従来に比べ、ベクトル化率が大幅に向上する。しかも、一重化機能により長大なベクトル長でベクトル実行されるので、最内側ループだけのベクトル長でベクトル実行される場合より、さらに有効にパイプライン演算器を使用でき、数倍高速処理される場合があることを初めて実証した。同時に、この一重化は間接参照型の主記憶アクセスを頻繁に行うため、この種のアクセスを高速処理できるアーキテクチャが重要であることも第8章で指摘したとおりである。これは、今後のベクトル計算機を設計する際に、重要な一指針となる知見であると考えられる。

これらのV-Pascalの特長的な機能は、第5章および第6章で述べたとおりソース言語に依存しない。それゆえ、既存のあるいは今後開発されるFORTRANその他の自動ベクトル化コンパイラに組み込むことも可能である。逆に、第7章および第8章で触れたV-Pascalが生成する目的コードのスカラ実行部分のさらなる高速化のためには、現在のFORTRANコンパイラ中に実現されている各種の最適化機能ならびに効率的なレジスタ割当て機能等を有する目的コード生成方式を、V-Pascalにも順次実現していく必要があると思われる。また、ベクトル化に関しても、さらにベクトル

化の対象ループを広げベクトル化率を上げるための研究、各種依存関係解析アルゴリズムの厳密な解析の適用範囲の拡大と高速化、ループ交換等のより強力なベクトル化手法の実現、一重化の適用範囲の拡大等今後のより一層の発展が期待される。特に、各種依存関係解析はベクトル化のみならず、並列化に際しても必要となる重要かつ基礎的な技術である。それゆえに、今後も重点的に研究をさらに深めていく必要がある。

現在NEC SX-3に代表される次世代ベクトル計算機のアーキテクチャが数台のベクトル-ユニットを持つ密結合のマルチ-プロセッサ方式を指向している。この次世代ベクトル計算機のマルチ-プロセッサ方式では、その性能を引き出すために、現在のベクトル計算機に対して自動ベクトル化コンパイラが果たした役割以上にコンパイラ、デバッガ等様々なソフトウェアによる支援が重要である。V-Pascalプロジェクトにおいても、この点に着目し今後の発展の重要な一目標として自動ベクトル化/自動並列化コンパイラとしての実現を目指すものである。

謝辞

本研究の機会を与えて頂いたうえ、本研究を遂行するにあたって、終始懇篤なご指導ご鞭撻を賜った恩師京都大学工学部津田孝夫教授に深甚なる謝意を表します。

常々本研究に関してご討論下さり、V-Pascalの開発のすべての段階においても、また本論文をまとめるに際しても、様々な有益かつ適切なご助言を頂いた京都大学工学部大久保英嗣助教授に心より感謝致します。

本研究に関して熱心にご討論下さり、本研究の遂行ならびに本論文の作成に対し貴重なご助言を頂いた九州大学大型計算機センター島崎眞昭教授、京都大学工学部岡部寿男助手に厚く御礼申し上げます。

また、本論文校閲の労をおとり下さいました京都大学工学部松本吉弘教授に拝謝致します。

自動ベクトル化に関し初めて研究を開始した当時、津田研究室で開発したFORTRANプリプロセッサの実現に協力された二宮正和氏(マツダ(株)勤務)、栗屋透氏(大阪市役所勤務)に感謝致します。

V-Pascalプロジェクトの初期の基本設計段階では、葉山悟氏(コントロールフロー解析等;富士ゼロックス勤務)、藪内健二氏(構文解析、データフロー解析等;三洋電機勤務)、三木英輔氏(中間コード設計;NTT勤務)の熱心な助力に負うところが大きくここに深く感謝致します。また、開発にあたり熱心にご討論下さり、作業を分担して努力下さったあるいは現在も助力下さる京都大学工学部津田研究室のベクトル化グループの卒業生ならびに在学生の諸氏に対し、ここに名を挙げ深く感謝致します。

大西達也氏(ベクトル化等;松下電器産業勤務)
笹倉万里子氏(依存解析;京都高度技術研究所勤務)
中田秀男氏(依存解析、ベクトル化等;三菱電機勤務)
Piyaporn Atipas氏(ベクトル化、最適化等;IBM Thailand勤務)
小塚裕史氏(ベクトル化;野村総合研究所勤務)
佐藤研治氏(ベクトル化)
末廣謙二氏(ベクトル化)
斉藤隆氏(目的コード生成)

金原弘明氏(ベクトル化)
児島彰氏(並列化)
田村哲氏(依存解析)
中村素典氏(ベクトル化)
水沼一郎氏(依存解析)
上原哲太郎氏(ベクトル化)
松本史氏(別名解析)

そして、V-Pascal開発作業を一部分担下さった、塩野純氏(沖電気工業勤務)、空一弘氏(NTT勤務)、永峰聡氏(松下電器産業勤務)、湊真一氏(日本電信電話勤務)、安岡孝一氏(京都大学大型計算機センター)、越智裕之氏に対しても感謝致します。また、研究会等でご討論下さった京都大学工学部津田研究室の他の卒業生ならびに在学生の諸氏に対し感謝致します。

V-Pascal開発にあたり快くPascalの処理系の使用を応諾下さいました明治大学理工学部疋田輝雄教授、東京工業大学総合情報処理センター前野年紀助教授に対し謹んで御礼申し上げます。

V-Pascal開発にあたり自社のベクトル計算機の命令セット、細かなアーキテクチャに関する資料等を提供して下さり、様々な便宜を与えてくださった株式会社日立製作所、日本電気株式会社、富士通株式会社の関係各位に対しここに厚く御礼申し上げます。

以上本研究を行う上には、多くの方々のご指導、ご支援等を頂戴致しました。いずれを欠いても現在のV-Pascalはありえませんでした。再度深く御礼申し上げます。

本論文を推敲する際、日本語文章推敲支援ツール『推敲』を利用させて頂きました。このツールは、非常に有益な数多くの示唆を与えてくれました。このツールの使用をご快諾下さいました九州大学工学部牛島和夫教授に対し謹んで御礼申し上げます。

最後に本研究の一部は、文部省科学研究費試験研究(2)課題番号60880007、同試験研究(2)課題番号63880009、同試験研究(2)(B)課題番号02558004、ならびに同奨励研究(A)課題番号02780025によるものであることを付記し、深く謝意を表します。

参考文献

◎スーパーコンピュータ、ベクトル計算機関連

[1] 島崎眞昭:スーパーコンピュータとプログラミング,共立出版, 1989.

[2] K.Hwang: Evolution of Modern Supercomputers, Tutorial of Supercomputer: Design and Applications, IEEE (1984).

[3] 唐木幸比古:スーパーコンピュータと行列計算,情報処理, Vol.28, No.11, pp.1441-1451 (1987).

[4] 河辺峻,小林二三幸,村山浩,桐生芳雄,半田洋光,田上文一,後藤志津雄,青山智夫:シングル-プロセサで最大性能2 GFLOPSのS820,日経エレクトロニクス, No. 437, pp.111-125 (1987).

[5] 小高俊彦,河辺峻:HITAC S-810/S-820,コンピュートロール, No.20, pp.96-101 (1987).

[6] 日立製作所:S-820処理装置, 6020-2-001.

[7] 平栗俊男,田畑晃,槌本隆光,田口尚三:マシン・サイクル7.5nsを達成した並列パイプライン処理方式のスーパコンピュータFACOM VP,日経エレクトロニクス, No. 314, pp.131-155 (1983).

[8] 田村宏:FACOM VPシリーズ,コンピュートロール, No.20, pp.102-108 (1987).

[9] 岡本哲郎,田村宏,内田啓一郎:スーパーコンピュータFACOM VPのハードウェア,FUJITSU, Vol.30, No.4, pp.465-475 (1984).

[10] 富士通:FACOM VPシリーズハードウェア機能説明書, 79HS-1010-1.

[11] 古勝紀誠,渡辺貞,近藤良三:最大性能1.3 GFLOPS,マシン・サイクル6nsのスーパコンピュータSXシステム,日経エレクトロニクス, No. 356, pp.237-272 (1984).

[12] 岩屋暁宏:NEC SXシリーズ,コンピュートロール, No.20, pp.109-115 (1987).

◎ベクトル化、依存解析

[13] Utpal Banerjee: Dependence Analysis for Supercomputing, Kluwer Academic Publishers (1988).

[14] Michael Wolfe : Optimizing Supercompilers for Supercomputers , The MIT Press , Cambridge , Massachusetts (1989) .

[15] J.R.Allen, Ken Kennedy, Carrie Porterfield, Joe Warren : Conversion of Control Dependence to Data Dependence, *10th Annual ACM Symposium on Principles of Programming Languages*, pp.177-189 (1983) .

[16] 金田 泰 , 石田和久 , 布広永示 : 配列の大域データフロー解析法, 情報処理学会論文誌 , Vol. 28 , No. 6 , pp. 567-576 (1987) .

[17] 石田和久, 金田 泰 : 多重ループの配列多重添字解析方式, 情報処理学会プログラミング言語研究会, Vol.87, No.9 (1987) .

[18] 津田孝夫 , 国枝義敏 , 二宮正和 , 粟屋 透 : ループ間にまたがるデータ参照関係をもつ多重ループの自動ベクトル化, 情報処理学会論文誌 , Vol. 26 , No. 3 , pp. 536-544 (1985) .

[19] 國枝義敏 , 津田孝夫 : 自動ベクトル化コンパイラのための制御関係解析法 , 情報処理学会論文誌 , Vol. 30 , No. 9 , pp. 1164-1174 (1989年9月) .

[20] 日立製作所 : HITAC プログラムプロダクト VOS2/VOS3 最適化 FORTRAN77 , HAP FORTRAN77 使用の手引き , 8080-3-258-50 .

[21] 富士通 : FACOM OS IV/F4 MSP FORTRAN77/VP プログラミングハンドブック V10用 , 78SP-5740-1 .

[22] 日本電気 : FORTRAN77 , 77/SX プログラミング手引書 , GGB12-1 , 1985年6月初版.

[23] 福岡 均 , 布広永示 , 田中義一 : FORTRAN 機能強化ー多重ループのループ一重化方式ー, 情報処理学会第34回全国大会論文集 (II) , 5V-3 , pp. 861-862 (1987) .

[24] 津田孝夫, 国枝義敏, 二宮正和, 粟屋透 : ソースプログラムの自動変換によるベクトル化率の向上, 1. 任意多重ループの一重化など, 情報処理学会第29回全国大会, 4B-8, pp. 153-154 (1984年9月) .

[25] 津田孝夫 , 国枝義敏 , 二宮正和 , 粟屋透 : ソースプログラムの自動変換によるベクトル化率の向上 , 2. ベクトル化プリプロセッサの開発 , 情報処理学会第29回全国大会論文集 , 4B-9 , pp. 155-156 (1984年9月) .

[26] T.Tsuda, Y.Kunieda : Mechanical Vectorization of Multiply Nested DO Loops by Vector Indirect Addressing, *Proceedings of the IFIP 10th World Computer Congress*, pp. 785-790 (Sep 1986) .

[27] Takao Tsuda , Yoshitoshi Kunieda : V-Pascal : an Automatic Vectorizing Compiler for Pascal with No Language Extensions , *Proceedings of SUPERCOMPUTING '88* , November 14-18 , 1988 , Orlando , Florida (IEEE Compuer Society) , pp.182-189 (1988) .

[28] Takao Tsuda , Yoshitoshi Kunieda , Piyaporn Atipas : Automatic Vectorization of Character String Manipulations And Relational Operations in Pascal , *Proceedings of SUPERCOMPUTING '89* , November 13-17 , 1989 , Reno , Nevada (ACM SIGARCH) , pp.187-196 (1989) .

◎言語処理系

[29] スティーブン・ペンバートン, マーチン・ダニエルズ共著, 武市正人, 木村友則共訳 : Pascal の言語処理系 Pascal-P4, 近代科学社 (1984) .

[30] 正田輝雄 ; コンパイラのキットを用いた PASCAL の移植, 日経エレクトロニクス , pp.100-131 (1976.12.13) .

[31] Hikita, T. and Ishihata, K. : PASCAL 8000 Reference Manual, Department of Information Science, University of Tokyo (1976) .

[32] Gordon Cox and Jeffrey Tobias : PASCAL 8000 IBM 360 / 370 Version for OS and VS environments Reference Manual Version 1.2, Australian Atomic Energy Commission Australia (1978) .

[33] 石畑, 疋田, 安村, 石田 : PASCAL コンパイラの開発, 東京大学計算機センター年報, Vol.6, pp.122-134 (1976) .

[34] Kiyoshi ISHIHATA and Teruo HIKITA : Bootstrapping PASCAL Using a Trunk, Technical Report 76-04 , Department of Information Science, Faculty of Science, University of Tokyo (1976) .

[35] 白濱律雄, 前野年紀 : Pascal による Pascal のための実行時支援システム, 情報処理学会第23回プログラミングシンポジウム, pp.11-20 (1982) .

[36] A.V. エイホ, J.D. ウルマン著, 土居範久訳 : コンパイラ, 培風館 (1986) .

[37] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman : Compilers / Principles, Techniques, and Tools , Addison Wesley Publishing Company (1986) .

◎コントロールフロー解析

[38] Frances E. Allen : Control Flow Analysis, *SIGPLAN Notices*, Vol.5 No.7, pp.1-19 (1970) .

[39] F. E. Allen, J. Cocke : GRAPH-THEORETIC CONSTRUCTS FOR PROGRAM CONTROL FLOW ANALYSIS, RC-3923(#17789) IBM Thomas J. Watson Research Center, Yorktown Heights, New York, pp.1-65 (1972) .

[40] Robert Tarjan : Finding Dominators in Directed Graphs, *SIAM J. Computing*, Vol.3, No.1,pp.62-89 (1974) .

[41] Thomas Lengauer, Robert Endre Tarjan : A Fast Algorithm for Finding Dominators in a Flowgraph, *ACM Transactions on Programming Languages and System*, Vol.1, No.1,pp.121-141 (1979) .

◎Alias 解析

[42] John P. Banning : An Efficient Way to Find the Side Effects of Procedure Calls and the Aliases of Variables, *Sixth Annual ACM Symposium on Principles of Programming Languages*, pp.29-41 (1979) .

[43] William E. Weihl : Interprocedural Data Flow Analysis in the Presence of Pointers, Procudure Variables, and Label Variables, *Seventh Annual ACM Symposium on Principles of Programming Languages*, pp.83-94 (1980) .

◎ループ検出

[44] Linore Cleveland and Yoshihiro Akiyama : Detecting loop structure in assembly code, *AFIPS Conference Proceedings, Vol.55 1986 National Computer Conference*, pp.259-265 (1986) .

◎大域的データフロー解析

[45] M. S. Hecht, J. D. Ullman : A Simple Algorithm for Global Data Flow Analysis Programs, *SIAM J. Computing*, Vol.4, pp.519-532 (1975) .

◎中間コード

[46] M. Nagl : Application of Graph Rewriting to Optimization and Parallelization of Programs,Computing (Suppl. 3) , No. 3, pp.105-124 (1980).

◎その他

[47] 大久保英嗣, 津田孝夫 : 順序保存ハッシュ関数による高速ジョインアルゴリズム, 情報処理学会論文誌, Vol.25, No.1, pp.59-65 (1984).

[48] 石浦菜岐佐, 高木直史, 矢島脩三 : ベクトル計算機上でのソーティング, 情報処理学会論文誌, Vol.29, No.4, pp.378-385 (1988).

[49] 津田孝夫 : 岩波講座ソフトウェア科学9 数値処理プログラミング, 岩波書店 (1988) .

[50] R. W. Hockney , C. R. Jesshope 共著 ; 奥川峻史 , 黒住祥祐共訳 : 並列計算機, 共立出版 (1984) .

[51] 島崎眞昭 : 数値計算におけるベンチマーク , 情報処理 , Vol. 31 , No. 3 , pp.313-320 (1990) .

論文一覧

◎本研究に関する主要論文

[1] 津田孝夫, 国枝義敏, 二宮正和, 栗屋 透: ループ間にまたがるデータ参照関係をもつ多重ループの自動ベクトル化, 情報処理学会論文誌, Vol. 26, No. 3, pp. 536-544 (1985).

[2] T.Tsuda, Y.Kunieda : Mechanical Vectorization of Multiply Nested DO Loops by Vector Indirect Addressing, *Proceedings of the IFIP 10th World Computer Congress*, Elsevier Science Publishers B.V. (North-Holland), pp. 785-790 (Sep 1986).

[3] Takao Tsuda, Yoshitoshi Kunieda : V-Pascal : an Automatic Vectorizing Compiler for Pascal with No Language Extensions, *Proceedings of SUPERCOMPUTING '88*, November 14-18, 1988, Orlando, Florida (IEEE Compuer Society), pp.182-189 (1988).

[4] 國枝義敏, 津田孝夫: 自動ベクトル化コンパイラのための制御関係解析法, 情報処理学会論文誌, Vol. 30, No. 9, pp. 1164-1174 (1989年9月).

[5] Takao Tsuda, Yoshitoshi Kunieda, Piyaporn Atipas : Automatic Vectorization of Character String Manipulations And Relational Operations in Pascal, *Proceedings of SUPERCOMPUTING '89*, November 13-17, 1989, Reno, Nevada (ACM SIGARCH), pp.187-196 (1989).

[6] Takao Tsuda, Yoshitoshi Kunieda : V-Pascal : An Automatic Vectorizing Compiler for Pascal with No Language Extensions, *The Journal of SUPERCOMPUTING*, Kluwer Academic Publishers, Vol. 4, No. 3, pp.251-275 (1990).

[7] 國枝義敏, 津田孝夫: 多重ループにわたる配列データ依存関係解析法, 情報処理学会論文誌 (投稿中).

◎本研究に関する口頭発表

[1] 津田孝夫, 国枝義敏, 二宮正和, 栗屋透: ソースプログラムの自動変換によるベクトル化率の向上, 1. 任意多重ループの一重化など, 情報処理学会第29回全国大会, 4B-8, pp. 153-154 (1984年9月).

[2] 津田孝夫, 国枝義敏, 二宮正和, 栗屋透: ソースプログラムの自動変換によるベクトル化率の向上, 2. ベクトル化プリプロセッサの開発, 情報処理学会第29回全国大会論文集, 4B-9, pp. 155-156 (1984年9月).

[3] 国枝義敏: ベクトル計算機に課せられる課題－マンマシン・インターフェース: 自動ベクトル化技術とチューンアップ作業－, 第3回ベクトル計算機応用シンポジウム－ベクトル計算機による超高速計算－論文集, 京都大学大型計算機センター, pp. 134-143 (1987年3月).

[4] 津田孝夫, 国枝義敏: 自動ベクトル化 PASCAL コンパイラ V-PASCAL, スーパーコンピューティングシンポジウム論文集, 京都大学大型計算機センター, 共催文部省科学研究費補助金総合研究(A)「スーパーコンピュータの高度利用に関する総合的研究」班, pp. 42-50 (1989年3月).

[5] 津田孝夫, 国枝義敏, アティパート・ピヤポーン: 言語 Pascal で記述された文字列処理と関係演算の自動ベクトル化, スーパーコンピューティングシンポジウム論文集, 京都大学大型計算機センター, 共催文部省科学研究費補助金総合研究(A)「スーパーコンピュータの高度利用に関する総合的研究」班, pp. 69-78 (1990年3月).

◎その他の論文/口頭発表

[1] 国枝義敏, 島崎眞昭, 津田孝夫: PASCAL をベースとした日本語データ処理及び漢字を用いるプログラミング, 情報処理学会第21回全国大会論文集, 5I-3, pp.1015-1016 (1980年5月).

[2] 国枝義敏, 島崎眞昭, 津田孝夫: 日本語データ処理用言語PLAK, 情報処理学会第22回全国大会論文集, 3M-9, pp.915-916 (1981年3月).

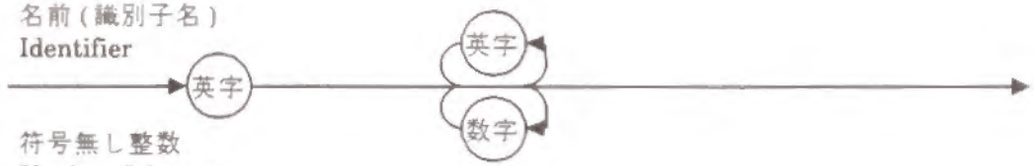
[3] 国枝義敏, 島崎眞昭, 津田孝夫: 言語PLAK のランタイムライブラリの日本語データ処理機能, 情報処理学会第23回全国大会論文集, 1H-3, pp.219-220 (1981年10月).

[4] 国枝義敏, 島崎眞昭, 津田孝夫：日本語データ処理用言語PLAKの処理システム, 電子通信学会電子計算機研究会報告, EC81-81, pp.59-72 (1982年3月).

[5] M.Shimasaki, Y.Kunieda, T.Tsuda : APPLICATIONS OF MODERN PROGRAMMING LANGUAGE CONCEPT TO TEXT PROCESSING WITH A LARGE CHARACTER SET *PLAK* : Programming Language with Abstract data type for Kanji processing, *Proceedings of the IFIP 9th World Computer Congress*, Elsevier Science Publishers B.V. (North-Holland), pp. 107-112 (Sep 1983).

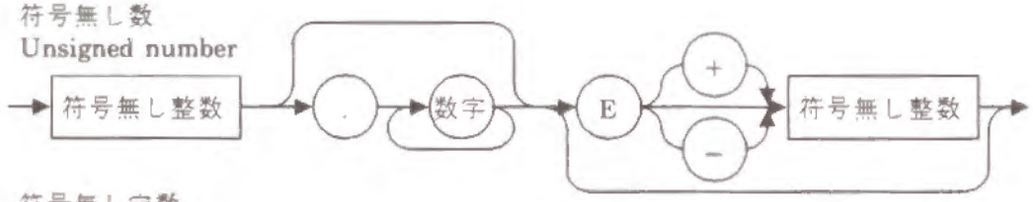
付録
Pascal構文図

名前（識別子名）
Identifier

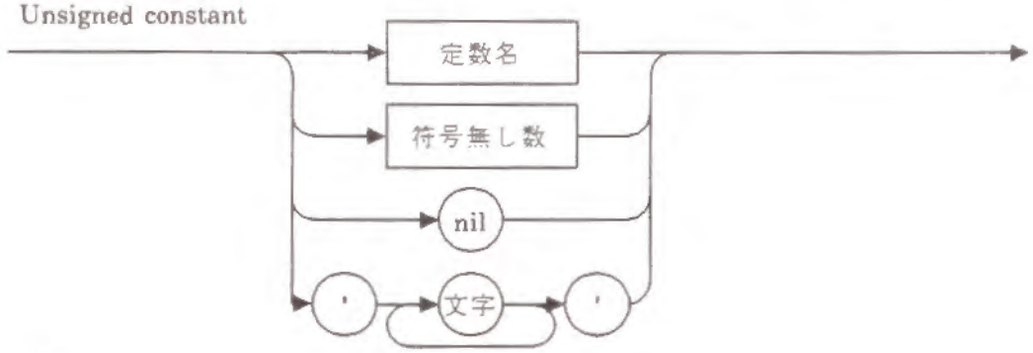


符号無し整数
Unsigned integer

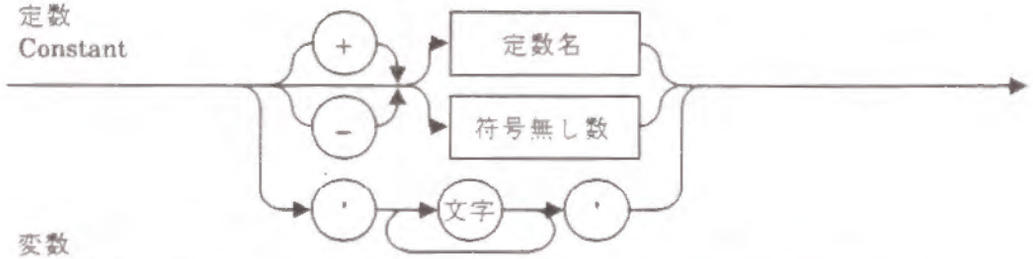
符号無し数
Unsigned number



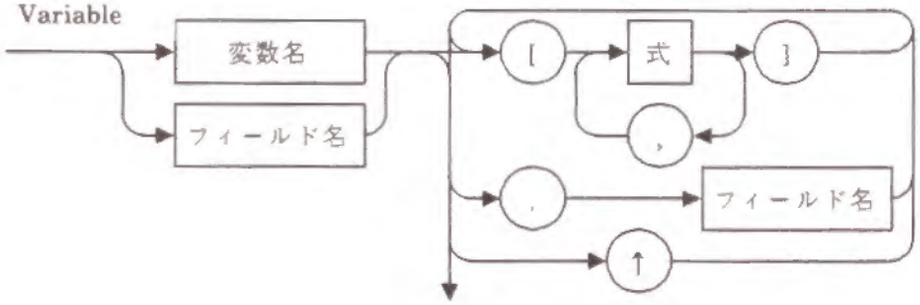
符号無し定数
Unsigned constant



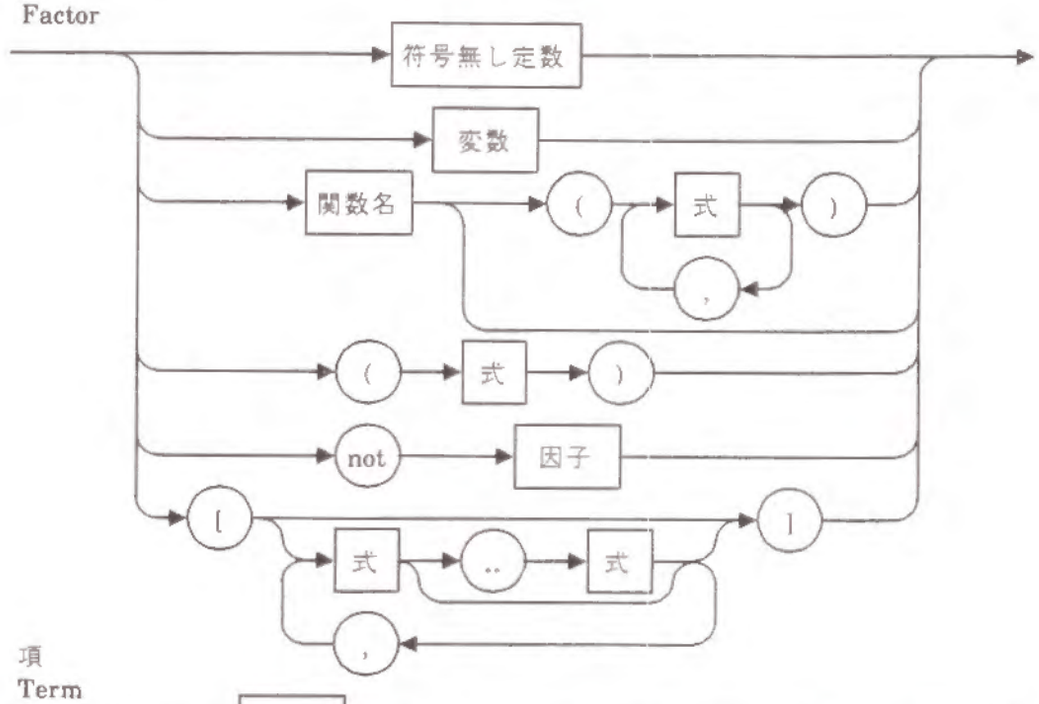
定数
Constant



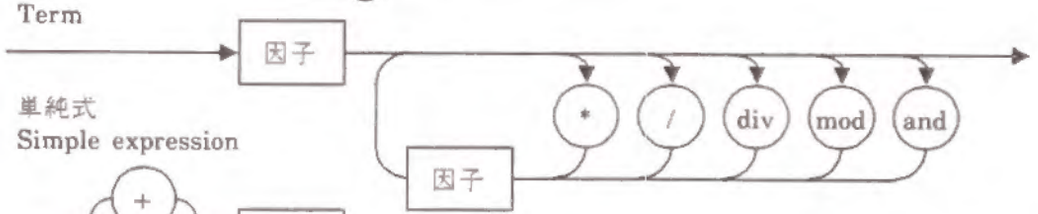
変数
Variable



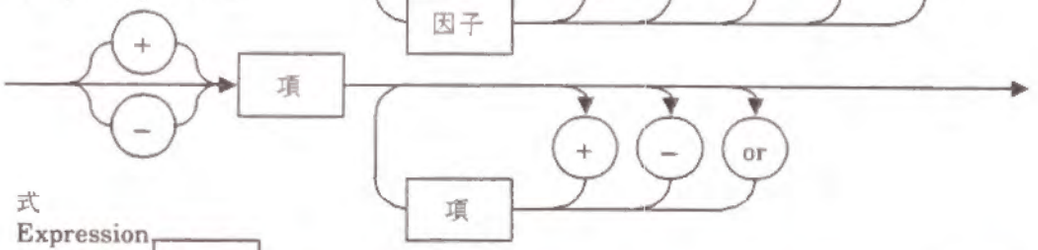
因子
Factor



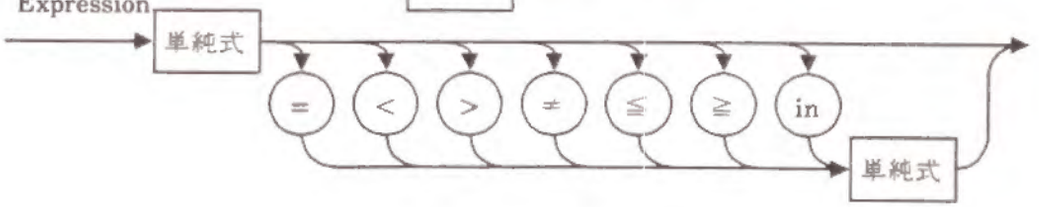
項
Term



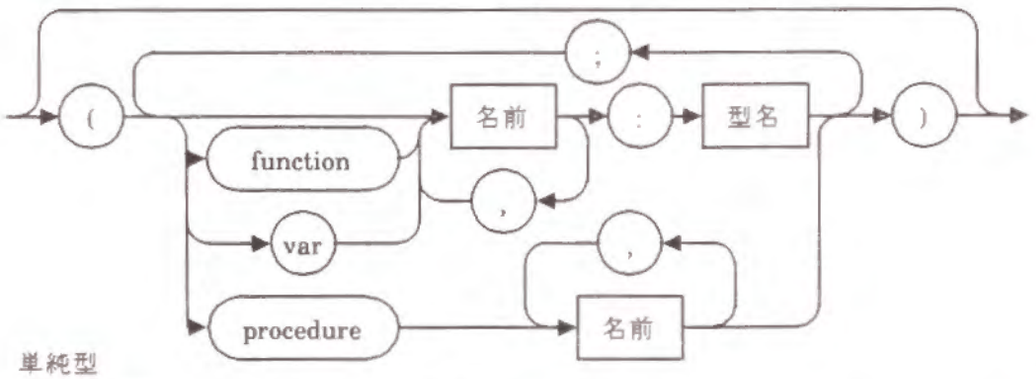
単純式
Simple expression



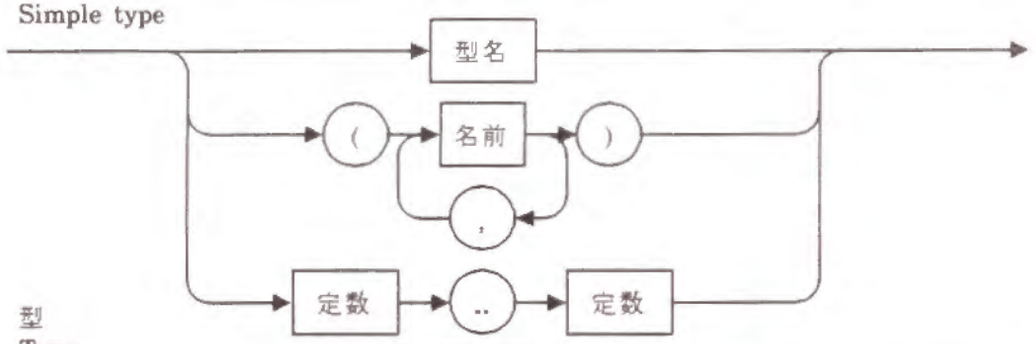
式
Expression



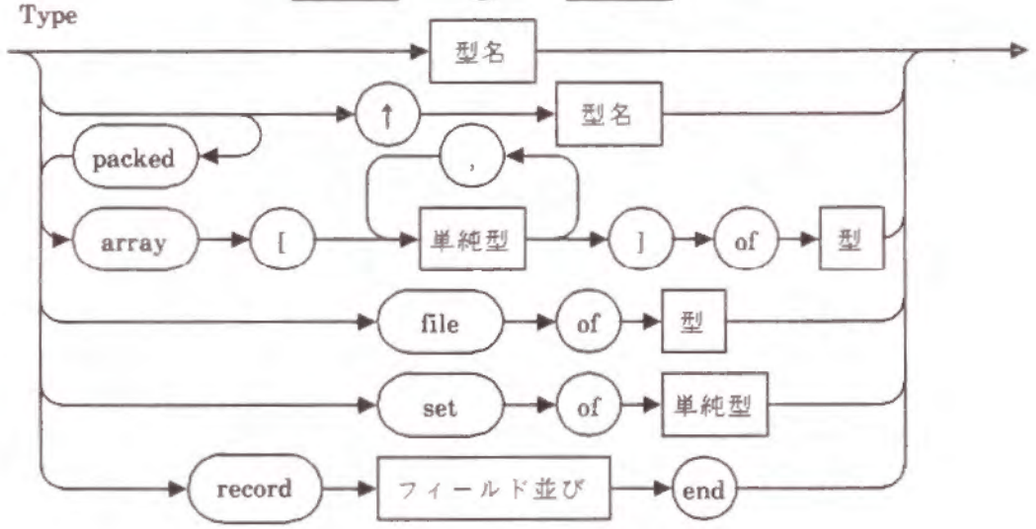
パラメータ並び
Parameter list



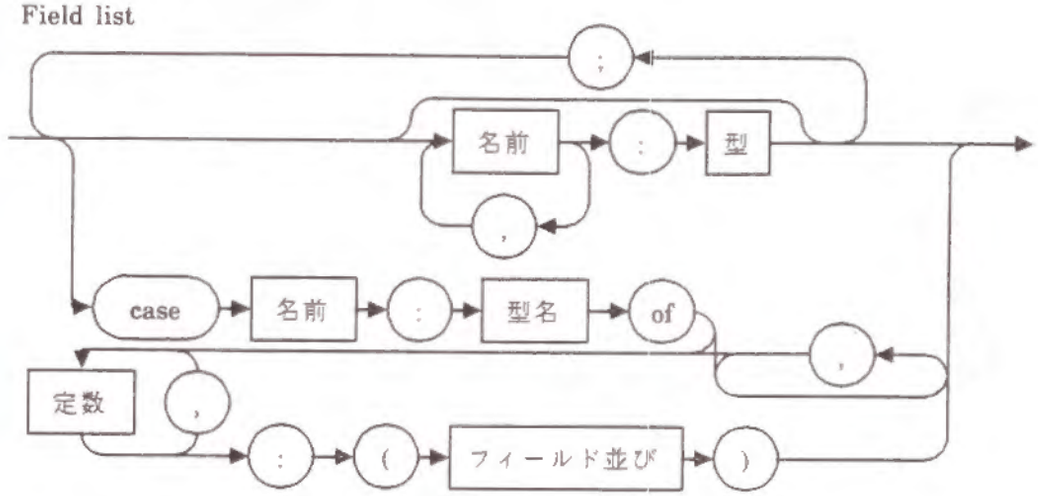
単純型
Simple type



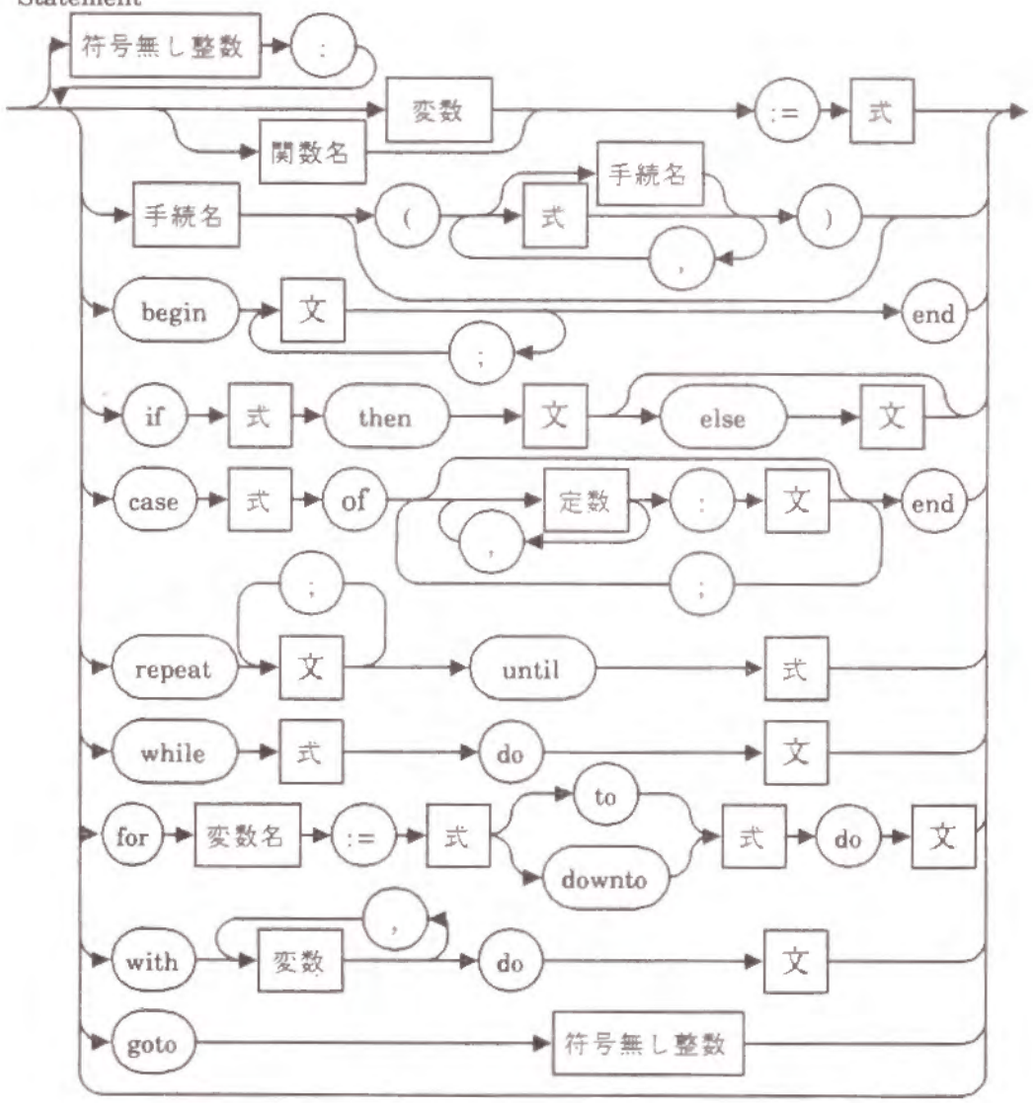
型
Type



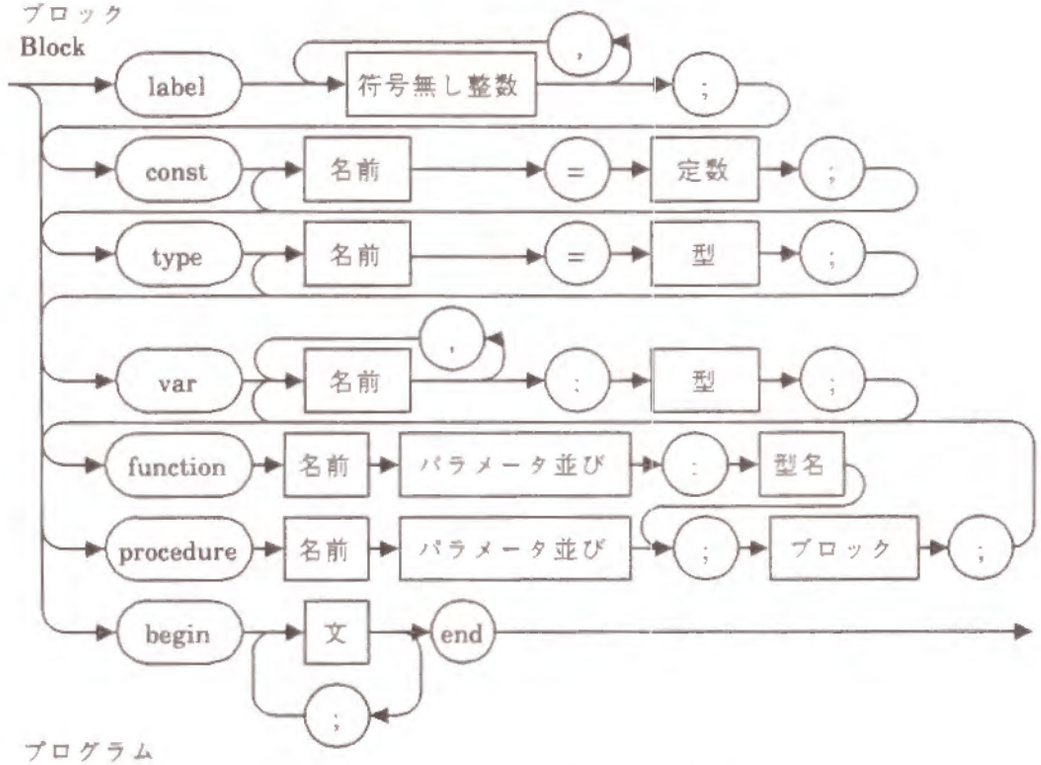
フィールド並び
Field list



文
Statement



ブロック
Block



プログラム
Program