属性処理機能に基づく情報処理技術とその応用に関する研究

井田昌之

はしがき

はしがき

事務情報処理や人工知能などにおける多くのソフトウェアにおいて，データのもつ論理的・物理的な属性の定義とこれに対する処理の整合性の保持はプログラマに任せられている。

事務情報処理の分野においては，従来よりCOBOL，PL/I等の汎用高級言語を中心にプログラミングが行われてきた。しかし，それらの高級言語を用いる場合，ファイルの入出力を正しく行うための記述，ファイル中のレコードの処理の記述，データの位置・長さ・型その他の属性記述は相互に関連しあっており，それらを正しく関連させるのはプログラマの責任となっていた。このため，それらの整合性を考慮しながら，信頼性が高く，実行効率がよく，また保守がしやすいようにプログラムする繁雑な作業がプログラマに強いられてきた。

その結果，事務情報処理ではプログラマに対する量的な負担が大きくなりがちで，表現形式の取違いによる誤変換，プログラム内で指定したデータ構造と実際のファイル構造の不一致，配列の添字の値の規定範囲からの逸脱，項目内の値の確定時機の誤認識などのかたちでデータに起因するトラブルが生じている。

そこで，ファイルの動的な処理を含む大局構造と，データの論理的・物理的属性の記述・処理と，レコード内の項目の処理の3つの機能にソフトウェアを分割して考え，生産効率と正確さを両立させるシステム構成法を案出した。この構成法を用いることにより，一般のプログラマは必要な項目に対する処理を記述するだけでよくなり，データの構造や属性，ファイルの入出力を正確にかつ，効率よく行う方法などを意識する必要がなくなる。また，ファイルの入出力を中心とする大局構造は，事務情報処理の特質から類型化できるので，少類の構造を熟練したプログラマが最も機械の能力を引き出せるように記述し管理することができる。これにより実行効率の

低下が防がれる。データの構造や属性を統一して管理することにより，処理の記述時と実行時を通してデータの妥当性を保証する機構も実現することができる。データの属性を記述子で表わすと，大局構造・データの属性記述・レコード内項目処理の3つの部分に分かれたプログラムを有機的に結合統合するシステムを作り上げることが可能となる。記述子を中心としてプログラムの生成が進められる。

こうした概念に基づいてFAST1システムを作成し，ある企業の協力を得て昭和49年から昭和51年にかけて，有効性を確認した。

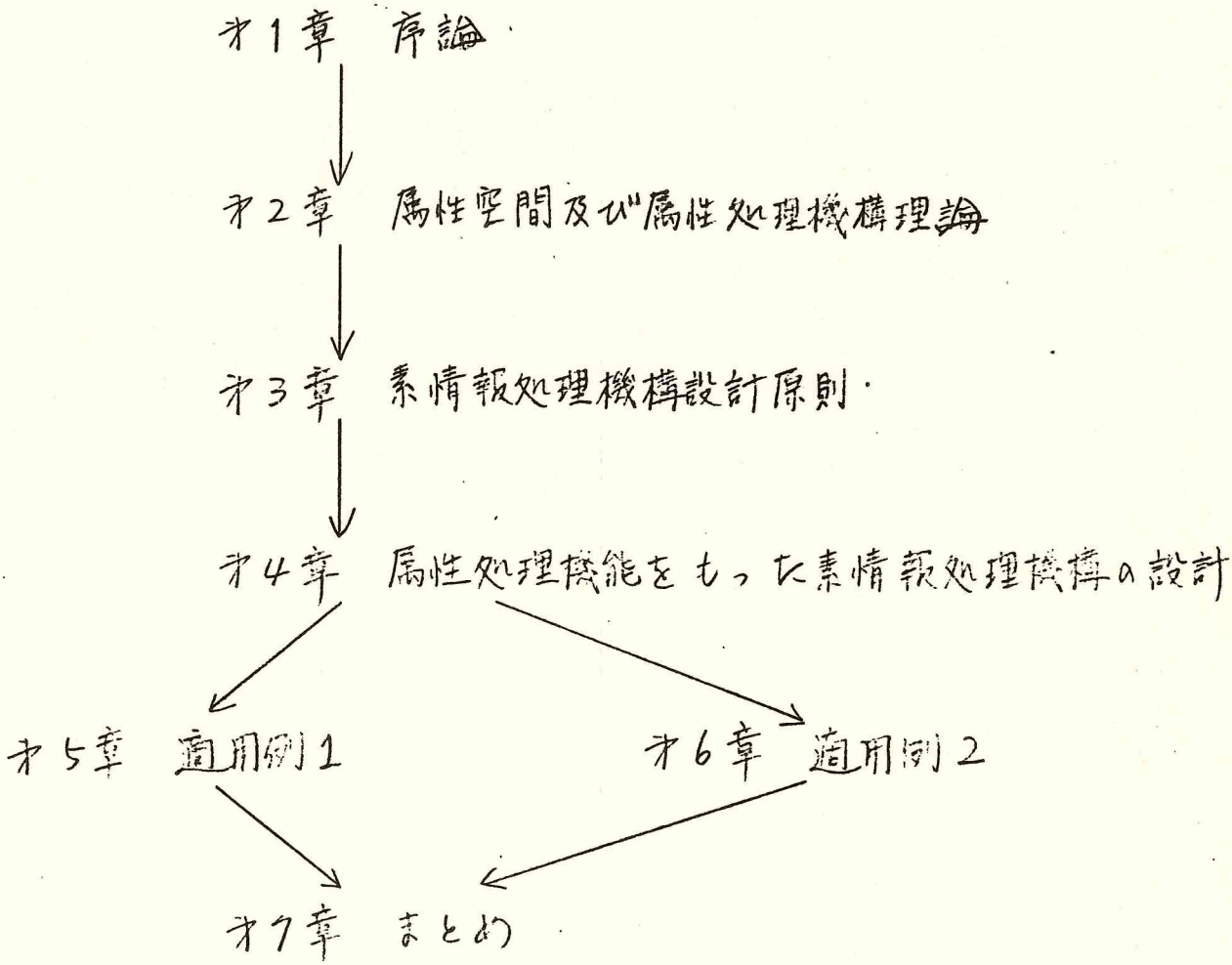
一方，数式処理等の人工知能分野においてもデータの属性処理に関しては同様の問題点が存在する。特に，知識辞書はデータの構造や属性を動的に追加・変更できることが必要であって，辞書の内容は不均質で疎なものとなる場合が多く，誤まった処理によりデータの属性や内容が失われた場合には，内容の修復は困難である。従って事務情報処理と比較して生産効率よりも正確さの保証と属性処理機能の高度さが必要となる。

そこで，データ名・属性名・属性値の三つ組を連想子として用いたデータ構造の自動管理機構と，データ属性の任意性と柔軟性を保った高速参照機構を案出した。連想子はハッシング技法に基づいて生成・参照される。記号処理言語Lispを拡張した形のシステムを設計することにより，目的を達成することができる。また会話型の人工知能処理用専用機を用いると，より効果を高められる。そこでLisp専用機ALPS/Eを作成し，この概念の有効性を確認した。その後の検討で，以上の考え方は1962年に発表されたが1977年になって我国に広く紹介された情報代数の考え方とほぼ等価であることが判明した。

従って，本論文は事務情報処理と人工知能に共通するデータ空間の枠組を定め，それに基づく基本機構の設計原則を演繹的に導くことを目的としている。この目的を達成するために，データの表現形式に依存しないデータ構造に関する共通理論として，抽象データ空間及び半識別属性空間を定義し，そこでのデータ操作の基本要素

を定め，空間を均質性により2つに分割し，その各々についてモデルから具体的な処理機構の設計と実施例を導いた。これらにより，ソフトウェアの自動生成や開発支援のための基本ソフトウェアに対して新しい設計法の確立をねらっている。

本論文の構成を次図に示す。



本論文は7章より構成されている。

第1章は序論であり，研究の対象・目的・方法がまとめられている。

第2章では半識別属性空間及び抽象データ空間を定義し，それにおける2段階のデータ抽象化及びそれに対する抽象手続き概念，各空間における基本要素である素情報及び抽象事象の定義及びそれらの関連を理論づけている。また，適用分野の側面からデータ空間を“均質性の違い”により2種類に類別し，均質データ空間と不均質データ空間を導いている。

次に，不均質なデータの抽象化に適した連想子とそれを中心とする処理機構論，データの抽象化に適した記述子とそれを中心とする処理機構論を展開している。特に記述子を中心とした処理機構は抽象事象概念と適合でき，この点から抽象手続きの構成論をまとめている。

第3章では，第2章で述べた理論を電子計算機上に実現するにあたって必要な設計原則について展開されている。システム自身の抽象化のために，データ独立性，機械独立性，応用独立性のある設計の必要性が述べられる。次にこれらの性質を満たすシステム構成法として，均質データの処理のための三層分化プログラム構成法が述べられる。三層分化プログラム構成法は，使用者の処理手続の記述から，データ定義と機械依存の共通手続（原形手続）を分離する概念に基づいた，機能の分化とプログラム合成理論である。最後に，不均質データの処理のための原則として，Reliability，Availability，Simplicity，Helpfulness，Adaptabilityが述べられる。

第4章では基本要素である素情報に対する処理機構の設計を行っている。属性の管理・格納手法として2つの方法を示している。それらの方法では基本要素は異なり，各々記述子・連想子とよぶ。記述子は固定された基本属性よりなる情報の属性管理に適している。

連想子は，動的に変化する属性をもつ情報の属性値処理に適している。

第5章では　事務情報処理支援システムとしての構成を例として第4章で述べた処理機構の1つである記述子による，属性処理の具体的な手法について述べている。包括的なシステムとするためには，プログラミング支援機能との緊密な連係が必要である。この点についてもまとめられている。また，3章で述べた三層分化プログラム構成法に基づく，プログラム機能の分化と合成の具体例を示している。言語の設計・翻訳手法・プログラム生成手法及び管理システムの実例と実施例の評価がまとめられている。

第6章では　記号処理システムとしての構成を例として第4章で述べた管理機構の1つである，連想子による属性処理を具体的に行う手法について述べている。ここで示す動的な属性処理機構が有効であることを示す例として，事実検索での手法，人工知能の一分野である数式処理システムにおける適用，動的計画法などで使われるbreadth-first型探索における適用について述べている。

また　これらをシステムとして動作させるのに必要な支援機構について，さらにコンパクトなシステムを構成するハードウェアについても，その設計原理と実現手法がまとめられている。

第7章では　本論文において得られた成果について要約し，まとめている。

目　　次

第1章　序論

1.1 ファイル処理とデータの属性

事務処理分野における情報処理の問題の多くは，ファイル形式に関する研究において議論されてきた。一般に広く用いられているファイル形式は，順アクセス構成と乱アクセス構成にわけられる。これらの構成法は用途に応じて得失も異なる。その特長を表1-1に示す。

順アクセス構成の長所は，ストレージコストが比較的小さくて済む点である。乱アクセス構成の長所はこれに比べて構造の柔軟性があり，また各レコードをより高速に参照できる点である。

一方，事務情報処理分野以外でも，情報の蓄積・参照機能に関する研究は行われている。

自然言語処理，数式処理または推論処理などの人工知能分野や，情報検索分野などにおけるデータ処理は，知識辞書を電子計算機上に構築したもので，そこへの情報の登録，あるいは辞書に対する情報の参照が基本機構となっている。したがって情報の蓄積構造は重要な論点となっており，ここにデータベースと呼ばれる事務処理のための集中化ファイル機構との共通点がみられる。すなわち，ストレージコストよりも，構造の柔軟性や高速参照能力を追求した機

表1-1 事務処理ファイルの形式

	構造の柔軟性	ストレージコスト	各レコードへのアクセスタイム比
順アクセス構成	小	小	$O(n)$
乱アクセス構成	大	大	$O(1)$

構の設計・開発が求められている。

情報処理関連技術の研究のうち，データ構造に着目したものの多くは，この構造の柔軟性と参照の高速性に関連するものである。たとえば，データベースはこの2つの性質を満足できる集中化ファイルシステムであるが，それ自身大規模なソフトウェアであり，基本的には大規模かつ長期的な事務システムを対象としており，小規模ないしは実験的な応用に利用することは困難である。情報検索のための組合せ論的ファイリングの研究は，[YAM75]などに代表されるが，想定される基本質問の論理演算で記述できる複合質問に対して，最も高速に解答できるような基本質問の群からなる索引表の並べ方に関する研究である。参照の高速性を追求しており，構造の柔軟性等を考慮していない。このためデータの動的な処理を前提とする一般の情報処理に適用することはできない。

また，人工知能関連分野において広く利用されているLispは，リンクリストを中心とする非常に簡単な構造をもつ言語システムでデータ間の関係を自由に定義・変更することができる特長がある。しかし，本来のLisp[MCC60]はストレージコストと参照の高速性については考慮されていないために，その実現にあたって各々いくつかの改良・拡張が行われている。

構造の柔軟性と参照の高速性は良い情報構造にするための基本的な条件となりうるが，これらにデータ自身に対する性質として項目の識別性と情報の任意性を加えることにより，電子計算機での情報の蓄積・参照に関する尺度をよりはっきりとすることができる。

項目の識別性は，順アクセス構成による事務処理ファイルにおいて従来より問題になっていた。すなわち，プログラム中で想定したファイルの構造と，実際の構造との一致を確認できないため，誤操作が生じることがある。また，各項目の位置・型・長さなどの表現形式や，配列の場合の添字のとりうる範囲や各々の異常値の存在を，プログラム作成時と実行時を通して管理することができない場合が多かった。

また，情報の任意性はデータの表現形式や内容について不必要な制約をうけないようにするために必要となる。これらの性質は，人工知能のための知識辞書を構成する場合には不可欠である。というのは，知識辞書の場合には保持する情報の構造や性質の動的な変更が中心的な作業であることと，誤操作により値や性質が失なわれた場合，修復がほとんど不可能であることによる。

このようにして，人工知能と事務情報処理における基本的な共通点として個々のデータの性質について注目し，それに対する問題点として両分野における議論を共通化することができる。

一般にこれらのデータの性質は属性と呼ばれる。属性概念をもとにデータの意味を厳密化して考えることができる。次のような概念が一般的に想定されている。

（1）データは1つ以上の属性をもつ
（2）各属性は値をもつ
（3）各データの各属性はただ1つの値をもつ。これを属性値とよぶ

属性の中で表現形式に関するものを整理すると次のようになる。

基本型（Basic Data Type）は次の5つに分けられる。

1. 整数型　2. 実数型　3. 論理型　4. 文字型
5. 名札型

各型には機械固有の副型がつけられうる。たとえばFACOM系では整数型は，二進固定小数点型，パック十進型，ゾーン十進型などに分けられる。

数式処理言語処理系においては整数型は，ショート整数型，整数型，無限桁整数型などに分けられる。

構造型は［HOA72］などにまとめられ，［WIRT71］に実現例があげられている。次の7つがある。

1. 列挙型（enumeration type）
2. 部分範囲型（subrange type）
3. 配列型

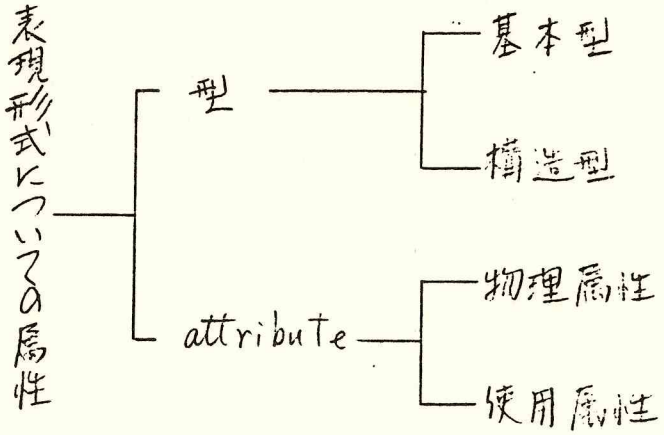
4. 集合型
5. 直積型
6. 再帰型
7. ファイル型

attributeは 2つの項目からなる。

1. 物理属性　2. 使用属性

物理属性には，(1) データ空間中の位置 (2) 大きさ(ビット長)の2つの指標がある。

使用属性には，(1) アクセス権 (2) usage の2つの指標がある。アクセス権とは その情報の参照方法として読み出しのみ許される(read only)，あるいは 書き込み可能であるという指定やその情報の生死(active or dead 有効か否か)についての指定である。usageとは 情報の型・物理属性に対する許容条件である。定義された型以外の型で参照することを許すのか否か，あるいはその情報の一部分だけ参照することなどを許すのか否か，などである。まとめていうならば 処理手続での参照の仕方の規定である。



データを構成する基本要素を本論文では素情報と定義する。情報処理を行うとき，対象となるデータ群はいくつかの値を持つ素情報の集まりであると考える。事務処理的な概念との対応をとるならば，「データ群」は「ファイル」に相当し，「素情報」は「レコー

ド」に相当する。人工知能的な用語との対応をとるならば，「データ群」は「リスト」に相当し，「素情報」は「アトム」もしくは素記号に相当する。

そこで，素情報には複数の属性と属性値があると考える。

このような概念を持つことにより，従来個別分野において考えられてきた情報の蓄積・参照について統一した議論を行うことができる。

形式的な定義を以下に示す。

$$S \equiv \mathbb{P}(\mathbb{D})$$

S：ソフトウェア
$\mathbb{P}$：処理手続
$\mathbb{D}$：対象データ群

$$\mathbb{D} = (E_1, E_2, \cdots, E_n)$$
$$E_i = ((p_1, v_{i1}), \cdots, (p_j, v_{ij}), \cdots, (p_m, v_{im}))$$

E_i：素情報（レコード）
p_j：属性
v_{ij}：属性値

1.2 情報処理技術の体系化とソフトウェア開発支援システム

ところで，近年ソフトウェア工学という言葉が定着しはじめている。これは従来手工業的な生産形態をとっていたソフトウェアに対して工学的なメスを入れるものであり，本研究もその中に入る。

ソフトウェアは多くの場合，反復して利用される。また利用が進むに従って機能の拡張・変更が要求される場合が多い。このためソフトウェアの生産現場においては，新規作成のみでなく，維持管理にかなりの比重が置かれることとなる。しかし，一旦作成されたソフトウェアの機能の修正や追加・変更には相当な工数を要することが一般に知られている。手工業的な開発形態では，以前に正しく処理されていたものが別の部分の修正により改悪され，誤動作を生じることさえよく見られる。また，現在の多くのソフトウェア開発体制ではこうしたことを自動的に検出し，改悪を未然に防ぐことはできず，事態を一層深刻なものにしている。そこでソフトウェア生産・管理の能率化のために迅速性・低コスト性に加え，正確性を保証できる手法が必要となっている。

これらのソフトウェア工学的な研究は情報処理技術の体系化とその結果構築できるソフトウェア開発支援システムに対する議論としてまとめることができる。ソフトウェアに関する問題点はソフトウェア自身の構造だけでなく，マンマシンインタフェイス，生産組織論に関係することも多いので多面的な研究が必要である。

たとえば，ソフトウェアは作成及び実行という2つの段階において各々問題点をもっている。構造化プログラミングなどやモジュラーデザインなどは作成時における問題に対する研究である。また機械の構造との関連などは　実行時における問題である。データに関連する問題は，作成時及び実行時の両者において統一して解決されねばならない問題の一つである。

また一方で，ソフトウェアは技術の進歩と応用分野の拡大の2

者の間を埋めるために本数の増大と多様化が必要となる問題点がある。

ハードウェアに対応して作成するオペレーティングシステム等の基本ソフトウェアにおいても こうしたことが生じるが，いまだに最も低級な機械語（アセンブラ）を用いた作成・修正に追われている。この現状については［Ida 79B］に述べたので省略する。

ソフトウェア開発環境について考慮されねばならない問題点は次のようなものである。

①互換性の問題

ソフトウェアには一般に機種依存性があり，他機種にそのままのせることは困難である。また対象となるデータの蓄積も機種依存性がある場合が多い。

②陳腐化の問題

自社内で充分機能しているシステムがあっても，社会的要請，商業的要請から変更を余儀なくされることも時々見られる。

③人事管理上の問題

日本の企業においては配置転換はつきものであり，また必要である。通常の手工業的なプログラミングを行う場合，それらの作成者には専門の技術者が必要であり，他部門との人事交流を行なわせることは難しい。そのためEDP部門の技術者の昇進・給与問題労働協約との関係で生じる残業時間の制限などの問題が生じる。

また図式的に考えられているオペレータ ⟶ プログラマ ⟶ シニアプログラマ ⟶ システムアナリスト ⟶ デザイナーという職務階層は，平社員 ⟶ 係長 ⟶ 課長 ⟶ 部長というような通常の事務部門で考えられる階層とは 全く異なったものである。オペレータ，プログラマ，システムアナリストなどに要求される能力と知識は，かなり異なった分野に属している。

④プログラマ要員確保の問題

SHAREの調査によると1975～85年の間のソフトウェア需要は毎年21～23%の割合で伸びるが，ソフトウェア生産力（労働力）

は毎年11.5～17%しか伸びず，両者の差が広がるであろうことが予測されており，プログラマの絶対数が不足している。

これらの諸問題の根本的な解決のために，ソフトウェアの生産は設計により多く比重をかけるべきだという主張が増え始め，これらが現在の要求工学ないしは要求仕様技術を形成している。

目に見える実際のコーディングのかげで，設計そして最後のテストの重要性は次第に認識されており，設計40%，コーディング20%，テスト40%という工数配分の考え方が最近いわれている。Boehmは［BOE 74］においてほぼそれに等しい値を実測値から出している。

コーディングは電子計算機の動作を順序だてて記述する作業であるため，かなりの緻密さが要求される。またその緻密さは明確な形で数量化できる。しかしソフトウェアの設計作業を具体的な形で行うための支援手法及びシステムがないため，設計時の誤りはコーディング時において発見されることが多い。設計誤りもしばしばコーディング作業者の誤りとして扱われてしまうことも多々見られる。

コーディングの支援のためには各種の言語及び言語支援機能が，テスト段階のためには各種のデバッグツール・手法が長年に渡って開発され，研究されてきた。しかし設計段階は現在ではほとんど人力にまかせられているといえる。設計とコーディングの間を円滑につなぐ手法が開発されれば，ソフトウェア開発の問題点もかなり解決することができる。現在，コーディングに直接結びつけられるような設計記述ツール（［ROB 77］など）が発表されはじめているが，全体的な処理効率の向上に役立つまでに至っていない。

設計のしやすい生産体系の研究や，
ソフトウェアのもう一つの側面である対象データの処理手法・蓄積手法の研究

が現在の代替ゴールとなるべきである。

1.3 従来の研究と問題点

1.3.1 事務ファイル処理技術研究

関連する研究としては，CODASYL による情報代数理論[COD62]，Lefkovitz によるオンラインファイル構造論[LEH69]，Codd による関係データベース基礎理論[COD70]，Hsiao&Harary による一般ファイル理論[HSi 70]などに示される各種モデルに関する研究と，DATE あるいは Martin によるデータベースの実際的諸問題に対する survey 研究[DAT77]，[MAR75]があげられる。情報代数は発表・研究時期が最も早い部類に属し，現在では先駆的な位置づけがされている。本論文における理論も情報代数を背景としているので1.3.2に特記することとする。

Lefkovitz はこれを受け オンラインファイル構造論として集中化されたファイルの形式・アクセス手法についてまとめ，実際に利用されている手法について分析を行った。定量的な測定をいくつかの例に対して行い，評価を試みている。しかし，理論的な扱いがされていないため普遍性に乏しい。

Hsiao はグラフ理論の大家としても知られている F. Harary と共に現代においても有効に利用されている各種のファイル形態に対して理論的な解析を行い，一般ファイル理論をまとめている。しかし，これはあくまで一般化理論が目的であり，新規の技術を導入していない。

Codd はこれらの研究を受け，高速参照に注目した関係モデル（リレーショナルモデル）をまとめ，多数の著作・研究活動を通して，表形式の記憶を基本とするファイル構成を展開している。さらにその理論に基づくデータベースシステムを実際に作成させ，実現可能性を示している。また このCoddの考え方はADABASその他の商用データベースの設計において大きな影響を与えている。しかし

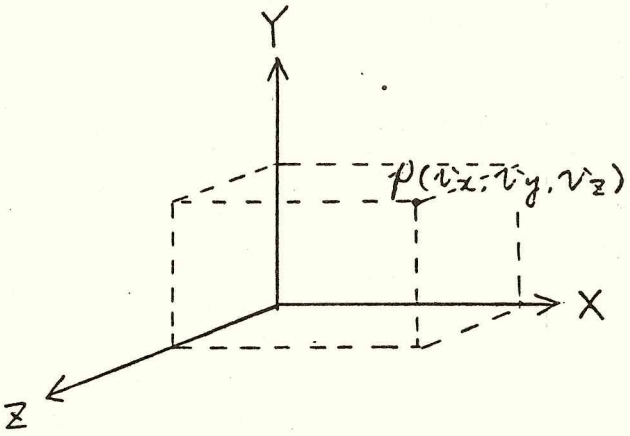
Coddの考え方は非常に大きな構想であり，必然的にその処理系も大型化し実用性においていくつかの問題点を含んでいる。Coddの所属するIBMで作成された関係モデルに基づくデータベースシステム System Rは商用化にいたらなかった。

また最近注目され始めている応用はソフトウェア生産支援に関するものである。はっきりとした成果は報告されていないが，その動向が注目されているものとしては，①SRIの設計用言語SPECIAL［ROB77］（Interlispで書かれており，O.S.の設計に使用されているという。）②SELECT［BOY75］（プログラムの検証のためのプログラムである。）③たとえばLPSのような形式的な記述からのプログラムの生成 ④たとえばEDA（in ladderシステム［SAG77］）のようなデータベースのエンドユーザーインターフェイスなどがある。

1.3.2 属性空間の概念と情報代数

情報代数はファイル処理に対する数学的な研究として先駆的な位置づけがされているが，ごく最近まで日本ではあまり知られていなかった。しかし，本論文における情報代数の概念の概略の紹介とそこでの問題点，そして本論文の2章以降で必要となるので，情報代数自身の定義について概要をまとめる。

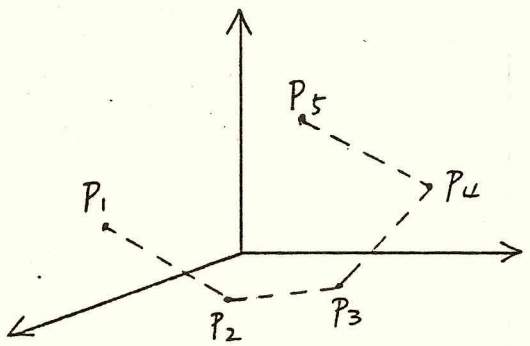
情報代数では電子計算機で処理しようとするデータを属性空間中の点もしくはその集合として扱っている。従来のファイル処理の概念と情報代数理論との対応を表1－2に示す。ファイルは領域に対応する。領域は属性空間の任意の部分集合である。ファイルに対して名前をつけ識別することは，その領域に対して領域関数として識別的な関数を用いることに対応する。ファイル中のレコードはこの領域に属するデータ点である。データ点には座標があり，座標集合は属性と考え，座標値は属性値と定義する。これによりレコード中の項目名，その項目の値の概念が対応する。また，ファイルのつきあわせその他のファイル間処理はバンドルという概念にまとめら



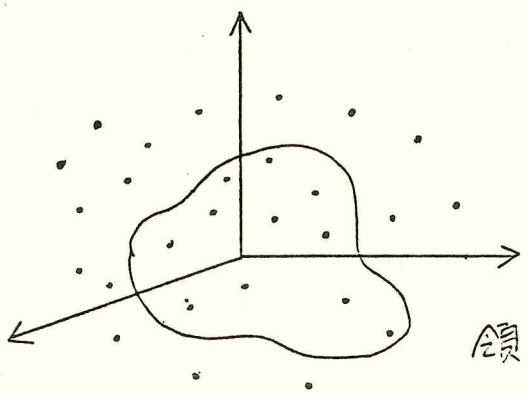
座標集合 $Q = (X, Y, Z)$
よりなる属性空間 P

$$P = V_X \times V_Y \times V_Z$$

ここで V_i は属性値集合
点 p：事象 e の データ点、



長さ5の線
$L = (P_1, P_2, P_3, P_4, P_5)$
(点の順序対)



領域（任意の部分集合）

図 1-1　属性空間の概念

れている。ファイル中の処理はグランプという概念にまとめられている。これらの基本概念を図1－1に示す。

表1－2　経験的事項と情報代数における用語との対応

経験的事項	情報代数における用語
レコード	事象(Entity)に対する値の集まり．属性空間中のデータ点。
レコード中の項目	属性値(事象の各属性はただ1つの値をもつ)
ファイル	領域(属性空間の任意の部分集合)
ファイル名	領域関数の値
ファイル間の操作とその出力ファイル	バンドル関数とバンドル
ファイル内の操作とその出力ファイル	グランプ関数とグランプ
現実世界のデータ	属性空間中の点、もしくはその集合
データ操作と対象データ	線と線関数(バンドル，グランプ，領域の概念を含む広い意味でのデータとその操作

情報代数は，経験的な概念を集合とその上での写像関係を用いて数学的に記述することを示した点で評価できる。

特にファイル間の処理，ファイル内の処理に対して明確な定義を与えた点，そしてこれらを包括する属性空間を導入し，それによりデータ処理の対象をすべて説明できることを示した意義が大きいと思われる。

しかし，情報代数は発表・研究時期が早く，当時の技術水準及び利用水準を越えた理論であったため，研究者自身もこの理論が一体どのような意味になるかわからなかったと述べている。したがってそこで形式化した利用形態は個別ファイルの逐次処理であるこ

と，単に枠組理論にとどまっており，実用手段との関連が述べられていないこと，属性というものに対して厳密な意味づけがされていないことなどの問題点があった。

ファイル処理概念についてはHsiaoらによって，一般ファイル理論〔HSIAO〕に発展をみている。属性空間の識別性についてはCoddによる関係データベース基礎理論に発展をみている。

そこで本論文では，基本となる属性とそれが構成する属性空間について拡張・改良を行っている。以下に情報代数の概要を示す。

〔情報代数の対象〕

情報代数は，事象（Entity），属性（Property），値（Value）という3つの概念の上に作られている。(1) 事象は1つ以上の属性をもつ。(2) 各属性は値をもつ。(3) 各事象の各属性はただ1つの値をもつ。これを属性値とよぶ。

〔属性値集合〕

属性値集合（V）は属性の値のとりうる範囲と種類を規定する。

未定義値Ωと欠落値θもまた，属性値集合の要素である。

〔座標集合〕

相異なる属性の有限集合を座標集合Qとよぶ。

〔属性空間〕

ある座標集合Qの属性空間Pとは直積，

$$P = V_1 \times V_2 \times V_3 \times \cdots \times V_n$$

である。

〔データ点〕

属性空間中の各点Pの座標は，

$$P = (a_1, a_2, \cdots, a_n)$$

$a_i \in V_i$：属性値集合

によって表わされる。

a_iがV_iから事象e_nに対して割当てられた値であるならば，Pはデータ点と呼ばれる。

〔識別属性空間〕

どのデータ点も2つ以上の事象を表わすことがない属性空間を識別属性空間という。

〔基本識別座標集合〕

Q，Q′を座標集合とし，Qは識別的であるが　Qに含まれるどのQ′も識別的でないならば，Qは基本識別座標集合であるという。

たとえば，（部番号，部内従業員番号）からなる座標集合がそれである。

〔線〕

属性空間Pから選出された点の順序対である。線の長さとは，線を構成する点の個数である。

長さnの線を

$$L = (P_1, P_2, \cdots, P_n)$$

と表わす。

〔線関数〕

線関数（FOL）は，線に対して一つの値を対応させる写像である。

〔順序つき線関数〕

順序つき線関数（OFOL）とは，その値の集合が，

非反射的　（a Ⓡ a が真でない）

反対称的　（a Ⓡ b ∧ b Ⓡ a）

推移的　（a Ⓡ b ∧ b Ⓡ C ならば a Ⓡ C）

を満たす演算Ⓡの定義できる集合に含まれるFOLをいう。

〔領域〕

領域とは　属性空間Pの任意の部分集合をいう。
領域には　合併・共通・補集合などの演算が定義できる。

〔領域関数〕

領域関数（FOA）とは　領域に対して一つの値を対応させる写像である。

〔領域集合〕

次数nの領域集合とは，n個の領域の順序対

$(A_1, A_2, \cdots, A_n)$

をいう。

〔バンドル〕

選択関数bにおける領域集合αのバンドル $B = B(b, \alpha)$ とは次のような線Lのすべてからなる集合のことである。

(1) $\alpha = (A_1, A_2, \cdots, A_n)$

(2) $L = (P_1, P_2, \cdots, P_n)$

P_i は A_i の点

Lはある関数 $b(L)$ = 真となるようなすべての線

〔バンドルの生成関数〕

選択関数bをバンドルBに対する　バンドルの生成関数とよぶ。

〔バンドル関数〕

バンドル関数(FOB)とは　一つのバンドルに対して　一つの領域を対応させる写像で，次の(a)，(b)，(c)を満たすものをいう。

(a) バンドル中の線と　領域中の点との間に多対1対応がある。

(b) 領域中の各点の各属性の値は，バンドル中の対応する線の線関数によって定義される。

(c) そのような線関数の値集合は，対応する属性値集合の部分集合である

FOBは，

$$F \equiv \begin{cases} g_1' = f_1 \\ g_2' = f_2 \\ \quad \vdots \\ g_k' = f_k \\ \quad \vdots \\ g_m' = f_m \end{cases}$$

として表わす。

g_k'はバンドル関数によって与えられる領域中の点に対するk番目の属性，mは属性の個数，g_{ij}は領域集合中のi番目の領域のj番目の属性，f_kはg_{ij} $(i=1,2,\cdots,n;\ j=1,2,\cdots,m)$の関数，nはバンドルを構成する線の長さとする。

〔バンドルにより生成される領域〕

バンドル関数Fによってバンドルにより与えられる領域Aは次のいずれかによって表わす。

$A = F(B)$

$A = F(b;\alpha)$　　b：バンドルの生成関数

$A = F(b; A_1, A_2, \cdots, A_n)$　　α：領域の順序対

[層]

層(Grump)とは，線関数gによる領域Aの類別をいい，$G=G(g,A)$により表わす。この類別の要素はgに対して同一の値をもつA中のすべての点よりなる。gはGに対する層の生成関数とよぶ。gはAの中の長さ1の線(すなわち，すべての点)の上で定義される線関数である。

gに対する値がCである点によって定義されるAの部分集合を

$$G(g,A)\mid_{g=c}$$

により表わす。

[層関数]

層関数(FOG)は層に対して領域を割当てる写像である。このとき，

(a) 層の要素と領域の点との間に1対1対応がある。

(b) 領域中の点の各属性値は，層の中の対応する要素の領域関数により定義される

(c) 領域関数の値域は　対応する属性値集合の部分集合である。

層関数Hは，

$$H\equiv\begin{cases} g_1'=f_1 \\ g_2'=f_2 \\ \vdots \\ g_k'=f_k \end{cases}$$

により表わす。ここで，

f_iは領域関数

g_i'は領域中の各点に対するi番目の属性

kは属性の個数

とする。g_i'が存在しないとき，$g_i'=\Omega$とみなす。

[層関数により割り当てられる領域]

層関数Hにより割り当てられる領域Aは

$A = H(G)$ 又は $A = H(g, A')$

のように表わす。

[領域の順序づけ]

長さ1の順序つき線関数fによる領域Aの順序づけは $\alpha(f, A)$ で表わし，Aのすべての点をつくし，かつ $f(p_i) \leq f(p_{i+1})$ となる点の集合 $(p_1, p_2, \cdots, p_n)$ を言う。

[領域の全順序，半順序]

$f(p_i) = f(p_{i+1})$ なる点が存在しないとき，全順序という。$f(p_i) = f(p_{i+1})$ となる点の組が少なくとも1組存在するとき，領域の順序づけは半順序であるという。

[線関数に対する演算]

線関数には，

加法(+)，乗法(*)，除法(/)，反転(−)，
論理和(⊕)，論理積(@)，論理否定(¬)，
等号(=)，不等号(<)，選択(← →)

の演算が定義できる。

[加法(+)]

加法(+)はその値が次表で与えられる2項演算である。

f_1 \ f_2	Ω	θ	R_2	A_2
Ω	Ω	Ω	Ω	Ω
θ	Ω	θ	θ	Ω
R_1	Ω	θ	R_1+R_2	Ω
A_1	Ω	Ω	Ω	Ω

ここで R_1 と R_2 は任意の実数，A_1 と A_2 は任意の他の値とする。

[乗法(×)] 加法に準ずる
[除法(/)]
$f_2 = 0$ ならば すべてΩである点を加えれば，加法に準ずる。

[反転(−)]

f	Ω	θ	0	R	A
$-f$	Ω	θ	0	$-R$	Ω

[論理和]
論理和($\oplus$, $f_1 \oplus f_2$)は その値が次表で与えられる2項演算である。

f_1 \ f_2	Ω	F	θ	T	A_2
Ω	Ω	Ω	Ω	Ω	Ω
F	Ω	F	θ	T	Ω
θ	Ω	θ	θ	T	Ω
T	Ω	T	T	T	Ω
A_1	Ω	Ω	Ω	Ω	Ω

$T \oplus \theta = T$
$F \oplus \theta = \theta$
に注意。

[論理積]
論理積(@)は略
ただし， $F @ \theta = F$
$T @ \theta = \theta$ に注意する。

[論理否定(¬)]

f	Ω	F	θ	T	A
$\neg f$	Ω	T	θ	F	Ω

[等号(=)]

等号(=, $f_1 = f_2$)は その値が次表で与えられる2項演算である。

	$f_1 = f_2$
$v_1 = v_2$	T
$v_1 \neq v_2$	F

ここでv_1とv_2は 各々 f_1とf_2 の値である

[不等号(<)]

不等号(<, $f_1 < f_2$)は その値が次表で与えられる2項演算である。

	$f_1 < f_2$
$v_1 < v_2$	T
他の場合	F

ここでv_1とv_2は 各々 f_1とf_2 の値である

[選択]

選択($\leftarrow$ $\rightarrow$, $f_1 \leftarrow f_2 \rightarrow f_3$)は その値が次表で与えられる3項演算である。

f_2	$f_1 \leftarrow f_2 \rightarrow f_3$
T	v_1
F	v_3
0	0
その他	Ω

ここでv_1とv_3は 各々 f_1とf_2の値とする。

1.4 研究の目的

本論文の目的は，データ構造の記述及び処理手続の記述などの設計製作段階と，それによって作られた情報処理システムの実行段階を通して，一貫して使用者がデータの属性を任意に処理し，管理し，検査するために必要な機構とそれを中心とするソフトウェアシステムの構成法についての理論の開発を行うことである。

この目的を達成するために，本研究では属性処理概念についてのmodeling，design，implementationが述べられている。能率化の尺度として1.1では迅速性，低コスト性，正確性があげられている。正確性は理論からの演繹としてとらえることにより保証できる。迅速性と低コスト性は相反する側面がある。そこで，モデリングにあたって直接電子計算機上に実現しやすい簡潔性のある理論を構築することにより，迅速性と低コスト性を補助し，両者のどちらかをより重視する場合を2つに場合分けし，各々における最適な機構を設計している。

本論文では，(1)データ自身の表現方法 (2)データ間の関連 (3)データに対する基本操作 (4)基本操作の結合方式 について順に理論展開を行い，実現機構を設計し，応用例の提示を行う。

本論文で述べる属性処理機構はソフトウェアの作成段階と実行段階とにおいて首尾一貫して動作することを目的としている。

基本操作処理機構及び基本操作自動結合機構の存在により，プログラム作成時において詳細なデータ加工手順（データに対する基本操作）を記述することが不要となり省力化することができる。またプログラム実行時においてプログラム中で仮定しているデータの状態と実際に対象となっているデータの状態について整合性を常時チェックし，処理の正確性を自動的に保証することができる。さらに現在報告されているソフトウェア生産管理上の問題の基本的な解決を助けることができる。

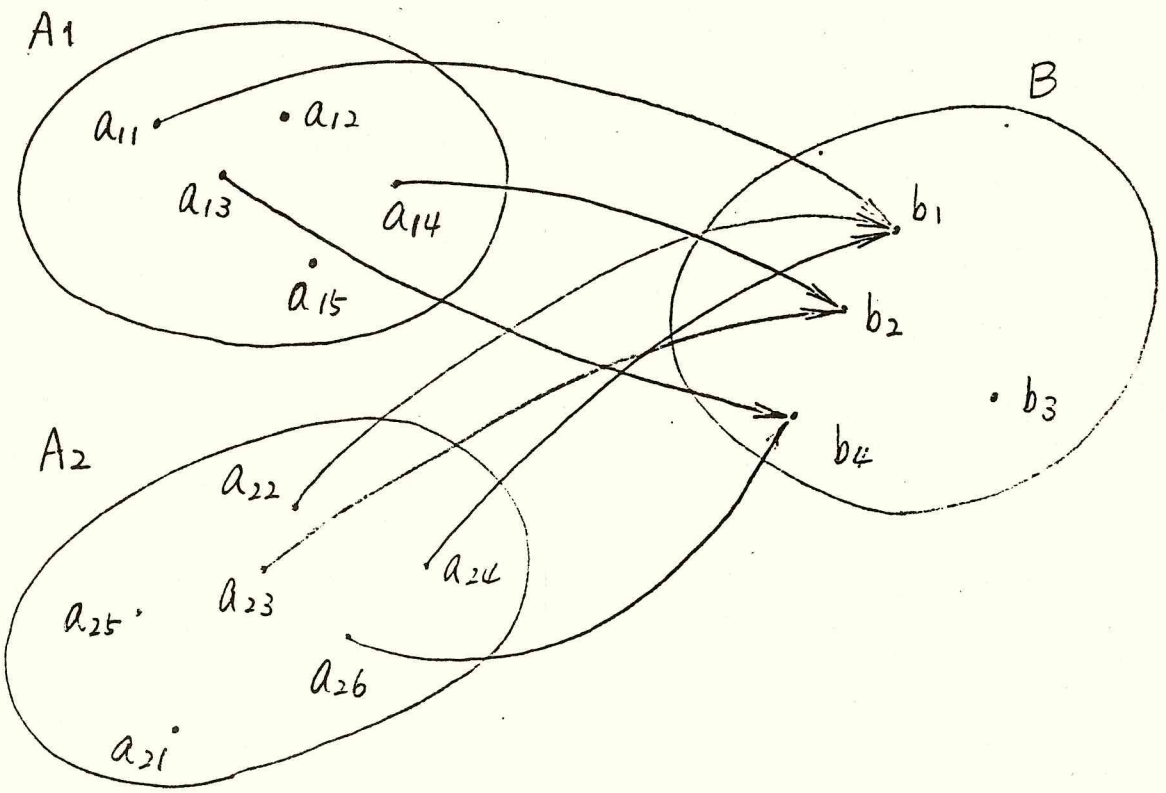
1.5 用語の定義

[ファイバー積]

集合 A_1, A_2, B と写像 $h_1 : A_1 \to B$, $h_2 : A_2 \to B$ が与えられたとき, $\{(a_1, a_2) \mid a_1 \in A_1, a_2 \in A_2, h_1(a_1) = h_2(a_2)\}$ を A_1, A_2 の B 上のファイバー積といい, $(A_1, h_1) \times_B (A_2, h_2)$ で表わす。

また, $h_i^{-1}(b) \subset A_i$ $(i = 1, 2)$ を $b \in B$ 上の A_i におけるファイバーと呼ぶ。

これを 図1-4 に示す。



A_1, A_2 の B 上のファイバー積

$= \{(a_{11}, a_{22}), (a_{11}, a_{24}), (a_{13}, a_{26}), (a_{14}, a_{23})\}$

図1-4 ファイバー積の例

〔レコード〕

データ d_j を要素とする集合を レコード R という

$$R = \{ d_1, d_2, \cdots, d_n \}$$

〔ファイル〕

レコード R_i を要素とする集合をファイル F とする

$$F = \{ R_1, R_2, \cdots, R_n \}$$

〔ファイル検索〕

ファイル F に対して，命題 $p(x)$ を満たす部分集合 V を作成する作業をファイル検索という。

$$V = \{ x \in F \mid p(x) \}$$

〔ファイルの更新〕

ファイル F 中の要素 f に対して $f' = p(f)$ を満たす f' より構成される F' を，F の更新済ファイルと呼び，写像 p を更新手続と呼ぶ。

$$p : F \longrightarrow F'$$

〔ファイル処理〕

入力ファイル群 $F_{in} = \{ F_{i1}, F_{i2}, \cdots, F_{in} \}$ に対して，

出力ファイル群 $F_{out} = \{ F_{o1}, F_{o2}, \cdots, F_{om} \}$ を対応させる手続をファイル処理プログラム P といい，この過程をファイル処理と呼ぶ。

〔ファイル処理の処理要素〕

Pを　次の関係を満たすように　P_1, P_2, …, P_mに分解する。

$$f_{i1}, \cdots, f_{in} \xrightarrow{P_1} f_{o1}$$
$$\vdots$$
$$f_{i1}, \cdots, f_{in} \xrightarrow{P_m} f_{om}$$

このときのP_iを処理要素と呼ぶ。

このP_iとしては　たとえば，ファイルの変換・更新・照合・併合その他の概念が存在する。

各々の概念については　3.6節で示す。

〔データ構造管理用記憶S〕

Sを次のように定義する.

$$S = \{C_1, C_2, C_3, \cdots, C_i, \cdots, C_m\}$$

ただし　C_i ：セル

i ：アドレス　$1 \leq i \leq m$

〔リンクリスト〕

セルC_iにより構成される任意の順序対をリンクリストとよぶ。

〔セル〕

C_iを次の順序対と定義する

$$C_i = (d_i, p_i)$$

d_i：データ

p_i：ポインタ　　$1 \leq p_i \leq m$ かつ $p_i \neq i$

このリンクリスト化することのできる記憶Sは　小規模な応用では自由プールと呼ばれることもある。

〔Lispマシン〕

Lisp言語の構造を意識して作成されたハードウェア及びファームウェアをもつ計算機。いわゆる高級言語マシンの1つである。

〔高級言語マシン〕 高級言語を機械語とする計算機

たとえばFortranマシンといったなら，Fortranによるプログラムを直接解釈実行できるものである。また，高級言語とは，広い一般通念のとおり機械語命令との直接的な対応をもつ低級言語（アセンブリ言語）以外の手続主体の問題向き言語をいう。

〔ファームウェア及びマイクロプログラム〕 機械語を構成するにあたって，それらが直接ハードウェアに写像されず，内在された別のプログラムによって解釈される場合，そのプログラムをマイクロプログラムとよび，それらを総称してファームウェアという。

ファームウェアはいうなれば機械語命令のインタプリタ・プログラムの総称である。ファームウェアのマイクロプログラムは ①固定化された記憶装置（ROM，書きかえ不可能） ②半固定記憶（PROM，オフライン書きかえ可能） ③通常の記憶（RAM，オンライン書きかえ可能）のいずれかにおさめられている。

〔マイクロ命令〕 マイクロプログラムを構成するための基本要素となる命令．ハードウェアの動作に直接対応する。

〔マイクロプロセッサ及びマイクロコンピュータ〕 基本的なCPU機能を1チップないしは数チップに収めたLSIをマイクロプロセッサとよぶ。マイクロプロセッサをCPUにもつコンピュータをマイクロコンピュータとよぶ。

第2章　属性空間と
属性処理機構理論

2.1 本章における研究の目的

本章では，情報及び情報構造についてのモデル化と，それに基づく処理機構論を述べている。

第1にデータを素情報に対する属性とその値の組としてとらえるモデル及びさらに抽象化を行ったモデルの2つを，情報代数理論［COD62］を基に展開している。前者のモデルで体系づけられているデータ空間を，半識別属性空間と呼ぶ。後者のモデルでの空間を抽象データ空間と呼ぶ。2.2に2つのモデルを述べている。

2.3では，半識別属性空間の意味的な解析として 濃度による類別を行っている。この類別は4章以降の設計実施例において利用される。

2.4では，半識別属性空間を直接電子計算機上に構築するために必要な体系と，その中で用いられる連想子を定義している。また連想子の利用法についてまとめられる。

2.5では，半識別属性空間に対してもう一つの抽象データ空間についてまとめている。またその中での基本要素である抽象事象に対して値の設定に必要な記述子表現と，その処理機構が導入される。

2.2 半識別属性空間と抽象データ空間

2.2.1 半識別属性空間と素情報

本論文におけるモデルでは，情報代数的な概念に基づく属性処理を電子計算機上で実現させるのに必要なインタフェイス概念を導入し，これを含めて理論的背景とする。

〔 属性空間の 電子計算機上での実現と，データ記述手続〕

一般に，データ処理時における属性空間Pは その中のデータ点dから記憶域Sの基本要素への写像fにより，電子計算機上に実現される。

$$f : P \longrightarrow S$$

写像fを データ記述手続とよぶ。

〔データの参照〕

データの参照（アクセス）とは S中のdに対して必要な属性値をとりだすことを表わす参照関数gをいう。

$$g : S \longrightarrow V$$

〔データ空間の確定とデータの内部表現〕

データ空間の確定とは，属性空間P中のデータ点dに対して合成関数$f \circ g$が定義された状態を言い，$f \circ g$の値をデータの内部表現という。

〔論理属性と記憶属性〕

属性空間を構成する座標集合を 論理属性と記憶属性に類別する。実現する機械によって，属性値を変更する必要がある属性を記憶属性という。 実現する機械によって属性値が変更されず，事象間の関係により属性値が変更される属性を 論理属性という。

〔属性の確定〕

属性の確定とは，その属性に対する属性値が存在することをいう。

〔論理属性の確定〕

論理属性は データ記述手続き f かつ/または 参照関数 g により確定する。

〔記憶属性の確定〕

記憶属性は 参照関数 g によってのみ確定する。

一般的にはデータ記述手続において，記憶属性が確定されることを許す概念もありうるが，本論文ではこれを禁止する。このことにより，データ空間は機械独立な処理と結びつけることができる。

〔半識別座標集合〕

相異なる論理属性の有限集合を半識別座標集合 Q' とよぶ。

〔半識別属性空間〕

半識別座標集合 Q' により構成される識別属性空間を 半識別属性空間 P とよぶ。

$$P = V_1 \times V_2 \times V_3 \times \cdots \times V_n$$

ただし，V_i は Q' の i 番目の論理属性の属性値集合である。半識別属性空間中の各点 p の座標は V_i のある値を a_i とするとき，a_i の n 個の組

$$p = (a_1, a_2, \cdots, a_n)$$

によって表わされる。

もし $n = 1$ ならば $P = V_1$ である。

もし $n = 0$ ならば P は零空間である。

〔素情報〕

半識別属性空間中の点 p を素情報という。

2.2.2 半識別属性空間の意味づけ

経験的にいって，データ処理プログラム中でのデータのかかわり方は 次の2つの段階に分けることができる。

phase 1： プログラム作成の段階
phase 2： プログラム実行の段階

phase 1では そのデータを利用するプログラム中での各文においてデータを識別するために，仮定されている論理属性が宣言される。実行文中では 宣言された論理属性をそのデータが持っていると仮定して，処理手順が構成される。COBOL，PL/I その他の多くの言語においては 位置，長さ，データ型などの記憶属性をも宣言し，実際にそのデータが機械の上でどのように表現されているかまでを phase 1で規定している。

phase 2では データの属性を仮定して作られた実行文により実際にデータの値，いいかえれば属性値が取り出され，格納され，処理をうける。phase 2の処理が実際にはじまる直前までに，記憶属性が確定していなければ 実行は進められない。処理時に仮定された属性と 実際の属性の間の妥当性のチェックが行われることもありうる。

本論文における概念では この記憶属性の確定が必要な，最も遅い時間ぎりぎりまで遅延させることにより，様々な柔軟性をうることと，（素情報，属性，属性値）の対をそのまま記憶させ，そのための特殊な命名された機構を付加することが特徴になっている。

このことを端的に表現すれば，半識別属性空間という概念の導入ということになる。

図示すると図2－1のようになる。これに処理手続との関連を加えた対応を図2－2に示す。

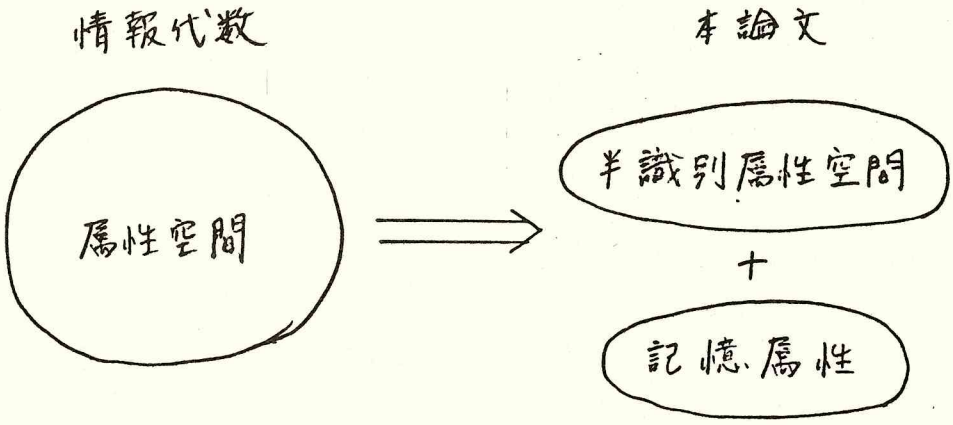


図2－1　抽象データ空間の展開

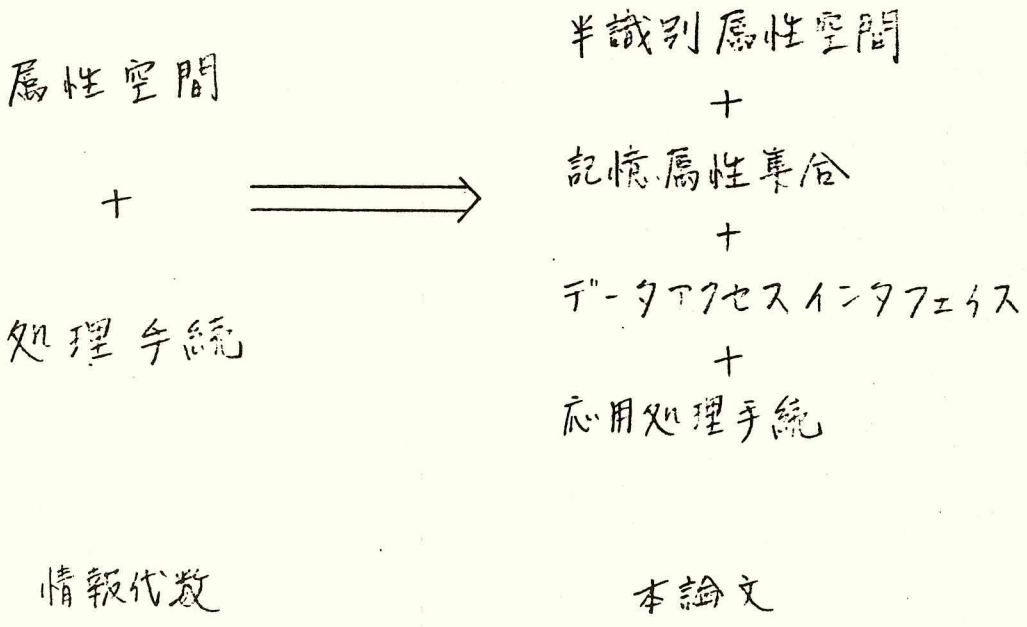


図2－2　本論文におけるデータ処理モデルへの展開

2.2.3 抽象データ空間と抽象事象

〔抽象事象〕

論理属性が未定義である（存在していない）事象を 抽象事象とよぶ。

〔抽象命令〕

抽象事象に対する電子計算機上の単位操作を抽象命令とよぶ。

〔抽象手続き〕

抽象命令を用いて構成される手続きを抽象手続きとよぶ。

〔抽象データ空間〕

抽象事象を要素とする集合を 抽象データ空間Aとよぶ。
抽象データ空間には 座標集合は存在しない。

〔抽象事象から素情報への全射〕

抽象事象と素情報との間には 1対多の対応関係が定義される。

〔抽象線〕

抽象データ空間から選ばれた点の 順序対をいう。
線の長さとは，線を構成する点の個数である。

〔抽象線関数・抽象順序つき線関数〕

属性空間，半試列属性空間と同様に，線に対して線関数及び順序つき線関数を定義できる。これを 抽象線関数及び 抽象順序つき線関数という。

〔抽象領域〕

抽象データ空間の任意の部分集合を抽象領域という。

〔抽象領域関数〕

抽象データ空間における領域に対して一つの値を 対応させる写像を 抽象領域関数という。

抽象事象とは，属性が存在していないデータ点（事象）のことである。抽象事象により構成されるデータ空間を 抽象データ空間と定義している。抽象事象に対して論理属性が付加されたものが素情報である。素情報に対して記憶属性が付加されたものが データ点（事象）である。この関係を 図2－3に示す。

抽象事象概念を導入することにより，電子計算機のためのソフトウェアを 対象となるデータの論理属性及び記憶属性から 独立して記述できる。この結果考えられる手続き概念を抽象手続きとよんでいる。

たとえば “ある数を100倍したものをある数に入れる” という概念を考える。“ある数” という言葉は識別的でないので，各々AとBと呼び，これを次のような抽象命令で表現する。

$$A \times 100 \longrightarrow B$$

この表現におけるAとBには 論理属性が与えられていない。すなわち Aおよび Bは 抽象データ空間中の点である。

$$A, B \in \mathcal{A}$$

このAおよびBに対応する半識別属性空間の点は 多数存在する。たとえば，論理属性の対（単価，売上金）を座標集合とする半識別属性空間中の素情報に 対応させることができる。
このとき，

$$A \times 100 \longrightarrow B$$

という抽象命令，いいかえれば 抽象データ空間中の要素の対応は，

抽象事象概念
＋
論理属性 ⟹ 素情報

素情報
＋
記憶属性 ⟹ データ点

図2-3 抽象事象，素情報，データ点の関係

単価×100 ⟶ 売上金

という半識別属性空間中の対応に 写像することができる。

論理属性の確定により，たとえば，

150円×100 ⟶ 1万5千円

となる。

このことにより 抽象手続きを意味のあるものにすることができる。しかし，この半識別属性空間上での操作概念は 機械上での実行はできない。記憶属性が存在しないからである。

たとえば 記憶属性として（番地，型，長さ）などを考えたとき，（1000，2進整数，4バイト）というように記憶属性が確定した場合，計算機上で動作することができる。

抽象データ空間，半識別属性空間，識別属性空間はこのような実現性をもたせることにより，単に理論的な概念にとどまらずに各々を計算機上に具体化することができる。

抽象手続きが 具体化される流れを図2－4に示す。

図2－4に示す手続的な流れを 自動的に行うシステムが第5章において設計・実施されている。その際の核になる原理は2.5節において基本構造が示される記述子を中心とするものである。

記述子は 記憶属性値を保持するものである。抽象手続きは原形手続きと呼ばれ，そのための定義言語が用意される。

実際の原形手続き定義言語には 抽象手続きを記述する機能に加えて，さらに動的に属性を処理することができるようなin stream - proc機能などが含められ，現実的なプログラムを記述できるように配慮されている。

記述子と同様に 2．4に原理が述べられている連想子は，図2－4に示す任意の段階での関係を直接計算機上に保持する記憶方式のことである。連想子を中心としたシステムは 第6章にその実現例がまとめられている。

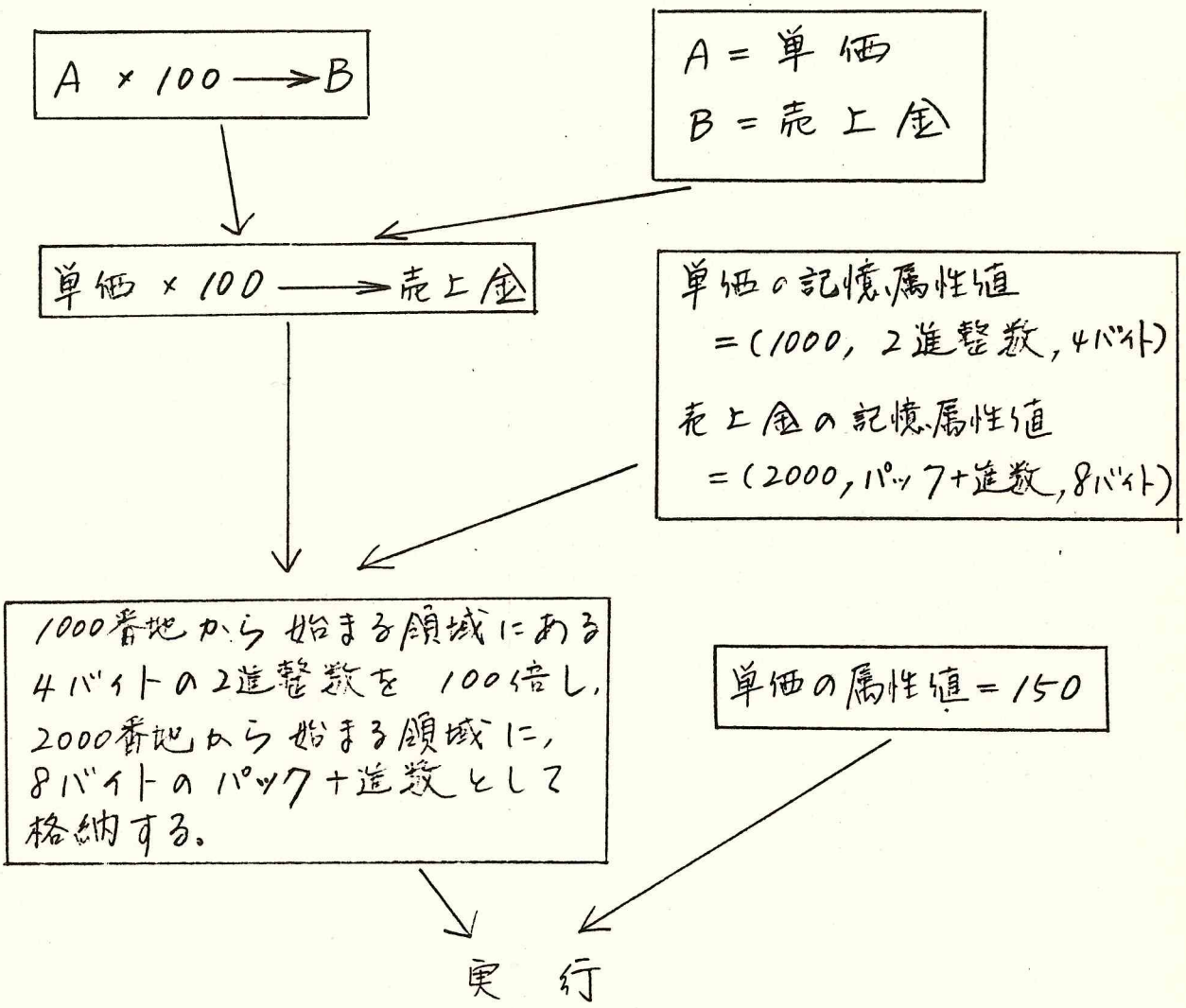


図2-4 抽象手続きの具体化

2.3 半識別属性空間の濃度による類別

2.3.1 均質データ空間と不均質データ空間

電子計算機上での処理Jは 処理手続Pと 対象となる情報群Dにより構成されると考えることにより一般化できる。

$$J = (P, D)$$ D:半識別属性空間上の領域

対象となる情報群Dは 素情報a_i よりなる。

$$D = \{a_1, a_2, \cdots, a_m\}$$

a_iは属性p_jと属性値v_{ij}との順序対よりなる。

$$a_i = \{(p_1, v_{i1}), (p_2, v_{i2}), \cdots, (p_m, v_{im})\}$$

このとき存在しうる属性集合をPとする。

$$P = \{P_1, P_2, \cdots, P_m\}$$

また，Pに対応した属性値の集合をVとする。

$$V = \{v_{11}, v_{12}, \cdots, v_{21}, v_{22}, \cdots, v_{nm}\}$$

v_{ij}は空であってもよい

半識別属性空間の表表現を図2－5に示す。

データ空間中の値，すなわち属性値はすべて存在するかどうかはわからない。情報処理のいくつかのケースにおいて2，3の素情報に対してのみ存在する属性も考えられるし，他の属性に対する属性値によって必ず空値となる場合もある。

また，データの処理・収集時において ある属性値が欠落することもある。

しかし，一方で定形的な処理を要なったデータの組に対して適用するような多くの事務処理の場合には，属性値はすべて存在することが前提となる。

[不均質データ空間]

空でない v_{ij} の個数を l としたとき，

$$l < nm/2$$

を満たすとき，これを不均質データ空間という。

[均質データ空間]

不均質でないデータ空間を 均質データ空間という。

すなわち，半分以上の要素が空であるものを不均質とよぶ。知識情報や計量経済における解析に利用されるデータ群などは，すべての素情報に対して同様の属性値が存在している場合は少ない。いいかえれば，空な属性値が多いデータ空間を形成している。

しかし，通常の事務処理ファイル・名簿管理ファイルなどを考えるとこれらは，ほとんど空なひがないものが多く，均質データ空間とみなせる場合が多い。

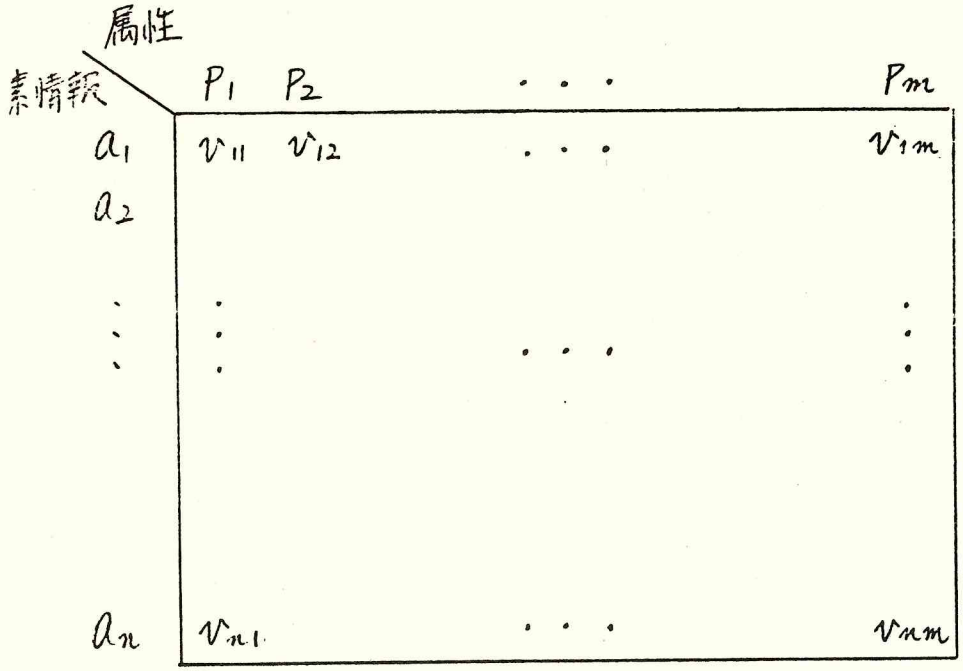


図 2－5　半識別属性空間と属性値

2.3.2 均質データ空間の例

事務処理では，対象となるすべてのデータに対して 定められた手順に基づく処理を，適用することが基本となっている。また，空値は異常値としての意味を持っている場合が多い。
表2－1に例を示す。

特殊分類 / 業種分類	計	国産分	素原材料	製品原材料	輸入分	素原材料	製品原材料
合計	205	168	49	119	37	28	9
公益事業	4	3	2	1	1	1	
石炭・亜炭鉱業	6	6	4	2			
製造工業	195	159	43	116	36	27	9
鉄鋼業	14	10	4	6	4	4	
非鉄金属工業	24	12	7	5	12	8	4
金属製品工業	4	4					
機械工業（船舶を除く）	19	19		19			
船舶	3	3	1	2			
窯業・土石製品工業	21	19	12	7	2	1	1
化学工業	40	33	5	28	7	6	1
石油製品	4	3	2	1	1	1	
石炭製品	6	3	3		3	2	1
ゴム製品工業	7	6	1	5	1	1	
皮革製品工業	2	1	1		1	1	
パルプ・紙・紙加工品工業	11	10	4	6	1		1
繊維工業	36	32	3	29	4	3	1
その他工業（プラスチック製品）	4	4		4			

表2－1 均質データ空間の例
——原材料消費指数分類別採用品目数［NIH72］p47より

表2－1の表の場合，$l = 99$である。これは表の大きさ$18 \times 7 = 126$の2分の1より大きい。

$$l = 99 > 18 \times 7/2$$
$$D = \{a_1, a_2, \cdots, a_{18}\}$$
$$= \{\text{合計}, \text{公益事業}, \cdots, \text{その他工業}\}$$
$$P = \{p_1, p_2, \cdots, p_7\}$$
$$= \{\text{計}, \text{国産分}, \cdots, \text{輸入分製品原材料}\}$$

2.3.3 不均質データ空間の例

属性が動的に増減する場合，もしくは多様な属性があり，空値の存在にも意味がある場合などは，不均質データ空間とみなすことができる。

たとえば個人情報ファイルについて考えると，氏名，性別，年令，住所，既婚／未婚，子供の数などの属性は前もって設定することができる。表2－1に示した名簿に準じて構成できる。

しかし，一時的に必要となる属性や，局所的に必要な属性がある。これらに対する属性及び属性値は，前もって存在を予定することができない。均質データ空間とみなすことよりも不均質データ空間とみなし，動的に属性の個数が変わることを前提として扱うことが望ましい。簡単な例を表2－2に示す。

表2－2の表の場合，$l = 28$である。これは表の大きさ$16 \times 4 = 64$の2分の1より小さい。

$$l = 28 < 16 \times 4/2$$
$$D = \{a_1, a_2, \cdots, a_{16}\}$$
$$= \{\text{合計}, \text{石炭}, \cdots, \text{織物}\}$$
$$P = \{p_1, p_2, p_3, p_4\}$$
$$= \{\text{素原材料}, \text{国産分}, \text{輸入分}, \text{製品}\}$$

	素原材料	国産分	輸入分	製品
合計	7	4	3	42
石炭	1	1		
鋼材				1
非鉄金属	3	3		5
機械器具				15
民生用電気機械				6
自動車				6
写真機・時計				3
石油製品				5
コークス				1
生ゴム	1		1	
洋紙				1
繊維原料・糸・織物	2		2	14
繊維原料	2		2	1
糸				7
織物				6

表2-2 不均質データ空間の例
——販売業者在庫指数分類別採用品目数〔NIH 72〕p87より

2.4 半識別属性空間の電子計算機上での実現

2.4.1 データ空間の包括的な取扱い

半識別属性空間の概念を直接的に電子計算機上に実現する方法について述べる。半識別属性空間の基本要素を連想子と呼ぶ。

連想子は個々の情報に対して与えられる属性と属性値の対を，

(対象，属性，属性値)

の形で直接表現するものである。いいかえれば，独立した各々のデータの定義方法を示している。

本節では この連想子を利用してどのように情報の管理を行うことができるか，従来の情報処理の概念とどのような対応が考えることができるかについてまとめる。

まず，データ空間をデータとその構造（関連）としてとらえるために，構造の包括的な取扱いはどのような方法で行なわれうるかを示す。次にこれに対して，各データの管理機構を付加する。こうした論法により，データ空間の包括的な処理概念を述べる。

データベースその他の集中化された情報を処理するためのファイルシステムでは，その中にまとめられたデータ自身，そしてデータ間の関連を処理するためにリンクリスト（linked list）が用いられている。リンクリストは それをおくことのできる電子計算機上の記憶装置が一括管理されていることを前提とした，ソフトウェア概念である。記憶装置としては主記憶，磁気ディスクあるいはそれらの上に構築された仮想メモリなど，乱アクセスの可能な記憶ならなんでもよい。

このように考えられるリンクリスト及びセル構成には，リングでとめる単語帳あるいはルーズリーフと対応した操作概念を与えることができる。

単語帳の用紙及びルーズリーフの用紙は 一括管理することができる。そして，必要に応じてそこから取り出し，情報を記述し，

目的別にまとめてとめることができる。目的別にまとめられたもののうち 不要な部分ができたならそれらはとりはずし，記述された情報を消すことにより再使用することができる。この概念を計算機の記憶上にもってきたものが一括管理技法である。このときの基本操作は次のようなものである。

1. get-a-cell ：セルを1つうる
2. return-a-cell：セルを返す

そして それらによって処理されるセルを目的別にまとめるためにリンクリストが利用されるが，その基本操作として次のようなものがあげられる。

1. add-a-cell-to-a-list：リストにセルをつなぐ
2. delete-a-cell-from-a-list：リストからセルを削除
3. find-a-cell-in-a-list：リストからセルを発見する

これらのリンクリスト処理の上に

stack，queue，その他

の意味的な概念を構築することもできる。

利用したセルが使用済になったとき，それを使用者が自発的に管理部に返還する方法の他に，管理部の側で使用済のセルを探して回収する方法が存在する。これを一般にガベジコレクション（Garbage collection）と呼んでいる。

ガベジコレクタは一般に，自由領域中の活きたセルに対して何らかの方法によってマークを行うマーキングフェイズと，マークのついていないセル（即ちガベジ）を回収し，かつ，活きたセルにつけられたマークをはずすリクレイムフェイズの2つの段階から成っている。

ガベジコレクションのアルゴリズムを整理すると次のようになる。

- ビットマーキング法 [MCC60]
  - ビットテーブル
  - セル内マーク
- ポインタ反転法 [SCH67]
- つめあわせ法 (Compactifying) [Han 69]

また，これらのプログラムの動作環境に応じて，

- シリアルG.C. [KNU68]
- シリアル リアルタイムG.C. [BAK78]
- パラレルG.C. [DIJ75], [STE75], [WAD76]

の3形態がある。

シリアルG.C.ではLisp処理系内の副手法として G.C.をおき，G.C.動作中は他の処理は停止する。

シリアル リアルタイムG.C.は 処理系内の副手続としてG.C.があるが，自由領域を2種置き，その間のスイッチングをすることにより G.C.オーバーヘッドを著しく減少させる特徴を持っている。

パラレルG.C.は 主にダイクストラの3色塗り分けアルゴリズムが有名であるが，主記憶共有型のマルチプロセッサ，あるいはマルチタスキングを行うユニプロセッサ上で動作することを考えたもので，処理系とは独立して自由領域の管理を専門とするプログラム形態である。

シリアルG.C.は自由領域がなくなったときに作動する。しかし残り少ない自由領域では G.C.の作動する回数が増え，オーバーヘッド過大となるおそれがあるので，たとえば90%のセルを使用したら働くようになっているシステムも考えられる。

このG.C.を全面的に採用し，かつ最初に成功したのは後述するLisp言語である。このことは [SAM69] にも述べられている。また本論文においても そのことによりLispを母言語としてシステム概念の提示を行っている。

リンクリスト表現は 理論的には有向グラフ $\vec{G}=(A,N,\phi)$ に帰着する。(図2-6) リンクリスト表現は ポインタのつけかえにより情報間の関係を変更できるので 構造の柔軟性がある。

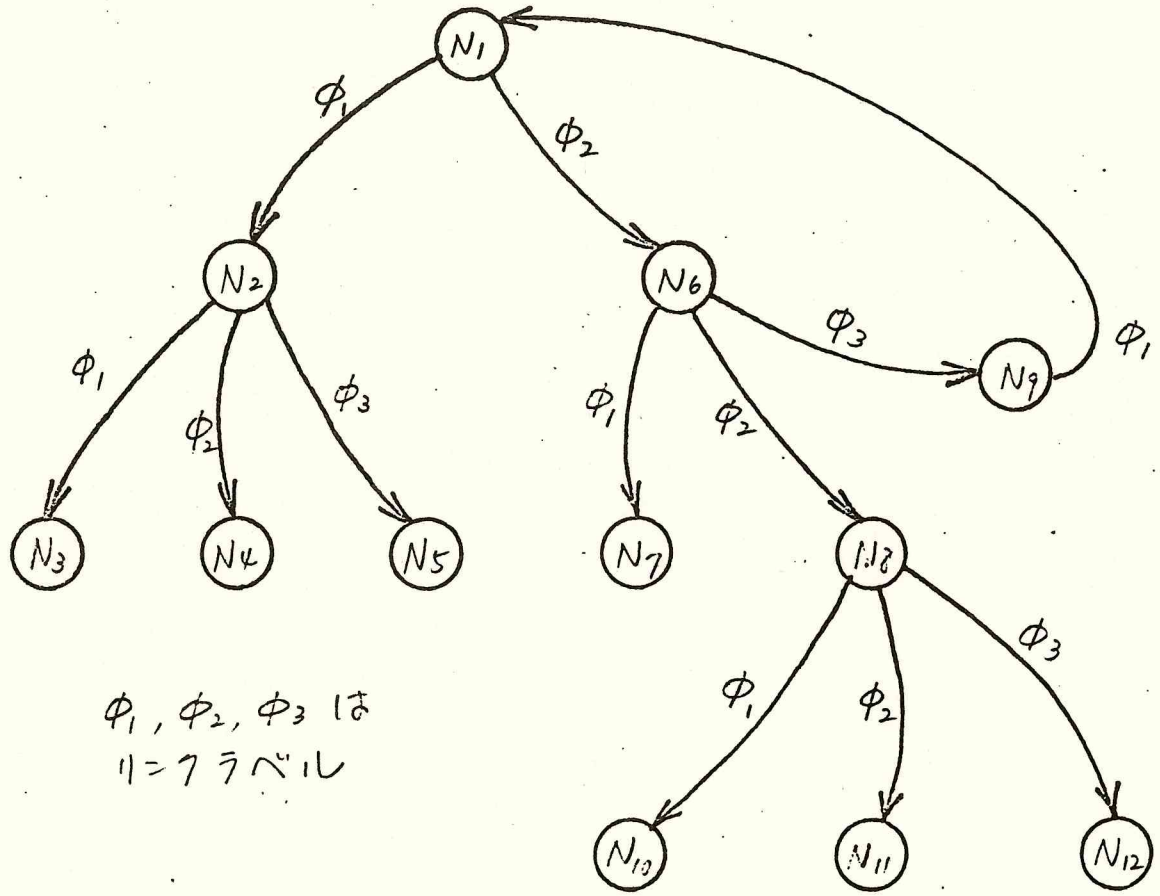


図2-6 有向グラフ $\vec{G}$

リンクリストに対して各種の操作を容易にほどこすことができる。またその結果，処理全体の見通しが一般に良くなることが指摘できる。たとえば図2-7に示したレコード $x_i=(a_{i1}\cdots a_{im})$ の集合 $x=\{x_1\cdots x_n\}$ (いわゆるファイルに相等) に対して，各レコードにフィールド a_{i0} を追加し，フィールド a_{i2} を削除する変更は次の定義を実行させればよい。

```
additem[x;a] = mapcar[x;λ[[i];
    progn[i:=nconc[list[car[a];car[i]];cddr[i]];
```

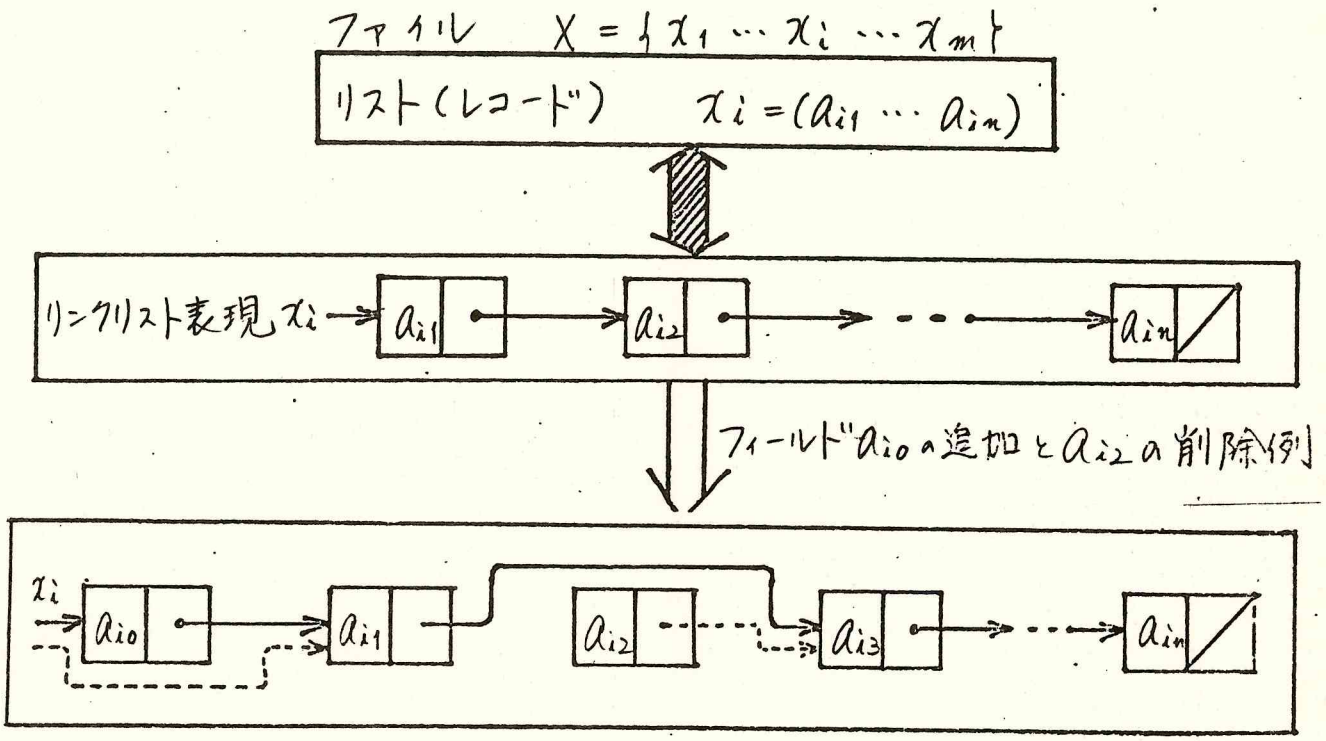


図2-7 リンクリスト表現

a; = cdr[a]; i]]]

説明：ファイルXのすべての要素xiに次のことを行なえ。

(mapcar [x; λ[[i] …]])

① a_{i0} を追加する。(list [car[a]; [car[i]])

② a_{i2} を削除する。(nconc [list[…], cddr[i]])

③ 次への準備処理

また，項をすべて数値としたとき，累計表 $y = \{y_1 \cdots y_n\}$，$y_j = \Sigma a_{ij}$ を更新する手続は次のようになる。

```
summerize[x; y] = mapcar[y; λ[[j]; progn[
    x := mapcar[x; λ[[i]; progn[k := sum[k, car[i]];
    cdr[i]]]]; j]]]
```

説明：累計表 y のすべての要素 y_j に次のことを行なえ。

(mapcar[y; λ[[j] …]])

① ファイル x の各要素 x_i に対して a_{ij} を y_j に足しこむ。

(x := mapcar[x; λ[[i] …]])

② 次への準備

このような特徴に着目した研究は 筆者らの関連分野においてもいくつか行われている。たとえば Lieberman [LIE75] は，図2-8の形で財務諸表の構造化を行い，これを基にイベント会計システムの提案を行っている。

しかしリンクリストのみによる構造化は，構造の柔軟性などの性質は満足するが，各フィールドの参照はリストをたどって行かねばならず，本質的に低速である。つまり含まれる項目数 n に比例して，求めるフィールドの値をとるためのリストたどりの手間が増加し，平均 $n/2$ 回を要することとなる（参照の手間 $= O(n)$）

含まれる項目数に依存せず，一意に（$O(1)$ で）値をとり出す機構が付加できれば参照の高速性が得られ，当初にかかげた目標をすべて満足する情報モデルとすることができる。そのために導入されたのが 連想子である。

2.4.2 データ空間の分割

データ構造の一括管理が可能となるリンクリスト表現を利用することにより，データ間の関連を任意に処理し，保持する機構を計算機上に実現できることを 前節において示した。

すべての情報をこのリンクリスト表現により表わし，またその処理系もリンクリストとして記憶でき，それらを実行できることは [MCC60] に示されている。

データ空間を次の3つの副空間に分割する。

(i)構造定義空間：リンクリスト表現を保持するための領域
(2.4.1参照)

(ii)素情報空間：素情報を保持するための領域

(iii)数空間　：　数値を保持するための空間

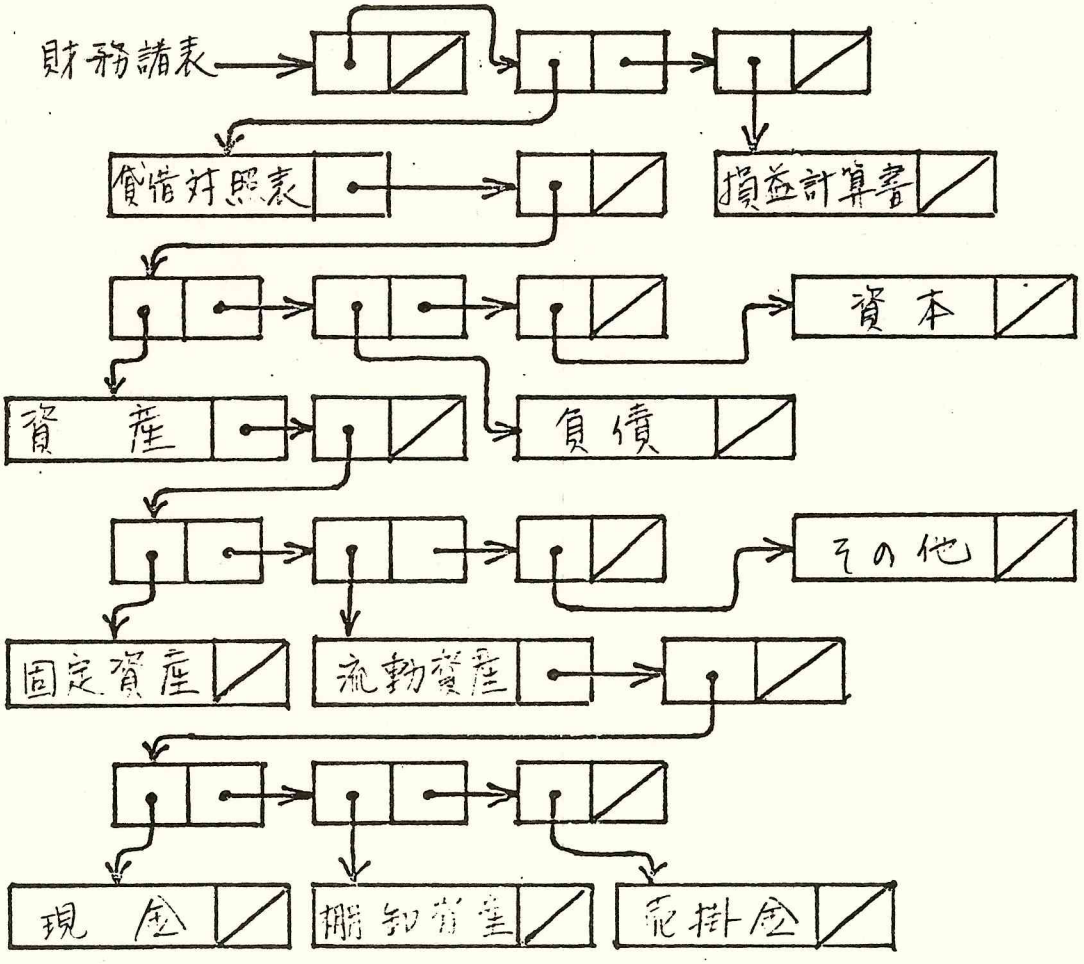
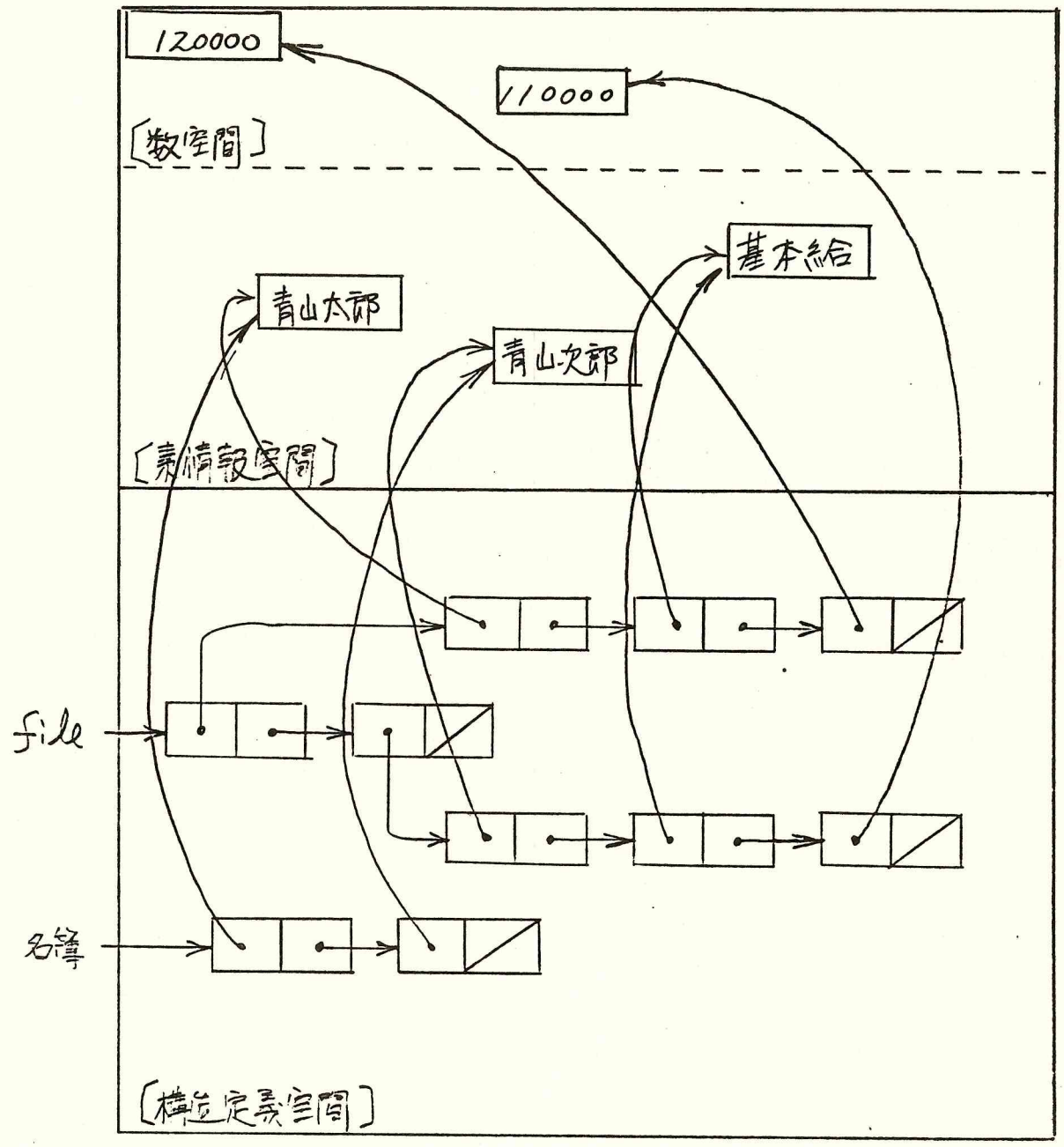


図2-8　Lisp型リンクリストによる財務諸表構造化の一例[LIE75]



名簿（青山太郎，青山次郎）

file（（青山太郎，基本給，12万），（青山次郎，基本給，11万））

図 2-9　リンクリスト表現

このとき，

((青山太郎，基本給，12万)，(青山次郎，基本給，11万))

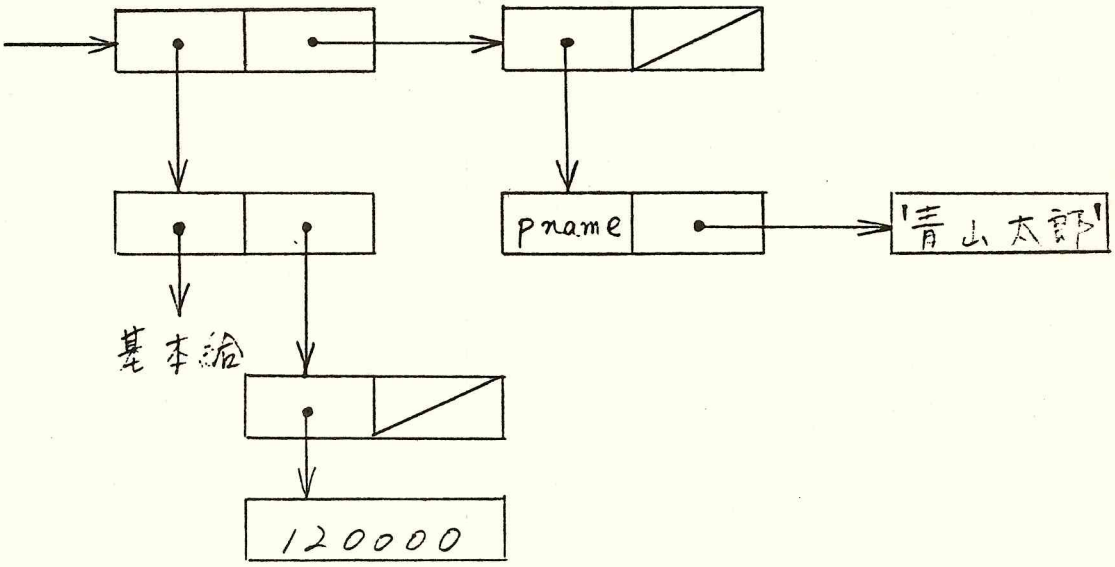
というファイル 及び

(青山太郎，青山次郎)

という名簿ファイルの計算機上での表現について考えてみる。素情報として名前だけを利用し，すべての素情報・数の間の関係は リンクリストにより構造定義空間において表現することが，その最も簡単な実現方法である。これを図2－9に示す。

これは[MCC60]において示唆され，実現された方法である。

次に青山次郎を表わす素情報に対する属性を規定するP-リスト概念での実現手法を図2－10に示す。P-リスト表現による場合，名前が与えられた場合にそれに付随する属性・属性値をすべて管理することができる利点があるが，情報をたどる手間はリンクリスト構成とかわらない。



青山太郎という素情報を表わすPリストに
(青山太郎，基本給，12万)を加える。

図2－10 属性・属性値のPリストによる表現
([MCC60]に準拠)

2.4.3 素情報概念と連想子

前節までに述べられた手法に対して連想子を導入することにより，高速参照能力と構造の柔軟性・簡潔性を併せ持つことができる。

素情報空間には　連想子と素記号の2種の要素がある。両者は同一の形式をもつ。

連想子の記憶形式の図的表現を　図2-11に示す。

また，連想子でない素情報は名前と値の対を持ちうるが，その表現形式を図2-12のように扱うことにより一般化することができる。

2.4.4 不均質データ空間への連想子の利用

これを利用したデータファイルの格納形を図2-13に示す。図2-13は図2-9に示す原形に比べて扱いやすくなっている。

図2-9の方式では　file中の要素の関係はすべて，構造定義空間において定義されていた。図2-13の連想子を利用した方法では（青山太郎，基本給，12万）等の項は　まとめられ，連想子として素情報空間に1単位を占めて格納されている。fileからはその連想子へのポインタをつけるだけでよい。

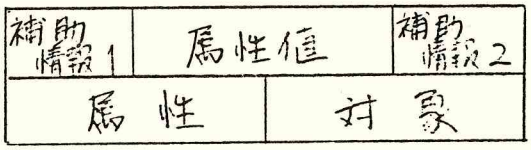


図2-11 連想子（対象，属性，属性値）の図的表現

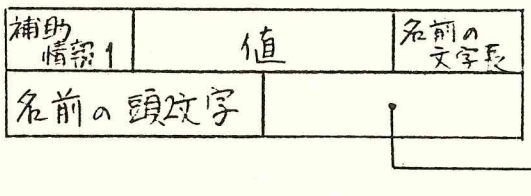
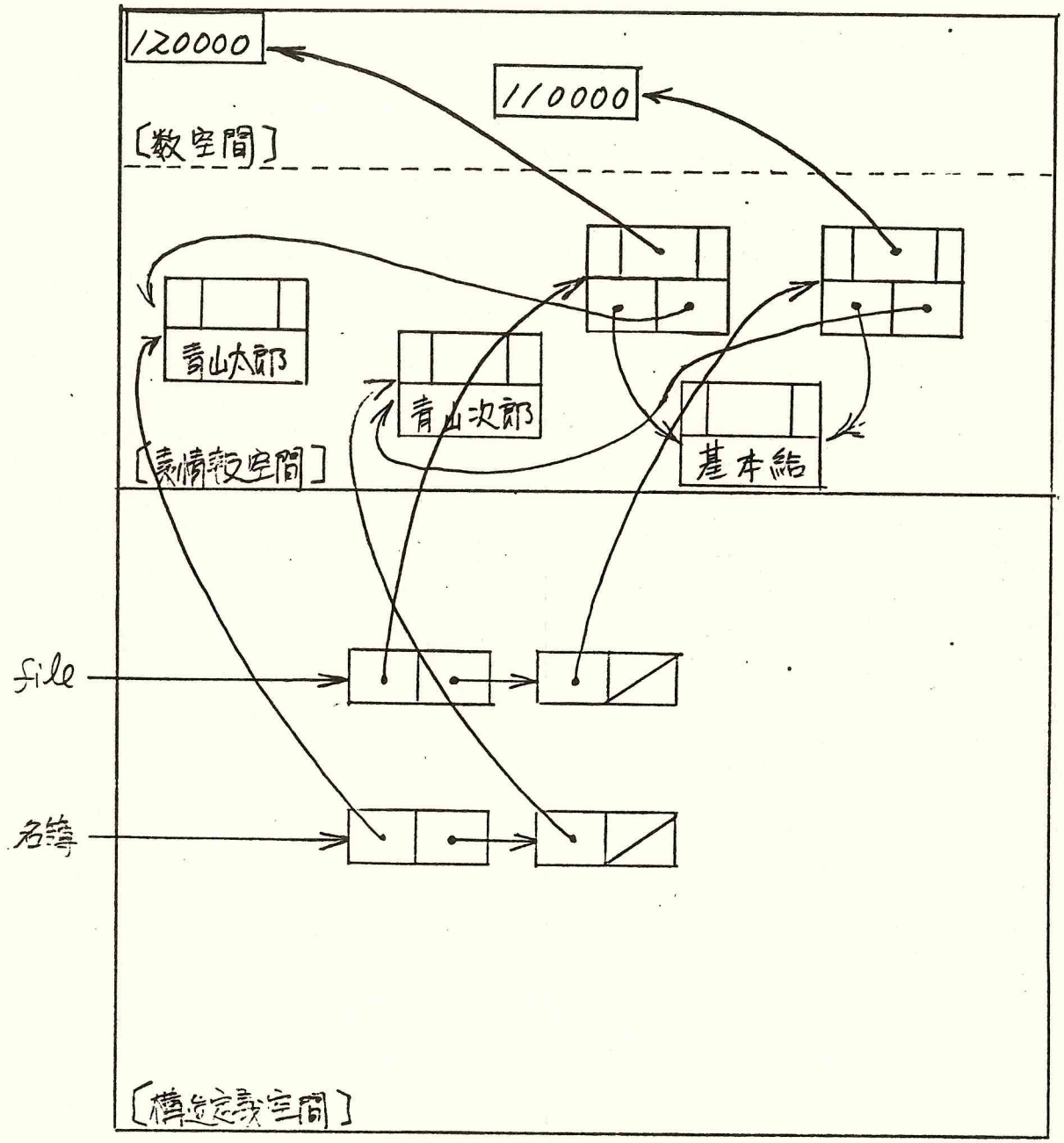


図2-12 素記号の図的表現（連想子との一般化）



名簿（青山太郎，青山次郎）

file ((青山太郎，基本給，12万)，(青山次郎，基本給，11万))

図2-13 連想子を利用したリンクリスト表現 — 中間段階

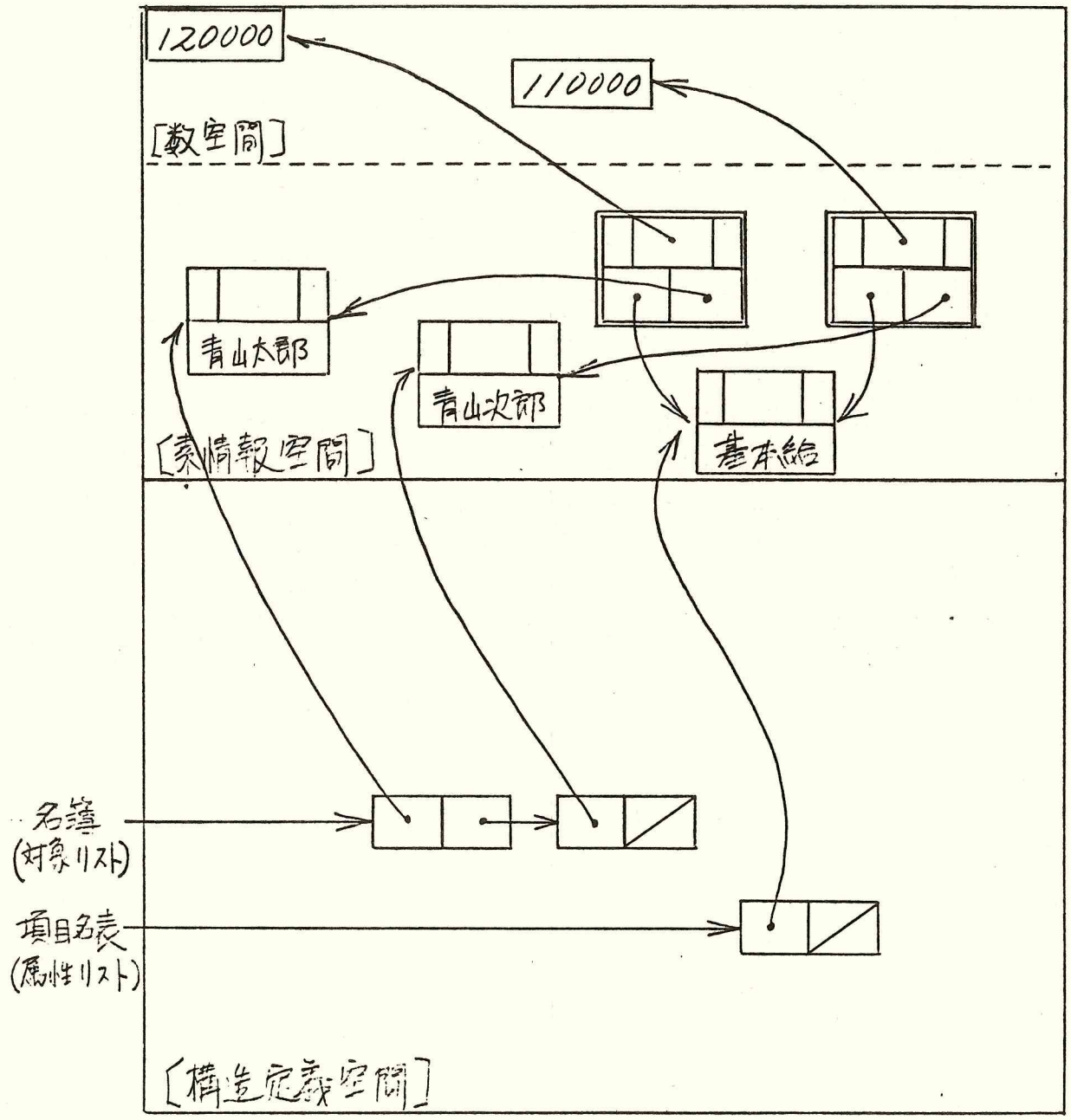
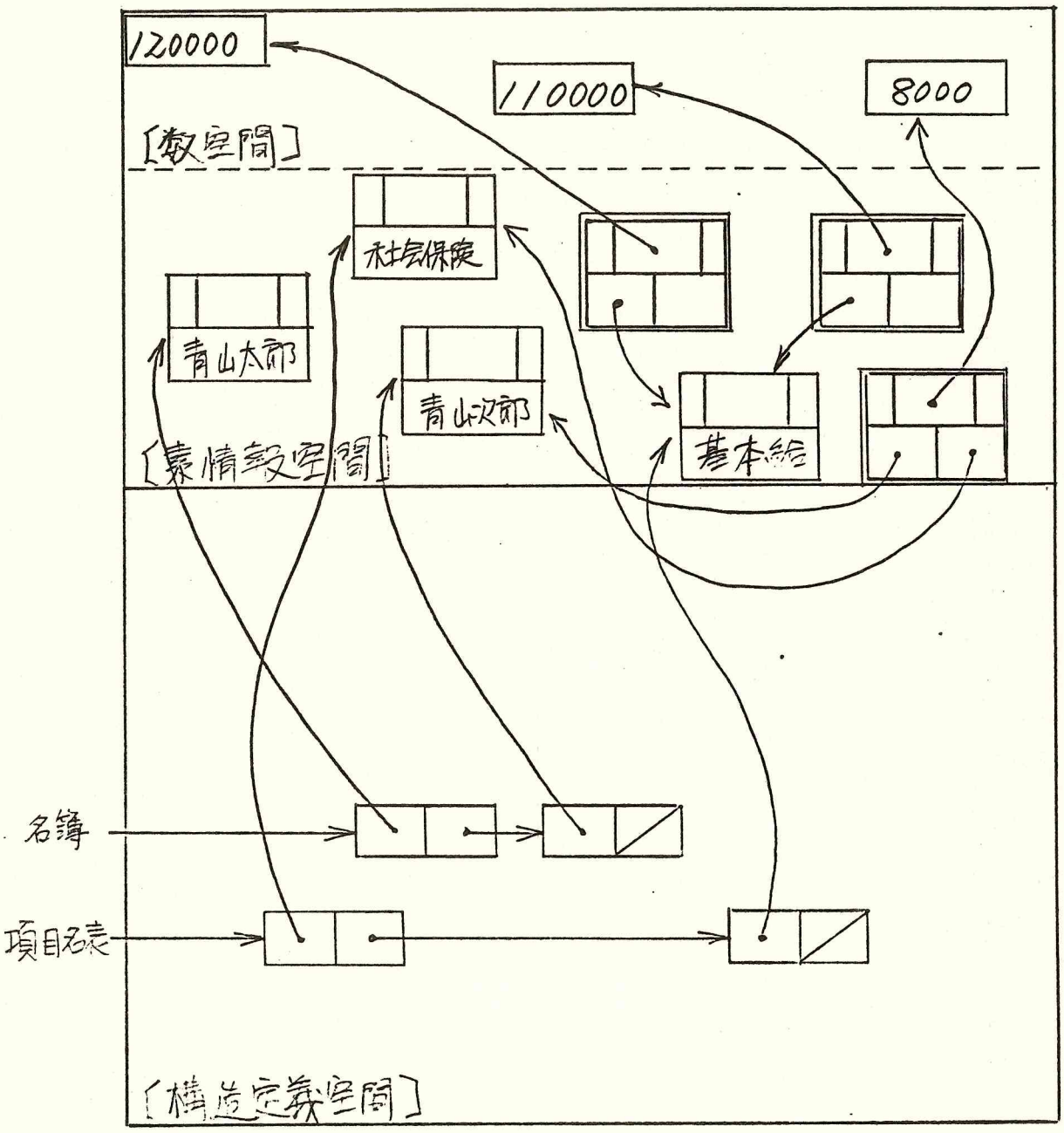


図2-14(i) 連想子を利用した表現
(各連想子はリンクリストをたどらずに
直接参照される。)



（青山次郎，社会保険，8000）の追加

図2-14（ii） 項目追加後の表現

記憶域への参照回数により連想子の効用を見てみよう。今，「青山次郎の基本給を問う」としたとき，図2－13の連想子を利用した構造では，

1. file中の最初の要素を参照する
2. その名前を参照する
3. （それは青山太郎であり，青山次郎でないので）次の要素を参照する
4. その名前を参照する
5. （それは青山次郎なので）その属性部を参照する
6. （それは基本給なので）値をとり出す

に示す6回の参照が行われる。

これに対して図2－9のリンクリストのみによる構造では，

1. file中の最初の要素を参照する
2. その値部を参照する
3. その値部を参照する
4. （それは青山次郎でないので）fileの次の要素を参照する
5. その値部を参照する
6. その値部を参照する
7. （それは青山次郎なので）5のポインタ部がさすセルを参照する
8. その値部を参照する
9. （それは基本給なので）8のポインタ部がさすセルを参照する
10. その値部より値をとり出す

に示す，10回の参照が必要である。

（対象，属性，？）という形の参照に着眼し，4．2に示した機構を仮定することにより，さらに高速にすることが考えられる。その場合の記憶域への参照は次の手順により行われる。

記憶域への参照回数により連想子の効用を見てみよう。今，「青山次郎の基本給を問う」としたとき，図2－13の連想子を利用した構造では，

1. file中の最初の要素を参照する
2. その名前を参照する
3. （それは青山太郎であり，青山次郎でないので）次の要素を参照する
4. その名前を参照する
5. （それは青山次郎なので）その属性部を参照する
6. （それは基本給なので）値をとりだす

に示す6回の参照が行われる。

これに対して図2－9のリンクリストのみによる構造では，

1. file中の最初の要素を参照する
2. その値部を参照する
3. その値部を参照する
4. （それは青山次郎でないので）fileの次の要素を参照する
5. その値部を参照する
6. その値部を参照する
7. （それは青山次郎なので）5のポインタ部がさすセルを参照する
8. その値部を参照する
9. （それは基本給なので）8のポインタ部がさすセルを参照する
10. その値部より値をとりだす

に示す，10回の参照が必要である。

（対象，属性，？）という形の参照に着眼し，4.ごに示した機構を仮定することにより，さらに高速にすることが考えられる。その場合の記憶域への参照は次の手順により行われる。

1. (青山太郎, 基本給)を鍵とする連想子を直接参照する
2. 値をとりだす

その場合の記憶域構成を図2-14に示す。この場合, 全データを関連づけるfileは不要となっている点に特徴がある。

図2-9及び図2-13は 図2-14を導入するために, 便宜的にただ1つの項目, すなわち基本給しか示さなかった。図2-9もしくは図2-13に複数個の項目, たとえば住所, 性別, 経歴, 社会保険, 累積給与支給額などを追加した場合, 非常に複雑な構造になるのは明らかである。しかし, 図2-14に示す連想子の利用方法によれば, 項目表に項目名をつけ, その後必要な連想子を作成するだけでよい。そのことを行うだけですべての情報の関連を保持し, かつ各要素に高速参照を行うことができる。この方式の実現例は4.2及び第6章に後述する。

2.4.5 連想子による属性処理機構の実現

鍵に対する値の高速参照機構を連想(association)とよぶ。図2-15に示すように, 連想は集合間の写像(mapping)の実現といえる。この写像は通常, 全単射的(bijective)に作成するが, 応用によっては全射性も単射性も要しない。

基本連想構成 $f: K \longrightarrow V$

K: 鍵集合

V: 値集合

写像関係をもつ $k \in K$, $v \in V$ の対 (k, v) を連想子(associator)とよび, 構成の基本要素とする。

さらにk及びvに対して, 文(文字列及び文字列のリンクリスト表現)を直接用いられるように計算機上に実現する。

このことにより, たとえば, 「子供」 「かわいい」, 「来月の生産可能台数」 「1500台以下」 などのような情報を, そのままの形で計算機上に格納することができる。

連想子は (k_i, v_i) を2×nの表に入れても擬似的に格納することができる。単純な記号表はその例である。しかしそうした場合には，表中の各要素を高速に参照する能力はなく，連想子とは呼びがたい。

これに対してVのみを配列として保持し，鍵から直接にその要素を選ぶ機構をおくことができれば，存在する連想子れに関係なくO(1)で高速参照することができる。この機構は一般に，非数値である鍵に対して一意に定められる数値を計算し，その数を配列への添字として直接利用することに相当する。

定義域Kを分割することにより，いわゆるファイルレコードを連想子として保持することが可能となる。

連想情報モデル：$A \times O \rightarrow V$

A：属性　(Attribute)
O：対象　(Object)
V：値　(Value)

連想子 $\equiv (a, o, v)$　　$a \in A, o \in O, v \in V$

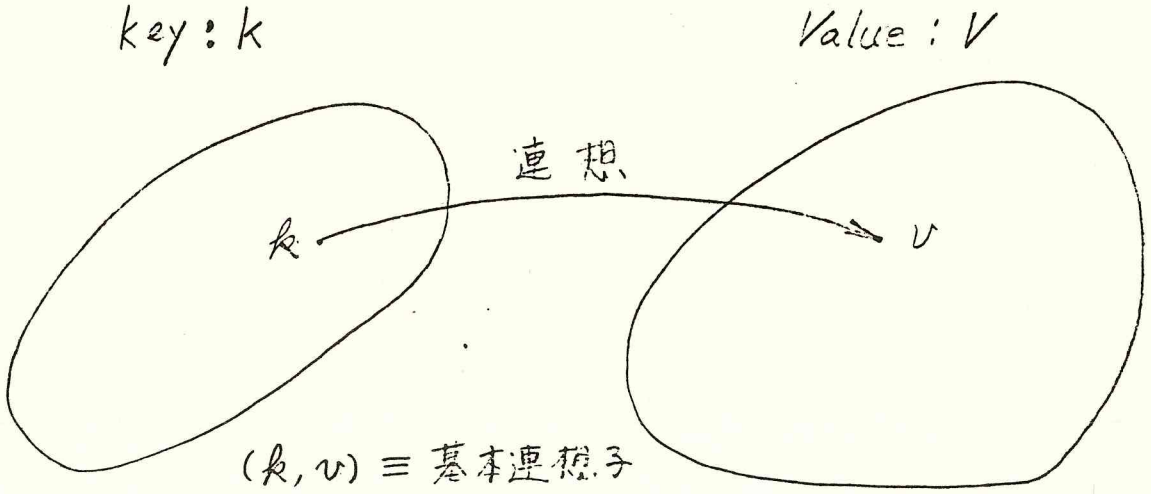


図2-15　基本連想構成

たとえば ある従業員に対する入社年度ないし基本給の格納は連想の概念を用いて次のように表現する。

「青山太郎」 $\xrightarrow{\text{「入社年度」}}$ 「1970年」

「青山太郎」 $\xrightarrow{\text{「基本給」}}$ 「12万円」

$A = \{$入社年度, 基本給$\}$
$O = \{$青山太郎$\}$
$V = \{$1970年, 12万円$\}$

このように $a(O) = v$ の関係を保つようにして a, O を定める。これにより各値の検索要求は

(入社年度, 青山太郎, ?)
(基本給, 青山太郎, ?)

という形で表現でき, (a, O) を鍵とする連想高速検索により値を得ることができる。(実際の計算機への実現においても, それらは記号のままコード化されることなしに格納される。しかしながら, 現在適当な漢字入出力装置がないのでローマ字表現で処理を行っている。)

2.5 抽象データ空間の電子計算機上での実現

2.5.1 抽象事象からデータ点への合成写像と記述子

2.2における抽象事象の具体化の概念は次のようなものであった。抽象データ空間Aと半識別属性空間Vの間に，次のような一対多の写像をおく。

$$f : A \longmapsto V$$

fの機能は抽象事象に対する論理属性の付与である。

次に実データ空間DとVとの間の写像gをおく。

$$g : V \longmapsto D$$

gの機能は素情報に対する記憶属性の付与である。これにより，実際に値を記憶し処理することが可能となる。

そこでプログラムを作成するときに，その手続の中で利用する記号の意味，いいかえれば変数名を手続を変更することなしに外部から自由に設定し，特定の変数名に対するプログラムを多数生成することができたならば そのプログラムは抽象手続と呼ぶことができる。またその中で，利用されている記号は抽象データ空間中の抽象事象と考えることができる。この機能の実現機構は第4章に示されている。

2.4において示した，半識別属性空間の実現機構はgを処理システム内部で統一化して処理し，画一的なセルという単位で事象を扱うことにより記憶属性の処理を省略化する構成法であった。不均質なデータ空間の場合や，データ処理のための特殊システムを構築する場合には こうした考え方が充分成立する。この応用例は第6章に示している。

しかし，他の汎用言語等で日常的に利用されているファイル形式は，属性値ひとのみからなるレコードの対，

$(v_{11}, v_{21}, \cdots, v_{n1}, v_{12}, v_{22}, \cdots, v_{nm})$

である。このレコード構成をそのまま使用しなければならない場合も多く存在する。均質データ空間からの典型的な例である。

そこで，本来の記録は，

$((P_1\ v_{11}), (P_2\ v_{21}), \cdots, (P_n\ v_{n1}), (P_1\ v_{21}), \cdots, (P_n\ v_{nm}))$

であるべきであるので，属性値のみからなるレコードと別に，

$(P_1, \cdots, P_n)$

という記録を作成する。

v_{ij}
の参照において P_j を利用すれば v_{ij} の属性を管理することができる。

[記述子]

記憶属性 P_j の電子計算機での記憶単位を記述子とよぶ。

この記述子を生成することにより記憶属性を画一化せずに，日常利用されているファイルに対して属性処理を行うことができる。

次にこの方式を系統的に実現する手順を示す。まず属性値のみからなるデータファイルの形式及び論理属性・記憶属性の定義を行い，電子計算機上に登録する。登録された属性に対して一つの名前を与える。次に抽象手続として，プログラム構造を電子計算機上に登録する。

このような準備のもとに，ある特定の処理をある特定のデータファイルに適応するプログラムは，抽象手続中で対象とするファイル名になるべき抽象事象と あらかじめ登録された属性群に対する名前を結合することにより生成される。

この結合では 抽象事象に対する論理属性と記憶属性を同時に与えている。論理属性の付与により，対象項目が識別的となる。記

憶属性の付与により，実際の操作手順を確定させることができる。両属性は同時に与えられるので，この過程は前記写像 f および g の合成写像 fg と考えることができる。

$$fg : A \longmapsto D$$

この合成写像において実際の操作手順の生成は，属性の動的管理を維持することを前提とすると 繁雑なものになる可能性がある。データを処理するために実行時ルーチンを利用し，その引数としてデータバッファの先頭番地と対象データの記憶属性を保持した記述子の対を与えることにより，プログラム生成の簡潔化・属性の動的処理機構の維持を達成することができる。

記述子としては，1.1節に示した素情報の記憶属性（型，物理属性，使用属性）と実際に値がはいっている場所へのポインタがその要素として含まれていればよい。構造例を図2－16に示す。

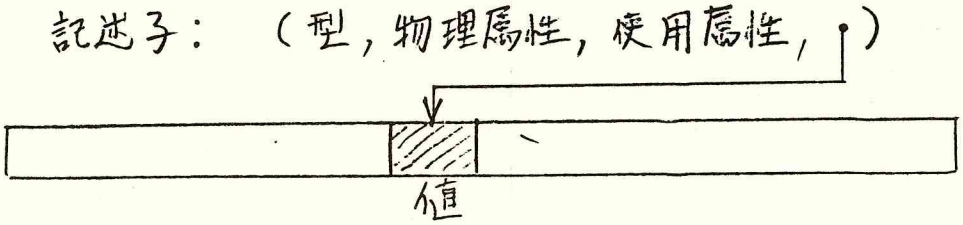


図2－16 記述子の例

2.5.2 均質データ空間への記述子の利用

記述子は2.5.1で述べたように，主に従来からあるファイル形式にもとづくデータ群に対して属性処理機能を付加することを目的としている。

したがってデータ自身の構造は，記述子の有無にかかわらず同一のものであることが望ましい。このために記述子はデータ群と独立に一括管理する。この概念を図2－17に示す。各記述子は3章で示した属性を 定められた形式で並べればよい。

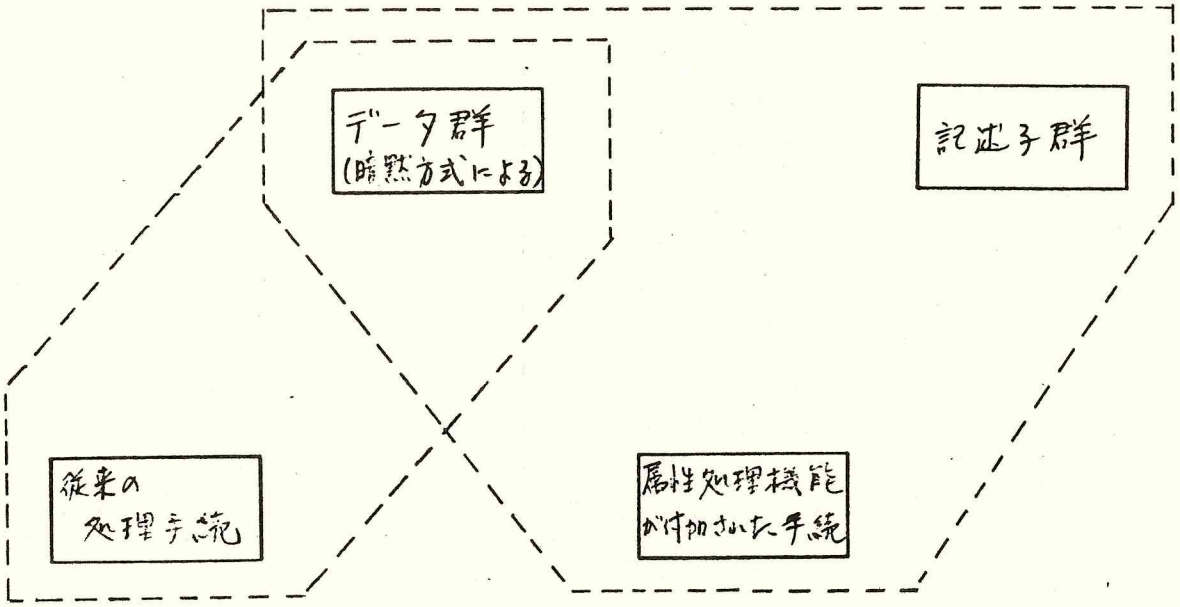


図2-17 従来の方式と共存できる記述子展開

記述子群には 必要となる属性値を前もって組み込んでおく。この組込みのためには 専用の機構がさらに必要となる。しかし，一度記述子群を定義すれば，それによって規定されるデータ群を利用する処理手続の記述の際には，その記述子群を引用しさえすればよい。それによって，データ構造の統一的な管理をすることができる。

たとえば商用データベースシステムの一つであるINQは，従来型データファイルに対してもデータベース的な処理を施すことができるように設計されている。そこでは記述子に似たものが用いられている。しかし，INQの方式では使用者が記述子状の定義体を直接処理・定義することができない。本論文では，データ構造の記述及び処理手続の記述などの設計製作段階から 作成されたプログラムの実行段階まで，一貫して属性を処理できる機構として記述子を扱っている。

2.6 研究成果の要約

データを属性空間中の点としてとらえる情報代数理論を発展させた半識別属性空間及び抽象データ空間について，その構造と各々におけるデータの処理機構概念をまとめた。

半識別属性空間理論は情報代数における属性の概念を論理属性と記憶属性にわけ，論理属性のみからなる座標集合に基づく空間を定義したものである。半識別属性空間は，データの概念から記憶属性をとり除いたものと考えてよい素情報を基本要素とする空間である。この理論に基づくデータ空間を電子計算機上にそのまま実現するために必要な機構として，セルとリンクリストを基本とする記憶域の構成法を，また半識別属性空間において基礎となる（対象，論理属性，属性値）の組を表現する連想子構成法を展開している。

次に，これらの概念を拡張した抽象データ空間理論をまとめている。抽象データ空間は素情報から論理属性をとり除いたものと考えてよい抽象事象を基本要素とする空間である。抽象事象を対象とする手続きを抽象手続と定義している。抽象手続によるソフトウェア開発をすることができれば，それらは機械の構造から独立しているだけでなくデータの論理属性からも独立させることができる。この具体的な機構については第4章で述べられる。その際に補助手続として必要となる記述子を本章で定義した。

第3章　属性処理システム設計原則

3.1 本章における研究の目的

本章における研究の目的は，第2章で述べた理論を実現する手法の設計に必要な設計原則を述べることである。

本章の内容は主に3つに分けることができる。すなわち，

1) 抽象データ空間の実現にあたって必要なシステム構成原則，
2) 従来より行われてきたデータ処理手続概念を抽象化し，抽象手続を構成するのに必要な抽象データ空間理論による形式化，
3) 会話形の属性処理に必要となる設計原則

である。

本章で述べる設計原則を用いた具体的な手法の展開は，第4章において記される。また実施例は第5章及び第6章においてまとめられている。

3.2. 支援システム設計原則

電子計算機上のシステムを設計するときに考慮すべき点は，次の3点である。

1. Machine Independent (機種独立性)
2. Data Independent (データ独立性)
3. Application Independent (応用独立性)

機種独立性とは そのシステムが実現されている，あるいは実現されるべき電子計算機の持つ特徴についての依存度である。一見すると機種依存度が高い程効率のよいシステムのように見えるが，技術革新の激しい電子計算機の場合，より高度な機能を求めて機械の構造，処理概念が劇的に変更されるかあるいは変更されなければならない場合が多々見られる。機械の構造に依存する部分を最小化することにより システムの寿命を，使用した電子計算機の寿命を独立させることができる。(図3-1，図3-2)

データ独立性とは 各属性値が実際に記録されている物理的な性質から，ソフトウェアシステム内部での処理をどれだけ独立することができるかという割合である。

データ独立性の低いソフトウェアシステムでは 各属性値の表現形式・レコード中での位置などが変更されるときには，それを利用している処理手続きも変更されなければならない。たとえば二進形式からパック十進形式への変更とか，32bit表現から64bit表現への変更その他である。

応用独立性とは そのソフトウェアシステムが対象としている応用に対しての柔軟性・拡張性である。たとえば給与計算システムにおいて処理できる手当の数・種類は部分的な修正をすれば任意に設定できるが，最大4つまでであるといった設計は，5つ以上の手当の種類をもつ給与体系には全く使用することができず，全面的な

設計変更をしなければならないことになる。また，書名，著者名による検索は許すが 掲載雑誌名による検索はできないという文献情報検索システムは，掲載雑誌名による検索を主とする業務形態に対しては全く無力となってしまう。

このように，ソフトウェアシステムはある対象に対するシステムではあるが，固定されたメニューしか持ちえない設計，あるいは機能を直接にシステム内部に保有してしまう設計では好ましくない。

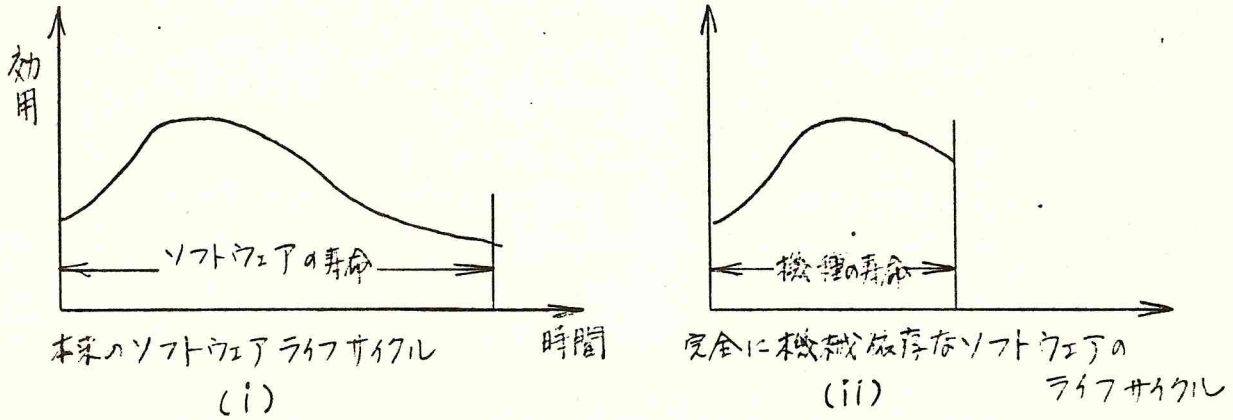


図3－1 ソフトウェアの ライフサイクル

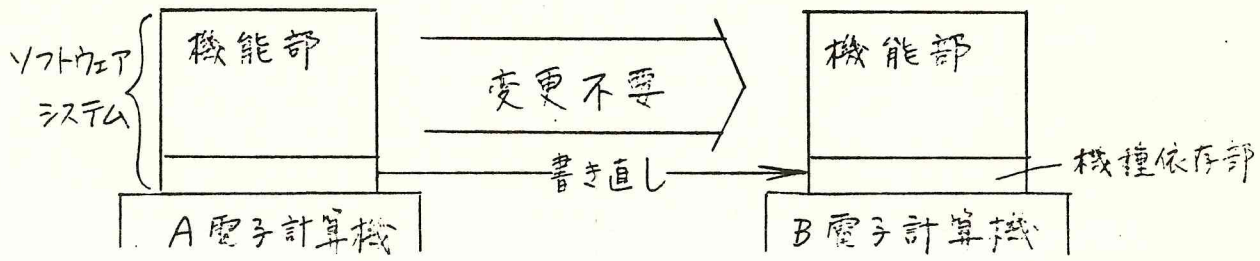


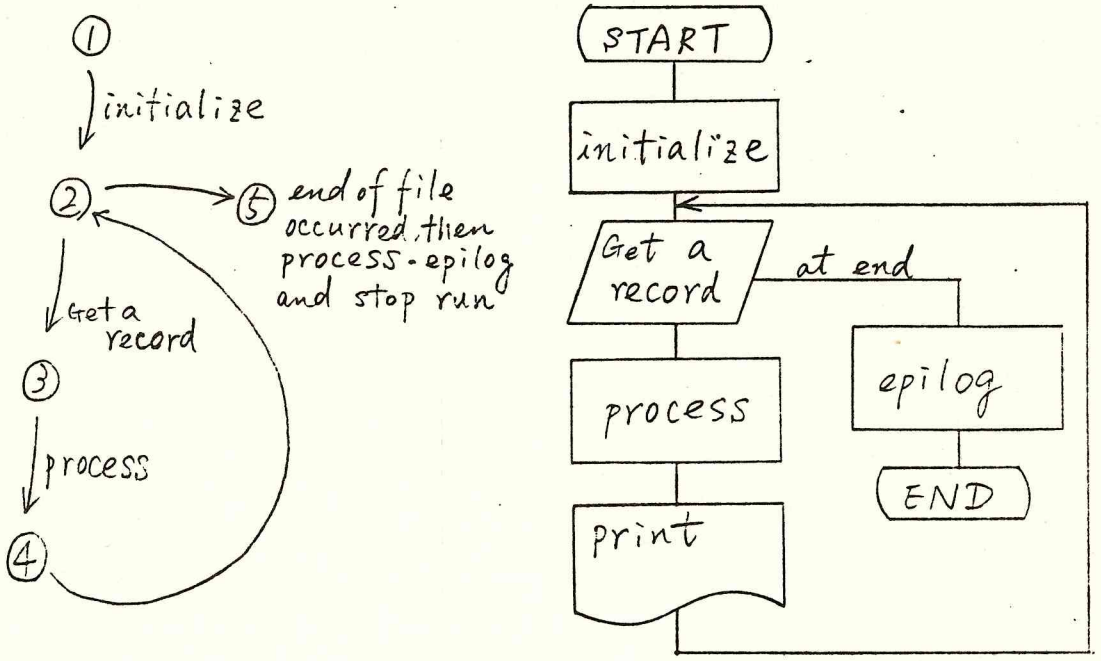
図3－2 機種独立なソフトウェアシステムの移行

3.3 三層分化プログラム構成法

前節までで考察された事項をもとに，組織的なソフトウェア生産管理に適したプログラム構成概念を展開する。

まずデータ構造を手続から分離する。次に手続きを類型化可能な部分とその他に二分割する。このことを例を示しながら導入する。

指摘された入出力パターンのうちMT→LPを例にとり，図にすると次のようになる。（図3－3）

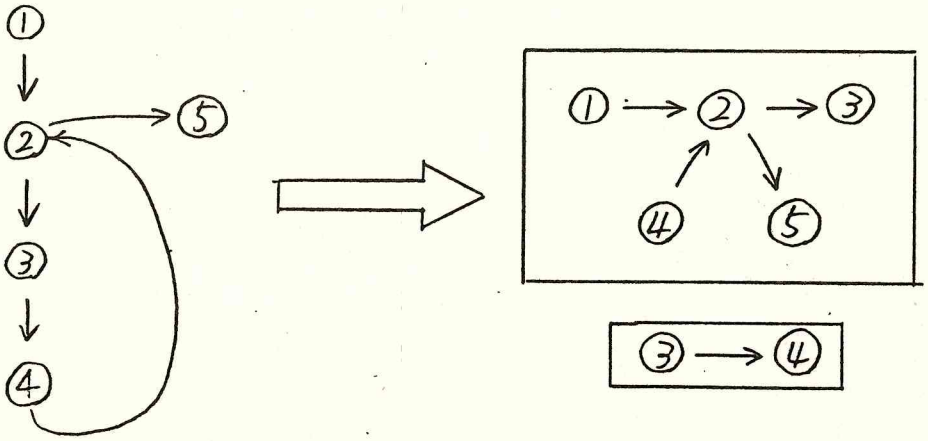


(1) グラフ表現　　(2) フローチャート表現

図3－3　単純なループ

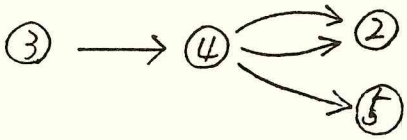
これには，初期化処理と終了処理が一般についている。入力ファイルから1レコードを読みこみ，それに対して処理を行う。このレコードに対して出力を行わないのであれば，印刷出力をバイパスしレコード読みこみを続ければよい。

プログラムによっては ④→⑤ ないしは ④→① といった経路が必要かも知れない。それらを自由につけ加えることができることが望ましい。このグラフからループを次のように分割する。



もちろん複数個の経路を許すから，必要に応じて

あるいは



などといった形で扱うことができる。
また，MT→MT，LP は図3－4のように表わせる。

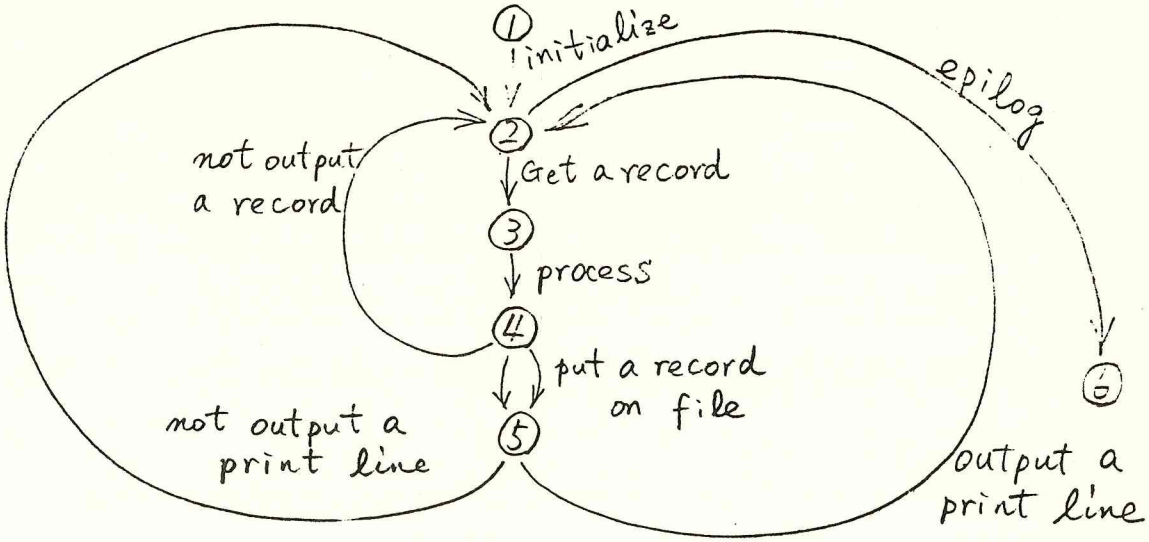


図3－4　1入力1出力のループ構造

その分割例を図3－5に示す。

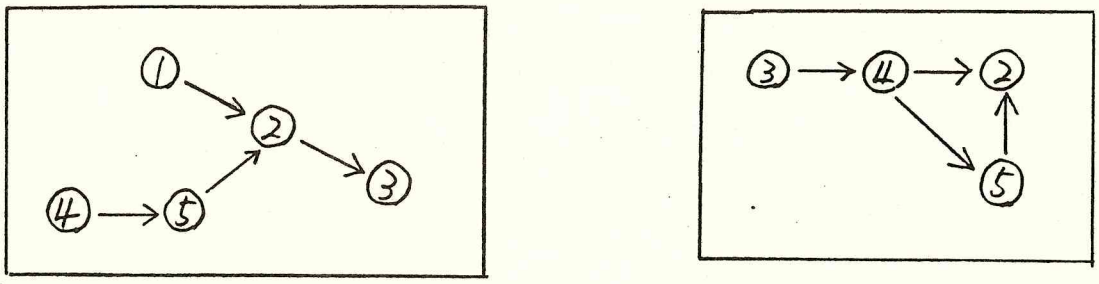


図3－5　1入力1出力ループの分割

他のものも同様にProcessの部分をカットし，入出力の一般処理を含む部分はprocess及びprocessの出口における行き先を含めれば完全なプログラムとすることができる。
この骨組となる原形構造をロジックユニットとよぶ。
　他方のprocessを中心とする部分グラフを細部項目処理部とよぼう。これらは原形構造であるロジックユニットを具体化するキーともなるので，生成パラメータとみなすこともできる。
2つの部分グラフの結合をプログラミングとよぶことができる。

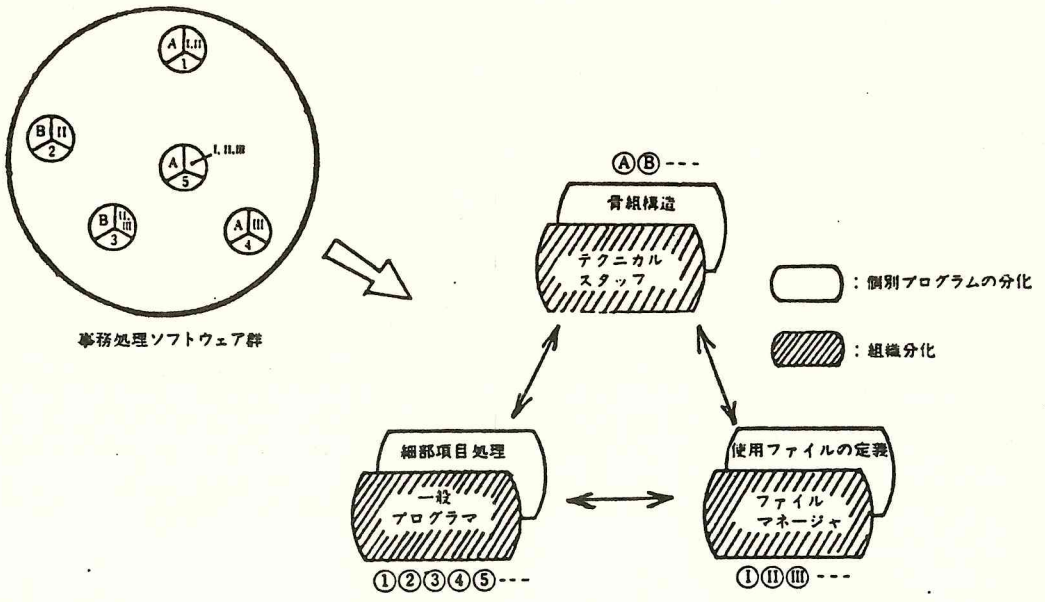


図3－6　機能の三分割と三層分化プログラム構成法

1つのプログラムを ①ロジックユニット，②生成パラメータ群，③データ構造の3部分に分けることができる。
各々は明確に分けることができ，また独立しているので，①ロジックユニットに対する管理者，②プログラム作成要求に応じてプログラムを生成するプログラマ，③データ管理者を独立におき，管理することも容易になる。

このときの個別プログラムの機能及び組織機能の3分割は完全に対応しており，個別プログラムを各々維持管理する場合に比べ，はるかに省力化を行うことができる。この関連を図3－6に示す。これを三層分化プログラム構成法とよぶ。

いくつかの実例によれば，ロジックユニットに吸収される部分は1つのプログラムに要する全ステップ数の50～80%程度となる。たとえば平均nステップのプログラムをn本持つ組織体で，それらがすべてk本にユニット化できると仮定すれば 全体で，

$$\alpha nm \text{ ステップ} \qquad (\alpha = 0.2 \sim 0.5)$$

の細部コーディングのみで済むことになる。

ロジックユニットの定義には 総じて $(1-\alpha)km$ ステップを要するので，

$$\begin{aligned} &nm - \alpha nm + (1-\alpha)km \\ &= m(n - \alpha n + k - k\alpha) \\ &= m(n + k - (n+k)\alpha) \\ &= m(n+k)(1-\alpha) \qquad \text{ステップ} \end{aligned}$$

を全体で 削減することができる。

またユニットの個数kはnに比して $n \gg k$ であるので
$n + k \approx n$ とみなすことができる。
したがって 上記削減量は $\underline{(1-\alpha)nm}$ と考えてよい。

3.4 処理要素

3.4.1 処理の類型化

昭和47年度に表3-1に示す2240本のプログラムについて類型化調査を行った[IDA 78]。

次の2種の集計を行った。

①入出力装置の使用区分による集計

②各プログラムの論理機能による集計

類型化がうまくいけば，それだけ個別プログラムの作成に要していた労力を減らすことができるはずである。

まず表3-2の規則を設定し，入出力装置の使用区分による集計を行った。その結果を図3-7に示す。13種のパターンで全体の95%を占めている。なかでも上位5種，すなわち，

表3-1 プログラム調査対象

種類	サブシステム数	プログラム本数	比率（本数）
銀行業務	12	1,316	58.8%
付随業務	6	415	18.5%
行内事務	13	509	22.7%
計	31 システム	2,240	100.0%

表3-2 集計規則

No.	内容
1	入出力の表記方法としてα→βと記す．ここでαは入力デバイス，βは出力デバイスを示す．
2	廃棄予定の近いシステムを考慮対象外とする
3	パラメータカードの有無は無視する
4	MT，ドラム，カードファイルの同一視
5	希少パターンの一般化による統合（図中のConversionは規則4の対象とならない周辺装置間の1入力1出力型処理を統合）
6	プリント出力有無の同一視

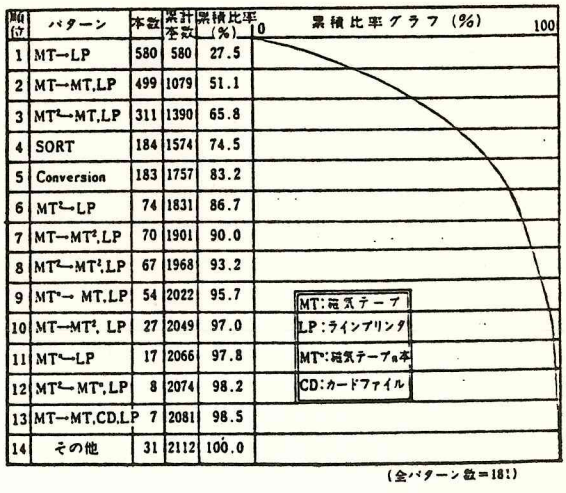


図3-7 入出力パターン調査結果

①単一ファイルの印刷（MT → LP）
②単一ファイルの編集及び印刷（MT → MT, LP）
③照合・統合等の2入力ファイルからの1ファイル出力及び印刷
（MT^2 → MT, LP）
④分類（SORT）
⑤1入力，1出力の媒体変換（Conversion）

のみで 83.2% を占め，ほとんどのプログラムがこれらのいずれかにはいっていることがわかる。

次に上記の分類を進めて論理機能による集計を試みた。論理機能は処理要素であり，この集計は意味がなかった。

これら2つの調査から次の2点を指摘した。

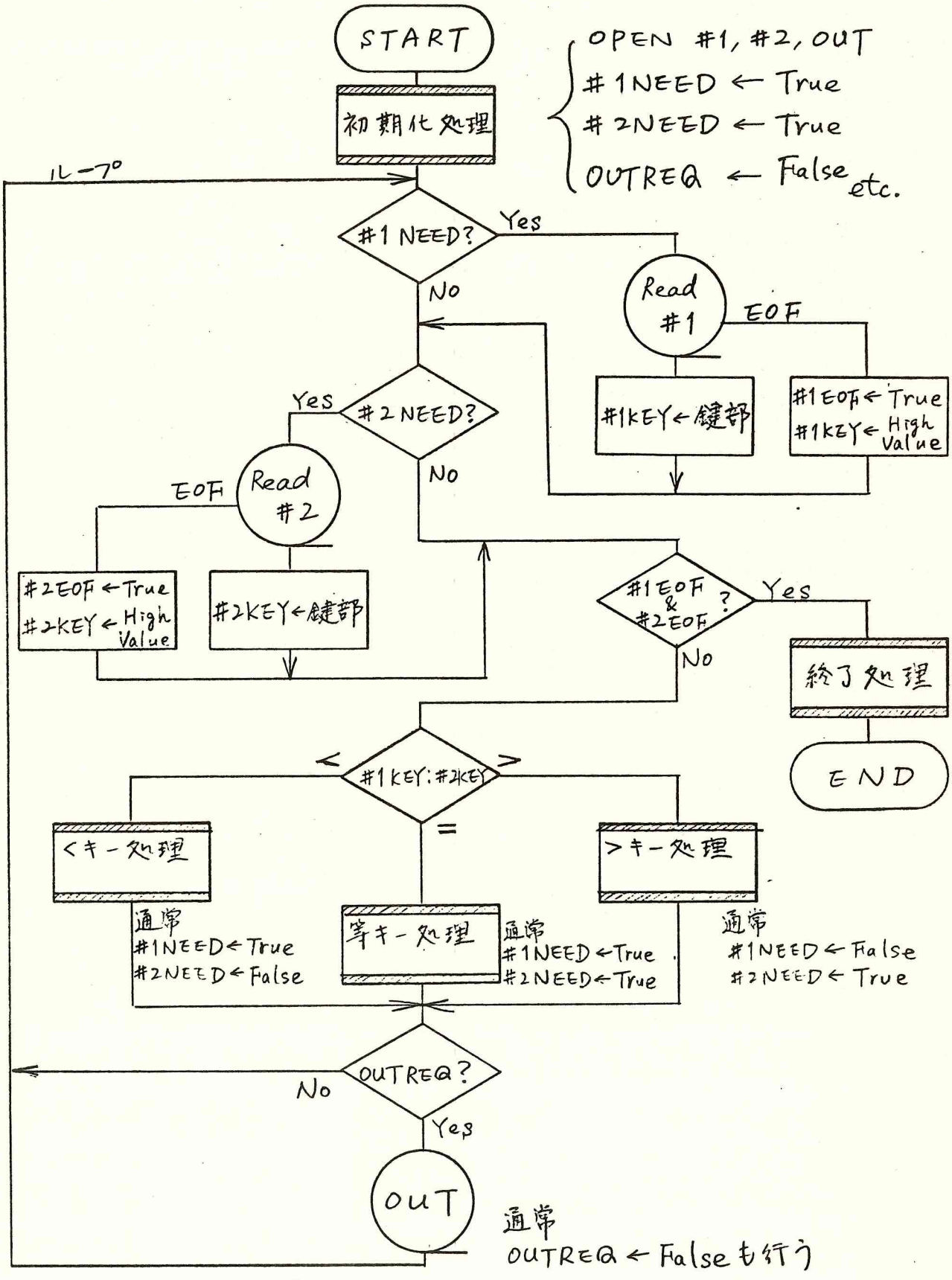
1) 論理構造の多様性：単能のプログラムは少なく，性質を一意に決定するのは無理がある。
2) 類型化可能な入出力処理：単純な入出力パターンのプログラムが多い。

そこで次のような方針に基づくソフトウェアの類型化の可能性を仮定した。

1) 入出力パターンを複数個に標準化する（ただし入出力パターンの拡張に対する補助手段は提供する）
2) 各プログラムのもつ複数の機能に対しては標準化ではなく生産性・管理性をあげるための記述手段の補助を提供する

たとえば マスター更新と併合・照合は2つのファイルから1つのファイルを作成する作業と考えることができる。この両者に共通になるようなフローチャートを 図3－2に示す。入力要求を行うべきか否かを決定するプログラムスイッチをおくことにより，共用することができる。図3－8は Data Driven Loop を形成している。この流れ図のうち□で囲まれた部分をかえることにより任意の機能を実現することができる。そのとき他の部分を変更する必要はない。

図3－8　2入力1出力の標準フローチャート



マスター更新の場合には#1をマスタファイル，#2をトランザクションファイルとして考える。このとき“＜キー処理”は更新対象マスターレコードにまだ出会わない状況である。したがって，そのときには#1（マスタファイル）を入力するスイッチをたて，ループを繰り返す。“＞キー処理”は，#2（トランザクション）のキーが小さくなっている場合であり，通常はトランザクションレコードの挿入（追加）処理が行われる。その後に#2を入力するスイッチを立て，ループを繰り返す。“等キー処理”はトランザクションによる修正処理を通常意味している。

照合・併合の場合，キーの等しいレコードに対して処理を行うのが通常であるので“＜キー処理”及び“＞キー処理”は，一方のファイルを読みすすめるためのスイッチの設定のみで十分なことが多い。

このように考えてきたとき，図3－8のような標準的なパターン，いいかえれば，あちこちに未コーディング部分のある流れ図を記憶させておき，それを後で引用して未コーディング部分を自動的に，あるいはハンドコードで埋めることができれば プログラミングの省力化を達成することができる。これをロジックユニットと呼ぶ。

ロジックユニットで類型化されるものは，形式論におけるバンドルに相当するものである。

3.4.2. 均質データ空間における処理要素

本節では，ロジックユニット化可能なファイル処理の共通パターンを 抽象データ空間理論に基づき，形式化している。形式化可能であることは，対象となるデータの論理属性及び記憶属性に独立な手続を構成できることを意味している。すなわち，形式化された処理要素は 2章に示した抽象手続に等しい。そこで，①マスタ更新，②媒体変換，③並べかえ，④編集・演算，⑤レポート印刷，⑥併合・照合 の6つの処理要素について考察する。

これらの処理要素を単独に，またそれらの複合概念を必要に応じてロジックユニットとして記述し，維持管理する。本節に示す形式化により，抽象手続の実現が可能となる。

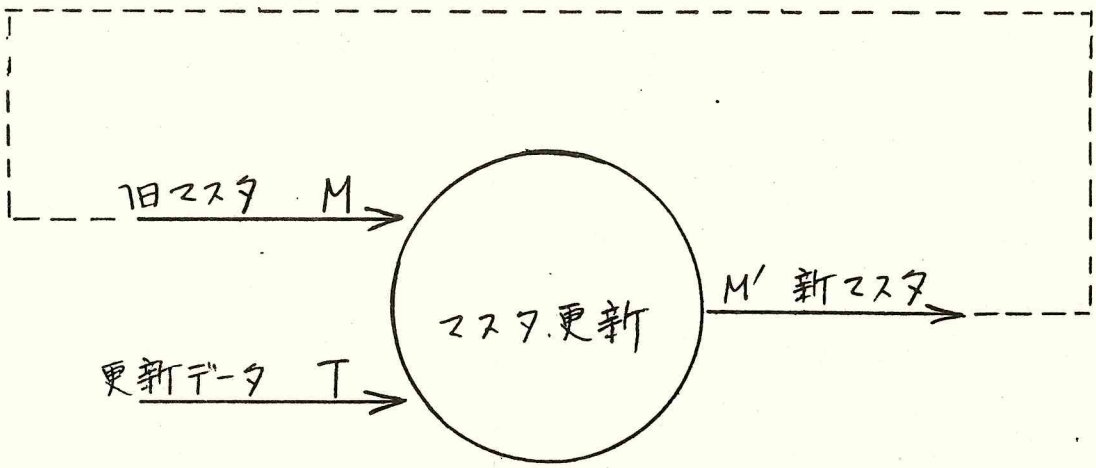


図3-9　マスタ更新

①マスタ更新（図3-9）

事務システムにおいては　主帳簿となるファイルが存在する。たとえば「入試処理システム」に対しては，受験生ごとの「受験番号，氏名，英語，数学，国語」のデータを集めたファイルをマスタファイルとする。

マスタファイル中に誤りが発見されたとき，あるいは追加・削除・修正を行うときにマスタ更新が実行される。
マスタ更新は　一括処理型のファイル処理において，中心的な役割を果たしている。

MとTとの対応は　キーとなる属性に対する属性値の一致により表わされ，バンドルの生成関数bにより決定される。この対応はファイバー積を利用して　次のように操作的に定義することができる。

$$(M, h_1) \times_K (T, h_2) = \{(m, t) \in M \times T \mid h_1(m) = h_2(t)\}$$

$h_1 : M \to K$, $h_2 : T \to K$

Kは鍵のとりうる値からなる集合とする

ファイル更新へのバンドルの応用

マスタファイルに対する挿入，削除，すなわち他のファイル中の照合されたレコードから得られた情報を使って マスタファイルのレコードを修正することは，ファイル更新として一般に知られている。

A_n：マスタファイルのレコードを表わす点からなる領域

A_1：追加レコードの集まり

$A_2, A_3, \cdots, A_{n-1}$：削除又は修正のためにマスタファイルのレコードと照合するトランザクションレコード(TX)のファイル

A_n^{new}：A_nを更新した結果の領域，すなわち更新されたマスタファイル

$$A_n^{new} = Up(b; F_1; A_1, \cdots, A_n)$$

ここで，

b：$A_2, A_3, \cdots, A_{n-1}$からのレコードと$A_n$のレコードを照合するために定義されたバンドルの生成関数である。これはバンドル$B(b, A_2, A_3, \cdots, A_n)$を作り出す。

F_1：FOBで個々の更新操作ごとに異なる。マスタをTX($A_2, \cdots, A_{n-1}$)のレコードで修正するための規制を定義している。

$$M = F_1(B)$$

レコードの削除はFOBがいくつかの線に零点(Ω)を与えることによって表現する。

このとき，

$$A_n^{new} = Up(b; F_1; A_1, \cdots, A_n)$$
$$= A_1 \cup F_1(B) \cup I_n'(A_n)$$

$I_n'(A_n)$：$I_n(B; A_n)$の補集合。($I_n(B; A_n)$とはバンドル関数F_1とA_nとの共通部分を表わす。)修正されなかったレコード。

$F_1(B)$：修正されたレコード

A_1：追加レコード

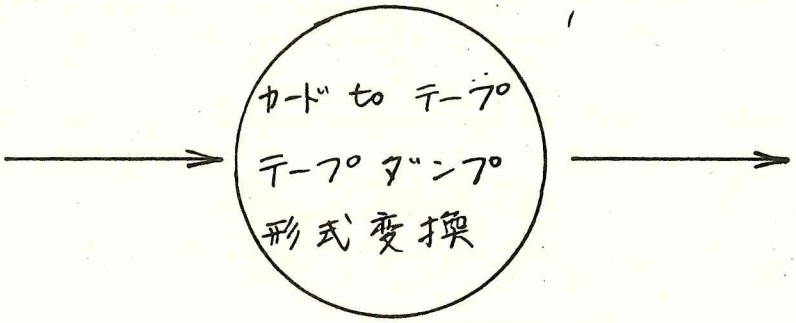


図3-10　　媒体変換

②媒体変換（図3-10）

マスタ更新の次に，マスタファイルの初期作成，ファイル内容のそのままの形での印刷（ダンプ），コード変換やブロック長などの形式変換などが システム運用のための重要な役割を果たしている。

それらは“情報の意味的な変換を含まないデータの転送”としてまとめられる。

媒体変換は 論理属性を変換させない変換であり，半識別属性空間においては形式的には意味をもたない。

③並べかえ（ソート，図3-11）

主に後続する処理を助けるために ファイルを特定の順序に並べかえる作業機能である。たとえば，受験番号順にならべられているファイルを英語の成績の順に ならべなおすことなどである。

一括処理形態でなく，集中化されたファイルを即時処理する場合には 並べかえは本質的に不要である。

並べかえは ファイル中のキーの公理順序に従う順序づけである。並べかえ対象ファイルを，領域A_1 キーをkeyとよぶと，次のように表わすことができる。

並べかえとは 長さ1の順序つき線関数fによる領域A_1の順序づけ $A_2 = \theta(f, A_1)$ である。 fは すべての$P_i \in A_1$のkeyに対して自然数を対応させる関数とする。

$$f: key \longrightarrow n$$

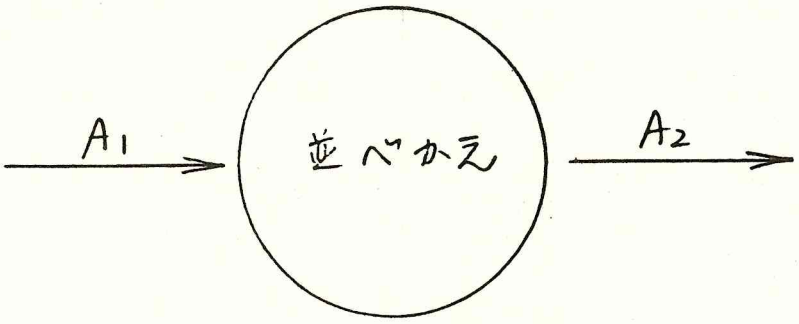


図3－11　並べかえ

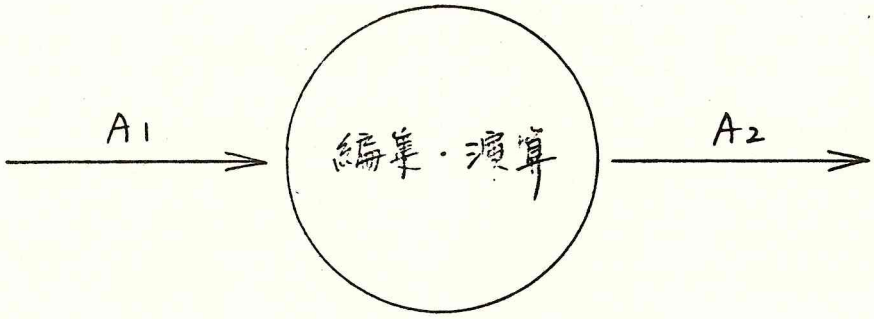


図3－12　編集・演算

④編集・演算（図3－12）

ファイル中の情報の意味的な加工を指す。入試処理を例にとると，総得点の計算や得点分布の計算及び それらに基づくレコード内の情報のセットなどである。

編集・演算は A_1 上での層により形式化できる。

$$A_2 = H(g\ A_1)$$

$$H \equiv \begin{cases} g_1' = f_1 \\ \vdots \\ g_k' = f_k \end{cases}$$

ただし，g_i' は A_2 中の属性

f_i は A_1 に対する領域関数

g は 抽出操作のために 抽出対象の鍵となる A_1 中のある g_{key} に対して真となる，長さ1のブール線関数である。

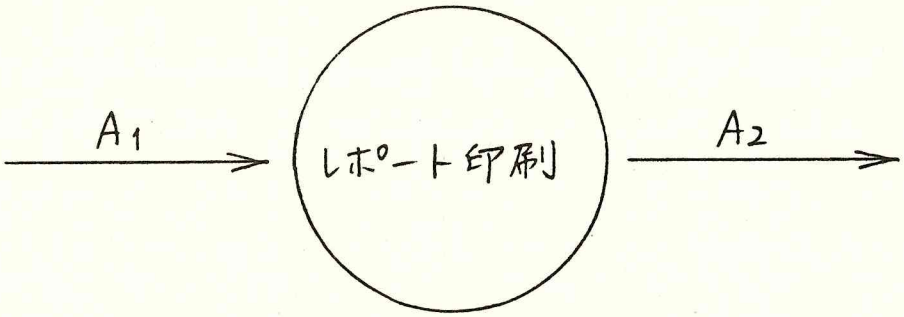


図3－13　　レポート印刷

⑤レポート印刷（図3－13）

ファイル上の内容を人間に見易い形で表示する機能である。レポート印刷プログラムは　適当な見出しをつけ，帳票用紙上にファイルからのデータを並べ，計をとり表示する，などが主要な機能である。

しかし，作業上の問題（たとえば　50万枚まで印刷したところで故障がおきたときには，JOBの最初ではなく　その直前の状態から再スタートできることが望ましい。これらはチェックポイントリスタート機能といわれる。）や，帳票レイアウト上の問題（たとえば下図のような複雑な書式）などから，難易度の高いプログラムも時として必要である。

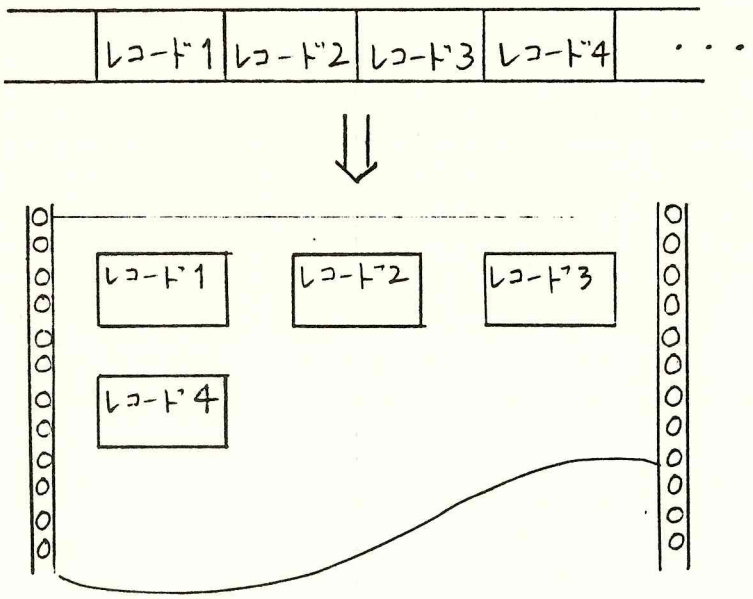


図3－14　　特殊な出力形式の例

レポート印刷の処理要素は，見出しづけ，全レコードの編集出力，多段階集計などの手順よりなっている。このうち多段階集計が本質的な作業である。

多段階集計を行うことのできるファイルA_1は，

$$A_1 = O(f_1, O(f_2, \cdots, O(f_n, A)\cdots))$$

により前もって順序づけられているものとする。

f_iは各データ点の属性g_jについて順序づけをするための長さ1の線関数である。

このとき出力A_2は層関数を用いて，次のように定義できる。

$$A_2 = A_{21} \cup A_{22} \cup \cdots \cup A_{2i} \cup \cdots \cup A_{2n}$$

$$A_{2i} = H(g_l, A_1)$$

$$H \equiv \{ g'_j = \Sigma g_j$$

ただし，g_lはf_{i-1}で対象とする属性，

g_jはf_iで対象とする属性とする。

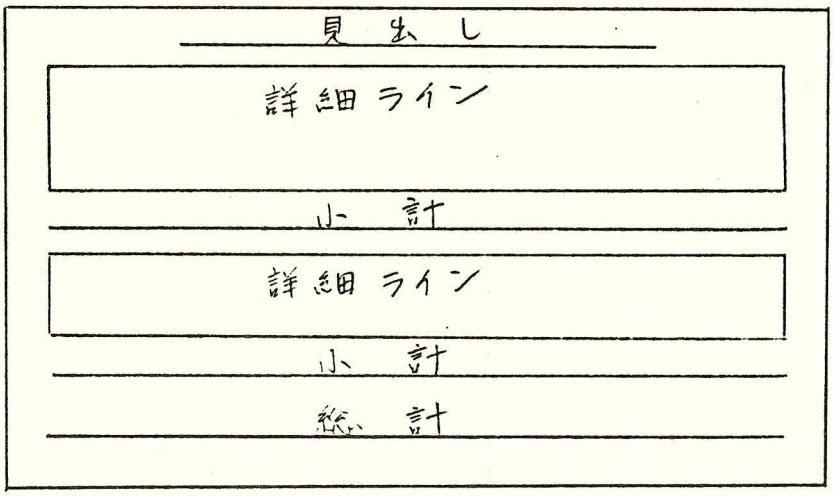


図3-15　出力レコードの形式例

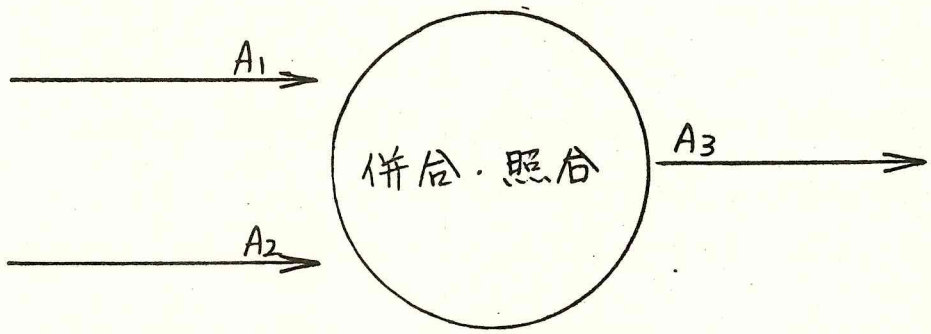


図 3－16　併合・照合

⑥併合・照合（図 3－16）

同じレコード列に対して 異なった情報をもつ2つ以上のファイルに対して，それらの間の情報を併合ないしは照合させる機能である。

たとえば，入試マスタに対して各受験者の受験番号と 住所をもつ住所マスタを併合する作業などがある。

照合は 次のように形式化できる。

$$A_3 = F(g_{1k} = g_{2k} ; H_1(f_1, A_1), H_2(f_2, A_2))$$

f_1及びf_2は各々キーとなるg_{1k}, g_{2k}の存在による類別のための関数である。

併合は ①のマスタ更新における定義と等しい。

3.4.3　不均質データ空間における処理要素

不均質データ空間の直接的な実現では，対象ファイルは乱アクセス構成であることが必須である。乱アクセス構成のファイルは，ファイルの更新のための全レコードの参照は不要であり，順アクセス構成のファイルに比べて維持の高速化がはかれる。

このとき，媒体変換・並べかえ・併合・照合などの概念は存在する必要がない。（正確にいえば，一般の利用者にとってそれらが不用となるように設計される。）

また，必要なデータを利用者が見るためには一括処理形態ではレポート印刷が存在する。即時処理形態では，必要なデータだけを即時に端末から参照し，表示すればよい。

これらの対応関係を次表に示す（表3－3）

表3－3　処理要素の対応

均質データ空間における要素	不均質データ空間における要素
マスタ更新	オフライン保守 （運用管理．一般利用者は行わない。）
媒体変換	——
レポート印刷	照会
並べかえ	——
編集・演算	更新 （レコードの削除・追加・修正）
併合・照合	

3.5 会話型不均質データ空間処理システム設計原則

3.5.1 集中化ファイルと不均質データ空間

データを集中管理する方式は最近よく見られるようになっている。集中化されたファイルは同質の情報のみからなっている場合は少なく，不均質データ空間と考えられる場合が多い。

現在普及しつつあるデータベースはその代表例であり，その中で対象となる事務システムで使用する情報自身に着眼し，それらを統一的に管理する。その結果個別的なプログラム作成の無駄を省くことが可能となり，EDP部門自身のシステム改善及び電子計算機上に記憶されたデータの現実の値への追従性が向上する。

データベース的な考え方は，多少複雑であっても情報の高速参照能力やその整合性・完備性への対処をシステム側に持たせることができるので，情報の検索・維持・格納に対して使用者は要求を簡潔に記述するだけですむメリットがある。

しかし既存のデータベースシステム自身は 使用する環境及び所要記憶域量の点で大規模なプログラムであり，簡単に実現することは不可能である。小規模な応用ないしは実験的な導入においてはデータベースは高価すぎるといえよう。

データベースの本質を事務情報の効果的な蓄積と参照機構の提供とみなしたいとき，これらを比較的小さいプログラムで実現できるならば その適用分野を広げることができる。以上のような認識に立ち，モデルと処理系の実現を考えるならば，コンパクトに作成でき，かつ使いやすくなり，有用性は高い。

3.5.2 会話処理システムの設計原則

1. Reliability ：信頼に足るシステムにしたい
2. Availability ：極力早期にシステムを完成し，実用に供したい
3. Simplicity ：構造を極力簡潔化し保守性を高めると同時に，
   system overhead の軽減をねらう。ハード面でもボードの改良など。
4. Helpfulness ：会話型処理を高効率に進めるための機能，実用上
   充分な数の組込関数の具備，atom の car，cdr のガードなど。
5. Adaptability ：ハードウェアの進歩・ソフトウェアの改良・拡張
   に対する適応性・拡張性。

3.6 研究成果の要約

属性処理システム及びその中心となる属性処理機構の開発に先立って設計原則を展開した。

3.2では，データ・処理手続，そしてシステム自身に対しての抽象化概念を，機種独立性，データ独立性，応用独立性の3つの性質に分解し，これらをすべて満たすことを設計原則として示した。

3.3では，3.2で示した設計原則にもとづいて，組織的なソフトウェア生産管理の行えるプログラム構成法として，三層分化プログラム構成法を述べている。三層分化プログラム構成法は手続を共通化可能な部分と個有の部分に二分し，あわせてデータの記述を手続から分離し，個別プログラムの作成にあたっての機能分化を可能としている。この機能分化に対した支援システムを設計することにより，組織的な機能分担がはかれる。このときの省力化の効率は，

$$(1-\alpha)nm$$

として表わせることを示した。

3.4では，データを処理する際の処理要素の形式化を，情報代数を基礎とした抽象データ空間理論により行い，ロジックユニットとしての類型化が現実的にも充分可能であることを示した。あわせて，不均質データ空間を対象とする処理要素についてまとめた。

3.5では，会話型不均質データ空間処理システムの設計に必要な原則として，

Reliability, Availability, Simplicity,
Helpfulness, Adaptability

を満たすことが必要となることをまとめた。

第4章 素情報属性の処理機構の設計

4.1 本章における研究の目的

本章における研究の目的は，第3章において述べた設計原則に基づき，半識別属性空間の実現，抽象データ空間及び抽象手続の実現に必要な機構を設計することである。

4.2節では，半識別属性空間の基本要素である素情報属性を保持する連想子について　その処理機構がまとめられる。

4.3節では，抽象データ空間の基本要素である抽象事象に対する記憶属性の確定に利用される記述子について，その構造とデータ処理時における手続との関連についてまとめられる。

4.4節では，抽象手続を実現するに際して必要な記述言語と，その際に必要となる機構についてまとめられる。

4.2 連想子及び属性の処理手順の設計

(a, O, v) 型連想子に対しては，次に示す手順により(a, O)に対する値 value を求めることができる。

```
procedure find (attribute, object, value);
  function addr;
  integer i;
  sparse-array vtable[m]    "m >> n";
  begin i := addr[attribute, object];
    if i = NIL then value := NIL
              else value := vtable[i]
```

addr は引数として与えられる(attribute, object)の対に対して一意にアドレスを決定する関数とする。

addr 関数は ハッシュ法とよばれる技術をその中核においている。ハッシュ法は文字通り与えられた文字列に対して，「細かく切りきざんでごちゃまぜにする」ことにより，定められた区間内の整数に対応させる技法である。その整数は ハッシュ領域とよばれる表のアドレスとして用いられる。ハッシュ法は，たとえば[MOR 68]などにまとめられている。

ハッシュ関数の場合，高速性を重んじ文字コードに対する単純な四則だけで済ませることが実用上一般的である。このため文字列と整数との間の写像の単射性は保証されず，関数値の衝突が生じてしまう。(しかし衝突の可能性を低くする工夫が実際には行われている。)衝突の対策としては チェイン法，リハッシュ法，バケットハッシュ法などが知られているが，その全体効率からリハッシュ法を採用している。

リハッシュ法は値の衝突時に，その値をもとに再ハッシュを行うものである。衝突の確率は ハッシュ領域中の連想子総数の比率（ロードファクタρ）により 定められることがわかっている。筆者の方法による平均探索回数（参照値が発見されるまでのハッシュ回数prob の平均E(prob)）は，

$$E(prob) = -\frac{1}{\rho}\log(1-\rho) \qquad (\rho\text{は負荷率})$$

によって与えられ，ρが80%であっても約2回で済むことが確率的にも，また実測上もわかっている。たとえば 1万個の情報も1万5千語程度のハッシュ領域に保持させることにより，各項目を他の手法と比較して高速に平均約2回の探索で参照でき，それは要素数に依存しない。またハッシュ領域は 少量（数万程度まで）なら主記憶上に，多量ならディスク等の補助記憶に割り当てることにより，このモデルの定量的な制限をはずすことができる。

このとき，関数addrは次のように作成できる。

```
function procedure addr[a, 0];
  sparse-array hash-area[m];
  begin i:= hash[a] + hash[0];
    while occupied[hash-area[i]] do
    begin if equivalent-key[hash-area[i]; a; 0]
          then return[i];
    i:= rehash[i]
  end;
return[NIL]
end;
```

occupied[x]はハッシュ領域中の要素xに連想子がはいっているなら真，さもなければ偽を返す関数，hashは図4－1に示す関数，equivalent-key[x; y; z]は要素x中の鍵が(y, z)と等しいならば真，さもなければ偽を返す関数とする。

このaddr関数はfind手続中のみでなく，情報の登録時の手続においても使用される。

ALPS/I（第6章）において，この部分は約200ステップ程度で実現できている。（他の計算機でもその程度で作成できる。）さらに，一意性を必要とする情報をすべて連想子として扱うことにより システム全体の統一性が保たれ，コンパクトに実現できる。

また，

(1) いわゆる変数。識別名称の文字列を鍵とする素情報連想子とよばれる基本連想子

(2) ハッシュ化配列連想子とよばれる連想情報モデルの中核をなす

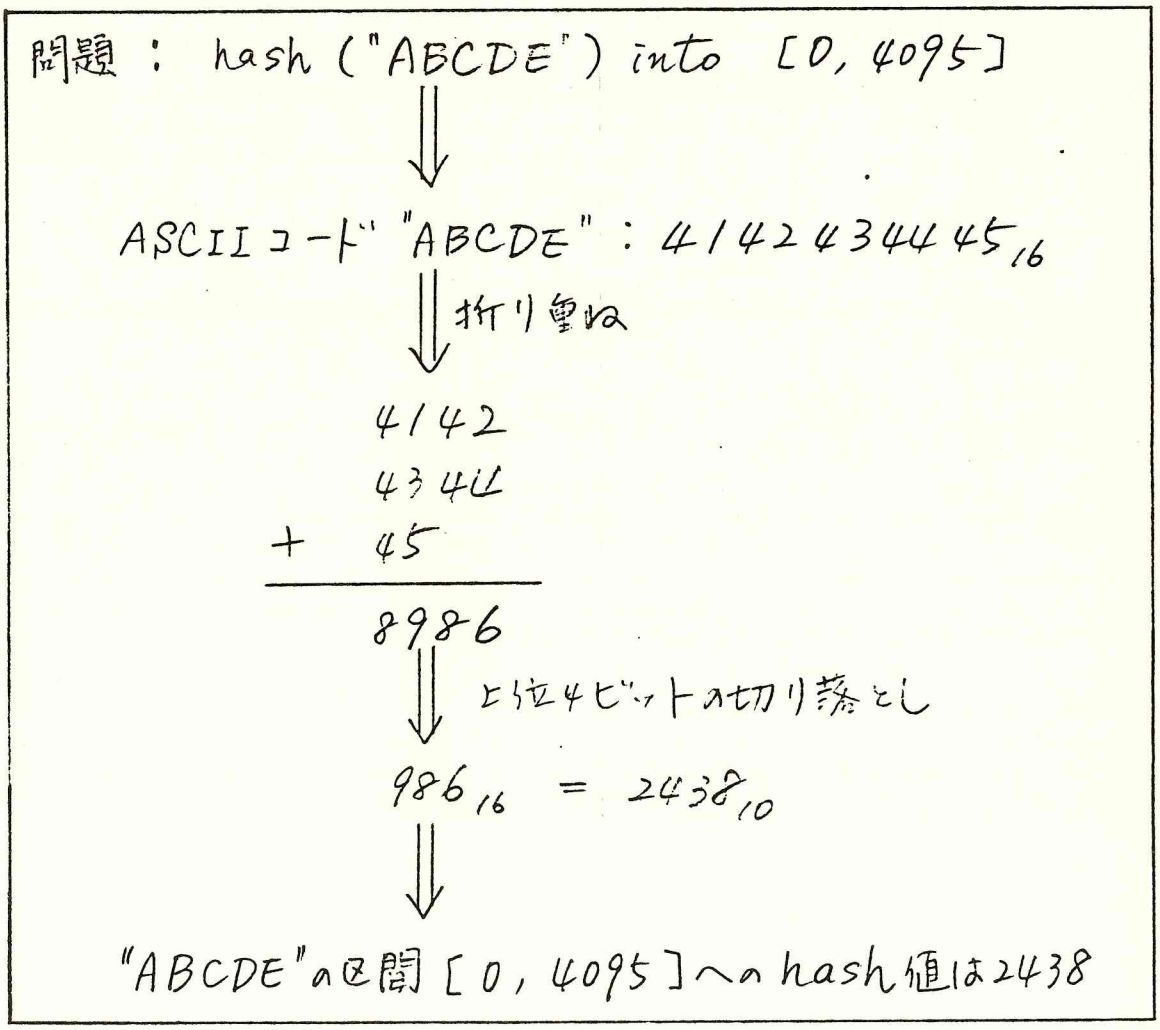


図4－1　ハッシュ技法の適用例

連想子

(3)連想計算連想子

の3語を図4-2のように統一的に扱うことができる。

連想子はハッシュ化配列(hashed array)として利用できる。「属性(attribute)」をハッシュ化配列名とよび,「対象(object)」はその引数(添字)とに表記する。これは疎な配列の実現方法としても使用できることからつけられた名称である。(たとえば1万×1万の疎な行列も連想子表現により実際に値を入れられた要素分だけの記憶域の使用で済む。) ハッシュ化配列の添字(「対象」)

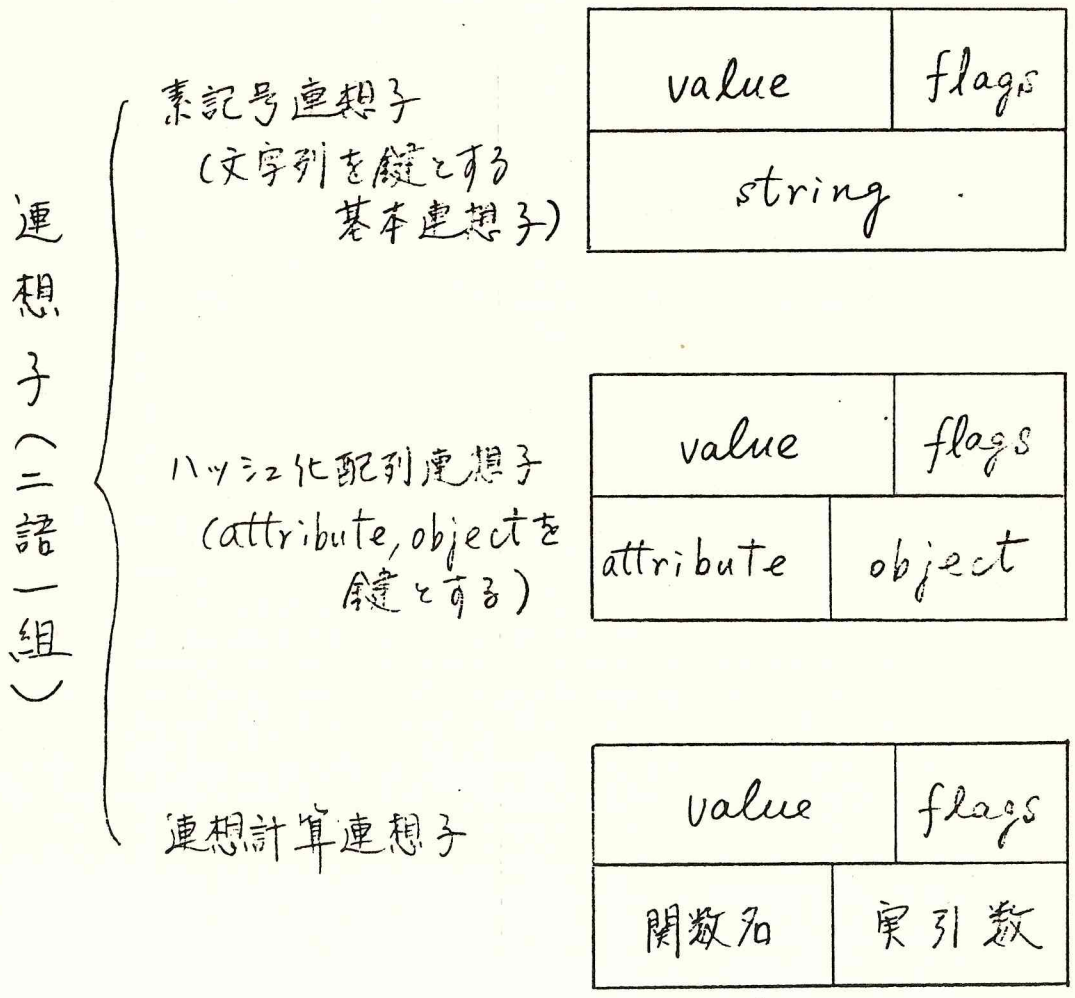


図4-2 連想子の構造

はいうまでもなく 数字，文字列，リンクリストのいずれであってもよい。用意された基本機能は 次の通りである。

array [a_1 … a_n]	a_iをハッシュ化配列名として宣言
dearray [a]	ハッシュ化配列名aとそれら属性部にもつ連想子をすべて除去
seta [a; o; v]	属性部にa，対象部にo，値部にvをもつ連想子の作成
deleta [a; o]	属性部にa，対象部にoをもつ連想子を除去
a [o]	鍵に[a, o]をもつ連想子の値ひを返す

これらは前節に示された実現手続により組込まれている。

ファイルをハッシュ化配列を用いて格納することにより，各項目の高速参照が可能であり，さらにその連想子をリンクリストで結合することにより連想子を階層的に構造化することができる。たとえば，(学生氏名，英語得点，数学得点)から成る成績ファイルの宣言は，

array [ENGLISH; MATHEMATICS]

ある学生の英語得点のセットは，

seta [ENGLISH; name; grade]

その参照は，

ENGLISH [name]

を実行させればよい。

4.3 記述子による属性処理機構の実現

4.3.1 記憶属性の記述

記憶属性は すべて独立して管理できる。もっとも簡単な記憶属性の管理方法は4.2節に示した方法であり，記憶属性を画一化することである。事務処理に対する場合 こうした方法は困難であり，このための記述子が導入されている。記述子として格納が必要な記憶属性は位置・長さ・型ですむ。

ファイル，いいかえれば属性空間中の領域，の各々について記憶属性値と論理属性値（名前）との対を記述し独立して管理することにより，プログラム作成時において繁雑な処理の必要な記憶属性を意識する必要がなくなり，また自動的な属性対処機構を利用することが可能となる。

記憶属性の記述は 階層的な記述を行うことにより，位置記憶属性の直接的な記述を避けることができる。事務処理用のCOBOL言語のDATA DIVISIONの記述に準じた方式により記述することが，その典型的な例である。第5章に後述するFAST1システムでは固定カラム形式のシート記入方式が採用されている。一例を図4－3に示す。またFAST1において許された型を表4－1に示す。FAST1における方式では，たより順に 階層レベル・項目名・長さ（語長及びビット長），型を記述する。図4－3には含まれてはいないが，型を記述する欄の右に配列型であれば配列の要素数を記述することができるように設計されている。また，階層レベルを77と指定し，同一の場所に対して異なった論理属性値・記憶属性値の対を定義することができる。これはCOBOLでは REDEFINEとよばれる機能である。

また階層レベル00には データ構造名称，全体の論理レコード長，物理レコード長（ブロックサイズ）などを記述することができ

る。このことにより記憶属性の定義を完了させ，データ処理実行時において，データファイルを参照するための補助情報をあわせて保持することができる。

データ構造名称は，生成時に使用する原形手続中で利用されるファイル識別名称として用いられる可変シンボルに結合される。データ構造定義中の各変数名は生成時においてすべて，レコードバッファの先頭からの相対位置及び属性コードにおきかえられる。型変換及びマッチングを含む操作中にそれらが利用される場合，実行時ルーチンのよび出しが生成され，その引数として相対位置及び属性レコードが渡される。

```
$LAY, C
00  CARDF          14    14    CARD
01  ALL            14    0     DSP
02  KEY            1     0     ZDC
03  A              3     0     DSP
03  B              10    0     DSP
99
```

図4－3　データ構造の定義例

表4－1　型の例

型	意味
BIN (Binary)	True Binary data
SBN (Signed Binary)	補数表現による Binary data
DSP (Display)	プリントイメージの数値データ（前ゼロの消去，編集記号の挿入あり）
ZDC (Zoned Decimal)	数字コードよりなるデータ
CHA (Character)	文字データ

レベル00で定義されたファイル構成情報は　すべて原形手続きの中で引用することができる。
それらは　次表のような形でファイル識別の可変シンボルに対する修飾形で引用できる。（表４－２）

表４－２　　ファイル編成情報

可変シンボル	意　味
&α	そのファイルのレコードバッファ名
&αL	外部ファイルの場合のラベル名
&αF	外部ファイル名称
&αV	ファイルが存在する装置名称
&αR	レコード長
&αB	ブロック長
&αA	データ構造定義名称自身

αはファイル識別名称を示す

4.3.2　データ処理時における記憶属性の確定

データの転送及び演算は　次のような形式の実行時ルーチンcallとする。

(例)　CALL MOVE (i, .i, j, .j)

i：送り側領域先頭番地
.i：iの為の記述子
j：受け側領域先頭番地
.j：jのための記述子

iとjの間の属性の違い・妥当性のチェック等は　転送実行時ルーチンにおいて処理される。

属性の組合せ判別を高速化することが必要な場合には，MOVE中にすべての組合せのための副エントリを用意し，そこへのCall命令にすればよい。

また配列型データの要素参照の場合には，i及びjには配列の先頭番地を常に指定し，記述子中もしくは次パラメータ以降に添字値を指定する。この方式により実行時における添字の妥当性チェックも合わせて行うことができる。

またデータ構造は1つのレコードに対する各属性値のならびを記述するようにされている。

このため領域の先頭番地は実際のデータが扱われる領域ではなく，それが属するレコードの先頭番地を示す方が，処理系作成上は好ましい。この実現例は次5章に示す。

ここでは概念的な説明を行う。図4－4に例を示す。2つのファイルdata-set1とdata-set2の間の部分的な転送を例にとる。data-set1中に24文字を占める変数Iがあり，これをdata-set2中の18文字を占める変数Jに転送したいものとする。このとき，長さが異なるので下6文字の切り捨て処理が行われねばならない。また，もしJの方が大きな文字数を占めるならば，空白を残りにつめる処理が行われねばならない。これらの処理はすべて実行時ルーチンMOVEが行う。MOVEには，もし必要ならば属性の変換，たとえば二進数のデータを十進表現のデータ形式を仮定している変数に送る場合には，相当する変換が行われる。

事務処理のための記述子には最低限次の3つの事項が含まれていればよい。

(1) 先頭番地
(2) 長さ
(3) 型属性　(二進，十進，文字，無型，その他)

この場合，先頭番地をそのファイルレコードの先頭番地からの相対位置により表現することにより実際に実行時ルーチンを作成するのに便利である。(図4－4)

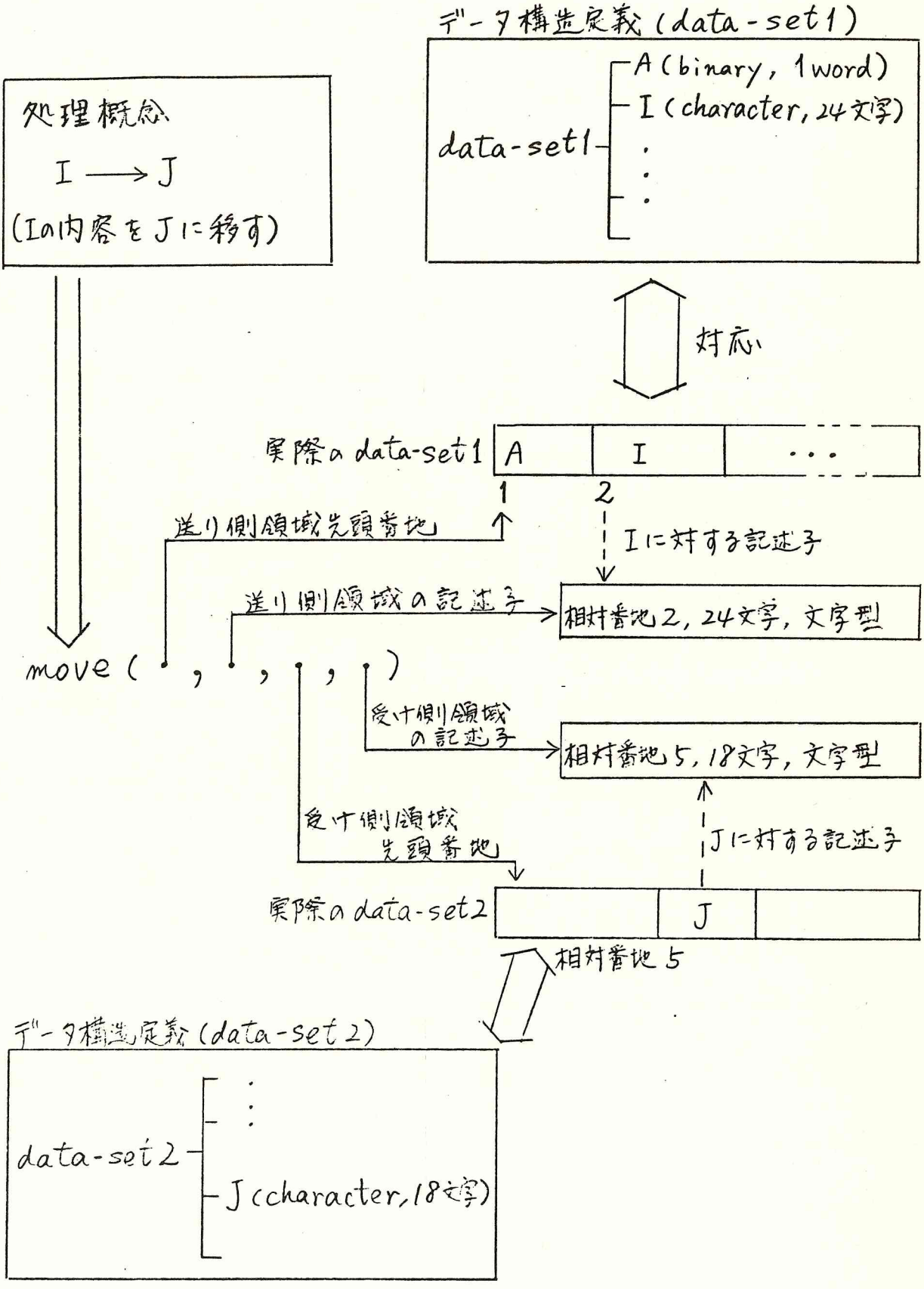


図4－4　転送における記述子の役割1

次に配列要素参照の場合について考察する。data-set 1中にある20語からなる配列A（型属性は＝進とする）の一語に，data-set 2 中のパック＋進型の変数Bの内容を転送する場合の例を図4－5に示す。

添字付変数（配列要素）は　実行時でなければ領域の先頭番地が決定できない。そこで　添字の値にあわせて記述子を更新する，subscript-update という実行時ルーチンを導入する。

subscript-update (..i, .i, value)

..i：更新された記述子が入れられる作業域
（これを作業用記述子とよぶ）
.i：配列全体を示す記述子
value：　添字値

subscript-update では　..i への記述子の作成を行うと同時に，添字値の妥当性の検査を行うことができる。

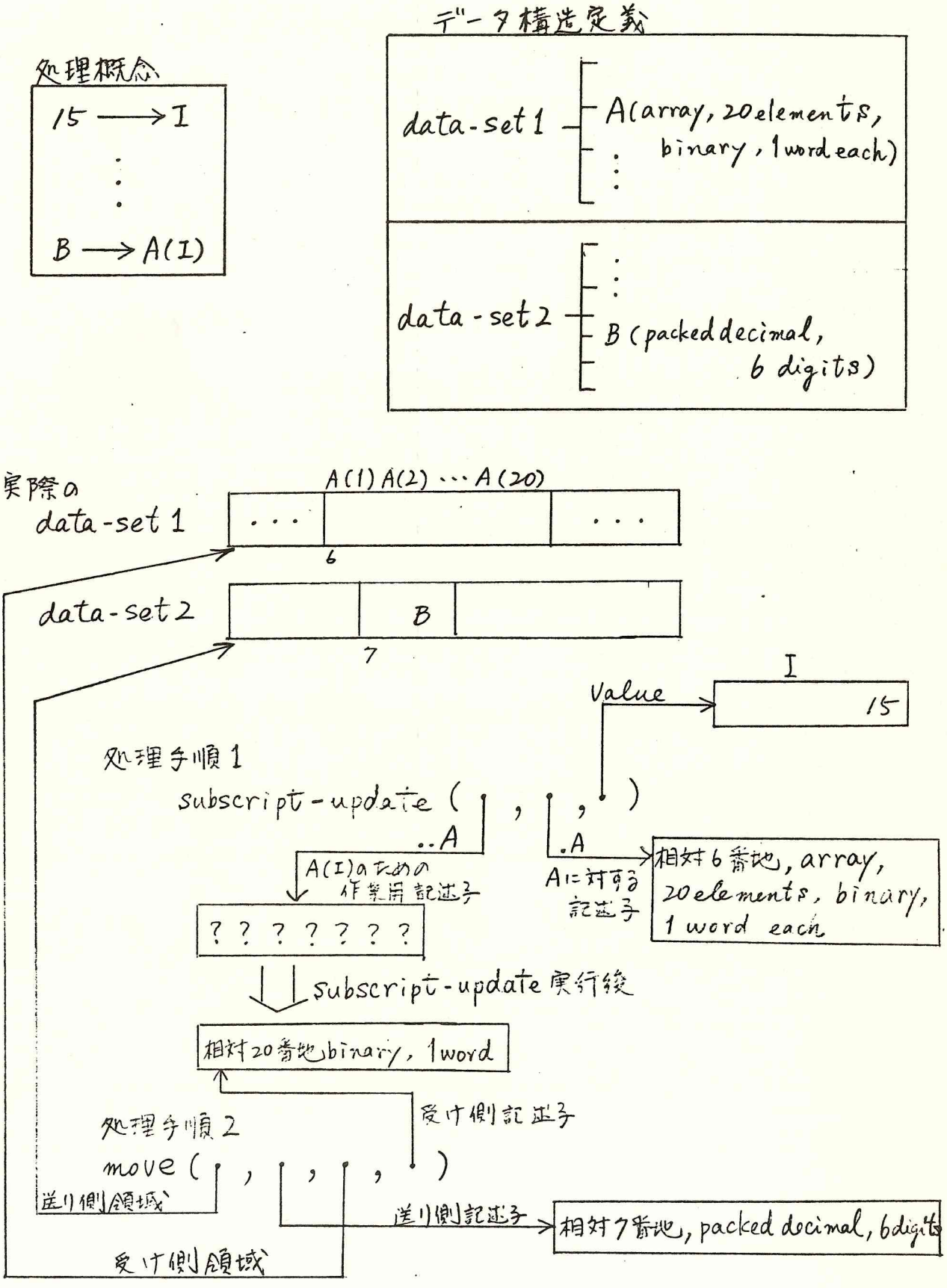


図4－5　転送における記述子の役割2（配列要素）

4.4 抽象手続の電子計算機上での実現

記述子処理を行うためには，①記憶属性の定義，②それによる記述子の生成及び処理手続中での引用機構，③実行時における支援機能が必要となる。

この①～③の機能に対する実現手法は4.3節に述べた。
第2章に述べたモデルを実現するために データのもつ記憶属性の分離に加えて，本節では論理属性の分離を行った抽象事象に対する手続構成法を中心に述べている。まず，分化された各々の情報を合成する手法の設計について述べる。この過程は抽象手続において対象となっている抽象事象に対する属性の確定とみることができる。

属性の確定の手順は図2－4にその概念が示されている。
図4－6にここまでに明らかになった事項を付加した 属性の確定手順との対応を示す。

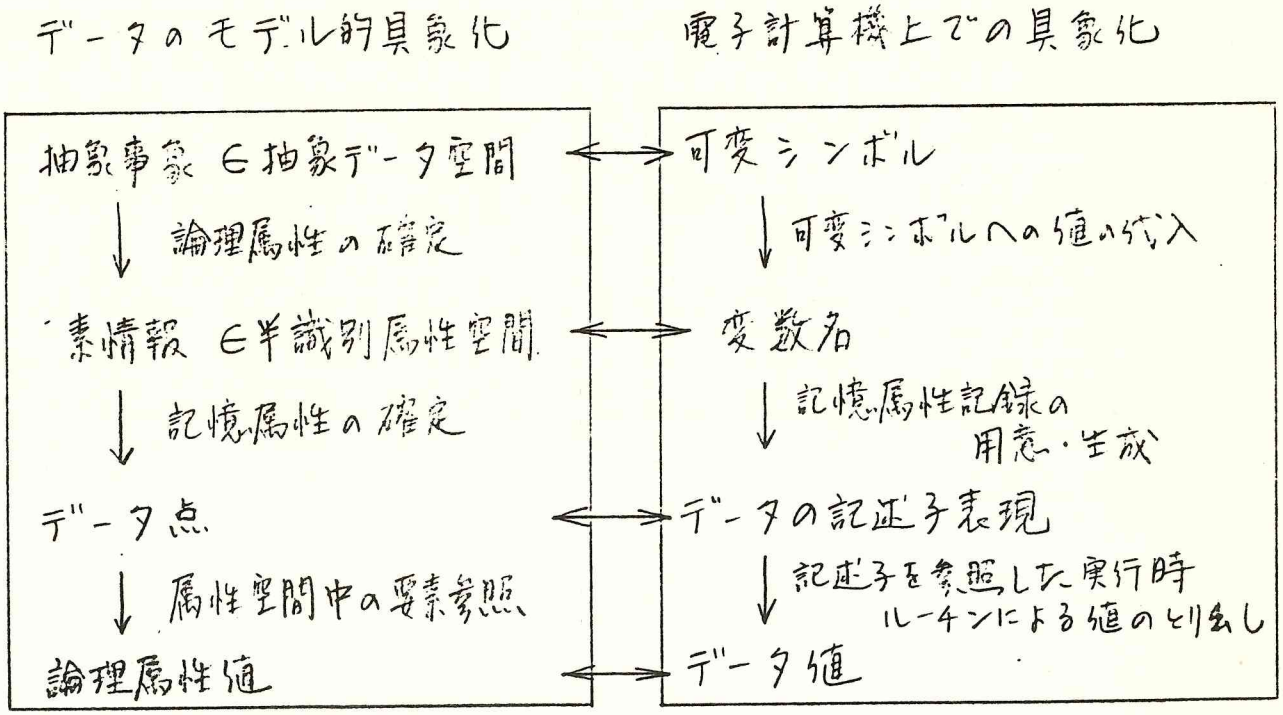


図4－6　データの段階的具象化

第2章に示したモデルにおいて定義された抽象事象は，「可変シンボル」として扱われる。素情報は一時的な変数の概念である。可変シンボルに対して名前，いいかえれば記号列を対応づけることにより，可変シンボルを用いて書かれた手続をある特定の対象に対するものに確定することができる。この確定は生成時において中間的に行われる。生成されたプログラム中にこの変数名が現われることはない。変数名は素情報の論理属性に相当する。

この論理属性に対応する記憶属性値を次に確定させる。確定した記憶属性値は記述子として保持される。実行に際しては この記述子を参照した実行時支援機能により，値の参照が行われる。このような関係をもつ抽象事象に対する処理概念を記述する言語体系を原形手続言語と呼ぶ。

原形手続言語には，

（1） 可変シンボル処理機能

（2） 生成制御機能

（3） 記憶属性処理機能

が含められる。

次に原形手続を基にするプログラム生成過程を表4－3に示す。生成過程は原形手続及び記憶属性値群の結合をもとに，原形手続の走査を中心として行うことができる。表4－3の示す手順は第5章において実現されたシステムにおける手順と一致する。原形手続及び記憶属性値群の選定は前もって登録されたものの中から，各々の名称を指定することにより行われる。この2つの準備作業ののち，可変シンボル値の原形手続外での代入が行われる。これにより一つの原形手続に対して 多数の異なったプログラムを生成することができる。次に選定された原形手続の内容を順に走査し，文の生成及び生成の制御を行う。このとき，可変シンボルを発見するたびに図4－6に示すデータの段階的具象化が生じる。合わせて対応する記述子を利用する文（実行時支援機能のよびだし）が生成される。

表4－3　プログラム生成過程

手順	内容
1.	原形手続の選定・結合
2.	記憶属性値群の選定・結合
3.	可変シンボル値の外部供給
4.	原形手続の走査とプログラム生成
4.1	可変シンボルに対しては図4－18に示した具象化
4.2	生成制御文に対しては対応する機能の実行
4.3	抽象命令に対しては確定した記述子を利用したステートメント生成
4.4	原形手続中の他の文に対してはそのまま生成
5.	生成されたプログラムの整形
6.	整形されたプログラムの翻訳
7.	記述子の利用を伴う実行

4.4.1　原形手続と論理属性の結合

結合は，可変シンボルディクショナリを通して行われる。

可変シンボルは原形手続を形成する重要な要素である。モデルとの対応で言うならば 抽象事象そのものである。可変シンボルディクショナリは 可変シンボルとその値（名前）との対応表であり，第6章の6.2.2に後述される変数束縛と同一の機構である。その概念を図4－7に示す。

可変シンボルディクショナリは原形手続を走査し，可変シンボルに出会うごとに参照される。既に登録されていなければ，そこで登録を行う。従って大きさが可変な表である。

可変シンボルの値は名前，いいかえれば記号列である。これはその可変シンボルが表わしている抽象事象に対する論理属性となることができる。この値は 生成制御文＊SETCや，第5章において後述される生成時のパラメータ代入文により確定する。単に記号列

のおきかえとして採用できるので，可変シンボルの値が特定のデータに対する論理属性となっているかどうかは　その可変シンボルの引用され方によって定まる。

あるデータの論理属性を構成するために可変シンボルが用いられた場合，その論理属性値すなわちデータ値は　対応する論理属性に対して与えられる記述子により間接的にさし示される。

可変シンボル(抽象事象)	その値(名前(論理属性))
x	A
y	B
⋮	⋮

生成制御文 &x　*SETC 'A'　あるいは
生成時のパラメータ代入文　x=A
などによる。

図4－7　可変シンボルディクショナリ

4.4.2　原形手続と記憶属性の結合

事象の論理属性が確定したとき，その記憶属性は選択された原形手続中で対象としているデータ群に対して必要な，前もって登録された記憶属性群（4.3節参照）から探し出される。記憶属性群には固有の名前が1つつけられており，論理属性とその記憶属性の集合から成り立っている。

この記憶の属性の探索は　次の3つの段階において行うことができる。（なお，*ATTRなどは4.4.3に示す生成制御文である。）

（1）*ATTR 生成制御文による直接的な引用。この場合，記憶

属性値は生成用カウンタにより出され，後続する生成制御文，抽象命令中で利用することができる。

(2) Instream-procedure 機能（＊STSUB～＊EDSUB中）における変数名引用。この場合，instream-procedure 処理系により，その記憶属性は自動的に記述子化され，Instream-procedure 中の文はその記述子を引用した文に翻訳される。

(3) 生成時手続機能（＊PROC）における変数名引用。この場合原形手続には存在していないデータ処理を生成時に与える点を除けば（2）に準ずる。

このときに生じる手順概念を 図4-8に示す。第5章に後述されるシステムを例にとり，これらの結合についてのより詳細な結合・確定過程を図4-9に示す。図4-9中の ⟺ は結合関係を示す。二重線（▭）で囲んだ生成パラメータがこれらの結合を起動する。

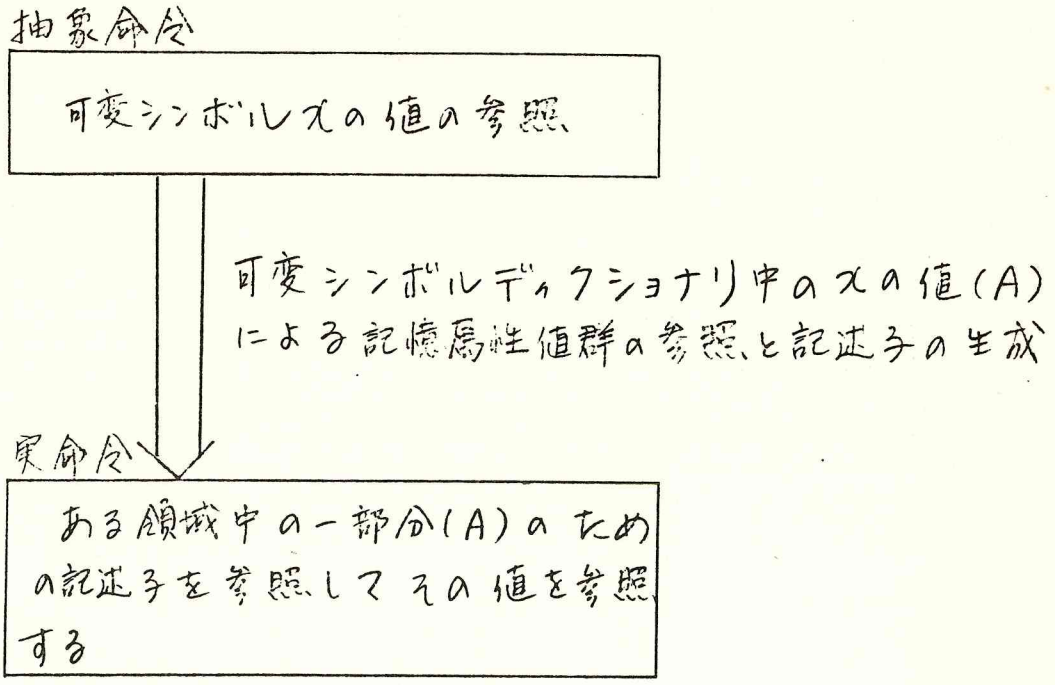


図4-8 可変シンボル・論理属性・論理属性値

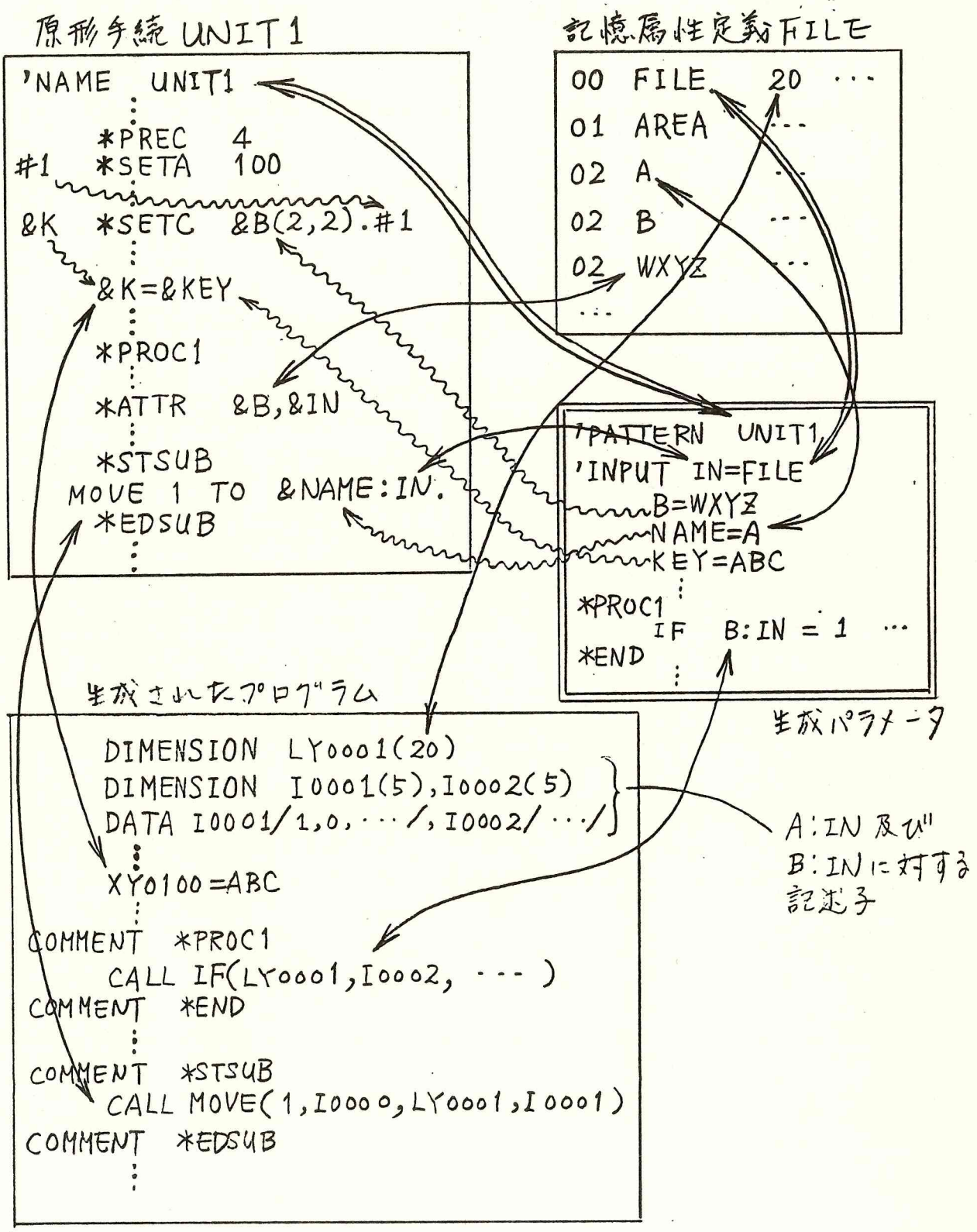


図4-9 原形手続に対する属性の確定とプログラムの生成

~~~~> により可変シンボルの値の確定を表わす。
<---> により注目すべき対応関係を表わす。

この図4－9には 図2－4に示した基本概念，図4－5，6，7，8に示された操作概念が集められている。図中の「生成されたプログラム」としては FORTRAN言語様のものが 記されているが，FORTRANである必要はない。そもそもこの結合において データの属性はすべて自動的に管理されるので，ターゲットとなった言語の性質には依存しない。ただ，5章で述べる実現システムにおいては，機種間の互換性の最も高い言語として FORTRAN を選んでいるにすぎない。

4.4.3 抽象手続記述のための原形手続記述言語

原形手続の記述法は マクロプロセッサにおける表記法に準じることにより，習得の容易さを高めることとする。すなわち，原形手続の定義は，可変シンボル，生成用カウンタ，生成制御ステートメント，原形文により構成される。

また，原形手続中には，生成時ラベルをふることができるので，分岐・バイパス・ループなどを行いながら生成をすることができる。文番号には&が前につけられる。これにより，手続中で定義された文番号を コンパイラによって生成された文番号との重複を避けられるようにすることができる。

可変シンボルは &記号が前につけられた変数であり，展開中はグローバルに利用される。可変シンボルの宣言文はなく，文脈中に最初に出現したときに宣言されたものとする。可変シンボルの利用により動的に結合される生成指示情報，生成時手続，データ構造定義間での情報の受け渡し及び原形手続内の作業域に使われる。

可変シンボルは基本的に記号列である。数字コードのみより成る記号列に対しては演算をさせることができるが，引用のためにコードの妥当性のチェック及び型変換が行われるので実行効率が悪い。このため数値を生成し，それを利用して展開を行う場合の便宜
~~~~

のために，生成用カウンタが9個用意されている。各々は#1～#9として引用できる。初期値はゼロにセットされている。生成用カウンタの評価に当たっては その表示桁数を制御できることが必要となる。後述する＊PREC文により桁数をコントロールする。
（＊PRECがないときは 5桁とされる。）

可変シンボル及び生成用カウンタは 必要に応じて相互に代入や連結をすることができる。

原形手続の走査により，特定の目的のためのプログラムが生成されることは前述した。一つの原形手続から多数の異なったプログラムを生成するための直接的な機能を行うのが 生成制御文である。必要となる生成制御機能を表4－4に示す。

表4－4 必要となる生成制御機能

生成制御機能名称	機能の内容
＊GO	無条件分岐機能
＊IF	条件分岐機能
＊SETA	可変シンボル等への算術的結合機能
＊SETC	可変シンボルへの記号的結合機能
＊PREC	表示桁数の制御機能
＊ERR	エラー表示
＊ATTR	可変シンボルの記憶属性の参照機能
＊PIF	パラメータの外部供給の有無のチェック
＊NOP	無操作指定
＊PROC0～＊PROC9	生成時手続埋込み指定
＊STSUB	In-stream procedure の定義
＊EDSUB	
＊END	展開終了

生成制御を行うためには，大別して次の5つの機能が必要である。

(1) 生成順序の変更

無条件分岐，あるいは生成時に強制的に結合された可変シンボルの値等により条件付き分岐を行う機能や，生成時手続の有無による分岐などである。

(2) 値の結合・参照

可変シンボルの値の代入・結合機能，あるいは原形手続中における直接的な記憶属性値の参照機能などである。

(3) 生成時手続埋込位置指定

原形手続中に生成時に副手続を付加する機能が存在できる。生成時手続の記述形式は 独自の言語形式をもち，次に示すIn-stream Procedure 機能における言語形式に等しい。

(4) In-stream Procedure

論理属性 ⟶ 記憶属性 ⟶ 記述子参照形 というデータの具象化を行うためのものである。論理属性名あるいは 可変シンボルを用いてそれら相互間の演算・転送その他の一般の高級言語にみられるデータ処理を記述する部分である。In-stream Procedure の言語形式は任意に設計することができる。その一例は 5.3節に詳述される。

(5) 展開・生成補助機能

終了指定，エラー表示指定などの補助機能である。

以下順に各個別機能の特長・形式について記す。

(1) 条件分岐機能

形式　　*IF (op1　operation　op2)　.TAG

op1とop2との間に関係演算operationを施し，その結果が真ならば　.TAG へ分岐する。
これにより，条件付生成が基本的に可能となる。

op1には可変シンボル又は　カウンタを指定することができる。
op2には可変シンボル，カウンタ，定数，文字列を指定することができる。
operationには

EQ
NE
GT
LT

を指定することができる。
operationは op1とop2の属性のチェックを行い，そののち指定された関係演算を行う。

(2) 無条件分岐機能

形式　　*GO　　.TAG

.TAG へ無条件分岐する

(3) パラメータの外部供給の有無による条件合岐機能

形式　*PIF { &XX / 又は / *PROCn } .TAG

その原形フレームの展開に先立って　その外側で可変シンボル又は*PROCnが定義されているかをチェックする。定義されていなければ .TAGへとぶ。

これにより　生成のキーとして用いられるパラメータの有無をチェックすることができる。Default処理や必須パラメータの強制などに利用できる。

(4) 可変シンボル等への算術的結合機能

形式　opØ　*SETA　op1 {*/} op2 {+−} op3

可変シンボル又はカウンタ(opØ)に算術的な値(数値)を代入する。必要ならば　演算をこのときに行うことができる。

opØは 可変シンボルまたはカウンタ

op1～op3は 可変シンボル，カウンタ，定数を記述できる。

可変シンボルが 用いられたときには　文字⟷数字変換とチェックが行われる。

(5) 可変シンボルへの記号的結合機能

形式　opØ　*SETC　op1

可変シンボル(opØ)への記号列(op1)の代入を行う文。

op1には次のものが書ける。
アポストロフィ(')で囲んだ文字列(例 'ABC')
可変シンボル名 (例 &OTHER)
可変シンボルの部分文字列
(例 &NAME(2,5):可変シンボル&NAMEの
2字目から5字)

(6) 可変シンボルと結合された論理属性に対して生成時に結合された記憶属性値群中における記憶属性参照機能

形式 *ATTR op1, op2

op1は可変シンボル又はリテラル
op2は可変シンボル
op2で示されるデータ構造グループ中でop1で示す要素が定義されていれば,その属性が,
#1～#5 に返される。
#1 ——— スタートワードポジション
#2 ——— スタートビットポジション
#3 ——— 属性
#4 ——— occurs 名なうその回数 else 0
#5 ——— 長さ(bit長)

(7) 生成用カウンタ表示桁数の制御機能

形式 *PREC n

生成用カウンタの原形文中の引用においての表示桁数の設定

(例)　#1　*SETA　123　とあるとき，

Case 1

*PREC 5 I=J#1	展開 →	I=J00123

Case 2

*PREC 2 I=J#1	展開 →	I=J23

(8) 生成時エラー指示機能

形式　*ERR　メッセージ

展開エラーを表示する。あわせてメッセージが印刷される。メッセージも展開scanがなされるので，メッセージ中にカウンタ引用や　可変シンボル引用があれば，各々のその時点での値がうめられる。

(9) 無操作指定機能

形式　*NOP

何もしない

(10) 展開生成終了機能

形式　*END

生成の終了を意味する実行文である。

(11) 生成時手続埋込み位置指定機能

形式　　*PROCn, 1=&m_1, 2=&m_2, …;
$0 \leq n \leq 9$
m_i は原形手続中の文番号

PROC文は 原形フレームの柔軟性を飛躍的に高めることができる特徴がある。

いわゆるマクロ定義では 前もって与えられた処理パターンの選択はできるが，新規の要求に対しては再定義をするしか対処できない。外部からの手続きの埋め込みを許すことができれば，原形フレームの可用性を向上させることができる。このために利用できる手続記述を

生成時手続

とよぶことにする。

生成時手続は 10通り記述することができ，各々PROC0 から PROC9と名付けられる。

PROC1～PROC9 は使用する原形手続の 展開において，PROC文に出会った時 その内容がコンパイルされ埋めこまれる。さらに 生成時手続の埋め込みに際しては，生成時手続から原形手続のあらかじめ設定された任意の個所に分岐できるようになっている (生成時手続記述言語中のRETURN文により)。

PROC文のオペランドには，そのための定義を記述する。
たとえば *PROC0, 1=&100, 2=&150;
とあったとき，生成側に *PROC0 があり，かつその中で，

RETURN　1

があれば，

原形手続中の文番号100への分岐

が生成される。

(12) In-stream Procedure 機能

形式　　*STSUB
　　〜
　　*EDSUB

原形手続中で，おそらくは定義されるであろうパラメータを用いたコーディングを生成するためには そのパラメータの属性，長さ，位置などを知って書かねばならない。

こうした場合のためなどに STSUB・EDSUBが用意されている。STSUBとEDSUBは 必ず対で用いられる。STSUBとEDSUBで囲まれた部分は In-Stream Procedure とよばれる。

In-Stream Procedure は まず可変シンボル，カウンタの評価がされ，それらがすんだソースがコンパイルされる。コンパイルされた結果はまた，可変シンボル等を含むので再展開される。

(例)

```
*STSUB
    LET   AREA:IN  = &KEY:IN + 1
*EDSUB
```

このときkeyが 何であっても その属性に応じたcodeが生成される。

図4-10に原形手続の例を示す。

```
'NAME=PROG1TO1
'PARM:&INPUT
'INPUT &10(&IN)
'OUTPUT &20(&OUT)
!ATEND &999(&IN)
'INITIAL
        *PREC 1
#1      *SETA 1
#2      *SETA 2
'MAIN
        *IF(&INPUT EQ 'CARD).CARD
&10     READ(#1,END=&999) &IN
        *GO .NEXT
.CARD   *NOP
        *IF(&INR LE 14) .OK
        *ERR INPUT RECORD SIZE IS MORE THAN 14
.OK     *NOP
&10     READ(5,&100,END=&999) &IN
&100    FORMAT(13A6,A2)
.NEXT   *NOP
        *PROC1,1=&10,2=&20,3=&999;
&20     WRITE(#2) &OUT
        GO TO &10
'FINAL
&999    CONTINUE
        *PROC2;
        *END
'END
```

図4-10
原形手続の例

'NAMEセクションは その原形手続に付される名前を示す。
'PARMセクションは 展開に使用される外部パラメータを宣言する。この例においては INPUTが宣言されている。INPUTに対する値の代入が生成時に指定されていなければエラーとなる。
'INPUT及び'OUTPUTは 入出力ステートメントの場所を示す。
'ATENDは 入力が終了したときの飛び先を示している。これらは覚え書きとして役立っている。また将来的には これらを用いて原形手続同志の結合などに結びつけることができる。

実際に展開される原形手続は 次の3つのセクションよりなる。

'INITIALは その原形手続の初期化部分の始まりを示す。この例では生成用カウンタの表示桁数を1桁にセットし，生成用カウンタの#1及び#2にそれぞれ1及び2をセットしている。
'MAINは プログラムのループを構成する主部分の始まりを示す。この例では 可変シンボルINPUTの値が'CARD'ならば .CARDから生成を行い，さもなければ次のREAD文の生成を行う。
そのとき #1，INなどは与えられた値でおきかえられる。
'FINALは そのプログラムの終了処理の始まりを示す。
'ENDは 原形手続定義の終了を示す。

この例の場合 これらの定義により，CardまたはTapeから（その選択は 可変シンボルINPUTにより決定される），データをINという名のエリアに読みこみ，PROC1で与えられる処理を行った後，OUTというエリアの内容を Tape上に書きこむプログラムの骨組を示される。

4.5 研究成果の要約

素情報属性の処理機構の設計を行った。まず半識別属性空間を電子計算機上に実現するために，包括的なデータの取り扱いを可能とする機構について述べた。これはセルと呼ぶ記憶属性を画一化し，その処理を不要にした自由領域管理概念の上に事象間の関連を保持するためのリンクリストと事象・素情報・連想子を統一的に扱う素情報管理を中心手法としている。この記憶構成法の上に，(素情報，属性，属性値)を直接に表現する連想子を用いた属性処理機構について，その利用概念・処理手順について明らかにした。この連想子機構により属性による値の参照が高速にできるようになった。第6章に示した実現例により，この機構の有用性が確認されている。

次に記憶属性を画一化できない応用のために記憶属性を分離し，それを記述子として独立管理する機構を述べた。これは理論的には半識別属性空間から記憶属性及び論理属性をとりさった，抽象データ空間の電子計算機上での実現である。さらに抽象データ空間を対象とする抽象手続を導入する。抽象手続を実現することにより，機種に依存しない，またデータの実際上の記憶形式に依存しないプログラムの管理が可能となる。これにより ソフトウェアの生産性・保守性が向上することは，実際に作成されたシステムに対して分析されており，これは第5章において明らかにされている。

最後に抽象手続のための原形手続記述言語の言語体系・原形手続中で使用されている抽象事象に対する論理属性の確定及び記憶属性の確定についてその仕組みを明らかにし，実現性を示した。

第5章　記述子を用いた事務情報処理支援システム

5.1 本章における研究の目的

本章では，記述子を用いた素情報の属性処理機構と抽象手続の概念を適用した電子計算機による事務情報処理に対する支援システムの構成法についてのべている。

5.2では，3章においてのべた機種独立性・データ独立性・応用独立性をもつ三層分化プログラム構成法の実現にあたって必要なシステムの形態について述べている。

5.3では，抽象手続とデータ構造の記述から1つのプログラムを生成する際に必要なジェネレータ入力にはどのようなものを与えるべきか，そしてその時の生成手順をまとめている。

5.4では 試行例と評価をまとめている。

5.2 支援システムの設計実施例

事務処理ソフトウェア生産を支援するシステムの設計実施例を示す。3.4節に示した設計原則に基づき，実現したソフトウェア作成管理システムをFAST1システムとよぶ。その概要は[IDA75]，[IDA78]に報告した。

5.2.1 機種独立性の実現

図5-1に示すソフトウェア作成管理システムを採用することにより，プログラム生産の総括的な管理が可能となる。

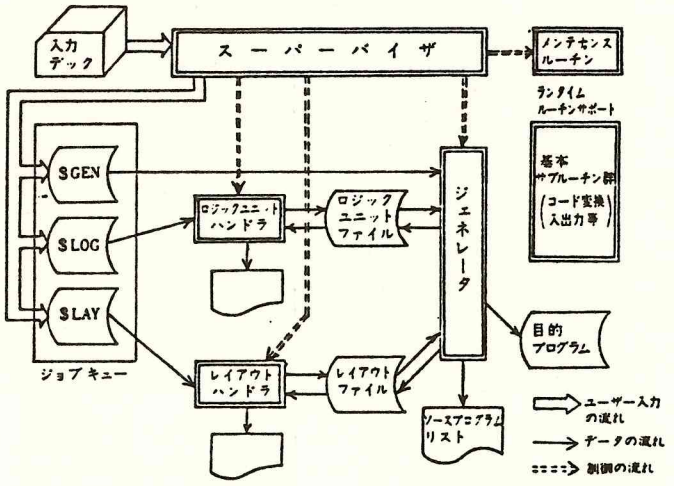


図5-1 システム構成図

□で囲まれた部分，すなわち，
スーパーバイザ：全体の作業の流れを管理
原形手続管理部（ロジックユニットハンドラ）
データ構造管理部（レイアウトハンドラ）
プログラム生成部（ジェネレータ）

保守管理部（メンテナンスルーチン）
基本サブルーチン群
により　システムプログラムは構成される。
⟹により，使用者の入力情報の流れを表わす
⟶により，データの流れを表わす
また
⋯⋯>により，制御の流れを表わす

使用者は任意の作業を組み合せ，スーパーバイザに対して与える。スーパーバイザはこれを解析し，指示された各部ごとに実行を行わせる。その核となるのはプログラム生成部である。生成指示情報及び細部処理用の言語形式によるコーディングを読みとり，これを変換して指示されたロジックユニットと結合し，目的プログラムを生成する。

データ構造・ファイル編成情報等の登録・修正・印刷などの管理操作はデータ構造管理部により行われる。また　ロジックユニットの登録修正などは　原形手続管理部により行われる。

これらのシステムプログラムは　すべて機種独立に作成することができ，FAST1 はFORTRANで記述された。初版はIBM1800で作成され，その後UNIVAC1108に移植された。

5.2.2　データ独立性の実現

ここではデータ独立性を実現するために必要な記述方式についてまとめている。

データ構造管理部に対する入力であるレイアウトには
①レイアウト名及びファイル編成情報（レコード長，ブロック長，編成手続）
②階層的なデータ構造の記述
③各レベルの変数名に対する「長さ，モード（BIN, CHA, DSP, SBNなど），繰り返し数（配列のため）」などの属性の記述

を含めることにより，プログラム中で扱われるファイルの構成について集中して管理することができる。

プログラム手続中に現われる変数名は，対応するレイアウトが探され，先頭番地に対する位置，長さ，モード，繰り返し数（単純変数ならば無し）により，オブジェクト中ではおきかえられる。これらの属性の記述子を解釈実行する実行時ルーチンを用意し，それらにより実際の情報の操作は行われる。

原形手続管理部に対する入力であるロジックユニットには，

①ロジックユニット名及び形式パラメータ
②入出力操作の行われている場所についての覚え書き（将来のドキュメントのため）
③初期化部についての原形手続
④主部についての原形手続
⑤後処理部の原形手続

を含める。

各々のプログラムを構成するプログラム生成部に対する入力は

①プログラム名称
②使用するロジックユニットの指定
③レイアウト及び外部ファイル名の指定
④プリント付加機能に対する記述
⑤記号パラメータの指定
⑥手続パラメータの記述
⑦手続パラメータ用サブルーチン（＊SUBPROC）の記述

に分ける。
④～⑦は必要に応じてつければよい。

データの記憶属性に関係した記述はレイアウトの部分にまとめられ，それらは生成時に引用される。この結果ロジックユニット及び生成時入力をデータ独立にすることができる。

5.2.3 応用独立性の実現

3.4で示した調査の結果からも明らかなように，各プログラムの機能の固定化は不可能である。したがって　どこまでを原形手続とみなすかは，その組織体及び用途によって異なることとなる。ただ　入出力ループは原形手続として共通化することが可能である。それ以上の共通化は　使用者で考えるべきことである。

本研究における考え方の特色は，このように原形手続さえも固定的な標準化をせずに記述・管理の手段を提供することに徹し，システムの柔軟性及び使用者の柔軟性を確保した点にある。
三層分化プログラム法を用いる組織体ではその要求を分析し，独自の原形手続を作成し，それをもとにプログラムの生産を行えばよい。原形手続の記述には容易に修得できる専用言語を用意した。　この言語は，本手法の中核となるものであり，4.4で述べた。

この専用言語を用いて作成・登録された原形手続を簡単に引用し，付加的な情報・処理手続との結合が自動的になされるように設計する。

利用者は，

1. 原形手続の指定
2. ファイル名及び作業域の呼出し
3. 1及び2に基づくプログラム生成に必要なパラメータ，部分コーディングの埋込み記述

により必要なプログラムを生成できる。

原形手続については　その各々についての機能概説，埋込み可能な部分コーディングの意味・位置，キーワードパラメータの種類などを記述した原形手続解説書を利用して生成を行う。

ファイル名，作業域については　登録されたデータレイアウトに対する資料を参照し，必要なものを決定する。

こうした機構をもつことにより，このシステム内で固定的に扱う事柄を，利用する業務・作業の内容・形式から独立にすることができ，汎用性のあるものにすることができる。

5.3 プログラム生成手法

5.3.1 ジェネレータ文法

ジェネレータ入力は大きく分けて次に示す3つのセクションより成る。

(1)パターンセクション

使用する原形フレーム(ロジックユニット)の指定，ユニット中で使用されるファイル名としての可変シンボルの値，外部ファイル名，ユーザがPROC-パラメータ中で特に使用を希望するワークエリアの定義および使用するレイアウト名．

(2)プリントフォーマットセクション(5.3.3参照)

このセクションはオプショナルである。ロジックユニットの定義とは独立して，ジェネレータの中だけで閉じたプリントのコントロールができるように作られている。したがって多くのプリント要求は特にロジックユニットを作らなくとも，このセクションを用いた定義とPROCパラメータ中のステートメントを働かせることによって行うことができる。事務処理プログラムにおいては様々な種類の帳票が要求されるが，このセクションを設けることによってロジックユニットの新規作成をおさえることができ，またロジックユニットの一般性を保つことができる。

(3)展開パラメータセクション

このセクションはキーワードパラメータの記述と，PROCパラメータの記述よりなる。プログラムの生成に対してユーザは，

選択的に記述できるものに対してはシンボリックパラメータのセットを行い，与えられた仕事に固有の処理はPROCパラメータで記述するように指導される。PROCパラメータのコーディングは，今までの例では長くて400ステップ，平均200ステップ程度であり，一般的には「4ページ コーディング」と言える。(5.3.2参照)

5.3.2 生成時手続及びIn-stream Procedureの入力言語

この言語の仕様の概要を拡張超言語記法で表わす。

```
<手続き>::=<行>;|<行>;<手続き>
<行>::=<ステートメント>|<ラベル>:<ステートメント>
<ステートメント>::=<IF文>|<GO TO 文>|<MOVE 文>|
        <LET文>|<PRINT 文>|<CALL 文>|<ASSIGN 文>|
        <UNSPEC 文>|<ENTRY 文>|<EXIT 文>|
        <RETURN 文>|<REPEAT 文>|<COPY 文>|
        <TRANSFER 文>|<EDIT 文>|<END 文>

<変数等>::=<変数>|<数>|'<記号列>'|<形象定数>
<形象定数>::=ZERO|SPACE|HVALUE|LVALUE
<変数>::=<データ構造中で定義された変数等>|
        <システム変数>|<ワーク変数>|
        <ワーク配列名>(<変数>)|<COL変数>

<データ構造中で定義された変数等>::=<英数字>|
        <英数字>:<英数字>|
        <英数字>(<変数>)[:<英数字>]|
        <英数字> OF <変数>
```

データ構造中で定義された変数等は 標準として :(コロン)

とその定義名称をつけることにより記述する。

(例)　　KEY:IN

INというファイル識別名称で関連づけられたデータ構造中のKEY

コロンがなければさがされる。

システム変数はI～Nの1字名の変数であり，すべてに共通している名前である。システム変数を介して簡単に情報の識別を行うことができる。

ワーク変数は，procedureの記述にあたって必要とされる作業域のための変数である。変数名自身に属性定義をもたせている。

COL変数はキャラクタイメージのレコードに対して，相対的な文字位置と長さの指定によるとりおしを可能にするための変数名である。

(1) IF文：

IF文では関係演算子による比較及びNUMERICチェック(IF … IS NUMERIC, …)が行える。また，ELSE節も扱える。

<IF>::= IF <変数>[NOT]<比較子><変数等>,
　　<statements>[ELSE <statements>]|
　　IF <変数> IS [NOT] NUMERIC,
　　<statements>[ELSE <statements>]
<比較子>::= >|<|=
<statements>::=<ステートメント>...

(2) GO TO文：無条件分岐を記述する

<GO TO 文>::= GO TO <ラベル>

(3) MOVE文：フィールド間の転送，定数のセット

<MOVE 文>::= MOVE <変数等> TO <変数>

(4) LET 文 ： 算術代入文

<LET 文> ::= LET <変数> = <変数> <演算子> <変数等>

<演算子> ::= + | − | * | /

(5) PRINT文 ： 後述するレポートライタ機能と関係する。

<PRINT文> ::= PRINT HD | PRINT DL | PRINT PT |
PRINT TL <数> [<英字>]

(6) CALL文 ； 外部手続き（サブルーチン）のよび出し

<CALL文> ::= CALL <名前> [(<引数並び>)]

(7) ASSIGN 文 ： システム変数間の算術代入文

(8) UNSPEC 文 ： 親言語（たとえば FORTRAN）の文をそのまま記述できる文。（セミコロンまでが すべて親言語に渡される）

(9) ENTRY文 ： 共通副手続きの入口の宣言を行う。必要ならば引数を記すことができる。

例： ENTRY SUB (I, J, k);

(10) EXIT文 ： 共通副手続き（SUBPROC）からの出口を表わす。

(11) RETURN文 ： 記された＊PROCから使用している原形フレームへの出口を表わす。原形フレーム中の＊PROC文には RETURN リストが定義できる。

例： ＊PROC 1 1 = &10, 2 = &20, 3 = &30

＊PROC1中での

RETURN 1は 原形フレーム中の文番号10への分岐を表わしている。

RETURN <整定数>

RETURN先の番号は原形フレーム中で意味を固定することができる。たとえば,

RETURN 1 ならば出力せずに次の入力を行う,

RETURN 2 ならば出力し,次の入力を行う。

などである。

(12) REPEAT文:

REPEAT <整数> TIMES [GIVING <システム変数>]

ENDまでで囲まれたブロックを指定された回数だけくり返す。<システム変数>の指定があれば 回数の現在値が与えられる。

(13) COPY文:

COPY <ファイル名1> TO <ファイル名2>

<ファイル名1>で示されたファイルの現在のレコードを<ファイル名2>で示されたファイルレコードにブロック転送する。

(14) TRANSFER 文:

TRANSFER <ファイル名1> TO <ファイル名2>

COBOLの MOVE CORRESPONDINGに相当する。ファイル間での同一の名前をもつ項目だけが転送される。転送時の型変換などは自動的に行われる。

(15) EDIT 文：

EDIT <変数1> TO <変数2> BY <PIC>

<変数1>を<変数2>のDSP型変数に編集転送する。
<PIC>には
'−', 'Z', 'X', ',', '.'
がかける。
'−' は負符号の指定
'Z' は数字の場合の不要ゼロ消去
'X' は無条件おきかえ
',' と '.' は 'X' 間又は 有効桁時の 'Z' 間で
有効となる ',' 及び '.' の挿入である。
規則は COBOLを簡略化したものである。

(16) END文： 記述終了を表わす

5.3.3 多段階集計及び帳票出力支援機能

ジェネレータ入力中に次のような形式のセクションをおき，多段階集計とそれに基づく帳票出力を容易にできるような支援機能を用意する。

(1) PRINT-FORMAT (図5-2)

5.3.2のPRINT文によって出力されるデータのフォーマットを定めるもので，フォームシートのプリントに適している。(汎用用紙へのプリントに適したものとして CHECK-LIST が準備されている。) 実際のプリントは，PRINT HD, PRINT DL, PRINT PT, PRINT TLn 等により出力する。

```
'PRINT-FORMAT   OUT= { DIRECT
                       GTP(ファイル名,キー[,パターンプリント
                                        のイメージ])
                ,MAXL=n
                ,MAXR=n  ;

 HD  }
 DL  }
 PT  } 行指定(プリント情報1)/ ・・・/(プリント情報n);
 TLn }
```

図5−2　PRINT-FORMATセクションの形式

① 'PRINT-FORMAT以外は1〜72カラム間でカラムフリー。

② OUTは アウトプット形態の指定

- DIRECT ・・・ 直接プリンターへアウトプットする。
- GTP ・・・ GTP用テープへアウトプットする。「ファイル名」はGTP用テープとしてアサインされるテープの名前(6文字以下)を記入し、「キー」はGTPのキー(1文字)を記入する。
  「パターンプリントのイメージ」は必要に応じてXX−nという形で記入する。XXはイメージとしてプリントする行のタイプ(HD, DL, PT, TLn)を記入し，nはその何行目を用いるかを示す数字を記入する。「パターンプリントのイメージ」を指定しない場合は，FAST1が＊印をMAXRで指定している文字数分，MAXLで指定している行数だけパターンプリントするよう考慮する。

OUTの指定をしなければDIRECT指定をとる。

② MAXLはプリントシート1ページにプリントできる最大行数を指定する。指定のない場合には66とみなす。

④ MAXRはプリントシート1行にプリントできる最大文字数を指定する。指定のない場合は132とみなす。

⑤ HD, DL, PT, TLnはプリント行の定義で, HDはヘッダー行, DLはディテイル行, PTはページトータル行, TLnはトータル行の定義で, nは0～9まで指定でき, 10個までトータルをプリントできる。"行指定"はプリントする行を指定するもので次のように行う。

HD = [絶対行]

DL = [始め行] − [終りの行], + [相対行]

PT = { [絶対行] / + [相対行] }

TLn = { [絶対行] / + [相対行] }

絶対行で指定すると, 頁内の指定した行にプリントされ, 相対行で指定すると直前にプリントした行から指定した行数だけフィードしてプリントされる。「プリント情報」は( )でかこり1行分の情報を記入する。多行プリントをする場合は1行の情報毎に/で区切り記入する。(プリント情報の間に/を連続して記入すると, その個数−1行分フィードされる)。このとき/は最高6個まで許される。

プリント情報には次の4つのタイプを置く。

○タイプ1　(プリントポジション　Name(Length))

記述したName(変数名)で指定したLength(キャラクター数)のエリアが確保される。エリアの属性はDISPLAYである。ファイル名としては、プリント情報の定義種類(HD, DL, PT, TLn)を使用する(GTPの場合はキーワードを付す。)

(例)

```
MOVE 'DATA' TO Name:DLk;
DL=8-60,+2(10 Name(4),18ABC:OUT(6));
```

○タイプ2　(プリントポジション　'…リテラル…')

このタイプはプリント情報中にリテラルを記述する場合使用する。

(例)

```
HD=5(10'*CHECK LIST*');
```

○タイプ3　(プリントポジション　変数(Length))
(プリントポジション　変数(Picture))

このタイプは変数の値をプリントする場合に用いる。FAST1は指定されたLength(キャラクター数)，又はPictureに従ってエディットしプリントエリアにMOVEする。

(例)

```
DL=8-60,+2(10 RISOKU:IN(7));
PT=63(10 TOTAL:IN(ZZZZZZZX));
```

○タイプ4　$\left(\left\{\begin{matrix}T\\TN\end{matrix}\right\}\right.$ プリントポジション 変数(Length)$\left.\right)$

$\left(\left\{\begin{matrix}T\\TN\end{matrix}\right\}\right.$ プリントポジション 変数(Picture)$\left.\right)$

このタイプはPT, TLnについてのみ有効である。FAST1は指定された変数について別個にエリアをとり，PRINT DL

命令が出されるたびに その変数が加算されて行くよう考慮している。(必ずしも その変数がPRINT DL命令によってプリントされるものである必要はない。)

プリントポジションの前のT又はTNの用法は，合計プリント後その合計をクリアーしたい場合Tを指定し，クリアーせずにさらに加算を続けたい場合TNを指定する。

⑥ 'PRINT-FORMAT使用の制限

a. OUT=DIRECT の指定は作成する1つのプログラム中に(1つの$GEN中に)，ただ1つしか指定できない。

b. 'PRINT-FORMAT で OUT=DIRECT を指定した場合は 'CHECK-LISTを併用できない。

⑦ FAST1が作り出す変数

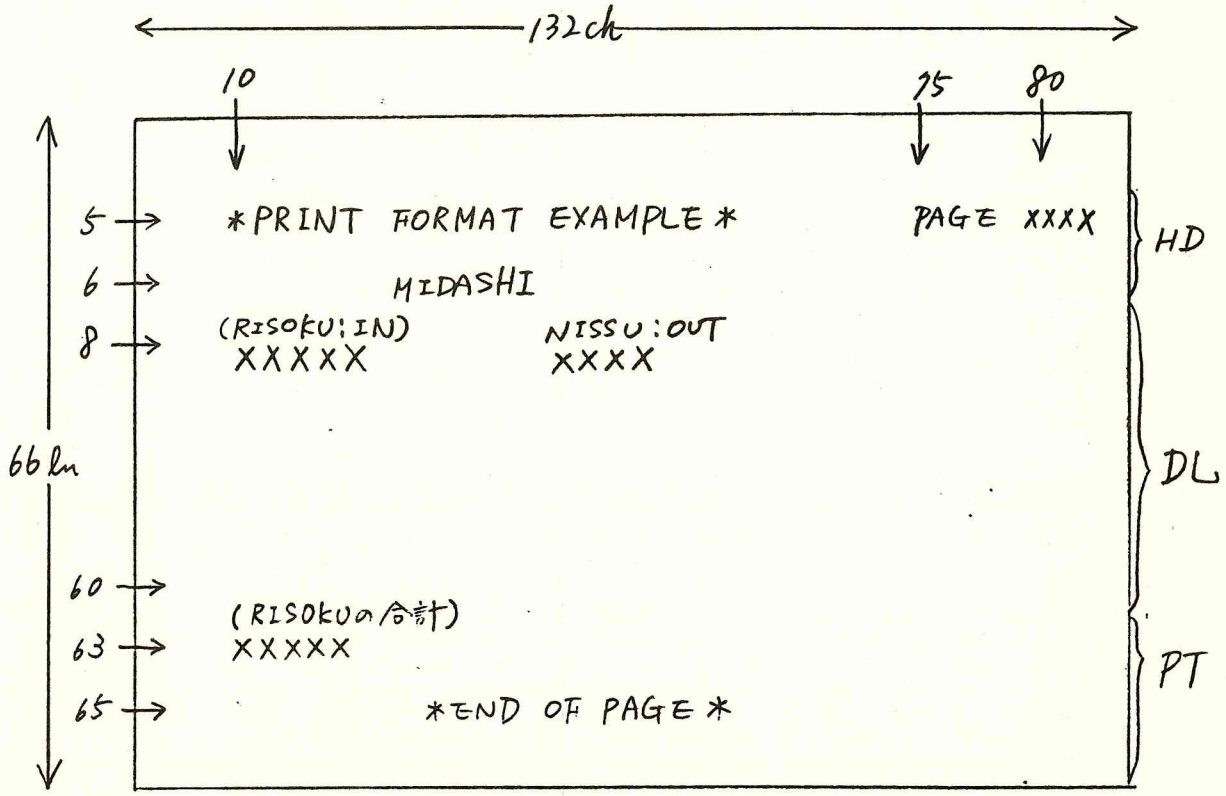
プリント関係でFAST1が作り出す変数として次のものがある。

a. PRPAGE:PRnn --- キーnnのプリントの頁数を保持

b. PRLINE:PRnn --- キーnnの現在プリントした行数を保持

c. PRNTSW:PRnn --- PRINT DLnn命令に対応しており，DLで指定した1頁の最大行を越えてPRINT DLnn命令をだした場合1がセットされている。(そのプリント命令は実行されない) その他の場合は常に0がセットされている。

d. PRGTID:PRnn --- GTPの場合のみ作り出される変数でGTPのPrint-IDが入っている。初期値として'GTP'がセットされているが，任意のものに変えてよい。ただし変更した場合はPRPAGE:PRnnを0にリセットしなければならない。



この他にある条件の場合にのみプリントするものとして
ERROR XXX(エラーコード) がある。

図5-3 PRINT-FORMATの例

```
'PRINT-FORMAT,OUT=GTP(GTPFIL,A,HD-1),MAXL=66,
 MAXR=132;
    HD=5(10'*PRINT FORMAT EXAMPLE*',75'PAGE',
        80 PRPAGE:PRA(4))/(17'MIDASHI');
    DL=8-60,+2(10 RISOKU:IN(5),25NISSU:OUT
        (ZZZX));
    PT=63(T10 RISOKU:IN(ZZZZX))//(20'* END OF PAGE');
    TL0=+1(10'*ERROR*',18 ERCODE:TBL(3));
```

図5-4 図5-3の形式に対するプリントフォーマット記述の例

nはGTPのキー（GTPでない場合省略）

これらの変数は'CHECK-LIST使用の場合も 同様に作り出される。

図5－3のフォーマットに従った'PRINT-FORMATの記述を例として図5－4に示す。

(2) 'CHECK-LIST（図5－5）

```
'CHECK-LIST;

HD
DL  } = 行指定(プリント情報1)/ ・・・ /(プリント情報n);
PT
```

図5－5 チェックリストセクションの形式

'CHECK-LISTのアウトプット形態はプリンターへの直接アウトプットである。HD, DL, PTは'PRINT-FORMATの場合に同じ。（TLnは使用できない。）行の制御，ヘッダープリント，ページトータルのプリントは自動的に行われる。'CHECK-LIST使用の制限としては，次の2つがある。

a. 'CHECK-LISTは作成する1つのプログラム中に（1つの$GEN中に）ただ1つしか使用できない。

b. OUT = DIRECTが指定されている'PRINT-FORMATとは併用できない。

このために必要な出力文をPROCパラメータ中に記述する。PRINT文は'PRINT-FORMATに対するプリント命令と，'CHECK-LISTに対するプリント命令に分かれる。

(1) 'PRINT-FORMATに対するプリント命令

'PRINT-FORMAT により定義されている情報をプリントする。

PRINT HDk; --- ヘッダー行のプリント
PRINT TLnnk; --- 合計行のプリント
PRINT PLk; --- ページトータル行のプリント
PRINT DLk; --- ディテイル行のプリント

kはGTPのキーを指定する。(GTPでない場合は省略)

(例) PRINT TL5B
GTPのキーとしてBを指定している'PRINT-FORMATのTL5で指定しているトータル行をプリントする。

(2) 'CHECK-LISTに対するプリント命令

'CHECK-LISTで定義されている情報をプリントする。

PRINT HD; --- ヘッダー行のプリント
PRINT DL; --- ディテイル行のプリント
PRINT PT; --- ページトータル行のプリント

用法は'PRINT-FORMATに対するものと同じであるが,'CHECK-LISTに対するプリント命令としてHD, DL, PTと個々にプリント命令を出さなくてもよいように,

PRINT CKLST

という命令が準備されている。PRINT CKLSTの実行により,ヘッダー行をプリントするタイミングであればヘッダー行をプリントし,ページトータル行のタイミングであればページトータル行をプ

リントし，改頁してヘッダー行をプリントした後ディテイル行をプリントすることなどを自動的に行う。

5.3.4 生成手順

次の手順に従って目的プログラムを生成する。

Phase 1. ロジックユニットファイルの参照

パターンセクションで指定されたロジックユニットを探し出しワークファイルに確保する。この過程は次のステップより成る。

1. ロジックユニットファイルのアサイン
2. ファイルディレクトリーのサーチ
3-1. Findしなかった場合，エラーとしてその後の処理を行わない。
3-2. Findした場合，ワークファイルへUNPACKしながら出力する。
4. ロジックユニットファイルのリリース

Phase 2. レイアウトファイルの参照

実行時のエリアの確保および後のPhaseで参照するファイルの項目の属性リストの作成，ファイル定義に関係する可変シンボルの値の付値などを行う。この過程は次のステップより成る。

1. レイアウトファイルのアサイン
2. ファイル定義カードを走査
3. 外部ファイルである場合，そのファイル名を付値
4. 与えられたレイアウトをサーチし，ワークファイル上へ展開する
5. レイアウトのサイズを定め，エリア確保用ステートメントの出力

6. ファイル付帯情報としての可変シンボルの値を付値する。
(表4-2)
7. レイアウトファイルのリリース

2～6のステップは，ファイル要求の数だけくり返される。

Phase 3. プリント出力の処理

'PRINT-FORMAT または 'CHECK-LIST カードがない場合はこのphaseは省略される。

各ライン定義の入力を解釈し，エリアの確保とリテラルのセットを行う。合わせて該当する機能をもつ内部サブルーチンを，すでにワークファイル上に展開されたロジックユニットの後につける。

ユニバック1108ではプリントの改行，改ページのコントロールはそのデフォルトとして標準のプリントシンビオントコントロールがあるが，FAST1のプリント機能を用いた場合，生成され付加された内部サブルーチンがすべてのコントロールを行う。

Phase 4. ロジックユニットの展開

この展開の過程は次の5つの機能より成り立っている。

1. シンボリックパラメータの評価
ジェネレータ入力のうちのシンボリックパラメータを評価し，可変シンボルの初期値として値を代入する。
2. 生成コントロールステートメントの処理
ロジックユニット中の生成コントロールステートメントを検索し，対応する処理を行う
3. PROC-パラメータのコンパイル
ロジックユニット中に*PROCステートメントが検索された場合，ジェネレータ入力中より対応するPROC-パラメータを検出し，コンパイルを行う。コンパイルされたコードは再び展開の対象となる。
4. 可変シンボルおよび生成用カウンタの評価

ロジックユニット中に現われた可変シンボル，生成用カウンタはそのときの値でおきかえられる。その際ステートメントの整形も同時に行う。

5. 擬似文番号の評価
ロジックユニット中の擬似文番号はグローバルな定数であるシステムインデックスにより修正される。これはPROC-パラメータのコンパイルの結果作りだされる文番号との重複をさけるためである。

Phase 5. プログラムの整形

作成されたプログラムを正しいFORTRANプログラムにするために配列宣言文，データ文等の整形，順序がえを行う。また，合わせて目的プログラム中の不要な部分の削除を行う。

5.4 生成実施例と評価

5.4.1 使用例

本節では，概念的な生成例とそのためのジェネレータ入力を示している。図5－6にこれを示す。ジェネレータ入力において，オーに必要なことはロジックユニット名の指定である。この例ではPROG1TO1を指定している。PROG1TO1は図4－10に示されている。PROG1TO1はINPUTをパラメータとし，CARDもしくは順編成ファイルをそれにより決定し，必要な作業を行ったのちに他の順編成ファイルに出力することをくり返すものである。

＊PROC1は，入力ファイルからの一レコード分のデータの読みこみが終了した時点に挿入される手続パラメータである。＊PROC1中のRETURN1は，出力せずに次のデータの入力への分岐を表わしている。RETURN2は，データの出力への分岐を表わしている。RETURN3は，強制EOFへの分岐を表わしている。この結合は，ロジックユニット中の

＊PROC1，1＝&10，2＝&20，3＝&999；

という文により示されている。（図4－10参照）

さてここでは，次に入出力に使用するファイル名及びレイアウト名を指定する。図5－6の，

IN＝（CARDF）

OUT＝OUTFIL（TAPEF）

はこのための記述である。

ロジックユニット中のIN及びOUTという入出力レコードを表わす抽象事象に対する属性の確定が，これにより行われる。INに対しては，レイアウトCARDFとして登録されているデータ名及び記憶属性の呼び出しが行われる。OUTに対しては，ファイル名

OUTFILの結合も指定されている。

INPUT－CARDの指定により入力はSYSINと結合される。PROCパラメータ中の

KEY：IN

はINと結合されたレイアウトCARDFの中にKEYがあるかを調べられ，KEYのもつ記憶属性に対応したコードが生成される。

図5－7は実際に利用され，既存プログラムとの機能比較テストの対象となったジェネレータ入力の一部である。図5－8に，生成に使用したロジックユニットSORT2の一部を示す。次節において本ジェネレータシステムにおける評価がまとめられている。

図5－9に比較テストの対象になった他のプログラムで使用したレイアウトを示す。このレイアウトは作業用変数群を定義している。

```
$GEN
'PATTERN PROG1TO1
        IN=(CARDF)
        OUT=OUTFIL(TAPEF)
'PARM
        INPUT=CARD
*PROC1
        IF KEY:IN NOT = 1, RETURN 1;
        MOVE ALL:IN TO ALL:OUT;
        RETURN 2;
        END PROC1;
'END
```

図5－6　ジェネレータ入力例（図4－10参照）

```
FAST-1 GENERATOR *
                        RUN ID      K-IDA          LIBRARY
                        DATE        03/19/75       23:46:27
*     1 **$GEN C.TEST
*     2 **.PATTERN SORT2
*     3 **        IN=LLKIN0(LOANK000)
*     4 **        OUT=LLKOU0(LOANK000)
*     5 **        OUT2=LLKOU2(LOANK200)
*     6 **        WORK = (WORKA)
        **03/19/75    23:47:16
*     7 **.CHECK-LIST
*     8 ** DL=8-60,+2 (14'******  FCONV03 ERR DATA    ',41PID:IN(3),47KAITENO:I
*     9 **            (3),53KOZANO:IN(7),62TORISNO:IN(7),71TORIANO:IN(6),79LOANCO:
*    10 **            IN(4),85YYN:IN(2),87'.',88MMN:IN(2),90'.',91DDN:IN(2),93'.',
*    11 **            95GANK:IN(11),108RISK:IN(11))
*    12 **.PARM SORT2:
        **03/19/75    23:47:19
*    13 **        KEY1=PID
*    14 **        KEY2=YAKUTEBI
*    15 **        KEY3=SIRALNO
*    16 **        KEY4=TORISNO
*    17 **        KEY5=TORIANO
*    18 **        CORE=20000
*    19 **        KEYS=5
*    20 **.PROC3
*    21 **      MOVE ZERO TO FFF:WORK ;
*    22 **       MOVE ZERO TO TTT:WORK ;
*    23 **       MOVE ZERO TO FRAG4:WORK ;
*    24 **       MOVE 25 TO NENGO:WORK ;
*    25 **       CALL LNTNRD (LNTIN:WORK ,M);
*    26 **      IF M = 0, GO TO TAGH;
*    27 **       CALL ERTRAN(2);
*    28 **    TAGH: CALL LDPACK (AREA:WORK ,5,N);
*    29 **       IF N = 0, GO TO TAGA;
*    30 **       CALL ERTRAN(2);
*    31 **    TAGA: RETURN 2 ;
*    32 **       END PROC3;
*    33 **.PROC4
*    34 **      MOVE PID:IN TO LOAN:WORK ;
*    35 **       MOVE LOAN:WORK  TO K;
*    36 **       IF IAREA OF AREA(K):WORK NOT = 1, GO TO TAGB;
*    37 **       ASSIGN I=I+1;
*    38 **      RETURN 1 ;
*    39 **    TAGB: RETURN 2 ;
*    40 **      END PROC4;
*    41 **.PROC1
*    42 **      MOVE PID:IN TO TEN:WORK ;
*    43 **       MOVE TEN:WORK  TO J;
*    44 **      MOVE ALNTIN OF LNTIN(J):WORK TO KUBUNA:WORK ;
*    45 **      IF KUBUNA:WORK = 3 , MOVE ZERO TO FRAG4:WORK , GO TO TAGD;
*    46 **      IF KUBUNA:WORK NOT = 2, GO TO TAGC;
*    47 **       MOVE ALNTIN OF LNTIN(J):WORK  TO KUBUNB:WORK ;
*    48 **      IF KUBUNB:WORK = 1 , MOVE ZERO TO FRAG4:WORK , GO TO TAGD ;
*    49 **    TAGC: IF FFF:WORK  > TEN:WORK ,MOVE ZERO TO FRAG4:WORK , GO TO TAGE;
*    50 **      IF TTT:WORK  < TEN:WORK ,    MOVE ZERO TO FRAG4:WORK , GO TO TAGD;
*    51 **    TAGE: IF KOZANO:IN = ZERO, GO TO TAGD;
```

図5-7 ジェネレータ入力の実例(一部分)

```
1  *        'NAME=SORT2
2  *        'PARM
3  *        'INPUT &100(&IN),&104(&CARD)
4  *        'OUTPUT &103(&OUT),&106(&OUT),&402(&OUT2)
5  *        'ATEND &900(&IN)
6  *        'INITIAL
7  *              DIMENSION ECT(60),FILE(2)          '
8  *        #1    *SETA &INR+6
9  *        #2    *SETA 2*#1
10 *            1   BUF(#2),MSG(2),ERRMSG(2)
11 *              DATA FILE/'&INF',' '/
12 *              DATA &INL/0,0,'&INF',0,0/
13 *              DATA MSG/'YYMMDD ?'/,ERRMSG/'* ERROR *'/
14 *              *PIF &CORE .NE
15 *              *GO .NEX1
16 *        .NE   *NOP
17 *        &CORE *SETA 15000
18 *        .NEX1 *NOP
19 *              DIMENSION SAREA(&CORE)
20 *        #4    *SETA 2*&KEYS+7
21 *              *PIF &FILES .NEX3
22 *        .NEX2 *NOP
23 *        #2    *SETA &FILES
24 *              *GO .NEO
25 *        .NEX3 *NOP
26 *        #2    *SETA 3
27 *        .NEO  *NOP
28 *        #4    *SETA #4+#2
29 *              DIMENSION SSPARM(#4)
30 *        #3    *SETA #2
31 *              *PREC 3
32 *        .NEX6 *NOP
33 *              DIMENSION SSF#3(2)
34 *              DATA SSF#3/'FAST1F#3'/
35 *        #3    *SETA #3-1
36 *              *IF(#3 GT 0).NEX6
37 *        C       START END SET
38 *              CALL FXSTBL(SSPARM,SSPARM)
39 *              SSPARM(2)= &INR
40 *              FLD(0,6,SSPARM(2)) = 1
41 *              CALL FXSTBL (SSPARM(#4),SSPARM)
42 *              FLD(0,6,SSPARM(#4))=57
43 *        C       FPOC,LPOC SET
44 *              CALL FXSTB1(SSPARM(3),SSPARM(4))
45 *              FLD(0,6,SSPARM(3))= 34
46 *              FLD(0,6,SSPARM(4))= 35
47 *        C   CORE ADDR. SET
```

図5－8　ロジックユニットの実例：SORT2
（図5－7のプログラムにおいて引用されている）

```
*FAST-1 LAY-OUT HANDLER*

LAY-OUT SOURCE STATEMENTS
00 WKPORT    12126 12126 FB
01 ALL       12126     0 CHA
02 OTEN          1     0 DSP
02 TTEN          1     0 CHA
02 SET           6     0 CHA
03 IPPAN         1     0 BIN
03 JUUTA         1     0 BIN
03 JIGYO         1     0 BIN
03 TOKU          1     0 BIN
03 SETU          1     0 BIN
03 TENB          1     0 BIN
02 TIPPAN        1     0 BIN
02 TJUUTA        1     0 BIN
02 TJIGYO        1     0 BIN
02 TTOKU         1     0 BIN
02 TSETU         1     0 BIN
02 TOT           1     0 BIN
02 IN           56     0 CHA
03 FILLER1      20     0 CHA
03 FILLER2       0    24 CHA
03 BLOCK         0    12 BIN
03 FILLER3      35     0 CHA
02 LBUF          1     0 CHA 1  4
03 FILLER4       0    18 CHA
03 LONKBN        0    12 BIN
03 FILLER5       0     6 CHA
02 LCODE         1     0 CHA
02 DBF       12000     0 CHA
02 STA           1     0 CHA
02 SRIA          1     0 CHA
02 CC            1     0 CHA
02 GT1           1     0 BIN 1  8
02 GT2           1     0 BIN 1  8
02 GT3           1     0 BIN 1  8
02 GT4           1     0 BIN 1  8
02 GT5           1     0 BIN 1  8
02 GT6           1     0 BIN 1  8
99 END OF WKPORT

WKPORT  DELETED

CATALOG COMPLETED
```

図5-9 レイアウトの例

5.4.2 評価

本節ではFAST1システムとして実現された属性処理機構及び三層分化プログラム構成法の有効性を，確認し評価する。

ジェネレータ入力は 本来記述せねばならない部分の大多数を原形手続の引用記述の1行に凝縮させることができるので，記述ステップ数を大幅に減少させることができる。また，構造を簡単化でき，プログラム作成所要日数も大幅に減少させることができる。実施テストを行ったある銀行では FAST1によるプログラム開発は，現行の普通の方式の3割の工数で済むと見積もっている。実施例のいくつかを 表5－1に示す。

一般にプログラムジェネレータによる機能分化は 実行効率において問題があると思われがちである。しかし，本論文における方式の実施例では次のような事態を観察している。すなわち，

原形手続中に緻密な設計に基づくOSとの間の高度なインタフェイスをおき，また記述子を利用する実行時ルーチンの最適化を行うことにより 実行速度の低下はかなりさけることができる。

その一例を表5－2に示す。システムの作動を確認する段階では，13分11秒を要したものを約9分の1の1分33秒に減らすことができる。これはジェネレータ入力（使用者による記述）を変更せずに達成した値である。1分33秒という値は普通の方式で開発された現行のプログラムの同一データに対する実行速度1分30秒にほぼ匹敵している。

これらのことから実行性能をほとんど落とさずに 所与の機能を得ることができることが明らかとなった。

実行性能については問題がないが，プログラムの保守性・プログラム開発効率の点で劣っていては実用性に欠けることになる。このシステムでは，三層分化プログラム構成法をとることにより，

$$(1-\alpha)nm$$

n: プログラム総本数
m: 平均ステップ数
α: 共有化率 (0.2～0.5)

に示す値だけ作業量を減らすことができる。（3.3節参照）この作業量の削減により，開発効率の上昇が認められることは表5-1のデータからも明らかである。

表5-1　プログラム作成所要日数及び記述ステップ名の比較［IDA75］

プログラム	使用言語	詳細設計	コーディング	デバッグ	記述ステップ数
A	FORTRAN	←――― 約1か月 ―――→			150
	FAST1	5時間	8時間	5日	86
B	FORTRAN	1日	6日	30日	319
	FAST1	4時間	11時間	20日	89
C	FORTRAN	3日	5日	20日	496
	FAST1	5時間	10時間	10日	105
D	FORTRAN	1日	7日	14日	444
	FAST1	5時間	6時間	10日	161

表5-2　プログラムA（表5-1）の実行性能の向上
（実行時ルーチン及び原形手続を最適化することにより手書きプログラムと同等になる。）

	FAST1				オリジナル
	VER1	VER2	VER3	VER5	FORTRANプログラム
実行時間	13分11秒	7分06秒	5分17秒	1分33秒	1分30秒
所要記憶域	54.7K	54.7K	53.9K	53.9K	48.8K

また，プログラムの保守性に関しては 記述ステップ数の大幅な減少（表5－1）による保守の容易化に加えて，属性処理機構の利用により実行時のデータの正当性がチェックされる効果が大きい。

オーにFAST1中で扱う変数は それぞれがレイアウトで定義されたデータの属性をもっているが，この属性はPROC－パラメータ上には直接現われず，実行時に実行時ルーチンが管理する。このためユーザーはコード変換などを考えることなくデータの転送，比較，演算のコーディングができる。これは逆に実行時の効率低下をひき起こすことにもなるが，コーディングの容易さとプログラムのドキュメント性という点から見れば，有利な点として考えられる。

オニに FAST1では データの転送，比較，演算の実行時ルーチン中に配列データの添字のチェック機能を含んでいる。このため ロジックユニットが完璧ならば 無関係なエリアをこわすことはない。もしそうした事態が発生したならば，実行時エラーとしてメッセージを出力する。この機能により デバッグが容易になった。

5.4.3 将来的な課題

FASTシステムの発展の方向として様々な形が考えられるが，今後の使用状況により流動的な面があるため FAST1 に組み入れなかった仕様，及び未解決の点を以下に記す。

1. 全体の流れを意識せずにプログラムを組めるように ジェネレータ入力の自己充足性を高める。

現在ジェネレータ入力は あくまで生成のパラメータという基本線に立っているため，使用するロジックユニットの流れを意識せずにコーディングすることはできない。

2. FAST1ジェネレータの機能の二分化

高効率と機種依存性は密接な関係があるため，FASTのPortabilityを保ちながら能力をあげるのには，最適化フェーズを分離する必要がある。さらに 仮想機械に対するマシンコードを生成するように ジェネレータを変更することの可否も検討の余地がある。

3. ロジックユニットの Inner MACRO

ロジックユニットのコーディングに対する支援は，現在ディクショナリ関係と 生成コントロールステートメントだけであるが，ロジックユニット定義のマクロライブラリを独立して維持できることが望ましい。これによりロジックユニットのコーディングの統一管理及び最適化フェーズの管理が容易となる。

4. レイアウトハンドラの機能

登録される項目の動的な属性（アクセス方法等）の扱いを導入すべきである。又，テーブルの扱いを考慮に入れるべきである。

5. PROCステートメントの拡張

大きな機能をもったステートメントを追加したい。これによりプログラムのstaticityが増し，コーディング・ステップ数も減少させることができる。しかし，これはステートメントの使用状況及び使用パターンを調査して決定する必要がある。さらにステートメントの数をふやすという方向は 習得すべき文法がふえ，ノン・プログラマー志向に反することになる。この点に注意が必要である。

6. プログラムの保管体制

現在のジェネレータ入力は RUN-STREAM からソースコーディングをとってくる形式をとっている。このため単純な形式をとると，ソースコーディングは カードで保管しなければならない。現在 実際には，他の方法でMT保管を行っているが FAST1 の機能としてソースライブラリの管理機能が要望される。

7. FASTシステムの作成・維持体制

FAST1の作成に当たって約3年を要したが，このためのマンパワーはごく一部を除き学生の手によるもので，通常のプログラマの作業に換算して約50人月程度であった。システマティック・プログラミングを目指した本システムもまた，その概念による産物であり，少ないマンパワーで作成したものである。このため本システムの保守（デバッグ，修正等）は しばしば特定の人物の作業能力がクリティカル・パスとなった。FAST2への展望としては この作成維持に対する手法の一層の合理化が必要である。

8. 教育体制

PROCーパラメータ用の言語には宣言文，入出力文がなく，またパーシャルコーディングであるため 独自の教育体制が必要である。

5.5 研究成果の要約

対象データの記憶属性及び抽象手続を各々独立して管理し，プログラム作成時にはそれらの引用と部分的なプログラムコードの埋込みを行うためのシステム概念を示し，その具体的な構成例としてFAST1システムについて述べた。

本章に述べられたプログラム構成法を三層分化プログラム構成法と呼ぶ。三層分化プログラム構成法は，プログラム機能の三分割とプログラム開発組織の三分割が直接的に対応できる点に特徴がある。

さらに，この三層分化プログラム構成法に基づくプログラム生成に必要な機構は普遍性があり，特定の電子計算機を仮定していない。また特定の応用システムをも仮定していない。

また，FAST1システムの試行結果より作成所要日数を半減させることが可能である点，所要メモリ数は1割程度増すが実行速度はほとんど損色がない点などが明らかとなっている。これは，使用する機械に合わせた最適化機構を組込むことにより可能となった。

これらのことから記述子を基本要素とする抽象データ空間概念と，それに基づくプログラム開発手法の有効性を確認できた。

第6章　連想子を用いた不均質データ空間管理システム

6.1 本章における研究の目的

本章では 第4章において示した半識別属性空間の電子計算機上での実現として，記憶属性の画一化による設計手法について，基本要素となる（素情報，属性，属性値）の対である連想子を中心に述べている。

設計手法としては 従来より存在した記号処理言語Lispを拡張し，連想子処理機能を付加することにより本論文に示したモデルを実現させている。このシステムをALPS/Iと呼んでいる。

ALPS/Iの記憶装置は 属性空間をすべて保持できるように充分大きくする必要がある。また属性空間に対する関数処理と使用者の便宜のためには，専用の単独計算機システムに構成することが考えられる。これらのことから，ALPS/Iを実現するコンパクトなハードウェアの設計手法とその実現についても論れられている。

このALPS/Iを用いたいくつかの例を通して，ALPS/Iの機能を確認する。

6.2 システム構成法とALPS/I

6.2.1 ALPS/I データ構造

本節ではALPS/Iで実現されているデータ構造についてまとめている。本節での内容は部分的に今までの記述と重複している。

① 一元構造

ALPS/Iで扱われる情報構造はただ一つであり，有向グラフを直接扱いうる形式をもつ。手続とデータの区別はなく，同一の形式をしている。このため手続を作り出し，それを動的に実行させることが容易である。

② 一元構造の実現

〔セル〕 ALPS/Iにおけるデータ点はセルと呼ばれる.

〔ドット対〕 ドット対には2セルの順序対がいれられる。これを

(a.b)

と記す。対の第1項をcar部，第2項をcdr部とよぶ。ドット対中の2つの値は対応するセルの識別子となっている。これを，ポインタとよぶ。

このときALPS/Iデータ空間は，

有向グラフ $G=(C,P,\Phi)$

C：セル集合
P：ポインタ集合
Φ：リンクラベル $|\Phi|=2$
(car及びcdrとよぶ)

として定義される。

ALPS/Iデータ空間においては一般に取り扱いの容易さから，

木(Tree)構造

を対象としているが，

孤立点，

巡回グラフ

も表現でき，利用されうる。

しかし完備性は必要で，

$|\phi| = 2$ または $|\phi| = 0$

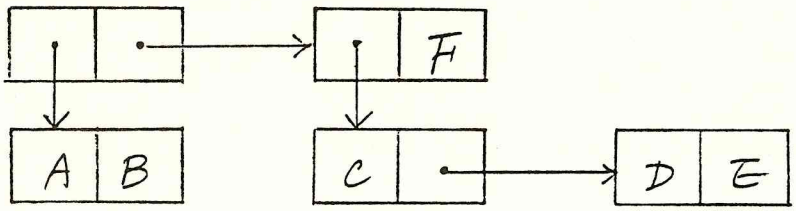
でなければならない。

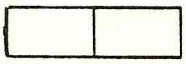
[S式] $g \in \vec{G}$ を S式とよぶ。

次のようなものはS式である。

$((A.B).((C.(D.E)).F))$

これを図的に次のように表わすこともできる。

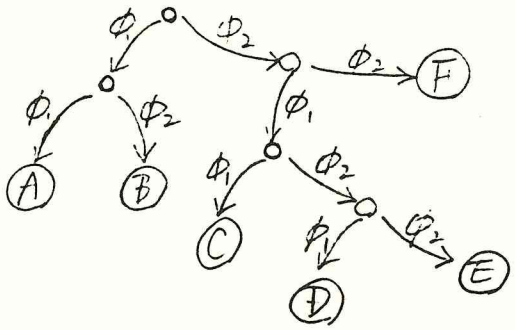


ここで  はドット対セルを，

ABCDでは表記号の一例を，

→ はポインタを表わしている。

また，グラフ表現すると次のようになる。



○を non-terminal node

ⓘを terminal node i

ϕ_1 を carとよばれるリンクラベル

ϕ_2 を cdrとよばれるリンクラベル

とする。

[S式のリスト表現]

$$(a_1 . (a_2 . (\cdots (a_n . a_{n+1}) \underbrace{)\cdots)}_{n個})$$

の形式をするS式を<u>リスト</u>と呼ぶ。(a_iは任意のS式)
これは次のような形をしている。

リストを表現するには途中のドットをとった次のような形式を用いることもできる。

$$(a_1 \ a_2 \ \cdots \ a_n . a_{n+1})$$

要素の間は必要ならばカンマで区切ってもよい。

a_{n+1}がNILという表記である場合
次のような表記を許す。

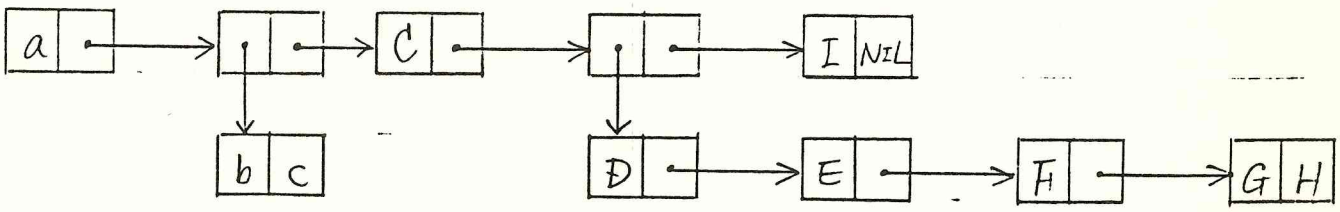
$$(a_1 \ a_2 \ \cdots \ a_n)$$

この場合NILという表記には次への連結がとめられるという特殊な意味が与えられていると考えることができる。

a_iには任意のS式をおいてよいので すべてのS式はリスト表現できる。
たとえば,

(a (b. c) C (D E F G. H) I)

は次のような形式をしたリストである。



〔対象データ及びデータ領域の区別〕

ALPS/I	Lisp 1.5〔MCC66〕
属性値集合(データ点の集まり)はブール関数atomによって2つに分割される.	対象データはすべてドット対により表わされる.
atom〔cell〕=Tとなるcellの集合を素集合とよぶ。 atom〔cell〕=Fとなるcellの集合をドット対領域とよぶ。	car部に-1をもつ部分グラフをアトムとよぶ。(素情報(終端記号)という概念は明確にはない) アトムはOBLSTとよばれる変数によりまとめられる。
素集合はさらにブール関数numberpによって2つに分割される。 numberp〔cell〕=Tとなるcellの集合を数領域とよぶ numberp〔cell〕=Fとなるcellの集合をH領域とよぶ (領域とは情報代数でいうAreaである)	アトムには数アトムと文字アトムがある。 ブール関数atomは.car部が-1であるかにより値を決定する。 numberpはそのcellがFullword Space中のcellであれば真を返す.
H領域に含まれるデータ点はその中の属性(atom indicator)により原化され, 素記号(atomic symbols)と 連想子(associators) にわけられる(連想子は本論文の中心事で記述される。)	連想子とよばれるものはない
数領域,H領域,ドット対領域はこの順に順序づけられている	数領域その他の副領域の間の順序づけは存在しない.

③ ALPS/I における基本演算

ALPS/Iで定義される基本演算は

car, cdr, null, eq, atom, cons, sum, and

である。これらにより他のすべての演算を合成することができる。以下の表現では

I は 任意の整数
A は 素情報
D は ドット対

とする。また ALPS/I - Lisp では 他の多くの Lisp と同様に Ω 及び ブール値 T を表わす記号として NIL をもつ。

[car] carは ドット対に対して定義される。

car[(a.b)] = a , a, b は任意のS式。

	Ω	I	A	D
car	Ω*	Ω**	Ω*	car[d]

* Lisp 1.5では -1
** Lisp 1.5では undefined

[cdr] cdrは carに準ずる

[null] null は反転ブール関数('NOT')の拡張である

	Ω	I	A	NIL	D
null	T	NIL	NIL	T	NIL

AはNIL以外の素情報とする

[eq] eqは その値が次表で与えられるブール線関数である

f_1 \ f_2	Ω	I_2	A_2	D_2
Ω	T	NIL	NIL	NIL*
I_1	NIL	$I_1=I_2$	NIL	NIL*
A_1	NIL	NIL	$A_1=A_2$	NIL*
D_1	NIL*	NIL*	NIL*	$D_1=D_2$*

D_1 eq D_2 が成立すること，Ω eq Ω が真であることに注意。

*Lisp1.5では undefined

[atom]

	Ω	I	A	D
atom	T	T	T	NIL

[cons] consはすべてのデータタイプに対して定義される

[sum]

f_1 \ f_2	Ω	I_2	A_2	D_2
Ω	Ω	Ω	Ω	Ω
I_1	Ω	I_1+I_2	Ω	Ω
A_1	Ω	Ω	Ω	Ω
D_1	Ω	Ω	Ω	Ω

[and]

f_1 \ f_2	Ω	I_2	A_2	D_2
Ω	Ω	Ω	Ω	Ω
I_1	Ω	T	A_2	T
A_1	Ω	A_1	A_1 and A_2	A_1
D_1	Ω	T	A_2	T

論理値以外のものに対しても論理積が定義できる点に注意。

6.2.2. ALPS/エデータ空間での関数の表現

処理手続はすべて関数として扱われる。関数は前置表現で表わす。$f(x_1 \cdots x_n)$をリストで表現するのに，

$$(f \quad x_1 \cdots x_n)$$

と表わす。
たとえば $f(x)+a/b$ は次のようになる。

$$(+ \quad (f\ x)(/\ a\ b))$$

基本的に値をもたない操作（入出力，代入，その他）も値を定義し，副作用をもつ関数として扱うことによりこの規則に例外はない。

たとえば $x+1$ を表わすのに 次のような入記法を用いることにより関数定義に必要な形式引数（この場合x）を指示する。

```
(LAMBDA  (X)(+ X 1))
```

入記法においては リストの第1要素はLAMBDAという表記号，第2要素は使用する形式引数のリスト，第3要素は関数形と約束されている。

こうして定義される関数形に対して 副作用をもつ関数defineを用いることにより 名前と結合することができる。（一時的に名を付ける記法としてラベル記法があるが，あまり用いられない。）また入記法を関数名としてそのまま用いてもよい。たとえば 次のようなものである。

```
((LAMBDA(X) (+ X  1)) 5)
```

手続構成のために用意されている機能は 次のような考え方によっている。

「単純化された制御構造と実用化のためには システム組込関数をふやし支援する。」

Lisp 1.5 は本質的には数個の組込関数が組込まれていれば動作するとされているが，それだけで実際的な手続きを記述するには手数がかかりすぎる。このため 多くの実際的なプログラムを流すことのできるLisp処理系では 100以上の関数を組込んでいる。

多くのLisp処理に共通して用意されている制御構造には次のようなものがある。

(1) cond式

これは一般にマッカーシーの条件式とよばれるものであり，

if P_1 then e_1
else if P_2 then e_2
⋮
else if P_n then e_n

という一般化された条件合成のためのものである。(もちろん e_i の中に他のcond式があってもよい)

この形式を次のようにリスト表現する。

(COND (P_1 e_1)
(P_2 e_2)
⋮
(P_n e_n))

(2) prog 関数

逐次的な実行をさせたい場合，このprogを利用することができる。またprog内のみで有効な局所変数(prog変数とよばれる)を宣言することができる。

progは 次のような形式をしている。

(PROG 局所変数のリスト 実行文並び).

実行文並びには 任意の関数をおくことができる。また，並びの中に素記号をおくこともできる。このとき その素記号はラベルとし

てみられる。prog関数中では このラベルへのGO TO 文 及び progを脱出するReturn 文などをかくことができる。

例を次に示す。

```
(PROG   (I V)
              (SETQ   I 0)
              (SETQ   V 0)
LOOP          (COND ((EQ   I N)
                     RETURN  V)))
              (SETQ   I (ADD1  I))
              (SETQ   V (SUM   V (TIMES I I)))
              (GO   LOOP))
```

これは $\sum_{i=1}^{n} i^2$ を行う部分である。

多くのLispではGO文の引数要素（上記例のLOOPの部分）には関数が書け，その関数の実行結果の値のラベルへ飛ぶことができるようになっている。このことにより多重分岐を許している。（Lisp 1.5では 許されていない。）

この関数処理系の特徴は次の2点にまとめられる。

(1) 一元構造の維持と手続の動的な確定

基本的には インタプリタで解釈実行されるように作られている。そのためLispコンパイラを作成しても，実行時のインタプリタはどうしても残ることになる。これは処理系記述の簡便さ・ポータビリティの維持という一般的な良い性質の追求に加えて，記号処理のもつ動的な側面を支援している特徴がある。

すなわち，手続とデータの区別がないので，たとえば，

```
(Cos   X)
```

というリストを見ても，これが単なるデータかあるいは，cos〔x〕という関数の実行を表わすのかを静的に決定することはできない。

そのリストが実行されるべきものとして「解釈しなければならない」という場所にあることをLisp処理系が動的に認識した場合にのみ cos[x] の評価がなされる。

したがって 処理手続をデータとして生成し，それを実行させることや，自分自身または他の手続の一部を他のものに置き換えてしまうことも可能であり，このことも他の汎用言語にはない特徴となっている。

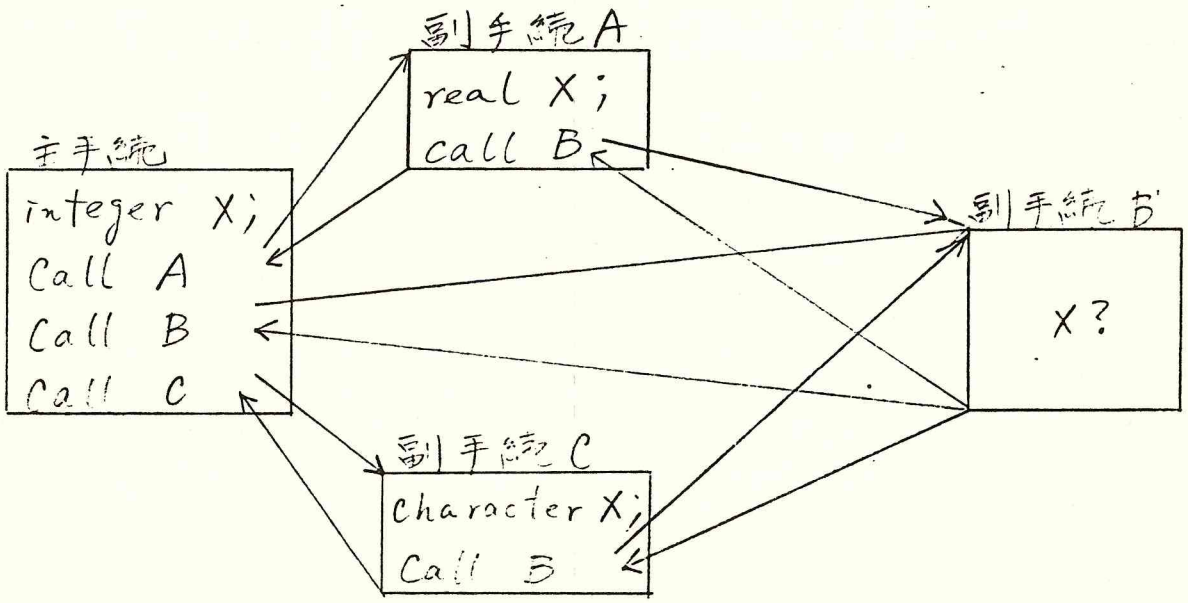


図 6-1 入れ子になった実行順序をもつ手続き

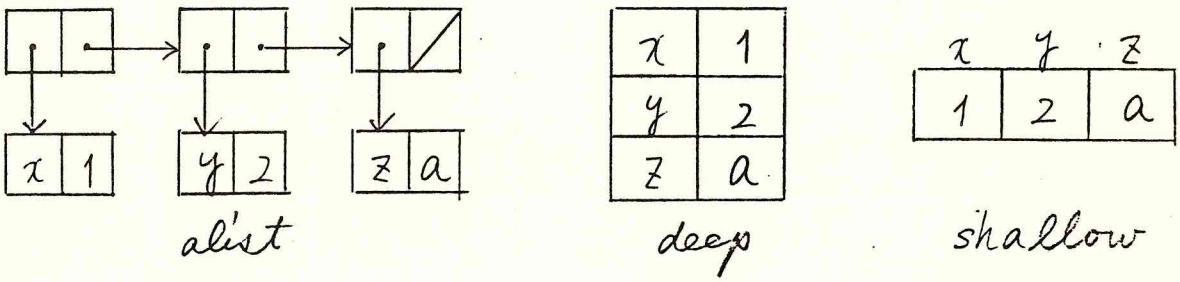
表 6-1 束縛形式

		属性の決め方	副手続B中の変数Xの属性
静的束縛	ブロック構造言語 (Algol, PL/I など)	静的な親の属性となる	integer (副手続A, B, Cが並列に主手続中に組込まれているとき)
	非ブロック構造言語 (Fortran など)	その手続の中だけで決定される	エラー
動的束縛	Lisp など	動的な呼出し順を逆のぼって探す	主手続から呼ばれた場合→integer 副手続Aから→real 副手続Cから→char.

(2) 動的結合による束縛

Lispは表6－1に示した動的束縛を行う言語である。実行の履歴に従って変数の値を探して行く。Lispのこの束縛方法を実現するのに，alist bind，deep bind及びshallow bindとよばれる3つの束縛方法がある．alist bindはリストを利用した束縛情報の記憶，deep bindはスタックを利用したもの，shallow bindは特定のセルに最新値を格納し，他は別のスタックに格納するものである。これを図6－2に示す。

図6－2· x=1，y=2，z=aの束縛状態



alistは値の探索に時間がかかるなどの欠点があるが，他のものと違い一般のリストと混在して管理され，柔軟性が高い。このためまったく異なった環境の導入などが容易で，静的束縛を要求する応用などもうまく処理することができる。

deep binding及びshallow bindingはこの点ではalistより劣る。shallow bindingは変数ごとに決められた場所に値をおき，旧値はスタックへ退避するというもので，値の参照の高速性がある。しかし頻繁の切り換えの多いプログラムの場合，スタックとの間の転送コストが増加し，deepの方が良い場合もあるという説をとなえる人もある。

この変数束縛に対する普遍的な認識はLispの研究者の間では薄いようである。Funarg問題とかfexprの環境保存などの問題は静的環境のLispへのとりこみということに帰着する。その構造は[BUL78]，[ORG73]など，あるいはブロック構造言語に対する文献などに含まれている。

6.2.3 ALPS/エ型連想構成の応用の方向

Lispの持つ特徴は主に

二進有向グラフに統一した一元構造

に起因しているといえる。

このため情報の構造と情報間の関係を処理する応用に向いていると言え，実記号処理とよばれる分野での利用が大きい。特に未知のアルゴリズムの探求などが必要となる場合には，プログラマは何を(what)書きたいかに集中し，どうやって(How)表現するかという側面の少ない言語が望ましい。Lispはその点でも情報間の関係の処理にはひいでている。

これらの処理は一口に人工知能分野の中に入れることができるが，その中でもっとも実際的な(いいかえれば実用化が早い)ものの1つに数式処理があげられる。

数式処理システムは「数値演算によらず記号のまま演算してほしい」という基本的な要求に根ざすものである。数式処理システムとLispとは直接的には無関係であり，以前にはPL/Iによるものや FORTRAN, Assembler を記述言語としたものが実験・研究用に作られてきたが，現在多くの数式処理アルゴリズムはLispで記述されるようになってきている。

またそれらの多くは A.C. Hearnの作成したReduce言語に組み込まれるような傾向がある。これにはHearn教授の精力的な普及活動によるところが大きいようである。この辺の事情については，「計算機による数式処理の現状」(後藤英一，IPSJ)などに詳しい。このReduceシステムは後述するALPS/エにも組込んでいる。小林氏の助力による初版によりその作動を確認し，遠峰氏の助力による第二版により現在に至っている。その例を図6-3に示す。また言語構造の解析や「積み木の世界」とよばれる環境認識問題に対する応用プログラムでも，Lispはかなりの実績をもつ。

① $H_n(x) = (-1)^n e^{x^2} \frac{d^n}{dx^n} e^{-x^2}$ による実行

```
EVALQUOTE ENTERED, ARGUMENTS...
BEGIN( )( )
REDUCE 2
FOR N:=1:5 DO BEGIN
      H(N):=((-1)**N)*(E**(X**2))*(DF((E**((-1)*X**2)),X,N));
      WRITE "H(",N,") = ",H(N) END;
H(1) = 2*X

                    2
H(2) = 4*X  - 2

                    3
H(3) = 8*X  - 12*X

                    4       2
H(4) = 16*X  - 48*X  + 12

                    5       3
H(5) = 32*X  - 160*X  + 120*X
```

② $H_n(x) = 2xH_{n-1}(x) - 2(n-1)H_{n-2}(x)$
$H_0 = 1,\ H_1 = 2x$ による実行

```
REDUCE 2 (MAR-21-79) ...
OFF ALLFAC;
ARRAY H(20);
H(0):=1$
H(1):=2*X$
FOR N:=1:10 DO BEGIN H(N+1):=2*X*H(N)-2*N*H(N-1);
  WRITE "H(",N,") = ",H(N) END;
H(1) = 2*X

                    2
H(2) = 4*X  - 2

                    3
H(3) = 8*X  - 12*X

                    4       2
H(4) = 16*X  - 48*X  + 12

                    5       3
H(5) = 32*X  - 160*X  + 120*X

                    6       4       2
H(6) = 64*X  - 480*X  + 720*X  - 120

                    7       5       3
H(7) = 128*X  - 1344*X  + 3360*X  - 1680*X

                    8       6       4       2
H(8) = 256*X  - 3584*X  + 13440*X  - 13440*X  + 1680

                    9       7       5       3
H(9) = 512*X  - 9216*X  + 48384*X  - 80640*X  + 30240*X

                    10      8       6       4       2
H(10) = 1024*X  - 23040*X  + 161280*X  - 403200*X  + 302400*X  - 30240
```

図 6-3 ALPS/エによる Reduce の実行例
——エルミート多項式の展開——

6.3 本管理システムの実現された記号処理専用コンピュータの設計と試作

ALPS/I (Aoyama List Processing System / I) は前述のような考え方で設計・試作されたLispマシンである。以下ではこのマシンの設計・試作について述べる。なお，本節の内容は[IDA76], [IDA77A], [IDA79A], [IDA79C]に発表されている。

6.3.1 背景

実際的なLispシステムの多くは大型機上で作成されている。([QUA70], [TEI72], [ETL76]) さらに国内におけるLisp処理系の絶対数が少なく，手軽にLispプログラムを作成実行できない現状がある。また，青山学院情報科学研究センターへのLispシステムの移植は所要メモリ量及び所要時間共に一般のJobの実行を著しくそこなう程の値である。

数式処理・自然言語処理などのためのLispプログラムの大きな蓄積や現在進みつつあるソフトウェア工学用へのLispの利用などを行うことを考えると，我々は手近な処理系を目指して価格/性能比志向のLisp専用機を開発することが必要となった。

Lispの特性及び使用上の便宜から専用機の満たすべき条件を次のように考える。

①電源投入のみで作動し，かつ安価である。

→最近急激に低価格化し，また拡張性・市場性がある8ビットマイクロプロセッサをCPUに採用し，処理系は半固定記憶、(PROM)に記憶させる。

②数10Kセル程度以上のLispデータの保持を許す大容量メモリの具備
——→その安価傾向から大容量のICメモリを採用する。参照の効率から語構成（1語 32bit以上，特によないし9の倍数bit長のものは入手が容易）を取り，物理的なアドレス空間はCPU能力とのかねあいから 16bitで構成する。（この点については後述する。）

③頻繁なLispデータ参照の高速化機構
——→接続されるバルフメモリに対する専用インタフェイスを作成し，低速なマイクロプロセッサの通常の入出力命令によらず，転送時のソフトステップ数を最小におさえ高速化をはかる。

④可用性向上のためのシステム装備と各種関数の組込み
——→約100の関数を組込み，会話型処理形態を提供する。

1.5のような定義に基づくLispマシンの設計を考えるにあたっては，単にハードウェアの高速化・マイクロプログラム化という微視的な見方は妥当ではない。その根拠は次のようなものである。

Lisp処理系自身の処理アルゴリズム，組込むべき関数の種類・仕様は，現存のシステムの間でも統一されていない。このためLisp処理系の高速化に重点をおく実験的なシステムは別として，何をファーム化するか，あるいは何を固定機能化するかについては 実際的な処理系の早期製作を目標とする我々にとって一義的な選択はなかった。

事実，マイクロプロセッサを中心とするハードウェアの進歩は著しいこと，今までに見られなかったLispの応用例が報告され始めており，Lispに要求される機能に影響を及ぼす可能性があること，一般に言語の普及においては 処理系の提供がまず第一であることなどがいえる。

高速性の追及は　それらが確定した後に，という方針をとった。

6.3.2　8ビットマイクロプロセッサとバルクメモリの採用

ALPS/エシステムは8bitマイクロプロセッサ，インテル8080，PROM 14kbyte，RAM 8kbyte，インクジェットプリンタ（PTR/PTP付），フロッピーディスク装置，バルクメモリ64kw（1w＝36ビット）より構成される。（図6－4，6－5）　8080及びバルクECメモリの採用に特徴があり，この点を中心に記述を行う。まず拡張性・市場性を考え，CPUに8ビットマイクロプロセッサ（8080）の採用が設計当初に決定された。その理由は次の通りである。

①設計当時（昭和50年3月）入手が容易であり，また段階的なシステム拡充が可能と思われた。（現在ではCPU，RAM，PROM，入出力などのチップ，モジュールの高速・高能力化が進み，これらに置き換えれば少なくとも倍以上の速度が見こまれる。）
②ハードウェアスタックポインタがあり，再帰呼出し等に便利でかつ比較的速い。（再帰呼出しの可能なCall及びret命令は　現在12.5μsec及び7.0μsecで0.5μsecのメモリを用いれば8.5μsec及び5μsecで実行できる。）
③1バイト語長の命令が多く，ステップ数に比してコンパクトに作成できる。（本体は7800ステップで11kbyteを占め，1命令当たり約1.4バイトである。またインタプリタのみでは2kbyte弱の大きさしか要しない。）
④算術演算に比して単純なデータ転送の多いLisp処理系の場合，②及び③により他の言語の場合と比べてマイクロプロセッサの低速性による速度低下は　それほど増えないと思われる。

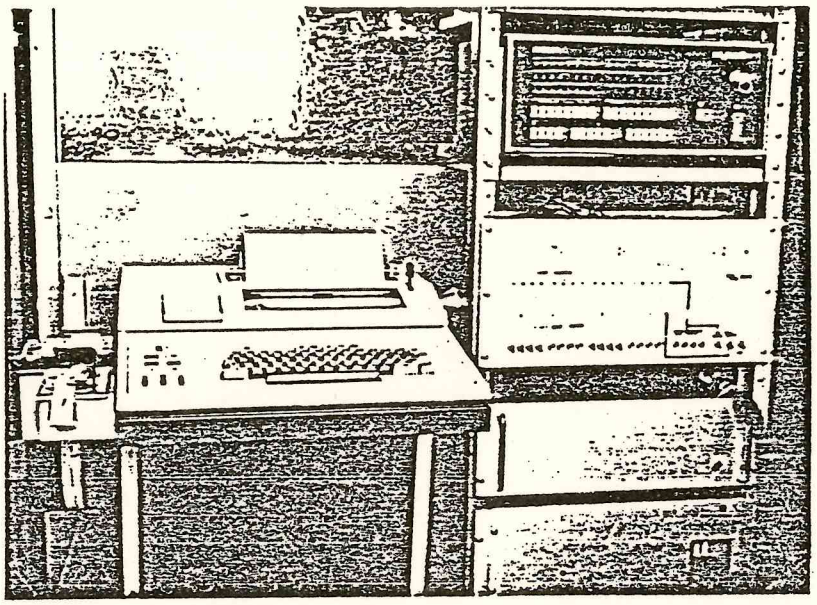


図 6－4　ALPS/I の外観

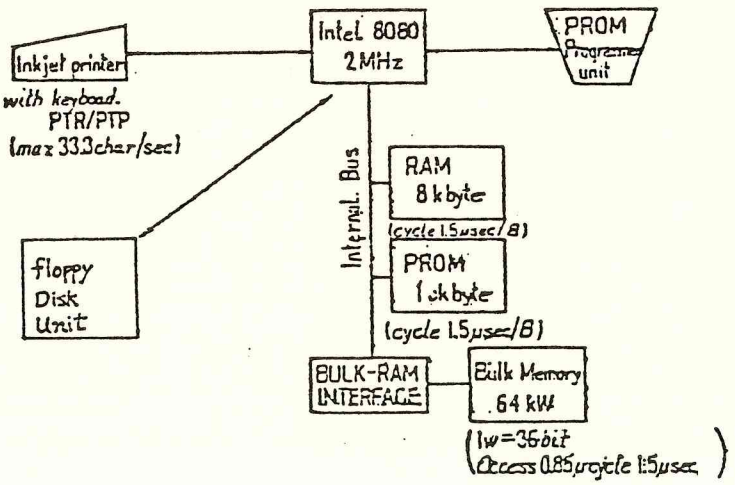


図 6－5　ALPS/I 構成図

Lisp処理系は明示的にリストを扱うため，メモリアドレス空間の設定方法は他の言語以上に処理能力及び速度に影響する。たとえば16ビットミニコンでのLisp処理系では，64K以上の記憶能力を与えるため二次記憶へのリスト退避手続を用意し，その管理を使用者にまかせることはよく行われてきた。また最近では，ソフトウェアによるページング機構を介して仮想ストレージを外部記憶上にもち，仮想アドレスをポインタ情報とする例も報告されている。前者の二次記憶の利用法は簡便である反面，Lisp使用者に負担がかかる。後者の仮想化は対象となるリンクリストの非局所性の問題，ソフトウェアによる場合はそのアドレス変換の低速性の是否が吟味されねばならない。

記憶能力の拡大には上記の他に，メモリバンクの切換えがミニコン等でよく見られる。バンクの切換えは仮想化に比べて一般に高速性があるので，メモリ参照の多いLisp処理系においては好ましい。

ALPS/Eではそのため，システム用の内部メモリ（PROMとRAM）とLispデータ格納用メモリ（バルクメモリ）を分離し，各々独立したアドレス空間を持たせて能力を上げている。

内部メモリはバイトアクセスされるが，バルクメモリは，

①バイトアクセスし，バンクを増す．

②Lispの基本単位（セル）を1ロケーションとした1バンクを構成し，語アクセスする。

の2方法が考えられる。①は一般的ではあるが，Lispではセルアクセスできればよいので②より低速となる。②の方法での問題点では内部メモリとアクセス幅が異なる（バイト対語）ことである。この点を解決することができればLispセルと処理系を切り離すことができ，かつ1ロケーション1セルとなるので，アドレス情報の処理が容易となり，8080を用いても比較的高速にデータを参照することができる。このために考えられた機構を次に示す。

6.3.3 バルクメモリ・RAMインタフェイス

バルク上の1セルは32ビット＋3ビットのフラグより成る。(他にパリティ用に1ビット使用。) 8080の命令ではこうした語データを処理することはできないため，内部メモリのRAMへ転送し処理を行うこととする。通常の入出力命令を介していたのでは(DMAチップの起動であっても)遅いので，転送用の命令を擬似的に追加する。転送の起動には5クロックしか要しない。命令実行時のアドレスバス16ビット，データバス8ビットのみにより転送制御情報はすべて合成され，他のメモリ参照は行われない。

転送において，バルクアドレスはすべて任意に指定できねばならないが，RAMアドレスは専用機である特質を生かせば任意性は不要で，複数指定できれば充分である。Lisp処理系の場合，次の指摘ができる。

①サブルーチン(Subr)呼出しの引数はたかだか4個である
②リンクのたどり，スタックの読み出しなどには，たかだか2個のバッファがあればよい
③再帰呼出しの数は比較的少ない

以上のことから作業域も含め，RAM側には16個のバッファがあれば充分である。そして引数の受渡しはバッファ上でそのまま行え，RAM上での二次的転送は不要である。

16通りのRAM側バッファの先頭アドレスはこのため動的に変える必要はなく，実際の転送時には識別のために4ビット(0～15)あればよい。作成された回路は次の4点にまとめられる(図6-6)。

1) RAMアドレス変換機構としてAAM(Address Association Memory)を持つ。転送時にはRAM論理アドレスとしてAAMレジスタ番号を指示し，実番地を生成させる。AAMは16ビット16語のRAM(アクセス

30 nsec) である。

2) 転送命令として，transfer (TR) 命令を擬似的に1バイト命令に加え，最小のCPUサイクルの消費で転送が行われる。本来必要なTR命令は3オペランド命令で，次の形式をもつ。

TR C, addr1, addr2

Cは転送制御を表わし，現在3種類存在する。

C = 1: (Balk[addr1]) ⟶ (Ram[AAM[addr2]], …, Ram[AAM[addr2]])

C = 2: C = 1 の逆方向転送

C = 4: AAM[addr2] ⟵ addr1;

ここで $0 \leq addr1 \leq 65535$

$0 \leq addr2 \leq 15$

このTR命令のために1バイト命令のうち，未実装領域(32768～)へのストア命令(「MOV M,r」及びSTAX命令)を代用し，Cに3ビット addr1に16ビット，addr2に4ビットをあてる(図6-7)。この結果入出力接続では100 μsec以上を要するバルク1語・RAM5バイト間の転送を約20.5 μsecで行う。回路は常時プロセッサから出されるメモリストア命令を監視する。

memory write cycle でかつ，アドレスバスの最上位bitが1であるとき，データバス8ビット，アドレスバス16ビットをラッチし，CPUをホールドさせ転送を行う。

3) 変則データバッファ(図6-8)を持つ。このデータバッファはバイト単位と36ビット単位の2種類のアクセスが許されている。RAM，バルク間の転送時にはパリティ生成・チェック及び第5バイトの下4ビットの無視・0の挿入が行われる。

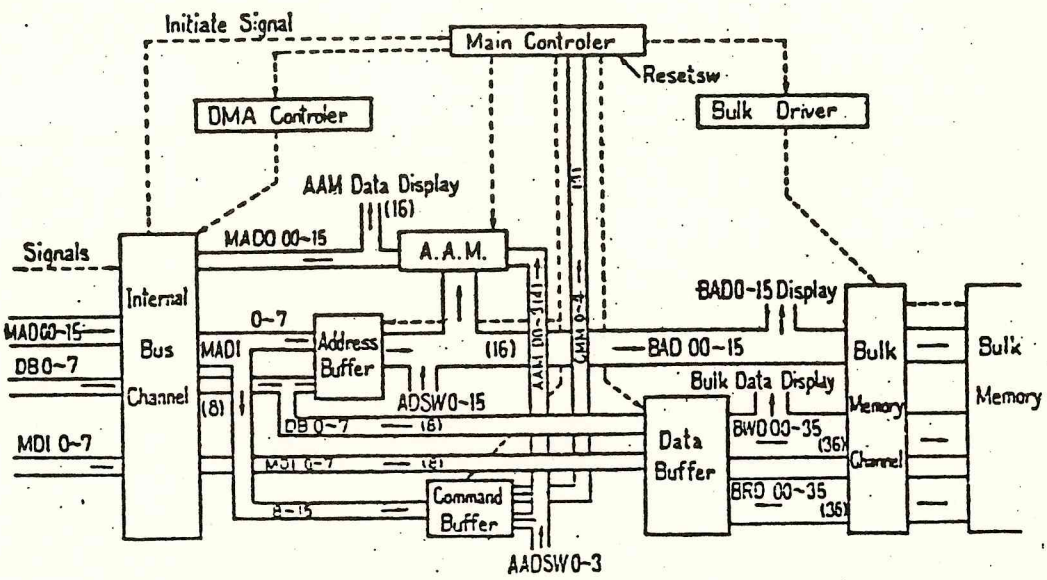


図 6－6　バルク-RAM インタフェース

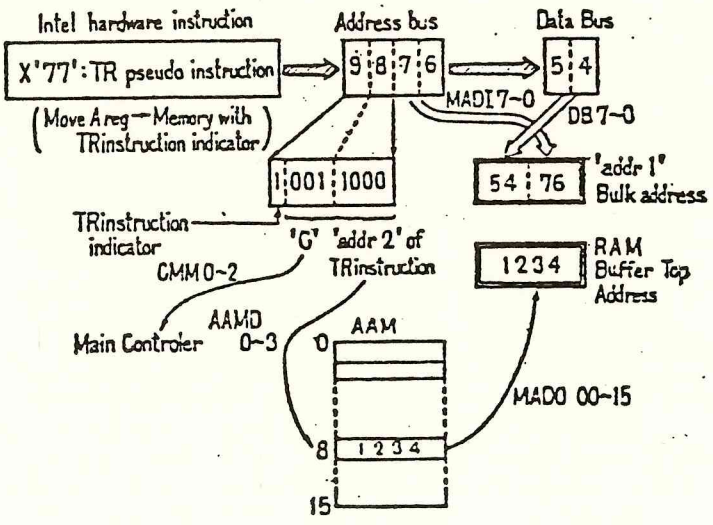


図 6－7　TR 擬似命令からのバルクアドレス及び RAM アドレスの形成例

4)独立した保守パネルがあり，リセット機能の他，バルクメモリ・AAMの内容の表示，データ転送エラー，コマンドエラーの表示と動作の停止を行う。

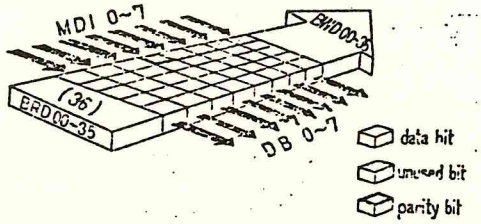


図 6－8　変則データバッファの構造

6.3.4　LISPプロセッサ構成

Lispプロセッサは　IBM370／135（後の370/138）上に作成されたクロスソフトウェア（アセンブラ及びシミュレータ）上でハードウェア作成・改良と並行して開発された。（またその開発には，370及びミニコンMelcom70が用いられ，それらの上にはいくつかのtoolがつくられた。）昭和51年1月に作動したLispプロセッサ初版をもとに改良がくり返されたが，昭和52年8月に大幅に変更され，現在に至っている。

このLispプロセッサはLisp1.5と対比した場合，次の特徴をもっている。

1) hashingによるアトムの格納，p-listはない。そのかわりに2)で述べる機構がつけられており，それによりP-listの機能は代行されている。
2) hashing機構を生かしたハッシュ化配列（連想三つ組の保持可能），連想計算機能をもつ。システム識別子Harray，Hexprを各々導入し，これらのインタプリタへの簡潔な組込み。
3) stackを用いたdeep bindingによる変数束縛
4) evlisの結果の即時回収
5) Lisp1.6型PROG＋ラベルの高速サーチ

6)機械語ルーチンの一時的なロード機能（PROMプログラマ，各種保守ルーチン，ユーザー組込subr等）

メモリ構成を図6－9に示す。バルクメモリ及びRAMの初期化は電源投入時に動作するルーチンにより行われる。（この間約2、33秒）

主記憶空間は次の順に割当てられる。

1) システム常駐域Ⅰ（0～1FFF番地）
2) スタック領域（2000～2FFF）
スタックポインタにより直接利用される領域，3072段可能である
3) バイナリプログラムスペース（3800～3DFF）
擬関数loadにより機械語プログラムをロードできる。
*fn2の名によりそのプログラムをLisp内から呼びだすことができる
4) システム作業域（3E00～3FFF）
5) システム常駐域Ⅱ（4000～5FFF）

またバルクメモリは次の順に割当てられる。

1) 変数束縛用スタック
1024段のスタックでcar部に引数，cdr部に値をもつ
2) フルワードスペース
基本整数以外の数が格納される。数の一意性は保証されている。
3) Prog中間言語ロード域
4) GC保護用スタック
1024段のスタック
5) アトムストリング域
アトムのpname及びストリングアトムの文字は各々4文字を越える場合その残りはこの領域にチェインされる

6) H-領域
　2語1組で利用される
7) フリーストレージ
　56Kセルを格納できる。

このバルクアドレス空間は，基本整数（0～1023）はアドレス自身を値とするなど，データの処理効率と識別性を高めるように配列されている。
たとえば numberp 及び atom 述語関数は，

numberp[x] = [addr[x] < 2048 → T; T → NIL]
atom[x] = [addr[x] < 8192 → T; T → NIL]
（ここで addr[x] は x の格納された各部を値とする関数とする）

と定義され，一回の比較により値が決定される。（Lispデータ空間の序列については　たとえば[KUR 76]などの文献がある。

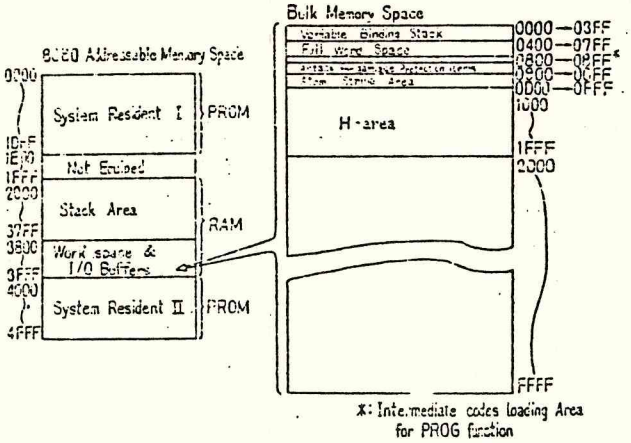


図 6－9　メモリ割付けと階層構造
Memory allocation and hierarchy.

ALPS/Iにおけるガベジコレクタは3段階に動作する

1)即時回収コレクタ

インタプリタ内で生成され，即座に不要となるものは通常のガベジコレクタの動作を待たずに各々の不要になった時点で回収する。たとえば 実引数リストの作成を行うインタプリタ内のevlisの値は目的関数の適用後は不要で，即時回収される。この即時回収を行うのに要するコストが過大であれば 却って不利益をもたらすものとなるが，evlisの場合後述するように定義をかえるだけで容易に回収をすることができる。各種の例題の実行においては消費セルのうち約20%が即時回収されている。

2)ガベジコレクタ(GC)

GCはフリー領域が一杯になったときに動作する。回収に先立つてマーキングは後述されるH-分子のうち属性値1から6の値部，属性値8,9の値部及び鍵部，束縛スタック，GC保護用スタック，current cons 対象の順に行われ，フリー領域の線形走査を行い，マークされなかったセルの回収及びマークの除去がなされる。

3)グランド ガベジコレクタ(GGC)

GGCはH領域のオーバーフロー時に動作する。GCに加えて不要になったH分子及びフルワードセルを回収する。

H領域は2048エントリを持つ4K語の領域で，H分子を格納する。H分子は属性部・鍵部・バリアント部・値部及びステータス部を持ち，システム内での一意性が保証されているものをいう(図6-10)。

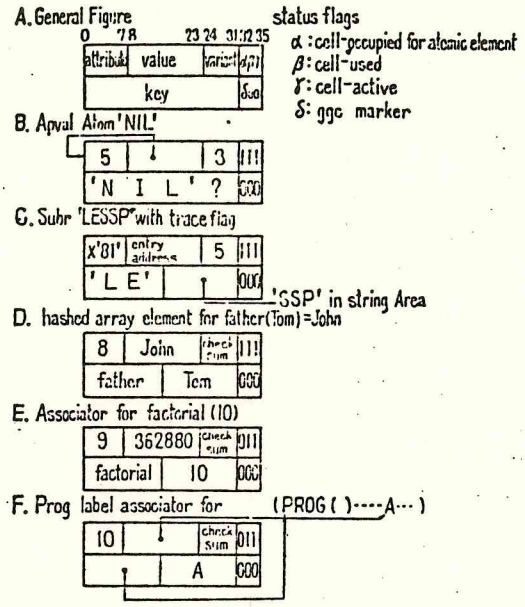


図6-10　H分子の構成例

```
hstore [h; v; va; k; attribute]=prog [[slot];
        slot=h;
  LOOP [cell-not-active [slot]→return [make-hmolecule []];
        and [eq [attribute; get-attrib [slot]];
             eq [variant [slot]; va]; eq [key[slot]; k]]→
                return [store-value [slot]]];
        slot =rehash [slot];
       [eq [slot; h]→ggc[]]; go [LOOP]]
```

図6-11　H分子の生成手順

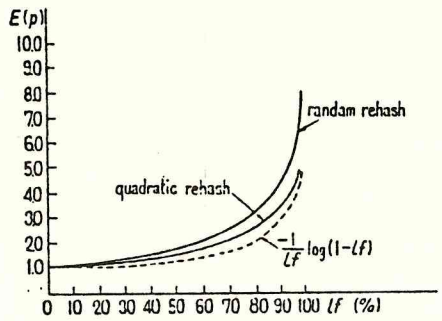


図6-12　非ランダムデータに対する平均探査回数

このH分子hに対しては　システム用に次の7種の基本関数が定義されている。

a)属性値の取り出し　　get-attrib[h]=a
b)値の取り出し　　value[h]=V
c)バリアント部の取り出し　variant[h]=va
va はatomic symbolに対しては keyとなるPnameの文字数，associator等には鍵部の情報のチェックサム
d)鍵部の取り出し　　key[h]=k
kはatomic symbolに対してはPname string，他の場合は鍵となる（key1*. key2*）のドット対
e)H分子の作成　　hstore[h:v:va:k;attribute]=h'
h'は（v, va, k, attribute）により決定されるH分子の格納される番地。h番地がすでに他のH分子により占められている場合には，空きがあるまでrehashされ，h'が決められる。このときand関数の定義の通りにバリアント部が等しい場合にのみオフワードがアクセスされ，keyが比較されるのでバルク参照ワード数が削減されている。（図6-11）
f)H分子の削除

delete[h]=set[status[h];not-active-bit-used]

さらに(a　o　v)型の連想子に対しては，
g)keyとしてa及びoを持つH分子の値vの取り出し

hassoc[a:o]=v

H分子の格納はhashingにより行われる。collisionはrehashにより解決する。rehashing手法はR=1としたquadratic searchを用い，次の手順により行う。

① Set k=hash[key], j=R
② もし k番目の要素が空であるか，等しいkeyを持っていれば終了
③ さもなければ k=k+j, j=j+1 としてステップ2へ戻る。

このときの平均探索回数E(p)は

$$E(p) = -(\log(1-p))/p \quad , \quad p: \text{load factor}$$

で与えられる。またサーチの最大区間は N-R+1である。
このrehashアルゴリズムは以前に用いられていたrandom rehashと比較して高速であり，またload factorが増えても実質探索回数がそれほど増えないように改良されている。(図6-12)

このように扱われるH/分子の生成及び消去の手順は表6-1にまとめることができる。

	atomic symbols								associators		
attribute name	—	subr	fsubr	expr	fexpr	apval	hexpr	harray	harray element	assoccomp associator	prog label associator
attribute value	0	1	2	3	4	5	6	7	8	9	10
created by	read etc.	—	—	define deflist	deflist	cset csetq	deflist assoccomp	array	seta	hexpr function refer	prog label
deleted by	ggc	—	—	—	—	—	—	dearray	delete dearray	ggc	ggc

表6-1　　素情報と連想子の分類

Lispインタプリタは Apply, Eval, Evlis, Evcon とよばれる副関数をもつ万能関数Eval quoteとして知られている。ALPS/Iに組込まれたもののうち，apply, eval, evlisのM式による定義を図6-13に示す。

Lispインタプリタは1命令あたり4バイトと，1バイト命令を比較的多用して構成されている。各々のsubr，fsubr関数の記述は次の規則を基本とする。

①引数はバルクバッファを直接用いる。
②引数の退避が必要となるモジュールにおいては　引数の性質に応じて，ハードウェアスタックないしはGC保護用スタックへのスタック命令を随時記述する
③関数の値は　D，Eレジスタへ返す

この規則の設定により手続呼出しのオーバーヘッドは　極小化されている。

作成されたインタプリタについては関数引数の処理とprogの処理について次に説明する。

関数引数は optional Freezeにより処理する。すなわち関数引数において，凍結の必要のある自由変数は第三引数以降につなげ，関数引数の実行は　凍結された値の束縛を行ったのちに行われる。この機構は＊functionにより引用するときに働き，通常のfunction関数による引用はQuoteと同じ処理がされる。

progインタプリタは中間言語域を使用して展開を行い処理する。展開に際してはラベル連想子の登録，prog変数のlocal宣言の追加が行われる。go文はLisp1.6の仕様に従い，飛び先を動的に指定できる。ラベル連想子により行先が保持されているので　登録されるラベル数に依存せずに高速に分岐が行われる。アトム引数のgo文の場合は中間言語域上の対応する部分が一度go関数に与えられると，無条件分岐を行う＊＊goに書き換えられ，short-cutが生じる。これを図6-14に示す。

```
                    (Evalquote, Evcon, Suppressed)
apply [fn; args]=prog [[templ; temp2; v];
 [atom [fn]→[eq [get-attrib [fn]; EXPR]→return [apply [value [fn]; args]];
              eq [get-attrib [fn]; SUBR]→return [progn [spread [args]; call [value [fn]]]];
              eq [get-attrib [fn]; HEXPR→[hdotp [fn; args]→return [
                                  hassoc [fn; args]]; T→progn [v=apply [value [fn]; args];
                                  hstore [fn; args; v]; return [v]]];
              eq [get-attrib [fn]; APVAL]→return [value [fn]];
              T→return [apply [sassoc1 [fn; ERRORA2]; args]]];
 templ=car [fn]; temp2=cdr [fn];
 [eq [templ; LAMBDA]→progn [stacksave []; stackpush-loop [car [temp2]; args];
                                  v=eval [cadr [temp2]]; stackrestore []; return [v]];
  eq [templ; LABEL]→ progn [stacksave []; stackpush [car [temp2]; cadr [temp2]];
                                  v=apply [cadr [temp2]; args];
                                  stackrestore []; return [v]];
  eq [templ; FUNARG]→progn [stacksave[]; stackpush-loop [cadr [temp2]; caddr [temp2]];
                                  v=apply [car [temp2]; args];
                                  stackrestore []; return [v]]];
   return [apply [eval [fn]; args]]]
 eval [form]=prog [[templ; temp2];
  [numberp [form]→return [form];
   atom [form]→return [[eq [get-attrib [form]; APVAL]→value [form];
                                  T→sassoc1 [form; ERRORA8]]];
  templ=car [form]; temp2=cdr [form];
 [not [atom [templ]]→progn [v=apply [templ; evlis [temp2]]; igc [*last]; return [v]];
  v=get-attrib [templ];
 [ev[v; EXPR]→progn [v=apply [value [templ]; evlis [temp2]]; igc [*last]; return [v]];
  eq [v; FEXPR]→return [apply [value [templ]; cons [temp2; NIL]];
  eq [v; SUBR]→progn [spread [evlis [temp2]; v=call [value [templ]]; igc [*last]; return[v]];
  eq [v; FSUBR]→return [call [value [templ]]];
  eq [v; HEXPR]→return[[prog2 [w=evlis [temp2]; hdotp [templ; w]]→
                          hassoc [templ; w]; T→progn [v=apply [value [templ]; w];
                          igc [*last]; hstore [templ; w; v]]];
  eq [v; HARRAY]→return [hassoc [templ; [eval [car [temp2]]]]];
  return [eval [cons [sassoc1 [templ; ERRORA9]; temp 2]]]]
evlis [m]=[null [m]→setq [*last; NIL];
          null [cdr [m]]→setq [*last; cons [eval [car [m]]; NIL]];
          T→cons [eval [car [m]]; evlis [car [m]]]]
hdotp [x; y] is a presence predicate.
      if the associator having (x°. y°) key is found, then true else false.
hassoc [x; y] is a function to obtain the value of the associator having (x°. y°) key.
```

図6-13 Lispインタプリタの M表現

```
go[x]=[atom[x]→[addr=hassoc [x; pform]→
                    progn [change-statement [**go; addr];
                           seq-counter=addr];
                    T→label-error[]];
       addr=hassoc [eval [x]; pform]→addr;
       T→lube-error []]
**go [x]=setq [seq-counter; x]
pform is an atom, whose value is a current prog-body
      top address, seq-counter is an atom to indicate a
      next prog statement to be eval-ed.
```

図6-14 go及び**goのM表現

この中の change-statement [x; y] は，現在sequence counterが指した中間言語のfunction部をxに，operand 部をyにおきかえる擬関数とする。この修飾は展開されたコードに対して行われ，元のS式はそのままで他の処理には全く影響しない。

6.3.5 ALPS/Iの評価

ALPS/Iは基本的には 1)経済性 2)機能 3)速度 の順に評価基準をおいていたが，これらは満足したといえる。表6-2は記号処理シンポジウムにおいて設定された，標準問題[IPS74]の実行結果でミニコンのもの[NAG76]を一例に比較したものである。

ほぼ同等の実行時間であるが，progインタプリタの高速化のためにsortの問題では速い値となっている。こうした程度のミニコンはALPS/Iに比して おおむね3倍以上の価格であるので，価格/性能比にして3倍以上の能力が得られたといえる。また，その後行われた処理系コンテストの結果を見ると超大型機のLispの100倍程度の時間を要する。

それらはLisp専用ではなく，汎用機であるので直接的な比較はできないが，ハードウェア価格はALPS/Iの100倍は越えている。また，現在の時点で製作すれば，ALPS/Iと全く同等のものを3分の1程度の価格で製作できる点からも 価格/性能元 志向は達成されたといえる。

機能的には 56Kの自由領域を用いて，今まで小型機のLisp処理系では扱えなかった既存大のLispプログラムの多くは，実行できるようになっている。Lispに基づく数式処理言語REDUCEは，そのプログラム自身で約44K語を占めるが，現在小型の応用問題なら処理できるようになっている。(図6-3)

また人工知能研究の一部であるTheorem Prover([CHA73])の実行も可能であり，実行時間を表6-3に示す。

	ALPS/I		Reference data	
	Execution Time* (msec)	Used cells	Execution Time (msec)	Average GC Times
WANGA	176	825	100	0
WANGB	970	1,235	600	0
BITA 6	8,750	4,738	5,532	0
BITA 7	29,600	15,536	22,432	0.3
BITA 8	99,500	54,514	75,925	0.8
BITB 6	1,980	1,283	1,495	0
BITB 7	5,950	3,476	4,345	0
BITB 8	19,400	11,145	15,825	0.1
Sort 60	77,500 (54,500)**	31,402	92,825	1.0
Sort 80	110,000 (83,000)**	47,227	164,200	1.7
Sort 100	127,000 (104,500)**	over 56 k	245,350	2.5

*: ALPS/I does not require garbage collection to process these test programs.
Time required for iterative execution of test programs was measured with a stopwatch.
**: When using system built-in subr EQUAL, APPEND not as expr.

表6－2 テストプログラム の実行時間
Execution time obtained from test programs.

CPU	ALPS/I 8080	L1.6* PDP-10	HLISP** H-8800	UTLISP4.1*** CDC6600
TPU-1	87.0	1.55	4.79	24.64 SEC.
TPU-2	390.0	7.51	13.41	70.71
TPU-3	157.0	3.42	5.68	29.43
TPU-4	194.0	2.65	8.03	41.52
TPU-5	28.0	0.67	0.87	4.38
TPU-6	820.0	2.30	22.85	123.2
TPU-7	150.0	3.56	5.08	26.84
TPU-8	142.0	5.18	3.46	18.68
TPU-9	133.0	4.05	2.66	13.78

* ORIGINAL PROGRAM'S DATA[CHA70]
** TOKYO UNIVERSITY'S LISP[GOT74A]
*** UNIVERSITY OF TEXAS'S LISP
DATA FROM [TAK78] EXCEPT L1.6

表6－3 EXECUTION TIME OF SAMPLE PROGRAMS[CHA73]

6.4 本管理システムの適用例1
――情報検索的基本操作の構成例

今までの理論の実現性を示すために構成した。本節の内容は〔IDA79D〕で発表したものである。

時間給制による従業員給与（payroll）システムを一例にとり，それに対してモデルを適用し有効性を確認する。

対象となるファイルは従業員名簿としてまとめられる。各員に対しては 時間給及び日々の労働時間が入れられる。図6－16にその構成を示す。各項目はハッシュ化配列連想子として，図6－15のように格納される。

図6－16の構成は 文字列を添字とする二次元の表といえるが要素はすべて満たされなくてもよく，疎な入力でも記憶効率がよい。このシステムの単位操作と実際のコマンドとの対応を表6－4に示す。

（このシステムは連想子の応用を例示するために 極力小さく作成されている。これをさらに現実的にするには，いくつかの項目の追加といくつかの手続の追加が必要であるが本質的な変更は不要である。）

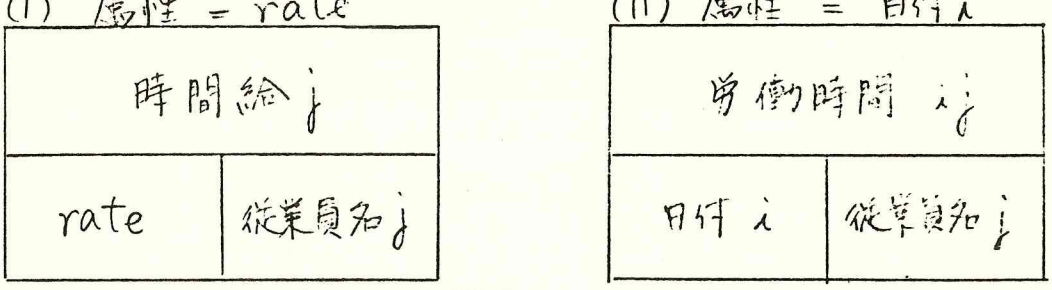


図6－15 連想子としての記憶。

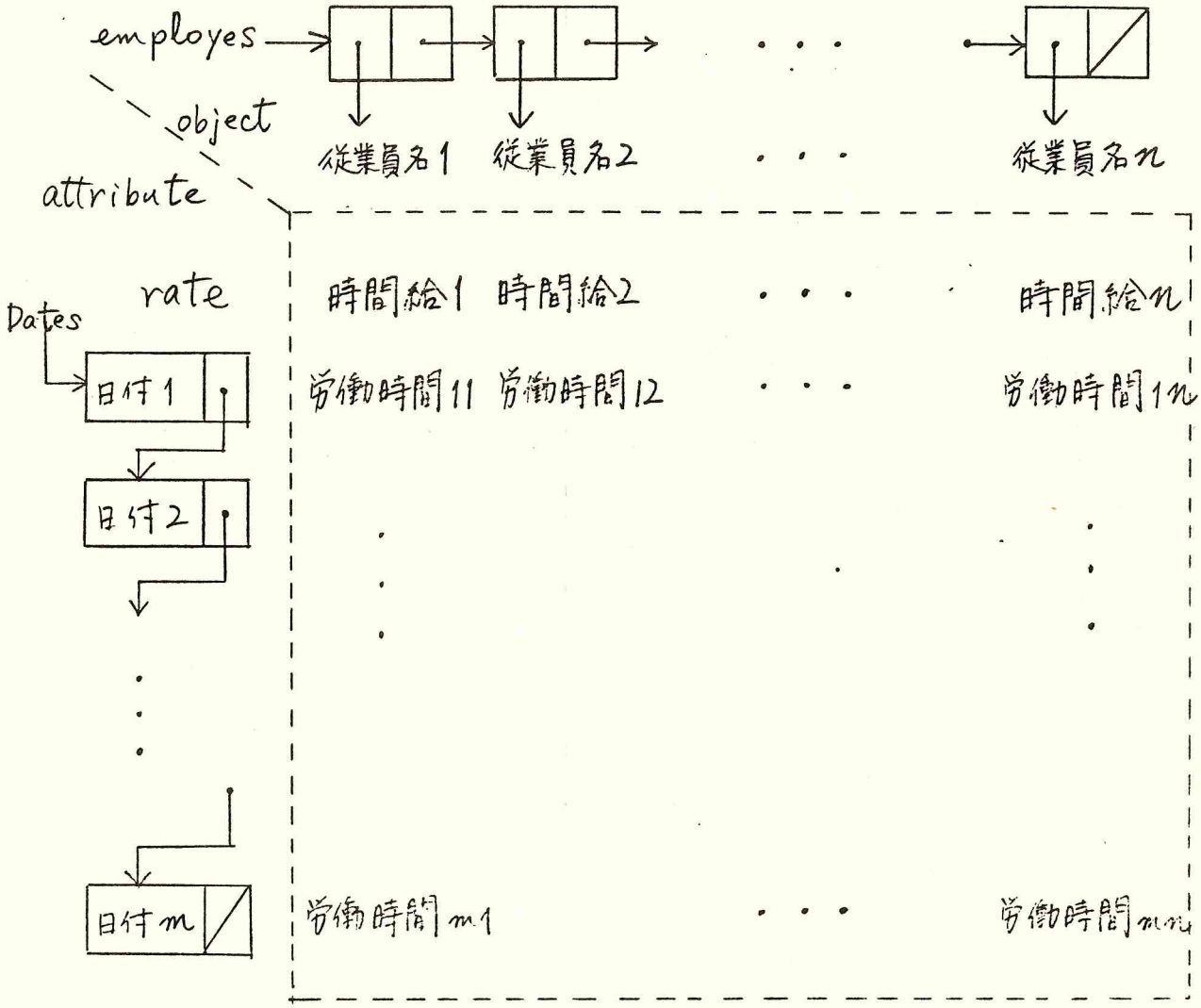
そしてこの単位操作に基づく会話型処理系のプログラムリスト（初期化部分を除く）を図6－17に，その実行例を図6－18に示す。

またこの会話型システムに対して「仮想データ」を導入することができる。連想計算機構を指定することにより，「一度手続的に計算した値は覚えておく」ことが行われる。これにより使用者は計算手続を書くだけでよい。二次的なデータの格納はシステムにより自動的に行われ，値の2度目以降の計算は実際に生ぜず連想子検索に自動的に切りかえられる。

表6－4　payrollシステムにおける単位操作

	単位操作	コマンド	機能
1	従業員の追加	ADD-EMP	従業員名簿への追加と時間給の設定
2	従業員の削除	REMOVE-EMP	従業員名簿からの削除
3	新日付の設定	NEWDATE	新しい日付を追加
4	労働時間の格納	SET	(日付，従業員名，労働時間)の連想子の作成
5	連想子の参照	DISPLAY	指定された連想子の表示
6	連想子の除去	ERASE	設定ミス等に対応して連想子を除去
7	給与表の作成	PRINT	各従業員の給与の印刷

図6-16 連想子によるpayrollシステムの構成
(破線内が連想子となる。それらは疎に散らばっている。)



たとえば「1月1日に労働を行ったもののリストアップ」が必要ならば、この手続を遠想計算指定すると2度目以降の参照は一度目に自動的に作成された連想子の検索に自動的におきかえるので、値を求める手続を無駄に実行させる必要がない。

Lispの特質をいかして、このようなデータ中心の検索システムに対して、推論的な扱いの導入も最近の話題の一つであり、それらの成果をとり入れることもできよう。

初期化部分

```
 PROGN(
(ARRAY RATE COMMAND)
(CSETQ DATES NIL)
(CSETQ EMPLOYES NIL)
(SETAQ COMMAND(QUOTE SET)(QUOTE(SETA(GETITEM)(GETEMP)(GETVALUE))))
(SETAQ COMMAND(QUOTE ERASE)(QUOTE(DELETA(GETITEM)(GETEMP))))
(SETAQ COMMAND(QUOTE DISPLAY)(QUOTE(PROGN(PRIN1(EVAL(LIST(GETITEM)(LIST(QUOTE QU
OTE)(GETEMP)))))(PRINT (QUOTE $$$ IS IN IT$)))))
(SETAQ COMMAND(QUOTE NEWDATE)(QUOTE(PROGN(GETDATE)(EVAL(LIST(QUOTE ARRAY)D))(CSE
TQ DATES(CONS D DATES)))))
(SETAQ COMMAND(QUOTE PRINT)(QUOTE(OUTPUT(MAKE-WAGE-LIST))))
(SETAQ COMMAND(QUOTE ADD-EMP)(QUOTE(PROGN(CSETQ EMPLOYES(CONS(GETEMP)EMPLOYES))(
GETRATE))))
(SETAQ COMMAND(QUOTE REMOVE-EMP)(QUOTE(DELETE EMPLOYES(GETEMP))))
(SETAQ COMMAND(QUOTE FIN)(QUOTE(RETURN(QUOTE PAYROLL-END))))
```

会話型部分

```
DEFINE((
(MAKE-WAGE-LIST(LAMBDA()(MAPCAR EMPLOYES(QUOTE(LAMBDA(K)
     (CONS K (MULT(RATE K)(TOTAL-HOURS K)))))))
(TOTAL-HOURS(LAMBDA(X)(PROGN (LOCAL I J) (SETQ I 0)
     (MAPC DATES (QUOTE (LAMBDA(Y)(COND((SETQ J(ERRSET(Y X)T))(SETQ I (SUM I J))
)))))
     I)))
(PAYROLL(LAMBDA()
 (PROG  (D COM)
LOOP  (TERPRI)(TERPRI)
      (PRIN1 (QUOTE $$$COMMAND ? $))
      (SETQ COM (READ))
      (EVAL (COMMAND COM))
      (GO LOOP)
)))
))
```

図 6-17　payroll システム処理系

```
EVALQUOTE ENTERED, ARGUMENTS...

PAYROLL NIL

COMMAND ? ADD-EMP
 EMPLOYE-NAME ? TOMY
 RATE ? 5                    時給500円

COMMAND ? ADD-EMP
 EMPLOYE-NAME ? JOHN
 RATE ? 4                    時給400円

COMMAND ? NEWDATE
 DATE ? S53-06-06            53年6月6日

COMMAND ? SET
 ITEM ? S53-06-06
 EMPLOYE-NAME ? TOMY
 VALUE ? 6                   労働=6時間

COMMAND ? NEWDATE
 DATE ? S53-06-07            53年6月7日

COMMAND ? SET
 ITEM ? S53-06-07
 EMPLOYE-NAME ? TOMY
 VALUE ? 5                   労働=5時間

COMMAND ? SET
 ITEM ? S53-06-07
 EMPLOYE-NAME ? JOHN
 VALUE ? 7                   労働=7時間

COMMAND ? DISPLAY            時間給の確認
 ITEM ? RATE
 EMPLOYE-NAME ? TOMY  5 IS IN IT


COMMAND ? PRINT              給与表の印刷

      WAGE LIST

      JOHN = 2800 YEN
      TOMY = 5500 YEN

TOTAL 2 EMPLOYES


COMMAND ? FIN                終了

END OF EVALQUOTE, VALUE IS..
PAYROLL-END
```

図6-18
payroll 実行例
(コマンドは表6-4に示す。下線部が使用者の入力.)

6.5 本管理システムの適用例2

――数式処理システム作成における適用例

人工知能応用分野の一つとして数式処理システムがある。これは基本的には数式の変形・整理等の記号的な操作を実行するシステム概念である。数式処理と呼ばれるこうした応用分野は人工知能の中でも用途と効果のはっきりと見積もることのできる分野である。しかし，数式処理システムは実行ステップ数・所要記憶量の両面で大規模であり，従来は大型機でのみ実現されていた。この数式処理システムをコンパクトなシステムとして作動させることができたならば，数式処理の普及に役立てることができるというのがALPS/I開発時の背景の一つでもあった。

数式処理システムとしてはMACSYMA，REDUCE等が知られている。ALPS/IではREDUCEを移植する。REDUCE[HEA73]はLispで書かれた数式処理言語である。REDUCEにおいて必要となる基本機能はstandard-Lisp[HEA69]としてその機能が定められている。それらの機能はすべてALPS/Iに用意された機能を用いて直接果たすことができる。

ALPS-REDUCEシステムの才一版[KOB77B]は，解析も含めて約4か月間で実動させることができた。64K存在する空間のうち40Kを占める大きさであった。才一版は行列演算機能等を削除したものであった。また新版のソースプログラムを入手することができたので，それをALPS-REDUCE才二版[TOO78]として製作した。この版の改良・保守作業は79年度末まで行われ[HEA73]に示すすべての機能を動作させることができた。400Kバイト以上必要とされるこのシステムを約260Kバイトの領域で実現できたことは，連想子機構の利用による処理系の縮少によるところが大きい。

◎基本機能の実現手法

REDUCEにおいて前提とされている基本機能は，約80種存在する。その多くは 本章で述べたシステムALPS/I中に機械語によって組込まれた基本機能と合成している。それ以外のものはALPS/I中の連想子機能及びそれを含めたALPS/I基本機能より合成される。その詳細については[KOB77B] [TOO78]などに述べられている。

ここでは，主な基本機能の本システムでの実現手法についてまとめる。

(1) LITER

機能：素情報の名称（1文字）が英字（A～Z）であるかを判別する。

LITER[x] = True if x is alphabetic
else NIL

ALPS/Iでの実現：

前処理とLITERに代わる本体の2段階で対応する仕事を行う。

まず前処理としてALPHAという属性を設定する（ハッシュ化配列宣言）。次に，

(A ALPHA T)
(B ALPHA T)
⋮
(Z ALPHA T)

という26個の連想子を作成する。

このような前処理をしておくと LITER[x]は,

LITER[x] = errset[alpha[x]; T]

として定義できる。

ここでerrset[exp; T]はexpの実行中にエラーがあっても,エラーメッセージを出力せずにただ,値をNILとして処理を続行し,通常はexpの値で処理を続行する機能とする。

alpha[x]は,

(x, alpha, ?)

という連想子検索に相当している。

たとえば 5は英字でないので, LITER[5]は偽になるはずである。この処理は次のようになる。

alpha[5] の引用

により (5, alpha, ?) の連想子検索が行われる。そうした連想子は存在しないのでエラーメッセージを出力し,値NIL(偽)を返そうとする。このときalpha[5]はerrset下で囲まれているので エラーメッセージは出力されず値NILのみが返される。

また, LITER[E]はEが英文字なので真になるはずである。

(E, alpha, ?) という連想子検索が行われ,属性値として格納されていたT(真)が値となる。

(2) FLAG, FLAGP, REMFLAG

機能:素情報に識別フラグをつける(FLAG),はずす(REMFLAG).

あるいはフラグの有無を判別する。

ALPS/Iでの実現：

連想子の生成・削除・参照を直接に利用する。

FLAG[x; flag] = seta[flag; x; T]

FLAGP[x; flag] = errset[flag[x]; T]

REMFLAG[x; flag] = delete[flag; x]

（実際にxはフラグをつけたい素情報のリンクリストを与える。本質的には上記の通りである。）

(3) GET, PUT, REMPROP

機能：素情報に属性と属性値をつける(PUT)，参照する(GET)，あるいは削除する(REMPROP)

PUT[atom; property; value]
= seta[property; atom; value]

GET[atom; property]
= errset[property[atom]; T]

REMPROP[atom; property]
= delete[property; atom]

◎FLAG, PROPERTY関係の前処理

格納にseta 関数，参照に連想子参照，削除にdelete関数を利用するためには，使用する属性名もしくは識別フラグ名は前もって属性名宣言をしなければならない。

REDUCE処理系の中には動的に属性名や識別フラグ名を生成することがないので，前もってREDUCE処理系ソースリストを調べ列挙しておく。

初版においては 53個の属性名が宣言されている。

◎連想子機構利用の効果

連想子の利用は，①速度②所要記憶量の両面における効果が認められる。①の速度効果は4.2に示した。ここでは②の所要記憶量について述べる。

前述したように（素情報，属性，属性値）の対はリンクリスト形式により記憶することもできる。しかし一つの対を記憶するのにリンクリスト形式では3つのセルが必要であり，これを属性別に連結するために1つセルが必要である。従って合わせて4セルを要する。素情報空間中に配置する連想子は一対につき2セルを要する。式の処理を行うときに，例に500個の連想子を必要とするとき，ALPS/Iの方式では1Kセルですむが，通常のLispの機能を利用した場合2Kセルが必要となる。このように参照の高速性に加えて所要記憶量が少ないことは，処理を進める上で有利である。

6.6 本管理システムの適用例3
——breadth-first型探索における適用例

箱入娘パズルの解法を例にとり，breadth-first探索における連想計算連想子及びハッシュ化配列連想子の有効性を確認した。[IDA77B]

まず文字通りハッシュ化配列は配列としても用いることができるので，各種の表及び盤面の記憶に用いられた。

箱入娘パズルに代表される盤面ゲームの解法の発見には次の手順に従う。

初期配置s_0を含む初期配置集合S_0及び最終配置s_nを含む最終配置集合S_nの間を結ぶ経路d_i（$1 \leq i \leq n$）を発見するために次の手順に従う。

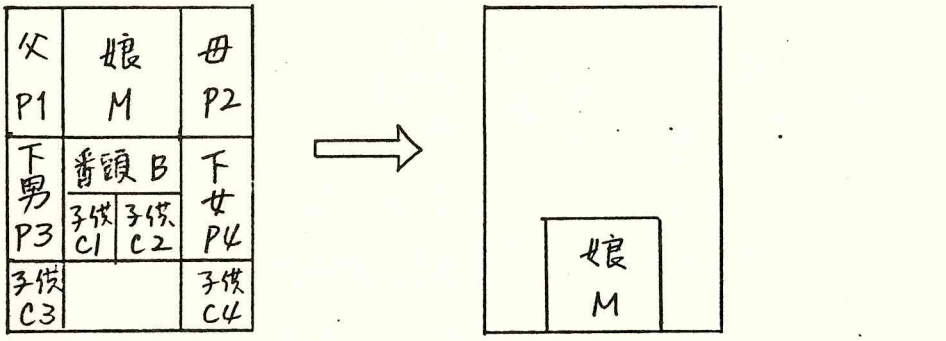
(1) S_{i-1}の各々に対して移動可能な配置の集合Hを作る。
そして $H - S_{i-1} - S_{i-2}$ によりS_iを作る。

(2) このS_iがS_nであれば(3)へ。さもなければ(1)へ。

(3) S_nから始めてS_iとS_{i-1}とをつなぐd_iを順にたどり，これを求める手順とする。

このときの(1)におけるS_{i-1}, S_{i-2}のマーキング，(2)におけるS_n終了判定，(3)における帰路手続などに連想子を利用し，高速化できた。特に(3)における帰路の記述に対しては，(1)の経路の手続をそのまま活用し，連想計算を指定することにより簡潔に高速な手続を記述できた。以上のようなことから，連想子による高速参照とそれらのリンクリスト結合による情報構造の有用性は確認できた。

箱入娘パズルは，初期配置C₀から娘を門の所へ連れ出した最終配置Cfに至る最短経路を見いだす問題である。



初期配置C₀　　最終配置Cf
（M以外の物の位置は任意）

初期配置C₀から1手で移りうる配置の集合をS_n，S_nに属する配置から1手で移り得る配置の集合を$S_{move}[S_n]$とする。このときS_{n+1}は次式で与えられる。

$$S_{n+1} = S_{move}[S_n] - S_n - S_{n-1},$$
$$S_0 = \{C_0\},\ S_{-1} = NIL$$

S_nの中にCfが含まれたら次式に従って，この目的配置に到達する手順として各々S_nよりd_nを1つだけ取り出す。

$$d_n = try[*S_{move}[d_{n+1}]; \lambda[[j]; member[j; S_n]]]$$
$$d_0 = C_0,\ d_n = C_f$$

ただし try[x; fn] は試行をxの各car部に対して繰り返し，NIL以外の値を持つ要素が発見されれば，それを値として帰るというALPS/I組込みのシステム関数である。箱入娘パズルは連想子の効用に加えて　このtry関数の有効性を示す実例にもなっている。

```
try[x; fn] = prog[[xx];
    xx := x;
Loop[atom[xx] ⟶ return[xx]
```

```
fn[car[xx]] ——→ return[car[xx]]
xx := cdr[xx]
go[Loop]]
```

計算機によってこのパズルは既に解かれており，n=81である。Lisp系の言語を用いた解としては，LiPQによる19分31秒というものが76年記号処理委員会報告集に報告されている。また，その手続の概略はbit誌Lisp入門No.13 (1974) に完全に記述されており，我々もこの手続をもとに作成した。

```
nextconf(n;snl;sn) = prog((dn);  print(list(n;length(sn)));
                                 (null(sn) → return(NIL);
                                 setq(dn;choosel(final(sn))) → go(z);
                                 setq(dn;nextconf(addl(n);sn;advance(snl;sn))) → go(z);
                                 T → return(NIL));
                           z     printconf(n;dn);
                                 return(choosel(intersection(snl;smove(list(dn)))))
```

ALPS/Iでの解はnextconf，ある1つの配置confから一手で動きうる配置の集合を求める*smove，配置のパック・アンパックを行う*fn2，駒位置の連続を調べるconnectなどよりなる。各ルーチンの説明は後述する。

さて，箱入娘パズルによる性能試験の項目としては，次のようなものがあると考えられる。

①すべての手続きをその言語で書き表わすことができる。
②手続き構成を支援するツール，関数が揃っている。
③その手続を動かすのに充分な記憶容量を持つ。
④実用上充分な高速性がある。

①を満たすことは明らかなので②，③，④が問題になると思われる。ALPS/Iのもつ機能のうちハッシュ化配列，関数Tryなどは②に対するサポートになっている。中間言語方式progインタプリタ，オブジェクトロード機能は④に対するサポートになっている。しかし記憶装置は64Kワードしかなく（カセットMTはソースファイルに使用），残念ながら81手解の初期配置から解を求めることができな

い。（各配置は1語にパックし，数として記憶するのが一番簡潔かつ高速であるが，数の保持は1024しかできない。配置を（M×B，P1×P2，P3×P4，S）の型でリストにして保持するにしても1配置に4語，さらにS_nにまとめるために1配置につき1語を要するのでd_{81}をおめるために覚えられる約1万2千の配置を保持するのに60k語リストセルが必要である。）

次に各手続を説明する。

① *fn2　1つの配置を表わす数 ⇄ M，B，P1～P4，S，x，y の間の変換を行う。このルーチンは高速化のためにオブジェクトロード機能を用いて8080機械語により作成されているが，Lispの仕様をまげることはされていない。

例：*fn2[0;d_n]：ある配置d_nの内容を各変数に分離する

② nextconf　ハッシュ化配列TAGがS_{n+1}を作るためのマーキング（S_n，S_{n-1}及びその時点までのS_{n+1}の各配置に対して）に用いられている。不要になったTAGの要素はdelete関数により消去される。H－領域は2kエントリーを持つので充分に使用できる。また*smoveをassoccompを指定することにより，帰路における再計算を不要にすることができる。

```
FUNCTION EVALQUOTE HAS BEEN ENTERED, ARGUMENTS...

 DEFINE((
 (NEXTCONF (LAMBDA ( )
 (PROG ( )
      (CSETQ  STACK (CONS SN (QUOTE (NIL))))
      (SETA TAG (CAR SN) 1)
    Y (PRINT N)
      (COND ((NULL SN)(RETURN NIL))
       ((TRY SN(QUOTE (LAMBDA (J)(PROGN (*FN2 O J)(EQ M 13)))))(GO X)))
        (CSETQ VALUE NIL)
      (MAPC SN (QUOTE (LAMBDA (J)(MAPC (*SMOVE  J)
                  (QUOTE (LAMBDA(K)(COND ((NOT (EQ (ERRSET (TAG K) T) 1))
        (PROGN (SETA TAG K 1)(CSETQ VALUE (CONS  K VALUE))))))))))))
      (MAPC SN1 (QUOTE (LAMBDA (J)(DELETE TAG J))))
      (CSETQ N (ADD1 N))
      (CSETQ SN1 SN)
      (CSETQ SN VALUE)
      (CSETQ STACK (CONS SN STACK))
      (PRINT SN)
      (GO Y)
    X (CSETQ STACK (CDDR STACK))(PRINT (QUOTE $$$ YATTAZO $))
      (PRINT N)(PRIN1 (QUOTE M=))(PRIN1 M)(PRIN1 (QUOTE B=))(PRIN1 B)
      (PRIN1 (QUOTE P1=))(PRIN1 P1) (PRIN1 (QUOTE P2=))(PRIN1 P2)
      (PRIN1 (QUOTE P3=))(PRIN1 P3)(PRIN1 (QUOTE P4=))(PRIN1 P4)
      (PRIN1 (QUOTE S=))(PRINT S)
      (CSETQ DN (TRY SN1 (QUOTE (LAMBDA (K)(TRY (*SMOVE K)(QUOTE (LAMBDA(J)
           (PROGN (*FN2 O J)(EQ M 13)))))))  ))
    Z (CSETQ N (SUB1 N))(CSETQ SN SN1)(CSETQ SN1 (POPUP NIL))
      (PRINT N)
      (*FN2 O DN)
      (PRIN1 (QUOTE M=))(PRIN1 M)(PRIN1 (QUOTE B=))(PRIN1 B)
      (PRIN1 (QUOTE P1=))(PRIN1 P1)(PRIN1 (QUOTE P2=))(PRIN1 P2)
      (PRIN1 (QUOTE P3=))(PRIN1 P3)(PRIN1 (QUOTE P4=))(PRIN1 P4)
      (PRIN1 (QUOTE S=))(PRINT S)
      (MAPC SN (QUOTE (LAMBDA (J)(DELETE TAG J))))
      (MAPC SN1 (QUOTE (LAMBDA (J)(SETA TAG J 2))))
      (CSETQ DN (TRY (*SMOVE DN)(QUOTE (LAMBDA (J)(EQ (ERRSET (TAG J) T) 2)))))
      (COND ((NULL DN)(RETURN NIL)))
      (GO Z))))))

END OF EVALQUOTE, VALUE IS..
(NEXTCONF)
```

Nextconf の説明

Yからはじまるループ

- Sn中に娘がまわりにいる配置 (*fn2[0;j]; eq[M;13]) があればXループへ

  (Try[Sn; QUOTE[λ[j]; progn[*fn2 ...] ...)

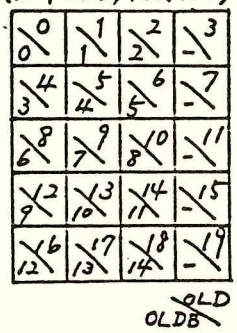
- なければ Snの各要素に*smoveを働かせ，それが Sn, Sn-1にいれば (eq[errset[tag[k]T]1]) はずし，それ以外のものは valueにつなぐ
- Sn-1からTagをはずす ((DELETE TAG J))
- Snを Sn-1にする
- valueを Snにする

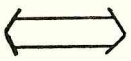
Xからはじまる部分
娘が出口に到達したので帰路の処理にはいる。
- S_{n-1}の各要素に対して *smoveをすると娘が出口にくるものを1つ発見し，それをd_nとする。

Zからはじまる部分
d_n ・・・ d_0 の発見
- S_{n-1}をS_nにし，さらに一つ手前のS_{n-1}を得る
- S_nの各要素のタブをとる
- S_{n-1}の各要素にタブをつける
- d_nから *smove をかけて動ける配置の各々にタブがついているかを探す
- タブがついていれば それをd_{n-1}とする
- これをd_{-1}(=NIL)になるまでくり返す。

③ *smove conf から移りうるすべての配置の集合を値とする関数。M, B, P1～P4についての移動可能配置は現在位置をもとに，ハッシュ化配列を引用し，手続を取り出し実行させる。これは演算ルーチンの速度に比べて配列参照の方が速いこと，このために必要とされるM, B, Pに対する手続のセル数は3K程度で済むことなどによる。子供C1～C4の移動に関しては，すきま(x, y)との位置計算により行う方が速いのでハッシュ化配列は使用しない。

基本位置番号
(左下は0, 他は右上)





Cの位置のための盤面位置番号

0	1	2	3
5	6	7	8
10	11	12	13
15	16	17	18
20	21	22	23

例 OLD(4)=5
OLDB(9)=15

```
FUNCTION EVALQUOTE HAS BEEN ENTERED. ARGUMENTS...
DEFINE((
 (*SMOVE (LAMBDA (CONF)
 (*PROGN
     (CSETQ VAL NIL)
     (*FN2 0 CONF)
     (CSETQ WW ((MM M) NIL))
     (COND (WW (CSETQ VAL (CONS WW VAL))))
     (CSETQ WW ((BB B) NIL))
     (CSETQ VAL (NCONC WW VAL))
          (CSETQ NOWP 1)
     (CSETQ WW ((PP P1) NIL))
     (CSETQ VAL (NCONC WW VAL))
     (CSETQ NOWP 2)
     (CSETQ WW ((PP P2) NIL))
     (CSETQ VAL (NCONC WW VAL))
     (CSETQ NOWP 3)
     (CSETQ  WW ((PP P3) NIL))
     (CSETQ VAL (NCONC WW VAL))
     (CSETQ NOWP 4)
     (CSETQ WW ((PP P4) NIL))
     (CSETQ VAL (NCONC WW VAL))
     (CSETQ NM (OLD M))
     (SETA MARK NM 1)
     (SETA MARK (ADD1 NM) 1)
     (SETA MARK (SUM NM 5) 1)
     (SETA MARK (SUM NM 6) 1)
     (CSETQ NX (OLD X))
     (SETA MARK NX 1)
     (CSETQ NY (OLD Y))
     (SETA MARK NY 1)
     (CSETQ NB (OLDB B))
     (SETA MARK NB 1)
     (SETA MARK (ADD1 NB ) 1)
     (CSETQ OP1 (OLD P1))
     (CSETQ OP2 (OLD P2))
     (CSETQ OP3 (OLD P3))
     (CSETQ OP4 (OLD P4))
     (SETA MARK OP1 1)
     (SETA MARK (SUM OP1 5) 1)
     (SETA MARK OP2 1)
     (SETA MARK (SUM OP2 5) 1)
     (SETA MARK OP3 1)
     (SETA MARK (SUM OP3 5) 1)
     (SETA MARK OP4 1)
     (SETA MARK (SUM OP4 5) 1)
     (CSETQ L 23)
     (CSETQ CLIST NIL)
 FINOC
     (COND ((NOT (OR
        (EQ (MARK L) 1)
        (EQ L 4) (EQ L 9)(EQ L 14)(EQ L 19)      ))
                    (CSETQ CLIST (CONS L CLIST))))
     (SETA MARK L 0)
     (COND       ((ZEROP L)  (GO FINOX)))
     (CSETQ L (SUB1 L))
     (GO FINOC)
```

現在のM,B,Pの位置に対応してハッシュ化配列MM,BB,PPの要素の手続きにより各々WWに移動可能配置をsetしVALへ結合させる。

以下の説明：一語にパックされたconfは手次の位置でなくすきまの位置関数Sを保持しているので，各駒とすきまの位置に目印をつけることにより C1～C4の位置を探し，それをCLISTにまとめる。このために配列MARKが用いられている。MARKには初期値ゼロが実行に先立って与えられている。配列OLD,OLDBはCの位置を求めるための統一化された駒位置の番号への変換を行なう定数テーブルである。

次ページへ続く

```
FINDX
    (CSETQ PREDXY (CONNECT NX NY))
    (MAPC CLIST
      (QUOTE (LAMBDA (C)
        (PROGN
          (CSETQ OC (NEWC C))
          (CSETQ PREDCX (CONNECT C NX))
          (CSETQ PREDCY (CONNECT C NY))
          (COND ((AND PREDCX PREDXY)
                 (CSETQ VAL (CONS (*FN2 6 OC Y)(CONS (*FN2 6 OC X) VAL))))
                ( PREDCX (CSETQ VAL (CONS (*FN2 6 OC Y) VAL)))
                ((AND PREDCY PREDXY)
                 (CSETQ VAL (CONS (*FN2 6 OC X)(CONS (*FN2 6 OC Y) VAL))) )
                (PREDCY (CSETQ VAL (CONS (*FN2 6 OC X) VAL)))
            )))))
    (RETURN VAL)
 )))))

END OF EVALQUOTE. VALUE IS..
(*SMOVE)
```

connect[x;y]はxとyが隣接していればT，そうでなければNILを値とする関数である。 $connect[x;y] = |x-y|=1 \vee |x-y|=5$

④ハッシュ化配列によるM,B,Pの移動可能配置の並べあげと現在位置による分割。実行に先立ち，配列MM, BB, PPの要素をセットしておく。

次に示すのはBBのセットの一部で，たとえばBが盤面の左上にいるとき（B=0：基本位置0と1を占める場合）に(BB B)の評価により呼びだされる値を与えている。もしすきまが下側に2つ並んでいたら(S=71)，そこへ移動させた配置を生成する。もしすきまの一方が右横にある場合(X=2)には一つ右へ移動した配置を生成し，さらに他方のすきまもその右に隣接している場合にはそこまで移動した配置の生成も付け加えられる。

```
FUNCTION EVALQUOTE HAS BEEN ENTERED. ARGUMENTS...:

PROGN (
   (SETA BB 0
        (QUOTE (LAMBDA ( ) (PROGN (CSETQ WK NIL)
         (COND ((EQ S  71) (CSETQ WK (CONS (*FN2 2 3   1) WK)))
               ((EQ X  2) (PROG2
                           (COND ((EQ Y  3) (CSETQ WK (CONS (*FN2 2  2   1
                                                          ) WK))))
                           (CSETQ WK (CONS (*FN2 3  1  0 Y) WK))
                                                              )))
                     ))     ))
        :
        : (以下省略)
```

以上のような箱入娘パズルの解法のためにH-分子212個，リストセル5200個が使用される。

⑤ 実行例

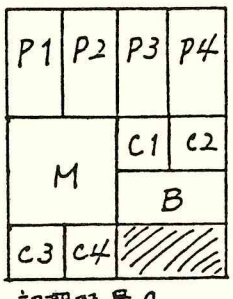
実行にはいる前に $S_n = \{C_0\}$, $S_{n-1} = NIL$, $N = 0$ がセットされる。

```
FUNCTION EVALQUOTE HAS BEEN ENTERED. ARGUMENTS...
ASSOCCOMP((*SMOVE))

END OF EVALQUOTE. VALUE IS..
T

FUNCTION EVALQUOTE HAS BEEN ENTERED. ARGUMENTS...
NEXTCONF NIL
0
(018EB023H 018GBC23H 018GBD23H)
1
(0188B823H 018GB923H 018GBA23H 018E9523H 018E9623H 018E9E23H 018E9023H)
2
(018E4127H 018E3A28H 018E4227H 018E3263H 019E8023H 018E29A3H 018E3163H 018E9223H
 01CB7D23H)
3
(41CB0923H 05CB1A23H 01CB8823H 01CB7E23H 028E17A3H 018E3AA2H 819E0423H 419E0C23H
 019EA623H 019E8423H 018E2E63H 018E2938H 018E3D27H)
4
(028E1736H 018E6736H 018E2D3BH 018E2667H 019E8523H 019E8923H 019EA723H 419E1023H
 809E1723H 018E74A2H 018E3EA2H 128E04A3H 01CB2B63H 01CB7F23H 05CB1C23H 01CB2C63H
 01CB9223H 01CB8923H 45CB0123H 41CB0A23H)
5
(01C83C27H 01C8B023H 01CB2E63H 01CB3D27H 05C81463H 05C81D23H 41C80B23H 01C83B27H
 41CB0263H 018E42A2H 018E9EA2H 018E78A2H 809E2913H 019CBD23H 019C8E23H 01908D23H
 0198A223H 419E1123H 018E313BH 018E953BH 018E6B3BH 128E043BH)
6
(019E803BH 018E9E3AH 4190BE23H 01984627H 0198B823H 0198A123H 01908623H 0190BB23H
 01908E23H 0190B623H 019CB823H 019CB723H 019CB623H 809E3A12H 018E95B2H 41CB0327H
 01CB2667H 01C8B323H 01C8B723H 01C8B823H 01C8B423H 05CB1527H)
7
(01D8A623H 01C8BE23H 019E80B2H 018E29BAH 018E3186H 809E7412H 809E3E12H 019CA023H
 019CA123H 019C9E23H 019CA223H 019EB923H 0190A923H 0190B523H 01908823H 01908C23H
 0190A923H 01908A23H 0190A023H 01908523H 01908723H 0198A023H 0198BE23H 0198B723H
 01984227H 01984527H 41908823H 41908723H 41908C23H 41908D23H 018E3A8AH 018E427AH
 819E043BH 419E0C3BH 019EA63BH 019E843BH)
8
 YATTAZO
8
M=138=8P1=0P2=1P3=2P4=3S=166
7
M=128=8P1=0P2=1P3=2P4=3S=179
6
M=128=8P1=0P2=1P3=2P4=3S=176
5
M=128=11P1=0P2=1P3=2P4=3S=146
4
M=128=11P1=0P2=1P3=2P4=3S=136
3
M=128=11P1=0P2=1P3=2P4=3S=125
2
M=88=11P1=0P2=1P3=2P4=3S=185
1
M=88=11P1=0P2=1P3=2P4=3S=188
0
M=88=11P1=0P2=1P3=2P4=3S=190

END OF EVALQUOTE. VALUE IS..
NIL
```



初期配置 C_0

C_0 の実行時間は印刷時間を含めて約3分20秒である。(内, 往路3分, 帰路は完全に印刷待ちで20秒) 所要メモリはリストセルが 44K, H-分子が 52 エントリーである。*Smove の直想計算を指定しないと帰路が約1分を要し, 4分の所要時間, リストセル 55K, H-分子 334 エントリーになる。直想計算は L-分子の消費を 60～70% におさえている。また, この時覚えられた配置の総数が 133, 所要時間 3分(往路) より 81手解の場合は約 300分程度と推測される。

6.7 研究成果の要約

半識別属性空間の直接的な実現方法について示した。また，そのために必要な処理概念とその設計及び マイクロプロセッサと大容量メモリを中心にした，コンパクトな専用ハードウェアシステムの設計と実現について示した。

データを素情報とその間の関係記述に類別し，それに対応してデータ領域も類別する。さらに，素情報と連想子の表現形式を統一する。次に連想子の導入により属性による検索を高速化することを明らかにした。また，連想子を利用して疎な配列の処理・知識辞書の記憶・不均質データの格納が効率よく行える点，後藤教授の発案になる連想計算機構を連想子処理に組込むことにより，発見的探索処理の記述性をよくし高速化できる点などを実例を用いて示した。

なかでも，従来汎用の大型機でなければ実行することのできなかった大規模な数式処理システムを，その上に機能を落とすことなく実現し，簡単な問題なら解くことができることは本モデルとその実現系の有用性を示している。

第 7 章　結　論

本章では，筆者の研究の総括として，本論文で述べた原理・モデル・その実現方法について要約する。本論文の中核は　属性処理機能を情報技術にもたせるために必要となる，記述子及び連想子の概念とその実現手法にある。下記のうち1～5が主旨である。

1. データには属性がある。属性の値を属性値という。属性は論理属性と記憶属性との2つに類別できる。記憶属性はそのデータの電子計算機上での物理的表現に関するものである。論理属性は記憶属性以外のものである。データを属性・属性値を用いて表現する場合，記憶属性を対象外とすることにより，データに普遍性を与えることができる。論理属性のみからなるデータ空間を半識別属性空間という。

2. 電子計算機における情報処理に対して　データ及びデータ間の関係に着目した統一的分析手法を示した。

事務処理・人工知能などの副分野をもつ非数値演算応用において，対象となるデータ空間Dは　素情報a_iより構成される半識別属性空間として統一的に扱うことができる。

$$D = \{a_1, a_2, \cdots, a_n\} \quad : \text{半識別属性空間}$$

素情報は論理属性P_jと論理属性値v_{ij}との順序対よりなる。

$$a_i = \{(P_1\ v_{i1}), (P_2\ v_{i2}), \cdots, (P_m\ v_{im})\}$$

$$P = \{P_1, P_2, \cdots, P_m\} \quad : \text{論理属性集合}$$

$$V = \{v_{11}, v_{12}, \cdots, v_{21}, v_{22}, \cdots, v_{nm}\} \quad : \text{属性値集合}$$

3. 半識別属性空間を電子計算機上に実現する直接的な方法としてデータのもつ記憶属性値を画一化し，統一することによる方法がある。このとき基本となる要素を連想子とよぶ。連想子は，

（素情報，属性，属性値）

の対である。

4. 記憶属性を画一化できない場合，記憶属性を分離して保持する設計概念がよい。各データに対応して分離して保持された記憶属性値を記述子と呼ぶ。

5. 論理属性及び記憶属性を分離したデータ事象を抽象事象とよぶ。抽象事象が構成する空間を抽象データ空間とよぶ。抽象データ空間を対象とする操作概念を抽象命令とよび，抽象命令により構成される手続を抽象手続という。

抽象手続を電子計算機上に実現することにより人間の思考に近いレベルで電子計算機上での情報処理を記述することができる。抽象手続は論理属性及び記憶属性から独立しているので，処理概念を使用する電子計算機の構造，使用するデータの物理的性質，これによって構成しようとする応用システムの特殊性から独立して扱うことができる。

6. データを処理する手続概念には定型反復処理と動的処理がある。この手続概念は対象となるデータの構成の仕方により特徴づけることができる。

定型反復処理におけるデータファイルはそのデータ空間を構成する属性値集合のみからなることが多い。

動的処理におけるデータファイルはその論理的なデータ空間をそのまま電子計算機の内部表現に写像させたものが多い。

7. 情報処理のためのデータ空間は その記憶密度の点から

均質データ空間と不均質データ空間

に分類することができる。

均質データ空間とは その空でない要素の個数 l が

$$l > \frac{nm}{2}$$, m は属性数, n は素情報数

を満たすものを言う。

均質データ空間以外の空間を不均質データ空間という。

8. 定型反復処理のためのデータ空間は均質と考えてよい。
動的処理のためのデータ空間は不均質と考えてよい。

9. 処理上の必要性・ソフトウェアの保守・実行時における妥当性のチェックなどの点から, 処理手続の作成・実行及び対象となるデータ群の定義・記憶形態を通して, 一貫した属性処理機構が必要となる。

記述子は均質データ空間のための処理機構の基本要素となりうる。連想子は不均質データ空間のための処理機構の基本要素となりうる。

10. 定型反復処理に使用される記述子のために, 定型的手続の記述手法とその実行時における環境との間の関係について考察し, 抽象手続概念を導入して 原形手続記述言語を設計した。

合わせて原形手続記述に対する, 論理属性, 記憶属性の確定概念, 及びその手順, 抽象手続の具象化のための生成制御方式について設計した。

11. 動的処理に使用される連想子のために，連想子の表現形式・ハッシングを利用した高速参照機構及び それを中心としたデータ空間の包括管理機構を設計した。

12. EDP合理化のために記述子を中心としたソフトウェアシステムを設計開発し，その実用性を確認した。このEDPシステムに利用された概念を特に三層分化プログラム構成法と呼ぶ。三層分化プログラム構成法は 本論文に示された，データ空間の処理機構とそのための手続記述法・プログラム生成法を中心としたもので，これは企業内の組織的な機能分担と対応させることができる。

13. (P_k, a_i, v_{ik}) を直接計算機内部で表現する連想子機構を中心に データ空間の包括管理とデータの動的処理を可能とするソフトウェアシステムを，記号処理言語Lispを拡張して設計開発した。そのシステムの上で作られた人工知能応用のいくつかの実施例を示し，連想子機構の有用性を確認した。

14. 13に示すデータ管理概念を効率よく実現することができるハードウェアシステムについて考案し，マイクロコンピュータと大容量ICメモリを用いて，価格/性能比の良い会話型計算機を設計・製作した。この計算機をALPS/Ⅰとよぶ。

ソフトウェアの特性に基づき，AAMとよぶ特殊インタフェイスを考案し，階層型のメモリ構造におけるデータ処理の速度を高めることに成功した。この方式は16ビットマイクロプロセッサなどを利用する場合にも可能である。ALPS/Ⅰでは，本論文で述べた機構とソフトウェアシステムを用いて数式処理言語Reduceが動作している。Reduceは従来，大型計算機上でしか動作することができなかった大規模システムである。このシステムを廉価な超小型機の上で初めて動作させたことは連想子機構とそれを中心とする体系が有用であることを示している。

15. 本論文で示した理論モデルにより，2つの応用分野の共通性をデータの属性処理の観点からまとめることができた。しかし，より一層の発展をするために次のような点を将来の課題としたい。

(1) 5.4.3項に述べたように，FAST1システムは処理の類型化と抽象化，そしてデータ定義に関して有効であったが，細部の記述方法については特別な考慮を払っていなかった。そこで構造化プログラミング的な記述支援を行うか，あるいは関数的な記述方式の導入を行うことが必要となる。

(2) ハードウェアの進歩が激しいので，人工知能研究用計算機については言語の構造と計算機の構造について今後もなお，研究をすすめる必要がある。

(3) ここでまとめた属性空間理論はまだ基礎的なものであって，その一部分だけが実用と結びついている。さらに一般的な形で展開するよう試みたい。

謝　　辞

本論文の作成にあたり，平素より終始ご指導を仰いだ青山学院大学理工学部経営工学科教授間野浩太郎氏，本論文の査読を通して貴重なご助言をいただいた同教授古谷野英一氏，青山学院大学理工学部数学教室教授古屋茂氏，東京大学理学部情報科学科教授後藤英一氏に感謝いたします。

この研究活動を進めるうえで，ご助言をいただいた情報処理学会　記号処理研究会及び　ソフトウェア工学研究委員会の諸先生ならびに諸兄，ご助言・ご協力をいただいた青山学院大学理工学部経営工学科講師矢頭攸介氏，同実験講師横山賢子氏などの諸先生及び森芳喜君，小林茂男君，重光恭之君，木村公則君，森剛一君などの諸兄；ならびに富士銀行業務管理部森建二氏に　深謝いたします。

参　考　文　献

- 著者名アルファベット順・年代順に並べられている
- 表記及び引用については　近年の情報処理関連分野における慣行に従い，著者名の頭3字及び発表年2ケタにより構成する。同名・同年の場合には識別のためにA，B，・・・の順に一字付加する。

例：　[DIJ70]

Dijkstra, E. W. : Structured Programming, Academic Press, 1970

[BAK72] Baker, F.: System Quality through structured Programming; Proc. of FJCC, (Dec. 1972)

[BAK78] Baker, H.G.: List Processing in Real time on a serial computer; C. ACM Vol.21, No.4, pp280-294 (April 1978)

[BOE74] Boehm, B.: The high cost of Software; Software World Vol.6 No.1 pp2~10, (Jan. 1974)

[BOE76] Boehm, B.: Software Engineering; IEEE on Computer, (Dec. 1976)

[BOY75] Boyer, R.S. et al: Select-A Formal System for Testing and Debugging Programs; Proc. ICRS, pp234-245 (April 1975)

[BUL78] Bullman, D.: Introduction to stack computers; IEEE on Computer, (Nov. 1978)
(邦訳は bit 誌 vol.11, No.5~7, 井田訳 (1979))

[CHA70] C.L.CHANG: The Unit proof and the Input proof in Theorem Proving; J.ACM Vol.17 No.4 pp698-707 (Oct.1970)

[CHA73] — and R.C.T. LEE: Symbolic logic and Mechanical Theorem Proving; Academic Press (1973)

[COD62] CODASYL L.S.G.: Information Algebra; C.ACM Vol.5 No.4 (1962)

[COD 70] Codd, E.F. : A Relational Model of Data for Large shared Data Banks; C.ACM Vol.13 No.6 pp377~387 (June 1970)

[COR 69] Corbató, F.J. : PL/I as a tool for system Programming; Datamation, pp68, 73~76 (May 1969)

[DAH 70] Dahl, O.-J. : The Simula 67 Common Base Language; Publication S-22, Norwegian Computing Center Oslo, 1970

[DAT 77] Date, C.J. : An Introduction to Data Base Systems 2nd edition; addison-wesley (1977)

[DIJ 70] Dijkstra, E.W. : Structured Programming, Academic Press, (1970)

[DIJ 75] Dijkstra, E.W. et.al. : On-the-fly Garbage Collection : An exercise in cooperation, EWD520-0, Technische Hogeschool Eindhoven, Netherlands, (Oct, 1975) もしくは G.Goos & J.Hartman edrs. "Lecture Note in Computer Science 46" pp43~56

[ETL 76] ETL : Lisp 1.9 User's Mannual, EPICS, 5-ON-2, (1976)

[FEL 69] Feldman, J.A. and Rovner, P.D : An Algol-Based Associative Language; C.ACM Vol.12 No.8 (Aug. 1969)

[FUJ 75] 富士銀行：FAST1解説書(1975)

[GOT74A] 後藤英一：Monocopy and Associative Algorithms in an Extended LISP，東京大学テクニカルレポート(1974)

[GOT74B] ―：連載Lisp入門，bit共立出版，Vol.6，No.1～No.13 (1974)

[GRE74] Greenblatt, R.：The Lisp Machine, MIT AI Lab. Working Paper 79 (1974)

[HAN69] Hansen, W.J.：Compact List Representation: Definition, Garbage Collection and system Implementation, CACM Vol.12 No.9 pp499～507 (Sept. 1969)

[HAS76] Haseman et.al.：Design of Multi-dimensional Accounting System; Accounting Review, pp65～79 (Jan. 1976)

[HEA69] Hearn, A.C.：Standard Lisp; Stanford Artificial Intelligence Report, Memo AI-90 (May 1969)

[HEA73] ―：REDUCE2 User's Manual; UCP-19 (March 1973)

[HOA72A] Hoare, C.A.R.：Proof of Correctness of Data Representations; Acta Informatica, Vol.1 pp271-281 (1972)

[HOA72B] ―：データ構造化序論；in [DIJ 72] (1972)

[HOP 72] Hopgood, F. and Davenport, J.: The Quadratic hash method when the table size is a power of 2; Computer Journal, Vol.15 No.4 pp 314～315, (1972)

[HSI 70] Hsiao, D. and Harary, F.: A Formal System for Information Retrieval from Files; C.ACM Vol.13 No.2 pp67～73 (Feb. 1970)

[IBM 71] IBM: Chief Programmer Teams: Principles and Procedures; FSD Report No. FSC 71-5108 (1971)

[IDA 75] 井田昌之, 木村公則: ソフトウェアマネージメントシステム FAST1; 情報処理学会 構造化プログラミングシンポジウム報告集 pp122-128 (1975)

[IDA 76] ―, et.al.: ALPS/Iのバルクメモリアクセス機構とLispインタプリタ; 情報処理学会大会予稿集 pp713-714, (1976)

[IDA 77A] ―: Lispマシン ALPS/I; 情報処理学会 記号処理研究委員会51年度報告集, pp142-159 (March 1977)

[IDA 77B] ―: ALPS/Iの性能評価; 情報処理学会 記号処理研究委員会 77-2-(1) (1977)

[IDA 78] ―: 事務処理プログラム作成の体系化とFAST1; 経営工学会誌 Vol.28 No.4 pp417-422 (1978)

[IDA 79A] ―, 間野浩太郎: マイクロプロセッサを用いたLispマシンALPS/I; 情報処理, Vol.20 No.2 pp113-121 (1979)

[IDA79B] 井田昌之，中田育男：基本ソフトウェアの記述ツール，情報処理 Vol.20 No.6 pp519-526 (June 1979)

[IDA79C] Masayuki Ida and kotaro Mano : An Adaptable Lisp Machine based on Micro Processors, proc. of IMMCC79, IEEE, pp210-215 (Nov. 1979)

[IDA79D] 井田昌之：コンパクトな処理系が可能な連想情報モデルについて；経営工学会誌，Vol. 30, No. 3 pp224-230 (Dec. 1979)

[INT72] INTEL Corp.: Intellec 8 MOD80 Reference Manual (1972)

[INT78] INTEL Corp.: MCS-86 Assembly Language Reference Manual (1978)

[IPS74] 情報処理学会：記号処理シンポジウム報告集 (1974)

[IPS75] ー：ソフトウェアエンジニアリング特集号；情報処理 Vol.16 No.10 (1975)

[KOB77A] 小林茂男，小方孝修：Simulationによる ALPS/I の hashing Algorithm の分析；情報処理学会 記号処理研究委員会報告集 pp166-176 (Mar. 1977)

[KOB77B] 小林茂男：ALPS-Reduce のインプリメンテーション；青山学院大学修士論文 (1977)

[KNU68] Knuth, D.E.: The Art of Computer Programming Vol.1; Addison-Wesley (1968)

[KUR76] 黒川利明：Lispのデータ表現；情報処理 Vol.17 No.2 pp127-132 (Feb. 1976)

[LEF69] Lefkovitz, D.: File Structures For On-line Systems; Spartan books, (1969)

[LIE75] Lieberman et.al.: A Structuring of an Events Accounting Information System; Accounting Review, PP246-258 (1975)

[LIS74] Liskov, B. and S. Zilles: Programming with Abstruct Data Types; proc. of ACM SIGPLAN conference on Very High Level Languages, SIGPLAN Notice Vol.9 No.4 PP50-59 (April. 1974)

[LIS76] Liskov, B.: An Introduction to CLU; proc. of New Direction in Algorithmic Languages 1975, PP139-156, IRIA, Paris (1976)

[MAN76] 間野浩太郎，井田昌之：マイクロコンピュータを用いたLispマシンALPS/I，情報処理学会大会予稿集 PP711-712 (1976)

[MAR75] Martin, J.: Computer Database Organization 2nd edition; Prentice Hall (1977)

[MCC 60] McCarthy, J.: Recursive Functions of Symbolic Expressions and Their Computation by Machine, CACM, Vol. 3 No. 4 pp184-195 (1960)

[MCC 66] McCarthy, J. et. al.: Lisp 1.5 Programmer's Manual, MIT Press (1966)

[MOR 68] Morris, R.: Scatter Strage Techniques, CACM Vol. 11 No. 1 pp 38-44 (Jan. 1968)

[NAG 76] 長尾真他：高速補助記憶を使用したミニコン用Lisp 1.6 システム；情報処理, Vol. 17 No. 8 pp 720-728 (1976)

[NIH 72] 日本経済新聞社編：経済分析のためのデータ解説；日本経済新聞社 (oct. 1972)

[ORG 73] Organick, E. I.: Computer System Organization; academic press (1973)

[PFA 77] Pfaltz: Computer-Data Structure; McGrawhill (1977)

[QUA 70] Quam, L. H. and Diffie, W.: Stanford Lisp 1.6 Manual; SAILON 28.4 (1970)

[ROB 77] Robinson, L. and Roubine, O.: SPECIAL — a specification and Assertion Language, Tech. Report CSL-46, SRI (Jan. 1977)

[SAG77] Sagalowicz, D.: IDA: An Intelligent Data Access Program; proc. of 3rd VLDB, pp293-302, (Oct. 1977)

[SAM69] Sammet, J.E.: Programming Languages, History and Fundamentals, Prentice-hall (1969)

[SCH67] Schorr, H. and Waite, W.: An Efficient Machine-Independent Procedure for garbage Collection in Variars List Structures, CACM Vol.10 No.8 PP501-506 (Aug. 1967)

[SEV74] Severance, D.G.: Identifier Search Mechanisms; A survey and Generalized Model; Computing Survey Vol.6 No.3 (Sept. 1974)

[STE75] Steele, G.L.: Multiprocessing compactifying garbage collection, C.ACM Vol.18 No.9 PP495-508 (Sept. 1975)

[SIG76] 重光宏之, 小林茂男: ALPS/I 固定プログラム開発のためのサポートシステム; 情報処理学会大会予稿集 PP715-716 (1976)

[TAK78] I. Takeuchi: The report of a Lisp Contest, preprint of WGSYM 5-3, IPSJ (Aug. 1978)

[TEI72] Teitleman, W. and Bobrow, P.G. et. al.: BBN Lisp Reference Manual; BBN (1972)

[TOO78] 遠峰隆好：REDUCE 2（新版）のALPS/Iへの Implementation；青山学院大学卒業論文（1978）

[WAD76] Wadler, P.L.：Analysis of an algoritihm for real time garbage collection; C.ACM Vol.19, No.9 pp491-500 (Sept. 1976)

[WIR71] Wirth, N.: The Programming Language PASCAL; Acta Informatica, Vol.1 pp35-63 (1971)

[WIR76] Wirth, N.: Algorithms + Data Structures = Programs; Prentice-Hall (1976)

[WUL76] Wulf, W.A. et.al.: An Introduction to the Construction and Verification of Alphard Programs; IEEE Trans. on S.E. Vol.2 pp253-244 (1976)

[YAM75] 山本純茶：情報検索；in「情報処理のための数学」赤，後藤他編，p164-179（1975）