資料1

第5世代のコンピュータ

ロジック・プログラミングの調査研究

(FGKL/Sマシンの基本構想)

昭和57年2月

財団法人 日本情報処理開発協会
社団法人 日本電子工業振興協会

この報告書は、日本自転車振興会から競輪収益の一部である機械工業振興資金の補助を受けて昭和56年度に実施した「第５世代電子計算機に関する内外技術動向調査」の成果をとりまとめたものであります。

序　　文

この報告書は，財団法人日本情報処理開発協会からの委託により，社団法人日本電子工業振興協会が昭和５６年度事業として実施した「第５世代コンピュータ基幹技術に関する調査研究」のうちの「ロジック・プログラミングの調査研究」の結果をとりまとめたものである。

この調査は，１９９０年代に実用化を目標とした第５世代コンピュータの重要技術であるロジック・プログラミングについて，各パイロット・マシンの基本設計に必要な基礎データの収集分析ならびに開発のための基本仕様の作成を「技術調査委員会」（委員長・電子技術総合研究所・横井俊夫氏）を設置して，実施した。

調査の実施にあたり，ご指導，ご協力いただいた関係官庁，関係各位ならびに直接労を賜わった委員各位に深く感謝する次第である。

昭和５７年２月

＜本調査で作成した報告書・資料＞

- 第５世代のコンピュータ研究開発計画
- 〃　　〃　　データフローマシン／データベースマシン
- 〃　　〃　　ロジックプログラミング
- 〃　　〃　　波及効果
- 〃　　〃　　関連技術動向調査
- 〃　　〃　　研究開発計画・付属資料

ロジック・プログラミング技術調査委員会
（敬称略・順不同）

委員長	横井俊夫	電子技術総合研究所	パターン情報部 数理基礎研究室室長
委員	佐藤泰介	〃	パターン情報部 推論機構研究室技官
	元吉文男	〃	パターン情報部 推論機構研究室技官
	梅村護	日本電気㈱	C&Cシステム研究所 コンピュータ・システム研究部
	国藤進	富士通㈱	国際情報社会科学研究所
	黒川利明	東京芝浦電気㈱	総合研究所情報システム研究所 ソフトウェアグループ
	後藤滋樹	電電公社 横須賀電気通信研究所	データ通信方式研究室研究専門調査員
	白石富勝	シャープ㈱	産業機器事業本部 システム機器事業部第1技術部
	中島秀之	東京大学	工学部計数工学科　和田研究室
	林弘	㈱富士通研究所	情報処理研究部　第1研究室室長
	上田謙一	松下技研㈱	研究開発部門技師
事務局	菊池英	㈳日本電子工業振興協会	開発部部長代理
	清紹英	〃	電子計算機担当課長代理
	小島謙二	〃	電子計算機担当
協力者	梅山伸二	電子技術総合研究所	制御部　論理システム研究室技官
	新田克己	〃	ソフトフェア部 情報システム研究室技官
	大谷木重夫	〃	制御部 論理システム研究室主任研究官
	松田利夫	東京理科大学理工学部	情報科学科助手

第５世代のコンピュータ
ロジック・プログラミングの調査研究　目　次

第1部　研究・調査活動の概要

1. 研究・調査活動の概要

1.1 基本方針

コンピュータ・システムの背景となるのがプログラム言語である。このプログラム言語を何に定めるかによって，システムの基本的な姿が定まる。言語は，それによって記述される様々なプログラムの特徴を反映し，それを処理する計算機のアーキテクチャを方向づける。第5世代コンピュータは，という設問も，第5世代プログラム言語が新しいものとして設定できるのか，できるとしたらその具体的な内容は，を問うことと見なすことができる。過去2年間の調査研究においても，最重要課題として検討され，第5世代プログラム言語，とりあえずＦＧＫＬ（５Ｇ－核言語）と呼ぶが，次のように設定された。

"論理型言語を母体にし，今まで提案されている様々の有用な記述要素をまとめ上げ，さらに，知識情報処理システムのため，高度な記述機能を加味したもの"

この設定に対し，いくつかの疑問（質問）が投げかけられた。

"十分な記述能力を提供できるのか"

"十分な実行速度を保証できるのか"

"プログラムし易いといえるのか"

"有用な記述要素を十分吸収できるのか"

"将来の大きな展開に十分対応できるのか"

もちろん，これらの疑問に，今，完全に答えることはできない。それ自体が，このプロジェクトの前期における重要な研究テーマである。しかし，この設定の方向に，誤りがなく，方向付けとしては正しいという大方の同意を得ることは必要である。この同意を得るに必要なデータを提供するのが，この技術調査委託の役割である。

この役割に答えるべく，次のような方針で作業を進めた。

① 言語仕様というより，高機能パーソナル・コンピュータ・システムのイメージという形で提案する。

ソフトウェア開発用ツールとしての逐次型推論マシン（核言語向き高機能パーソナル・コンピュータ）は，先進的な研究・開発テーマを数多く含む本プロジェクトの推進には不可欠のものである。そこで，マシンを含めた全体像として描いた方が，その位置付けが明確になる。また，ロジック・プログラミングの現状は，言語の外面的な仕様を切離して議論できる段階ではなく，その処理系のインプリメンテーションの手法も合わせて議論しなければ，その具体的なイメージを与えることはできない。さらに，これからの言語は，支援システムやオペレーティング・システ

ム等を含めたプログラミング・システムとしてとらえねばならないが，そのようなとらえ方は，まだ広く行き渡っているわけではないので，この見方を強調する意味で，システム全体像として描く必要がある。

基本的な言語仕様は，ここで述べる逐次型マシンでも，第五世代の目標機である並列型（駆動型）マシンでも同じであると考えられる，その意をくみ，報告書の副題としては，'ＦＧＫＬ／Ｓマシンと銘うった。Ｓは逐次型の意味で，"'"は，引用（ｑｕｏｔｅ）の印で，まだ評価（ｅｖａｌ）を受けていないことを表わす。

② プログラム言語としてのまとまりに重点を置くが，論理系としての整合性も念頭に置いて作業を進める。

プログラム言語としてまとめ，効率の良さを追求していくのか，論理系としての体系を重んずるのか，２つのアプローチがある。本来は，相反するものではないのであろうが，具体的な作業を進める場では，対立する２つの見方となる。ここでは，論理系として体系立ったものとするのは，将来の研究テーマとし，重点をプログラム言語としてまとめることに置いた。実用化されている代表的な記号処理(知識情報処理の基本操作)言語であるＬｉｓｐの諸機能，抽象データ型によるモジュラー化機能，並列処理機能や入出力機能の整理等，まだ未消化であるが，極力，ロジック・プログラミングの枠内にとり込み，記述能力の一般性と高さを示すのに努力した。

③ 基本部分の合意にのみ議論を集中した。

時間や作業形態の制約から，同意のとれぬもの，断定するだけのデータのないものについては，可能性を併記した。また，高機能部分については，検討をしつくしたというより，１つの考え方を示したものである。ただし，１つとはいえ，担当者が，時間の許す限りの考察を加えた結果である。

④ あくまで，１つのイメージ，１つの近似である。

委員として，それぞれの分野ですぐれた人達に集まっていただいた。しかし，ロジック・プログラミングの細部にまで精通した人は少数であった。新しい分野であるから，当然である。まずは，実際にプログラムを書き，勉強をするということからスタートした。また，各委員，忙しい中を，月々１～２度集まって議論するという作業形態で進められた。このような制約の上で，ここで対象とした種々システムの細部にわたって仕様を作り上げることは，不可能である。したがって，ここでの提案も，未消化で，整合性のとれぬままである。しかし，委員会の役割をはたすに十分な状態とはなっているはずである。

当委員会も，委員各員，協力者と参加していただいた人達，それぞれの職場で作業や議論に加わっていただいた人達，多くの人々の高い知的好奇心と情熱に支えられた。この熱気を素直にス

タートへと持続させることが，本プロジェクト成功への第一歩である。

なお，富士通・国際研の安達統衛，竹島卓，沢村一，加藤昭彦および松下電器の安川秀樹の諸氏への協力を得たことを付記しておく。

1.2 活動の経緯

1.2.1 活動の概略

会合回数	日時	場所	主たる議題
LPG(1)	7／22(水)	ETL	第5体制，LPG基本理念と作業案の紹介
LPG(2)	8／24(月)	ETL	Prolog 調査報告，例題記述成果の突き合わせと整理
LPG(3)	9／29(火)	ETL	Prolog 言語仕様の改良・拡張への提案
LPG(4)	10／30(金)	ETL	仕様案作成のための基本的問題点の討議
LPG(5)	11／18(水)	電子協	FGKL Machine に対する要望の提案
LPG(6)	11／30(月)	電子協	報告書全体構成に関する討論，章別概要の提案
LPG(7)	12／17(木)	電子協	報告書の目次と担当者の決定
LPG(8)	57年 1／11(月)	電子協	報告書作成の基本事項の調整
LPG(9)	1／28(木)	電子協	報告書原案の検討，調整

1.2.2 資料一覧

（LPG－1－1）：第5世代コンピュータ研究開発調査実施体制；

（LPG－1－2）：第5世代コンピュータ・プロジェクト技術調査委託（案）；

（LPG－1－3）（社）日本電子工業振興協会：ロジック・プログラミング技術調査委託作業詳細案；

（LPG－1－4） J.F.Sowa：A Prolog to Prolog.IBM System Res.Inst.1981

（LPG－1－5） 坂村・元吉：スーパー・パソコン比較一覧

（LPG－2－1） L.M.Pereira, F.C.N.Pereira & D.H.D.Warren: User's Guide to DEC system-10 PROLOG, Sept. 1978, pp.1－50,

L.Byrd, F.Pereira & D.Warren：A Guide to Version of DEC－10 PROLOG, June 1980, pp. 1－17.

（LPG－2－2） D.H.D. Warren： Implementing PROLOG－

Compiling predicate logic programs, Vol.1, Dep. of A.I., Uuiv of Edinburgh. D.A.I. Res. Rep. No.39, pp.1−102

(LPG−2−3) D.H.D. Warren: Implementing PROLOG−Compiling predicate logic progams, Vol.2, ibid.No.40, pp.1−61

(LPG−2−4) D.W. Clark: The Dorado: A High-Performance Personal Computer Three Papers, PALO ALTO Res, Center, Xerox, CSL−81−1, Jan 1981 pp.1−80.

(LPG−2−5) (社)日本電子工業振興協会：第5世代ロジック・プログラミング技術調査委員会(第1回)議事録, Aug.1981, 4p.

(LPG−2−6) 後藤：述語型プログラミング言語DURALとその処理系, 電電公社電気通信研究所, 成果報告第16892号, May 1981, 48p.

(LPG−2−7) H.Nakaskima：PROLOG／KR User's Manual for Version C−2 Uuiv.of Tokyo, Aug 1981, 33p.

(LPG−2−8) 中島, 後藤：現在日本で使用可能なProlog システム LPG配布資料, Aug.1981, 2p.

(LPG−2−9) 奥乃：各種プログラミング言語による「レイアウト」プログラムの比較 Aug.1981, 8p.

(LPG−2−10) 古川：問題解決・推論メカニズム ー ハードウェア化によって期待される性能の明確化, SG配布資料, Aug.1981, 3p.

(LPG−2−11) 古川：SG1宿題検討, SG配布資料, Aug.1981, 3p.

(LPG−2−12) 古川：現PROLOGの問題点, LPG配布資料, Aug.1981, 2p.

(LPG−2−13) 上田：8−Queen PROLOG Program, LPG Report, Aug.1981, 3p.

(LPG−2−14) 梅村：宣教師と食人種のPROLOG Program, LPG Report Aug.1981, 5p.

(LPG−2−15) 林：The Missionary and Cannibal PROLOG Program, LPG Report, Aug.1981, 4p.

(LPG−2−16) 国藤：8−puzzle の PROLOGによる解法, LPG Report,

Aug.1981, 4p.

(LPG-2-17)後藤：Graphsearch PROLOG Program, LPG Report, Aug.1981, 7p.

(LPG-2-18)黒川：Problem-reduction Graph の PROLOG Program, LPG Report, Aug.1981, 9p

(LPG-2-19)梅山：命題論理式の妥当性を調べるプログラム，LPG Report, Aug.1981, 6p.

(LPG-2-20)新田：PROLOG作譜例～The Resolution Method～, LPG Report, Aug.1981, 20p.

(LPG-3-1) (社)日本電子工業振興協会：第5世代ロジック・プログラミング技術調査委員会(第3回)議事録，Sept.1981, 4p.

(LPG-3-2) T.Yokoi, et.al.: Logic Programming and a Dedicated Personal Computer, 第5世代国際会議下書き，Oct.1981, 19p.

(LPG-3-3) 後藤：仕様の改良と拡張，分担(a)高階述語の整理と制御構造の導入について，Sept.1981, 2p.

(LPG-3-4) 佐藤：変数の依存解析と後戻り，Sept.1981, 4p.

(LPG-3-5) 梅山：Logic Programming と Intelligent Backtracking, Sept.1981, 7p.

(LPG-3-6) 国藤：PROLOG機能拡張に対する一提言，Sept.1981, 30p.

(LPG-3-7) 国藤，若木：知識ベースと統合性制約 - その演繹的質問応答への利用法，電子通信学会 信学技報AL79-72, Dec.1979, pp.27-34.

(LPG-3-8) 新田：PROLOGにおけるdebug機能，Sept.1981, 14p.

(LPG-3-9) 黒川：PROLOGに対するデータ抽象化とデータベース機能，Sept.1981, 6p.

(LPG-3-10)元吉：PROLOGの拡張 - Higher-Order Extensions to Prolog - Are they needed?, Sept.1981, 3p.

(LPG-3-11)林：LISPマシンの画面制御について，Sept.1981, 11p.

(LPG-3-12)林：高解像度ディスプレイ端末について，Sept.1981, 2p.

(LPG-3-13)白石：Color Graphics Display 仕様案，Sept.1981, 5p.

(LPG-3-14)上田：PROLOG - Editor の仕様について，Sept.1981, 2p.

（LPG－3－15） 新田：エジンバラ版PROLOG処理系，Sept.1981，6p.

（LPG－3－16） 新田：PROLOGプログラム例 (I)ロジック・シミュレータ，(II)ルービック・キューブ，(III)数式処理，Sept.1981，38p.

（LPG－3－17） 国藤：8－puzzle の PROLOG プログラム，Sept.1981，21＋3p.

（LPG－3－18） 安達：Combinatory Logicの式を評価するPROLOG言語プログラム，Sept.1981，30p.

（LPG－3－19） Symbolics，Inc：Symbolics Software，1981，37.

（LPG－3－20） 後藤：ソフトウェア技術委よりLPGへの要望事項，Sept.1981，1p.

（LPG 3.5－1） D.H.D.Warren: Higher-order Extensions to Prolog-Are they needed? D.A.I. Res.Paper, No.154, 1981, 13p.

（LPG 3.5－2） A.Bundy and B.Welham: Utility Procedures in Prolog. D.A.I. Ocasional Paper, No.9, Sept.1977, 15p.

（LPG 3.5－3） L.Byrd: Understanding the Control Flow of Prolog Programs D.A.I. Res. Paper No.151, 1980, 13p.

（LPG 3.5－4） A.Bundy: My Experiences with PROLOG, D.A.I. Working Paper, No.12, March 1976, 8p.

（LPG 3.5－5） D.H.D.Warren: Logic Programming and Compiler Writing, D.A.I. Res. Paper, No. 128, 1980, 53p.

（LPG 3.5－6） D.H.D.Warren: PROLOG on the DEC－system 10, D.A.I. Res. Paper, No.127, 1979, 11p.

（LPG 3.5－7） D.H.D.Warren: An Improved PROLOG Implementation Which Optimises Tail Recursion, D.A.I. Res. Paper No.141, 1980, 13p.

（LPG 3.5－8） D.J.Wright: PROLOG as a Relationally

Complete Database Query Language Which can Handle Least Fixed Point Operators, Dep.of Comp. Sci., Univ. of Kentucky, June 1980, 18p.

（LPG－4－1） （社）日本電子工業振興協会： 第5世代ロジック・プログラミング技術調査委員会（第4回）議事録, Sept.1981, 4p.

（LPG－4－2） W.F. Clocksin and C.S.Mellish: The RT－11 Prolog System (Version NU7 with Top Level) －Software Report 5a (Second Edition), Dec. 1980, 17p.

（LPG－4－3） 国藤, 安達, 沢村, 竹島：＂PROLOG機能拡張に対する － 提言＂の要約, Oct.30, 1981, 5p.

（LPG－4－4） 黒川：PRŌLŌGの仕様改良のキーポイント, Oct.30, 1981, 1p.

（LPG－4－5） 新田：エジンバラ版PROLOGの処理系(2), Oct.30, 1981, 5p.

（LPG－4－6） 中島：TMSとIntelligent Backtracking, Oct.30, 1981, 3p.

（LPG－4－7） 佐藤：失敗の原因追求と後戻り, Oct.30, 1981, 13p.

（LPG－4－8） 梅村：プロログ用エディタ仕様方針案, Oct.30, 1981, 1p.

（LPG－4－9） 白石：Color Graphics Display 基本述語と全体構成(2), Oct.30, 1981, 3p.

（LPG－4－10） 新田：Cut についてのコメント, Oct.30, 1981, 1p.

（LPG－4－11） 梅山：Logic Programming MEMO, Oct.30, 1981, 1p.

（LPG－4－12） 後藤：IF－THEN－ELSE, NOTについての補足, Oct.30, 1981, 2p.

（LPG－4－13） 横井：論理型プログラミングのパラダイム, Oct.30, 1981, 9p.

（LPG－5－1） （社）日本電子工業振興協会：第5世代ロジック・プログラミング技術調査委員会（第5回）議事録, Oct.30, 1981, 5p.

（LPG－5－2） 横井：ロジック・プログラミング技術調査委託報告書の骨子, Nov. 5, 1981, 4p.

（LPG－5－3） 横井：コメントのまとめ，Nov.18, 1981, 3p.
（LPG－5－4） 林：言語・マシン仕様についてのコメント，Nov.18, 1981, 1p.
（LPG－5－5） 国藤，安達，竹島，加藤，松尾：Super Personal FGKL Machine に対する要望事項（案），Nov.18, 1981, 2p.
（LPG－5－6） 後藤：言語・マシン仕様について，Nov.18, 1981, 2p.
（LPG－5－7） 黒川：Prolog マシンへの要求項目，Nov.18, 1981, 1p.
（LPG－5－8） 新田：PROLOGとグラフィックス，Nov.18, 1981, 5p.
（LPG－5－9） 横井：マシン仕様への提案，Nov.18, 1981, 10p.
（LPG－5－10） 佐藤：IBと単一化について，Nov.18, 1981, 19p.
（LPG－5－11） 黒川：From PROLOG to KUROLOG: Logic, Knowledge, AI, and Tool, Nov., 1981, 8p.
（LPG－5－12） 黒川：KUROLOGソース・リスト，Nov., 1981, 3p.
（LPG－5－13） 中島：Portable Prolog Interpreter, Nov.9, 1981, 5p.
（LPG－5－14） 横井：レジメの書式，Nov.18, 1981, 1p.
（LPG－5－15） Boley, H.: AI languages and AI Machines: An Overview, Univ. Hamburg, FB Inform., IFI－HH－M－86/81, Apr. 1981, 22p.
（LPG－5－16） Boley, H.: Bibliography on AI Machines, Univ. Hamburg, FB Informatik, 1980(?), 18p.
（LPG－5－17） Symbolics,Inc.:Symbolics Software, 1981, 37p. {LPG3－19}
（LPG－5－18） Moon, D.A. & Wechsler, A.C. : Operating the Lisp Machine, Symbolics, Inc., 45p.
（LPG－5－19） Weinreb, D.& Moon, D.A.: Introduction to Using the Window System, Symbolics, Inc., 87p.
（LPG－5－20） Weinreb, D.& Moon, D.A.: Lisp Machine Manual, Fourth Edition, Symbolics, Inc, July 1981, 544p.

（ＬＰＧ－５－２１）　ソフトウェア技術委員会報告書目次案，Ｓ８－２，Nov.12，１９８１，４p.

（ＬＰＧ－６－１）　（社）日本電子工業振興協会：第５世代ロジック・プログラミング技術調査委員会（第６回）議事録，Nov.18，１９８１，６p.

（ＬＰＧ－６－２）　国藤，竹島，安達，加藤，沢村：ＬＰＧ報告書章別概要案，Nov.30，１９８１，８p.

（ＬＰＧ－６－３）　国藤：ロジック・プログラミング技術調査委託報告書（案），Nov.30，１９８１，ＫＪ法Ａ型図解

（ＬＰＧ－６－４）　国藤：ＬＰＧ資料一覧，Nov.30，１９８１，７p.

（ＬＰＧ－６－５）　後藤：ＬＰＧ報告書に向けて，Nov.30，１９８１，２p.

（ＬＰＧ－６－６）　中島：ＬＰＧ報告書章別概要案，Nov.30，１９８１，９p.

（ＬＰＧ－６－７）　梅山：Logic Programming MEMO(2)，Nov.30，１９８１，６p.

（ＬＰＧ－６－８）　黒川：報告書概略案，Nov.30，１９８１，４p.

（ＬＰＧ－６－９）　新田：報告書レジュメ，Nov.30，１９８１，６p.

（ＬＰＧ－６－１０）　安川：報告書概要，Nov.30，１９８１，６p.

（ＬＰＧ－６－１１）　梅村：レジュメ，Nov.30，１９８１，５p.

（ＬＰＧ－６－１２）　林：報告書レジュメ，Nov.30，１９８１，１p.

（ＬＰＧ－６－１３）　上田：報告書レジュメ，Nov.30，１９８１，８p.

（ＬＰＧ－６－１４）　元吉：ＬＰＧ資料，Nov.30，１９８１，３p.

（ＬＰＧ－６－１５）　白石：ＬＰＧ報告書レジュメ，Nov.30，１９８１，８p.

（ＬＰＧ－６－１６）　大谷木：ＦＧＫＬについてのメモ，Nov.30，１９８１，４p.

（ＬＰＧ－６－１７）　佐藤：ＬＰＧ報告書分担分の前書きと粗すじ，Nov.30，１９８１，９p.

（ＬＰＧ－6.5－１）　ソフトウェア技術委員会（編著）：第５世代コンピュータ・ソフトウェア技術委員会報告書，Dec.１９８１，２０p.

（ＬＰＧ－6.5－２）　アーキテクチャ技術委員会（編著）：第５世代コンピュータ・アーキテクチャ技術委員会報告書，Dec.１９８１，２５p.

（ＬＰＧ－6.5－３）　R.Ferguson:PROLOG A Step Toward the Ultimate Computer Language，BYTE，Nov.１９８１，pp. 384−399.

(LPG-6.5-4) R.A.Kowalski: Prolog as a Logic Programming Language, AICA Congress, Pavia, Italy. Sept. 23-25, 1981, 6p.

(LPG-6.5-5) R.Kowalski: Logic as a Database Language, July 1981, 30p.

(LPG-6.5-6) K.A.Bowen and R.A.Kowalski: Amalgamating Language and Metalanguage in Logic Programming, School of Computer and Information Science, Syracuse Univ., June 1981, 30p.

(LPG-6.5-7) K.L.Clark and S.Gregory: A Relational Language for Parallel Programming, Res. Rep.DOC 81/16, Dep.of Computing, Imperial College of Science & Technology, July 1981, 26p.

(LPG-6.5-8) K.L. Clark and F.G. McCabe : PROLOG: a language for implementing expert systems, Nov. 1980, pp.1-19.

(LPG-7-1) (社)日本電子工業振興協会:第5世代ロジックプログラミング技術調査委員会(第6回)議事録, Nov.30, 1981, 7p.

(LPG-7-2) 国藤:LPG報告書章別エントリィ状況, Dec.17, 1981, 1p.

(LPG-7-3) 国藤:報告書作成手順についてのコメント, Dec.17, 1981, 1p.

(LPG-7-4) 黒川:Commentary, Dec.17, 1981, 1p.

(LPG-7-5) 上田:特殊入出力機器としてFGKLマシンが備えるべきもの, Dec.17, 1981, 1p.

(LPG-7-6) 榎本, 宮地, 片山, 榎本: Lorel-2 言語について, 情報処理, Vol.19 No.6, June 1978, pp.522-530.

(LPG-7-7) A.Colmerauer(Chairman): Call for Papers of First International Logic Programming Conference, Marseille, France, Sept. 1982, 1p.

（ＬＰＧ－７－８）　横井：ロジック・プログラミング技術調査委託報告書（第０版），Dec.17, 1981, 81p.

（ＬＰＧ－７－９）　ＪＩＰＤＥＣ・５Ｇ事務局（編）：Ｓ－ＷＧ報告書原稿執筆要領，Nov.19, 1981, 1p.

（ＬＰＧ－８－１）　（社）日本電子工業振興協会：第５世代ロジックプログラミング技術調査委員会（第７回）議事録

（ＬＰＧ－８－２）　後藤：報告書（Ⅱ－１）内容案

（ＬＰＧ－８－３）　中島：2.1 基本データ表現とその管理

（ＬＰＧ－８－４）　梅山：2.2.1 基本論理式

（ＬＰＧ－８－５）　新田：2.2.2 基本計算メカニズム

（ＬＰＧ－８－６）　安達，竹島：2.3　拡張論理式と拡張計算メカニズム

（ＬＰＧ－８－７）　黒川：2.4 抽象化機構

（ＬＰＧ－８－８）　中島：3.1 インタプリタ（コア部）

（ＬＰＧ－８－９）　黒川：4.1 内部データベース・システム

（ＬＰＧ－８－１０）　国藤：4.2 外部データベース・システム（概要案）

（ＬＰＧ－８－１１）　黒川：4.3 ファイル・システム

（ＬＰＧ－８－１２）　元吉：5.1 インタプリンタ（インタフェース部）

（ＬＰＧ－８－１３）　白石：5.2 ＴＶディスプレイシステム

（ＬＰＧ－８－１４）　上田：5.3 ネットワーク

（ＬＰＧ－８－１５）　後藤：ＬＰＧ報告書について（6.1 エディター）

（ＬＰＧ－８－１６）　中島：6.2 デバック支援ツール

（ＬＰＧ－８－１７）　黒川：6.4 ライブラリアン

（ＬＰＧ－８－１８）　林：6.5 マイクロプログラム作成支援ツール

（ＬＰＧ－８－１９）　沢村：Ⅲ－１言語仕様の論理学からの解釈

（ＬＰＧ－８－２０）　中島：Ⅲ－1.1　ＰＲＯＩＯＧ／ＫＲ

（ＬＰＧ－８－２１）　林：Ⅳ－3.1　ＬＩＳＰマシン

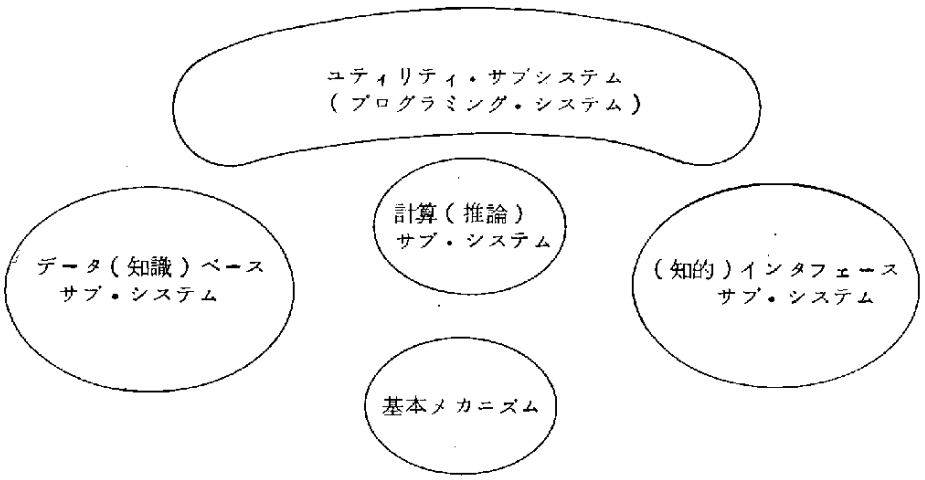
（ＬＰＧ－8.5－１）　ＳＰＲＡＷＯＺＤＡＮＩＡ　ＩＮＳＴＹＴＵＴＵ　ＩＮＦＯＲＭＡＴＹＫＩ　ＵＮＩＷＥＲＳＹＴＥＴＵ　ＵＮＩＷＥＲＳＹＴＥＴＵ　ＷＡＲＳＺＡＷＳＫＩＥＧＯ

（ＬＰＧ－8.5－２）　Predicate Logic as a Language for Parallel Programming by M.H. Van Emden & G.J.de

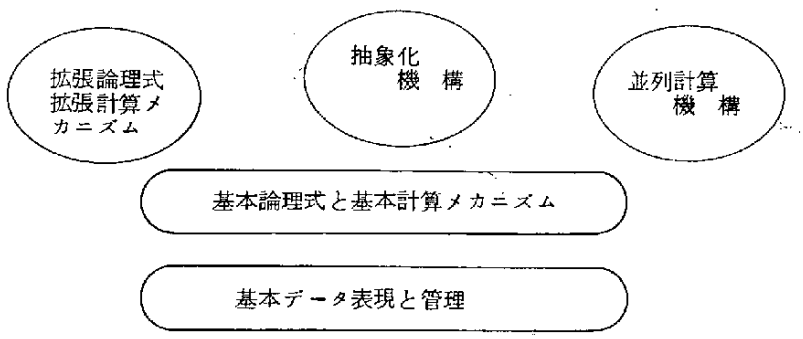
Lucena
(LPG-8.5-3) WATERLOO PROLOG USER'S MANUAL
(LPG-8.5-4) An Algorithm for Interpreting Prolog Programs
(LPG-8.5-5) PERQ(A High Performance Single User Workstation)
(LPG-9-1) (社)日本電子工業振興協会:第5世代ロジック・プログラミング技術委員会(第8回)議事録
(LPG-9-2) 中島:とりとめもなく
(LPG-9-3) 梅山:Ⅱ-2.2.1 原案
(LPG-9-4) 安達, 竹島:Ⅱ-2.3 原案
(LPG-9-5) 国藤, 加藤:Ⅱ-4.2 原案
(LPG-9-6)
(LPG-9-7) 梅村:Ⅱ-6.1 原案
(LPG-9-8) 沢村:Ⅲ-1 原案
(LPG-9-9) 大谷木:Ⅲ-4(プロログのSemanticsへ向けて)原案
(LPG-9-10) 林:マシン例
(LPG-9-11) 白石:Ⅳ-3.2 原案
(LPG-9-12) 白石:Ⅳ-3.3 原案
(LPG-9-13) 白石:Ⅳ-3.4 原案
(LPG-9.5-1) Logic Programming News Letter
(LPG-9.5-2) Centro de Informatica da Universidade de Lisboa Intelligent backtracking and side-tracking in Horn clause programs - the theory
(LPG-9.5-3) Selective Backtracking
(LPG-9.5-4) Backtracking intelligently in AND/OR trees
(LPG-9.5-5) Revision of top-down logical reasoning through Intelligent backtracking

1.3 報告書の構成

'FGKLマシンの全体構成は



であり，第2章に述べられている。さらに，基本メカニズム部分は



となり，第2章，2.2項に詳述されている。

サブシステム群は，

計算サブシステム

インタプリタ（コア部）

コンパイラ

データベースサブシステム

内部データベース

外部データベース

ファイルシステム

インタフェースサブシステム

インタプリタ（インタフェース部）

TVディスプレイシステム

ネットワーク交信システム

ユティリティサブシステム（プログラミングシステム）

エディタ

デバッグ支援システム

デバッガ

トレイサ

ステッパ

探索空間アナライザ

ライブラリアン

という構成になり，第2章，2.3項から，2.6項まで，順番に紹介されている。

以上が，「FGKLマシンのイメージであるが，将来への展開という観点から，第3章に，ロジック・プログラミングの基礎理論に関する4つの論文を収録した。そして，出発の土台ということで，第4章にロジック・プログラミングの現状とマシンの実例というタイトルで，国内で開発され，利用されているシステム例，Prologによるプログラム例，米国で開発された高機能パーソナル・マシンの例をまとめてある。

第2部　'FGKL/Sマシン

2 'FGKL/Sマシン(逐次型推論マシン)

2.1 システム構成

本項では'FGKL/Sマシン[注)]の全体としての機能を紹介し，合わせて本マシンを提案するに至った基本的な考え方を述べる。マシンの細部にわたる記述は2.2項以下に展開される。

2.1.1 システムのスケッチ

'FGKL/Sマシンは，'FGKLと呼ばれる新しいプログラミング言語(2.1.2参照)を高速に実行することの出来るコンピュータである。この意味で'FGKL/Sマシンはいわゆる高級言語マシンと呼ばれる範疇に属する。

'FGKL/Sマシンは，パーソナル・コンピュータとして使われることを想定している。このため形状・重量・消費電力・騒音等は極力小さく抑える計画である。ここで，パーソナル・コンピュータという言葉を使ったが，この言葉は今日ある特定のクラスのマシンを指すのに使われている。そこで，誤解を生じないように若干補足説明をしておく。

今日，パーソナル・コンピュータと呼ばれるマシンは表2.1.1の左側の欄に対応している。これに対し，表2.1.1の最中の欄にはスーパー・パーソナル・コンピュータと呼ばれるクラスを示している。'FGKL/Sは右側の欄である。この表からもわかるように，'FGKL/Sをパーソナル・コンピュータと呼ぶのはその使用形態を指しているのであって，処理速度，メモリ容量ともに今日のパーソナル・コンピュータのクラスを大きく上回っている。

表2.1.1 パーソナル・コンピュータの分類

	いわゆるパソコン	スーパーパーソナルコンピュータ	'FGKL/S
CPUの例	Z80 8080	AMD2901 AMD2903	AMD2903 ～ECL(?)
メモリ容量	～64KB	256KB～ 4MB	4MB～数10MB
ディスク装置	フロッピーディスク (256KB～1MB)	ハードディスク (20MB～80MB)	ハードディスク (200MB～)
ディスプレイ	640×200 ドット	1024×1024 ドット程度	2048×2048 ドット
外部接続	EIA.RS232C 300ボー～9600ボー	Ether Net 10Mbit/S	Ether Net 10Mbit/S

注) 他の報告書中では「逐次型推論マシン」という名前で呼んでいる場合もある。

また肝要な点として，単に量的な尺度（速度，容量）で大きくなっているだけでなく，むしろ質的な面で従来のコンピュータとは一線を画するものである。すなわち，'ＦＧＫＬ／Ｓは利用者に高度のプログラミング環境を提供するものであり，'ＦＧＫＬの言語仕様はその点を熟慮して設計されている。またハードウェア構成上もディスプレイ装置を重視している。

本報告書の段階では，'ＦＧＫＬ／Ｓマシンの内部構成については幾つかのａｌｔｅｒｎａｔｉｖｅがあり得る。したがって完成予想図を掲げるには早過ぎるのではあるが，外観の近似的なイメージとしてはＳｙｍｂｏｌｉｃｓ社のＬＩＳＰマシンが参考になる（4.3項参照）。ただし，'ＦＧＫＬ／Ｓは既有のＬＩＳＰマシンよりもハードウェア量が多いため，第１号試作機は実装上の都合で若干大きめになるかも知れない。また何よりも重要なのは，われわれの目標はＬＩＳＰを走らせることではなく，'ＦＧＫＬにある。ＬＩＳＰと'ＦＧＫＬとの比較については1.2節で言及する。

ソフトウェアの外観に相当するのは，ディスプレイ画面の様子である。ビット・マップ方式でマルチウィンドウを制御する方式はスーパー・パーソナル・コンピュータでも広く採用されているが，'ＦＧＫＬ／Ｓではよりドット数を増加させる。また'ＦＧＫＬ／Ｓの基本応用システムのひとつとして機械翻訳[注]を想定しているから，漢字等の取扱いは十分考慮されている。図2.1.1に'ＦＧＫＬ／Ｓの画面の近似解として，漢字フォントを含むマルチウィンドウの例を示す（〔斉藤・野村１９８２〕より引用）。

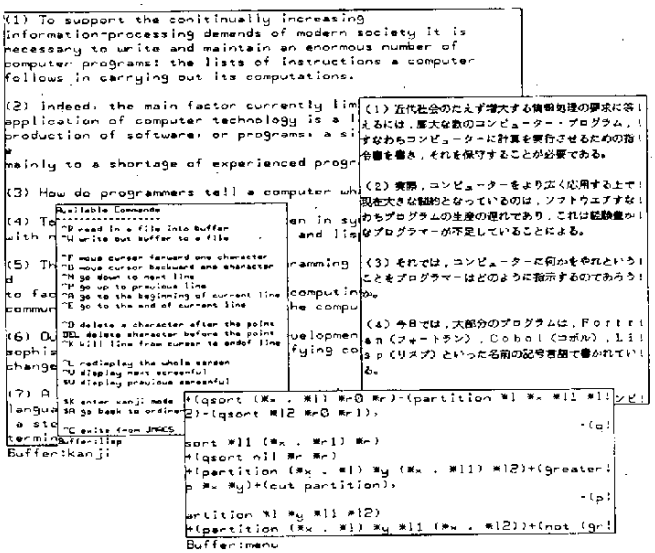


図 2.1.1

注）　ソフトウェア技術委報告書に詳述されている。

2.1.2 ソフトウェア・システムの構成

'ＦＧＫＬ／Ｓマシンは，'ＦＧＫＬと呼ばれる新しいプログラミング言語を実行する。'ＦＧＫＬの言語仕様は，現存するプログラミング言語の中ではＰＲＯＬＯＧに最も近い。ただし，'ＦＧＫＬはＰＲＯＬＯＧそのものではなく，より高度の機能を提供するものである。以下に'ＦＧＫＬの特徴を列挙しておく。

① 抽象化機構を持つ。

② 並列計算機構を持つ。

③ 外部データベース・システムと自然に接続されている。

④ ＴＶディスプレイを重視している。

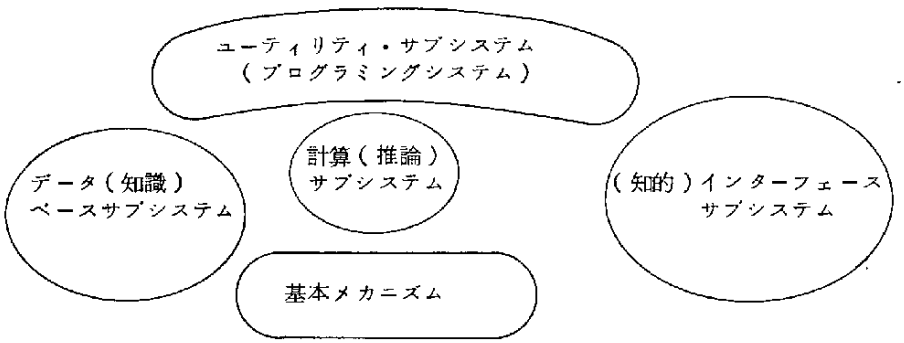
⑤ ネットワーク機能を持つ。

⑥ プログラミング環境（エディタ・デバッガ）を重視している。

これらの特徴は'ＦＧＫＬの性格を物語ることになる。すなわち，'ＦＧＫＬは知識情報処理において中心的な役割を果たす言語であり，システム記述[注]から知識表現に至るまで幅広く使われる核言語（Ｋｅｒｎｅｌ Ｌａｎｇｕａｇｅ）である。[注]

上のような特徴を盛り込むためには'ＦＧＫＬの論理的な基盤を確立する必要がある。本報告書の中でも論理的な議論に幾つかの節を当てている。'ＦＧＫＬはロジックの面から見てもＰＲＯＬＯＧを大きく上回る新語である。

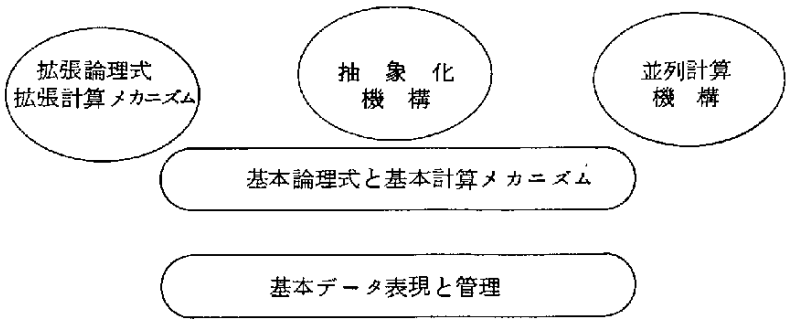
図2.1.2に'ＦＧＫＬ／Ｓマシンのソフトウェア・システムの概略図を示す。



(a) システムの全体構成

注） 例えば，エディタは'ＦＧＫＬ自身で書かれることになるであろう。

注） 'ＦＧＫＬ／Ｓマシンのエンド・ユーザ言語としては別の素人向のプログラミング言語が用意される可能性もある。



(b) 基本メカニズム拡大図

図 2.1.2

従来，知識情報処理の分野ではＬＩＳＰ が広く用いられて来た。LISP と 'FGKL とを比べると，'FGKLの方がより高位の概念を自然に内包している。すなわち，３段論法のような基本的な推論，プロクション・ルールのような非手続き的な記述，パターンマッチング，バックトラックなどの機能はＬＩＳＰ で書けば何がしかのプログラムとして実現されるのであるが，'FGKL では言語自身の中にこれらの機能が内蔵されている。

もちろん，ＬＩＳＰ には学ぶ点も多い。例えばエディタ等のプログラミング環境も含めた構成法や，アセンブラ言語やＣ言語にも比すべき低レベルの細かい記述が可能であることはＬＩＳＰ の優れた特徴である。'FGKL は，これらの諸点について吟味し，現存するＬＩＳＰ 文化を正しく継承するよう設計されている。

2.1.3 ハードウェア・システムの構成

ソフトウェア（'FGKL）に比べると，ハードウェアの構成はまだ幾つかの選択肢を残している。これは，今回の議論の進め方が，言語からアーキテクチャに至るという攻め方をしたためである。今後 'FGKL の言語仕様の細部が固まるにつれて，ハードウェアの最適な構成も自ずから明らかになるであろう。

基本的なハードウェア上の論点としては次のようなものがある。

(1) 仮想メモリVS.実メモリ

仮想メモリ方式は実装メモリ容量が小さい場合でもマシンを一応使うことができる。またアプリケーションが予想以上に膨張した場合にも安全である。これに対し，例えば２０ＭＢ程度の実メモリを付ければ充分で，仮想機構を省いてシンプルな構成にした方が良いという考え方もある。

(2) ディスプレイ：VRAM 方式 VS. バッファ転送方式

速度を重視するならVRAM(Video RAM) 方式が優位である。ただし，VRAM はメモリ空間を必要とするので，ドット数の多いカラー表示の場合にはページ・バッファを設けて転送する必要がある。

(3) データ幅：３６ビットVS.３２ビット

基本的なデータ幅（必ずしもメモリからの読出し幅と同一ではない）は，各種の要因を勘案して決定しなければならない。３６とか３２とかいう数字は，たまたま現存の計算機の例を引合いに出したまでである。[注)] 他にもっと合理的な数字があるかも知れない。

いずれにしても，'FGKL／Sのハードウェアは次の様な基本的な特徴を持つはずである。すなわち，'FGKL の中心部分（例：ユニファイア，データタイプのチエック）をハードウェア化するとともに，ディスプレイ画面を重視した構成とする。さらにローカル・ネットワーク機能を持ち，知識情報処理に携わる研究者・技術者によってパーソナル・コンピュータとして使用されるマシンである。

— 参考文献 —

1) 〔斉藤・野村１９８２〕斉藤康己・野村浩郷

「日本語スクリーン・エディタJMACS の概要」情報処理学会第２４回（昭５７年前期）全国大会，3G－3，１９８２年３月。

2.2 表現と計算（推論） メカニズム

2.2.1 基本データ表現とその管理

ハードウェア・メモリの構成単位（bit, byte, word）を適当に切り分け，システムの基本構成単位となるデータ表現が構成される。本節で述べるデータ表現は，あくまでもシステム構成のための基本となるものであって，'FGKL がユーザに提供するものとは異なる。ユーザ側からは，さらにこれらの上に皮をかぶせたもの（述語，ファイル等）が見えることになる。また，タグ・ビットが開放されているため，システム・インプリメンタあるいはユーザが任意のデータ・タイプを作ることも可能である。

データはポインタと実体の組として表現される（１語ですむ数値の表現は例外）。一般にはひとつの実体に対し複数のポインタが存在しうる。データ・タイプを示すタグはポインタ側に置く

注） DEC2060 …３６ビット，VAX－１１…３２ビット

ものと実体側に置くものの２通りが考えられ，それぞれ長所，短所がある。ここではタイプチエックの早さを優先し，データ・タイプはポインタ側で識別されるものとする。ポインタには通常のものの他に不可視ポインタを用意する。これは（タイプ・チエック時以外には）ハードウェアによって自動的にdereference が一回多くかかるポインタである。

データ・タイプには以下の種類がある。注）

(1) 変数（var）

(2) シンボル（symbol）

(3) 整数（number）

(4) 実数（浮動小数点数）（flonum）

(5) 多倍長整数（bignum）

(6) 多倍長実数（bigflo）

(7) 文字列（string）

(8) ビット列（bits）

(9) ベクター（vector）

(10) ハッシュ・ベクター（hash－vector）

(11) コード（code）

(12) リスト（list）

データ・タイプの判定には括孤内の名前にｐを付加したもの（例，varp，codep）を述語名として用いる。注）又，タイプ交換には括孤内の名前をそのまま用いる。たとえば

```
list( abc , *x)
```

によって *x＝97. 98. 99.＜＞ となる。注） var ⟷ symbol 変換は言語の表現力を増す上で特に重要であると思われる。

新しいデータ・オブジェクトを作るには，new－を用いる。たとえばLisp の gensym に相当する述語は，new－symbolとなる。

A．ポインタ

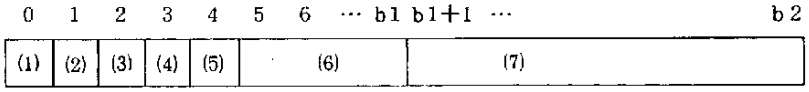
a．ポインタ

注） file－descriptor やディスプレイのbit－map 等はシステム・インプリメンタが定義するデータ・タイプとする。各節を参照のこと。

注） (1)～(8)までをatomp で判定する必要があるかもしれない。

注） コードはASCIIを仮定した。

ポインタは1語を占めるものとし，次のようなビット構成をとる。



ここでb1，b2 はハードウェアによって適当に定めればよいが，ここではb1=8，b2=35 を仮定している。もしb2=47位に大きくとれるのであれば，ポインタにより多くの情報（たとえば変数の場合にはフレーム（ 3.1.1.B 参照）の先頭からの相対番地等を持たせ，より一層の高速化が可能となる。

各領域の意味は以下の通りである。注）

(1) 1（可視ポインタを示す）
0 1
(2) データ／ポインタ

(3) 固定長／任意長

(4) ポインタ領域　無／有

(5) 連続領域　無／有

(6) データタイプ

(7) アドレス

ただし，(2)が0の場合には，(3)〜(7)は以下のとおり。

(3) 整数／浮動小数点数

(4)〜(7) データ

また，(6)のデータタイプのうち，システムで使用されていないものはユーザに開放する。

b．不可視ポインタ

不可視ポインタとは，ハードウェアによって自動的にスキップ（参照が一回多くかかる）されるポインタで，ユーザからは見えない。これは2通りに使われる。

・GCによるデータの引越し先を示す。

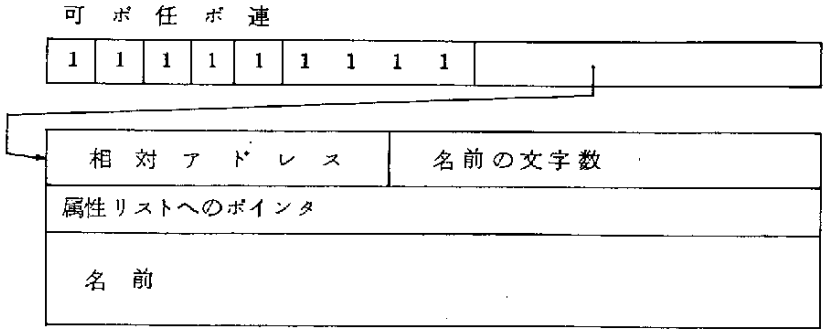
・不可視セルの一部として，リストセルや数値等に属性リストを付加するのに用いる。

不可視ポインタは先頭ビットが0であることの他は可視ポインタと同じ構造である。

B．データ表現

a．変　数

注）(4)，(5)の情報は不要かもしれない。

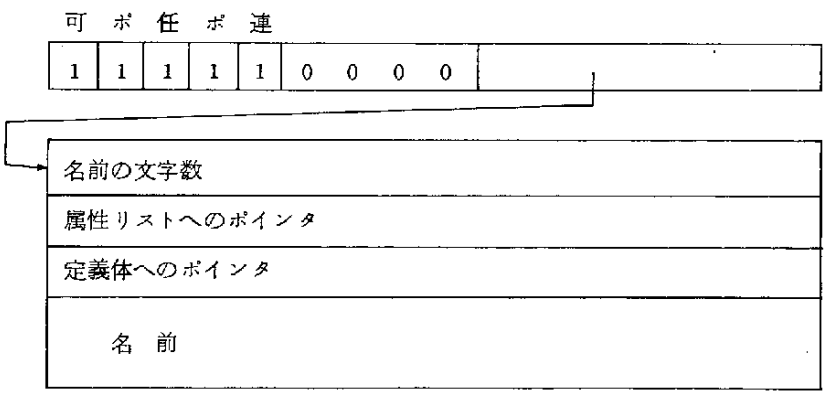


変数であるかの判定には，ポインタの上位1バイトがすべて1かどうかを使う。この判定にはスピードが要求されるので，これは一命令で実現すべきである。

属性リスト[注]は，変数ごとのデータ・タイプの宣言等の際に用いる。名前の情報はデバッグ時に重要である。

相対アドレスとは，フレーム（3.1.1.B）の先頭からの相対アドレスのことで，そこに変数の値が書き込まれる。この情報は，できればポインタ側に移動させ，（フレームの先頭アドレスはレジスタに入れておいて）変数参照の際にはこの相対アドレスでindexing して一度に変数の値がとれることが望ましい。

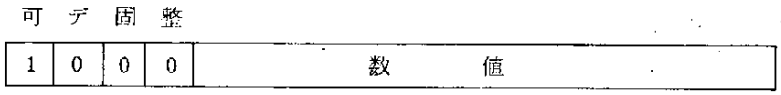
b．シンボル



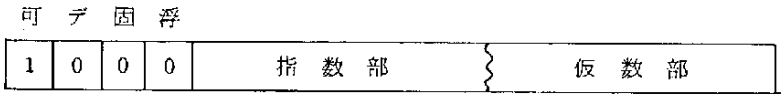
変数が値を持つ代わりに，シンボルは述語としての定義を持つ。unification の方式を指示するためのポインタを加えても良い。

注） 以下でも「属性リスト」という名前を用いるが，これはリストで表現するとは限らず，hashed vector 等を用いても良い。

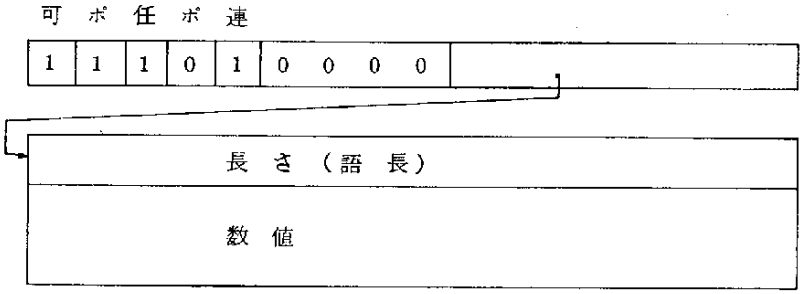
c．整　数



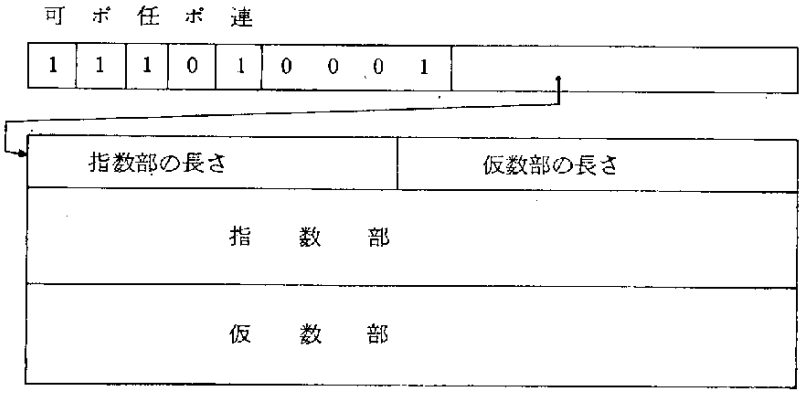
d．浮動小数点数



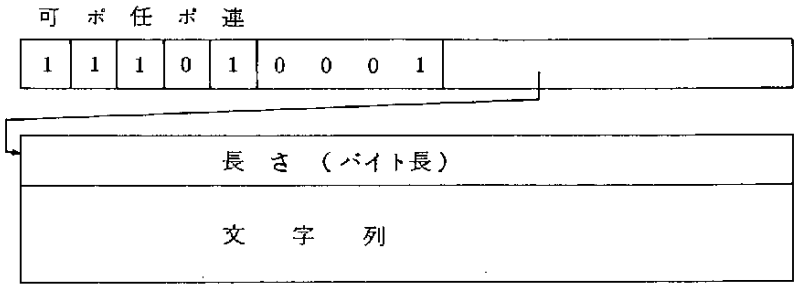
e．多倍長整数



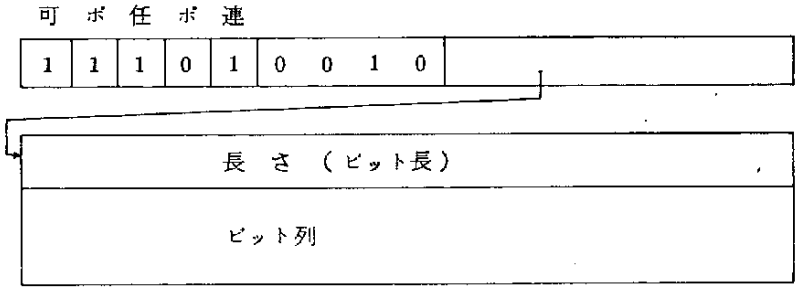
f．多倍長実数



g．文字列

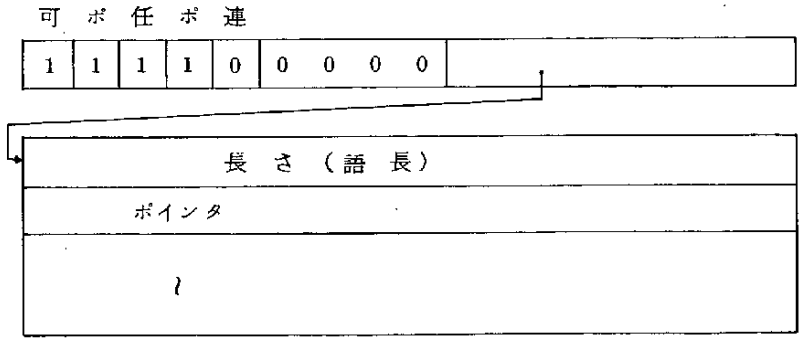


h．ビット列



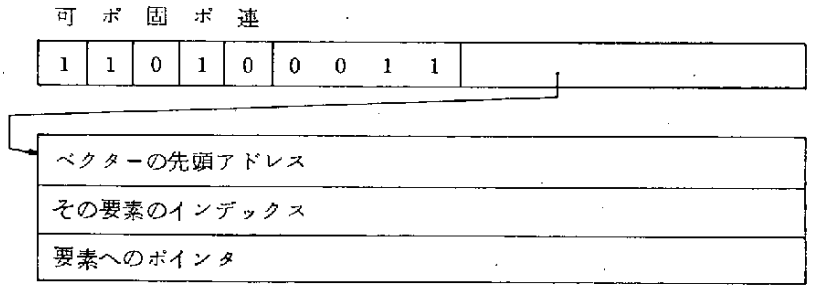
セットの表現等にも用いる。

i．ベクター



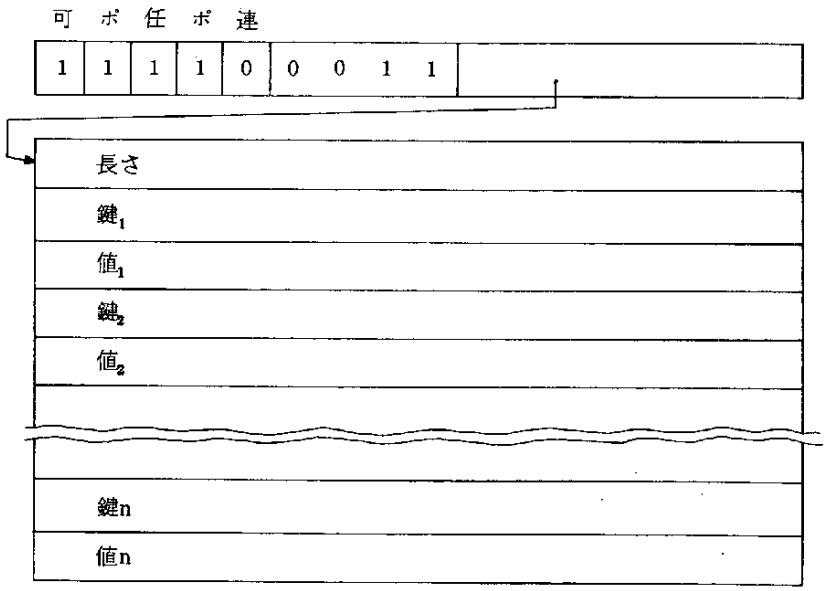
ベクターはリストの縮約形だと思っても良い。

j．ベクターの要素へのポインタ



ベクターの要素を直接参照している方が便利な場合に用いる。

k．ハッシュ・ベクター



ハッシュ・ベクターはベクターの特殊な使い方をしたものだと考えても良い。属性リスト等が大きな場合，リニア・サーチでは時間がかかり過ぎるのでハッシングを用いて高速化する。このような場合にハッシュ・ベクターが用いられる。

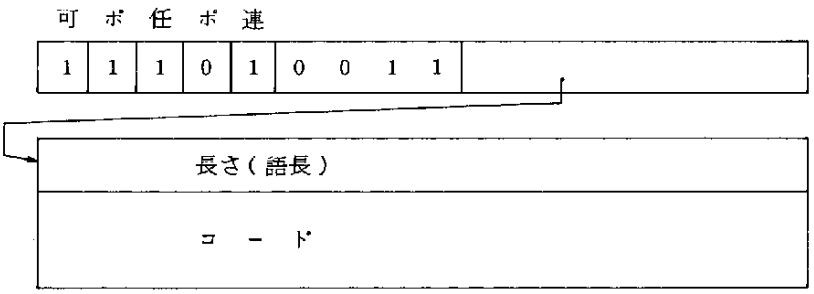
ハッシュ・ベクターは，一般に2種類のデータを関連づけるのに用いられる。たとえば，シンボルの文字列とその実体（読み込みルーチンが用いる）との対応を示すのに用いる。こ

の二種類のデータのうち，元となる方を鍵データ，もう一方を値データと呼ぶことにする。

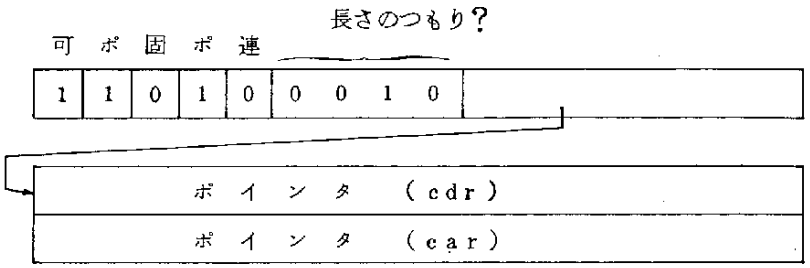
鍵データにハッシュ関数を作用させると，鍵データが格納されているアドレスになる。その次のアドレスに値データが格納されている。異なった鍵データに対し，同じアドレスが割当てられた場合には，もう一度別のハッシュ関数を作用させる方式と，かまわずリストにして同一アドレスに格納する方式があるが，ここでは特に指定しない。

鍵データは，値データともに任意のデータ・タイプで良い。ただし，鍵データがリストの場合には，同一構造のリストは同一アドレスに変換される方が好ましいと思われるので，リストの各要素（Lisp風にいうとcarとcdr）のハッシングの値を合成してアドレスを決めることにする（この手続きは，要素がリストでなくなるまで再帰的に用いられる）。数値の場合にはその値をそのまま用いれば良いし，他のデータ・タイプではそのアドレスを用いれば良い（もちろん，ハッシュ・ベクターの大きさを法とした値にする）。

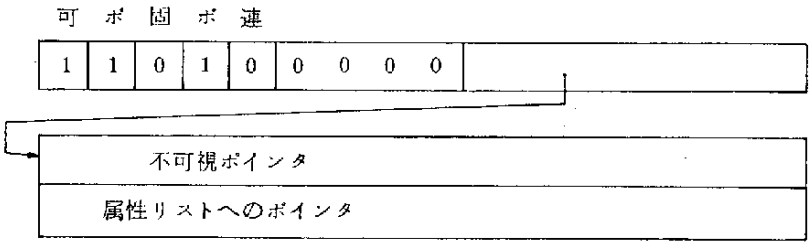
l．コード



m．リスト



n。不可視セル



これは通常属性リストを持たないデータタイプ（数値，文字列，ビット列，ベクター，コード，リストおよびユーザ定義のもの）に属性リストを与えるために用いられる。不可視ポインタの方はハードウェアによってスキップされるが，参照されているデータタイプはそこに書かれている。

C．メモリ管理モジュール

メモリの割り当て，回収（GC）は，このモジュールによって管理される。GC は計算過程と並行して走るのが望ましい（ロボットの昼寝を避けるため）。GC には色々な方法があるが，（表2.2.1参照），注）ここでは，コピー方式を用いることにする。従ってメモリは常に2面用意し，これらを交互に使用する。

	マーク・ビット方式	参照カウンタ方式	コピー方式
コンパクションの有無			
仮想記憶の有無			
並列処理か否か			

表2.2.1　ガーベイジ・コレクションの方式の大別における4方向のベクトル
（他にスタックの有無等の細別がある）

注）Jacques Cohen："Garbage Collection of Linked Data Structures"，ACM Computing Surveys, Vol.13, No.3, 1981
に詳しい解説がある。

メモリが一杯になるともう一方のメモリに必要なデータだけをコピーしてそちらに移動する。そうすると，もとのメモリには不要なデータが残ることになるから，これを捨ててしまえば良い。この方式の長所としては，ＧＣに要する時間が短いこと（メモリの大きさに比例しない），コンパクションが容易であることが挙げられる。又並列化も容易で，新しいデータが作られるごとに，一定個のデータを反対側のメモリに移動してゆけば良い。引越し跡には不可視ポインタを置いてゆく。

スタックの管理もこのメモリ管理モジュールが行う（2.3.1Ａ参照）。変数のbindingの情報を保持するスタック（BS)は，その変数が属している述語の実行が終わった後にも，他から参照されている場合があるので，ただちにたためない。そのため，このフレームの回収はＧＣが担当することになる。ただこの場合，フレーム中のただひとつの変数の値だけが参照されている場合にも，そのフレームへの参照が残っていることになる（変数の値は，フレームのアドレスとそのからの相対アドレスの２本で参照される）ので，他の変数（もはや不要になっている）からのポインタの先のデータも回収されずに残ってしまうことになる。これを避けるためにはＧＣがＢＳだけを特別扱いしてやれば良いのであるが，複雑になるので当面は行わない。

Ｄ．インデックス管理モジュール

インデックスを管理するためには．任意のデータからその出現場所へのポインタが必要となる。不可視セルを一段かませることにより，任意のデータ（たとえばリスト・セル）に属性リストを付加することが可能となる。また，同一リストは一回しか作らないこと（ＨＬＩＳＰのHCONS)を可能にするために，ハッシュ・ベクターが必要となる。

インデックスは属性リスト中のＩＮＤというインディケータの下にベクターとして格納される。そのベクターの第ｎ要素は，その要素がｎ番目に出現する述語のリストとなっている。このリストは線型ではなく何らかの形でソートされた構造体（たとえばバイナリーリスト）あるいはハッシュ・ベクターが望ましい。

Ｅ．データ操作用述語

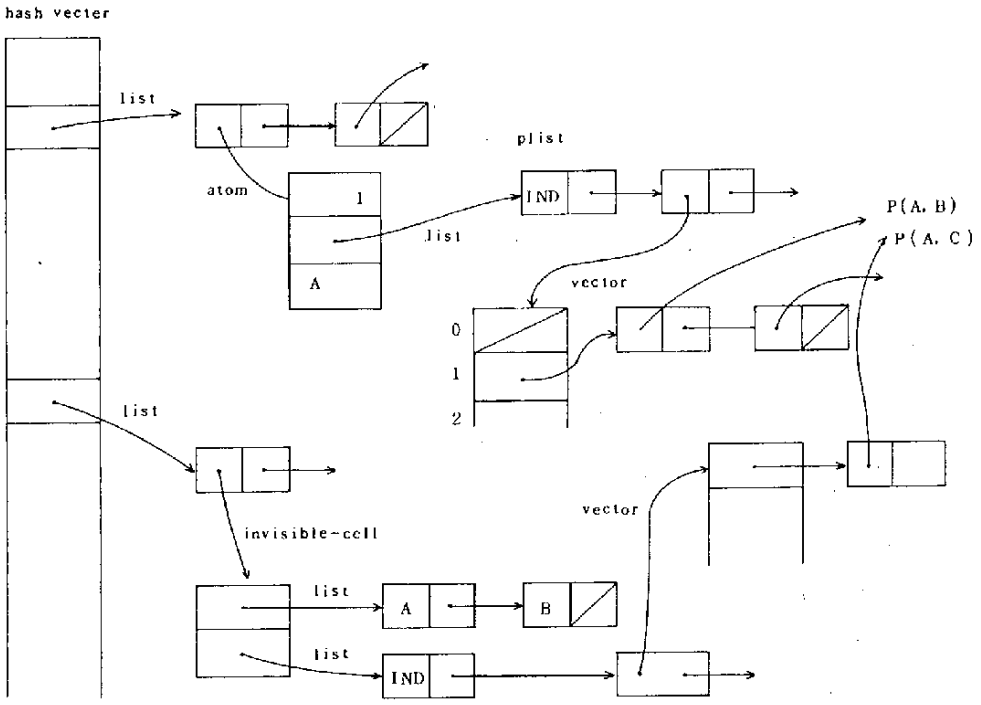
以下データタイプごとに，そのデータを操作するのに必要な基本述語を列挙する。特に説明を要すると思われるもの以外は引数の形を示すにとどめる。｛　｝内はオプションである。

ａ．変　数

・varp(＊x)

＊x がまだ値を持っていない変数かどうかを判定する。

・new-var（〔＊symbol,〕〔＊number,〕＊v）



新しい変数名（Lispのgensymと同様に，シンボルの後に数字が続く）を作って*V をそれにbindする。obvector （アトムのハッシュテーブル）に登録しないので，たまたま他に同一名の変数が存在しても別の物として扱われる。

- var（ *x, *v ）

  *x をタイプ変換し，変数にしたものを*v の値とする。*x としては，シンボル，整数，文字列あるいはnew-varで作られた変数が許される。new-varと異なり，obvectorに登録する。

- deref('*var', *value)

  *var(quote しなければならない）の値を一段だけたぐったものを*valueに入れる。

- length(*var, *n)

  プリントした際の文字数

b．シンボル

- symbolp(*x)
- new-symbol([*symbol,] [*number,] *s)

  new-var 同様新しいシンボルを作るが，obvector には登録しない。
- symbol(*x, *s)

  *x としては，変数，整数，文字列あるいは new-symbol で作られたシンボルが許される。
- length(*x, *n)

  プリントした際の文字数

c．数値（整数，浮動小数点数，多倍長整数，多倍長実数）

- numberp(*x)
- flonump(*x)
- bignump(*x)
- bigflomp(*x)
- number(*x, *n)
- flonum(*x, *f)
- bignum(*x, *b)
- bigflo(*x, *b)
- ＋(*x, [*y]$_{ooo}$ *z)
- －(*x, [*y]$_{ooo}$ *z)
- ／(*x, [*y]$_{ooo}$ *z)
- mod(*x, *y, *z)

  ⋮

（この他に三角関数や対数等）

- length(*x, *n)

  プリントした際の文字数

d．文字列

- stringp(*x)
- string(*x, *s)
- new-string([*number,] [*string,] *s)

  *number で与えられた長さの文字列を *s に入れる。*string で初期値が与えられな

ければ，空白がつめられる。

new-string("abc", *s)

は文字列のコピーと同じ。

・sub-string(*string〔*from〕〔*to,〕*s)

*string の *from 番目から*to番目までが*s に入る。*from も *to も省略された場合には文字列のコピーになる。

・string-append(*s1,〔*s2,〕$_{...}$ *s)

*s1から*sn までを連結する。

・string-reverse(*s1,〔*s2〕)

*s2が省略されるともとのstringが変更される。

・sset(*string, *number, *char,〔*s〕)

*stringの*number 番目を*char に変えたものを*s に入れる。*s を省略するともとの文字列が変更される。*char としては，シンボル，文字列，整数(文字コード)が許される。*char が2文字以上の場合には，第1字目のみが有効となる。

・sref(*string, *number, *char)

*string の*number 番目を*char に入れる。

・length(*s, *n)

文字数を返す。

e．ビット列

・bitsp(*x)

・bits(*x, *b)

・sub-bits(*bits,〔*from,〕〔*to,〕*b)

・bits-append(*b1,〔*b2,〕$_{...}$ *b)

・bits-reverse(*b1〔,*b2〕)

・bset(*bits, *number, *n〔,*b〕)

・bref(*bits, *number, *n)

・length(*b *n)

f．ベクター

・vectorp(*x)

・vector(*x, *v)

リストをベクターに変換する。

・new-vector([*number,][*vector,] *v)

*number（長さ），*vector の一方，または両方を与える。

*vector が与えられた場合には，その中味を*v にコピーする。

・new-vector（*number. *x., *v)

長さ*numberのvector(*v) の各要素を*x に初期化する。

・sub-vector(*vector, [*from,][*to,] *v)

・vector-append(*v1, [*v2,]。。。 *v)

・vset(*vector, *number, *x[, *v])

*vector の第*number 要素を*x に変更する。

*v を与えたときは，元のベクターは変更されずに，*v に新しいベクターが作られ，他の要素は元のベクターからコピーされる。

・vref(*vector, *number, *x)

*vector の第*number 要素の値を返す。

・length(*vector, *length)

g. ベクターの要素へのポインタ

・rset(*ptr, *value)

・rref(*ptr, *value)

・rvector (*ptr, *vector, *index)

ベクターの要素へのポインタを与えると，そのベクターと，要素のインデックスを返す。

h. ハッシュ・ベクター

・new-hash-vector(*number, *v)

*number でハッシュ・ベクターに格納したい鍵データの個数を指定する。システムではそれより大きい適当な長さのハッシュ・ベクターを用意する。

・hset(*v, *key, *value)

・keyを鍵データ，*valueを値データとして*v に格納する。

・href(*v, *key, *value)

i. コード

・codep(*x)

・code-append(…)

・cset(…)

・cref(…)

・length(*c, *n)

j. リスト

・listp(*x)

・list(*x, *l)

ベクターをリストに変換

・append(*x1, [*x2,]$_{\circ\circ\circ}$ *y)

・reverse(*l1, *l2)

・lset(*list, *number, *value)

・lref(*list, *number, *value)

・length(*list, *length)

k. ユーザによるタイプ定義のための述語

・new-type(*type, {fix-length, var}, {pointer, no-pointer}, {連続域の有無})

タイプ名*type の新しいデータ・タイプを定義する。固定長／可変長の別，ポインタ領域の有無及び大きさ，連続領域の有無及び大きさを与える。システムは，他のデータ・タイプとまぎらわしくないよう，タグ・ビットの値を設定する。タグ・ビットが一杯の場合（同一形式で，異なるデータ・タイプ名のものが多く宣言された場合）にはエラーとなる。

2.2.2 基本論理式と基本計算メカニズム

A. 基本論理式

ここでは'FGKL のプログラム構成要素として用いられる基本論理式について述べる。

'FGKL プログラムは次に定義されるような（基本論理式である）non-guarded clause, guarded clauseの集合であり，このようなプログラムに対して目的節（question clause)を与えることにより実行が開始される。与えられた目的節がプログラム中の節によって証明された場合，目的節の実行は成功し，目的節中の変数にunify された情報が質問に対する解答として返される。そうでない場合，目的節の実行は失敗する。

この章では，まず構文の基本単位であるtuple について述べ，次に節，定義体，プログラムの基本的動作等について記述する。

(1) tuple

'FGKL では次のように定義されるtuple が構文の基本単位となる。

tuple ::= < > | <element {, element}>

element::=定数 | 変数 | tuple

<>を空 tuple と呼ぶ。定数および変数は次のように定義される。

(i) 定数

定数はアトムおよび数から成る。

アトムは1文字以上の文字列である。ただし次のような場合には"'"で両側を囲まなければならない。

(a) 文字列中に特殊文字(under bar "_"を除く)あるいは空白を含む場合。ただし特殊文字のみを含む場合で,先頭の文字が"*"で無い場合は囲まなくとも良い。

(b) 数字,あるいは数字と小数点から成る文字列をアトムとしたい場合。

例 apple my-book 'Daigo-Sedai' -->

'**' '1982'

"'"をアトム中で用いる場合には2度続けて使用する。

例 'It''s me'' ''

数は整数の場合数字列であり,実数の場合整数部と小数部を表わす数字列を小数点"."でつないだものである。

例 1982 197.65

("."の両側共,空白があってはならない。)

(ii) 変数

変数は"*"で始まる文字列である。"*"を除いた残りの文字列はアトムの表現法に従わなくてはならない。注)

例 *apple *my-book *'Daigo-Sedai' *-->

*'**' *'1982'

ここでtuple の例をあげる。

<member, *x, *y> <+, *x, <+, *y, *z>>

また, tuple が1個以上のelement を持つ場合,その第1 element を特にprincipal element と呼ぶ。tuple を$<e_1, e_2, \cdots\cdots, en>$とすると,principal element ($e_1$)を特別に外へ取り出して

e_1 (e_2, e_3, ···, en)

注) 匿名変数(anonymous variable : 節中に1度しか表われない変数)は,"*"によって表現する。

と記述することが許される。[注] 例えば上の例は次のように記述できる。

member(*x, *y)　　+(*x, +(*x, *y))

以降では tuple の表現として，ほとんどこの表現を採用する。[注]

(2) 節

FGKL プログラム中で手続き等を表現する単位となるのが節である。ここでは，まずゴール，sequential component，quarded component の構文を定義し，その後，節の構文を定義する。

(1) ゴール

ゴールは次の goal で定義される。後述するように goal1 は節頭に用いられ，goal2 は節体中に用いられる。

goal ::= goal1 | goal2

goal1 ::= <<., 定数, 定数>{, element}>

goal2 ::= 変数 | <変数{, element}>
| <<., 定数, 変数>{, element}>
| <<., 変数, 定数>{, element}>
| <<., 変数, 変数>{, element}>
| goal1

注) tuple が1個のelement しか持たない場合(<e>)，その tuple をこの記法に従って記述するとe( )となる。

注) これは e_1 をprefix のoperator 的にとらえた表現となっている。実は後述する属性宣言節を用いて，定数に対して infix，postfix，right associative，left associative，優先順位等のoperator 宣言が可能である。

例えば"+"をinfix の riqnt associative な operatorと宣言すると上述の例

<+, *x, <+, *y, *z>> は

*x+*y+*z

と記述できる。

また"."をinfix の right associative な operator と宣言すると

<., a, <., b, <. c, <>>>>

<., *x, *y>

は，

a . b . c . <>

*x . *y

と記述できる。

ゴールがtuple であり，かつそのprincipal element も tupleであった場合で，そのprincipal elementの第2 elementが定数であるならば，その定数をこのゴールのモジュール名と呼ぶ。また，principal elementの第3 elementが定数であるならば，その定数をこのゴールの述語名と呼ぶ。

例　<<.，A，append>，*x，*y，*z>

この例ではAがモジュール名，append が述語名である。"."がinfixのoperator として宣言されていると，これは

A . append(*x，*y，*z)

となる。以降の例では"."はinfixのoperatorと宣言されているとする。

(ii) sequential component

sequential componentは次のscで定義される。

sc ::=< > | goal2
| <∧, sc, sc>
| <∨, sc, sc>
| <not, sc>

例　<∨, <∧, A.p(*x), A.q(*y)>, A.r(*z)>

"∧", "∨"がinfixのright associative な operator として宣言され，また"∨"より"∧"の結合優先順位が高いと宣言されているとするとこれは

A.p(*x)∧A.q(*y)∨A.r(*z)

となる。以降の例では，"∧", "∨"は上記のように宣言されているとする。

(iii) quarded component

quarded componentは次のgcで定義される。

gc ::=<|, sc, sc>

guarded componentの第2 elementをガード部，第3 elementを実行部と呼ぶ。

例　<|, A.p(*x), A.q(*y)>

この例では，A.p(*x) がガード部，A.q(*y) が実行部である。"|"がinfixのoperatorとして宣言されているとすると，これは

A.p(*x) | A.q(*y)

となる。以降の例では，"|"はinfixのoperatorとして宣言されているとする。

(iv) 節

節は次のclauseで定義される。節には，non-guarded clause，guarded clause，目的節（question clause）の3種がある。

clause :: = non-guarded clause | guarded clause |
| question clause

non-guarded clause :: = <←, goall, sc>

guarded clause :: = <←, goall, gc>

question clause :: = <←, < >, sc >

節の第2 element を節頭，第3 element を節体と呼ぶ。

例 <←, A.p(*x), A.q(*x)∧A.r(*y)>

<←, A.p(*x), < >>

<←, B.p(*x), B.q(*x) | B.q(*y)>

<←, B.p(*x), < > | < >>

1番目の例では，A.p(*x) が節頭，A.q(*x)∧A.r(*y)が節体である。"←"がinfix の operator として宣言されていると，これらはそれぞれ

A.p(*x)←A.q(*x)∧A.r(*y)

A.p(*x)←

B.p(*x)←B.q(*x) | A.r(*y)

A.p(*x)←|

となる。（記法として，このような場合，空tuple <> は省略できるものとする。）以降の例では，"←"はinfixのoperatorとして宣言されているとする。この例で，1番目と3番目の節がホーン節における手続き節，2番目と4番目の節がホーン節における表明節に対応している。

(3) 定義体

節頭のprincipal elementが同一である節の集合を定義体と呼ぶ。定義体に含まれる節は，全てが guarded clause であるか，全てがnon-guarded clauseであるか，いずれかでなければならない。

(4) プログラムの基本的動作

章の初めで述べたように，'FGKL プログラムは non-guarded clause，guarded clauseの集合であり，このようなプログラムに対して目的節を質問として与えることにより，プログラムの実行が開始される。目的節の実行が成功するか否かは次のように定義される。

ある目的節の実行が成功するのは，その節体である sequential component の実行が成功した場合である。sequential component の実行が失敗した場合，目的節の実行も失敗する。

sequential component の実行が成功するのは，次の場合に分けられる。それ以外の場合には sequential component の実行は失敗する。

(i) sequential component が空 tuple <>の場合。

sequential component は直ちに成功する。

(ii) sequential component がゴールの場合。

そのゴールの実行に成功すれば sequential component も成功する。

(iii) sequential component が<∧, sc1, sc2> の場合。

sc1 が成功し，次に sc2 が成功すれば sc も成功する。

(iv) sequential component が<∨, sc1, sc2>の場合。

sc1が成功するか，あるいは sc1 が失敗しても sc2 が成功すれば，sc も成功する。

(v) sc が<not, sc1>の場合。

sc1 が失敗すれば，sc は成功する。

ここでゴールの実行が成功するとは，そのゴールと unify 可能な節頭を持つ節が選ばれ，かつその節の節体の実行が成功した場合である。[注] それ以外の場合には，ゴールの実行は失敗する。

ただし，節の節体の実行が成功するのは，次の２つの場合である。それ以外の場合には節体の実行は失敗する。

(i) 節が non-guarded clause の場合

節体は sequential component であり，この場合は上記の sequential component の成功および失敗の定義による。

(ii) 節が quarded clause の場合

節体の guarded component を<|, sc1, sc2>とする。sc1 が成功し，次に sc2 が成功すれば，節体の実行も成功する。

ただし，ガード部（sc1）の実行が成功し，更に実行部（sc2）の実行を開始

注）'FGKL/Sマシン上の実行では，節に順序付けがなされ，その順序に従って，ゴールへの適用を試みるものとする。

した場合，この節は"選択された"と呼ばれ，同じ定義体に含まれる，この節以外の節を実行することは禁止される。

(ii)のような節の選択法をguarded selectionと呼び，これに対して，(i)のような節の選択法を non-guarded selection と呼ぶ。

(5) モジュール

プログラムは節の集合である。この集合はいくつかのモジュールに類別される。ある節がどのモジュールに属しているかは，その節の節頭のモジュール名によって識別される。また前述したように，モジュール内の節はさらにいくつかの定義体に，（その節頭の述語名によって）類別される。

定義体にはそれぞれglobal，localの属性が属性宣言節（後述）を用いて宣言される。global と宣言された定義体に含まれる節は他のモジュールからの参照が可能であるが，local と宣言された定義体に含まれる節は，外部からの参照ができない。（図2.2.2）

例＊　A.access-attribute(foo, global)

A.access-attribute(first, local)

A.foo( ) ← B.fix( ) ∧ A.first( )

A.foo( ) ← C.fix( ) ∧ A.first( )

A.first( ) ←……

A.first( ) ←……

（fix は，B，Cというモジュールでglobal宣言されているとする。（図2.2.2参照））

注）モジュールの記法として，例えば次のような記法も考えられよう。

```
module A of
  foo( ) ← B.fix( ) ∧ first( )
  foo( ) ← C.fix( ) ∧ first( )
        where
          first( ) ←……
          first( ) ←……
end-module
```

モジュールを用いることにより，プログラムのモジュール化を行うことが可能となり，またモジュールごとのコンパイル（分割コンパイル）も支援できよう。[注)]

組込み述語等の，システム定義の述語に対応する定義体は，systemと呼ばれる特別のモジュールに含まれているものとする。

注） また，将来，モジュールのtemplate の定義，およびそのtemplate から実体(instance)をinstantiate するための特別な述語を導入することにより，object－oriented なプログラミングへの対応も可能となろう。この場合，local と指定された定義体（節）は，Smalltalk の instance variable 的に用いられる。

例えば，turtle（ディスプレイ上で，線を描くために用いられる方向を持った点）は，template として次のように定義できる。（template を module－class と名付けている。）turtleの最初の位置は（0,0）に設定されており，記法は前の脚注の記法を採用している。

```
module－class turtle of
    current－position(*x, *y) ← position(*x, *y)
    go－ahead (*distance)
            ← position(*x, *y) ∧ direction(*degree)
            ∧ new－position(*x,*y, *degree, *distance, *new－x, *new－y)
            ∧ system.retract(position(*x,*y))
            ∧ system.assert(position(*new－x, *new－y))
    change－direction(…) ← …
    ……
    ……
                where
                    position(o, o)
                    direction(o)
                    new－position(…) ←……
                    ……
end－module－class
```

プログラム中で例えば

…… ∧ system.create(turtle, my－turfle) ∧ …

と行なうことにより，my－turfle というモジュール名を持つモジュールが生成される。my－turfle は，例えば

…… ∧ my－turfle. go－ahead(5) ∧ ……

とすることによって，移動させることができる。

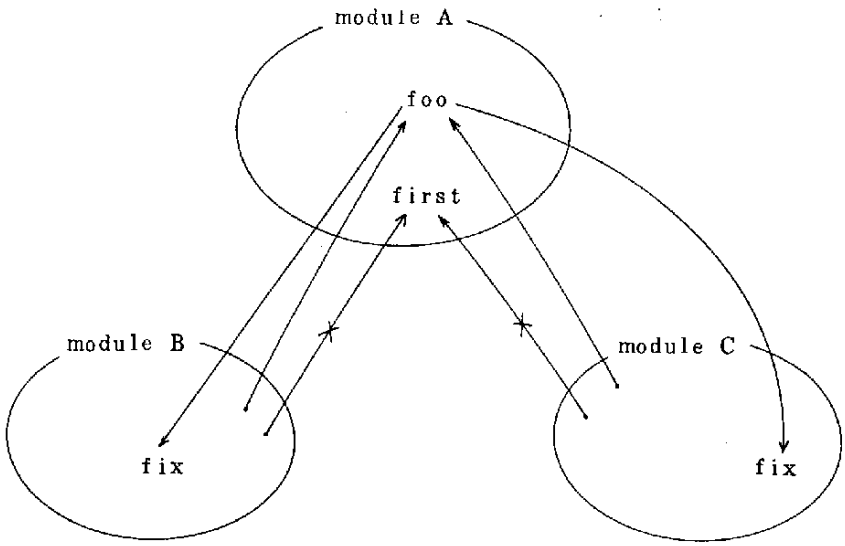


図2.2.2

(6) 属性宣言節

今までに何度か述べてきたように，定数に対してはさまざまな宣言が可能である。それらの宣言は属性宣言節として宣言される。属性宣言節はシステム・プログラムが実行に際して参照する節と見ることができる。

属性宣言節としては，例えば

① operator-attribute

ある定数がinfix 等のoperator であることを宣言する。

② access-attribute

その定数を述語とする定義体が外部モジュールから参照可能か否かを宣言する。

等が考えられる。

(7) Concurrent Programming〔参1〕

Concurrent なプログラミングをサポートするため"//"記法を導入する。注）

$S_1 // S_2 // S_3$ （S_1, S_2, S_3 は sequential component）

"//"は意味的には"∧"と同じであるが，操作的にはS_1, S_2, S_3 がconcurrent

注） ここでは concurrent programming の一例を紹介するにとどめる。

この内容に関しては，今後詳細な議論が必要である。

に実行されることを示す。

sequential component 間で共有されている変数をチャネル変数と呼び，sequential component 間の通信に用いられる。ただし，producer，consumer となる component は静的に決定されているものとする。producer となる component（チャネル変数へのバインドを行う component）は唯一個であるとし，producer であることを示すため，"^"を変数名の肩に付ける。"^"の付いていないチャネル変数を持つ component は consumer（チャネル変数への参照のみを行う）となる。

```
p ( *x^ ) // q ( *x ) // r ( *x )
```

このような記述[注)]は，図2.2.3のようなネットワークに対応している。

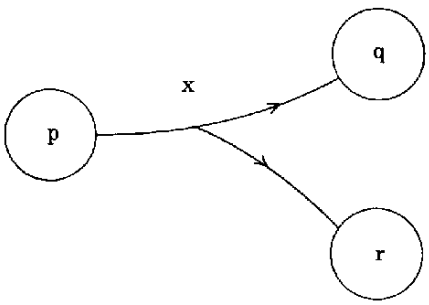


図2.2.3

例　自然数列の2項づつの和を求めるプログラム

```
←integers(1, *x^ ) // add( *y, *x).
 integers( *u, *u : *x) ← | sum( *u, 1, *v)
                          ∧ integers( *v, *x)
 add(*x12 : *y, *x1 : *x2 : *x)
          ← | sum( *x1, *x2, *x12) ∧ add( *y, *x).
```

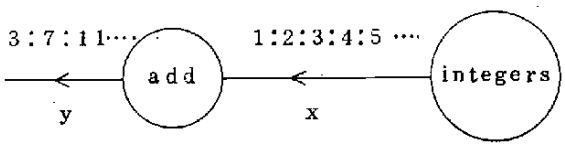


図2.2.4

注）モジュール名は省略されている。以降，誤解を生じない限り，例の中では，モジュール名を省略することがある。

integers の変数 *x に unify されるのは，次第に近似されてゆく（無限）リストであるが，このようなリストのことをstreamと呼び，特に":"を用いて表現している。

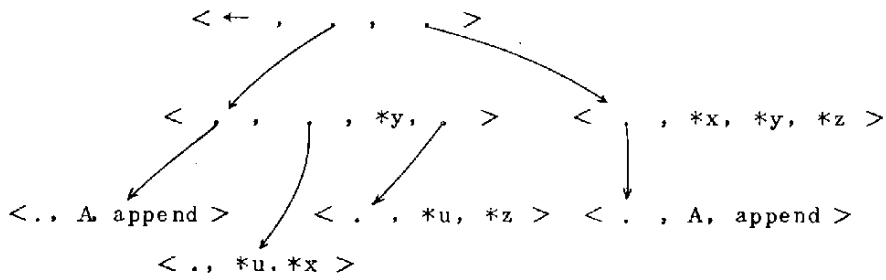
(7) 内部表現

変数および定数は，基本データタイプのうち変数，シンボル，整数等によって内部表現される。またtuple はベクターによって表現される。

例

A.append(*u.*x, *y, *u . *z)

← A.append (*x, *y, *z)



モジュールは節の集まりであるが，内部的には，例えば1個のハッシュ・テーブルと見ることもできよう。このテーブルによって，述語がハッシュされ，モジュール内の対応する節のリストを見つけることができる。節管理は例えば図2.2.5のように行われる。注）

注）モジュールの template の instantiate(create) を行うためには，単純には例えば clause hash table をコピーすれば良い。（もちろん local 節のコピー等も必要であろう。）

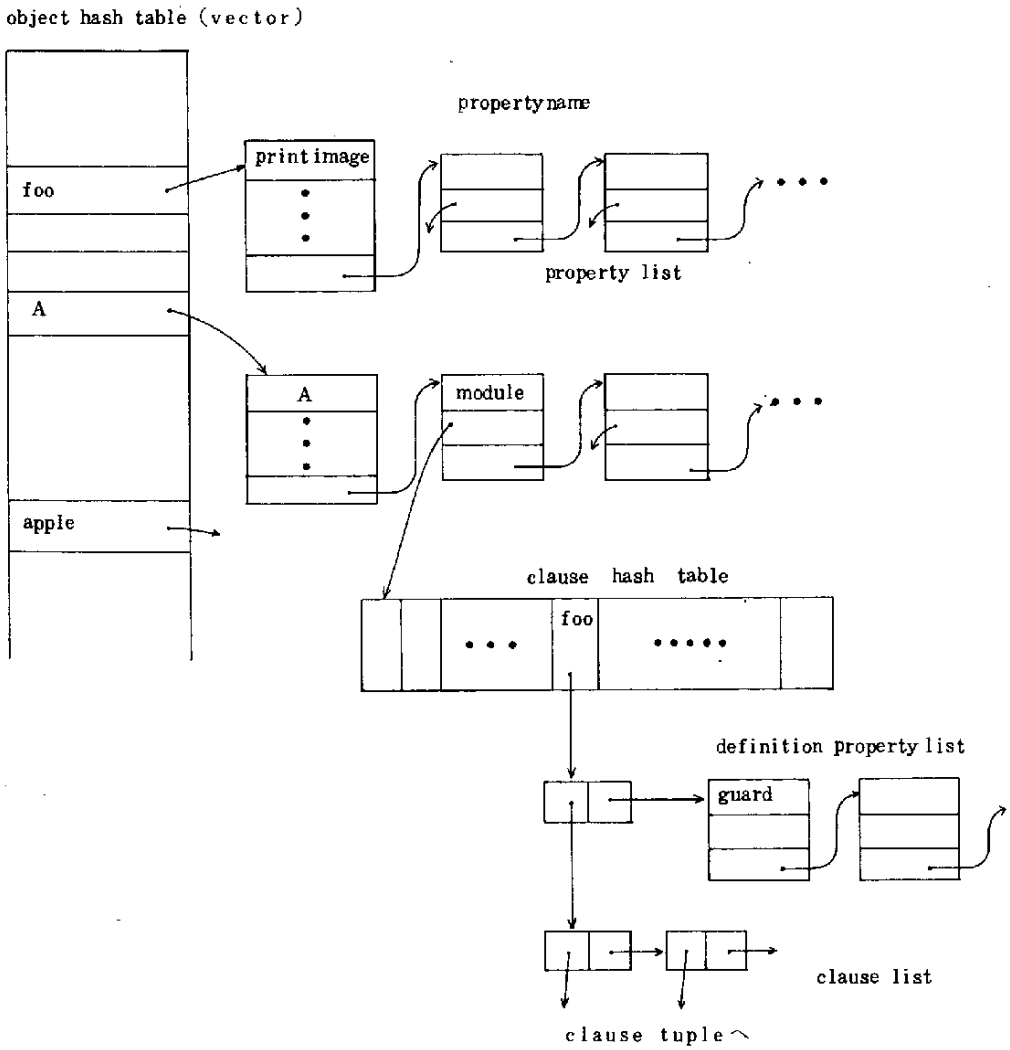


図2.2.5

図2.2.5

— 参考文献 —

1) K.L.Clark & S.Gregory
"A Relational Language for Parallel Programming"
Imperial College of Science & Technology,
Research Report DOC 81/16. July 1981.

B．基本計算メカニズム

ここでは'FGKL のプログラムをsequential に実行するための基本的なメカニズムについて概説する。

a．ユニフィケーション[1][2]

'FGKL における証明にはユニフィケーションが繰り返し行われる。

ユニフィケーション

定数の集合をC，変数の集合をV，CとVから成るtuple の集合をUとする。tuple の対(ui，vi)，ui，vi ∈U が与えられたとき，$\sigma \triangleq \{ (tj/xj) \mid xj \in U \vee C \vee V \}$ なる代入が$\sigma(ui)=\sigma(vi)$ を満たすとき，σをユニファイア(unifier)とよぶ。

2つのユニファイア$\alpha=\{ t_1/x_1, \cdots, t_m/x_m \}$．$\beta=\{ s_1/y_1, \cdots, s_n/y_n \}$ のcombination $\alpha \cdot \beta$ は $\{ t_1 \cdot \beta/x_1, \cdots, t_m \beta/x_m, s_1/y_1, \cdots, s_n/y_n \}$ で定義される。(ただし，$t_i \beta = x_j$，$y_j = x_j$ となるものを除く。また，代入は，各変数に同時に起こる)

あるtuple の対(vi，ui)について，ユニファイアθがあって，この対の任意のユニファイアαに対して$\alpha=\theta \cdot \beta$ なる代入βが存在するときθをmost general unifier (m．g．u)と呼ぶ。m．g．uを求める操作がユニフィケーションである。

例1：(f(*x，*y)，f(g(*z),*w)) に対するm．g．u は

$$\theta = \{ g(*z))/*x, *w/*y \}$$

である。

例2：(f(*x)，f(g(*x))) ではm g uが

$$\theta = \{ g(g(g(\cdots)))/*x \}$$

という無限の代入を起こすため，ユニファイできない。

Occur check

例2に示すような無限の代入を予め検知することをOccur check という。Occur Check は，定理証明等，論理的な完全性を必要とする場合には必要であるが，これを含むユニフィケーションのコストは，パラメータ数の増加と共に指数関数的に増加する。(Occur Checkを含まなければ，コストはパラメータ数に比例)従って，ユニフィケーションとしては，オプションでOccur Checkを選択できることが望ましい。

代表的なユニフィケーションのアルゴリズムを挙げる。

Robinson のアルゴリズム

tuple の対をWとする。

step1　　k：＝O，Wo：＝W，σ_o：＝ε

step2　　Wk が singleton ならstop.（σkがWのm・g・u）さもなくば，
Wk の中で一致しないものをDk にセット

step3　　Dk の中に次の条件を満たす*vk と tk があればstep4
さもなくば，stop（Wはユニフィケーションできない）
条件：*vkは tk に含まれない

step4　　σk+1：＝ σk・{ tk／*vk }，Wk+1：＝Wk { tk／*vk }

stop5　　k：＝k＋1，step2へ

このアルゴリズムは最も基本的なものであるが，ユニフィケーションのコストがパラメータ数と共に指数関数的に増加することがある。

Linear Unification アルゴリズム

tuple をそれぞれ有向グラフ（direct acyclic graph）で表わし，principal element に対応するノードをfunction node，変数に対応するノードをvariable node とする。

例3：(a(b(*v), c(*u, *v)), a(b(*w), c(*w, d(x, y))))

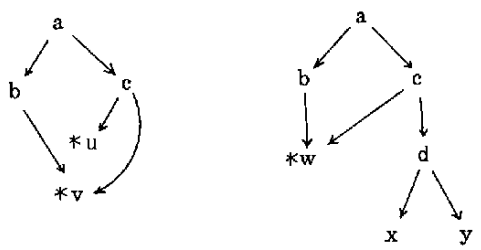


図2.2.6　direct aciclic graph

(function node は a, b, c, d
variable node は *u, *v, *w)

このアルゴリズムにより，図2.2.6のグラフの各ノードは図2.2.7に示すような同値類に分割され，この結果から，m. g. u {d(x, y)／*u, d(x, y)／*v, d(x, y)／*w } が得られる。

```
Algorithm C
Comment: to test <u,v> for unifiability
   Create undirected edge (u,v)
   While there is a function node r, Finish (r).
   While there is a variable node r, Finish (r).
   Print "UNIFIED".

Procedure Finish (r)
 Create new pushdown stack with operations
      Push(*) and Pop.
 Push(r)
 While stack non-empty do
 Begin
  s:=Pop
  If Pointer (s) defined then begin
     If Pointer(s)≠r, Print "FAIL-LOOP" and halt
       end
    else
     Pointer (s):=r
  If r,s have different function symbols,
    print "FAIL-CLASH".
  While there is a father t of s, Finish (t).
  If s is variable node, print "s→r"
   else create undirected edges
     {(j^th son(r),j^th(s))|j=1,...,outdegree(r)}
  While there is an undirected edge (s,t),
     Push (t) and delete (s,t).
  If r≠s, delete node s and all dag arcs out of s.
 End.
 Delete node r and all dag arcs out of r.
End of Finish
End of Algorithm C.
```

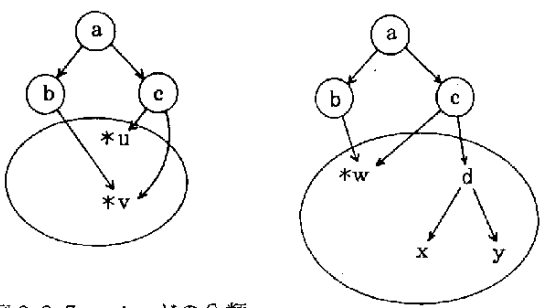


図2.2.7　ノードの分類

このアルゴリズムはユニフィケーションのコストがパラメータ数に比例するとされている。

(2) 基本計算制御[1)3)]

'FGKL プログラムは，Horn Clause から成る公理と対応付けられる。このプログラムの実行は推論規則

(i) $B \leftarrow \Gamma \wedge A(\alpha)$

(ii) $A(\beta) \leftarrow \Lambda$

(iii) $B' \leftarrow \Gamma_{\wedge} \Lambda$ （'は$\alpha' = \beta'$となるようなユニフィケーション）

を繰り返し用いて，公理から仮説を証明する過程と一致する。

° input resolution

仮説の証明には，(i)公理を組み合わせて定理を次々と生成して仮説を導く方法と(ii)仮説を否定して公理に加え，この公理から矛盾する定理を導く方法がある。

さらに，背理法には，推論規則の適用において(i)少なくとも一方が公理である方法（input resolution）と(ii)少なくとも一方が単位節である方法（unit resolution）がある。'FGKL／Sにおいては，input resolution を基本的な実行メカニズムとするが，unit resolution もオプションとしておくと良い。

°実行プロセス

定理の証明過程はresolventを節点とする証明木の探索に対応させることができる。'FGKL／S では，探索をdepth first に行うこととし，パスの選択は，プログラム中に表われる節の順序やゴールの順序に依存することにする。

'FGKL のインタプリタの動作を実行プロセスの点から説明する。

例として，'FGKL プログラム

a←b∧c .　　b←e∧f .

b←g .　　c←h .

e←. f←. g←. h← .

←a .

を考える。（モジュール名およびゴールの引数は省略する）

図2.2.8(a)に示すように，初めに，プロセスP_0 が生成され，P_0 はaを節頭に持つ節を探すためプロセスP_1 を生成する。P_1 は節 a←b∧c . を見つけ，ユニフィケーションが成功したら，節体の各ゴールb，cのマッチングを行う。マッチングは左のゴールbから行われ，そのため，プロセスP_2 を生成し（図2.2.8(b)），制御を移す。このように各プロセスは，まず，ゴールとマッチする節頭を持つ節を見つけ，次に，その節体の各ゴールについて左から順にプロセスを生成する。節体のない節のとき，または，生成した

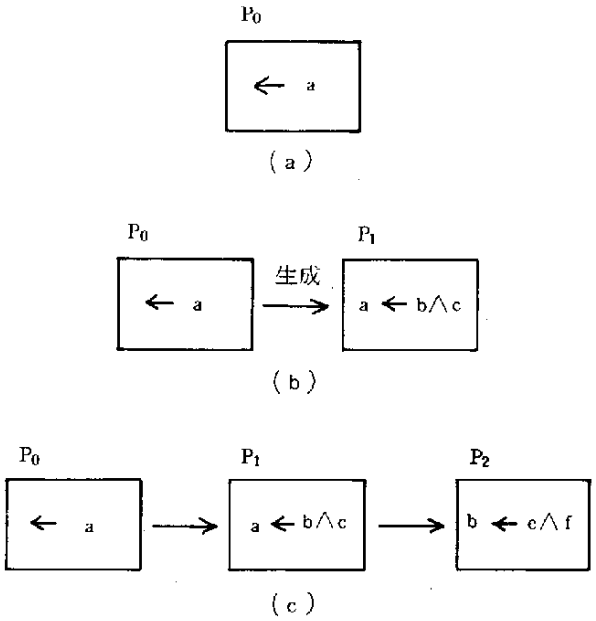


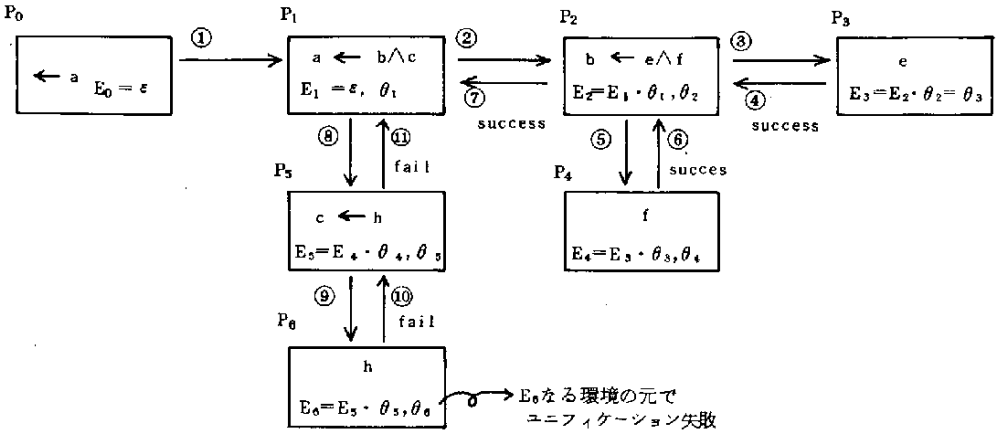
図2.2.8　実行プロセスの生成

全てのプロセスからsuccess 信号が返ったとき，このプロセスはsuccess 信号を親プロセスに返す。図2.2.9(a)において，プロセスP_2，P_3，P_4 からsuccess 信号が親プロセスに返っているのがこれにあたる。

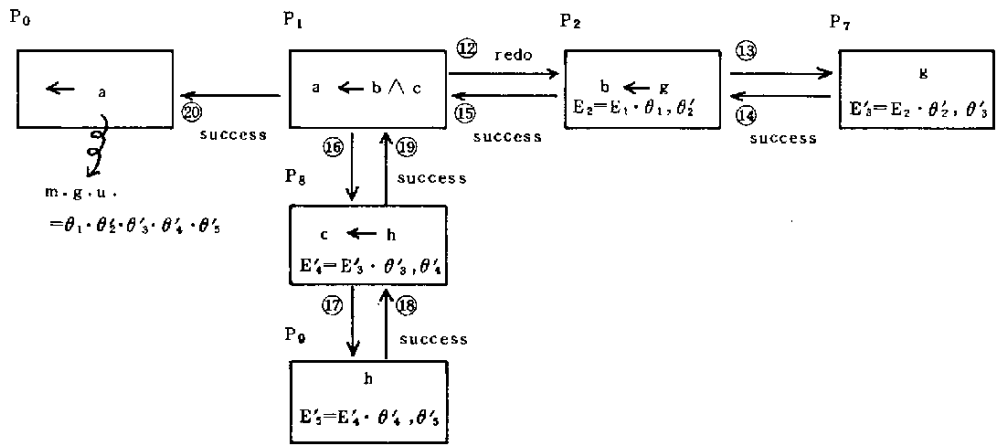
この図のプロセスP_6 のようにユニフィケーションができる節が見つからないとき，または，P_5 のように節体から生成されたプロセスが失敗したときにはfail 信号を親プロセスに返し，これらのプロセスは消滅する。注）

子プロセスからfail 信号が返ったプロセスは，その子プロセスの前に生成した子プロセ

注）ユニフィケーションができる節が存在しないとき（ゴールと同じ述語を節頭にもつ節が初めから存在しないとき）には，fail ではなく，error であり，この場合には，プロセスは，特別の処理を行う。



(a)



(b)

図2.2.9　プログラムの実行過程

各プロセスにおいて
丸数字は時間の経過。
Ei はユニフィケーションを行うための環境。
θi はユニフィケーションの結果得られたユニファイアを表わす。
(厳密にはθi の情報はそのプロセスと親プロセスに分散して記憶される)

スは，その子プロセスの前に生成した子プロセスがあるときには，その子プロセスを再実行し，他の節とのマッチングを試みる。（backtrack）図2.2.9(b)でプロセスP_2が再実行され，b←e∧f．に代わる節b←g．とのマッチングが行われている。

この例で，プロセスP_3，P_5は選ばれる節がそれぞれ1つずつしかないため，success信号を返して，制御を親プロセスに移した後，backtrackにより再実行が行われても，必ずfailとなる。従って，P_3，P_4は親プロセスにsuccess信号を返した後，消滅している。（図2.2.9(b)におけるP_0以外のプロセスも同様）

また，guarded selectionされたプロセスも，guard部を通過した後は，再実行の余地がないので，successした後でも，failした後でも消滅する。

以上に述べた実行プロセスの動作をPAD*で図5に表わす。この図では，実行プロセスを，節の選択を行うOr－プロセスと，節体の管理を行うAnd－プロセスに分けて記述した。両プロセスはAnd－Or treeにおけるOrノード，Andノードにおける処理に対応している。注)

- Concurrent Programming

Concurrentなプログラムにおいては，実行プロセスは同時に生成される。例えば，

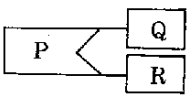
a←| p(*x^)// q(*x) // r(*x)

においては，図2.2.11に示すように3つのプロセスP_1，P_2，P_3が生成される。sequentialな処理と異なり，goalの順番は意味を持たない。

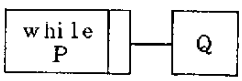
プロセスは，入力データが揃ったものだけが実行できる。注) 従って，図2.2.11の例では*xのproducerであるP_1が実行され，その後でP_2，P_3が実行される。

'FGKLインタプリタにおいては図2.2.10に示したAndプロセスを一部変更することによりconcurrentな処理が実現できる。すなわち，節体のゴールの列からOr－

注） PAD（Problem Analysis Diagram）において条件判定，繰り返しは以下のように記述されている。



if P then Q else R



while p do Q

注） この意味で，各processは，data駆動型である。

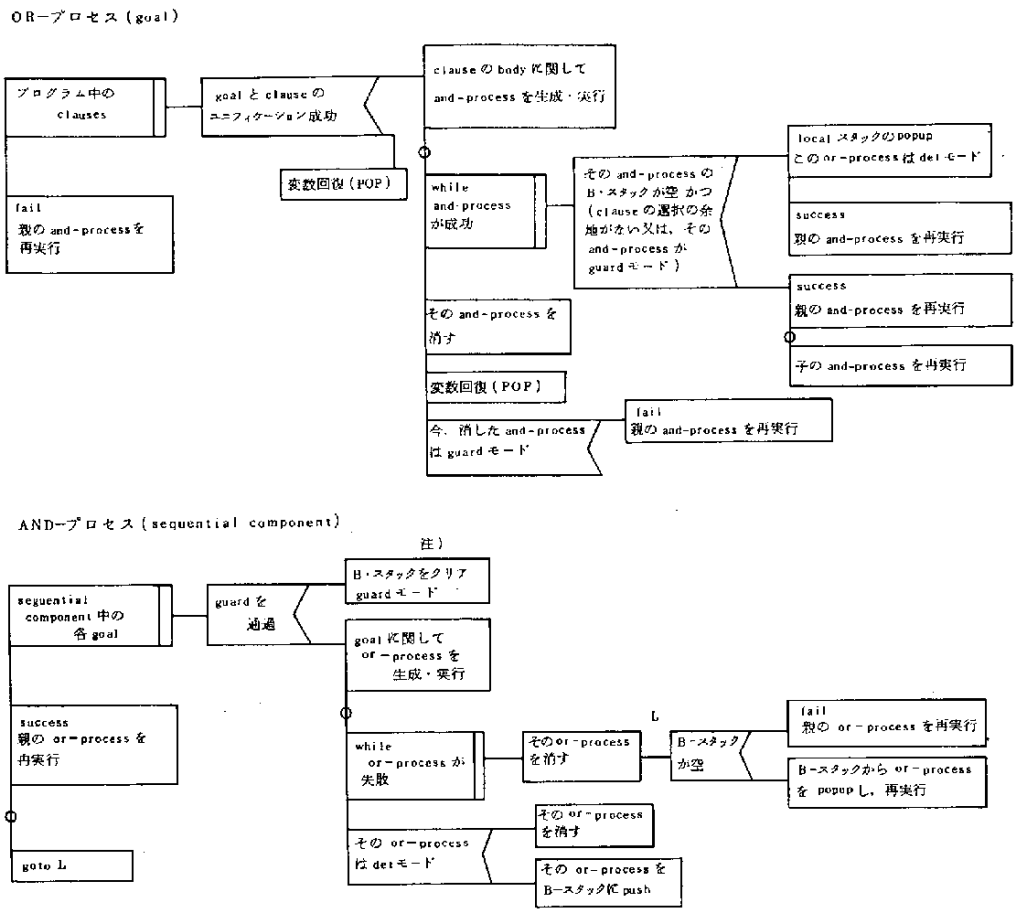


図2.2.10　実行プロセスによるPrologインタプリタの記述
（○は，制御が一時，他のプロセスに移行することを示す）

注）　B－スタックは，各And－プロセスに1つずつあり，そのAnd－Process が生成したOr－プロセスのうちで，success し，かつ，alternative のclause があるものを保持する。

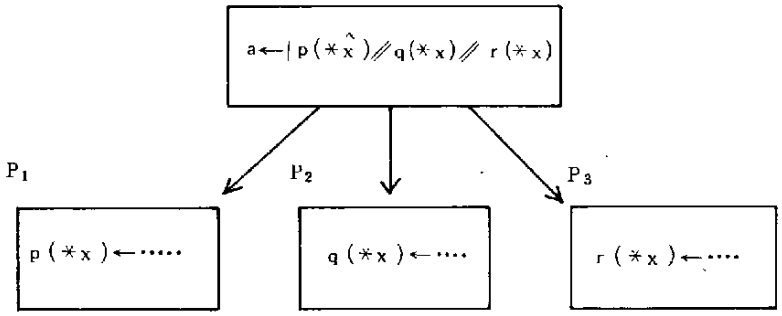


図2.2.11　Concurrent Programming

プロセスを発生するにあたり，変数の依存関係に応じてプロセスの生成の順番を決めれば良い。

stream の処理は，プロセスをコルーチンとして実行することによって実現される。

前項で述べた自然数和を求めるプログラムにおいては，図2.2.12に示すように無限のプロセスが生成される。注）この場合，integers，add に関するプロセスは，部分解が得られると親プロセスに渡す。部分データを受けたプロセスは実行できるだけのデータが揃うと即実行される。注）このようにして図2.2.12に示すようなstream 処理が実現される。また，2つのプロセスP_1 とP_2 は，P_1 が*x の producer，P_2 が*x の consumer としてコルーチンとして動作している。注）

注） コンパイルする場合には，tail recursionの最適化により，このような無限のプログラムが同時に存在するのを防ぐことが可能である。

注） 実際には，各プロセスの持ち得る入力バッファ・出力バッファの大きさにより，プロセス実行の待ち合わせが行われる。

注） 変数の依存関係によってはdead lockを生じることがある。

例えば，a←| p(*x∧ , *y)//q(*x, *y∧) のときには，pもqも実行できない。ただし，a←| p(*x∧, a：*y)//q(*x, *y∧) のように，部分的にデータが決定しているときには，p の定義によっては実行できる。

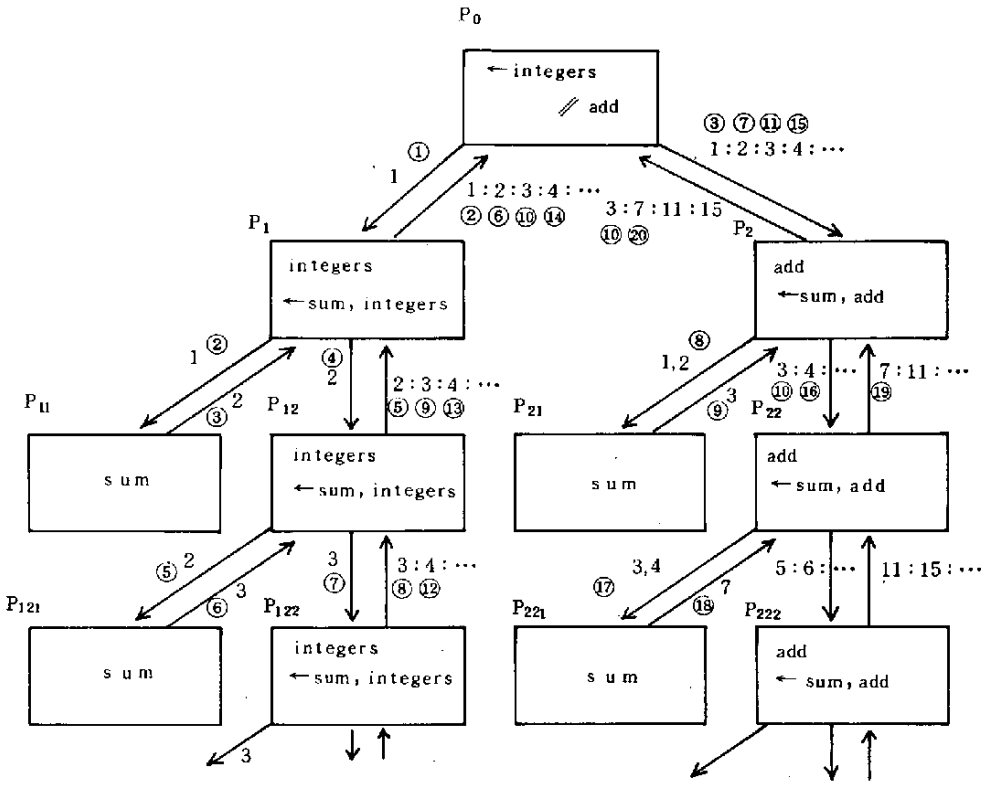


図2.2.12　Stream　の処理とコルーチン

(3)　インプリメントの技法 〔4〕〔5〕〔6〕

'FGKL/S　を効率良く実行するためのスタックの技法について述べる。

図2.2.9に示したように，ユニフィケーションの行われる環境をE_i，ユニフィケーションの結果得られた代入をθ_iとすると$E_i \cdot \theta_i$が次に実行されるユニフィケーションの環境E_{i+1}となる。また，backtrack時には環境を元の状態に戻す必要がある。

このように変数のインスタンスは後で，元の状態に戻す必要があるため，スタックで表現する。計算が進行すると，新たなユニフィケーションの結果がスタックにpush され，その情報が不要になった段階でpop　すれば良い。

スタックがpop　できるのは

①　backtrackが起こったとき

②　プロセスが再実行の余地なくsuccess 終了したとき*（図2.2.9(a)におけるP_3，

注)これをdeterminate successとよぶ

P_4 など）

の２つの場合である。①の場合，捨てられる情報は他から参照されることはない。②の場合，捨てられる部分が他から参照されていないことが保証されているときのみpopできる。

スタックに値を記憶するのに，(i)構造を共有する方法（ structure sharing ; ss）と(ii)構造をコピーする方法（ copy ）がある。

° structure sharing

この方式では，スタックに記憶される情報は次のようになる。

インスタンス	スタックに記録するもの
変数 注)	その変数のスタックにおけるアドレス
定数	その定数の値
tuple	heapにあるプログラム中のtuple へのポインタと，環境へのポインタの対

この方式では，その節でtuple 中に表われる変数（ global変数）をglobalスタックで，それ以外の変数（ local 変数）をlocal スタックで管理する。

例として，次のプログラムにおけるスタックの内容を図2.2.13に示す。

append(*w.*x, *y, *w.*z) ← append(*x, *y, *z)

append(<>, xp, xp) ←.

← append(a.b.<>, c.d.<>, *r).

この図で*r に関する参照のchain を辿ることにより，解a.b.c.d.<> が得られる。脚注により，local スタックにおいては，低位アドレスから高位アドレスへの参照はないため，再実行の余地がないプロセスにおけるユニフィケーション情報（図2.2.13ではP3 ）がpop up できる。trail は backtrack時に未定義に戻すべき変数を記録している。

注） 変数$*x_0$ がゴール中に表われ，変数*x が節頭に表われたとき，ユニフィケーションの処理は４つの場合に分かれる。

(i) $*x_0$: local　*x : local　　*x ← address ($*x_0$)

(ii) $*x_0$: local　*x : global　　$*x_0$ ← address ($*x_0$)

(iii) $*x_0$: global　*x : local　　*x ← address ($*x_0$)

(iv) $*x_0$: global　*x : global　　*x ← address ($*x_0$)

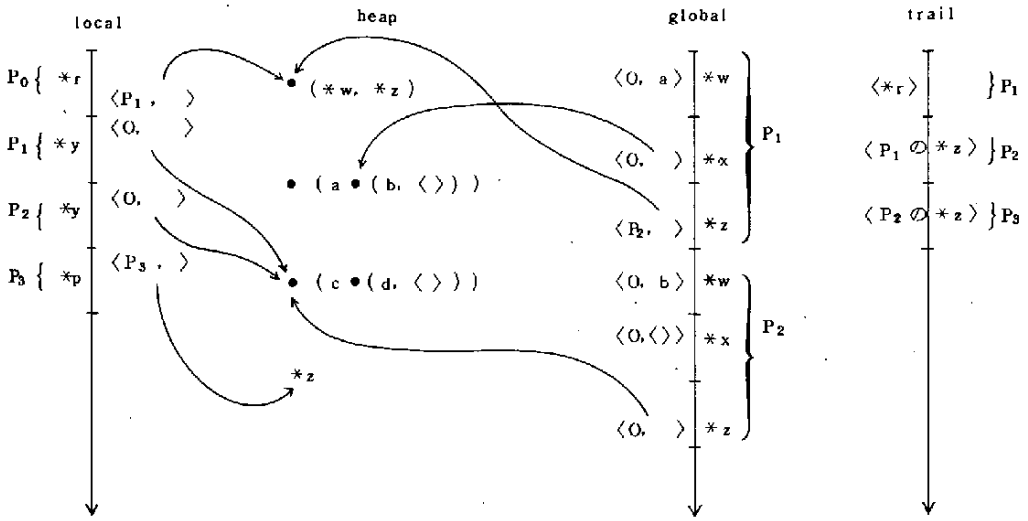


図2.2.13　Structure Sharingにおけるスタックの内容

◦ copy

この方式では，スタックに記憶される情報は次のようになる。

インスタンス	スタックに記録するもの
変数 注)	その変数のスタックにおけるアドレス
定数	その定数の値
tuple （未定義変数を含む）	そのtuple のコピーへのポインタ
tuple （未定義変数を含まない）	heapにあるプログラム中のtuple へのポインタ

tuple のコピーはheap（またはコピー用スタック）上に作られ，図2.2.14(a)に示すようにtuple 中の未定義変数へ，スタック中の対応する変数cell からポインタが付けられる。この変数の値が決まるとポインタが除去され，逆にコピー中の変数cell からこの値へのポインタが付けられる。（図2.2.14(b)）backtrack時に環境を元に戻すた

注）$*x_0$ ，*x がそれぞれ，ゴール中に表われた変数，節頭に表われた変数とするとユニフィケーションの結果は，次のようになる。

$$*x \leftarrow address(*x_0)$$

めには，コピー中の変数cell へのポインタを記憶しておく必要がある。（図2.2.14における trail 2）

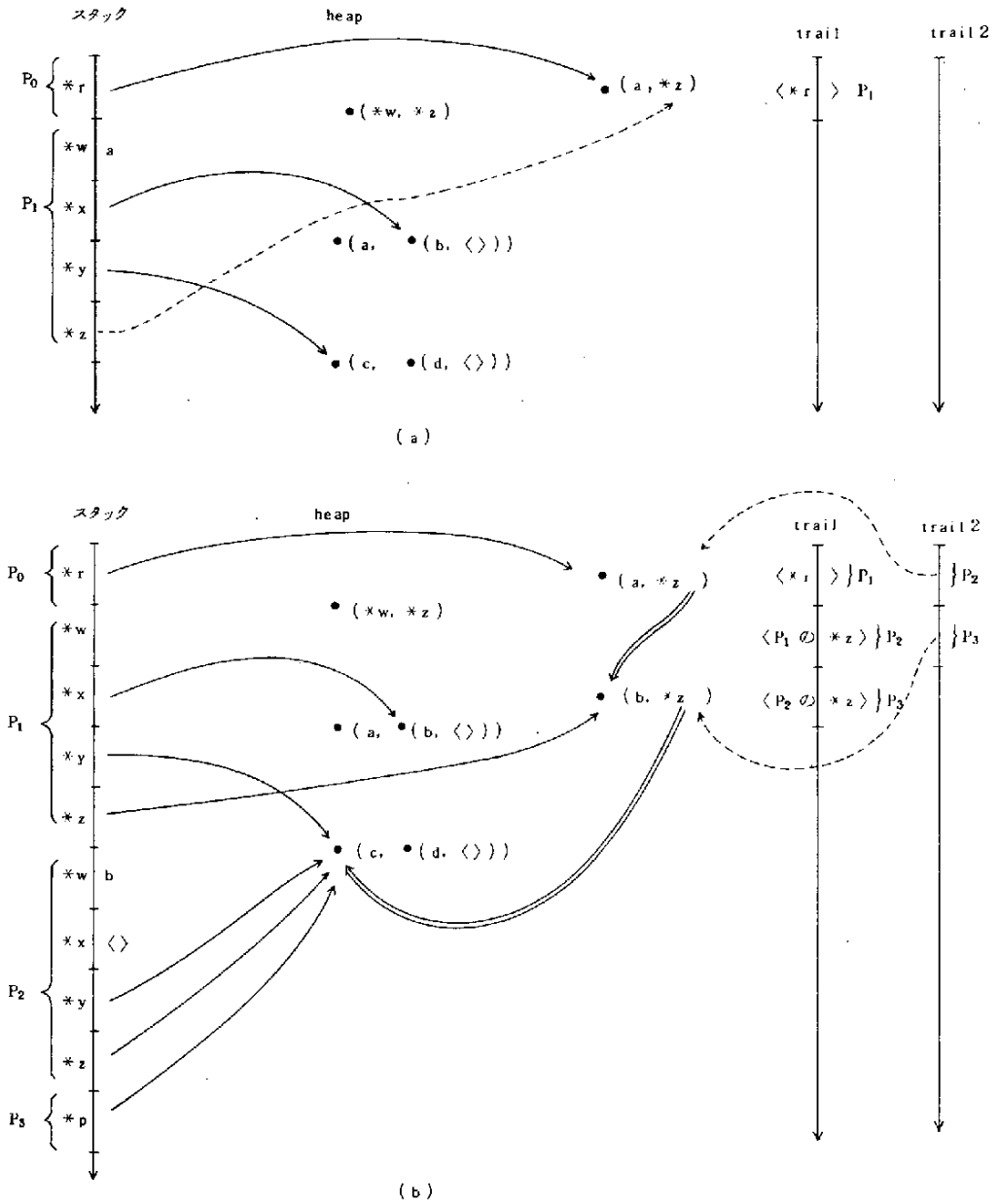


図2.2.14　copy 方式におけるスタックの内容

また，copy 方式の別の方法として，tuple は必ずコピーする（未定数変数の有無を問わない）ことも考えられる。（図2.2.15）

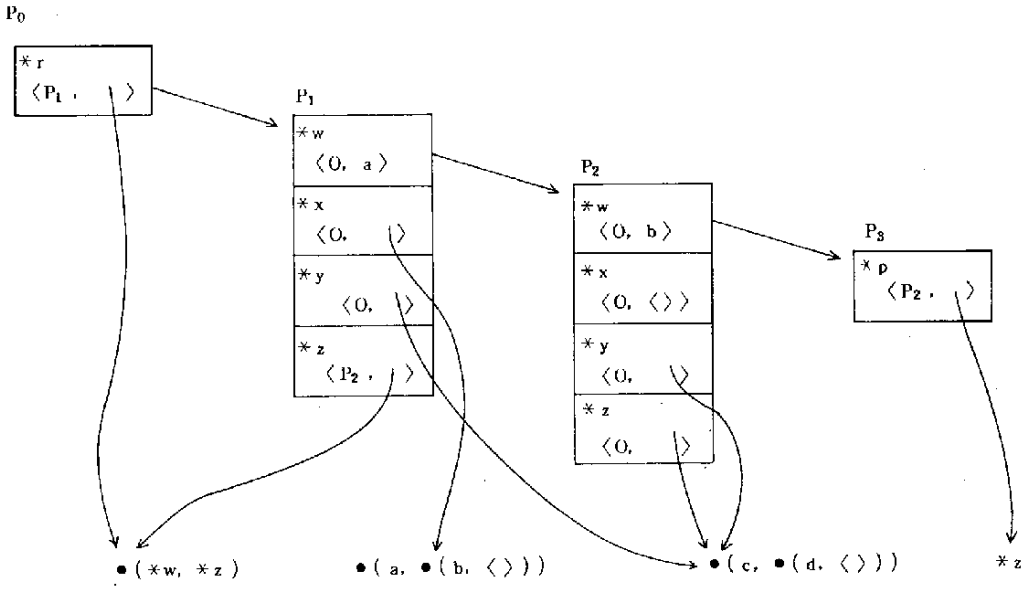


図2.2.15　copy 方式におけるスタックの内容（続き）

プログラムの性質によってどの方式が有効かは異なる。

(4) ハード化ユニフィケーション

ユニフィケーションをハード化する場合の基本命令について述べる。

① レベル0

deref(any)

引数が変数ならばreference の chainを辿って，その値（定数かtuple ）を出力する。未定義変数ならばchain を辿ったときの最後の変数のアドレスを出力する。

copy(any, sp)

引数のコピーを作る。この中に変数があるときには，その値を求め（スタックポインタspを用いる），その値で変数を置き換える。変数の値が未定義ならば，スタック中のその変数cellからコピーの変数cellへのポインタを保持する。

varp(any)

引数の中に変数が含まれているか否かを出力する。

type(any)

引数の種類（定数，変数，tuple）を出力する。

② レベル1

bind(unbound variable, any, sp)

未定義変数を特定の値に結びつける。第2引数の種類（定数・変数・tuple）により処理が異なる。[注)] このunbound variableが節頭中でなく，ゴール中に発生したものならば，そのアドレスをtrailに記録。

atom−unify(atom/integer, any)

定数と第2引数とのユニフィケーションを行う。まず，第2引数の値を求め，第1引数との比較を行う。第2引数が未定義変数ならば，bindが呼ばれる。

var−unify(var, any)

変数と第2引数とのユニフィケーションを行う。まず，第1引数の値を求め，未定義か，定数か，tupleかによって，bind，atom−unify，tuple−unifyが呼ばれる。

tuple−unify (tuple, tuple)

tupleと第2引数とのユニフィケーションを行う。まず，両tupleの成分の数を比較する。一致していたら，tupleの各成分について，その種類（定数，変数，tuple）によって，atom−unify，var−unify，tuple−unifyが呼ばれる。

frame−list

'FGKL/Sマシンでは，スタックをハードウェアで取り扱う必要がある。しかし，前に述べたように，スタック内のセルが他から参照されているときには，backtrack時にはpop upできるが，determinate success時にはpop upできない。structure sharing方式ではlocal変数とglobal変数を区別したのはこの欠点

注）structure sharing方式，copy方式における処理は前に述べた。

を改善する一つの方法であるが十分でない。[6] 'FGKL/S マシンとしては，図2.2.16に示すように単なるスタックではなく，frame （1つの節に含まれる変数のcellから成るvector）をリストで結んだ構造が有利である。

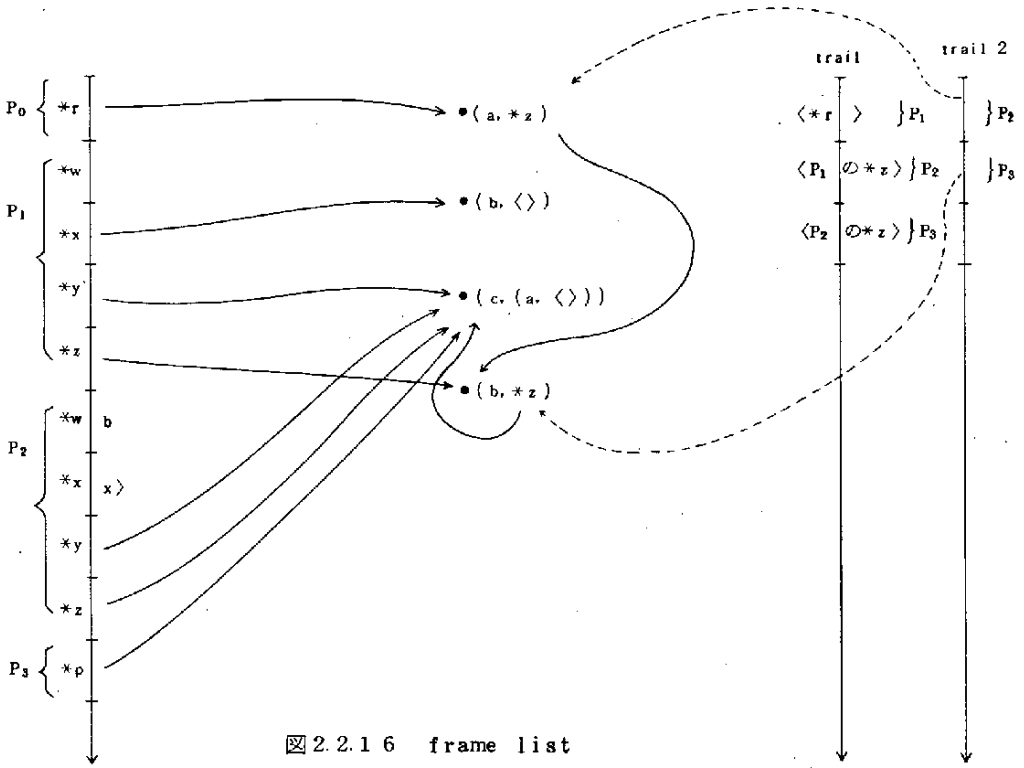


図2.2.16 frame list

このようなframe listにおいては，他から参照されていないframeは，それがスタックの最上部でなくとも削除できる。従って，determinate success 時だけでなく，必要なときにはいつでも，ガーベジコレクションによって他から参照されていないframeを削除し，メモリの有効利用ができる。また，local，global の区別も不要である。

(5) ハード化計算制御

'FGKL/S マシンでインタプリタをハード化するための基本命令について述べる。

① OR－プロセス関連

select－clause(goal)

goalと同じprincipal elementを持つ節を選択する。

create－and(sequential－component)

AND－process を生成し，制御を移す。

delete－and(process)

AND－process を消す。

pop(sp)

スタックのpop up

redo(process)

AND－process を再実行させる。

② ＡＮＤ－プロセス関連

select－goal(sequential－component)

sequential component から，goal を左側から１つずつ選ぶ。

create－OR(goal)

ＯＲ－process を生成，制御を移す。

delete－OR(process)

ＯＲ－process を消す。

push－B(process)

ＯＲ－process をＢ－スタックにpush

pop－B

Ｂ－スタックからＯＲ－process を１つ Pop して再実行させる。

redo(process)

ＯＲ－process を再実行させる。

○ プロセスを支援するハードウェア

プロセスの生成，消滅にはプロセス管理用のスタックが必要であり，このスタック中に各プロセスにおけるレジスタ群等の情報が記憶される。これを３台のプロセッサにより，高速実行できる。

すなわち，あるＯＲ－プロセスが実行されているとき，次に実行されるＯＲ－プロセスは，そのプロセスがsuccess したときに実行されるプロセスか，fail したときに実行されるプロセスのいずれかである。従って，プロセスの実行中に，次に実行される可能性のある２つのプロセスのレジスタ群を先行回復することにすれば，速度の向上が期待される。

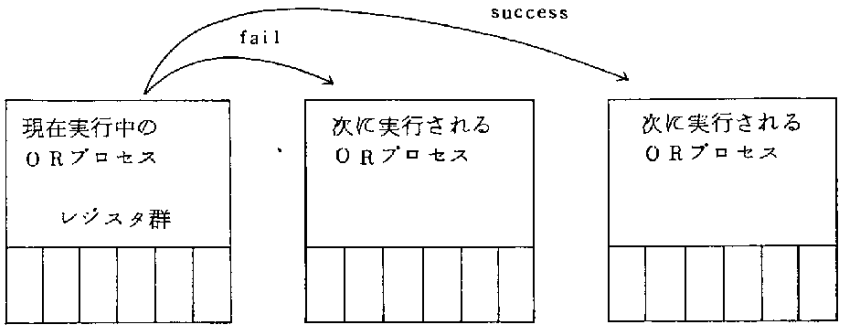


図2.2.17　レジスタ群の先行回復

— 参考文献 —

1) C－L. Chang and R. C.－T. Lee

"Symbolic Logic and Mechanical Theorem Proving"

Academic Press, 1973

2) M.S. Peterson

"Linear Unification"

3) 後藤滋樹

"述語型プログラミング言語DURALとその処理系"

4) D.H.D.Warren

"Implementing PROLOG"

D.A.I. No.39～40 University of Edinburgh, 1977

5) M. Bruymooghe

"The memory management of PROLOG implementations"

Logic Programmmg Workshop 1980

6) C.S.Mellish

"An alternative to structive sharing in the Impremementation

of A PROLOG INTERPRETER"

Logic Programming Workshop, 1980

2.2.3 拡張論理式と拡張計算メカニズム

プログラミング言語としての使いやすさと，その言語の論理的基盤の確固さとを両立させることは必ずしも簡単ではない。それにもかかわらず，本節では，プログラミング言語 'ＦＧＫＬ／Ｓに対するいくつかの拡張とその拡張機能の意味づけ，拡張の論理的実現法について述べる。

拡張を必要とする理由は，ＫＩＰＳの種々の目的にかなう言語機能を提供することが核言語に要求されており，基本論理式とその計算メカニズムのみではそれらの目的にとって必ずしも十分でない点があるからである。

ここで述べる拡張機能は（基本計算メカニズムの拡張等によって）いろいろな計算メカニズムによりインプリメントすることができるであろう。しかし現状は，拡張機能そのものの取捨選択についてまだ多くの議論が必要な段階であると思われる。それゆえここでは，具体的実現方法について触れることなしに可能な拡張機能について述べることにする。

A　拡張概念について

'ＦＧＫＬ／Ｓに対する拡張のｎｅｅｄｓは多岐にわたるが，次のものが代表的である。

(i) 述語変数，述語間演算子

(ii) 名前のスコープ制御

(iii) 実行環境の多重化，変更

(iv) 節選択順序の制御

(v) バックトラックの制御

(vi) パタンマッチの制御

(vii) マクロ機能

(viii) 引数渡しの方法

一方，拡張は，2.2.2基本論理式と基本計算メカニズムと対比していえば，次の３つのクラスに大別できるであろう。

(L) 言語機能（基本論理式）の拡張

(R) Ｒｅｓｏｌｕｔｉｏｎ法（基本計算メカニズム）の拡張

(M) 意味領域の他次元への拡張

この分類に強いて従えば，上述の(i)~(viii)は表 2.2.1のように分類される。

表 2.2.1　拡張の分類

	(L)	(R)	(M)
(i)	◎	○	
(ii)	◎	○	
(iii)		○	◎
(iv)		◎	
(v)		◎	
(vi)		◎	
(vii)	◎	○	
(viii)	◎	○	

◎　主な拡張

○　2次的に生じる拡張

他方，needs に対する seeds の立場から，次の6つの主たる実現方法を考えることができる。

(a)　ホーン性を越えるロジカル・コネクティブの導入

(b)　高階概念（高階論理式）の導入

(c)　ラムダ抽象化記法の導入

(d)　様相概念（様相演算子）の導入

(e)　メタ概念の導入

(f)　よりProceduralな意味論の導入

すると，上述の(i)～(viii)は，(a)～(f)を，例えば，表2.2.2のように用いることによって直接的，間接的に実現可能となる。

表 2.2.2　実現方法

	(a)	(b)	(c)	(d)	(e)	(f)
(i)		◎	△		○	
(ii)		◎	△		◎	
(iii)				◎		
(iv)	◎	◎	△		◎	◎
(v)					○	◎
(vi)					○	◎
(vii)					◎	○
(viii)					○	◎

◎　直接的な実現

○　間接的な実現

△　補助的に必要

(a)と(b), (c), (d), (f)は言語の意味領域に対する拡張を基礎とし, (e)は言語自身の拡張であり, 言語の意味化, 換言すればメタ言語の対象言語化を基礎としている。しかし, (f)は, はっきりした論理的基礎がまだ与えられてはいない。

意味領域に対する拡張法では, 言語とその意味とを全く異なる対象として扱うことを原則としており, プログラムをデータとして扱う方法を(原則として)提供しない反面, プログラムとデータとの混合に起因する誤りを生じない。一方, 言語の対象化は, プログラムによってプログラムを生成できる機能を与えるため, プログラム作成の量的困難を回避するのに役立つが, 本来の意味でのプログラムとデータの混同から生じる誤りの可能性を増大する要因ともなる。

形式的論理体系における言語の対象化は論理的に不完全であらざるをえず, これゆえ高度のプログラムにおいては言語の能力の限界についての正確な認識が要求される場合もあることに注意しなければならない。

ここに述べる拡張のうち, 高階の概念およびそれに伴う言語の拡張であるラムダ抽象化はメタ概念の対象化によってかなり一般的に実現し得ると考えられる。(注)

本節2.2.3の構成は, B～Fまではseeds サイドから概念の説明と例題を挙げている。後半G～Mは, いくつかのseeds 概念が複合しており, しかも, needs として代表的と思われるものについてneeds サイドから個別に列挙している。

B ホーン性を越えるロジカル・コネクティブの導入等

ホーン節の制限をゆるめ非ホーン節による表現をゆるすものである。

次式で定義される関数fを'FGKL風に述語化して表現してみよう。

f(x) ⇦ if x=0 then 1 else 2

例1. ロジカルコネクティブ not

f(0, 1) ←

f(*x, 2) ← not(*x=0)

このnotの引数はsequential component であり, 従ってnot はロジカル・コネクティブとみなされる。(高階概念, メタ概念のnotを参照のこと。) 引数が真のとき偽を引数が偽のときは真を値とする。

例2. ロジカルコネクティブ ifthenelse(または if)

f(*x, *y) ← if(*x=0, *y=1, *y=2)

ifは3引数でsequential component をとる。第1引数が真のとき第2引数をif全

注)例えば, 2.2.3 F例3. Demo あるいは, 文献3)を参照のこと。

体の結果とし，第1引数が偽のとき第3引数をif 全体の結果とする。ここで，第1引数が真のときの第3引数，第1引数が偽のときの第2引数は必ずしも値が定まらなくてもよいものとしておく。

例3. 非ホーン節

例1はさらに次のように書き直すことができる。

(a) f (0 , 1) ←

(b) f (＊x , 2) ∨ (＊x＝0) ←

上記(b)の節でユニフィケーションの際に対象となるのはf (＊x , 2)であって，f (＊x, 2)とのユニフィケーションが成功でしかも(＊x＝0)が偽であるとき，(注)第2節の節頭とのマッチングが成功する。この例では現われていないが，もし節体があればそれを実行する。

fの定義体の中では，呼出しに対するマッチングはfをprincipal elementとする項に対してのみ試みられ，節頭に現われているそれ以外のリテラルはユニフィケーションの可否を判定するための付帯条件とみなす。

例3をsyntactic sugarを用いてわかりやすく書き直すと次のようになろう。

f (＊x , 2) unless ＊x＝0 ←

f (0 , 1) ←

unless部分は2.2.2.A. guarded componentのguardに似ているが，unlessはユニフィケーションの制御であるのに対して，guardは手続きの(バックトラックの及ぶ)本体であると考えられる。

C 高階概念の導入

ここでは高階論理の変数と関数，述語を導入する。これら高階プリミティブの利用に役立つラムダ抽象化記法については，Dで触れる。

2階以上の述語論理では，述語変数，関数変数，述語の述語，関数の関数等が現われる。高階概念のうち，高階述語および高階述語変数は述語論理型言語の基本的形式に比較的良く整合するが，高階関数，高階関数変数は基本形式におけるスコーレム関数とは異なりevaluableでなければほとんど実用的に意味を持たない。従って，(スコーレム関数を扱う)基本形式に忠実であろうとするならば，高階関数でなく高階述語を採用するのが妥当であろう。しかし，スコーレム関数として高階関数を用いることはまず考えられないので，高階関数はすべてevaluableとする，と割切って，システムでいくつかのものを提供することが考えられよう。

注) 2.2.3.E 例1. notの用例を参照のこと

他方，（2階以上の）高階述語論理では完全な証明系が存在しないことが知られており，導入された高階概念をカバーする適切な証明系を見い出すことは容易でない。また，（逐次型）ユニフィケーションにおいては，理論的な限界として二階以上のタームについては決定不能であることが知られている。従って，実際の処理系作成では便宜的な技法を用いることになろう。

(i) （高階）関数，（高階）述語の具体的表現技法として，ＬＩＳＰ言語のラムダ関数表現の採用

(ii) 様相概念，メタ概念による等価なあるいは近似的な実現

などが考えられる。

例えば，(a) 述語変数に対してはその時点で定義されている述語（定数）全体から一つずつ選び出してはユニフィケーションを試みる。(b) 関数変数については（evaluableな）関数（定数）全体から一つずつ選び出してはユニフィケーションを試みる。(c) 高階述語（定数），高階関数（定数）は（予め定義されている関係，手順等に従い）等価な（通常は一階の）表現に置き換えて実行する。即ち，述語を表わす式e（p，q，r，…）について，

e（p，g，r，…）（δ）は，

e′（p（δ），q（δ），r（δ），…）と翻訳し直して実行する。ここで，eは高階述語，p，q，r，…は述語，δは述語式全体への実引数，e′（…）は等価な1階の表現をそれぞれ表わす。

例1. 高階関数　not　（その1）

notは（高階）述語から（高階）述語への関数である。2.2.3.B. の例1.は次のように書き表わされる。

f（0，1）←

f（＊x，2）← not（＝）（＊x，0）

notの働きにより，not（＝）は2引数述語≠を表わすことになり，引数が等しいとき偽，等しくないとき，真を与える。なお，not（＝）の部分は'ＦＧＫＬ／Ｓ（resolution法）のユニフィケーションレベルで処理するのは困難である。

例2. 高階述語 not（その2）

notは2つの述語を引数とする述語で，2つの述語が互いに否定の関係にあることを表明する。これによれば，上述の例1は次のように書ける。ここで＊neqは述語変数である。

f（0，1）←

注）＊x＝0は　prefix形　＝（＊x，0）の省略形であることに注意

f (＊x, 2) ← not (=, ＊neg) ∧ ＊neg (＊x, 0)

例1の高階関数よりこの高階述語の方がresolution 法の形式的枠組を変えることが少ない。しかし，実現上の困難は依然として残る。

例3. 高階関数 if1, if2

(i) α, β, γが(高階)述語のとき，if1(α, β, γ)は(高階)述語である。即ち，

if1(α, β, γ) (δ) と，if $\alpha(\delta)$ then $\beta(\delta)$ else $\gamma(\delta)$

は等価である。

2.2.3.B. の例2はラムダ抽象タームを用いて次のように書き直すことができる。

f (＊x, ＊y) ←

if1(λ＊u. λ＊v. (＊u＝0),

λ＊u. λ＊v. (＊v＝1),

λ＊u. λ＊v. (＊v＝2)) (＊x, ＊y)

(ii) αがsequential component, β, γが(高階)述語である　き，if2(α, β, γ)は(高階)述語である。即ち

if2(α, β, γ) (δ) と

if α then β (δ) else γ (δ)

は等価である。

用法は(i)の場合と同様である。

例4. 高階述語 apply

αを(高階)述語，βをタームβi または(高階)述語βi (1≦i≦n)のリスト β1. β2. ……・βn. <>とするとき，apply (α, β)は引数β1, β2…, βnにαを作用させた結果(真または偽)を返す。 即ち，

apply (α, β1. β2. …. βn.<>) ←

α(β1, β2, …, βn)

例5. 繰返し forall

forallは2引数高階述語である。2つの引数は，引数の個数が同じの述語あるいは述語を表わす(一般にはラムダ抽象化されている)表現で，自由変数を含んでいてもよい。この実行は次の例で示されるとおりとする。

自由変数＊yと＊z，＊wには値η, ζ, ωがそれぞれユニファイされているものとする。このとき，

forall (λ＊x. p(＊x, ＊y, ＊z), λ＊z. q (＊x, ＊y, ＊w))の実行がsuccessするのは，p(ξ, η, ζ)がsuccessするようなすべてのξに対して，q (ξ,

η，ω）もやはりsuccessするときである。

この繰返し実行は，メタ概念のforall 2を用いて

forall 2（＊x，p（＊x，＊y，＊z），q（＊x，＊y，＊w））

とした方がわかりやすいと思われる。ただし，第1引数の＊xが走査のための変数であることを示しているものとする。

他方，別な表現として，

forall 3（p（＊x，＊y，＊z），q（＊x，＊y，＊w））が考えられる。これではどの変数がforallで走査されるのかがわからない。別の解釈としては，実行時点で，定数にユニファイされていない変数すべてを対象として走査することも考えられる。注）

例6. 関係の閉包を与える高階述語closure　rを2引数（高階）述語とするとき，closure（r）は2引数（高階）述語であって，次式で定義されるようなcと等価である。

c（δ，ε）← r（δ，ε）

c（δ，ε）← r（δ，ξ）$\wedge$ c（ξ，ε）

例7. 高階関数　map

αを（高階）関数または（高階）述語，βをリストβ1．β2．…βn．<>　とするとき，map（α，β）はリストr1．r2．….rn．<>　を与える。ここで，riはα（βi）の結果である。

このmapは高階述語map′の形で導入しておくことも考えられる。

map′（α，β，r）はr＝map（α，β）ということを表明している。

D　ラムダ抽象（lambda abstraction）

ラムダ抽象は，ラムダ計算と同様にして命題（述語）の（言語レベルの）表現から述語の（言語レベルの）表現，タ－ム（関数）の（言語レベルの）表現から関数の（言語レベルの）表現をつくり出す。

例1. grandparent

λ＊x．λ＊y．parent（＊x，＊z）$\wedge$ parent（＊z，＊y）

＊xは＊yの祖父母grandparentであることをparentを用いて定義している。

ラムダ抽象は，一般に，例えば　高階述語（高階関数）の実引数として述語や関数を表わすために用いる。特に実行時に動的に定まる関数や述語をあらわすのに不可欠である。

ラムダ抽象は，高階概念を表わすのに適した言語機構であり，高階述語や高階関数を自由に

注）Prolog／KR[5]を参照のこと

扱うためには次例のように不可欠の言語要素である。

例2.

← minus (3, 2, *u) ∧
apply (λ*x. λ*y. plus (*x, *u, *y), 0. *w. <>)

このゴール節に対して，*u＝1，*x＝0，*w＝*y＝1 となる。ここで apply は 2.2.3.c の例4を参照のこと。

例2において，ラムダ抽象を用いた述語 λ*x. λ*y. plus (*x, *u, *y) に対応する次の表現は正しくない。

addu (*x, *y) ← plus (*x, *u, *y)
← minus (3, 2, *u) ∧ addu (*x, *y)

なぜなら，両方の節の*u が同一のものとみなされないからである。(2.2.3 H例3も参照)。

ラムダ抽象を用いた述語，関数は，便宜上，(どの一部分もβ変換できない) 標準形であると仮定する。このような述語，関数のユニフィケーションは，(従来どおりの) ラムダ表現の意味が等しくなるように拡張される。

(i) ラムダ束縛変数とラムダ束縛変数とだけユニファイできる。

(ii) ラムダ表現全体と変数はユニファイできる。

例3. λ*x. p (*x, *u) と λ*y. p (*y, *y) はユニファイできない。

例4. λ*x. p (*x, *u) と λ*x. p (*x, 3) は*u＝3でユニファイできる。

例5. λ*x. p (*x, *x) と λ*y. p (*y, *y) はユニファイできる。

(iii) ラムダ表現は，高階述語や高階関数の引数として用いられ，引数が適用されたときには常に標準形になるようにラムダ変換 (特にβ変換) が行なわれるものとする。(このことは，ラムダ表現が evaluable な性質を持つこと，及び，ユニフィケーションアルゴリズムの変更が必要なことを意味する。)

例6. λ*x. p (*x) と p とはユニファイできる。

E 様相概念の導入

今までは，意味解釈の領域 (世界) が唯一つであった。ここでは，意味解釈の領域が複数であることを仮定する演算子 (様相演算子) を考察する。

Prolog においてバックトラックの制御に用いられた cut 演算子は，様相演算子と考えられる。cut を用いた not の定義

(a) not (*p) :－*p, !, fail

(b) not (*p).

において，(a)式で　*pがsuccessし，cutが実行されたあとはnotの定義全体が次のように変わったのと同じ効果を与える。

(a')　not (*p) : − fail .

(b')　(対応なし)

即ち，cutによって「世界」が変わったととらえることができる。

つぎに，様相概念を直接言語機能に含む例をあげる。

例1.　Strong Provability[4)]

述語型言語ＤＵＲＡＬでは２つの節定義集合を持っている。通常はこれら両方を合わせ用いて計算を進めるが，様相演算子Ｓが前置されたゴールは「高速版」と定義された節定義集合だけを用いて計算を進める。

証明（計算）に用いる公理（節定義）が少なくなるので Strong provabiliy と呼ばれる。

例2.　EXecute[4)]

同じくＤＵＲＡＬでは，様相演算子ＥＸの前置されたゴールは，Ｌisp言語のexpressionとみなして，Ｌisp世界で評価される。その結果がｎｉｌであるときそのゴールは偽(failure)であるとみなし，そうでないときそのゴールは真（ success ）であるとみなす。

F　メタ概念の導入

本来は，言語構成要素である節やsequential componentをあたかも操作対象のようにして扱う述語や関数をメタ述語，メタ関数という。従って，システムの自己記述等に用いることができる（例３参照のこと）。

例1.　メタ述語　not[注)]

αをsequential componentとするとき，not（α）は次のような意味を持つ。

実行時（証明時）にαが（それまでのユニフィケーションの結果変数に値が割当てられて）α'になっているとする。その時点で定義されている節全体をΣとするとき，Σからα'が証明不能のとき（即ち，$\Sigma \nvdash \alpha'$）[注)]

not（α'）はsuccessとする。そうでないときfailureとする。α'中に変数が現われている場合には注意を要する。

例えば，次のように２つの節が定義されているものとしよう。

p (a) ←

注）2.2.2.A．基本論理式で示されたnotはメタ概念とみなされる。

注）　ここでは，有限時間内に証明不能か可能であるかが決まるものと仮定しておく。

r (*x) ← not (p (*x))

ゴール節←r (b) はsuccessであるが，ゴール節←r (*z) はfailureとなる。他方，通常の本来のロジカル・コネクティブのnotの意味としては（存在すれば）a以外の*z＝b，c，d，……なる解が得られる筈である。これは，not (p (*z)) を，¬(∀z p (z)) としてでなく∀z (¬p (z)) として証明（計算）を試みていることに起因している。

例2. メタ述語 if

αを sequential component，β，τを節とするとき if (α，β，τ) は節を表わす。これは次に示すように節の選択に用いる。

p (*x，*y) ← if (*x＝0，
q (*z，1) ← ，
q (*z，2) ←)$_{\wedge}$
……∧ q (*y，*u)∧……

*x＝0のときは，qは，q (*z，1) ← として定義され，そうでないときは，q (*z，2) ← として（局所的に有効でしかも）動的な節定義を与える。

なお，これは前述の様相制御ともみなされる。

例3. メタ述語 Demo[3)]

Σを定義節集合，α を sequential component とする。αがΣから証明できる（$\Sigma \vdash \alpha$）とき[注)]，Demo (Σ，α) は真を与える。そうでないときは偽を与える。

例4. メタ述語 setof[1)]

αを変数，βを変数αが（自由に）現われている sequential component，τを集合（を表わすリスト）とする。setof (α，β，τ) はβを満たすような変数αの実現値の集合がτであるとき真となりそうでないとき偽となる。

上述の Demoと setofを用いると，←literal (*c)$_{\wedge}$setof (*x，clause (*x)，*s)$_{\wedge}$Demo (*s，*c)

によって，*cにはシステムから演繹できるすべてのリテラル型の定理が得られることになる。このDemoを用いてnot等が定義できる。

not (α) ← Demo (Σ，α) | failure

not (α) ← |

注）ここでは，有限時間内に証明可能か不能かが決まるものと仮定しておく。

G　マクロ機能

マクロ機能は，節やsequential component 等の言語要素から，別の言語要素を与える点でメタ概念である。しかし，マクロ処理が処理対象となっている言語それ自身（の一部）を用いて記述されるため（即ち，メタ言語レベルとオブジェクト言語レベルとが明確に区別されなくなるので），注意が必要である。

また，マクロ処理によって得られた言語要素はさらに実行の対象にする必要がある。[注)]

ここでは，マクロ定義を次のように与える。

macro（α，β）← τ

ここで，αは節頭のパタン，βは節体のパタン，τはαからβを得る処理を表わす節体である。α，β中にはいくつかのパラメタ（変数）が含まれており，これらの変数には次に述べるマクロ呼出しとの間のユニフィケーションにより値が与えられる。

マクロ呼出しは，マクロ定義で与えた節頭パタンの実例をgoalとして用いることにより行われる。

……$\wedge^{\alpha'}\wedge$……

α'は節頭パターンの実例

呼出しの実行は，まずマクロ呼出しα'とマクロ定義の節頭パターンαとの間でユニフィケーションが行われる（いくつかの変数に値が与えられる）。[注)] 次いでマクロ定義の節体τが実行されて節体βの実例β'が求められる。最後にこのβ'を実行してその結果がマクロ呼出しα'の結果となる。[注)]

なお，β'中の自由変数の有効範囲（scope）はβ'の中に限定され，α'から持込まれた同じ綴りの変数とは区別する。

また，α'とαとのユニフィケーションにおいて，両者の対応する位置でα'側が変数νでα側が定数ρのとき，マッチングが起きないように抑止するために，特にα側を$'\rho$と表わすものとする。（この'は2.2.3.1例1で述べられる。）

例1.　繰返し　while

pを述語，sを（恒真）述語とするときpが真である間sを実行するような述語while（p，s）はマクロ機能を用いて次のように表わせる。

注）　基本計算メカニズムそのものの変更や拡張を必然的に伴なう。

注）　通常のユニフィケーションとは異なっていることに注意

```
macro ( while ( *p, *s ),
        if  *p  then  *s ∧ while ( *p, *s )
          else  success          ) ←
```

H　名前の有効範囲の制御

2.2.2.A．基本論理式で述べたように，述語名については，global 宣言と，モジュール名の前置とにより，モジュール間を越えた述語の参照が可能である。他方，節で用いられる変数の有効範囲はその節の中だけである。このため，いわゆる大域変数への代入参照ができない。ここでは，Algol 流のブロック構造を導入する。[6)] 注）

例1.　節定義の入れ子構造

```
p 1 ( … ) ← { q 1 ( … ) ∧ q 2 ( … ) ∧ p 2 ( … )
              where  q 1 ( … ) ← …
                     q 2 ( … ) ← …          }
```

述語名 q1，q2 の有効範囲は括孤 { } の中だけである。節頭 p1 (…) に対応する節体は q1 (…) ∧ q2 (…) ∧ p2 (…) である。

節の中に現われる変数の有効範囲はその節の中だけである。

例2.　変数のスコープの拡大

```
p ( ……*x …… ) ←
    {  ……*x ……
       where ……
             ( *x ) : q ( … ) ← ……*x ……
    }
```

q の定義節の前に置かれた (*x) は import リストである。節頭 p (…*x…) に含まれる *x，節体 ……*x …… に含まれる *x，q の定義節に現われる *x とが同一の変数 *x であることを，import リストに *x を記入することで指定することができる。

例3.

2.3.4 ラムダ抽象化の例2は，節定義の入れ子と変数名の有効範囲を拡大したうえでつぎのように書くことができる。

```
  ← { minus ( 3, 2, *u ),
```

注）プログラムの入れ子構造はプログラム構成単位の階層間での (上下の) スコープを制御するのに対し，モジュール構造は，内一層内の (相互間の) スコープを制御する。

apply (addu , 0. *w. <>)

where (*u) : addu (*x, *y) ← plus (*x , *u , *y)]

I　パターンマッチ制御

ユニフィケーション時において，システムが行うパターンマッチを制御する方法を考える。この方法は，本来は言語要素である sequential componentや節などを処理対象として扱うときに利用されるであろう。

例1. 引用符号

通常のユニフィケーションでは，変数と変数，変数と定数，変数と構造を持つ項とが一般にマッチする。そのため特に変数と変数とだけマッチさせたいことがあるときでもそうすることができない。引用符号 'を次のようなマッチング制御機能を持つものとしよう。変数 *u に引用符号が前置された '*u は，変数 *u または，（同じ綴りの）変数 *u に引用符号が前置された '*u のいずれかとマッチすることができる（マッチング自体は *u と *u，または *u と *u との間で行われる）。定数 c に引用符号が前置された 'c は，定数 c または 'c とだけマッチできる。構造を持つ項の場合，引用符号が前置されると，その項の構成要素ごとにそれぞれの対応する位置にある要素との間で，上記の規則を用いてマッチングが試みられる。

なお，一般的なパターンとのマッチングの制御を行うのは容易でない。'FGKL／Sの項定数は，一般に，木（タップル）構造を成しているので，例えば，パターンを正則式表現しようとすれば，木構造上の正則言語が与えられていることが必要である。しかしながら，木構造上の適切な正則言語はまだ得られていないからである。

J　引数渡しの制御

'FGKL／Sの計算過程はユニフィケーションの繰返しで遂行され，手続き呼出しはいわゆる名前呼び（call by name）である。しかし，値を表わす実引数（項）について値呼び（call by value）としたい場合，例えば，ゴール呼出し，p（plus（3, 2），*y）において，pの定義がp（*u, *v）←. . .であるとき，*u はplus(3, 2)となるよりも5としたい場合がある。

一般に，関数呼出しでは，(i)実引数を評価する／しない，(ii)引数に関数（本体）を適用する／しない，の組み合わせにより表2.2.3のように4種類の方式が考えられる。なお，スコーレム関数は表の項目1に相当する。

ここでは項目3に相当する機能を導入する。この関数適用を指定する方法として2つ考えられる。(i)関数適用する項のパターンを予め宣言しておく，(ii)関数適用すべき項を記述するごとに印をつけていく。(iii)専用の述語を用意しておく（is [1]）。無論，いずれの方法であっても関数本体（値を与える手順）は別に定義しておくことが必要である。

表2.2.3　関数呼出しの方式

	実引数評価	関数本体適用
1	しない	しない
2	する	しない
3	しない	する
4	する	する

ここでは，(j)を採用し，宣言と定義とを次のような節定義で与えることにする。

byvalue (α , β) ← γ

ここで，αは項f（…）の形をしている。βはαの値となる項である。γにはαからβを得る手順を書く。得られたβが値呼びの項であっても再び計算されることはない。（必要ならば，γでそのことを記述する。）

関数適用は，ユニフィケーションの際に次のようにして行われる（ユニフィケーションアルゴリズムが拡張される）。

項δとεとのパターンマッチングにおいて，項δが関数適用であるとしよう。すると，はじめに項δは関数の定義に従い計算されて，その結果（値）をδ'とする。のち，δの代わりにδ'を用いてεとのパターンマッチングがはじめられる。δが関数適用でなければ直ちにδとεとのパターンマッチがはじめられる。注）

例1. 階乗 fact

(a) byvalue (fact (＊n) , ＊m) ←

　　＊n＞0 | eq (＊m , ＊n×fact (＊n－1))

(b) byvalue (fact (0) , 1) ← |

(c) byvalue (fact (＊n) , fact (＊n)) ← |

(d) eq (＊x , ＊x) ←

ここで，関数×，－は関数適用の指定があると仮定した。(c)節は項をそのまま返すことを示す。

なお，(a)節を，

(a') byvalue (fact (＊n) , ＊n×fact (＊n－1)) ← ＊n＞0 |

とすると，例えば，fact (1)の値は1ではなくて，1×fact (1－1) となることに注

注）byvalueは，ユニフィケーション時のmacroと考えられる。

意されたい。（ただし，関数×，－，factがいずれも関数適用なので最終的に1になる可能性は大きい。）

K　バックトラック制御

'FGKL／Sでは，実行に際してバックトラックを基本にしているために，バックトラックの制御を行わなければ無駄なバックトラックを行うことも多い。また，プログラムのバグのために使用者の予測を越えたバックトラックを行うこともあるためデバックが極めて困難になり，大規模なプログラムを作成する際には問題が多い。

バックトラックの制御機構として2.2.1基本論理式で述べたようにguard barが用意されている。ここではバックトラック動作をある範囲だけ抑止するメタ概念，control block cb（ε）を導入する。εは sequential component である。cb（ε）の代わりに〔ε〕と記述してもよいとしておく。cbはバックトラックの際にのみ意味を持ち，cbの中にバックトラックが及んでくると，cb 全体は直ちにfailureとなる。

例.

P1∧〔P2∧P3〕∧P4

P1から順に実行していき，P4で失敗した場合，P3をやり直しにいくのでなく（〔〕全体が直ちにfailure するので），P1をやり直すことを指示する。

control blockはそのblock内へ後でバックトラックが及んでこないことを指定しているわけであるから，block を成功して出る時点では，block 内へバックトラックを行うための情報を保存する必要がなく，そのために必要な記憶域を節約することができる。

L　エラー処理

プログラムの実行中に発生するエラーに対しては，通常システムによって適当なメッセージと回復処理が行われるであろう。ここでは，利用者固有のエラー処理を与える方法を考えてみたい。一般にエラー発生後に何らかの回復処理のち処理を継続するわけであるが，必ずしもエラー発生直後から再開するとは限らないため，継続すべき位置を特定できるようにしなければならない。これは正常な実行順序を大きく変更することになり，非決定的な計算方式を採る'FGKL／Sではインプリメントが容易でない。そこでここでは，単にエラー発生時に実行されるべきゴール節を指定する方法を与えるにとどめる。

αをエラー識別子，β，γを sequential component とするとき，述語 errorset（α，β，γ）の実行は次のように行われる。

βの実行中にエラーαが発生するとγが実行されて，その結果がerrorset 全体の結果となる。エラーが発生しなければ，βの結果が全体の結果となる。

なお，プログラム中において任意にエラーを発生できるような述語error（α）を用意することにより，複雑な実行順序の指定が可能となろう。しかし反面，論理的な裏付けは複雑にまた困難になる。

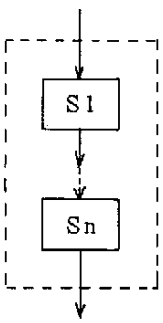
M　構造化プログラミング

プログラムを限られた，しかし，よく整った構造だけを用いてつくり上げるという構造化プログラミング[7]の考え方の'FGKL／Sへの導入を検討する。

従来の構造化プログラミングは，手続き型言語のプログラミングパラダイムとして提案されたもので，論理型言語のパラダイムとして必ずしも通用するとはいえない。ここでは，手続き型言語を機械的に翻訳するための'FGKL／Sでのプログラム構造の表現法やsyntactic sügarを考えていくことにする。

構造化プログラミングの3大構造，連接（concatenatoin）と選択（selection），繰り返し（repetition）は'FGKL／Sでは，次のようになるであろう。

(a) 連接

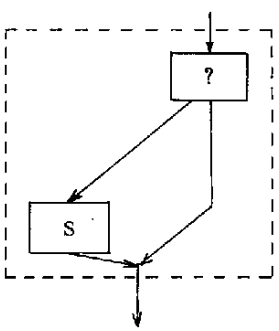


S1，S2，…，Sn　は
呼出し　seq（…）　に
翻訳される。

seq（ν1，ν2，νm）←
　s1（…）∧s2（…）∧……sn（…）

ここで，seqはこのプログラム（全体）の名前，ν1，ν2，…νmはこのプログラムの入力または出力を表わす項，s1（…），s2（…），…，sn（…）は文S1，S2，…，Snを述語化したsequential componentである。

(b) 選択（1）



if　?　do　s　は
呼出し　ifdo　（……）　に
翻訳される。

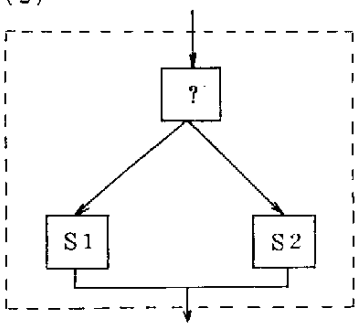
ifdo (ν1, ν2, …, νm) ←

p (…) | s (……)

ifdo (* ν1, * ν2, …, *νm) ← |

ここで，p (……) は？に対応する sequential component，s, (…) は文Sに対応した sequential component である。ν1，…νmはこのプログラムの入力または出力を表わす項である。

(c) 選択 (2)



if ? then S1
else S2

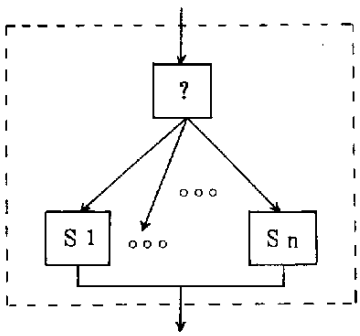
ifthenelse (ν1, ν2, …, νm) ←

p (…) | s1 (…)

ifthenelse (ν1, ν2, …, m) ←

| s2 (…)

(d) 選択 (3)



case i of (S1, …, Sn)

caseof (1, ν1, ν2, …, νm) ← | s1 (…)

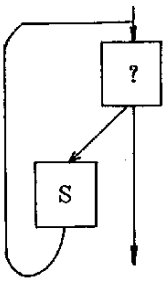
caseof (2, ν1, ν2, …, νm) ← | s2 (…)

……

caseof (n, ν1, ν2, …, νm) ← | sn (…)

繰返しは再帰的 (終端) 呼出しで実現する。

(e) 繰返し (1)



while ? do S

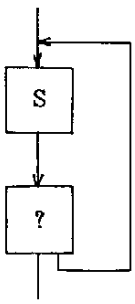
whiledo (ν1, ν2, … νm) ←

p (…) | s (…) ∧ whiledo (……)

whiledo (ν1, ν2, …, νm) ← |

ここで, ν1, …, νmはこのプログラムに対する入力または出力を表わす項であるか, あるいは sの中だけで用いられてくりかえしごとに更新されるような変数である。

(f) 繰返し (2)



repeat S until ?

repeatuntil (ν1, ν2, …, νm) ←

s (…) ∧ repeatuntiltail(……)

repeatuntiltail (ν1, ν2, …, νm) ←

p (…) | repeatuntil (……)

repeatuntiltail (ν1, ν2, …, νm) ← |

また代入文に対しては次のような技法を用いる。

(g) 連接, 条件の中で代入される変数

代入が行われるごとに新しい変数を用意する (sinfle assignment化)。

(h) 繰返しの中で更新 (代入) される変数

繰返しの所で述べたように, 更新変数は節定義の際に引数として記述する。

次に, いくつかの syntactic sugar を示す。これは, ゴール節呼出しや節定義におい

て述語の引数がすべて変数である場合に効果が大きい。

(i) 節の呼出しと節の定義を融合させる。即ち，例えば，条件 ifdoにおいて呼出しが，

…∧ifdo(＊Y1，…＊Ym)∧…

で， 定義が，

ifdo(＊X1，＊X2，…，＊Xm) ← p(…) | s(…)

ifdo(＊X1，＊X2，…，＊Xm) ← ¦

のとき，呼出しを次のように表わす。

…∧(＊Y1，…，＊Ym)〔if p(…) do s(…)〕∧……

ここで，元々p(…)，s(…)で用いられた変数＊X1，…，＊Xmは系統的に項 ＊Y1，…，＊Ymに名前変えを行っておく。

他の場合についても，呼出しを次のように表わす。

(＊Y1，…，＊Ym)〔if p(…) then s1(…) else s2(…)〕

(＊N，＊Y1，…，＊Ym)〔case ＊N of (s1(…)，s2(…)，…，sn(…))〕

(＊Y1，…，＊Ym)〔while p(…) do s(…)〕

(＊Y1，…，＊Ym)〔repeat s(…) until p(…)〕

(ii) 同一変数への代入が2回以上あるときは，新しい変数名を用意して書く代わりにnext ν （νは変数名）と書く，以後，単にνと書かれたときは新しい変数を指すと約束する例として，構造化プログラミングに従ったエイトクィーンプログラムを例1に挙げておく。ここで，α−＞βは節定義αをβで置き換えることを表わす。

ここで述べた構造化プログラミングは，'FGKL／Sによって従来型言語のプログラムにできるだけ忠実に機械的翻訳ができるための指針を与えたにすぎない。'FGKL／Sのような論理型プログラミング言語にはそれにふさわしいプログラミングスタイルが考えられるはずである。

Prologによる解を例2に挙げておく。プログラムの小ささに注目されたい。

例1.

```
x(0,0). x(1,0). x(2,0). x(3,0). x(4,0). x(5,0). x(6,0). x(7,0).

col(0,true). col(1,true). col(2,true). col(3,true). col(4,true).
col(5,true). col(6,true). col(7,true).

up(-7,true). up(-6,true). up(-5,true). up(-4,true). up(-3,true).
up(-2,true). up(-1,true). up(0,true). up(1,true). up(2,true). up(3,true).
up(4,true). up(5,true). up(6,true). up(7,true).

down(0,true). down(1,true). down(2,true). down(3,true). down(4,true).
down(5,true). down(6,true). down(7,true). down(8,true). down(9,true).
down(10,true). down(11,true). down(12,true). down(13,true). down(14,true).
down(15,true).

generate_1(N) :-
  H=0,
  (N, H){repeat (N,H){if col(H, true),up(N-H,true),down(N+H,true) do
                          x(N,_)->x(N,H),
                          col(H,_)->col(H,false),
                          up(N-H,_)->up(N-H,false),
                          down(N+H,_)->down(N+H,false),
                          next N=N+1,
                          (N){if N=8 then K=0,
                                          (K){repeat x(K,T),
                                                     print([K|T]),
                                                     next K=K+1
                                              until K=8},
                                          newline
                                      else generate_1(N)},
                          next N=N-1,
                          down(N+H,_)->down(N+H,true),
                          up(N-H,_)->up(N-H,true),
                          col(H,_)->col(H,true)],
              next H=H+1
          until H=8].
```

例2.

```
eight_queen(Q) :- queens([1,2,3,4,5,6,7,8], [], Q).

queens([], Y, Y).
queens(X, Y, Z) :- select(U, X, V),safe(U, Y, 1),queens(V, [U|Y], Z).

select(X, [X|Y], Y).
select(U, [X|Y], [X|V]) :- select(U, Y, V).

safe(U, [], _) :-!.
safe(U, [P|Q], N) :- nodiag(U, P, N),M is N+1,safe(U, Q, M).

nodiag(U, P, N) :- T1 is P+N,T2 is P-N,not(U=T1),not(U=T2).

not(P) :- P,!,fail.
not(_).
```


－ 参考文献 －

1) L. M. Pereira, et al. : Use s Guide to DEC system－10 Prolog, Univ. of Edinburgh, 1978

2) D. H. D. Warren : Higher－order Extensions to PROLOG－Are they needed ?, D.A.I.R.R, No. 154, 1981

3) K. A. Bowen and R. A. Kowalski : Amalgamating Language and Metalanguage in Logic Programming, School of Computer and Information Science, Syracuse University, 1981

4) S. Goto : DURAL : An Extended Prolog Language, 京都大学数理解析研究所講究録, No. 396, 1980

5) 中島秀之 : Prolog／KRの概要, 情報処理学会第20回記号処理研究会, Mar. 1982

6) S. B. Jones : Structured Programming Techniques in Prolog, Logic Programming Workshop, 1980

7) O.-J. Dahl, et al. : Structured Programming, Academic Press, 1972

2.2.4 抽 象 化 機 構

抽象化（abstraction）は人間のもつ知的な機能の中でも極めて特徴的なものとされる。

プログラム言語での抽象化の流れは，一つはデータ抽象化であり，今一つはプロセスの抽象化である。

本節の残りの部分では主としてデータの抽象化を扱うので，ここではプロセスの抽象化を取り上げておきたい。

プロセスの抽象化は電子計算機の歴史から言えば，詳細から概略へ，手順（how）から目的（what）へと変化している。'FGKLにおける論理プログラミングという基本思想もこのプロセス抽象化の努力の反映であると言うことができる。

一方，プロセス抽象化の別の動きは関数型プログラミング（functional programming）のようなプロセス自体を取り出して演算しようという考え方や，データ抽象化と連結したオブジェクト志向（object oriented）計算という手順などに表われている。（オブジェクト志向についてはメッセージ処理機構として後に述べる）。

一方機能クラスのような考え方は，各種のプロセス間の共通する機能や構造をとり出して分類のキーとする。これはこれで1つの抽象化のステップとすることができる。（しかも分類キーの寄せ集めでプロセス自体が再合成されるならば……）

プロセスの抽象化の最終形式については議論が分かれている。「何をしたい」という要求の表明がそうであるという考えと，プロセス＝δ（process element）という定式化がそれであるという考え方との間で各種の試行がなされている。

前者はある意味でプログラミングの否定だが，後者はいわばプログラムの公式化ということになる。いずれにせよ，今日の時点では難かしい。しかも，この2つの立場の差異は，「プログラミングとは何か」という根本的な観点の差異を物語っている。

平均的なプログラムはこの両者の観点の中間にいる。なぜなら，プログラムを説明するのに，その目的を話し，どうして実現するのかという概容を付け加えるからである。プログラマ（ないし彼／彼女の書いたプログラム）が，両観点の間隙を埋めて見せるのである。

プロセスの抽象化は，従って，そう簡単なことではない。アルゴリズムやデータの抽象化と合わせて考えねばならず，あるいは人工知能とは何かという問題にプログラミングの場で答える努力が必要なのかも知れない。

'FGKLでもプロセスの抽象化に手をこまねくわけではなく，機能クラスによるプログラム・ライブラリの作成や，高機能エディタなどのプログラミング補助手段などの形式でプロセスの抽象化への試みがなされるものと考えて良いであろう。

抽象化とは以下ではデータ抽象化（data abstraction）を主として意味する。[注1] データ抽象化の定義としては，「データをそのデータに適用可能な操作（operation）の集合として

注1） データ以外のオブジェクトの抽象化について今述べたようにもいずれ考える。

規定する」のが本来の定義である。

この報告書の文脈では，しかし，実用的な観点から「データ型と述語との適用関係を整理して，型の不整合やそれによるエラーを防ぐ機構」を抽象化機構の定義として用いる。いわゆるデータ型チェックを拡張整理したものと考えて良い。

この抽象化機構について考慮すべき内容には次のようなものがある。

(1) オブジェクトに対する型指定方式。

(2) 関数に対する引数の型指定方式。

(3) ユーザ定義による型の導入方式。

(4) データ型に関する情報の集積方式。

(5) ハードウェア的表現形式。

(6) 抽象化機構を支えるハードウェア命令。

(7) 組み込みデータ型。（プリミティブ型）。

(8) データ型の型変換機構。

一方データ抽象化に関してこれまで行われた試みとしては次のようなものがある。

① ＳＩＭＵＬＡにおけるクラス化機構。[1]

② 抽象データ型の代数的，操作的，公理的定義。[2]

③ 抽象データ型に基く言語試作。ＣＬＵ[3]，ＡＬＰＨＡＲＤ，[4]

④ 抽象データ型とプログラムの証明。[5]

⑤ 抽象データ型とプログラム合成。[6]

⑥ 抽象データ型と関数型プログラミング。[7]

⑦ 機能クラス。[8]

⑨ ＬＩＳＰへのデータ型導入。[9]

これらの経験を活かして'ＦＧＫＬのデータ抽象化を次のように考える。

A．抽象化指定方式

抽象データ型では本来データ型が適用可能な操作集合で与えられる。ところが'ＦＧＫＬにおいては，元来変数が無型であり，操作に対応する論理関数のどこに変数が与えられるかは，実行時にならないとわからない。しかもデータ型の不一致がそのまま実行時エラーとなるのではなく，これが適当な論理式を選択する鍵となっている。

これに対応する方向には二つある。

(1) 変数型宣言を実施する。

(2) 論理関数式に引数型宣言を実施する。

変数型宣言の問題点は，型宣言の範囲をどう定めるか，どういう形式で型宣言を行うか，にある。また'FGKLでは知識情報処理の観点から論理関数を動的に追加，変更，削除する。これは抽象データ型を関数の集合として定義しようとすることを困難にする。

定義体をグループ化するのにbegin－def ～end－def という括弧構造を使えるならこの1つのブロック内で変数を宣言することができる。

(例)

```
begin－def
  def－var  X, Y, Z : list  ;
  append  ( nil , X, X )
  append  ( A . X, Y, A . Z ) :←append ( X, Y, Z ).
eud－def
```

もう一歩範囲を進めて，複数の定義体の集合をモジュール化して，このモジュールの中で共通の変数とそのデータ型を宣言してゆく方法もある。

あるいは各節ごとに変数宣言するという方法もある。

一方論理関数式に引数型宣言を与える方式ではＬＩＳＰにおける経験が役立つ。[8) 9)] これには機能クラスを用いる方式と単に引数部分に型を指定する方式とがある。

機能クラスでは型指定の記述部が論理関数本体と切り離せる利点がある反面，引数パターンの中に埋め込まれた変数の型宣言までは手が回らない。

一方単純に引数部分子たる引数の仮変数に型を指定させる方式では，細かい指定，特に型不一致における処理等の記述が不便であり，また型指定の変更，省略等の運用にも難点がある。

B. 型定義方式

関数集合としてデータ型を指定する方式を取らないとすると，基本データ型から何らかの演算によってユーザが型を定義することとなる。ＰＡＳＣＡＬやＡＤＡなどでは例えば次のようなものを用いている。

① スカラー型　　順序付き有限集合

② 部分範囲型　　スカラーの部分集合

③ 配列型　　スカラーの直積とデータ型を対応させたもの。

④ 集合型　　スカラーの部分集合自体を要素とするもの。

⑤ ポインタ型　　データへの参照関係をデータとするもの。

⑥ ファイル型　　データの順序集合。配列型と違ってスカラーとの対応関係は陽に表われ

ない。

⑦ レコード型　　データ要素を有限個集め各々に名前を付けて参照可能とする。

'FGKLのもとになっているPROLOG ではデータ型は構文上でのみ区別され，ポインタ型はリストとして表現される。

ユーザによるデータ型の拡張を許すにはこのような構文面への配慮も必要となる。

一方，LISPマシンではFlavorと言うオブジェクト志向型のデータ型定義の方式を採用している。

Flavorではデータ型の枠をユーザに提供する。この枠組に対して各種の基本演算がまず定義され，さらにこの基本演算を組み合わせて，複雑な処理をメッセージという形式で実現する。

例えば，XY軸上に位置と速度を持ち，荷物をのせたShipというFlavorは次のように定義できる。

```
(defflavor ship (x-position y-position
                 x-velocity y-velocity mass)
                ()
  :gettable-instance-variables)
```

X－position以下massまでが，shipの基本属性である。この部分はPASCALのrecordに相当する。

:qettable－instance-variablesはshipに対する操作記述の一部である。この場合，shipの属性値を知る操作が適用可能であることを示す。

具体的にFlavorに属するデータを作るには以下のようにmake－instanceという命命を使う。Flavorはデータの抽象型定義になっている。

```
(setq my-ship (make-instance 'ship))
```

データ属性はfuncallという（メッセージ管理の）汎用関数を使って下のように表現される。

```
(funcall my-ship ':x-position)
```

このFlavorでは下のようにshipというデータの属性を変更することはできない。

```
(funcall my-ship ':set-x-position 3.0)
```

:qettable－instance－variablesしか定められていないからである。

shipのFlavorを次のように変更すれば（抽象的には全く別のデータ型が出来上るということなのだが）属性値の変更ができる。

```
(defflavor ship (x-position y-position
                 x-velocity y-velocity mass)
                ()
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)
```

次の処理により，

```
(funcall my-ship ':set-mass 3.0)
```

属性値は以下のようになる。

```
X-POSITION:          unbound
Y-POSITION:          unbound
X-VELOCITY:          unbound
Y-VELOCITY:          unbound
MASS:                3.0
```

属性値設定は，funcallを使わなくても，make－instanceの引数として与えることもできる。これは:initiable－instance－variablesという機能宣言によって可能となっている。

```
(setq her-ship (make-instance 'ship ':x-position 0.0
                                    ':y-position 2.0
                                    ':mass 3.5))
```

属性をすべて出力する関数 describe を使うと，

```
(describe her-ship)
```

の出力は下のようになる。

```
#<SHIP 13756521>, an object of flavor SHIP,
 has instance variable values:
        X-POSITION:          0.0
        Y-POSITION:          2.0
        X-VELOCITY:          unbound
        Y-VELOCITY:          unbound
        MASS:                3.5
```

このようなデータ作成時のデータ属性の指定には暗黙値(default)が使用できる。

```
(defvar *default-x-velocity* 2.0)
(defvar *default-y-velocity* 3.0)

(defflavor ship ((x-position 0.0)
                 (y-position 0.0)
                 (x-velocity *default-x-velocity*)
                 (y-velocity *default-y-velocity*)
                 mass)
```

```
                    ()
  :gettable-instance-variables
  :settable-instance-variables
  :initable-instance-variables)

(setq another-ship (make-instance 'ship ':x-position 3.4))
```

上の例でdefaultは単に数値のマクロ定義だと思えば良い。x－positionからy－velocityまでが暗黙値をもったわけである。

make－instanceのときに，引数を与えると，引数の方が暗黙値に優先する。この時点で属性を出力させれば下のようになる。

```
(describe another-ship)
#<SHIP 14563643>, an object of flavor SHIP,
 has instance variable values:
        X-POSITION:          3.4
        Y-POSITION:          0.0
        X-VELOCITY:          2.0
        Y-VELOCITY:          3.0
        MASS:                unbound
```

このようにFlavorに対してはmake－instanceとかdescribeとか，あるいはfuncallのような関数（メッセージ）が用意されている。これらをLisp Machineではmethodと呼んでおり，Flavorに対するmethodは次のようにしてユーザが作ることもできる。

```
(defmethod (ship :speed) ()
  (sqrt (+ (^ x-velocity 2)
           (^ y-velocity 2))))

(defmethod (ship :direction) ()
  (atan y-velocity x-velocity))
```

methodは常にFlavorと対になって定義される。この点がいわゆるLispの関数（適用データが何であろうと構わない）と異なる点である。

'FGKLにおいてFlavor的なもので実現した場合に，どうなるかは，2.2.1の基本論理式のところでモジュール・クラス（module class　）という名前で紹介されている。

Flavorのような抽象データ型の場合には，δ（X，Y，…）のような'FGKLの一般形式を用いる代わりに，　X.f（Y）のようなメッセージ呼び出し機能を活用する。

C. メッセージ処理機構

抽象データ型の考えを一歩推し進めると，述語に対して各引数のユニフィケーションを行う

という本来の'ＦＧＫＬでの処理機構から離れて，抽象データ型をもつデータに対して述語をメッセージという形式で割り当て，実行させるという処理機構が考えられる。

この間の関係は次のように捕えることもできる。

述語レベル　　f（X_0，Y_0）←

f（X，Y）　　← X＝X_0，Y＝Y_0，∞∞

抽象データ型レベル

X_0　のデータについて

applicable　f

such that　f（X，Y）← ……

メッセージ処理機構

X_0 ……f（Y_0）

メッセージ

f（X_0，Y_0）← ……

メッセージ処理機構は各データ要素に注目して，そのデータ要素がどう働くべきかを記述する。従って，複数個のデータ要素が存在し，各々が相互作用をするというような場合に，簡単に記述することができる。

'ＦＧＫＬでの内部付な述語の形式はf（X_1，X_2，∞∞）または＜f，X_1，X_2，……＞となっているので，メッセージ処理のためには汎用メッセージ処理述語message を用いる。

message 述語は次のように用いられる。

(1) message（Obj，〃メッセージストリング〃）

(2) message（Obj，＜メッセージリスト＞）

しかし，この形式では，「見易い形式」ではないので，次のような便宜的な書式を用いることもある。

(3) Obj.主メッセージ（追加メッセージ）

例えば

turtle.move　（X，5，Y，－4）

というような具合である。

ここで重要なことは，このように抽象的に定義されたデータとは一体何物なのか，ということである。

概念的にはこの「抽象的なデータ」は実際にある場所を占有する必要はない。あくまでも，

各データと同時に定義された操作（述語）に対して，何らかの作用を返せば良いのである。しかし，現実にはこれらの抽象的なデータを（例えばフォン・ノイマン型計算機上に）具体的に作成する必要がある。

'ＦＧＫＬの場合には，具体的にはタプルがこれらの抽象的なデータ型のために用いられることになるであろう。

データ型が抽象的に定義されたとき，その型名，その構成要素の名前と型，そのデータ型に対する操作（述語）の一覧表と，各操作の内容とかすべて記述された表ができて，これが抽象データの一種類として登録される。

これはデータの種類としてはタプルであるが，例えばPROLOGの世界では存在しなかったデータである。

この抽象的なデータの枠組にもとづいて，実際に，create,newなどの述語で作られるデータは，また新しい形式をもったタプルとなるであろう。

下図のように，タプルの先端はこれがどのような抽象データ型に属するか，またこのタプルの名前は何かを示し，そのあとに，枠組の中で存在が表明された種々の成分が続くことになる。

このタプルはユニフィケーションの結果作られてゆく中間的なタプルでもなく，入力（登録）された節のように直接タプルの各成分に要素が代入されたものでもない。いわばユニフィケーション（より具体的にはモジュールの基本操作）の結果として副作用的に生成されたものである。

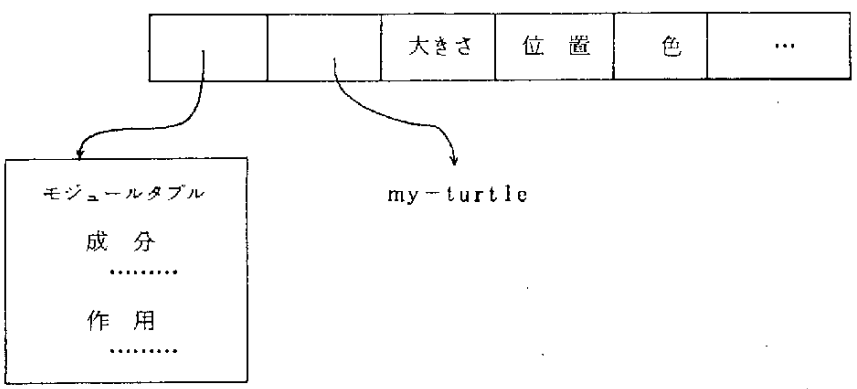


図2.2.18　抽象的データの具体的な実現

しかも，この種のデータは各成分がユニフィケーションの度に動的に変化する。（新規作成，旧体の廃棄という手順でシミュレートもできるが）したがって従来のPROLOG では見られなかったデータの型と考えることができる。

D. 組み込みデータ型

データ型をユーザによる定義にまかすつもりでも基本的なデータ型とその基本演算系は前もって用意せねばならない。

最低限必要なのは，（2.2.2.A参照），

① 整数

② アトム

③ 変数

④ タプル

である。

この他に実数，配列，集合，構造体，スタック，キュー，ファイルなどがあれば当面目標とする第5世代のシステム作成には役立つであろう。これらは基本的には上の要素の組み合わせで実現される。

考慮しておきたいデータ型として，ディスプレイ上の図形，並列処理制御用データなどがある。

リストを始めこれらは主としてタプルを使って実現される。

E. データ型変換

データ型変換には自動的なものと，データ型変換の関数を利用するものとがある。自動型変換の対象となるのは，整数と実数間の数値変換，文字列とデータ型との変換（入出力で行うこと）などである。

自動型変換にはプログラムの表面上に現われない操作が伴うので，プログラムの制御上の問題がある。

型変換の関数を一々使用するのは面倒な場合がある。

機能クラスを用いて，関数処理部で型を変換する方法なら，書式上は型変換が表面に現われず，一方ユーザが型変換の有無を検知できるので良かろう。

── 参考文献 ──

1) Dahl，O－J et al，〃 Common Base Language 〃 Norwegian Conpnt－iug Center （Oct，1978）

2) Liskov，B. and V. Benjins，〃 AN APPRAISAL OF PROGRAM

SPECIFICATIONS", in Wagner ed, "Research Directions in Software Methodology".The MIT Press, (1979)

3) Linlov,B, "An Introduction to CLU",Memo 136, LCS, MIT, (1976)

4) Wulf, W, A, R, London ane M, Shaw, "ABSTRACTION and VERIFICATION in ALPHARD －Introduction to Language and Methodology", CMU, (1976)

5) Nakajima, R, et al, "THE τ PROGRAMMING SYSTEM － A SUPPORT SYSTEM FOR HIERARCHICAL AND MODULAR PROGRMMING", IFIP 80 PP 299－304, (1980)

6) Futatsugi, K, and K, Okada, "SPECIFICATION WRITING AS CONSTRUCTION OF HIERARCHICALLY STRUCTURED CLUSTERS", IFIP 80, PP 287－292

7) Guttarg, J, T, Horning, and J, Williams, "FP With Data Abstraction and Strong Typing", Proc 1981 Functional Programming Language and Computor Architecture, PP 11－24, 1981

8) Kurokawa, T, "The Function－class", Conf. Record of 1980 LISP Conf., 1980

9) Cartwright, R, "A Constructive Alternative to Axiomatic Data Type Definitions", Conf. Record of 1980 LISP Conf. 1980

10) Weinred, D, and D, Moon, "Flavors: Message Passing in the Lisp Machine", MIT AI Memo 602, (Nov (1980)

2.3 計算（推論）サブシステム

2.3.1 インタプリタ（コア部）

インタプリタのコア部では，以下のものが処理される。

- 述語呼び出し（refute）
- エラー処理 （error）
- 割り込み処理（break）
- ステッパ呼出し（step）

インタプリタは，スピードはもちろんのことだが，柔軟性（拡張性）にも注意をはらって設計

されなければならない。上記の４つの処理も，ユーザが適当な述語を定義すれば変更できるようになっていることが望ましい。従って，これらの処理はできる限り，述語呼び出しの形式をとる。しかし，述語呼び出しの手続きは，マシンコードを直接呼び出すのに比べてオーバーヘッドが大きいので，スピードに大きく影響する部分は，直接呼び出しを採用せざるを得ないであろう。

以下ではインタプリタ構成の具体的手法について述べる。これはProlog／ＫＲの経験を基礎として，ＦＧＫＬのインプリメント手法を具体化したものであるが，これが最善という意味ではなく，一例として見ていただきたい。一般論を述べた2.2.2.B節の記述とは必ずしも一致していないが，おおまかな対応はとれるはずである。

A　スタック

基本的な制御や，変数の値のbindにはスタックが用いられる。スタックとしては以下の４本を用意する。

- alternative－stack（ＡＳ）注）
- bind－stack（ＢＳ）注）
- control－stack（ＣＳ）注）
- disengage－stack（ＤＳ）注）

BSを除いては普通の構造のスタックで良いのだが，co－routine等の並列計算をサポートすることを考えて，各フレームをvectorで実現し，そのvectorの連鎖としてスタックを表現することにする。不要になったスタック・フレームの回収はすべてＧＣ（garbage collector）にまかせることになる。

以下の各項では述べないが，ＡＳ，ＣＳ，ＤＳの各フレームには，（上記の理由により）自分の親のフレームへのポインタが入る。デバッグ用に各スタックの中味をダンプする必要のある場合には，このポインタをたどりながら各フレームの中味を見てゆくことになる。フレームはvectorで表現されているから，そのフレームへのポインタが与えられれば，中味にアクセスするのは2.2.1.E節で与えられた述語群を使用すれば良い。実際のインプリメントが定まれば，これらはparent－frame（フレームを入力とし，その親を出力とする）等のスタック操作用述語群として与えられるべきである。

注）　2.2.2では「ＯＲ－プロセス」として表現されている

注）　2.2.2の「スタック」に相当

注）　2.2.2では「ＡＮＤ－プロセス」として表現されている

注）　2.2.2の「trail」に相当

a　alternative−stack

ASはバックトラックに用いられる。ASに積まれる情報としては，以下のものがある：

呼び出し側の述語

まだ試みていない主張のリスト

対応するCSへのポインタ

対応するDSへのポインタ

まだ試みていない主張が存在しない場合（即ち計算がdeterministicな場合，あるいは，現在試みているのが最後の選択肢の場合）にはASは伸びない。

AS制御用述語には以下のものがある：

push−as（＊pred，＊alternatives，＊cs，＊ds）

pop−as（＊pred，＊alternatives，＊cs，＊ds）

push−as及びpop−asの4引数はそれぞれ上記の4項目に対応している。

as−empty？

as−top（＊as−ptr）

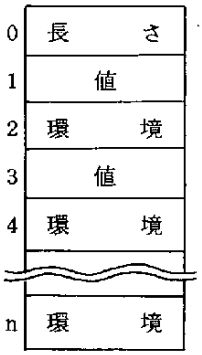
ASのトップへのポインタを返す。実際のインプリメンテーションでは，AS〜DSのトップへのポインタはレジスタに格納されているはずである。従って実際には，その値を一度変数にbindすることは不要で，直接参照することになるのだが，ここでは記述上述語呼び出しを介して値をとってくることになっている。cs−top，ds−topも同様である。

b　bind−stack

BSは変数のbind状況を示すのに用いる。一般に変数の値は述語呼び出しが終了した後も参照される可能性があるので，Lisp等のように即座にスタックをたたむことができない。EdinburghのPrologのインプリメンテーションでは後で参照される可能性のあるものはglobalスタックに，そうでないものはlocalスタック上に置くことによってできる限り，スタックをpop−upできるよう工夫している。ここではそういう方式は用いず，各レベルごとの変数結合をフレーム[注)]（以下BSフレームと呼ぶことにする）で表現し，そのフレームの連鎖としてBSを表現する。不要なフレームの回収はGC（ガーベイジ・コレクタ）の仕事となる（2.2.1C参照）。

変数の値は，その構造のスケルトン（プログラム中に現われる）へのポインタと，その環境（BSフレーム）へのポインタの対で表現される。従って各BSフレームは下図のような構造を持つ。

注）データ構造としてはvectorを用いる。



各変数はＢＳフレームの先頭からのオフセットによってその値がアクセスされる。値が未定の変数は，環境へのポインタをｎｉｌ にすることによって表現する。

ＢＳ操作用の述語としては，以下のものがある。

```
bind ( *var , *bs , *value , *env )←
  if ( varp(*value )∧
       lookup ( *value , *env , *nv , *ne ),
     bind ( *var , *bs , *nv , *ne ) ,-- then part
     loc ( *var , *loc )∧              -- else part
       setv ( *bs , *loc , *value )∧
       setv ( *bs , *loc+1 , *env ) )
lookup ( *var , *bs , *value , *env )←
     loc ( *var , *loc )∧
     sref ( *bs , *loc+1 , *env )∧
     not ( =(*env , nil ) )∧
     sref ( *bs , *loc , *value )
unbind ( *var , *bs )←
     loc ( *var , *loc )∧
     sset ( *bs , *loc + 1 , nil )
new-bs ( *length , *bs )←
     =( *vl , *length × 2 )∧
     new-vector ( *vl . nil.<>, *bs )注)
```

注) 長さ＊ｖｌのｖｅｃｔｏｒの各要素をｎｉｌに初期化し，そのｖｅｃｔｏｒへのポインタを＊ｂｓにｂｉｎｄする。

c. control－stack

ＣＳは述語呼び出しの制御に用いられる。基本的にはLisp等のバックトラックのない言語のそれと同じであるが，ＡＳへのポインタを格納している点が異なる。これは，non local exit等を行った際にＡＳを適切な場所までpop upするのに用いられる。

ＣＳには以下の情報が格納される。

親のＣＳへのポインタ

次に実行すべき述語又はcode（自分が最後の呼び出しの場合にはnil）

ＢＳへのポインタ

ＡＳへのポインタ（自分に対応するＡＳのひとつ上を指している ── deterministic な場合には自分に対応するＡＳは存在しない。全計算がdeterministic な場合にはこれはnilである。）

ＣＳ操作用述語としては以下のものがある：

push－cs（＊next，＊env，＊as）

pop－cs（＊next，＊env，＊as）

peek－cs（＊next，＊env，＊as）

cs－empty？

cs－top（＊cs）……stack topへのpointer

d. disengage－stack

ＤＳは変数がどの時点でbindされたかを記憶しておき，backtrackの際にそれらの変数をunbindするために使われる。これは普通のスタックで良い。[注)]ＤＳスタックの各要素はＢＳの各変数の環境部の番地を直接に示しているので，unbindの際にはその中味をnilに書き換えるだけでよい。即ち：

```
disengage(＊ptr)←rset(＊ptr, nil)
undo(＊ds)←
  ds－top(＊top)∧undol(＊top, ＊ds)
undol(＊top, ＊top)←
undol(＊top, ＊until)←
  ds－contents(＊top, ＊ptr, ＊next)∧
  disengage(＊ptr)∧undol(＊next  ＊until)
```

注） データ構造としてはveceorが用いられる。

B　retute

述語呼び出しとその環境（ＢＳへのポインタ）を引数とし，その述語呼び出しを実行する。実行の成否（成功か失敗か）がrefuteの成否となる。

refuteはrecursiveに呼ばれる場合もある（例えばifから）が，前述の４本のスタックはglobalに１本ずつ存在するだけである。

refuteのrecursive callの情報もＣＳにpushされる。この場合のＣＳのフレームは前述のものとは異なるであろうが詳細は'ＦＧＫＬマシンの機械語に依存するので，ここでは述べない。

ここではrefuteの定義を'ＦＧＫＬ自身で示すことにする。高階述語（and，or，if，not，etc.）に関しては一例を示すに留めるが，それらもここで示す道具を用いて自由に定義できるはずである。

実行の基本モードをdeterministicにするか，non－deterministic にするかは，[注)]意見の分かれるところである。しかしながら，いずれを採用しても，他は容易に実現できるので，ここでは，前者を仮定（即ち各実行が成功しようがしまいが次に進む。また backtrackは行わない）した上で，後者用のインタプリタを作製することにする。即ち，ここで示すrefute自身はdeterministic なインタプリタ上で動くことを仮定しているが，このrefuteによってインタプリトされる言語はnon－deterministicな動きをすることになる。

refuteは下請けとしてresolveを用いる。これは述語呼び出しとその環境，及びその述語の定義（主張のリストとして表現される）を入力とし，パターンがマッチする主張，その環境，およびまだ試みていない主張のリストを出力とする述語である。

以下にrefuteの定義を与える：

```
refute ( *pred, *env )←
    get－def ( *pred, *def )∧
    if ( codep ( *def ), call ( *def ),
        ref1 ( *pred, *env, *def ) )


ref1(nil , *, * ) ←
    if(cs－empty?,  succeed, ),
```

注）ここではdeterministicということばはbacktrackしないということ，又，non－deterministicということばはbacktrackするかもしれないということとほぼ同義である。

```
    --else
      pop-cs(*pred.*rest, *env, *as)∧
      push-es(*rest, *env, *as)∧
      refute(*pred, *env))

refl(*pred, *env, *def)←
    if(resolve(*pred, *env, *def, *body・*rest, *nenv,
                                                    *alt),
    --then
    as-top(*as)∧ds-top(*ds)∧
    push-cs(*rest, *nenv, *as)∧cs-top(*cs)∧
    push-as(*pred, *alt, *cs, *ds)∧
    refute(*body, *nenv),
    --else
    pop-as(*npred, *nalt, *cs, *ds)∧
    undo(*ds)∧
    get-euv(*cs, *env)∧
    refutel(*npred, *env, *nalt))
```

ここで，undoは3.1.1.Dで定義された，変数のbindを解く述語，get－envはcsから，環境へのポインタを取り出す述語である。

```
get-env(*cs, *env)←
      vref(*cs, 3(注), *env)
```

高階述語の例としてnotの定義を以下に与える：

```
not(*pred)←
      as-top(*as)∧cs-top(*cs)∧ds-top(*ds)∧
      get-env(*cs, *bs)∧
      push-as(true(), ←, true().<>, *cs, *ds)∧
      push-cs(fail(), <>, <>, *as)∧
```

注）実際のインプリメンテーションにより異なる。

```
refute ( * pred , * pred , * bs ).
```

notの基本動作は，ASにtrue，CSにfailをプッシュしておいて，引数を実行することである。引数の実行に成功した場合には，CSがポップされてfailが，そうでない場合にはASがポップされてtrueが実行される。failは，falseと異なり，親のゴールを強制的に失敗させる（falseは自分が失敗するだけである）。

C. error

エラーの際にはerrorが呼び出される。標準の動作は、メッセージを印刷した後ステッパを呼び出すことだが，ユーザが任意に動作を（エラーの種類ごとに）定義できる。たとえば，未定義の述語が呼ばれた場合の動作を，エラーではなく，単なる失敗に置き換えるには，

```
error ( " Undefined predicate ",* pred )←fail( )
```

とすれば良い。システムの標準定義は，

```
error ( * mes , * at )←
      princ ( * mes )∧print ( * at )∧ step( )
```

である。

D. break

ユーザからの割込みの際にはbreakが呼ばれる。標準動作はステッパに制御を渡すことである。実行再開には，そこで標準実行を続けるコマンドを入力すれば良い。

breakもerror同様，ユーザが任意に定義できる。

E. step

ステッパを呼び出すには，stepを呼ぶか，あるいはステッパ呼び出しのスイッチをオンにすれば良い。後者の場合には，refuteが，述語呼び出しの都度stepを呼び出す。stepの定義をユーザが変更することによって，MaclispのEVALHOOKと同様の機能が実現できるわけである。

Bでは記述されていないが，stepを毎回呼ぶためには，refufeを以下のように変更すれば良い：

```
refute ( * pred , * env )←
    get－def ( * pred , * def )∧
    if ( step- switch( ), step( ))∧
    if ( codep ( * def ), call ( * def ),
        refl ( * pred , * env , * def ) )
```

2.3.2 コンパイラ

コンパイラの役割は，一般に，処理速度の向上を目的とした機械命令への翻訳であり，この意味で特定のハードウェア仕様なしにコンパイラの詳細について抽象的な議論をするのは困難である。そこで，ここでは既存の論理型言語ＰＲＯＬＯＧ（Edinburgh版）のコンパイラを参考にし，論理型言語を処理効率の点で実用的にするための方法について論じる。

論理型言語の利点が，非決定性の記述にすぐれていることや，論理変数の持つ柔軟な性質などにあることはいうまでもないが，既存の逐次型計算機上でこれを実現しようとするとき，利用者から見たこれらの利点が，記憶領域の使用効率や実行速度の面で障害となる場合が多くある。実際，一般的な計算機プログラムで用いられているアルゴリズムの多くは決定的な性質のものであり，これらの利点が十分に生かされない。したがって，例えばＬＩＳＰのような決定的言語と同じような実行効率を期待できる目的コードを生成するためには，コンパイル時に非決定的な性質や論理変数の柔軟性をできるだけ排除する必要がある。ＰＲＯＬＯＧが実用言語として一般に受け入れられた理由の１つが，このような点に留意した処理効率向上の努力にある。

A．非決定性と決定性

一般の再帰的プログラミング言語と同様に，論理型言語においても，実行時環境の管理はスタックを用いて行うのが適切である。特に，論理型言語が静的な環境管理法（static scoping）をとるという点を考えると，動的な環境管理法（dynamic scoping）をとるＬＩＳＰよりも，スタックによる環境管理に適しているといえる。

一般の再帰的プログラミング言語では，手続き（procedure）からの復帰時に，その手続きの実行に要した局所的な環境を持つスタック・フレームをポップ・アップするが，論理型言語では一般的にこれをすることができない。これは，論理型言語の手続きが，通常，非決定的に解釈されるためである。このため論理型言語プログラムのすべての手続きを非決定的なものとして処理すると，本来決定的に解釈してよい手続きに関しても，後戻り（backracking）処理のためのオーバー・ヘッドが付加されてしまう。また，後戻りが起こるまで，途中で起動された手続きのスタック・フレームをポップ・アップすることができないため，記憶領域（スタック）が多量に消費されてしまう可能性がある。したがって，論理型言語の手続き呼び出しが，決定的に行われるのか，非決定的に行われるのかを判別することが処理効率の面から重要になる。この判定を実行時に行うことは勿論可能であるが，コンパイラでは，この決定性の判定をコンパイル時に細部にわたって自動的に行うことは困難であり，決定性を示唆する宣言などが必要となる。ＰＲＯＬＯＧで悪玉扱いされている〝cut〟オペレータは，決定性の宣言と解釈するならば，コンパイラにとって善玉の役割を演じることになる。

〝 cut 〞 オペレータの役割は，論理式の手続き的解釈のための補助というだけでなく，コンパイラに対する指示宣言子の１つとも考えられる。〝 cut 〞オペレータの排除が提案されているが，コンパイラ側から見ればこれにかわるguardのような宣言手段が望まれる。

B. last－call (tail－recursion)

決定性の判定は，最適化問題に多くの効果を持たらす。

$p \leftarrow g_1 \wedge g_2 \wedge g_3$

のような例において，pが決定的に起動されたとしよう。このとき，g_1 ，g_2 が決定的な手続きを呼び出してあれば，g_3 の起動的にpのスタック・フレームをポップ・アップすることができる。このような最適化をlast－call最適化と呼ぶ。

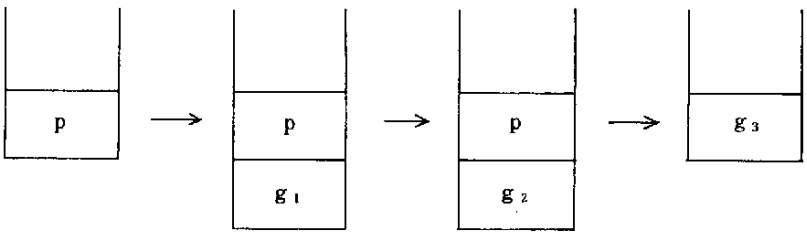


図2.3.1　last－call最適化の場合のスタックの状態

これに対して，g_1 ，g_2 が決定的な手続き呼び出しでなければ，スタックの状態は図2.3.2のようになり，スタック領域が多量に消費されてしまう。

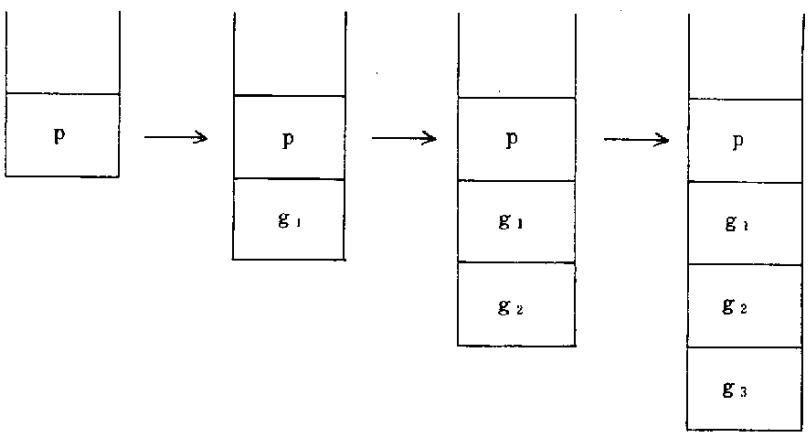


図2.3.2　last－call最適化ができない場合

この last－call の例で，g_3 がp自身であるとき，tail－recursive であるという。tail－recursion の最適化問題は，今や斬新しい問題ではないが，実際に用いられている例はきわめて少ないようである。論理型言語は，その言語スタイルと論理変数の評価を遅延するという性質のおかげで，tail－recursive なプログラミングが容易であり，また，そのような状況を検出することが比較的容易な言語構造をしている。

```
append ( <>, *X, *X )←
append ( *A. *B, *C, *A. *D )←append ( *B. *C, *D )
←append ( a.b.c. <>, <>, *R )
```

図 2.3.3　tail－recursive なプログラム例

図 2.3.3 は論理型言語による tail－recursive なプログラムの例である。

tail－recursion は，一般のプログラミング言語の loop 構造に相当する記述力を持つが，最適化を行うことにより，loop 構造と同等の効率のよい，目的コードを生成することが可能になる。

C.　local スタックと global スタック

論理変数が持つ評価を遅延するという効果は，言語の記述力という面で有用であるが，実行時の記憶領域の管理を複雑にする原因となっている。この現象は，ＬＩＳＰでいうFUNARG問題に類似している。スタックを用いて環境管理を行うＬＩＳＰ処理系では，upward FUNARGを生じた時に，その時の環境をa-リストにして heap 上へコピーするという方法をとる。この場合の heap 上のａ－リストの役割が，ここでいう global スタックの働きに似ている。すなわちＰＲＯＬＯＧでは，その手続きを起動した親の環境から参照される可能性のある変数をすべて global スタック上に割り当て，それ以外の変数を local スタック上に割り当てる。この判定は，次のように行う。すなわち，手続きの head 部で複合項（ compound term ）中に現われる変数は，親の環境から参照される可能性があるので global スタックへ，単独で現われる変数は，その可能性がないので local スタックへ割り当てる。したがって図 2.3.3 の append の例では，変数Ａ，Ｂ，Ｄは global スタックへ，変数Ｒ，Ｃ，Ｘは local スタックへ割り当てられる。これを示したのが図2.3.4－ａである。

local スタック・フレームは親の環境から参照されることがないため決定的な手続きからの復帰時にポップ・アップすることができるが，global スタック・フレームをポップ・アップすることはできない。global スタックは，後戻りが起るまでポップ・アップされることがない。また，global スタックは，last－call や tail recursion の最適化の対象とすることもできない。したがって，global スタックは，記憶領域を多量に消費する主要な原因と

なる。図2.3.4の例を注意して見ると解るように変数Bは必ずしもglobalスタック上にある必要はなく，localスタック上に置くことができる。このように，複合項中の変数でもlocalスタックに割り当てることのできるものがある。globalスタックによる記憶領域の消費を軽減するには，このように変数をよく吟味して，localスタックに割り当てることのできるものを判別する必要がある。globalスタックについては，不必要な変数の割り当てを避けるという消極的な方法でしか記憶領域の消費を抑制できない。この変数の判別をコンパイラ自身が行なうことは困難であり，変数割り当て方法を指示する宣言が必要である。

D. 入力変数と出力変数

ＰＲＯＬＯＧが成功した原因の１つに統一化（unification）アルゴリズムの簡略化があげられる。ＰＲＯＬＯＧプログラムをいろいろ吟味してみると，統一化の多くが単純変数に対する定数項の統一化であることが解る。これらの統一化は，単なる代入演算で代用できることが多い。変数に対する統一化が，代入演算で代用できるか否かを判別するには，その変数が入力変数として働くのか，出力変数として働くのかを知る必要がある。換言すれば，その変数が定数と統一化されるのか，変数と統一化されるのかを知らなければならない。Edinburgh版ＰＲＯＬＯＧでは，mode宣言を用意して，この問題に対処している。

```
←mode  append（＋，＋，－）
```

この例では，手続きappendのcall時に，第１，第２引数には入力定数が，第３引数には出力変数が与えられることを宣言するものである。

E. 今後の展望

Edinburgh版ＰＲＯＬＯＧでは，コンパイラにおける最適化に必要な情報の多くを，利用者による宣言に頼っている。このようにコンパイラに対する指示宣言をいろいろ用意し，木目細い最適化情報を与えられるようにするというのも１つの方向である。しかし，宣言を多様化させるという方法は，過度に行われると，利用者の負担を増す怖れがある，したがって，コンパイラの研究に際しては，安易に宣言を増すことをせず，最適化情報の抽出法についてさらに研究を進める必要がある。この方法として，データ・フロー解析，記号実行，連結グラフを用いた解析などを応用した技術の研究成果が持たれる。これらは論理型言語を手続き的に解釈した場合の最適化技法と考えられるが，さらに論理型言語の宣言的な性質を利用したソース・プログラム・レベルでの最適化技法についての研究も必要である。

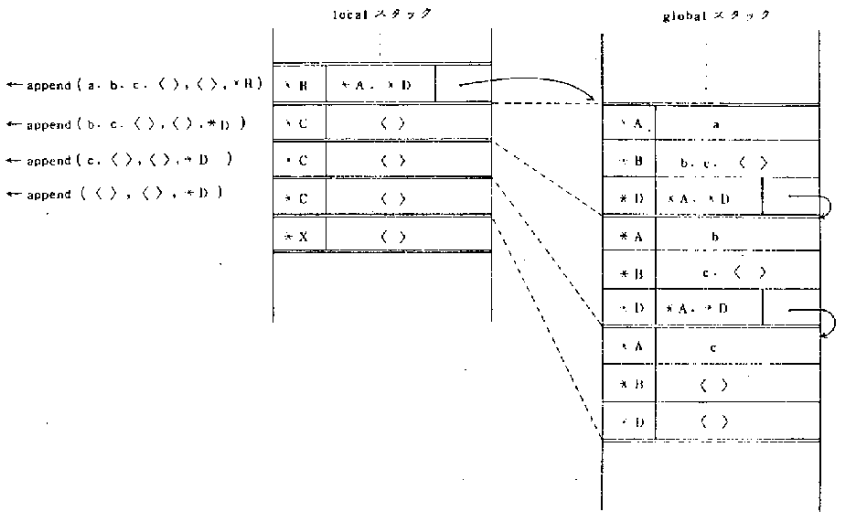


図 2.3.4 － a　append（図 2.3.3.）の実行時スタックの状態
（tail－recursion 最適化をしない場合）

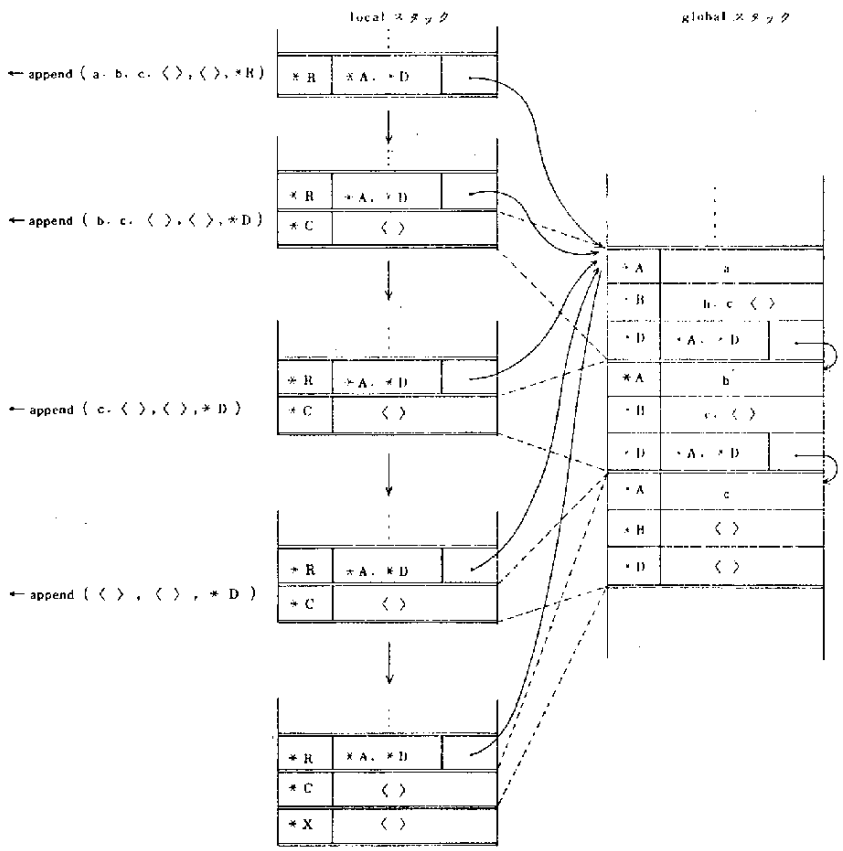


図 2.3.4 － b　tail－recursion 最適化をした場合のスタックの状態

2.4 データベース・サブシステム

2.4.1 内部データベース・システム

内部データベースは 'FGKLの計算遂行のためのデータベースである。

データベースの内容は 'FGKL の基本となる論理関数式である。いわゆる大域変数に含まれるデータはここでは扱わない(注)。また，計算途中での変数等は計算途中の作業域でのデータ処理なので問題としない。

内部データベースの処理機構には，論理関数式の検索，追加，変更，動的ないし仮想的な内部データベースを構成するコンテキスト機構がある。

具体的なハードウェアの支持機構としては検索に必要な（ハードウェア）ハッシュ機構，大規模な内部データベースを効率良く処理する仮想記憶機構が必要となる。

A. データベース・オブジェクト

内部データベース・オブジェクトである論理関数式は次のような内容から成っている。

① 論理式の名前

② 論理式の機能クラス(注)

③ 論理式の定義体

④ 2次記憶　（swap－in，swap－outのため）

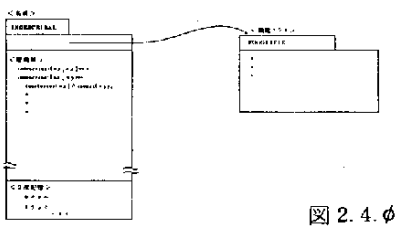


図2.4.φ　内部データベースの内容例

抽象データやモジュールなどの導入によって，内部データベースの要素も上記の基本要素だけではなくなる。

抽象データの導入に伴うもの。

① 抽象データの定義式

いわば型枠（template）を示すもの。型枠そのものが同じ成分の同じ配列から成っていても，操作が違うと別のデータだと思う。

実用的にはこの定義式がかなり複雑で膨大なものとなるため，何らかのクラス化機構（データ・クラスなど）が有用となるであろう。

注）大域変数をどう処理するかは　FGKLでも難かしいところである。論理式に含めるのも一方法か？

注）あるいは，他の抽象化機構のための論理単位

② 抽象データの実体

型枠に従って，システムの中で動的に作り出される。不必要になった時点で積極的に消されるべきなのか，メモリ管理ルーチンが適当な時期にガーベジ・コレクションすべきなのかは用途によって異なるだろうが，システムとして考慮の余地がある。

③ モジュール

モジュールに属する述語のリスト，モジュール内で共通に使われる変数の属性宣言などが必要となる。

利用上は述語の方からその述語が属しているモジュールを検索する必要があるだろう。

B. データベース機構

① 名前のハッシング

論理関数式の名前は 'FGKLの計算メカニズムの重要事項なのでハッシングにより高速化をはかる。

② 機能クラス

機能クラスの内容は包含関係を有し，これ自体が小さなデータベースという形をもつ。機能クラスの内容は引数に対する処理（evaluation）と，この引数のデータ型に対する検査項目および，不適合なデータ型をもつ引数に対する処理とである。

これらの内容の集合体が1つの機能クラスとなるので，各機能クラスは内容集合へのポインタをもつことになる。

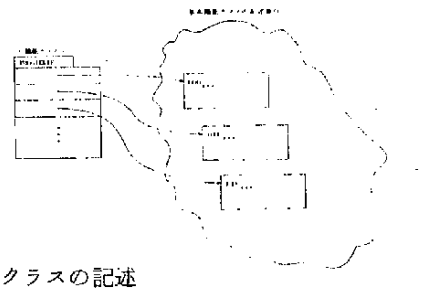


図 2.4.φφ 機能クラスの記述

論理式の機能クラスが（値域の拡大などによって）変更されることはありうるが，機能クラス自体が動的に変化することはない。したがって上記の内容をコンパイルして，手続き化（マイクロ・コンパイル）することもできる。

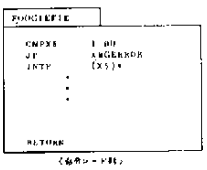


図 2.4. φφφ コンパイル化された機能クラス記述

③ 論理関数の定義体

論理関数の定義体はいわゆるホーン節の集合となる。集合の要素である各ホーン節には'ＦＧＫＬの定める規則に従って順序関係がついている。

エディタないしはプログラムによってこの定義体の内容は動的に追加，削除，変更の処理が行なわれる。

順序づけはホーン節の左辺の内容に従って整理されることが望ましい。同一の左辺に対して異なる複数の右辺が存在することはプログラムの虫だと考えられるから。

C　ハードウェア・サポート

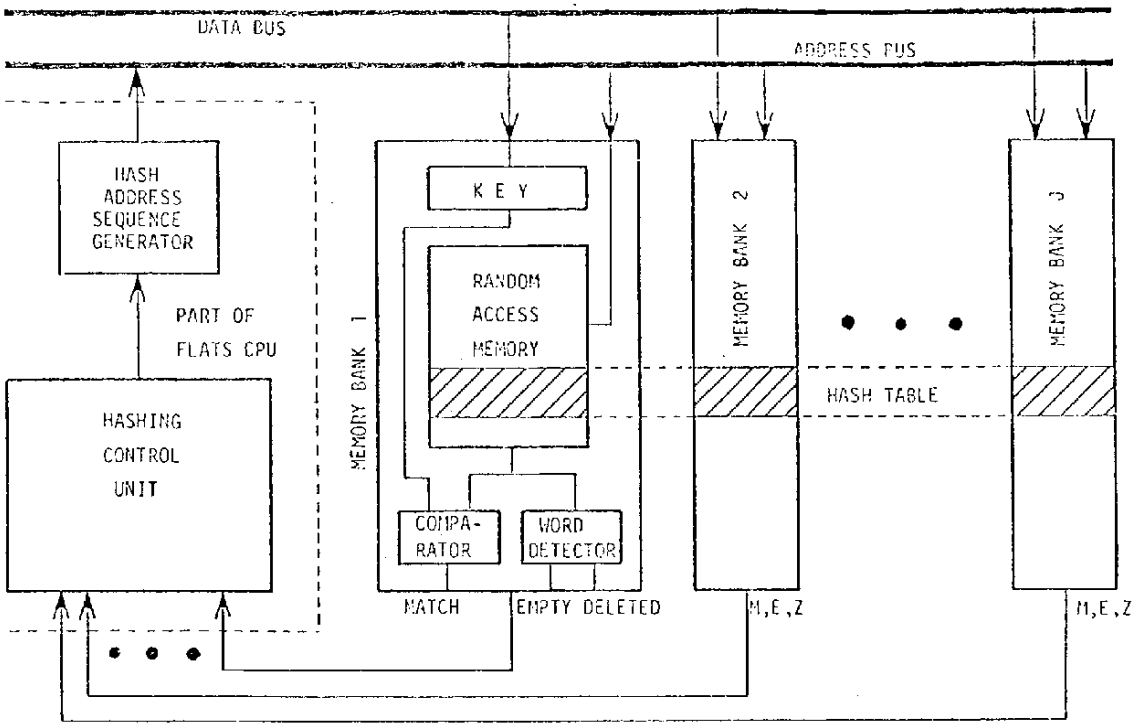
内部データベースのハードウェアには，論理式の述語名を検索するハッシング・ハードウェア，内部データ型の検出ハードウェアが主たるものである。

ハッシング・ハードウェアには理科学研究所や東京大学で開発されたＦＬＡＴＳ用の並列ハッシング・ハードウェア（図 2.4.1 ）が参考になる。ＦＬＡＴＳの場合にはリスト構造を含めたハッシングが計算機構（連想計算 associative computation）の上からも必要となっていた。一方，'ＦＧＫＬにおいては，ユニフィケーションの関係で連想計算が直接には使えない。もっとも，論理式全体の等価値性を判定するためにハッシングを使うことはできる。ただし，結果的にどれだけ計算時間の短縮化がはかれるかどうかは定かではない。

またハッシングの対象を述語名に限定するなら，ハッシング回路をかなり簡潔化することができる。

内部データ型の検出ハードウェアは，基本計算メカニズム（ 2.2.2 Ｂ ）を支持する'ＦＧＫＬ／Ｓマシンのマイクロプログラム上で実現され，活用される。

ユーザが抽象的に定義したデータ型についても動的に適応する機能をもつハードウェアというものが考えられはするが，'ＦＧＫＬ／Ｓ上で実現するのは，当面は困難であろうから考えない。



〔Goto 他 1979〕より

図2.4.1 Parallel Hashing Hardware

参考文献

1) E. Goto et al, "FLATS, A MACHINE FOR NUMERICAL, SYMBOLIC AND ASSOCIATIVE COMPUTING", Proc 6th IJCAI, pp. 1058－1066

2.4.2 外部データベース・システム

A 外部データベース・システムの基礎

a 関係データベース・システムとロジック・プログラミング

電子計算機を用いるデータ処理の歴史は，ファイル処理中心から外部データベース・システム（ＥＤＢＳ：External Data Base System ）へと発展してきた。その流れをデータ構造という側面から分類してみると，主としてネットワーク型ＥＤＢＳ，階層型ＥＤＢＳ，関係型ＥＤＢＳ，の３つの方式に整理される。各方式はそれぞれ長短をもつが，本節では以下の理由で関係型ＥＤＢＳのみを取上げる。

(a) 第５世代コンピュータ・システム（ＦＧＣＳ：Fifth Generation Computer System ）で採用した第５世代核言語（ＦＧＫＬ：Fifth Generation Kernel Language）が，Ｐｒｏｌｏｇ ベースの論理型言語であること。

(b) 論理でいう関係と関係モデルでいう関係データベース（ＲＤＢ：Relational Data Base ）が，モデル論的アプローチを採用すれば，概念的にも１対１に対応すること。

(c) 論理型言語の採用によりＲＤＢ，知識ベース（ＫＢ：Knowledge Base ），推論機構の知識表現を一様ならしめること。

(d) 論理型言語は堅固な数学的基盤というしっかりした理論的背景をもっているので，その結果，透明な記述力（分り易さ，表現し易さ，簡潔さ）を提供すること。

関係モデルは１９７０年のE. F. Coddの論文[1]に端を発し，データ処理の世界の経験的／直感的職人芸を，計算機科学／工学という学問の世界へ昇華させるという功績を残した。現在は関係モデルの基礎理論の樹立と本格的な関係データベース管理システム（ＲＤＢＭＳ：Relational Data Base Management System ）の開発の方向へと２極分化[2]しつつある。この２極分化を食止め，ＲＤＢとロジック・プログラミングのインタフェースの仕様を提案するのが，本項の役割である。ＦＧＣＳに対する要求分析からすれば，ここでは次の２種のＥＤＢＳとのインタフェースを考えれば十分である[3]。

(1) ＥＤＢＳ－Ⅰ （試作ＲＤＢマシンとの結合）

ＦＧＣＳ研究開発課題となっているデータベース・マシンの試作機であるＲＤＢマシンと接続できるものであること。

(2) ＥＤＢＳ－Ⅱ （逐次型推論マシンの外部にあるＲＤＢとの結合）

汎用大型機上の大規模ＲＤＢＭＳが利用できるようなインタフェースをもつこと。

さてＥＤＢＳとしてのＲＤＢに対する基本操作にはデータ定義言語（ＤＤＬ：Data

Definition Language ）とデータ操作言語（ＤＭＬ：Data Manipulation Language）がある。ＦＧＣＳユーザにとって，よりフレンドリなＤＭＬである問合せ言語（Query Language ）を，数学的基盤を背景に分類[2]すれば，例えば要素操作型，関係代数型，関係論理型，写像型，例題型，自然言語型等がある。本項では 'ＦＧＫＬ／ＳマシンとＥＤＢＳ－Ⅰ，ＥＤＢＳ－Ⅱとのインタフェースをとることを主眼に，ロジック・プログラミングの原点とＲＤＢ提唱者であるCoddの主張の原点との共通部分をとり，関係代数型と関係論理型の問合せ言語について 'ＦＧＫＬ／Ｓマシンの立場からの仕様提案を行う。すなわちＥＤＢＳの親言語としては，関係代数型と関係論理型の述語を装備した問合せ言語を考えることにする。

b　関係データベース・モデルの基本概念とロジック・プログラミング

(a)　関係データベースの集合論的意味

互いに識別可能な属性と呼ばれる軸a_i（$i\in\{1,\cdots,n\}$）があり，各a_iに対応し，定義域と呼ばれる集合D_i（$i\in\{1,\cdots,n\}$）が存在するものとする。a_iを要素とする集合$\mathfrak{A}\triangleq\{a_1,\cdots,a_n\}$のことを属性集合と呼ぶ。各$d_i$を，それぞれ$D_i$の要素とする時，n項リスト$(d_1,\cdots,d_n)$を$\mathfrak{A}$上の（n－）タップルと呼び，集合$\{(d_1,\cdots,d_n)\mid(d_1\in D_1)\wedge\cdots\wedge(d_n\in D_n)\}$を$\mathfrak{A}$上の直積（Cartesian Product）と呼び，$\prod_{i=1}^{n}D_i$あるいは$D_1\otimes\cdots\otimes D_n$と表記する。直積$\prod_{i=1}^{n}D_i$の部分集合$R(\subseteq\prod_{i=1}^{n}D_i)$のことを，n個の定義域$D_1,\cdots,D_n$に関するn項関係（n－ary relation）あるいは単に関係*という。$\mathfrak{A}$の部分集合Aが$\{a_{i_1},a_{i_2},\cdots,a_{i_m}\}$と表わされる時，タップル$d=(d_1,\cdots,d_n)$のA上への射影は，次のように定義される：$d.A\triangleq(d_{i_1},\cdots,d_{i_m})$。

以上の概念構成で注意しなければならないのは，属性名と属性値，関係名と関係の実体とを明確に区別して扱わなければいけないことである。

(b)　関係データベースの１階論理的意味

'ＦＧＫＬ／ＳのベースとなったPrologが拠所とする論理は，１階述語論理（１ＰＬ：First－order Predicate Logic）の部分クラスである１階Horn 論理である。ところで前述のＲＤＢ概念構成法で明らかなように，ＲＤＢモデルは複数の定義域に関する直積の部分集合である関係の集合として構成されているので，単一の定義域しか

注）Codd の定義した関係[4]においては，属性の順序やタップルの順序は自由に入換えてよいことになっている。この点まで含めて概念設定することも可能である[5]が，ここでは簡単のため必要最小限の表記法や諸定義のみ導入する。

取扱わない１ＰＬでは，厳密なフィットした概念構成法を提供するのに必ずしも適切でない。ＲＤＢモデルを論理的に正当化するのに最も適切な論理系は，Ｈｏｒｎという制約のついた１階多類論理（１ＭＬ：First-order Many－sorted Logic）[6] であると考えられる。１ＭＬの構文法，構造等の厳密な議論は文献[6]に譲るとして，ここでは関係の意味をロジック・プログラミングの立場から正当化するのに必要最小限の事柄を述べる。

類（ｓｏｒｔ）ｉからなる類集合Ｉ（｜Ｉ｜＝ｍ），素類 i_0（$\in I$），類対 $\sigma_I \triangleq < i_1, \cdots, i_n > (i_1, \cdots, i_n \in I)$，$i_{n+1} \in I$，$n \in \{0\} \cup N$（Ｎ:自然数の集合），$j \in N$ という記号類を準備する。１ＭＬの構文法を，次のようにして帰納的に定める。

〔項〕 1. 類ｉの各個体定数 c^i と各個体変数 x^i は，ともに類ｉの項である。2. もし $t_1^{i_1}, \cdots, t_n^{i_n}$ がそれぞれ類 $i_1, \cdots, i_n$ の項であり，$f^{\sigma_I : i_{n+1}}$ が類対 σ_I から類 i_{n+1} へのｎ引数関数定数ならば，$f^{\sigma_I : i_{n+1}}(t_1^{i_1}, \cdots, t_n^{i_n})$ は類 i_{n+1} の項である。3. もしＷが命題で，t_1^i と t_2^i が類ｉの項ならば，（if W then t_1^i else t_2^i）は類ｉの項である。

〔素命題〕 1. ＴとＦは素命題である。2. 各命題定数 pi_0 は素命題である。3. もし $t_1^{i_1}, \cdots, t_n^{i_n}$ がそれぞれ類 $i_1, \cdots, i_n$ の項であり，$p^{\sigma_I : i_0}$ が類対 σ_I 上のｎ引数述語定数ならば，$p^{\sigma_I : i_0}(t_1^{i_1}, \cdots, t_n^{i_n})$ は類対 σ_I 上の素命題である。4. もし t_1^i と t_2^i が類ｉの項ならば，$t_1^i \approx_i t_2^i$ は素命題である。

〔命題〕 1. 各素命題は命題である。2. もし W_1，W_2，や W_3 が命題ならば，$\sim W_1$，$W_1 \supset W_2$，$W_1 \wedge W_2$，$W_1 \vee W_2$，$W_1 \equiv W_2$，や（IF W_1 THEN W_2 ELSE W_3）も命題である。3. もしＷが命題であり，x^i が類ｉの個体変数ならば，$\forall x^i W$ や $\exists x^i W$ も命題である。ただし簡単のため，このルールは $\forall x^i$ や $\exists x^i$ がＷに出現しない場合のみ適用されることにする。

このようにして与えられた１ＭＬの構文法に対して，その構造 $\mathcal{M}$ は次のように与えられる。

〔構造 $\mathcal{M}$〕 1. 各類ｉ（$\in$Ｉ）に対して，$\mathcal{M}$ は類解釈領域 U_i を割当てる。2. 各ｎ引数述語定数 $p^{\sigma_I : i_0}$ に対して，$\mathcal{M}$ は $U_{i_1} \times \cdots \times U_{i_n}$ の全述語を割当てる。3. 各ｎ引数関数定数 $f^{\sigma_I : i_{n+1}}$ に対して，$\mathcal{M}$ は $U_{i_1} \times \cdots \times U_{i_n}$ から $U_{i_{n+1}}$ へ写像する全関数を割当てる。4. 各個体定数 c^i に対して，$\mathcal{M}$ は U_i に属するある点を割当てる。5. 自由に出現する各個体定数 x^i に対して，$\mathcal{M}$ は U_i に属する点を割り当てる。

(c) 非評価方式と評価方式

知識情報処理システム（KIPS：Knowledge Information Processing System）のサブシステムである問題解決推論システムの構成要素としてRDBを考えるとき，1PLあるいは1階Horn論理での証明法を活用することが必須である。これには，次の2つのアプローチがある[8]。

① 非評価方式（non－evaluational approach）：RDBは他の一般的知識と共に，ある形式論理体系の公理（axiom）を成すと考える。従って推論方式としては，1PLで確立されたRobinsonの分解証明法にもとづくTheorem Proverテクニック，Prolog流にいえばある種のパターン・マッチングにもとづく手続き的解釈に準拠し証明していく。

② 評価方式（evaluational approach）：RDBと他の一般的知識は全く別個のものとみなし，一般的知識からなるKBは，ある形式論理体系の公理を成すと考える。このときRDBはその理論のひとつの解釈（interpretation），すなわちモデル（model）をなすと考える。評価方式での推論方式は，RDBとKBとの分離とそれに伴うIntensional EvaluatorとExtensional Evaluatorの役割分担に，その特徴がある[8]。ここにIntensional Evaluatorは，問合せを通常のEDBSで処理可能なコマンドへと次々に変換していく専用の解探索プラン生成機であり，Extensional Evaluatorとは通常のEDBSのことである。具体的なシステム構成については後述する。

(d) 関係データベースとロジック・プログラミング

以上の準備の下に，RDBとロジック・プログラミングの橋渡しをする。

RDBへの演繹的質問応答（QA：Question Answering）システム研究には，古川[9]，Reiter[10]，Chang[8]，國藤・若木[7]らの研究があるが，これらはいずれも前述の評価方式のEDBS設計方法論にもとづいて考察している。ReiterのIntensional Data BaseとExtensional Data Base，ChangのVirtual RelationとBase Relationは，それぞれ本項の文脈ではKBとRDBに対応しているが，これらはどれも1PLの命題のみを知識とする計算機処理を前提としている。1PLの命題のみを利用するという制約を設ける理由は，ある知識が充足不能かどうかを定める部分的決定手続きが，現状の計算機処理システムの機構を前提に確立しているからである。

RDBを集合論的見地でとらえた時，関係の特性は内包（Intension）と外延

（Extension）で把握される[7]。関係の内包では，その関係に固有な普遍な意味的制約であり，関係の外延とは，その関係の実例であるタップル集合のことである。他方ＲＤＢを１ＰＬ的見地でとらえた時，タップルｄがある関係Ｒに属することが述語Ｒ(d)に真値ｔｒｕｅを割当てることに，タップルｄがその関係Ｒに属さないことが述語Ｒ(d)に偽値ｆａｌｓｅを割当てることに対応する。すると関係の外延とは，１ＰＬでのモデルを規定する。証明論的には，定数記号のみを引数とする正の単位リテラル（positive unit literal）のみからなる節集合のことである。関係の内包とは，解釈が変化しようと，そのようなモデル上で，常に真となる公理，通常proper axiom と呼ばれるが，によって規定される。関係の内包と外延をこのように定めた時，関係の更新はモデルの変化に相当する。ここで述べたインフォーマルな議論に従えば，ＲＤＢをロジック・プログラミングの立場からみた時，次のようなことに留意しなければならない。

① 集合としてのＲＤＢは，１ＭＬでは述語定数に対して割当てられた全述語に相等する。従って定義域についても，任意のｊ（∈{１，…，ｎ}）について$U_{ij}=D_j$が成立する。

② ＲＤＢモデルではＲＤＢのスキーマとインスタンスを区別して取扱う。ＲＤＢスキーマで扱う対象は，関係の族（ｆａｍｉｌｙ）なので２階述語／多類論理を用いて考察すべきである。しかしながらＲＤＢインスタンスで扱う対象は，個々の関係なので１ＰＬ／１ＭＬを用いて考察するのが妥当である。

③ 論理における証明論的見地からすれば，ＲＤＢとはどの項も定数記号である正単位リテラルからなる節の集合のことである。

④ “否定的情報（Ｎｅｇａｔｉｖｅ Ｉｎｆｏｒｍａｔｉｏｎ）”を論理でどのように表現するかは，Ｐｒｏｌｏｇでの“ｎｏｔ の扱い”問題と関連し，極めて難しい問題を派生する。基本的には“真であると証明されるもののみ真であるとみなす”開世界仮説（ＯＷＡ：Ｏｐｅｎ Ｗｏｒｌｄ Ａｓｓｕｍｐｔｉｏｎ）という立場にたつか，“真であると証明されないものは偽であるとみなす”閉世界仮説（ＣＷＡ：Ｃｌｏｓｅｄ Ｗｏｒｌｄ Ａｓｓｕｍｐｔｉｏｎ）という立場にたつかの選択の問題[10),5)]である。この問題の論理学からの解釈については，3.1を参照されたい。

c ’ＦＧＫＬ／Ｓでの関係データベース表現

親言語としての論理型言語’ＦＧＫＬ／Ｓに，関係論理（relational calculus）型と関係代数（relational algebra）型の問合せ言語を埋め込む方式について考察する。

(a) 関係論理的アプローチ

論理型言語 'FGKL／Sに関係論理型問合せ言語を埋込むのは，極めて自然な考えである。[注)] RDBの各タップルを 'FGKL／Sの表明節に対応づけることになる。しかしながらHorn節を主体とした論理型言語に，Coddの1PLをベースとした関係論理を埋込むには，幾つかの注意が必要である。例えば，次のような問題に対する配慮が要する。

① notの扱いが両者の間で全く異なっている。[注)]

② 親言語として十分な機能を提供しているか。例えば関係完備性[4)]を保証しているか。

③ RDB更新をどのように扱うのか。

関係論理を 'FGKL／S に埋込むにも，次の2つの方式が考えられる。これらはともに，基本的に関係を述語で表わし，関係の操作を述語操作で行うものである。

(i) 属性値変数／定数を使う方法

関係を述語とみなすやり方である。"タップル＜a，b，c＞が関係rに属する"ことを"r(a，b，c)←"と表記する。CoddのRDBモデルでは，更に属性やタップルの出現順序は随意であるという条件がつくが，この表記法ではそれらの出現順序は，手続き的解釈の過程で有意である。

"＜a，b，c＞，＜a'，b'，c'＞∈r"の意味は通常，"r(a，b，c)← r(a'，b'，c')← "と解釈されるが，"r(∗x，∗y，∗z)↔(∗x＝a∧∗y＝b∧∗z＝c)∨(∗x＝a'∧∗y＝b'∧∗z＝c)"[注)]という解釈の仕方もある。後者の"iff"解釈問題については，文献[1)]を参照されたい。

(ii) タップル変数／定数を使う方法

タップルを引数とする1変数述語で関係を表現するやり方で，Coddの関係論理により近い表現法を与える。"＜a，b，c＞∈r"ことを"r(＜a，b，c＞)←"と表記する。この方法もタップル定数や変数を直接扱えるため，属性の個数が多数ある関係表現等において便利な場合がある。

(i)，(ii)両方式の相互変換は容易である。例えばbからaへの変換には"r_a(∗x，∗y，∗z)←r_b(＜∗x，∗y，∗z＞)"とし，aからbへの変換には"r_b(＜∗x，∗y，

注) non－Hornの1PL命題も，このやり方では表現できる。

注) Prologのnotを用いると，"p(a)← r(∗x)← not〔p(∗x)〕"という節集合を与えて，"?←r(b)"という質問をすると"yes"という解答が，"?←r(∗x)"という質問をすると"no"という解答がフィードバックされる。

注) この"iff"解釈だと更新(update)の意味付けが困難だし，Horn論理の枠内では記述できないことが多い。

＊z＞）←r_a（＊x，＊y，＊z）”という述語を実行すればよい。従って，どちらか一方をサポートすればよいが，本項では主として，方式(i)を用いて考察することにする。

(b) 関係代数的アプローチ

論理型言語‘ＦＧＫＬ／Ｓに関係代数的問合せ言語を埋込むのは，多少不自然な所もある。しかしながらハードウェア特性やＥＤＢＳとの結合を考慮すれば，関係代数レベルの集合演算をインタフェースとすることは，性能向上の目途がつきやすく，また将来的にも並列型問題解決推論マシンの研究開発へと発展していく可能性が高いので，極めて魅力あるアプローチである。

このアプローチでは関係をひとつの項として扱うので，抽象データ型に“関係型”なる型を導入しなければならない。そこで‘ＦＧＫＬ／Ｓの型定義や宣言方式との整合性が必要となる。特別な場合として，関係型をタップルのリスト（a list of tuples）として表現すると約束すれば，関係代数の基本演算を全てリスト処理で実現することができる。このタップル・リスト型を関係型そのものとみるか，関係型を別に作って，タップル・リスト型との相互変換述語を作るかは，意見の分れるところであるが，‘ＦＧＫＬ／Ｓでは既存Prolog処理系の現状を考慮し，主として前者の考えで概念を整理しておくことにする。

以上述べてきたアプローチ(a)，(b)を融合する方式として，関係型とタップル・リスト型との相互変換を行う双方向性の高階述語“relof（＊relation，＊a-list-of-tuples）”を準備しておくと便利がよい。

B 基本操作

a 関係モデルの定義

(a) 関係の宣言

関係インスタンスの宣言は，次のような述語にもとづき行う。

```
define-relation〔関係名（属性名1：定義域型1，属性名2：定義域型2，…，
属性名n：定義域型n）where （節1，節2，…，節n）←〕←
```

ここに節1，…，節mは関係名を含む節で，その関係が常に満足する論理的条件を示す。

関係スキーマの宣言も，同様な述語にもとづき行う。

```
define-type〔Reltype，関係名（属性名1：定義域型1，…，属性名n:定義域型n）
where （節1，…，節m）←〕←
関係名1（Reltype）←
関係名2（Reltype）←
```

これにより同じ枠組をもつ複数の関係を宣言する機能も，ユーザに提供することになる。

(b) 関係の表記

ＥＤＢＳとのインタフェースをとるために関係宣言では属性名も記述するが，関係そのものを述語表現する際には，冗長かつ見難いので，必要に応じ属性名を省略することにする。ただしその場合，関係宣言に出現する属性順にある定義域型上の値をとりうる属性値変数／定数が並んでいるものとする。従って，例えば次の述語表記は全て同等である：

r (a_1 : *x, a_2 : *y),

r (a_2 : *y, a_1 : *x),

r (*x, *y) .

b 関係代数サポート

関係代数をサポートする述語を列記し，それらの 'ＦＧＫＬ／Ｓでの表記法を導入し，その意味を説明し，また可能ならばその 'ＦＧＫＬ／Ｓでの定義を述べる'[12]。注）

(a) 集合演算

① 基礎になる演算を表わす述語

(i) メンバシップ (membership) 述語

member (*element, *set)
ある要素*element が集合*setに属するかどうかを判定する述語（集合論的表記法によれば，*element∈*set ）
member (*e, *e . *r)←
member (*e1, *e2 . *r)←member (*e1, *r)

(ii) 部分集合 (subset) 述語

subset (*subset, *set)
集合*subset が集合*setの部分集合であるかどうかを判定する述語 (*subset⊆*set)
subset (<>, *s)←
subset (*e . *r, *s)←member (*e, *s)∧subset (*r, *s)

注）本項では基本論理式で表記すると約束したモジュール名は，繁雑なので省略する。

(iii) 等価集合(set equality)述語

seteg (* set1, * set2)
集合 * set1 と集合 * set2 が等しいかどうか判定する述語 (* set1 = * set2)
seteg (* s1, * s2) ← subset (* s1, * s2) ∧ subset (* s2, * s1)

② 通常の集合演算を表わす述語

タップルのリスト型を引数として処理する関係代数型の集合演算述語(注)について述べる。

(i) 和集合(set union)述語

union (* set1, * set2, * set)
集合 * set1 と集合 * set2 の和集合が * set かどうか判定する述語 (* set1 ∪ * set2 = * set)
union (<>, * s, * s) ← | union (* e. * r, * s, * a) ← member (* e, * s) | union (* r, * s, * a) union (* e. * r, * s, * e. * a) ← | union (* r, * s, * a)

(ii) 差集合(set difference)述語

difference (* set1, * set2, * set)
集合 * set1 と集合 * set2 の差集合が * set かどうか判定する述語 (* set1 − * set2 = * set)
difference (<>, * s, <>) ← | difference (* e. * r, * s, * a) ← member (* e, * s) | difference (* r, * s, * a) difference (* e. * r, * s, * e. * a) ← | difference (* r, * s, * a)

注) 関係論理型の集合演算述語については, 2.4.2.B.c で述べる。

(iii) 共通部分集合 (set intersection) 述語

intersection (* set1 , * set2 , * set)
集合 * set1 と集合 * set2 の共通部分集合が * set かどうか判定する述語 (* set1 ∩ * set2 = * set)
intersection (<> , * s , <>) ← | intersection (* e . * r , * s , * e . * a) ← member (* e , * s) | intersection (* r , * s , * a) intersection (* e . * r , * s , * a) ← | intersection (* r , * s , * a)

(iv) 直積 (Cartesian product) 述語

cartesian (* set1 , * set2 , * set)
集合 * set1 と集合 * set2 の直積が * set かどうか判定する述語 (* set1 ⊗ * set2 = * set)

(b) 関係演算

(i) 射影 (projection) 述語

projection (* rel , * attrs , * result)
関係 * rel の属性集合 * attrs 方向への射影が * result かどうか判定する述語 (* rel [* attrs] = * result)

関係宣言述語により p (a_1 : * t_1 , a_2 : * t_2 , …, a_n : * t_n) や q (a_{i1} : * t_{i1} , …, a_{im} : * t_{im}) (m ≦ n) が既知のときは, " projection (p , q) " という略記法も許すことにする。

(ii) 結合 (join) 述語

join (* rel1 , * attr1 , θ , * rel2 , * attr2 , * result), ただし θ は " =, ≠, <, ≦, >, ≧ " のどれかとする。
関係 * rel1 の属性 * attr1 と関係 rel2 の属性 * attr2 の間の θ 一結合した関係が * result かどうかを判定する述語 (* rel1 [* attr1 θ * attr2] * rel2 = * result)

θが"＝"の場合を自然結合（natural join）といい，RDBに対する問合せコマンドとしては最もよく使われる。従って述語 naturaljoin(*rel1, *attr1, *rel2, *attr2, *result)を，独立したコマンドとしておくのが望まれる。

自然結合と直積，共通部分，制約等との間のカテゴリー論的構造については，文献5)を参照されたい。

(iii) 制約（restriction）述語

restriction (*rel1, *attr1, θ, *attr2, *result)
関係*rel1の属性*attr1の値と属性*attr2の値同志がθという2項述語を満たすθ－制約関係が，*resultかどうかを判定する述語
(*rel1〔*attr1 θ *attr2〕＝*result)

(iv) 商（division）述語

division(*rel1, *attr1, *rel2, *attr2, *result)
関係*rel1の属性集合*attr1方向の射影を関係*rel2の属性集合*attr2方向の射影で割算を行い，その結果の商が関係*resultかどうか判定する述語
(*rel1〔*attr1÷*attr2〕*rel2＝*result)

(c) 集合の性質を操作する述語

(i) 性質判定述語

some (*property, *set)
集合*setのある要素が，性質*propertyを満足するかどうか判定する述語
some (*p, *e. *r)←apply (*p, (*e))
some (*p, *e. *r)←some (*p, *r)

(ii) 性質集合生成述語

pset (*property, *set, *subset)

集合＊setからある性質＊propertyを満足する要素を取出し，そのような要素からなる部分集合＊subset を生成する述語
pset (＊p , <> , <>) ← |
pset (＊p , ＊e.＊r , ＊e.＊a) ← apply(＊p, (＊e)) | pset(＊p, ＊r, ＊a)
pset (＊p , ＊e.＊r , ＊a) ← | pset (＊p , ＊r , ＊a)

(iii) 高階作用述語

apply (＊l , ＊args1)
第2引数＊args1の要素のそれぞれに＊lを作用させた結果（真または偽）を返す述語
apply(＊l , ＊a1) ← ＊l = . .＊p.＊a2 ∧ append (＊a1 , ＊a2 , ＊a) ∧ ＊nl = . .＊p.＊a ∧ ＊nl 注）

(d) その他の集合操作述語

(a)～(c)以外の集合を操作する述語で標準装備しておいた方がよいものにソート／マージ述語がある。これには種々のヴァリエーションが考えられるので，最も簡単なソート（ sort ）述語を例示するにとどめる。

```
sort(＊x, ＊r , ＊order) ← relof ( ＊r , ＊x) ∧ sorted(＊x , ＊order )
sorted (<>, ＊order) ←
sorted ( ＊n.<>, ＊order ) ←
sorted ( ＊n. ＊m. ＊l , ＊order ) ← ＊order ( ＊n , ＊m ) ∧
                                    sorted(＊m.＊l , ＊order)
```

ここに＊orderは第1引数と第2引数があるオーダを満足すれば真値を返す述語である。例えば，次のような述語である。

$*order(*x_1.\cdots.*x_n.<>, *y_1.\cdots.*y_n.<>) \leftarrow *x_1 \leqq *y_1$

(a)②で述べた集合演算述語や(b)で述べた関係演算述語の全ては，最後の引数＊resultが結果の値を返す役割にしか使われない場合は，むしろevaluable predicatesとして，その引数を陽に表現しない表記法を採用した方が便利である。例えば射影の場合，"＊result is projection(＊rel , ＊attrs)"となるが，これはまさに直接，関係

注）"＝．．"の意味はエジンバラ大学版Prologマニュアル[14]を参照のこと。

代数のコマンド“ *rel〔 *attrs 〕” に変換可能である。

c 関係論理サポート

Prolog系言語は関係論理となじみやすい。そこで FGKL／Sという論理型言語を用いて，集合演算や関係演算につながる概念を記述することは易しい。ここではそのような基本となる関係論理サポート述語と拡張された関係論理サポート述語について述べる。

(a) 集合演算

(i) 論理和述語

r(*tuple)
所与の関係p(*tuple) と関係q(*tuple) の論理和が関係r(*tuple) かどうか判定する述語
r(*tuple)←p(*tuple) r(*tuple)←q(*tuple)

(ii) 論理差述語

r(*tuple)
所与の関係p(*tuple) と関係q(*tuple) の論理差が関係r(*tuple) かどうか判定する述語
r(*tuple)←p(*tuple)∧not〔q(*tuple)〕

(iii) 論理積述語

r(*tuple)
所与の関係p(*tuple) と関係q(*tuple) の論理積が関係r(*tuple) かどうか判定する述語
r(*tuple)←p(*tuple)∧q(*tuple)

(iv) 論理直積述語

r(*tuple1 ∩ *tuple2) 注)
所与の関係p(*tuple1)と関係q(*tuple2)の論理的な直積がr(*tuple1 ∩ *tuple2)かどうか判定する述語
r(*tuple1 ∩ *tuple2)←p(*tuple1)∧q(*tuple2)

(b) 関係演算

(i) 論理射影述語

$q(a_{i_1} : *x_{i_1}, \cdots, a_{im} : *x_{im})$
所与の関係$p(a_1 : *x_1, a_2 : *x_2, \cdots, a_n : *x_n)$の属性集合$(a_{i_1}, \cdots, a_{im})$方向への射影が$q(a_{i_1} : *x_{i_1}, \cdots, a_{im} : *x_{im})$かどうか判定する述語
$q(a_{i_1} : *x_{i_1}, \cdots, a_{im} : *x_{im}) \leftarrow p(a_1 : *x_1, \cdots, a_n : *x_n)$ あるいはそあるいはその略記法として, $q(*x_{i_1}, \cdots, *x_{im}) \leftarrow p(*x_1, \cdots, *x_n)$

属性名と定義域型を並記する表記法は，属性の名前替え，重複，拡大といった属性に対する操作を表現するのに便利である。

例えば，

① $q(b_1 : *x_1, b_2 : *x_2) \leftarrow p(a_1 : *x_1, a_2 : *x_2)$

② $q(a_1 : *x_1, b_1 : *x_1, a_2 : *x_2) \leftarrow p(a_1 : *x_1, a_2 : *x_2)$

③ $q(a_1 : *x_1, a_2 : *x_2, a_3 : *x_3) \leftarrow p(a_1 : *x_1, a_2 : *x_2)$

ただし③においては属性a_3の型t_3に属する値が個体変数$*x_3$の値として代入されるものとする。

(ii) 論理結合述語

$r(*x_1, \cdots, *x_i, \cdots, *x_n, *y_1, \cdots, *y_j, \cdots, *y_m)$
所与の関係$p(*x_1, \cdots, *x_i, \cdots, *x_n)$と関係$q(*y_1, \cdots, *y_j, \cdots, *y_m)$の論理的な$\theta$－結合関係が$r(*x_1, \cdots, *x_j, \cdots, *x_m, *y_1, \cdots, *y_j, \cdots, *y_m)$

注) *tuple1 ∩ *tuple2は，*tuple1と*tuple2の連接(concatenation)タップルを意味する。

かどうか判定する述語
r(*x_1, …, *x_i, …, *x_n, *y_1, …, *y_j, …, *y_m)←p(*x_1, …, *x_i, …, *x_n)∧q(*y_1, …, *y_j, …, *y_m)∧θ(*x_i, *y_j) ここにθ(*x_i, *y_j)はevaluable predicateである。

(iii) 論理制約述語

q(*x_1, …, *x_i, …, *x_j, …, x_n)
所与の関係p(*x_1, …, *x_i, …, *x_j, …, *x_n)の属性a_iとa_jの成分値が，2項述語θを満たすかどうか判定する述語
q(*x_1, …, *x_i, …, *x_j, …, *x_n)←p(*x_1, …, *x_i, …, *x_j, …, *x_n)∧θ(*x_i, *x_j)

(iv) 論理商述語

r(*x_1, …, *x_{i-1}, *x_{i+1}, …, x_n)
所与の関係p(*x_1, …, *x_i, …, x_n)の属性a_i方向への射影が常に，関係q(*y_1, …, *y_j, …, *y_n)の属性b_j方向への射影を包含するならば，そのような属性値の組<*x_1, …, *x_{i-1}, *x_{i+1}, …, *x_n>に対して真値を返す述語

(c) 拡張関係論理サポート述語

(i) 外延化述語[14),12)]

setof(*x, *pred, *set)
述語*predを証明可能とするような*xのあらゆる実現値の集合を*setとする述語
setof(*x, *p, *)←|asserta(found(mark))∧call(*p)∧ asserta(found(*x))∧fail setof(*, *, *s)←collectfound(<>, *m)|*s=*m collectfound(*t, *s)←getnext(*x)|collectfound(*x.*t, *s)

```
collectfound(*s, *s)←|
getnext(*x)←retract(found(*x))|*x\==mark
```

(ii) 存在化述語

exists(*x, *pred) 注)
述語*pred を真ならしめる変数*x が存在するならば真値を返す述語

(iii) numerical quantifier [8]

exists(θ, *number, *x, *pred)
*number を非負整数N, *x を量限定される変数, θ を論理記号"≧, >, <, ≦, =, ≠"のどれかとする。このときθの値に応じてそれぞれnumerical quantification"∃*x≧N, ∃*x>N, ∃*x<N, ∃*x≦N, ∃*x=N, ∃*x≠N"を表わす述語

(d) Aggregation Function [8]

実際のRDBへの問合せでは，種々のaggregation functionを準備しておく必要がある。ここでは最小限必要なaggregation functionとして，DEDUCE [8]やKAUS [15] にならって次のものを準備しておく。

(i) リスト長計算述語

count(*list, *length)
リスト*list の長さ*lengthをカウントする述語
count(<>, 0)← count(*x.*l, *n1)←count(*l, *n)∧plus(*n, 1, *n1)

(ii) 最大値述語

```
maximum(*list, *max)
```

注) Prologの*x^*pred に相当する。[14]

実数のリスト＊list の中から，その最大値＊maxを計算する述語

(iii) 最小値述語

minimum(＊list, ＊min)
実数のリスト＊list の中から，その最小値＊minを計算する述語

(iv) 平均値述語

average(＊list, ＊ave)
リスト＊list の成分である各実数の平均値＊aveを計算する述語

(v) 総和述語

total(＊list, ＊tot)
リスト＊list の成分である各実数の総和＊totを計算する述語

C データベース管理

a スキーマ定義

ＥＤＢＳ基本構成のインタフェースを徹底的に分析したANSI／X3／SPARC[16]によれば，ＥＤＢＳモデルの記述は３層スキーマ構成（図2.4.2 参照）からなる。概念スキーマ，外部スキーマ，内部スキーマという３層スキーマは，それぞれ組織体管理者，応用管理者，ＤＢ管理者から管理される。ＲＤＢの定義，生成，消滅，名前替え（ renaming ）といったＲＤＢの初期化（ initialization ）[注)]に関する管理を実現するには，まずＲＤＢ上にどのようなデータが，どのようなデータ構造をなしているかの全体像を，スキーマとして記述する必要がある。従ってここではＲＤＢの表わすスキーマを，上記３層スキーマのもとで考察し，ロジック・プログラミングに結び付ける基礎を与える。

注) ＲＤＢの生成，消滅，名前替えに関する管理述語は，本項で述べるスキーマ定義と対応がとれる形式で導入しなければならないが，詳細に立ち入ることになるので省略する。

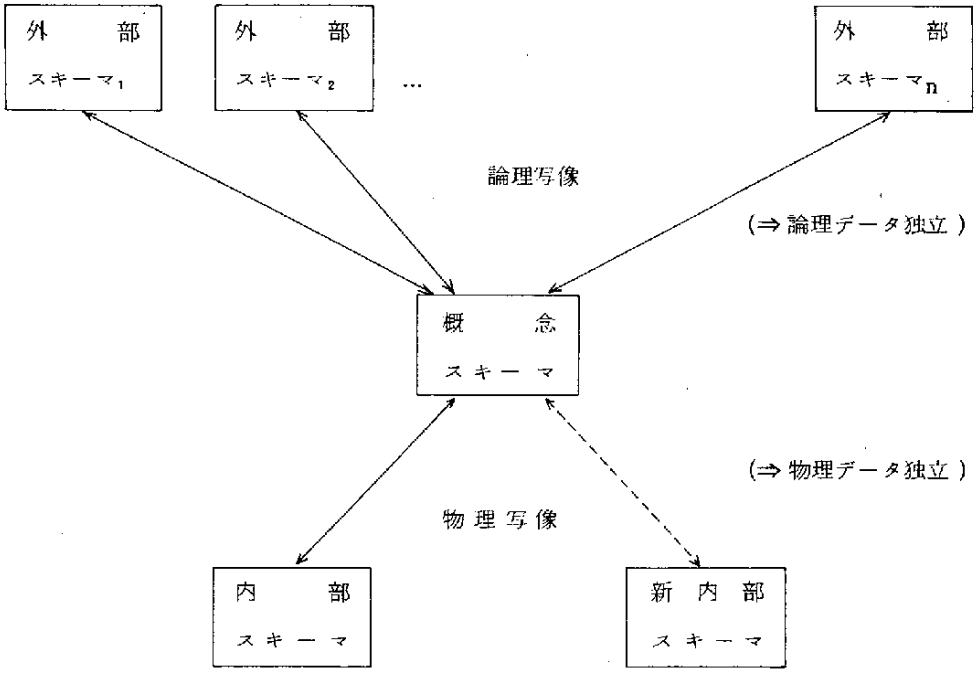


図2.4.2　3層スキーマ構成[2)]

'FGKL／Sではスキーマ定義は，ひとつのモジュールをなす。RDB生成後，関係の追加や削除等でスキーマ変化が起こるので，スキーマ定義モジュールはRDBの初期スキーマを与えると考えられる。ここでは概念スキーマを例にとり，スキーマ定義モジュールの述語としての形式を提示する。概念スキーマはRDBに含まれる（初期）関係スキーマと関係間の相互関連を与えるので，例えば次のような情報を含む。

```
conceptual － schema（スキーマ名）←
  administrator0（管理者0）←
  date（作成日）←                          } 概念スキーマに対する説明
  …
  remarks（注意）←
  define － relation〔
    関係名1（…　）where（…　）←           } 関係1とその関係内制約条件記述
    関係名2（…　）where（…　）←           } 関係2とその関係内制約条件記述
    …
```

```
    関係名n(…  ) where (…  )←]←          } 関係nとその関係内制約条件記述
  interrelational－constraints[           } 関係間の制約条件記述
    節1, …, 節m]←
end－of (スキーマ名)←
```

この概念スキーマ内の高階述語 define-relation において，関係内の制約条件を表わす述語は，例えば次のような形で示される。

```
r1(a1:char(＝15),
    a2:integer(0..999),
    a3:char(≦100),
    a4:real)
where(*x2≦*x4←r1(*x1, *x2, *x3, *x4),
      *x3＝*y3←r1(*x1, *x2, *x3, *x4)∧
                r1(*x1, *x2, *y3, *y4),
      r1(*x1, *x2, *x3, *x4)←
                r1(*x1, *x2, *y3, *y4)∧
                r1(*x1, *y2, *x3, *x4))←
```

ここで第2の節は関数従属 $\{a_1, a_2\} \rightarrow \{a_3\}$ を表わし，節3の節は多値従属 $\{a_1\} \rightarrow\rightarrow \{a_2\} \mid \{a_3, a_4\}$ を表わしている。制約条件を機械的に効率よくチェックするためには，節の形に適当な制限を加えて，単純な形に限定しておく必要がある。

内部スキーマや外部スキーマを定義するモジュールも，同様の表記法をとる。内部スキーマは概念スキーマによるRDBの実現方法を記述するものなので，EDBSのアーキテクチャを反映する述語を用意しておく必要がある。外部スキーマは応用プログラムからみたRDBの論理構造を定義するものなので，応用プログラムの視野(view)を反映する述語を豊富に用意しなければならない。

b　データベース更新 注)

RDBの更新(update)処理述語をエジンバラ大学Prolog第3版の内部データベース[13),14)]のもつ述語との整合性を保ちつつ提案する。

(a)　タップルのキーを調べる述語

注) 更新は理論的には評価方式ではモデルの変更である。

(i) キーを与えてタップルを取り出す述語

lookup(*relation, *key, *tuple)
関係*relationに関するキー値*keyを与えて，そのキーに対応する関係の元であるタップル*tupleを取出す述語。（instance[13),14)] 参照）

(ii) タップルを与えてキーを取出す述語

lookuped(*relation, *tuple, *key)
関係*relationに属するタップル*tupleを与えて，そのタップルのキー値*keyを取出す述語。（recorded[13),14)] 参照）

(b) タップルそう入（insertion）述語[注)]

(i) 関係にタップルをそう入する述語

insert(*relation, *tuple, *result)
関係*relationにタップル*tupleをそう入し，その結果の関係を*resultとして取出す述語（record[13),14)] 参照）

(ii) 関係の先頭にタップルをそう入する述語

inserta(*relation, *tuple, *result)
関係*relationの先頭にタップル*tupleをそう入し，その結果の関係を*resultとして取出す述語（recorda[13),14)]参照）

(iii) 関係の最後尾にタップルをそう入する述語

insertz(*relation, *tuple, *result)

注）“そう入，修正，削除”述語に出現する結果の関係*resultは，更新述語の実行が行われた場合，もとの関係は変更するということは自明である。そこで副作用を利用するということを前提に，*resultを陽に出さない表記法もありうる。

関係＊relationの最後尾にタップル＊tupleをそう入し，その結果の関係を＊result として取出す述語（ recordz[13],[14]参照）

(c) タップル修正（ modification ）述語

(i) 関係のタップル1をタップル2に修正する述語

modify(＊relation, ＊tuple1, ＊tuple2, ＊result)
関係＊relationに属するタップル＊tuple1を，別のタップル＊tuple2に修正し，その結果の関係＊result を値として返す述語

(ii) あるキーのタップルを別のタップルに修正する述語

modifyk(＊relation, ＊key, ＊tuple, ＊result)
関係＊relationに属するキー値＊keyをもつタップルを，別のタップル＊tupleに修正し，その結果の関係＊result を取出す述語

(iii) あるキーのタップルを，別のキーのタップルに修正する述語

modifykk(＊relation, ＊key1, ＊key2, ＊result)
関係＊relationに属するあるキー値＊key1 をもつタップルを，別のキー値＊key2 をもつタップルに修正し，その結果の関係を＊result として取出す述語

(d) タップル削除（ deletion ）述語

(i) 関係からタップルを削除する述語

delete(＊relation, ＊tuple, ＊result)
関係＊relationに属するあるタップル＊tupleを削除し，その結果の関係＊result を取出す述語

(ii) 関係からあるキー値のタップルを削除する述語

deletek(*relation, *key, *result)
関係 *relation からキー値 *key のタップルを削除し，その結果の関係 *result を取出す述語

(iii) 関係の先頭にあるタップルを削除する述語

deletea(*relation, *result)
関係 *relation からその先頭にあるタップルを削除し，その結果の関係 *result を取出す述語

(iv) 関係の最後尾にあるタップルを削除する述語

deletez(*relation, *result)
関係 *relation からその最後尾にあるタップルを削除し，その結果の関係 *result を取出す述語

D 知識ベース管理

a スキーマ管理

3層スキーマの概念スキーマ・レベルで外部スキーマを内部スキーマに埋込むには，外部スキーマや内部スキーマそれぞれの論理的な整合性（ logical consistency ）を保有しつつ，それらを概念スキーマへと統合管理していくスキーマ管理機構[注)]が必要である。これには静的スキーマ管理という考えと動的スキーマ管理という考えがある。

(a) 静的スキーマ管理（ Fig 2.4.3 参照 ）

静的スキーマ管理とは，内部スキーマや外部スキーマそれぞれの整合性を維持しつつ，両者を概念スキーマへと併合し，外部スキーマのイメージをユーザに提供するために，それにもとづく濾波（ filtering ）手続きを前処理手続きの実行という形式で実現するものである。外部スキーマとしてユーザの視野（ view ）を考え，スキーマ情報として統合性制約（ integrity constraints ）の一種である従属（ dependency ）関係のみに限定して考察したもの[17)]があるが，その場合の概念図式が Fig. 2.4.4 である。

注）スキーマ管理の基本問題は，次の2つである。① 充足可能性問題が可解なクラスにおちいるかどうか，② 妥当性問題が可解なクラスにおちいるかどうか[7)]。

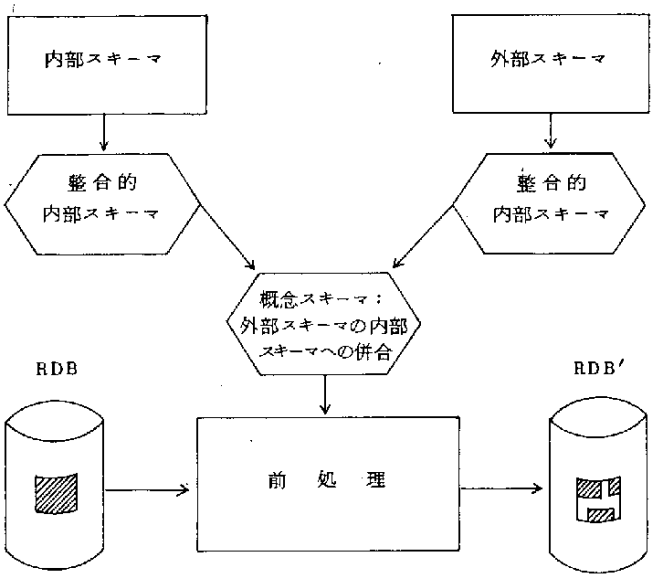


Fig. 2.4.3　静的スキーマ管理

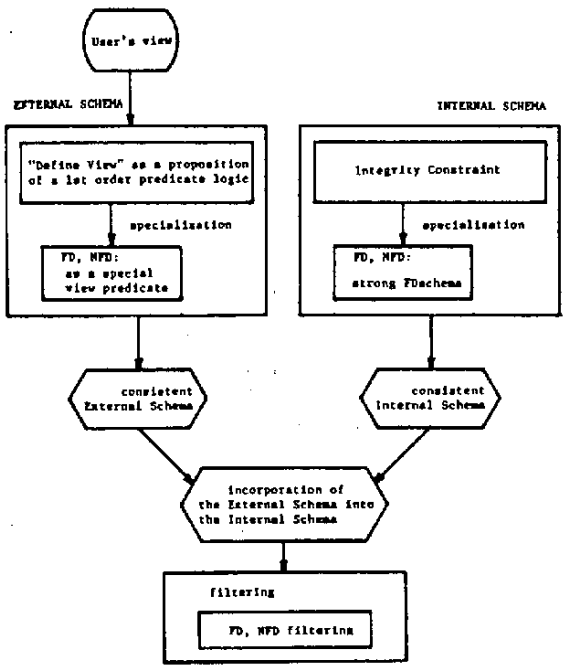


Fig. 2.4.4　Framework of view incorporation [17]

(b) 動的スキーマ管理

前処理でなく実時間処理の場合の動的スキーマ管理機構を実現したものが，通常，評価方式の関係データベース管理システム構成法[8]といわれるものである。これについては，ＦＧＫＬ／Ｓマシンと ＥＤＢＳとの結合方式を考慮した考察が必要なので，Ｅのシステム構成法のところで詳しく述べることにする。

b 推論根拠管理[18]

ＫＩＰＳ研究開発課題である問題解決推論システムや知識ベース管理システムが結合し適切な推論機能を発揮するためには，推論の根拠を支える管理機構が，ＫＢやＲＤＢに適切な知識を獲得する機能を保有している必要がある。このような機能をもつシステムは，常に更新される可能性をもつ開いたＫＢやＲＤＢをもち，不完全知識に対する推論根拠管理を行う。さて知識獲得プロセスには，同化と調節という２つの側面があるが，ＫＩＰＳ研究の発展につれ，この両機能をもつ論証のメカニズムの研究が行われるようになってきた。ここでは人工知能の分野における運用論の導入として取扱われていたDefault Reasoningや非単調論理のプリミティブな考えを述べ，'ＦＧＫＬ／Ｓマシンにおいても知識の同化や調節のためのメタ述語を用意すべきであることを，ひとつの例をもって示すことにする。

Default Reasoning[19]は不完全知識のもとで，問題解決のために強制的に発動される"与えられたＫＢやＲＤＢの知識Γから，ある知識Ａが証明されないが故に，Ｂと結論づける"という推論図式のことである。Kowalskiのメタ述語demo[21]を用いて，'ＦＧＫＬ／Ｓ流にその考えを形式化すれば，次のようになる：

demo (Γ, Ｂ) ← not〔 demo (Γ', Ａ) 〕

ＢとしてnotＡをとったものが，前述の閉世界仮説に相当する。このような状況下での論証プロセスを，より形式的に処理しようというのが非単調論理（non-monotonic logic[19]）であり，不完全知識下での知識の同化や調節の過程を演繹論理の拡張の方向でとらえようとするものである。知識の同化や調節の過程は種々のスペクトルがあり，一概には論じられないが，Kowalskiは知識の同化に関して，次のようなメタ述語assimilate[20]を提案している。

```
assimilate(*currdb, *input, *currdb) ← demo(*currdb, *input)
assimilate(*currdb, *input, *newdb) ←
  belongsto(*info, *currdb) ∧ seteg(*interdb, difference
  (*currdb, *info)) ∧ demo(union(*interdb, *input), *info) ∧
```

```
  assimilate(*interdb, *input, *newdb)
assimilate(*currdb, *input, *currdb)←
  demo(union(*currdb, *input), false)
assimilate(*currdb, *input, union(*currdb, *input))←
  independent(*currdb, *input)
```

ただしここでの述語union，differenceはevaluable predicatesとする。

E　外部データベース・システム構成法

a　システム構成の概略

FGKL／Sマシン単独で，内部データベース（IDB：Internal Data Base）に格納されている公理集合をもとに，質問応答している際のシステム構成イメージは図2.4.5のように与えられる。このシステム構成は，2.4.2.A.cで述べた分類によれば，非評価方式の構造をしている。これに対して従来から検討されている評価方式のシステム構成図は，図2.4.6や図2.4.7で与えられている[7]。

'FGKL／Sマシンと外部データベース・システムEDBS－I（試作RDBマシン）やEDBS－II（汎用大規模RDB管理システム）との接続法を，評価方式のシステム構成を参考に模式化すれば，図2.4.8のようになる。'FGKL／SマシンとEDBSを結合し，ひとつのシステムに構成するやり方は，現状の技術的／理論的背景を前提にすれば，

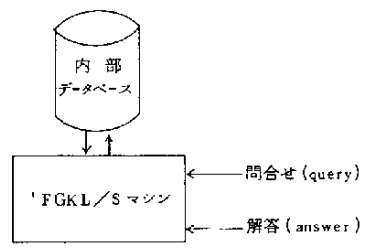


図 2.4.5　'FGKL／Sマシンのシステム構成

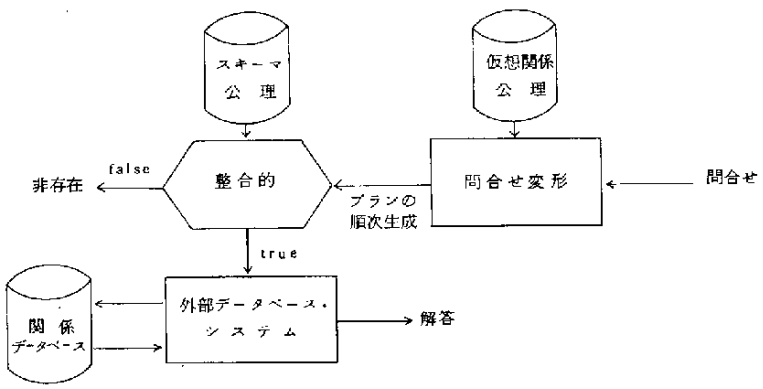


図2.4.6　評価方式演繹的質問応答システムーI[7]

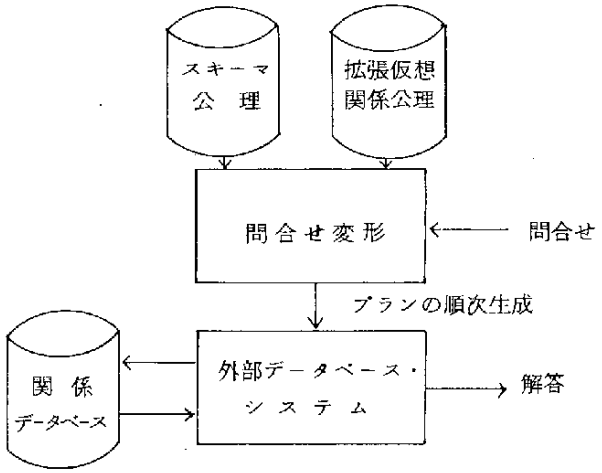


図 2.4.7　評価方式演繹的質問応答システム－II[7]

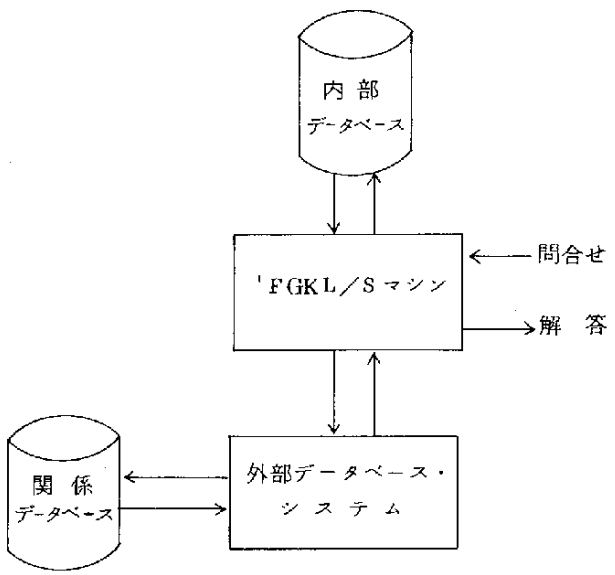


図 2.4.8　FGKL／Sマシンと外部データベース・システムの結合されたシステム構成

2.4.2.E.b と 2.4.2.E.c で述べる２つの方式が実現可能性が高い。

b　関係論理の関係代数への埋込み方式

'ＦＧＫＬ／Ｓマシンへの問合せ言語は，主として2.4.2.B.ｂで述べた関係代数型述語と2.4.2.B.ｃで述べた関係論理型述語からなる。ＥＤＢＳは通常，関係論理型あるいは関係代数型のどちらかを問合せ言語にとることが多い。ここでは今後の研究開発課題である試作ＲＤＢマシンへの接続可能性，並列型ＫＢマシンへの拡張可能性，やハードウェアの特性等を十二分に考慮した上で，関係論理型問合せ言語を全て関係代数型問合せ言語に埋込む方式を提案する。'ＦＧＫＬ／Ｓマシンのユーザは，一見したところ関係代数型問合せ言語のみを使用しながら，そのうちある種のコマンドがＥＤＢＳのもつ関係代数コマンドに，マクロとして受渡され，その部分はＥＤＢＳで処理される。さて本方式を実現するメカニズムのキー・テクノロジーは，高階の外延化述語 setof [14] である。setof 述語を用いれば，主要な関係論理述語が簡単に関係代数述語にマクロ変換される。例えば最も難しい関係代数の商演算の場合，その関係代数での定義[4]は次のようになる。

$*r[*a \div *b]*s \triangleq \{t.\overline{*a} \mid *r(t) \wedge \mathrm{subset}(*s[*b], g_{*r}(t.\overline{*a}))\}$，

ただし $g_{*r}(t.\overline{*a})$ は，タップルｔの属性＊a 以外への属性方向への射影の関係＊r に関するイメージ集合とする[4]。

この関係代数による商定義に対して，関係論理による関係＊r，＊s の定義（例えば，＊r(＊a：＊x，$\overline{*a}$：＊y)，＊s(＊b：＊x，$\overline{*b}$：＊z)とする。）を用いて，同一概念を構成すると次のようになる。

```
division(*r, *a, *s, *b, *d)←
  setof(*y, setof(*x, *r(*x, *y), *f)∧
                setof(*x, *s(*x, *z), *g)∧ subset(*g, *f), *d)
```

実際のProlog処理系では，setof の性能は余り良くないといわれている。そこでＥＤＢＳのもつ高速の集合／関係演算をフルに利用する関係代数コマンドへ置換した後，必要な演算処理を行えば，大巾な性能向上が期待される。ここで注意したいのは，meta logical 述語 setof のない pure Prolog[13] での商述語記述の困難さである。pure prolog に setof を導入すればProlog 系言語が問合せ言語として関係完備（relational complete）であることが証明できる[23]。

c　結合グラフ法を用いる評価方式

図 2.4.6 や図 2.4.7 で与えられる評価方式の特徴は，次の２つにある。

(i) KB((拡張)仮想関係／スキーマ公理の格納されているＩＤＢ)とＲＤＢの完全分離

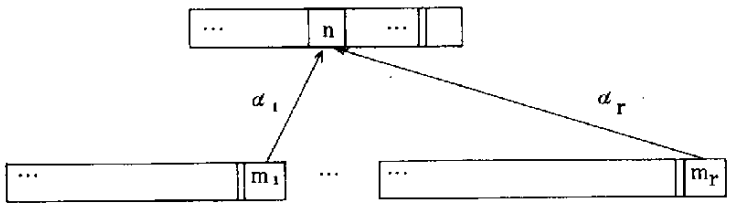
(ii) 結合グラフ法という評価方式の利用

このうち(i)は通常，RDBの濃度がKBの濃度より膨大だから採用されることが多い。ここではIntensional Evaluatorと呼ばれることの多い問合せ変形部の基礎理論である結合グラフ法(ii)について，最も基本的な事柄を述べる[7]。

Chang[8]は，Sichel[22]の結合グラフ(connection graph)法のアイディアをもとに，問合せを変形し，EDBSで処理可能なプランを次々と順次生成するための書換え規則(rewriting rule)を抽出している。書換え規則は1階多類Horn論理と相性がよく，次の3種の規則よりなる[7]。

① 全ての節体中のゴールnについて，節頭のゴールm_1，…，m_rから直接nを指す枝をもつならば，次のような書換え規則を与える：

$$W(n) = \alpha_1 W(m_1) \cup \cdots \cup \alpha_r W(m_r)$$



② 全ての節頭のゴールnから出る枝がある時，次の書換え規則を与える：

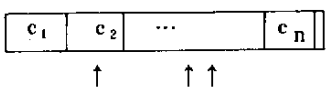
$$W(n) = P_1 \cdots P_r$$

ここに
$$P_i = \begin{cases} W(m_i) & : m_i \text{がKBで定義される関係を含むゴールの場合，} \\ m_i & : \text{それ以外。} \end{cases}$$

③ 問合せについては，その否定形について次のような書換え規則を与える。

$$T = Q_1 \cdots Q_m$$

ここに
$$Q_i = \begin{cases} W(c_i) & : c_i \text{がKBで定義される関係を含むゴールの場合，} \\ c_i & : \text{それ以外。} \end{cases}$$



以上の書換え規則にもとづき1階多類Horn論理における子孫概念を，論理として記述したのが次式，その結合グラフ表現が図2.4.8，そしてその 'FGKL／Sで問合せ言語で記述したのが図2.4.9である。

$\forall *x^h \forall *y^h (parent(*x^h, *y^h) \supset ancestor(*x^h, *y^h))$,

$\forall *x^h \forall *z^h \forall *y^h (parent(*x^h, *z^h) \wedge ancestor(*z^h, *y^h) \supset ancestor(*x^h, *y^h))$,

⋮

$\supset *y^h\ ancestor(桃太郎, *y^h)$ {問合せ}

ただし $*x^h$, $*y^h$, $*z^h$ は全て human 型の変数とする。

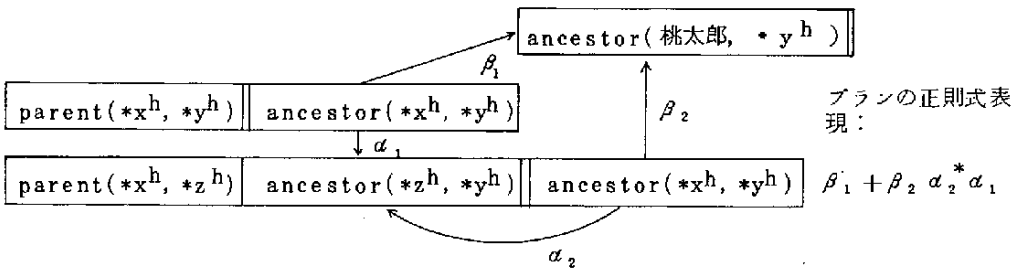


図 2.4.8　1階多類論理での結合グラフ法

```
define_relation[ancestor:KB;
  var  *x, *y, *z : human;
  relation  parent : RDB;
  ancestor(*x, *y)←parent(*x, *y)
  ancestor(*x, *y)←parent(*x, *z)∧ancestor(*z, *y)]

  ...

? − ancestor(桃太郎, *y)
```

図 2.4.9　　1階多類論理での問合せ

結合グラフ法を用いる評価方式は明確なシステム設計方針を与え，理論的基盤もしっかりしているが，論理型言語 'ＦＧＫＬ／Ｓの基本計算メカニズムや拡張計算メカニズムを用いて実現できるかどうかについて，今後解決していかなければならない技術的課題をかかえている。

参考文献

1) Codd, E.F.: A Relational Model of Data for Large Shared Data Banks, CACM, １３, ６, １９７０, pp.３７７～３８７.

2) 植村俊亮： データベースシステムの基礎，オーム社，１９７９.

3) (財)日本情報処理開発協会，(社)日本電子工業振興協会(編)： 電子計算機基礎技術開発調査報告書ソフトウェア技術編，昭和５７年３月。

4) Codd, E.F.: Relational Completeness of Data Base Sublanguages, in *Courant Computer Science Symposium* 6 Data Base System, Prentice-Hall, １９７２, pp.６５～９８.

5) 加藤昭彦： 射の導入による関係データモデルの表現独立性への接近と従属性への応用，情報の記憶と利用に関する理論的研究，京都大学数理解析研究所講究録４２３, April １９８１, pp.２７５～２９３.

6) Kunifuji, S.: Second-order Many-sorted Boolean Logic for knowledge Representation Language, IIAS-SIS Fujitsu Ltd., Research Report №１４, Feb. １９８１.

7) 國藤 進，若木利子： 拡張仮想関係と知識フィルタの演繹的質問応答への応用について，Proc. of the JIPDEC Information Systems Seminar on Semantic Aspects of Databases, (財)日本情報処理開発協会，Feb. １９８０, pp.４９～８１.

8) Gallaire, H. and Minker, J.(eds.): Logic and Data Bases, Plenum Press, New York, １９７８.

9) 古川康一： データベースの知的アクセスに関する研究，東京大学学位論文，昭和５４年.

10) Reiter, R.: An Approach to Deductive Question-Answering, BBN Report №３６４９, Bolt Beranek and Newman, Inc., １９７７.

11) Kowalski, R.: Logic for Problem Solving, North Holland, １９７９.

12) Clocksin, W.F. and Mellish, C.S.: Programming in Prolog, Springer-Verlag, １９８１.

13) Pereira, L.M., Pereira, F., and Warren, D.: User's Guide to DEC

system-10 Prolog, Laboratorio Nacional de Engenharia Civil, Lisbon, Portugal, 1979.

14) Byrd, L., Pereira, F., and Warren, D.: A Guide to Version 3 of DEC-10 Prolog and Prolog Debugging Facilities, DAI Occasional Paper 19, Dept. of A.I., Univ. of Edinburgh, Edinburgh, Scotland, 1980.

15) Ohsuga, S.: Multi-layer Logic for Knowledge Representation(in Japanese), Preprint of Working Group of Artificial Intelligence and Man Machine Interaction 17－6, Information Processing Society of Japan, Sept, 1980.

16) The ANSI/X3/SPARC DBMS Framework, Report of the Study Group on Database Management Systems, Information Systems, 3, 3, 1978, PP. 173～191.

17) 國藤 進, 若木利子, 竹島 卓: ユーザ・ヴューにもとづくフィルタリング法について, 電子通信学会, オートマトンと言語研究会AL78－68, Dec.1978.

18) 國藤 進: 知識情報処理システムから創造科学へ, オペレーションズ・リサーチ, May 1981, pp.261～268.

19) Artificial Intelligence-Special issue on non-monotonic logic, Vol. 13, №1－2, North Holland, 1980.

20) Kowalski, R.: Logic as a Database Language, July 1981.

21) Bowen, K.A., and Kowalski, R.A.: Amalgamating Language and Metalanguage in Logic Programming, School of Computer and Information Science, Syracuse University, June 1981.

22) Sickel, S.: A Search Technique for Clause Interconnectivity Graphs, IEEE Trans. on Computer, C－25, 1976, pp. 823～834.

23) 國藤 進, 加藤昭彦, 安達統衛, 竹島卓, 沢村一: ロジック・プログラミングとデータベース, 情報処理学会第25回人工知能と対話技法研究会／第20回記号処理研究会, March 1982.

2.4.3　入出力システム

'ＦＧＫＬの入出力は下図のような段階を経る。

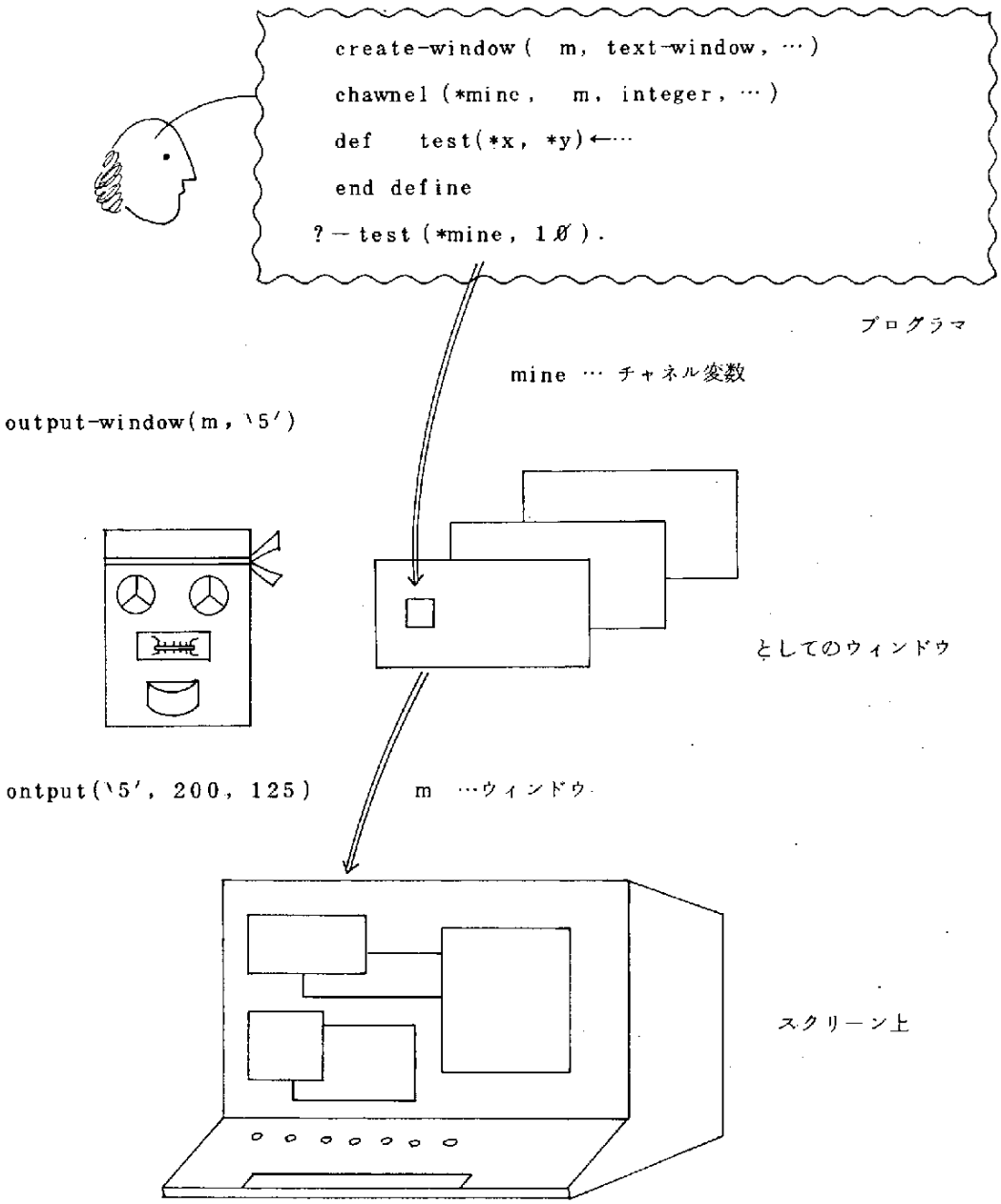


図 2.4.1 0　入出力の段階

'ＦＧＫＬの入出力はユーザのレベルと，システム上の（マイクロプログラム的な）レベルとで大きな相違を見せる。

ユーザ・レベルにおいては，入出力はすべてチャネル変数を介して（副作用的に）実行される。すなわち，チャネル変数に値が割りつけられると，これはそのチャネルへの出力を意味し，チャネル変数に対して値が要求されると，これはチャネルに対する入力要請となる。

例えば，test という述語を次のように定義したとする。

test (*X, *Y) ← *Y is 2×*X

ここで，入力チャネル変数 *mi に対して，

test (*mi, 8)

とすると，まず *mi に対して入力が要求される。例えば4が入力されれば論理式は true となり，5が入力されれば false となる。

チャネル変数でなければ，test (*X, 8) は X＝4という答を得て停止する。

従って，チャネル変数を用いないと，

test-in (*X, *Y) ← inpnt (*X) ∧ *Y is 2×*X

という別の節を作り，test-in (*X, 8) を実行することとなる。

一方 *mo を出力チャネル変数であるとすれば，

test (*mo, 8)

は，*mo のチャネルに4という答を出して，終了する。

*m が入出力チャネルの場合はどうなるかであるが，この場合まず *m はあたかも出力チャネルのように扱われる。ただし，

test (*m, *Y)

のような *m の値が決定されない場合に，入力が要求されて，計算が進められるという手順を踏む。

上のような手段によりユーザは，

read (*X, channel-in) write (*X, channel-out)

のような入出力命令を書く必要が無くなる。

（コメント）

read, write のような入出力命令が無意味になるというわけではない。実際的な意味は残るにしても，本質的ではないということ。

チャネル変数には後に述べるような各種属性が付属し，高レベルの入出力概念を扱うことができる。

（コメント終了）

一方，システムのレベルにおいては，'ＦＧＫＬ／Ｓマシンがマイクロプログラム制御のフォン・ノイマン型マシンであることから，通常の基本入出力命令が用意される必要がある。この基本入出力命令は，'ＦＧＫＬのユニフィケーションによる計算メカニズムより下位のレベルで動作する。

先程の例にあったread，writeや図2.4.10のoutput-window（m，'5'）などもこのレベルとして把握できる。

より詳しく述べれば，このレベルにも２つの階層がある。

ウィンドウであるとかファイルであるとかあるいは（ファイルの一種としての）プリンタとかに出力する「概念的」な部分と，具体的な論理入出力機器に対してコード（バイトまたは語）を送受信する「物理的」入出力の部分とである。

概念的な部分までは（チャネル変数も含めて），いわゆるメッセージ処理機構が基本となる。一方論理入出力のレベルではフォン・ノイマン型の手続きという概念が主となる。

メッセージ機構ではチャネル（もしくはウィンドウなど）に対して要求メッセージが出され，これに対する反応という形で入出力が行なわれる。

この関係を図2.4.11に示す。

図2.4.11のような処理階層を具体的に実現するには，図2.4.12に示すように，物理入出力機器，入出力オブジェクト（ファイル，ウィンドウなど），チャネル変数，プログラム（プロセス）の各レベルで入出力処理の管理が必要となる。

プロセス管理ではユニフィケーションの中でチャネル変数に対する入出力メッセージを出し，またチャネル変数の属性変更のメッセージを処理する。

チャネル管理では，チャネル変数に対する入出力や属性変更のメッセージを処理する。この部分の処理はメッセージの発生による事象駆動（event-driven）的処理となる。

入出力オブジェクト管理も，チャネル変数と似ていて，入出力オブジェクトに対する入出力メッセージを受け取ってから処理をする。特にこの時点で重要になるのは，入出力オブジェクトから物理入出力を行う段階で，入出力資源に対する配慮を必要とする点である。

ディスプレイやファイルなどの場合には，入出力オブジェクトと物理入出力との間で，ファイル・システムとかディスプレイ・システムだとかの調整を必要とする。

A　チャネル属性

次のような属性が割り当てられる。

(α)　コマンド・インタープリタ

チャネルは，データの仲介だけでなく特別な処理も引受けることができる。各チャネルご

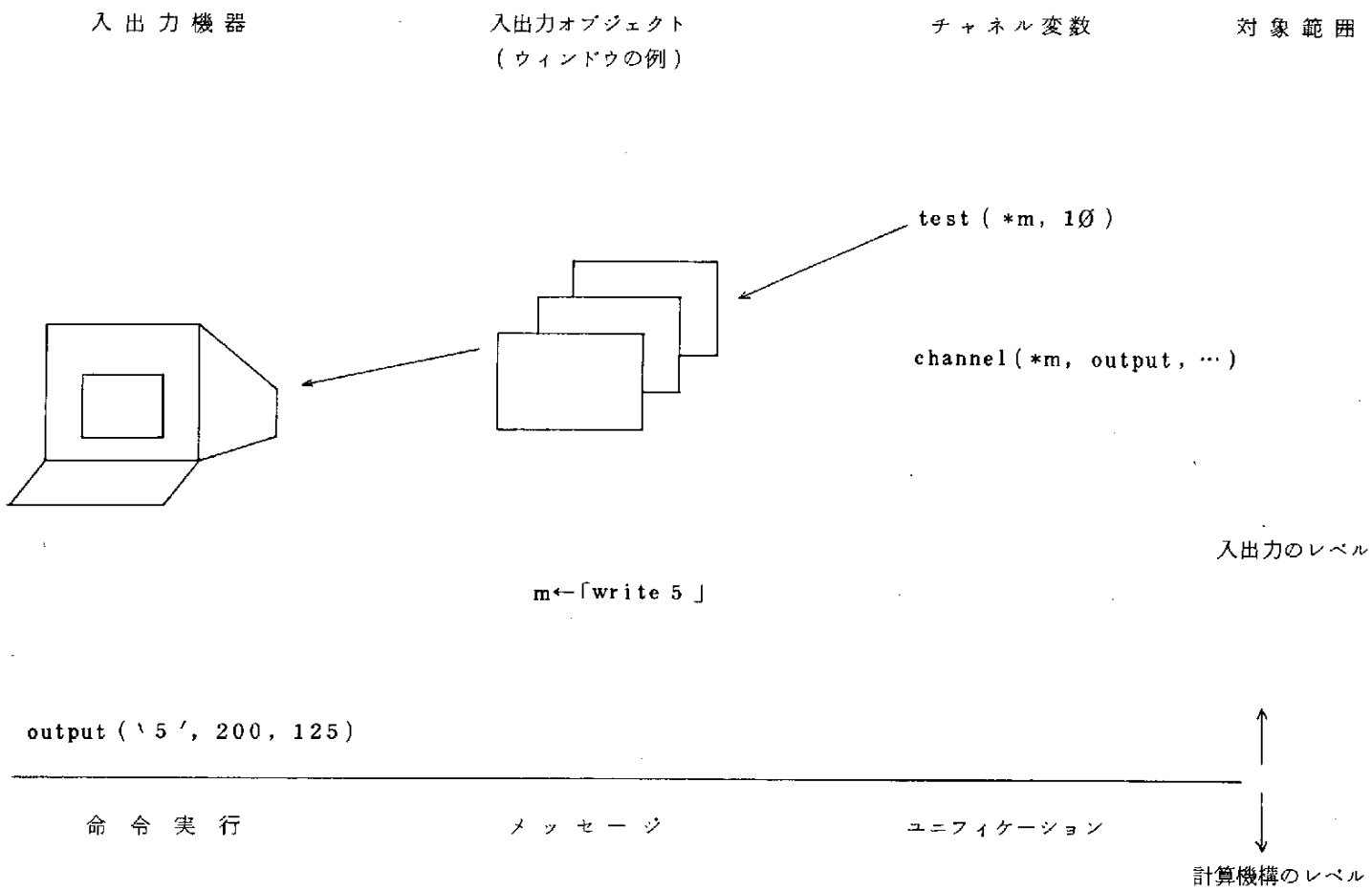


図 2.4.1 1　計算機構のレベルと入出力のレベル

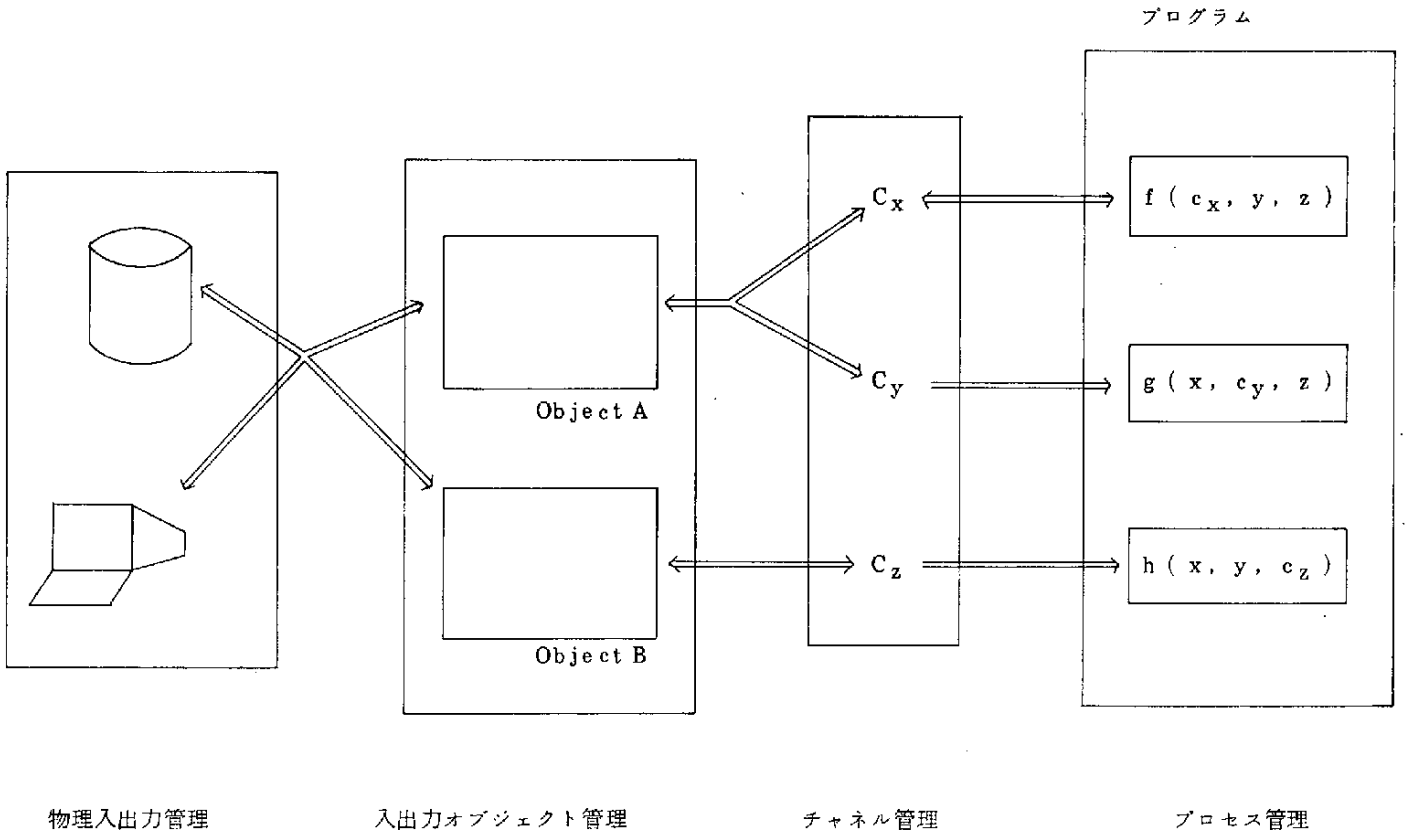


図 2.4.1 2　チャネルと入出力オブジェクト

とに，コマンド・インタープリタを定義できる。（Interlispの top-level Lispx に相等）

(1) データの種別

文字（普通のテキスト入出力），数値， etc.

(2) データの区分

文字の場合なら1行の行数を言う

(3) end-of-file 処理

end-of-file における処理

(4) アクセス権

このチャネルに対するアクセス権を示す。

(5) モニター・チャネル

このチャネルへの入力または／および出力をモニターするチャネルを指定する。

チャネルを会話端末といわゆる唖のファイル端末に分類すると，会話端末には次のような属性が割り当てられる。

(6) 入力要請

入力待ちの表示をする。通常は何らかの文字列を打ち出す。

(7) return処理

入力時の return keyは特別な用途として使用できる。

(8) tab 処理

ハード的に可能なら各チャネル毎に tablation の処理をする。（この項不要？）

(9) 物理属性

どのような端末，どのような物理ファイルかを指定する。

B．チャネルの機能

チャネルの機能としては入力と出力が典型的なものである。通常は入力チャネルと出力チャネルとかそれぞれ1つずつ属性化されている。これらは現在のチャネルということでカレント・チャネル（ current channel ）と呼ばれる。

'FGKLの入出力関数において，特にチャネルを指定しなければカレント・チャネルが入出力に使用される。

データ処理のチャネルには障害時の回復用にトランザクション機能を与えられているものがある。

C．チャネルの割り当て

'ＦＧＫＬでは原則としてチャネルの個数に制限はない。

ＦＧＫＬでは原則としてチャネルの個数に制限はない。チャネルはユーザが動的に（コマンドまたはプログラムで）割り当てるものと，システムが事前に割り当てるものとから成っている。

事前割り当ての対象となるチャネルは，

(1) 端末標準チャネル

このチャネルには，ユーザとの対話をファイルするモニター・チャネルuser-sessionが割り当てられている。

(2) 端末モニター・チャネル

ユーザとの対話を記録する。

(3) デバッグ・チャネル

端末用でデバッグ時に通常利用される。このモニター・チャネルにはdebug-sessionが割り当てられている。

(4) トレース・チャネル

トレース時に利用される。普通はカレント・チャネル（出力）を使うが，大量のトレース情報などはここに貯える。

D. ディスプレイの構成

ディスプレイの画面はビットマップ・メモリにより構成されている。

論理的にはディスプレイはウィンドウの階層構造になっており，ウィンドウの中には一部しか画面に出ていないものや，あるいはまったく画面上に姿を現わしていないものもある。

'ＦＧＫＬのプログラム上では，チャネル変数が各ウィンドウに一つまたは複数個割り当てられており，ウィンドウはあたかも仮想的なファイルのようにしてデータの入出力を行う。

ウィンドウの論理的な階層構造，および，同一レベルのウィンドウの組において，どのウィンドウを一番上にもってくるか，ということはユーザが指定できる。

しかし，ウィンドウの細かい配置については各ウィンドウのクラス毎に適当に定めてシステムが処理するものとする。

実際には各ウィンドウに対応するバッファなりファイルなりがメモリやファイル・システム上に確保されており，その一部がビットマップ・メモリ上に移されて，画面に出てくる。

キー・ボードやマウスなどのディスプレイに対する入力機器は，プログラム上では現在活性化されているチャネル変数に接続されており，ディスプレイの論理的構造からは，現在活性化されているウィンドウに結びつけられている。キー・ボードは，従って，プログラムによって，現在活性化しているウィンドウが変更されていない限りそのウィンドウの外に出ることはできない。しかし，マウスの場合には，機器としての特性上，ウィンドウの中にあることも，ウィン

ドウの外に出ることもできる。すなわちマウスについては，すべてのウィンドウの元締めとも言うべき原ウィンドウ，ディスプレイ・ウィンドウ，が常に結びつけられている。

もち論，マウスの入力が意味をもつかどうかは，特定の活性化ウィンドウの中にあるかどうかが決まり，マウス入力のもつ意味もウィンドウごとに変化する。

ただし，ウィンドウの物理的な位置を変更するような役割は，マウスのようにディスプレイ・ウィンドウに対して入力を提供できるデバイスでないと処理できない。

例えば，'FGKL上のテキスト・エディタを考えてみよう。

極端な場合として，一回変更するごとに，新しくウィンドウが作られてゆくとしよう。

そうするとテキスト・エディタのプロセスedはある時点でN個のウィンドウを持つ。ユーザの眼には最近のウィンドウが見えている。テキストの編集操作は新しいウィンドウを作る操作となる。キーボードはこの現在一番上にあるウィンドウに結びついている。

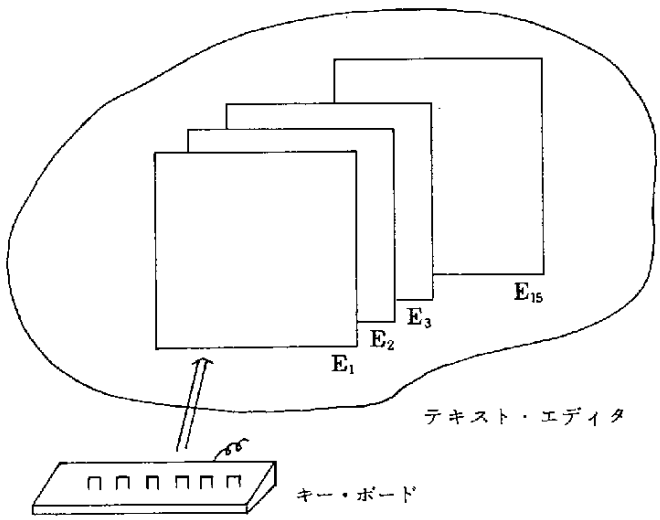


図2.4.13　ウィンドウの集合

編集操作において，元の状態への回復は下にあるウィンドウを上に持ち上げる操作に対応する。この操作は例えばマウスを使う。

これを余り頻繁にやるとどれが新しいのかわからなくなる。ウィンドウ選択の記録をとるプロセスをedの中に仕込んでおき，ここにウィンドウ選択の経過をまた別のウィンドウに書いておく。このウィンドウ（Historyと呼ぶ）は，常に画面に出ている必要はない。

通常のE_0～E_nの編集テキストのウィンドウと，このHistoryウィンドウとは入力命令やマウスに対する反応が異ってよい。

このウィンドウに対するメッセージ解釈は2.2.2.Aの基本操作や2.2.4の抽象化機構に述べられたモジュール的な部分で処理される。すなわち，各ウィンドウを定義した時に，特定のメッセージに対する反応を仕込んでおく。（この部分は，特殊なユニフィクーション処理ともいうべきものとなる。）

どのウィンドウをディスプレイ上に具体的に出力するかはディスプレイ・システムの管轄下にある。各ウィンドウがディスプレイ・システムに出力要請をしたり，あるいは出力要求度に応じてディスプレイに出力する。

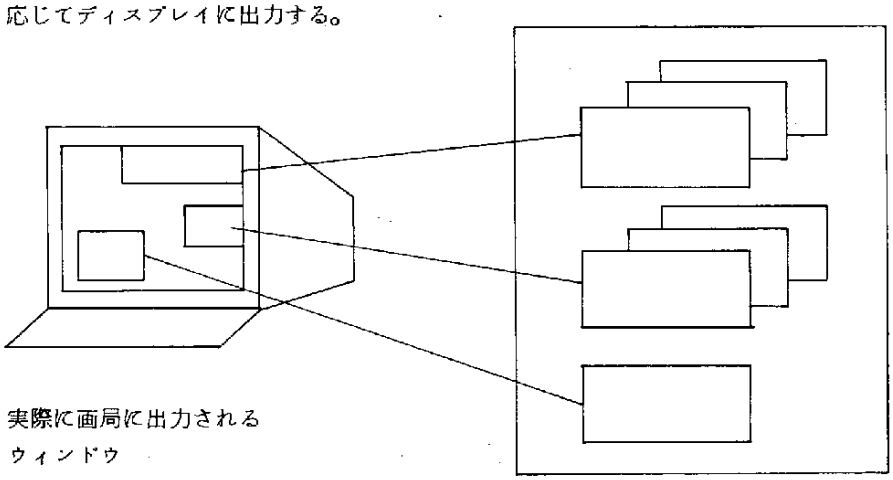


図2.4.14　ウィンドウの出力

出力時のフォント（文字形）指定も，論理的なウィンドウにおいては，ウィンドウの属性として定められているだけで，画面に出力される時に（ビットマップ上で）フォントが決められる。

ウィンドウの機能属性は2.5.2節のＴＶディスプレイ・システムで詳しく述べられるが，例えば，

- 境界表示
- フォント指定
- 図形表示
- 色指定

などがあり，これらのイメージ（より正確には画面上のビットパターン）は，論理的なウィンドウのいわばコード列としてのデータから，ディスプレイ・システムにより，具体的に画面

上に移されることとなる。

キー・ボードやマウスの入力も，ディスプレイ・システムに受け取られて，後に個々のウィンドウに渡され，次にチャネル変数にもってこられることになる。

E．ファイル・システム

データベースを支える概念の１つにファイルがある。ファイルには２次記憶媒体としての物理的イメージと，データの順序序合としての論理的イメージとがある。

一般に最近のオペレーティング・システムでは入出力機器をもファイルに含めることが多いが，これはデータの順序集合としての論理的性格をファイルの特性を考えるからである。

'ＦＧＫＬでは，ファイルの論理的性格をさらに前面に出して，概念チャネル（ conceptual channel[注)] ）をファイル処理の中心にすえる。入出力はすべてこの概念チャネルを通じて行われるものとする。

F．概念チャネルの構成

概念チャネルには属性と機能とが割りあてられる。

属性とは，チャネルのもつ種々の性質を言う。このチャネルにはどのようなデータが，どのような形態で，流れてゆくのかということである。

機能とは，このチャネルが現在入出力でどのように使われているかを示す。

機能面を独立させると，現在の入力チャネル，出力チャネル，あるいは，トレース用チャネル，エディット用のチャネルという具合になる。

概念チャネルの利点は，属性と機能を独立させ，同一のデバイス上に複数の概念チャネルを設定することによって，入出力でトラブルの原因となる副作用を封じ込める（ないしは活用する）点にある。

G．ファイルの物理的構成

'ＦＧＫＬ／Ｓでのファイルは５１２バイト単位のページから構成される。データの内容はすべて連続的な文字の集りであるとする。ファイル容量が５１２バイト以内のとき，このファイルは１つのページで構成される。

容量が５１２バイトを越えると，先頭のページはディレクトリとなる。このディレクトリがページ・サイズを越えると，先頭ページはディレクトリのディレクトリとなる。

一般に先頭ページは（ディレクトリの）^{n}ディレクトリとなる。

シーケンシャル・アクセスにおいては，ディレクトリ部分はアクセスできない。一方，ペー

注）Kurokawa，T.，“ Iuput-Output Facilities in LISP1.9 ”，SOFTWARE Practice and Experiences，８，pp.２７７－２８４，（１９７８）参照

ジ・アクセスにおいては，ディレクトリも含めたすべてのページが開放されている。ページ・アクセスによりファイル内のデータがどうなるかは，システムの問題としないところである。

ディレクトリの構成法は実際のファイルの物理的容量にも関係するが，ＵＮＩＸ^{TM} の構成法などが参考となるであろう。

H. システムの入出力命令

以下に述べる入出力命令は，ユーザ・レベルのチャネル変数を用いる処理ではなく，物理入出力に近いシステムのレベルの命令である。

(1) ファイル関係

- create（ファイル名，フラッグ）

  ファイル名のファイルを作る。フラッグはその結果を示す。ファイルの容量は指定できない。

- delete（ファイル名，フラッグ）

  ファイル名のファイルを削除する。

- open（ファイル名，ステータス，フラッグ）

  ファイル名のファイルを開く。ステータスはopen時の条件，（入力／出力，アクセス法等）を規定する。フラッグは結果を示す。

- close（ファイル名，フラッグ）

  ファイル名のファイルを閉じる。必ずしもend-of-fileを書き込むことを意味しない。end-of-fileはソフトウェア的に扱う。

- get（ファイル名，バッファ・アドレス，フラッグ）

  ファイルからバッファへの転送。転送単位は５１２バイト固定長とする。バッファ・アドレスの示す先がどうなっているかには無関係。getはシーケンシャル・アクセスを行う。

- put（ファイル名，バッファ・アドレス，フラッグ）

  バッファからファイルへの転送の転送単位は同じく５１２バイト。既存の容量を越えてputした場合，ファイルの容量が自動的に増やされる。

- readpage（ファイル名，ページ番号，バッファ・アドレス，フラッグ）

  ページ指定入力。

- writepage （ファイル名，ページ番号，バッファ・アドレス，フラッグ）

  ページ指定出力。

- addpage（ファイル名，フラッグ）

注）ＵＮＩＸは米国WESTERN ELECTRIC社の登録商標である。

ファイルの末尾に１ページ分容量を増やす。

- deletepage（ファイル名，フラッグ）

ファイルの末尾から，１ページ分削除する。

（コメント）

物理的なページの差し替え，番号を指定したページの削除はここでは除外してある。

また，ファイルのいわゆる物理的なセクター番号も一切，ここでは扱っていない。

処理能率を考えれば，デバイスの特性を考慮したファイル・システムが必要となるであろうが，'ＦＧＫＬ／Ｓが目標とするプロトタイプ実験では，物理的な機器の特性に余り深入りしたくないからである。

(2) ディスプレイ関係

- create-window（window名，window-class，条件）

window-classの下に，window名のウィンドウを作る。条件は主として，ディスプレイ時の条件（サイズ，配置など）を示す。

- delete-window（window名，window-class）

ウィンドウの削除。

- top-window（window名，window-class）

window-class下のウィンドウの内，window名 で指定したものを，トップに（画面に写るように）持ってくる。

- next-window （window-class）

window-class下のウィンドウを一枚めくる。現在，トップにあったものは最下位にゆく。

- change-working-window（window経験）

ディスプレイ全体を占めるworking-window を切り替える。これによって，ウィンドウの階層構造の中に入ってゆくことができる。

- change-window-face（window名，状態）

ウィンドウを現在表示されている状態から指定された状態へと変更する。位置や大きさなど。

- input-window（window名，データ）

ウィンドウから一文字分のデータを読み込む。

- output-window（window名，データ）

ウィンドウに一文字データを送る。

（コメント）

論理的なウィンドウ（チャネルもそうだが）はメッセージ処理機構で動いている。すなわち入力データに応じて適当な処理（たとえばそれをプログラム側に送る）をし，適当な出力を出す。論理的にはウィンドウはユーザとプログラムの間でメッセージを媒介する働きをもつと考えてよい。

しかしながら，マイクロプログラム・レベルではそんなことは出来ないので，上に示したinput-window，output-windowを使って，メッセージ処理を完成することになる。

ウィンドウの受け渡すデータはマイクロプログラム・レベルでは一字分だが，ユーザにとっては，漢字（２バイト）や，座標（４バイト）のこともあり，極端には，図形のように多数のバイトを必要とするものもある。

I．その他の入出力機器

ファイルとディスプレイ以外の入出力機器の代表としては，ネットワーク・インタフェース[注]がある。

プログラム上ではこれもチャネルで表わされ，大概はウィンドウが付属していて，ネットワークの状態をディスプレイ画面上に表示する。

以下にはネットワークも含めて，一般的に何らかの入出力機器が'ＦＧＫＬ／Ｓマシンに割り当てられるものとして，マイクロプログラム・レベルでの命令を示す。

- allocate（論理機器番号，物理機器番号）

  'ＦＧＫＬ／Ｓで扱う論理機器番号に，物理的な機器番号を与える。通常の入出力機器の場合，この操作は不要になるかも知れない。

- input （論理機器番号，データ）

  データに論理機器番号の機器から入力する。入力単位は通常１文字。ネットワークなどの場合，１パケットということもありうる。

- ontput（論理機器番号，データ）

  データを論理機器番号に出力する。

（コメント）

既述のファイル入出力も，ディスプレイ入出力も，細かいことを言えば，このinput

注）ネットワーク・インタフェースの考え方は問題になる。物理的なレベルとパケットなどの使用状況的なものとがある。使用状況，ネットワークの相手に対して適切な処理をする。

とontputによって実現される。ネットワーク関係のユーザに近いレベルの命令は，ネットワーク・インタフェースの項目で説明される。

2.5. インタフェース・サブシステム

2.5.1 インタプリタ（インタフェース部）

この節では 'FGKL／Sのユーザとのインタフェースについて述べる。ここでは，人間工学的配慮を払うことを第一に考えることにする。

A. 外部表現と内部表現

'FGKL／Sでは構造体の表現としてタプルを使用するが，その基本的外部表現を説明する。

タプルとはn個の要素の組であり，計算機内部では2.2項でも述べたように次のように表わされている。

g c	タグ1	長さ	第1項	タグ2	第2項	.....	タグn	第n項

このタプルの基本的な外部表現は次のようにn個の要素をコンマで区切って並べ，それを '＜' と '＞' でくくることにする。

＜第1項，第2項，…，第n項＞

この各要素には，定数，変数，さらにタプルを書くことができる。

定数は，文字アトムまたは数であり，この表現は通常のプログラミング言語と同様である。

例　append　　20　　1.5

変数の表現は，文字アトムの先頭に '*' 印を付けて書くことにする。

例　*x　　*variable

以上がタプルの基本的な表現法であるが，タプルの第一項に特別の意味を持たせて，関数あるいは述語などのオペレータとしてみるときには，次のように表現することもできる。

第1項（第2項，…，第n項）

この表記法は通常のプログラミング言語における関数呼出しと同じものである。

さらに，通常infix operatorとして使用される記号に対しては，'FGKL／Sでもinfix で表現することを許す。よって次の2つは内部表現としては同じものである。

1. a＋2／x

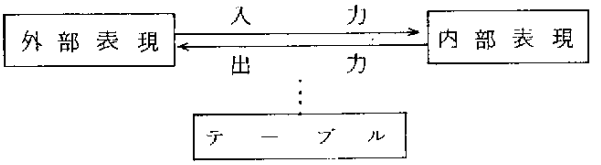
2. ＋(a , ／(2 , x))

なお，タプルはＬＩＳＰにおけるリストと表現法が似ているので混同が起こるかも知れないが，タプルはｎ個の要素が連続した領域にとられること，長さを自由に変えることができない，などの点で異なっている。リストを表現するには，＇．＇というinfix operatorを使用して，

a . *x . <>

のように表現する。

'ＦＧＫＬ／Ｓでは，以上のような外部表現を内部表現に変換する構文解析部で，テーブルを利用して構文解析を行うようになっており，ユーザはこのテーブルを書き替えることによって，自分の好みの表現法を利用することができる。また，出力部においてもこのテーブルを利用して，入力時と同じ形の出力が得られるようになっている。この様子を図に示すと次のようになる。



B．マクロ機能

前節では，外部表現と内部表現の関係について述べたが，この節では，ユーザがプログラムを書くときに書きやすくするためのマクロ機能について述べる。

ＬＩＳＰやPrologのようなインタプリタを基本とする言語には，マクロには，読み込み時に展開を行うものと，実行時に展開を行うものの２種類がある。読み込み時のマクロは，ＬＩＳＰのＱＵＯＴＥ記号＇のようにマクロ起動文字を定義しておくと＇abcという文字列を読み込んだ時に(ＱＵＯＴＥ　abc)のように置き替えてくれるものである。

これに対し，実行時のマクロはインタプリトを実行中にその述語を実行しようとする時に始めて展開を行うものである。これは，入力時に展開しないでおくとプログラムのデバッグやエディットの際に，入力のときに使用した形で出力されるのでユーザの混乱が避けられる。またデバッグが済んでコンパイルして実行するときには，当然，マクロを展開して行うので効率が悪くなることはない。

マクロを使用すると，シンタックスシュガーを容易に実現することができる。たとえば，エ

ジンバラ版Prologでは関数の値を評価するには 'is' という述語を使用しているが，これをマクロだと思えば

```
x  is  y+z
```

は

```
+(y, z, x)
```

と展開することにより処理ができる。このisというマクロを強力にすればisの右辺に関数の入れ子があってもそれを自動的に展開してくれるようになる。

例 *x is f(*n−1)+f(*n−2)

⇓

```
−(*n, 1, *n1)∧f(*n1, *x1)∧
−(*n, 2, *n2)∧f(*n2, *x2)∧
+(*x1, *x2, *x)
```

また，引数の数を不定にしたいような述語についても，マクロを利用することによって'FGKL／Sのインタプリタは引数の数が一定だと思って処理をすることができる。

マクロを定義するには，まず，マクロにすべき述語を宣言する。

例 macro(is)

次に定義を書く。

例 is(is(*x, *f(*y, *z)), *f(*y, *z, *x))←.

つまり，マクロと宣言された述語は2引数であり，第一引数は展開前の形であり，第二引数は展開後の形になるようにプログラムすればよい。

C．プリティプリント・省略プリント

前節までに述べた表現法は，いわば一次元の話であるが，プログラムやデータを見る場合には二次元的なものを見るので，そのための出力法も考慮しなければならない。

最も単純な例では，一行に入り切れないデータを二行に分ける場合に，単に定まった欄で切るのではなく区切りのよいところで切るようにする。

例
```
append(*x.*l1, *l2, *x.*l3)←app
    end(*l1, *l2, *l3)·

    append(*x.*l1, *l2, *x.*l3)←
      append(*l1, *l2, *l3)·
```

また将来，構造化制御機構を導入したときは，その構造を反映できるようにしておく。

定義されたプログラムの全体を出力するときには，同一の節頭を持ったものをまとめて出力する等というように，人間が出力を見る場合になるべく見易くなるように工夫した出力法を用いる。

また，データをすべて出力するだけでなく適当に情報を切り捨てた省略プリントの機能も重要である。

たとえば次の例に示すように，節を出力するときに述語名だけを出力したり，関数の入れ子をあるレベルまでしか出力しなかったりとする省略法がある。

例 1. p (*x , f n (*x , *y)) ←

q_1 (*x) ∧ q_2 (*y , *z) ∧ q_3 (*z , *ω) ∧

q_4 (*ω , *x , *y) .

⇓

p ← q_1 ∧ q_2 ∧ q_3 ∧ q_4 .

2. s (s (s (0) , s (t (s (0))) , t (s (0 , s (0 , 0))))

⇓

s (s (# , #) , t (#))

または

s (s (s (0) , s (#)) , t (s (0 , #)))

D. 日本語入出力

'ＦＧＫＬ／Ｓでは入出力はアスキー文字だけでなく，ひらがな・漢字等の入出力を行なうことも可能となっている。

入力の物理的手段としては，タブレット・符号化タッチ入力・ローマ字漢字変換などを用意してユーザの好みの方法で入力が可能である。出力はグラフィック端末上に，ストロークまたはドットで行う。

さらに，日本語文字をアスキー文字と混合して使用することも可能であり，述語名・関数名などにも日本語文字を使用してもよい。

例 1 足す 2 は *z

(これは ' *z is 1＋2 ' と同じ)

dictionary (本 , book) ← .

(第一引数の英訳が第二引数となる)

2.5.2 ＴＶディスプレイシステム

A. 概　要

○このディスプレイシステムは核言語 'ＦＧＫＬの開発を強力にサポートするエディタ，デバ

ッガ等の開発支援ツールの操作性向上と機能アップに貢献し，かつ'ＦＧＫＬ／Ｓマシン上で行われる評価用プログラムのアプリケーションを出来るだけ幅広いものにするために高解像度のディスプレイ画面に裏付けされたマルチウィンドウの概念と高度なグラフィク機能を兼ね備えたものになる。

- ディスプレイ画面はビットマップメモリ（bit-map memory）により構成され，この画面上に複数のウィンドウが階層的に表示される。ウィンドウには他のウィンドウによって全体が覆われて全く見えていないものと，一部が覆われて部分的に見えているものと，一番上にあって全てが見えているものとがある。
- ユーザはこのウィンドウを使って複数のプロセスを同時に処理することが出来る。各プロセスは異なったウィンドウを操作し，他のプロセスによってそのプロセスのウィンドウの情報や状態がこわされることはない。プロセスの切換えはマウス（mouse）を操作してプロセスに相当するウィンドウを指示するだけでよい。即ち他のウィンドウによって覆われ，部分的にしか見えていないウィンドウをつかみ上げて一番上に持ってきてそのウィンドウに対して処理を行うことが出来る。あたかも机の上に積み重ねられた書類のようにそれをめくったり，差し換えたりして目的のウィンドウを一番上に持ってきてその上で仕事を行うことが出来る。
- 'ＦＧＫＬ／Ｓマシンの入出力は4章に述べられているように概念チャネルに対して行なわれる。チャネルにはチャネル変数（機能と属性を示すもの）があり個々のオブジェクトに対してチャネル変数が割り当てられる。ウィンドウもこのチャネル変数を割当てることによって定義される。各ウィンドムへの入出力は割当てられたチャネル変数へメッセージを送ることによって行われる。メッセージはそのファンクションを指示する述語列によって構成される。述語にはウィンドウの形や位置を変えるものから，ウィンドウ内に文字や図形を出力させるものまで色々なレベルのものが準備される。グラフィック機能もこのメッセージのラージセットとして実現される。
- キーボードやマウスなどの入出力機器もこのウィンドウ単位で管理されそのウィンドウ毎に異なった処理が行われる。キーボードはプログラムによって現在選択されているウィンドウと接続され，入力はそのウィンドウに対してのみ行われる。マウスはスクリーン上を自由に動かすことが出来るが，そのカーソル位置を含む最も手前のウィンドウに対してボタン入力（click）が指示される。これら入力装置から入力するためのメッセージもウィンドウへ送られる。
- ウィンドウの内にサブウィンドウを作ることが出来る。このサブウィンドウをパン（pan）と呼び，そのパンを含むウィンドウをフレーム（frame）と呼ぶ。この概念はインタラクテ

ィブな処理を行うプログラム作成に効果を発揮する。メニュー画面やエラー処理用画面をパン上に構成しメニューの選択やエラー処理を行うことが出来る。パンは複数個定義出来，移動やサイズの変更，パンとフレーム，パンとパン間の情報の移動やパンとパン間のロジカル演算（ＡＮＤ，$\overline{\text{O}}$Ｒ，ＥＸ$\overline{\text{O}}$Ｒ）も可能である。

○ グラフィック機能については直線，四角形，三角形，円等を描く基本的な述語が準備される。（述語については後記する。）'ＦＧＫＬ／Ｓマシンではこれらの述語を組合せて'FGKL／Ｓ－グラフィックパッケージを作り強力なグラフィク機能をユーザに提供する。グラフィク・パッケージの製作に当っては，ＡＣＭ／ＳｉGGRAPH のグラフィックス標準計画委員会（ＧＳＰＣ）によって提案されているＣ$\overline{\text{O}}$ＲＥ[注]システムを参考にする。このパッケージを利用して'ＦＧＫＬ／Ｓマシン上でかなり高度なＬＳｉのマスクパターンの設計が行えるＣＡＤシステムが構築出来るであろう。

○ 日本語処理についてはＪＩＳに規定された符号系[注]を用いて行われる。漢字パターンとしては２４×２４ドットのフォントをＪＩＳ第１水準，第２水準ともにサポートする。漢字入力方式としてはカナ漢字変換方式，タブルレット入力方式の２方式で提供されるが将来的には手書入力認識も追加される予定である。

B．画面構成の基本オブジェクト

a．構　成

図2.5.1に示すようにディスプレシステムはユーザのアクセス対象とするウィンドウと，そのウィンドウが実際に表示されるスクリーン（ビットマップメモリ上に展開されることによって表示される）部分との２つに大きく分けられる。このウィンドからスクリーンへの変換を行うのがディスプレイシステムのコントロール部である。

ウィンドウはユーザが概念チャネルの１つとしてチャネル変数を定義することによって作り出される入出力の対象である。ユーザはこのウィンドウに対してファンクションを指示することによって目的の画面を作ることが出来る。このウィンドウへの入出力はメッセージの形でウィンドウへ送られる。送られたメッセージはウィンドウ毎に指示されたファンクションにもとづいて解読実行される。メッセージはファンクションに対する詳細述語で構成されている。以下に各オブジェクトの説明を行う。

b．ウィンドウ（Window）

ウィンドウはスクリーン上に設置される長方形の領域で，ユーザはこのウィンドウを任意

注）Computer Graphics Volume 1 3，Number 3，August 1 9 7 9．

注）JiS C 6226－1978「情報交換用漢字符号系」

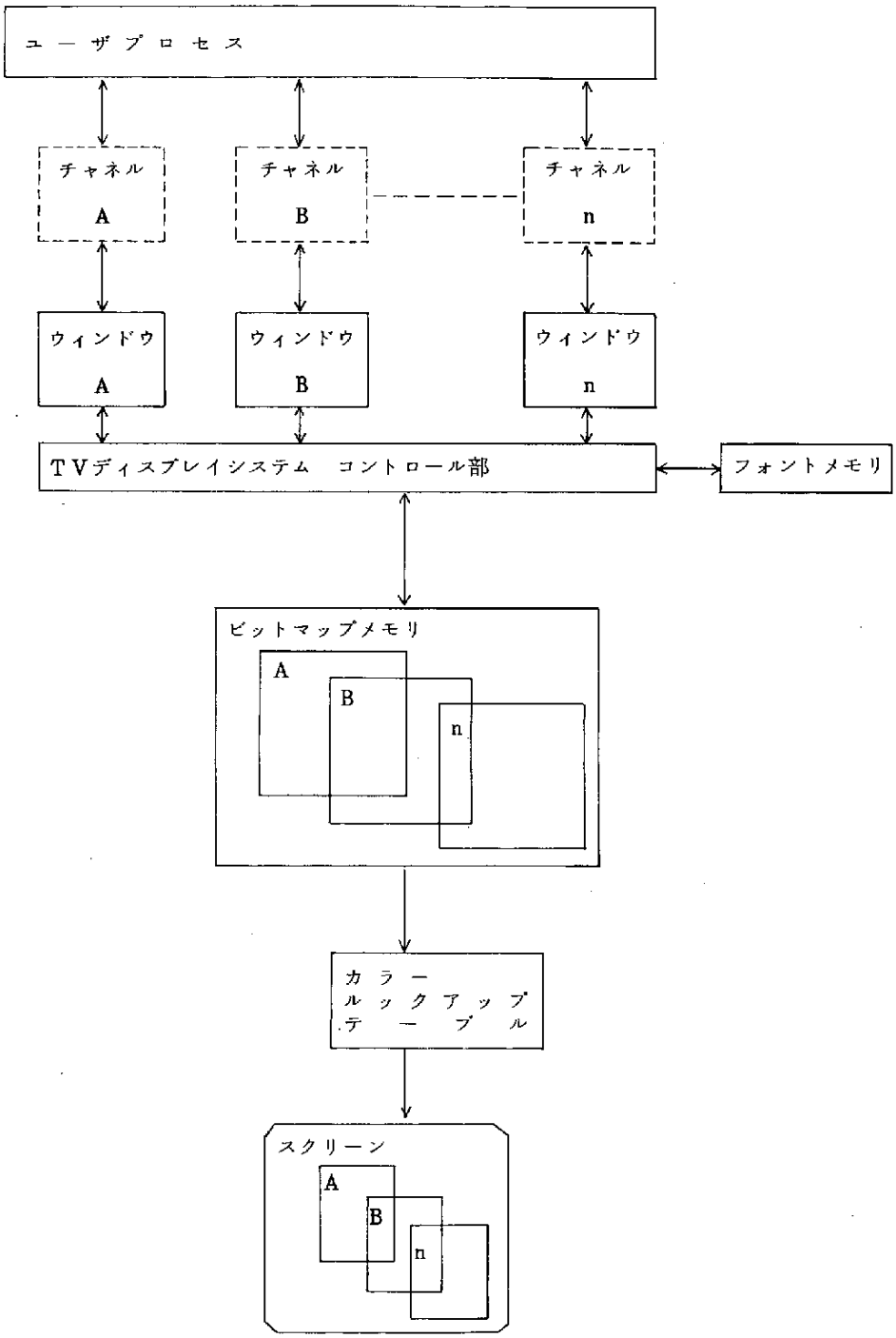


図 2.5.1　ディスプレイシステム構成図

の大きさで，任意の位置に配置することが出来る。ユーザはウィンドウをスクリーン上に複数個配置出来，個々にその機能や属性を定義可能で，全く独立した入出力の対象としてとらえることが出来る。ウィンドウはその機能を示す*functionとその属性を示す*attributeとを持っている。ディスプレイシステムはこの*functionと*attributeをもとにしてこのウィンドウへ送られてくるメッセージを解読し処理を行う。*functionと*attributeはユーザが定義することが出来る。*functionはサブセットの形でシステムがライブラリに登録している。そのミニマムセットはウィンドウをスクリーン上に表示する機能のみで，その属性としてウィンドウを活性化するか否か。スクリーン上に表示するか否かの指定が含まれる。ユーザはこのミニマムセットに必要なサブセットを結合して新しい*functionを定義することが出来る。*functionとして準備されるサブセットの代表的なものを次に示す。

- ウィンドウ境界の表示機能
- ウィンドウ名の表示機能
- マウスによるピックアップ機能
- キャラクタ入出力機能
- グラフィク機能
- フレーム及びパンの操作機能

*attributeは*functionによって得られる機能を実現するうえでより具体的な性質を規定するものであり，例えばウィンドウの境界線を表示する際の境界線の太さ，文字出力時の書体を決めるフォントの指定，文字サイズ（行間隔，列間隔），カラーグラフィクス時の色指定等を決定するのが*attributeである。

このウィンドウの生成，位置変更，サイズ変更重ね合わせ順の制御等はこのディスプレイシステムによって行われる。ユーザはこれらの要求をメッセージとしてこのディスプレイシステムへ送出することによってこれらの操作が行える。そのための述語が特別に準備されている。ウィンドウに対する入出力を行うための述語も準備されており，ユーザはウィンドウへ出力する場合に事前にそのデータを作る必要がある。そのデータは一般に述語列[注]で構成される。例えば次のようになる。

font（KBNJi）	① KANJiフォントを指定
curser（X，Y）	② カーソルを（X，Y）へ移動

注）これらの述語は内部データとして表現されるがその表現の方法としてはESCコードを先頭とするエスケープシーケンスの方法が一般的であるが，他にも種々の方法がありどの方法を取るかは今後の検討項目としたい。

text（'ＦＧＫＬ／Ｓマシン'）　　③ カーソル位置より「ＦＧＫＬ／Ｓマシン」の文字列を出力

c　スクリーン（ディスプレイ画面）

実際に文字や図形が表示されるところで，次項のビットマップメモリの内容がピクセル（pixel ＝表示単位）毎に表示される。表示はビットマップメモリに情報を書き込むことによって自動的に表示される。マルチカラー表示スクリーンの場合はビットマップメモリとスクリーンとの間にカラールックアップテーブルを一般的には配置しこのテーブルの内容を変えることによって容易に色の変更が可能である。スクリーン上のアドレスはスクリーンの左上隅を絶対原点（0,0）として左右にＸ軸，上下にＹ軸座標を取り，Ｘ軸は右方向に，Ｙ軸は下方向に進むにつれ値が増加する。単位は１ピクセル毎にカウントされる。

スクリーンはユーザ表示エリアとシステム専用表示エリアの２つに分けられる。システム表示エリアはこのディスプレイシステムがユーザに提供する情報が表示される。これには時刻，年月日，現在実行中のプロセスの状態（ＲＵＮ，ＳＴＯＰ，ＷＡｉＴ等），コンソールの使用状況等を表示することが出来る。ウィンドウはユーザエリア内に位置とサイズを指定して配置される。その場合ウィンドウの左上隅がウィンドウの原点（0,0）として，その原点がスクリーン座標のどの位置にくるかを指定する。サイズについては文字数（行数，列数）で指定する方法とピクセル数で指定する２通りの方法がある。

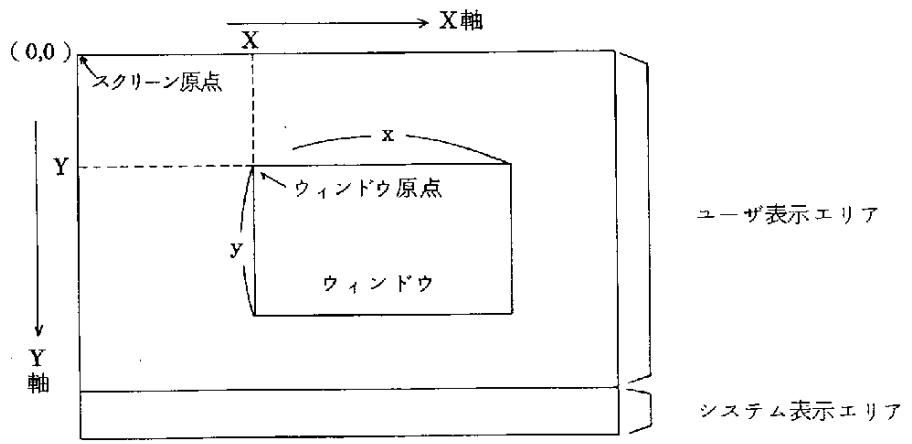
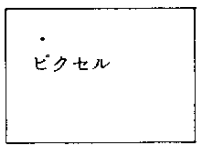


図 2.5.2　スクリーン上の座標

d．ビットマップメモリ（Bit-map memory）

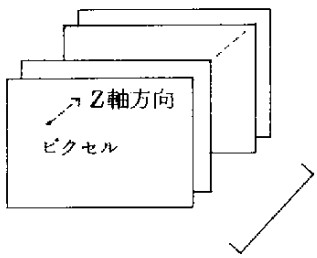
ディスプレイ画面の表示単位（ピクセル）に対応したメモリで白黒表示の場合は１ピクセル当たり１ビットが対応し，カラー表示の場合は１ピクセル当たり８ビット（２５６色同時表示の場合）が対応している。このビットマップメモリはユーザから直接アクセスすることはなく，ビットマップメモリの読み出し，書き込みは全てディスプレイシステムにて行われる。

（白黒）



１ビットで点，滅の状態を表わす。

（カラー）



８ビット（ｚ軸方向）で１ピクセルの色番号を表わす。

図 2.5.3　ビットマップメモリの構成

e．カラールックアップテーブル（Color look-up table）

カラー表示の場合，このカラールックアップテーブルによって実際の色指定が行われる。先のビットマップメモリの１ピクセルに対応する８枚（８ビット）の情報がこのカラールックアップテーブルの色番号（０～２５５）を指定する。各色番号にはRed，Green，Blueの階調を指定するためにそれぞれ４ビットが割り付けられておりこの組み合わせにより４０９６色を作り出すことが出来る。

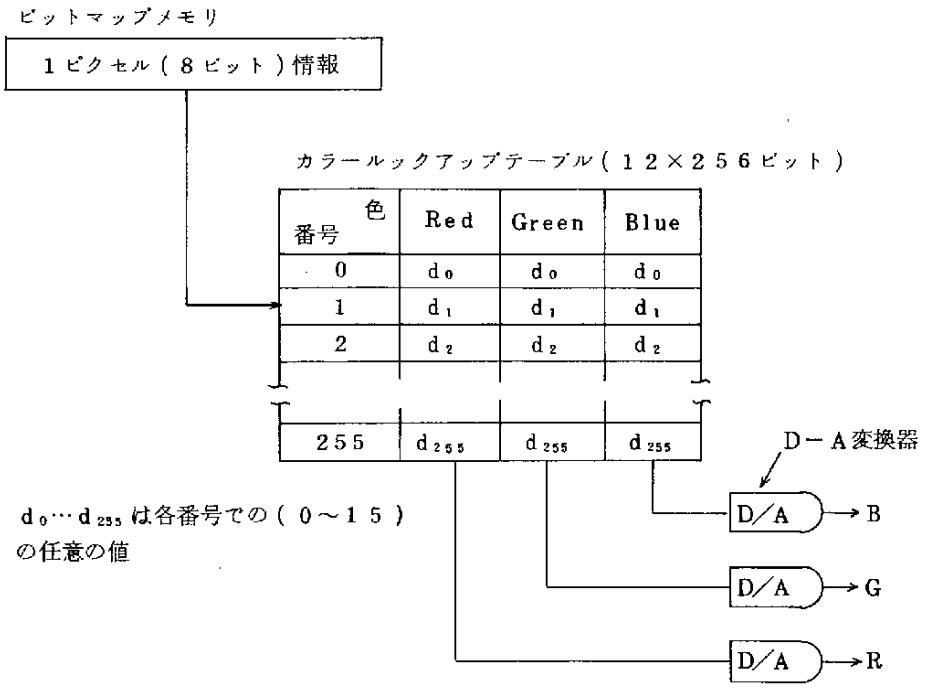


図 2.5.4　カラールックアップテーブルの構成

f. フォント（Fonts）

キャラクタや特殊記号は 'FGKL／Sマシン内ではコード化されて処理されている。このコード化された文字または文字列をユーザは各種の書体を使ってディスプレイ上に表示可能である。フォントはあらかじめ作成されてファイルに格納されている。登録されているフォントには名前がつけられており，各ウィンドウ毎に使用するフォントをこの名前で選択指定出来る。複数個のフォントを指定して同一ウィンドウ上に混在して表示することも可能である。指定されたフォントはフォントメモリ上に移されディスプレイシステムの管理下におかれる。複数のウィンドウが同一フォントを指定した場合は共有して使用される。いづれかのウィンドウで使用中のフォントは全てフォントメモリ上に配置されるよう管理される。

フォントには大別してドットマトリクス方式のものとストローク方式の２種に分けられる。一般にドットマトリクス方式の方が小容量のメモリで高品質の書体が得られる。ストローク方式の利点は書体を傾けたり，拡大したり，回転したりすることが比較的容量に行なえる。しかしストローク方式では一文字当りの容量が変化するためキャラクタコードとフォントメモリ上のアドレスとを対応さすフォントテーブルが必要となる。

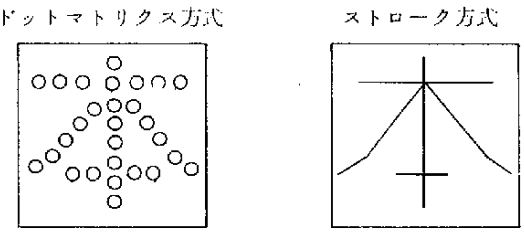


図 2.5.5　フォントの２方式例

C. 操作述語

a. ウィンドウ操作述語

(1) ウィンドウの生成

・ create-window (ウィンドウ名, *function, *attribute)

(2) ウィンドウの削除

・ delete-window (ウィンドウ名)

(3) ウィンドウの属性変更

・ change-window (ウィンドウ名, *attribute)

(4) ウィンドウの選択

・ select-window (ウィンドウ名)

(5) ウィンドウへの出力

・ write-window (ウィンドウ名, 述語列)

(6) ウィンドウの属性読み込み

・ sense-window (ウィンドウ名, attribute名)

(7) ウィンドウ内のデータ読み込み

・ read-window (window名, 述語列)

b. グラフィック述語

グラフィック機能を記述するために必要と思われる述語を項目のみ列挙する。個々の詳細な記述方法，述語の種類については既存のグラフィックディスプレイターミナルを参考にして決定される。

(1) セグメント管理述語

・ ディレクトリーのイニシャライズ

・ セグメントの登録

・ セグメントの削除

・ セグメントの結合

・ セグメントの出力

(2) テキスト操作述語

・ フォント指定

・ 文字サイズ指定

・ 文字回転角度指定

・ テキストの出力

(3) グラフィック述語

・ カーソル移動

・ 直線（ベクトル）

・ 円又は円弧

・ 三角形，四角形，扇形

・ テクスチャー（ぬりつぶし，ハッチング）

・ 楕円，２次曲線

(4) 画面制御述語

・ 拡大，縮小

・ スクロール

・ カラー変更

・ ビューポートの移動

(5) カーソルコントロール述語

・ カーソル位置のリード

・ カーソルのブリンク

・ カーソルのＯＮ／ＯＦＦ

・ カーソルの選択

c．記述列

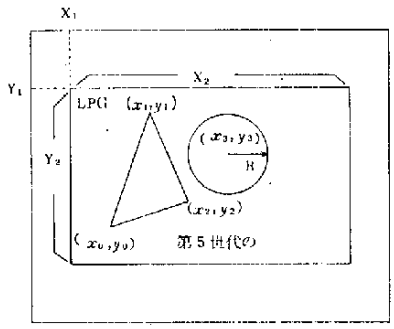
(1) create-window ('LPG', Graphics, position (X_1, Y_1)
・size(X_2, Y_2) ・point(0, 0) ・< >)

(2) select-window ('LPG')

(3) write-window('LPG', point (x_0, y_0) ・line (x_1, y_1) ・line (x_2, y_2) ・
line (x_0, y_0) ・circle (x_3, y_3, 10, 360°) ・<>)

(4) change-window('LPG', font(KANJi)・char-size(24,24)・<>)

(5) write-window('LPG', cursor(x_4, x_5)・text(「第5世代の」)・<>)



D. 装置仕様

ここでは 'FGKL／Sマシンのディスプレイ装置として要求される仕様について述べる。

a. 表示仕様

(1) 解像度

・白黒表示　　2000×2000ピクセル以上

・カラー表示　　1000×1000ピクセル以上

解像度については簡単な図形処理では1000×1000程度の解像度で十分と思われるが，本格的なグラフィックス処理ではより高解像度の画面が要求される。微妙な濃淡表示が可能なマルチカラー表示画面が要求される場合とモノクロ表示でも，より高解像度の画面が要求される場合もあるので，'FGKL／Sマシンではこのため白黒表示とカラー表示の2つのモデルを準備する。

(2) 画面サイズ

・20インチ程度

コンパクトな実装が要求されるが，1つの画面に多量の情報を表示するために高解像度の画面が要求される。その結果，表示が小さく見辛くなる傾向があり，人間工学の見地からも最小表示文字サイズは3㎜以上にする必要がある。(注)

ホントサイズや表示文字数等を合わせて検討する必要がある。

(3) リフレッシュレート

・・60HZ　ノンインタレース方式

チラツキ防止，高解像度の点からもこの仕様が必要である。

(4) 色・階調

注) DiN 66234 PART 1 では目と画面の距離が50cm以下の時でも最小3㎜と規定している。

・カラー表示　４０９６色

（同時表示は１６～２５６色）

・モノクロ表示　白黒表示

マルチカラー表示は当面のアプリケーションではそんなに多色を必要としないがＣＡＤシステム，画像処理，デザイン等へのアプリケーション拡大とメモリコストの低下を予測し４０９６色中２５６色選択表示可としたい。

b. 入力装置

入力装置としてはキーボードとマウスのようなポインティングデバイスを標準装備する。キーボードはＡＳＣＩＩタイプとＪＩＳタイプの２種を用意する。マウスについては今回の調査では十分な検討がなされなかったが今後の検討課題としてぜひ装備したい。

グラフィック処理，漢字フルキータブレット入力，手書入力，手書文字認識等の用途としてタブレットを接続する。サイズは３０cm×３０mm，解像度１０本／mm，サンプリングレート２００ドット／秒程度が必要と思われる。また手書入力認識（サインの認識等）ではＸ軸，Ｙ軸方向の情報のみでなくＺ軸方向の情報も必要である。

注）

c. その他の特殊装置

(1) ハードコピー

ユーザが要求する任意の時点でのディスプレイ画面の表示内容をハードコピーとして出力する。このプリンターには種々のレベルの装置が考えられるが，高価なレーザプリンタ，カラープリンタ等はローカルエリアネットワークに接続される形で提供される。

(2) イメージ入力装置

イメージ情報を取り扱う際にそのインプットは大変な作業となるので容易に入力する手段として，ビデオカメラ，イメージスキャナ，ファクシミリ等が考えられる。これらはローカルエリアネットワークに接続するか，他の伝送回線もしくは交換用媒体を利用して'ＦＧＫＬ／Ｓマシンへ入力される。

2.5.3 ネットワークシステム

A. 概要

'ＦＧＫＬ／Ｓマシンは，他システム，他の'FGKL/Sマシンと接続して，より大きな機能を実現する上で，ネットワークの構築，とくに，ローカルネットワークの構築が出来る機能を有していなければならない。

注）図面からのディジタイズを規定するともっと大型のものが必要となるが，この値は机上でのグラフィック処理，手書入力等の処理を想定したものである。

このローカルネットワークの具備すべき機能としては，一般的に以下のことがいわれている。

(1) 応答性……リアルタイム性，および大量データの高速伝送が必要であり，このため，伝送速度が１０Mbit／ＳＥＣのオーダが要求される。

(2) 高機能性……単なるデータ収集だけでなく，機器の起動，停止や制御を含み，通信と制御を融合させた高機能が要求される。

(3) 信頼性……ローカルネットワークとしては，構成するライン上でのインタフェース，コンピュータの故障がシステム全体に影響を及ぼさないよう，ケーブルルートの選択，ケーブルの２重化，インタフェースのバイパス機能などを考慮する必要がある。

(4) トランスペアレンシイ（透明性）……個々のシステムに固有性を持つことなく，自由にユーザがシステムを意識することなく利用出来ることが要求される。

ローカルネットワークとしては，現在種々の方式が提案，実施されている。代表的な商用ローカルネットワークとしては，Ethernet(Xerox)がある。このEthernet の動きに刺激され，Z－Net(Zilog)，Net／One（Ungermam-Bass），Local Net (Sytek)，ＨＹＰＥＲ channel（Network Systems），などが相次いで発表されている。これに対して，米ＩＥＥＥは，ローカルネットワークの標準化を１９８０年春から「プロジェクト８０２」として進めており，その標準案は，商用Ethernet に近く，それを拡張したものとなっている。

以上のことから，'ＦＧＫＬ／Ｓマシンが備えているネットワーク機能としては，Ethernet を基本とし，それを拡張したものとなっていることが必要である。

B．基本プロトコル

ＩＳＯ（国際標準化機構）は，１９７７年から異種コンピュータ・ネットワークを接続するための標準化を行っている。その標準案を，ＯＳＩ（Open Systems Interconnection）と呼ぶ。

図2.5.6は，ＯＳＩのプロトコルの階層図で，全部で７層からなり，下位から物理レベル，データリンク，ネットワーク，トランスポート，セッション，プレゼンテーション，アプリケーションと呼ぶ。

物理レベルは，システム間に物理的回線を設定，維持，解除するための電気的，機械的な条件を規定する。例として，ＣＣＩＴＴが定めたX.21，V.24，V.35 などがある。データリンク層は，システム間にデータリンクを提供し，その上でデータ伝送と誤り制御を行う。ＨＤＬＣやX.25 レベル２プロトコルが代表的な例である。３層目のネットワーク層は，１つまたは複数のネットワークを通じたルーチング，交換の機能を上のトランスポート層に提供する。

X.25レベル3プロトコルが代表的である。

レベル3は，情報のフレームの内容，すなわちパケット交換制御の大半を規定するパケットレベルの通信規約を定めており，端末と網間のデータリンク上でパケット多重通信を可能にしている。X.25には，バーチャルサーキット方式（相互通信する端末間に，網機能としてバーチャルサーキットが設定され，網が個々の呼びを識別できるパケット転送方式で，バーチャルコール（VC）と，パーマネントバーチャルサーキット（PVC）サービスがある。）と，データーグラム方式（呼びの概念がなく，端末はパケットごとに通信相手（アドレス）を指定する方式）がある。

ローカルネットワークでは，Ethernetをはじめとして，物理レベル，データリンク層の2層を規定し，それ以上の層については定めていない。これらのことをふまえて，'FGKL／Sマシンは，物理レベル，データリンク層の2層をサポートし，次のネットワーク層についてもサポート可能な設計としておくことが必要である。

パケットフォーマットとしては，Ethernet ，ＩＥＥＥ標準案については，図2.5.7のようになっている。 'ＦＧＫＬ／Ｓマシンのパケットフォーマットとしては，これらを包含したものとすることが必要である。

C．基本仕様

a．ハードウェア構成

'ＦＧＫＬ／Ｓマシンが基本とするEthernet のシステムは，図2.5.8のように構成され，基本性能は以下のようになっている。

データ伝送速度	１０Ｍビット／秒
ステーション間最大距離	2.5Km
最大ステーション数	１０２４
伝送媒体	同軸ケーブル
変調方式	ベースバンド
網形態	Branching non-rooted tree
アクセス方式	ＣＳＭＡ／ＣＤ

従って，'ＦＧＫＬ／Ｓマシンのハードウェアの基本性能は，上記のEthernet の基本仕様と同等もしくはそれ以上であってEthernet と互換性があることが必要である。すなわち，Ethernet では，衝突の処理に問題があり，'ＦＧＫＬ／Ｓマシンでは，この衝突の処理の部分がハードウェア的にも改善されていることが必要であると考えられる。

b．ソフトウェア構成

'ＦＧＫＬ／Ｓのソフトウエア構成は，図2.5.9に示すように，データリンク層におけるサポートが主体となり，Ethernet と同様に，図に示すようなモジュール構成に分解される。各モジュールは次のような処理を担っている。

(1) 送信データ組立部……上級階層が供給したデータに，誤り検出に必要なフレームチェックシーケンスを付加してフレームを作る。このフレームは，送信リンク管理部に渡される。

(2) 送信リンク管理部……キャリヤ検知信号を監視しており，通信中のトラフィックを優先させてチャネル上での他のトラフィックとの衝突を避ける。チャネルが開放されていれば，フレームの送信が開始される。そして，データリンク階層は物理階層へシリアルビットストリームを送出する。衝突が起った場合，Ethernet では，送信リンク管理部が，衝突処理を行い，衝突が起るつど再送信を行っているが，'ＦＧＫＬ／Ｓマシンでは，この衝突処理の改善が必要になると考えられる。

(3) 受信リンク管理部……キャリヤ検知信号がオンである事を確認しつつ，受信されているビットが送られてくるまで，待ちの状態となる。受信リンク管理部は，キャリヤ検知信号がオンである限り，物理階層からのビットを取込み続ける。キャリヤ検知信号がオフになった時，フレームは受信データカプセル分解部に引渡される。

(4) 受信データ分解部……受信フレームがこのステーションで受信されるべきものかどうか，フレームのディスティネーションアドレスフィールドを調べ，受けとるものであれば適当なステータス情報を付加し，このフレームの内容を上級階層に引き渡す。

D. ネットワーク述語

'ＦＧＫＬ／Ｓ言語によるネットワーク述語としては，例えば次のものが考えられる。

a. 送信

Transmit Data(*destination, *source, *type, *data, *CRC, *frame)←Data Encap(*destination, *source, *type, *data, *f_1)∧CRC Gen(*CRC, *f_1, *frame)

これは，送信データのフレーム組立と，フレームチェックシーケンスの生成述語である。

Transmit Mgmt(*frame, *carrier sense, *time, *length)←
Traffic(*carrier sense)∧Frame(*time)∧Send(*frame)∧
Collision(*length, *error)∧Retry(*error, *frame)

これは，組み立てられた送信データフレームを送信する述語であり，carrier sense 信号をみてチャネル使用中の判断をし，フレーム時間間隔を見て送信開始をし，衝突を検出すると，フレームの再送処理を行うというものである。

b. 受信

Receive Mgmt(*carrier sense, *length, *destination, *CRC, *frame)←Traffic(*carrier sense)∧ Detect(*length, *destination)∧Receive(*CRC, *frame)

これは，フレームデータを受信する述語であり，carrier sense 信号をみて受信フレームを識別し，フレームの長さ，アドレスを識別してアドレスが自己の物理アドレスと一致した場合，そのフレームデータを受信するというものである。

Receive Data(*destination, *source, *type, *data, *CRC, *frame)←CHECK(*CRC, *frame, *error)∧ Data Decap(*error, *frame, *destination, *source, *type, *data)

これは，受信フレームデータの各フィールドの分解をして，上位階層へ通知する述語である。

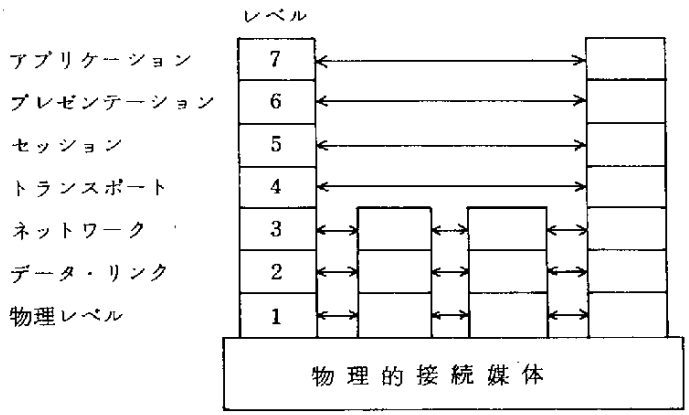


図 2.5.6　ＯＳＩの概念図

64	48	48	16	368～12,000	32（ビット）
プリアンブル	相手アドレス	ソースアドレス	タイプ	データ	CRC

(a) Ethernet

16M	16M	8	0～8N	32
相手アドレス	ソースアドレス	制御	データ	CRC
（1≦M≦3）			（N：整数）	

(b) IEEE標準案

図2.5.7 パケット・フォーマット

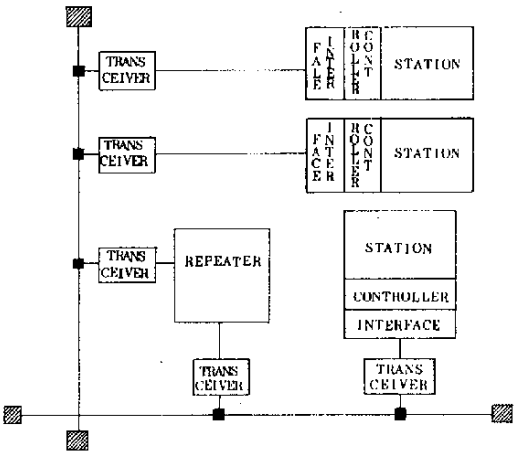


図2.5.8 Ethernet システム

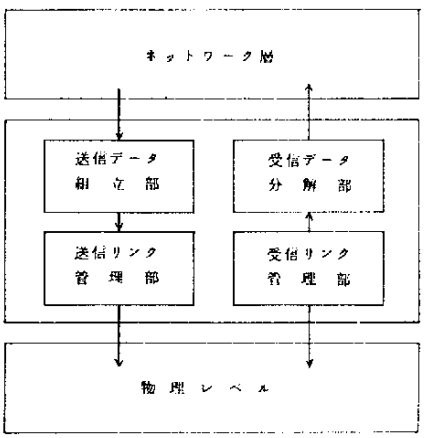


図2.5.9 ソフトウェア構成

参考文献

1) Metcalfe, R.M. and Boggs, D.R., "Ethernet：Distributed Packet Switching for Local Computer Networks," Communication of the ACM, vol.19, №7, July 1976.

2) The Ethernet－A Local Area Network, Data Link Layer and Physical Layer Specifications Version 1.0 September 30, 1980 Digital Eguipment Corporation, Intel Corporation, Xerox Corporation

3) 松本允介，坂井敏，小野欽司，浦野義頼：パケット交換網のプロトコル，情報処理，Vol.21，№8，pp.845－853（1980）

4) 桑原啓治：近付く普及期を前に商用化が活発なローカル・ネットワーク，日経エレクトロニクス，1981年7月6日号，№268，pp.110－135。

5) 上谷晃弘：ローカルコンピュータネットワーク，電子通信学会誌，Vol.62，№11，pp.1310－1316（1979）

2.6 ユーティリティ・サブシステム

2.6.1 エディタ

'FGKL／Sマシンを使用するユーザの作業時間の大半は，エディタを介したマシンとの対話で占められることになろう。したがって，エディタの設計においては人間工学的な使い勝手の良さが最も重視されなければならない。'FGKL／Sマシン用エディタは，'FGKLによるプログラミングを支援する機能と，図面を含むテキストの編集機能とを備えたシステムである。ハードウェアとして2.5.2節に述べられる高機能TVディスプレイシステム，ポインティングデバイス，イメージ入出力装置等を活用し，真に使い易い編集／清書システムを構築する。

以下に'FGKL／Sマシン用エディタが備えるべき基本機能とその特長について述べる。

A．画面エディタ

'FGKL／Sマシン用エディタは，TVディスプレイ上に表示される画面を2次元的に扱うオンライン画面エディタである。すなわち，画面上に表示されているイメージ全体が編集処理の対象となり，画面上で何らかの変更を施せばその結果が直ちに反映され，再表示される。編集作業は2次元画面上の1点を指示するカーソルを基点として行われる。

画面は複数の区画に分割して使用することができる（ウィンドウ機能）。各区画には異るプログラムやテキスト，又は制御コマンドレパートリ等を割り当てて表示させ得る。ユーザはこの機能によって画面上でカーソルを移動させるだけで制御対象を切換えたり，カーソルで指示

した制御コマンドをボタン操作のみで実行させたり，あるいは一時的にエラーメッセージを特定の位置に重ねて表示するといった操作が可能である。

キーボードは，カナ，漢字[注]，英字，数字，特殊文字等の文字キーと，数種類の機能キーを備えたものを用いる。また，とくに頻繁に使用されるコマンド用のキーを備えるのもよい。キーボードとシステムとの回線接続は全2重方式が望ましく，エコーネゴシェーション機能もサポートされるべきである。制御コマンドは原則として機能キーと文字キーの組み合わせによって入力されるものとし，テキスト入力とは明確に区別してユーザの徒らな混乱を避けるよう配慮する。

B．実行モード

実行モードとして，プログラミングモードとテキストモードを設ける。プログラミングモードでは'FGKL（および他のプログラミング言語）に依存する段落付け（インデンテーション）や，言語のシンタクスを考慮した扱いが可能である。[注]たとえば，'FGKLの特定のモジュール内の節体に含まれるtupleの特定のprincipal elementを検索するといった処理や，プログラムを見易い形でプリントする等の機能がある。このモードは，次節に述べられるデバッグ支援ツールとの密接な連携により，有機的な総合支援システムを達成するよう設計される。

テキストモードは和文，英文等の自然言語および図形を扱うモードで，語，文，パラグラフ，ページ等自然言語を扱う上で対象とすべき単位を意識した編集コマンドや，図形の縮小，拡大，回転等の制御コマンドが用意される。

上記実行モードとは独立に補助モードとして自動詰めモード，省略形モード，自動セーブモード等が指定できる。補助モードについては下記の各項目の説明の中で述べる。

C．文字入力と制御コマンド

a．文字入力

キーボードの文字キーの打ち込みにより，テキストやプログラムが入力される。ここでは文字入力方法の一例を示すが，細部については更に使い易い仕様に改められ，洗練されたシステムとすべきである。

文字キーが入力されるとカーソル位置にその文字が挿入され，カーソル以降が段落を単位としてすべて右へシフトされる。消去キーが入力されるとカーソルの直前の文字が消去され，カーソル以降は左へシフトされる。復改キーを押すと次の行の先頭へ制御が移行する。

注）漢字の入力方法はカナ漢字変換，又はタブレットによる入力等でもよい

注）Lispの構造エディタの概念を含む

補助モードとして自動詰めモードが指定されているときには，入力時に行の終了を意識する必要がない。このとき，プログラミングモードではシステムがステートメントの適切な切れ目を認識し，改行およびインデンテーションを自動的に行う。テキストモードでは「語」を認識し，語の切れ目で改行を行う。

b．制御コマンド

キーボードの機能キーと文字キーを組み合わせた入力と，ポインティングデバイス上のスイッチのクリックにより，種々の編集制御コマンドが実行される。

画面エディタではカーソルがすべての動作の基点となるため，カーソルの移動はコンパクト且つ柔軟に行えなければならない。ポインティングデバイスはこの目的に叶ったもので，物理的に動かした軌跡にしたがってカーソルが移動する。カーソル移動コマンドとしてプログラミングモードでは言語シンタクスに応じた各種のコマンドを用意する。テキストモードでは字，語，文章，パラグラフ，ページおよびファイル全体等を単位とする移動が可能である。

カーソルを目的の位置に移動した後，語，文等の単位を対象に消去，変換，置換等の編集処理を指示するコマンドが投入される。更に，印付け（マーク）コマンドによってある領域を設定し，その内容をレジスタに保存することができる。保存された内容は任意のカーソル位置に復元することが可能である。また，この領域を単位として，すべての小文字を大文字に変換する等の操作を行うことができる。

編集処理を支援する検索コマンドとして，文字列や'FGKLシンタクスを意識した高度なコマンドが用意される。例えば，'FGKLステートメント群から特定の述語を含むステートメントを検索し，リストアップするといった動作をアーギュメント付の単一コマンドで指示することができる。

編集時に誤ってテキストの一部又は全てを消してしまったり，思い直して消した部分を復活させたいときのためのコマンドも用意される。システムは消去コマンドによって除かれた部分をバッファに退避しておき，復活コマンドが投入されると退避された内容を再生する。また，ユーザが陽に版管理を行うことにより，誤った更新を行った版をキャンセルして，原状に復帰する手段も有効である。

システムは編集処理のための基本的なコマンドを準備しているが，ユーザが独自に新しいコマンドを作成したり，キーボードとコマンドの対応を変更して使い易くする等の拡張が可能である。追加，変更された機能はシステムに登録することにより，他のユーザも使用することができる。

補助モードの1つとして省略形モードを設定できる。このモードは，ユーザ毎に特定の文字列に対応する省略形を登録しておき，省略形が入力されたときにはシステムが自動的に正規の文字列に変換するというモードである。例えば，"wam" とタイプインすると，システムは"word abbrev mode" と書き換える。

D．ファイルとバッファ

データを保存するための基本単位はファイルである。個々のプログラムやテキストは通常はファイル中に存在し，編集作業を行うときにファイルの写しをシステム内のバッファに取り込んだ上で行われる。バッファ上で編集を施してもその結果が直ちにファイル内容に反映されるのではなく，コマンドによってファイルへの書き戻しが指示される。エディタが処理する対象はあくまでもバッファであり，ファイルとの対応や動的な実行モード，補助モード，カーソル位置，ウィンドウとの対応等はバッファ毎に属性情報としてシステムが維持する。

エディタは同時に複数個のバッファを並行して処理できるが，編集処理は各時点では1つのバッファに対して行われる。ユーザは簡単なキー操作や，ポインティングデバイスによるカーソル移動によって処理対象となるバッファを切換えることができる。また，バッファの消去やファイルへの書き戻しが単一のコマンドによって指示できる。

バッファ上で編集処理した結果が正しくファイルに保存されたか否かは常に確認しなければならない。このため，ユーザはすべてのバッファの状態を見ることができる（ディレクトリ）。バッファ上で更新が行われたにも拘わらず，ファイルへの書き戻しが行われていないものには印が付けられる。

補助モードとして指定される自動セーブモードは，ファイルへの書き戻し忘れやシステムクラッシュによって編集処理途上のバッファ内容が失われることを防ぐためのモードである。このモードが指定されると，システムが一定期間毎に自動的にファイルへの書き戻しを行う。自動書き戻し時には，例えば，前回の書き戻し時に比べて極端に容量が減少した等の異常と思われる事象を検出するとユーザに確認のコメントを送る。これによりユーザの操作誤りを未然に防ぐよう配慮する。

E．セルフドキュメント

エディタを使用する前に分厚いマニュアルを読破しなければならないようなシステムは煩わしい。FGKL/S マシンエディタは特定のキーを押すことによってその時点で投入可能な制御コマンドのリストや各コマンドの使用法に関する説明を表示する。この機能によってユーザはエディタを使用しながら使い方をマスターすることができる。また，ユーザが追加したり拡張したコマンドに関する問い合わせを行うこともできる。

F. 清書システム

テキストやプログラムを読み易い形に整形して出力し，ハードコピーをとればそのままドキュメントとして使えるような機能が備えられる。プログラムの出力に関しては5.1節に述べられているプリティプリント機能がある。テキスト出力に於てはランオフ方式の清書システムを備える。即ち，テキスト中に予めランオフコマンドを挿入しておき，特定のコマンドを投入すれば整形された出力が得られる。例えば英文テキストの右端をそろえるためのランオフコマンドが挿入されていれば語の間に適当なスペースを配分して右端をそろえ，見易い形に編集出力する。また，図面の挿入位置と大きさを指定しておくことにより，システムは図面の適度な縮小，拡大を行って指定位置への図面はめ込みを行う。

2.6.2 デバッグ支援ツール

デバッグ支援ツールはステッパを基本とする。従来用いられてきたトレーサはステッパの表示部のみ，またデバッガはコマンドインタプリタのみと考えることができる。

エラーやユーザによる割込みの際にもステッパに入ることにする。ユーザは実行状態を調べた後，実行を続けることもできるし，他の作業に移ることも可能である。

上記のような使われ方を想定した場合に，ステッパに必要と思われる機能について，以下に順に述べる。

A. 実行モードの切り換え

実行モードには大別して通常モードとステップ実行モードがある。ステップ実行モードはさらに，毎回コマンドを読み込むモードと，表示だけを行なって勝手に走るモードとに分かれる。ステップ実行モードは選択的にオンになるものとする — 定められた述語呼び出しだけで止まるとか，ある変数の値が更新された場合のみに止まるとかである。

ステップ実行モードでコマンドを読み込まない（あるいは毎回決められたコマンドが実行される — たとえば，ある変数の値を出力した後，ステップ実行を続ける — と考えても良い）のが，従来のトレーサに相当する。デバッグはディスプレイの前で対話的に行うのが標準であるとすると，このモードの表示は，充分人間に読みとれるスピードであることが要求される。1200 boudでは若干速すぎるし，300 baudでは遅すぎるという感じである — これは印字速度の目やすであって，回線の速度ではない。ハードコピーへの出力は別の様式が望まれる（後述）。

デバッグをする場合，まずバグの居場所をつき止める必要がある。その際には最初は上記の表示のみのモードで全体を眺め，ある程度の情報を得た後，（割込みを使って）コマンドを受け付けるモードに変換する。この場合にも必要な場所だけで止まるように指示できる必要があ

る。また必要な箇所を通り過ぎてしまった場合に最初からやり直すというのでは能率が上がらないから，ステップ実行のバックトラックも是非欲しい機能である。

実行モードの切り換えに関して必要なコマンドを列挙する。

- ステップ実行に入る
- ステップ実行を続行する（注）
- 通常モードに戻る
- 表示だけのモードに入る
- 実行をバックトラックする
- 表示及びコマンド受け付けの条件を決める（述語や変数の値による条件判断）
- 実行をやめる

最初の4つのコマンドには，各々，以後全部そうするか，下のレベルだけそうするかの2通りがある。コマンドの条件判断は，述語を用いても書けるようにするのが一般的であるから，そのためには述語呼び出しのパターンを返す述語parent-callが必要である。

parent-call（*x）

*x に親の呼び出しパターンを返す。

たとえば

p(*x)←parent-call(*y)へ　print(*y)

としておいて，p((abc))と呼ぶと，

p((abc))

が印刷される。

parent-call(*x, *y)

とした場合には，*x の親が返される。これを任意回くり返せばどんどん祖先がたどれることになる。

デバッグ支援ツールに限らず，すべてのコマンドは述語として呼び出せるようになっているのが望ましい。たとえば，ステップ実行に入るコマンドは，

step（述語呼び出し）

としても呼び出せるべきである。

B. 実行状況の表示

ステッパは述語呼び出しの状況を表示しながら実行してゆく。詳細はコマンドを使って見れ

注）1文字I／Oが可能なら，このコマンドは空白1文字が望ましい。

ば良いから，この表示はできるだけ簡潔であることが望ましい。必要な情報としては，

① 呼び出しのＩＤ（次節で使う）

② 呼び出し側のパターン

③ どの主張が使われたか

④ 変数の束縛状況

⑤ 呼び出しの深さ（レベル）

⑥ バックトラックの状況

等が考えられる。これらを，たとえば図2.6.1のように画面上に表示する。画面上の各エリアは独立にスクロール・アップ／ダウンができることが望ましい（これはソフトで制御してもかまわない）

C．環境の表示

環境とは次の３つをまとめたものを言う。

① 過去の実行の履歴

② 変数の値

③ バックトラックした場合の選択肢

これらがコマンドによって表示されなければならない。①や③は膨大な量になることが予想されるので，部分的表示が望まれる。その場合にはエディタの利用が効果的である。構造エディタを使うのならこれらの情報を構造体のままエディット（基本的には表示に使うのだが，変更もユーザの責任において許してしまうのが面白い）すれば良いし，ＥＭＡＣＳ風のファイル・オリエンテドなエディタを使うなら，いちどこれらの情報を文字列に変換してからエディットすることになる。構造エディタの方が，細部を無視した表示が可能であるので，膨大なデータを扱ってそこから必要な情報をとり出す（目的の部分の細部を表示する）のには向いていると思う。

①～③の各々の要素間の対応をつけるためにも，各述語呼び出しに一意的な名前が割り当てられていることが望ましい。これには，前節で表示されていた呼び出しI.D.を用いる。

D　環境の変更

デバック中には，色々な環境のもとでプログラムを実行してみることが望まれる。バグの出現は，たとえば1時間プログラムが走った後でないと到達しない場所かもしれないから，ひとたびその状況に到ったなら，そこでできる限り多くのテストができることが望ましい。そのためには，過去のある時点まで環境を戻す等，環境をコマンドによって変更できることが望ましい。

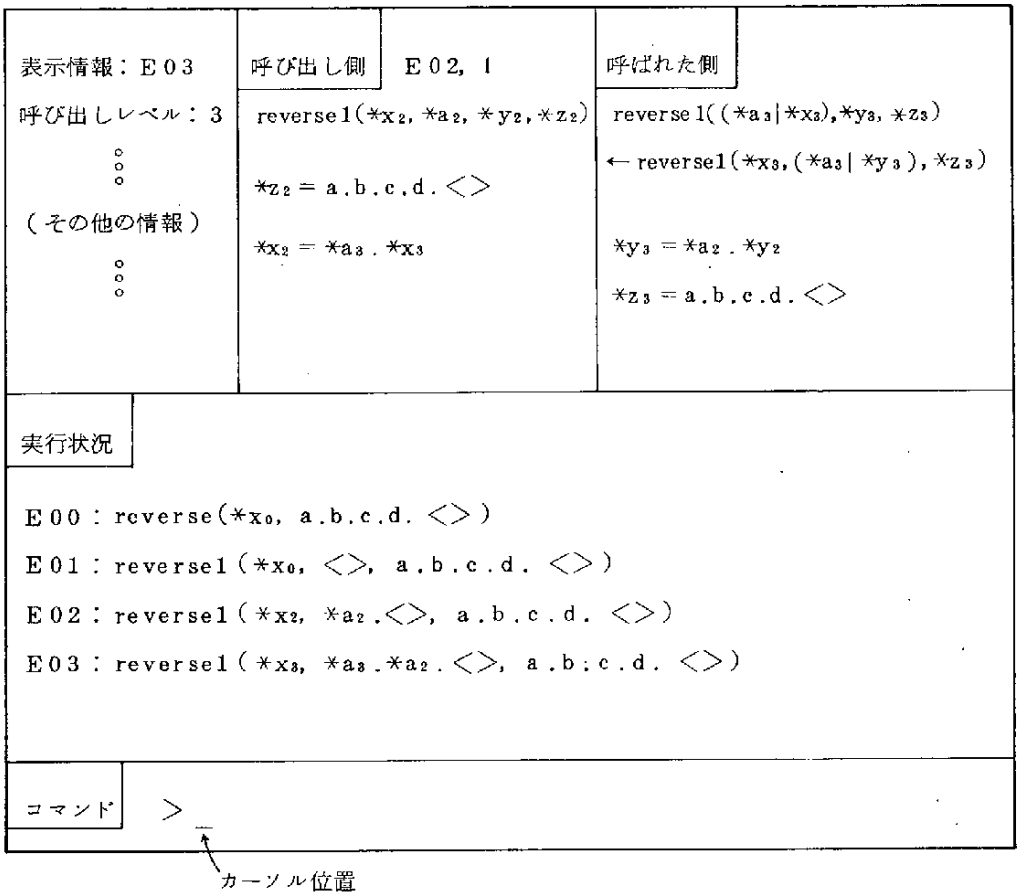


図261 reverse(*x, *y)←reverse 1 (*x, <>, *y)
reverse 1 (<>, *x, *x)
reverse 1 (*a.*x, *y, *z)←reversel(*x, *a.*y, *z)
において，←reverse(*x, a.b.c.d.<>)をステップ実行中のウィンドウ

環境変更のためのコマンドとしては，

① 過去のある時点の環境まで実行をバックトラックする

② 変数の値を変更する

③ 選択肢の順序を変更する

④ 現在の呼び出しを強制的に成功／失敗させる

特に①は，バグの場所を通り過ぎてしまった場合（選択的ステップ実行を行っていると良く起こる）に有効である。

E 述語呼び出し

ステップ実行中に，エディタの呼び出しや，その他のテストを行うために，述語呼び出しができることが望ましい。また，エラーでステッパに入った場合等，いちいちトップレベルへ戻らなくてもそのまま色々な述語が実行できる方が都合が良い。

F 拡 張 性

以上の節で述べた機能はすべてユーザにも定義できることが望ましい。そのためには，MaclispのEVALHOOKのような機能をユーザに開放しておけば当面は充分であろう。

G ハードコピーへの出力

ハードコピーへの出力は，後でゆっくり検討するためのものであるから詳細な情報が全部含まれていることが望ましい。

呼び出しの順序は縦に，バックトラックの順は横にといった2次元的出力が望ましい。大きさも各種とりそろえ，今のLP用紙より大きいものもあった方が都合が良い。

2.6.3 探索空間アナライザ

ある呼び出しが失敗となったら，その失敗の原因が取り去られるところまでバックトラックしなければ成功とはならない。原因となっているところまでのバックトラックによる試行錯誤は無駄な選択を繰返すことになる。そして，あるリテラルの失敗の原因を生む可能性のあるのは，同じ変数を含む既出のリテラルである。節内の変数間の依存関係が失敗の因果関係を表わすことになる。この性質を利用して，不必要なバックトラックを省略しようという手法が知的（選択的）バックトラックと呼ばれているものである。この手法を用い効率の良いインタプリタを作成しようという提案もあるが，ここでは探索空間のアナライザとして活用し，プログラムの最適化を進めるためのツールとして活用する。知的バックトラックモードで実行すると，失敗が生ずるたびにその失敗を回復できる所までの戻りの距離や探索木の状態が表示される。ユーザは，探索空間が小さくなるようプログラムの改良を行う。

さらに，より一般的には，論理プログラムの結合グラフ（connection gnaph)の展開（partial evaluation）によって次のような情報を得ることができる。

(1) regular horn（A←Bの形の節）のプログラムはmguを語とするオートマタになる。
e.g $\sigma_1 : (\sigma_2 : \sigma_3)^* ; \{\sigma_4 + \sigma_5\}$ 即ちloop構造を得る。

(2) 探索木のOR一分枝の分枝条件をあらかじめ計算できる可能性がある。→Cond文

(3) 結合グラフのmguを集める。それらは等式の集合でもある。（mgu $\{a/x, f(x)/y\}$ は $\{x=a, y=f(x)\}$ という等式と見なす）

等式
$$\begin{aligned} x_1 &= t_1(x_1 \cdots\cdots x_n) \\ &\vdots \\ x_n &= t_n(x_1 \cdots\cdots x_n) \end{aligned}$$

より，変数の型の無矛盾性，変数の値の範囲（例えば，$x=0$，$x=x+1$があれば，xの値は自然数の全体になることが予想される）入出力関係の方向性，無矛盾性のチェックができる（可能性がある）。

(4) 変数のinstantiationの伝播を解析し，サブゴールの実行順序を最適化できる。
（よりinstant されたゴールから実行する）

これらを利用して，高度な最適化や虫取りを行うシステム（アナライザ）を作ることができる。

2.6.4 ライブラリアン

従来のシステム構成において，ライブラリ関係は軽視される傾向があった。システムが本当にユーザに役立つものとなるにはユーザのライブラリ管理，ドキュメント管理を助けるライブラリアンが不可欠である。

ライブラリアンには2つの機能がある。

(1) 問題解決のプログラムをライブラリとして登録する。

(2) システムに関するドキュメントを管理する。

この両機能に対して'FGKLライブラリアンは，データベース機能と問い合わせ機能をもってユーザに対処する。（この問い合わせ処理自体を'FGKLの知的応用プログラムと考えることができる。）

A．プログラム・ライブラリ

プログラム・ライブラリはユーザ・プログラムや各ユーザに役立つと思われる論理関数式のひとまとまりを集めて管理する。

ライブラリに登録されるのは，

① 登録名称

② プログラム自身

③ 作 成 者

④ 登録日時（ライブラリアンが自動的につける。）

⑤ 変更，修正記録（同上）

⑥ プログラムの使用目的

⑦ プログラムの使用条件

⑧ 利用者に対する制限事項（利用者を特定することができる。）

⑨ コメント（ユーザが陽に書き込む）特徴，セールス・ポイント，使用料など

⑩ 利用者の評価

などといった項目である。

プログラム・ライブラリの利用形態には，

(1) ライブラリの登録名による指定

(2) ライブラリの使用目的による指定

とがある。

(1)の場合には，ライブラリアンは利用者に利用権がある事を確認してライブラリを渡す。（大抵はソース・コードで。コンパイル・コードの場合もある。）

(2)の場合には，ユーザとライブラリアンの間で問い合わせ処理が行われる。ユーザの使用目的に合致すると思われるライブラリをライブラリアンが示し，ユーザはその中から適当なものを選び出して使う。

プログラム・ライブラリにおいては，プログラムの評価が重要なもので，利用者が利用後このプログラムに対する感想，評価などを書くことができるようになっている。

プログラムのクレームなどは作成者のところに直接電子メイルで伝えてもらう。

プログラムの自動検定システムが出来れば，ライブラリ登録時にチェックするようにしたい。

a．機能クラスによる分類

プログラムの登録時には，プログラムの自動分類を容易にする意味からも，プログラムの引数に関する条件を揃える意味からも，機能クラス分類が奨励される。

機能クラスによる記述には任意性があるので，なるべく詳しく機能記述をするために，機能クラス登録サポート・プログラムを用意することが望ましい。

b．機能クラスの構成

詳細は文献〔Kurokawa 80〕（注）に譲るが，ここで機能クラスの概略を述べる。

注)〔Kurokawa 80〕Kurokawa, T., " The Function -class ", Conference Record of 1980 Lisp Conference, (1980)

機能クラスは本来,関数(function)を分類，記述することを主目的の1つにしていた。

'FGKLの場合には論理式の内，いわゆる定義体の分類，記述が1つの目的となる。

関数では入力と出力とがあり，機能クラスはこの両者の関係に注目したのであるが，論理式の場合には，一般には入力と出力とは区別されず，一様に入力（変数）として与えられる。ただし，2.2.4 節で触れたように，短期記憶のような形態で論理式の値を扱うことができれば，関数の場合と良く似た議論ができる。

論理式の定義体の各節では，変数が入力または出力の受け入れ口になるが，この場合各節毎に変数の個数が異なるので，機能クラスとしては，論理式の引数（個々の変数ではなくて，「，」で区切られた各々）を1つの処理単位と見なす。

機能クラスの成分としては，今述べた引数についての制限事項：

(i) 引数の個数

(ii) 引数のデータ型

(iii) 引数相互の関係（この場合は'FGKLの場合，定義体に含まれることが多い。これを定義体に含めるか，機能クラスに含めるかの得失は参考文献参照。）

(iv) 引数の条件付き処理

と，出力値についての制限事項：

(i) 出力の値についての制限

(ii) 出力と引数との関係（ FGKLの場合には，出力という概念の中に副作用を含めた方が良いのかも知れない。）

および，定義体の内容に関する項目：

(i) 再帰的定義を使っているかどうか

(ii) 他の下請け的述語にはどんなものがあるか

(iii) 定義体の基本的構造としてどのようなものが用いられているか　帰納法，選択方式，くり返しなど。

この内容に関する項目の中には，目的，作者を含めた，ライブラリアンの基本情報を含めることもできる。

上に示した各成分の記述に伴う困難点は，普通のやり方では記述が決して簡単ではなく，むしろ煩雑なために実施が困難なところがあった。

これに対して機能クラスでは同一の記述をもつ論理式（正確には論理式の機能クラス）を分類してゆくことにより，機能クラスの共通部分は特定のクラス名を宣言することで済ましてしまえる。

例えば，リストを引数とする機能クラスはまとめてlist class を形成し，このクラスでは引数のリストというデータ型が要求される。これに対して引数の個数が3個というクラスは，arg 3 classを形成し，3引数のリストをとる機能クラスは list-class と arg 3 class の共通クラスとして宣言できる。

従って，この機能クラス名は一種のマクロ的な働きをして，多くの論理式に必要な処理をこの機能クラスの部分で済ましてしまうことができる。

例えば引数として整数値をそのもの以外に整数値を結果として返すような整数式をとり，それ以外はエラーとしたい場合など，この機能クラスの部分にその処理をまとめておけばそれ以降の同様な仕事を必要とする論理式について一々処理内容を記述する必要がなくなる。

この種の処方の拡張としては，既存の論理式をライブラリに登録するときに，機能クラスを変更することで，適用範囲を広げるという手法がある。例えばリストの接続（append）のような場合に，引数の個数制限を撤廃し，機能クラスの処理で，引数のマクロ展開を処理しようとするものである。

c　機能クラス登録システム

機能クラスの登録に際してシステム・サポートのあることは機能クラスの整合化の面から非常に意味がある。

登録システムの形態には，①可能な要素（属性）についてYes／No を1つずつ尋ねる方式と②ある程度ユーザが自己申告をし，システムが抜けた部分をチェックして確認する方式，③システムが対象となる定義体を調べて，自動的に機能クラスを割り当てるもの，の3種類が考えられる。

①の形態はごく初歩的な知識工学応用システムとして作成可能であるが，ユーザにとって決して使い易いものではない。③の形式は記号実行（Symbolic execution）を必要とする。②の形式でより自由な混合指導（mixed-initiative）型の処理ができれば一番理想的である。

機能クラスの合成演算は参考文献に記した様にいくつか考えられるが，ここでは単純な共通部分をとる積操作のみを考えるとするとユーザがある程度自己申告をし，この申告した機能クラスがすでに何らかの名前をつけて登録済みかどうか，とか，矛盾が無いかどうかのある程度のチェックはできる。

その上で，一般的な①型の問い合わせで実行する手法は医療診断などでも良く用いられている手法である。

できれば，記号実行にある程度の最適化処理を絡めさせて，機能クラス宣言と実際の定義

との間の矛盾や，無駄なことをしていないか，というあたりの処理をしたい。そのような効果があれば，ライブラリ登録という作業が，単に他の人への親切心からだけでなく，自分自身の仕事のケジメとしても役に立つからである。

B．ドキュメント・ライブラリ

ドキュメントとして重要なものに，'FGKLのユーザ・マニュアルがある。マニュアル以外には，システムの現状のレポート，クレーム，使い方のヒントなどがある。

ドキュメント・ライブラリの内容の一部は電子メイル・システムによってサポートされる。

ユーザ・マニュアルについては，通常のマニュアルと同様にして検索する場合，エラーに対する処理の一部として用意される場合，初心者のための教育プログラムの一環として使われる場合と，管理担当者が内容を更新する場合とがある。

ユーザが通常のマニュアルのようにして検索する場合には，一種のコンサルテーションとして処方することができる。すなわち，ユーザの知りたい事項について，関係する項目をリスト・アップし，その中から適宜ユーザが内容を選ぶというわけである。

2.6.5 マイクロプログラム作成支援ツール

'FGKL／Sマシンは制御記憶を用いたマイクロプログラム制御の計算機であるため，マイクロプログラム作成支援ツールの整備がマシン開発に際しては極めて重要となる。またマシンが開発され，ユーザに提供された後においてもユーザ自身によるマイクロプログラミングを可能とするためには，使い易い支援ツールを用意することは不可欠なことといえる。ここでは次の4点につき重点的に述べる。

(1) クロスアセンブラ

(2) シミュレータ

(3) ハードウエアサポート

(4) マイクロセルフコンパイラ，アセンブラ

A．クロスアセンブラ

'FGKL／Sマシンの開発に並行して他の計算機上で動作するマイクロ用のクロスアセンブラを作り上げる必要がある。クロスアセンブラの動作環境は図2.6.2.が望ましい。第5世代コンピュータ開発のために用意される大型計算機上でＬＩＳＰあるいはＰＲＯＬＯＧ言語を用いてアセンブラを作成する。アセンブラ自身特に問題となりそうな点はないが，検討すべき項目としては発生したマイクロコードをどのようにして'FGKL／Sマシン上に持ち込むかという点がある。これは単に媒体（フロッピー，ＭＴ）を決めるだけでなく'FGKL／Sのイニシャルマイクロプログラムロード（ＩＭＰＬ）をどのように実行するか，あるいは'FGKL

／Sマシンの試験保守をどのように行うかという事と密接にからむ。基本的には試験保守の媒体として一番簡便なフロッピーディスクをマイクロコードの格納媒体として考えるのが素直であろう。従って大型計算機上で動作するプログラムとしては，クロスアセンブラだけでなく，マイクロコードのリンカー，制御記憶の管理プログラムも必要となってくる。特にフロッピーの媒体管理プログラムは試験保守の方式を充分理解して作成しなければならない。

B．シミュレータ

マイクロプログラムのデバッグは'FGKL／Sマシン上で行わなければならないため，マシンの開発当初においてはハードウェアとマイクロプログラムのバグの切り分けが難しい。このためマイクロプログラムのデバッグ効率が問題となる。これを解決するために作成したマイクロコードを事前に試験する必要があり，この目的のためにソフトウェアによるマイクロコードのシミュレータが有効となる。シミュレータを用いたマイクロコードの試験手続を図 2.6.3 に示す。シミュレータはユーザからの指示により擬似的な'FGKL／S のハードウェアを設定した後ユーザあるいはハードウェアによって指示されるマイクロ命令アドレスにもとづいてマイクロコードを取り出す。このマイクロコードが擬似ハードウェアに供給されるとシミュレータによって擬似ハードウェアが動作する。この結果をシミュレータはユーザに表示する。ハードウェアの表示結果はモードによっていろいろ変えることができる。たとえばマイクロ命令毎，決められたマイクロ命令を走行する毎，あるいは特定の命令アドレスに合致した時が考えられる。シミュレータの機能としてはハードウェアをデバッグする時に有効な次のような機能はすべて必要であろう。

(1) ハードウェア設定表示機能

レジスタ，ＬＳ等ハードウェアを自由に外部から設定・表示する機能である。

(2) マイクロ命令の走行モード可変機能

マイクロ命令毎，指定された命令数を実行した後に停止する機能である。

(3) アドレスコンペア機能

ユーザが指定したアドレスにマイクロ命令アドレスが一致した時に停止する機能である。

(4) 命令アドレス設定機能

任意のマイクロ命令から走行できる機能である。

本機能を持ったシミュレータをユーザは端末から自在に使用することによってマイクロプログラムのデバッグを効率的に行うことが可能となる。ソフトウェアによるシミュレータは図2.6.3のように擬似ハードウェアを計算機上で動作させるために，マイクロプログラムのデバッグにはかなりの計算機時間を必要とする。さらに，マイクロプログラムの開発と並行して擬似

ハードウェアを作成することも重要になってくる。シミュレーションを効率よく実行するには高いレベルでハードウェアを記述することも要求される。またユーザからみた時に書き易く，読み易いという目標も満足させねばならず，シミュレータが用いるハードウェア記述言語の選択には十分注意を払わねばならない。仕様記述の言語としてはＩＳＰ，機能記述の言語としてはＤＤＬ，ＣＤＬ，ＡＨＰＬ等があり，これらをうまく利用してシミュレータを作成しなければならない。

C．ハードウェアサポート機能

作成したマイクロコードを高速にデバッグするには'FGKL／S マシンを実際に使用することが考えられる。シミュレータによるデバッグはソフトウェアによるものであり，大型計算機を用いたとしても長大なマイクロプログラムをデバッグすることは時間的な点から困難となる。実際の'FGKL／S マシンを利用すれば時間的な問題は解決されるが，一方ハードウェアのバグも残っている可能性もあるため，マイクロプログラムとハードウェアの切り別けが重要となる。これが適切に行われなければ，ソフトウェアシミュレーションによるデバッグにくらべて必ずしも時間的に有利とは限らない。この切り別けを有効に行い，マイクロプログラムのデバッグを効率的に行うために次のようなハードウェアツールが是非とも必要となる。

(1) ＩＭＰＬ 機能

マイクロプログラム制御の計算機では必須であるが高速に外部媒体から制御記憶上にマイクロプログラムをロードする機能を'FGKL／S マシンでも備える必要がある。制御記憶をＲＯＭ化する案もあるが，ユーザマイクロプログラムをサポートするためには制御記憶はＲＡＭで構成せねばならず，ＩＭＰＬ 機能は必須となる。

(2) マイクロ命令の変更機能

マイクロプログラムのデバッグ中にバグを発見した時一時的に制御記憶上の命令を正しい命令に変更して試験を続行したいものである。この機能がなければ１つのバグが見つかる毎にマイクロプログラムを再アセンブルする必要があり，デバッグの効率が大幅に低下する。

(3) ハードウェアの設定・表示機能

'FGKL／S マシンの初期設定，マイクロプログラムを走行させる為の各種レジスタ，インタフェース等を設定する機能，さらに動作した後の結果をユーザに見易い状態で表示するための機能が必要である。これらの機能を用いればマイクロプログラムを分割してモジュール単位でデバッグすることも可能となる。

(4) 動作モード変更機能

マイクロプログラムの走行モードを制御できる機能であり，ユーザと一命令進む毎に会話

するか，ユーザの指示があるまで命令を連続実行するかを区別する。

(5) マイクロ命令の走行・停止

ユーザの指示により命令を任意の時点で停止させたり進めたりできる機能である。

(6) アドレスコンペア機能

マイクロ命令アドレスがユーザが指定した特定アドレスに一致した時命令が停止する機能である。

(7) 命令アドレス設定機能

ユーザが指定した任意のマイクロ命令から走行可能なように，アドレスを自由に変更できる機能である。

以上の様なハードウェアの機能を用いれば実際のマシン上でマイクロプログラムを効率的にデバッグできる。

D．マイクロセルフアセンブラ，コンパイラ

'FGKL／S マシンは第5世代プロジェクトのソフトウェア開発用パイロットマシンであるためにユーザがこのマシン上でマイクロプログラムを開発する機会が多くなる。この時ユーザマイクロプログラムの開発効率をあげるためには，'FGKL／S マシン上で動作するセルフアセンブラが是非とも必要になる。この場合もクロスアセンブラと同じように発生したマイクロコードをどのようにして自身の制御記憶および次の IMPL 時のための補助記憶媒体に格納するかが問題となる。図 2.6.4 にセルフアセンブラの動作環境を示す。図中のディスクに格納されたマイクロコードは制御記憶管理プログラムの制御のもとに所定の制御記憶領域に割りつけられる。割り付けられたアドレスに基づいて制御記憶更新プログラムはディスク内のマイクロコードを制御記憶にロードする。図 2.6.5 にユーザマイクロプログラムコードを制御記憶上に登録するまでのステップを示した。登録されたマイクロプログラムは適当な時期に'FGKL／S 全体を制御している SVP によって外部媒体（ここではフロッピーディスク）に格納される。

マイクロコンパイラは次の目的のために用いられる。ユーザマイクロプログラムを作成することは通常のユーザは大変である。なぜならば'FGKL／S のハードウェアを十分理解しなければマイクロプログラムの作成は不可能となるからである。従ってユーザが FGKL の機械語あるいは Kernel 言語を用いて作成した関数をコンパイラが自動的にマイクロプログラムまで展開できるならばユーザの負担は非常に軽減される。

このようにマイクロアセンブラ，コンパイラを'FGKL／S マシンの上で実現できれば第5世代プロジェクトのソフトウェア開発は非常に効率的に行えることが予想される。

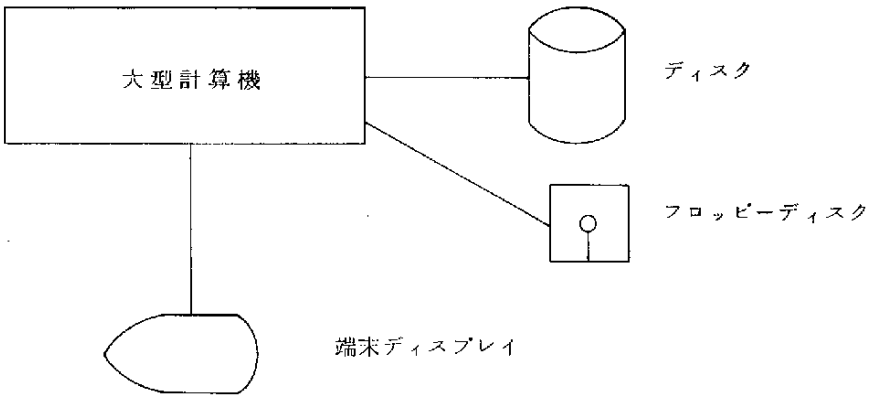


図 2.6.2　マイクロクロスアセンブラ動作環境

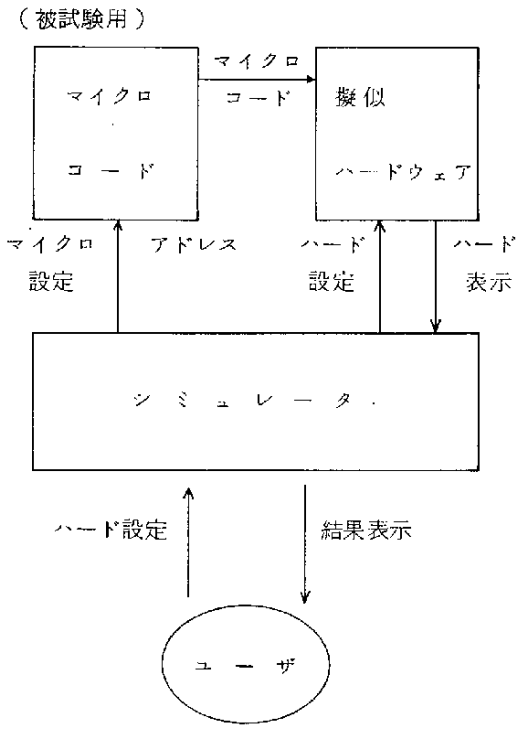


図 2.6.3　マイクロコードのシミュレータ

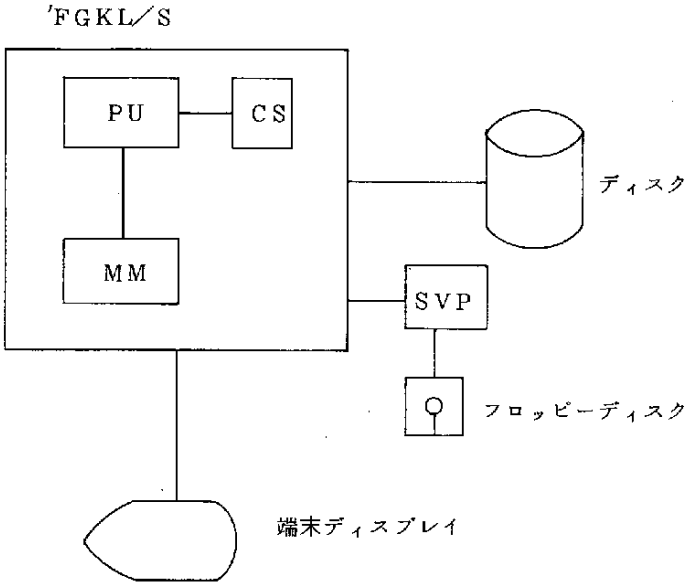


図 2.6.4　マイクロセルフアセンブラ動作環境

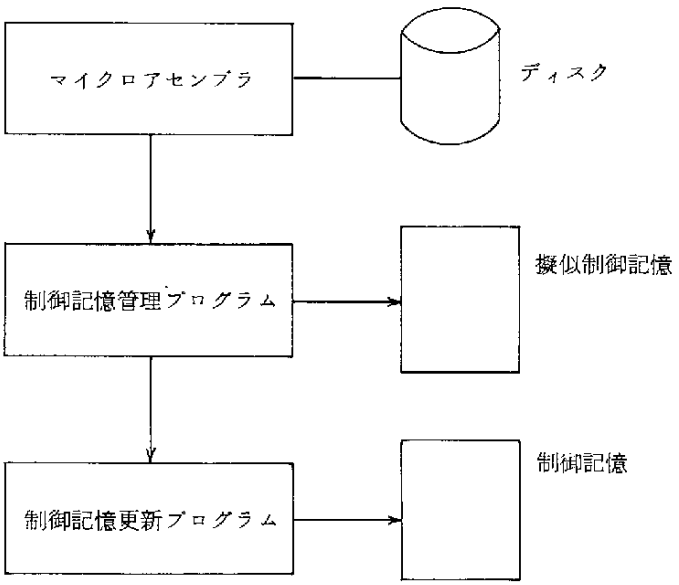


図 2.6.5　ユーザマイクロプログラムの登録

第3部　ロジック・プログラミングの基礎理論

3. ロジック・プログラミングの基礎理論

3.1 言語仕様の論理学からの解釈

この節では論理型の算法言語であるProlog に対するいくつかの論理的考察および論理的拡張について述べる。しかしながら，ここでは言語仕様全体に対する統一のとれた論理学的あるいは数学的な解釈法について述べるのではなくその中で重要と考えられる話題について非形式的に取り上げる。また，Prologの今後のさらなる展開のために，Prologとは異なる論理型言語である Lucid や等式論理に基づく算法言語についても触れその特質を見ることにした。

次にPrologのような論理型言語の意味論およびその論理的拡張を考えるとき論理的概念の源となる論理学について少し眺めてみることにする。論理学は大きく西洋論理学（形式論理学，内容（哲学的）論理学）と東洋論理学（因明学（印度），陽明学（中国））に分かれ形式論理学はさらに演繹論理，帰納論理に分類される。ここでは特に演繹論理の中で現代に入って急速に発展させられてきた記号あるいは数理論理学に焦点を当てる。数理論理学には代表的なものに次のような論理系がある。

- 古典論理
  - 一階論理*，高階論理*（型の理論）
  - 無限長論理
  - 組み合わせ論理　など
- 非古典論理
  - 様相論理*，内包論理*
  - 多値論理
  - 直観論理*
  - 量子論理　など

＊印のついた論理で提供される概念の利用，適用法については直接的あるいは間接的に以下の節で述べられる。残りの論理については，さらに複雑な情報処理を論理型言語によって表現しようとするとき，それらの論理固有の論理的方法が有効な概念を与えることになるであろう。

3.1.1 ロジック・プログラミングの原理

A　論理と計算

計算（Computation）とは，部分帰納関数（計算可能な関数）の値を体系的に決めていく方法とみなすのが伝統的である。ある点ではより一般的な他のアプローチもあってそれは計算というものを公理から定理を生成していくプロセスであると考えるものである。すなわち計算

に対する演繹的アプローチである。

計算を論理系としてとらえるという発想はKawalski[1]に始まる述語論理をプログラミング言語として解釈する研究にその発端があるのではなくすでに計算の理論研究の中に見い出される。例えば，Herbrand-Gödel Computability[2]では帰納関数の値を一歩一歩決めていく計算過程を形式体系の中でとらえ直し計算可能性を超数学的に考察していこうとする。

計算に対する演繹的アプローチでは関数あるいは関係の演繹的計算による値の取り出しのみならず命題の真理性（証明可能性）をも問うことができる。このような計算と論証が同一のレベルで遂行できる計算（推論）メカニズムは将来の知識情報処理システムにぜひとも必要な情報処理メカニズムである。さらに，計算を論証と見ることによって手続き性をあまり要求しない記述的な算法言語の研究にも有力な方法論を提供している。

B　論理型言語

ここでは代表的な論理型言語であるProlog[3]，Lucid[4]，等式論理[5]にもとづく算法言語について概観する（表3.1.1参照）。

すべてに共通しているのは非手続き的言語であることと，プログラムの性質の証明に適しプログラムテキストからの直接的証明が可能である点である。また，プログラムのシンタックス上ではLucid と等式論理は類似し，プログラムの計算過程はProlog と等式論理はよく似た側面をもっている。

以下に階乗プログラムのこれら3者によるプログラムを上げその特質を見る。

Prologプログラム

```
fact ( 0, s(0)).
fact ( s(x), u ) ← fact ( x, v ), times ( s(x), v, u ).
```

等式プログラム

```
fact (0) = s (0)
fact ( s(x)) = times ( s(x), fact(x))
```

Lucid プログラム

```
n = first input
first f = 1
first i = n
next f = f × i
next i = i − 1
output = f as soon as i = 0 .
```

比較項目 ＼ 論理型言語	Prolog	Lucid	等式論理
•基礎論理系	一階論理の部分系（Horn論理）	様相的（時制）論理	一階論理の部分系
•論理的完全性	完全	不完全	完全
•計算の方式	導出原理	コンパイル方式	リダクション
•記述系			
論理記号	∧，→ ¬（間接的）	¬，∧	なし
述語記号	有	＝	＝のみ
関数記号	有	first，next，as soon as など	有
•アルゴリズム（仕様）構成の基本概念			
決定性vs.非決定性	非決定性	決定性	決定性
関数型vs.関係型	関係型	関数型	関数型
•意味論	モデル論的意味	モデル論的意味	モデル論的意味
	操作的 〃	操作的 〃	操作的 〃
	指示的 〃		
•プログラムの証明	適している	適している	適している

表 3.1.1　論理型言語の比較

Prolog は階乗プログラムをその入力と出力の関係間に成立する性質をHorn 論理式で記述し関数プログラムを関係プログラムとして表現している。等式プログラムは入力と出力の関数関係を陽に等号記号で結び，等式で表現している。この両者は，Prologプログラムから出力変数を除去することにより等式プログラムが得られ，逆に等式プログラムに新たな変数を導入し関係的にとらえることによりPrologプログラムが得られるという意味で互いに関係している。そしてこのことはプログラムの仕様記述段階での関係型プログラムの記述に適したPrologと関数型プログラムの記述に適した等式論理の記述の融合のための一方法を示唆している。Lucidプログラムは ALGOL風のくり返し構造で書かれた階乗プログラムをそのくり返し構造をずたずたにし代入文を等式としてしまうことによって得られた形式をしている。そしてその記述は構造をもたない単なる論理式の集合となっている。このようなLucid の非手続き性はfirst，next，as soon as，・・・，といった変数の実行履歴上の洩算子の導入によって可能となっている。これはPrologに対して新しい静的な制御構造の導入の指針を与える。

3.1.2 Horn 論理，Prologの意味論

通常のプログラミング言語に対する意味論には大別して，操作的意味論（operational），公理的意味論（axiomatic），指示的意味論（denotational）の3つの意味論がある。プログラミング言語としての述語論理のHorn 文に対してもやはり上記3つの方法論からの意味付けの仕方があり得る。[6)]

A　宣言的意味と手続き的意味

1つのHorn クローズの宣言的意味は通常のモデル論的意味によって定められ，プログラムとしてのクローズの集合｛C_1，・・・，C_n｝はconjunction $C_1 \wedge \cdots \wedge C_n$ として解釈され，さらにそれらが変数x_1，・・・，x_n を含んでいるときそれらの変数はすべて暗黙のうちに全称的に束縛されているものと見なされる。他方Hornクローズの手続き的解釈においてはHorn クローズの形によってそれぞれ停止ステートメント，ゴールステートメント，手続き宣言文といった解釈が与えられるが，クローズの集合としてのプログラムの手続き的意味はクローズの順序やHornクローズの右辺のゴールの並びの順序の指定の仕方によって変わり，単なる1つ1つのクローズの手続き的解釈の集合体ではない。この点がクローズの宣言的解釈と異なる。

B　Horn論理の種々の意味論

a　操作的意味論

通常のプログラミング言語の操作的な意味はその言語のコンパイラあるいはインタプリタ

を与えることによって意味が定められたと考えるものである。そしてコンパイラ，インタプリタの記述は具体的なマシンを想定している場合とアブストラクマシンを想定する場合とがある。

Horn 論理の操作的意味論はHorn 論理式の証明手続をHorn 言語のインタプリタと見なし，したがって，証明手続きが意味を定めるものと考えるものである。

この意味論では，Horn 文の集合Γに現われるn項述語記号PはΓのHerbrand 領域上のn項関係を指示する。すなわち，操作的意味論はPの指示対象D(P)を次のように一意に定める，

$$D(P)=\{(t_1,\cdots,t_n)\mid \Gamma\vdash P(t_1,\cdots,t_n)\}$$

ここで各t_iはΓのHerbrand 領域の元を表わし，$\Gamma\vdash P(t_1,\cdots,t_n)$は$\Gamma\cup\{\sim P(t_1,\cdots,t_n)\}$の導出原理による反証証明が存在することを示す。

次に，Hornプログラムでは出力結果を与えてそれを生ぜしめる入力を求めるというような計算を操作的意味論としての導出原理を用いて行なうことができる。その妥当性は次の2つの事実にもとづく。

「Γが無矛盾でかつ$\Gamma\vdash P(t_1,\cdots,t_n)$であれば各変数$x_i$が項$t_i$に具象化されるような$\Gamma\cup\{\sim P(x_1,\cdots,x_n)\}$の反証が存在する」，

「Γが無矛盾でかつ$\Gamma\vdash P(t_1,\cdots,t_n)$であればPの引数$t_1,\cdots,t_n$の任意の部分集合に対して，入力としてPのその引数をとり出力として残りの引数を計算するような計算（反証）が存在する」。

一階論理のモデル論的意味論に従ってHorn 文の集合Γに含まれる述語記号の指示対象D'(P)は次のように与えられる，

$$D'(P)=\{(t_1,\cdots,t_n)\mid \Gamma\models P(t_1,\cdots,t_n)\}$$

ここで各t_iはΓのHerbrand領域の元である。このとき，一階論理の完全性によってD(P)＝D'(P)であり，操作的意味論はモデル論的意味論と同値であることがわかる。[注)]

b　公理的意味論

通常のプログラミング言語の公理的意味論の方法はプログラムの正当性の証明に代表されるプログラムの諸性質の証明の方法を論理系として提示するものと表裏一体をなしている。[8)]そしてその方法は大きく分けて次の2通りの方法がある。

注）Clausal logic，Horn logic のモデル理論の詳しい説明については JIPDEC 資料第5世代の電子計算機に関する調査研究報告書，昭和55年3月[7)]を参照

(1) プログラム言語と証明言語が分離されている場合，例えばMannaの方法[9]

(2) プログラム言語と証明言語が一体化している場合，例えばHoare[10]，Pratt[11]の方法。

(1)の場合はプログラムの意味するものを証明言語で表わし，それが公理的に証明されるという方法が取られるが，Horn形式で書かれたプログラムはそれ自体でプログラムであると同時に一階述語論理という証明言語で表現された意味記述でもあるという2面性をもつことになる。この意味において，Horn論理はプログラムの公理的意味論の枠組そのものを与えている。(2)の立場からのHornプログラムの公理的意味論の方法に対しては今後の研究が必要である。

c 指示的意味論

この意味論については3.4節で詳しく論じられるので，ここでは触れない。

C Notの意味

ここではNotの意味について諸側面から考察し間接的にその意味を特徴づける。Notは Prologのシンタックス上特異な存在でありその操作的な意味は明確であるが，Hornプログラムの論理を複雑にする要因の1つになる場合がある。

NotはPrologでは次のように定義される，

```
{ not(P) ← P, !, fail.
  not(_).
```

すなわち，Pのすべての可能な証明が失敗すればnot(P)が成功し，そうでなければ失敗するというものである。notは一階論理の論理記号の否定を表わすものではなく証明の失敗としての否定(negation as failure)[12]であるがこれをさらに以下のように概念分類する。

(1) Notの高階の否定化関数としての解釈

Notは述語Pの指示する対象をNot(P)が指示する対象に写す高階の関数と解釈できる。すなわち，Notは

$D(P)=\{(t_1,\cdots,t_n) \mid \Gamma \vdash P(t_1,\cdots,t_n),$ 名 t_i は Γ のHerbrand領域の元$\}$

に対して

$D(Not(P))=\{(t_1,\cdots,t_n) \mid \Gamma \not\vdash P(t_1,\cdots,t_n)\}$

を与える関数である。

(2) Notのメタ言語の要素としての解釈

これはNotを含む命題の証明は一階論理の論理記号としての否定を含む命題の証明のメタ言語レベルでの証明であるとみなす解釈である。この解釈の妥当性はClarkの結果にもとづく。すなわち，Clarkは，証明の失敗としての否定をもつ命題not(P)のメタ証明

が与えられると，～Pの構造的に類似した一階論理の証明が存在することを示した。例えばnot(B)の証明が次のクローズを用いて与えられたとする

$B \leftarrow B_1$, ・・・, $B \leftarrow B_n$

このとき，Bの

$B \equiv B_1 \vee B_2 \vee \cdots \vee B_n$

なる定義と等号の公理を用いて～Bの構造的にnot(B)の証明に類似の一階論理の証明が存在する。

(3) Not と多世界意味論

Not の証明の失敗としての否定においてはデータベースとの関連において次のような問題が起る。ある時点でnot(P)が証明されたとするとデータベースが更新された時点ではPの証明が可能になりnot(P)の証明に失敗する。しかしながらこの場合にはデータベースの更新によって世界が変わったと見ることができ不都合は存在しない。

また，次のようなクローズの集合が与えられているとする，

q(a)

r(x) ← not(q(x))。

このとき←r(b)とするとYesとなり，他方←r(x)と問うとNoとなってr(b)コヨxr(x)が成立しない。この場合は，r(b)が証明された時点で公理が

q(a)

r(x) ← not(q(x))

r(b)

から成る世界（知識が増えた世界）へ移ったと考えることによって問題を回避することができる。

このようにNotは多世界意味論と関係が深いが，直観主義論理あるいは強い否定を含む直観主義論理[13]における順序集合上の多世界意味論に基づく否定のKripke 流の意味

V(～A, w)＝t ⇔ 順序集合(W, ≦)において，w≦w′ なるすべてのw′∈Wに対してV(A, w′)＝f

とは異なる。

3.1.3 Prologの論理的拡張

この節では，Horn 形式と整合性を保ちつつHorn プログラムを制御する方法，プログラミング上の便宜をはかるために論理系を拡張する話題などについて記す。またそれらに対する意味論についても議論する。

A　様相（的）記号の導入

様相論理体系にはいろいろな種類の体系が知られているが代表的なT，S4，S5 といった体系においてすら，いわゆる一階論理の論理式の冠頭標準形，積和標準形に相当する標準形は存在しない。しかしながら，Horn形式をした様相論理の部分系は定義可能であり，手続き的解釈を与えることができる。ここでは様相（的）記号を含んだHorn論理式の利用法について述べる。

a　様相記号の証明論的解釈にもとづく利用法

形式的体系Sとその部分体系S′が与えられたとき体系Sの論理式Aが $\vdash_{S'} A$ であるときAは強い意味で証明可能であるとかAは必然的に成り立つといい□Aと表わされる。このときAがS′で証明可能であるとき□AはSで証明可能であると定義する。すなわち，

$$\vdash_{S'} A \quad \Leftrightarrow \quad \vdash_{S} \Box A$$

したがって，

$$\vdash_{S} \Box A \quad \Leftrightarrow \quad \vdash_{S'} A \quad \Rightarrow \quad \vdash_{S} A$$

述語論理型言語Dural[14]での様相記号S（Strong provability），Ex（Execute）はこのような背景を基に導入されている。具体的には，SはHorn節の区別を行うために用いられる。Duralでは一般の節と高速の節があり様相記号Sのついた問い合わせSqは高速の節の上だけで解が求められる。すなわち，高速の節からなる部分系での証明が逐行されるのである。またExはDural上でLisp関数を評価するために用いられる。すなわちDuralの中でLispを部分系としてもつことによりExは部分系での証明を促す働きをする。

b　様相記号のモデル論的解釈にもとづく利用法

historical データのような多世界データベース，あるいは分散型データベースが与えられている状況において，必然的なゴール，可能的なゴールの記述を許す。例えば

P（y）← ・・・，□Q（a，y），・・・

といったクローズにおいて，P(y)を達成するにはゴールの1つとして□Q（a，y）を達成するがこれは多世界データベースにおいてQ（a，y）の証明を要求する。すなわち，すべての世界においてQ（a，y）の証明が成功すること（必然であること）を要求する。この手続的意味は単項の失敗としての否定Notの意味と正反対である（Notの場合はQのあらゆる可能な証明が失敗すると成功するというものであった）。◇Rのようなゴールに対しては可能な世界におけるRの証明が成功すればそのゴールは成功し，そうでなければ失敗となる。このような□と◇の手続き的解釈において，

□P ≡ not (◇ not(P))

が成立する。

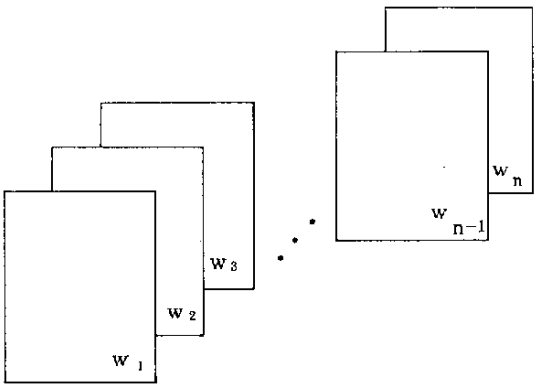


図 3.1.1. 多世界データベース

B 論理記号としての制御構造の導入

ここではクローズ間およびリテラル間に制御構造あるいは順序を明示するための論理記号の導入について述べる。

新しい論理記号を導入しHorn論理を拡張することの利点としては次のようなものが上げられる。

(1) プログラムを論理式の集合とすることに徹することができる。

(2) アルゴリズム的要素もある程度明示した方がわかり易い場合もある。

(3) 定理証明(インタプリタ)の戦略をプログラムする人が知る必要性を要求しない

(4) 制御構造をProverに任せるのではなく必要なときは陽に入れる方がよい

など。

以下では，様相制御記号として○(and then)と⋎(or else)をもつHorn論理風の論理系についてその一端を記す。

Syntax

1) h_1, h_2がHorn文のとき, $h_1 \circ h_2$, $h_1 \veebar h_2$ を論理式(プログラム)とする,

2) l_1, l_2がリテラルのとき, $l_1 \circ l_2$をリテラルとする。

Semantics

モデルを$m = \langle D, N, \leq \rangle$, ここで

D：基礎集合　N：自然数の集合　≤：N上の大小関係

付値関数をV_nとし, ○, ⋎の意味を次のように定める。

○(and then)の意味

$V_n(A \circ B) = t \iff V_n(A) = t$ and $V_{n+1}(B) = t$

$\veebar$(or else)の意味

$V_n(A \veebar B) = t \iff V_n(A) = t$ or $V_n(A) = f$ and $V_{n+1}(B) = t$

さらに，次のようにクローズの右辺において。（and then）と，（and）を区別する方が自然な場合が多い。

$R \leftarrow P_1 \circ P_2, P_3, P_4 \circ P_5$

C Dynamic Logic との融合

Dynamic Logicはdynamicなobjectとしてのプログラムの記述やその性質の証明に有効な手段を提供している。またその意味論は状態概念に基づく関係型の意味論である。したがってHorn logic との整合性が高く，Horn sentenceの評価をHorn sentenceで制御するのに用いられたり[15]また条件式，くり返しなどの制御構造をProlog に取り入れるのに都合がよい。

ここではDynamic Logic の示唆する２つの適用方法についてのみ簡単に述べる。

(1) 構造化された関係の導入

条件式，くり返しなどの制御構造をPrologに，入力，出力の関係としての述語上の正則式を用いて，入れる。例えば，

Dynamic Logic では

くり返しの関係は　$(P?;S)^*;\neg P?$

条件分岐の関係は　$(P?;S_1) \mid (\neg P?;S_2)$

と書かれる，ここで P，S，S_1，S_2 はすべて入力と出力の間の関係を表わす述語であるとする。さらに記号；は関係の合成，* | は関係の和，＊はKleene operator，？はテスト文を作り出すoperator を表わす。Horn文のように手続き的に解釈するならば，

R；Sは　Rを証明しその次にRを証明する，

R | S は　非決定的にRあるいはSを証明する，

R^* は　非決定的な回数だけRの証明をくり返す

R? は　単にRの証明を要求する

ことを表わす。

例1.　F(x, y) ← H(x, z)；G(z, y)

例2.　R ← (P?；S)[注)]；not(P)

注)；は1.3.2節で述べた。に相当する。

P ← Q ; T,

S ← U, V, W

例3. Prologのevaluable predicate 'is'を用いた次のような階乗プログラムを考える。

fact(X, Y) :− Y is 1 ; (X＞0 ? ; Y is X×Y ; X is X−1)*; X≦0 ?

このプログラムにおいて, factの右辺はALGOL風のくり返しプログラム

x：=n ; y：=1 ; while x＞0 do y：=x×y ; x：=x−1 od

にほとんど類似したプログラムになっていることがわかる。

(2) 様相概念としてのプログラムの導入

HornプログラムもDynamic logicのプログラムも一般に非決定性プログラムである。Dynamic logicでは非決定性プログラムaの性質の記述に[a]P, <a>Pといった記述を許し様相論理の中でその証明が行われる。それらの意味は次の通りである。

V([a]P, I)＝t ⇔ プログラムaを入力と出力の関係ρ(a)と解釈するとき Iρ(a)J なるすべてのJに対して V(P, J)＝t, 他方, <a>P ≡ ～[a]～P であるから,

V(<a>P, I)＝t ⇔ Iρ(a)J なるあるJに対して V(P, J)＝t。

ここでは, Horn文の右辺にこのような解釈をもつ表現[R]P, <R>Pを許しHorn logicを拡張する。このとき, それらの手続き的解釈は次のようになる。

〔R〕Pの手続き的解釈： Rのあらゆる証明に対してPの証明が成功するとき[R]Pの証明が成功する。

<R>Pの手続き的解釈： Rのある証明に対してPの証明が成功する。

このような解釈の下では関係

[R]P ＝ not(<R>not(P))

が成立する。

[R]P, <R>P 共にPの証明がRの証明によって制限されていることになる。この意味で, 3.1.3 bで述べた様相記号□, ◇の手続き的意味は単一世界では様相記号としてのプログラムの手続き的解釈の特殊な場合に相当する(Rが命題定数 true あるいは success のとき, 〔t〕＝□, <t>＝◇)。

以上(1), (2)の議論をHerbrand領域上で正当化する必要があるがここではこれ以上述べない。

D その他の拡張について

次のようなHorn論理に対する拡張は, それらの論理的裏付けには難かしい問題が潜んでは

いるが，今後極めて重要となると考えられる。しかしながらここではそれらの詳細には立ち入らず文献のreferのみにとどめる。

(1) 高階Horn論[16]

(2) Object言語とMeta言語のamalgamation[17]

(3) Concurrent Logic[18,19]

この章の最後として自然言語と論理型言語に関連して自然言語プログラミングについて触れておく。

E 自然言語プログラミング

自然言語プログラミングへアプローチする1つの方法は自然言語あるいはそのサブセットで書かれた文を意味表現としての論理式に翻訳するというものである[20]。そのさいターゲットとしてHorn論理を想定するのが最適であるが自然言語表現を直接そのような論理式に翻訳することはほとんど不可能に近い。したがって，自然言語により近い論理系，例えば内包論理[21]を翻訳の中間に設定する必要がある。このとき，自然言語文の翻訳としての内包的表現をHorn的な文に変換していくことは論理式の同値変形によって行われる。またこのプロセスはHornプログラムの合成論，検証論の方法とも関連する。

<参考文献>

1) R.A. Kawalski : Predicate Logic as Programming Lauguage, Proc. IFIP 74, North-Holland, 1974.

2) S.C.Kleene : Introduction to Metamathematics, North-Holland, 1967.

3) L.M.Pereira et al. : User's Guide to DEC system-10 Prolog, Univ. of Edinburgh, 1978.

4) E.A. Ashcroft et al. : Lucid : a Nonprocedural language with Iteration, CACM, Vol. 20, No.7, 1977

5) J.D. Monk : Mathematical Logic, Springer, 1976.

6) M.H.van Emden and R.A. Kowalski : The Semantics of Predicate Logic as a Programming Language, JACM, Vol. 23, No.4, 1976.

7) 日本情報処理開発協会編：第5世代の電子計算機に関する調査研究報告書，昭和55年3月.

8) 沢村一：算法論理，情報処理，第20巻，第8号，1979.

9) Z.Manna : Properties of Programs and First-order Predicate

Calculus, JACM, Vol. 16, No. 2, 1969.

10) C.A.R.Hoare : An Axiomatic Basis for Computer Programming, CACM, Vol. 12, No. 10, 1979.

11) V.R.Pratt : Semantical Considerations on Floyd-Hoare Logic, 17th IEEE Symp. on Found. of Comp. Sci., 1976.

12) K.L. Clark : Negation as Failure, in Logic and Data Bases, edited by H. Gallaire and J. Minker, Plenum Press, 1978.

13) 高野道夫： 強い否定を含む直観主義論理について，科学哲学11，早大出版部，1978.

14) S. Goto : DURAL : An Extended Prolog Language, 京大数理解析研究所講究録No.396, 1980.

15) 大谷木重夫： Horn Sentenceの評価とその制御，情報処理学会記号処理研究会12-1, 1980.

16) D.H.D. Warren : Higher-order Extensions to PROLOG－Are they needed ?, D.A.I.R.R. No. 154, Univ. of Edinburgh, 1981.

17) K.A.Bowen et al. : Amalgamating Language and Metalanguage in Logic Programming, School of Computer and Information Science, Syracuse University, 1981.

18) L. Monteiro : An Extension to Horn Clause Logic Allowing the Definition of Concurrent Processes, L.N. in C.S., 1980.

19) K.L. Clark et al. : A Relational Language for Parallel Programming R.R. DOC 81／16, Imperial College, 1981.

20) 渕一博： 自然言語表現と形式的表現－Montague 風翻訳の試み，第18回プログラミングポジウム，1977.

21) D.R. Dowty et al. : Introduction to Montague Semantics, D.Reidel, 1981.

3.2 失敗の原因追求の理論

3.2.1 失敗の原因追求による後戻りとは？

現在のPrologの証明探索法は top-down，serial であり，行き詰ると後戻り（back track）を開始して，選択（未だ試みていない入力節）の残っている地点迄戻り，他の道を探す。このような後戻りは単に今迄辿って来た道を逆順に辿るだけであり，失敗（単一化（unification）の原因を分析することなく行うもので，Naive Backtracking，略してNB，と呼ばれる。

NBの欠点は次の例に明白に見てとれる

S≡
- ←A(x,y),B(x),C(y)　（AND−ゴール）
- A(z,a) ←
- A(z,b) ←
- B(a) ←
- B(b) ←　　x,y,z は変数
- C(b) ←　　a,b は定数

リテラルは左から右へ，節は上から下へ access されるとする。AND−ゴールA(x,y)，B(x),C(y) に対しまずA(x,y)がcallされる。するとA(a,a)と単一化して，代入（変数と値の束縛）{z/x,a/y}が生じる。次にB(x)がcall されると。この時xの値はzであるから実際のB(x)の値はB(z)である。このゴールも入力節にB(a)← があるのですぐに成功する。代入{a/z}が生じる。

最後にC(y)がcallされるが，実際の値はyにaが代入されているのでC(a)である。

すると C(a)と単一化できる入力節がないので後戻りを開始する。NBによると直前の推論ステップ，即ち，B(z)とB(a)を単一化した地点に戻り，これをやり直し，他の入力節 B(b)を選ぶ。この選択が失敗に終わることは明らかである。何故なら，後戻りを開始する原因となったC(y)のyの値a（最初のゴールA(x,y)を解く時に生じた）に対して何ら影響を及ぼさないからである。AND−ゴール中にC(y)があり，且つyの値がaである限り，このゴールを解きうる入力節が存在しないから，直前のB(x)の解き方をいくら変えてみても，それはx，zの値に影響を及ぼすだけであり，無駄である。

このような無駄を避けるには，失敗の原因となった{a/y}の代入を生じた地点，即ち，A(x,y)とA(z,b)の単一化を行った地点に戻ってやり直しを行わなければならない。

以上の考察を一般化したものが，単一化の失敗の原因を分析し，分析結果に応じて後戻りを行う“知的後戻り（Intelligent Backtracking）”の方法である。元来このような発想は，

"依存関係による後戻り（Dependency Directed Backtracking）"と言われ，起源は７０年代半ばである。そこで"依存関係による後戻り"の研究の流れを極く簡単に眺めてみよう。

後戻りをNBのように年代順に行わず，行き詰りの原因を除去する順に行うことを明確に宣言したのは恐らく〔Ｓ＆Ｓ７６〕が最初である。〔S&S 76〕は電気回路のＱ.Ａ.であり"依存関係による後戻り"を行うプログラムを持っていたが，その後この部分がJohn DoyleによりＴＭＳ（Truth Maintenance System）〔Ｄ 78〕として独立発展させられた。ＴＭＳは信念（命題）の依存関係を木構造として記録し，信ずべき命題とそれ以外の命題の区別を教えてくれる。矛盾命題（信じてはいけない命題）Ｑが見つかった時はＱを支える祖先の命題を木構造を辿ることにより探し出す（依存関係による後戻り）。この探索順序は命題の依存関係のみに依存し，命題のシステムへの入力順序に依存するのではない。後戻りの結果矛盾Ｑを支えている仮定命題（default assamption）が発見されるが，仮定命題に対する信念を翻がえすことによりＱを信ずる根拠が無効になるので，矛盾Ｑは信じられない状態になる。

ここで後戻りについて２，３整理を行う。ここで考える後戻りは必ずしも以前の状態に戻ることを意味せず，行き詰りの打破の為のより良いalternativesを求める方法を意味するものとする。行き詰りを解決するalternativesをもし存在すれば見逃すことのない後戻りをsafeと呼び，行き詰りから有用な知識を吸収し同じ行き詰りを２度と繰り返さないように努める後戻りをwiseということにする。

このsafeとwiseの観点からTMSの後戻り見てみよう。TMSは命題に対する信念の間に矛盾がないように，ある命題を信じ他の命題を信じないといった"調整"を行う。

調整の可能性は通常複数個あるが，ＴＭＳはすべての可能な調整を行うわけでなく，従ってＴＭＳの後戻りはsafeでない（と推察される）。しかしＴＭＳが矛盾命題を見つけた場合その矛盾命題を再度信じることがないような手立をして後戻りを行うのでwiseといえる。

〔Ｓ&Ｓ 76〕と同じ頃Cox等は結合グラフ（connection graph）を使ったグラフ的導出証明システムを作製した〔Ｃ&Ｐ 76,81〕.

このシステムはまず入力節集合の結合グラフをたよりに証明プランを表わす証明グラフを作る。次にこの証明グラフの辺に付いている単一化子（unifier）をすべて同時に満たす単一化合成（unifying composition）を作ろうとする。もし単一化合成ができれば証明は終わるが，単一化合成が失敗すると"後戻り"を行う。この場合の後戻りはもとの証明プラン（グラフ）を修正することを意味しているが，目的はその修正を最小限にとどめることにある。

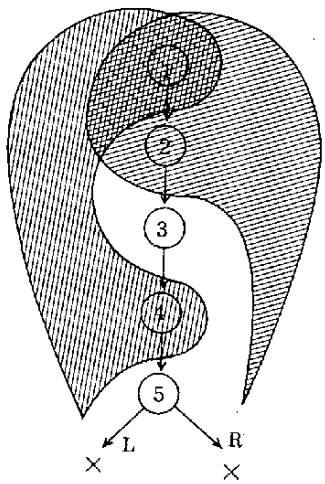
証明プランの修正部分を確定する際，そのプランが作られた経緯を年代順に辿るわけではない。〔Ｓ&Ｐ 81〕ではこの証明プランの探索戦略が述べられていないので後戻りがsafeであると

もwiseであるとも判断できない。しかし行き詰り（単一化合成の失敗）の原因をそれに関係する所のみに絞って探すという意味で"依存関係に基づく"後戻りであり，NBの単純な後戻りに比べ失敗に無関係な所を無視するという点で明らかに優れている。

〔B78,80〕は，論理プログラムの枠組みの中で"知的後戻り"を追求している。扱う節はhorn節で，実行はtop-down，serialであるが，リテラル（subgoal）の選択が自由である点がPrologと異なる。問題は導出ステップ D_0，D_1，・・・，D_n のある点Diで失敗(単一化の失敗である）が起きた時，どの推論ステップのどの選択を取るか定めることである。推論は反証木（refutation tree）を上から下へ行って行く。ある推論ステップDiを有効ならしめている以前の推論ステップをDiのinput setと呼ぶ。従ってDiのinput set以外の推論ステップをやり直しても無意味，即ちDiの失敗の原因を除いたことにならないのは明らかである。

ある推論ステップDiの失敗の各々に付きその失敗を導びいたinput setが付随している。推論ステップDi，即ちあるゴールを解決する試みがすべて失敗に終った時．戻るべき地点（やり直すべき推論）はそれらの失敗のinput setの集合和＝supectsの中で任意の推論ステップに対しその前提になっていないような推論ステップである。これをculprit（容疑者）と呼ぶ。あるculpritをやり直すことによりそのculpritを前提の推論ステップとして持つ推論系列はすべて無効になる。従ってあるcall（入力節の呼び出し）が以前に失敗していたとしてもそのcallが依存しているculpritをやり直せば，失敗に終わった入力節も再度適用可能になる。（その入力節を使っても失敗に終わるとは限らなくなる）

さて，今迄後戻りの際戻るべき地点を簡単に述べたが，注意しなくてはならないのは"連続した失敗"の現象である。



今推論ステップが①→②→③→④→⑤と進み，第5ステップで　の選択肢が失敗し，その時の input setを{①，④}，Rの選択も失敗してその時のinput setを{①，②}とする。第5ステップのすべて選択肢がすべて失敗したので後戻りをする。第5ステップの失敗のsuspects は{①，④} U {①，②}={①，②，④}であるので，やり直すべきculprit は④である。さてそこで④も失敗したとする。その時のculpritを①とすると①に戻りそうになるが，これは実は⑤で失敗した時のinput setの内②を無視し，②を飛び越すこ

とになる。⑤の失敗は｛①，②，④｝に起因するのであるから，②をやり直すことによりステップ⑤が成功する可能性があるのにこれを無視することになる。従って，④の失敗で①に戻ることは間違いで，②に戻るべきであるということになる。

このように失敗が続く場合，その後戻りには以前の失敗迄考慮する必要があるのである。"連続した失敗"の現象を考慮しない後戻りは解を見逃す可能性があるので safe でない。

〔B 80〕ではこの"連続した失敗"に考慮を払っているが，〔P 80〕は全く考えに入れていない。

さて〔B 80〕の後戻りは，あるゴールを解決するすべての試みが失敗に終った時，推論ステップ間の依存関係を基に今述べた"連続した失敗"に注意を払いつつ culprit に戻る。

従って失敗の原因に無関係な後戻りをしないという点でwiseである。とは言うものの，このような後戻りが，解決の可能性のある道を捨てていないかどうか，即ち"探索の完全性"を壊していないかどうか疑問が残る。たとえそのような可能性がないにしてもそのことは証明が必要である。

〔P 80〕は〔B 80〕に刺激を受け，"知的後戻り"の implementation を論じたものである。

〔P 80〕ではゴールGの失敗を単一化に於ける定項間の mismatch に求めている。occur check は全く考えていない。失敗に関係した定項記号をG迄運んで来た推論ステップが失敗の input set である。その為各定項記号（a，f（x，y），等々）が変数に束縛された地点で各定項記号にそのことを示す tag を付ける。例えばP（x）とP（f（y，a））を単一化する時x，y に tagPを付ける。実際に単一化の時，aとbの間にmismatch が生じたら，a についている tag と b のそれとを比較して，最も最近（most recent）に付けられた tag に対応する推論ステップに戻る。

〔P 80〕は知的後戻りの具体的 implementation 迄踏み込んだものであるが，多くの点で不安を残している。1つは〔B 80〕が指摘した"連続した失敗"に考慮を払っていない，少なくとも言及していない点である。もう1つは tag 付けにより失敗の原因となる推論ステップの同定を行うが，この tag 付けが正しいという証明がない。即ち，例えば tag付けの間違いで戻り過ぎを起こし，成功の可能性のある選択肢を見逃してしまう危険はないのだろうか？〔P 80〕の議論は（〔B 80〕もそうだが）idea の説明にとどまり，証明が欠けているのでハッキリとした議論ができない。

〔L 80〕はCoxとPereira の仕事を引用し，Bruynoghe を引用していないが，議論の枠組みはBruynogheと全く同じである。失敗が生じた時（もとの地点に戻ることを含めて）

どのような選択肢を選ぶべきかをCox流の考えとPereira式のbirding記法を使って論じているが，非常に大雑把であり，例えば失敗の定義，探索法の規定もなく，後戻りが safe かも議論していない。単に1つの見返り図を考えてみたというに止どまると言うべきであろう。

以上で"知的後戻り"に関する5つの論文を概観したが，共通していえることは，

(1) 失敗の原因を何らかの依存関係を手かりに追求し

(2) その原因を取り除くことにより同じ失敗 繰り返すことを防ごうとする

姿勢である。従って失敗すると1つ前の地点に戻って他の選択肢を採すNBに比べwiseであることは間違いない。しかし単に"wise"だけを追求しているのでは1つのheuristics にとどまる。問題とすべきは

(3) 後戻りにより捨てられた選択肢が実際に解を与える可能性がない

(4) 解へ導く可能性のある選択を必ず試みるような探索法を取っている

ということに関する明確な議論がない点である。従って彼等の方法にもとづく後戻りが safeか否か断定できない。safeでない後戻りは解を見逃す可能性があり，我々は是非ともこのようなことを避けなければならない。

さらにまた〔B 80〕〔P 80〕〔L 80〕の議論はhorn 節に限定されているが将来Prologを拡張して（OL反証法）完全な証明システムにした時の知的後戻りの方法も研究する必要がある。

次からはOL反証法における知的後戻りの理論をもとに，Prolog における知的後戻りの方法を議論する。ここで提案する方法はwiseかつSafeであり，現在のPrologの implementationで良く使われているstructure sharing に良く適合する。

3.2.2 Prologにおける"知的後戻り"基本編

任意の順序節（ordered clause）の集合Sに対する完全な（complete）反証手続き（refutation procedure）であるOL（ordered linear）反証法のIB（intelligent backtracking） として提案された方法をProlog 向けに簡単化してここで述べる。

Prologの計算過程＝証明過程の途中状態を中心節（center clause／AND－ゴール）の型（template）＋代入（substitation／binding）に分けて表れそう。

最初のゴールを ←P(x) とすると，これは ＜P(x)，ε＞ と表わせる。P(x)が型，ε（null の代入）が代入である。今このゴールに入力節（input clause）P(a)←Q(b)，R(y)を適用すると，次のAND－ゴールは＜Q(b)R(y)，{a／x}＞となる。Q(b)R(y)が現在の中心節の型，{a／x}が現在迄単一化（unitication）により得られた代入である。

この例からも分かるように，Prologのある時点のゴールの状態は常に型＋代入の形に表わ

せるのである。

以下L,M,N・・・はリテラル（またはその型）を，α, β, τ…はリテラル型の列を，θ, λ, μ…は代入を，$\theta \circ \nu$の合成を，Eθは表現Eに対する代入θの適用を示している。

次に一般的に証明中のAND－ゴールの変化を追う。

公証明を開始して$\nu-1$回導出を実行した時のAND－ゴール$<L\alpha, \theta_1 \circ \cdots \circ \theta_{i-1}>$としよう。実際のAND－ゴールは$L\alpha\theta_1 \circ \cdots \circ \theta_{i-1}$であり，次の解くべきゴールは$L\theta_1 \circ \cdots \circ \theta_{i-1}$である。i回目の導出にLと単一化可能なhead M を持つ入力節M←βを選んだとする。LとMのmgu（most general unifier）をθiとおくと，i回目の導出が終った時のAND－ゴールは$<\beta\alpha, \theta_1 \circ \cdots \circ \theta_{i-1} \circ \theta i>$となる。但し M←$\beta$は変数の名前がえ（renaming）をしておくものとする。$\theta_1, \cdots, \theta_i$を代入因子と呼ぶ。

最後にAND－ゴールが空になり状態が<□, θ>（□は空節）となれば，証明が成功で,その時の答（answer substitution）がθである。

以上のような証明過程は下図のような証明の探索木[注)]の探索過程に対応する。

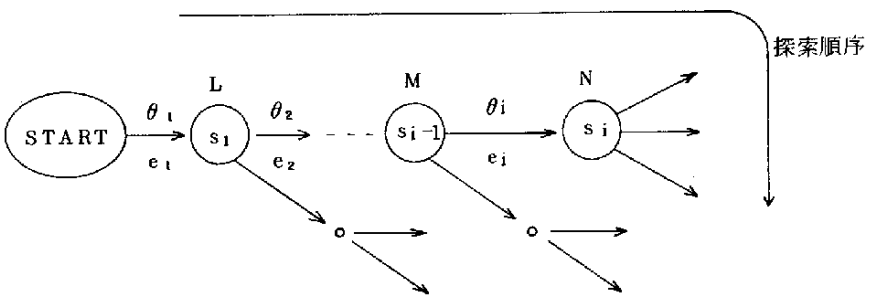


図 3.2.1. 探索木

各節点s_iはAND－ゴール＝<型，代入>を表わしている。各節点はその時resolveされるゴール[注)]（AND－ゴール中の最左端のゴール）のゴール型でラベル付けしてあり，節点から出ている辺e_iは導出の予定（resolveすべきリテラルpairの指定）を表わしている。図3.2.1の状態s_{i-1}は辺e_iで指示される導出を行うとmguθiが得られ次の状態s_iに移り，そこでは可能な導出が3通りあることを表わしている。

上に示したような探索空間（探索木）は最初にゴールと入力節の集合（Prologプログラム）

注）要するに結合グラフ（connection graph）の木への展開である。従って，探索空間図は，AND－ゴールの型によってのみ決定される。

注）ゴールとリテラルを区別しないで使っている。

Sが与えられた時点で完全に定まる。探索は探索空間を左から右，上から下へ辺を辿ることにより行われ（勿論辺を渡る時に導出を行う。導出が失敗した時には辺に×印を付けることにする），ゴールが空である状態に達っすれば証明は成功裡に終れる。その節点を空節（null clause）の印"□"でラベル付けをする。探索の始点から□でラベル付けされた節点迄の道を解または解の道と呼ぶ。注）

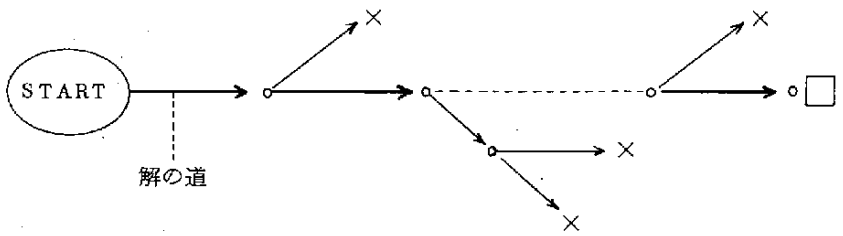


図 3. 2. 2.　解の道

ある節点sを通る解がないことが分かった時後戻りをして他の道をtryしなければならない。その際sの1つ前（sの左隣り，sの親節点）に戻る方法がNBである。失敗の原因を分析し，結果に応じて後戻り先きを決めるやり方をwiseな後戻り法と呼び，後戻りの開始点sと後戻り先きの節点s'の間の節点から出ている道が必ず失敗に終わる保証がある方法をsafeな後戻り法と呼ぶ。

NBはsafeであるがwiseとは限らない。

われわれはNBに代わるsafeかつwiseな後戻り法を追求する。その為用語を幾つか定義する。

（Def.1）導入点：ある節点sでresolveされるゴール型（リテラル型）Lについてそれが初めてAND－ゴール中に現われた節点s'をLの導入点という。s'をintro(s,L)またはintro(L)と表わす。

例　状態sに於けるAND－ゴールを＜P(x)Q(y)，{a／x，b／y}＞とする。実際のANDゴールはP(a)になる。ここで，入力節P(z)←R(z)S(z)によりP(a)を展開すると，次の状態s'は　＜R(z)S(z)Q(y)，{a／x，b／y}∘{a／z}＞になる。R(z)，S(z)の導入点はs'である。

導入点に関し次の事実が成立する。

F1）　Lの導入点sを通る解の道の上には必ずLでラベル付けされた節点がある。これは，

注）われわれの探索法は，解があればそれを見逃すことがない，即ち完全な探索である。

一旦ＡＮＤ－ゴールに導入されたリテラル型は，そのＡＮＤ－ゴールが達成される為に resolve される必要があることから明らか。

F2） Ｌの導入点ｓを祖先（左側に辿って行くと達する節点）に持つＬでラベル付けされた節点ｔ，ｔ′ ‥ を頂点とする部分探索木はすべて同一である。これは，探索木はＡＮＤ－ゴールの型のみで決まることから明らか（ modulo renaming ）状態節点ｓに於けるＡＮＤ－ゴールを$<\beta L\alpha, \theta_1 \circ \cdots \circ \theta_i>$とするとｔ，ｔ′… はすべて$<L\alpha, \theta_1 \circ \cdots \circ \theta_i \circ \nu>$（$\nu$はｔ，ｔ′,… による）という形をしている。

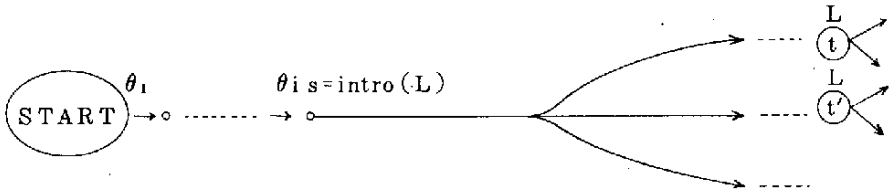
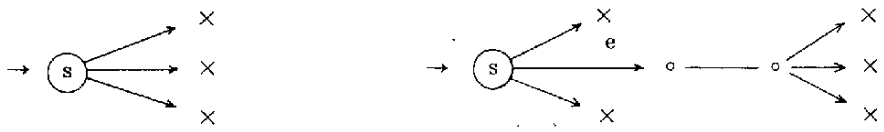


図3.2.3　リテラル型Ｌの導入点と，Ｌでラベル付けされた節点

（Def.2）辺の失敗：節点ｓから出ている辺ｅの指示する導出が成功しなかった時，辺ｅは直接失敗したという。ｅの指示する導出が成功して節点ｓ′に移ったが，その後の探索によりｓ′を通る解がないことが解った時ｅは間接的に失敗したという。

（Def.3）節点の失敗：節点ｓから出ているすべての辺が直接失敗したといい，ｓから出発するある辺が間接的に失敗し，かつ他の辺が間接または直接に失敗した時，ｓは間接的に失敗したという。



ｅは間接的に失敗している。

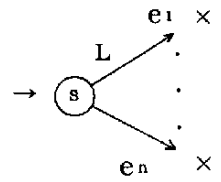
（節点ｓの直接の失敗）　　（節点ｓの間接的失敗）

図3.2.4　節点の失敗

例　今状態ｓに於けるＡＮＤ－ゴールが$<P(x), \{a/x\}>$で入力節が｛ P(b)←，P(c)← ｝とする。実際のゴールの値はP(a)であり，どの入力節の headとも単一化できないからｓは直接的に失敗する。今度は状態ｓは同じで，入力節が｛ P(y)←Q(y)，Q(b)←，Q(c)← ｝であるとしよう。するとｓの次の状態ｓ′は$<Q(y), \{a/x\} \circ \{x/y\}>$になる。ｓ′は直接失敗する。従ってｓは間接的に失敗することになる。

さて，プログラム（節集合）Ｓと証明すべきゴールが与えられて，証明が始まったとしよう。

最初に後戻りが必要になるとすればそれは，あるゴールPに単一化するheadがない時，即ち，ある節点の直接の失敗が起きた時である。そこでわれわれは，Prologにおける知的後戻りの考察を，節点の直接の失敗の場合から始める。



今状態sから出ている辺$e_1 \cdots$ ， e_n が指示する単一化がすべて失敗したとする。ei を実行する際，UA（Unification Algorithm —単一化アルゴリズム）が参加した代入因子 $\{\lambda_1, \cdots, \lambda_k\}$ $(k \geq 0)$ を ei の（直接の）失敗の agent と呼び agent（e_i）と記す。

（Def. 4）辺eの直接の失敗のagent ：

agent(e) ＝eが指示する単一化の際，UAが参照した代入因子の全体

さて辺 ei の失敗を繰り返さないためにはどの地点に後戻りすべきであろうか？

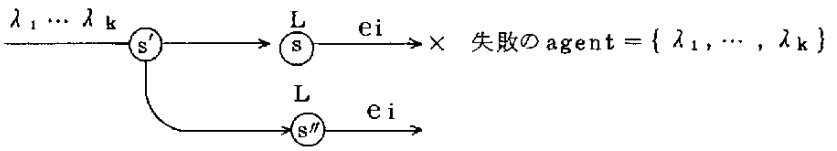


図3.2.5 同じ辺の失敗

次のような場合に，われわれは e_iの失敗が繰り返されることが断言できる。

今sから後戻りをしてs'に戻ったとする。もしs'において

① AND－ゴールの型に，単一化に失敗したリテラル型Lを含み

② 辺e_iの失敗のagent $\lambda_1, \cdots, \lambda_k$ がs'における代入の中に因子として存在する

ならば，s'から出発して辺 e_iを通る時同じ失敗が起こる。（図3.2.5参照）何故なら，もし後戻りした地点s'が上の条件をみたせば，その時の状態は $<\beta L \alpha, \theta_1 \circ \cdots \circ \theta_m>$ と書ける。

$\{\lambda_1, \cdots, \lambda_k\} \leqq \{\theta_1 \circ \cdots \circ \theta_m\}$ である。

s'からs迄辿り終えた時の状態は$<L\alpha, \theta_1 \circ \cdots \circ \theta_m \circ \theta_{m+1} \cdots \circ \theta_n>$ であった。そして辺 e_iの失敗は，辺 e_iの指示するLとM（Mはある入力節のhead）の単一化が行えなかったことを意味している。即ち，$L\theta_1 \circ \cdots \circ \theta_m \circ \theta_{m+1} \circ \cdots \circ \theta_n$とMが単一化不能であることを示している。

その際UAが参照した代入因子は$\{\lambda_1, \cdots, \lambda_k\}$は$\{\theta_1 \cdots \theta_m\}$ に含まれていたのだから，$L\theta_1 \circ \cdots \circ \theta_m$ とMが単一化不能であったことになる。さてs'に後戻りをして別の道を辿り，AND－ゴールの中でLが最左端の位置にあるような状態s''になったとする。（図3.2.5参照）s'' における状態は $<L\alpha, \theta_1 \circ \cdots \circ \theta_m \circ \theta_{m+1}' \circ \cdots \circ \theta_\ell'>$ と書ける。そこでは再度辺ei

の指示に従って，

$L\theta_1 \circ \cdots \circ \theta_m \circ \theta_{m+1}' \circ \cdots \circ \theta_\ell'$ とMの単一化を試みるのであるが，過去に節点sで，

$L\theta_1 \circ \cdots \circ \theta_m$ とMが単一化不能であったことより，$L\theta_1 \circ \cdots \circ \theta_m$ をより instantiate した $L\theta_1 \circ \cdots \circ \theta_m \circ \theta_{m+1}' \circ \cdots \circ \theta_\ell'$ がMと単一化することは不可能である。即ち辺 e_i で同じ失敗が起きるわけである。

例：最初の状態 s_1 を $<P(x)Q(y)R(x,c), \varepsilon>$，節集合を $S = \{ P(a)\leftarrow,\ P(b)\leftarrow,\ Q(c)\leftarrow,\ Q(d)\leftarrow,\ R(b,c)\leftarrow \}$ とし，Sの節は左から右へ順々に try されるとする。2回導出を行うと，AND－ゴールは $s_3 = <R(x,c), \{a/x\} \circ \{c/y\}>$ という状態になる。そこで，$R(x,c)\{a/x\} \circ \{c/y\}$ とR(b,c)の単一化を行う。するとUAは代入因子 $\{a/x\}$ を参照して，R(a,c)とR(b,c)の単一化を試みるが失敗する。（図3.2.6参照）

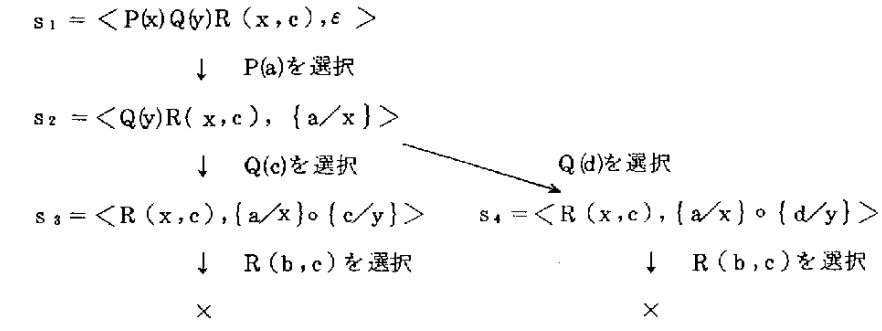


図3.2.6　同一の辺の失敗例

s_2 においては，s_3 において解こうとしたゴール型R(x,c)，及び，s_3 でUAが参照した代入因子 $\{a/x\}$ が存在している。従って s_2 に戻って他の道を選択しても（図3.2.6の右側の分岐）再度同一の失敗を繰り返す。

ゴール型Lでラベル付された節点sから出ている辺 eiが直接失敗した時，その原因はLの存在と，失敗のagent，$agent(e_i) = \{ \lambda_1 \cdots \lambda_k \}$ の存在にある。Lと $\{ \lambda_1 \cdots \lambda_k \}$ が始めてAND－ゴール中に揃った時点s'で，s'の子孫節点sから出ている辺 e_j の失敗は運命付けられていたのである。その意味で，s'を辺 e_j（直接の）失敗の運命点 $det(e_i)$ と記す。

辺 e_i の失敗を繰り返さないためには $det(e_i)$ の親節点，あるいはそれ以前に戻らなければならない。改めて辺 e_i の直接の失敗の運命点 $det(e_i)$ を定義しよう。

（Def.4）　代入因子の集合 $\{ \lambda_1 \cdots \lambda_k \}$（$k \geq 0$）を含む最小prefix：AND－ゴール中の代入が証明を開始して以来始めて $\lambda_1 \cdots \lambda_k$ を含んだ時の節点sの代入をいう。sを $\{ \lambda_1 \cdots \lambda_k \}$ を含む最小prefix の出現点という。

$k > 0$ ならsは $<\bullet, \theta_1 \circ \cdots \circ \lambda_k>$ の形，$k = 0$ ならsは証明の開始節点である。

(Def.5)　Lでラベル付けされた節点sから出ている辺eの直接の失敗の運命点det(e)：eの直接の失敗のagent，agent(e)を含む最小prefixの出現点とゴール型Lの導入点intro(L)の内，most recent（探索木の頂点から遠い方）の節点

辺e_iの直接の失敗の運命点det(ei)は<…L…，$\nu \circ \nu'$>但し，Lは辺e_iが単一化しようとしたゴール型，νはagent(ei)を含む最小prefixの形をしている。

話をもとに戻そう。直接に失敗した，ゴール型Lでラベル付けされた節点sからは$e_1 \cdots e_n$のnコの辺が出ていた。各eiにはその失敗のagent，agent(ei)と，失敗の運命点det(ei)が定まっている。{ $det(e_1) \cdots det(e_n)$} の内most recentな節点をs'としよう。sからs'に戻り証明を続行するとそれは必ず失敗する。何故なら，s'から証明を続けて行く内に（もし途中で失敗がなければ）Lでラベル付けされた節点に達っし，そこからは辺ei … enが出ている（導入点に関する事実F1，F2参照）が，それらの指示する単一化は，s'がdet(e1)と等しいか，もしくはdet(ei)の子孫の節点であることから，すべて失敗する運命にあるからである。それ故

(Def.6)　節点sの直接の失敗の運命点det(s)：sから出ている辺を$e_1 \cdots e_n$とする。

det(s)＝{ $det(e_1) \cdots det(e_n)$} の内most recentな節点

(Def.7)　節点sの直接の失敗，及びdet(s)の失敗のagent：sから出ている辺を$e_1 \cdots e_n$とすると，

agent(s)＝agent (det(s))

＝agent $(e_1) \cup \cdots \cup$ agent(e_n)

sで直接失敗した時，det(s)，あるいはそれより後（det(s)の子孫の節点）に戻ったのでは必ず失敗する。det(s)より1つ前に戻れば，sから出ているある辺の直接の失敗（単一化の失敗）の原因の少なくとも1つを取り除いたことになるので，det(s)の1つ前の節点に戻り，証明を再開すれば，成功する可能性がある。従って節点s'が直接失敗した時戻るべき点btk(s)は

(Def.8)　失敗した節点sの後戻り点：btk(s)＝det(s)の1つ前の節点（親節点）である。このような後戻りがsafeかつwiseであるのは明らか。

直接失敗した節点s，agent(s)，det(s)に関し次の事実が成立する。

F3)　失敗した節点をs＝<α，ν>とする，agent(s)[注)]を含む最小prefixをν'とす

注）agent(s)は直接または間接に失敗した節点，及びその失敗の運命点である節点sに対して定義される。今迄の所，直接に失敗した節点とその運命点のみについて定義されているが，後で間接的に失敗した節点及びその運命点についても失敗のagentが定義される。

る。もしある節点s'が$<\alpha, \nu' \circ \nu''>$と表わされるならば，s'を通る解の道はない。

F4) 失敗した節点sの運命点をdet(s)，btk(s)をdet(s)の1つ前のsの後戻り点とする。btk(s)からdet(s)に移る際作られた代入因子λが，det(s)の失敗のagent（＝sの失敗のagent）に入っているならば，λはそのagentの中でmost recentな代入因子であり，det(s)の代入νはν'をある代入として$\nu = \nu' \circ \lambda$と書ける。

次に節点sが直接失敗した場合のagent(s),det(s),btk(s)の関係を図示する。

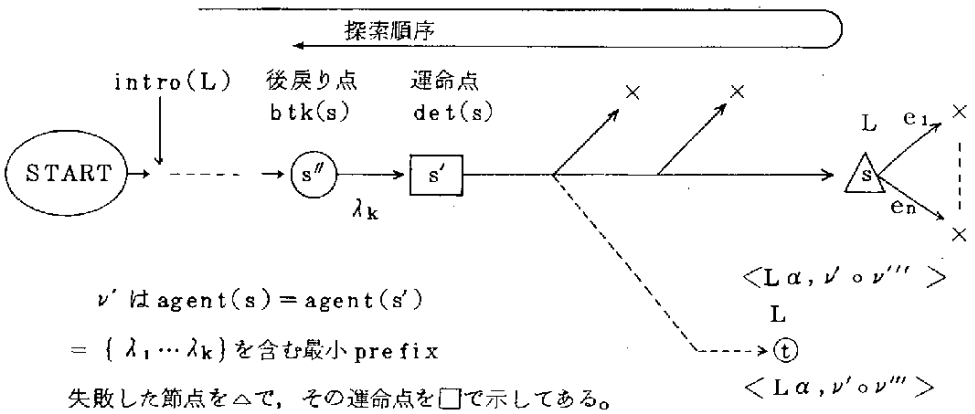


図3.2.7　節点sの直接の失敗とdet(s),btk(s)の関係

次に節点sが間接的に失敗した場合の後戻りを考えよう。

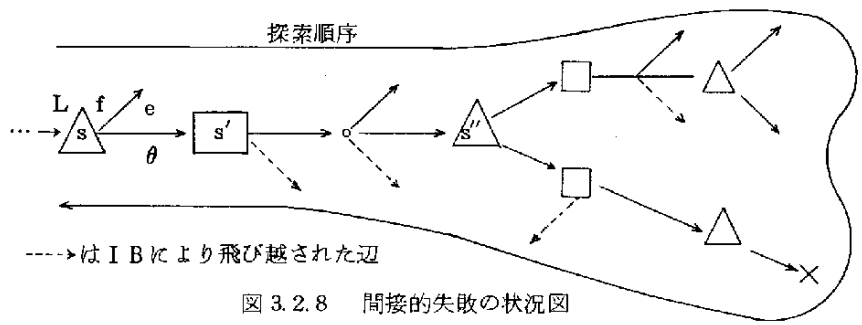


図3.2.8　間接的失敗の状況図

上の図でs''は間接的に失敗し，s'はその失敗の運命点，sはs''の失敗の後戻り先きである。s'，s''については失敗のagentが既に定められ，s'，s'，s''に関し先のF3，F4が成立しているものとする。

われわれは上の図でまず間接的に失敗している辺eの失敗のagentと運命点を定める。

(Def.9)　Lでラベル付けされた節点sから出ている間接的に失敗した辺eの失敗のagent，

agent(e)，及びその失敗の運命点det(e)：（図3.2.8参照のこと）

eの行き先である，ある失敗の運命点をs′，eを実行する時生み出された代入因子をθとする。s′に対してはagent(s′)が既に定義されている。

(i) agent(s′)＝ϕ　この時はs′のAND－ゴール型のみが失敗の原因である。

agent(e) ＝ ϕ

det(e) ＝ intro(L)

(ii) agent(s′)≒ϕかつagent(s′)∋θ

eの指示する単一化が生み出した代入因子θはs′の失敗に寄与していない。

agent(e)＝agent(s′)

det(e)＝agent(s′)を含む最小prefixの出現点とintro(L)の内most recentな節点。

(iii) agent(s′)≒ϕかつagent(s′)∋θ

agent(s′)＝{$\lambda_1 \cdots \lambda_k$}とおく。$\lambda_1 \cdots \lambda_k$はこの順に作られ，$\lambda_k$が最も最近に作られた代入因子とする。F4)より$\lambda_k = \theta$である。$\theta$を作る際UAが参照した代入因子を{$\mu_1, \cdots, \mu_\ell$}{$\ell \geq 0$}とおく。

agent(e)＝{$\lambda_1, \cdots, \lambda_{k-1}$}U{$\mu_1, \cdots, \mu_\ell$}

det(e)＝agent(e)を含む最小prefixの出現点とintro(L)の内でmost recentな方

間接的に失敗した辺eに対し，ここで与えたその失敗の運命点det(e)，またはそれより後に戻ったのでは，再度eの失敗を繰り返すことは（F3），（F4）より確かめられる。（証明略）

間接的に失敗した節点sから出ている，直接失敗した辺（図3.2.8のF）の失敗のagent，運命点については，Def.1及びDef.5で与えてある。

以上の準備のもとに間接的に失敗した節点sの失敗のagent，その失敗の運命点を定める。

（Det.10）間接的に失敗した節点sの失敗のagent，運命点：sから出ている辺を$e_1 \cdots e_n$とする。それらはすべて直接または間接に失敗している。

agent(s)＝agent(e_1)U ･･･ Uagent(e_n)

det(s)＝det(e_1),･･･det(e_n)の内でmost recentな節点

また，det(s)も間接的に失敗しているわけであるが，その失敗のagentは，

agent(det(s))＝agent(s)

間接的に失敗した節点sに対し，その失敗の運命点det(s)を定義した，det(s)より後に戻ったのでは失敗に終わることは，det(s)について（F3）が成立している（証明略）ことより明

らか。また少なくともdet(s)の1つ前に戻って証明をやり直せば，sから出ているあの辺の失敗の原因の1つを取り除いたことになるので，証明が成功する可能性がある。

従って，間接的に失敗した節点sの後戻り先btk(s)はdet(s)の1つ前である。

Def.9 間接的に失敗した節点sの後戻り先btk(s)はsの失敗の運命点det(s)の1つ前の節点である。（Def.8）

この場合，btk(s)とdet(s)について（F4)が成立していることも容易に確かめられる。

以上ですべての失敗の場合について，"知的後戻り先き"が定まった。このような後戻りがsafeかつwiseであることは，"知的後戻り"の回数に関する帰納法により常に（F3),,(F4)が成立していることが証明されることより明らかである。

3.2.3 Prologにおける"知的後戻り"応用編

2.2節で"知的後戻り"の基本アルゴリズムを示した。ここでは"知的後戻り"を実際に行う場合の注意事項について述べる。

A．探索木が直線の場合

探索木が直線→ • →○→ …○ → Ⓢ であり，sが失敗した場合，NB(注)は単に分岐のある地点迄後戻りをする。しかし"知的後戻り"の場合，そのようなことをすると，失敗の原因が分からなくなるので，基本的には一歩一歩"知的後戻り"をする必要がある。しかし，うまく考えるとこのようなstep by stepの"知的後戻り"を高速化できる。しかし紙面の都合上，詳細は省略する。

B．偽の失敗の場合

(1) cut symbol「!」が入った時の後戻り。

!が入ると，証明の道を切り捨ててしまうので，失敗の原因が正確につかめなくなる。

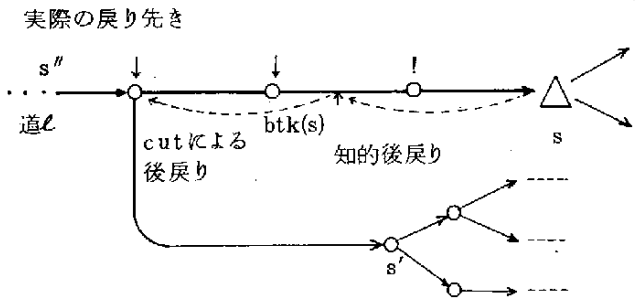


図3.2.9 cutがある場合の後戻り

実際，図3.2.9のようにsで失敗が起り，後戻り先きbtk(s)が，cut(!)の導入点と！

注）Navie Backtracking

の導出点の間（両端を含む）にある時，cutの働きにより，btk(s) より更に以前に戻ってしまう。この時，cutが捨てた道は成功する可能性もあり，実際の戻り先きがたとえ失敗に終わっても，それは本当の失敗ではなく，失敗の原因（失敗のagent）も分からない。

われわれは安全な後戻りを求めているので，cutがある場合の後戻りを次のように定める。

「cutが導出された後の知的後戻りによりcutによる強制的後戻りが起った時，その実際の後戻り先をs″とすると，証明の開始点からs″に至る道の上での後戻りはNBを採用する。（図3.2.9道ℓ）」

従って図3.2.9のs′を頂点とする探索木内では今迄通りの"知的後戻り"を行う。

(2) Prologでは1つの解を見つけた後，他の解を見つけるために強制的に失敗させることがある。このような場合も，失敗の原因が単一化に起因するわけではないので，われわれの"知的後戻り"が適用できなくなる。

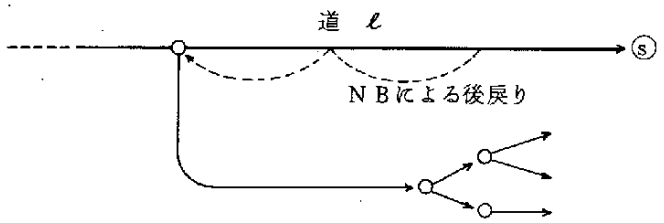


図3.2.10　他の解を求めるための強制的失敗

この場合cutと同様，

「節点sでAND一ゴールが空になり，1つの解が見つかったとする。そこで強制的に失敗を起こした時，証明の開始節点からsに至る道（図3.2.10のℓ）の上ではNBを行う」

従って図3.2.10の節点s′を頂点とする探索木内では"知的後戻り"ができる。

以上偽の失敗が節点で起きた場合，証明の開始節点とsに至る道の上では"知的後戻り"が禁止され，NBが採用される。これをNBの指定範囲と呼ぶ。

C．同一の失敗の防止

今迄述べた"知的後戻り"は失敗の複数の原因を1つ取り除くに過ぎないので，再度同じ失敗をする可能性がある。（図3.2.11参照）

この図では節点sで辺e_1，e_2が失敗している，辺e_1の失敗のagentを$\{\theta_1\}$，辺e_2のそれを$\{\theta_2\}$とし，θ_2の方が後で生成されたとすると，btk(s)はθ_1が生成された時点より後

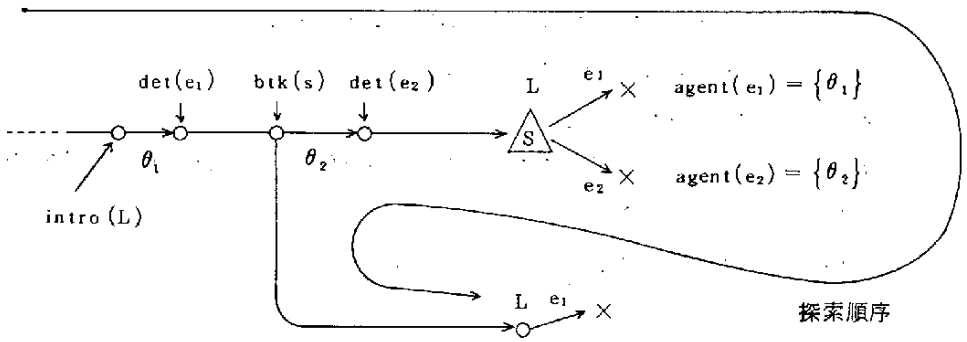


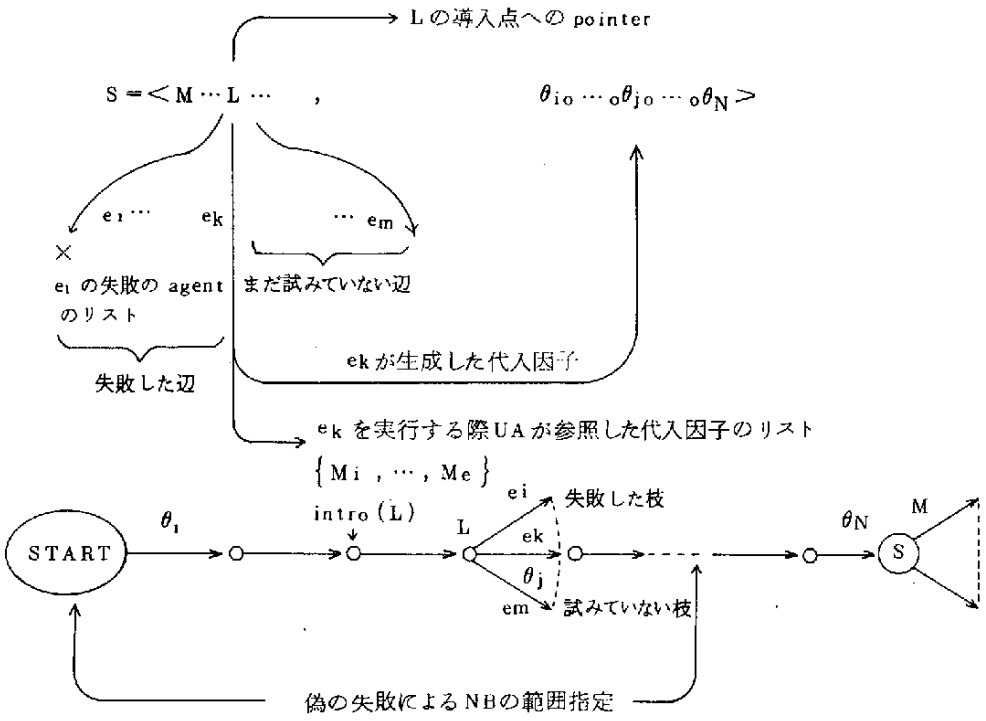
図 3.2.11　同一の辺の失敗

になるので，btk(s) から再度証明を開始し，辺 e_1 を渡る時失敗が起こる。

このようなことを防ぐには，辺 e が失敗した時，その失敗の運命点 det(e) より前に戻らない限り e を実行しないようにすればよい。

D. 実　働　化

インプリメンテーションの見取り図を示す。



"知的後戻り"を行うため記録しておく必要のあるものは,

(1) ゴール型Lの導入点(導入ステップ)

(2) 失敗した辺eの失敗のagentのリスト

(3) 成功した辺eに対し,辺eの生成した代入因子θ(へのpointer)と,θを作る際UAが参照した代入因子のリスト

である。勿論代入因子を指定するのに,それを作り出した推論ステップの番号を使えば簡単になる。

E. 簡単化

"知的後戻り"の簡単化を考える。"知的後戻り"を節点が直接失敗した時だけに限れば,記録しておく必要のあるものは,辺の直接の失敗のagentの内most recentなものだけである。また!や強制的失敗によるNBの範囲指定も不必要になる。

F. 一般化

Prologのsubgoalの選択を自由にした時の"知的後戻り"を考える。AND一ゴールはリテラルのbagと,代入により表わされ,探索木は,resolveすべきリテラルpairを指定することにより定まる。

後戻りはAND一ゴールのあるリテラル(ゴール)が解ける可能性が完全に失われた時に起こる。このような場合の"知的後戻り"の議論は,今までの議論を全く平行にできるので省略する。

3.2.4 結論

われわれは,Prologにおけるsafeかつwiseな"知的後戻り"を,OL反証法またはさらにリテラルの選択を自由にしたOL反証法用に提案された"知的後戻り"の方法をもとに議論してきた。

大事な点は,Prologによる証明の途中状態を<AND一ゴールの型,代入>という形に表わすことにより,精密な議論が可能になり,かつわれわれの提案する"知的後戻り"がsafeであることが示せたことである。

この枠組みにより,探索空間,辺,節点の失敗,そのagent,失敗の運命点等々が定義され,最後に後戻りすべき点が決められた。

また,この枠組みにより,従来触れられることのなかった,cut,強制的失敗の場合も適切な後戻り地点を設定することができた。('not'がある場合の議論は紙面の都合上省略)

勿論われわれの後戻り法は失敗の原因を1つ取り除くだけで,より成功の見込みの大きい地点に戻るわけではない。その意味で,最低限の"知的後戻り"である。よりよい後戻りは,失敗の

可能性だけでなく，成功の可能性も考慮に入れたものでなければならない。

しかし，そのような"より知的な後戻り"の研究は将来の課題とし，ここで，一応"知的後戻り"に関する議論を終える。

—参考文献—

1) 〔B 80〕 Bruynooghe, M. "Analyis of Dependencies to Improve the behavior of Logic Programs" 5th Cont.on Automated Deduction, Lec. Note in Comp. Science 87, Springer, 1980.

2) 〔C&P81〕 Cox, P.T. & Pietrzykowski, T. "A Basis for Intelligent Backtracking", IEEE trans. on pattern analysis and machine intelligence, Vol. PAMI—3, No.1, 1981.

3) 〔D 78〕 Dogle, J. "Truth Maintenance System for Problem Solving", MIT, AI—TR—419, 1978.

4) 〔P&P80〕 Pereiva, L.M. & Ports, A. "Selective Backtracking for Logic Programs", 5th Cont. on Automated Deduction, Lec. Notes in Comp. Science, Springer, 1980.

5) 〔L&G 80〕 Lasserre, C. & Gallaire, H. "Controlling Backtrack in Horn Clauses Programming", ACM Logic Programming Workshop, Budapest, 1980.

6) 〔S&S 76〕 Stallman, R.M. & Sus man, G.J. "Forward Reasonning and Dependency Directed Backtracking in a System for Computer—Aided Circuit Analysis", MIT AI memo 380, 1976.

3.3 ユニフィケーション・アルゴリズム

現在Prologの単一化（unification）アルゴリズムではOCK（Occur ChecK)を行っていない。このことが"論理的に建全でない"ことは明らかである。この節では，問題を高階（ラムダ式）の単一化や，特殊な項（bag, acsociative, idempotent)の単一化を除いた，第一階述語論理の項の単一化に絞って議論する。

われわれの目的は，論理的に健全な高速の一階の項の単一化アルゴリズムを探し出すことにある。

高速単一化アルゴリズムとしては〔P76〕の提案した，"equivalence class"に基づくO(n)のアルゴリズムがある。しかしこの方法は，従来の標準的な単一化アルゴリズム(〔C&L73〕参照，以下Rと略す)に比べ，データ構造が非常に複雑であり，実用性は疑問である。この方法をもとに，$O(n^3)$のより簡単なアルゴリズムが考えられるが，やはりそれでも実用性に乏しい。そこでわれわれはRの実用性(指数オーダーにも拘わらず！)を考慮して，Rの改良を考える。

提案は2段階に分かれる。最初はRにおいてOCKが不要な場合それを省くというものである。次はもしOCKがどうしても必要な場合，OCKの手間を(平均的に)減らす為の細工をするというものである。まずOCKなしでRが正しく働く場合を考えよう。

3.3.1 OCK不要の場合

ここで扱うデータはリスト(で表現した木構造)で，変数を通じて構造の共有があるものとする。

(Def.1) 表現"E"に対し<E>は"E"を示すリスト構造へのpointerである。

E_1 E_2は構造E_1(アトムまたはリスト)がE_2の部分構造であることを示す。

a Eは"a"が"E"の中に出現していること，またはaがリストEのアトムであることを示す。

$E_1-<E_2>$はリストE_1から$<E_2>$で示される部分構造を除いたリスト構造を示す。

リスト集合 $\{s,t\}$のRobinson式単一化アルゴリズムR

① $i:=0$, $si:=s$, $ti:=t$, $\sigma i:=\varepsilon$ (εはnullの代入を表わす)

$i=0,1,2,\cdots$に対し，

② $Dis:=\{s_i,t_i\}$の不一致集合

if $Dis=\phi$ then return σi

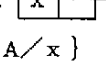
if $Dis=\{f(\cdot),g(\cdot)\}$かつ"f"≒"g", then return "clash"

;$Dis=\{x,A\}$の形でxは変数である。

③ if x A (OCK) then return "occur-tailure"

④ $S_{i+1}:=(Si-<x>-<A>)\{A/x\}$

$t_{i+1}:=(t_i-<x>-<A>)\{A/x\}$

(x←Aの代入は  とする)

⑤ $\sigma_{i+1}:=\sigma_i\circ\{A/x\}$

⑥ go to ②

このアルゴリズムが指数オーダーになる原因は②と③の所である。②の所は動しようがない

ので③を省略して良い場合（OCKなしでよい場合）を検討しよう。

（Def.2） 変数xが項tでlinear

変数xが項tでlinear ⟺

tにおけるxの出現回数が高々1回

項tがlinear ⟺

tのすべての変数がtでlinear

従って，a，f(x,a)，g(x,y,z)等はlinearであり，f(x,x)はnon-linearである。

〔命題1〕 変数を共有しない項s，tの単一化において，もしs，tの一方がlinearならばRobinson式単一化アルゴリズムのステップ③は不要，即ちOCK不要である。

証） われわれは入力{s,t}について，sがlinear かつsとtが変数を共有しないなら，ステップ①のs_i,t_iについて

α：s_iはlinearかつs_iとtiは 変数を共有しない。

ことを示す。αはi＝0の時明らかに成立している。αがiについて成立しているとする。

ステップ②で得られるDis＝{x,A}は{s_i, t_i}の部分構造である。場合に分けてステップ③を検討しよう。

(a) x s_i，A t_i： αよりxはt_iに出現していないから

β：x A

が成立する。従ってOCKは不要。ステップ④では

$$\begin{cases} s_{i+1} := (s_i - \langle x \rangle)\{A/x\} = s_i - \langle x \rangle & (\because x \text{は} s_i \text{中に1回だけ現われている。}) \\ t_{i+1} := (s_i - \langle A \rangle)\{A/x\} = t_i - \langle A \rangle & (\because x \text{はt中に現われていない}) \end{cases}$$

より，s_{i+1}はs_iの，t_{i+1}はt_iの部分構造である。従ってi＋1の時もαが成立する。

(b) A s_i，x t_i：αよりxはs_i中に出現していないから，やはりβが成立している。ステップ④では，

$$\begin{cases} s_{i+1} = (s_i - \langle A \rangle)(A/x) = s_i - \langle A \rangle & (\because x \text{は} s_i \text{中に現われていない}) \\ t_{i+1} = (t_i - \langle x \rangle)(A/x) \end{cases}$$

となる。t_{i+1}中の変数は，t_iの変数またはA中の変数であるが，t_i中の変数はαよりs_{i+1}に現われず，A中の変数は，s_iがlinearであり，従って$s_i - \langle A \rangle = s_{i+1}$に現われることがない。従ってi+1の時も$\alpha$が成立する。

以上で常にαが成立する。従ってβが常に成立するので，OCK不要であることが分

かる。

論理プログラミングにおいてOCKが必要な場合は次のようにまとめられる。

Gはゴール，HはGと単一化すべき入力節のheadを示している。

Goal ＼ Head	linear	non－linear
linear	不　要	不　要
non－linear	不　要	不　要

図3.3.1　OCKの必要性

3.3.2　OCKを簡単化するための工夫

次にOCKの簡単化を考える。ideaを絵で説明する。単一化アルゴリズムRのステップ③で

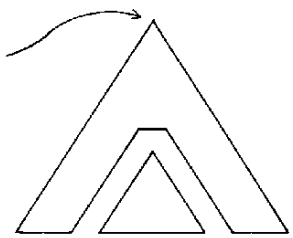


図3.3.2　x　Aの場合

$x \in A$ となる場合は左図のような場合である。

この時，xまたはxを含む項tが既にある変数yに束縛されていなければならない。

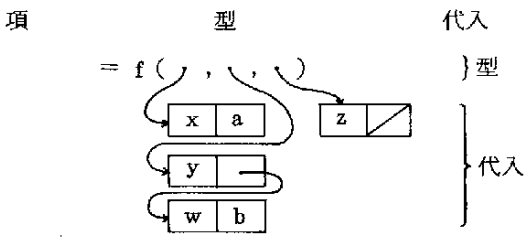
従ってx，またはxを含む部分構造を参照している変る変数yがなければ，$x \in A$ と考えられる。

このような考えに沿ってOCKの簡単化を計る。

データ構造をより正確に考えよう。

構造共有（structure sharing）方式のPrologのimplementationの場合，項は型と代入により表わされる。

例1.　f(a,b,z)＝f(x,y,z){a／x}∘{w／y}∘{b／w}



一般に項tはAθのように型Aと代入θにより表現されるとする。すると

xがAθに自由変数として存在

⟺ xはA中の変数でθにより束縛されていない，かまたは

θ中の束縛されていない変数である。

が成立する。θ中の自由変数（$\theta=\{t_1/x_1\}\circ\cdots\circ\{t_n/x_n\}$とした時，$x_1\cdots x_n$以外の変数）を$Fv(\theta)$と表わすと，上のことは，

$$x\in A\theta \Leftrightarrow x\in A \vee x\in Fv(\theta)$$

従って，束縛されていない変数xに関し，

$$\neg(x\in Fv(\theta)) \Rightarrow (x\in A\theta \Leftrightarrow x\in A)$$

が得られる。$x\in A \Rightarrow x\in A\theta$は明らかであるから，結局

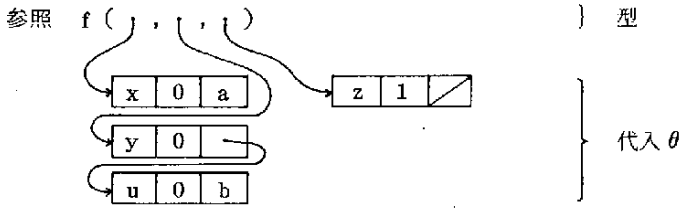
〔命題2〕 型s，t と代入θにより表現されている項 $s\theta$，$t\theta$の不一致集合を$\{x, A\theta\}$とする。この時，自由変数xについて

$$x\bar{\in}Fv(\theta) \Rightarrow (x\in A\theta \Rightarrow x\in A)$$

従って，θの自由変数$Fv(\theta)$が記録してあれば，$x\bar{\in}Fv(\theta)$の時，$x\in A$を調べるだけで$x\in A\theta$のOCK ができる。θは計算の進展と伴に幾らでも大きくなるので，θの中（bindigのスタック）を見ずに$x\in A\theta$のOCKが行えることは大きなメリットである。

θの自由変数を記録しておく為，変数セルに1 bitのフィールドを設け，θの自由変数であれば，bitをonにしておく，このbitをcheck bit と呼ぼう。

例2. $f(a,b,z)=f(x,y,z)\{a/x\}\circ\{u/y\}\circ\{b/u\}$がある変数から参照されている場合



x，y，z，uは変数

即ち変数のセルは<名前，Check bit，値>の3つ組みになり，check bit＝1なら，その変数はbinding のスタックの中で自由である。

以上の考察より標準的単一化アルゴリズムRのステップ③の所に工夫を加えた，改良型Robinson式 単一アルゴリズムのR′ができ上がる。

型s，t と代入θにより表現された項$s\theta$，$t\theta$ の改良型単一化アルゴリズムR′

① $i:=0$，$s_i:=s\theta$，$t_j:=t\theta$，$\sigma_i:=\theta$，θ中の自由変数の heck bit は既にon になっているものとする。

$i=0,1,2,\cdots$について

② $Dis=\{s_i, t_i\}$の不一致集合

if　Dis＝ϕ　then return σ_i

if　Dis＝{f(・),g(・)}かつ"f"≒"g"

then return "clash failure"

③　$x \varepsilon A\sigma_i$を調べる（OCK）。命題2より$A\sigma_i$ 全体を調べる必要があるのは，xのcheck bitがonの時だけである。

3−1　if　変数xのCheck bitがon

then if x $A\sigma_i$

then return "occur failure"

else if x A

then return "occur failure"

3−2　Aの変数でσ_iにより束縛されていない変数yのCheck bit on

④　$s_{i+1} := s_i - \langle x \rangle - \langle A \rangle$

$t_{i+1} := t_i - \langle x \rangle - \langle A \rangle$

⑤　$\sigma_{i+1} := \sigma_i \circ \{A/x\}$，xのCheck bit off.

⑥　go to ②

単一化アルゴリズムRとR′を比べると形の上では，

R′＝R＋check bit の確認＋ステップ 3−2

ということになるが，実際は大した手間の増加にならないと考えられる。

勿論R′の入力{$s\theta$, $t\theta$}の一方がlinear ならばステップ③全体を省略してよい。

そこでPrologにOCKを導入する方法としては次の案が妥当であると考える。

(イ)　Robinson式 アルゴリズムの改良型，R′単一化アルゴリズムを採用し

(ロ)　節$A_0 \leftarrow A_1, \cdots, A_m$が入力された時，$A_0 = A_0(t_1, \cdots, t_n)$として

（ロ−1）　A_0がlinearならばその事を記録しておいて（A_0は計算中に変化しない），後でA_0との単一化が生じた時はステップ　3−1 を省略する。

ステップ 3−2 は省略できない。何故なら，A_0との単一化により生まれる自由変数の記録が後で必要になる可能性があるから。

（ロ−2） A_0がnon−linearでも，$A_0 = A_0(t_1, \cdots, t_i, \cdots, t_n)$に対し，$<t_1, \cdots, t_i>$まではlinearであるようなiを記録しておき（$i \geq 0$），$<t_1 \cdots t_i>$を単一化している間は（ロ−1）に従い，$<t_{i+1} \cdots t_n>$の単一化になって初めて3−1 のステップを使う。

これはPrologのコンパイル時に行える。

このようにしてもPrologの単一化が指数オーダでなくなるわけではない。それは"同一構造に対する重複したaccess"取り除いているわけではないからである。

R'は単に同一構造に対する重複したaccess の回数を減らしているに過ぎない。（しかしその効果は充分あると考えている）

−参考文献−

1）〔C&L 73〕 Chang, C.L. & Lee, R.C.T.
: Symbolic Logic & Mechanical Theorem Proving, ACADEMIC PRESS, pp 77, 1973.

2）〔P 76〕 Paterson, M.S.
: Linear Unification, ACM, proc. of 8 th annual ACM Symposium on theory of Computing, 1976.

3）〔R 79〕 Raulets, P., Siekmann, J., Szabó, P., and Unvericht, E.
: A Short Survey on the State of the Art in Matching and Unification Problems, ACM, SIGSAM, Vol.13, No.2, 1979.

3. 4　意味論（記述）

―意味記述の新しい可能性―

3.4.1　概　論

―計画、到達点、意義等―

本論では、プロログのdenotational semanticsを与えるのが目的である。ただlogic programmingという観点に立つと、計算が証明の過程になっているということが明らかにされねばならない。従って計算と論理が十分融合して展開される体系が用意されるべきであろう。

プロログの建前は「第一階論理の証明＝計算」ということであるが、実際の使用では第一階論理という前提が多少くずれても差しつかえない。例えば

$$P(P(x)) \wedge Q(x) \rightarrow R(x)$$

等といった文もプロログの中では機能し得る。これは一種の高階論理である。こういう式がどうモデル解釈されるべきかという問題に解答を与える必要がある。

もう一つの問題としてプロログの計算の各ステップが関数としてどう解釈されるべきかという問題がある。Herbrand model上ではP(x)という式はxがHerbrand universe全てを動くものとして、その全てに於いてP(x)が成立すると解釈される。

すると情報としてP(a)、P(b)、P(f(a))、・・・といった集合が与えられているのだと解釈され得る。こう考えれば1つのHorn 節

$$P(a) \wedge Q(f(x)) \rightarrow R(g(x))$$

はP(a)とQ(f(・・・))という情報からR(g(・・・))という情報を得る関数と解釈できる。

Horn 節で表わされる関数は入力情報が増加すると出力情報も増加するという意味でScottの意味での連続性が期待できる。こうしてScott model とその上でのHorn 節の関数解釈が可能となる。

しかるにPrologの証明は背理法であり、情報の移動は

$$\neg R(g(x)) \rightarrow \neg P(a) \vee \neg Q(f(x))$$

なる論理式の関数解釈により行なわれねばならない。立ちどころにして否定の問題と選言の問題が生ずる。直観論理では命題¬Pが成立するということはPが成立しないことを知ることから得られる情報である。これが問題をこじらせる原因である。

我々が¬Pが成立するという情報をシステムを与えたとするとこれはPが証明することができないことを示すまでもなく成立するのでなければならない。従ってこれはnegativeではあるが

情報が与えられたのだと解釈することが可能である。こうして全ての情報はpositive な情報とnegative な情報の集まりからなるという解釈を行うことができる。

こうして基本的な点でのmodelの改定が行なわれる。

撰言の問題は極めて容易に解釈される。

$$p \vee g = (\neg p \rightarrow g) \wedge (\neg g \rightarrow p)$$

とすることで∨記号を消去するのである。

これで問題が全て解決したかのごとくである。残念ながら事態はこれに反していた。

$$(p \rightarrow g) \wedge \neg g$$

という情報が与えられていたとするとこの情報から計算の結果

$$\neg p$$

という情報が得られなければならない。

しかるに→の関数解釈ではpという情報が欠落しているという結論しか得られないのである。

今考えられる最も容易な解決法は空間を

$$(p \rightarrow g) \wedge \neg g \rightarrow \neg p$$

が成立する空間に制限してしまうことである。そこで再度空間の論理が何であるかを問うことになる。我々が或る時点で知っている情報の全てが状態であると考える。すると命題の真理値とはその命題の成立を知っている状態の集まりであると考えることが可能である。

$$\text{truth-value of } p = \{ s \mid s \supset p \}$$

この集合は

$$\perp \rightarrow P$$

という連続関数の不動点集合になっている。一般に連続関数の不動点集合はScottの意味でのレトラクトになっている。

Scott〔 〕ではレトラクトのなすCategory の集合値反変関手圏を考えることによりScottモデルを高階の直観論理のモデルへ拡張する仕事が行われた。ここでレトラクトカテゴリーの部分カテゴリーとして連続関数の不動点集合の作るカテゴリーを考えることができる。すると集合値反変関手圏の作り方から連続関数の不動点集合は真理値に相当する役割を演ずることになる。

このモデルの上で論理式を再解釈すると最初に述べた

$$P(P(x))$$

といった命題に意味を与えることが可能になる。このとき¬Pであることを知っているという命

題は

$$P^c$$

また¬Pは

$$P \to \bot$$

と書かれることになる。

P^c と $P \to \bot$ とが等価になる世界にモデルを制限することが可能であれば

$$(p \to g) \wedge p^c \to g^c$$

が得られるわけである。

たぶんそれは商カテゴリーを作る操作で可能ではなかろうか。

もしそうだとしても問題が完全に解決しているかどうか疑わしい。というのもこのモデル以上にもとの連続関数を回復することが可能かどうか調べる必要があるからである。

さらにもう一つ重要な問題がある。Prologのようにbacktrackを許すsystemでは情報がなくなった状態に達するとそれを判別してbacktrackすることになる。ところで

$$Cond(x=\bot)(y)(z) = \begin{cases} y & x=\bot \\ z & x \neq \bot \end{cases}$$

という関数が連続であるためには常に

$$z \sqsupseteq y$$

が成立せねばならない。これはCond 文の精神に反するわけで

$\bot$＝情報がなくなった

という解釈のもとではbacktrackをする制御関数が見付からなくなってしまう。これは極めて重大な問題であって論理式を単なるCodingされた結果のtreeとして取扱う分には$\bot$の判別ではなくunification の失敗としてbacktrackを働かせることが可能であるが、今迄述べてきたような論理式の関数解釈をxに代入することは上記のような困難を引起こす。

Scott model の範囲内でlogic programmingを正当化するのはなかなか困難であり、もしかすると不可能なのかも知れない。むしろトポスの中で論ずることの方が自然であるような気もするのである。

もしそうであれば逆にトポスの中でlogic programmingを如何にすべきか、より積極的に考えるべきであろう。

プロログのsemanticsとしてはlogical information の取扱いもさることながらinterpreter の制御の流れを記述することも重要である。この部分は完全にプログラム

であり、λ-Calculus の範囲である。従ってmodelについて心配することはない。logical information の部分を black-box にしたinterpreter を仮に abstract interpreter と称することにして、まずabstract interpreter を考えることにした。第一番手にsimple な General top down backtrack syntax analyser に従ったinterspreter を記述し、しかるのちに coroutine 概念に基づくinterpreter を記述してみることにした。プログラムカウンターやスタックに相当する部分がcontinuation として抽象化されればdenotational semantics としての役割を十分にはたすことができるのであるが、時間の都合のため省略させていただくことにした。

最後にimplementation technique についても詳述すべきであるがこれは技術編の問題でもあるので3.2 節にゆずる。

基礎理論の基本的な目的はlogic programmingが展開し得るに十分な広さをもったmodelと論理系を提供することにあろう。そうすればProlog や Lispを拡張した言語をその中で捜すことも可能であろうとし、又言語そのものについて論証することが可能となる。Prologの前提とする第一階論理ではλ一計算を正当化できない。そこで高階の直観論理系をもってきた次第である。

3.4.2 Abstract Prolog interpreter

Prolog の syntax として次のようなものを採用しよう。

```
Program ::= Clause-groups
Clause-groups ::= Clause-group |
                  Clause-group Clause-groups
Clause-group ::= Head Clauses
Clauses ::= Clause | Clause Clauses
Clause ::= Clause-Head :-∧-part
∧-part ::= terms
terms ::=Λ | term,terms
term ::= Predicate-Functor|/
Head ::=Λ
Clause-Head ::= Predicate-Functor
```

即ちプログラムはClause 群の並びであり、Clause 群は同一名のHeadをもつClause の集まり、Clause は、Head:-Predicate-Functor 列で/を許すもの

である。Predicate-functorはskolemizeされた述語そのもののこととする。

いまプログラムが

$d_1\ d_2\ \cdots\ d_n$

ただし　$di = {}'p_i : -p_{i1}, \cdots, p_{in}^{i}{}'$

と並んでいるものとしよう。

d_i はlogical stateを移す関数として働らくからその関数を$\rho(di)$で指定する。またd_i に対しそのHead の述語名を得る関数をhead(d_i),d_i に対し同じHead をもつ次のClauseを得る関数をnext(d_i)、1つ手前のClauseを得る関数をpre(d_i)とする。

Stackは一本であり、Stack top current state となっているものとする。stackのdata formatは次のようなものである。

stack-element : <process-status、logical state>

process-status :

<Clause-name `stack-pointer-of-caller

subprocess-status >

subprocess-status : list of

<restart-index stack-pointer>

いま文d=$'p'$ が与えられたときのこのプログラムの実行過程全体を1つのプロセスと仮に呼ぶことにして、プロセスステータスとはそのプロセスの最初に適用されたClause名、そのプロセスを呼び出したプロセスのstart時点でのstack-topの位置、そのプロセスが呼んだ乃至は呼ぶべきプロセスの状態記述からなっている。

入力文をd_e とする最初のlogical stateは

$\rho(d_0)(\perp)$

である。

process statusはclause-name=d_0 が決まっており、callerがないからcallerのprocess statusへのpointerは自分自身のそれ即ち、

stack-pointer-of-caller=1

subprocess-statusは、

restart−index = first(head(d_0))

stack−pointer = ⊥

である。

即ち初期状態は

Stack : ≪≪d_0,1, ≪≪first(tail(d_0)),

,norm≫≫,ρ(do)(⊥)≫

となる。

但しここで関数first(q) は q を Clause−head の述語名とするとき q で決まる Clause 群の最初のClause 名を与えるものとする。

いまstp でStackのある番地を示すものとする。このときstp 番地の subprocess−status へのindex ix が与えられているとしてStack(stp) は、

≪d,n, ≪≫,・・・,≪d',pt≫,・・≫≫,s≫

ix

である。

d='p:=p_i,・・・,p_{ix},・・・,p_p'

となるからp_{ix} の述語名q が定まる。従ってstp とix が定まればq はredundunt な情報となるがそれは一応おくとして、最初のClause をClause 群q から撰んで進むルーチンを考える。

これをm(q,stp,ix) とするとmain routineは

```
main:
     Define rec
       Casem(tail(d0),1,1)=
fail:if Stack(1).subprocess-status(1).
          restart−index ≠ φ then rec;
SUC:RTN(success);


        if Stack(1),subprocess−status(1).
         restart−index= φ
         then RTN(fail);
       end;
```

```
initial set;
RTN(rec);
end;
```

となりルーチンm(q,stp,ix) は

```
sub1:m(q,stp,ix)
  Do
   d←Stack(stp). subprocess−status(ix).
    restart−index;
   Stack(stp). subprocess−statas(ix).
   restart−index←next(d);
   if ρ(d)(state,logical−state)¬= Ø
    them STN(m(d,stp,ix));
   untill next(d)=φ;
    RTN(fail);
   end;
```

と定義される。

即ち、まずfirst(fail(d_0)) なるclauseを適用してみてunification がとれるか調べる。

unification が可能であればこのclause に従ってsubprocess を起動していく。unification がとれなければ次々と進み最初のunification がとれるClause を選択してそのclause に従ってsubprocess を起動していく。その結果が失敗であれば次のselectionを求めていく。

次にClause の適用とそれが生みだすプロセスの管理を行うルーチンを定めるとしよう。

Clause 名dが

$$d='p:=p_1,\cdots,\ pi,\cdots,pr'$$

と書かれるものとする。このとき

length(d)=r

Pred−name−of(pi)=pi の述語名　or ／

⊥=未定

なる関数を定める。

すると目的とするルーチンは、

```
sub2:m(d,stp,ix)
 Push(<<d,stp,
   <<first(Predname-of(p1)),⊥>, ...
   first(Predname-of(pr)),⊥ >>>,
   <ρ(d)(state,logical-state)>>);
Stack(stp). subprocess(ix). stack-pointer
←length(Stack);
if length(subprocess-status)=0
  then RTN(success);
stp←length(Stack);
ix←1;
r←length(d);
While ix≤r
  do
     q←head(Stack(stp). subprocess(ix).
          restart-index);
     if m(q,stq,ix)=fail
        then
        While m̂(q,stp,ix)=fail
          do
        if Stack(stp). subprocess-status(ix-1)=/
          . backtrack-imfo=abnorm
           then PoP(length(Stack)-stp);RTN(fail);
        if ix=1
       then PoP; RTN(fail);
        Pop;
           Stack(stp). subprocess-status(ix)
          . restart-index
```

```
                ←first(head(Stack(stp).
                 subprocess-status(ix).
                 restart-index));
                 ix←ix-1;
                 Stack(stp). subprocess-status(ix)
                . restart-index
                 ←next(Stack(stp).
                 subprocess-status(ix).
                 restart-index));
                 end;
        if q=/and ix+1≧r
           then POP(length(Stack)-stp);
           RTN(fail);
             ix←ix+1;
           end;
           RTN(success);
        end;
```

となる。

まず始めにProcess-statusとlogical stateを作ってStackにしまい、callerのsubprocess-statusのstack-pointer欄にcurrent-stack-topの位置をしまう。

次にsubprocessを次々に呼び失敗したらBacktrack routineを呼ぶ、失敗が不可避的であればfailをもってreturnし、成功であればsuccessをもってreturnする。

次にBacktrack用ルーチンを定めよう。

```
        sub3:△m(q,stp,ix)
           stp1←stack(stp). subprocess-status(ix).
                stack-pointer;
           d←stack(stp1). Clause-name;
           r←length(d);
           if r≠0 and △m(d,stp1)=success
              them RTN(success);
```

```
        d'←Stack(stp). subprocess-status(ix).
            restart-index;
        While d'¬=Ø
        do
         Pop;
         q←head(d');
         if m(q,stp,ix)=success
             them RTN(success);
       end;
      RTN(fail)
    end;
```

親プロセスのprocess-statusの格納されたstackの位置stp とそのプロセスでの子プロセスとしての位置ixからそのプロセスのprocess-status の格納されたstack の位置stp1を知る。そのプロセスにBacktrack $\hat{m}(d,stp1)$ を掛けて失敗したらそのプロセスのalternative を試みるというわけである。

ここで使ったもう一つのBacktrack用ルーチンを次に定める。

```
sub4: m̂(d,stp)
    ix←length(d);
    if Stack(stp). subprocess-index(ix). restart-
        index = ／ then PoP(length(Stack)-stp);
        RTN(fail);
  While 1≤ix≤length(d)
      do
      q←head(Stack(stp). subprocess-
        index(ix). restart-index);


    if m̂(q,stp,ix)=success
      then ix←x+1;
      else if Stack(stp). subprocess-index(ix-1).
              restart-index = ／
```

```
                then RTN(fail);
                else ix←ix-1;
            fi;
        fi;
        end;
    if ix>length(d) then RTN(success);
                    else RTN(fail);
        end;
```

子プロセスのうち一番最後に実行したものにBacktrackをかける。失敗なら1つ前の子プロセスにBacktrackをかけ、成功したら1つ後の子プロセスにBacktrackをかける。
Whileループを失敗で抜ければ失敗、成功で抜ければ成功である。

第4部　ロジック・プログラミングの現状とマシンの実例

4. ロジック・プログラミングの現状とマシンの実例

4.1. システム例

4.1.1. Prolog／KR

Prolog／KRは高度に会話的なシステムとすることを目標に，東京大学大型計算機センターのUtilisp上に開発された。言語仕様としてはPrologを基本としているが，その欠点をカバーするために大幅に拡張してある。

Prologの最大の特徴はそのパターン・マッチングの機能（ユニフィケーション）にある。これは構造を持ったデータを扱う上で非常に便利である。一方，バックトラックの機能は有用ではあるものの両刀の剣の感がある ― 制御されない全域的バックトラックは望ましくない。そこでこのバックトラックを制御するために，多くの処理系ではカット・オペレータを用意しているが，これはあまりにも原始的で大規模なシステムを構成するのには使いづらい。Prolog／KRでは高階の制御用述語を用意してカット・オペレータに替えている。

PrologやLispといった言語は人工知能的プログラミングに向いているといわれている。そのようなプログラミングでは，アルゴリズムが確定している場合は少なく，むしろプログラムを変更しながらアルゴリズムのテストを行っている。そのような環境ではプログラムの実行速度もさることながら，そのテストの容易さが問題となる。会話的にプログラムを構成してゆくためには 何よりもまずそのプログラムが動くことが重要である。自動バックトラックはその意味では非常に有効である。効率の良い制御は，プログラムが動き出してデータが集まってから，それらをもとにして与えてゆけば良い。もちろん，バックトラックが本質であるような部分は最後まで残ることになるであろう。

Prolog／KRは上記のようなプログラミングを想定して作られている。そのようにして出来上がったプログラムはバックトラックが用いられる部分とそうでない部分が混在しており，どちらも重要であると考えられる。

A. シンタックス

Prolog／KRはLispと同じシンタックスを採用している。たとえば，'FGKLのシンタックスで，

```
append(( ), *x, *x)←
append(*a . *x, *y, *a . *z)←append(*x, *y, *z)
```

となるappendの定義は，Prolog／KRでは，

```
(assert (append ( ) *x  *x))
```

```
( assert ( append ( *a . *x ) *y ( *a . *z ) )
         ( append *x  *y  *z ) )
```

と書かれる。＊で始まるシンボルは変数を表わしている。

このシンタックスには，以下の利点がある：

- プログラムの内部表現（リスト構造）と外部表現が1対1に対応しているので，構造エディタの使用が楽である。
- プログラムとデータが完全に同一の形で表現できる。
- 引数の受け渡しに融通性がある。

たとえば，Prolog／KRでは引数の数が不定個（LispのFEXPRのようにひとまとめのリストで渡されるという意味ではない）の述語が定義できる。一例として，ある述語をリストの各要素に順に作様させるような（2階の）述語mapを考えてみよう。

```
( map reverse ( ( a  b ) ( c  d ) ) *x )
```

を実行すれば，

```
*x = ( ( b  a ) ( d  c ) )
```

となるし，

```
( map append ( ( f  o  o ) ( foo bar ) )
             ( ( b  a  r ) ( poo ) )
               *x )
```

を実行すれば，

```
*x = ( ( f  o  o  b  a  r ) ( foo bar poo ) )
```

となるはずである。最初の例は3引数（2引数述語とその引数），後の例は4引数（3引数述語とその引数）である点に注意してほしい。このmapは図4.1.1のように定義できる。

```
(ASSERT (MAP *FN . *ARGS)
        (DIVIDE *ARGS *CARS *CDRS) (*FN . *CARS)
        (RETURN (MAP *FN . *CDRS)))
(ASSERT (MAP *FN . *ARGS) (NILS *ARGS))

(ASSERT (DIVIDE NIL NIL NIL))
(ASSERT (DIVIDE ((*CAR . *CDR) . *REST)
                (*CAR . *CARS)
                (*CDR . *CDRS))
        (DIVIDE *REST *CARS *CDRS))

(ASSERT (CARS NIL NIL))
(ASSERT (CARS ((*CAR . *CDR) . *REST) (*CAR . *RESULT))
        (CARS *REST *RESULT))

(ASSERT (CDRS ((*CAR) . *REST) *Y) (FAIL))
(ASSERT (CDRS ((*CAR . *CDR) . *REST) (*CDR . *RESULT))
        (CDRS *REST *RESULT))
(ASSERT (CDRS NIL NIL))

(ASSERT (NILS NIL))
(ASSERT (NILS (NIL . *NILS)) (NILS *NILS))
```

図4.1.1　MAPのプログラム

B. パターン・マッチング

Prolog／KRは，ユニィフィケーションに基づいたパターン・マッチングの他に，いくつかの機能を持っている。

a. 実行可能パターン

Prologで用いられているユニフィケーションは，構造データを取り扱う上で非常に便利であるが，その反面，判定をともなった複雑なパターンは記述できない。たとえば，「FOOあるいはBARで始まるリスト」といったパターンは書けないので，パターンをいくつか用意するか，又は本体でその判定を行わねばならない。Prolog／KRの実行可能パターンを用いれば，これらは解決できる。

実行可能パターンとは，！で始まるリストのことである。そのパターン内では＊だけの変数は特別扱いされる ― ＊がまずマッチングの相手に束縛され，その後そのパターンの！以降の部分が実行される。実行の成否が，マッチングの成否となる。たとえば次の２つのパターン

```
( ( !member *( foo bar ) ) poo )
```

```
( bar  *zoo )
```

は以下の手順でマッチする。

(1) ＊がbarに束縛される。

(2) (member bar (foo bar))が実行され，成功する。

(3) ＊zooがpooに束縛される。

(4) 全体のマッチングが成功する。

実行可能パターンは関数の代用とすることもできる。Prologでは，述語の引数として現われる関数はスコーレム関数と呼ばれ，実行されることはないが，Prolog／KRでは，そこに実行可能パターンを書いてやれば，あたかもそれが関数として評価されたような効果を持たせることができる。たとえば，階乗は次のように定義できる。

```
( assert ( factorial *n
  ( ! cond ( ( =*n  0 ) ( =  *  1 ) )
           ( ( true )
             ( times  *n
                      ( ! factorial ( ! minus *n  1  * )
                                    * )
                      * ) ) ) )
```

実行可能パターンが入れ子になって用いられている。！と＊の対応を示すためにそれぞれに添字をつけてみよう。

```
( assert factorial *n
  ( !1 cond ( ( =*n  0 ) ( =  *1  1 ) )
            ( ( true )
              ( times *n
                      ( !2 factorial ( !3 minus *n  1 *3 )
                                     *2 )
                      *1 ) ) ) )
```

その上で，

```
( factorial  3 *f )
```

の実行を追ってみよう。

(1) ＊n＝3

(2) $*_1 = *f$

(3) (times 3 (!$_2$ factorial …) *$_1$) が実行される。

(4) そのために，まず(factorial (!$_3$ …) *$_2$) が実行される。

(5) そのために，まず(minus 3 1 *$_3$) が実行される。

(6) *$_3$ = 2

⋮

（中略）

⋮

(7) *$_2$ = factorial (2) = 2

(8) *f = *$_1$ = times (3, 2) = 6

b．マッチングの制限

パターンを ' でクオートすることによって変数とマッチすることを防止できる。クオートされたパターンは，変数名を含めてそれと同一のパターンとしかマッチしない。この機能は，たとえば定義の一部を消去する場合に用いられる。以下のような定義を考えてみよう。

```
( assert ( p a 1 ) )
( assert ( p a 2 ) )
( assert ( p b 36 ) )
( assert ( p *x 0 ) )
```

そして，Pの第1引数がaのものだけを消去したいとすると，

```
( retract ( p a *any ) )
```

をくり返せば良いのだが，これでは，最後の

```
( p *x 0 )
```

まで消えてしまう。これを防止するためにはaをクオートして，

```
( retract ( p 'a *any ) )
```

としてやれば良い。'aはaあるいは'aにしかマッチしない。同様に，

```
( retract ( p '*x *any ) )
```

とやれば，最後のものだけが消去される。

```
( retract '( p *x *any ) )
```

ではどれも消去されない。

c．変数のプリフィックスの変更

プログラム自身をデータとして取り扱いたい場合（コンパイラやエディタ）には，プログラム中の変数を定数として扱った方が便利である。そのような場合には，変数のプリフィッ

クスを変えてしまえば良い。たとえば，Prolog／KRの構造エディタでは＆を変数のプリフィックスとして採用している（これも，さらに変更可能である）ので，findコマンド（F）で，

F　＊x

とやると，＊xという変数だけが見つかる。もし変数のプリフィックスが＊のままだと，＊xは何とでもマッチしてしまう。

ただ，この解決法は完全ではなく，たとえばコンパイラ自身をセルフコンパイルする場合には困ってしまう。

蛇足ではあるが，プリフィックスをAからZまでの大文字にしてしまえば，Edinburgh風の変数のシンタックスが実現できる。

C. 制御構造

a. if-then-else

Prologでは条件分岐は，カットオペレータ！を用いて，たとえば，

P：－Q，！，R．

P：－S．

のように表わす。しかし，これは，

P：－if Q then R else S.

と書いた方がわかりやすい。Prolog／KRではこれは，

（assert P（if Q R S））

となる。

b. 否　　定

～Pという述語を導入するとホーン節の枠組からはみ出してしまう。たとえば，

P：－～Q，R

は，

P∨～～Q∨～R

という意味で，これは，

P∨Q∨～R

と同じであるから，正項がふたつになって，ホーン節ではない。Prolog／KRでは，手続き的な否定，即ち「Pの実行に失敗する」という意味でnot（P）を導入している。

またPではないという主張もPrologでは取り扱えない。否定の形はゴールだとして実行してしまうからである。Prolog／KRでは，否定の主張は，

(assert P (fail))

のようにする。failは親の呼び出し(この場合はP)を失敗させる働きを持つ。そこで,

たとえば,

(assert (fly !penguin) (fail))

(assert (fly *x) (bird *x))

としておけば,一般にbirdは飛ぶが, penguinは飛ばないという主張になる。penguinの前の!は,

(fly *x)

が最初の主張につかまって失敗するのを防ぐためにつけてある。

c. くり返し

たいていのProlog処理系ではくり返しは,バックトラックを使って実現される。まず,

repeat .

repeat:-!, repeat .

と常に成功する述語を用意しておき,

repeat, P, false.

とすれば,Pを無限にくり返す。Prolog/KRではくり返し用の述語を用意している。

(1) for-all

for-allは

$\forall x\ P(x) \rightarrow Q(x)$

にちなんで付けられた名前で,

(for-all (P *x) (Q *x))

と書けば,Pを満たすすべての*xについて,Qを実行する。たとえば,

(for-all (member *x (apple orange grape))

(eat *x))

とすれば, (eat apple), (eat orange), (eat grape)を実行することになる。

(2) loop

loopはその引数が成功する限りそれらをくり返す。repeat-untilやwhile-doはこのloopの特別な場合として定義できる,即ち

(assert (repeat *s until *p)

(loop *s (not *p)))

(assert (while *p do *s)

```
( loop *p *s ) )
```

ただし，＊sはいつも成功するものと仮定している。

d．すべての解を集める

ある述語を満たすものすべてのリストを作るという作業は，バックトラックを使っては容易に作れない。Prolog／KRではaccumulateという述語を用意している。

```
( accumulate *struct *pred *var )
```

は＊predを満たすすべての＊struct（変数でも，構造を持ったデータでも良い）のリストを＊varの値とする。たとえば，

```
( assert ( intersection *x *y *i )
  ( accumulate *z
        ( and ( member *z *x )
              ( member *z *y ) )
        *i ) )
```

は＊xと＊y（それぞれリスト）の共通要素のリスト（＊i）を求める述語の定義である。

e．コルーチン

Prolog／KRのコルーチンには，値を生成する部分と，それを消費する部分とがある。値を生成する側は，普通のプログラムで良いが，それはいくつかの（無限個でも良い）解を持っている必要がある。消費側は，そのプログラムをinitiateを用いて初期化し，次にnextで1つづつ値をとり出す。もう解がなくなっていれば消費側も失敗する。例を示そう。

```
( and ( initiate ( member *x ( 子 丑 寅 …… ) )
                 *gen )
      ( next *gen *e1 )
      ( next *gen *e2 )
            ⋮
```

とすれば，＊e_1＝子，＊e_2＝丑……となる。

f．並列実行

Prolog／KRの基本戦略はバックトラックを供なった深さ優先の探索であるが，場合によっては広さ優先の探索が必要になる。その場合のためにpor（parallel orのこと）が用意してある。

parallel prologのような完全な並列実行は，現存のハードウェア上では非能率で使いものにならない。

D. 多重世界

Prolog／KRにはworldと呼ばれる，いくつかの独立な述語定義の世界がある。一般にはシステムが用意した述語の世界（これは，どこからでも見える）とユーザの標準世界（standard-worldと呼ばれる）の2つが存在する。ユーザはこの他に任意の世界を構築することができる。それらには名前つきのものと名前なしのものがあり，名前つきの世界からは外の世界は（システム述語を除いては）見えない。また他の世界を覗くには，その世界に入らねばならない。これらの機能により述語間のモジュラリティが保たれる。

名前つきの世界は，

（create-world　名前・述語定義の並び）

あるいは，

（load-world　名前　ファイル名）

によって作られる。そして，

（with　名前．述語呼び出しの列）

によって参照される。自分の世界以外を参照するには（システム述語を除いて）このwithを用いなければならない。

名前なしの世界を作るには，withの第1引数に名前の代わりに述語定義の列を与える。このようにして作られた世界からは，外の世界がそのまま見える。

たとえばpretty printerの定義の一部を変更して新しい定義を作ろうと思えば，以下のようにすれば良い。

```
(assert (pretty-print *x)
    (definition princ *princ)
      ; princの定義を*princに入れる
    (with pretty-printer
      ; pretty-printerの世界に入る
      (with ((prin1 *princ))
        ; prin1の定義を外側のprincと同じにする。
          (print *x)
            ; pretty-printerの世界のprintを呼ぶ))))
```

Prologでグローバル変数が必要な場合には述語の形式で値を保存する。たとえばxの値をaにするには,

```
( and ( retract x )
      ( assert ( x a ) ) )
```

を用いる。これと，withを組み合わせて用いると，Lisp風の動的スコープを持った変数を実現できる。名前なしwithで変数の新しいスコープを開くわけである。

E. 会話的デバッグ機能

a. ステッパとトレイサ

ステッパは新しい述語が呼ばれるごとに停止しながら述語呼び出しを実行する。停止の時には，その述語呼び出しの様子を出力し，ユーザからのコマンドを受けつける。ユーザはそこで以下のようなコマンドを入力できる。

- ステップ実行を続ける。
- 正常実行に戻す。
- 現在の述語呼び出しの実行が終わるまでは止まらない。
- 指定した述語が呼ばれるまでは止まらない。
- 実行を止める。
- エディタを呼ぶ。
- 呼び出しの履歴を見るetc.

このステッパは,

```
( step  述語呼び出し )
```

で呼び出される他，エラーやユーザによる割入みの際にも呼び出され，デバッグ・パッケージとして用いられている。

トレイサはいちいち止まらないステッパだと思えば良い。もちろん全部の呼び出しを出力する必要が無い場合には，必要な述語呼び出しのみを出力させることが可能である。

b. エディタ

Prolog／KRのエディタは構造エディタになっており，プログラムの実行中いつまでも呼び出せる。

このエディタはPrologに似たパターン・マッチャを備えている。変数が&で始まる点とマッチングが一方向である点だけが異なる。このパターンは，たとえば，全部とりかえ,(replace all)コマンドRAの引数として書けるが，非常に強力かつ有用である。

```
RA( foo &1 &2, &x )( foo &2 &1, &x )
```

により，fooの全呼び出しについて第1引数と第2引数の入れ替えができる。

また，エディタコマンダはexecmという述語を介してプログラム中から呼べるので，マクロを作る際にはProlog／KRの全機能が使える。

このエディタは定義体の入ったファイルをエディットするのにも使える。述語名を変更したいとき等には，ファイルに対してRAコマンドを発効するのが楽である。しかし，一般にはプログラムを実行しながらのエディットが多いので関数単位に使うことになる。この場合には，変更された新しい定義を再びファイルに書き込んでおく必要がある。そのために，Prolog／KRではファイルから定義体をロードした際にそれらを全部記録しておき，

```
( store  ファイル名)
```

で定義体を全部ファイルに再格納することが可能になっている。従って，典型的なProlog／KRのセッションは以下のようになろう：

```
( load   " a.file " )
        ⋮
( do-some-test )
( edit predicate )
        ⋮
( store " a.file " )
( quit )
```

c．エラーと割込み

エラーやユーザからの割込みの際にはステッパに制御が移る。正確には，ステップモードをオンにして，errorあるいはattentionが呼び出される。errorは2引数（メッセージと場所），attentionは0引数の述語である。ユーザはこれらを適当に定義することによってエラーや割り込みの際の動作を変更できる。たとえば，未定義述語が呼ばれた場合の標準動作はエラーだが，これを単に呼び出しが失敗することに変更するのは容易である：

```
( assert ( error *mes *p )
    ( select *mes
      ( " UNDEFINED PREDICATE "  ( false ) )
      ( ?  ( standard-error *mes *p ) ) ) )
```

とすれば良い。この例からもわかるように，エラーの種類はメッセージによって区別される。standard-errorは本来のerrorと同一の動作をする。

割り込みでステッパに入った場合は，正常の実行モードに戻すコマンドを入力すれば実行

が再開できる。ただ，現在割り込みに関する処理能力が充分でないので，多くの場合，実行の再開は，割り込み時点の直後からとはならず，正しい実行再開は保証されていない。

4.1.2. DURAL

DURALはPROLOGをベースに種々の機能拡張をはかった述語型プログラミング言語である。PROLOGの持つ論理的な整合性を損わないでPROLOGの機能を拡張することが，DURALの設計目標の中心となっている。したがって，以下の各項目のようなDURALの新しい機能は，すべて論理的な裏付けが行われている。

(1) PROLOGではホーン節で表現されるが，DURALでは相対ホーン節（relativized Horn clause）に拡大されている。

(2) DURALでは様相記号の使用が認められ，簡単な様相論理（modal logic）の表現が許されている。

(3) DURALには2種類の異なる証明機構が内蔵されており，ユーザが自由に切り換えて使用することができる。

DURALの処理系は，DEC System−2020のTOPS−20オペレーティングシステムの下でMacLispおよびInterLispで書かれている。処理系はLisp上に実現されているので，Lispの会話環境が活用できるだけでなく，DURAL独自の会話環境の充実にも考慮が払われている。

DURALはプログラム・シンセシスやデータベースへの演繹的な問合せなどに応用されている。本節では，DURALの設計思想，特徴について報告する。

A. DURALの構文

DURALの構文を図1に示す。構文はBNF記法を拡張したAN記法で書かれている。AN記法では｜は選択枝を，〔 〕は随意要素を $_{\circ\circ\circ}$ は $_{\circ\circ\circ}$ の任意個の並びを表わす。

<プログラム> ＝ <相対ホーン節>$_{\circ\circ\circ}$
<相対ホーン節> ＝ <正のリテラル>〔<リテラル>$_{\circ\circ\circ}$〕｜<ゴール節>
<ゴール節> ＝ <空節>｜<負のリテラル>〔<リテラル>$_{\circ\circ\circ}$〕
<リテラル> ＝ <正のリテラル>｜<負のリテラル>
<正のリテラル> ＝＋〔<様相記号>〕<素論理式>
<負のリテラル> ＝−〔<様相記号>〕<素論理式>

図4.1.2 DUARLの構文

<素　論　理　式> ＝ （<述語><項>$_{000}$）

<述　　語> ＝ <Lispのアトム>

<　　項　　> ＝ <Lispのアトム>｜<変数>｜（<関数><項>$_{000}$）

<変　　数> ＝ ＊<Lispのアトム>

<関　　数> ＝ <Lispのアトム>

<様　相　記　号> ＝ S｜EX

図4.1.2　DURALの構文

B．相対ホーン節

PROLOGではプログラムはホーン節で構成される。ホーン節には高々1個の正リテラルしか含まれない。AN記法では，<ホーン節>＝〔<正のリテラル>〕〔<負のリテラル>$_{000}$〕と表現される。一方，DURALではプログラムは相対ホーン節で構成される。相対ホーン節（relativized Horn clause）は一般化されたホーン節であり，PROLOGで扱うホーン節を包含する概念である。ホーン節に対する証明機構（導出原理）は相対ホーン節にも適用できるように拡張できる〔Loveland 78〕。

正のリテラルが相対ホーン節の先頭に現われる場合は，その相対ホーン節をPROLOGと同様にDURALでも手続きの定義あるいは事実の表明（登録）と見なしている。これ以外の正のリテラルはLispにおいて評価（eval）される。（DURALの処理プログラムはLispで記述されていることに注意。）一方，負のリテラルは，PROLOGと同様にゴール節と見なされ，DURALにおいて評価される。

正のリテラルをLispで評価したときに返される値がnilのときに限り統一化（unification）が成功したと見なす。正のリテラルのLispでの評価と統一化との関係を図4.1.3に示す。

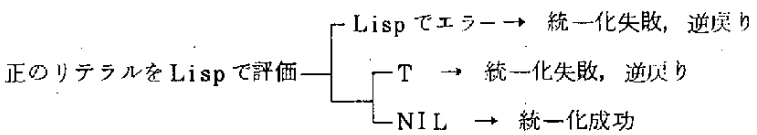


図4.1.3　正のリテラルの評価と統一化との関係

強制逆戻りを起させる述語FailureはDURALでは相対ホーン節を用いて

＋（Failure）＋T

と書かれている。

相対ホーン節を用いると，DURALとLispとを混用したようなプログラムが書ける。例えば，大小判定のような簡単な述語や算術演算の類は単に結果を求めるだけならばLispで書いた方が高速である。また，正のリテラルを用いることで，実行可能述語（evaluable predicate）の目印ともなっており，PROLOGのように実行可能述語が予約語となっている場合よりもプログラムの読み易さが向上している。例で説明しよう。

例1. 10との大小比較により答1または0を返すプログラムとその実行。

(1) ＋（Result *x 1）－（GE *x 10）

(2) ＋（Result *x 0）

(3) ＋（GE（S *x）（S *y））－（GE *x *y）

(4) ＋（GE *x 0）

ゴール1 －（Result 11 *r） ゴール1の答（*r）は1

ゴール2 －（Result 9 *r） ゴール2の答（*r）は0

ただし，Sは後続関数（successor function）である。つまり，（S x）＝x＋1．

例1ではGE（≧の意味）は数を0と後続関数Sとで表現することによって定義されている。しかし，この方法ではいかにも速度が遅いのでGEの部分にLispの関数を使うことにする。すなわち，GEを実行可能述語に指定して，GEをLispで評価する。

例1' 例1の高速実行版プログラム

(5) ＋（Result *x 1）＋（LESSP *x 10）

(6) ＋（Result *x 0）

例2 加算のプログラム

(1) ＋（Add 0 *y *y）

(2) ＋（Add（S *x）*y（S *z））－（Add *x *y *z）

ゴール1 －（Add 2 3 *z） ゴール1は加算としての使い方。

ゴール2 －（Add 2 *y 5） ゴール2は減算としての使い方。

ゴール3 －（Add 2 3 5） ゴール3は述語としての使い方。

ゴール4 －（Add *x *y 5） ゴール4は生成関数（generator function）としての使い方。

例2' 加算の高速実行版プログラム

(3) ＋（Add *x *y *z）＋（ASSIGN（PLUS *x *y）*z）

ここにPLUSはLispの加算関数である。また，Lisp関数ASSIGNは第1引数を評価し，その値を第2引数で与えられるDURALの変数へ渡す。(3)を用いるとゴール1は1ステップで答えがでること，およびゴール2，3，4に対して(3)は全く役に立たないこと，が容易にわかる。つまり，柔軟性を犠牲に（加算用に専用化）して高速化を果したものである。

ゴールが満されたときに答を陽に出力させたいときには実行可能述語ANSを用いたゴールを指定する。例えば，

ゴール5　－（Add　2　3　＊z）＋（ANS　＊z）

（ANS　＊z）を実行すると＊zが出力される。ゴール5を論理的に同値変形すると，

〔（Add　2　3　＊z）⊃（ANS　＊z）〕

となる。これは「もし＊zが（Add　2　3　＊z）を満せば，それが答（ANS）である」と読むことができる。一方，PROLOGでは実行可能述語OUTを用いて，

ゴール6　－（Add　2　3　＊z）－（OUT　＊z）

と書く。ゴール6を論理的に同値変形すると

¬〔（Add　2　3　＊z）∧（OUT　＊z）〕

であり，これは導出原理（背理法）でとられている論理式である。ゴール5の方がゴール6よりも直截的で見やすい。また，ゴール5は，導出原理を質問応答システムに応用する場合に良く用いられる"answering predicate"の形をしており，ゴール6よりも自然である。

ANSに似た実行可能述語としてQUERYがある。QUERYはANSと同様に答を出力する他，副作用としてPROLOGに本来備っている逆戻り（backtrack）のメカニズムを起動する。つまり，答を一つ得たならばこれを出力し，一方ではこの答が見つからなかったかのようにバックトラックを始める。これを繰返すと，複数の答を持つゴールに対して全部の答を集めることができる。QUERYはデータベースの問合せに相当する。

C. 様相記号

例2と例2'のプログラムを混ぜて書いておくと，ゴール節に対してどちらのプログラムが反応しているのか分からなくなる。例えば，

－（Add　2　＊y　5）＋（ANS　＊y）

と書いて，例2'のプログラムではうまく動かないだろうと思っていると，例2のプログラムの方が動いて答が出て来たりする。そこで，節を区別するためにDURALでは様相記号Sを用いる。SはStrong provabilityに由来している。すなわち，Sを付けたリテラルは予め「高速版の節」と宣言した節とだけ統一化できる。これに対して，Sを付けない負のリテラルはどのような節とも統一化してよい。したがって，例2'のプログラムを

3') ＋S（Add ＊x ＊y ＊z）＋（ASSIGN（PLUS ＊x ＊y）＊z）

と定義しておくと，

ゴール －S（Add 2 ＊y 5）＋（ANS ＊y）

は高速版の節だけで答を探すので，答が見つからない。このような節の種別を述語論理の立場から見ると一種の様相論理といえる。すなわち，｛高速版の節｝は｛節｝の部分体系〔松本71〕を成しており，Sは通常の様相記号で言えば□（またはL）に相当する。

節の区分をする機能を生かしてデータベースの問題を扱うことができる。データベースの内容を考えてみると，外延的部分（個々の事実を集めたもの）は時間とともに刻々変化するのに対し，内包的部分（一般的ルールのようなもの）はあまり変化しない。そこで，DURALでは様相記号の表現力を応用してこの2つの部分を区別して扱う。下の例ではSを「変化しない部分」に対応付ける。

例3. 有向グラフの辺（Arc）と経路（Path）の問題。

＋S（Path ＊x ＊x）

＋S（Path ＊x ＊y）－（Arc ＊x ＊z）－（Path ＊y ＊z）

＋（Arc 1 2）

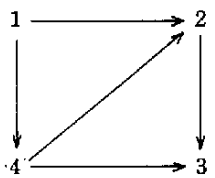
＋（Arc 1 4）

＋（Arc 2 3）

＋（Arc 4 2）

＋（Arc 4 3）

辺の関係を下図に示す。



（数字は頂点を表わす）

このとき，ゴール

－（Path 4 ＊w）＋（QUERY ＊w）

を与えると，｛4，2，3｝が答として求まる。一方，ゴール

－S（Path 4 ＊w）＋（QUERY ＊w）

を与えると，Sが効いて答は｛4｝だけになる。

様相記号を用いてデータベースの無矛盾性条件を取扱うというデータベースへの応用も行われている〔奥乃 81ab〕。

. 様相記号の使い道は広い。DURALではSの他にもう一つEXという様相記号も用意されている。EXはEXecuteの略であり，正のリテラルと同様に，Lisp関数を実行可能述語として書くのに用いる。ただし，正のリテラルとは異なり，Lisp関数の返す値がTのときに統一化が成功したと見なす。例を示そう。

例1" 様相記号EXを用いて書いた例1のプログラム

(1) ＋(Result ＊x 0)－EX(LESSP ＊x 10)

(2) ＋(Result ＊x 1)

D. 述語変数

DURALの構文では素論理式が(Add ＊x ＊y ＊z)のように書くようになっている。DEC－10 PROLOGではadd(x, y, z)と書く。この2つを比べると前者ではAddと＊x等が同列に並んでいるから，Addのところにも変数が書きたくなる。述語変数というのは，このような発想から生まれたものである。論理的に言えば，述語論理の範囲を少し越えることになる。しかし，実際の使い方としてはそれ程大袈裟なことはない。例えば，ゴール

－(＊p 2 3 5)＋(QUERY (＊p 2 3 5))

は，DURALの内部データベースに手続きか事実として登録された述語のリストを参照しながら，答を探す。

述語変数のこのような使い方は，データベースの問合せや，昔書いたプログラムを思い出したり，あるいは類似のプログラムを比較したりするのに役に立つ。また，プログラム・シンセシスにも活用されている[Goto 79]。。

E. 逆戻りの制御

DURALではDEC－10 PROLOGのカットシンボル(！)の機能を強化したLisp関数CUTによって，逆戻りの制御を行う。例えば，DURALではNotは次のように定義されている。

例4. Not の定義

＋S(Not ＊p)－＊p＋(CUT Not)＋T

＋S(Not ＊p)

CUTは引数で与えられる述語に対して，後で逆戻りが生じた場合，それ以上同じ述語名の手続きが無いかのように見せかける効果を果たす。IF－THEN－ELSEは次のようになる。

例5. IF－THEN－ELSEをCUTを用いて書く。

＋(IF ＊p THEN ＊q ELSE ＊r)－＊p＋(CUT IF)－ ＊q

＋(IF ＊p THEN ＊q ELSE ＊r)－＊r

もう少し，技巧的な例を挙げる。

例6. 知的な逆戻りの制御。

(1) ＋（ｐ１）－…－（ｐ２）－（ｐ３）－…　＋（CUT　ｐ３）－（ｐ４）…

(1)の例はｐ４に入る前にCUTを実行するので，ｐ４が失敗したときには，ｐ２へ直接逆戻りする。

(2) ＋（Ｑ１）－…＋（CUT　＊ｐ）－（Ｑ２）…

(2)の例はＱ２に入る前にCUTを実行し，＊ｐで与えられる述語名に対してCUTを実行する。このとき，その述語名は一つのホーン節に中に閉じている必要はなく，動的に探される。

PROLOG／KRのように，種々な制御機構を持つシステムもあるが，DURALではCUTしか提供していない。

F．双モード証明機構〔後藤８１〕

DURALは証明機構としてinput resolution〔Chang 73〕を基本としている。これは他のPROLOGと同じであるが，DURALではこの他にunit resolutionという証明機構でプログラムを実行することもできる。

理論的にいえば，input resolutionもunit resolution も証明機構としての力は等しい。すなわち，input resolutionで証明できるものはunit resolutionでも証明でき，その逆も正しい〔Loveland 79〕。いずれも一般の節に適用すると完全（complete）ではないが，相対ホーン節に対しては完全である。

このように，input resolutionとunit resolutionとは理論的に同値であるから，わざわざunit resolutionを組込むだけの値打ちはないように思える。しかし，同値であるのはゴール節の実行が成功した場合の話である。ゴール節が失敗する場合（論理的にいえば証明できない場合）には証明機構の違いによってプログラムの動作が大幅に異なってくる。DURALではinput，unitの双モード証明機構を利用することによって，プログラムのデバッグを行うことができる。例で説明しよう。

例7. 積木の世界

(1) ＋（On　a　b）

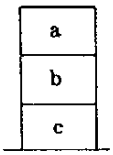
(2) ＋（On　b　c）

(3) ＋（Above　＊x　＊y）－（On　＊x　＊y）

(4) ＋（Above　＊x　＊y）－（Above　＊x　＊z）－（On　＊z　＊y）

プログラムの１行目と２行目はそれぞれ「ａがｂの上（On）にある」，「ｂがｃの上にあ

る」という事実を表わしている。また3行目と4行目はAboveという関係ををOnの推移的閉包（ transitive closure ）として定義している。



2つの証明機構の違いを見るために，具体的なゴール節を与えて動作を追跡してみよう。例えば，ゴール節として

－(Above a *v)

を与えると， input resolutionでは図4.1.4に示すような導出木をもとに*vがbであることを証明する。また， unit resolutionでは図4.1.5に示すような導出木をもとに*vがbであることを証明する。この場合，両モードとも同じ答を出す。

次にゴール

－(Above a d)

を考えてみる。この場合，dについてはプログラムで何もいっていないので，証明はできない。input resolutionでは次のような無限のゴールの系列が生成されて，プログラムは停止しなくなる。

given goal (5) －(Above a d)

(4)と(5)から (6) －(Above a *z)－(On *z d)

(4)と(6)から (7) －(Above a *z')－(On *z' *z)－(On *z d)

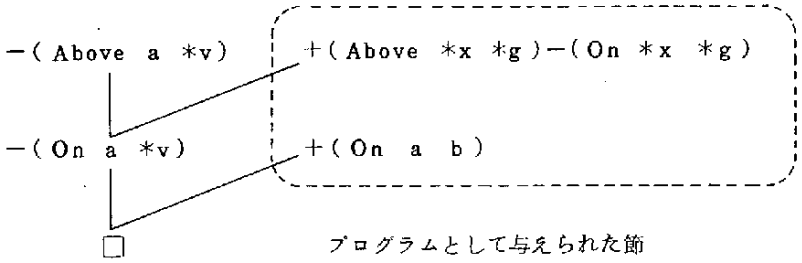
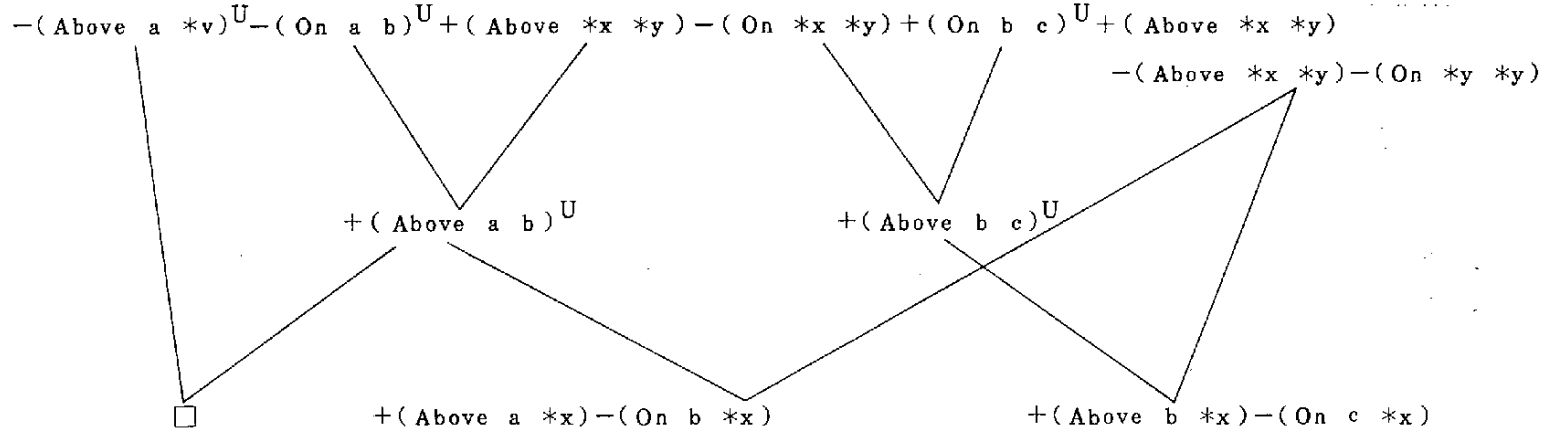


図4.1.4 －(Above a *v)の
input resolutionでの導出木



UはUnit 節を示す。

図4.1.5　unit resolutionによる導出木

一方，unit resolutionでは有限回の操作で証明不可能なことが分かる。unit resolutionでは証明の途中で生成される節の長さが必ず与えられた節よりも短くなるから，input resolutionのような爆発（explosion）が起らないからである。もちろん，unit resolutionでも節の長さが与えられた節と同じ場合にはループに入る。

DURALでの証明の効率化は次のようにしている。input resolutionの場合にはright−fixedという順序付け（ordering）の規則を設けて証明の効率を向上させている。これに対してunit resolutionの場合には，atom−orderingを採用することにより，ほぼ同等の順序付けを実現している。

最後に，例7のプログラムは次のように改良すると，input resolutionでも証明不可能という答えが得られる。

例7'　例7の改良

(3) ＋（Above ＊x ＊y）−（On ＊x ＊y）

(4) ＋（Above ＊x ＊y）−（On ＊x ＊z）−（Above ＊z ＊y）

G. 処 理 系

DURALの処理系は，DEC System−2020上に，TOPS−20オペレーティングシステムの下のMacLisp，InterLispで書かれている。処理系はLispで書かれているので，移植性が高く，国内の数か所で種々のLisp上に移植され，現在稼動している。

DURALはTSS環境での使用のために会話環境が充実している。会話環境の充実は，プログラミング言語を実用に供する時の重要な点である。DURALはLispの上に作成されているので，Lispが提供する会話環境は十分に活用することができる。

Lispレベルの会話環境としては，ブレイク・パッケージ，構造エディタ，スクリーン・エディタなどがある。

DURALレベルの会話環境としてはスクリーン・エディタ，証明過程のモニター，ステッパ，トレーサなどがある。また，内部データベースのファイルへの格納，ファイルからのロードもできる。〔奥乃句82〕DURALで特徴的な2つの機能について説明する。

a. スクリーン・エディタ

プログラム（節）はDURALの内部データベースに内部形式で格納されているので，Lispの構造エディタを用いて修正することができる。しかし，構造エディタを用いることは，ユーザに内部形式を意識させることになる。DURALでは，スクリーン・エディタを用いて相対ホーン節という外部形式で節の修正・削除ができる。したがって，ユーザは内部形式を意識する必要がない。

DURAL用スクリーン・エディタはEmacsであり，Lispの構造エディタとしてのコマンドも備えたエキスト・エディタである。Emacsへの入り方は3通りある。

(1) 直前に呼ばれたときと同じ画面を表示。

(2) 内部データベースのすべての節を画面に表示。

(3) 指定された述語名を持つ節だけを画面に表示。

EmacsからDURALへの戻り方には4通りある。

(1) カーソル位置からSTOPまでの節を定義または実行。

(2) カーソル位置からSTOPまでの節を削除。

(3) カーソル位置からSTOPまでの節で内部データベースの対応する節を置換。

(4) 何もせずに戻る。

DURALでは，Emacsを用いて内部データベースの管理が完全に行える。

b．証明過程のモニター

ゴール節の証明過程で未定義の述語に遭遇したときは，DURALではブレイク・パッケージが呼ばれる。ブレイク・パッケージで使えるコマンドには7種類あり，エディタを呼んだり，再帰的にDURALに入ることができる。図4.1.6に示したように，単なる入力誤りのときには，自動スペル修正機能が便利である。

```
25-Feb-82 08:50:09
Hi, DURAL user.
2_load(dural.com)
DURAL.COM.7
3_load(dural-emacs.com)
DURAL-EMACS.COM.5
4_(RESTORE-DATABASE) ←データベースのファイルからのロード(内部形式で)
(Database (SOUTHERN-HEMISPHERE EASTERN-HEMISPHERE NORTHERN-HEMISPHERE
...) is restored from DURALDATABASE)
5_(SETQ DEBUGMODE T) ←証明過程のモニターON
(DEBUGMODE reset)
T
6_(DURAL)  ↙述語名が間違っている。
DRL>-(REGIN *X FAR-EAST)-(LANDLOCKED *X) ←問合せ
(REGIN is not defined) ←Breakに入るときのメッセージ
Break>H ←メニューの出力コマンド

     C[ONTINUE]          Continue refutation
     D[URAL]             Enter recursive Dural
     S[PELL]             Correct spelling automatically
     R[PLACE] pred       Correct spelling to pred
     E[VAL] sexpr        Eval sexpr
     A[BORT]             Abort refutation
     H[ELP]              Print this text

Break>S ←自動スペル修正コマンド
=REGION ←修正された述語名
(MONGOLIA) ←問合せの答
DRL>
DRL>L(DURAL.EMACS)
     ↖ DURALの中からLispが呼べる。
DRL>OK                    DURAL用スクリーン・エディタを呼んでいる。
NIL  ↖
7_     DURALから出る。
```

図4.1.6　証明過程のモニターの例〔奥乃82〕

—参考文献—

1) 〔Chang 73〕 C—L, Chang and R.C—T.Lee ; Symbolic Logic and Mechanical Theorem Proving, Academic Press, 1973.

2) 〔Goto 79〕 S. Goto, Program Synthesis from Natural Deduction Proofs, Proc. of 6 th IJCAI, pp.339-341, 1979.

3) 〔Goto 80〕 S. Goto, DURAL : An Extended Prolog Language, 京都大学数理解析研究所講究録№396, pp170-189, 1980.

4) 〔後藤 80〕 後藤滋樹, 述語型プログラミングのdebugについて, 「作譜工程の評価・改良・自動化」シンポジウム報告集, pp.48-53, 情報処理学会, 1980.

5) 〔後藤 81〕 後藤滋樹, 述語型言語DURALにおける双モード証明機構, 情報処理学会第22回全国大会予稿集, 1B—5, 1981.

6) 〔Loveland 79〕 D.W. Loveland, Automated Theorem Proving : A Logical Basis, North-Holland, 1978.

7) 〔松本 71〕 松本和夫, 数理論理学, 増補版, 共立出版, 1971.

8) 〔Nakashima 81〕 H. Nakashima, PROLOG／KR USER'S MANUAL for version C-5, 東京大学, 和田研究室, 1981.

9) 〔奥乃 81a〕 奥乃博, 演繹機能を有したデータベースの一構成法, 記号処理研究会, 16—1, 情報処理学会, 1981.

10) 〔奥乃 81b〕 奥乃博, 演繹データベースにおける更新演算について, 情報処理学会第23回全国大会予稿集, 3G—10, 1981.

11) 〔奥乃 82〕 奥乃博, 述語型言語の支援ユーティリティについて, 情報処理学会第24回全国大会予稿集, 4P—7, 1982.

12) 〔Pereira 78〕 L. M. Pereira, F.C.N.Pereira, and D.H.D.Warren, User's Guide to DEC System-10 PROLOG, University of Edinburgh, 1978.

4.1.3. そ の 他

A. KUROLOG (Knowledge Utilization and Representation Oriented Logic Programming Language)

KUROLOGは現在試作が進められている論理関数型言語の一種である。

KUROLOGは実用性に主眼を置き, 通常のPROLOG言語に比べて, 実行制御に重きを置い

ている。

a. 制　御

KUROLOGにはPROLOGのカット・オペレータ（cut operator）を用いない。代わりに次のような制御関数が用意されている。

① sequence（f_1, f_2, …）

f_iを順に実行する。f_iの間での逆戻りはない。

② try-all（f_1, f_2, …）

f_iをすべて真になるまで実行する。実行順序は問わない。

③ try-one（f_1, f_2, …）

f_iのどれか一つが真となるまでf_1…, f_nを実行する。

④ select（f_1；f_2；…）wohen（t_1；t_2；…）

互いに素なテストt_iを調べて，その内成立したt_jに対応したf_jを実行する。

⑤ iterate（f_1；f_2；…f_n）while C

条件Cの成立している間，f_1, f_2, …f_nを順次実行する。

b. 引数処理

KUROLOGの引数処理は基本的には機能クラスの枠組の中で処理される。すなわち，各述語の属する機能クラス中に，引数をどう処理すべきかの処方が記述されている。

したがってKUROLOGでは単純なパターン・マッチングだけが引数処理のすべてではない。引数の評価（特に数値演算式の場合），引数のデータ型の合致，さらにパターン・マッチングの制御などが可能である。

c. 実行環境

KUROLOGでは実行環境もレッキとしたデータとしてユーザに供給される。KUROLOGではこの実行環境を短期記憶（Short Term Memovy）と呼んでいる。

短期記憶には実行開始時の情報，実行途中の情報，実行終了後の述語の「値」，各述語間での交信の場となるブラックボードが含まれている。

実行途中の情報は，述語の実行を途中で停止して，適当な条件下で再開することのできる機能，すなわちソフトウェア的な意味での並列処理を実現する。

述語の「値」は，KUROLOGにおいては関数呼び出し，もしくはメッセージ処理な実行機能を実現するベースとなる。

ブラックボードは，プロセスの協調的問題解決などにも有効である。この述語に関係する他の述語が，各種のデータ，指令，情報などをブラックボードに書き込む。このブラックボー

ドの内容を見ては，この述語の実行内容が変更されることがある。

c. 議　論

KUROLOGの基本的な考え方は，実行制御を陽に開放し，副作用を短期記憶の形で陽に取り出そうというもので，いわゆるPROLOGの行き方とは矛盾する。

実用的な観点では，制御と副作用の開放が不可欠だというのがKUROLOG の基本主張となる。並列制御の部分には実際はバックトラックを用いるが，概念上はあくまで実行順序に関係なく成立すべきだという考えである。この部分はKUROLOGを支えるハードウェアの問題である。

引数の評価などの関数的な部分はできればもう少しスマートな形にしたい。メッセージで処理の形とか関数型プログラミングの形とかを取り入れた方が良かった。

ー参考文献ー

1) Kurokawa, T., "Comments on Prolog from the functional points of view", 情報処理学会，記号処理研究委員会，(Jan. 1982)

2) Kurokawa, T., "Logic Programming for Knowledge Utilization and Representation", Prolog Conference, (Mar.1982)

4.2. プログラム例

4.2.1. エイト・クイーン

A. 問題記述

8個のクイーンを互いにとることが，できないようにチェス盤上に配置する。その配置を解として求める問題である。クイーンを1つずつ配置してゆくが，その際に，縦・横・斜めの8方向をチェックし，どの方向にも，すでにクイーンが置かれていない場合に，その位置にクイーンを置くという手順を繰り返し適用し，解を得る。

B. 述語の説明

eight_queen(Q)

解の配置がQにunifyされて戻る。

queens(X, Y, Z)

X：クイーンが未配置な行(列)のリスト。

Y：queensが呼ばれた時点での盤上のクイーンの配置を示すリスト。

Z：解の配置を示すリスト。

その時点までの盤上のクイーン配置から，縦・横方向に空いている位置を選択し，その位

置が斜め方向にも空いていればその位置を部分解(Y)に入れ,再帰的に,残りのクイーンの配置を決定する。斜め方向が,それまでのクイーンの配置により占められている場合は,バックトラックにより別の位置を選択し,同様の処理を行なう。

queen([], Y, Y).

8個のクイーン全ての配置が定まった(Y)ので,eight-queen(Q)の変数Qに解のリストをunifyする。

select(U, X, Y).

クイーンが未配置な位置を求める。

safe(U, Y, N).

盤上のある位置の斜め方向に,すでに配置されたクイーンがないかをチェックする。

C. プログラム・ソースリスト

エジンバラ版Prolog及びMaclispによるインプリメント例を以下に示す。

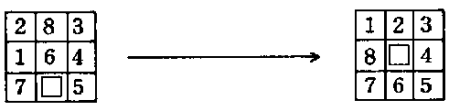
```
eight_queen(Q) :- queens([1,2,3,4,5,6,7,8], [], Q).

queens([], Y, Y).
queens(X, Y, Z) :- select(U, X, V),safe(U, Y, 1)
                  ,queens(V, [U|Y], Z).

select(X, [X|Y], Y).
select(U, [X|Y], [X|V]) :- select(U, Y, V).

safe(U, [], _) :-!.
safe(U, [P|Q], N) :- nodiag(U, P, N)
                    ,M is N+1,safe(U, Q, M).

nodiag(U, P, N) :- T1 is P+N,T2 is P-N
                  ,not(U=T1),not(U=T2).

not(P) :- P,!,fail.
not(_).
```

```
(DEFUN QUEEN (SIZE)
       (PROG (N M BOARD)
             (SETQ N 1)
             LOOP-N
             (SETQ M 1)
             LOOP-M
             (COND ((CONFLICT N M BOARD) (GO UN-DO-M)))
             (SETQ BOARD (CONS (LIST N M) BOARD))
             (COND ((GREATERP (SETQ N (ADD1 N)) SIZE)
                    (PRINT (REVERSE BOARD))))
             (GO LOOP-N)
             UN-DO-N
             (COND ((NULL BOARD) (RETURN 'FINISHED))
                   (T (SETQ M (CADAR BOARD)
                            N (CAAR BOARD)
                            BOARD (CDR BOARD))))
             UN-DO-M
             (COND ((GREATERP (SETQ M (ADD1 M)) SIZE)
                    (GO UN-DO-N))
                   (T (GO LOOP-M)))))

(DEFUN CONFLICT (N M BOARD)
       (COND ((NULL BOARD) NIL)
             ((OR (THREAT N M (CAAR BOARD) (CADAR BOARD))
                  (CONFLICT N M (CDR BOARD))))))

(DEFUN THREAT (I J A B)
       (OR (EQUAL I A)
           (EQUAL J B)
           (EQUAL (DIFFERENCE I J) (DIFFERENCE A B))
           (EQUAL (PLUS I J) (PLUS A B))))
```

4.2.2. 8－パズル

A. 問題記述

3×3の枠の中に収められた，8個の駒と1個の空白があり，空白と，それと隣接する駒を入れ換えることができる。8個の駒と1個の空白の初期状態が与えられた時，駒と空白の入れ換え操作により，定められた最終状態へと変換するシーケンスを求める。



初期状態　　最終状態

図4.2.1　8－パズル問題

各述語の説明

```
puzzle ( I, G, SOL ) :- fhat ( I, G, 0, FI ),
                        puz1 ( [ [ I, FI, [], 0 ] ], G, [], SOL ) .
```

ヘッド　puzzle (I, G, $\overline{\text{SOL}}$)

初期状態I，最終状態Gを入力リストとし，IからGに至る状態の系列の逆順リスト$\overline{\text{SOL}}$を出力リストとして返す述語。

ボディのゴール1　fhat (I, G, 0, FI) .

初期状態I，最終状態G，探索木の深さのレベル0を入力として与え，ヒューリスティックな評価関数$\hat{f}$(I)の値FIを出力として求めてくる述語。

ボディのゴール2　puz 1 ([[I, FI, [], 0]], G, [], $\overline{\text{SOL}}$)

解の候補集合$\overline{\text{OPEN}}$の初期値として，初期状態I，その評価関数値FI，その先祖状態リスト[]，その深さのレベル0からなる4項リスト[I, FI, [], 0]を設定する。第2引数は最終状態Gをそのまま受渡し，第3引数は解の処理済状態集合CLOSEDの値を空集合として初期設定している。第4引数は解答SOLを出力として返してもらうための変数。

以上より明らかなように，この節(clause)は8－パズルの初期設定する述語からなっている。

```
puz 1 ( [ ], _, _, _ ) :- !, fail .
```

もし集合$\overline{\text{OPEN}}$が空集合ならば，解は見つからなかった(失敗した)ので，述語puz 1の値を偽(fail)として強制終了(!)しなさい。

```
puz 1 ( O, G, Z, SOL ) :- fmin ( O, [ S, V, P, L ] ),
                          union ( C, [ [ S, V, P, L ] ], C2 ),
                          difference ( O, [ [ S, V, P, L ] ], O2 ),
                          concatenate ( [ S ], P, CH2 ),
                          puz 2 ( O2, G, C2, CH2, SOL, S, L ) .
```

ヘッド　puz 1 ($\overline{O}$, G, C, $\overline{SOL}$)

解の候補集合$\overline{O}$，最終状態G，処理済の状態集合Cを入力し，解答$\overline{SOL}$を出力する述語。

ボディのゴール1　fmin ($\overline{O}$, 〔 S, V, P, L 〕)

解の候補集合$\overline{O}$を入力し，評価関数値が最小である状態を計算し，その状態S，評価関数値V，その先祖状態リストP，その深さのレベルLからなる4項リスト〔 S, V, P, L 〕を出力して返す述語。この出力リストは代案が複数個ありうるので，一般にバックトラッキングの可能性あり。

ボディのゴール2　union (C, 〔〔 S, V, P, L 〕〕, C2)

処理済状態集合Cと，評価関数値が最小の状態Sに対応するリスト〔 S, V, P, L 〕のみからなる集合〔〔 S, V, P, L 〕〕を入力し，それらの和集合C2として出力する述語。

ボディのゴール3　difference ($\overline{O}$, 〔〔 S, V, P, L 〕〕, $\overline{O}$2)

解の候補集合$\overline{O}$と集合〔〔 S, V, P, L 〕〕の差集合$\overline{O}$2を生成する述語。

ボディのゴール4　concatenate (〔S〕, P, CH2)

最小の評価値をもつカレント状態Sからなる1項リスト〔S〕とSの先祖頂点リストPを連接し，初期状態からカレント状態に至る系列の逆順リストCH2を値として返す述語。

ボディのゴール5　puz 2 ($\overline{O}$2, G, C2, CH2, $\overline{SOL}$, S, L)

カレント状態S，そのレベルL，Sを除く解の候補集合$\overline{O}$2，Sを含む処理済状態集合C2，Sに至るチェインCH2，および最終状態Gを入力とし，解答$\overline{SOL}$を出力とする述語。

```
puz 2 ( —, G, —, CH, CH, G, — )  :-!.
```

もしカレント状態Sが最終状態Gと一致すれば，解答$\overline{SOL}$の値をチェインCHと一致させ，解は見つかった（成功した）として終了する述語。

```
puz 2 ( Ō, G, C, CH, SŌL, S, L ) :- expand ( S, W, Ō, C ),
                                    L1 is L+1,
                                    feval ( W, CH, L1, G, FW ),
                                    union ( Ō, FW, Ō2 ),
                                    puz 1 ( Ō2, G, C, SŌL ).
```

ヘッドpuz2(Ō, G, C, CH, SŌL, S, L)

カレント状態S，そのレベルL，Sを除く解の候補集合Ō，Sを含む処理済状態集合C，Sに至るチェインCH，および最終状態Gを入力とし，解答SŌLを出力とする述語。

ボディのゴール1　expand (S, W, Ō, C)

カレント状態S，候補集合Ō，処理済状態集合Cを入力とし，Sに対する拡張状態集合Wを生成する述語。ここに拡張状態集合とは，Sに対して可能な拡張状態集合から，既に探索済のŌとCの和集合を差し引いた集合のこととする。これによりループに陥るのを避けている。

ボディのゴール2　L1 is L+1

拡張された状態のレベルL1は，カレント状態のレベルLに1を足したものである。

ボディのゴール3　feval (W, CH, L1, G, FW)

拡張状態集合Wに属する状態Wのそれぞれに対して，Wのレベル L1と最終状態Gをもとにヒューリスティックな評価関数値$\hat{f}$(W)を求め，W，$\hat{f}$(W)，Wの親状態のチェインCH（すなわちWの先祖状態リスト），WのレベルL1からなる4項リストの集合FWを出力として返す述語。

ボディのゴール4　union (Ō, FW, Ō2)

解の候補集合Ōと拡張状態集合FWの和集合を，新たに解の候補集合Ō2として生成する述語。

ボディのゴール5　puz 1 (Ō2, G, C, SŌL)

解の候補集合Ō2，既に解でないと分っている処理済状態集合C，および最終状態Gを入力し，解SŌLを求めてくる述語。

```
fhat ( X, G, L, FX ) :- ghat ( X, L, GX ),
                        what ( X, G, WX ),
                        FX is GX+WX .
```

ヘッド　fhat (X, G, L, FX)

レベルLの状態Xを与えて最終状態Gに対する評価関数の値FXを出力として返してくれる述語。

ボディのゴール1　ghat (X, L, GX)

状態X，そのレベルLを入力とし，初期状態からその状態Xに至る探索木内の経路の長さを評価関数値GXとして返す述語。

ボディのゴール2　what (X, G, WX)

状態X，最終状態Gを入力とし，正しく置かれていない駒とスペースの数を評価関数値WXとして出力する述語。

ボディのゴール3　FX is GX+WX

ヒューリスティックな評価関数値FXが，GXとWXの和であることを規制する述語，すなわち，$\hat{f}(x)=\hat{g}(x)+\hat{w}(x)$を意味する。

```
ghat ( _, L, L ) :- ! .
```

状態のいかんにかかわらず，探索木の深さのレベルLを，評価関数値として返す述語。この述語は，毎回，1度しか呼ぶ必要がないので，!を入れた。

```
what ( [X, ··L], [X, ··M], WX) :- !, what ( L, M, WX) .
```

第1引数と第2引数のリストのcar部同志が等しい場合の評価関数値WXとは，それらリストのcdr部同志の評価関数値WXである。

```
what ( [X, ··L], [Y, ··M], WX) :- what ( L, M, WX2), WX is
                                                    WX2+1 .
```

第1引数と第2引数のリストのcar部同志が異なる場合の評価関数の値がWXであるためには，それらリストのcar部同志の評価関数値がWX2であり，しかもWXがWX2に1を足したものである必要がある。

```
what ( [ ], [ ], 0 ).
```

空リスト同志の評価関数値は0である。

```
feval ( [ ], —, —, —, [ ] ) :— ! .
```

評価される集合Wが空集合となれば，評価値集合FWも空集合として値を返して終了する。

```
feval ( [X, .,W], CH, L, G, FW ) :— feval (W, CH, L, G, FW2),
                                    fhat ( X, G, L, FX ),
                                    concatenate ([[ X, FX, CH,
                                    L ] ], FW2, FW) .
```

ヘッド　feval ([X, . , W], CH, L, G, FW)

拡張状態集合[X, ‥W]に属する状態Xに対して，Xのレベル L と最終状態Gをもとに評価値 $\hat{f}(x)$ を求め，[x, $\hat{f}(x)$, CH, L] (CHは，Xの先祖状態リスト) からなる集合FWを値として返す述語。

ボディのゴール1　feval (W, CH, L, G, FW2)

拡張状態集合[X, . . W]のcdr部Wに属する状態xに対して，前述のリスト[x, $\hat{f}(x)$, CH, L]の集合FW2を値として返す述語である。

ボディのゴール2　fhat (X, G, L, FX)

レベルLの状態Xを与えて，最終状態Gに対する評価値FXを出力として計算してくれる述語。

ボディのゴール4　concatenate ([[X, FX, CH, L]], FW2, FW]

リスト[[X, FX, CH, L]]とリストFW2を連接した結果をFWとして返す述語。

C. プログラム・ソース・リスト

エジンバラ版Prologによる，８－パズルのプログラム・リストを以下に挙げる。

```
ex1([X1,X2,X3,X4,X5,X6,0,X7,X8]
    ,[[X1,X2,X3,0,X5,X6,X4,X7,X8]]).
ex1([X1,X2,X3,X4,X5,X6,X7,0,X8]
    ,[[X1,X2,X3,X4,0,X6,X7,X5,X8]]).
ex1([X1,X2,X3,X4,X5,X6,X7,X8,0]
    ,[[X1,X2,X3,X4,X5,0,X7,X8,X6]]).

ex2([0,X1,X2,X3,X4,X5,X6,X7,X8]
    ,[[X1,0,X2,X3,X4,X5,X6,X7,X8]]).
ex2([X1,0,X2,X3,X4,X5,X6,X7,X8]
    ,[[X1,X2,0,X3,X4,X5,X6,X7,X8]]).
ex2([_,_,0,_,_,_,_,_,_],[]).
ex2([X1,X2,X3,0,X4,X5,X6,X7,X8]
    ,[[X1,X2,X3,X4,0,X5,X6,X7,X8]]).
ex2([X1,X2,X3,X4,0,X5,X6,X7,X8]
    ,[[X1,X2,X3,X4,X5,0,X6,X7,X8]]).
ex2([_,_,_,_,_,0,_,_,_],[]).
ex2([X1,X2,X3,X4,X5,X6,0,X7,X8]
    ,[[X1,X2,X3,X4,X5,X6,X7,0,X8]]).
ex2([X1,X2,X3,X4,X5,X6,X7,0,X8]
    ,[[X1,X2,X3,X4,X5,X6,X7,X8,0]]).
ex2([_,_,_,_,_,_,_,_,0],[]).

ex3([0,X1,X2,X3,X4,X5,X6,X7,X8]
    ,[[X3,X1,X2,0,X4,X5,X6,X7,X8]]).
ex3([X1,0,X2,X3,X4,X5,X6,X7,X8]
    ,[[X1,X4,X2,X3,0,X5,X6,X7,X8]]).
ex3([X1,X2,0,X3,X4,X5,X6,X7,X8]
    ,[[X1,X2,X5,X3,X4,0,X6,X7,X8]]).
ex3([X1,X2,X3,0,X4,X5,X6,X7,X8]
    ,[[X1,X2,X3,X6,X4,X5,0,X7,X8]]).
ex3([X1,X2,X3,X4,0,X5,X6,X7,X8]
    ,[[X1,X2,X3,X4,X7,X5,X6,0,X8]]).
ex3([X1,X2,X3,X4,X5,0,X6,X7,X8]
    ,[[X1,X2,X3,X4,X5,X8,X6,X7,0]]).
ex3([_,_,_,_,_,_,0,_,_],[]).
ex3([_,_,_,_,_,_,_,0,_],[]).
ex3([_,_,_,_,_,_,_,_,0],[]).

ex4([0,_,_,_,_,_,_,_,_],[]).
ex4([X1,0,X2,X3,X4,X5,X6,X7,X8]
    ,[[0,X1,X2,X3,X4,X5,X6,X7,X8]]).
ex4([X1,X2,0,X3,X4,X5,X6,X7,X8]
    ,[[X1,0,X2,X3,X4,X5,X6,X7,X8]]).
ex4([_,_,_,0,_,_,_,_,_],[]).
ex4([X1,X2,X3,X4,0,X5,X6,X7,X8]
    ,[[X1,X2,X3,0,X4,X5,X6,X7,X8]]).
ex4([X1,X2,X3,X4,X5,0,X6,X7,X8]
    ,[[X1,X2,X3,X4,0,X5,X6,X7,X8]]).
ex4([_,_,_,_,_,_,0,_,_],[]).
ex4([X1,X2,X3,X4,X5,X6,X7,0,X8]
    ,[[X1,X2,X3,X4,X5,X6,0,X7,X8]]).
ex4([X1,X2,X3,X4,X5,X6,X7,X8,0]
    ,[[X1,X2,X3,X4,X5,X6,X7,0,X8]]).

puzzle(I,G,SOL) :- fnat(I,G,0,F)
                  ,puzi([[I,F],[],0],G,[],SOL).

puzi([],_,_,_) :- !, fail.
puzi(O,G,C,SOL) :- fmin(O,[S,V,P,L])
```

```
member(X,[X|L]).
member(X,[Y|L]) :- member(X,L).

concatenate([],L,L).
concatenate([X|L1],L2,[X|L3]) :-concatenate(L1,L2,L3).

remove([],Y,[]).
remove([Y|L],Y,Z) :- !, remove(L,Y,Z).
remove([X|L],Y,Z) :- remove(L,Y,Z1)
                    ,concatenate([X],Z1,Z).

diff2(X,[],X).
diff2(X,[[Y,V,P,L]|LL],Z) :- remove(X,Y,ZZ)
                            ,diff2(ZZ,LL,Z).

difference(X,[],X).
difference(X,[Y|L],Z) :- remove(X,Y,ZZ)
                        ,difference(ZZ,L,Z).

union([],X,X).
union([X|L],Y,Z) :- member(X,Y),!,union(L,Y,Z).
union([X|L],Y,[X|Z]) :- union(L,Y,Z).

fmin([[S,V,P,L]],[S,V,P,L]) :- !.
fmin([[S,V,P,L]|R],[SS,VV,PP,LL]) :- fmin(R,[SS,VV,PP,LL])
                                    ,V >= VV,!.
fmin([[S,V,P,L]|R],[S,V,P,L]) :- fmin(R,[SS,VV,PP,LL])
                               ,V < VV,!.

feval([],_,_,_,[]) :- !.
feval([X|W],CH,L,G,FW) :- feval(W,CH,L,G,FW2)
                        ,fhat(X,G,L,FX)
                        ,concatenate([[X,FX,CH,L]],FW2,FW).

fhat(X,G,L,FX) :- ghat(X,L,GX),what(X,G,WX),FX is GX + WX.

ghat(_,L,L) :- !.

what([X|L],[X|W],WX) :- !,what(L,W,WX).
what([X|L],[Y|W],WX) :- what(L,W,WX2),WX is WX2 + 1.
what([],[],W).

expand(S,W,O,C) :- ex1(S,S1),ex2(S,S2)
                  ,ex3(S,S3),ex4(S,S4)
                  ,union(S1,S2,S12)
                  ,union(S12,S3,S123)
                  ,union(S123,S4,SS)
                  ,diff2(SS,O,SS1)
                  ,diff2(SS1,C,W).

ex1([W,_,_,_,_,_,_,_,_],[]).
ex1([_,W,_,_,_,_,_,_,_],[]).
ex1([_,_,W,_,_,_,_,_,_],[]).
ex1([X1,X2,X3,W,X4,X5,X6,X7,X8]
    ,[[W,X2,X3,X1,X4,X5,X6,X7,X8]]).
ex1([X1,X2,X3,X4,W,X5,X6,X7,X8]
    ,[[X1,W,X3,X4,X2,X5,X6,X7,X8]]).
ex1([X1,X2,X3,X4,X5,W,X6,X7,X8]
    ,[[X1,X2,W,X4,X5,X3,X6,X7,X8]]).

                  ,union(C,[[S,V,P,L]],C2)
                  ,difference(O,[[S,V,P,L]],O2)
                  ,concatenate([S],P,CH2)
                  ,puz2(O2,G,C2,CH2,SOL,S,L).

puz2(_,G,_,CH,CH,G,_) :- !.
puz2(O,G,C,CH,SOL,S,L) :- expand(S,W,O,C)
                        ,L1 is L + 1
                        ,feval(W,CH,L1,G,FW)
                        ,union(O,FW,O2)
                        ,puz1(O2,S,C,SOL).
```

4.2.3 Wangのアルゴリズム

A. 問題記述

与えられた命題論理式の妥当性をチェックする。

～（否定），&（and），！（or），=>（条件），<=>（同値）なる記号を割り当てる。

*Wangのアルゴリズム

○&○&○=>○！○！○……（I）が妥当か？

○：wff（well formed formula）

(i)左辺・右辺で同じwffがあれば妥当

○&○&（P=>q）=>○！（P=>q）！○

(ii)左辺・右辺で同じwffがなく，wffが全てafomicであれば妥当ではない。

P&q=>r！s

wffがnon-atomicであれば，以下のruleに従って分解し，(i)，(ii)が成立するかを調べる。

－rule－

○&～A=>○ → ○=>A！○ 1a

○&（A&B）=>○ → ○&A&B=>○ 1b

○&（A！B）=>○ → 〔○&A=>○〕and〔○&B=>○〕 1c

○&（A=>B）=>○ → 〔○=>A！○〕and〔○&B=>○〕 1d

○&（A<=>B）=>○ → 〔○&A&B=>○〕and〔○=>A！B！○〕 1e

○=>～A！○→A&○=>○ 2a

○=>（A=>B）！○ → A&○=>B！○ 2b

○=>（A！B）！○ → ○=>A！B！○ 2c

○=>（A&B）！○ → 〔○=>A！○〕and〔○=>B！○〕 2d

○=>（A<=>B）！○ → 〔A&○=>B！○〕and〔B&○=>A！○〕 2e

B. 述語の説明

wang（X，Y，Z，U，LVL）

X：(I)式の左辺のwffのうちatomicなもののリスト

Y：(I)式の左辺のwffのうちnon-atomicなもののリスト

Z：(I)式の右辺のwffのうちatomicなもののリスト

U：(I)式の右辺のwffのうちnon-atomicなもののリスト

LVL：splitting rule（1c，1d，1e，2d，2e）を適用した回数。

red（L，X，Y，Z，U，LVL）

X，Y，Z，U，LVLはwangと同一。

L：〔〔wff1，rℓ1〕，〔wff2，rℓ2〕……〕なる形のリスト。

(a) rℓiが"r"でwffiがatomicならば，wffiをzにつなぎ，

(b) rℓiが"r"でwffiがnon-atomicならば，wffiをUにつなぎ，

(c) rℓiが"ℓ"でwffiがatomicならば，wffiをXにつなぎ，

(d) rℓiが"ℓ"でwffiがnon-atomicならば，wffiをYにつなぎ，

wangを実行する。

C．プログラム・リスト

エジンバラ版Prolog及びMaclispによるプログラム・リストを以下に挙げる。

```
:- op(150,yfx,[<=>]).
:- op(140,yfx,[=>]).
:- op(130,yfx,[!]).
:- op(120,yfx,[&]).
:- op(110,fx,[~]).


prove(WFF) :- red([[WFF,r]],[],[],[],[],0).


wang(X,Y,Z,U,LVL) :-
        ( notdisjoint(X,Z) ; notdisjoint(Y,U) ),
        N is LVL*2, tab(N), write('IS VALID'), nl.

wang(X,[~A,..Y],Z,U,LVL) :- red([[A,r]],X,Y,Z,U,LVL).
wang(X,[A&B,..Y],Z,U,LVL) :- red([[A,l],[B,l]],X,Y,Z,U,LVL).
wang(X,[A!B,..Y],Z,U,LVL) :-
        LVL1 is LVL+1,
        red([[A,l]],X,Y,Z,U,LVL1), red([[B,l]],X,Y,Z,U,LVL1).
wang(X,[A=>B,..Y],Z,U,LVL) :-
        LVL1 is LVL+1,
        red([[A,r]],X,Y,Z,U,LVL1), red([[B,l]],X,Y,Z,U,LVL1).
wang(X,[A<=>B,..Y],Z,U,LVL) :-
        LVL1 is LVL+1,
        red([[A,l],[B,l]],X,Y,Z,U,LVL1), red([[A,r],[B,r]],X,Y,Z,U,LVL1).

wang(X,[],Z,[~A,..U],LVL) :- red([[A,l]],X,[],Z,U,LVL).
wang(X,[],Z,[A=>B,..U],LVL) :- red([[A,l],[B,r]],X,[],Z,U,LVL).
wang(X,[],Z,[A!B,..U],LVL) :- red([[A,r],[B,r]],X,[],Z,U,LVL).
wang(X,[],Z,[A&B,..U],LVL) :-
        LVL1 is LVL+1,
        red([[A,r]],X,[],Z,U,LVL1), red([[B,r]],X,[],Z,U,LVL1).
wang(X,[],Z,[A<=>B,..U],LVL) :-
        LVL1 is LVL+1,
        red([[A,l],[B,r]],X,[],Z,U,LVL1), red([[A,r],[B,l]],X,[],Z,U,LVL1).

wang(X,[],Z,[],LVL) :-
        N is LVL*2, tab(N), write('NOT VALID'), nl, fail.
```

```
red([],X,Y,Z,U,LVL) :-
        prinline(X,Y,Z,U,LVL), wang(X,Y,Z,U,LVL).
red([[F,r],..Q],X,Y,Z,U,LVL) :-
        atom(F) -> red(Q,X,Y,[F,..Z],U,LVL) ; red(Q,X,Y,Z,[F,..U],LVL).
red([[F,l],..Q],X,Y,Z,U,LVL) :-
        atom(F) -> red(Q,[F,..X],Y,Z,U,LVL) ; red(Q,X,[F,..Y],Z,U,LVL).


notdisjoint([X,..Q],Y) :- member(X,Y).
notdisjoint([X,..Q],Y) :- notdisjoint(Q,Y).


member(X,[X,..._]).
member(X,[_,..Y]) :- member(X,Y).


prinline(X,Y,Z,U,LVL) :-
        N is LVL*2, tab(N),
        append(X,Y,L), writelem(L), writelem(['===>']),
        append(Z,U,R), writelem(R), nl.


writelem([]).
writelem([X]) :- write(X), tab(1).
writelem([X,Y,..Z]) :- write(X), write(','), writelem([Y,..Z]).


append([],L,L).
append([X,..L1],L2,[X,..L3]) :- append(L1,L2,L3).


...
| ?- prove( ~(p&q)<=>(~p)!(~q) ).
===> ~(p&q)<=> ~p! ~q
  ~(p&q) ===> ~p! ~q
  ===> p&q,~p! ~q
    ===> p,~p! ~q
    ===> p,~q,~p
    q ===> p,~p
    p,q ===> p
    IS VALID
    ===> q,~p! ~q
    ===> q,~q,~p
    q ===> q,~p
    IS VALID
  ~p! ~q ===> ~(p&q)
    ~p ===> ~(p&q)
    ===> p,~(p&q)
    p&q ===> p
    q,p ===> p
    IS VALID
    ~q ===> ~(p&q)
    ===> q,~(p&q)
    p&q ===> q
    q,p ===> q
    IS VALID

yes
| ?- prove( ~(p=>q)<=> ~p!q ).
===> ~(p=>q)<=> ~p!q
  ~(p=>q) ===> ~p!q
  ===> p=>q,~p!q
  p ===> q,~p!q
  p ===> q,q,~p
  p,p ===> q,q
  NOT VALID

no
```

```
;
;-------------------------------------------------------------
;
(DEFUN PROVE FEXPR (WFF)
       (TERPRI)
       (WANG (ADDR (CAR WFF) (LINE 3 NIL NIL NIL NIL))))
;
;-------------------------------------------------------------
;
(DEFUN WANG (L)
       (PROG2
        (PRINLINE L)
        (COND ((PRIM-NO L) NIL)
              ((PRIM-YES L) T)
              ((NULL (LF L))
               (REDR (CAR (RF L)) (RF L (CDR (RF L)))))
              (T (REDL (CAR (LF L)) (LF L (CDR (LF L))))))))
;
;-------------------------------------------------------------
;
(DEFUN PRIM-NO (L)
       (COND ((AND (NULL (LF L))
                   (NULL (RF L))
                   (DISJOINT (LA L) (RA L)))
              (PRIN3 % (LVL L) NOT VALID <>))
             (T NIL)))
;
;-------------------------------------------------------------
;
(DEFUN PRIM-YES (L)
   (COND ((NOT (AND (DISJOINT (LA L) (RA L))
                    (DISJOINT (LF L) (RF L))))
          (PRIN3 % (LVL L) IS VALID <>))
         (T NIL)))
;
;-------------------------------------------------------------
;
(DEFUN DISJOINT (LA LB)
   (COND ((NULL LA) T)
         ((MEMBER (CAR LA) LB) NIL)
         (T (DISJOINT (CDR LA) LB))))
;
;-------------------------------------------------------------
;
(DEFUN REDL (WFF L)
   (COND ((EQ (CAR WFF) '/^)
          (WANG (ADDR (CADR WFF) L)))
         ((EQ (CADR WFF) '/&)
          (WANG (ADDL (CAR WFF) (ADDL (CADDR WFF) L))))
         ((EQ (CADR WFF) '/!)
          (AND (WANG (ADDL (CAR WFF) (BUMP L)))
               (WANG (ADDL (CADDR WFF) (BUMP L)))))
         ((EQ (CADR WFF) '=/>)
          (AND (WANG (ADDL (CADDR WFF) (BUMP L)))
               (WANG (ADDR (CAR WFF) (BUMP L)))))
         ((EQ (CADR WFF) '/<=/>)
          (AND (WANG (ADDL (CAR WFF) (ADDL (CADDR WFF) (BUMP L))))
               (WANG (ADDR (CAR WFF) (ADDR (CADDR WFF) (BUMP L))))))))
;
;-------------------------------------------------------------
;
(DEFUN REDR (WFF L)
   (COND ((EQ (CAR WFF) '/^)
          (WANG (ADDL (CADR WFF) L)))
         ((EQ (CADR WFF) '/=/>)
          (WANG (ADDL (CAR WFF) (ADDR (CADDR WFF) L))))
```

```
          ((EQ (CADR #FF) '/!)
           (#AND (ADDR (CAR #FF) (ADDR (CADDR #FF) L))))
          ((EQ (CADR #FF) '/%)
           (AND (#AND (ADDR (CAR #FF) (BUMP L)))
                (#AND (ADDR (CADDR #FF) (BUMP L)))))
          ((EQ (CADR #FF) '/<//=/>)
           (AND (#AND (ADDL (CAR #FF) (ADDR (CADDR #FF) (BUMP L))))
                (#AND (ADDL (CADDR #FF) (ADDR (CAR #FF) (BUMP L))))))))
;
;-----------------------------------------------------------------
;
(DEFUN ADDL (#FF L)
   (COND ((ATOM #FF)
          (LA L (INSERT #FF (LA L))))
         (T (LF L (INSERT #FF (LF L))))))
;
;-----------------------------------------------------------------
;
(DEFUN ADDR (#FF L)
   (COND ((ATOM #FF)
          (RA L (INSERT #FF (RA L))))
         (T (RF L (INSERT #FF (RF L))))))
;
;-----------------------------------------------------------------
;
(DEFUN BUMP (L)
   (LVL L (PLUS 2 (LVL L))))
;
;-----------------------------------------------------------------
;
(DEFUN PRINLINE (L)
       (PROG ()
             (PRIN3 % (LVL L))
             (PRLST (LA L))
             (PRLST (LF L))
             (PRIN3 ===/>)
             (PRLST (RA L))
             (PRLST (RF L))
             (PRIN3 <>))))
;
;-----------------------------------------------------------------
;
(DEFUN PRLST (L)
   (EVAL (CONS (QUOTE PRIN3) L)))
;
;-----------------------------------------------------------------
;

(DATA LINE (LVL LA LF RA RF))
(SETQ H '(PROVE ((/* (P & Q)) <=> ((/* P) ! (/* Q)))))
;
;-----------------------------------------------------------------
;
(DEFUN REPEAT FEXPR (QQQQ)
       (PROG (QQQ1 QQQ2)
        POP  (SETQ QQQ1 QQQQ)
        NEXT (COND ((EQ (CAR QQQ1) 'UNTIL)
                    (COND ((SETQ QQQ2 (EVAL (CADR QQQ1)))
                           (RETURN QQQ2))
                          (T (SETQ QQQ1 (CDR QQQ1)))))
                   ((EQ (CAR QQQ1) 'WHILE)
                    (COND ((EVAL (CADR QQQ1))
                           (SETQ QQQ1 (CDR QQQ1)))
                          (T (RETURN NIL))))
                   (T (EVAL (CAR QQQ1))))
```

```
            (COND ((SETQ 0001 (CDR 0001))
                   (GO NEXT))
                  (T (GO TOP)))))
;
;-------------------------------------------------------------
;
(DEFUN DATA FEXPR (0003)
       (CONS (PUTPROP (CAR 0003)
                      (SUBST (CAR 0003)
                             'TP
                             '(LAMBDA (0004)
                                      (CONS 'TP
                                            (MAPCAR
                                             (FUNCTION
                                              '(LAMBDA(X) (EVAL X)))
                                             0004))))
                      'FEXPR)
             (DATADEF (CADR 0003) 2)))
;
;-------------------------------------------------------------
;
(DEFUN DATADEF (0005 0006)
       (COND ((NULL 0005) NIL)
             (T (PUTPROP (CAR 0005)
                         (SUBST 0006
                                'N
                                '(LAMBDA (0007)
                                         (COND ((CDR 0007)
                                                (MAKENTH N
                                                         (EVAL (CAR 0007))
                                                         (EVAL (CADR 0007))))
                                               (T (NTH N
                                                       (EVAL (CAR 0007)))))))
                         'FEXPR)
                (DATADEF (CDR 0005) (ADD1 0006)))))
;
;-------------------------------------------------------------
;
(DEFUN NTH (N L)
       (COND ((EQ N 1) (CAR L))
             (T (NTH (SUB1 N) (CDR L)))))
;
;-------------------------------------------------------------
;
(DEFUN MAKENTH (N L S)
       (COND ((EQ N 1) (CONS S (CDR L)))
             (T (CONS (CAR L) (MAKENTH (SUB1 N) (CDR L) S)))))
;
;-------------------------------------------------------------
;
(DEFUN PRIN3 FEXPR (0007)
       (REPEAT UNTIL (NULL 0007)
               (SETQ 0007 (APPEND (PREF-OFF (CAR 0007)) (CDR 0007)))
               (COND
                ((EQ (CAR 0007) '*)     (PRINC (EVAL (CADR 0007)))
                                        (PRINC BLANK)
                                        (SETQ 0007 (CDDR 0007)))
                ((EQ (CAR 0007) '**)    (EVAL (CADR 0007))
                                        (SETQ 0007 (CDDR 0007)))
                ((EQ (CAR 0007) '%)     (PRIN-N (EVAL (CADR 0007)) BLANK)
                                        (SETQ 0007 (CDDR 0007)))
                ((EQ (CAR 0007) '<>)    (TERPRI) (SETQ 0007 (CDR 0007)))
                ((EQ (CAR 0007) ':)     (PRIN-N (EVAL (CADR 0007)) (CADDR 0007))
                                        (PRINC BLANK)
                                        (SETQ 0007 (CDDDR 0007)))
```

```
                ((EQ (CAR 0007) '!)    (SETQ 0007 (SET-UP (EVAL (CADR 0007))
                                                          (CADDR 0007)
                                                          (CDDDDR 0007)
                                                          )))
                (T (PRINC (CAR 0007))
                   (PRINC BLANK)
                   (SETQ 0007 (CDR 0007))))))
(SETQ BLANK '/ )
(SETQ LPAR '/()
(SETQ RPAR '/))
;
;-----------------------------------------------------------------------
;
(DEFUN PREF-OFF (A)
       (PREF-OFF1 (EXPLODEC A)))
(DEFUN PREF-OFF1 (XA)
       (COND ((NULL (CDR XA)) XA)
             ((EQUAL XA '(* *)) '(**))
             ((EQUAL XA '(< >)) '(<>))
             ((AND (CDDR XA) (EQ (CAR XA) '*) (EQ (CADR XA) '*))
              (LIST '** (READLIST (CDDR XA))))
             ((AND (CDDR XA) (EQ (CAR XA) '<) (EQ (CADR XA) '>))
              (LIST '<> (READLIST (CDDR XA))))
             ((MEMBER (CAR XA) '(* % : !))
              (LIST (CAR XA) (READLIST (CDR XA))))
             (T (LIST (READLIST XA)))))
;
;-----------------------------------------------------------------------
;
(DEFUN PRIN-N (N S)
       (REPEAT  UNTIL (LESSP N 1)
                (PRINC S)
                (SETQ N (SUB1 N))))
;
;-----------------------------------------------------------------------
;
(DEFUN SET-UP (0008 0009 0010)
       (PROG (0011 0012)
             (SETQ 0011 0)
             (REPEAT
              WHILE 0010
              (SETQ 0010 (APPEND (PREF-OFF (CAR 0010)) (CDR 0010)))
              UNTIL (EQ (CAR 0010) '!)
              (COND
               ((E) (CAR 0010) '*)
                (SETQ 0012 (CONS (LIST 'QUOTE
                                       (EVAL (CADR 0010)))
                                 (CONS '* 0012)))
                (SETQ 0011 (PLUS 0011 (FLATSIZE (CADAR 0012)) 1))
                (SETQ 0010 (CDDR 0010)))
               ((EQ (CAR 0010) '**)
                (EVAL (CADR 0010))
                (SETQ 0010 (CDDR 0010)))
               ((EQ (CAR 0010) '%)
                (SETQ 0012 (CONS (EVAL (CADR 0010))
                                 (CONS '% 0012)))
                (SETQ 0011 (PLUS 0011 (CAR 0012)))
                (SETQ 0010 (CDDR 0010)))
               ((EQ (CAR 0010) ':)
                (SETQ 0012 (CONS (CADDR 0010)
                                 (CONS (EVAL (CADR 0010))
                                       (CONS ': 0012))))
                (SETQ 0011 (PLUS 0011
                                 (TIMES (CADR 0012)
                                        (FLATSIZE (CAR 0012)))
```

```
                                1))
                  (SETQ 0014 (CDDDR 0010)))
                 (T (SETQ 0012 (CONS (CAR 0010) 0012))
                    (SETQ 0011
                          (PLUS 0011 (FLATSIZE (CAR 0012)) 1))
                    (SETQ 0010 (CDR 0010)))))
            (SETQ 0011 (DIFFERENCE 0008 0011))
            (SETQ 0010
                  (COND
                   ((EQ 0009 'L) (CONS '% (CONS 0011 (CDR 0010))))
                   ((EQ 0009 'C)
                    (CONS '% (CONS (ROUND (QUOTIENT 0011 2.0))
                                   (CDR 0010))))
                   (T (CDR 0010))))
            (REPEAT
             WHILE 0012
             (SETQ 0010 (CONS (CAR 0012) 0010))
             (SETQ 0012 (CDR 0012)))
            (SETQ 0010
                  (COND
                   ((EQ 0009 'C)
                    (CONS '% (CONS (QUOTIENT 0011 2) 0010)))
                   ((EQ 0009 'R) (CONS '% (CONS 0011 0010)))
                   (T 0010)))
            (RETURN 0010)))
;
;-------------------------------------------------------------
;
(DEFUN FLATSIZE (U) (LENGTH (EXPLODE U)))
;
;-------------------------------------------------------------
;
(DEFUN ROUND (X) (FIX (PLUS X 0.5)))
;
;-------------------------------------------------------------
;
(DEFUN INSERT (S L)
       (COND ((MEMBER S L) L)
             (T (CONS S L))))
;
;-------------------------------------------------------------
(DEFUN REMOVE (ITEM LIST)
       (PROG (RLIST)
             (SETQ RLIST NIL)
             (REPEAT
              WHILE LIST
              (COND ((EQUAL ITEM (CAR LIST))
                     (CONCAT RLIST (CADR LIST)))
                    (T (SETQ RLIST (CONS (CAR LIST) RLIST))
                       (PRINT RLIST)))
              (SETQ LIST (CDR LIST)))
             (RETURN RLIST)))
(DEFUN CONCAT (LIST1 LIST2)
       (PROG ()
             (MAPCAN (FUNCTION (LAMBDA (X)
                                       (SETQ LIST2 (CONS X LIST2))))
                     LIST1)
             (RETURN LIST2)))
```

4.2.4. その他のプログラム例

A. ロジック・シミュレータ

作成者　内　田　俊　一　氏（ＥＴＬ）

論理回路の構成と初期状態及び入力信号を与えた時，それ以降の出力信号値を求めるシミュレータ。

人間とマシンの対話システムとして，プログラムに工夫がなされており，またスタック使用量を少なくするための技法が組み込まれている。

B. ルービック・キューブ

作成者　古　川　康　一　氏（ＥＴＬ）

ルービック・キューブの初期状態を与え，それを最終状態（解）へ移す手順を示す。

Prolog上でプロダクション・システムを構成し，そのルールの集合としてプログラムを作成している。

C. 数式処理

作成者　新　田　克　已氏（ＥＴＬ）

与えられた数式の形を整えるプログラム。

(1) 分配則によりカッコをはずし，

(2) 各項の整理を行ない，

(3) 降べきの順にならべ，

(4) 同類項の整理を行なう。

という手順によるものである。

D. MILISYJ

作成者　田　中　穂　積氏（ＥＴＬ）

日本語ＱＡシステム。Lispで書かれていたものをProlog上に移植したものである。日本語のパージングに，ＤＣＧ（Definite Clause Grammar）を用いている。

以上のプログラム例の詳細については，作成者に問い合わせていただきたい。

4.3. マシン例

本節では現在米国で発表されたスーパーパーソナルコンピュータと称される商品群について説明する。これらのコンピュータの特長は高性能ながらシングルユーザの利用に的を絞り，ユーザインタフェースを最優先したシステム設計が行われている点にある。

以下，次の機種について各項で説明する。

(1) ＬＩＳＰマシン

(2) DORADO

(3) STAR

(4) PERQ

表4.3.1にマシンの簡単な比較を示す。

4.3.1. ＬＩＳＰマシン

A. 概　要

ＭＩＴのＡＩラボで１９７７年に完成したプロトタイプをベースとして，ＬＭＩ（Lisp Machine Inc.）とSymbolics社で市販されようとしている。このＬＩＳＰマシンはスタンドアロン的な使用を目的としたもので，ミニコン規模ながらPDP－１０のMACLISPと同等機能を備え，大規模ＬＩＳＰプログラムを実行させてみた処理性能はMACLISPと同等ないし数倍速いといわれている。

B．ハードウエア構成

マシンは２５６Ｋ語（１語＝３２ビット），制御記憶１２Ｋ語（１語＝４８ビット）で８０Ｍバイトのディスクと白黒のＣＲＴモニタを備えている。このマシンは，ＭＩＴが開発したローカルネットワーク「CHAOS ＮＥＴ」に接続する機能を持っている。またXerox社のETHERNETへも容易に接続できる構成となっている。このネットワークを通してＬＩＳＰマシンはファイルプロセッサがアクセスできる。図4.3.1に内部で使用されているマイクロプロセッサの構成を示す。内部データ幅３２ビットのマイクロプログラム制御のプロセッサである。制御記憶の他，プッシュダウンバッファ（１Ｋ語×１５ビット），マイクロ命令レベルのプログラムカウンタ・スタック（３２語×３２ビット），２５６語×３２ビットのローカルメモリを備えている。主記憶に対する論理アドレス幅は２４ビットで１６Ｍバイトの仮想空間をサポートすることができる。データ・バスは主記憶やディスクに対する３２ビットのＸバスと称するもののほかにPDP－１１の１６ビットバス（UNIBUS）をも備えており，それと互換性のある入出力装置や他システムのファイルとの接続が容易にできるようになっている。このＬＩＳＰマシンのデータ表現は，MACLISPに沿っているが図4.3.2に示すように各基本データ要素に５ビットのデータ・タイプを持たせている点が最も異なる。この５ビットタグによって各種の機能を持つことができる。また２ビットのＣＤＲコードによってリスト要素のＣＤＲ（次の３２ビット）を省略する構造を採用し，セル空間の効率的運用を行っている。図4.3.3にこのマシンの機械語を示した。

マシン名			LISP マシン	DORADO	STAR	PERQ
プロセッサ	処理方式		マイクロプログラム方式	マイクロプログラム方式	—	マイクロプログラム方式
	処理速度		クロック 180ns	60ns，パイプライン方式	i8086の6倍高速	170ns，1million P-code／sec
	その他		CONSプロセッサ	ECL使用	TTL（16ビット）	Bipolar bit slice
記憶部	主記憶	仮想空間	2^{24}	2^{28}	—	2^{32}
		実装	256K語（32ビット）	16M語（16ビット）	192K語（16ビット）	1MB
	制御記憶		16K語（48ビット）	4～16K語	—	16K語（48ビット）
	キャッシュ		—	4～16K語（16ビット）	—	—
二次記憶			80MB	80MB	29MB	12MB／24MB
ネットワーク			CHAOSNET，（ETHERNET）	ETHERNET	ETHERNET	ETHERNET
インタフェース マンマシン	ディスプレイ		700×900ビットマップ	0.5M～1Mドット ビットマップ	809×1024， ビットマップ	768×1024， ビットマップ
	キーボード，ポインティングデバイス		キーボード マウス	キーボード マウス	キーボード，256×256フォント マウス	タブレット キーボード
ソフトウェア	オペレーティングシステム		マルチプロセス・シングルユーザ用	マルチタスク・シングルユーザ用	—	マルチプロセス・シングルユーザ用
	言語		LISP	BCPL，Mesa Inter LISP，Smalltalk	Mesa	Pascal （Lisp，Ada，C，Fortran）
	エディタ		スクリーンエディタ マルチバーチャル画面	スクリーンエディタ マルチバーチャル画面	スクリーンエディタ マルチバーチャル画面	スクリーンエディタ マルチバーチャル画面
その他			MIT AIラボで開発 LISP Machine Inc. Symbolics	ALTO 改良版 Xerox PARC	Xerox	Three Rivers Computer Corp.

表4.3.1 マシン例

C．ソフトウエアの構成

LISPマシンはここで述べるソフトウェアによって高機能で強力なプログラミング開発環境を提供できる。システムのプログラミング言語としてはZetalispが用いられている。ソフトウェアの体系を図4.3.4に示す。テキストエディタはZmacsと呼ばれており，リアルタイムのディスプレイをベースとしたエディタである。又，Zmacsはディスプレイのグラフィック入出力を操作できる機能もあわせてもっている。ZmacsはAI分野で有名なEmacsエディタのほとんどのコマンドを実行できる。LISPマシンのファイルはファイルシステムによって制御される。システムは自系のファイルとファイルサーバと称するネットワークを介した手段によって他系ファイルへアクセスできる。このファイルシステムは非常に強固にできている。ディレクトリが破壊されても別のルートを用いて復旧できる。ファイルシステムの情報は冗長をもって格納されているためディスクの1ブロックがダメになってもただ1つのファイルが壊れるだけである。

他の強力な機能としてディスプレイ画面を十分に使った電子メールシステム（Zmailと呼ばれる。）がある。画面上で複数のメッセージを同時に扱えるし，ユーザの希望に応じたメッセージの制御が可能である。ビットマップディスプレイの画面はウィンドウシステムによって管理されている。ウィンドウは画面内の任意の大きさの方形の領域である。このウィンドウは全域見えている時もあるし，一部だけ表示されている時，全域ディスプレイ画面からかくされている時がある。それぞれのウィンドウは違ったタスクで制御される。タスクのスイッチはウィンドウをマウスで指示し，クリックすることによって実現できる。このような機能はプログラムを開発している時に特に大きな効果を発揮する。Zetalispシステムと Zmacsを画面の半分づつの大きさをもつウィンドウで代表させ，プログラムのデバッグと変更を任意タイミングで実行することが非常に簡単にできるようになる。その他プログラム開発用の道具としてデータ構造を眺めることができるインスペクタあるいはプログラムの開発を効率的に実行できるディスプレイデバッガ等がある。

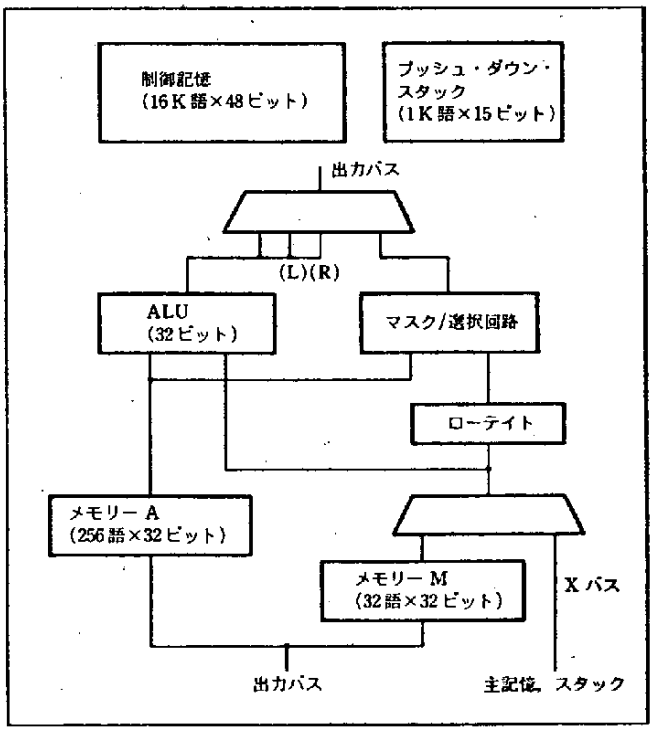


図 4.3.1　マイクロプロセッサの構成

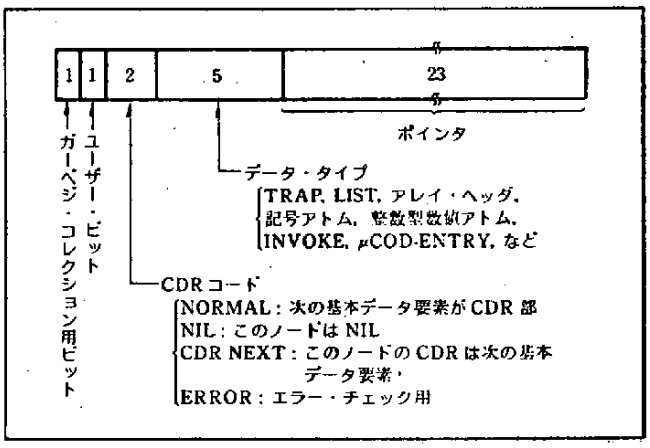


図 4.3.2　基本データ要素形式

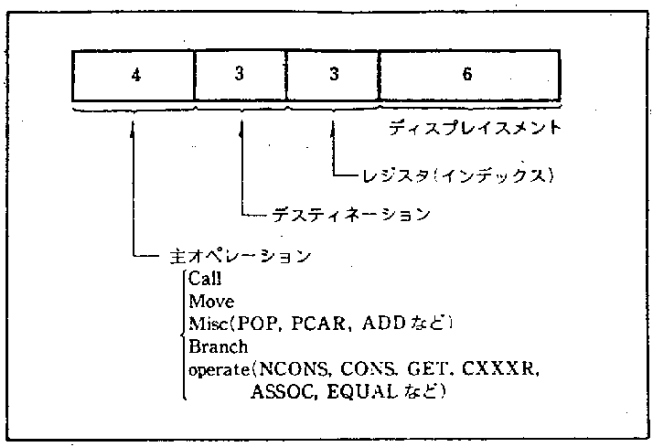


図 4.3.3　LISP マシンの機械語

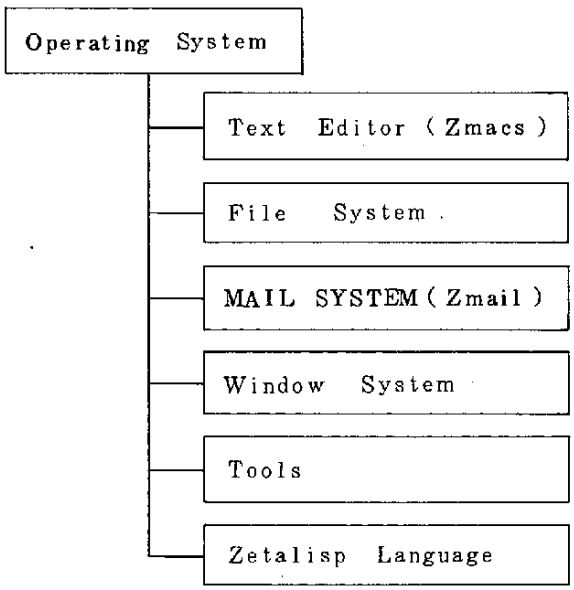


図 4.3.4　ソフトウェアの体系

—参考資料—

1) Symbolics Software Symbolics, Inc.

2) 松崎 稔「LISPマシン開発の現状を探る」
日経エレクトロニクス 1979. 3. 19

3) 坂村 健「LISPマシンのアーキテクチャ」
Bit vol. 13, №3

4.3.2. 高機能パーソナルマシンDORADO

A. 概 要

DORADOはXerox社Palo Alto Research CenterのCSL(Computer Science Laboratory)においてALTOの改良機として開発されたパーソナルコンピュータである。ALTOは1973年にCSLでパーソナルコンピュータの実験モデルとして開発され,以後6年間に諸々の改良が加えられこれまでに約1,000台が同研究所や他の研究機関で使用されてきたが,仮想アドレス空間,実記憶の大きさ,プロセッサ速度等が現在の研究活動には不十分で,これらの改良を目的としてDORADOが計画され,1979年春にそのMODEL 1が製作されている。しかし従来ALTOが持っていた次の特徴はそのまま受け継がれている。

(1) マイクロプログラム方式のプロセッサで全ての入出力制御をプロセッサシェアの方法で行う。

(2) メインメモリ上にビットマップメモリを持った高解像度のディスプレイシステムである。

(3) ディスプレイ上のイメージを示すための装置(マウス)を持つ。

(4) 高速の通信ネットワークをサポートしている。

B. 設計目標

DORADOは次の4点を設計目標にして製作されている。

(1) 強力なパーソナルコンピュータ作りに徹している。そのためマイクロインストラクションから高級言語までのさまざまなレベルでのプログラミング環境を整備している。またマシンは事務室や研究室のユーザのすぐ近くに設置可能となるように十分に小さく,かつ数多く配置出来るように安価でなくてはならない。

(2) 従来機ALTOとの互換性は余り問題としない。理由はALTO上でつくられたソフトウェアがすでに陳腐化しており,より速く実行可能なDORADO上で新しく作りかえる必要があった。

(3) その代わり,Mesa, Inter Lisp, Smalltalkなどの高級言語を高速にインタプリー

トするための高速なマシンサイクル，強力なマイクロインストラクションがあり，これらの高速動作をマイクロストア，パイプライン，キャッシュ等のインプリメンテーション上の技法で実現している。

(4) 高速入出力機能がサポートされる。CSLでの研究項目にはカラーモニタ，ラスタースキャンプリンタ，高速コミュニケーション等の高速転送を必要とするものがある。これらの研究に十分なだけの高速入出力がエミュレータのエミュレーションスピードを低下させることなしに並行動作し，かつ低速デバイスの転送が高速デバイスのデータ転送に影響を与えないようにサポートされなければならない。

C. システム構成と実装方法

図4.3.5にDORADOのブロック図を示す。

DORADOはコンパクト性を保つためにデータ幅を１６ビットに制限している。アドレス空間としては２８ビットの仮想アドレス空間を持ち，MAP・RAMによって２４ビットの実アドレスへ変換される。その変換の様子を図4.3.6に示す。４～１６KWのキャッシュを持ちマシンサイクル６０nsで動作する。主記憶装置は１６kbit，６４kbitのN-MOS RAMで構成され最大３２メガバイトまで実装される。比較的遅い素子（サイクルタイム375 ns）を使用しているが288ビット（２５６ビット＋３２ビットECC）を一括して８マシンサイクル(480 ns）でアクセスする方式を取っており高速入出力装置の転送スピード530MBit／SECを実現している。

図4.3.7にシャーシ実装図を示す。２１インチ×１５インチ×２６インチと比較的小さな筐体に２４枚の基板と電源および2個の冷却ファンが実装されている。マシン全体で約3,000個のMSIを使用しその大部分がECL１０Kであり，その約３５％がプロセッサに使用されている。AC電力は80MBのディスクに供給される分も合せると約2.5KWとなる。

D. アーキテクチャ上の特色

DORADOのアーキテクチャの特色は入出力制御にある。すなわちそれぞれの入力装置にプロセッシング能力を持たせバスをシェアしたリスイッチしてメモリをアクセスするよりもプロセッサをエミュレータと入出力装置で共有する方式を採っている。この方式はALTOから受け継がれた方式であるが次の理由によりこの方式が採用された。

第一の理由はシステムが複数のメモリバス（すなわちマルチポートメモリ）を持ちかつ独立に動作可能なメモリモジュールを持たなければシステムのスループットはメモリの競合の度合によって決定される。わかりやすくいうと入出力装置がメモリを参照している時，プロセッサはメモリが開放されるまで待たなければならない。このような状況下ではプロセッサが入出力

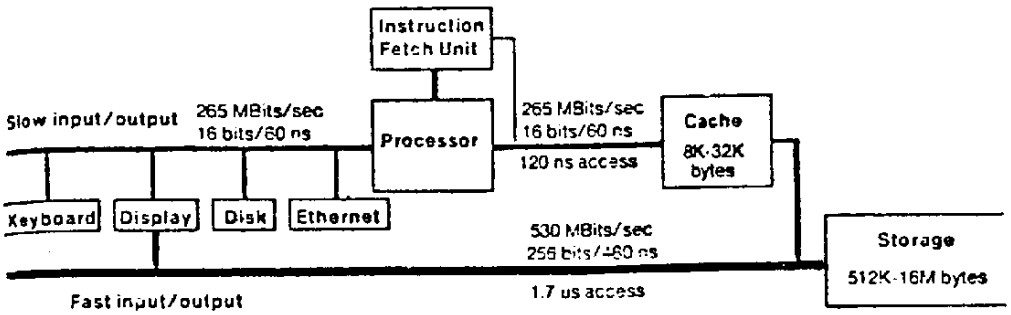


図 4.3.5　DORADOのブロック図

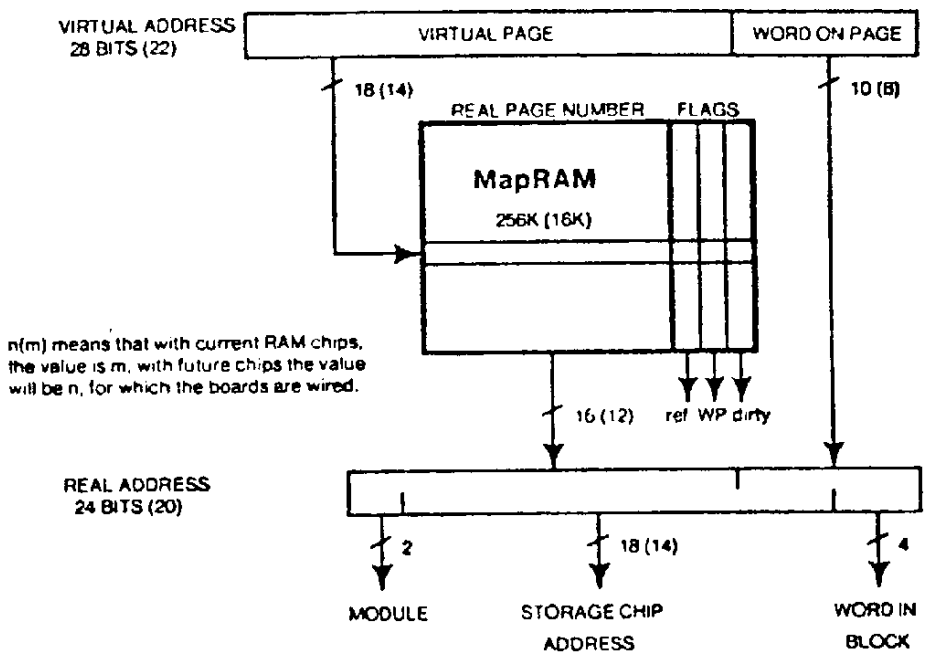


図 4.3.6　仮想アドレス－実アドレス変換

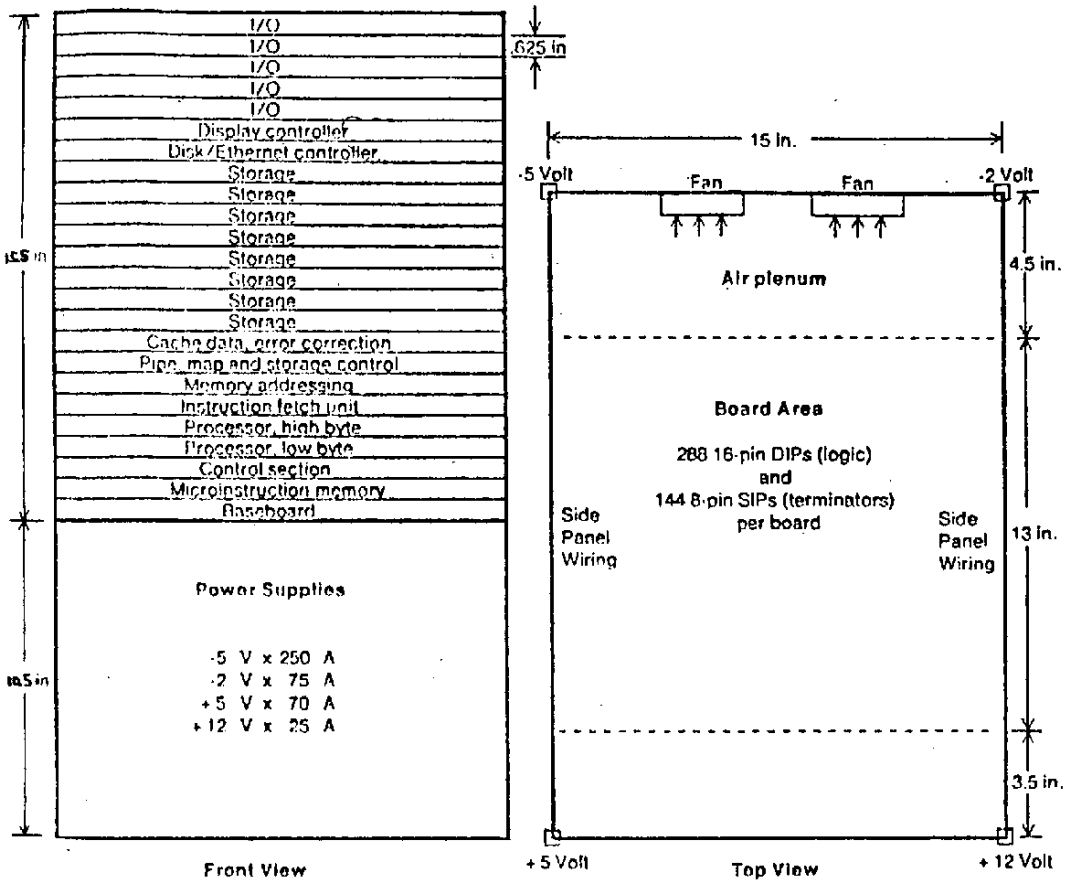


図 4.3.7　シャーシ実装図

装置の制御を代って行なっても同性能が得られると判断した。

第2の理由はプロセッサを各装置の制御に利用出来れば，各々のインタフェイスの中に重複してその制御用のハードウェアを持つ必要がなくなり複雑な装置とのインタフェイスも比較的少ないハードウェアで実現可能になると判断した。

E. 性　能

DORADO では BCPL，LiSP，Mesa，Smalltalk の4つのインストラクションをイン

タブリートするエミュレータがあり，LOADやSTOREのような基本的なインストラクションであればMesa，やBCPLの場合は1～2マイクロインストラクションで，Lispの場合は4マイクロインストラクションで実現される。フィルドやアレイエレメントに対するリード・ライトのような複雑なオペレーションではMesaで5～10マイクロインストラクション，Lispで10～20マイクロインストラクションを要する。また関数呼び出し（Function Call）ではMesaで約50マイクロインストラクション，Lispで200マイクロインストラクションを要する。1マイクロインストラクションは60nsで実行されることからしてかなり高速なマシンであるといえる。

DORADOは主記憶上に0.5～1Mbitのビットマップメモリを持つ方式でラスタスキャンディスプレイのリフレッシュを行っている。このビットマップを操作するためにBit・Blt（bit boundary block transfer）という特別な操作方法がありイレーズやスクロールのような簡単な操作であれば34Mbit／secのスピードでディスプレイを操作可能である。もっと複雑な操作であっても24Mbit／secのスピードで操作可能である。10Mbit／sec程度の入出力装置はI／Oバスを介してプロセッサによって制御される。例えばディスクから2ワード（32ビット）の転送を行うためにはプロセッサは3サイクルのマイクロコードを要するが10Mbit／secの転送スピードのディスクではプロセッサ負荷の約5％程度で制御される。

F．ま と め

文献1）にもとづいてDORADOのアウトラインについて紹介したが，DORADOはその設計目標にもあるように強力なパーソナルコンピュータを目指して設計されており，現時点では決してコンパクトで安価なマシンとは云い難いが，その処理スピード，機能目標は今後のパーソナルコンピュータが目指すべき方向を示唆したものである。特にディスプレイを主眼に置いた設計は以後のパーソナルコンピュータやLispマシンにも引き継がれている。

－参考文献－

1） The Dorado： A High-Perfomance Pensonal Computer by Butter W. Lampson and kenneth A. Pier.

4.3.3. STAR

ゼロックス社パロアルト研究所で開発されたALTOは，大型のTSSシステムで実現されている機能を低コストのパーソナルコンピュータ上でいかに実現すべきかを探るための実験システム

であった。前節のDORADOがALTOの研究用改良版であるのに対し，STARはその商用版である。STARの主眼はアーキテクチャやソフトウェア構成にあるのではなく，ALTOの使用経験を活かして優れた人間機械間の親和性を実現した，中堅管理者および"knowledge worker"向ワークステーションを提供したことにある。STARはEthernetをベースとしたローカルネットワークの主要コンポーネントとして1981年4月に発表された。

STARの開発方針として以下の四項目が挙げられている。

○ 従来システムのように多種多様なコマンドを覚えてタイプインするのではなく，視覚的に指示するだけでシステムが動作するよう設計する。

○ 処理の進行に伴って徐々にその先が見えていくようにする。すなわち，各時点で情報が過多にならないよう，必要十分な事項のみをユーザに示す。

○ 処理全体をとおして一貫したコマンド体系とし，基本的なロジックを理解すればシステムを使いこなせる。

○ ユーザに見えるものが即ちユーザに与えられるものであるようにする。

STARの特徴は大型のビットマップディスプレイと改良型マウスにある。ディスプレイスクリーンは809×1,024ドットを表示し得る横長のもので，同時に二枚のドキュメントを並べて表示できるよう考慮されている。画面には最大6個のウィンドウを設定できる。またリフレッシュサイクルを38.7サイクル／秒とし，フリッカを少なくするよう改良している。マウスは従来の3つボタンを2つボタンとし，シンプルにして非プログラマが混乱することなく使えるよう工夫されている。また，キーボードには使用頻度の高いコマンド用の特別キーを設けると同時に，ディスプレイ上に表示したコマンドの1つをマウスで指示し，ボタンをクリックするとそのコマンドが実行されるようになっている。

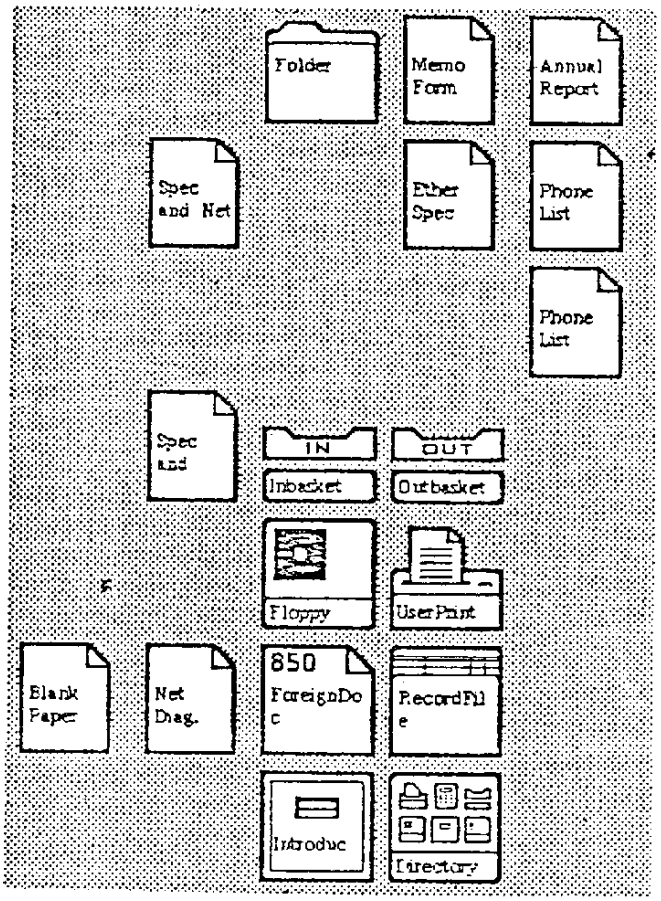
プロセッサは16ビットのLSIで，インテル8086の6倍高速という。メモリは192KW（1ワードは16ビット）で，50KWはビットマップのリフレッシュ用，140KWはOSとアプリケーション用である。プロセッサは6枚のボードから成り，机と同じ高さのキャビネットに収められている。

ソフトウェアはゼロックス社のMesaというプログラミング言語で作られている。STARのソフトウェアはALTOと同様に，すべての機能を実行できる単一の統合化されたプログラムを作るのではなく，夫々のユーザが必要なプログラムを積み上げることで作られる。ALTOにおいて最も広く使われたパッケージがドキュメントの作成と配布であり，これがビジネス用システムとして結実したものが，STARであるといえよう。STARの表示画面では事務机の上の配置と同様な環境が視覚的に表示される。ドキュメントやフロッピー，メール箱，プリンタ，ファイル等が夫

々一見してわかるようなシンボル（iconという）を用いて示される。（図4.3.8参照）

STARのディスプレイは解像度が1／72インチで，最大256×256種のフォントを表現できる。文字フォントとして，スモールキャピタル，ギリシャ文字，Σ, π を含む数学シンボル，データ処理シンボル，漢字等を表示できる。キーボードは仮想化され，フォントセットを切換えれば対応するキーボード配置がスクリーン上に表示される。

レコード処理のためには，文字列のフィルタリング，スペルチェック等のプログラムが用意されている。フィルタリングは，検索時に値の範囲を指定してその条件に合うものを探す機能である。スペルチェックはバッチ的に使用するもので，外部の辞書との照合を行い，辞書に無いスペリング語が入力されると警告を与える。



Hard copy printout of icons as they appear on the Star screen. We think that most of the objects are explicit enough to require no explanation. Clearly, the pieces of paper with the corner turned down are individual documents, the folder is a file folder, the drawer labeled "Introdu" is a file drawer, and "UserPrint" is a Print Server. Note the in- and out-baskets and the floppy.

図4.3.8　ＳＴＡＲ画面表示例

STARはXerox 8000というローカルネットワークの主要構成要素である。このネットワークは10メガビット／秒の商用Ethernetで，ファイル，プリンタ，メール機能の多数ユーザ間での共有を可能にしている。STARはこのシステムの最上位ワークステーションとして位置付けられている。

—参考文献—

1) "Xerox's Star" The Seybold Report vol.10, No.16 April 27, 1981.

4.3.4 PERQ

PERQはスリーリバース・コンピュータ・コーポレーションが1981年に発表した最初のスーパー・パーソナル・コンピュータである。PERQにおけるスーパー・パーソナル・コンピュータの定義は，以下の項目を含むものである。

○単一ユーザに高速処理機能を与えるCPU。

○大型プログラムを走らせることができる大容量メモリと大容量仮想記憶空間。

○高品質ディスプレイ。

○ローカル・ネットワーク・インタフェース。

○大容量高速ディスク装置。

○ポインティング・デバイス

PERQの処理性能はPascalのPコードを1秒間に百万回実行できる速度という。

A．ハードウェア構成

PERQのハードウェアは，マイクロプログラマブルなCPU，256KB～1MBの主記憶，12又は24MBのウィンチェスタ型ディスク，高解像度ディスプレイから成る。

CPUは16ビットのデータパスを基本とし，32ビット演算が可能なマイクロプログラムマシンである。（図4.3.9参照） マイクロ語長は48ビットで，170 nsのサイクルで動作し，コントローラとしてAMD 2910が用いられている。4KWの書込み可能制御記憶（WCS）と，256個の汎用レジスタおよびハードウェア化されたシフタ，スタックを備えていることが特徴的である。WCSを用いてユーザがマシン機能を拡張できるようマイクロプログラム仕様が開放されている。また，多数のレジスタとハードウェアスタックは，PascalのPコードインタプリターを作り易くしている。

主記憶は256KBから最大1MBまで容量を増加することができる。サイクルタイム680 nsのMOSメモリで構成されており，1語64ビット，バンド幅は95MB／Sである。

ディスプレイは1024×768ドットのビットマップディスプレイで縦長に配置されている。ポインティングデバイスとしてはビットパッドを用いたペン型タブレットが用意されている。

外部インタフェースとして10Mb／S Ethernet，音声出力ポート，RS232Cポート，GPIBインタフェースが用意され，他装置とのフレキシブルな接続が可能である。

B．ソフトウェア

PERQの論理アドレス空間は4GBで，最大1MBの実空間とのマッピングにはセグメント概念が用いられている。システム言語としてPascalが採用されており，そのコンパイラが標準装備されている。PERQのPascalはJensen，Wirth仕様に準拠しているが，大容量空間で大規模なシステムを組み易くするための拡張が施されている。主な拡張点は，ストリング，ダイナミックアレイ，Others節，アディショナルプリデファインドタイプの導入とモジュール概念の導入である。モジュール概念は大プログラムをモジュール毎にコンパイルし，リンカによってバインドして大プログラムのオブジェクトを作る機能によってサポートされている。このためすべてのコードはリエントラントであり，コードセグメント毎にグローバルデータブロックと外部セグメントテーブルを備えて，互いに他のセグメントを参照できるよう構成されている。PERQはPascalのPコードに酷似したQコードを備えている。

スリーリバースコンピュータコーポレーションは米国カーネギメロン大学と共同でSpice（Scientific Personal Integrated Computing Environment）プロジェクトの開発に当っている。このプロジェクトではSpice Lispというリスプインタプリタを開発中である。このLispはMITのリスプマシンリスプに似ており，Lisp Byte Code，メモリ管理およびガーベジコレクション用のマイクロコードが含まれている。

SpiceはPascal，Lispの他にAde，C，Fortranコンパイラを持つ。CおよびFortranコンパイラは共にQコードを作るが，性能向上のためにマイクロコードでサポートされる。また，ディスプレイおよびネットワークサポート付きのUnixをPERQ Unixとして提供する予定という。

—参考文献—

1）"PERQ A High Performance Single User Workstation"

2）Three Rivers Computer Corporation → 理経

3）"PERQ System Software Summary" 理経

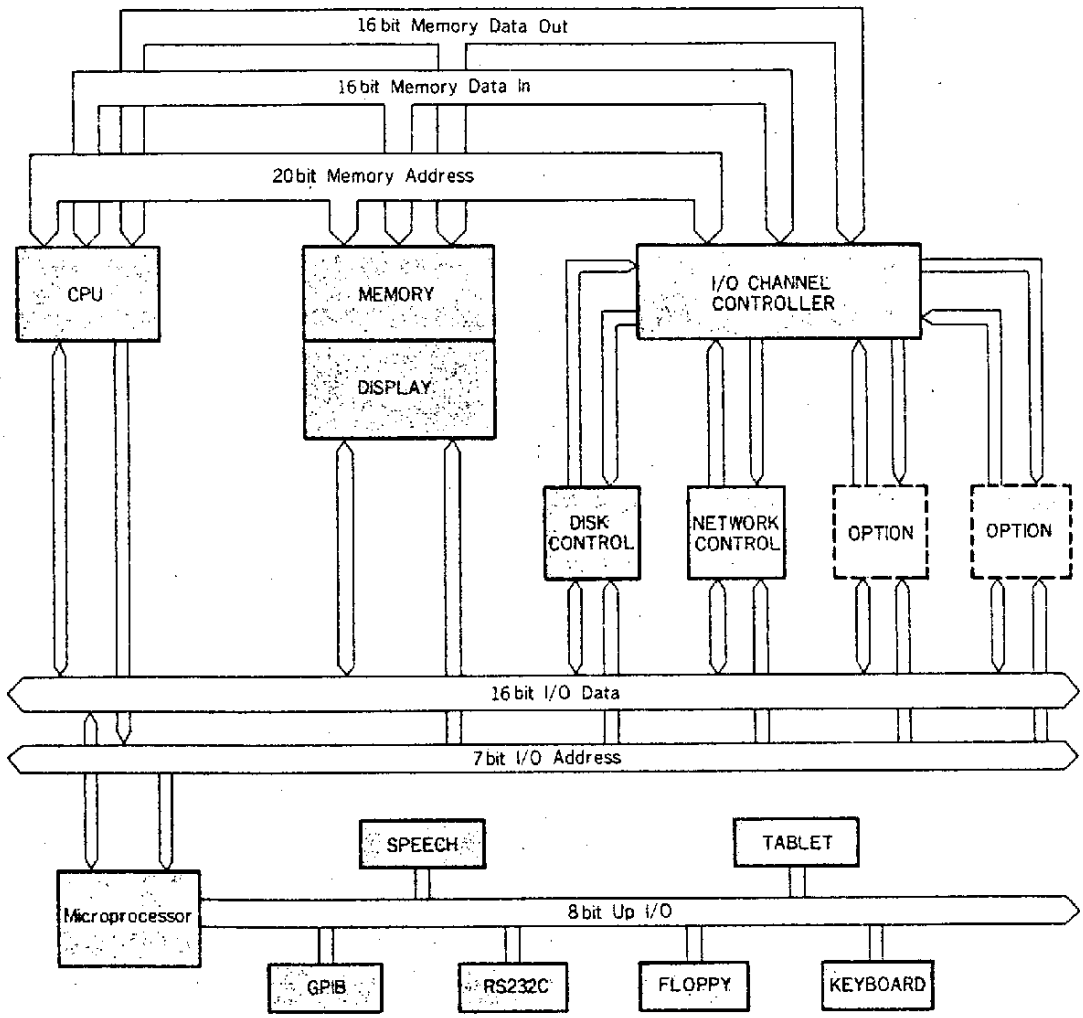


図 4.3.9　ＰＥＲＱハードウェア構成

昭和57年2月発行

発行所　財団法人 日本情報処理開発協会
社団法人 日本電子工業振興協会
東京都港区芝公園3—5—8
機械振興会館内
TEL 03(434)8211(大代表)

印刷所　有限会社 サンコピー印刷部
東京都大田区西蒲田4—23—7
TEL 03(754)7076