ICOT Technical Report: TR-054

TR-054

核言語第1版概念仕様書

古川康一、国藤　進、竹内彰一
上田和紀（日本電気）

1984. 3

ICOT

Mita Kokusai Bldg. 21F
4-28 Mita 1-Chome
Minato-ku Tokyo 108 Japan

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

核言語第1版概念仕様書

{古川　康一（ＩＣＯＴ），
国藤　進　（ＩＣＯＴ），
竹内　彰一（ＩＣＯＴ），
上田　和紀（日本電気）}

１９８４年　３月

1. 序 論

1.1 概 要

核言語は，本プロジェクト前期で開発される逐次型推論マシンＳＩＭの機械語であるＫＬＯをベースとして，中期以後は並列実行機能の拡張を主としたＫＬ１，さらに後期には知識表現サポート機能の拡張を伴ったＫＬ２への発展を考えている。

ＫＬＯについては，その設計は完了し，その言語仕様はすでに逐次型推論マシンＳＩＭの提供する機械語の仕様として開発過程に取り込まれている。

本概念仕様書では，主としてＫＬ１の仕様について述べることにする。ＫＬ１を考える上で最も重要な点は並列性であることはすでに述べた。しかし，それは単なる並列性ではなく，図１に示すように，その言語の上で知識表現，知識ベース管理，協調問題解決などの，知識情報処理技術を展開する上で土台となる諸機能が自然に，しかも，効率良く実現されることが必要である。そのために必要とされるより基本的な機能は，柔軟なプログラミング機能であることは言うまでもない。知識情報処理技術はこれまで人工知能の研究を中心として発展をとげてきたが，そこで最も重要な役割を演じたのがLispに代表される柔軟なデータ構造の扱い，すなわちリスト処理能力を備えたプログラミング言語である。プログラミング言語の柔軟性を考えるとき，データ構造とともに忘れてはならないのは制御構造である。Lispは，その点でも従来のFortran などの言語にない再帰呼び出しの機構が自然に備わっていた。一方，われわれは，Lispに代わって，Prologを本プロジェクトのプログラミング言語の出発点として考えたが，それはPrologが，柔軟性に関してLispと同等以上であると判断したからである。

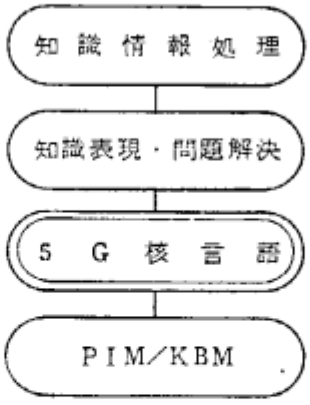


図 1　　５Ｇ核言語の位置づけ

さて，並列実行環境は大きな制約となり，一般に，逐次実行環境における柔軟性をそのまま維持することは大変困難である。そして，これまで，汎用の並列プログラミ

ング言語／マシンが成功した例は見当らない。本プロジェクトは，そのような困難な問題に挑戦することになる。

われわれが選択した道は，第１がConcurrent Prolog （ＣＰ）［Shapiro 83］である。ＣＰは，Relational Language ［Clark 81］を基にして設計されたプログラミング言語で，現在のPrologが逐次実行環境に適した論理型プログラミング言語であるのと同じような意味で，それは並列実行環境に適した論理型プログラミング言語であると言える。ＫＬ１はＣＰそれ自身ではないが，その最も重要な部分となっていると言えよう。

ＫＬ１の構成要素でＣＰについで重要なものは，探索機能である。知識情報処理のうちでも人工知能的な色彩の強い応用では，多くの可能性の中から解を探し出す探索問題に帰着できる問題が多い。ルールベースのエキスパート・システム，碁や将棋などのゲームなどがその類である。この種の問題を高速に扱える機能は，本プロジェクトの目標を達成するためにも欠かすことができない。そのための基本機能が全解探索機能である。本プロジェクトの前期で開発してきた関係データベースマシンは，一種の全解探索マシンと考えられる。また，逐次型Prologから逐次制御を取り除いて得られる純粋Prolog（Pure Prolog ）は，全解探索を表現する言語と考えることができる。

以上の２つの構成要素を結合するためのインタフェースが集合ーストリーム・インタフェースである。すなわち，全解探索サブシステムでは，すべての解を集合として求め，ＣＰサブシステムではそれをデータの流れ（ストリーム）として扱うことになる。これらの２つの構成要素は，その並列性の力点のおき方の違いから，全解探索の方をＯＲ関係，ＣＰの方をＡＮＤ関係と呼ぶ。

ＫＬ１はＣＰを中核としていることから，並列実行環境に適した制御構造を，その言語の中に自然な形で含んでいる。それは複数の実行主体を同時に走らせることができるもので，プログラミングの観点から見るとマルチ・プログラミング機能であり，オブジェクト指向プログラミング機能である。オブジェクト指向プログラミングは，一方，知識情報処理の応用の観点からも重要な技術である。それは，例えば，自然言語理解を考えるときに，複数の問題解決機による共同作業として形式化する場合に用いられる枠組であるからである。複数の問題解決機の中には，用語専門家，構文解析専門家，意味解析専門家，文脈解析専門家などが考えられる。また，これらは，ただ１つづつ存在するのではなく，入力の各語，各節，各文などの単位で適当に作ることができる。この種のプログラミング技術は，応用分野から並列性を抽出する上で不可欠である。そして，このようなプログラミングを支援するものとして，知識プログラミング言語／システムを考えているが，前期にＣＰ上で開発が進められている知識プログラミング言語／システムMandala［Furukawa 84］は，ＣＰをＫＬ１の中核とすることの妥当性を裏づけていると言えよう。

Mandala 自身は，ＣＰ上の階層型モジュラー・プログラミング・システムの土台としても考えられている。その基本は，is_a ，part_ofのような概念間の階層関係に基づくモジュラー・プログラミング機能である。そして，その考えは，ＫＬ０上のシステム記述言語ＥＳＰと全く同じである。すなわち，ＣＰを中核として選択することによって，ＫＬ０およびＫＬ１は，その上に作られるシステム記述言語のレベルで，少なくとも思想的な連続性を達成できたことになる。現実問題としてのソフトウェアのコンパティビリティは，今後の課題として残されている。

Mandala の効率的な実現もＫＬ１にとって大切な条件である。そのために必要な機能はモジュール化のサポートとメタ推論機能である。これらは，お互い深く関連し合っている。モジュール化機能は，システム・プログラムのレベルでソフトウェア工学的側面（プログラムの作成，管理等）の改善のため，および世界の階層化，多世界化など，知識の構造化のサポートのための２つの目的を有している。そして，前者ではとくに効率が重視され，後者では柔軟性が重視される。このような相異なる要請を満足させるための基本的な道具立てをどのように決めるかが問題となる。ＫＬ１では，この種の機能の基本として，機械語型というデータ型を用意し，それに対して，プログラムをコンパイルする操作，および，コンパイルされたプログラムを指定して与えられたゴールを解く操作，の２つの基本操作を用意する。そして，これらの基本操作を用いて，あるモジュール内だけで使われる局所的な述語の実現，モジュールのパラメータ化のための外部参照機能の実現，およびモジュールの階層化の実現を行うものとする。

メタ推論機能の効率化は，大変困難な課題であるが，すでに述べたようにMandalaの実現にとって重要であるばかりでなく，並列プロセスの知的エディタ，デバッガの実現，並列に動作するマシン間のインタフェース合せ，協調問題解決への応用など，その及ぼす効果は大きい。メタ推論機能の基本は，モジュール化サポート機能であるコンパイルされたプログラムを指定して与えられたゴールを解く操作と考えられるが，それを核にして，より柔軟な制御を行うための制御方式のパラメータ化などを実現することが必要となる。これは，プロセスのスケジューリング，探索空間の制限などといったＣＰ実行環境の操作機能をも含むものである。このレベルのプリミティブをどの程度ユーザ・レベルに解放するかどうかを決めることが，今後の課題である。

メタ推論機能のうち，メタインタプリタによって実現しなければならないものは，直接実行に比して１桁程度の速度の低下が見込まれる。その速度低下を回避するためには，オブジェクト・レベルの計算をできるだけ多く取り込んだ形でメタ・レベルとオブジェクト・レベルの融合を図ること，および可能ならばメタ・レベルの一部もコンパイルすることである。このような技術は未だ見えていないが，今後の課題として検討して行きたい。

図２は，核言語ＫＬ１の概念的な構成と，他のサブ・システムとの関連を示したも

のである。ＫＬ１の概念的な構成は，これまでに述べた事柄をまとめて表わしたものである。そして本関連図が示しているように，ＫＬ１は，本プロジェクトの目指す第５世代コンピュータの要石である。そのため，本言語の役割は大きく，その設計の良否は，本プロジェクトの成否をも左右する大切なものである。従って，ＫＬ１は今後とも並列推論，集合表現，モジュール化，およびメタ推論といった諸機能の詳細検討を進め，その妥当性をさらに強化して行くこと，およびＰＩＭ／ＫＢＭの設計に耐え得る厳密性を確保していくことを目指すものとする。

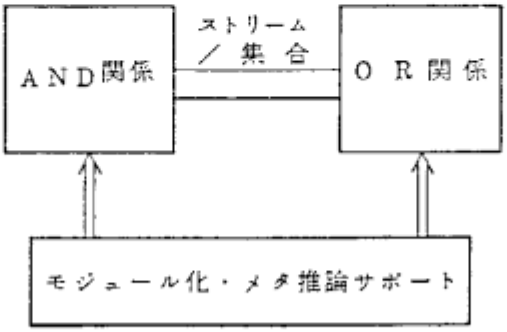


図２　ＫＬ１の概念構成

1.2　設計思想

ＫＬ１は，並列推論マシンおよび知識ベース・マシンの機械語を与える言語である。ＫＬ１は，その上で知識プログラミングあるいは知識情報処理を展開する土台となるべき言語であるが，その満足すべき条件は，以下の通りである。

(1) 並列実行環境下での柔軟なリスト処理並びに実行制御機能。
(2) 並列実行環境下でのデバッグ，トレースなどのプログラミング環境のサポート。
(3) 知識情報処理のための基本機能（知識表現機能，知識ベース管理機能，協調問題解決機能）のサポート。
(4) 探索型問題のサポート。

以上の４条件を満足する言語を設計する上で，われわれが考慮した議論の枠組は，以下の通りである。

(a) 論理性の維持
(b) 並列実行環境での自然な実行制御機能の追求
(c) Pure Prolog の記述力の維持
(d) オブジェクト指向プログラミング機能の実現

(e) ＫＬＯ，ＥＳＰとの連続性の維持

以下に，これらの６項目について，われわれがどのような点を特に考慮したかについて述べる。

(a) 論理性の維持

「論理性を維持する」ということは，プログラムの実行結果が，プログラムをHorn Logicにおける公理の集合とみなしたときに，必ず公理から証明できることを意味する。この逆は必ずしも成り立たないことに注意しなければならない。しかし，少くともこの論理性によって，プログラムの正しさが公理系としての正しさにのみ依存することが言える。これはプログラムが停止した場合の正当性であり，その意味でそれほど強力な性質でないかも知れないが，デバッグ，トレースなどを容易にし，プログラムの可読性を向上させる原動力となっている。

また，この性質は，今後の理論的研究成果を産み出しうる土台を提供しているので，新たな発展の可能性を秘めているといえよう。例えば，現在逐次型Prolog上で記号実行，プログラム変換，アルゴリズミック・デバッギング，プログラムの合成・検証などの研究が展開され，他の言語では実現できなかった成果が上がってきている。このような成果の少なくとも一部は，並列実行環境においても有用なものとなることが期待される。

(b) 並列実行環境での自然な実行制御機能の追求

逐次実行環境における自然な実行制御機能を備えた論理型プログラミング言語がPrologあるいはＫＬＯであると言えるが，われわれが必要としているのは，並列環境での同様な自然さである。Prolog（ＫＬＯ）の実行制御機能はＡＮＤ－ＯＲ探索木の左から右，深さ優先の探索，失敗したときの自動後戻り機能，および不要な探索枝を刈り込むカットである。

これに対して，並列環境では，巾優先探索が自然であり，自動後戻りではなくＯＲ並列探索が自然である。さらに，並列度を上げるためにＡＮＤ並列もとり入れたい。そのときに枝刈りをどうするかが問題となる。ＡＮＤ並列は無制限に行うと，たちまち不要な計算が大量に発生する危険性を有する。ＡＮＤ並列のサブクラスで扱いうるものにストリーム並列があることが知られている。ストリーム並列性は，論理的なレベルでのパイプライン処理であるが，パイプライン処理自身，ＯＲ並列の一部として，すなわちその計算が無効になるかも知れないものとして行うことを許すかどうかが大きな問題となる。Concurrent Prolog におけるコミット・オペレータは，これを許さない仕様になっている。その意味で，かなり強力な制御プリミティブとなっているが，並列実行環境の下では，その制限は，

妥当なものと考えられる。その代わり，ＯＲ並列計算能力を他の手段により補ってやることが必要である。それが次に述べるPure Prolog の記述力を維持する問題である。

(c) Pure Prolog の記述力の維持

探索型問題は逐次型Prologにおいては現在良く知られているように自動後戻りを利用した逐次型のdepth-first , left-to-right の探索が行われる。一方，Concurrent Prolog (KL1) においては，Stream-AND並列やOR並列を利用した並列探索が行なわれる。これを簡単な例により説明する。

例題は下記の記述のように，human であり，かつgreek であるようなものを求めるということである。

［Horn節による記述］

```
human(turing) .
human(socrates) .
human(aristotle).
greek(socrates) .
greek(aristotle).
```

［逐次型Prologプログラム］

この問題の解は逐次型Prologでは次のように書かれる。良く知られているように，このプログラムはバックトラックを用いた逐次型探索プログラムである。

```
:-human(X), greek(X).
```

［並列探索プログラム］

Concurrent Prolog プログラムによる同じ問題に対する解は，例えば次のようになる。このプログラムはConcurrent Prolog のStream-AND並列性を利用した並列探索プログラムであり，探索型問題に対する標準的解の一つである。

```
:-clauses(human(X), L), select_greek (L? , R).                    (1)

select_greek([human(X) | L], X) :- greek (X) | true.
select_greek([ _ | L], R) :- otherwise | select_greek(L ? , R).
                                                                  (2)
```

ゴール(1) はhuman(X)を満足するものをすべて集めて，

[human(turing), human(socrates) , human(aristotle)]

というストリームを生成するclauses というプロセスと，このストリームを受取り，greek であるものだけを返すselect_greek というプロセスの２つの並列なプロセスを記述している。(2) はselect_greek のプログラムである。select_greek プログラム(2) の第１節はストリームからhuman(X)というデータを受取ったときに，このＸに対してgreek (X) かどうかを調べ，正しければそれを第２引数に返すということを記述しており，第２節はそれ以外，すなわち，受取ったＸがgreek でなかったときにストリームの次のデータを見に行くことを表わしている。一般にStream-AND並列性を利用したプログラムは，候補となるデータを生成するプロセスとそれをテストするプロセスの組合せで問題を解く，いわゆる“generate and test ”型のプログラムとなっている。

このようにして，制限の強いＯＲ並列，Stream-AND並列の組合せによっても，探索問題が処理し得ることが示された。

(d) オブジェクト指向プログラミング機能の実現

知識情報処理の応用の方から考えてみると，一般に多くの並列性を引出すことは大変困難であるといわれている。たしかに現状の技術では，それは正しいかも知れない。しかしながら，われわれは同時に新しいプログラミング手法も手に入れている。その中でも，オブジェクト指向プログラミング，データ指向プログラミングの２つは，その計算モデル自身に並列性の概念が含まれており，応用から並列性を引出す上での有力な方法と考えてよいであろう。少なくとも，このような手法自身未だ開発がし尽されていないので，可能性を探ることは十分に意味のあることであると思われる。Concurrent Prolog はその言語自身に，この２つのプログラミング手法を実現するメカニズムが備わっており，その意味でもＫＬ１検討の出発点にふさわしい言語であると言えよう。またConcurrent Prolog 上で開発中の知識プログラミング言語Mandala は，知識ベース管理機能をも有しており，知識情報処理をその上で展開し得る見通しも得つつある。

(e) ＫＬＯ，ＥＳＰとの連続性の維持

ＫＬＯ，ＥＳＰとの連続性を考えるにあたり，ＫＬＯ－ＫＬ１，ＥＳＰ－Mandala の対応を考えることにする。すなわち，ＫＬＯ，ＫＬ１はともにＳＩＭ，ＰＩＭの機械語であり，ＥＳＰ，Mandala はそれぞれＫＬＯ，ＫＬ１のユーザ言語である。以下，プログラムの蓄積の観点から重要であるユーザ言語レベルの連

続性について考える。

ＥＳＰおよびMandala は少なくとも次の２点で連続性があると考えられる。

(i) ベース言語（ＫＬＯ，ＫＬ１）が共に論理型言語である。

(ii)いずれもモジュール構造を有し，かつオブジェクト指向プログラミングをサポートしている。

ベース言語（ＫＬＯ，ＫＬ１）間の相違点は，(b) で述べた通りである。(ii) の点についての相違点は，メッセージ通信の方法とオブジェクトの状態の更新の方法の違いがあるが，それは細かい実現レベルの話であり，言語レベルでの統一化は可能である（厳密に言えば，通信の自由度はConcurrent Prolog の方がＥＳＰより大であるが，upward compatibleにすることは可能である）。

これらの相違を吸収して，ＥＳＰをMandala へ変換するトランスレータが実現可能かどうかが問題となる。現状では未だ解決されていない問題もあるが，今後ＥＳＰ，Mandala の双方を改訂していく過程でそのギャップを縮めていき，トランスレータ実現を可能とするようにしたい。また，この方向を推進するためにも，われわれはMandala の記述言語およびベース言語となっているConcurrent PrologをＫＬ１の出発点とすることが妥当であろう。

本プロジェクトにおけるこれからのソフトウェアの蓄積が問題となるが，上に述べたようなトランスレータが実現すれば，ＥＳＰで書かれたソフトウェアは容易にMandala プログラムに変換され得る。完全なトランスレータが不可能な場合も，トランスレート支援システムは実現されよう。また，エミュレータの開発も可能であろう（POPS［Hirakawa 84 ］参照）。

2.　基本計算メカニズム

2.1　AND関係節と OR 関係節

KL1は Horn 論理に基づくプログラミング言語であり、論理型言語の基本的な特徴である関係記述能力を保ちつつ、 Horn 論理プログラミングの部分集合とも言うべき並列プログラミング機能を強化し計算の基本としている。具体的には KL1で記述される述語は静的な関係記述用の OR 関係と動的な並列プログラミング用の AND関係に分類され、それぞれの計算メカニズムが KL1において独立に定義される。 OR 関係は、基本的にその関係を満足するものすべてを求めるように計算が行なわれ、データベースなどの don't know non-determinism に対応した集合をベースとした計算を記述するのに適している。一方、 AND 関係の計算は don't care non-determinism に基づき、バックトラッキングのないものであり、プロセスの状態遷移の記述などに適している。本仕様書においては OR 関係と AND 関係をシンタクティックには前者を non-guarded節、後者を guarded節とすることにより区別する。

KL1 はこの２つの関係を融合する。すなわち、基本プログラミング言語は並列プログラミングのできる AND関係とし、 OR 関係については、 AND関係の中から集合表現という特別のインタフェースを介して呼出すこととする。

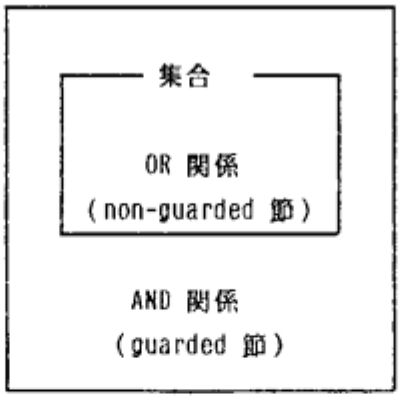


図　　3

このような言語構成は Parlog [Clark 83]において初めてとられた。 AND関係、 OR 関係等の名称は Parlog での名称を引継いだものである。本章では最初に本仕様書で用いる仮のシンタックスについてまとめ、AND 関係における計算メカニズムについて説明し、その他の制御プリミティブについても解説する。OR関係の計算については３章で解説する。

2.2 シンタックス

[プログラム]　KL1ではプログラムは節の並びとして表わされる。節にはguarded 節とnon-guarded 節の２種類があり、前者は AND関係、後者はOR関係を記述する。

[Guarded 節／commit operator]　Guarded 節は次に示すように常に右辺に“｜”をもつ節である。

A :- G｜B.

G及びBは、論理的にand で結ばれた素命題（atomic formula）の並びであり、それぞれ、guard 部、body部と呼ばれる。“｜”はcommit operator と呼ばれ、論理的には ANDを表すが、オペレーショナルにはcut symbolの拡張概念になっている。

[non-Guarded 節]　“｜”を含まない節。

[And] 論理的and は次の２種類のオペレーショナルには異なった意味をもつand で表わされる。

逐次 and	“&”
並列 and	“,”

名前から明らかなように、“&”は逐次的に実行されるべきand を示し、“,”は並列に実行されるべきand を意味する。

[Or]　論理的orはオペレーショナルにはparallel or を意味する。すなわち、ゴールPとunify 可能な節が並列に探索される。これについては次節でまた述べる。

[Read Only Annotation]　Read only annotation　“？”　は変数に付加することのできる制御情報であり、“X？”のように書く。“？”は論理的な意味を持っていない。従って、同一節にXとX？が出現すれば、これらはunification に対する制御情報を除けば論理的には同一のものである。“？”の付いた変数は変数とは常にunify できるが、それがinstantiate されない間は変数以外のものとunify してはならないことを示す。Read only annotationの付いた変数とunify した変数はこの性質を自動的に引継ぐ。また、X？についてXがnon-variable term にinstantiate されれば、このannotationも自動的に消滅する。Annotationは変数の個々の出現について独立に付加でき、通常は複数プロセスで共有される変数に対して個々のプロセスが付加したりしなかったりする。共有変数にanno

tationを付加したプロセスは、この変数をinstantiate できなくなり、他のプロセス（annotationをもたない）がそれをinstantiate するまで待つことになる。（これについては、2.3で再び述べる）。

2.3 基本計算メカニズム[Clark 81],[Shapiro 83a],[Shapiro 83b]

―― AND 関係の計算 ――

ゴールが与えられたとき、 AND関係の節を使ってそれがどのようにしてサブゴールに展開されるかについて述べる。 AND関係の計算ではプログラムは常にguarded 節のみからなってなくてはならない。上述のように各ゴールはparallel or モード、すなわち、ゴールと unify可能な節が並列に探索されるというモードで実行される。

今、ゴール Ａがあるとする。また、プログラムとして次のclauseがあるとする。

```
A1 :- G1  | B1.
A2 :- G2  | B2.
      .
      .
      .
An :- Gn  | Bn.
```

Giは空であってもよい。ゴールＡに対してＡ１，…，Ａｎは次の３つに分類される。

① 候補節　Ai :- Gi | Bi.
Read only annotationの付いた変数にnon variable term をunify することなしに、ＡとＡｉがunify し、かつ、guard 部Ｇｉが解くことができる場合。

② 待機節　Aj :- Gj | Bj.
read only variableにnon variable term をunify することを除けば、ＡとＡｊがunify し、かつ、guard 部Ｇｊを解くことができる場合。

③ 失敗節　Ak :- Gk | Bk.
それ以外。

ゴールＡは、それが１つ以上の候補節を持てば、その中の１つを選択し、（それを Ai :- Gi | Biとすると）body部 Bi に展開される。原則として、このときの選択のメカニズムは各節が並列に等しい計算資源を用いて探索され、一番早く見つかったもの、すなわち、一番先に commit operatorの処理に到達したものが選ばれる（この並列な節探索のスケジ

ューリングやより一般のスケジューリングは未検討項目である）。従って、この方法により互いに競争し合う節探索を利用した非決定的な処理が可能となる。一度ゴールＡが展開されると、他の候補節の探索は放棄される。この意味でcommit operator “｜”はカット・シンボルとして機能する。

ゴールＡが候補節を全然持たず、かつ少なくとも１つ待機節を持つ場合は、このゴールＡは少なくとも１つの候補節が見つかるか、あるいは可能なすべての節が失敗節に分類されて完全にfailするまで展開が待機させられる。

このリダクションの途中で起きるいかなる変数のbinding も、“｜”を過ぎて他の可能性が放棄されるまでは確定しない。従って、read only annotationのついていない共有変数については、仮にそれがリダクションの途中でnon variable term にinstantiate されても、“｜”を過ぎるまでは、他のプロセスにアクセスさせないようにしている。

2.4 otherwise の導入

Horn論理の世界から見るとメタ的概念である否定の扱いの導入をするために Concurrent Prolog で考慮されているシステム述語 otherwise をそのまま導入する。これはオペレーショナルには Or 並列性に対する制限付の逐次性を導入することに相当する。

ある述語を定義する節集合の中の１つ節が guard部に otherwiseを持つとき、そのotherwise の論理的意味は他の節がヘッドおよび guard部で規定している条件の否定すべての conjunctionである。ただし、この否定は Clark の“Negation as Failure ”[Clark 78]と同様に解釈される。従って、otherwise を持つ節以外のすべての節が失敗節になったときに otherwiseは真となる。

otherwise を持つ節は１つの述語を定義する節集合中に多くても１つであり、かつ、otherwise を持つ節は otherwiseを guard部の中に単独で置かなければならない。

Otherwise の並列環境下での implemetation では１つのゴールで起動されるすべての並列探索プロセスの状態を監視し、otherwise を guard部に持つ節以外のすべての節が失敗節になるのを検出する機構が必要となる。

Otherwise はプログラミングの立場からは “failure”を扱うためのプリミティブなシステム述語である。これを用いて Negation as failure流の “not ” の実現ができる。

```
not(P) :-  p | fail.
not(P) :-  otherwise | true.
```

failはシステム述語であり、常に偽である。第１節と第２節は並行して調べられるが、 otherwiseにより実際には第１節が失敗したときにはじめて第２節が調べられる。もしゴール pが成功すれば、第１節により、not(p)は失敗する。また、もしゴール pが失敗すれば otherwiseにより第２節が選択され、not(p)は成功する。

また、変数が instantiate されるまで suspendするというシステム述語 wait もother wise を使うと実現できる。

```
wait(X) :- wait0(X?).

wait0(foo).
wait0(X) :- otherwise | true.
```

述語 wait は常に述語 wait0 を引数に read only annotation を付して呼出す。述語 wait0 は１引数の述語であり、ゴール wait0(X?)が与えられたとき、第１節および第２節は並行して調べられるが、第２節は guard部に otherwiseを持つので第１節が失敗したときしか選択される可能性はない。ゴール wait0(X?)中の変数 Xが instantiateされていないときは第１節のヘッドとのunification において suspension が起き、従ってゴールは suspendする。変数 Xがinstantiate されているときは、その値が fooであったときは第１節により成功し、それ以外の場合は第２節により成功する。このように、wait0 、従って wait は引数が instantiateされるまで suspendする機能を otherwiseを用いて実現している。

2.5 write early annotation

2.3で述べたように、ゴールと unify可能な節の並列探索においては節の間の相互干渉を防止するためにゴールに含まれる元の変数のbind環境のコピーを節ごとに持っている。これにより、ゴールとヘッドの unificationや guard部の実行の際の共有変数に対する instantiationは commit operatorを過ぎるまでは各節に付随する局所的な環境にとどめておかれ、commit operator を過ぎると同時に外部に公開される。一般に共有変数に対する instantiationはプロセス間の通信と見なせるが、上に述べたことは commit operatorによる通信の発信の実質的遅延を意味している。

しかし、応用プログラムを書く立場からすると、 guard部におけるメッセージの発信がすべて commit operatorを過ぎるまで遅らされるとなると非常に不便である。たとえば guard部のゴールが外部へメッセージを発信し、その返事を受取らない限り先の計算ができないような場合を考えると、これは発信が commit operatorまで遅延される限りデッドロックを引起こす。（発信されていないメッセージの返事を待ちつづけることになる。）また、上述のメカニズムのもとでは、commitされた節の通信情報だけが公開され、 commitされなかった節の発信の延期されていたメッセージは放棄される。すなわち、commitされなかった節の情報を取出そうとしてもその手段は全く存在しない。このことは実行のトレース等をとる立場からは全く不都合である。

write early annotation は以上の問題を解決するために導入されたプリミティブである。write early annotationは read only annotation のように変数の個々の出現に付加できる制御情報であり、論理的意味はない。write early annotationは呼出し側（ゴール）の変数に付加され、そのオペレーショナルな意味は annotation の付加された変数に対するinstantiation （通信）は commit operatorによる発信の遅延を例外的に解除し即時に外部へ公開するということである。

write early annotationは“^”で表わされ、変数Xに次ぎのように付加される。“X^”。“^”の付加された変数のことを write early 変数と呼ぶ。write early annotationのオペレーショナルな性質を以下に列挙する。

1. write early annotationはunification によりinherit する。すなわち、write early annotationの付いた変数と通常の変数が unifyした場合はその変数は write early変数となる。

2. write early 変数が変数を含むデータ構造と unifyされた場合、その構造中の変数も write earlyとなる。

3. write early 変数と read only変数の間の unificationは read only変数がinstantiate されるまで suspendされる。

write early 変数のimplementationは比較的単純である。通常、ゴール中の変数は節の並列探索のために節ごとにコピーが作られ、その変数に対する instantiationは各々の節が各々のコピーに対して行ない、元の変数に対するinstantiation は commit する節が commit のときにコピーと元の変数を unifyすることにより行なう。しかし、ゴール中の変数が write earlyである場合にはそのコピーの生成をやめ、元の変数自身を各節に渡せばよい。各節は元の変数自身を持っているわけだから、その変数に対する instantiationはた

だちに外部に伝わる。ただし、この instantiationは同一の write early 変数を持って並行に走る節探索プロセスすべてにも伝わる。しかし、これはこのimplementationに起因するものではなく、write early という概念そのものが副作用的なものを持っているためである。

write early annotationは非常に強力であり、かつ非常に有用でもあるが、プログラムの意味をわかりにくくするという欠点があるので十分注意して使う必要がある。５章では write earlyによる通信を抽象化して用いる例が述べられている。

2.6 計算の制御に関するコメント

KL1 では、計算資源の管理は有限長バッファ通信[Takeuchi 83] となんらかのスケジューリングのプリミティブにより実現できると考えている。有限長バッファ通信はすでに KL1のベースをなしているConcurrent Prolog で実現されており、これをプロセス制御に用いた応用プログラムも存在している。この意味では有限長バッファは制御のためのプリミティブとは考えていない。しかし、スケジューリングのためのプリミティブについては本仕様書の範囲ではまだ未検討であり、今後の課題として残されている。

3．　集合表現

3.1　集合表現の導入

KL1 の集合は、並列性やメタ述語と結合してプログラミング上の種々の基本概念を実現するプリミティブである。その導入の基本的動機は、次の２つである。

1．　ゴールを満足するすべての解の表現
2．　高階表現の実現

２章で述べたように、 AND関係の計算においては、あるゴールに対して複数個の別解があったとき、処理系はその中の高々１つの解を非決定的に選択して残りの別解をすべて捨ててしまう。これに対し OR 関係の計算においては、あるゴールを満足するすべての解を求める。集合は、この両関係の世界を結合するインタフェースとして導入される[Clark 83]。手続き的意味では、集合は、 AND関係の世界から OR 関係を用いた計算を起動する唯一の手段である。一方、宣言的意味では、集合は AND関係の世界において OR 関係で表わされるゴールを満足する解全体を表わす抽象化されたデータ構造である。言替えると OR 関係の世界の計算は AND関係の世界のおいては常に集合表現の中に押し込められ抽象化されるといえる。

加えて、集合表現は節表現にかわるまたもう１つ述語表現とみなすことができる[McCabe 83] 。この場合には OR 関係に限らず AND関係も集合表現の中で用いられ、λ式が関数を定義するように集合表現が１つの述語を定義すると考えることができる。この集合による述語表現により述語引数のような高階性も集合を引数として渡すことにより実現できるようになる。

これらの場合において集合が表わす概念を一言で言えば、それは『関係』である。ゴールを満足するすべての解とは、このゴールが表わす関係の具体的表現に他ならず、また、述語というのも関係に他ならない。

集合表現は AND関係の世界でのみ用いられ、上述のようにメタ的な概念を表わしている。従って、取扱には注意を要するが、KL1 においては集合を上述の関係という意味において操作する述語を後述の特別なメタ述語に限り、それ以外の述語で操作されるときは集合を単なる項としてしか扱わないようにすることによって解決している。逆に言えば、 AND関係の世界における集合の基本的意味は項であり、後述の特別なメタ述語により操作されるときのみに『ゴールを満足するすべての解』あるいは『述語表現』というような関係としての意味で扱われる。

3.2 集合表現のシンタックス

集合が表わす概念を関係としたが KL1には AND関係とOR関係の２種類の関係があり、集合がどちらの関係を表わすかによってその評価の仕方も異なる。KL1 では集合を表現する方法として外延的表現と内包的表現の両方を採用するが、外延的表現を OR 関係の表現に、内包的表現を AND、OR両関係の表現に用いることにする。

集合の外延的表現では、その集合の表わす関係を満足する項の組(tuple) すべてを列挙することにより表わす。組は項の並びであり、“(”, “)”で囲んで表現される。１つの組が持つ項の数、すなわち、組の長さは固定されており、さらに、１つの集合ではその要素となる組の長さは一定でなければならない。長さｎの組よりなる集合はｎ項の OR 関係を表わしており、従ってメタ的にはｎ項述語を表わしている。

例： 組　　(a,b,c) , (f(a),f(b))

ただし、長さ１の組においては“(”, “)”は省略する。集合の外延的表現は、この組の並びを“{”, “}”で囲んだもので表わされる。

例： 集合の外延的表現

{(a,b,c),(e,f,g),(h,i,j)}

{(f(a),f(b)),(g(a),g(b))}

{red,green,blue}

一方、集合の内包的表現においては、集合の表わす「関係」を述語の並びによって表現する。その表現方法は具体的にはZermelo-Frankel のSet Abstraction に基づいており、一般に次のように書かれる。

{ <組> | <述語の並び> }

<述語の並び> はこの集合の要素が満足すべき関係を述語の形で書きくだしたものである。ただし、 <述語の並び> はすべて AND関係の述語からなるか、すべて OR 関係の述語からなるかのどちらかでなければならず、前者の場合、集合は AND関係を表わし、後者の場合、

集合は OR 関係を表わすという。

例： 集合の内包的表現

```
{X | color(X)}

{(X,Y) | parent(X,Z)&father(Z,Y)}
```

単なる項としての集合が変数を含む場合、その解釈は通常のリスト等の項が変数を含む場合と同じである（集合中の変数は後述のメタ述語による集合の評価のときに特別な意味を持つ）。また、集合どうしも項と項のユニフィケーションという意味でユニファイ可能である。ただし、この場合外延的表現を持つ集合は内部で内包的表現に変換されているものと考える。この変換は次の例のように OR 関係の述語 member 述語を用いて容易に行なわれる。

```
{a,b,c,d}   →   {X | member(X,[a,b,c,d])}
```

集合どうしのユニフィケーションは項どうしのユニフィケーションと見なすので、以下のユニフィケーションはすべて失敗する。

```
{a,b,c,d} ≠ {b,c,d,a}
{(X,Y) | parent(X,Z)&father(Y,Z)}  ≠  {(X,Y) | father(Y,Z)&parent(X,Z)}
```

また、次のようなユニフィケーションは成功する。

```
{a,b,c,d} = {X | Y}          →   X = A,  Y = member(A,[a,b,c,d])
{(X,Y) | parent(X,Z)&father(Y,Z)}  =  {P | parent(Q,R)&S}
                             →   P = (X,Y),  Q = X,  R = Y,
                                 S = father(Y,Z)
```

3.3 集合の評価メカニズム

集合を項としてではなく、前述のメタ的意味での集合として評価するメタ述語は基本的にメタ述語Apply とその特殊形であるメタ述語 Enumerateに限られる。

メタ述語 Apply

Apply(<集合>, <組>).

<集合> の表現については内包的表現でも外延的表現でも良いとする。ただし、外延表現を持つ集合については前述のようにして内部で内包表現に変換するものと考え、ここでは <集合> は次のような内包表現により表わされているものとする。

<集合> = { <組1 > | <述語の並び> }

前述のように、 <集合> は AND関係か OR 関係のどちらかを表わすので、それに対応して、 <述語の並び> も AND関係の述語だけからなるか、 OR 関係の述語だけからなるかのどちらかである。また、拡張あるいはマクロ的意味で、 <集合> として述語名も許す。n引数の述語名 pを Applyの中で使う場合、それは次の意味で使われる。

p = {(X1,...,Xn) | p(X1,...,Xn)}

例

integer = {X | integer(X)}

メタ述語 Applyの評価は次のステップで行なわれる。

(1)集合表現中の変数でその集合外と共有されている変数はすべて新しい変数に rename される。つまり、すべての変数をその集合表現内だけの local 変数にする。

(2)Apply の第2引数の <組> と集合表現中の <組1 > を unifyする。

(3)集合表現中の <述語の並び> を評価する。ただし、この評価は述語の性質に依存しており、述語が AND関係を記述するものであるなら、 AND関係として評価するし、述語

が OR 関係を記述するならば、 OR 関係として評価する。

(3-1) AND 関係として評価する場合

<述語の並び> 中の各述語はすべて guarded節で定義されていなければならない。それらの評価は２章でのべた AND関係の基本計算メカニズムに従い行なわれる。この場合のApply の効果は DEC-10 Prologの call 述語に相似である。

(3-2) OR関係として評価する場合

<述語の並び> 中の各述語はすべて non-guarded節で定義されていなければならない。 <述語の並び> はそのすべての述語を満足する <組> を１つ求めるように評価される。そのような組が１つあれば Applyは成功するし、１つも存在しない場合には Applyは失敗する。可能な組が何種類もある場合にその中のどれが得られるかについては本仕様書では指定しない。

メタ述語 Enumerate

Enumerate(<集合>, <組>, <変数>).

Enumerate は OR 関係を表わす集合だけにしか適用できない。Enumerate の評価は Apply の評価と基本的に同じである。ただし、 3-2の集合の評価時には集合は <述語の並び> を満足するすべての <組> を求めるように評価され、求めたすべての <組> を第３引数にストリームとして返す（そのような組が１つも存在しないときは [] を返す）。ストリームとして生成される <組> の順序には特別な指定はないので、述語の並びの評価順序についても本仕様書の範囲内では特別な規定はしない。

3.4 集合による計算

ここでは、メタ述語 ApplyとEnumerate を用いた種々の計算例について説明する。

(1) membership check

<組> とOR関係を表わす <集合> を入力として、membershipのcheck をする。

例

```
?-  Apply(integer,3)        →    true

?-  Enumerate(integer,3,X)   →    X = [3]

?-  Apply({a,b,c},a)        →    true

?-  Enumerate({a,b,c},a,X)   →    X = [a]

?-  Apply(integer,a)        →    fail

?-  Enumerate(integer,a,X)   →    X = []
```

(2)　generating elements

OR 関係を表わす <集合> を入力として、集合の要素をストリームとして生成する。

```
?- Enumerate({X | color(X)},Y,Z).          →       Z = [red,green,blue]
```

ただし、color は OR 関係を記述している。

```
color(red).
color(green).
color(blue).
```

ここで、{X | color(X)} は{red,green,blue}と書いても良いし、また colorと書いても良い。また、OR関係 parent に対して次のような使い方もできる。

```
?- Enumerate({(X,Y) | parent(X,Z)&parent(Z,Y)},(A,taro),B).

  →    B = [(hideki,taro),(sumiko,taro),(rikio,taro),(ikuko,taro)]


?- Enumerate({(X,Y) | parent(X,Z)&parent(Z,Y)},(A,B),C).

  →    C = [(hideki,taro),(hiroko,jiro),(sumiko,taro),(akikazu,jiro),
```

```
(rikio,taro),(ikuko ,taro), ...]
```

(3) Constraint

```
Apply(S,X?).
```

変数X がinstantiate されていれば、1の membership check と同じ。X がinstantait e されていない場合は Xがinstantiate されるまで計算をsuspend する。従って通常並列 ANDでつないで使う。この機能をもちいれば demonなどが容易に実現できる。

例

```
・・・ ,plus(X,Y,Z),Apply({A | positive(A)},Z?) ,・・・
```

上の例では、Apply は変数 Zがinstantiate されたときに起動され、 Zの値が正であるかどうかをチェックする。

(4) 述語引数／λ-abstaction

集合を引数として受け渡すことにより、述語引数のような高階の機能を実現することもできる。下の例では入力として受取った集合の表わす関係を2回適用する述語を作り出す高階述語 twice を定義している。

```
twice(P,{(X,Y)| Apply(P,(X,Z)) & Apply(P,(Z,Y))}).
```

使用例

```
?- twice({(X,Y) | plus(X,1,Y)},Q) & Apply(Q,(1,A)).  →  A = 3.

?- twice({(X,Y) | twice(X,Y)},Q) &
    Apply(Q,({(X,Y) | plus(X,1,Y)},R)) & Apply(R,(1,A)).  →  A = 5.
```

[Enumerate の実現と制御に関するコメント]

Enumerate の実現方法としては、現在2通りの方法が分かっている。1つは AND関係の計算における AND並列性を用いるものであり[Hirakawa 84] 、他は同じく AND関係におけ

る OR 並列性を用いるもの[Kahn 83] である。Fnumerate は通常１つのプロセスとして用いられ、自律的にストリームを生成する。この自律的生成を制御する手段としては Concurrent Prologで確立された有限長バッファ通信[Takeuchi 83] が有効である。

4．　モジュール化機能構成

4.1　機能検討の指針

本章の目標は，論理型のユーザ言語の上にモジュール化機能を実現するためには，核言語上の基本操作としてどのようなものを用意すればよいかを検討することである。

このため，まず，モジュール化とそれに付随する諸概念，たとえば

- 局所化
- 情報隠蔽
- 外部参照
- 階層化

などを，論理型言語にいかにとりいれるべきか（あるいはとりいれるべきでないか）を，事例調査をつうじて検討し，核言語上の基本操作を設定し，それを用いて，どのようなモジュール化機能がユーザ言語上で実現可能かをみることとする。

ここで用語について若干の説明を行なっておこう。論理型言語におけるモジュールとは，基本的には，一括管理される節の集合のことである。モジュールは，単独でひとつのプログラムとなることもあるが，通常は，他のモジュールと組合わされてひとつのプログラムを構成する。プログラムの側からみれば，モジュールは，管理上の最小構成要素ということになる。管理上の最小構成要素という意味は，たとえばコンパイルという概念が適用できる最小の単位であり，結合編集系の最小入力単位であるということである。モジュールは，ソース形式をとることもあろうし，コンパイルされた形式をとることもあろう。両者のちがいは形態上のものであって，論理上のものではない。

なお，モジュールは通常プログラムの構成要素であるが，それ自身しばしばプログラムと呼ばれることに注意されたい。

4.2　基本機能の選定

事例調査［Bowen 81］，［Clark 82］，［Szerdi　82］，［Caneghem 82 ］，［Nakashima 83］の結果をまとめると，次のことがいえる。

① モジュール化機能の事例の中には，
　ａ．　ソフトウェア工学的側面（プログラムの作成，管理等）の改善を目的とするもの
　ｂ．　世界の階層化，多世界化など，知識の構造化を目的とするもの
　がある。

② ａ．に関しては，何らかの方法で局所化の機能が実現されている。しかし，モジュールを動的に生成し，使用することがあまり考えられていない。

③ ｂ．に関しては，モジュールを動的に生成することを考えた例もみられる。しか

し，情報の隠蔽や，モジュール単位のコンパイルについてはあまり考えられていない。

核言語の上に実現されるであろうユーザ言語においては，問題によって，①の両方の側面をサポートする必要が生じる可能性がある。そこで核言語レベルでは，それらのユーザ言語上の機能を効率よく実現するための基本機能を考えておくべきである。ここで，「効率よく」という条件は重要である。なぜなら，効率を問題にしないならば，モジュール化機能を実現したインタプリタを核言語の上に載せてしまえばよいからである。しかし，モジュール化機能を用いたことによって，プログラムの実行効率が大幅に低下するという事態は避けなければならない。

上の要請に加えて，核言語第１版においては，メタ推論機能の実現［５章参照］という要請がある。メタ推論機能とは，「ある節集合の下であるゴールが証明可能か」という問題を扱う機能である（プログラム言語の用語で言い直すと，プログラム中で扱うデータを，プログラムとして呼出す機能となる）。

この，モジュール化機能とメタ推論機能との間には，「節の集合として，いわゆる内部データベース以外のものを考え，その中の述語を呼出す」という点で深い関連性がある。したがって，この両者は同じ基本機能によって実現することが望ましい。ここでメタ推論の効率的実現のための基本機能をまず考えると，それは，機械語プログラムをデータとみなし，それを作成したり呼出したりする機能であることが直ちにわかる。そこで，この機械語プログラムの作成・呼出し機能を 4.3に示し，それを用いていかなるモジュール機能が実現できるかを 4.4で考察することとする。なお，4.3では名前表の管理方式をどうすべきか－モジュールごとに持つべきか，大域的なものが１枚あればよいか－についても検討する。これも，核言語のレベルで考えるべき重要事項である。

4.3 機械語プログラムに対する操作と，名前表の管理問題

機械語プログラムを処理対象とするために，機械語型というデータ型を用意する。機械語型に対しては，少なくとも次の操作を用意するものとする。

① compile（S，M）

ソースプログラムSをコンパイルし，機械語プログラムMを得る。Sは，節のリストとして与える。

② call（M，G）

機械語プログラムMを用いて，ゴールGを解く。

これらに対して，若干の補足を行なう。

① ソースプログラムSは，たとえば

```
[append([],X,X),
(append([A|X],Y,[A|Z]):-append(X,Y,Z))]
```

という形で与えることも考えられるが，これには次のような問題がある。

- S中の変数の論理的な有効範囲は，それぞれの節で閉じているはずである。つまり，ある節の中の変数は，他の節の中やSの外にある同一の変数とは，本来区別されるべきものである。
- S中の変数は，ユニフィケーションによって値を獲得してはならない性質のものである。

この点からは，S中の節に含まれる変数を，特殊なアトムによって示す方式の方が望ましいかも知れない。この方式には，Sの中に，「プログラムの不定部分としての変数」を導入できるという利点がある。（ 4.4参照）。

② ［Kunifuji 84 ］においては，直接call（M，G）で実現できるメタ推論用述語simulate（World ，Goal）のほかに，その各種拡張形－たとえば，副作用による入出力を扱う能力をもったもの－が提案されている。これらを効率よく実現するためには，call述語に，副作用による入出力の処理機能などを追加しなければならない。

③ compile 述語によって得られる機械語が，もとのソースプログラムに含まれていた情報をすべて保持していなければならないかについては，議論の余地があろう。この問題は，機械語型に対して，上記のcompile とcall以外に，逆コンパイル，節の追加削除などの操作を定義すべきかという問題と関連する。

④ ここでは，ソースプログラムも機械語プログラムも，論理型言語における値として扱っている。したがって，「プログラムを書きかえる」という考えかたは適用できない。あるプログラムに対して節を追加，削除したい場合は，

modify（旧プログラム，変更情報，新プログラム）

というように，別の新たなプログラムを得るという方法で行なう必要がある。

核言語レベルで，いまひとつ検討すべき問題は，名前表の問題である（ここで名前表というのは，名前の同一性（identity）を調べるための表のことである。名前と述語定義の対応づけ等，名前の属性にかかわることは，当面考えないことにする）。名前表をモジュールごとに管理することができれば，名前表の論理的分割が達成でき，また局所的な名前を外部から保護することが可能になる。

しかしながら，名前の局所化機能は，プログラムの動的作成・呼出し機能とはなじみの悪いものである。プログラムの動的作成・呼出し機能と名前の局所化機能を組み

あわせると，名前表の構造が階層的になるが，この場合，

- 外部から入力された名前をどこに登録するか
- 局所的でない名前をいかに指定するか

などの問題が発生し，複雑なプログラムでは管理しきれなくなる危険があるからである。そこで，名前表は，大域的なものがひとつだけあると仮定することとする。

なお，名前と述語定義との対応表は，モジュールごとに保持する必要がある。これは，コンパイルされたモジュールにおいては，述語の数が多ければハッシュ表の形式をとり，少なければ，逐次的判断のための命令列の形式をとることとなろう。

4.4 各種モジュール化関連機能の実現

A． 述語の局所化の実現

あるモジュールが述語定義ｐ，ｑ，ｒ，ｓから成っていて，そのうちのｐとｑだけを外から呼び出しの対象にしたいとしよう（ｒとｓは，ｐとｑの定義のために必要な補助述語とする）。この場合，このモジュールをコンパイルしてできるプログラムの述語表（名前と述語の対応表）には，ｐとｑのみが登録されるようにすべきである。それには，プログラムを，

① 局所化を考えずにコンパイルしたプログラム
② 外部からのｐとｑの呼出しを，①に引きつぐインターフェースプログラム

の２段構えにし，①の方をアクセスできないようにすればよい。この処理をするための３引数のcompile 述語を

compile （Ｓ，Ｐ，Ｏ）

Ｓ：ソースプログラム（節のリスト）
Ｐ：呼出しの対象となる述語の，最も一般的な呼出し形のリスト
Ｏ：機械語プログラム

とすると，これは図４のように定義できる。

```
compile(Source, Public_List, Object) : —
    compile(Souce, Local_Object)&
    make_interface_module(Public_List, Local_Object, Interface)&
    compile(Interface, Object).
make_interface_module([], Local_Object, []).
make_interface_module([P | Ps], Local_Object,
                [(P : —call(Local_Object, P)) | Is]) : —
    make_interface_module(Ps, Local_Object, Is).
```

図4　コンパイル述語

B．　外部参照の実現

あるモジュルの中で，他のモジュールの中にある知識を使用したくなることがしばしばある。たとえば，整列に関するモジュールは，その中で列の要素の大小関係を判断しなければならないが，整列モジュールを汎用のものとするためには，大小関係の知識をその中に置くわけにはゆかない。

このような場合の解決法は３つある。

① 整列述語は，要素の大小関係に関する知識（プログラム）を，引数として受けとるようにする。この知識は，大小比較の際に，call（Order ，X＞Y）のように用いるものとする。

② 大小関係はcall述語によって行なうが，そこで用いるプログラムは，整列述語の引数とするのではなく，整列述語を定めているプログラム中の不定部分とする。そして不定部分を残したままコンパイルする（コンパイラは，少なくとも，call述語の第一引数が不定であるようなプログラムを受けつける能力をもつものとする）。できた機械語は，ソースプログラムと不定部分を共有するが，それに大小関係の知識をユニファイすることによって，完全なプログラムとなる。

③ 大小関係を，call述語を用いずに直接記述する。この場合，このモジュールの知識集合は不完全なものであり，コンパイルしても単独では使えない。そこで，このような不完全なモジュールとそのコンパイルを許す場合は，機械語プログラムどうしの結合編集機能を用意し，それを用いて呼出し可能なプログラムを作成，使用するものとする。

①，②と③では，機械語プログラムの自己完結性という点で大きな差がある。結合編集を前提とした③の方式においては，そのための情報を保持する必要がある。

C.．　階層化の実現

モジュールの階層化を考える場合，個々のモジュールとしては，必然的に非自己完

結的なものを考えることになる。そのモジュールをどのように組合わせて問題を解くかということは，継承機構（inheritance mechanism ）の設計の問題であるが，これは，複数のコンパイルされたモジュールの結合編集方式と深く関係する。さきにB．③で考えたケースにおいては，ふたつのモジュール間の上下関係が問題とはならなかったが，階層化を行なう場合は，下位のモジュールの知識を上位のそれよりも優先使用するようにすることが通常望ましい。したがって，結合編集のための述語もそのような上下概念（あるいは順序概念）を考慮した処理の能力を持つことが望まれる。並列論理型言語の節選択に順序概念を導入した例としては，Concurrent Prolog の otherwise ［２章参照］があるが，この記法は，階層化に伴う順序概念の表現能力という点からは，不十分であると考えられる。

なお，モジュールの結合編集機能を用意する場合は，新たに生成されるプログラムが，結合されるプログラムとできるだけ多くの部分を共有できるように，機械語プログラムの構成を設計すべきである。

5　メタ推論用基本述語

5.1　メタ推論の必要性

ＫＬ１においては，対象知識（Object Knowledge：　実世界の対象（Object）に関する知識）とメタ知識(Meta Knowledge ：「対象知識あるいはメタ知識」の使い方に関する知識）との間で通信を行いながら，与えられた問題を解くプロセスをシミュレートする並列メタ推論（Parallel Meta Inference ）機構を実現するために，メタ述語simulateを導入する。

並列メタ推論方式導入の必要性は，２章や３章で議論されたＡＮＤ並列やＯＲ並列を用いて出現する並列プロセス間の通信や同期以外の並列プロセスの管理（モニタリング，スケジューリング，エディティング，デバッギング，等）を支援する必要性，および４章で導入されたモジュール構造を十二分に使いこなす機能導入の必要性から生じた。

導入された並列メタ推論支援用メタ述語は，基本的に並列実行環境下での知識プログラミング能力を飛躍的に高める。その結果ＫＬ１は，階層的モジュラー・プログラミング・システム，知識ベースへの知識同化システム，並列プロセスのエディト／デバッグ，並列現象のシミュレーション，および並列問題解決といった知識情報処理志向の応用分野の知識を記述する知識プログラミング言語／システム作成のための有力なツールとなる。

以上のような必要性に応えるため，5.2以降では並列メタ推論概念の明確化，各種並列メタ推論方式の提案，並列メタ推論用メタ述語の実現法の検討，およびその応用可能性について述べることにする。

5.2　並列メタ推論

5.1で考察したようにメタ推論とは，「メタ知識を用いた推論」のことである。ここにメタ知識とは，「対象知識の使い方に関する知識」あるいは「（それによって再帰的に定義される）メタ知識の使い方に関する知識」のことである。対象知識としては，ＫＬ１（あるいはその部分集合を仮にＫＬ１と呼ぶこともあるが，）で表現可能なホーン論理の節集合のみ取扱うことにする。ＫＬ１上で並列メタ推論を実行するには，ＫＬ１処理系の使い方に関する知識を表現し利用するメタ述語を，ＫＬ１ユーザに提供する必要がある。そこでＫＬ１でゴールを解く過程を反映するメタ述語simulateを導入し，各種の並列メタ論理方式をその表現利用技術に還元し，並列メタ推論機能をＫＬ１ユーザに解放することにする。

メタ知識と対象知識を単一の論理プログラミング言語で表現し，両知識を融合［Bowen 81，Kowalski 81 ］した形式で利用することにより，対象知識の更新を含む高度なメタ推論機能を簡単に実現できる。逐次実行型論理プログラミング言語DEC-10

Prologあるいはその部分集合の場合，このことはdemo述語という概念の有用性として，文献［Bowen 81, Kowalski 81 ］で提案され，文献［Kunifuji 83 ， Miyachi 84 ，Kitakami 84 ］でその実現法，各種逐次メタ推論方式の提案，およびそれら諸方式の応用例の開拓がなされた。そこでここでは，ＫＬ１上での各種並列メタ推論方式を提案し，並列メタ推論用メタ述語simulateの実現法に関する諸検討を行い，simulate述語の応用可能性等について述べる。

5.3 並列メタ推論の方式［Kunifuji 84 ］

simulate述語に基づくメタ推論を実行するため，我々は知識表現の対象としている対象知識の世界，すなわち対象世界，とそれを操作対象とし管理しているメタ知識の世界，すなわちメタ世界，とをそれぞれ論理的に閉じた世界として構築する。ここに，論理的に閉じているとは，対象世界のfailがメタ世界のfailに波及しないことを意味する。これによって，ある世界の内と外との間の論理的つながりを，write early による通信用論理変数を除いて，切断することができる。

ここで，提案される最も基本的な並列メタ推論用simulate述語は，次のような形式をしている。

(1) simulate (World,Goal)

simulateは，ある世界World において，与えられたゴールGoalを解くプロセスをシミュレートする。

世界World をいかに実現するかは，本述語の設計方針，実現法に依存する。(1)をベースにして，simulate述語にさらに次のような機能を追加することが考えられる。

(a) ゴールを解く過程で，Woldに対して節の追加，削除を（assert/retractによって）命令的に行うことが考えられる。この場合，引数World そのものを書換えるのではなく，World の変更結果を返すような別の引数を用意すべきである。

(b) 対象世界とメタ世界との通信，すなわち入出力を，命令的にではなくストリームによって行うならば，それらの入出力チャネルをwrite early 変数としてGoalの引数に含ませておけばよい。

(c) simulateの実行過程そのものから抽出されるメタ情報（たとえばproof tree）を利用可能ならしめるためには，simulate述語に，その取出しのための引数を追加する必要がある。これは，対象世界のsuccess/fail情報の取出しにも利用できる。

(d) simulateの実行プロセスに対して，制御情報（たとえばdepth-first とbreadth-first の切替え）を与えることが考えられる。そのためには，simulate述語に制御情報入力用の引数を追加すればよい。

以上の機能拡張をほどこしたsimulate述語の一般形は，たとえば次のような形式をしている。

(2) simulate(World,NewWorld,Goal,Result,Control)

このsimulate述語は，ある世界World において，他の世界とwrite early 変数を用いた通信を行いながら，与えられたゴールGoalを制御条件Control 下で解くプロセスをシミュレートする。シミュレーションの結果，世界World は新世界NewWorldに更新されると同時に，ゴールを証明していくプロセスそのものから，必要なメタ情報Resultが順次抽出されていく。

なお，対象世界として，内部データベース自身を用いる場合は，(1) の第１引数World は省略できる。この場合“ＫＬ１ interpreter in ＫＬ１”という概念に相当する。

このsimulate述語(2) の各引数のもつ役割を明らかにするため，多少の補足説明をすると，次のようになる。

(a) World,NewWorldについて：　　ＫＬ１あるいはその部分集合からなる世界

World NewWorldの構成法はsimulate述語(2) の実現法を左右する。World 構成法として，次のような方法がある。

(a.1) World として世界の名前を指示する方法。世界の名前と実体との関連付けは，ＫＬ１処理系に則して与えられる。

(a.2) World として世界の実体を与える方法。節集合の実体として，節のリストを直接与える方法やコンパイルド・コードを与える方法が考えられる。

またWorld からNewWorldへの変更には，（assert/retractといった）副作用を用いる方法と副作用を用いない方法の両者が考えられる。

(b) モジュール構造との関連について：　　ある知識プログラミングをする際，しばしばひとまとまりの何らかの意味のつながりのある対象世界やメタ世界の構造体を取扱う必要がある。この構造体は，４章で導入されたモジュール構造を用いて表現される。モジュール構造の表現法には，基本的に次の二種類がある。

(b.1) 構造を知識としてあらかじめ与えておく方法。KRL ，LOOPS ，やHandala［Furukawa 83 ］といった知識表現言語で採用されているように，part-of 関係やis-a関係といった知識として，その構造を静的に与える方法である。

(b.2) 構造をプログラム実行時に与える方法。Prolog／KRのwith［Nakashima 83］のように，その構造をプログラムの引数として実行時に動的に与える方法である。

(c) Goalについて：　　World が対象知識の世界ならば対象世界の言語で記述されたゴール，メタ知識の世界ならばメタ世界の言語で記述されたゴールである。

GoalはWorld で規定された対象世界やメタ世界のクラスを逸脱してはいけない。

(d) Resultについて：　　simulate述語(2) によって，その世界World で与えられたゴールGoalを解くプロセスを実行し，（success ／failの両パターンを含む）proof treeの生成過程そのものから，特定の問題を解くのに必要なメタ情報Resultを抽出し，その結果を第４引数へ反映できる。Resultとしては，次のようなものがしばしば利用される。

(d.1) 証明の結果を返すResult

(d.2) 証明過程そのものを返すResult（無限に走る計算プロセスの途中結果を，メタ情報Result経由で利用できることに注意せよ。）

(e) Control について：　　ホーン論理に基づく論理プログラミングは，その本来の主旨からすれば，“Algorithm ＝Logic ＋Control ”［Kowalski 79 ］という哲学をもっていた。それ故，Logic とは分離した形態で，Control 情報を付与することが考えられる。その際，次のような諸点に注意されたい。

(e.1) 多数の並列プロセスに対する計算資源の割付のコントロール，すなわち並列プロセスのスケジューリングが最も重要なControl である。

(e.2) Logic における証明とは別枠にあるproof treeの探索の仕方（たとえば，depth-first searchやbreadth-first searchの切替え，探索空間範囲の制限，探索打切り条件の指定，等）を変えるControl が，ある種の応用では有効である。

(f) write early 変数について：　　write early annotationの付加された変数に対するインスタンシェーションは，コミット・オペレータによる発信の遅延を例外的に解除し即時に外部へ公開する。その結果，write early annototionは非常に強力な機能を提供するが，同時にそれはプログラムに非論理的なものを導入する危険がある。ここでは，メタ述語simulateがwrite early 変数の非論理的通信を外界に対して論理的に見せかける役割を持っていることを例を用いて説明する。

```
f(X)  :- X = [hello | Y ] & fail | true.
```

上のプログラムをゴールf(X)で起動する場合，そのプロセスはガード部で必ず失敗するため，変数Ｘにいかなる項もユニファイされない（ガード部で行なわれたユニフィケーションはガード部の失敗により取り消される）。しかし，同じプログラムをゴールｆ(X^）で起動する場合，そのプロセスはガード部で失敗するにもかかわらず，変数ＸがWrite early annotationを付加されているために，Ｘは［hello ｜Ｙ ］にインスタンシェートされたまま残る。このことは，変数Ｘに値を書き込んだプロセスが失敗して消滅したにもかかわらずその書き込んだ値だ

けは残るという非論理的副作用と同じ効果を残す。simulate述語はこのような write early 変数に値を書き込むプロセスが失敗して消滅しかねないときに，このプロセスを“simulateが生成する論理的に閉じた対象世界”に押し込めることにより，みせかけ上，write early 変数に値を書き込んだプロセスを失敗させないようにすることができる。これはsimulate（f(X^),Result）というゴールを起動することにより実現される。なぜならsimulateはその中で行なわれる計算の結果を第２引数に返し，決して失敗することのない述語だからである。simulateとwrite early 変数のこのような使い方は，write early 変数の強力な通信の一種の抽象化と考えることができる。

5.4 実現法の検討

並列メタ推論用simulate述語(1) を，ＫＬ１の親言語となっているConcurrent Prolog(CP) 上でインプリメンテーションすることを考える。メタ述語simulateの実現過程を通して，その実現のためのプリミティブを明らかにしていく。世界の構成法はWorld として世界の名前を指示する方法を採用する。世界を構成する節は，共通の世界名を冠して，

世界名（(A :- G1#...#Gm | B1#...#Bn)）.
（ただし，＃は逐次and “& ”あるいは並列and “, ”）

という形式で，ＫＬ１の内部データベースにあらかじめ登録されているものとする。すなわちsimulate述語の第１引数World は，世界のユニークな識別名を与えるものとする。

上述のような節構成法の下，文献［Shapiro 84］にあるメタ述語“reduce(Goal)”，すなわちConcurrent Prolog interpreter in Concurrent Prolog，を参考にしながら，simulate述語（１）を実現すると，例えば次のようになる。

```
simulate(World,true).                                          ①
simulate(World,(A,B)) :—true |
        simulate(World,A?),simulate(World,B?).                 ②
simulate(World,(A&B)) :—true |
        simulate(World,A?)&simulate(World,B?).                 ③
simulate(World,A) :—
        w_clauses(World,A,Clauses) |
        resolve(World,A,Clauses,Body),
        simulate(World,Body?).                                 ④
simulate(World,A) :—cpsystem(A) | A.                           ⑤
```

```
resolve(World, A, [(A : -(Guard | Body)) | Cs], Body) : -
        simulate(World, Guard) | true
resolve(World, A, [C | Clauses], Body) : -
        resolve(World, A, Clauses, Body) | true.

w_clauses(World, A, Clauses) : -
        wait(A, A1)&cpsystrem(A1) | fail.
w_clauses(World, A, clauses) : -
        wait(a, A1) |
        prolog((P = ..[World, (X : -Y)],
        bagof((X : -Y), (P, unif(A1, X)), Clauses))).

unif(X, Y) : -
        (var(X); var(Y)), !, not(not(X = Y)).
unif(X?, Y) : -!,
        nonvar(X), unif(X, Y).
unif(X, Y?) : -!,
        nonvar(Y), unif(X, Y).
unif([X | Xs], [Y | Ys]) : -!,
        unif(X, Y), unif(Xs, Ys).
unif(X, Y) : -
        X = ..[F | Xs], Y = ..[F | Ys], unif(Xs, Ys).
```

simulate述語の各節のもつ手続き的意味は，以下の通りである。

① あるWorld においてプロセスtrueをシミュレートすると，正しく停止する。

② あるWorld においてプロセス（A，B）をシミュレートするには，そのWorld においてプロセスAをシミュレートすることとプロセスBをシミュレートすることを並列に実行しなさい。

③ あるWorld においてプロセス（A&B）をシミュレートするには，そのWorld においてプロセスAをシミュレートすることとプロセスBをシミュレートすることを逐次的に実行しなさい。

④ あるWorld においてプロセスAをシモュレートするとは，もしその世界World においてプロセスAと統一可能なヘッドをもつあらゆる節のリストをClauses とすれば，そのClauses からガード部分をシミュレートした結果が真となるような節のボディ部分Bodyを取り出し，そのWorld においてそのBodyをシミュレートすることを並列実行しなさい。

⑤ あるWorld においてプロセスAをシミュレートするとは，もしAがConcurrent Prolog（あるいはＫＬ１）のシステム述語ならば，そのAを直ちに実行しなさい。

resolve 述語，w_clauses 述語およびunif述語のもつ意味の詳細は省略する。上記simulate述語のサンプル・コーディングから分かるように，resolve と w_clauses とunif，特に w_clauses の高速なインプリメンテーション法が提供できるかどうかが，simulate述語の処理性能に大きく利いてくる。このことは最初に指摘したように，World の実現法に強く依存し，その妥当性検討が極めて重要である。

なおsimulate述語の実現イメージを正確に伝えるため，素数生成を素材にした簡単な実行例を示しておく。

［例題　１］　素数生成プログラム実行例

〔プログラム〕

$$
\begin{aligned}
&w((primes:-true\mid\\
&\qquad integers(2,I),sift(I?,J),outstream(J?))).\\
&w((integers(N<[N\mid I]):-N1:=N+1\mid\\
&\qquad integers(N1,I))).\\
&w((sift(([P\mid I],[P\mid R1]):-true\mid\\
&\qquad filter(I?,P,R),sift(R?,R1))).\\
&w((filter([N\mid I],P,R):-0=:=NmodP\mid\\
&\qquad filter(I?,P,R))).\\
&w((filter([N\mid I],P,[N\mid R]):-0=\backslash=NmodP\mid\\
&\qquad filter(I?,P,R))).\\
&w((outstream([X\mid S]):-writesp(X)\mid\\
&\qquad outstream(S?))).
\end{aligned}
$$

〔実行例〕

```
|: -cp  simulate(w,primes).
2  3  5  7  11  13  17  19  23  29  31  37  41  43  47   ...
```

5.5　応用可能性

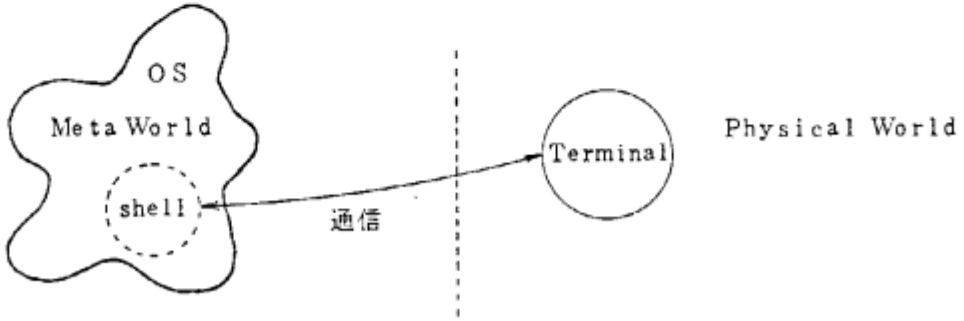
ＫＬ１の一機能として提案したsimulate述語の応用例として，次のようなものが考えられる。

(a) 知識プログラミング言語Mandala ［Furukawa 83 ］における多重相続の実現
(b) 知識同化を含む知識ベース管理への応用
(c) ＯＳの核“shell ”の実現
(d) 並行プロセスの知的なエディタ，デバッガへの適用
(e) 並列に動作するマシン間のインタフェース合せ
(f) 協調問題解決への応用

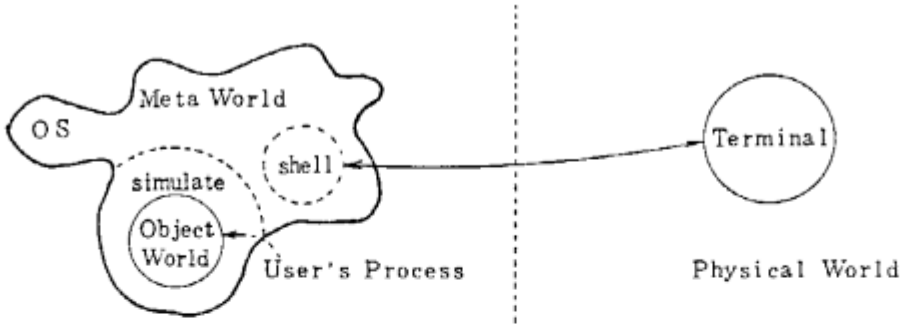
5.3で導入されたsimulate述語(2) の応用可能性を示唆するため，UNIX-like のＯＳの核“shell ”の実現例を述べる。

［例題　２］　ＯＳ核“shell ”の実現

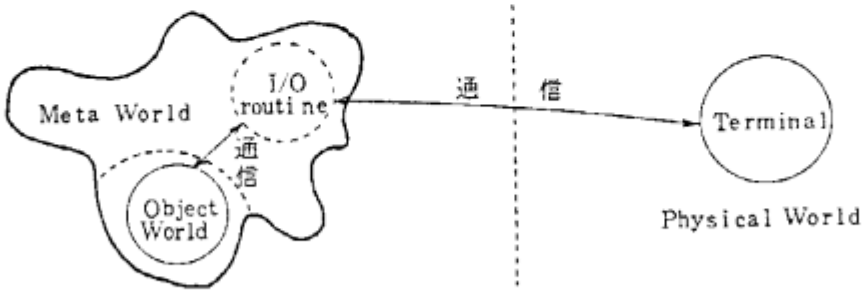
OSとユーザとの通信を行ないながら画面に表示していくshell の例を考える。shell プログラムはタイプ(2) のsimulate述語"simulate(Goal,Result)" を利用している。ここに引数World,NewWorld,Channelは、簡単のため省略するものとする。このsimulate述語は、「メタ世界にあって与えられたゴールGoalに対してオブジェクト世界をひとつ生成し、その中でゴールを解き、その結果をResultに返す」メタ述語である。以下、shell プログラムに至るシナリオを述べる。



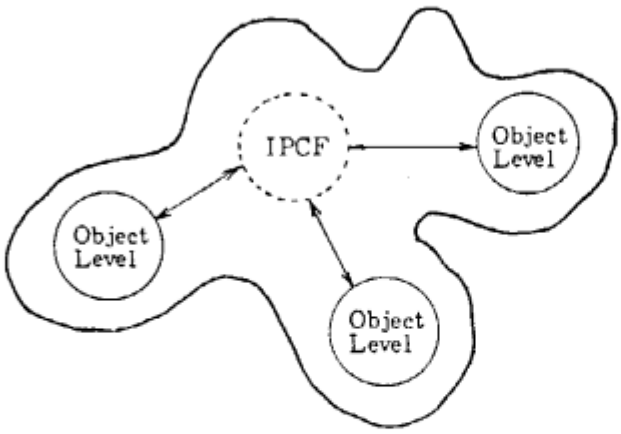
最初ユーザ・プロセスはなく、メタ・レベルにOSのユーティリティのみが存在しているとする。端末からユーザのプロセス(-Goal)をひとつ受取り、それを解くユーザのプロセスをsimulate(Goal,Result) で生成する。



ユーザ・プロセスのI/O ルーティンもOSを介して行なわれる。



simulate で別々に作られたユーザ・プロセス間の通信もＯＳを介して行われる。



IPCF : Interprocess Communication Facility

以上のシナリオのもとに簡単なshell の例をサンプル・コーディングすると，次のようになる。

shell(X,Y)::ユーザからコマンドのストリームＸを受取り，それらのゴールををオブジェクト世界を創成しながら解き，ストリームＹに答を表示するプログラム

ただしコマンドは，次の二種からなる。

X　　:: ゴールＸを解く

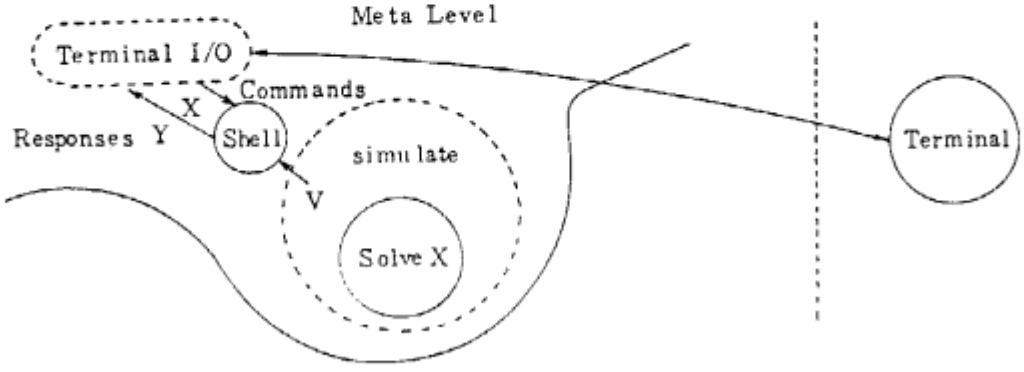
fork(X)　:: バックグラウンドでゴールＸを解く

$$shell([X \mid Commands], [V \mid Responses]) : -$$
$$simulate(X, V) \mid shell(Commands?, Responses).$$

$$shell([fork(X) \mid Commands], Responses) : -$$
$$(simulate(X, V)\&R = [background\_responses(V)]),$$
$$merge(R?, Rs1?, Responses), shell(Commands?, Rs1).$$

ゴールは次のように記述される。

$$: -terminal(Commands, Responses?), shell(Commands?, Responses).$$



最後に、メタ推論に関する今後の課題は、次のような諸研究の成果を踏まえて、並列メタ推論のための言語仕様を洗練し、その実現法を検討することである。

(a) 高速な多世界データベース・アクセス機能の実現に関する研究
(b) メタ情報 ((2) のresult) 抽出法に関する研究
(c) 制御条件 ((2) のControl) 付与の方法に関する研究

[References]

[Bowen 81a] D.L. Bowen: DECsystem-10 PROLOG USER's MANUAL, Department of Artificial Intelligence, University of Edinburgh, Dec. 1981.

[Bowen 81b] K.A. Bowen and R.A. Kowalski: Amalgamating Language and Meta Language in Logic Programming, Tech. Rep. of School of Computer and Information Sciences, University of Syracuse (1981).

[Caneghem 82] M. Van Caneghem: PROLOG II Manuel D'Utilisation, Groupe Intelligence Artificielle, Faculte des Sciences de Luminy, Marseille (1982).

[Clark 78] K.L. Clark: Negation as failure, in Logic and Data Bases, Gallaire and Minker eds., Plenum Press, New York, (1978) pp293-322.

[Clark 81] K.L. Clark, S. Gregory: A Relational Language For Parallel Programming, Proceedings of the ACM Conference on functional Programming Languages and Computer Architecture (1981).

[Clark 82a] K.L. Clark, J. R. Ennals, and F. G. McCabe: A Micro-PROLOG Primer (2nd ed.), Logic Programming Associates Ltd., London, 1982.

[Clark 82b] K.L.Clark, F.McCabe, S.Gregory: IC-Prolog language features, in Logic Programming, ed. K.L.Clark, S.A.Tarnlund, Academic Press (1982).

[Clark 83] K.L. Clark, s. Gregory: PARLOG: A Parallel Logic Programming Language, Research Report DOC 83/5, May (1983).

[Furukawa 83] K. Furukawa, A. Takeuchi, and S. Kunifuji, **Mandala** : A Knowledge Representation System in Concurrent Prolog, Information Processing Society of Japan,Preprints of WG on Knowledge Engineering and Artificial Intelligence, Nov. 1983.

[Furukawa 84] K. Furukawa, A. Takeuchi, S. Kunifuji : **Mandala**: Knowledge Programming and System in the Logic-type Language, Institute of New Generation Computer Technology, ICOT TR-043, February 1984.

[Hirakawa 84] H. Hirakawa, T. Chikayama: Eager and Lazy Enumerations in Concurrent Prolog, ICOT TM-0036 (1984).

[Kahn 83] K. Kahn: Pure Prolog Interpreter in Concurrent Prolog, Presentation in ICOT 1983.

[Kitakami 84] H. Kitakami, S. Kunifuji, T. Miyachi, and K. Furukawa: A Methodology for Implementation of A Knowledge Acquisition System, Proc. of the 1984 International Symposium on Logic Programming, Atlantic City, U.S.A., Feb. 6-9, 1984 (TR-037).

[Kowalski 79] R.A. Kowalski: Algorithm = Logic + Control, CACM, Aug. 1979.

[Kowalski 81] R.A. Kowalski: Logic as a Database Languages, Tech. Rep. of Department of Computing, Imperial College (1981).

[Kunifuji 83] S. Kunifuji, M. Asou, A. Takeuchi, T. Miyachi, H. Kitakami, H. Yokota, H. Yasukawa, and K. Furukawa: Amalgamation of Object Knowledge and Meta Knowledge by Prolog and its Applications, Preprint 30-1 of Knowledge Engineering and Artificial Intelligence Working Group of the Information Processing Society of Japan, June 1983 (TR-009).

[Kunifuji 84] S. Kunifuji, A. Takeuchi, K. Furukawa, and K. Ueda: Meta Inference and Its Applications–Meta predicate simulate for parallel meta inference–, 28th National Conference of the Information Processing Society of Japan, the University of Electro-Communications, March 1984 (TM-0039).

[McCabe 83] F.G. McCabe: Lambda Prolog, Logic Programming Workshop '83, Portugal (1983).

[Nakashima 83] H. Nakashima: A Knowledge Representation System : Prolog/KR, METR 83-5, Dissertation of Dept. of Mathematical Engineering, University of Tokyo, Feb. 1983.

[Shapiro 83a] E.Y. Shapiro: A Subset of Concurrent Prolog and Its Interpreter, ICOT Technical Report TR-003 (1983).

[Shapiro 83b] E. Shapiro, A. Takeuchi: Object Oriented Programming in Concurrent Prolog, New Generation Computing Vol. 1, No. 1 (1983).

[Shapiro 84] E. Shapiro: Systems Programming in Concurrent Prolog, 11th Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages, Salt Lake City, Utah, Jan. 1984.

[Szerdi 82] P. Szerdi: Module Concepts for PROLOG, Proc. of Workshop on Prolog Programming Environments, Linkoping, 69-79, 1982.

[Takeuchi 83] A. Takeuchi, K. Furukawa: Interprocess Communication in Concurrent Prolog, Proc. of Logic Programming workshop '83 (1983).

[Turner 81] D.A. Turner: The Semantic Elegance of Applicative Languages, Proc. of Functional Programming Languages and Computer Architecture (1981).