TM-0290

PIMOS 第一版概念仕様書

ICOT 第四研究室編

March, 1987

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

PIMOS第一版概念仕様書

第０．１版

1987.3.4

ICOT第四研究室　編

改定履歴

１９８７年２月１９日	第０版作成	
１９８７年３月４日	第０．１版作成	６．３節追加

目　次

1．はじめに

並列推論マシンPIM の研究は、プロジェクトの中期にはいって増々熱気を帯びてきている。本書はPIM のオペレーティング・システムPIMOS(Parallel Inference Machine Operating System)について、その概念仕様を述べるものである。

並列マシンと並列ソフトウェアとはシステムを稼働させる上でちょうど車の両輪の関係にあるべきものであり、またそれぞれの研究が進むことにより相互に他の研究を促進させる効果をもつ。しかるに、並列ソフトウェアの研究は並列マシンの研究に比べ大きく遅れをとっていた。PIMOS は、並列推論マシン・システムを満足に稼働させるためのオペレーティング・システムであると同時に、並列論理型言語KL1 で記述される初の本格的実用プログラムとなるべきものでもある。このことはすなわち、PIMOS が実用的ソフトウェアを目指すと同時に、研究項目が山積する並列ソフトウェアであることを意味している。したがってPIMOS の研究開発は、その構成部分により研究phase と開発phase の部分があり、それらを平行して進めなければならないところに難しさがある。PIMOS は研究開発のターゲットとして、プロジェクトの最後に完成する1000台PIM を目標とするが、そこには研究要素数多く含まれることから途中での大幅改訂が必要になる場合も起こり得る。この意味から本仕様書の名称は、特にPIMOS 第1版概念仕様書と呼ぶ。

PIMOS は、その名称が定められてから、いく人もの研究者によりイメージの呈示が重ねられてきたが、その共通イメージあるいは、それから先の議論の土台となる確かな概念仕様というものがまとめられていなかった。本書は、従来の議論をふまえた上で、概念仕様の共通項をまとめるとともに検討課題を明らかにし、今後の機能検討、詳細検討のためのバイブルとして役立つ概念仕様書を目指すものである。バイブルを目指すゆえに、記述方法は機能仕様のイメージ呈示に重点を置き、敢えて詳細記述は避けることを原則とした。但しイメージ呈示を助ける意味あいから、機能項目毎に実現手法のイメージを併記した部分も随所にある。また実現手法に関するある程度細かなアイデアがあるものについては最後の章にまとめた。本書は、PIMOS の機能仕様、詳細仕様の検討に加わる人々およびKL1 処理系の開発者を主な対象読者とする。もちろんPIM の検討者やPIMOS のユーザとなる人々にも広く読んでいただき議論に参加していただくことを希望するものであるが、記述内容の説明が不十分な部分もあることをあらかじめお断わりしておく。

本書は次の各章から構成される。第2章ではPIMOS の定義、PIMOS のプロジェクトにおける開発範囲、およびPIMOS がどのようなOSを目指すかについて述べる。第3章ではPIMOS が搭載されるべきターゲットマシンの形態と、システムのラフな実現イメージを述べる。第4章ではPIMOSを記述するために特に重要となるKL1 の言語機能と、言語仕様と密接な関係にあるOS実現上の基本概念について述べる。第5章ではPIMOS を構成する各々の機能について各論を述べる。概要の節では、PIMOS の全ての機能を横断して眺め概要をまとめる。そして次節以後では、機能毎に各論を述べる。ここでは実現階層が毎なるものでも、機能に着目してまとめて記述する。第6章では、実現方式に関する様々なアイデアやコメントに関して提案されているものすべてを掲載する。概念仕様書としては付録的な章であるが、検討を進める上で有用となろう。

2．PIMOSの定義とその特徴

2．1　PIMOSの定義と開発範囲

PIMOS とは並列推論マシン用のオペレーティング・システムであり、対象としているマシンはマルチPSI 第2版、 100台PIM（中期PIM ）、1000台PIM（後期PIM または5Gマシン）である。但し1000台PIM に搭載されるのはPIMOS 第1版の改訂版となろう。

狭義のPIMOS は、並列推論マシンのための資源管理(CPU 資源やメモリ資源)や負荷分散制御の機能を中心としたオペレーティング・システムであり、並列推論マシンの本体上で動作する。プログラミング・システムも含まれるが、SIMPOSに比べてその比率はずっと少ない。

一方PIMOS の開発項目には、上記の狭義のPIMOS（レジデントPIMOS ）に加えて次の2項目が含まれる。第1はフロントエンド機能であり、これは本体といっしょになって動作するものである(入出力デバイス・ハンドラやメンテナンス機能等)。第2はクロス・システムであり、これは本体と全く独立して動作するものである(コンパイラやシミュレータ等)。

ネットワーク・ハードウェアの制御や、効率の良いゴール・スケジューラ、GCなど、多くの機能がファームウェアで実現される。これには非論理的な操作をKL1 の世界には論理的に見せるという意味と、効率を高めるという意味がある。従来マシンではOSが行なっていた仕事のかなりものもが、KL1 の言語実行メカニズムに取込まれているという点に注意する必要がある。

PIMOS の利用形態は、シングル・ユーザ、マルチ・タスクを中心に考える。すなわち、1人のユーザはシェル・コマンドを用いて複数の独立タスクを起動できる。しかし、複数ユーザがタイムシェアリングで公平にシステムを利用するといった厳密な意味のマルチユーザ・サポートは、まじめには考えないものとする。これは、大規模マルチプロセッサ・システムの公平なマルチユーザ化がきわめて難しいことによる。

一時に1人のユーザしか利用できないにしても、マシンの使用権が得られたときはそれぞれのユーザが異なる端末からマシンを使いたいであろうし、あるいは1台のマシンを時分割でなく物理的に分けて複数人で使いたいこともあろう。あるいは、公平ではないサポったマルチユーザなら実現が可能かもしれない。いずれにしても、何セットもあるわけではない並列推論マシンを複数の利用者間でなるべくむだなく使える何らかの方法を実現する。

２．２　ＰＩＭＯＳの特徴

ＰＩＭＯＳは以下のような特徴を持つシステムとして設計した。

（１）ＫＬ１で記述された純粋なロジックＯＳ

論理型プログラミングの機能に基づいて，できる限り超論理的な機能を用いずに記述する。ＰＳＩ上のオペレーティングシステムＳＩＭＰＯＳはＫＬ０に基づいたオブジェクト指向言語ＥＳＰで記述されていたが，全体の構成は論理型というよりオブジェクト指向の概念に基づいており，プロセス間の通信に関わる部分や低レベル部分の記述などはＫＬ０の超論理的な機能を多様している。刻々と状態の変化する並列的な計算過程を表現するにはＧＨＣはＰｒｏｌｏｇよりはるかに優れており，超論理機能をほとんど用いることなく，オペレーティングシステムとして必要とされる機能の（あるレベル以上の部分の）すべてを記述できるものと考えている。

（２）本格的なＯＳとして必要な機能を網羅

ＰＩＭＯＳでは本格的なＯＳとして実用に耐えるために必要な機能は一応網羅する。工程上，実際のインプリメンテーションでの実現手段は一部フロントエンドプロセサ（ＰＳＩなど）のＯＳの機能に依存することになろうが，本質的には本来すべてがＫＬ１で書けるような仕様に設計する。

（３）集中型単一ＯＳ

ＰＩＭＯＳはマルチプロセサ用のＯＳではあるが，いわゆる分散ＯＳではない。分散型ＯＳでは，独立して動作するプロセサごとのＯＳを組み合わせ，全体としても一体的に動作するようにするわけであるが，ＰＩＭＯＳでは一台一台のプロセサは独立しては意味ある動きはできない。ソフトウェア，つまりＯＳとその上で動作するアプリケーションプログラムに関する限り，対象となるマルチプロセサシステムは一台の計算機とみなされる。

（４）実験システムのＯＳとして高い水準の使い勝手

実験的なシステムのＯＳではあるが，Ｍｕｌｔｉ－ＰＳＩやＰＩＭのハードウェアや，並列アルゴリズムの実験／評価がスムースに行なえるように，ある程度使い勝手のよいシステムをめざす。フロントエンドにＳＩＭＰＯＳが乗ったＰＳＩを使うことによって良好なユーザインタフェースを実現する。とくに，デバグの困難な並列言語を扱う，使いやすいデバッガの開発は重要な課題のひとつである。ハードウェアや並列アルゴリズムの実験／評価を支援するために，各種の評価データを収集する機能も用意する。また，こうした実験／評価をスムースに行なうためには，正常な動作が正常に行なえるだけではなく，誤った使い方に対するＯＳの堅牢性も重要である。

（５）クロスシステムの充実

Ｍｕｌｔｉ－ＰＳＩにしてもＰＩＭにしても実験機であるし台数も限られているので，ソフトウェアの開発にそのまま用いるのは得策でない。そこで，ＰＩＭＯＳではその機

能の一部として，ＫＬ１プログラム開発環境をホスト計算機（たとえばＰＳＩ）の上に提供する。クロスシステムは単にＫＬ１の論理的なレベルでの実行が可能なだけではなく，ゴールの実行順序をランダマイズするなどして実行順序に依存するテストや，複数プロセサに負荷を分散させる方針の良否の判断に必要な計測値を集める機能など，可能な限りのツールを用意する。プログラム開発はクロスシステム上で，最終的な実行と計測データ集めのみを実機上で，というのが理想型であろう。

３．ターゲット・マシンと実現イメージ

３．１　ターゲット・マシンの形態

PIMOS が対象としているマシンは、マルチPSI 第2版とPIM であるが、どちらも次のような共通のマシン形態を有することを仮定してPIMOS を設計している。

(1) フロントエンド・マシンがあること

並列推論マシンの本体にはPSI 相当のフロントエンド・マシンが接続されており、本体からの要求により入出力デバイスの制御をしたり本体のメンテナンス・サポートを行なう。フロントエンド・マシンはLAN にも接続されている。

(2) ネットワーク結合によるマルチプロセッサ構成であること。

複数のPE(プロセッシング・エレメント)またはクラスタ(複数PEが共有メモリにより密結合されたもの)がネットワークで結合された構成である。ネットワーク結合の意味はPEまたはクラスタ内で閉じる処理に比べてネットワークを渡る処理の方がある程度以上コスト高となること、また大規模システムになるとネットワーク上にも距離に見合った通信コストが存在するという意味である。このようなシステムでは、プロセス間の通信のローカリティを高めることが重要となるわけで、PIMOS の負荷分散ではそれを目指している。

３．２　システムの実現イメージ

PIMOS には、本体上のレジデントPIMOS 、フロントエンド機能、クロス・システムが含まれる。レジデントPIMOS は、多くのOSプロセスから成る1個のOSであり、それらのプロセスは、並列推論マシンの本体上に分布して存在する。ユーザ・タスタは、OSのサブ・タスクとして生成され、本体上で実行される。

フロントエンド機能のうち、入出力デバイスの制御機能は、本体側からの入出力指令にもとづいて働く。入出力デバイスをKL1 の世界のプロセスとして見せる変換がフロントエンド上(又は本体上のもっともフロントエンド寄り)で行なわれ、KL1 の世界からはストリーム通信で入出力が行なわれる。本体とフロントエンドとはKL1 用の通信線で接続される他に、メンテナンス用の通信線で接続されるのが普通である。メンテナンス、IPL 、計測データ収集などはこのパスを用いて行なう。

コンパイル、リンク、シミュレータによるテストまではクロス・システム上で行なうことができる。フロントエンドとは別のマシン上でクロス・システムを使用したときは、LAN を用いてコンパイル&リンクした結果をフロントエンドまで転送しライブラリに登録しておく。それをレジデントPIMOS 上のローダにより本体上にロードすることにより実行可能となる。ユーザが本体上のユーザプログラムとやりとりするためのコンソールは、フロントエンドのビットマップディスプレイが普通であるが、LAN を介したリモート端末機能のサポートなども検討する。

４．記述言語と基本概念

ＰＩＭＯＳの記述には，ＰＩＭＯＳの下で走るユーザ・プログラムと同じく，並列型論理言語ＫＬ１を用いる。ＫＬ１は，その基本的な機能としてはＦＧＨＣであるが，ＰＩＭＯＳの記述にあたっては，種々の機能および概念の拡張が不可欠である。以下この章では，どのような拡張がなぜ必要なのか，それが言語仕様上はどう実現されるかについて述べる。

４．１　機能拡張の要点

ＰｉＭＯＳを記述する際に，ＦＧＨＣに対して必要な機能拡張は，以下のように分類できる。

（１）メタプログラミング機能

ＯＳは，たとえば，ユーザ・プログラムがどのぐらいの時間走っていたか，などということを管理する必要がある。こうしたことは，ユーザ・プログラム自身が対象とする問題自体ではなく，問題を解く解き方に関する問題，つまり，一段メタな問題である。ＯＳをＫＬ１で記述し，しかもそれが，ＫＬ１で記述したユーザ・プログラムをメタな立場から管理できるようにするには，ふたつの方法が考えられる。

ひとつは，メタ機能の対象となるユーザ・プログラムをすべてインタプリティヴに走らせることとし，ＫＬ１を純粋にメタプログラム言語としてしか用いない方法である。しかし，この方法には，すべてがインタプリティヴなので，極度に実行効率が低下するという重大な欠点があり，そのままでは実用的なＯＳの記述手段とはなりえない。

いまひとつは，ＫＬ１に自分自身のメタレベルをも記述できるような機能を導入し，これを用いてＯＳを記述する方法である。この方法をとれば，実行効率の低下はないが，言語の本質的な拡張が必要である。

インタプリタ方式の改良として，インタプリタとユーザ・プログラムを合わせて部分計算する手法が提案されているが，この場合も部分評価したプログラムを作成し，それを動作させるメタ機能は，最低限必要である。また，その他のメタ機能についても，なんらかの言語レベルでのサポートなしで効率よく実現するのは難しい。

こうした，メタ機能としての拡張は，以下のようなものが必要である。

① プログラム動的作成機能

ＯＳの下でプログラムの開発を行なうことができるためには，プログラムを動的に生成し，これを実行できる機能が必要である。このためには，実行されるコードを動的に生成し，これに制御を移す機能が必要である。これは，コードをデータ・オブジェクトと

してみる機能であるから，高階の機能にあたる。

② 資源管理機能

プログラムの実行には，計算時間，メモリなどの資源が必要である。ＯＳは，ユーザ・プログラムの実行過程で必要となるこうした資源を管理し，適正な配分，暴走などによる過度な消費の防止を行なう必要がある。特に注意が必要なのは，ＯＳとユーザ・プログラムが同じＫＬ１で走るのだから，ユーザ・プログラムが消費する資源は，ＯＳが動作するのに必要なものと同じものである，という点にある。

マルチプロセサ上での効率的な実行のためには，計算負荷の各プロセサへの適正な配分が必要になる。これをすべて言語処理系が自動的に行なうことは非常に困難であるので，プログラム中，少なくともＰＩＭＯＳのプログラム中では，なんらかの形で負荷の配分を指定する機構が不可欠である。また，そのためには，各プロセサの負荷を計測する方法を用意する必要がある。

③ 異常処理機能

ユーザのプログラム中には当然誤りがあると思わなくてはならない。この誤りが原因で生じる異常な事態によって，ＯＳが正常な動作を続けられないようなことがあると，安心して使えるＯＳにはならない。これを防ぐには，ＯＳ内の異常と，ユーザ・プログラム内の異常を，切り分けるような機能が必要である。また，どのような異常が発生したのかをユーザに知らせることは，ユーザのプログラム開発を助けるために重要である。

（２）速度向上のための機能

ＧＨＣを単純にインプリメントしたのでは，逐次型言語と同じオーダのアルゴリズムを記述できない問題がある。たとえば，動的に内容が変化するような大きな表を，定数オーダで検索／更新するようなアルゴリズムが記述できず，検索／更新ともに，表に登録されたデータの数の対数に比例するような手間がかかる表現しかできない。同様の問題は，複数のストリームをマージする操作にもあり，Ｎ本のストリームをマージする場合に，一個の要素につきｌｏｇ Ｎに比例するだけの手間がかかることになる。これらは通常の逐次計算機上では定数オーダの手間で実現できる仕事であるから，問題のサイズが大きくなったときに，並列計算機よりも，その要素プロセサひとつよりも低性能の逐次計算機の方が高効率に実行できる，という逆転現象が生じる。

これを解決するには，配列やストリームの大域的な最適化を行なうコンパイラを用意することが考えられる。しかし，そういったコンパイラの作成は容易ではないし，最適化が実際に行われているかどうかが明白でないので，プログラムの表わすアルゴリズムの手間のオーダを把握するのが困難になる，という欠点もある。このため，むしろこうした機能を言語のプリミティブとして導入することが望ましいと考えられる。

速度向上のために導入すべき機能としては，具体的には以下のふたつが候補としてあげられる。

① 配列

要素の個数に関わらず，一定の手間で参照／更新できるような配列の機能。これを用いてハッシュ表を用意すれば，検索／更新ともに，要素の数によらず一定の手間でできるような表を実現することができる。

② 一定の手間でマージできるストリーム

マージする本数に関わらず，一定の手間でマージできるようなストリームの機能。ストリームによる通信は，ＰＩＭＯＳとアプリケーション・プログラム，ＰＩＭＯＳのモジュール間の通信，また，荘園や割込み機能など，言語の基本的な機能の実現に重要な役割を果たす。このため，ストリーム・マージの効率化は，ＰＩＭＯＳの実行効率全体を左右しうる，重要な課題である。

（３）否定を表現する機能

ＦＧＨＣは，原則としてはＰｒｏｌｏｇと同様，単調な論理しか扱えず，否定を表現することはできない。これを根本的に解決することは容易ではなく，ＧＨＣ自体の設計方針にかかわる問題である。しかし，記述の簡素化と効率のよい実現を容易にするために，ｏｔｈｅｒｗｉｓｅないしは逐次ＯＲの機能を導入することは有益である。

（４）　大規模プログラムの構成のための機能

ＦＧＨＣは平坦な構造しかないため，そのままではＯＳのような大規模なプログラムを構成するには不便な点が多い。なんらかのモジュール構造の導入が必要であるが，ここでは特に触れないこととする。

４．２　拡張言語仕様

必要なＫＬ１の拡張機能は，ＫＬ１言語の仕様上では以下のように表現される。

（１）ゴール間の優先度指定

資源配分の指定の仕方のひとつとして，ＡＮＤ関係にあるゴールに実行優先度を設け，優先度の高い計算ほど先に行なうようにする機能を導入する。

優先度は，たとえば０から適当な上限値までの正整数で指定する。優先度の高いゴールほど先に処理するのを原則とするが，マルチプロセサシステムでは，優先度を厳格に守る効率のよい実現方法がないので，処理系は必ずしも厳格に優先度指定を守らなくともよいものとする。したがって，優先度はあくまでも計算時間という資源を割り当てる際の目安に過ぎず，本質的な実行順序の制御に用いてはならない。

優先度の指定は，ゴール単位に行なう方法と，後述する荘園の単位に行なう方法が考えられる。荘園の設定にはある程度のオーバヘッドが伴うことは避けられないので，頻繁に優先度指定を行なう必要があるのなら，ゴール単位の指定が有利である。ゴール単位指定を行なうと，優先度を用いない場合でも，サスペンド時にややオーバヘッドがあるが，大きな違いはない。

（2）クローズ間の優先度指定

資源配分の指定の仕方のひとつとして，ＯＲ関係にあるクローズに実行優先度を設け，候補となるかどうかの検査を優先度の高いクローズほど先に行なう機能を導入する。

ＫＬ１では原則として，クローズの試行順は処理系の都合で任意に決め手よい。たとえば，第一クローズを用いたリダクションの可否の判定に離れたプロセサ上の情報が必要で，第二クローズには局所的な情報で十分な場合などは，第一クローズでリデュースできる可能性があっても，第二クローズを用いてリデュースしてよい。

クローズ間の優先度はこの自由度に対する制限であり，述語を構成するクローズ群のうちの指定したサブセットが，「その時点」でフェイルまたはサスペンドする場合以外は，他のクローズを用いてリデュースしてはならない，という指定により行なう。

クローズ間の優先度を指定しても，他の部分の計算の実行順によって，選ばれるクローズが非決定的であることに変わりはない。したがって，ＡＮＤゴール間の優先度指定が目安にすぎない以上，クローズ間の優先度も目安としての意味しか持ちえない。

指定の仕方としては，クローズに優先度を指定するかわりに，特定の変数がまだ具体化されていないことを確かめる述語ｖａｒ（Ｘ）を導入する方式も考えられる。

（3）負荷分散制御

負荷の分散を静的にプログラム中で完全に指定することは，非常に困難であろう。一方，計算の局所性は，プログラム中に指定することは比較的容易であるが，動的に検出することは困難である。したがって，静的には局所性を意識して大まかなところを指定し，静的な指定と物理的なプロセサとの対応は，実際のプロセサ負荷を観測しつつ，動的に修正していくのが適当である。

① 動的な負荷分散指定

動的な負荷分散のためのプリミティヴの導入が必要であるが，詳細は今後の研究が必要な部分が多く，ここでは触れないこととする。

② 静的な負荷分散指定

静的な負荷分散指定は，動的な再分散が行ないやすいような方式を取ることが必要であ

る。提案されている指定方式としては，以下のようなものがある。

○ 実行するプロセサを静的に確定してしまい，動的な再分散の対象としない。

○ Ｎ次元（Ｎ＝１から３）の立方体中にプロセシング・パワーが平均に分散しているモデルを考え，静的にはこの中のどの領域で計算すべきかを指定する。立方体中の領域と物理的なプロセサとの対応関係は，立方体内の領域間のトポロジーを変えない範囲で，動的に変化させる。

ＰＩＭＯＳの記述には両者ともに必要な局面があるので，両方を実現するのが適当であろう。

（４）資源管理と異常処理

資源管理と異常処理のためのプリミティブとして，以下のようなメタ組込述語を考える。

ｅｘｅｃｕｔｅ（ＧＯＡＬ，ＣＯＮＴＲＯＬ，ＲＥＳＵＬＴ）

ここで，各引数は以下のようなものである。

① ＧＯＡＬ

実行されるべきゴールないしゴール列。通常のボディ中のゴールと同様，コンパイル時に呼出し先が明白に与えられているものとする（ゴールが変数になっているようなものではないものとする）。ｅｘｅｃｕｔｅは従来メタコールと呼ばれていたものの機能のうち，資源管理と異常処理の機能だけを抜き出したものになっているわけである。

このゴールの実行にあたって必要とされる資源について，以下に述べるような資源管理機能が，実行中の異常に対して以下に述べるような異常処理機能が働く。以降では，この実行に関わる資源管理／異常処理単位を，「荘園」と呼ぶ。ｅｘｅｃｕｔｅはｅｘｅｃｕｔｅの中で実行してもよい。したがって，荘園は任意にネストする。

② ＣＯＮＴＲＯＬ

資源割当制御ストリーム。外部から各種のコマンドをここに与えることにより，実行中に消費してよい資源に関するなんらかの制限が付けられたり，その制限が緩和されたりする。具体的には以下のようなものが考えられる。

ｓｔａｒｔ
ｓｔｏｐ
ａｂｏｒｔ

荘園の実行の開始／中断／放棄を指定する。いったん中断を指定しても，再び開始を指定することによって，実行を再開できる。放棄を指定した場合は，この限りではない。マルチプロセサシステムでは，開始／中断／放棄の指定も，

実際に効果を発揮するまでにある程度の遅れを伴うことがあることに注意。

execution_time (T)

荘園の実行中に消費してよい実行時間を指定する。この実行時間を費やしても実行が完了しない場合には，実行は中断され，その旨ＲＥＳＵＬＴに報告される。マルチプロセサシステムでの効率よい実現のためには，必ずしも正確にこの実行時間だけ実行したときにただちに実行を中断しなくてもよい。誤差範囲が明らかにされていれば，ある程度の誤差（余分な実行時間）は許される。このコマンドをひとつの荘園に対して複数回発行した場合は，それらのＴの合計に当たる実行時間が許されるものとする。

reduction (R)

荘園の実行中に行なってよいリダクション数を指定する。この回数のリダクションを行なっても実行が完了しない場合には，実行は中断され，その旨ＲＥＳＵＬＴに報告される。マルチプロセサシステムでは，必ずしもこの指定が正確に守られるとは限らないことは，実行時間の場合と同様である。このコマンドをひとつの荘園に対して複数回発行した場合は，それらのＲの合計に当たる回数のリダクションが許されるものとする。この機能は，実行時間計測の効率的な実現が困難な場合，その代替機能として用意するのが適当である。

allocation (M)

荘園の実行中に割り当ててよいメモリの語数を指定する。この語数以上のメモリを割り当てないと実行が完了しない場合には，実行が中断され，その旨ＲＥＳＵＬＴに報告される。マルチプロセサシステムでは，必ずしもこの指定が正確に守られるとは限らないことは，実行時間の場合と同様である。このコマンドをひとつの荘園に対して複数回発行した場合は，それらのＭの合計に当たる語数のリダクションが許されるものとする。

execution_time_so_far (T)
reduction_so_far (R)
allocation_so_far (M)

荘園の実行中に行なった資源消費量の問い合わせ。その時点までの実行時間／リダクション数／割り当て語数を，それぞれＴ，Ｒ，Ｍとユニファイする。荘園の実行中は，これらの値は必ずしも正確ではなくてよい。

資源の割り当てはいずれも，ネスティングの外側の荘園に対する指定が，内側に対しても有効である。すなわち，外側の荘園の実行の中断を指定すると，その内側の荘園の実行ももすべて中断される。実際に有効な優先度は，指定された値と外側の荘園の優先度のうち低いほうである。また，ある荘園の実行時間，リダクション数，割り当て語数の制限は，その内部のネストした荘園の実行時間，リダクション数，割り当て語数を合計したものに対する制限である。

なお，荘園プリミティヴの初期状態としては，許される実行時間，リダクション数，メモリ割り当て語数はみな０，ｓｔａｒｔコマンドを発行するまでは停止状態にあるものとする。

③ ＲＥＳＵＬＴ

荘園の実行状態に関する報告が行われるストリーム。以下のようなメッセージが送られてくる，

ｓｕｃｃｅｓｓ
ａｂｏｒｔｅｄ

荘園内の実行がそれぞれ，成功／放棄の形で完了したことを意味する。このメッセージが得られた後には，資源消費量の問い合わせは，正確な値を得ることができる。

ｅｘｅｃｕｔｉｏｎ＿ｔｉｍｅ＿ｌｉｍｉｔ
ｒｅｄｕｃｔｉｏｎ＿ｌｉｍｉｔ
ａｌｌｏｃａｔｉｏｎ＿ｌｉｍｉｔ

荘園内の実行がそれぞれ，実行時間，リダクション回数，メモリ割り当て語数の制限によって中断されたことを意味する。不足した資源の追加を行なえば，実行は再開される。詳細は異常処理の項に譲る。

ｅｘｃｅｐｔｉｏｎ（ＩＮＦＯＭＡＴＩＯＮ，ＧＯＡＬ，ＮＥＷＧＯＡＬ）

荘園の実行中に例外事象が生じたことを意味する。例外事象には，ゴールの候補節がすべて失敗した場合，ボディ部でのユニフィケーションの失敗，算術演算時のオーバフロー，（もし検出可能なら）デッドロックなどを含む。ＩＮＦＯＲＭＡＴＩＯＮにはどのような例外事象が生じたかについての情報，ＧＯＡＬには例外事象の原因となったゴールが渡される。ＮＥＷＧＯＡＬを適当なゴールに具体化すると，例外を起こしたゴールをこれに置き換えて，実行を継続する。詳細は異常処理の項に譲る。

デッドロックはゴミ集めの際に検出できる場合がある。参照カウントなどによるインクリメンタルなゴミ集めを行なうなら，リダクションごとに少しづつゴミ集めをするわけだから，この際に検出されることがある。

組込述語によって検出される例外と，ソフトウェアによって検出される異常事態を，同じ枠組みで処理できるようにするために，また，例外処理のデバッグを容易にするために，ソフトウェアで積極的に例外を起こす機能（レイズ機能）を用意する。

ｒａｉｓｅ（ＩＮＦＯＲＭＡＴＩＯＮ）

（５）実行可能コードの扱い

実行の対象となるコードを作成し，これに制御を渡す機能として，以下のような組込み機能を用意する。なお，コードの更新を許すためには，コードをプロセスとして扱う必要があるが，ここでは，コードは静的なオブジェクトであり，更新は行われないものとする。

① コード型の導入

実行可能コードをデータとして扱うために，コード型のデータオブジェクトを用意する。

② コードの生成

create_code (CODE, [INSTR, ...])

第二引数の命令列からなる実行可能コードを作成し，CODEとユニファイする。第二引数の命令リストが完結し，コードが完成するまでは，CODEは具体化されない。

③ コードの呼出し

apply (CODE, [ARG, ...])

CODEを第二引数で指定した引数に適用する。CODEが具体化され，第二引数のリストが完結するまでは，実行は遅延させられる。

（6）配列

以下のような仕様をもつ配列を，組込み機能として用意する。

① 配列型の導入

配列を効率よく扱うために，配列型のデータオブジェクトを用意する。

② 配列の生成

create_array (ARRAY, SIZE)

要素数SIZEの配列を作成し，ARRAYとユニファイする。

③ 配列要素の参照

aref (ARRAY, INDEX, ELEMENT)

ARRAYのINDEX番目の要素を，ELEMENTとユニファイする。ARRAY及びINDEXが具体化するまでは，実行は遅延させられる。

④ 配列要素の更新

aset（ARRAY，INDEX，ELEMENT，NEW_ARRAY）

ARRAYのINDEX番目の要素を，ELEMENTで置き換えたような配列を，NEW_ARRAYとユニファイする。ARRAY及びINDEXが具体化するまでは，実行は遅延させられる。

配列の更新操作には，原則としては配列全体のコピーが必要になる。これを常に一定の手間で実現するためには，配列要素の更新時に，更新前の配列を直接書き替えてしまうしかない。MRBなどにより参照数の管理を行なえば，参照数が1の配列については，このような更新が可能である。実現の都合上，このような参照数が1の配列以外の更新を許さないものとしても，あまり問題はないであろう。また，必要なら，Prologにおけるマルチ・バージョン・ストラクチャと同じく，ひとつの物理的な配列と連想リストの組み合せによる実現手法を取れば，かなり効率的な実現も可能である。

（7）ストリーム

以下のような仕様をもつストリーム通信機能を，組込み機能として用意する。

① ストリーム型変数の導入

ストリーム通信を効率よく扱うために，ストリーム型の変数を用意する。ストリーム型変数は，リストまたはアトム[]および通常の変数としかユニファイできないものとする。また，ストリーム変数に対する多重書込みも禁止する。これを直接禁止するのは容易でないので，多重参照を持ったストリーム型変数への書込みは，多重書込みの可能性があるので，すべて禁止することにする。さらに，多重参照の有無の判定は，安全方向（より多くを多重参照と判定する方向）ならば，必ずしも正確でなくてもよい。

② ストリームの生成

create_stream（IN，OUT）

ストリーム型変数INを生成し，それと対応する通常の変数OUTを入出力とする仮想的なプロセスを生成する。このプロセスは，以下の項に述べるような機能を実現するものである。

③ ストリーム型変数とのユニフィケーション

ストリーム型変数をリストセル［X｜Y］とユニフィケーションは，新たなストリーム型変数Iを作り，リストセルのcdr部Yをこの変数Iとユニファイする。このとき（任意の遅れをもって）もとのストリーム型変数に対応する通常の変数，すなわち入力ストリームは，ユニファイされたリストセルのcar部Xと，新たな通常の変数Oとからなる新たなリストセル［X｜O］とユニファイする。仮想プロセスの出力ストリームは通常の変数であるから，このときすでに具体化されている可能性もあり，したがって

このユニフィケーションは失敗する可能性もある。新たに作られた変数ＩとＯとが，仮想プロセスの新たな入出力ストリームとなる。

ストリーム型変数とアトム［］とのユニフィケーションでは，対応する仮想プロセスの出力ストリームをアトム［］とユニファイする。

この動作は，出力ストリームへの多重書込みの禁止を除けば，入出力ストリームの対応づけを行なう仮想プロセスが

```
p（［X｜I］，OUT）：－　OUT＝［X｜O］，p（I，O）．
p（［］，　　OUT）：－　OUT＝［］．
```

のように動いている，と説明することができる。

④ ストリームのマージ

```
merge（IN1，IN2，OUT）
```

新たにふたつのストリーム型変数ＩＮ１，ＩＮ２を作成し，それらをストリーム型変数ＯＵＴにマージする仮想プロセスを生成する。仮想プロセスは，多重書込みの禁止を除けば，以下のような述語で表現される，通常のマージプロセスで説明できる。

```
m（［］，X，X）．
m（X，［］，X）．
m（［W｜X］，Y，WZ）：－WZ＝［W｜Z］，m（X，Y，Z）．
m（X，［W｜Y］，WZ）：－WZ＝［W｜Z］，m（X，Y，Z）．
```

以上のような制限を設けると，マージするストリームの本数によらず，一定の手間でストリームをマージすることができる。

（８）逐次ＯＲ

あるゴールが複数のクローズでリデュース可能な場合，どれをリダクションに用いるかは通常は処理系の自由である。逐次ＯＲの機能は，クローズ群のあるサブセットすべてがフェイルする場合のみ，他のクローズをリデュースに用いてよいことを指定する機能である。たとえば，以下のような記法を用いる。

```
    max（X，Y，M）：－X＞＝Y｜M＝X．
otherwise
    max（X，Y，M）：－M＝Y．
```

逐次ＯＲとは，クローズ間のプライオリティ指定とは異なる。プライオリティ指定では，あるサブセットのクローズ中にサスペンドしたものがある場合も，残りのクローズを用いてよいが，逐次ＯＲでは，すべてフェイルするのを確認した後でしか残りのクローズは用いない。

４．３　プロセスの概念

ＫＬ１がその基礎を置くＧＨＣは，非常に低いレベルの操作まで並列に実行できるセマンティクスを持つ。このため，実現方式の上でのプロセスは，ひとつひとつのゴールリダクションや，ユニフィケーションのレベルまで分解できる。

しかし，こうした細かいレベルの並列性は，プログラムを記述する上では余り意識しないし，また，意識しないで済むことがＧＨＣの特徴でもある。そこで，以下この文書中では，「プロセス」という言葉を，ＫＬ１のひとつのプログラミング技法である，再帰呼出しで実現され，「ある程度長い」ライフタイムを持つ計算過程を指す用語として用いる。したがって，何をプロセスと呼び，何がサブルーチンかは，はなはだ主観的な基準によるものである。

5　機能イメージ

5．1　概　要

5章では、ＰＩＭＯＳを構成する各機能について述べ、本節では、この各機能の概要について述べる。ＰＩＭＯＳの各機能は、ユーザから見た機能別に以下のように分けている。

（1）資源管理
ユーザの使用するＣＰＵ消費量、メモリ量、Ｉ／Ｏ資源などの管理、負荷分散の制御を行う。これらは荘園の機能を使って実現されるが、Ｉ／Ｏ資源についてはフロントエンドに資源の確保・解放を依頼する形で行われる。

（2）異常処理
デッドロック、資源の超過使用、ゼロ除算等の異常に対して、実行を継続できるような処置を行う。また、ユーザがプログラムによりこの処置を定義できるような機能も提供する。

（3）ユーティリティ
ＰＩＭＯＳやアプリケーション・プログラムを記述するのに便利なツール群を提供する。例えば、任意のデータをメモリ上に格納する容器（プール）や、内部データ構造と文字列表記との変換を行うトランスデューサ等がある。ただし、変数の表示に関しては、並列環境であるため、いつ値を持つか分からず、正しく行えるとは限らない。

（4）プログラム管理
コンパイラ，リンカ，ローダ，ライブラリ，シンンボル管理システム等の機能を提供する。ただし、コンパイラ，リンカは当面クロス・システムに置き、ライブラリ，シンボル管理システムはフロントエンド・マシンで実現される。

（5）入出力
入出力は、フロントエンド・マシンとのストリームによる通信によって行い、端末，ファイル，プリンタ等の機能を提供する。また、時計の機能も提供する。

（6）タスク管理
名前を持ったあるひとかたまりの仕事の単位をタスクと呼び、ユーザはそれに対して荘園で提供されているような操作を行うことができる。またタスクは、Ｉ／Ｏ資源の管理単位でもある。そしてユーザが簡便にタスクに対して操作を行えるようにするツールであるシェルの機能も提供する。

（7）デバッグ支援
ユーザ・プログラムのデバッグを支援する機能には、文法ミスのチェック等を行う静的にプログラムを解析するプログラム・アナライザとプログラムを実行させて虫取りを行うトレーサがある。後者については、並列言語のデバッグ方式という大きな課題があり、その方法については今後十分な検討が必要である。

（８）低レベル・サポート

システムの立ち上げ，低レベル・デバッグ，メインテナンス，ハード・エラー処理等の機能を提供する。

（９）計測評価機能

ハードウェア，処理系，ユーザ・プログラム等の計測評価をスムーズに行え，各種評価データを収集／編集する機能を提供する。

各機能を実現するに当たっては、４章で述べたようなＫＬ１の言語機能を必要とする。また、以下のようにそれぞれの機能を利用し合う必要もある。

・資源管理と異常処理

資源の消費超過時に、異常処理の機能を利用してユーザにその旨を知らせる。

・トランスデューサとシンボル管理システム

アトムからアトム名への変換及びアトム名からアトムへの変換では、シンボル管理システムによりその対応を調べる。

・シンボル管理・ライブラリとプール

シンボルやユーザ・プログラムのモジュールを効率的に管理するために、プールの機能を利用する。

・入出力機能と資源管理

入出力装置（ストリーム）を資源として扱うために、資源管理の機能が必要である。

・タスク管理と資源管理

ユーザ・タスクの資源を操作するには、資源管理の機能が必要である。

・デバッグ支援とプログラム管理・トランスデューサ

ユーザ・プログラムをデバッグするには、コンパイラ等のプログラム管理の機能によりプログラムを実行可能な形式に変換する必要がある。また、デバッグしたいプログラム（ゴール）の入力，トレース出力等は、トランスデューサの機能を利用する。

これらは一例にすぎないが、相互に深い関連を持っており、今後の詳細機能及び実現方式等の検討において、それぞれの機能項目間ＫＬ１言語機能の関連をより明確にする必要がある。

５．２　資源管理

ＰＩＭＯＳは，ユーザが利用するＣＰＵ（計算時間），メモリ，Ｉ／Ｏ等の資源を管理し，適当な配分，暴走などによる過度な消費の防止を行う。ただし，Ｉ／Ｏ管理だけは他の資源管理とは性質が異なっている。すなわち，ＣＰＵやメモリは有限にしか存在しないものとして扱っているが，Ｉ／Ｏ資源だけは無限に存在するものとして扱っている点である。一般のＯＳのＩ／Ｏ管理の主な仕事は，この有限にしか存在しない資源をあたかも無限に存在するかのごとく見せることである。しかし，ＰＩＭＯＳではその仕事をフロント・プロセッサ上のＳＩＭＰＯＳに委せており，ＰＩＭＯＳではタスクに付随したＩ／Ｏの管理のみを行っている。

（１）プロセサ管理

プロセサ管理では，ユーザ・プログラムの暴走によるＣＰＵ時間の過度な消費を防ぐためのＣＰＵ消費量の管理と実行優先度（priority）の管理を行う。

・ＣＰＵ消費量の管理

ユーザが消費できる計算（ＣＰＵ）時間は，荘園単位でその上限を持っておくことにより制限されている。ユーザの消費量がその上限を超えた場合はＫＬ１処理系がＰＩＭＯＳに例外事象として割り出す。従って，処理系はプログラムの実行中にＣＰＵの消費量をカウントし，上限を超えているかどうかをチェックしなければならない。ただし，実際の時間でカウントするのは困難なので，リダクション数で代用することが考えられる。

・実行優先度の管理

ＫＬ１処理系は，実行優先度を基にＣＰＵを割り付ける（スケジューリングする）。この実行優先度の設定は，荘園単位又は goal 単位のどちらかで行えるようにする。また，設定できる実行優先度の上限と下限を荘園単位で持ち，設定・変更時にそれを超えないようにＫＬ１処理系がチェックする。実行優先度の動的な変更は，行えるようにしたい。しかし，それを可能にするためには，goal を別の実行優先度のスケジューリング・キューに移動する時に，総てのキューを調べて回らなければならず，実現が困難である。ただし，実行優先度を下げるだけなら，goal の dequeue 時に別（低い優先度の）キューに移動するだけなので，実現は楽である。

（２）メモリ管理

メモリ管理では，ユーザが消費するメモリ消費量を管理する。ただし，実際にメモリを割り付けるのは，ＰＩＭＯＳより下のレベルであるＫＬ１処理系が行い，ＰＩＭＯＳは消費量の上限の設定／変更を行うだけである。ユーザが消費できるメモリの量の上限は荘園単位で管理されており，ユーザの消費がそれを超えた場合は，ＫＬ１処理系がＰＩＭＯＳに割り出す。従って，処理系はプログラムの実行中にメモリの消費量をカウントしなければならない。この消費メモリのカウントは，これまで消費した総量で行い，現在消費している量ではない。従って，ＧＣ（ゴミ集め）が起きてもユーザが消費でき

るメモリ量が増えることはない。また，このＧＣは，ＰＩＭＯＳより下のレベルである
ＫＬ１処理系が行うが，ＧＣが起きたことの通知をＰＩＭＯＳが受け取り，ユーザに知
らせたほうが良いと思われる。

（３）許容消費量を超えた時の処理

前述したようにユーザが資源（ＣＰＵやメモリ）の制限を超えた場合，ＫＬ１処理系
から例外事象として割り出される。消費量が超えた場合，ＰＩＭＯＳはユーザにその旨
を知らせ，対処を訪ねる。この対処として考えられるのは，（ＣＰＵ又はメモリの）消
費量の上限を上げる（制限を緩和する），あるいは資源を消費しすぎたタスクを放棄す
る（殺す）等がある。このような処理は，原則的には（ユーザが特に指定しなければ）
ＰＩＭＯＳが行うが，ユーザがその処理を行う（プログラムにより定義する）ことも
できるようにする。ただし，その実現方式についての詳細は異常処理の項に譲る。

（４）子荘園における資源管理

子荘園（ネストしたexecute）で消費できる資源は，あくまで親荘園から分け与えられ
るものである。つまり，子荘園の資源の制限は，親荘園のそれを上回ることはけっして
ない。従って，資源の制限の指定方法としては，親に対する割合で指定する機能も提供
したほうがよいと思われる。例えば，子荘園での消費資源の制限がないという意味は，
資源を無制限に消費してもよいということではなく，親荘園から無限に分けてもらえる
ということであると解釈する。従って，子荘園が親荘園の制限を食い潰したら，例外事
象として割り出される。ただし，このときは親荘園が制限を超えたことになる。

（５）負荷分散制御

ユーザ・プログラムで一部のＰＥに割り付けた仕事が先に終了してしまっても何のメ
リットもない。そこで，全ＰＥになるべく均一に負荷が分散するように制限する必要が
ある。それには，ユーザ・プログラム中に挿入するプラグマによる負荷分散の他に，近
傍のＰＥの忙しさを定期的に監視し，各ＰＥの負荷が均一になるように制御する負荷分
散がある。前者は，プログラム中の指定に従ってＫＬ１処理系が行うが，後者はＰＩＭ
ＯＳが行い，これは，論理プロセサ平面と物理プロセサ平面との対応を変えることに相
当する。つまり，忙しい論理ＰＥの面積を小さくすることで，忙しいＰＥから暇なＰＥ
へ仕事を割り振る（goal を送出する）ことになる。この goal 送出は，goal のスケジ
ューリング・キューへの enqueue 時に行われるのがよいと思われる。また，論理平面
と物理プロセサとの対応を変更する時に，送出する goal の量に注意しなければならな
い。つまり，暇なＰＥから忙しいＰＥへ goal を送出した為に負荷が逆転し，すぐに逆
方向の goal の送出が行われるという現象が起こりうるからである。このように goal
の送出が波をうつ（何度も行ったり来たりする）ことのないようにしなければならない
が，その方法は検討を要する。

・各ＰＥの負荷

前述したような負荷分散を行うためには，各ＰＥの忙しさ（負荷）とは何かを定義し
なければならない。ＫＬ１の各 goal は，実行優先度を基にスケジューリングされるた
め，単にスケジューリング・キューの長さでは不十分である。そこで実行優先度を考慮

に入れた負荷を考えなければならない。この負荷として考えられるものに，ある一定時間の間に実行された goal の実行優先度の平均値がある。例えば，１０秒間に優先度１００のゴールを５秒間，２００のゴールを２秒間実行し，あとの３秒はアイドルであったとすると，このプロセサの１０秒間の平均負荷は：

（１００＊５＋２００＊２）／１０　＝９０

と算定される。従って，負荷を均一にするということは，その平均値ができるだけ等しくなるようにすることに相当するが，実現方式については検討が必要である。また，アイドルの扱いについても，検討が必要である。

（６）タスクに付随した資源管理

ユーザが，Ｉ／Ｏ資源を使用するためには，そのＩ／Ｏ資源を確保し，使い終わったら解放しなければならない。ＰＩＭＯＳでのＩ／Ｏ資源の操作は，Ｉ／Ｏストリームへコマンドを送ることにより行われる。従って，Ｉ／Ｏ資源の確保とは，Ｉ／Ｏストリームを取ることに相当すると考えられる。また，Ｉ／Ｏ資源の解放とは，そのＩ／Ｏストリームに〔〕を送る（Ｉ／Ｏストリームを閉じる）ことに相当する。ただし，Ｉ／Ｏ資源の解放には，そのようなユーザによる積極的なものの他に，タスクが放棄された時のそのタスクが使用していたＩ／Ｏ資源の解放が必要であり，これはタスクを管理しているＰＩＭＯＳが行わなければならない。そこで，ＰＩＭＯＳは，ユーザが確保したＩ／Ｏ資源をタスク単位で管理する。従って，ユーザは，Ｉ／Ｏ資源の確保をＰＩＭＯＳに依頼する形で行わなければならない。その依頼の方法には，ＰＩＭＯＳへの割出しが考えられる。

５．３　異常処理

５．３．１　エクセプション

エクセプションとはプログラムの実行においては重要であるが，プログラムの実行のロジックには関係づけられない状態である。したがってエクセプションはメタ論理的意味のあるものである。

(1) 種類

・デッドロック

並列言語特有のエクセプションで，全てのゴールがサスペンドしている状態である。どのゴールも具体化しない変数が具体化されるのを待っているゴールがある場合が考えられる。

・資源の超過使用

実行において使用を許された資源(実行時間,リダクション数,メモリの語数)を実行中に使い果たしてしまった状態である。

・ゼロ除算/オーバフロー/アンダフロー/未定義述語呼出し

閉世界のフェイルに対応しているが，通常のフェイルと区別することは有用である。

・フェイル

オブジェクトレベルのフェイルはメタレベルにおいてはエクセプションとして認識される。つまりオブジェクトがフェイルしたからといってメタレベルがフェイルするわけではない。そういう点でメタ論理的である。

フェイルの例としては，ゴールに対して候補節が全て失敗節になってしまった場合やアクティブ部においてユニファイに失敗した場合などがある。

・レイズ

ソフトウェアで積極的にエクセプションを起こす機能をレイズという。実行の文脈によって処理方式を変えたいとき，局所的情報のみでは処理を行うのが困難なとき，そのようなときにレイズを使用する。

(2) 認識方法

エクセプションを正しく処理するために，オブジェクトプログラムの状態としてエクセプションを導入する必要がある。これはｅｘｅｃｕｔｅのＲＥＳＵＬＴストリームを通じてメッセージとして知ることができる。このメッセージには，

```
ｅｘｅｃｕｔｉｏｎ＿ｔｉｍｅ＿ｌｉｍｉｔ/ｒｅｄｕｃｔｉｏｎ＿ｌｉｍｉｔ/
ａｌｌｏｃａｔｉｏｎ＿ｌｉｍｉｔ          （資源の超過使用）
ｅｘｃｅｐｔｉｏｎ（ＩＮＦＯＲＭＡＴＩＯＮ，ＧＯＡＬ，ＮＥＷＧＯＡＬ）
                                        （その他）
```

がある。ＩＮＦＯＲＭＡＴＩＯＮはエクセプションのタイプ及びそれに付随した情報，ＧＯＡＬはエクセプションを引き起こしたゴール，ＮＥＷＧＯＡＬは具体化されていない変数である。

例えばゼロ除算の場合，ＩＮＦＯＲＭＡＴＩＯＮはｚｅｒｏ－ｄｉｖｉｓｉｏｎ，ＧＯＡＬはＸ：＝５／０である。特にデッドロックの場合，ＧＯＡＬはデッドロックしているゴールの列である。ゴールＧＯＡＬは新しいゴールＮＥＷＧＯＡＬに置き換えられ継続実行するが，ＮＥＷＧＯＡＬがメタプログラムによって具体化されるまでサスペンドする。ガード部でエクセプションが起こった場合はその節を呼び起こしたゴールをサ

スペンドする方法と，ガード部のゴールをサスペンドする方法が考えられる。

(3) 処理方法

(2)の機構とＣＯＮＴＯＲＯＬストリームにコマンドメッセージを与えることにより，荘園内のエクセプションに対処することができる。例えばメタプログラムはＣＯＮＴＲＯＬストリームにａｂｏｒｔを流すことによってオブジェクトプログラムを放棄することが出来るし，ＮＥＷＧＯＡＬをｔｒｕｅに具体化することによってオブジェクトのゴールを成功させることもできる。一般にＮＥＷＧＯＡＬに具体化させるものによって，様々な動きを指定するができる。ＰＩＭＯＳはこの機構を利用してユーザにエクセプションの処理法について尋ねることができる。

資源の超過使用の場合は注意が必要である。この場合ＣＯＮＴＲＯＬストリームへの資源追加割当てメッセージによって対処できる。しかし言語仕様上，資源追加割当てメッセージはエクセプションとは関係無く出せる。資源の超過使用メッセージが届く前に資源追加割当てメッセージを送った場合，実際に資源の追加割当てを行った後にその資源の超過使用メッセージが出されたのか，前にそのメッセージが出されたのか分からない。つまりメッセージの行違いが起こりうるので，プログラマはこのような使用を避けるべきである。

このようにしてユーザはエクセプション処理を記述できるが，記述されていないエクセプションに対する処理はデフォルトの処理が行われる。ｅｘｅｃｕｔｅが起動されるとＲＥＳＵＬＴストリームを入力とするデフォルトのエクセプション処理プログラムが起動するものと考える。このプログラムはエクセプション処理が記述されていないエクセプションについては上述のレイズを発生させる。したがってｅｘｅｃｕｔｅがネストしている場合，あるレベルでエクセプション処理が記述されていない場合エクセプションはその外側に伝えられる。デフォルトの処理の実体はその最外部にあるとみなすことができる。

５．３．２ システムエクセプション

システムエクセプションとは，荘園に閉じていないエクセプションである。ＰＥ内メモリの欠乏，全体的なメモリの欠乏が考えられる。システムエクセプションへの対処はすみやかに行う必要がある。この対処法は検討を要する。

５．４　ユーティリティ

ユーティリティでは、種々のモジュールからサブルーチン的に呼び出すことのできる、以下のような機能を用意する。

５．４．１　プール

プールは、任意のデータ等をメイン・メモリ上で格納するための容器である。ひとつひとつのプールに対しては、dynamic にデータ等の格納、検索、削除等を行いたいので、それぞれをプロセスとして実現する。

以下にプールとして実現する代表的なものをあげる。

・リスト
　いわゆるリストであり、無限長の構造を有する。
・アレイ
　1次元の配列であり、サイズは固定である。
・スタック
　後入れ先出しを行える構造データを有する。
・ハッシュ・テーブル

プールに対しては、以下のような操作を行うことができる。

・データ等の格納
・データ等の取り出し
・データ等の削除
・プール中のデータ等の格納情報に関する問い合わせ

また、プール上のデータ等へのアクセス方法としては、次のものが用意される。

・順次アクセス
・直接アクセス
・インデックス・アクセス

ハッシュ・テーブルのような直接アクセス機能を効率よく実現するには、低レベル・サポートが必要だろう。

５．４．２　トランスデューサ

トランスデューサは、内部データ構造とテキスト文字列とのデータ変換を行う。機能としては、次のものが用意される。

(1) テキスト文字列を内部データ構造に変換するパーザ
(2) 内部データ構造をテキスト文字列に変換するアンパーザ
(3) 主に、構造体でない内部データ(たとえばアトムや数)を印字表現の文字列（16bit ストリングまたは文字コードのリスト）に変換したり、文字列を内部データ構造に変換するシンボライザ

それぞれ、普通のPrologのタームを読み書きする程度の機能を持つ。パーザ、アンパーザは、サブルーチン・パッケージとして実現されるが、シンボライザでは名前表の集中管理が必要であるから、シンボル管理システムはプロセスとして実現される。また、シンボライザは、扱うデータの種類により、以下のような変換機能を持つ。

・アトム	印字名
・変数	印字名
・整数・実数	印字名
・ベクタ	関数表記、リスト表記、ベクタ表記
・ストリング	ストリング表記

パーザは、対象とする文字列は解析中も変化しないから、実現に当たって特に問題はない。

しかし、アンパーザの実現には多少の問題点が生ずる。すなわち、アンパーザは、プログラムの実行中に稼動するので、KL1で走るアンパーザとプログラムは並列に実行されてしまう。そのため、アンパーザが対象とするデータは常に動的である(ある時点で変数であると解析されたデータは、まさにその瞬間にある数値等に具体化してしまうかもしれない)。たとえば、p(X)は、最初Xは変数だったが、書いている間に3にバインドされる可能性があるから、そのタイミングによって、p(X)がp(X)と書かれるかp(3)と書かれるか、わからない。また、r(X,X)というゴールがあったとする。KL1の設計思想として、最初のXと2番目のXが同じ変数であるということはわからないことになっているから、アンパーザが、Xが変数の状態のときにプリントアウトしようとしても、r(X,X)と書くことはできず、r(X,Y)となってしまう。ふたつの変数が同一であることがわかるための組込みを用意することはできないわけではないが、たとえそうしても、アンパーザとプログラムの並列実行の問題から、必ずしも、r(X,X)と書くことができるとは限らない(たとえば、r(X,X)と書こうとしている間にXが3にバインドされてしまうかもしれない。そのタイミングによって、r(X,X)は、r(X,3)と書かれるかもしれないし、r(3,3)と書かれるかもしれない)。この点については、r(X,X)とプリントアウトするのは不可能とみた方がよいかもしれない。

5．4．3　ブロッキング

プロセス間、プロセサ間の同期や、通信を減らして効率をあげるために、通信データをブロッキングする機構を用意する(6.X.実現方式に関するコメント:ブロッキング参照)。

５．５　プログラム管理

プログラム管理システムは、並列型論理言語ＫＬ１で記述されたユーザ・プログラム及びＰＩＭＯＳプログラムの実行可能型式への変換、登録、管理機能を提供するシステムである。これらの機能は、具体的には次に示すシステムによって実現される。

（１）コンパイラ
（２）リンカ
（３）ローダ
（４）ライブラリ
（５）シンボル管理システム

プログラム管理システムの開発に当たっては、ＰＩＭＯＳ開発の各段階で最も必要かつ適当なものを選んで段階的に開発する。従って、表５．５－１に示すように、ＰＩＭＯＳ第１版ではクロス・システム上にコンパイラ及びリンカを作成する。ＰＩＭＯＳプログラムの開発もこれらクロス・システム上で行なうと同時に、ユーザ・プログラムのコンパイル及びリンクもクロス・システム上で行なう。ＰＩＭＯＳ第２版ではコンパイラ及びリンカも本体上に作成する。この章では、まずクロス・システムの概要を述べた後、上記（１）～（５）の各システムの機能及び実現イメージについて述べる。

表５．５－１　プログラム管理システムの開発段階

	ＰＩＭＯＳ第１版	ＰＩＭＯＳ第２版
クロス・システム	クロス・コンパイラ クロス・リンカ	クロス・コンパイラ クロス・リンカ
フロント・エンド	ライブラリ シンボル管理システム	ライブラリ シンボル管理システム
本体	 ローダ	コンパイラ リンカ ローダ

５．５．１　クロス・システムの概要

クロス・システムは、クロス・コンパイラ、クロス・リンカから構成される（図５．５－１）。ＰＩＭＯＳの下で走るユーザ・プログラム及びＰＩＭＯＳプログラムはＫＬ１－ｕ（モジュール化機能を持ったユーザ言語）で記述される。ソース・プログラムは、クロス・コンパイラによってモジュール単位でモジュール内相対形式コードにコンパイルされて、ファイルに出力される。この時点では、シンボルは自字名がそのまま書き出されている。

複数個のモジュール内相対形式コードは、クロス・リンカによってモジュール外呼び

出しの処理及びシンボルの内部データへの変換を行なって、相対形式コード（機械語命令列）をファイルに出力する。また、ライブラリ初期化情報ファイル、シンボル・テーブル・ファイル及びデバッグ用情報ファイルも出力する。

イニシァル・プログラム・ローダ（ＩＰＬ）は、クロス・リンカが出力した相対形式コードをファイルから読み込み、本体上のローダを用いて全要素プロセッサの主記憶上にロードする。相対形式コードから絶対形式コードへの変換はローダが行なう。次に、リンカが出力したライブラリ初期化情報ファイルとシンボル・テーブル・ファイルより、それぞれフロント・エンド上のライブラリ及びシンボル管理システムの初期化を行なう。

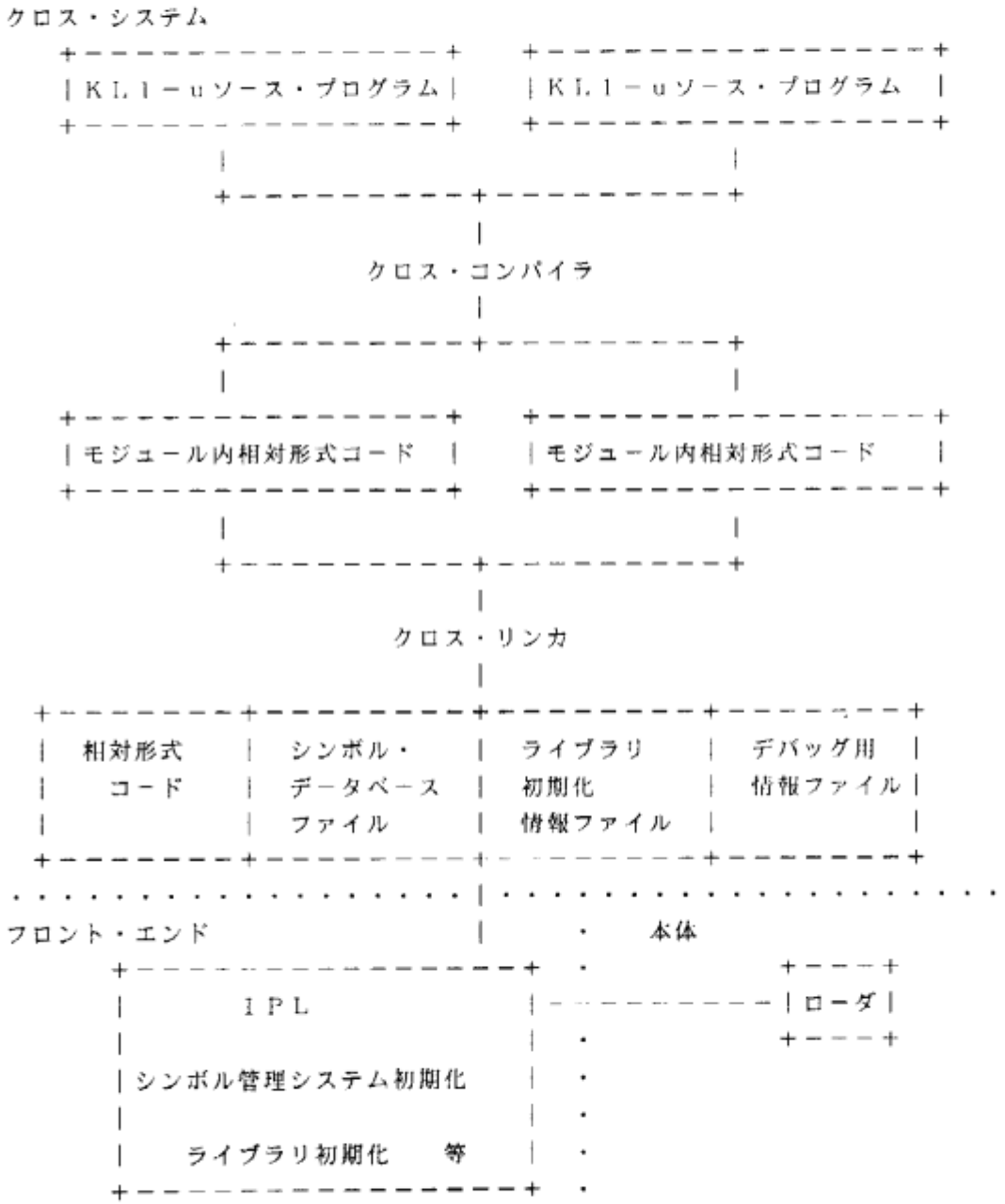


図５．５－１　クロス・システムの構成

５．５．２　コンパイラ

コンパイラは、ＫＬ１－ｕで記述されたユーザ・プログラム及びＰＩＭＯＳプログラムを、対象とするマシンの機械語（ＫＬ１－ｂ）にコンパイルするためのシステムである。ＰＩＭＯＳ第１版では、クロス・システム上のクロス・コンパイラを開発する。

クロス・コンパイラは、モジュール単位にＫＬ１－ｕでコーディングされたソース・プログラムを読み込み、－構文解析、－モジュール解析、－最適化、－オブジェクト形式変換を行なった後、モジュール内相対形式コードのファイルを出力する。この時点では、シンボルは印字名がそのまま出力されている。

５．５．３　リンカ

リンカは、コンパイラが出力した複数個のモジュール内相対形式コード・ファイルを読み込み、本体上のローダによって各要素プロセッサの主記憶にロード可能な相対形式コードを作成するためのシステムである。ＰＩＭＯＳ第１版ではクロス・システム上のクロス・リンカを開発する。

クロス・リンカは、クロス・コンパイラが出力した複数個のモジュール内相対形式コード・ファイルを読み込み、－モジュール間呼び出しの解析、－シンボルの内部データへの変換及びシンボル・データ・ベースへの登録を行なった後、（１）相対形式コード・ファイル、（２）シンボル・データベース・ファイル、（３）ライブラリ初期化情報ファイル、（４）デバッグ用情報ファイルを出力する。

５．５．４　ローダ

ローダは、リンカが出力した相対形式コードを絶対形式に変換し、各要素プロセッサの主記憶上にロードし、更に自要素プロセッサ上のコードの管理を行なうシステムである。ローダは本体上に開発される。

相対形式コードを絶対形式に変換して主記憶にロードするための方式には、次の２つが考えられる。

（１）静的なローダ

この方式では、全てのコードを全要素プロセッサにあらかじめロードしておく。従って、実現は容易であるが、主記憶の使用効率は悪い。

（２）動的なローダ

この方式では、あらかじめ最低限必要なコードのみを全要素プロセッサにロードしておき、その後は実行時にコードが必要となった時にエクセプション（ＵＮ－ＬＯＡＤＥＤ－ＣＯＤＥ－ＣＡＬＬ）機能を用いてフロント・エンドに通知した後、ライブラリを経由してコードを得た後ロードする。この方式では主記憶の使用効率は良いが、実現は

難しい。ＰＩＭＯＳ第１版では静的なローダと動的なローダを段階的に開発する。

静的なローダは、フロント・エンド上のＩＰＬからの要求に従って相対形式コードを絶対形式コードに変換した後、間接語テーブルにそのコードを登録してポインタを介して主記憶上にロードする。

動的なローダは、間接語テーブルに登録してから主記憶上にコードをロードするところまでは静的なローダと同じである。その後、ロードされていないコードが実行されようとした時に発生するエクセプション（ＵＮ－ＬＯＡＤＥＤ－ＣＯＤＥ－ＣＡＬＬ）に対して、フロント・エンドのライブラリに問い合わせて必要なコードを得、主記憶上にロードする、エクセプション・ハンドラの機能も有する。

また、動的なローダはコードのＬＲＵ（Ｌｅａｓｔ　Ｒｅｃｅｎｔｌｙ　Ｕｓｅｄ）管理を行ない、主記憶が不足してコード領域の一部を一時的に解放する必要が生じた時には、ＬＲＵコードから順に間接語テーブルから外す。外されたコード領域の解放は、ＧＣによって行われる。

５．５．５　ライブラリ

ライブラリはＫＬ１－ｕプログラムのモジュール情報、述語情報、相対形式コードの管理を行なうシステムである。本システムはフロント・エンド上に実現される。以下にライブラリの機能を列挙する。

（１）モジュール情報の管理

モジュールの名称とソース・プログラムの対応、及びモジュール間の関係等を管理する。ただし、ＰＩＭＯＳ第１版ではコンパイラ及びリンカはクロス・システム上にあるため、ソース・プログラムの管理は行なわない。

（２）述語情報の管理

モジュールに定義されている各種述語情報を管理する。

（３）相対形式コードの管理

モジュールに定義されている述語の相対形式コードを管理する。

ＰＩＭＯＳ第１版では、ライブラリはＩＰＬ後リンカが出力したライブラリ初期化情報ファイルを読み込んで、（２）、（３）の情報をハッシュ・テーブルを用いて初期化して管理する。

５．５．６　シンボル管理システム

シンボル管理システムは、シンボルの印字名と内部データ（アトム）の変換を行なう

システムである。シンボルはシステムで一括管理する必要があるため、フロント・エンド上に実現される。シンボル管理システムの機能をまとめると、次のようになる。

（１）印字名から内部データへの変換

シンボル管理システムに印字名を与えると、それが既にシンボル・テーブルに登録されているアトムであればその内部データを返す。まだ登録されていないアトムであれば、新しく内部データを生成してシンボル・テーブルに登録した後、その内部データを返す。

（２）内部データから印字名への変換

シンボル管理システムに内部データを与えると、シンボル・テーブルに登録されていれば、その印字名を返す。登録されていなければ、その旨を何等化の方法で伝える。

シンボル管理システムはプロセスとして実現され、他のシステムとはストリームを通して通信を行う。常にこのプロセスは他からデータ変換を要求するメッセージを待っており、要求が来る度に変換処理を行う。シンボル管理システムは印字名と内部データとの対応をシンボル・テーブルと呼ばれるハッシュ・テーブルを用いて管理している。ＰＩＭＯＳ第１版では、シンボル・テーブルはクロス・リンカが出力したシンボル・テーブル・ファイルを読み込んで初期化される。

また、シンボル管理システムが呼ばれる時には通常連続して呼ばれる事が多いと考えられる。この場合、その度にフロント・エンドと通信を行っていたのでは、通信のオーバヘッドが大変大きい。従って、各要素プロセッサにはシンボル・テーブルの一部をキャッシュしておき、通信量がなるべく少なくなるように作るのが望ましい。

次に、シンボル管理システムとは別の話になるが、アトム管理方式について簡単にのべる。シンボル管理システムは、印字名を持ったアトムを管理するためのシステムである。これとは別に、アトムというのは必ずしも印字名を持っているものだけではなく、単なる内部データとしてのアトムも存在するわけで、それらのデータを管理するための方式も検討する必要がある。印字名のないアトムは組込述語の実行によってのみ生成される。従って、各要素プロセッサが各々アトムを生成した場合、アトム管理を行わないと番号が重なってしまって矛盾が生じる。そこで、次のような方式でアトム番号を管理すれば、この問題は解決する。

・各要素プロセッサ毎に生成出来るアトム番号の範囲をあらかじめ規定しておく。

例：要素プロセッサが６４台の時は、次のようにアトム番号（２４ビット）の上位８ビットでプロセッサ番号を示し、下位１６ビットでそのプロセッサ内のユニークなアトム番号を示す。

フロント・エンド	１６' １００００００	～	１６' ＦＦＦＦＦＦ
ＰＥ＃０１	１６' ００００００	～	１６' ００ＦＦＦＦ
ＰＥ＃０２	１６' ０１００００	～	１６' ０１ＦＦＦＦ

PE#03　　16'020000 ～ 16'02FFFF

:

PE#64　　16'0F0000 ～ 16'0FFFFF

この方式のアトム管理を行う場合はファーム・ウェアのレベルで実現しなければならず、今後もさらに検討を要する。

５．６　入出力

入出力は、デバイス側のプロセスとシステム側のプロセスとの間で、ストリームにデータを流すという形で通信のやりとりを行うという形をとる(5.2.資源管理参照)。その際、ストリームに無制限にデータが流れるのを防ぐために、PIMOSでは、バッファを設けてその制御を行うようにする。また、PIMOSでサポートされる入出力は、プロセス間、プロセサ間の同期、通信を減らして効率をあげるために、基本的にはユーティリティでサポートされるブロッキング機構を用いて、ブロック単位で行われるが、インタフェースとしてキャラクタ単位の入出力もサポートする。

５．６．１　フロントエンド

本体上のPIMOSでは、KL1のストリームにデータを送るが、それを受けるフロントエンド側ではSIMPOSを用いる(6.X.実現方式に関するコメント:フロントエンド・インタフェース参照)。

フロントエンドには、次のものがあり、それぞれの主だった機能イメージは、以下のようである。

・端末

端末からの入出力には、SIMPOSのウインドウ・システムで提供する、標準入出力ウインドウ程度のウインドウを使用することができる。

・ファイル

SIMPOSのファイル・システムで提供する、バイナリ・ファイルを用意する。

・プリンタ

プリンタへの出力は、SIMPOSのスプール機能を使用して実現することができる。

５．６．２　時計

PIMOSは、ハードウェアの時計として、リアルタイム・クロックと、インターバル・タイマを持つ。ハードウェアとして用意されるリアルタイム・クロックは、プロセサごとに持つと、時計が一致しなくなる可能性があるため、システム全体でただひとつだけ持つこととする(プロセサ間で同期したクロックを用いることができるなら、プロセサごとに持っていてもよい)。PIMOSは、これを管理する組込みの時計プロセスとストリーム通信を行うことによって、様々な時計の機能を実現する。また、PIMOSは、論理上の時計を複数個持ち、それらを同時に管理することができる。

時計の代表的な機能としては、次のようなものがあげられる。

・日付と時刻

現在の日付、時刻を得る。

・ストップ・ウォッチ
荘園の実行時間を計測する。しかし、荘園のインプリメントのし具合によっては、うまく実行時間を計測するのは、むずかしいかもしれない。

・アラーム・ウォッチ
あらかじめ設定した時刻まで、プロセスの実行を中断して待たせる時計である。プロセスの中断、再開は、変数の具体化待ちという形で実現される。

５．７　タスク管理

タスクとは，あるひとかたまりの仕事の単位であり，ＰＩＭＯＳ全体でソフト的に管理されるものである。このタスクという概念は，ＫＬ１の言語プリミティブとしては存在しないが，荘園の機能を用いて実現される。ただし，荘園との違いは，ユーザが認識できる名前（識別ＩＤ）が付いている点である。従って，ユーザはその名前を用いることにより（プログラム中からまたは後述のシェルにより）タスクに対して種々の操作を行うことができる。

(a) タスクの生成（開始），中断，再開，放棄
(b) タスクで実行されるプログラムの実行優先度の指定／変更
(c) 資源（リダクション数，メモリ量）の割り当て／変更
(d) 資源が不足した時の対処（資源の再割り当て，タスクの放棄 等）
(e) 実行分散の位置と範囲の指定
　　例：　あるＰＥを中心にして，ある範囲で実行させる。

また，前述したようにＰＩＭＯＳは，タスク単位にＩ／Ｏ資源を管理しており，Ｉ／Ｏ資源の確保及び解放の機能を提供する。

ＰＩＭＯＳのタスクでは，その子供として任意個のサブ・タスクを生成することができる。このとき，ＰＩＭＯＳはそれらのサブ・タスクを階層的には管理していない。この意味は，あるタスクの中からＰＩＭＯＳが管理している総てのタスクが見えるということであり，例えば，兄弟タスクを殺す（放棄する）といったことが可能である。

（１）シェル

ユーザのタスクに対する操作を簡便に行えるようにするユーティリティがシェルである。ユーザは，このシェルにより上記 (a)～(e) のような操作を行える。ただし，シェルにおいては，そこで生成したタスクの階層的な管理も行っている。例えば，あるタスクのサブ・タスクだけを表示して，それらに対して操作をするといったことが可能である。

また，ＰＩＭＯＳのシェルは，ＵＮＩＸのパイプのように複数のプログラムをストリームにより繋げて実行させることができる。このパイプは，単なるストリーム通信の機能の他に，データ転送のブロッキング／デブロッキングの機能を持っている。また，パイプによりプログラムを繋げた場合，ＰＩＭＯＳは全体をタスクとして扱うので，ユーザは個々のプログラムに対して上記のような操作を行うことはできない。

５．８　デバッグ支援

デバッグ支援は、並列型論理言語ＫＬ１で記述されたユーザ・プログラムのデバッグを支援するための各種機能を提供するシステムである。ユーザ・プログラムに限らず、ＰＩＭＯＳプログラム自身のデバッグも支援できなければならない。この章では、まずデバッガの種類について述べた後、そのうちの代表的なプログラム・アナライザ、トレーサ及びシミュレータの機能と実現イメージについて述べる。

５．８．１　デバッガの種類

デバッガには、大きく分けて静的なデバッガと動的なデバッガとがある。また、これらはクロス・システムのシミュレータ上に実現されるものと本体上に実現されるものとがある。

（１）静的なデバッガ

静的にＫＬ１プログラムを解析して初歩的な虫を見つけるためのデバッガで、プログラム・アナライザと呼ばれる。。

（２）動的なデバッガ

ＫＬ１プログラムを実行させて、その実行状態あるいは実行結果を見ながらデバッグするためのツールである。１つには実行状態をトレースするトレーサがある。また、実行結果をバグ検出の各種アルゴリズムに従って解析して虫を見つけるアルゴリズミック・デバッガがある。その他にも各種の動的なデバッガが今後考えられるであろう。

静的及び動的なデバッガは本体上で実現されるとともに、本体の動きをクロス上でシミュレートするシミュレータ上で実現される。次に、ＫＬ１プログラムをデバッグする一般的な手順を示す。

ＫＬ１で記述されたプログラムのデバッグは、まずプログラム・アナライザを用いてプログラムを静的に解析して初歩的な虫取りを行う。

次にクロス上のシミュレータを用いて１台プロセッサ上で実際にプログラムを動作させ、動的なデバッガを用いてプログラムの論理的な虫取りを行う。次に、プラグマ指定方法を各種変え、プロセッサの台数も変えて負荷分散アルゴリズム等のデバッグも行なう。

次に本体上で実行し、動的なデバッガを用いて最終的な虫取りを行なう。

５．８．２　プログラム・アナライザ

ＫＬ１のプログラムは、Ｐｒｏｌｏｇのプログラムと同様にソース・プログラムを解析してある程度の初歩的な虫を見つけることが出来る。この解析ツールをプログラム・アナライザと呼ぶ。以下にその機能を列挙する。

- 文法ミスを見つける。

- 変数名のミス・スペルを見つける。

1クローズ中に1度しか現れない変数名はミス・スペルした可能性がある。

- 未定義述語呼び出しを見つける。

モジュール内の述語呼び出しのクロス・レファレンスを得、未定義述語呼び出しを見つける。

- 静的に判るデッドロックの検出

モード・インファレンスを行ない、デッドロックを検出する。

- マルチプル・ライタの検出

モード・インファレンスを行ない、複数のゴールが同じストリームに書き出している部分を検出する。

このほかにも、Ｐｒｏｌｏｇプログラムの静的解析ツールの多くが利用でき、更に今後の研究で各種の機能が考えられであろう。

５．８．３　トレーサ

トレーサは、ＫＬ１プログラムが実行される過程をモニターし、その途中で実行を制御する事によってデバッグを行うためのシステムである。Ｐｒｏｌｏｇのような逐次型プログラムは、記述されている順序に従って実行されるのでトレーサは作成が容易で、虫取り作業も人間にとって一般的には操作及び理解がしやすい。ところが並列プログラムのデバッグは、プログラムに記述されている順序通りには動かないため、単純に実行過程をモニターしても虫取り作業は大変困難だと考えられる。従って、どのようなトレーサがＫＬ１にとって一番ふさわしいかはこれから更に検討を要する。ここでは、ＫＬ１プログラムが実行される過程をモニターするためのトレースの方式、トレーサに必要な機能及びトレース対象の限定方式に着目し、それらの特長及び問題点について述べる。

（１）トレーサの方式

トレース方式を実現イメージから分類すると次の４方式が考えられる。

① Ｍｅｔａ　Ｉｎｔｅｒｐｒｅｔｅｒ方式

・ＧＨＣプログラムをターム形式で保持し、荘園機能を用いてＧＨＣでインタプリタを記述する。インタプリタの中にトレース表示及び実行制御を行うためのトレーサを組み込む。

・荘園機能ではＳＵＣＣＥＳＳ／ＦＡＩＬしか判らないため、ＳＵＳＰＥＮＤ／ＤＥＡＤＬＯＣＫのトレースは表示できない。

・ＦＡＩＬ－ＳＡＦＥなユニファイがＧＨＣにはないため、インタプリタの記述は大変注意を要する。すなわち、ユーザ・プログラムのＦＡＩＬがインタプリタに浸透しないようにインタプリタを作成せねばならない。

・実行をコントロールするのが難しい。

② Ｌｏｗ Ｌｅｖｅｌ Ｉｎｔｅｒｐｒｅｔｅｒ方式

・ＧＨＣプログラムを機械語（ＫＬ１－ｂ）と同じレベルの中間言語にコンパイルし、この中間言語を実行するインタプリタをＧＨＣで記述する。すなわち、ファームウェアがやっているのと同じ事をＧＨＣで記述する。

・ファームウェアの処理と同じ事をＧＨＣで記述するため、インタプリタの作成は大変困難で、かつ実行速度は大変遅くなる。

・実行のコントロール等、大抵の事が実現出来る。

③ Ｆｉｒｍｗａｒｅから情報をもらう方式

・トレースを行うタイミング（例えばリダクションを行うタイミング等）をファームウェアが検出して、エクセプションの機能を用いてトレーサに情報を渡す。

・ファームウェアでトレースを行うのに近く、実行速度は速い。

・どのような情報をファームウェアからもらうか、及びインターフェースの仕様を明確にしなければならない。また、ファームウェアの処理が重くなる可能性があるため、充分な検討を要する。

④ Ｐｒｏｇｒａｍ Ｔｒａｎｓｆｏｒｍａｔｉｏｎ方式

・コンパイル時にトレース用のコードを埋め込む。トレーサを用いて実行する時は、このトレース用のコードによって渡されるデータをもとにトレース表示を行う。

・コンパイル時にすべてのコードにトレース用のコードを埋め込んだ場合は、通常の実行が遅くなる。ただし、トレース時とトレースを行わない時の実行は同じである。

・コンパイラにトレース実行用と通常実行用の２つのモードを設けた場合は、通常の実行時の速度低下はなくなるが、トレース時とトレースを行わない時の実行が違う場合がある。以上のように、ＫＬ１プログラムの実行過程をモニタするためのトレース方式には幾つかあるが、どれを採用するかは今後検討した結果決定する。

（２）トレーサに必要な機能

（１）で示したどのトレース方式を採用するにせよ、トレーサには次に示すような機能を備えているのが望ましい。但し、トレース方式によっては実現不可能な機能もある。

① どのようなリダクションが実行されたかを表示する。

② どこでＦＡＩＬしたかを表示する。

③ どのようなゴールがいつＳＵＳＰＥＮＤしたかを表示する。

④ 実行のコントロールが出来る。

ＲＥＡＤＹ－ＱＵＥＵＥの内部を表示し、ゴールの実行順序を変えられる。このためには、ゴールのプライオリティをトレーサが変えられる機能が必要である。

⑤ クローズレベルのトレースが出来る。

Ｐｒｏｌｏｇのプロシジャレベルのトレースだけでなく、どのクローズのガードでＳＵＣＣＥＳＳ／ＳＵＳＰＥＮＤ／ＦＡＩＬしたかを表示する。

⑥ 変数／ストリームのモニタが出来る。

特定の単純変数及びストリームに対する値のバインド過程をモニタし、バインドされる度に表示する。

⑦ プロセスの特定の入力に対応した出力を表示する。

各プロセスをフィルタと考え、ある入力に対する出力との対応関係を表示する。

トレーサに必要な機能は以上の通りであるが、トレーサを検討していく上で更に機能は追加されていくものと思われる。

（３）トレース対象の限定法

トレースを行う際には、実行過程等をすべて表示するのは情報量が多すぎて実際上大変使いにくい。従って、トレースするべき対象を限定する事が必要である。次に、そのための方式について述べる。

① ＳＰＹ方式

・述語名を指定して、その述語が実行された時にのみトレース表示を行う。

・パターンを指定して、あるデータがそのパターンと一致した時のみトレース表示を行う。

② ＴＲＥＥ方式

・特定のゴールの子孫のみをトレースする。

③ データ依存方式

・特定の変数がバインドされたらトレースを開始する。又は、特定の変数が指定した値にバインドされたらトレースを開始する。

・データに注目して、そのデータが流れて行く様子をトレースする。

④ ＰＲＯＣＥＳＳ方式

・特定のプロセスに関係するゴールの実行のみをトレースする。

⑤ ＦＩＬＴＥＲ方式

・特定のデータがプロセスを渡って行く様子をトレースする。

５．８．４　シミュレータ

シミュレータはクロス上で本体の動きをシミュレートするシステムである。シシュレータはトレーサ及びその他の動的デバッガを備えており、パラメータの指定等によって要素プロセッサの台数を変えることが出来る。従って、プロセッサ台数を１台にして実行させればプログラムの論理的な虫取りが出来、その後台数を増やして負荷分散アルゴリズムのデバッグを行なえる。

以下にシミュレータの長所及び短所を述べる。

（長所）

・必ず再現性がある。

実行環境を再現させて、全く同じ現象を再現させる事が出来る。

・ネットワークに関連する各種状況をシミュレート出来る。

ネットワークに関連した各種状況を疑似乱数等を用いる事によって簡単にシミュレート出来るため、特にＰＩＭＯＳのデバッグを行なう際に役立つ。更に、疑似乱数を用いれば同じ現象を再現する事が出来る。

（短所）

・実行速度が遅い。

・本体の動きを完全にシミュレートすることは出来ない。

5．9　低レベル・サポート機能

・立ち上げ

まず，フロントエンドがPIMOS の最小限のカーネル部分だけをメンテナンスパスで各PEに転送し，ネットワークを立ちあげる。次に，このネットワークを用いて各PEの立ち上げを行う。
（注：　カーネル部分は，インプリメントによる。）

・低レベル・デバッグ
ファームサポートで，KL1-B ステッパーの様なものを提供する。

・メンテナンス
本体を停止せずに，実行監視できるようにしたい。また，PE個数が数十，数百のオーダーになった場合，常にそのうちの一台や二台メンテナンスしなければならない状態にある可能性がある。このとき，運用中の PE と物理的に結合したままで，一部のPEをメンテナンスすることは困難が予想される。したがって，実機の運用から筐体単位ぐらいで自由に切離し可能なシステム構成にされることが必要である。

・エラー処理
ここでは，KL1 ではなくファームウエアで扱うエラーについての処理について述べる。

・1ビット・メモリパリティエラー
フロントエンドに報告をすることは，まず必要である。さらには，頻繁にエラーの起こるページを切離すような処理をフロントエンドの指定にしたがって，あるいは，勝手にPE内でするか，という問題がある。

・ハードエラー
CPU 内パリティエラー，キャシュパリティエラー，電源異常などが考えられるが，これも上と同様にフロントエンドに報告する必要がある。

5．10　計測評価

・計測評価項目

(i) 単体性能

まず,従来の逐次型マシンで行なわれたような計測は,必要であろう。すなわち,
各種ベンチマークテストを用いた測定...
ユニフィケーション速度
データベースの検索速度
組込み述語の処理速度
KL1-B 命令の種類別実行頻度 (ヒストグラム)
単体の処理速度(単位時間あたりのリダクション数)

このため,以下のような測定ができることが必要となる。
リダクション数,サスペンション数,実行(稼動)時間,
メモリ使用量,GC時間,GC回数,GCによるメモリ回収量

さらには,デリファレンス回数も可能であれば計測が望まれる。
以上の測定は,PEの内部コストを求めるという意味を持つ。

(2) 通信に関する性能

(1) に対して,PEの外部(通信)コストを求めるために,以下のような項目が必要となろう。

メッセージの種類別
通信のためにかかる処理時間(encode/decode,輸出表のメンテナンス,etc.)
通信メッセージ数(送信/受信)
通信パケット長(の総数)
通信ヒストグラム

(3) システム全体の性能

システムの性能評価で知りたい事は,システム全体で処理した方が単体で処理するよりもどれだけ良い/悪いか,なぜそうなるのか,という事である。
そのためには,以下が計測できれば良いだろう。
全体の処理速度
各PEの仕事量,各PEの稼動率,全体から見た仕事の分散度合
各PEの通信コスト,全体の通信コスト
外部に投げ出す仕事量と内部との割合　・計測方法

逐次型マシンの計測では,計測用の処理を組入れても,そのオーバーヘッド分が別に計測でき,実際のシステムの性能が正確に測定できた。

ところが並列マシンでは,実際的プログラムでオーバーヘッドを定量的に求めることが極めて難しい上,その非同期的動作ゆえに実行に再現性が期待できない。

従って,システム全体の評価のためには,実際のシステムに手を加えないもの,あるいは測定オーバーヘッドの極めて少ないものによるべきである。

以上の観点から,以下の方法が考えられる。

(1) ハードサポートにより,測定オーバーヘッドを考慮せずに済むツールによるもの
例) 現在のマルチPSI では,GEVCで通信パケット数を計測している。

(2) システムに組込まれているもの,
あるいは,組込んでしまってもオーバーヘッドの少ないもの

例) リダクション数,サスペンション数,稼動時間

(3) 要素プロセサ単体,ないし二台で,計測できるもの
これは,従来の逐次型マシンでと同様に,測定オーバーヘッドが測定できるものでオーバーヘッドがあっても構わない。

例) 通信あたりのパケット長,通信あたりのコスト ... 2台
ユニフィケーション速度,データベースの検索速度,
組込み述語の処理速度 ... 1台

(4) 上記の実測値から計算(推測)できるもの

例) 通信パケット長がシステム組み込みで計測でき,単体でパケット長あたりの通信コストがわかれば,そのときの通信コストが計算できる。

(5) 実行とは別に計測できるもの

例) メモリ使用量は,実行の前後に実行とは別に計測すればよい。
(これについては,MRB を導入すると正確には計測できないであろう)

・インプリメント・イメージ

計測は各要素PEで行い,計測結果はメンテナンス・パスを通じてフロントエンドに回収する方がよいだろう。 ネットワークを通じてデータを回収するならば,回収用の通信が計測値に加算されないような工夫が必要である。また,時間が大幅にかかるような計測を自動運転ですますような工夫や,回収したデータの集計・編集ツールも必要である。

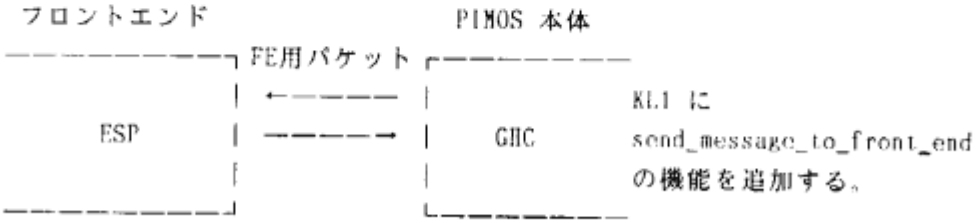
6．実現方式に関するコメント

本章では、PIMOS及びＫＬ１のPIMOSサポート機能の実現方式に関して、その概略を述べる。

6．1　フロントエンドとＰＩＭＯＳ　本体とのインタフェース

フロントエンドとPIMOS 本体の通信は，パケットで行なうものとする。実現イメージとしては，以下のような方法が考えられている。

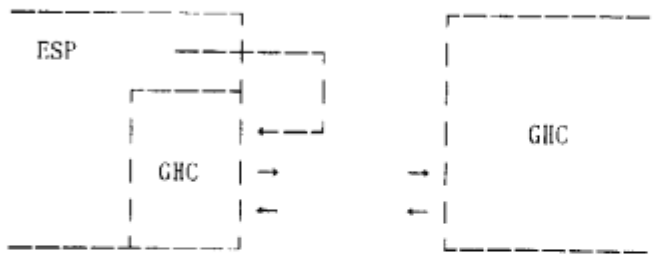
方法1　フロントエンドには，GHC 処理系を置かず，フロントエンド用ネットワークを作成する。



ESP から GHC へは，ファームサポートが必要となる。

方法2.1　フロントエンドにも GHC 処理系を置き，GHC 用のネットワークを作成する。
フロントエンド内で ESP-GHC の変換を行なう。
しかし，GHC から ESP への変換は，新しくtrapのサポートを追加することですむとしても，ESP から GHC への変換は困難である。

方法2.2　フロントエンドにも GHC を置き，ESP からGHC への変換は，本体からの通信のような形で行なう。



また，この方法であれば，pseudoを作成する際もネットワーク以外は，同じにすることが可能である。

方法3　ネットワークはGHC 用とし，フロントエンドには，GHC 用パケットを解釈するESP プログラムを置く。

６．２　ＫＬ１のＰＩＭＯＳサポート機能の実現イメージ

ＰＩＭＯＳのような実際的なプログラムの実現に必要なＫＬ１の拡張機能仕様と，その実現イメージについて述べる。

６．２．１　ゴール間の優先度指定

（１）　ＫＬ１Ｂでの表現

① 優先度付きのエンキュー命令の用意

```
enqueue_with_priority Predicate, An
```

ｅｎｑｕｅｕｅ＿ｇｏａｌ命令と同様だが，通常のｅｎｑｕｅｕｅ＿ｇｏａｌが親ゴールと同じ優先度のキューにエンキューするものであるのに対し，レジスタＡｎで指定した優先度のキューにエンキューする点が異なる。なお，エンキュー命令を用いないのなら，ゴール作成命令中に同様のプライオリティ指定をすることになる。

勝手に優先度の高いゴールを作ることができないようにする方法は何か必要である。たとえば，現優先度を超える優先度指定はエラーとするのは簡単な方法である。

② ｅｘｅｃｕｔｅ／ｐｒｏｃｅｅｄ命令での高優先度ゴール確認操作

ゴールリダクション終了後，次のリダクションを開始する前に，リダクション中により高い優先度のゴールがレディにならなかったかどうかを確認する必要がある。もしそのようなゴールがあれば，その優先度の実行を先に行なう。高優先度のゴールがレディになる原因としては，ボディユニフィケーションの結果サスペンドしていた高優先度のゴールがレディになった場合と，上述の優先度付きのエンキュー命令によって高優先度のゴールが新たにエンキューされた場合がある。

（２）　実現イメージ

① 優先度別のレディキュー

優先度ごとに別々のレディキューを持つ。優先度の種類だけの大きさを持った配列（キューテーブル）を用意し，各キューの先頭を配列の要素とする。キューは単方向リストとして実現する（ゴールレコード中のフィールドを用いる）。レディ状態のゴールに関しては，どのキューに入っているかで優先度がわかるので，ゴールレコードには特に優先度情報を持つ必要はない。

② サスペンドしたゴールの優先度

サスペンドしたゴールについては，変数の具体化を契機に起動する際に優先度を知る必

要がある。そこで，サスペンションレコードには優先度を記録するフィールドが必要である。

③ 実行中のキュー

処理系は常にその時点で最大の優先度を持つキューのゴールに対してリダクションを行なう。処理の高速化のために，処理系は実行中のキューの先頭（これを単にレディキューと呼ぶ）とその優先度（現優先度）をレジスタ上にキャッシュする。

④ 高優先度レディゴールの検出

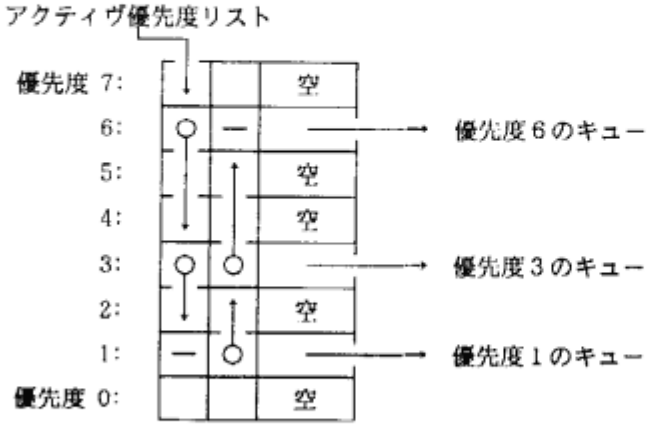
実行中優先度より高い優先度のゴールがレディになったことは，ｅｘｅｃｕｔｅ／ｐｒｏｃｅｅｄ命令でチェックする割出し原因のひとつに加える。この割出しフラグは優先度付きのエンキュー命令や，変数の具体化の際のサスペンディドゴールのエンキュー等の際に，エンキューするゴールの優先度と，現優先度を比較して，必要なら立てる。このとき同時に，その優先度をどこかに記録する。すでに高優先度のゴールがエンキューされている場合は，その優先度と比較し，高い方を記録する。

⑤ 次優先度キューの探索

最高優先度のキューが空になったときは，つぎに実行すべき優先度を探さなくてはならない。単純な構成では現優先度を開始点に空でないキューを順探索しなければならず，優先度の種類が多い場合，このオーバヘッドは大きい。そこで，キューが空でない優先度をソートした双方向リスト（アクティヴ優先度リスト）で管理すれば，この探索の手間をコンスタントにできる。

⑥ アクティヴ優先度リストの管理

現優先度よりも低い優先度の空のキューに新たにエンキューする場合，アクティヴ優先度リスト中にその優先度を挿入する必要がある。適当な挿入位置を見付けるには順探索が必要になる。この探索の対象は空でないキューだけなので，可能な優先度の種類にはよらない。より高速の探索が必要なら，空でない優先度を均衡二分探索木で管理することもできる。



６．２．２　クローズ間の優先度指定

クローズ間の優先度を表現するために，クローズの試行に逐次性を持たせる命令を導入する。

```
try_me_thoroughly_else Next_Clause
```

ｔｒｙ＿ｍｅ＿ｅｌｓｅ命令と同様であるが，前後のクローズの失敗またはサスペンションを確認するまでは次のクローズに進まない。

６．２．３　負荷分散制御

（１）負荷の定義

プロセサの負荷は，ある時間間隔中に実行したゴールの優先度の時間平均値であるとする。たとえば，１０秒間に優先度１００のゴールを５秒間，２００のゴールを２秒間実行し，あとの３秒間はアイドルであったとすると，このプロセサの１０秒間の平均負荷は：

（１００×５＋２００×２）÷１０　＝　９０

と算定される。この定義は以下のような性質を持つ。

① 優先度が負荷の定義に反映されている。同じ量の仕事をしていても，優先度の高い仕事をしているか低い仕事をしているかで「忙しさ」は異なる。

② アイドルする時間が長くても，非常に優先度の高い仕事を時々処理するプロセサは，低い優先度の仕事を常に処理し続けているプロセサより「忙しい」。アイドル時間の評価を変えるには，たとえば負荷の計算に優先度自体ではなく優先度プラス適当な定数を用いればよい。

③ 同じ優先度の仕事を休みなく実行し続けている限り，キューが長かろうと短かろうと，同じ「忙しさ」と考える。どちらかのキューが空になってはじめて「忙しさ」の差がでる。優先度をまったく指定しないと，株分け方式と同様，アイドルプロセサにのみ仕事を分け与えるという負荷分散方式になる。

（２）　実現イメージ

① 優先度別の実行時間測定と乗算による方法

処理系はある優先度のキューを実行開始してから，なんらかの原因で他の優先度のキューに移るまでの経過時間を計測する。それに優先度を乗じたものを一定時間積算すれば，負荷を測定したことになる。ひとつの優先度のキューの実行がどの程度の時間続くかによって乗算が必要となる頻度は変わる。

③ 速度可変の時計による方法

通常のタイマは単位時間あたり１づつ加算ないし減算される。この加減量を可変にできるようなタイマを考える。このようなタイマがあれば，単位時間あたりの加算量として現優先度を与えることによって，乗算なしで負荷の積算が可能である。

６．２．４　荘園の機能

（１）ＫＬ１Ｂでの表現

① 荘園の作成

```
create_manor Code/Arity, Ac, Ar
```

新たな荘園を現在実行中の荘園の子荘園として作り，その荘園のＣＯＮＴＲＯＬ，ＲＥＳＵＬＴの両ストリームへポインタをそれぞれＡｃ，Ａｒに置く。Ｃｏｄｅ／Ａｒｉｔｙは荘園のトップレベルとして実行すべきゴールとその引数個数である。

この命令は，実現方式によってゴールのｃｒｅａｔｅ命令またはｅｎｑｕｅｕｅ命令のかわりに用いられる。ｃｒｅａｔｅ命令のかわりとして用いる仕様なら，ゴールレコードの作成もこの命令で行なう。引数の用意は通常のゴールと同様，セット系の命令によって行う。

② レイズ機能

```
raise An
```

積極的に例外を生起する命令を用意する。

③ 通信メッセージ中のアトム

荘園の実行を制御するＣＯＮＴＲＯＬストリームや，実行結果などを知るＲＥＳＵＬＴストリームには特定のアトムやファンクタが流れるものとした。これらのアトムやファンクタの表現形（アトム番号など）は，あらかじめ決めておく必要がある。

（２）荘園とそのネスティングの実現イメージ

① 荘園は荘園レコードで表現され，荘園に関する各種の情報はここに置く。

② 各ゴールレコード，里親，荘園は，それぞれが属する荘園または里親（以下，単に親と呼ぶ）のレコード（親レコード）へのポインタ（親ポインタ）を持つ。現在リデュース中のゴールについては，引数とともに，親ポインタ，さらに親レコード中の頻繁に用いるフィールドもレジスタ上にキャッシュして置く（親キャッシュ）。

③ 単純な再帰呼出し以外の新たなゴールのリダクションを開始する場合は，リデュース対象ゴールの親が今までと変わる可能性がある。このような場合は，新たにリデュース対象となるゴールの親ポインタと，キャッシュしてある親ポインタを比較し，もし異なっていれば親キャッシュのスワップを行なう。ひとつの荘園の中のゴールのみを実行し続けていれば，親キャッシュのスワップはまったく不必要である。

④ 親レコードには実行状態フラグを置く。このフラグの値としては，

実行中：正常に実行中。子ゴールのリダクションを行なってよい。

中断中：なんらかの理由で実行を中断している。この状態のときにリダクションを行なおうとした子ゴールは，サスペンドしてこのフラグにキューする。フラグが「実行中」に変わったときに，サスペンドキューからレディキューに移す。

放棄：　実行は放棄された。この状態のときにリダクションを行なおうとした子ゴールは，実行を放棄する。

⑤ 荘園レコードからは，すべての子荘園と子里親にアクセスできるパスを設ける。このためには，子荘園や代理親を双方向リストなどで管理する必要がある。

（3）実行の中断と放棄

① ＣＯＮＴＲＯＬストリームから荘園内の実行の中断／放棄を指定すると，その荘園の実行状態フラグを中断／放棄にするだけではなく，すべての子孫荘園と子孫里親の実行状態フラグも再帰的に中断／放棄にする。

② 荘園の実行を中断したとき，すでに子荘園は実行を中断していることもある。この場合，片方の中断だけが解けたときに実行を再開してはならない。実行を中断しようとする荘園がすでに中断状態にあったなら，自分の荘園の実行を中断するような高い優先度の特殊なゴールを実行状態フラグにエンキューしておく方法が考えられる。片方の中断が解けると，このゴールが最初に実行され，再び中断することになる。この方式は複数のパスから同時に中断を指定する場合のオーバヘッドは大きいが，その頻度が小さいとすると，荘園レコードにカウンタを設けるようなメモリ面でのオーバヘッドがないという利点がある。

（4）　資源管理の基本方針

資源割当ては必要に応じて，ゴールは親に，親はまたその親に要求して，親に許された資源割当量の一部を貰ってくる，という方式を基本とする。

① ゴールの直接の親に許された資源割当量は，親キャッシュの一部になっている。通常は，ゴールリダクションのため消費した分をここから減算していく。減算結果が負になった場合は，親に割当てた資源は尽きたことになる。

② 親に割当てられた資源が尽きたら，親はそのまた親（以下，祖親）に要求して自分

に資源を割当てる。祖親から資源を割当てられるまでは，その親に属するゴールの実行は延期される。具体的には，その親の実行状態フラグを「中断」にし，ゴールはそこにエンキューする。

③ 荘園には祖親に要求してよい資源量が記録されている。これが荘園に対するＣＯＮＴＲＯＬメッセージで追加されるようなものである。荘園に対して子供の里親，ゴール，子荘園から割当て要求が来たときに，自分にすでに許されている資源量が尽きていて，しかも祖親に要求してよい資源量が０である場合は，資源を使いきった旨の例外を起こす。

④ 祖親に要求を出したところ，そこでもすでに割当てられている資源が尽きたら，②の操作を再帰的に適用する。

⑤ 祖親から資源を貰ってきたら，実行状態フラグを「実行中」に戻し，そこで待っていたゴール群をレディキューに移す。祖親への資源割当て要求がサスペンドしているものも，通常のサスペンディドゴールと同様の形式にしておくと，統一的に扱えるであろう。

⑥ 親に対する割当て要求の回数を減らすために，一回の割当量はある程度大きくする。たとえば実行時間１秒，１００Ｋ語など。割当てた資源を使いきる前にその親に属するすべてのゴールのの実行が終了した場合は，残った資源は祖親に返却する。資源割当ての制御制度は，最悪一回の割当量と里親を作ったプロセサ台数の積程度になる。

（６）　実行時間の管理

計算の実行時間を資源として管理するには，インタバルタイマを親キャッシュとして用いる以下のような方式を取ればよい。

① 新たな親に関するリダクションを開始するときには，その親にすでに許された計算時間をインタバルタイマに設定する。

② その親の実行中にタイマが尽きた場合は，上述の方針にしたがって，祖親から資源を貰ってくる。

③ タイマが尽きる前に親が他に切り替わる場合は，インタバルタイマの残り時間を親レコードに書き戻す。

（７）　資源消費量の検査時期（ガードでの「大きな」計算）

① 資源管理のための資源消費量の検査は，後の続行に必要な情報がひとまとまりになっている，ゴールリダクションの切れ目に行なうのが便利である。具体的には，ｐｒｏｃｅｅｄ／ｅｘｅｃｕｔｅ命令で行なう。

② あるリダクションでどのぐらいの資源を消費するかは予測しにくい。そこで，資源消費量の検査はリダクションの終了時に，後処理方式で行なうのが適当であろう。すると，１リダクション中に消費する程度の資源は，制限を超過する可能性もある。この程度は誤差として認めてよいであろう。

③ １リダクション中の資源消費を誤差と考えてよいためには，誤差をある程度小さくする必要がある。ひとつのクローズ内の普通の実行は，資源消費の面でも「小さい」と考えてよかろう。しかし，「大きな」組込述語，たとえば配列の割付けのような大きなメモリ割付けをする（かもしれない）組込述語や，文字列探索のような長い計算時間がかかる（かもしれない）組込述語は，ガード部ではなく，ボディゴールとして実行し，必要なら途中でサスペンドできるような枠組みを考えるのが適当である。

（８）例外処理

例外発生時の処理系の動きは以下のようになろう。

① 例外の検出

例外は，組込述語実行中の例外ならガード部の実行中に，候補節が全部フェイルしたことはサスペンド命令の実行中に，ボディユニフィケーションの失敗はユニフィケーションの実行中に，レイズによる積極的な例外の生起はレイズ命令の実行中に，それぞれ検出される。ボディゴールの定義体がないことの検出（後述のモジュール間リンケージ機能に必要）は，ゴールのエンキュー時に行なってもよいが，デキューしてリダクションにかかるときに行なうのが適当であろう（エンキュー時に例外を出してコードをとってきても，デキューするまでにスワップアウトされてしまうこともある）。

② 例外情報の準備

例外原因情報（例外情報中のＩＮＦＯＲＭＡＴＩＯＮフィールド）を用意する。例外原因は検出者が知っている。レイズについては引数中に指定がある。

例外情報は，ＰＳＩ／ＫＬ０の例外処理とほぼ同様である。しかし，例外を生じたゴールとして報告されるものが，例外の直接原因であるガード部中の組込述語呼出しではなく，リダクション中に例外を生じたようなゴール全体であるから，ＧＯＡＬの情報以外に直接原因となった組込述語の引数情報なども渡す必要がある。たとえば加算でのオーバフローについては以下のようなものがあろう。

割込原因：	オーバフロー
原因となった述語：	組込みのａｄｄ
原因となった引数：	加算したふたつの数値データ
例外場所の詳細：	クローズ番号？ 述語内置負セット？

③ 例外原因ゴール情報の準備

例外原因となったゴールをＧＯＡＬフィールドに渡す準備をする。

まずどの述語かの情報を用意する。処理系としては述語に対応する命令番地，ないしはそのための間接語番地は知っているが，述語名までは知らない。処理系としては分かる範囲の情報，たとえば，コード型のオブジェクトや述語番号（後述）を供給し，あとの操作はソフトウェアに任せる。

つぎに，引数の情報を準備する。リダクションを開始前に検出した場合や，ガード部の組込述語やサスペンド命令で検出した場合は，ゴール引数はすべてレジスタ上にあるので，これを用いればよい（コンパイラは，例外を生じる可能性のある組込述語の実行までは，引数レジスタを保存するように命令を生成しなくてはならない）。引数個数については，実行中のコードから知るしかないであろう。ボディユニフィケーションの失敗に関しては，失敗の原因となったふたつのタームだけでよい。

④ 続行ゴール指定の準備

続行ゴール指定を行なうための変数セル（ＮＥＷＧＯＡＬに渡すもの）を生成する。この値としてゴールが与えられたときにそれを実行するようなゴールを生成し，これを変数セルにキューする。続行ゴールの指定も処理系に分かるような形式（コード型オブジェクトと引数列など）で与えられるものと思ってよい。

⑤ 例外情報タームの作成と送出

異常で用意したデータをもとに例外情報のタームを作成し，例外を起こしたゴールの親荘園のＲＥＰＯＲＴストリームに流す。

６．２．５　配列

配列操作の組込述語に対応する命令を用意する。ただし，ａｒｅｆ以外はガード部では実行できないものとする。

配列の更新操作には，原則としては配列全体のコピーが必要になる。これを常に一定の手間で実現するためには，配列要素の更新時に，更新前の配列を直接書き替えてしまうしかない。ＭＲＢなどにより参照数の管理を行なえば，参照数が１の配列については，このような更新が可能である。実現の都合上，このような参照数が１の配列以外の更新を許さないものとしても，あまり問題はないであろう。また，必要なら，Ｐｒｏｌｏｇにおけるマルチ・バージョン・ストラクチャと同じく，ひとつの物理的な配列と連想リストの組み合せによる実現手法を取れば，かなり効率的な実現も可能である。

６．２．６　コードの扱い

（１）　ＫＬ１Ｂでの表現

① コードの生成

```
create_code  Asize, Ainstr, Acode
```

Ａｓｉｚｅ（正整数）で指定する大きさのコード型オブジェクトを作成し，Ａｉｎｓｔｒで指定する内容に初期化して，Ａｃｏｄｅとユニファイする。Ａｓｉｚｅ，Ａｉｎｓｔｒおよびその内容が具体化されるまではこのプロセスはサスペンドする。Ａｉｎｓｔｒは上述の配列型であるとし，大きさの情報まで持たせてしまうのが簡単か？

② コードの呼出し

```
apply  Acode, Aargs
```

通常のボディゴールの作成，エンキューと同様の操作を行なうが，呼び出されるコードを動的にＡｃｏｄｅで与える点が異なる。コードが動的に決まるため，静的には引数個数のチェックはできない。実行時にＡａｒｇｓの引数個数とＡｃｏｄｅの引数個数を照合し，異なる場合はエラーとする。

（２）モジュール間リンケージ

コードの動的な分配を可能にするためには，以下のような機構が必要になる。

① モジュール外の述語の呼出しはいったん間接語を経由できるようにする。

② 相対形式コードをロードする時，ローダはモジュール外の参照を，述語番号等のソフト管理情報から間接語へのポインタに翻訳する。未ロードモジュールエントリについては，間接語に未ロードであることと，ロードに必要な情報を格納しておく。

③ 処理系はモジュール外呼出しの際に間接語に未ロードと記されていたら，例外を起こす。ローダは例外を解析し，必要なローディングを行なう。

（３）コードのスワップ

コードに使用するメモリ領域が大きくなると，不必要なコードは（正本が別の場所にあるとすると）消去したくなる。この際，どのコードを消去すべきかは，たとえばＬＲＵなどによって管理したいところである。完全なＬＲＵ管理は容易でないが，疑似ＬＲＵ管理ならば，比較的簡単に実現できる。

① モジュール間呼出しに用いる間接語に，１ビットのフラグ（ユースフラグ）を設ける。処理系は，モジュール外呼出しで間接語を用いたら，このフラグをオンにする。すでにオンならそのままでよい。

② メモリ不足の際に，コードの消去を行なうソフトウェアは，モジュール間リンク用の間接語を順次探索する。ユースフラグがオフの間接語は最近使われていないのだから，その指すコードは消去対象とする。ユースフラグがオンの間接語はオフにし

ておき，次の探索までにオンにならなければ次回に消去対象とする。必要なだけのコード量を消去したら，それで終了する。次回の消去対象の探索は，前回終了した間接語のところから始める。

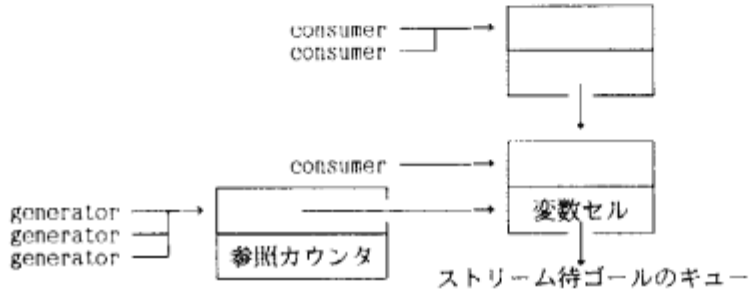
この方式は仮想メモリシステムのページテーブル管理に広く用いられている。

６．２．７　ストリーム

ＭＲＢ管理（ゴミ集めは必須ではない）を行なえば，ユニフィケーションの拡張によって，以下のような実現方式が可能である。

（１）ストリーム型オブジェクトの表現形式

複数のジェネレータがひとつのストリームにマージしており，マージ結果を複数のコンシューマが（それぞれ違う個所を）読んでいる場合。



（２）ストリームとリストとのユニフィケーション

```
ｉｆ（ストリームのＭＲＢがｏｎ）｛
    エラー；
｝ｅｌｓｅ｛
    Ｘ ＝ ストリームオブジェクトの最初の語の内容；
    リストセルをひとつ作る；
    ユニファイ相手のリストのｃａｒ部をリストセルのｃａｒに入れる；
    リストセルのｃｄｒは未定義にする；
    ストリームオブジェクトの最初の語はリストセルのｃｄｒ部を指させる；
    Ｘと作ったリストセルとをユニファイする；
｝
```

（３）ストリームと［］とのユニフィケーション

```
ｉｆ（ストリームのＭＲＢがｏｎ）｛
    エラー；
｝ｅｌｓｅ｛
    ストリームの参照カウンタを１減らす；
    ｉｆ（参照カウントが０になった）｛
        ストリームオブジェクトの最初の語を［］とユニファイする；
```

```
        }
    }
```

（4）ストリームとリスト／[] 以外のもののユニフィケーション

エラー；

（5）ストリームのマージ

マージする２本のストリームをマージ先ストリームをそのままユニファイする；

【付録】　ゴール間の優先度／資源管理等に必要な情報のまとめ

（１）ゴールレコード

親ポインタ
述語コード番地
引数個数
引数

（２）荘園レコード

親ポインタ
直接の子ゴールの数（子荘園，子里親の数を含む）
兄弟荘園／兄弟代理親リスト（双方向，里親は代理親経由）
子荘園／子代理親リスト（双方向，里親は代理親経由）
実行状態フラグ（実行中／中断／放棄，中断時はキューを持つ）
ＲＥＰＯＲＴストリーム
割付け許可資源量：メモリ量，計算量
割付け済資源量：メモリ量，計算量

（３）里親レコード

代理親ポインタ（プロセサ間ポインタ）
直接の子ゴール数（子荘園の数を含む）
子荘園／子代理親リスト（双方向，里親は代理親経由）
親キャッシュ：
　　実行状態フラグ
　　割付け済資源量：メモリ量，計算量

（４）代理親レコード

親ポインタ
里親へのポインタ（プロセサ間ポインタ）
兄弟荘園／兄弟代理親リスト（双方向，里親は代理親経由）

（５）サスペンションレコード

ゴールレコードへのポインタ
優先度

（６）制御レジスタ上のキャッシュ

親ポインタ
親キャッシュ：
　　実行状態フラグ
　　割付け済資源量：メモリ量，計算量（計算量はタイマ上か）

実行中のキューのキャッシュ
　　キューの先頭
　　優先度
負荷積算値

6．3 ブロッキングの実現方法

プロセス間，ひいてはプロセサ間の通信回数を減らすには，データの転送をある程度まとめて行なうブロッキングが重要である。ブロッキングは以下のような方法で行なうことができる。

（1）処理系のレベル

① アトミックなデータからなる構造体

文字列のようにアトミック・データだけからなる構造体を用意する。このような構造体は，いったん値が決まればそれ以降変更されたり具体化が進行したりすることはない。したがって，いくつコピーを持っても，それらのコピーをコンシステントに保つことを考えなくてよい。

KL1のレベルでは，たとえば文字列型を用意し，

```
new_string（String，Size，Elements）
```

のような組込述語で生成する。ここで，Sizeは文字列の要素数，Elementsは要素のリストである。SizeとElementsの全要素が確定したら，処理系はそのような要素からなるひとかたまりの文字列データを作成し，Stringとユニファイする。

文字列の要素の参照には，

```
sref（String，Index，Element）
```

のような組込述語を用いる。

② プロセサ間通信におけるブロック転送

プロセサ間でデータをやりとりする場合，文字列のような構造体データはまとめて送受信する。たとえば，あるプロセサ上で文字列の要素が参照されたら，別の実際に文字列データのあるプロセサから文字列全体のコピーを作って参照するプロセサに送ってしまう。

（2）PIMOSでの記述

PIMOSのレベルでは，データのやりとりを行なうストリームにフィルタとしてブロッキング／デブロッキングを行なうプロセスを付加することによって，ブロッキングをストリームの両端からは意識しなくて済むようにできる。

① プロセスの構造

ブロッキングを行なう場合のプロセスの構造は以下のようになる。

図中で，①と③はブロッキングされていないストリーム，②はブロッキングされたストリームになる。ストリーム中を流れるデータの論理的な総量は①，②，③ともに同じであるが，通信回数では②は①や③のブロッキングファクタ分の一である。ジェネレータとブロッカ，デブロッカとコンシューマをそれぞれ同じプロセサに置けば，プロセサ間の通信回数は②を流れる回数だけ，すなわちデータ総数のブロッキングファクタ分の一で済む。

② ＫＬ１での記述

ＫＬ１でのブロッカ／デブロッカの記述は概略以下のようになる（以下のコードは理解の助けとするために記述したものであり，効率面での配慮は十分でない）。

```
blocker(0, In, BlockSize, Block, EOB, Out) :-
        EOB = [],
        Out = [Block|Out1],
        new_string(Buffer, BlockSize, Block1),
        blocker(BlockSize, In, BlockSize, Block1, Buffer, Out1).
blocker(Buffered, [X|In], BlockSize, Block, EOB, Out) :-
        Buffered \= 0 |
        EOB = [X|EOB1],
        blocker(Buffered+1, In, BlockSize, Block, EOB1, Out).

deblocker([Block|In], BlockSize, Out) :-
        deblock(BlockSize, Block, 0, Out, Out1),
        deblocker(In, BlockSize, Out1).

deblock(0, Block, _, Head, Tail) :-
        Head = Tail.
deblock(Left, Block, N, Head, Tail) :-
        Left \= 0,
        sref(Block, N, Data) |
        Head = [Data|Head1],
        deblock(Left-1, Block, N+1, Head1, Tail).
```

要求駆動型の通信を実現したい場合は，ジェネレータ，ブロッカ，デブロッカ，コンシューマそれぞれの間に逆方向の要求送信用ストリームを張り，データの送信を制御するのが適当であろう。

③ 最適化

ブロッキングは上述のような文字列型の生成機能を組込述語にすればかなり高速化でき

るであろう。デブロッキングについても同様に，

```
string_elements(String, Head, Tail)
```

のような，文字列型データをリストに展開する組込述語を用意すれば高速化に有利である。