ICOT Technical Memorandum: TM-0503

TM-0503

PIMOS機能設計書

ICOT第4研究室編

May, 1988

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

目次

図目次

表目次

第1章

はじめに

1.1 本書の内容

PIMOS はマルチ PSI、PIM などの KL1 をベースとした並列推論マシンを対象とする、オペレーティングシステムである。

PIMOS の設計は昭和 61 年度に着手され、その仕様の概要は PIMOS 第一版機能仕様書[1] としてまとめられた。その後昭和 62 年度には、機能仕様書に述べられた仕様をどのように実現するかの検討がなされてきた。この機能設計書は、その検討結果をまとめたものである。

1.2 PIMOS の概要

PIMOS は計算機資源 (CPU 資源、メモリ資源、入出力機器) の管理を行い、ユーザの過ちからシステム全体を、ひいてはユーザ自身を守ることが、もっとも基本となる機能である。このためには、計算機資源をそのまま応用プログラムに見せることはできないので、計算機資源に保護のための枠組を付加してを仮想化してから提供することが必要になる。こうした管理機能が PIMOS の中核となる部分で、以下 PIMOS 本体と呼ぶ。 PIMOS 本体では基本的な資源管理機能の他に、最小限のプログラミングシステムを提供することとしている。

PIMOS 本体は KL1 で記述することとしたが、KL1 の記述レベルが高いため従来のシステムではオペレーティングシステムで実現しなければならなかった機能のかなりのものが言語レベル以下で実現されることになっている。

入出力機器について上述の仮想化を行うためには、入出力機器を仮想化し高機能化するのが有利である。PIMOS ではこうした入出力の仮想化 / 高機能化はフロントエンドプロセサで行うものとした。フロントエンド部の記述には ESP 言語をもちいることとした。

ソフトウェアの研究開発を円滑化するために、マルチ PSI、PIM などの並列処理ハードウェアの他、逐次処理システム (たとえば PSI) の上にも KL1 の並列実行を援助する機能を提供する。 また、PIMOS の開発のために必要なコンパイラ / リンカなどもこうした逐次処理システムの上に準備する。 これらを総称して PIMOS のクロスシステムと呼ぶ。

今回設計した PIMOS は、基本的に単一ユーザ、複数タスクのシステムを考えた。複数タスクを許すことはシステムの使い勝手の上から必須であるし、並列処理マシンのためのオペレーティングシステムとしては当然必要な機能である。複数ユーザを考えるとユーザ間の競合関係を調整する必要が生じる。これは逐次 / 並列の処理形態によらず困難な問題なので、当面は本格的には扱わないこととした。

1.3 本書の構成

本書は次の各章から構成される。第2章では、PIMOS設計における基本的方針について、第3章では、PIMOSが搭載されるターゲットマシンの形態とシステムの実現イメージを、第4章では、PIMOS及び応用プログラムを記述するにあたって必要となる核言語の機能拡張について述べる。次に、第5章では、PIMOSの核となるPIMOS本体について概要と各論を、第6章では、入出力を行うFEP(Front End Processor)について述べる。最後に、第7章では、核言語のソースプログラムからその抽象機械語へのコンパイラについて述べる。

1.4 執筆分担

本書の執筆者および執筆箇所は以下のとおり。

木村 康則 (ICOT 第4研究室) 7.1、7.3、7.4

越村 三幸 (ICOT 第4研究室) 5.3.5、5.4

佐藤 裕幸 (ICOT 第4研究室) 5.1、5.2.1、5.2.2、5.2.3、5.3.1

佐藤 令子 (三菱電機) 6.3.3、6.4

関田 大吾 (三菱総合研究所) 7.2、7.5、7.6

近山 隆 (ICOT 第4研究室) 第1、2、3、4章

藤瀬 哲朗 (三菱総合研究所) 5.2.4、5.3.2、5.3.3

堀 敦史 (三菱総合研究所) 6.1、6.2、6.3.1、6.3.2、

松尾 正浩 (三菱総合研究所) 5.5

和田 久美子 (沖電気) 5.3.4

第 2 章

設計の基本方針

PIMOS 設計の基本方針は以下のようなものである。

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、超論理的な機能を用いずに記述する。このためには、超論理的な操作が言語の表層上に現れずに済むように、 KL1 言語を拡張 / 改良する必要があった (第4章 「核言語機能の拡充」 参照)。

2. 本格的な OS として必要な機能を網羅

   本格的な OS として実用に耐えるだけの基本機能を網羅する。工程上、機能の一部 (たとえばファイルシステム) をフロントエンドプロセサ (PSI) の OS に依存することになるが、そうした部分も KL1 で記述できる形に仕様を設計する。

3. 集中型単一 OS

   PIMOS は並列処理マシン用の OS ではあるが、複数の OS の集合体ではない。プロセサごとに独立して動作する OS が協調して全体の機能を果たすのではなく、全体として一体であるシステムの並列動作可能な部分 (それが大部分であることを期待するわけだが) を並列に実行するものである。

4. 高水準の使い勝手

   実験的なシステムのための OS ではあるが、ハードウェアや並列アルゴリズムの実験 / 評価が円滑に行えるように、十分な使い勝手のシステムを目指す。そのためにはシステムの堅牢性も重要な要素である。

5. クロスシステムの充実

   実験的なハードウェアのためのオペレーティングシステムであるので、ハードウェアはいつでも利用可能とは限らない。したがって、ソフトウェアの研究開発を促進するために、 PIMOS では KL1 による並列プログラムの開発環境をホスト計算機 (たとえば PSI) の上に提供する。

第3章

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は、対象とする並列処理計算機として以下のような形態のものを仮定して設計した。

1. 疎結合マルチプロセサ構成であること

   メモリを共有しない複数のプロセサないしクラスタ(共有メモリにより複数プロセサが密結合したもの)を、ネットワークにより疎結合した構成のマシンを対象とする。プロセサ/クラスタ内の通信に比して、ネットワークを経由する通信が高コストであることを意識してシステムを設計する。

2. フロントエンドプロセサがあること

   対象マシンには、本体を構成するプロセサの他に、PSI程度の能力を持つフロントエンドプロセサが接続され、入出力機器の制御などを行える。

3.2 システムの実現イメージ

PIMOSには、本体部、フロントエンド部、クロスシステムが含まれる。

本体部は多数のKL1プロセスからなるひとつのOSである。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し、応用プログラムからの要請に応じて生成消滅を繰り返す。応用プログラムが動くユーザプロセスは、OSのサブプロセスのひとつとして本体上で実行される。

フロントエンド部は、本体からの入出力指令に基づき入出力機器の制御を行う。フロントエンドの機能により、本体からは入出力機器がKL1のプロセスであるかのように見えるようにする。

PIMOSおよび応用プログラムの開発に当たっては、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うことができる。クロスシステムは並列推論マシン本体とはまったく隔絶されていても動作可能であるが、LANなどによりテスト済みプログラムを本体上に(フロントエンド経由で)転送/実行できるようにする。

第4章

核言語機能の拡充

並列推論マシンに共通して用いる核言語であるKL1は、並列論理型言語GHCのサブセットであるFlat GHC (FGHC)を基本とする。しかし、FGHCの機能だけではオペレーティングシステムや応用プログラムを記述するのに不十分な点がある。こうした点を補うために、KL1はFGHCに種々の機能や概念の追加・拡張を施したものとした。このようなKL1の拡張機能について、その必要性と仕様について述べる。

4.1 実行管理機能

4.1.1 メタプログラミング機能

必要性

大規模で複雑な制御を必要とするプログラムを記述する際には、単純で平板なプログラム構造のみでは記述が容易ではない。その解決策のひとつは、モジュール構造を取り入れ、ひとつひとつのモジュールを小さく保つことである。これは、KL1に翻訳されるようなより高レベルの言語をKL1の上に構築することによって実現できる。モジュール間のより複雑な関係を表現する方法としては、直接問題解決に当たるオブジェクトレベルプログラムと、オブジェクトプログラムを監視・制御するメタレベルプログラムに分けて記述する方法がある。

オペレーティングシステムとその下で動くユーザプログラムの関係は、まさにこのメタレベルプログラムとオブジェクトレベルプログラムの関係になっている。ユーザプログラムの中にさらにメタ/オブジェクトの階層が重なることもあり、仮想機械オペレーティングシステムでは、オペレーティングシステムの中にもメタ/オブジェクトの階層があることになる。

メタプログラムの機能を実現するにはインタプリタを用いるのが簡単である。しかし、インタプリタを用いると実行効率の面で通常十倍程度以上の損失が生じる。部分評価技術を用いてこの損失を軽減はできるが、現在の技術水準では数倍程度の損失はまぬがれない。ことに、メタ/オブジェクトの階層が幾段も重なると、この損失は階層一段ごとに乗ぜられるので、最終的な効率は指数関数的に低くなっていく。

したがって、この問題はKL1に翻訳されるような言語階層を積み重ねることでは解決できず、メタレベルプログラムとオブジェクトレベルプログラムを同じ土俵(機械語)で動かすことが必要である。そのためには、メタプログラミング機能を言語自身に導入し、機械語で実行するふたつのプログラムをメタ/オブジェクトの関係に置くことが不可欠である。以下に、そのようなメタプログラムの持つべき機能を上げる。

保護機能 FGHCでは、すべてのゴールは論理積関係にある。このため、ひとつのゴールが失敗すると、全体が失敗することになる。このような言語でメタレベルプログラムを記述すると、オブジェクトレベルプログ

ラムに何らかの問題があって実行に失敗すると、メタレベルプログラムを含めたシステム全体が失敗することになってしまう。 そこで、失敗の伝播範囲を制限する機構が必要になる。

メタ制御機能　メタレベルプログラムには、たとえば、暴走したオブジェクトレベルプログラムを強制的に停止させるなど、オブジェクトレベルプログラムの実行の制御を行えるメタ制御機能が必要である。

監視機能　メタレベルプログラムはオブジェクトレベルプログラムの実行の様子を何らかの意味で監視できなくてはならない。

例外処理機能　メタレベルプログラムはオブジェクトレベルプログラムの実行中に生じた例外事象を検出し、例外事象に応じた適当な処理を行わなくてはならない。

荘園機能

概要　メタプログラミング機能を導入するためには、すべてが平板に論理積となっている FGHC に、なんらかの構造を持ち込む必要がある。 これを実現するために導入するのが荘園の機能である。

荘園は以下のようなプリミティブを用いて生成する。

execute(Goal, Control, Result, Tag)

ここで、各引数は以下のようなものである。

Goal: 荘園の中で実行すべきゴール。

Control: 荘園内の実行を制御するためのコマンドを荘園の外部から送るストリーム。

Result: 荘園内の実行の結果によるさまざまな情報を、荘園外部に伝達するためのストリーム。

Tag: 荘園がネストしている場合、それらを識別するためのタグ。

各々の引数の詳細については後述する。 なお、実際の実現上はひとつひとつの引数の機能がさらに複数の引数に分けて実現されることもありうる。 たとえば、ゴールは述語と引数に分けて別の引数にするなどである。 また、後述の優先度管理などのためには、他の引数も必要になる。

荘園の実行　上述の方法で生成した荘園内のゴール群は、荘園外とは独立した論理積を成す。 すなわち、荘園内での失敗は荘園内に閉じたものであり、荘園外のゴールを巻き添えにすることはない。 また、荘園内のゴールの実行の終了は実行結果のストリームに通知される (後述)。 概念上、荘園はインタプリタを機械語レベルで実現したものであると考えて良い。

荘園の制御　荘園内の実行を制御するためには、その荘園を生成した際の引数 Control に以下のようなメッセージを送る。

start: 実行開始。 生成直後の荘園は停止状態にある。 荘園はこの実行開始メッセージを受けて初めて実行を開始する。 同じメッセージは後述の中断メッセージによる中断の後に実行を再開する場合にも用いる。

suspend: 実行中断。 荘園の実行を中断する。 中断は即座に行われるとは限らず、任意の有限時間の遅れがあっても良い。 中断状態の荘園は上述の開始メッセージを送ることによって再開できる。

abort: 実行放棄。 荘園の実行を放棄する。 中断と同様、放棄は即座に行われるとは限らず、任意の有限時間の遅れがあっても良い。 放棄された荘園は中断した荘園と異なり、次して再開できない。

制御ストリームにはこの他に資源管理のためのメッセージも送られる (後述)。

この機能および後述の資源管理機能を用いて、メタレベルプログラムはオブジェクトレベルプログラムの実行を制御できる。

実行結果の報告 荘園の実行結果の情報は、荘園生成の際に引数として指定した実行結果報告ストリーム Result に送出される。 送出されるメッセージには以下のようなものがある。

succeeded: 荘園内の実行がすべて成功裏に終了したことを示す。

aborted: 荘園内の実行が放棄されたことを示す。

報告ストリームにはこの他に例外処理や資源管理のためのメッセージも送られる (後述)。

この機能および後述の例外処理や資源管理の機能を用いて、メタレベルプログラムはオブジェクトレベルプログラムの実行状態を監視できる。

ネストした荘園の実行管理 荘園の中でまた荘園を作ることも可能である。 意味的にはメタインタプリタでメタインタプリタをインタプリトする、ということになる。

外側の荘園に実行の中断を指令すれば、内側の荘園を実行しているインタプリタの実行が中断されるわけであるから、内側の荘園の実行も当然中断される。外側の荘園の実行を再開すれば、内側の荘園の実行も自然に再開される。

このような管理方式を取れば、メタレベルはオブジェクトレベル以下にどのようにメタ / オブジェクトの階層構造があるかを意識することなく、全体として管理することができる。

例外処理

例外の報告 荘園内の実行中に例外が生じると、以下のようなメッセージを実行結果報告ストリームに流す。

exception(Info, Goal, NewGoal)

ここで各引数の意味は以下の通り。

Info: どのような例外が生じたかについての情報。

Goal: 例外を起こしたゴールの情報。

NewGoal: 例外処理として、例外を起こしたゴールの代わりに実行すべきゴールを指定するための変数。

実際の処理系では、例外メッセージの引数個数や内容は異なっていてもよい。内容として上記の情報が伝われればよいわけである。

例外の原因別に適当なタグを割当てる。 例外の報告は、例外を起こした荘園, その親荘園、そのまた親、という先祖荘園の中で、例外原因タグにマッチするタグを持つ最も近い先祖荘園の報告ストリームに流す。 たとえば、荘園生成時に指定するタグをビットマスクとし、例外原因タグとビットごとの論理和を取って、ゼロでない荘園に報告する。 こうすることによって、特定の例外だけを取り扱い、他の例外はより外側の荘園に任せるような記述が可能になる。

例外の原因　例外の原因としては、ボディ部の組込み述語に対して誤った型の引数を与えた場合、処理できない値の引数を与えた場合 (演算あふれやゼロ除算など)、そしてゴールが失敗した場合などがある。 KL1 では unification の失敗や, ガード部の成功する候補節がないことによる失敗は、例外として扱う。 したがって、荘園の実行が失敗によって終了することはない。 失敗した際に荘園の実行を放棄したければ、制御ストリームから放棄メッセージを送れば良い。

ガード部では例外が生じることはない。 ガード部の組込み述語の型が述語の引数型にあっていない場合 (たとえば加算の引数にアトムを与えたような場合) は、例外とせず単に失敗とする。 これは、例外による中断/再開の機構を簡素化できるようにするためである。

積極的な例外の生起　ユーザが積極的に例外を起こすことができるように、以下のような組込み述語を用意する。

raise(Info, Data, Tag)

ここで、各引数は以下のようなものである。

Info: 例外に関する情報。 この引数が完全に (構造体の場合はその要素も、要素が構造体の場合はそのまた要素と、構造体全体が) 具体化されるまで例外の生起は遅延される。

Data: 任意のデータ。 この引数は具体化されていてもいなくてもよい。

Tag: 例外のタグを指定する。

例外を取り扱うプログラムは例外を出すプログラムのメタプログラムである。このメタプログラムがデッドロックすることを防ぐためには、具体化されていることが保証された例外情報が必要である。組込み述語 raise の引数 Info はこのような目的に用いるためのものである。 一方、引数 Data は、メタプログラムが中身を読まなくても良いデータを渡すのに用いる。

例外後の実行の継続　荘園内では例外を起こしたゴールの実行は中断され、NewGoal の具体化を待って、元のゴールの代わりに NewGoal を実行する。 この際、荘園には NewGoal が具体化されたらそれを実行しようとするゴールがあると考えて良い。 このゴールがあるため、NewGoal を具体化しないうちに荘園全体が先に成功裏に終了してしまうことはない。

例外処理を行う側で NewGoal を適当に与えることによって、言語仕様を拡張することができる。 たとえばあるパターンのユニフィケーションの失敗に対する処理を記述することによって、ボディ部でのユニフィケーションの仕様を拡張することもできる。 このように積極的に例外を利用する場合、荘園の実行が自動的に中断されてしまうような仕様は効率面から好ましくない。 そこで、同じ荘園内でも、例外を起こしたゴール以外のゴールは正常に実行が継続されるものとする。

4.1.2　優先度管理

必要性

GHC では任意のふたつのプログラム部分の実行順については以下のふたつのどちらかしか指示できない。

1. ふたつの計算の順序はまったく自由。
2. データの依存関係から、ふたつの計算の順序が一意に決まっている。

限られた計算資源を有効に用いて、効率良く問題を解くためには、

- ふたつの計算を同時に進めても良い。
- 両方を同時に進めるための資源 (例えばプロセサ) が不足している場合は、一方をもう一方より優先的に実行したい。

という場合が少なくない。

たとえば、AND/OR 木の探索に用いるアルファベータ探索アルゴリズムでは, なるべく速く深さ優先の探索を行い、その結果を他の枝の探索の効率化に生かすことが肝要である。 逐次的にしか実行できないプログラム言語では、厳格に深さ優先探索を指定することになる。 同様の厳格な指定は GHC でももちろん可能ではある。 しかし、完全に深さ優先に探索を行うことを指定してしまうと、たとえアイドル状態のプロセサがあってもそれは用いず、深さ方向の探索のみを行うことになる。 これでは並列処理の可能性を放棄していることになる。

これを解決するためには、データの依存関係による厳格な順序以外に、柔軟性のある優先度の概念が必要である。 優先度が異なる計算が複数ある場合、プログラムに指定した依存関係からはどちらも実行可能状態で、しかし実行に必要な計算資源 (例えばプロセサ) は片方の分しかない場合、優先度の高い方を先に実行する。 データの依存関係による順序との差は、計算資源が豊富にある場合に両方同時に実行することを許す点にある。

優先度の概念

優先度は常に完全にに守らなければならないものではなく、処理系が計算順序 (別の言葉で言えば、計算資源の配分法) を決める際の目安に過ぎない。 したがって、システム中により優先度の高いゴールが実行可能状態にあるにもかかわらず、優先度の低いゴールが実行されてしまうことがあってもかまわない。優先度が完全に守られないと正しく動作しないようなプログラムは、誤ったプログラムである。 優先度指定は、あくまで実行効率向上のためのプラグマである。 処理系がどの程度優先度を守るかは、処理系の良否の問題であって, 正誤の問題ではない。

優先度を単なる目安としたのは、並列処理系で完全に優先度を遵守するためには、ひとつひとつのリダクションを開始するごとに、全システムに渡ってより優先度の高いゴールは存在しないかを調べる必要があり、処理の局所性を著しく損なうことになるからである。

指定方式

優先度の指定は、荘園単位におおまかに指定する方法と、各ゴールごとに細かく指定する方法の両者を用意した。 メタレベルプログラム的な意味でおおまかに制御する場合 (たとえばオペレーティングシステムがユーザプログラムの優先度を管理する場合) には荘園単位の指定、オブジェクトレベルの知識を元に細かい制御をしたい場合 (たとえばアルファベータ探索プログラムで、深さ優先探索をしたい場合) にはゴール単位の細かい制御ができるようにするためである。

ゴール単位の指定はそのゴールそのものの実行の優先度を指定する。 荘園に対する指定は、荘園内のゴールの優先度の最大 / 最小値を指定することによって行う。 荘園生成のための述語にはこの優先度範囲指定のための引数を追加する。

指定は荘園単位の指定・ゴール単位の指定のいずれも、優先度の絶対値ではなく、実行中の荘園の優先度範囲や指定を行う計算自身の優先度と比べての相対的な値として与える。 具体的には以下のいずれかの方法を用いる。

1. 指定を行う計算の属している荘園内での相対値。 属している荘園の優先度範囲内のどのあたりで実行するかを、割合で指定する。

2. 指定を行う計算自身の優先度と、属している荘園の最大 (または最小) 優先度との間のどのあたりで実行するかを、割合で指定する。

絶対値ではなく相対値を用いることには、以下のような利点がある。

- 優先度の絶対値の範囲は処理系ごとに異なることが考えられるが、相対記述ならプログラムには処理系独立な優先度指定記述ができる。
- 局所的に相対的な優先度指定をしてあるプログラムを、大局的に荘園全体の優先度範囲を変えて呼び出すことができる。

暴走の停止

優先度の概念がなかったり、オブジェクトレベルの優先度がメタレベルの優先度以上になったりすると、オブジェクトレベルプログラムの暴走をメタプログラムから制止できなくなる。 たとえオブジェクトレベルの実行を停止するためのメタプログラムのゴールを実行しようとしても、オブジェクトレベルプログラムの暴走のためにメタレベルプログラムはたったひとつのリダクションも実行されないことがあり得る。

この問題を解決する方法のひとつに、スケジューリングに公平さを導入する方法がある。公平なスケジューリングでは、実行可能状態にあるどのプログラム部分も有限時間内には少なくとも少しは実行されることが保証される。 これを保証する方式としては、幅優先スケジューリングや限度つき深さ優先スケジューリングなどがある。

一方、優先度の機構を用いてもこの問題は解決できる。 オブジェクトレベルプログラムが決してメタレベルプログラムよりも高い優先度を持たないように、荘園を生成する際の優先度範囲指定を行えばよい。 逆に、優先度機構を持っていれば、プログラムの暴走を防ぐために公平なスケジューリングを行う必要はない。

スケジューリングが公平でなくてよければ、プログラムの局所性を最大限に生かすような、あるいは実装が最も容易なスケジューリング方式を自由に選択できる。 極端な例では、公平なスケジューリング方式で最も簡単である幅優先スケジューリングより、公平さを保証しない最も簡単なスケジューリング方式である深さ優先スケジューリングの方が、通常はるかに時間的な局所性を高くワーキングセットを小さくできるし、実装の簡素化のためにもスケジューリングキューをスタックとして管理できる利点が大きい。

暴走の停止が保証するためには処理系がある程度優先度を守る保証が必要である。 それは、低優先度の計算がいくらあろうとも、有限時間内に終了するような最高優先度の計算は有限時間内に終了することである。

4.1.3 クローズ間の優先度

GHC では原則として複数の候補節のどれを用いてもリデュース可能な場合、どの節を用いるかは言語仕様としては定めないことになっている。 すなわち, 処理系が処理の上で都合が良い (例えば、速度面で有利) と考える節を用いて良いわけである。 この仕様は、処理系による最適化の自由度を上げる意味で有利である。

しかしこの機構だけでは、A B のどちらの方法を用いても良いが、もし A の準備ができていれば (方法 A を用いるのに必要なデータが揃っていれば) A の方を使いたい、準備がまだなら B の方を用いる、といったことは記述できない。 これでは、複雑な解法戦略を記述し難い。

そこで、節の選択に関しての優先関係を指定するために、優先度の高い節群と低い節群の間に、**alternatively** というキーワードを挿入することとする。

例: 節の優先関係

```
p(stop, In, Out) :- true | Out=In.
```

```
alternatively.
    p(Stop, In, Out) :- true |
        q(In, In1), p(Stop, In1, Out).
```

上記のプログラムの p は、第一引数が stop というアトムに具体化するまで第二引数に述語 q を適用し続けるような述語である。第一引数が stop になっても第二節でもリデュースできるのだが、優先度関係から第一節が選ばれ、再起呼出しによるループは停止する。

この alternatively による節間の優先度指定はゴール間の優先度と同様、あくまで効率向上のための情報を処理系に知らせるためのプラグマである。上記プログラムで言えば、第一引数が stop に具体化された後に第二節が選ばれることが決してあってはならない、というものではない。ゴール間の優先度と同様、処理系がどの程度優先度を守るかは、処理系の良否の問題であって、正誤の問題ではない。完全に守ろうとすると、並列実行の局所性を著しく損なうことになる。

4.2 資源管理機能

4.2.1 必要性

メタレベルプログラムがオブジェクトレベルプログラムを管理するには、オブジェクトレベルプログラムがどのようなふるまいをしているかを、少なくともマクロには把握していなければならない。前節で述べた実行管理機能を用いれば、オブジェクトプログラムの実行の終了、異常事態の生起といった、計算の進行状況の質的な把握は可能である。しかし、それだけでは量的な側面, たとえばどのぐらい計算が進行しているのか、といったことはわからない。

計算がどのぐらい進んでいるのか、というようなメジャーは解いている問題に依存するので、一般的なメタレベルの機構として導入するのは不適当であり, 個別の問題ごとに適当な機構を設定するしかない。しかし、どれぐらいの時間計算しているのか、どのぐらいのメモリを費やしているのか、といった計算機構そのものに関わるメジャーは、同じ計算機構を用いているかぎり共通のものであり、メタレベルでオブジェクトレベルの計算の戦略を立てるためのデータとして有効である。

一方、オペレーティングシステムのようなプログラムを考えると、たとえばユーザプログラムがメモリをすべて消費してしまうと、オペレーティングシステムが動くためのメモリが残っていない、ということも生じ得る。これを防ぐには、メタレベルにはオブジェクトレベルでの消費計算資源を制限する能力が必要である。

そこで、こうしたデータを大きなオーバヘッドなく得るために、言語のメタレベル機能の一環として資源管理機能を導入することとした。

4.2.2 方針

管理単位

どのような計算ごとに資源消費を管理するかという単位は、実行管理と同じく荘園単位とした。これは資源管理の結果として実行を制御したいことが多く, 資源管理単位を実行管理単位と一致させれば制御が容易であること、そして処理系実現上も資源消費単位を別に設けると余分なオーバヘッドになること、が理由である。

資源消費の報告の仕方として、単純にはオブジェクトレベルからメタレベルに資源の消費状況を逐一報告することが考えられる。しかし、この方法ではメタレベルが常にオブジェクトレベルを監視していなくてはならず、そのオーバヘッドが大変大きくなってしまう。そこで、あらかじめ適当な資源消費許容量をメタレベルから設定し、それが尽きるまではオブジェクトレベルはメタレベルに特に報告することなく動作できるものとした。許容量が尽きた段階でそれを知らされたメタレベルでは,

- 許容量を追加して計算を続行させる。
- その計算は打ち切る。
- その計算は中断し、他の方法をためすが、将来継続して見ることもできるようにしておく。

などの選択が可能である。 許容量を与えて行く単位を小さくすれば精度良く資源管理できるが、それだけオーバヘッドも増えることになる。

管理対象

管理の対象としては以下のようなものが考えられる。

計算時間: 当該荘園の計算のためにどのぐらい時間をかけてよいか。

メモリ消費量: 当該荘園の計算のためにどのぐらいメモリを消費してよいか。

計算時間については、たとえばリダクション数で代用することも考えられる。

メモリについては、ごみ集めによって再利用が可能であるから、本来は一時にごみでないメモリをどれぐらい使ってよいかを指定したいところである。 しかし、いったんメモリ上に作ったデータ構造をひとつの荘園から別の荘園に渡した場合、それを記録するのはオーバヘッドが大き過ぎる。 また、ごみ集めの際に集めたごみがどの荘園に属しているかを知ることができるようにする機構も大きなオーバヘッドを伴う。 メモリ割り付けの時点で把握することは容易なので、メモリ消費量はメモリ割り付け量で捉えるのが適当であろう。

計算時間とメモリ消費の両者を独立に管理するには、それぞれについてオーバヘッドがかかる。 そこで、計算時間とメモリ消費量を適当に按分した資源消費ミックスを設定し、これだけを管理することも考えられる。

プロセサが複数ある場合、

- 総消費量の許容限度を指定。
- プロセサごとに許容消費限度を指定。
- ひとつのプロセサでの許容消費限度の最大値を指定

が考えられる。 しかし、KL1 のプログラムは原則としてプロセサの個数など, ハードウェア依存のことを意識せずに書くものとしたいので、指定は総消費量で行うものとする。

資源管理のネスティング

荘園がネストしている場合の資源管理の考え方は、実行管理と同様、すなわち荘園はインタプリタを機械語で実現した機構である、という考え方に基づく。荘園の実行に必要な計算資源量はその荘園がインタプリトしながら数えている、と考える。 荘園を実現するインタプリタは効率が良く、インタプリトするのに直接実行したのと同じだけの資源しか消費しないものとする。

荘園が二重になっている場合、内側の荘園の実行を行っているインタプリタは、外側の荘園によってまたインタプリトされていることになる。 内側の荘園で資源を消費すると、その消費をインタプリトするために外側の荘園でも同じだけの資源が消費される。 外側の荘園の資源が尽きた場合は、内側の荘園を実行しているインタプリタも停止せざるを得ないのだから、内側の荘園も停止する。

このような管理方式を取れば、メタレベルはオブジェクトレベル以下にどのようにメタ / オブジェクトの階層構造があるかを意識することなく、全体として管理することができる。

4.2.3 言語仕様

資源管理機能は KL1 には荘園機能の機能追加という形で導入する。

荘園の資源消費許容量は、荘園生成時にはゼロとする。 すなわち、生成したばかりの荘園は何も計算できない。 以下のようなコマンドを荘園の制御ストリームから送ることによって、荘園の資源消費許容量を増やすことができる。

more-resource(Amount): Amount だけの量を資源消費許容量に追加するコマンド。

荘園の実行中に許容量だけの資源を消費し尽くした場合は、以下のようなメッセージを報告ストリームに流し、実行を中断する。

resource-limit: 資源が尽きたことを報告するメッセージ。

上述の資源消費許容量追加コマンドによって許容量が追加されれば、実行は再開される。

なお、複数の種類の資源を区別する場合は、それぞれの資源ごとに異なるコマンド / メッセージが必要である。

第5章

PIMOS本体

5.1 概要

PIMOS本体は、並列推論マシン(PIM, マルチPSI)の本体上で動作し、PIMOSの核となる部分である。このPIMOS本体は、ユーザが使用するさまざまな資源を管理する機能、それらの資源をアクセスするための機能、プログラムの実行中に発生した異常(例外事象)を処理する機能、ユーザのプログラムを起動し、実行の制御を行なうためのユーティリティ(シェル)などの機能を提供する。以下にこれらの機能について、その概略を示す。

1. 資源管理
   PIMOSは、ユーザが消費する計算時間、メモリ、入出力装置等の資源を管理し、適当な配分、暴走などによる過度な消費を防止する。PIMOSでは、これらの資源を管理する単位を「タスク」と呼んでいる。このタスクは任意にネストすることができ、タスク内で生成された子タスクも資源の1つとして管理される。これらの資源を管理するためのテーブルを「資源木」と呼んでいる。資源木は、タスク単位で入出力装置や子タスク等のさまざまな資源を管理しており、ユーザからの要求によって、それらの資源の制御を行なう。

2. 資源へのアクセス機能
   PIMOSが管理している資源に対して、ユーザがアクセスする場合は、ユーザとOSとが共有変数を用いて通信を行うことになる。PIMOSが動作するような並列環境では、ユーザ-OS間の通信で、ユーザがなんらかの誤りを犯すとシステム全体がダウンしてしまうことがある。PIMOSでは、ユーザーOS間を流れる通信メッセージを監視するプロセスをユーザ・タスク内に置くことで、ユーザの誤りからシステムを保護している。PIMOSが管理する入出力資源には、ウインドウ、ファイル、タイマなどがある。これらに対するユーザの操作要求は、最終的にFEP(フロントエンド・プロセッサ)に通知されるが、PIMOSでは、これらの入出力装置をユーザが効率よく使えるようなバッファリング、パーザ、アンパーザなどの機能を提供している。

3. 例外処理
   PIMOSでは、プログラムの実行中に発生した例外事象に対して適切な処置を行なう。この例外には、フェイル、組込述語エラー、未定義述語呼出し、資源の超過使用、デッドロックなどのPIMOSの記述言語であるKL1によって定義されいるものと、ソフトウェアにより積極的に発生させるソフトウェア定義例外がある。タスク内で例外事象が発生すると、そのタスクの祖先に通知される。従って、ソフトウェア例外を利用することにより、OSとの通信にも使用できる。また、ユーザは、タスク単位で例外処理を追加/変更することができる。

4. シェル

シェルは、ユーザが指定したゴールを起動し、実行/制御するためのユーティリティである。シェルが管理するひとかたまりの仕事の単位を「ジョブ」と呼ぶ。ジョブは、1つ以上のタスクから構成され、それぞれのタスク間は、シェルが提供するタスク間通信機能(パイプ)によって通信することができる。ユーザは、ジョブ単位で、実行の開始、中断、再開、放棄(強制終了)、状態の問い合わせなどが行なる。

5.2 資源管理

5.2.1 概要

PIMOSは、ユーザが使用する計算時間、メモリ、入出力装置等の資源を管理し、適当な配分、暴走などによる過度な消費の防止を行なう。PIMOSが管理する資源には、計算時間やメモリ等のプログラムの実行に必要不可欠なものと、入出力装置等のようにプログラムを実行する上で手足(道具)となるものに分けられる。前者は、PIMOSの記述言語であるKL1(核言語第1版)によって定義された資源(言語定義資源)であり、KL1の荘園機能を用いて管理される。一方後者は、KL1ではなくPIMOSによって定義されている資源(OS資源)であり、これらは資源木によって管理される。

これらの資源を管理する単位を「タスク」と呼んでいる。ユーザは、タスクを生成してその上でプログラムを実行することにより、プログラム実行の開始、中断、再開、放棄等を行なえ、言語定義資源の使用許容量を設けることによりユーザ・プログラムの暴走からシステム全体を保護することができる。また、入出力装置等のOS資源に関してもタスク単位で管理され、タスク内で使用されているOS資源の問い合わせ等が行なえ、タスクの実行が放棄された時には、その中で使用されていた未解放のOS資源がPIMOSにより強制的に解放される。このタスクは、任意にネストすることができ、タスク内で生成された子タスクも、OS資源の1つとして管理される。

これらのOS資源を管理しているものを「資源木」と呼んでいる。資源木は、ユーザ・タスクとは別の場所(PIMOSのタスク内)に存在し、タスクの階層構造にそって木状になっており、1つの階層の中では、タスク内で使用されているOS資源が輪状に繋がれて管理されている。ユーザがOS資源に対して制御要求を送ると、PIMOSは資源木内の木及び輪をたどることにより、指定されたOS資源を制御する。

PIMOSは、ユーザが要求した入出力装置の生成等に対して資源木を用いて処理を行なう。このようなユーザの要求に対する処理は、PIMOS側でかなりの部分が行なわれ、ユーザ・タスク内の言語定義資源はほとんど消費されない。しかし、このような処理によって消費された資源は、その要求を発行したユーザ・タスクが消費したと見なすべきである。そこで、ユーザがPIMOSに対して要求を行なうと、そのタスク内で言語定義資源を強制的に消費させる機構を設けて、できるだけ正確かつ公平な資源管理を行なうようになっている。

5.2.2 タスク

タスクとは、ひとかたまりの仕事の単位であり、また、資源管理の単位である。タスクで管理される資源には、リダクション数等のKL1で定義されている言語定義資源の他に入出力装置等のOSで定義されているOS資源がある。ユーザは、これらの資源に対して、タスク単位で制御することができる。

1. タスクの階層構造

   PIMOSのタスクでは、その子供として任意個の子タスクを生成することができ、これらの子孫タスク群は階層的に管理されている。この子タスクとそれを生成した親タスクとの間には、以下のような関係がある。

(a) 子タスクが消費する言語定義資源は、親から分け与えられる。つまり、子孫タスクが消費した言語定義資源は、それらの親タスクが消費したと見なされる。従って、子孫タスクが消費できる資源量は、親タスクの許容資源量を超えることはできない。

(b) 親タスクの実行が放棄されると、その中で生成された総ての子孫タスクの実行も放棄されてしまう。また、親タスク内のゴールの実行が総て終了しても、子孫タスクが放棄されない限り、親タスクの実行が終了したとは見なされない。

タスクは、KL1の荘園機能を用いて実現されており、これらの関係はその荘園機能の階層関係と同じになっている。また、PIMOSではタスク自身もOS資源として扱っているので、子タスクはその親タスクが使用している一つの資源として管理される。

2. タスクに対する操作
ユーザはタスクに対して以下のような実行の制御及び資源の変更 / 問い合わせが行なえる。

(a) プログラム実行の開始、中断、再開、放棄

(b) 実行優先度の範囲 (最高値 / 最小値) の指定

(c) 言語定義資源の許容量 (リダクション数等) の設定 / 変更 / 問い合わせ

(d) OS資源 (入出力装置等) の割当て / 状態問い合わせ / 解放

(e) 例外処理の追加 / 変更

ユーザがタスクを生成すると、そのタスクを制御するためのストリーム (タスク制御ストリーム) とタスクの実行状態や例外事象を知らせるタスク報告ストリームが返される。この制御ストリームに対して要求 (メッセージ) を送ることにより、タスクを制御することができ、報告ストリームから知らされる情報により、タスクの実行状態を監視したり例外処理を行なうことができる。

また、PIMOSが管理する総ての資源 (OS資源) には、ユーザが認識できるID(資源ID、詳しくは後述) が付いており、孫タスクのようなタスク制御ストリームを直接持っていない場合にも、このIDを指定することによって制御することができる。ただし、親タスク等の祖先タスクに対する操作は、そのタスクが持っている資格によって制限が設けられる場合がある。

3. 資源の強制解放
ユーザ・プログラムの暴走等により、タスクの実行が強制的に放棄された時には、そのタスク内で使用していた資源は、PIMOSによって解放される。例えば、ファイルをクローズしたり、ウインドウをスクリーンから消したり、子孫タスクの実行を放棄したりする。このような資源の強制解放は、タスクの実行が放棄されることによりタスク単位で行なわれるが、その具体的な実現方式については後述する。

5.2.3 資源木の管理

入出力装置や子タスク等のOS資源を管理するのが資源木である。資源木は、タスクの階層構造にそって木状になっており、また、1つの階層の中では各資源が輪状に繋がれた構造をしている。

1. 資源木で管理される資源
ユーザが入出力装置の確保や実際の入出力は、PIMOSとの間に通信路 (ストリーム) を張り、そこに対して要求 (メッセージ) を送ることにより行なわれる。資源木では、このようなユーザとPIMOSとの間に張られるストリームを資源として扱っており、それらは以下のように分類できる。

(a) ジェネラル・リクエスト・ストリーム
ウインドウ・システムやファイル・システム等の 各サブ・システムへのストリーム (サブシステム・ストリーム) を 獲得するための資源である。 これは、ユーザが PIMOS との通信を行う時の根源となる資源であり、 プログラム実行中の任意の時点で例外事象を上げることにより 獲得することができる。

(b) サブシステム・ストリーム
ウインドウやファイル等の個々の装置 (デバイス) を獲得するための 資源である。 これは、ジェネラル・リクエスト・ストリームに要求を送ることにより獲 得することができる。

(c) デバイス・ストリーム
ウインドウやファイル等の個々の入出力装置に対して、 実際の入出力要求を送るための資源である。 また、子タスクの実行開始、中断、放棄や許容資源量の問い合わせ、変更等 子タスクの資源に対する要求を送るためのストリームもこれに含まれる。

これらの資源に対する (流せる) 具体的な要求については、後述する。

2. 資源木の構造

タスクの階層構造及び資源木の構造を例を図 5.1に示す。この例では、4つのタスクが存在しており、それぞれのタスクは以下のような OS 資源を使用している。

(a) 親タスク：1つの入出力資源と「子タスク 1」と「子タスク 2」を持っている。

(b) 子タスク 1：2つの入出力資源を持っている。

(c) 子タスク 2：3つの入出力資源と「孫タスク」を持っている。

(d) 孫タスク：3つの入出力資源を持っている。

この図の右側の部分を資源木と呼び、タスクの階層にそって木構造になっている。また、各タスク内の資源は、それぞれが輪構造で結ばれている。上記のジェネラル・ストリーム、サブシステム・ストリーム、デバイス・ストリームの3種類の資源には、その獲得方法によって階層関係が存在するが、資源木上では特に階層的に管理していない。この理由は、例えばサブシステム・ストリームが閉じられてもそこから獲得されたデバイス・ストリームは有効であり、階層的に結合するとその子供群が宙に浮いてしまい、資源木から切り放されてしまうからである。

以下に、資源木を構成する各要素及びタスク管理を行なうための構成要素を示す。

(a) タスク H(タスク・ハンドラ)
タスク制御ストリーム (図では下向き1点鎖線) からの要求を受付け、 タスク・モニタに要求を送る。 タスクは、KL1 の荘園機能を用いて実現されており、 その実行状態報告ストリーム (図では上向き2点鎖線) によってタスクの実行 状態を管理している。その子タスクの報告ストリームやタスク・モニタからの 報告を受けて、タスク報告ストリーム (図では上向き2点鎖線) に報告する。このようにユーザが子タスクを制御する時の直接の窓口になっている。 これは親タスク (ユーザ・タスク) 上で実行されるが、 プログラムとしてはシステム (PIMOS) が提供するものである。

(b) Mess-F(メッセージ・フィルタ)
ユーザの誤りからシステム全体を保護するためのものである。 ユーザとシステムとが通信を行なう場合には、 必ずこの種の保護フィルタが付けられる。 従って、タスク・ハンドラはこの保護フィル

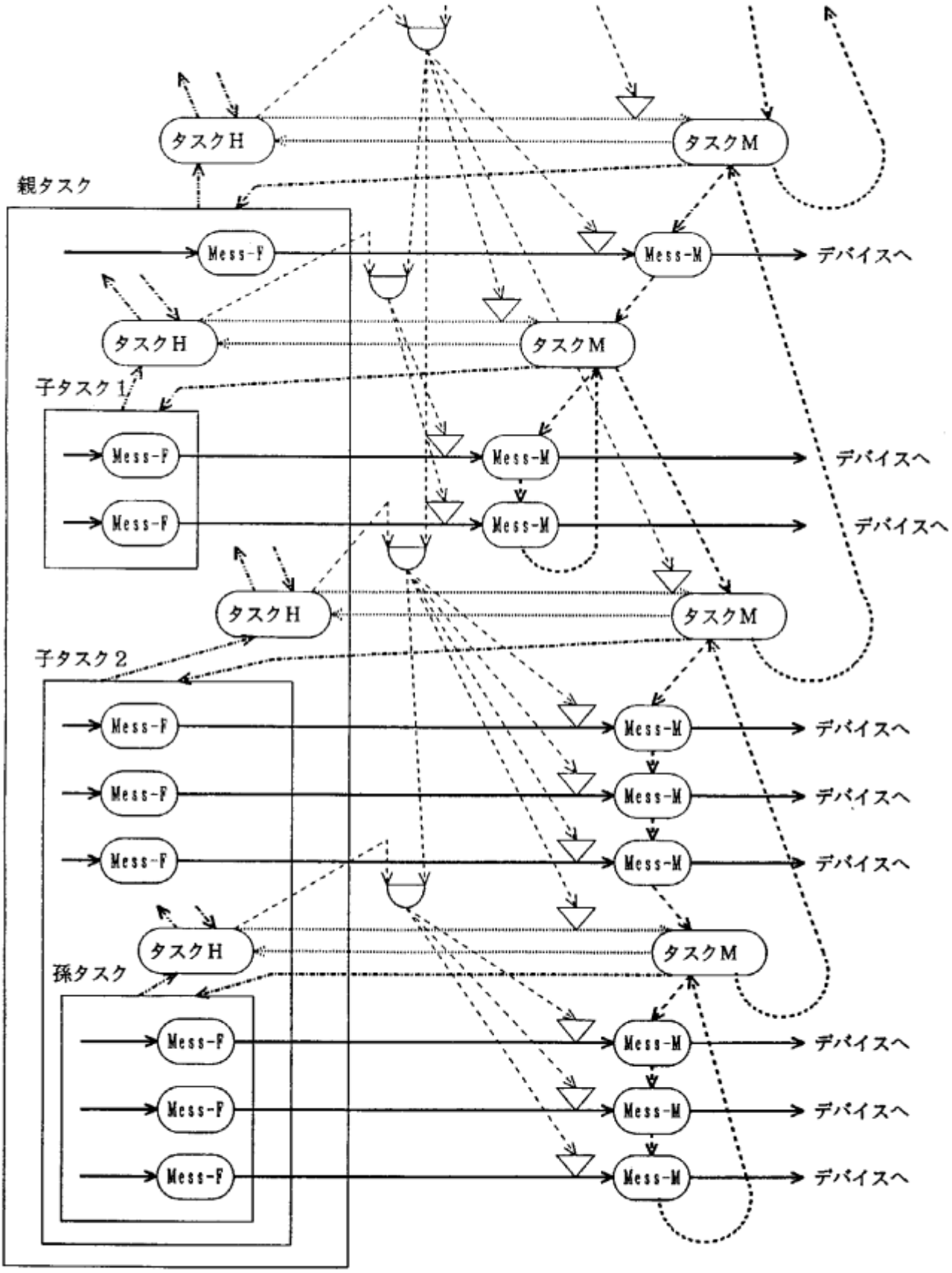

図5. 1:タスクの階層構造と資源木の構造

タとしての働きもする。また、これもユーザ・タスク上で実行されるが、プログラムとしてはシステムが提供するものである。なお、保護フィルタの詳細については、5.3「OSとの通信方式」の節で記述する。

(c) タスク M(タスク・モニタ)
タスク内の資源を管理しているものであり、ユーザがそれら資源を操作する時に、タスク・ハンドラを経由してここに要求が渡ってくる。また、親のタスク・モニタ(タスク・モニタから見れば兄弟関係にある)からもここで管理している資源に対する操作要求が送られて来ることもある。ここでは、タスクを実現している荘園の実行制御ストリーム(図では下向き1点鎖線)へ要求を送ることにより、タスクの実行を制御している。本来、タスク・ハンドラがそれを用いてタスクを制御すべきである。ところが、タスクの外側にユーザ荘園が作られておりそれが停止状態の場合には、タスク・ハンドラが動作しなくなるため、タスク・モニタが荘園の実行制御ストリームを管理しているのである。

(d) Mess-M(メッセージ・モニタ)
タスク以外のOS資源を管理しているものである。通常の入出力要求等は、ユーザから(メッセージ・フィルタを通して)直接送られてくる。一方、資源の状態の問い合わせ等の資源にとってメタな操作に関しては、タスク・モニタから送られて来る。

(e) ▽(バルブ)
タスクの実行が放棄された時に、そのタスク内で使用されていた資源を強制的に解放するためのものである。ユーザ・タスクから資源木へ向かうストリームには総てこれが付いている。PIMOSでは、資源の解放を「ストリームを閉じる」と言う統一した方法によって行なっている。バルブは、タスクの実行状態を監視しているタスク・ハンドラからの終了メッセージを受け取ると、ストリームを閉じる働きをする。従って、タスクの実行が放棄された時には、そのタスク内で使用していた資源は総て解放されることになる。また、親タスクの実行が放棄されるとその子孫タスクの実行も放棄されるので、親からの終了メッセージも監視している。これは、「祖先タスクの実行が放棄されたから自分も死ぬ」といった場合には、自分を監視しているタスク・ハンドラには、「実行が放棄された」と言う報告が行なわれる保証がないからである。

PIMOSが管理する総てのOS資源には、ID(資源ID)が付けられている。この資源IDは、タスクの階層によって深くなって行き、原則的に自分のタスクを基にした相対番号となっている。例えば、[5]であれば、タスク内の5番目の資源であり、[3.4.2]であれば、タスク内の3番目の資源(タスク)内の4番目の資源(タスク)内の2番目の資源を表わしている。

タスク・モニタやメッセージ・モニタ(これらを総称してモニタと呼ぶ)は、それぞれ自分が管理している資源のIDを知っており、指定されたIDが自分のIDと一致するかを調べて、一致すればその要求に対する処理を行ない、一致しなければ隣の兄弟に要求を再送する。また、タスク・モニタの場合は、指定されたIDの先頭が自分のIDと一致すれば、指定されたIDの残りを付けて子供にその要求を再送する。また、自分が再送した要求がそのまま返ってきた場合は、指定されたIDに相当する資源が存在しなかったことになり、エラーとして扱われる。

資源が解放された時には、モニタが自分の左の兄弟からのストリームと右の兄弟へのストリームをユニファイすることにより、消滅する。

3. 資源木の機能
資源木で管理している資源を操作するには、タスク制御ストリームに要求を送ることで行なえる。この時、資源IDを指定すればその資源に対する操作となり、指定しなければそのタスク自身に対する操作と

なる。これらの方法では、自分の子孫資源に対する操作しか行なえない。例えば、「自殺行為（自分で自分の実行を放棄する）」や「システム全体の状態を問い合わせる」等の操作は、この方法では行なえない。これらの操作は、祖先タスクでしか行なえないので、祖先タスクに例外事象を上げることによって要求を送る。ただし、この方法は異常パスであって、タスクが持つ資格によっては不可能な操作も存在する。例えば、「兄弟タスクの実行を放棄する」ことはできない場合がある。

以下にタスクに対する操作、タスク内の資源に対する操作を示す。

(a) 子孫タスクに対する操作

i. 生成

ジェネラル・ストリームからタスク管理システム・ストリームを獲得し、そこへタスク生成要求を送ることにより、タスクを生成することができる。タスクが生成されると、タスクを制御するためのストリーム（タスク制御ストリーム）と、タスクの実行結果（success または aborted）や子孫タスク内で発生した例外事象が報告されるストリーム（タスク報告ストリーム）が返される。ユーザは、これらのストリームを使ってタスクを制御したり、例外処理を行なったりすることができる。

ii. 実行の中断 / 再開

タスク制御ストリームに中断要求を送ることにより、タスクの実行を中断させることができる。この時、中断原因と中断解除用の変数を付ける。この変数を具体化することにより中断は解除される。タスク・モニタでは、中断原因毎にこの変数を管理しおり、総ての原因が解除されたらタスクの実行を再開する。また、資源 ID を指定することにより、子孫タスクの実行を制御することもできる。

iii. 実行の放棄

タスク制御ストリームを閉じることにより、タスクの実行を放棄することができる。タスクの実行が放棄されると、その中で使用されていた資源は、総て解放される。同様にタスク ID を指定することにより、子孫タスクの実行を放棄することができる。

(b) タスク内の資源の変更 / 状態問い合わせ

i. タスク情報の問い合わせ

タスク制御ストリームに情報問い合わせ要求を送ることにより、以下のようなタスクの情報を得ることができる。

- 資源 ID
- 実行状態（実行中または中断状態）
- 言語定義資源の許容量
- その時点までの言語定義資源の使用量

この要求には、指定したタスクだけの情報を取り出すものと、総ての子孫タスクの情報を取り出すものがある。

ii. OS 資源の問い合わせ

タスク制御ストリームに OS 資源問い合わせ要求を送ることにより、タスク内で使用されている OS 資源（入出力装置や子タスク等）に関して、以下のような情報を得ることができる。

- 資源 ID
- 資源 REMARK(資源に付いた名前のようなもの)

同様に、この要求には、指定したタスク内だけの OS 資源の情報を取り出すものと、総ての子孫タスク内の OS 資源の情報を取り出すものがある。

iii. 言語定義資源の追加

タスク制御ストリームに言語定義資源の追加要求を送ることにより、 そのタスクの言語定義資源の許容量を大きくすることができる。 ただし、これは自分の資源を分け与えることになるので、 自分の許容量を超えて指定することはできない。 同様にタスク ID を指定することにより、 子孫タスクの言語定義知資源を追加することができる。

(c) OS 資源に対する操作

i. 生成

ウインドウやファイル等の入出力資源を生成するには、 それらに対応するサブシステム・ストリームに対して生成要求を送り、 デバイス・ストリームを獲得する。 以後は、このデバイス・ストリームに対して入出力要求を送ることにより、 入出力装置を制御できる。

ii. 解放

入出力資源の解放は、通常、デバイス・ストリームを閉じることにより 行なわれる。また、プログラムの暴走等が原因で入出力資源を 強制的に解放したい場合は、資源 ID を指定してタスク制御ストリームに 資源解放要求を送ることによっても、資源の強制解放は行なえる。 また、タスクの実行が放棄された時には、 解放されていない OS 資源は、バルブ機構によって強制的に解放される。

5.2.4 資源消費量の管理

資源消費量管理の必要性

プログラムの実行には、計算時間、メモリ等の言語定義資源やファイル、ウインドウ等の OS 資源を必要とする。 PIMOS が実行過程で必要となるこうした資源の管理および利用者プログラムへの提供を行うにあたり、資源の適正配分や暴走等による過度な消費の防止を行う必要がある。ここで注意すべき点は、 PIMOS と利用者プログラムは同じ KL1 で動作することである。 OS 資源を消費することがそのサービスを行なう PIMOS 側の言語定義資源をときには大きく消費してしまうのである。一般に PIMOS が利用者プログラム実行のために提供する全ての OS および言語定義資源に対して、その消費量を管理する必要がある。

資源消費量の管理単位

各資源に対してその消費量を別々に管理することは、管理の手間や拡張性等からみても困難である。そこで PIMOS では、言語定義資源や OS 資源の共通管理単位として基準資源量単位を設け、資源消費量を一意的に管理することにする。

PIMOS の提供する各資源rに対して、その資源量 (場合によっては複数資源の資源量) から近似的に基準資源量単位へ換算するための関数ϕ_rを定義する。そしてプログラムが使用している各資源の消費量θ_rをそれぞれ基準資源量単位へ換算し$\phi_r(\theta_r)$を求める。そして全ての資源に対する換算値の総和$\Sigma_r(\phi_r(\theta_r))$ を求め、それをプログラムの資源消費総量として管理することにする。

基準資源量単位の一例として、プログラム中のゴール・リダクション数が挙げられる。

資源消費量とタスク階層

PIMOS では、タスクを資源管理単位のひとつとして位置付けているので、資源消費量もタスク単位で管理することにする。

利用者が生成したタスクの資源許容量を利用者自身が設定することも可能である。しかしながら、利用者に対して無制限に資源許容量を設定することを許すのは、他タスクや PIMOS への影響を考えると望ましくな

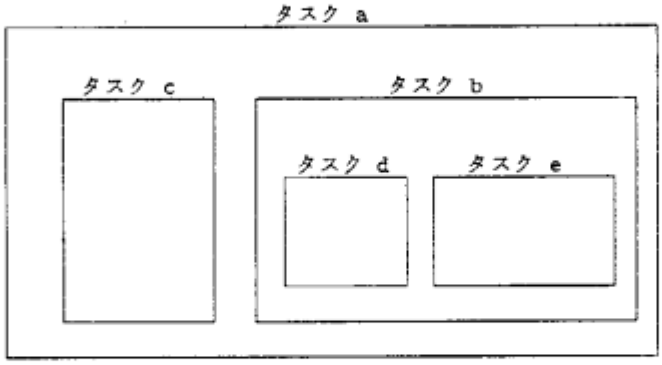

図 5.2: タスク階層例

い。 PIMOS では、タスクの階層性を利用した次の方針によりこの問題を解決する。

- PIMOS では、タスク全体が消費する資源量を一定量とする。
- タスクは、自分が生成する子タスクが消費する資源量は親タスクの 資源消費量でもあることとする。 親タスクは子タスクに対して、 自分に与えられた資源許容量の一部を譲渡することとする。

例えば、タスクa,b,c,d,e間に図 5.2 で示すタスク階層が存在したとする。すると資源消費量R_a,R_b,R_c,R_d,R_e間には次の関係式が成り立つ。

$$R_a \geq R_b + R_c, R_b \geq R_d + R_e$$

PIMOS はタスク階層の頂点となるタスクをまず生成し、タスク全体が消費可能な資源量をこのタスクに設定する。今後発生するタスクはすべてこのタスク中で生成されることとする。タスク階層の頂点であるタスクは他のすべてのタスクを子孫とするため、タスク全体として消費する資源量は正しく管理できる。また、複数の兄弟タスクに対して資源許容量を適切な配分を行なうことによって、複数タスク間における資源消費の競合を避けることが可能となる。

もちろんタスクが資源を超過使用した場合も、その親タスクが処置を行なうこととする。タスク階層の頂点となるタスクが資源を超過使用した場合には、全タスクの資源消費量を管理する意味で PIMOS がその処置を行なう。

資源消費量管理の実現方式

資源消費量管理を行なうためには、次の機構が必要とされることがわかる。

- タスクが基準資源量単位により資源総消費量を管理できる。
- 各資源消費量を基準資源量単位による消費量でタスクに報告できる。

PIMOS の記述言語 KL1 のもつ荘園管理機構をタスク管理へ適用することにより、資源消費量管理が実現される。また OS 資源に関しては、通信路に消費量換算機構を設けることによる実現方式も考えられる (図 5.3)。

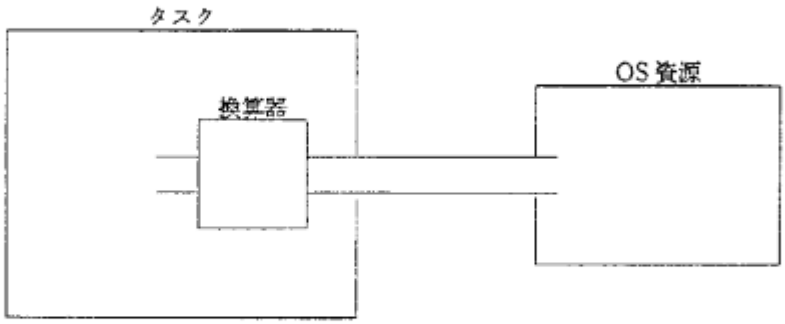

図 5.3: 基準消費量単位への換算

5.3 OS との通信方式

5.3.1 概要

ユーザが入出力装置に対して文字列の読み書きを行なうには、 OS に対して入出力要求を送らなければならない。このようなユーザと OS との通信は、その間に共有変数を用いて通信路 (ストリーム) を張り、そこにユーザの要求メッセージや OS の各種報告メッセージを流すことにより行なわれる。

このようなユーザと OS との通信において考慮しなければならない点は、ユーザの誤りを OS 側に浸透させてはならないことである。つまり、ユーザの誤りから OS を保護しなければならない。ユーザ・ゴールの失敗は、タスク (KL1 の荘園機能) の機構によって (OS が失敗しないように) 保護されている。しかし、 PIMOS もユーザ・プログラムも並列論理型言語 KL1 で記述され、その性質上、ユーザが誤った要求メッセージを OS に流した時に、 OS 側で失敗してしまう場合がある。また、ユーザが OS に対して要求メッセージを送った時点では、その要求に対する OS の処理が完了したことにはならないので、たとえユーザのプログラムが総て終了していても、ユーザ・タスクを解放してはいけない。さらに、ユーザ・タスクが強制解放された時に、ユーザと OS との通信路を正しく解放してやらなければ、 OS 側にユーザからの要求メッセージを受付けるプロセスが永遠に残ってしまう。

以上の理由で、ユーザ–OS 間の総ての通信路には、 OS を保護するためのプロセス (保護フィルタ) が付いている。この保護フィルタをユーザ・タスク上で動作させることにより、ユーザの誤りは OS 側に浸透しない。この保護フィルタを生成するには、ユーザ–OS 間の通信プロトコルを定義しなければならない。逆に、この通信プロトコルが定義されれば、保護フィルタのプログラムを機械的に生成することができる。 PIMOS では、この通信プロトコルを定義するための言語を設け、それから保護フィルタ・プログラムを生成するトランスレータ (プロトコル・コンパイラ) を用意した。

ユーザ–OS 間の具体的な通信メッセージは、主に入出力装置への入出力要求、状態の問い合わせ / 変更である。これらのメッセージは、 OS を介して最終的に FEP(フロントエンド・プロセッサ) に送られる。 FEP が提供する入出力装置には、ウインドウ、ファイル、タイマがある。ウインドウは FEP 上の SIMPOS が提供する Pmacs ウインドウの機能、ファイルはバッファ単位の入出力機能やディレクトリの操作機能等が用意されている。

ユーザのウインドウやファイルに対する操作には、標準入出力プロトコルが用意されている。一方、 OS 側で用意している入出力機能は、ウインドウでは行単位の入力、バッファ単位の出力であり、ファイルではバッファ単位の入出力である。この標準入出力プロトコルと OS が提供する機能との間を埋めるのが、 I/O ユー

ティリティである。I/O ユーティリティは、バッファリング、パーザ、アンパーザの機能を提供し、ユーザ・タスク上で実行される。

5.3.2 OS の保護方式

PIMOS– タスク間通信と OS 保護

タスクと PIMOS は、次の KL1 節で示される通り、共有変数を介して通信を行なう。

```
pimos(SV), task(SV)
```

PIMOS では、共有変数をストリームとして利用することによって PIMOS– タスク間で連続的なメッセージ通信をモジュラリティよく実現する。利用者はファイルやウインドウへの読み書きや OS への問い合わせを行なうために、タスク上のプログラムからこのストリームを利用して要求メッセージを送る。

なおこの項での OS の保護とは、上記 KL1 節における PIMOS 側プロセス中にデッドロック・プロセスや異常終了 (例えば失敗してしまうこと) したプロセスを生成しないことである。

OS 資源の保護

1. 通信終了の確認
   PIMOS も利用者プログラムも並列論理型言語である KL1 で実現されるため、利用者が OS 資源へのメッセージ送信を終了したとしても、メッセージ通信を監視する OS 側プロセスの実行が終了しているとは限らない。そのため OS 資源へのメッセージ通信より利用者プログラム・タスクが先に終了してしまうことも考えられる。そこで利用者が OS 資源へのメッセージ通信が行なわれたことを保証する機構が必要である。 OS– タスク間に実際の OS 資源までの到着確認付きメッセージを用意する。このメッセージを利用することによりそれ以前に送信したメッセージが全て OS 側に到着し、かつ利用者側に到着確認用プロセスが残っていることにより、利用者プログラム用タスクが通信終了により先に正常終了してしまうことを防ぐこともできる。

2. タスク異常終了時の OS 資源の解放
   タスク実行中に異常が生じる等のためにタスクの実行を強制終了させた場合、正しく資源を解放 (資源の利用を終了すること) しないと不具合を生ずることが多い。例えばビットマップ・ディスプレイ上にウインドウが残ったり、他タスクから利用不能なファイルを形成してしまう状況が起こる。

   タスクが異常終了した場合に PIMOS– タスク間の通信路を閉じ、 PIMOS 側に存在する通信路監視用プロセスを終了させ、 OS 資源を正しく解放しかつ PIMOS 中のプロセスのデッドロックを防ぐ必要がある。

   そこで次の機構を用意する。

   - タスクの異常終了を監視し、 PIMOS 中の通信路監視プロセスへ伝達する。

   すなわち、 PIMOS– タスク間全通信路に通信路を閉じるためのバルブ機構を設け、タスク異常終了監視プロセスにバルブ閉じるための全通信路共通なノブを握らせる。タスクが異常終了した場合に監視プロセスはこのノブを回し、タスクがもつ全通信路を閉じる機構である。

   KL1 プログラムとしてのイメージは図 5.4の通りである。通常時にはバルブが開いたままであるので、メッセージは正常に通信路を通過する (図 5.4 第3節)。正常に通信路を閉じた場合には (図 5.4 第2節)、通信が終了する。もしタスクが異常終了した場合にはノブを回す (図 5.4、 Knobをshutにユニファイする) ことによりバルブを閉じ、通信を終了させることができる。

```
  valve(shut,_,Out) :- true | Out = [].
alternatively.
  valve(_,[],Out) :- true | Out = [].
  valve(Knob,[Msg|In],MsgOut) :- true |
          MsgOut = [Msg|Out], valve(Knob,In,Out)
```

図 5.4: バルブ実現イメージ

```
  task(Req) :- true | Req = [getb(N,String)|ReqT],...

  pimos([getb(N,String)|ReqT]) :- true |
          readFromKbd(N,KBDString), KBDString = String,
          . . .
```

図 5.5: 単純な通信方式による例

3. タスク階層と OS 資源の解放
   PIMOS ではタスクの階層が形成されている場合、最も上位であるタスクが異常終了した場合には子孫タスクも異常終了する。そのため、タスクのバルブが閉じられた場合にはそのすべての子孫タスクのもつバルブも閉じられる必要がある。

4. 実現方式の留意事項
   実現方式の留意事項は以下の通りである。

   - バルブに相当するプロセスは、通信路を監視するプロセス側に取り込む。わざわざ図 5.4で示すプロセスを形成する必要はない。
   - 異常終了自体もちろん異常な状況を示しているため、ノブをまわしたり、子孫タスクに対してノブがまわされたことを伝達するプロセスは できるだけ速やかに実行されることが望ましい。

OS の保護

1. OS 異常の防止と保護フィルタ
   利用者はファイルやウインドウ等の OS 資源を利用するために、PIMOS– タスク間通信路にメッセージを流す。PIMOS および利用者プログラムは KL1 で実現されるため、メッセージ中の共有変数を利用して値の受渡しを行なう。しかしながら単純に PIMOS– タスク間で変数を共有した場合に問題が生ずる。指定した文字数の文字列を PIMOS から読み込む例 (図 5.5) を利用して説明する。タスク側は PIMOS 側に第 1 引数に文字数を設定すると第 2 引数に文字列が PIMOS 側から設定されるメッセージ getb/2を送る (図 5.5)。ここでは`N`と`String`が共有変数となる。 PIMOS 側はメッセージ getb/2 を受け取ると`readFromKbd/2`から必要な`N`字分の文字列を読み込み、その文字列を`String`とユニファイしタスク側に返す。しかしこのプログラムには次の問題点が含まれている。

   - タスク側が文字列読み込み用共有変数に PIMOS 側で設定される値と 異なった値をユニファイした場合には、あるゴールが失敗する。 PIMOS は並列論理型言語 KL1 で実現されるため、 図 5.5のゴール実行順序は規定できない。 もしタスク側の共有変数へのユニフィケーションが先に実行された 場合には、PIMOS 側で変数ではなく誤った値と読み込んだ値との ユニフィケーションが起こ

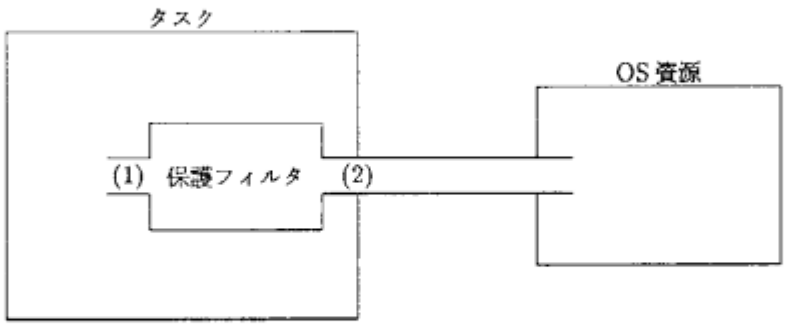

図 5.6: 保護フィルタ付き通信

り、そのゴールが失敗してしまう。 PIMOS 側で失敗が起これば、処理系全体がダウンすることになる。

- タスク側が文字数設定用共有変数に値を設定しなかった場合には、 PIMOS 側にある共有変数の設定を待つプロセスがデッドロックしてしまう。

これらの現象を利用者が簡単に起こすことができることに注意しなくてはいけない。そこで、上記問題を次の方法で解決する。

- PIMOS-タスク間通信路に保護フィルタ・プロセスを設け、上記問題を吸収する。保護フィルタ・プロセスをタスク内で実行することにより、PIMOS を保護する (図 5.6)。

タスクは PIMOS とつながっている通信路に直接要求を送らずに、保護フィルタへの通信路 (図 5.6 (1)) へ要求を送る。このフィルタはタスク側の要求を絶対に失敗しない形式に変換し、また、 PIMOS 側で待ち続けないことが保証されるまで待ってから、 PIMOS への通信路 (図 5.6 (2)) へ変換された要求を送信する。

このように PIMOS 側通信路をタスク側に直接見せないため、タスク側の誤りが PIMOS 側へ波及するのを防ぐだけではなく、タスク側からの悪意を持った PIMOS への侵入を保護フィルタが監視して防ぐことができる。具体的な保護フィルタの役目は以下のようなことである。

- タスク側で設定すべき要求部分は、 フィルタ内で設定されたことを確認してから PIMOS 側へ送る。
- PIMOS 側から設定される要求部分は、 フィルタ内でタスク側とは別の通信路で受け取り、 設定が終了したことを確認してからフィルタがタスク側へ要求部分を送る。

例えば、図 5.6の保護フィルタ・プログラムは図 5.7のように定義できる。簡単に説明する。読み込む文字数Nに値が設定されるまでのを待ってから PIMOS へ要求を送信する。一方、読み込まれた文字列が設定される変数Stringの代わりにまず未定義変数への文字列設定を待ち、この設定の終了を確認してから、保護フィルタによってタスク内変数Stringへ文字列が設定される。従って、タスク側で変数Stringへ誤った値をユニファイしたとしてもフィルタが失敗するだけで PIMOS には影響はない。

保護フィルタ実現のために重要な留意事項がある。あたり前のことなのであるが、タスク (利用者プログラム) 対して PIMOS 側通信路を見せないことである。

```
filter([getb(N,String)|ReqT],OSReq) :- wait(N) |
        OSReq = [getb(N,OSString)|OSReqT],
        waitAndUnify(OSString,String),
        filter(ReqT,OSReqT).

waitAndUnify(PIMOSV,TaskV) :- wait(PIMOSV) | PIMOSV = TaskV.
```

図 5.7: 保護フィルタの例

```
'+a$filter'('+a'(B),OS) :- true |
        OS = '+a'(OSB), '-b$filter'(OSB,B).
'-b$filter'('-b'(C),Task) :- true |
        Task = '-b'(TaskC), '+c$filter'(TaskC,C).
'+c$filter'('+c',OS) :- true | OS = '+c'.
```

図 5.8: 保護フィルタの例

2. OS– タスク間の通信プロトコル

PIMOS– タスク間通信において PIMOS を保護するためには保護フィルタが必要であることがわかった。保護フィルタを形成するにあたり、通信メッセージのどの部分をタスク側が設定し、どの部分を PIMOS が設定するかを明確に定義する必要性がある。すなわち通信プロトコルが存在することがわかる。ここでは、このプロトコルの性質についてまとめる。

PIMOS– タスク間では、どのようなメッセージ通信でもできるわけではない。例えば'+a'('-b'('+c'))のメッセージが PIMOS– タスク間で通信されたとする。このメッセージに対応して次の処理が行なわれる。

(a) まず, '+a'(B)が PIMOS 側へ送られる。

(b) 次に'+a'(B)中のBに PIMOS 側から'-c'(C) が設定される。

(c) 最後に'-c'(C)中のCにタスク側から'+c' が設定されて通信を終了する。

この通信のための保護フィルタは、例えば図 5.8のように記述できる。ところが、2b 終了後、PIMOS 側では 2c の通信に備えて新しくメッセージ監視プロセスが生成されるにも係わらず、タスク内で異常が発生して 2c が実行されない可能性がある。この場合には PIMOS 内にデッドロック・プロセスが発生してしまう。

以上の現象を防ぐには、 5.3.2で述べたバルブ機構を全メッセージ監視プロセスへ導入する必要性があるのだが、非常に煩雑な実現方式になってしまう。そこで特殊なプロトコルを導入することによってこの問題を回避する。通常のプロトコルに加えて、 5.3.2で述べた連続したメッセージ通信のためにストリーム型プロトコルを導入する。ストリーム型プロトコルのみバルブ機構によってストリームを閉じるという統一した概念のもとで PIMOS 側に異常なプロセスを生じないこととする。実はストリーム型プロトコルでは、メッセージをひとつ受信した後に必ず、メッセージを受信するプロセスを発生させるため (ストリーム自身、ストリームを閉じるまで不完全なメッセージであることに起因する)、バルブ機構を必要としているのである。

以上の概念をもとに PIMOS– タスク間通信プロトコルに従ったメッセージの制限を表 5.1に挙げる。

表 5.1: メッセージ領域

分類	領域	記号	定義
メッセージ	入力メッセージ	IM	$IP + (IP \times ARGS)$
	ストリーム内入力メッセージ	SIM	$IP + (IP \times IARGS)$
	出力メッセージ	OM	$OP + (OP \times OARGS)$
	入力ストリーム・メッセージ	ISM	$ISP + (ISP \times IARGS)$
	出力ストリーム・メッセージ	OSM	$OSP + (OSP \times OARGS)$
	任意メッセージ	ANY	$ANYP$
プロトコル	入力プロトコル	IP	
	出力プロトコル	OP	
	入力ストリーム・プロトコル	ISP	
	出力ストリーム・プロトコル	OSP	
	任意プロトコル	$ANYP$	
引数群	引数群	$ARGS$	$ARG + (ARG \times ARGS)$
	入力引数群	$IARGS$	$IARG + (IARG \times IARGS)$
	出力引数群	$OARGS$	$OARG + (OARG \times OARGS)$
引数	引数	ARG	$IM + OM + ISM + OSM$
	入力引数	$IARG$	$SIM + ISM$
	出力引数	$OARG$	$OM + ISM + OSM + ANY$

5.3.3 プロトコルコンパイラ

プロトコル定義言語

PIMOS-タスク間通信のための保護フィルタ・プログラムは PIMOS で提供される。このプログラムは、PIMOS-タスク間の通信プロトコルが正確に定義できれば、機械的に生成することができる。このような定義の簡単さと比較してプログラム・バグを防ぐためにプロトコル定義言語を設定し、定義からフィルタ・プログラムを生成するトランスレータを用意する。このトランスレータのことをプロトコルコンパイラと呼ぶ。

プロトコル定義方法として、通常プロトコル定義の他にパラメータ付きプロトコルの定義も許すことにする。パラメータ付きプロトコル定義は、実際には実パラメータにプロトコルを与え、通常プロトコル定義形式に展開したうえフィルタ・プログラムに変換される。プロトコル定義言語の文法を図 5.9に示す。

基本プロトコル定義としては次の 4 つを用意することにする。

```
 definition ::= name[ ( formals ) ] = [ inheritance ] body.
inheritance ::= { super & }
      super ::= invocation
       body ::= [ sign ] ( invocation | { enumeration } )
 invocation ::= name [ ( actuals ) ] | parameter
enumeration ::= pattern { ; pattern }
    pattern ::= atom [ ( actuals ) ]
    formals ::= parameter { , parameter }
    actuals ::= body { , body }
  parameter ::= name
       sign ::= + | -
```

図 5.9: プロトコル定義言語文法

- atomic
  メッセージが atomic 型であるときに、通信を行なう。

- any
  保護フィルタなしで通信を行なう。

- stream(P)
  プロトコル P に従ったメッセージをストリーム型プロトコルに従って 通信を行なう。パラメータ付きプロトコルである。

- vector(P)
  プロトコル P に従ったメッセージを要素とするベクタを メッセージとする通信を行なう。パラメータ付きプロトコルである。

フィルタ・プログラム生成法

トランスレータは、基本的に指定されたプロトコル用のフィルタ・プログラムを必要とする最小限のプログラムのみを生成する。展開規則の概要は下記の通りである。

- プロトコルの参照がある場合には、 その参照が基本プロトコル定義もしくは構造をもった定義になるまで トップダウンに参照を続ける。 途上でででてきたプロトコル(単に左辺の右辺での置き換えで済むプロトコル) 定義に対するプログラムは生成しない。

- 参照の探索中でループが存在した場合には、誤りとする。

- 符号は記述上、定義の入れ子に対して相対的になっているため、 すべて絶対的符号に変換する。

- 符号が+(タスク側が設定する)である参照プロトコルがあれば、 KL1 節を 2 つに分け、第 1 節で指定されたプロトコルの 保護フィルタを呼び出し、第 2 節のガードでこの保護フィルタを 通ったメッセージ部分を監視する。 そして、ガードを通った時点で PIMOS と通信を行なう。

- 符号が−(PIMOS 側が設定する)である参照プロトコルがあれば、 KL1 節をひとつだけ生成して、 その節のボディでユニフィケーションの失敗に備えるために 指定されたプロトコルの保護フィルタを呼ぶ。フィルタをメッセージ部分が通った時点でタスク側変数とユニファイする。

- ストリーム型プロトコルはストリームを流れるメッセージの タスク側設定部分が完成する度に PIMOS 側へ送信する。

- enumeration はひとつひとつの pattern に対してそれぞれ フィルタ・プログラムを生成する。

例外的展開法として次のことが挙げられる。

- ＋と指定された参照プロトコルが atomic であるならば、生成される最初の節内のガードにフィルタに相当する述語を生成し、 2 番目の節中のガードにその監視用述語を生成しない。 よって、もし符号が+である参照プロトコルがすべて atomic ならば 2 番目の節を生成する必要はない。

- any プロトコルはフィルタを生成しない。

5.3.4 通信プロトコル

ここでは、ユーザ/OS間の具体的な通信プロトコルについて述べる。 PIMOSがユーザに提供する入出力機能として、次のようなものが必要とされた。

1. ウインドウに対する入出力機能
   - 標準入出力機能。
   - ウインドウ生成の際、pmacs-バッファを指定することができること。
   - ウインドウのサイズは、あるフォントにおける文字単位、行単位で指定することができること。
   - 入出力処理の最中でも、OSからのリクエスト、またはユーザからのウインドウへのアテンション・キーによって処理をアボートするためのラインを設けること。
2. ファイルに対する入出力機能
   - ファイルのオープン・モードとして、read、write、appendの3種類を用意すること。
   - 標準入出力機能。
   - ディレクトリに対していくつかの操作が可能なこと。

これらの入出力機能を実現するため、通信プロトコルとして、以下のようなメッセージをユーザに提供することとした。各メッセージのStatusは、メッセージの処理が正常に終了したかどうかを返す変数であり、すべてのメッセージに存在する。また、変数についている^記号は、その変数がOS側でバインドされるものであることを意味する。

ジェネラル・リクエスト・ストリームへのメッセージ

ユーザは、次のようなメッセージをジェネラル・リクエスト・ストリームに流すことによって、各サブ・システムへのリクエスト・ストリームを得ることができる。

`window(Window,^Status)`

ウインドウ・サブシステムへのリクエスト・ストリームを得る。

`file(File,^Status)`

ファイル・サブシステムへのリクエスト・ストリームを得る。

`timer(Timer,^Status)`

タイマへのリクエスト・ストリームを得る。

サブシステムへのメッセージ

ユーザは、各サブシステムへ、各サブシステム・リクエスト・ストリームを通じて次のようなメッセージを送ることができる。ユーザは、それによってそれぞれのデバイスへのストリームを得る。

1. ウインドウ・サブシステムへのメッセージ

```
create(Fep_number,Initiation,Window,Abort,^Attention,^Status)
create(Fep_number,Window,Abort,^Attention,^Status)
create(Initiation,Window,Abort,^Attention,^Status)
create(Window,Abort,^Attention,^Status)
```

指定されたFEP上にウインドウを生成し、そのウインドウへのストリームを得る。

このメッセージによって、ウインドウの生成と同時にresetメッセージ(後述)が発行される。

Abort、Attentionはそれぞれ、resetメッセージによって張られたアボート・ライン、アテンション・ラインである。

Initiationは初期設定情報で、リスト形式で以下の要素が許される。

- inside_size(Characters,Lines) :

  生成するウインドウのサイズを、文字数、行数で指定する。

- size(manipulator) :

  生成するウインドウのサイズを、マニピュレータで指定する。

- position(X, Y) :

  生成するウインドウの左上のポジションを、X、Y座標(ドット数)で指定する。

- position(manipulator) :

  生成するウインドウのポジションを、マニピュレータで指定する。

- title(String) :

  タイトルを指定する。

- buffer(Buffer_name) :

  ウインドウ生成時に割り当てられるテキスト・バッファを名前で指定する。指定されたバッファが存在しない場合、新しいバッファをその名前で生成し、ウインドウに割り当てる。

- deactivate :

  ウインドウ生成時にウインドウをスクリーン上に表示しない。初期設定にdeactivateを記述しなかった場合には、ウインドウ生成と同時に、そのウインドウは自動的にスクリーン上に表示される。

上記のInitiationの要素のうち、記述されなかった項目については、デフォルト値が設定される。

デフォルト値は、

size : スクリーンと同じ
position : 0,0
title : なし
バッファは新規に作成

である。

初期設定のうちいずれかが満たされない場合、ウインドウは生成されない。初期設定情報を特に指定しない場合、デフォルト値が用いられる。FEPを特に指定しない場合は、ユーザがログインしているFEP上に生成する。

```
get_max_size(X,Y,Font_pathname,^Characters,^Lines,^Status)
get_max_size(X,Y,^Characters,^Lines,^Status)
get_max_size(Font_pathname,^Characters,^Lines,^Status)
get_max_size(^Characters, ^Lines, ^Status)
```

フォントをパス名で指定し、X、Y座標をドット数で指定すると、そこをウインドウの左上端と仮定した場合に生成することができるウインドウの最大の大きさを、そのフォントにおける文字数、行数で返す。

フォントを特に指定しなければ、デフォルトのフォントにおける文字数、行数を返す。

X、Y座標を特に指定しなければ、スクリーンのサイズを文字数、行数で返す。

2. ファイル・サブシステムへのメッセージ

```
open(Fep_number,Pathname,read,File,Abort,^Attention,^Status)
open(Pathname,read,File,Abort,^Attention,^Status)
```

指定されたFEP上の、Pathnameで指定された既存のファイルを入力ファイルとしてオープンし、そのファイルへのストリームを得る。ファイルのオープンと同時にresetメッセージ(後述)が発行される。Abort、Attentionはそれぞれ、resetメッセージによって張られたアボート・ライン、アテンション・ラインである。

ファイルが存在しない場合はオープンされない。

FEPを特に指定しない場合、ユーザがログインしているFEPに対して行われる。

```
open(Fep_number,Pathname,write,File,Abort,^Attention,^Status)
open(Pathname,write,File,Abort,^Attention,^Status)
```

指定されたFEP上に出力ファイルを作成オープンし、そのファイルへのストリームを得る。ファイルのオープンと同時にresetメッセージ(後述)が発行される。Abort、Attentionはそれぞれ、resetメッセージによって張られたアボート・ライン、アテンション・ラインである。

Pathnameで指定されたファイルがそのFEP上に存在しない場合は、ファイルを新規に作成してオープンする。

ファイルがすでに存在する場合は、同じ名前でバージョンアップされたファイルが作成オープンされる。

FEPを特に指定しない場合、ユーザがログインしているFEPに対して行われる。

```
open(Fep_number,Pathname,append,File,Abort,^Attention,^Status)
open(Pathname,append,File,Abort,^Attention,^Status)
```

指定されたFEP上に出力ファイルを追加オープンし、そのファイルへのストリームを得る。このメッセージによって、ファイルのオープンと同時にresetメッセージ(後述)が発行される。

Abort、Attentionはそれぞれ、resetメッセージによって張られたアボート・ライン、アテンション・ラインである。 Pathnameで指定されたファイルが存在しない場合は、ファイルを新規に作成してそのファイルをオープンする。

ファイルがすでに存在する場合は、そのファイルをオープンする。

FEPを特に指定しない場合、ユーザがログインしているFEPに対して行われる。

directory(Fep_number,Pathname,Directory,Abort,^Attention,^Status)
directory(Pathname,Directory,Abort,^Attention,^Status)

指定されたFEP上のディレクトリを検索し、そのディレクトリへのストリームを得る。このメッセージによって、ディレクトリへのストリームの獲得と同時に resetメッセージ(後述)が発行される。

Abort、Attentionはそれぞれ、reset メッセージによって張られたアボート・ライン、アテンション・ラインである。 Pathnameで指定されたディレクトリが存在しない場合は、ストリームは接続されない。

FEP を特に指定しない場合、ユーザがログインしている FEP に対して行われる。

3. タイマへのメッセージ

get_count(^Count,^Status)

午前0時0分から現在までのミリ秒数を返す。

on_at(Count,^Now,^Status)

指定された時刻になると、Nowを wake_up にバインドする。

on_after(Count,^Now,^Status)

指定された時刻が経過すると、Nowを wake_up にバインドする。

デバイスへのメッセージ

ユーザは、各デバイスへ、各デバイス・ストリームを通じて次のようなメッセージを送ることができる。ユーザは、それによって入出力を行う。

1. 各デバイスに共通なメッセージ

reset(Abort,^Attention,^Old_stream,^Status)

デバイスに対して、新規にアボート・ライン、アテンション・ラインを張る。 resetメッセージは、デバイスの生成と同時に送られる。

以降、デバイス側でユーザ割り込みが発生した場合は、デバイスによって変数Attentionが control_C にバインドされる。

また、デバイスに対して入出力処理の中断を指示する場合は、変数Abortを abort にバインドすることによって通知する。中断を指示した場合、新たに各ラインを張り直すため、このメッセージを再発行しなければならない。それが到着するまでの間にデバイスに届くその他の I/O メッセージはすべて中断される。resetメッセージの到着と同時に各ラインは新たに張り直され、中断されたメッセージ列を含む今までのストリームが、Old_streamに返される。Old_streamは、中断されたメッセージ列を最施行したい場合、用いることができる。

next_attention(Attention,^Status)

デバイスに対して、新規にアテンション・ラインを張る。

デバイス側でユーザ割り込みが発生したが、中断を指示しない場合には、アテンション・ラインのみを張り直せばよい。そのような場合に有効である。

flush(^Status)

最新のバッファの内容をただちにデバイスに出力する。

2. ウインドウへのメッセージ

(a) 入出力メッセージ

getl(^Line,^Status)

ウインドウから1行入力する。

putb(String,^Status)

ウインドウにストリングを出力する。ストリング長に制限はない。

beep(^Status)

ベルを鳴らす。

(b) ウインドウ属性メッセージ

set_inside_size(Characters,Lines,^Status)

文字数、行数でサイズを指定する。

set_size(manipulator,^Status)

マニピュレータでサイズを指定する。

set_size(screen,^Status)

ウインドウのサイズをスクリーンと同じにする。

set_position(X,Y,^Status)

X、Y座標(ドット数)でウインドウの左上端のポジションを指定する。

set_position(manipulator,^Status)

マニピュレータでポジションを指定する。

set_title(String,^Status)

タイトルを指定する。

set_font(Pathname,^Status)

フォントをパス名で指定する。

reshape(X,Y,Characters,Lines,^Status)

X、Y座標(ドット数)、文字数、行数でリシェイプする。

reshape(manipulator,^Status)

マニピュレータでリシェイプする。

select_buffer(Buffer_name,^Status)

テキスト・バッファを名前で指定する。指定されたバッファが存在しない場合は新しいバッファをその名前で生成し，ウインドウに割り当てる。

get_inside_size(^Characters,^Lines,^Status)

そのウインドウのサイズを文字数、行数で返す。

get_position(^X,^Y,^Status)

そのウインドウの左上端のポジションをX、Y座標(ドット数)で返す。

get_title(^String,^Status)

タイトルを返す。

get_font(^Pathname,^Status)

そのウインドウのフォントをパス名で返す。

get_buffer(^Buffer_name,^Status)

そのウインドウに指定されているテキスト・バッファの名前を返す。

activate(^Status)

ウインドウをスクリーン上に表示する。

deactivate(^Status)

ウインドウのスクリーンへの表示をやめる。
画面イメージは保存されており、再びメッセージactivate(^Status)を流すと元のイメージが表示される。

show(^Status)

ウインドウが他のウインドウによって隠されていない状態となる。ただし、ウインドウがスクリーン上に表示されていないとき(deactivate状態)は、メッセージactivate(^Status)と同じ働きをする。

hide(^Status)

ウインドウを他のウインドウ中の最下層に移動する。

clear(^Status)

ウインドウのテキスト・バッファの内容を消去する。

3. 各ファイルへのメッセージ

(a) 各ファイルに共通のメッセージ

pathname(^Full_pathname,^Status)

そのファイルの完全パス名を返す。

(b) 入力ファイルへのメッセージ

getb(Size,^String,^Status)

ファイルから、サイズ長のストリングを返す。
ファイルから取り出せるデータがサイズ長に満たない場合は、有効な長さ分だけを返す。

end_of_file(^Ans,^Status)

入力の終了を問い合わせる。

(その他の入力メッセージについては、5.3.5 I/Oユーティリティ参照)

(c) 出力ファイル、追加ファイルへのメッセージ

putb(String,^Status)

ファイルにストリングを出力する。ストリング長に制限はない。

(その他の出力メッセージについては、5.3.5 I/O ユーティリティ参照)

4. ディレクトリへのメッセージ

`pathname(^Full_pathname,^Status)`

そのディレクトリの完全パス名を返す。

`listing(Wildcard,^Filename,^Status)`

ワイルドカードに適合するファイルのファイル名を返す。

`listing(^Filename,^Status)`

ディレクトリ上にあるすべてのファイルのファイル名を返す。

`delete(Wildcard,^Status)`

ワイルドカードに適合する名前のファイルを削除する。

`undelete(Wildcard,^Status)`

削除されているファイルで、名前がワイルドカードに適合するものを回復する。

`deleted(Wildcard,^Filename,^Status)`

削除されたファイルで、ワイルドカードに適合するもののファイル名を返す。

`deleted(^Filename,^Status)`

削除されたすべてのファイルのファイル名を返す。

`purge(Wildcard,^Status)`

ファイル名がワイルドカードに適合するファイルのうち、最大バージョンを持つものだけを残し、それ以外のものを一括抹消する。 Wildcard にバージョンの指定はできない。このメッセージが送られた時点で削除中のファイルは抹消される。

`purge(^Status)`

すべてのファイルのうち、最大バージョンを持つものだけを残し、それ以外のものを一括抹消する。このメッセージが送られた時点で削除中のファイルは抹消される。

`expunge(^Status)`

削除中のファイルを一括抹消する。

5.3.5 I/O ユーティリティ

I/O ユーティリティは、I/O バッファリング、パーザ、アンパーザ等の機能を提供する。ユーザはデバイス・ストリームを I/O ユーティリティ付で獲得することが出来る。そして、このストリームにコマンドを流すことにより所要の入出力を行える。I/O ユーティリティのプログラムはユーザ荘園内で起動されるので、資源管理、例外処理など注意を要する。

バッファリング

PE間、荘園間、荘園内外の通信回数を減らして効率を上げるために、バッファリングが必要となる。I/Oユーティリティでは、入出力バッファを用意しユーザの入出力要求はこのバッファを介して行う。そして、PIMOSへの入出力要求はバッファ単位で行われる。

パーザ、アンパーザ

項単位に入出力を行うために、パーザ、アンパーザが必要となる。これは演算子順位文法に従って行われる。

演算子とその優先度の対応表をオペレータ・テーブルといい、パーズ、アンパーズはこれに基づいて行われる。オペレータ・テーブルはデバイス・ストリームの属性であり、その変更要求はこのストリームに対して行う。ストリーム獲得時にはKL1標準のオペレータ・テーブルが設定されている。

1. パーザ
   パーザは文字列から項を構成するものである。パーザはオペレータ・テーブルと変数表を参照しながらパーズを進めて行く。

   変数表は変数名と変数の対応表である。パーズ中にテキスト文字列中に変数表に登録してある変数名が現れた場合、同じ変数として扱われる。また、変数表に未登録の変数名は変数表に登録される。

   パーズはオペレータの型とその優先順位に基づいて行われる。これはオペレータ・テーブルに登録してある。オペレータの型と優先順位が複数ある場合、その全てについてパーズを試みる必要がある。KL1では、これはプロセスのフォークで実現されている。

   パーズの結果、1つの項も得られなかったり複数の項が得られた場合パーズ結果としてそのことが報告される。

2. アンパーザ
   アンパーザは項から文字列を生成するものである。アンパーザはオペレータ・テーブルを参照しながらアンパーズを進めて行く。

   構造体に対してアンパーズする範囲を指定することができる。これは構造の深さと長さで指定する。指定された範囲以上のアンパーズは行われない。これは項の出力時に冗長な部分を出力したくない時に使用される。

   アンパーザはKL1で記述されているため変数を含むような項をアンパーズしようとするとサスペンドするので注意が必要である。

I/Oユーティリティで提供される入出力単位は、文字、行、項、ブロックである。以下に、コマンドを列挙する。Cは文字コード、Lは行、Tは項、Bはブロックを意味する。

入力コマンド

- getc(C)

  1文字Cを入力する。読み込みが終了した場合そのことを示す特別なアトムがCとユニファイされる。

- getl(L)

1行Lを入力する。読み込みが終了した場合そのことを示す特別な アトムがLとユニファイされる。

- gett(T,VT,NVT)

  項1個を構成する文字列を読み込んで、変数表VTを用いてパーズしTとユニファイする。更新された変数表をNVTとユニファイする。変数表を特に指定しないgett(T)もある。
  読み込みが終了した場合そのことを示す特別なアトムがTとユニファイされる。

- getb(B,Size)

  Size文字読み込みブロック化し、Bとユニファイする。読み込みが終了した場合そのことを示す特別なアトムがBとユニファイされる。

- skip(N)

  文字コードNまで読み飛ばす。次の入力要求はNの次の文字から行われる。

出力コマンド

- putc(C)

  1文字Cを出力する。

- putl(L)

  1行Lを出力する。

- putt(T,Length,Level)

  項Tを長さLength深さLevelの範囲でアンパーズし、結果の文字列を出力する。

- putb(B,Count)

  ブロックBの初めのCount文字を出力する。

その他のコマンド

- do(Stream)
  入出力要求はストリームに対してなされ、そのストリームには複数のストリームがマージ入力されているかもしれない。したがって、続けて出した入出力要求の間に他の入出力要求が挿入されることもある。このコマンドは、プロンプトの出力要求とそれに対する入力要求といった不可分な要求をする時に使用される。Streamには任意のコマンドを流すことができ、このコマンド間には他のコマンドは挿入されない。

- flush(Done)
  出力バッファに蓄えられた出力をPIMOSに送る。実際に送られるとDoneが具体化される。ストリームを閉じることにより資源の開放は行えるが、パルプ機構により入出力要求はPIMOSに届いてない可能性がある。このコマンドによってPIMOSに実際に要求が届いたかどうか確かめてからストリームを閉じる必要がある。

5.4 例外処理

例外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。したがって例外はメタ論理的意味のあるものである。

例外処理機能は、オペレーティング・システム構築といったシステムプログラミングには必須の機能である。KL1では例外処理や資源管理といったメタ操作のために荘園機能を提供している。このことは例外処理の最小単位が荘園であることを示している。ユーザは荘園機能を用いて例外処理を記述することができる。

本節ではKL1で提供される例外処理機能について触れ、それを用いた例外処理方法について述べる。

5.4.1 例外の種類

KL1で扱うことのできる例外は大きく2つに分けられる。1つは言語定義例外、もう1つはソフトウェア定義例外である。

1. 言語定義例外
   言語定義例外には次のようなものがある。

   - フェイル
     荘園内のフェイルは荘園外では例外として認識される。つまり荘園内のフェイルは荘園外に漏出しない。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。
   - 組込み述語エラー
     組込み述語で扱うことのできないデータを扱おうとした。0除算などの算術演算エラー、型不一致などがある。
   - 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デバッグ等で有用である。
   - 資源の超過使用
     荘園内で消費できる資源(リダクション数)の上限は予め決められている。実行中にこの上限を超えて資源を消費しようとすると例外となる。但しマルチ・プロセッサ上で実行しているため、実際には上限を超えていなくてもこの例外が発生する場合がある。
   - デッドロック
     並列言語特有の例外で，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。GC時に発見されることがある。

2. ソフトウェア定義例外
   ソフトウェア定義例外はソフトウェアで積極的に起こした例外で、この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。また、荘園の外側[1]との通信にも使用することができる。レイズについては5.4.4で詳しく述べる。

[1] OS等

5.4.2 例外処理機能

execute のレポート・ストリームのメッセージによって荘園内の例外を知ることができる。ガード部で例外が起こった場合は、その節が失敗節になる。

メッセージにはその例外に対する処理を記述出来るものがある。以下で示す変数 NewGoal がそれで、これは実行を継続するのに用いられる。 NewGoal にゴールをユニファイさせると例外事象を発生させたゴールの代わりにそのゴールが実行される。 NewGoal は具体化されていないことが保証されている。

5.4.1で述べた例外の種類によって次のような意味のメッセージが流れてくる。

- ゴールを実行しようとしたら例外事象が発生した。 このメッセージには次のような情報が含まれる。 例外を起こしたゴール。例外のタイプ。 NewGoal。
- 実行中に許されたリダクション数を超えてリダクションをしようとした。
- デッドロック状態が発見された。 デッドロックしているゴール群の情報も含まれる。
- raise(Type, Information, NewGoal) 組込み述語 raise/3 が実行された。 Type、Information は raise/3 で渡された項である。 Type は基底項であることが保証されている。

  Information は変数を含んでいるかも知れないので、 これを参照するようなプロセスはデッドロックする可能性がある。 これは荘園内の例外事象が荘園外に浸出することになるので 十分な注意が必要である。 Type に関しては基底項であることが保証されているので、 このようなことはない。

5.4.3 例外処理記述

5.4.2の機構とコントロール・ストリームにコマンドメッセージを与えることにより、荘園内の例外に対処することができる。例外処理プロセスはレポート・ストリームとコントロール・ストリームを監視する必要がある。

例外処理プロセスはコントロール・ストリームに abort を流すことによってその荘園を放棄することができるし、 NewGoal を true に具体化することによって例外事象を発生させた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。

例えば 0 除算の場合、例外のタイプはゼロ除算、例外を起こしたゴールは$X := 5/0$となる。この時 NewGoal に$X := 999999$をユニファイすることにより実行を継続できる。

またレイズ時の Information には任意の項が許されるから、これを用いて荘園の外から中に様々な情報を渡すことができる。 PIMOS はこの機構を利用してユーザにジェネラル・リクエスト・ストリームを供給する。

資源の超過使用の場合は注意が必要である。この場合コントロール・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。資源追加割当てメッセージは例外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、このような使用を避けるべきである。

荘園起動時にその荘園で処理する例外の種類をタグによって指定する。その荘園内指定されている以外の例外事象が発生した場合、その荘園のレポート・ストリームには通知されず、外側でその例外事象を処理しようとしている最も内側の荘園のレポート・ストリームに通知される。

ユーザ荘園は全て PIMOS 管理の荘園内で起動される。この荘園は全ての例外を処理するようにタグが設定されている。したがって、ユーザが処理しない例外事象についてはこの荘園に対する PIMOS 定義の例外処理が行われる。

5.4.4 レイズ

レイズは次の組込み述語によって起こすことができる。

raise(Tag, Type, Information)

Tag と Type が基底項になるまでサスペンドし、基底項になると例外事象が発生する。

一般に荘園は多重にネストする。例外事象が発生した場合、その直近の荘園のレポート・ストリームにメッセージが流れるのは煩わしい場合がある。例えば、例外処理プロセスを荘園毎につけなくてはならなくなる。この様な時にある例外については、何段か飛び越して外側の荘園のレポート・ストリームにメッセージを流す機能があると便利である。

レイズを1段づつ繰り返す方法によってこの機能を代替できそうであるが、この方法では例外事象が途中の荘園に浸出したり、例外処理の対処法がうまく伝わらなかったりする可能性がある。これは Information に NewGoal の情報を安全に渡せないからである。

荘園起動時にその荘園内のどのような例外をレポート・ストリームに受信するかを示すタグが設定される。ある例外事象が発生した場合、その例外事象を受信するタグが設定されている最内側の荘園のレポート・ストリームにメッセージが流れる。raise/3 の第1引数 Tag はこのタグを示すものである。Tag によってどの荘園にレイズを知らせるかを指定することができる。

Information を通して荘園の外側に変数を含む任意の項を渡すことができる。これを用いて荘園の外側と通信することができる。ジェネラル・リクエスト・ストリーム獲得にもこの手法を用いる。

5.5 シェル

5.5.1 シェルの機能

シェルは、ユーザが(インタラクティブに)指定したゴールを起動し、実行するためのユーザ・インタフェースの一つである。換言すれば、シェルは、ユーザがシェル用に開発したプログラムを起動、管理するために用意されたコマンド・プロセッサである。以下に、その機能を概説する。

1. タスクの生成
   ユーザの指定したゴールを実行する際には、シェルは新たに作成したタスク(ユーザ・タスク)の中にそのゴールを挿入し、実行する。

2. 標準入出力
   シェルがユーザ・タスクを起動する際には、タスクに対して、標準入力、標準出力、標準メッセージ出力の三つの OS 資源(ストリーム)を割り当てる。これら資源は、ユーザがゴールと共に指定することが可能であるが、特に指定がなければ、シェル自身が持っている標準入出力を与える。もちろん、ユーザがゴールの中で他の OS 資源を獲得し利用することには、なんら問題はない。

3. タスク間通信機能
   あるタスクの出力を他のタスクの入力としてタスク同士を連結したい場合がある。その様な場合には、シェルが提供するタスク間通信機能(パイプ)を利用してタスクの標準入出力間を連結し、タスク間で通信を行うことができる。

4. ジョブ
   パイプを用いて連結された一連のタスク群をジョブと呼ぶ。シェルは、自身が起動したユーザ・タスクを個々のタスクごとに管理するのではなく、このジョブを単位として管理する。なぜなら、ジョブを起

動したユーザの立場から考えると、 ジョブ内にあるタスクのうちの一つにでも異常があれば、 それはジョブ全体が異常を起こしたことと同様であるし、 さらにまたジョブの中の一つのタスクでも終了していなければ、 ジョブが終了したとは考えられないためである。

なお、ここで言う管理とは、ジョブの起動、停止、再開、終了などの ジョブの状態変更およびジョブの状態問い合わせの事をさす。

5. シェル・ウインドウの共有
   シェルは通常、ユーザとのインタラクションを行うためのウインドウ (シェル・ウインドウ) を持つ。しかし、このウインドウは、 シェルが起動したタスクに標準入出力として割り当てられる場合が あるため、複数のタスクが競合して使用する可能性がある。 その際、複数のタスクが同時に出力を行うと、表示内容が乱れ、 ユーザが出力内容を認識できなくなる恐れがあり、 またさらには複数のタスクが同時に入力を行うことは事実上不可能である。 すなわち、ある一つのタスクがシェル・ウインドウに対して、入出力動作を実行中には、他のタスクがそのウインドウに対して 入出力動作を行うことを許してはならない。 そのため、シェル・ウインドウに対し、排他制御を行う必要がある。

6. フォア・グラウンドとバック・グラウンド
   シェルは、ジョブの実行環境として、 フォア・グラウンドとバック・グラウンドと呼ぶ二種類の環境を提供し、 シェル・ウインドウの排他制御を行う。 フォア・グラウンドに投入されたジョブとシェルは、排他的にシェル・ウインドウを利用する。 一方、バック・グラウンドに投入されたジョブは、 シェル・ウインドウに対するアクセスを常に抑制されている。 仮にアクセスした場合、その入出力動作の実行は開始されない。

7. シェル・コマンド
   バック・グラウンドのジョブがシェル・ウインドウにアクセスする 必要があれば、そのジョブをフォア・グラウンドに移行させねばならない。 この操作を行うため、シェルは特殊なコマンドを用意する。

   また、フォア・グラウンドのジョブを停止させるための機構も必要である。 これは、ウインドウのアテンションの機構によって実現する。

5.5.2 ジョブの構成

シェルは、各ユーザ・タスクにシェル専用の例外ハンドラを付与して起動する。この例外ハンドラの機能は、各タスクの実行状態の監視および変更である。以下に、必要とする機能を列挙する。

- タスクの終了検出およびその報告
- タスクの異常終了検出およびその報告
- タスクの起動 (再開)
- タスクの中断
- タスクの強制終了
- タスクの状態問い合わせ

ジョブの管理は、これらの例外ハンドラ同士をジョブ単位にストリームでつないで行う。このことからジョブは、共通の例外処理を行う一連のタスク群としても捉えられる。ジョブの構成を図 5.10に示す。

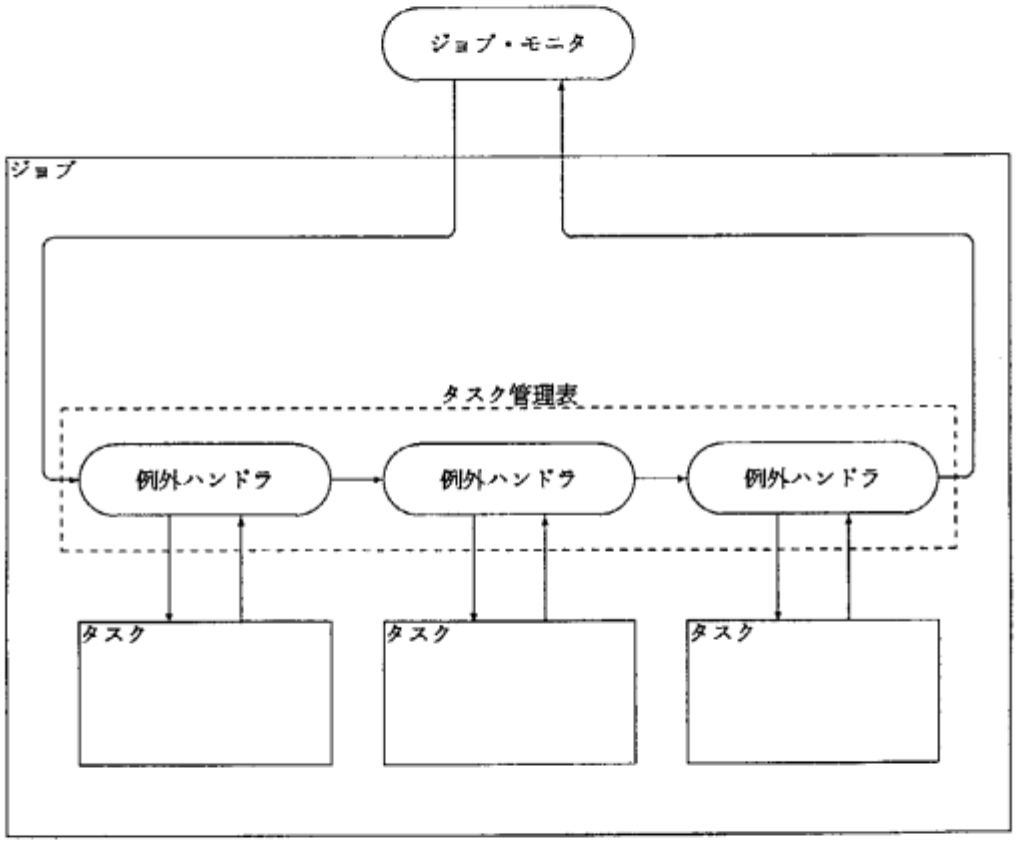

図 5.10: ジョブの構成

図に示したように一つのジョブに対応する例外ハンドラ群は、ループ状に結合される。このループをタスク管理表と呼ぶ。ループ内には、そのループ自身を管理する (すなわちジョブを管理する) ためのプロセスが必ず一つ存在する。このプロセスをジョブ・モニタと呼ぶ。シェルは、このジョブ・モニタを用いてジョブを管理する。

ジョブ・モニタは、シェルからのメッセージ (例えばジョブの中断) を受け取ると、そのメッセージをループ上に送出する。メッセージを受け取った各例外ハンドラは、メッセージを解読し、自身の管理しているタスクのコントロール・ストリームにメッセージを送る。

また逆に、例外ハンドラがタスクのレポート・ストリームからメッセージ (例えばジョブの終了報告) を受け取ると、例外ハンドラがループ上にメッセージを送出する。さらにそのメッセージをジョブ・モニタが受け取り、シェルに報告する。

5.5.3 ジョブ・プールの構成

前述の通り、シェルは、各ジョブにジョブ・モニタを付与する。このジョブ・モニタの機能は、各ジョブの実行状態の監視および変更である。以下に、必要とする機能を列挙する。

- ジョブの終了検出およびその報告
- ジョブの異常終了検出およびその報告
- ジョブの起動 (再開)
- ジョブの中断
- ジョブの強制終了
- ジョブの状態問い合わせ

ジョブ・プールの管理は、これらのジョブ・モニタ同士をストリームでつないで行う。ジョブ・プールの構成を図 5.11に示す。

図に示したようにジョブ・モニタ群は、ループ状に結合される。このループをジョブ管理表と呼ぶ。ループ内には、そのループ自身を管理する (すなわちジョブ・プールを管理する) ためのプロセスが必ず一つ存在する。このプロセスをジョブ・マネージャと呼ぶ。シェルは、このジョブ・マネージャを用いてジョブ・プールを管理する。

ジョブ・マネージャは、シェルからのメッセージ (例えばジョブの中断) を受け取ると、そのメッセージと宛先 (何らかのジョブ識別子) を組にしてループ上に送出する。宛先に該当するジョブ・モニタは、メッセージを受け取ると、そのメッセージを解読し、自身の所属しているタスク管理表のループにメッセージを送る。

また逆に、ジョブ・モニタがタスク管理表を巡回してきたメッセージ (例えばジョブの終了報告) を受け取ると、ジョブ・モニタがジョブ管理表のループ上にメッセージを送出する。さらにそのメッセージをジョブ・マネージャが受け取り、シェルに報告する。

ジョブ・マネージャの機能は、ジョブの実行状態の監視および変更である。

5.5.4 タスク間通信機能 (パイプ)

シェルは、タスク間の通信を行うための機能としてパイプと呼ばれる専用の通信路を提供する。このパイプを用い、タスク群の標準入出力間を結合することで、ジョブを構成している。

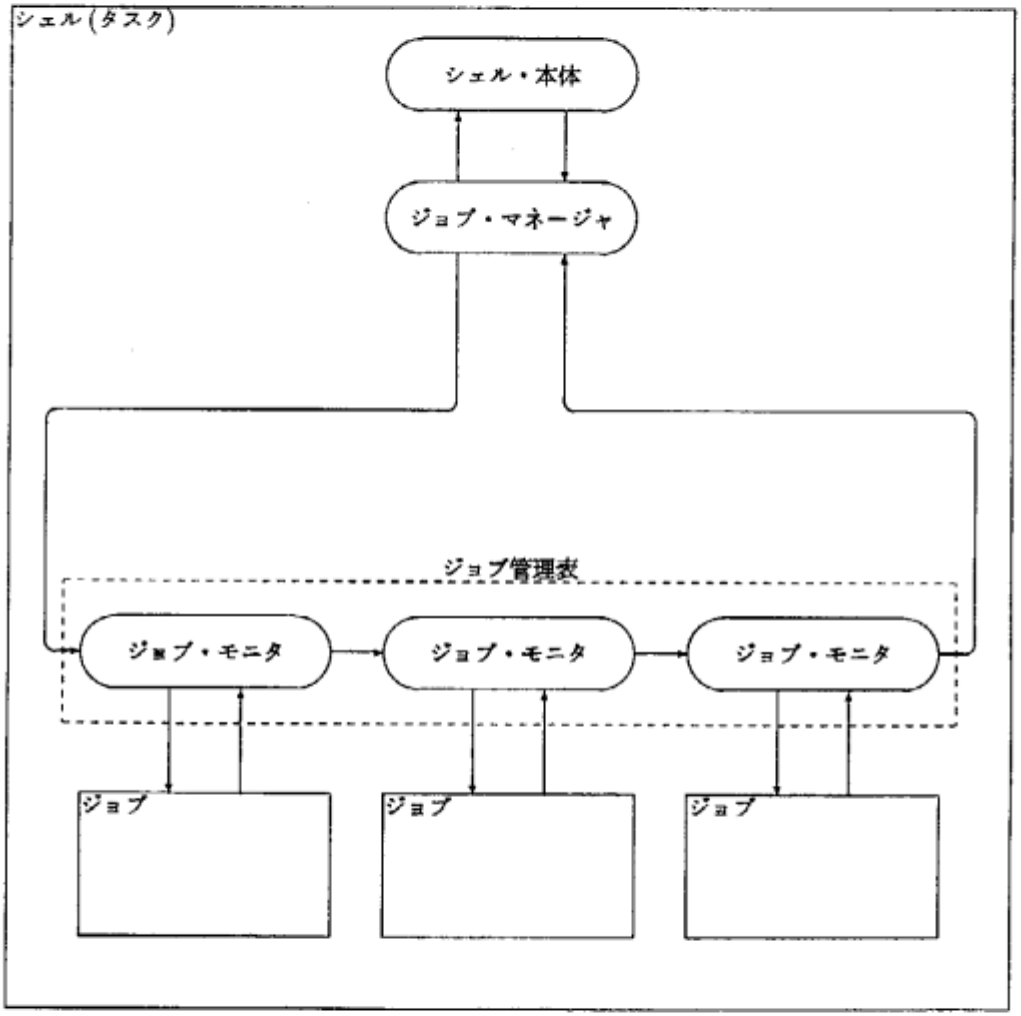

図 5.11: ジョブ・プールの構成

またパイプは、処理速度の異なる二つのタスク間をバッファを介して接続することで、全体としての処理効率を向上させる役割を持つ。処理効率を向上させるためには、パイプ両端のタスク内にある入出力バッファのバッファ・サイズとパイプのバッファ・サイズとの関係が重要なファクタとなる。

さらにパイプは、入力側タスクと出力側タスクとの間のプロトコルの整合をとる役割も持つ。

パイプに要求される機能としては、以下のようなものがあげられる。

1. バッファ・フル検出

パイプの持つバッファは、有限長である。したがって、パイプへメッセージを入力する側(生産者)の能力が、パイプからメッセージを取り出す側(消費者)の能力を上回っている場合には、いずれバッファがあふれる。そのような場合には、生産者を一時的に停止させるために、バッファ・フルの状態を検出する必要がある。

実際には、バッファ・フルの状態を「次入力メッセージ・サイズがパイプ内の空き領域サイズ以上である場合」と定義する。

ただし前提条件として、パイプのバッファ・サイズがパイプ入力側の タスク内にある入出力バッファの持つバッファ・サイズよりも大きいことが必要である。また、ここでメッセージのサイズとは、メッセージの個数のことではなく、メッセージ内に含まれるデータのキャラクタ・サイズを表す。

2. バッファ空き検出

1のバッファ・フル状態検出の結果として停止させられた生産者を再開するために、再開のタイミングを検出する必要がある。

実際には、以下のような場合をそのタイミングとして検討している。

(a) パイプ内バッファの空き領域サイズが次のメッセージのサイズと較べて十分に大きくなった。例えば、バッファ領域の半分が空いた。

(b) パイプ内バッファの空き領域サイズが次のメッセージのサイズと同じか、それよりも大きくなった。

2aの特徴は、再開のタイミングは2bの場合に較べ遅れる場合があるが、生産者を停止させる回数が2bに較べて一般に少なくなる。一方2bの特徴は、これとは逆で、再開のタイミングは早いが、生産者を停止させる回数が増加する可能性がある。

上記2a 2bのいずれを採用するかは、入出力バッファのサイズ、パイプのバッファ・サイズ、タスクの停止・再開のためのコスト等に依存して決定する必要がある。

3. ブロークン・パイプ検出

二つのタスクがパイプを介して通信を行う際に、パイプによる結合になんらかの異常が生じ、通信の継続が不可能となった場合には、それを検出し、適切な処置を施す必要がある。この異常状態をブロークン・パイプと呼び、以下の場合を検出せねばならない。

(a) 消費者側の異常

- パイプ内にメッセージがあるのに消費者が死んだ。
- 生産者がメッセージを入力したのに消費者側がクローズされている。

(b) 生産者側の異常

- 生産者が異常終了した。

ここで 3aの異常を検出するために パイプ自身がクローズされたか否かを記憶している必要がある。 また 3bの異常は、 タスクの異常終了であるためタスクの例外ハンドラが 検出すべきものであり、パイプでは特に何も行わない。

4. その他の機能その他にユーザに提供する機能としては、次のようなものが考えられる。

   - 空き領域のサイズ問い合わせ
   - 入力可否の問い合わせ

5.5.5 有限長出力バッファ付きタスク

シェルが、タスクの標準入出力間をパイプで結合しジョブを作成する際には、タスクとパイプを各々独立に作成して後から結合するのではなく、あらかじめタスクの標準出力にパイプを結合させた、シェル定義の専用タスクを生成し、それを結合してジョブを形成する。この専用タスクを標準出力に有限長バッファ (パイプ) を持つタスクという意味で、有限長出力バッファ付きタスクと呼ぶ。シェルは、この有限長出力バッファ付きタスクを生成するためのユーティリティを提供する。

タスクとパイプの組合せを有限長出力バッファ付きタスクとして提供することの利点は、パイプのバッファ・フル検出、バッファ空き検出、ブロークン・パイプ検出の各時点で行う、パイプへのメッセージ入力側 (生産者) タスクの停止・再開に必要な機構の組み込みを容易に行える点にある。この機構は、パイプの状態を検査しているプロセスが直接に入力側のタスク (生産者) のコントロール・ストリームを取り扱うことで実現している。

有限長出力バッファ付きタスクを使用してジョブを構成した場合、最終段のタスクの出力は、(最終段の) 有限長出力バッファ (パイプ) に出力されるが、そのままでは、最終段のタスクの標準出力には出力されない。そのため、このバッファからデータを取り出してメッセージに変換のうえ、ジョブの出力として最終段のタスクの標準出力に送出するための出力の専用タスクをジョブに付加する必要がある。

また、ユーザ・タスクが入力を要求していない時点での先行入力を実現するため、第一段のタスクの前には、入力の専用タスクを付加する。この入力の専用タスクは、有限長出力バッファ付きタスクで実現される。

この入力および出力の専用タスクを利用すると、ジョブの停止・再開は、これら専用タスクのみを停止・再開することで代用できる。

5.5.6 標準入出力の獲得方法

シェルが起動するタスクには、あらかじめ標準入出力用の OS 資源を指定しておく必要がある。しかし、シェルがファイルやウインドウなどを獲得 (オープン) して、そのストリームをタスクに渡す方式をとると、ファイルやウインドウは、それらを生成したタスクであるシェルの資源として登録されてしまう。さらにその場合、資源の入出力バッファや保護フィルタがシェルと起動したタスクとの両方に二重にできてしまい、無駄が生じる。

以上のような状況を避けるため、ファイルやウインドウの獲得は、シェルが行うのではなく、ファイルやウインドウの識別子 (名称等) および獲得するためのゴールをユーザ・ゴールの引数として渡し、ユーザ・タスク内で実際にそれらのストリームを得るという方法で行う。

5.5.7 ユーザ・インタフェース

シェルは、ジョブやタスク管理のユーザ・インタフェースとして、次のようなコマンドを提供する。ここに示したコマンドは、ユーザが PIMOS 上でプログラムを実行するために最低限度必要と思われるもののみである。これ以外のコマンドについては、順次追加する。

1. ジョブ管理用のコマンド

   - STOP(job_id) : job_id で指定されたジョブを停止する。
   - FORE(job_id) : job_id で指定されたジョブをフォアで再開する。
   - BACK(job_id) : job_id で指定されたジョブをバックで再開する。
   - KILL(job_id) : job_id で指定されたジョブを強制終了する。
   - STATUS : そのシェルが起動した全ジョブの情報を表示する。

2. タスク管理用のコマンド

   - TS(task_id) : タスクの情報を表示する。
   - STOP(task_id) : task_id で指定されたタスクを停止する。
   - RESTART(task_id) : task_id で指定されたタスクを再開する。
   - KILL(task_id) : task_id で指定されたタスクを強制終了する。

第 6 章

入出力

6.1 概要

マルチ PSI は IO デバイスを持っていないため、FEP(Front End Processor) である PSI が入出力を行う。実際の IO デバイスは SIMPOS のデバイスに相当する。したがって PIMOS のデバイス・ハンドラは IO デバイスの制御は行わず、資源管理やユーザへのサービス提供を目的とする。その意味では、FEP は実質的なデバイス・ドライバである。 FEP と本体は KL1 ネットワークで結ばれる。マルチ PSI は PE64 台で構成され、最大 4 台の FEP を接続することができる。図 6.1にマルチ PSI の構成例を示す。

本体の PE から見た FEP は本体の他の PE と全く同じである。一般にファイルやウインドウなどの入出力は逐次性を持つため、本体からの入出力要求はストリームを通じて通信が行われる。しかし、インタラクティブなマン・マシン・インターフェイスに必須のユーザ割込や入出力処理の中断といった非同期的な処理も必要である。ここで、「非同期」とはストリームの逐次性で実現できないものを指す。

ここでは、 PIMOS の立ち上げるときの本体と FEP の関係、本体と FEP の定常的な通信、非定常的な通信、さらに具体的な通信メッセージ内容について記す。

6.2 本体との通信方式

6.2.1 世の中の始まり

PIMOS の立ち上げは CSP(Console System Processor) が行う。 CSP、 FEP および本体の関係と、それぞれの間を結ぶ通信路を図 6.2に示す。 CSP と同じ PSI にある FEP は、 SIMPOS のストリームにより接続される。 CSP と違う PSI にある FEP は直接の通信路はない (オフライン)。

以下に、 PIMOS 立ち上げの様子を順を追って説明する。

1. IPL のロード

   PIMOS の IPL は、メンテナンス・バスを通じて行われる。すなわち、CSP が本体の電源を投入して初期化完了後、全 PE に対し IPL コードをロードし，ネットワークのパス・テーブルを設定する。 (図 6.3)

2. IPL の開始直前

   1 が完了すると、各 PE に対し IPL に必要なコードがロードされ、ネットワークの初期化が終了する。 FEP は CSP の (SIMPOS の) 子プロセスとして生成される。この時点で、すでに本体と FEP の間にはストリームが張られている。このストリームは FEP に対し様々な IO スト

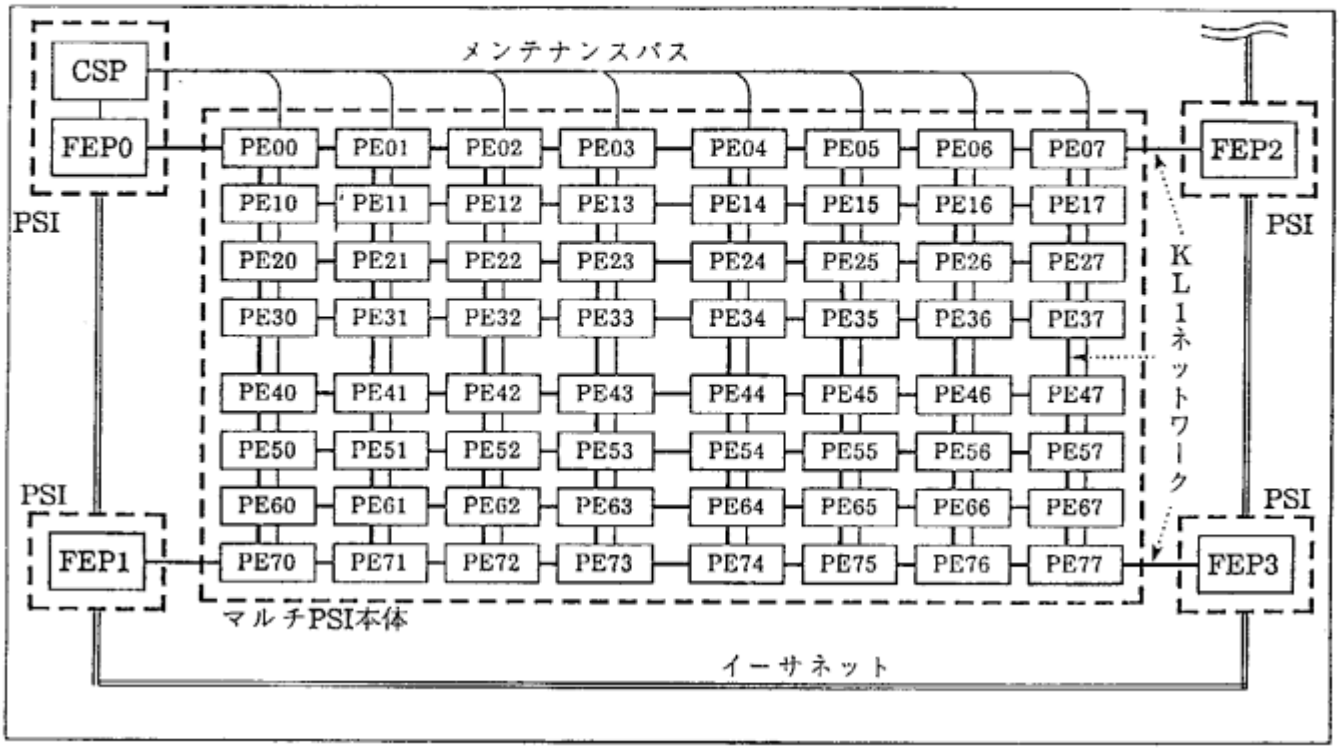

図 6.1: マルチ PSI-V2 の構成

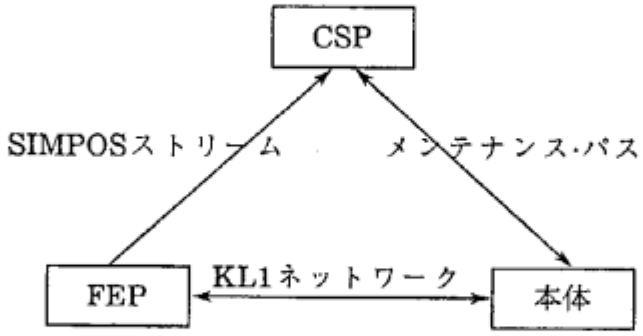

図 6.2: 本体、FEP、CSP を結ぶ通信路

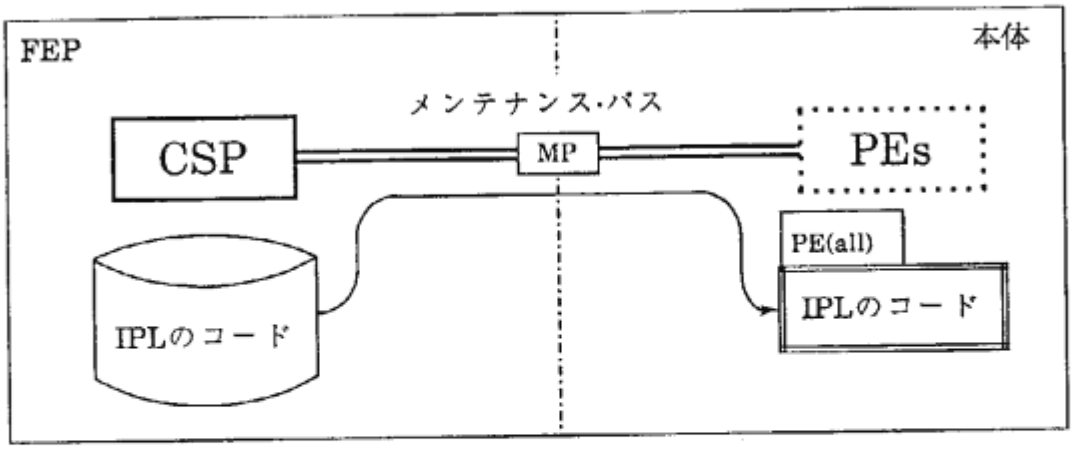

図 6.3: IPL のロード

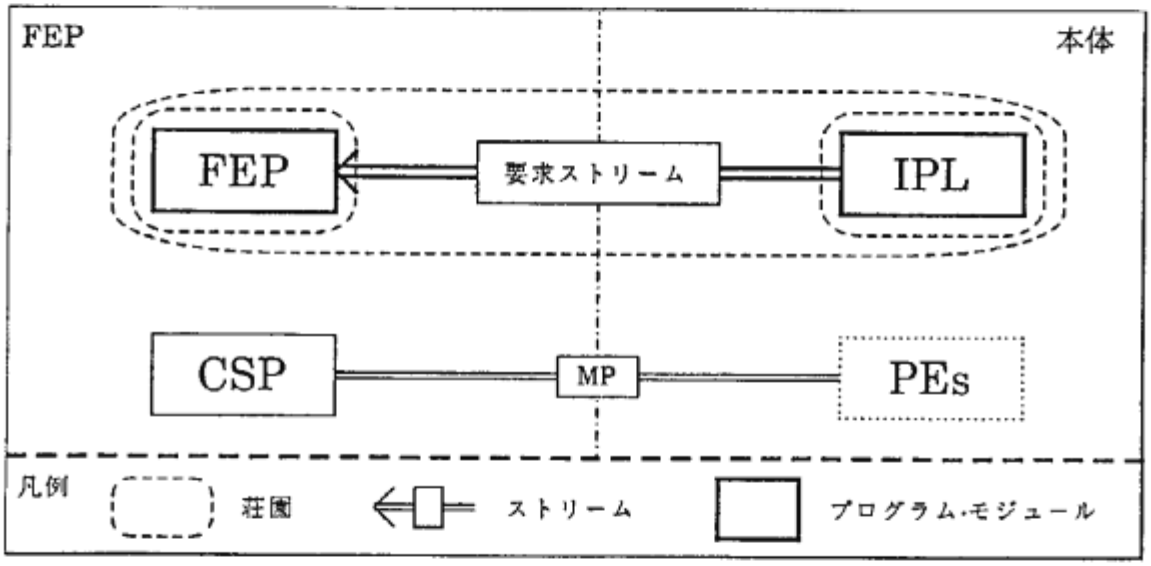

図 6.4: IPL の開始直前

リームを生成するためのもので、「要求ストリーム」と呼ばれる。 荘園は図 6.4に示されるように、 FEP と本体にひとつずつとそれらの親荘園の計 3 個存在することになる。 FEP は荘園の機能をサポートしないので、 FEP の荘園は仮想的なものである。

3. IPL-PIMOS のロード

2 でロードされた IPL コードを使って、 FEP 経由で PIMOS をロードする。 IPL デバイスはファイル・デバイスの一種であり、この内容は PIMOS のコードである。この IPL デバイスに必要なストリームは要求ストリームを経由して作られる。本体の IPL がこのファイルを読み込み、 PIMOS が実行可能な状態にする。図 6.5は PIMOS のロードの様子を、図 6.6 には IPL が完了した状態を示す。

4. PIMOS の初期化

IPL が完了すると PIMOS の初期化が開始される。 PIMOS の初期化では PIMOS に必要な

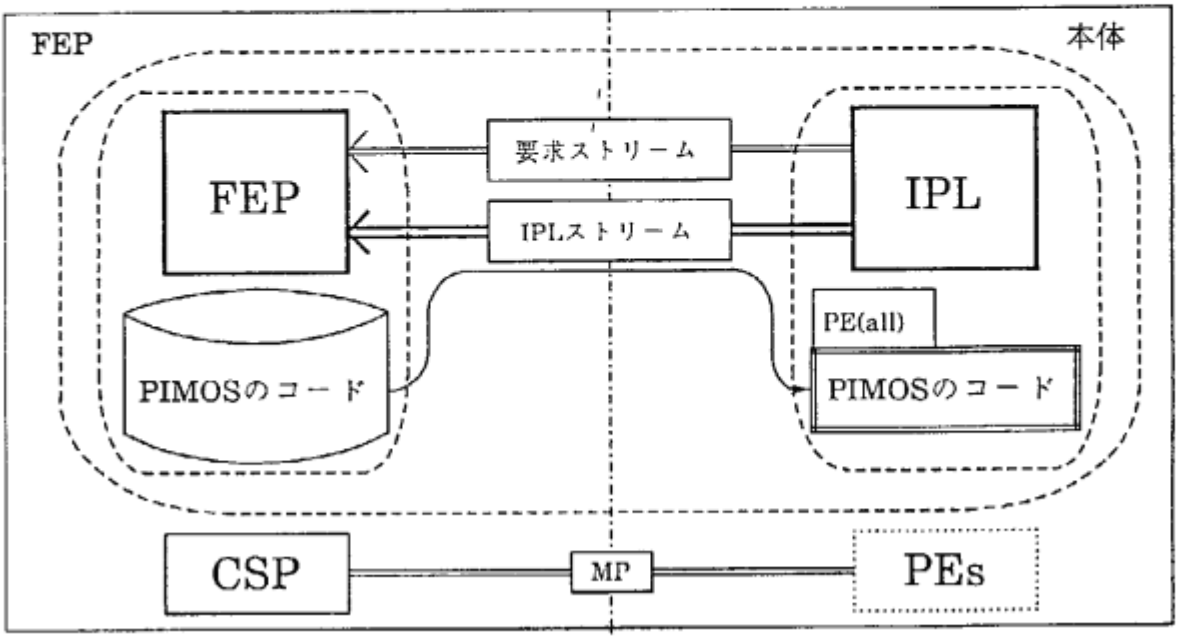

図 6.5: PIMOS のロード

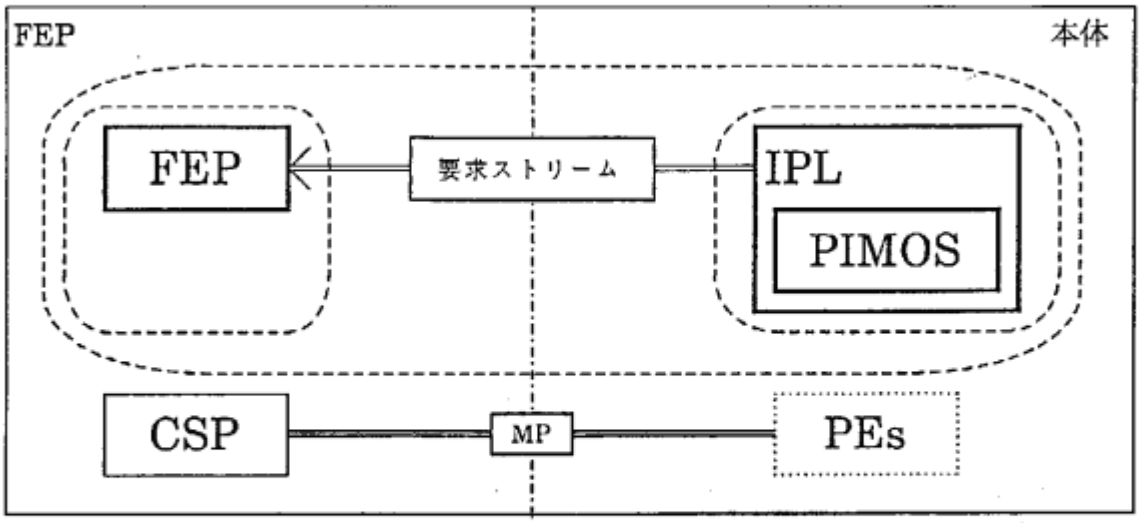

図 6.6: PIMOS の開始直前

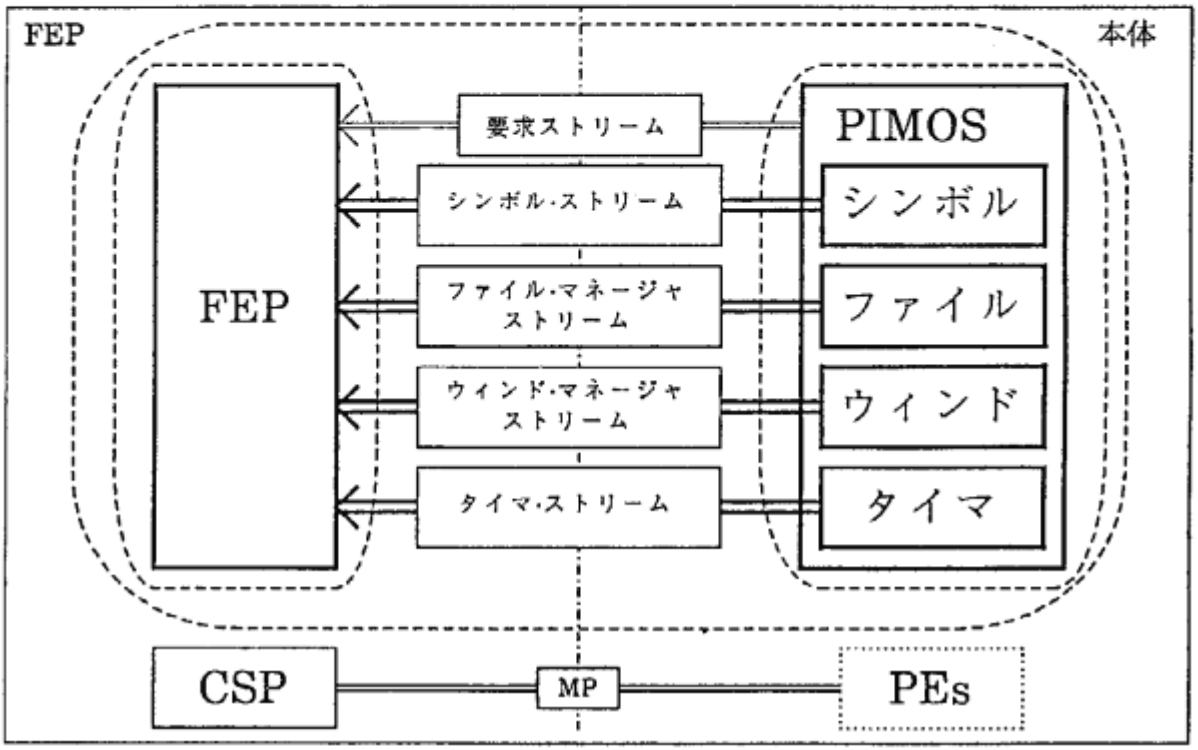

図 6.7: デバイス・ストリームの生成

IO デバイスを管理するデバイス・マネージャ・ストリームが生成される。個々のデバイス・ストリームはこのマネージャ・ストリームにより生成される。(図 6.7)

PIMOS のシンボル管理に必要なデータは、FEP のシンボル・デバイスを使って PIMOS に取り込まれる。シンボル・デバイスは内容がシンボル・テーブル (アトム番号と表示文字列の対応) であるようなファイルの一種である。 PIMOS の初期化が完了した状態を図 6.8に示す。

6.2.2 通信方式

前述したように、本体と FEP の通信はストリームを通じて行われる。 IO メッセージは本体から FEP に送られ、結果がメッセージ中の変数を具体化することで本体に返される。メッセージは KL1 のベクタであり、その第 1 要素はメッセージ ID、最後の要素は次のメッセージへのポインタである。本体-FEP 間のストリームは、処理を簡略化するために、 GHC でいうところのリスト構造を用いていない。

本体から FEP に送られるメッセージに出現するデータ型は、整数、ストリングおよびベクタであり、FEP から本体に送られるデータ型は整数またはストリングに制限されている。アトムを通信に用いないのは、シンボル管理を PIMOS で一元管理するためである。

本体 -FEP 間では、通信に必要な機構は KL1 のユニフィケーションである。したがって、本体では他の

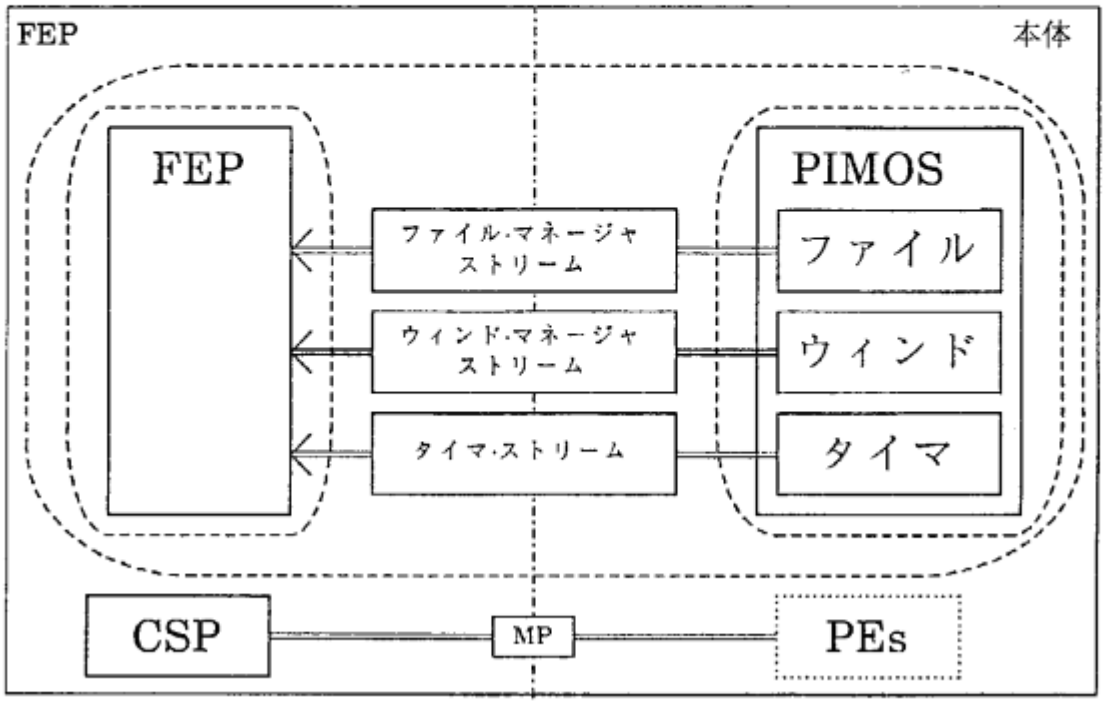

図 6.8: PIMOS の初期化完了

PE との通信とほとんど同じコーディングで済む。本体からユニフィケーションを行うには、FEP に KL1 データ・オブジェクトを作る必要が生じる。FEP は基本的に KL0 の世界なので、ESP のクラスとして KL1 データ・オブジェクトを作ることは可能であるが、処理が複雑になる。したがって、ユニフィケーションの方向を FEP から本体に限定し、必要な変数、ベクタなどは全て本体上にあるものとする。この取り決めにより、FEP が扱う KL1 ネットワーク・メッセージの種類も限定される。

FEP では KL1 の待ち合わせ処理およびストリームの機構を模擬する必要がある。この処理は KL1 の変数やストリームに相当するものを ESP のクラスで実現している。

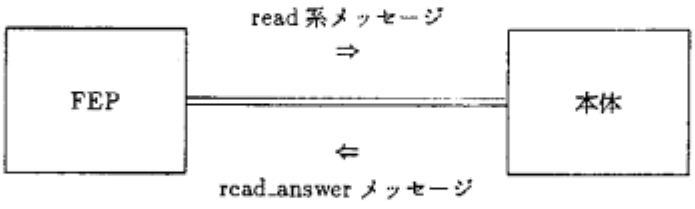

図 6.9: 本体側の値の参照

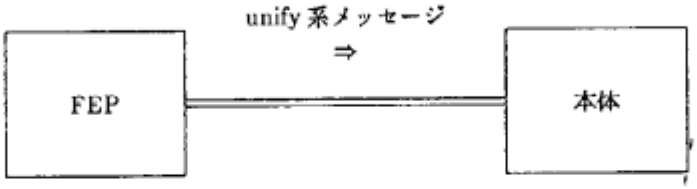

図 6.10: 本体に値を返す場合

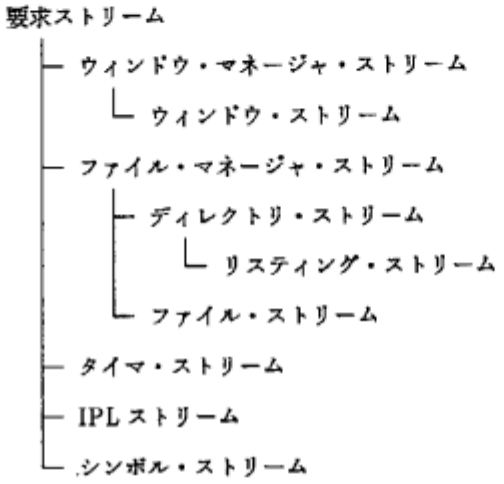

図 6.11: 階層ストリームの構成

6.3 提供機能

6.3.1 概要

FEP で用意するデバイスは、要求ストリームをルートとする樹状構成になっている。要求ストリームは、前述したように FEP と本体を最初に結ぶもので、全てのデバイス・ストリームはこの要求ストリームから生成される。ファイルやウィンドウなどのデバイスは、要求ストリームを通じて生成された、それぞれのマネージャ・ストリームから生成される。場合によっては、デバイス・ストリームからもデバイス・ストリームが生成される。一般的にいえば、IO ストリームはコマンドを受け付け、コマンドによっては、その結果として下位のストリームを生成することになる。このように、ストリームの生成を階層にすることにより、ストリームのモジュール性を高めることができる。注意を要するのは，この階層構成はストリームの生成に関するものであり、生成されてしまったストリームの関係をしめすものではない、ということである。例えば、ファイル・マネージャ・ストリームを経由してファイル・ストリームを生成し、その直後にマネージャ・ストリームを閉じても、生成されたファイル・ストリームは開いたままである。

6.3.2 異常終了と割込み

従来型のOSでは、ユーザからの割込を受け付ける。例えば、UNIXのシェルではControl-Cを端末から入力することにより，実行中のフォアグランド・プロセスを終了させることができる。この仕組みは、プログラムが無限ループに陥った等の状態から抜け出すのに、必要な機能である。このような「ユーザ割込」は、

1. ユーザがOSの注意をひきつけ、

2. プログラムを強制終了させる。

ことを意味する。FEPから見れば、2は入出力処理の強制終了を意味する。

以上のような処理をPIMOSで実現するのに、単にストリームで入出力要求を本体からFEPに送る、という枠組みだけで対処するのは難しい。

例えば、ユーザ割込を受けつけるために、全てのメッセージに割込があったことを示す変数を付け加えることも考えられる。しかし、IOストリーム上にメッセージがないときには、ユーザ割込が効かなくなってしまう。また、入出力処理の中止を指示するメッセージをIOストリームに流すことにすると、応答が悪くなってしまう。

以上の結果から、ストリームの枠組みの外に上記のような非同期処理のための仕組みが必要である。基本的な戦略は、FEPと本体で共有する変数を持ち、この変数を具体化することにより非同期処理を取り扱うのである。このように、FEPと本体で共有されストリームの基本的な使用方法と異なる変数を、ストリームと区別する意味で「ライン」と呼ぶ。ラインはストリームの「外」にあるので、ストリーム中のメッセージを飛び越えて通信することができる。

ラインは、共有される変数を含む特別なメッセージをストリームで送ることで生成される。このことを「ラインを張る」という。一般のメッセージに現れる変数は、対応する入出力処理が完了した時点で具体化されるが、ラインに使われる変数はメッセージ処理完了後も具体化されるとはかぎらない。当然ながら、ラインは一度使われて(変数の具体化と同じ意味)しまうと、二度と使えなくなるので、再びラインを張る必要がある。

そこで、すべてのIOストリームに共通なふたつのメッセージを設ける。

- `reset(Abort,ATN,Status,Cdr)`
  FEPデバイスの内部状態を正常な状態にすると同時に、FEP-本体間にアボート・ラインを張る。

- `next_attention(ATN,Status,Cdr)`
  FEP-本体間にアテンション・ラインを張る。

ここで、AbortはFEPに対し入出力処理の中断を指示するときに本体側で具体化する変数であり、ATNはユーザ割込がFEPで発生したときにFEP側で具体化する変数である。以下、前者をアボート・ライン、後者をアテンション・ラインと呼ぶ。これらのメッセージを受け取る直前までに使われなかったラインは捨てられる。Statusはメッセージの処理が正常に終了したかどうかを本体に返す変数であり、Cdrは次のメッセージへのポインタである。これら、StatusおよびCdrは全てのメッセージに存在する。

以下に、IOストリームとこれからのラインの使われ方を典型的なユーザ割込の例を順を追って示す。

(1) ユーザ割込
FEPプロセスがウィンドウからのユーザ割込を検出すると、その旨をアテンション・ラインを通じて本体に知らせる。本体では、アテンション・ハンドラがアテンション・ラインの変数が具体化されるのを待っている。

(2) 入出力の中断
ユーザ割込の結果として、その時点での入力または出力処理を中断する必要を生じた場合は、アボート・

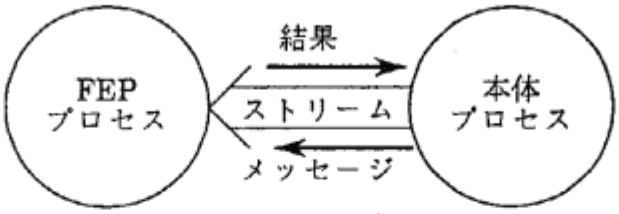

図 6.12: IO ストリーム

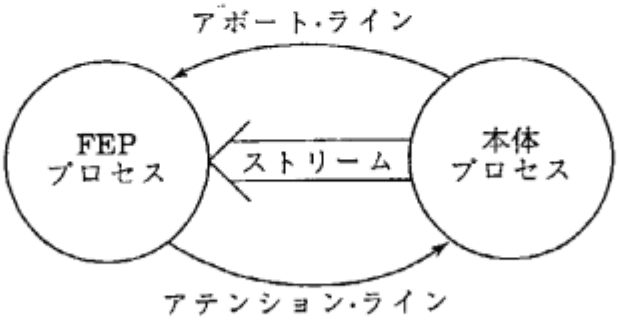

図 6.13: ストリームとラインの関係

ラインを通じて FEP に通知する。 FEP ではその時点から次のresetメッセージ (後述) までの IO メッセージに対して、実際の入出力は行わずに，異常終了をStatusに返す。

(3) 入出力の再開

ユーザ割込処理が終わり、中断された入出力を再開するには、IO ストリームにresetメッセージを FEP に送る。 FEP はresetメッセージを受け取ることにより、正常な状態に戻る。 また、アボート・ライン、アテンション・ラインとも張り直される。

アボートからresetメッセージを受け取るまでの間、 IO メッセージを無効にするのは、 FEP- 本体間でアボートの同期をとるためである。

ユーザ割込みの結果として何も入出力に影響を与えない場合は、上記の (2)(3) の代わりに以下の (2') を行う。

(2') 入出力を中断しない場合

次のユーザ割込を受け取るために、next_attentionメッセージを送り、アテンション・ラインを張り直す。もし、それ以上のユーザ割込を検知したくなければ、next_attention メッセージさえも送らずに、普通の IO メッセージを送ればよい。アテンション・ラインが張られていないときのユーザ割込みは FEP で無視される。

6.3.3 一般の入出力機能

本仕様は、 FEP 及び本体デバイス・ハンドラ間を渡る I/O 命令について詳述したものである。

以下で記述を行う I/O 命令の各引数は、常に本体側にその実体を持ち、本体・ FEP の間のネットワークを渡るデータは、 KL1 における整数、小正整数 (0 以上 255 以下)、ストリング、ベクタ、及びポインタ (exref,exval) である。

また、各命令の引数のうち、`status、?cdrという記述は、それぞれ

^ status FEP から本体へ I/O 命令終了状態を返すための KL1 変数
?cdr その命令がバインドされた KL1 ストリームの尾部

を表す。また、その命令の終了の状態について特に記述されていない場合には、statusは以下のような状態をとりうるものである。

$$\text{status} = \begin{cases} \texttt{fep\#normal} & \text{正常終了} \\ \texttt{fep\#abnormal} & \text{異常終了} \\ \texttt{fep\#aborted} & \text{アボートされた} \end{cases}$$

statusがfep#abonormal及びfep#abortedの時、FEP 側でバインドされるべき値は、すべてfep#nilになるものとする。但し、'fep#XXX'は、XXXを表すコード (整数) のマクロ表現である。

また、本仕様では、各命令の引数についているマークによりその引数がどこでバインドされるものかが分かるようになっている。マークの種類には、以下のものがある。

なし： 本体側でバインドされるべきもの。タイミングにより、ポインタのみの時がある。ポインタのみの時には、一旦readしてからでないと、この引数を含む I/O 命令は実行できない。

?： 将来、本体側でバインドされるもの (本体側でバインドされるストリーム及びライン)。FEP 側では特に考慮せず、この変数に対して readメッセージを発行する。

^： FEP 側でバインドされる変数。この変数の値が分かったら、FEP から本体へのunifyメッセージを発行する。

1. メタ・デバイス

   PIMOS における世の中の始まりには、本体から FEP に向かって Request ストリームが張られている。この Request ストリームを通じて本体から発行される命令を実行するのが FEP 上にあるメタ・デバイスである。

   このデバイスは、以下に示す命令を受け付ける。

   - window(?window, ^status, ?cdr)
     ウインドウ・マネージャ・デバイスを生成し、そこへのストリームを返す。
   - file(?file, ^status, ?cdr)
     ファイル・マネージャ・デバイスを生成し、そこへのストリームを返す。
   - timer(?timer, ^status, ?cdr)
     タイマを生成し、そこへのストリームを返す。
   - ipl(?ipl, ^status, ?cdr)
     ipl を行うためのストリームを生成する。ipl デバイスはファイルであり、ipl ストリームを通じてコードをロードすることができる。
   - symbol(?symbol, ^status, ?cdr)
     シンボル・テーブルはファイルであり、このストリームを通じて、シンボル・テーブルをロードすることができる。

2. I/O マネージャ・デバイス

   Request ストリームを用いてメタ・デバイスに生成命令を発行することにより、本体から FEP に向かってデバイス・ストリームを張ることができる。このデバイス・ストリームを通じて発行される命令を実行するのが、FEP 上の I/O マネージャ・デバイスである。

このデバイスは以下に示す命令を受け付ける。

(a) ウインドウ・マネージャ・デバイス

- create(?window, ^status, ?cdr)
  ウインドウ・オブジェクトを生成し、そこへのストリームを返す。
- create_with_buffer(buffer_name, ?window, ^status, ?cdr)
  ウインドウ・オブジェクトを生成し、そこへのストリームを返す。但し、ウ インドウ・オブジェクト生成時に割り当てられる入出力先のバッファを指定 することができる。
- get_max_size(x, y, pathname, ^characters, ^lines, ^status, ?cdr)
  位置(x,y)を左上頂点とするウインドウの最大サイズ (文字数×行数) を返す。 但し、一文字のサイズは、pathnameで指定される フォント・ファイル中のフォントのサイズを基準としている。

(b) ファイル・マネージャ・デバイス

- read_open(pathname, ?input_file, ^status, ?cdr)
  pathnameで指定されるファイルを入力用にオープンし、 そこへのストリームを返す。
- write_open(pathname, ?output_file, ^status, ?cdr)
  pathnameで指定されるファイルを作成オープン (すなわち生成) し、 そこへのトリームを返す (詳細については表 6.1に示す)。
- append_open(pathname, ?output_file, ^status, ?cdr)
  pathnameで指定されるファイルを追加オープンし、 そこへのストリームを返す (詳細については表 6.1に示す)。
- directory(pathname, ?directory, ^status, ?cdr)
  pathnameで指定されるディレクトリ・オブジェクトを生成し、 そこへのストリームを返す。

(c) タイマ

- get_count(^count, ^status, ?cdr)
  午前0時0分から現在までのミリ秒数をcountに返す。
- on_at(count, ^now, ^status, ?cdr)
  指定された時刻になると、nowをfep#wake_upにバインドする。
- on_after(count, ^now, ^status, ?cdr)
  指定された時間が経過すると、 nowをfep#wake_upにバインドする。

3. I/O デバイス

I/O マネージャ・デバイスにデバイス・ストリームを用いて生成命令を発行することにより、本体からFEP に向かってI/O ストリームを張ることができる。このI/O ストリームを通じて発行される命令を実行するのが、 FEP 上の各I/O デバイスである。I/O デバイスとしては、現在ウインドウとファイルがある。

これらのデバイスは以下に示す命令を受け付ける。

(a) ウインドウ・コマンド

i. ウインドウ・I/O コマンド

表 6.1: #pimos_binary_file出力用作成 / 追加オープンの動作

			write モード	appendモード
versionなしでファイル名を指定	指定されたファイル名のファイルが存在		versionを+1してファイルを作成 マーカはファイルの先頭	指定されたファイルをオープン マーカはファイルの末尾
	指定されたファイル名は存在するが、最新versionはdeleted		versionを+1してファイルを作成 マーカはファイルの先頭	deletedでない最大versionをオープン マーカはファイルの末尾
	指定されたファイル名が存在しない		version=1のファイルを作成 マーカはファイルの先頭	version=1のファイルを作成 マーカはファイルの先頭
versionつきでファイル名を指定	指定された名前のファイルが存在	指定versionのファイルが存在	指定versionファイルを作成 ファイルの中身はクリア、マーカは先頭	指定されたファイルをオープン マーカはファイルの末尾
		指定versionのファイルがdeleted	指定versionファイルを作成 ファイルの中身はクリア、マーカは先頭	指定されたファイルを作成 ファイルの中身はクリア、マーカは先頭
		指定versionのファイルが存在しない	指定versionのファイルを作成 マーカは先頭	指定されたファイルを作成 マーカは先頭
	指定された名前のファイルが存在しない		指定versionのファイルを作成 マーカは先頭	指定されたファイルを作成 マーカは先頭

#pimos_binary_fileは、SIMPOSのクラスとして用意されるPIMOS 用の標準入出力ファイル・クラスである。

- getl(^line, ^status, ?cdr)
  ウインドウから一行分 (<LF>まで) 入力を行い、lineを返す。 一行分がKL1 ネットワークのバッファ・サイズより大きい時、lineにはバッファ・サイズ分を返し、その際statusはfep#continueとなる。
- putb(string, ^status, ?cdr)
  ウインドウにstringを出力する。
- flush(^status, ?cdr)
  現在、バッファに溜まっているものをすべて表示する。
- beep(^status, ?cdr)
  ウインドウでベルを鳴らす。

ii. ウインドウ・属性コマンド

- set_inside_size(characters, lines, ^status, ?cdr)
  ウインドウのサイズをcharacters(一行の文字数)× lines(行数) に設定する。
- set_size(^status, ?cdr)
  ウインドウのサイズをマウスからの入力で設定する。
- set_position(x, y, ^status, ?cdr)
  ウインドウの左上頂点を位置(x,y)に設定する。
- set_position_by_mouse(^status, ?cdr)
  ウインドウの位置をマウスからの入力で設定する。
- set_title(string, ^status, ?cdr)
  ウインドウのタイトルをstringにする。
- reshape(x, y, characters, lines, ^status, ?cdr)
  ウインドウのサイズをcharacters(一行の文字数)× lines(行数) に、ウインドウの位置を(x,y)に変更する。
- reshape_by_mouse(^status, ?cdr)
  ウインドウのサイズ及び位置をマウスからの入力に合わせて変更する。
- set_font(pathname, ^status, ?cdr)
  ウインドウのフォントをpathnameで指定される フォント・ファイルのフォントに設定する。
- select_buffer(buffer_name, ^status, ?cdr)
  ウインドウの入出力先バッファを、 buffer_nameで指定されるバッファに切り換える。
- buffer_name(^buffer_name, ^status, ?cdr)
  ウインドウの入出力先バッファの名前をbuffer_nameに返す。
- activate(^status, ?cdr)
  ウインドウをスクリーン上に表示する。但し、表示されていなくても、createされていれば出力は可能である。
- deactivate(^status, ?cdr)
  ウインドウの表示をやめる (再度activate可能である)。
- show(^status, ?cdr)
  ウインドウを他のウインドウによって隠されていない状態にする。
- hide(^status, ?cdr)
  ウインドウを他のウインドウ中の最下層に移動する。

- clear(^status, ?cdr)
  ウインドウ中に表示されている内容を消去する。
- get_inside_size(^characters, ^lines, ^status, ?cdr)
  ウインドウの表示可能部分 (タイトルは除く) のサイズ (文字数×行数) を返す。
- get_position(^x, ^y, ^status, ?cdr)
  ウインドウの左上頂点の位置(x,y)(ドット単位) を返す。
- get_title(^string, ^status, ?cdr)
  ウインドウのタイトルをstringに返す。
- get_font(^pathname, ^status, ?cdr)
  ウインドウのフォントを、フォント・ファイルのパス名で返す。

(b) ファイル・コマンド

i. ファイル・I/O コマンド

- getb(size, ^string, ^status, ?cdr)
  ファイルから、サイズ長のストリングを取り出し、stringに返す。 ファイルから取り出せるデータがサイズ長に満たない時は、 有効な長さ分だけを返す。この時のstatusは、fep#normalである。
  取り出せるデータが無くなった時は、 stringにはfep#nilが、 statusにはfep#eodが返る。 既に取り出せるデータが無くなった後のgetb命令に対しては、 statusはfep#abnormalとなる (stringの内容は、fep#nil)。
- putb(string, ^status, ?cdr)
  ファイルにstringを出力する。
- flush(^status, ?cdr)
  現在、バッファに溜まっているものをすべてファイルに吐き出す。この命令 は主に出力が終了しているかどうかの確認のために用いられる。

ii. ファイル・属性コマンド

- end_of_file(^eof, ^status, ?cdr)
  ファイルの入出力開始位置を示すマーカがファイルの最後を指していたら eofにfep#yesを、それ以外の場所を指していたらfep#noを返す。
- pathname(^pathname, ^status, ?cdr)
  ファイルの完全パス名をpathnameに返す。

iii. ディレクトリ・属性コマンド

- pathname(^pathname, ^status, ?cdr)
  ディレクトリの完全パス名をpathnameに返す。
- listing(wildcard, ?file_names, ^status, ?cdr)
  wildcardで指定されるファイルの完全パス名の リスト[1] を返す。
- delete(wildcard, ^status, ?cdr)
  wildcardで指定されるファイルをディレクトリから削除する。

[1] リストは本体側でバインドされるストリームの形で表現される。
?file_names = next_filename(^ file_name, ^ status, ?cdr)
file_name n 個目のリストの要素
cdr n+1 個目以降を送るためのストリーム用変数
リストの要素の個数が k 個の時、k 個目を送信した時のstatusは fep#normal、 k+1 個目を要求するとfile_nameにはfep#nilが、statusにはfep#eodが返る。 k+2 個目の要素に対しては、statusはfep#abnormalとなる (file_nameの値はfep#nil)。

- undelete(wildcard, ^status, ?cdr)
  wildcardで指定されるファイルで、 deleteにより削除されているファイルをディレクトリに回復する。
- purge(wildcard, ^status, ?cdr)
  wildcardで指定されるファイルのうち、最大バージョンのファイル以外のファイルをディレクトリから抹消する。 また、削除中のファイルも抹消される。 抹消されたファイルは回復することができない。
- deleted(wildcard, ?file_names, ^status, ?cdr)
  wildcardで指定されるファイルのうち、 削除中のファイルの名前のリスト[1]を返す。
- expunge(^status, ?cdr)
  deleteにより削除中のファイルをすべて抹消する。

4. すべてのI/Oストリームに共通な命令

- reset(?abort, ^attention, ^status, ?cdr)
  本体から各デバイスに対して、I/O命令をアボートするためのアボート・ライン(abort)と、FEPからPIMOSの注意を喚起する為の アテンション命令を送るためのアテンション・ライン(attention)を張る。
  すべてのデバイスは、生成時にはこれらのラインが張られていない状態(アボート状態)である為、この命令はI/Oストリーム(及びI/Oデバイス)が生成された直後(属性コマンドの発行以前)に出される必要がある。これらのラインが張られた後、 アボート・ラインにfep#abort(引数なしのアボート命令)が バインドされると、対応するデバイスはアボート状態になり、 再度のresetメッセージによりアボート・ラインと アテンション・ラインが張り直されるまで、 すべてのI/O命令をアボートする。この時、古いアテンション・ラインには FEP側からfep#nilがバインドされる。アテンション・ラインにFEP側からfep#control_cがバインドされると、本体側では、I/O命令をアボートしてその後 リセットするか、I/O命令をアボートせずに アテンション・ラインのみを張り直す。 abort命令とreset命令は必ずペアで発行されるか、 または、abort後reset命令なしにI/Oストリームがfep#nilで閉じられる。
- next_attention(^attention, ^status, ?cdr)
  アテンション・ラインのみを張り直す。 ユーザのアテンション入力はあったが、abortしたくない時に resetメッセージの代わりに用いられる。
- nil
  I/Oストリームを閉じ、I/Oデバイスを終了する。この時、アテンション・ラインにfep#nilをバインドして終了する。 本体側ではこの時、アボート・ラインを終了させるために以下の作業が必要である。
  (a) flushして入出力が終了したことを確認する
  (b) アボート・ラインにfep#nilをバインドする
  (c) ストリームを閉じる。

6.4 プログラムの構成

6.4.1 プロセス構成

FEP・I/Oにおけるプロセスの構成は、図 6.14に示すようになっている。

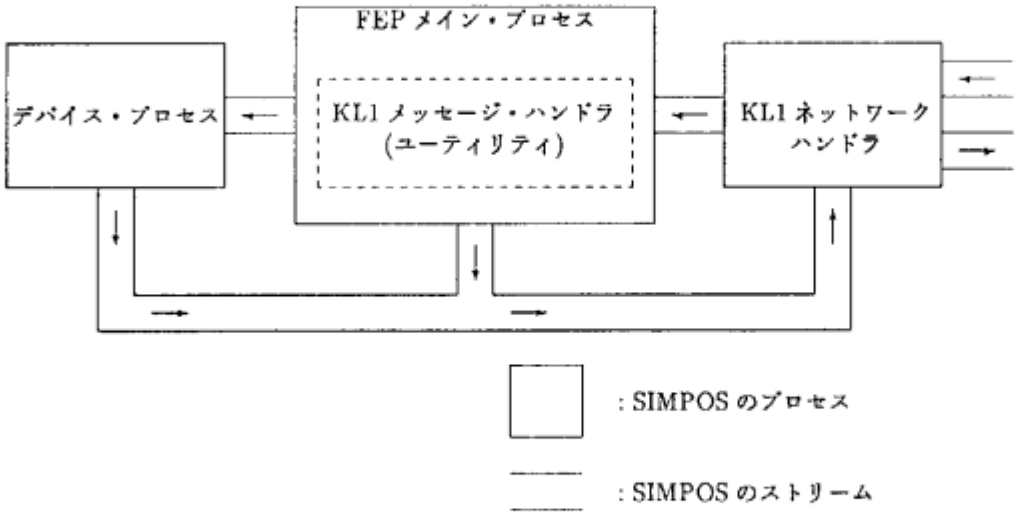

図 6.14: FEP のプロセス構成

FEP・I/O プログラムは、KL1 ネットワーク・ハンドラ、FEP メイン・プロセス、デバイス・プロセスの 3 つの SIMPOS プロセスから成る。各プロセスは、SIMPOS のストリームを通じて通信を行い、それぞれの環境で、独立に動作する。

1. KL1 ネットワーク・ハンドラ
   KL1 ネットワーク・ハンドラは、KL1 ネットワークからの割り込み (パケット受信) を受け取ると、KL1 ネットワークの read_buffer からパケットを取りだし、1 パケットずつに切り離して、ストリングにセットする。このストリングは、ストリームを通じて FEP メイン・プロセスに送られ、デコードされ、処理される。

2. FEP メイン・プロセス
   FEP メイン・プロセスは、KL1 のデータを KL0 に変換し、I/O の為に必要なデータが総て揃うまで、'待ち' を行うプロセスである。

   FEP メイン・プロセスは、1 パケット分のデータを受け取ると、KL1 メッセージ・ハンドラ (ユーティリティ) を呼び、KL1 データを KL0 データに変換する。また、変換した KL0 データ (I/O 命令) は、その I/O 命令を実行する I/O デバイスに送られる。

   各 I/O デバイスは、自分自身が実行できる I/O 命令の引数について、入出力の方向を知っており、入力が必要なデータに関しては、読み込み命令 (6.2本体との通信方式参照) を発し、そのデータを待つ。(ただし、この時 I/O デバイスは、FEP メイン・プロセス上で動作する事に注意する。)

3. デバイス・プロセス
   実際に I/O を行うためのデバイス・オブジェクト (すなわち SIMPOS のオブジェクト) が動作するプロセスである。ここで、実際にデバイスと呼ばれるのは、ウィンドウ・デバイスとファイル・デバイスの 2 種である。

   ウィンドウに関しては、SIMPOS の都合により 1 ウィンドウ 1 プロセスであり、ファイルに関しては、総てのファイルが 1 プロセス上で動作する事になる。1 ウィンドウ 1 プロセスにする事により、各ウィンドウは独立に動作する事が可能である。

デバイス・プロセスではSIMPOSのI/O機能が呼びだされ、実行される。出力系命令においては、本体側に返されるのは、実行が正常終了 / 異常終了 / アボートした事を示すステイタス・フラグのみである。入力系命令においてはステイタス・フラグに加えて、ユーザから入力されたintegerやstringを本体側に返す事になる。これらのデータの送出は、デバイス・プロセスにおいて、ユニファイ命令 (6.2本体との通信方式) を発する事によって行われる。

6.4.2 クラス構成

FEPプログラムは、主に以下のようなクラスから構成されている。

pimos_front_end FEPのメイン・プロセスに当たる。KL1ネットワーク・ハンドラからデータを受け取るための口 (KL0ストリーム) を持つ。

as_physical_device デバイス・プロセスに当たる。FEPメイン・プロセスからデータを受け取る口 (KL0ストリーム) を持つ。

as_pimos_device PIMOSのデバイスとしての基本機能を記述したクラス。基本的に、KL1・I/O命令をどこまで実行したかというコンティニュエイション・ポインタ持っており、I/O命令コード、入力引数、KL1ストリームの尾部、等の読み込みを行う。

as_abortable_device PIMOSのデバイスにI/O命令の実行をアボートする機能を提供する。
(as_pimos_deviceの一部である。)

kl1_message_handler KL1データとKL0データの変換を行う。

invocation_counter I/O命令を実行するためのデータが揃うまで待ちを行う為のカウンタ。入力引数個数をセット、データが到着する度にデクリメントし、0になるとI/O命令を実行するデバイスを起こす。

kl1_value_cell KL0の上でKL1データを表現するのに用いているオブジェクト。

external_reference 本体側へのポインタを表している。

booking_table 本体側へ問い合わせたデータを管理するための返信表。FEP上へのポインタは、FEPのPE番号とこの返信表のエントリのペアで表される。

以上のクラスを用いたFEP上のプログラム構造について図6.15に示す。

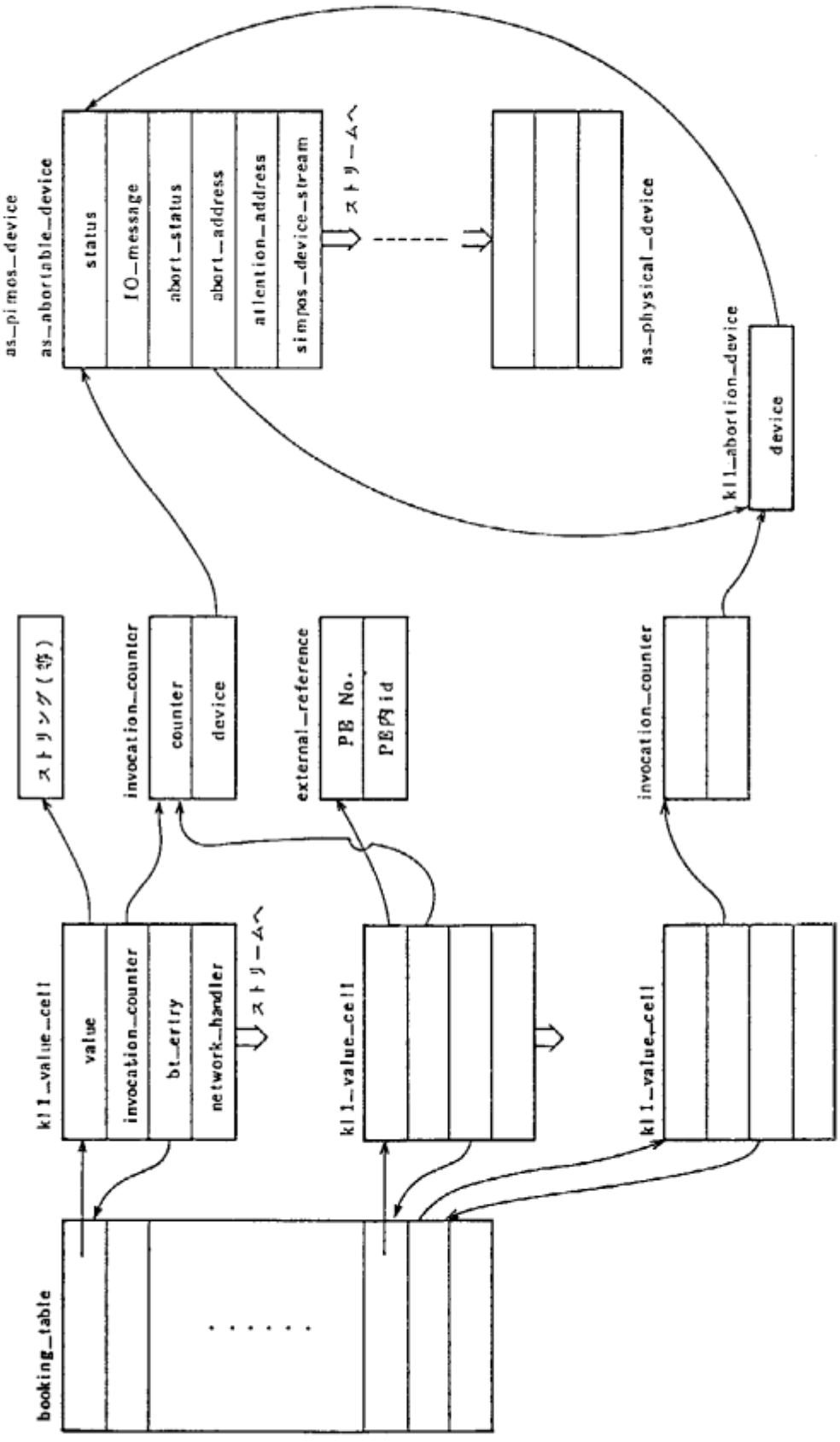

図 6.15: FEP のプログラム構成

第7章

KL1コンパイラ

7.1 概要

本章では、KL1-Cソースプログラムから KL1-B抽象機械語へのコンパイラについて説明する [10][2]。

KL1-Cは、フラットGHCに基づいてICOTで設計された並列論理型言語で、PIM、マルチPSIシステム上の言語として使用されるものである。また、PIMOS開発用のシステムとして逐次計算機上に実現されたPDSSシステム上の言語としても使用される。本コンパイラは、これらのシステム上で開発されたKL1プログラムを、KL1-Bと呼ばれる抽象機械語にコンパイルするプログラムで、マシン依存の部分はアセンブラによって解決するようにしている。

KL1コンパイラは、言語 Prologで記述したものと、KL1自身で記述したものの2つの版があり、前者は、PIM、マルチPSIの開発用ツールとして用いられ、後者は、PDSSシステム上のセルフコンパイラとして用いられる他、KL1自身で記述した大規模な応用プログラムの一つとしてKL1のプログラミングスタイルや、KL1の効率的な処理系開発のためのベンチマークプログラムとしても用いられる。以後本章では、Prologで記述したコンパイラをKL1/Prologコンパイラ、KL1自身で記述したものを、KL1/KL1コンパイラと呼ぶことがある。

本コンパイラは、フラットGHCの基本機能のほか、PIMOSサポート用の機能や、MRB-GC用の機能、クローズインデキシングの機能なども備えている。

本章は、以下の5つの節から構成されており、概要を以下にまとめる。

1. 7.2 内部構成
   KL1コンパイラの内部構成、特にクローズ単位のコンパイル方法ついて説明している。クローズ単位のコンパイルの方式は、Prologによるもの、KL1によるもの共、そのアルゴリズムは殆ど同じであり、以下の5つのモジュールから構成されている。
   (a) プリプロセッサ
   (b) クローズの正規化 (normalization)
   (c) クローズのコード生成
   (d) MRB-GC用命令の生成
   (e) レジスタ割付
2. 7.3 MRB管理
   KL1のデータをMRB管理することによって、インクリメンタルなGCや、アレイの効率的な実現が可能になる。またプロセッサ間にまたがったデータを効率良く管理することが可能になる。本節では、こ

れらの詳細については、言及せず、MRB用命令のレパートリと、コンパイラによるその生成アルゴリズムについて説明する。

また、コンパイラにはコンパイルオプションとして次の3種類の指定のしかたがあり、使用者によって自由に選択できる。

(a) MRB なしモード：MRB によるデータ管理を全く行わないモードである。

(b) MRB の更新のみのモード：MRB ビットのメインテナンスは行うが GC は行わないモードである。アレイの効率的な実現のみを目的とする場合には、このモード指定でよい。

(c) MRB-GC を行うモード：MRB ビットのメインテナンスも GC も行うモードである。インクリメンタル GC を行いたい場合に指定するモードである。

3. 7.4 クローズインデキシング
KL1 では、ガード部の実行で、呼出し側変数を具体化しようとしたり、組込述語の実行に失敗するとそのクローズの選択の試みは失敗し、次の候補節が試される。従って、このようなクローズの実行は行わず、選択される可能性のあるもののみを実行することが無駄な実行を省き、速度向上に役立つ。本節では、このためにコンパイラに実装したインデキシングの方式について説明する。また、インデキシング用に導入した命令についても説明する。ただし、今回説明したインデキシング方式は、暫定的なものであり、Prolog 版のコンパイラのみに実装されている。

4. 7.5 KL1 コンパイラの特徴
KL1 による KL1 コンパイラは、基本的には Prolog 版のコンパイラを基に書き直したものである。KL1 自体は、かなり平板な言語であることから、コンパイラのような大規模なプログラムを開発しようとすると、なんらかの形のモジュール化、あるいは‘書きやすさ’向上のためのシンタックスシュガリングが必要となる。本節では、KL1 コンパイラ開発のために KL1 に導入した拡張シンタックスについて説明する。

具体的には以下の2点である。

(a) DCG 拡張：コンパイラのコード生成の部分で用いた。これにより、クローズの引数個数が2ずつ減りプログラムの書きやすさ、読みやすさが増した。

(b) Case 文：KL1 のクローズのボディ部で、Prolog と同様のケース文が書けるようにした。これにより、Prolog から KL1 への変換が楽になり、また余分なクローズを定義しなくてもよくなった。

5. 7.6 使用法、その他
本節では、Prolog 版、KL1 版コンパイラを色々なマシン上で稼動させるときの留意点および変更しなければならない点などについて説明する。また、コンパイル時のエラーメッセージの意味などについても説明する。

7.2 内部構成

この章では、KL1 コンパイラの内部での処理構成について説明を行う。KL1/Prolog コンパイラ、KL1/KL1 コンパイラとも、インデキシングを除いてほぼ同じアルゴリズムを使用しており、大まかな構造は以下の5つのフェーズに分かれる。

1. プリ・プロセッサ

2. クローズの正規化

3. クローズのコード生成

4. MRB-GC 用命令生成

5. レジスタ割付

各フェイズの説明を以下で行う。

7.2.1 プリ・プロセッサ部

ここでは、ファイル、端末から読み込んだ KL1-C プログラムの展開、構造化を行う。

展開 後に述べるような拡張表記された KL1 プログラムを通常のプログラムに展開する。

構造化 連続した同じ節名、引き数の数の節群を一まとめにする。'otherwise'、'alternatively'の入ったプログラムでは、後に述べるインデキシングの対象となるブロック毎に候補節群を区切る。

例 7.1

```
a(1, X):-....
a(2, X):-....
otherwise.
a(A, X):-....

b(X) :-.....
 .....
```

```
[[a(1, X):-...., a(2, X):-....], otherwise, [a(A, X):-....]].
```

a/2 の'otherwise'迄の節群　　a/2 の'otherwise'以降の節群

```
[[b(X):-....]....]
```

b/1 の候補節群

7.2.2 正規化部

コードの生成に先立って、節内の変数の構造を明らかにするため正規化を行う。ここでは、ヘッド、ガード、ボディに出現する変数を atomic なレベルまで、展開している。これは、以下の例に示すことに相当する。

```
append([A|X1], Y, Z) :- true |
    Z = [A|Z1], append(X, Y, Z1).

append(X', Y', Z') :- X' = [A|X1], Y' = Y, Z' = Z |
    List = [Car|Cdr], A = Car, Z1 = Cdr, Z = List,
    X1' = X1, Y'' = Y, Z1' = Z1,
    append(X1', Y'', Z1')
```

正規化の代表的な例を以下にいくつか挙げる。(以下の例で'$VAR'(_)とあるのは、コンパイラが生成した変数名である。)

1. ヘッドユニフィケーションの展開
   . . . ゴール引数とヘッド引数のユニフィケーションに展開する。

```
a([A|X], Y) :- ....

a('$VAR'(0), '$VAR'(1))
'$VAR'(0)=list          % '$VAR'(0) はリスト
unify(A)                % リストの Car を読む
unify(X)                % リストの Cdr を読む
Y='$VAR'(1)             % '$VAR'(1) と Y をユニファイ
```

2. 構造体のアクティブユニフィケーションの展開

. . . 構造体を用意し、各要素ごとのユニフィケーションに展開する。

```
X=[Car|Cdr]

'$VAR'(0)=list          % '$VAR'(0) はリスト
write(var(Car))         % 'Car' を書く
write(var(Cdr))         % 'Cdr' を書く
X<='$VAR'(0)            % '$VAR'(0) と X をユニファイ
```

3. ユーザ定義ゴールの展開

. . . ゴールレコードを用意し、各要素ごとのユニフィケーションに展開する。

```
...., bar(1, X, [A|B]), .....

create_goal(bar/3)      % ゴールレコード 'bar/3' をつくる。
1 := atomic(1)          % 第1引数をゴールレコードに書く
2 := var(X)             % 第2引数をゴールレコードに書く
'$VAR'(0)=list          % '$VAR'(0) はリスト
write(var(A))           % Car に A を書く
write(var(B))           % Cdr に B を書く
3 := var('$VAR'(0))     % '$VAR'(0) をゴールレコードに書く
enqueue_goal(bar/3)     % ゴールレコードをエンキューする
```

また、TRO 実行する場合はレジスタ間転送を行う。

```
'$VAR'(1) = atomic(1)   % 第1引数をレジスタ転送する
'$VAR'(2) = var(X)      % 第2引数をレジスタ転送する
'$VAR'(3) = list        % '$VAR'(0) はリスト
write(var(A))           % Car に A を書く
write(var(B)            % Cdr に B を書く
'$VAR'(4) = '$VAR'(3)   % '$VAR'(0) をレジスタ転送する。
execute(bar('$VAR'(1), '$VAR'(2), '$VAR'(4)))
```

ここで出力される中間語と、KL1B のオペコードはほぼ1対1対応している。

7.2.3 コード生成部

ここでは正規化部で生成された中間語を KL1B のオペコードに落とす。(ただし、まだ各命令語のオペランドは変数表示のままである。) この時、変数が初出かどうかで、XXX_variable系命令 (初出の時) XXX_value系命令 (既出の時) を出し分ける必要がある。正規化部で挙げた例に対応して生成されるコードを以下に示す。

```
a([A|X], Y) :- ....

wait_list('$VAR'(0))
read_variable(A)          %  'A' は初出
read_variable(X)          %  'X' は初出
wait_variable(Y, '$VAR'(1))   % 'Y' は初出


X=[Car|Cdr]

put_list('$VAR'(0))
write_variable(A)(or write_value(A))
write_variable(B)(or write_avlue(B))
get_list(X, '$VAR'(0))


...., bar(1, X, [A|B]), .....

create_goal(bar/1)
set_constant(1, 1)
set_varable(X, 1)(or set_value(X, 1))
put_list('$VAR'(0))
write_variable(var(A))(or write_value(var(A)))
write_variable(var(B))(or write_value(var(B)))
set_value('$VAR'(0), 3)
enqueue_goal(bar/3)

put_constant(1, '$VAR'(1))
put_variable(X, '$VAR'(2))( or put_value(X, '$VAR'(2)))
put_list('$VAR'(3))
write_variable(A)( or write_value(A))
write_variable(B)( or write_value(B))
put_value('$VAR'(3), '$VAR'(4))
execute(bar('$VAR'(1), '$VAR'(2), '$VAR'(4)))
```

7.2.4 MRB-GC 用命令生成部

ここでは MRB-GC を行うため、MRB のメインテナンス、変数セルの回収用の命令を生成する。
各命令語のオペラントの変数が、

- ガードでのみ出現するときはcollect_XXX系の命令を追加する。
- ガードとボディの両方で出現する時はXXX_reused_value系の命令に置換える。
- ガードとボディ、またはボディのみに出現し、参照数が増加する時はXXX_marked_XXX系の命令に置換えする。(詳しくは 7.3 を参照)

7.2.5 レジスタ割り付け

各変数に、テンポラリ・レジスタを割り付ける。レジスタの割り付けはなるべく使用するレジスタの数を少なくするように行われる。この時、wait_variable, put_value命令 (共に単純なレジスタ転送を行う) については、可能な限り 2 つの変数に同じレジスタを割り付けるようにする。

上記のレジスタの最適化により、wait_variable, put_value命令の 2 つのオペランド変数が共に同じレジスタに割り付けられたときにはこれら命令は最終的なコードには出力しない。現在 PDSS 等エミュレータには wait_variable命令は存在せず、put_value命令に置き換えられる。

MRB 操作命令のget_marked_value(A_i, A_j)は現在存在せず、 put_marked_value(A_i, A_i), get_value(A_i, A_j)と展開される。また、 KL1/KL1 コンパイラでは、 PDSS(PIMOS Development Support System) 用フォーマットの KL1B 命令を出すこともできるが、その変換はここで行っている。

各フェーズの中間結果、及びコンパイルに必要な変数情報テーブルの例 (append, queen) を以下に示す。

コンパイル例

1. append

- ソースプログラム

```
a([A|X], Y, Z) :- true | Z=[A|Z1], a(X, Y, Z1).
a([], Y, Z) :- true | Y=Z.
```

- 正規化後
  - 第 1 節
    - ヘッド・ガード

```
0       L = list            % a([
1       unify(var(I))       %    A|
2       unify(var(F))       %      X],
3       E = var(K)          %          Y,
4       H = var(J)          %             Z) :- true |
```

    - ボディ

```
0       G = list            % [
1       write(var(I))       %  A|
2       write(var(D))       %    Z1
3       H <= list(G)        %      ]=Z,
4       F = var(C)          %           a(X,
5       E = var(B)          %                Y,
6       D = var(A)          %                   Z).
7       execute(a(C, B, A))
```

- 第2節
  - ヘッド・ガード

```
0       E = atomic([])      % a([],
1       B = var(D)          %      Y,
2       A = var(C)          %         Z) :- true |
```

  - ボディ

```
0       B = var(A)          % Y=Z.
1       proceed
```

- コード生成後
  - 第1節
    - ヘッド・ガード

```
0       try_me_else(a/3/1)
1       wait_list(L)        % a([
2       read_variable(I)    %    A|
3       read_variable(F)    %      X],
4       wait_variable(E, K) %          Y,
5       wait_variable(H, J) %             Z) :- true |
6       commit
```

    - ボディ

```
7       put_list(G)         % [
8       write_value(I)      %  A|
9       write_variable(D)   %    Z1
10      get_list_value(G, H)%      ]=Z,
11      put_value(F, C)     %           a(X,
12      put_value(E, B)     %                Y,
13      put_value(D, A)     %                   Z).
14      execute(a/3)        %
```

  - 第2節
    - ヘッド・ガード

```
0       try_me_else(a/3/2)
1       wait_constant([], E)% a([],
2       wait_variable(B, D) %      Y,
3       wait_variable(A, C) %         Z) :- true |
4       commit
```

    - ボディ

```
5       get_value(A, B)     % Y=Z.
6       proceed
```

- MRB-GC用命令生成後
  - 第1節
    - ヘッド・ガード

```
0       try_me_else(a/3/1)
1       wait_list(L)
2       read_variable(I)
3       read_variable(F)
4       wait_variable(E, K)
5       wait_variable(H, J)
6       collect_list(L)      <--  追加
```

* ボディ

```
7       put_list(G)
8       write_value(I)
9       write_variable(D)
10      get_list_value(G, H)
11      put_value(F, C)
12      put_value(E, B)
13      put_value(D, A)
14      execute(a/3)
```

- 第2節

* ヘッド・ガード

```
0       try_me_else(a/3/2)
1       wait_constant([], E)
2       wait_variable(B, D)
3       wait_variable(A, C)
```

* ボディ

```
4       get_value(A, B)
5       proceed
```

• 出力コード

```
procedure((a), 3).
label(a/3).
        try_me_else((a/3/1)).
        wait_list(1).
        read_variable(4).
        read_variable(5).
        collect_list(1).
        put_list(1).
        write_value(4).
        write_variable(4).
        get_list_value(1, 3).
        put_value(5, 1)
        put_value(4, 3).
        execute((a/3)).
label((a/3/1)).
        try_me_else((a/3/2)).
```

```
        wait_constant(([]), 1).
        get_value(3, 2).
        proceed.
label((a/3/2)).
        suspend((a/3)).
```

- 変数情報テーブル

  これらのテーブルはコンパイラ内部での作業用テーブルであり、変数の出現区間等を記憶するために用いられる。これらの情報は、レジスタ割り付け時に、同時に複数の変数が同一のレジスタを競合しないため、また効率の良いレジスタ割り付けのために用いられる。詳しい内容は以下の通り。

  第1コラム 変数名

  第2コラム 変数が初めて出現するプログラムカウンタ (以後 PC) の値

  第3コラム 変数が最後に出現する PC+1 の値

  第4コラム 変数が割り付けられたレジスタの番号

  - ヘッド引数用、TRO 実行用にレジスタを割り付ける。この時、各変数の出現区間は登録済み。

    * 第1節

L	0	2	1	append/3 呼び出し時は1 ～ 3レジスタが
K	0	5	2	ヘッド引数 (J,K,L) 用に割り当てられる。
J	0	6	3	
I	3	9	[]	
H	6	11	[]	
G	8	11	[]	
F	4	12	[]	
E	5	13	[]	
D	10	14	[]	
C	12	15	1	ボディ・ゴール append/3 呼び出し時に1 ～ 3
B	13	15	2	レジスタが TRO 引数用に割り当てられる。
A	14	15	3	

    * 第2節

E	0	2	1	append/3 呼び出し時は1 ～ 3レジスタが
D	0	3	2	ヘッド引数 (C,D,E) 用に割り当てられる。
C	0	4	3	
B	3	6	[]	
A	4	6	[]	

  - MRB-GC 用命令挿入による PC のずれを直した状態。

    第1節ではput_list命令の挿入された分ボディの PC が1ずつ増加している。また、ゴール引数用を受けるレジスタは次候補節でも値を読むため、その節が選択される事が決定する (つまりコミットする) まではレジスタの内容を保存する必要がある。

    * 第1節

L	0	7	1

K	0	5	2
J	0	6	3
I	3	10	[]
H	6	12	[]
G	9	12	[]
F	4	13	[]
E	5	14	[]
D	11	15	[]
C	13	16	1
B	14	16	2
A	15	16	3

＊第2節

E	0	2	1
D	0	3	2
C	0	4	3
B	3	6	[]
A	4	6	[]

－ レジスタ割り付け後

同一レジスタを使用する変数間で、出現区間が重ならぬようレジスタを割り付ける。

＊第1節

L	0	7	1
K	0	14	2
J	0	12	3
I	3	10	4
H	0	12	3
G	9	12	1
F	4	13	5
E	0	14	2
D	11	15	4
C	13	16	1
B	14	16	2
A	15	16	3

＊第2節

E	0	2	1
D	0	6	2
C	0	6	3
B	0	6	2
A	0	6	3

2. queen

- ソースプログラム

```
queen([P|U], C, L, I, O) :- true |
    append(U, C, N),
    c1(P, 1, N, L, L, I, X),
    queen(U, [P|C], L, X, O).
```

- 正規化後

  - ヘッド・ガード

```
0     Q = list              % queen([
1     unify(var(J))         %        P|
2     unify(var(F))         %          U],
3     E = var(P)            %             C,
4     I = var(O)            %                L,
5     H = var(N)            %                   I,
6     K = var(M)            %                      O) :- true |
```

  - ボディ

```
0     create_goal(queen/5)  % queen(
1     1 := var(F)           %       U,
2     L = list              %         [
3     write(var(J))         %          P|
4     write(var(E))         %            C
5     2 := var(L)           %             ],
6     3 := var(I)           %                L,
7     4 := var(G)           %                   X,
8     5 := var(K)           %                      O
9     enqueue_goal(queen.5) %                       ),
10    create_goal(c1/7)     % c1(
11    1 := var(J)           %    P,
12    2 := atomic(1)        %       1,
13    3 := var(D)           %          N,
14    4 := var(I)           %             L,
15    5 := var(I)           %                L,
16    6 := var(H)           %                   I,
17    7 := var(G)           %                      X
18    enqueue_goal(c1/7)    %                       ),
19    F = var(C)            % append(U,
20    E = var(B)            %           C,
21    D = var(A)            %              N
22    execute(append(C, B, A))%               ).
```

- コード生成後

  - ヘッド・ガード

```
0       try_me_else(queen/5/1)
1       wait_list(Q)            % queen([
2       read_variable(J)        %        P|
3       read_variable(F)        %          U],
4       wait_variable(E, P)     %             C,
5       wait_variable(I, O)     %                L
6       wait_variable(H, N)     %                   I,
7       wait_variable(K, M)     %                      O) :-
```

– ボディ

```
8       create_goal(queen/5)    % queen(
9       set_value(F, 1)         %       U,
10      put_list(L)             %         [
11      write_value(J)          %          P|
12      write_value(E)          %            C
13      set_value(L, 2)         %              ],
14      set_value(I, 3)         %                 L,
15      set_variable(G, 4)      %                    X,
16      set_value(K, 5)         %                       O
17      enqueue_goal(queen/5)   %                        ),
18      create_goal(c1/7)       % c1(
19      set_value(J, 1)         %    P,
20      set_constant(1, 2)      %       1,
21      set_variable(D, 3)      %          N,
22      set_value(I, 5)         %             L,
23      set_value(I, 4)         %                L,
24      set_value(H, 6)         %                   I,
25      set_value(G, 7)         %                      X
26      enqueue_goal(c1/7)      %                       ),
27      put_value(F, C)         % append(U,
28      put_value(E, B)         %           C,
29      put_value(D, A)         %              N
30      execute(append/3)       %               ).
```

• MRB-GC用命令生成後

– ヘッド・ガード

```
0       try_me_else(queen/5/1)
1       wait_list(Q)
2       read_variable(J)
3       read_variable(F)
4       wait_variable(E, P)
5       wait_variable(I, O)
6       wait_variable(H, N)
7       wait_variable(K, M)
```

```
8       collect_list(Q)          <-- 追加
```

– ボディ

```
9       create_goal(queen/5)
10      set_marked_value(F, 1)
11      put_list(L)
12      write_marked_value(J)    <-- 変更
13      write_marked_value(E)    <-- 変更
14      set_value(L, 2)
15      set_marked_value(I, 3)   <-- 変更
16      set_variable(G, 4)
17      set_value(K, 5)
18      enqueue_goal(queen/5)
19      create_goal(c1/7)
20      set_value(J, 1)
21      set_constant(1, 2)
22      set_variable(D, 3)
23      set_value(I, 5)
24      set_value(I, 4)
25      set_value(H, 6)
26      set_value(G, 7)
27      enqueue_goal(c1/7)
28      put_value(F, C)
29      put_value(E, B)
30      put_value(D, A)
31      execute(append/3)
```

- 出力コード

```
procedure((queen), 5).
label(queen/5).
        try_me_else((queen/5/1)).
        wait_list(1).
        read_variable(7).
        read_variable(6).
        collect_list(1).
        create_goal((queen/5)).
        set_marked_value(6, (1)).
        put_list(1).
        write_marked_value(7).
        write_marked_value(2).
        set_value(1, (2)).
        enqueue_goal((queen5)).
        create_goal((c1/7)).
        set_value(7, (1)).
```

```
        set_constant((1), (2)).
        set_variable(5, (3)).
        set_value(3, (4)).
        set_value(3, (5)).
        set_value(4, (6)).
        set_value(1, (7)).
        enqueue_goal((c1/7).
        put_value(6, 1).
        put_value(5, 3).
        execute((append/3)).
label(queen/5/1).
        suspend(quuen/5).
```

- ヘッド引数用、TRO 実行用にレジスタを割り付ける。

Q	0	2	1	queen/5 呼び出し時は1 ～ 5レジスタが
P	0	5	2	ヘッド引数 (J,K,L) 用に割り当てられる。
O	0	6	3	
N	0	7	4	
M	0	8	5	
L	12	15	[]	
K	8	18	[]	
J	3	21	[]	
I	6	25	[]	
H	7	26	[]	
G	17	27	[]	
F	4	29	[]	
E	5	30	[]	
D	23	31	[]	
C	29	32	1	ボディ・ゴール append/3 呼び出し時に
B	30	32	2	1 ～ 3レジスタが TRO 引数用に
A	31	32	3	割り当てられる。

- MRB-GC 用命令挿入による PC のずれを直した状態。
  第1節ではput_list命令の挿入された分ボディの PC が1ずつ増加している。また、ゴール引数用を受けるレジスタは次候補節でも値を読むため、その節が選択される事が決定する (つまりコミットする) まではレジスタの内容を保存する必要がある。

Q	0	2	1
P	0	5	2
O	0	6	3
N	0	7	4
M	0	8	5
L	13	16	[]
K	8	19	[]

J	3	22	[]
I	6	26	[]
H	7	27	[]
G	18	28	[]
F	4	30	[]
E	5	31	[]
D	24	32	[]
C	30	33	1
B	31	33	2
A	32	33	3

- レジスタ割り付け後
  同一レジスタを使用する変数間で、出現区間が重ならぬようレジスタを割り付ける。

Q	0	2	1
P	0	5	2
O	0	6	3
N	0	7	4
M	0	8	5
L	13	16	1
K	8	19	5
J	3	22	7
I	6	26	3
H	7	27	4
G	18	28	1
F	4	30	6
E	5	31	2
D	24	32	5
C	30	33	1
B	31	33	2
A	32	33	3

7.3 MRB 管理

KL1 のデータを MRB[9] 管理することによって、インクリメンタルな GC や、アレイの効率的な実現が可能になる。またプロセッサ間にまたがったデータを効率良く管理することが可能になる。本節では、これらの詳細については、言及せず、MRB-GC 用命令のレパートリと、コンパイラによるその生成アルゴリズムについて説明する。

まず、MRB-GC を正しく管理するにはどのような操作が必要になるのかを述べ、次に MRB-GC 用の命令について述べる。

7.3.1 MRB-GC のための管理

MRB-GC の管理は、大きく分けると以下の 2 つに分類できる。

1. MRB ビットのメインテナンス
   KL1 の実行をメモリの参照 / 消費と言う観点から見ると、ガード部でのユニフィケーションに依る構造体データの分解、デレファレンスおよびボディ部での新たな構造体データの生成およびゴールから受け継いだデータの子ゴールへの分配と言う操作から成る。これらの操作の時には MRB は正しくオン / オフが付けられなければならない。

2. 不要データの回収
   ガードを越えた時点で、不要になったデータ構造を回収すること。

MRB ビットのメインテナンス

KL1 の実行は以下の様な操作をメモリ領域に対して行いながら進む。このとき MRB ビットは、余分な回収を引き起こさないように正しく管理されなければならない。

1. 変数、構造体などの生成
2. 変数、構造体などのゴール間にわたる分配
3. 変数のデレファレンス
4. ユニフィケーション
5. 構造体要素の参照

以下に、個々の場合についての MRB 管理の方法を説明する。

1. 変数、構造体などの生成
   クローズのボディ部で新たに変数セルを割り付けたり、構造体を作る場合である。

```
p :- true | q(X), r(X, [car|cdr]).          (1)
p :- true | q(X), r(X, [car|cdr]), s(X).    (2)
```

   クローズ(1)の場合は、ボディ部で新たに割り付ける変数Xの出現が 2 なので、このセルに対するポインタの MRB は○である。一方、クローズ(2) では、3 回 (以上) 現れているので MRB は●となる。構造体[car|cdr] に対するポインタの MRB はクローズ(1), (2) とも○である。

2. 変数、構造体などのゴール間にわたる分配
   クローズのヘッドで受けた引数をボディ部で分配する場合である。

```
p(X) :- true | q(X).                        (1)
p(X) :- true | q(X), r(X).                  (2)
```

   クローズ(1)では、ヘッドで受けた引数をそのままボディゴールに移すだけなので MRB に変化はない。
   クローズ(2)では、2 つ (以上) に分配するので、MRB は●にする。

3. 変数のデレファレンス
   変数をデレファレンスを行う場合で、デレファレンス結果の MRB はデレファレンス途中の間接ポインタのなかで 1 つでも MRB が●のものがあれば●、それ以外の場合は○とする。

表 7.1: ユニフィケーション時の MRB 管理

	S1	S2	U1	U2○	U2●	U3
S1	★	★	I1	I1	I3	I4
S2	★	★	I3	I4	I3	I4
U1	I1	I3	I1	I2	I3	I2
U2○	I1	I4	I2	I2	I4	I4
U2●	I3	I3	I3	I4	I3	I3
U3	I4	I4	I2	I4	I3	I4

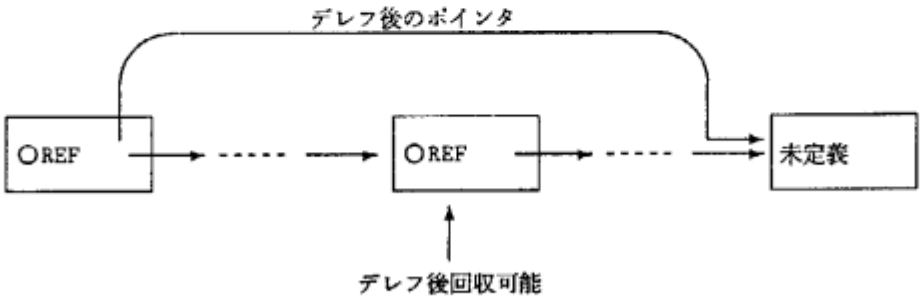

図 7.1: 変数のデレファレンス

4. ユニフィケーション
   ユニフィケーションの場合の MRB の管理は、まず、2 つの変数をデレファレンスし、どちらかの MRB が●であれば●で具体化、さもなくば○で具体化する。表 7.1に個々の場合を示す。表で、★は、構造体同士のユニフィケーションが行われることを示し、U2○及び、U2●はそれぞれ ○のポインタ / ●のポインタ を使ってユニフィケーションが行われることを示す。

5. 構造体要素の参照
   構造体要素の参照時の参照値の MRB は、ストラクチュアレジスタの MRB と要素自体の MRB の OR を取ったものとする。

不要データの回収

上記の様に KL1 のデータを MRB 管理すると、KL1 の実行中に、あるデータへの参照が最後でかつ MRB が白であれば、そのデータは、回収し、再利用できることになる。例えば、以下の場合に回収が可能となる。

1. 変数のデレファレンス
   変数のデレファレンス時の間接ポインタは、その間接ポインタセルへの参照が単一で (ポインタ MRB は白)、この間接ポインタの MRB が白であれば、デレファレンス時に回収できる (図 7.1)。

2. 構造体のユニフィケーション
   受動部で、ゴールの引数の一つが構造体との単一化に成功したとき、ゴール引数として与えられた構造体は MRB がオフであれば回収できる。例えば以下の様なクローズがあったとき第一引数として与えられたコンスセルは MRB がオフであれば回収できる。

```
p([X|Y]) :- true | b(X), c(Y).
  -----
```

3. ボイド変数とのユニフィケーション
ゴール引数がボイド変数とユニファイしたとき、そのゴール引数のMRBがオフであれば回収できる。

```
p(foo, X) :- true | true.
```

4. 回収される構造体の要素
ある構造体が回収されるとき、その構造体の要素も各々のMRBがオフであれば回収してよい。例えば、以下のクローズで、第一、第二引数とも同じ構造体が与えられ、構造体同士のユニフィケーションが行われたとき、その構造体は回収され、さらに、要素も再帰的にたぐって(MRBがオフであれば)回収できる。

```
p(X, Y) :- X = Y | true.
           -----
```

以上の内、回収できるタイミングがコンパイル時にわかるものについては、コンパイラで回収のための命令を陽に出すものとする。また、実際の回収はそのクローズが選択されるとわかった時点で行なわなければならない。従って、回収のための命令は概念的には、コミットバーを越える時点に生成される。

7.3.2 MRB − GCのための命令

本節では、前節で説明したMRBメインテナンスのための命令について説明する。この命令セットに対する抽象マシンは、先に提案したものと殆ど同じでよいが、MRB導入にあたっての変更点について簡単に述べる。次に、MRBビットメインテナンス用の命令について、最後に、回収用の命令について説明する。

抽象マシンの変更点

抽象マシンの変更点について以下にまとめる。

1. MRBビット用のフィールド
MRBのフィールドを各KL1データセルに設けなければならない。これには、タグフィールドの1ビットを充てることとした。

2. 変数セル、構造体用メモリ領域の管理
変数セル、構造体のためのメモリ領域は、動的な獲得、回収が容易にできるようにそれぞれフリーリストにより管理することにした。

3. ガーベジスタック
一般にガーベジセルの回収は、そのクローズが選択されるとわかってから行わなければならない。そこで、受動部のvalue系命令実行中にガーベジとなったデータを一時的に保持しておくためのスタックを新たに導入した(これをガーベジスタックと呼ぶ)。

4. 構造体中の未定義セル
構造体中に未定義セルを直接取るとその構造体のMRBがで回収できる時でも、未定義セルを指している他のパスがあるかも知れないので回収出来ないことになってしまう。これは、回収効率を著しく低下させるものと考えられる。そこで、未定義セルは別に取り、構造体には、それへの参照ポインタを入れることにした。

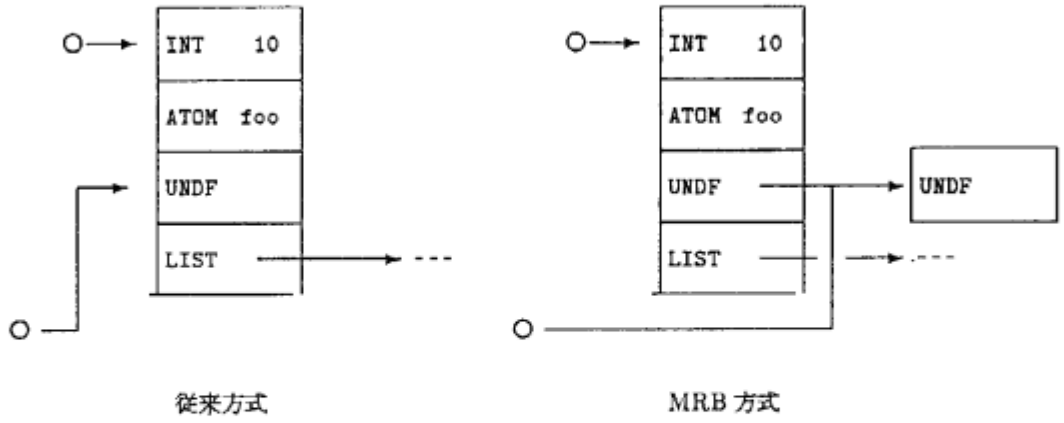

図 7.2: 構造体中の未定義セル

MRB ビットメインテナンス用命令

1. 受動部の命令

```
wait_reused_value  Xi, Aj
read_reused_value  Xi
```

受動部では、各々の命令は実行中に回収の条件を満たせば積極的にセルを回収しようとする。しかし、そのデータが能動部で使われている場合には、たとえ MRB が○であっても回収してはならない。これらの命令は回収を抑制するための命令である。レジスタXiが保持しているデータは MRB が○でも回収しない。

2. 能動部の命令

```
set_marked_value Xi, Gj
put_marked_value Xi, Aj
write_marked_value  Xi
set_marked_variable Xi, Gj
put_marked_variable  Xi, Aj
write_marked_variable  Xi
```

能動部において、MRB を●にする命令である。前 3 つの命令は、ヘッド引数として受け取ったデータを能動部で複数のゴールに分配するときに用いる命令である。残りの 3 つは、能動部で新たに 3 箇所以上の参照パスを持つ変数を割り付けるときに用いる。

回収のための命令

ガーベジセル回収のための命令である。なお、現在の命令セットはデレファレンス命令を陽に持っていないので、デレファレンス時の回収は個々の命令中で行なわれる。

1. **collect_list Ai**
   Aiから指されるリストの MRB がオフであればリストセルを回収する。これは、ソースプログラム上陽にリスト構造が書かれている場合に生成される。

2. collect_value Ai

   Aiから指されるデータのMRBがオフである限り再帰的に回収する。これは、或る変数(Aiで受けた変数)が、ボディ部で一度も使われていない場合に生成される。

3. collect_stack

   ガーベジスタックから指されているデータを回収する。これは、受動部で、value系命令が一箇所でも現われると生成される。

7.3.3 コンパイル例

MRB-GC用命令を生成するコンパイル例を以下に示す。

```
p([X|Y], X) :- true | q(X).
```

```
p/2: wait_list  A1              % 第一引数はリスト?
     read_variable X3           % car を読む
     read_variable X4           % cdr を読む
     wait_reused_value X3, A2   % 第二引数の単一化
     collect_list  A1           % リストの回収
     collect_value X4           % ボイド変数の回収
     collect_stack              % ガーベジスタックの回収
     put_value  X3, A1
     execute    q/1
```

7.4 クローズインデキシング

本節では、KL1ソースプログラムから抽象機械語への効果的なコンパイル手法、特にクローズインデキシング手法[3]について報告する。

7.4.1 基本的な考えかた

KL1は、GHC(Guarded Horn Clauses)に基づく並列論理型言語であり、そのソースプログラムは以下の様に記述される。

$$\underbrace{a(X_{i,1},\ldots,X_{i,l}) :- g_{i,1},\ldots,g_{i,m_i}}_{\text{受動部}} \mid \underbrace{b_{i,1},\ldots,b_{i,n_i}}_{\text{能動部}}. \quad (i \geq 1,\ l \geq 0,\ m_i \geq 1,\ n_i \geq 1)$$

いま、あるゴール$a(Y_1,\ldots,Y_l)$が与えられると、このゴールは以下の様に実行される。

1. 候補節群の中から1つ候補節を取り出し、ヘッドのユニフィケーション、ガード部の実行が試みられる。この時、ゴール変数を具体化しようとしたり、ガードの組込述語の実行に失敗すると、この候補節に対する選択の試みは失敗する。

2. もし、1の実行が成功したならば、能動部のゴールが新たに実行可能ゴールとなり、これらのゴールの実行が同様に試みられる。

3. 失敗ならば、別の候補節に対して1が行われる。

4. 全ての候補節に対して失敗したならば、このゴールの実行は不可能であり、他ゴールによって(ゴールの変数が具体化することなどによって)実行可能になるまで実行が中断させられる。

KL1の言語仕様では、候補節の選択の順番は決められておらず、どの候補節から行ってもよいことになっている。従って、最も素直に候補節群をコンパイルするには、ソースプログラム上先に出てきたクローズから順にコードを生成し、最後にサスペンション時の処理を行う命令を置けばよい。しかし、この方法によると、i番目のクローズの選択が失敗したことにより、$i+j$番目($j \geq 1$)のクローズの選択も失敗することが分かってしまう場合でも実行が行われてしまう。これは、非常な実行時間の損失である。そこで、コンパイル時に分かる情報により、無駄な実行を抑え、実行が中断してしまうゴールを出来るだけ早く(実行時に)検出するようなコード列を生成することが望まれる。幸い、KL1では、Prologと異なり、ソース上に書かれたクローズの順番と実行順序を保証する必要がないこと、ガード部のユニフィケーションが一方向性であることより、クローズのインデキシングは、比較的簡単に行えることが期待できる。

7.4.2 処理の概要

本節では、KL1コンパイラに実装したクローズインデキシングの方法について説明する。まず、基本的な方針を挙げる。

1. インデキシングの対象とする引数は、各クローズのヘッドの第1引数からとし、各クローズが一意に決まるまで第2、第3引数に対象を拡げていく。

2. 引数のデータ型は、整数、アトム、リスト、ベクタ、変数と、変数であるがガード部で或る決まったデータ型が来ることが期待されるもの(例えば、`a(X, Y) :- Z := X + Y | p(Z).`の様なクローズの第1、第2引数は整数でなければならない。)に分類する。

3. インデキシング対象の引数がベクタの場合には、まず、その要素数で分類し、それで決まらない場合には、その要素をインデキシングの対象とする。(リストの場合もリスト要素をたぐってインデキシングの対象とする。)

4. インデキシング対象の引数のデレファレンス結果は、引数レジスタに書き戻し、デレファレンス回数を1回に抑える。

この様な方針のもとで、コンパイラの処理手順の概略を示すと次の様になる。

1. まず、同じ述語名及び引数を持つクローズを一纏めにして扱い、各々のクローズのヘッド引数に対して、そのクローズが選択されるための引数の条件を求める。図7.3の例の場合は以下の様になる。

```
クローズ(1)    < atom(□), integer(10) >
クローズ(2)    < [{atom(f),any}|any], any >
クローズ(3)    < [{atom(g),INT}|any], INT >
```

ここで、atom(□)は、アトム'□'が実引数として与えられなくてはならないことを、同様に、'{a, b}'は2引数ベクタで、その要素が'a','b'であること、'INT'は、任意の整数であることを示す。'any'は何でもよいことを示す。

2. 次に、この条件を基に、各ノードが引数の条件による分岐、リーフ(葉)が分類されたクローズに対応するトリーを作る。この時、各ノードまでの分類で分かった引数の条件も各ノードに持たせておく。

```
a([], 10) :- true | true.                          (1)
a([f(X)|Cdr], Y) :- true | p(Y).                   (2)
a([g(X)|Cdr], Y) :- add(X, Y, Z) | p(Z).           (3)
```

図 7.3: サンプルプログラム

```
a/2:    try_me_else  a/2/0
        switch_on_type  A1, a/2/1, a/2/2, a/2/0
a/2/1:  test_constant  □, A1   /* クローズ(1) */
        wait_constant  10, A2
        proceed.
a/2/2:  read_variable  X3      /* リストの分解 */
        read_variable  X4
        jump_on_non_vector  X3
        test_arity  2, X3      /* ベクタの引数   */
        read_variable  X5      /* チェックと     */
        read_variable  X6      /* 要素の読み出し */
        jump_on_non_constant X5
        switch_on_constant X5, [(f, a/2/5),(g, a/2/6)]
a/2/5:  put_value  A2, A1      /* クローズ(2) */
        execute  p/1
a/2/6:  integer  X6            /* クローズ(3) */
        integer  A2
        add  X6, A2, A1
        execute  p/1
a/2/0:  suspend  a/2
```

図 7.4: コンパイル結果

3. その後、このトリーをたぐって往きながらクローズのコンパイルを行う。各ノードがインデキシングの命令に対応し、クローズのコンパイルでは、そこまでのトリーたぐりによって既に分かった引数に対するコードは生成しないようにする。図 7.3のプログラムをコンパイルした結果を図 7.4に示す。

7.4.3 インデキシング用の命令

本節では、KL1 コンパイラの出力するインデキシング用命令の一部について説明する。

1. **switch_on_type** A_i, *Atomic*, *List*, *Vector*
   A_iをデレファレンス後、そのデータタイプに依ってそれぞれの分岐先に分岐する。 *Atomic*はA_iが整数かアトムの時の分岐先である。 A_iがリストかベクタの時は、分岐する前にSR(ストラクチャポインタ)で第1要素を指させる。

2. **switch_on_arity** A_i, $[(N0,\ L0),\ldots,(Nk,\ Lk)]$
   A_iの引数個数$(N_0,\ N_1,\ldots,N_k)$に依って分岐先 $L_0,\ldots,L_k$に分岐する。 A_iはデレファレンス済で、ベクタを指していなければならない。

3. switch_on_constant A_i, $[(C0, L0), \ldots, (Ck, Lk)]$
   A_iの内容$(C_0, C_1, \ldots, C_k)$に依って分岐先 $L_0, \ldots, L_k$に分岐する。 A_iはデレファレンス済で、整数かアトムでなければならない。

4. test_constant $Cnst$, A_i
   A_iが$Cnst$と等しいかどうか調べる。 A_iはデレファレンス済で、整数かアトムでなければならない。

5. test_arity $Arity$, A_i
   A_iの引数個数が$Arity$かどうか調べる。 A_iはデレファレンス済でベクタを指していなければならない。

6. jump_on_non_XXX A_i
   A_iのデータタイプを調べ、それが'XXX'で指定したものと同じならば成功、さもなくば失敗する。 A_iはデレファレンスされる。この命令は、switch_on_type命令で分岐先が二方向しかない場合の最適化命令である。

7.4.4 考察

ICOTで開発した逐次マシン上のKL1処理系(PDSSシステム)に実装し、評価を行った。 BUP(ボトムアップパーザ)やKL1で書いたKL1コンパイラを実行した時には、実行時間で、1~2割の向上が得られている。 Prologに較べて、インデキシングの効果は小さい。これは、

1. KL1のガード部の実行では、呼出し側引数を具体化することがないので、トレイルの必要がないこと、
2. MRBによるGCのために、引数レジスタを壊さないようにコンパイルしているので、引数を退避しなくてよい、

などの理由で、ガード実行の失敗時のコストが比較的軽いためである。

7.5 KL1コンパイラの特徴

KL1/KL1コンパイラを記述するに当たって、プログラムの構造化の度合を高め、KL1の記述性を向上させるため、いくつかのシンタックス・シュガーを導入した[6]。ここでは、それらの説明を行う。尚、ここで紹介したシンタックス・シュガーは、 Prolog版、KL1版何れのコンパイラでもサポートされている。

7.5.1 DCG拡張

オブジェクト指向プログラミング

KL1では、以下の例のようにリストを使ったプロセス間ストリーム通信によるオブジェクト指向プログラムを記述することができる。この記法はプログラムのモジュラリティを高め、コンパイラのような大規模なプログラム開発には有効なプログラミング・スタイルであると考える。

```
/*Window0-Window はウインドウ制御用ストリーム、 Input0-Input は入力ファイル制御
用ストリーム、 Output0-Output は出力ファイル制御用ストリームで何れも差分リストで
ある。 */

level1([message(In)|Msg_Cdr], Window0, Window, Input0, Input,
                                        Output0, Output) :- true |
```

```
        Window0=[write(In)|Window1],
        message_convert(In, In1),
        level2(In1, Window1, Window2, Input0, Input1, Output0, Output1),
        level1(Msg_Cdr, Window2, Window, Input1, Input, Output1, Output).
level1([], Window0, Window, Input0, Input, Output0, Output) :- true |
        level2([], Window0, Window1, Input0, Input1, Output0, Output1),
        Window1=Window,
        Input1=Input,
        Output1=Output.
.........
.........
```

通信用ストリームの整理

上記のようなプログラミングスタイルで大規模プログラムを記述した時、オブジェクト－ストリーム構造は複雑なものとなり、また各述語引数も増加しプログラマの大きな負担となる。そこで、以下に述べるようなプログラミング技法を提案する。これは、KL1 コンパイラ記述の基本方針でもある。

1. あるオブジェクトを表すプログラム群のトップレベルに以下の例で示されるような 'router' と呼ばれるディストリビュータ・プロセスをおく。このプロセスは、通常 1 本の入力ストリームと、複数本の出力ストリームに接続しており、入力メッセージにより該当する出力ストリームを判別し送り出す働きを持っている。
2. オブジェクトを構成する各プログラムは、上記 router の入力ストリームにつながる 1 本の出力ストリームを持ち、外部への通信は、すべてこのストリームを通じて行われる。

プログラムの例

```
router([window_putt(X)|Cdr], Window_Stream, Input_File_Stream,
        Output_File_Stream) :- true |
        Window_Stream=[putt(X)|Window_Stream_Cdr],
        router(Cdr, Window_Stream_Cdr, Input_File_Stream,
                Output_File_Stream).
.......
router([output_file_putt(X)|Cdr], Window_Stream, Input_File_Stream,
        Output_File_Stream) :- true |
        Output_File_Stream=[putt(X)|Output_File_Stream_Cdr],
        router(Cdr, Window_Stream, Input_File_Stream,
                Output_File_Stream_Cdr).
.......

router([input_file_gett(X, N)|Cdr], Window_Stream, Input_File_Stream,
        Output_File_Stream) :- true |
        Input_File_Stream=[gett(X, N)|Input_File_Stream_Cdr],
        router(Cdr, Window_Stream, Input_File_Stream_Cdr,
            Output_File_Stream).
```

```
.......

router([], Window_Stream, Input_File_Stream,
        Output_File_Stream) :- true |
        Window_Stream=[],
             Input_File_Stream=[],
        Output_File_Stream=[].
```

1本のストリームにまとめられたメッセージを複数プロセスに分配することによるオーバーヘッドの増加及び並列性の減少という短所がこの方法にはあるが、プログラムのネットワーク構造を簡素化し開発効率が高まるというメリットは大きいと考える。KL1 コンパイラのプロセス－ストリーム構成を示す。(図 7.5)

DCG 拡張の導入

上記の [1本のストリーム~router] というプログラミング・テクニックで用いる各述語に接続されている唯一の出力ストリームは、DCG 拡張による差分リストを用いることによりプログラム上に陽に引数を記述する必要が無くなり、プログラマにとっては大きな負担軽減となる。

ここでいう DCG(Definite Clause Grammar) 拡張とは Prolog のそれと、ガード部のゴールは展開しないという点を除いて同じものである。具体的には以下のプログラム例のように、ヘッド、ボディ・ゴールに差分リストを表す引数 2 つが追加される。また'{}'で囲まれたゴールについては、展開をしないことにした。

```
/*DCG  拡張を用いたプログラム */
foo(X, Y) --> X > 0 |
        {bar1(X, X1)},
         bar2(X1,X2),
         bar3(X2, Y).

/* 上記プログラムのプリプロセス出力 */
foo(X, Y, S0, S) :- X > 0 |
        bar1(X, X1),
        bar2(X1, X2, S0, S1),
        bar3(X2, X3, S1, S).
```

但しこの記法は Prolog での有効な応用である構文解析に用いることはできない。これは KL1 にはバックトラック機能がないということに起因する。

先に記したオブジェクト指向プログラムを上記の記法を用いて書き直すと以下のように著しく簡素化される。

```
/*DCG 拡張を利用して書いたオブジェクト指向プログラムの例。ウィンドウに対する
出力メッセージは、ファイルに対するものと区別する必要があるので、'window_write'
とした。 */
        level1([message(In)|Cdr]) --> true |
                {message_convert(In, In1)},
                [window_write(In)],
                level2(In1),
                level1(Cdr).
```

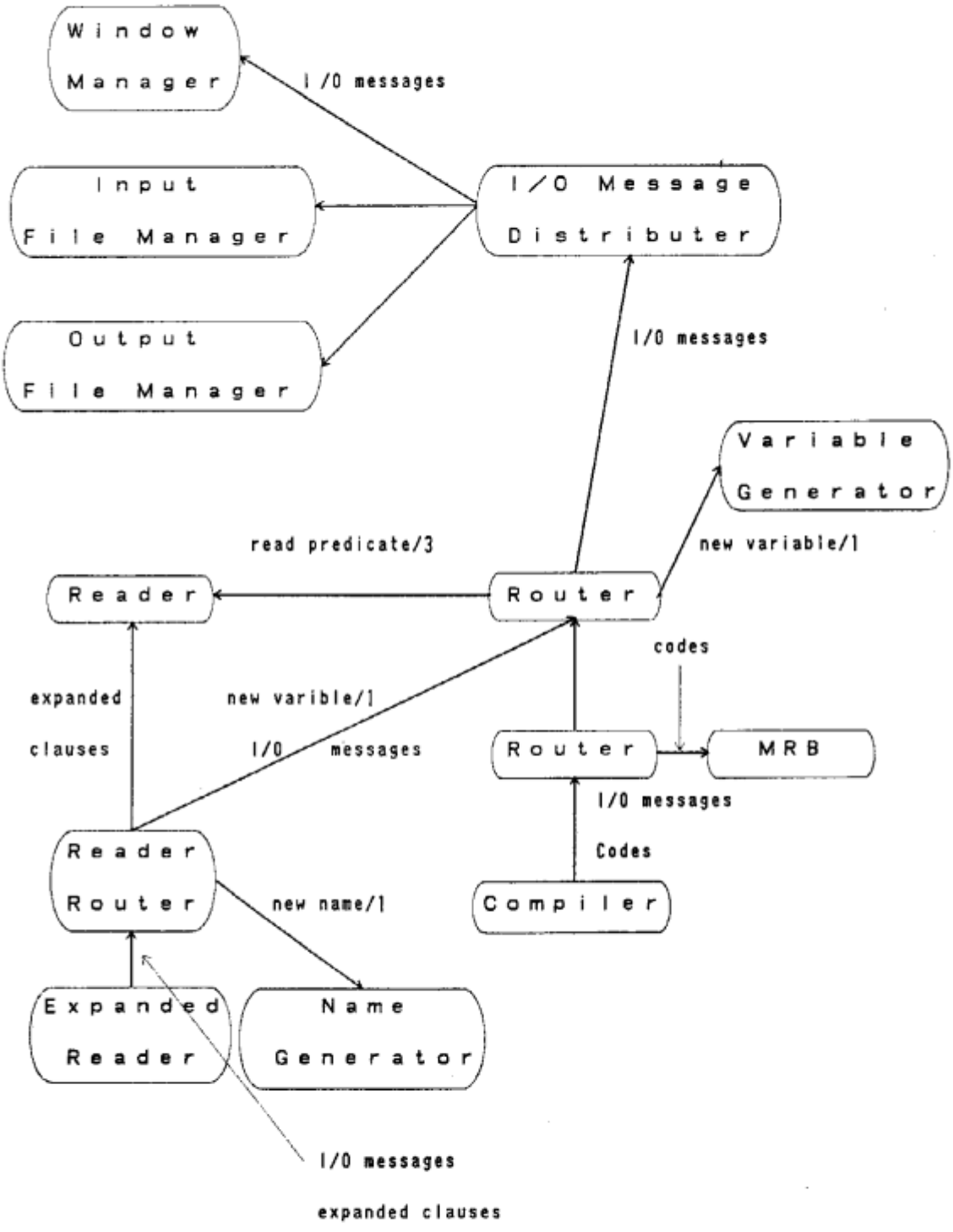

図 7.5: KL1 コンパイラのプロセス－ストリーム構成

```
        label1([]) --> true | [].
```

KL1のプログラムでは、入出力もデバイスマネージャにメッセージを送ることにより行われ、Prologプログラムとは全く異なる記法になる。デバグのために一時的に変数の値を出力することはプログラム開発において日常的なことであるが、KL1ではストリームの管理を伴うため、困難な作業になりやすい。DCG記法をもちいてI/Oストリームを引き回しておけばこういった作業もPrologライクに比較的簡単に行うことができる。

```
/*Prologで書いたプログラム*/
        foo(X) :- write(X), nl.

/*通常のKL1で書いたプログラム*/
        foo(X, Stream0, Stream) :- true | Stream0=[write(X), nl|Stream].

/*DCG拡張を利用して書いたKL1プログラム*/
        foo(X) --> true | [write(X), nl].
```

この拡張を用いて書いた実際のKL1/KL1コンパイラのプログラム例を以下に示す。

KL1/KL1コンパイラのプログラム例(各部処理の内容については、7.2を参照のこと)

```
        compile_one_block([], _, C0, C) --> true | {C0=C}.
        compile_one_block([One|Rest], Pred, C0, C, Opt) --> true |
/*  コード生成部  */
                {compile:compile_one_clause(One, Pred/C0, From0-To0-Reg0,
                                Argtbl, Nv, PC0, O0, [])},
/*  コード生成部の出力メッセージをI/Oメッセージとコードに分ける。*/
                 normalize:router(O0, Code0, []),
/*  MRB メインテナンス用命令生成部  */
                {mrb:mrb_optimization(Opt, Code0, PC0-PC,
                                Nv, Argtbl, (From0-To0)-(From1-To1), O1, []) },
/*  コード生成部  */
                 normalize:router(O1, Code, []),
/*  レジスタ割り付け部  */
                {register_predefine(Nv, From1, To1, Reg0, From2, To2, Reg1,
                                Used0, Used1) },
                 register_preference(Code, From2-To2-Reg1, From3-To3-Reg2,
                                Used1, Used2),
                 determine_registers(Nv, 0, From3, To3, Reg2, From4, To4, Reg3,
                                Used2, _),
                {reader:hurry(Nv, [], Type0),
/*  コード出力部  */
                 output:erace_or_convert_instructions(Code, From4-To4-Reg3,
                                0, Type0, 0, out_code, []),
                 output:convert_code_for_PDSS(Out_code, Out_code1)},
```

```
        output:write_object_code(Out_code1),
       {C1 := C0+1},
        output:write_label(Pred/C0),
        compile_one_block(rest, Pred, C1, C, Opt).
```

7.5.2 Case 拡張

KL1 にはPrologのような'OR'の機能を表す演算子(Prologでは';')がなく、ユーザが別の節を設けて展開するより他なかった。今回、以下のようなシンタックス上の拡張を設けることにより、KL1でもProlog同様の記法が可能となり、プログラムの記述性を高めることができた。

```
/*Prologで書いたプログラム*/
        foo(X, Y, Z) :-
                bar1(X, Ans),
               (Ans = yes, !, bar2(Y, Z);
                bar3(Y, Z) ).

/*通常のKL1で書いたプログラム*/
        foo(X, Y, Z) :- true |
                bar1(X, Ans),
                foo2(Ans, Y, Z).

        foo2(Ans, Y, Z) :- Ans = yes |
                bar2(Y, Z).
        foo2(Ans, Y, Z) :- Ans = no |
                bar3(Y, Z).

/*Case拡張を使用して書いたプログラムの例
このプログラムは上記KL1 プログラムと同じプログラムに展開される。*/
        foo(X, Y, Z) :- true |
                  bar1(X, Ans),
               (Ans = yes -> bar2(Y, Z);
                Ans = no  -> bar3(Y, Z) ).
```

7.6 KL1 コンパイラの使用法

7.6.1 使用法

本節では、KL1 コンパイラの使用法について説明する。

KL1/Prologは、Prologで書かれており、DEC10 Prolog互換のProlog上で稼働できる。現在はICOTのDECSystem20上のDEC10 Prolog、VAX上のQuintus Prolog、Balance Symmetry上のSICStus Prolog上で動作している。また、KL1/KL1 コンパイラは、現在、DEC20の上田版GHC処理系上、Balance SymmetryのPDSS上で動作している。以下の説明では、特に断わりのない限り、Prolog版、KL1版で共通であるものとする。また、Prolog版は、Symmetry上のコンパイラ、KL1版は、PDSS上のコンパイラであるものとして説明する。

起動

- Prolog 版 : Dynix コマンドレベルより、以下のように入力する。
  % pdss/release/compiler/kl1cmp
- KL1 版 : Symmetry 上の Emacs 上の PDSS/μPIMOS 上で、以下のファイルをロードする。(ディレクトリは何れも ' pdss/kl1cmp')
  - reader.sav(読み込み部、プリ・プロセッサ部のモジュール)
  - norm.sav(正規化部のモジュール)
  - comp.sav(コード生成部のモジュール)
  - mrb.sav(MRB 用命令生成部のモジュール)
  - reg.sav(レジスタ割り付け部のモジュール)
  - outp.sav(コード出力部のモジュール)
  - blt.sav(Builtin 述語用テーブル)

次に、 PDSS-SHELL ウィンドウのプロンプトから、

```
:- mrb:kl1cmp.
```

と入力すると、'PDSS-kl1cmp' ウィンドウが作られるので、 Emacs のウィンドウを切り替える。

各種パラメータ入力

上記の操作を行うと、

```
**** KL1/B Compiler Var XXX(0XXXXX) ****
Indexing    :
MRB Option  :
Input  file :
Output file :
```

と入力を要求してくる。各要求に対しては、以下のように答える。

Indexing: インデキシング用命令を出力すかどうかを入力。 0 を入力すれば、 Indexing 用命令は出ず、 1 で出るモードになる。

MRB Option: 0~2 を入力。各々以下のような意味を持つ。

0: MRB メインテナンスをしない。

1: MRB メインテナンス用命令は出すが、回収用の命令は出さない。

2: MRB メインテナンス用命令を出し、回収も行う。

Input/Output file: 各々ファイル名を入力。(Prolog でアトムと認識され　る必要があり、'.' 等を含むときはクォーテーションで囲むこと。)

但し、現在の KL1 版では、インデキシングはサポートをしておらず、 Indexing オプションは無い。

コンパイラの版による相違点

1. Prolog 版
   SICStus Prolog 上のコンパイラを標準と考えると、 DEC10、Quintus 上のコンパイラでは、テーブル類がすべて DEC10 Prolog の構造体で書かれている。(SICStus 上では、配列構造が使用できる。)

2. KL1 版
   PDSS 上処理系と、上田 GHC 上処理系では、ビルトイン、I/O 周りが大きく異なる。さらに、両者の GHC(KL1) のシンタックスにも若干違いがある (例. リストと構造体の関係)。ただ、出力されるコードは同じものである。

現在 Prolog 版は、SICStus 上のもの、 KL1 版は PDSS 上のものが主にメインテナンスされており、他の版では、改修が遅れることがある。

7.6.2 エラーメッセージ

現在のコンパイラで出されるエラーメッセージをまとめる。

- `module name is duplicated`
  module宣言が複数箇所でされている。
- `module name is not specified`
  module宣言がされていない。
- `otherwise or alternatively was found irregurally.`
  otherwise, alternativelyの前後の節のfunctor/arityが異なる。
- `Error in pragma specification`
  プラグマの記述に誤りがある。
- `Error in Priority specification`
  プライオリティの記述に誤りがある。
- `Left hand side variable in ':=' is instantiated`
  ':='の左辺が未定義変数でない。
- `Undefined binary operator`
  未定義の二項演算子を発見した。
- `Guard goal not implemented`
  未定義のガード・ゴールを発見した。
- `Body goal not implemented`
  現在のコンパイラでサポートできないボディ・ユニフィケーションをしようとした。
- `Variable should be appeared in goal argument`
  ガードの組み込み述語integer/1, wait/1の引数が、初出である。
- `Some non-implemented unification in guard`
  現在のコンパイラでサポートできないガード・ユニフィケーションをしようとした。

- `Var-var unification not implemented in the guard`
  ガードで初出変数同士をユニフィケーションしようとした。
- `Some non-implemented unification in guard`
  ガードでインプリメントされていないユニフィケーションが書かれている。
- `Value-void variable unifivation appear in wait_variable`
  ボイド変数とのユニフィケーションが記述されている。
- `Unification too much complicated`
  ユニフィケーションが複雑過ぎて、レジスタが足り無くなった。(現在、使用可能レジスタ数は32個である。)
- `Illegal case`
  Case文中にCaseオペレータがない。
- `Uninstanciated case`
  Case文がプログラム中未定義変数になっている。

参考文献

[1] ICOT 第四研究室編：PIMOS 第一版概念仕様書 *ICOT TM-290 1987*

[2] 木村、近山、久門、中島：並列推論マシン PIM-KL1 の抽象命令仕様とコンパイラ－ 情処全大 34 回 62 年前期

[3] 木村、関田、近山：KL1 コンパイラにおけるコード生成の最適化 情処全大 36 回 63 年前期

[4] 佐藤、近山、杉野、瀧：PIMOS の概要－並列推論マシン用オペレーティング・システムの構築 情処全大 34 回 62 年前期

[5] 佐藤、越村、近山、瀧：並列論理型 OS-PIMOS(1)－資源管理方式－ 情処全大 35 回 62 年後期

[6] 関田、J.Barklund、木村、近山：KL1 の拡張シンタックス 情処全大 36 回 63 年前期

[7] 松尾、藤瀬、佐藤、近山：PIMOS のタスク管理方式－タスク終了時の資源解放 情処全大 36 回 63 年前期

[8] 宮崎：PDSS 使用手引き *ICOT TM-437 1988*

[9] T.Chikayama, Y.Kimura "Multiple Reference Management in Flat GHC" *4th ICLP, 1987*

[10] Y.Kimura, T.Chikayama "An Abstract KL1 Machine and its Instruction Set" *SLP'87*