ICOT Technical Memorandum: TM-0883

TM-0883

PIMOS速修マニュアル
(第1.5版)

近山　隆，寿崎かすみ

May, 1990

ICOT

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

(03) 456-3191～5
Telex ICOT J32964

Institute for New Generation Computer Technology

PIMOS 速修マニュアル
(第 1.5 版)

目次

第 1 章

はじめに

本書は初めて PIMOS を使うユーザの方々が KL1 プログラムを作成し，PIMOS 1.5 版上でそのプログラムを実行できるようになるための入門書として書かれています．本書では，PIMOS の初期設定から起動，KL1 プログラムの作成から，その実行，デバッグ，そして性能評価に至る一連の操作の流れを具体的な例によって説明します．PIMOS の各種の機能についても簡単に紹介してあります．本書を読んだ後，PIMOS 及び KL1 について，より詳細な説明を必要とされる方は「PIMOS マニュアル」[3] を参照して下さい．また，KL1 のプログラミング技法については，[5]「KL1 プログラミング 入門編 / 初級編 / 中級編」に詳しく解説してあります．

本書は、(Pseudo) マルチ PSI 上の KL1 処理系、PIMOS、CSP、クロスシステム CAL をインストール済みであることを前提にしています．インストールについては [1] を参照して下さい．

また、各ユーザの方が PSI の OS である SIMPOS に関する操作のうち、以下については既知であるとしています．したがって，PSI 及び SIMPOS の操作については必要と思われる場合を除き説明をしていません．わからない点がある場合は，「SIMPOS 操作説明書」[7] などを参考にして下さい．

- 基本操作

  PSI-II のログイン / ログアウト / シャットダウンができる．
  マウスの基本操作 (システムメニューの呼びだし、メニューの選択、ウインドウ作成) ができる．

- ユーザ登録

  ユーザ登録ができる．login.com を編集して、システムメニューに項目を登録できる．論理名をディレクトリに付けることができる．

- PMACS エディタ

  PMACS エディタで、テキストの編集ができる．

- ファイルマニュピレータ

  ディレクトリの作成，ファイルのコピーができる．

本書は次の 3 つの部分から構成されています．

1. PIMOS の初期設定と起動及び終了の方法 (第 2 ～ 3 章)
   PIMOS を起動する前に各ユーザが行なうべき準備と、PIMOS の起動，ログイン，終了の方法を示します．

2. KL1 プログラムの作成と実行 (第4～7章)
   KL1 プログラムの書き方と負荷分散の方法，プログラムの実行，デバッグ，性能の評価の方法，そしてトラブルの対処法について説明します.

3. PIMOS の各機能の説明 (第8～11章)
   PIMOS のシェル，リスナ，その他のユーティリティについて，個別の機能とその使い方を説明します.

本書の使い方としては，実際に (Pseudo) マルチ PSI の前に座って，実際に操作しながら読むことをお勧めします.

さあ早速マルチ PSI を使ってみましょう！！

第 2 章

PIMOS 起動のための準備

この章では，SIMPOS のユーザの登録と，PIMOS を使用するために SIMPOS 上で行うユーザごとの環境設定について説明します．

2.1 ユーザ登録

まず，PSI にログインしてみましょう．標準ユーザがつぎの名前で用意されています．

Pseudo マルチ PSI ユーザは，	pmpsi
マルチ PSI ユーザは，マスタ FEP に	mpsi
スレーブ FEP に	slave

パスワードはユーザ名と同じです．PIMOS および CAL(KL1 のクロス・システム) の起動は，システムメニュウのつぎのものを選択して行います．

- Pseudo Multi-PSI あるいは Multi-PSI
  PIMOS を起動する際に選択する項目．
- CAL
  KL1 のクロスシステム．コンパイル・アセンブル・リンクが行える．CAL を使用すると，コンパイルしたプログラムは，PIMOS 本体のオブジェクト (pimos:pimos.kbn) にリンクされる．

この標準ユーザは，サンプルとして用意されたものなので，KL1 プログラムの開発者は自分用のユーザを登録して下さい．手順は次の通りです．

1. ユーザ登録 (実機マルチ PSI ではマスタ FEP のみ行なう)
   ユーザ登録は，特権ユーザ (superuser) でログインしなおし，
   システムメニュー から次の順で項目を選択することにより行なうことができます．

   others → user → maintenance → register

   ユーザのクラスは，つぎのように設定します．

Pseudo マルチ PSI ユーザならば，	ランク	general,	初期モード general
マルチ PSI ユーザならば，	ランク	general,	初期モード general

として下さい．登録が終ったらログアウトして，今登録したユーザ名でログインして下さい．
ユーザ登録・管理の詳細な説明は，「小型化 PSI/SIMPOS システム管理説明書」[6] を参照して下さい．

2. 環境設定

各ユーザ用の環境を設定します．
login.com を標準ユーザからコピーして，自分の環境に合うように修正します．次の例を参考にして下さい (PMACS エディタで編集します)．

例) ユーザ名が Taro さんの場合

(a) ファイルマニピュレータで，標準ユーザの login.com を Taro さんのディレクトリにコピーします．

```
Pseudo のとき
>sys>user>pmpsi>login.com  を >sys>user>Taro>*.*  にコピーします．
マルチ PSI 実機のとき
>sys>user>mpsi>login.com  を >sys>user>Taro>*.*  にコピーします．
```

(b) PMACS エディタで

- システムメニューに以下内容が登録されていることを確認します．

```
menu :-
    items_list(
        ....................................
%%%  Pseudo マルチ PSI                    Pseudo の時のみ必要
     {"Pseudo Multi-PSI",  mpsicsp##pseudo_csp_main_program},

%%%  マルチ PSI                           マルチ PSI 実機の時のみ必要
     {"Multi-PSI",         mpsicsp##csp_main_program},
            ....................................
%%%  クロスシステム CAL
     {"CAL",               cal##cal_manipulator},
```

- 論理名 me の定義を自分のディレクトリに変更します．

```
define :-
%%%%        "me" := ["user:pmpsi"],    コメントアウト
            "me" := ["user:Taro"],
        ....................
```

- 次の論理名が定義されていることを確認します．

```
Pseudo マルチ PSI の場合

  "root"  := ["user:pmpsi"],
  "psys"  := [">sys>csp>SYSTEM.DIR"],
  "pmpsi" := [">sys>csp>PMPSI.DIR"],
  "pimos" := [">sys>csp>PIMOS.DIR"],


マルチ PSI 実機の場合

  "root"  := [">sys>csp"],
```

```
"mpsi"  := ["root:MPSI.DIR"],
"msys"  := ["root:SYSTEM.DIR"],
"pimos" := [">sys>csp>PIMOS.DIR"],
```

(c) いったんログアウトして，ログインしなおします.

(d) システムメニューに，"Pseudo Multi-PSI" あるいは "Multi-PSI" と "CAL" の項目があることを確認します.

(e) マルチ PSI 実機の場合はマスタ FEP の,

msys:MPSI.CONFIG

を編集し，各 FEP のネットワークアドレスを設定して下さい．設定方法は，CSP マニュアル [2] の 6.3.5 の (3)feps を参照して下さい．msys:MPSI.CONFIG には，インストール時にマシン構成にあわせた内容が記述してあるので，ネットワークアドレスのみ変更してください.

2.2 カスタマイズ

- 自分用のシステムディレクトリとして，次のディレクトリを作ります.

```
me:PIMOS.DIR
me:SYSTEM.DIR   ( Pseudo マルチ PSI の時のみ. )
```

- Pseudo マルチ PSI の場合は，自分の me:SYSTEM.DIR に， psys: の全ファイルをコピーしますコピーされるファイルはつぎの 4 個です.

```
MPSI.CONFIG
MPSI.PARAM
boot.init
usercom.init
```

- 自分の me:PIMOS.DIR に pimos: にある pimos.mac 以外の 5 つのファイルをコピーします.

```
pimos.conf
pimos.kbn
pimos.ldb
pimos.sym
pimos.users
```

- 今コピーした pimos.users の以下の箇所を修正します.

```
%%% [system(*,*,"sys>user>ShellUser",*,
    [system(*,*,"sys>user>Taro",*,

%%% [system(*,*,"sys>user>ListenerUser",*,
    [system(*,*,"sys>user>Taro",*,
```

このファイル (利用環境設定ファイル) は，PIMOS へのログイン時に実行するタスクを指定するためのものです.

あなたは，ファイルに自分のログイン名とログイン時に実行するのタスク (あなたのプログラムの実行を含む) を指定することができます.

利用環境設定ファイルの詳細は PIMOS マニュアル [3]12.3 を参照してください.

- login.com を次のように修正します.

```
%%%  "psys"   := [">sys>csp>SYSTEM.DIR"],
  "psys"   := ["me:SYSTEM.DIR"],       (Pseudo マルチ PSI のみ)
%%%  "pimos"  := ["sys>csp>PIMOS.DIR"],
  "pimos"  := ["me:PIMOS.DIR"],
```

- Pseudo マルチ PSI では，コンフィギュレーション・ファイル

```
psys:MPSI.CONFIG
```

を変更することによって，使用するプロセッサ数などのシステムの構成を変更できます.
(ファイルの変更の方法については，CSP 操作説明書 [2] の 4 章，6 章を参照して下さい.
実機マルチ PSI の場合は変更できません.)
以上で，SIMPOS 上での準備はできました.

- もう一度，ログインしなおして PIMOS を起動しましょう.

第 3 章

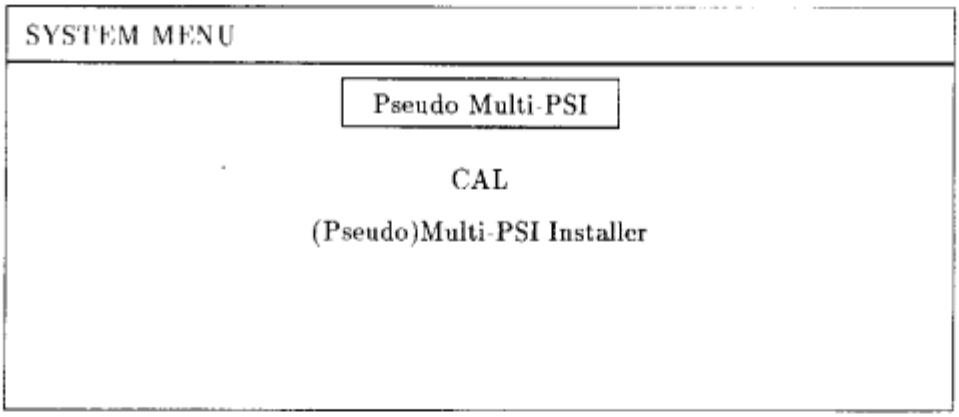
PIMOS の起動，ログイン，シャットダウン

この章では PIMOS の起動，ログイン，シャットダウンの方法について説明します．また PIMOS が正常に起動しなかった場合の対処方法についても簡単に示します．

3.1 (Pseudo) マルチ PSI の起動

3.1.1 Pseudo マルチ PSI の場合

SIMPOS にログインした直後，あるいは，ログイン後にマウスの右ボタンを 2 回クリックするとシステムメニューが現れます．Pseudo マルチ PSI はこのメニューの "Pseudo Multi-PSI" という項目を選択して起動します．



マルチ PSI-CSP が起動されると，左に『状態監視パネル』，右に『CSP コマンドウインドウ』が現れます．（SIMPOS を立ち上げた後，PIMOS を最初に起動する際には，CSP プログラムのローディングのため，多少時間がかかります．）

CSP コマンドウインドウには，次のようなメッセージが出力されたのち，プロンプト $ が表示されます．

```
Initiating all Pes from configulation file MPSI.CONFIG
Current PEs are
    0, 1, 2, 3, 4, 5, 6, 7
Current Default PEs are
    0, 1, 2, 3, 4, 5, 6, 7
PE auto power on... All PEs power ready.
PEs initiation completed
```

```
$
```

3.1.2 マルチ PSI の場合

ここでは標準的なユーザ (全ての FEP が PSI ネットでつながっている) の場合について説明します. それ以外の場合および, 詳細は CSP マニュアル [2] の 4 章, および 6 章を参照して下さい.

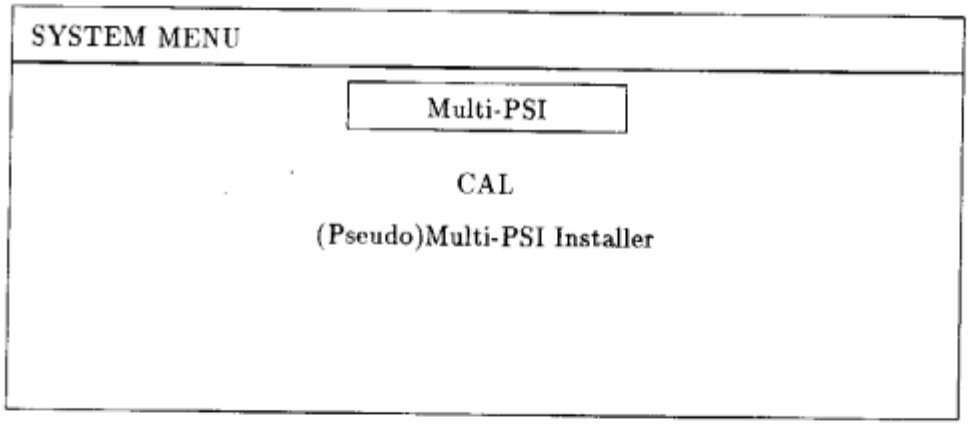
1. マスタ FEP および, スレーブ FEP の電源を入れ, それらをすべて立ち上げます.
2. スレーブ FEP にユーザ名

   ```
   slave ( パスワードも同じ )
   ```

   でログインします.
3. マスタ FEP からの通信待ちというメッセージが表示されるのを確認します.

   ```
   ただいまマスタ CSP からの通信待機中です.  しばらくお待ち下さい.
   Now Waiting Orders from Master CSP.  Please wait ...
   ```

4. マスタ FEP に自分の登録したユーザ名でログインします.
5. システムメニューからマルチ PSI に対応する項目を選択します.



6. マスタ, スレーブ両 FEP 上で CSP が起動されます.
7. スレーブ FEP には sys:MPSI.CONFIG というファイルが存在しないというメッセージが表示されますが, 無視してください.

   ```
   Configuration File does't exist.  sys:MPSI.CONFIG
   Slave FEP-CSP Program is Starting !!!
   ```

8. マスタ FEP には, 左に『状態監視パネル』, 右に『CSP コマンドウインドウ』が現れます. (SIMPOS を立ち上げた後, PIMOS を最初に起動する際には, CSP プログラムのローディングのため, 多少時間がかかります.)
9. CSP コマンドウインドウには, 次のようなメッセージが出力されます.

```
Initiating all Pes from configulation file MPSI.CONFIG
Current PEs are
    0, 1, 2, 3, 4, 5, 6, 7
Current Default PEs are
    0, 1, 2, 3, 4, 5, 6, 7
PE auto power on... All PEs power ready.  Memory Configuration...
PEs initiation completed

$
```

$ は CSP のプロンプトです．次節で示すオートブートを設定すると，プロンプトは表示されず，直ちに PIMOS を起動します．このプロンプトに対し，? または help を入力すると，全ての CSP コマンドが表示されます．各コマンドの機能については「CSP 操作マニュアル」[2] を参照して下さい．

3.2 オートブートの設定

PIMOS の立ち上げは，オートブートスイッチを設定しておくことにより，システムメニューから "(Pseudo) Multi-PSI" を選択して自動的に行うことができます．設定の方法は次の通りです．

CSP コマンドウインドウで，プロンプトに続いてコマンド

```
$ panel
```

を入力します．

すると，『CSP Console Panel』が表示されます．『CSP Console Panel』は，いくつかのサブメニューから成っています，これらのサブメニューを次のように設定します．

1. オートブートのスイッチ群を以下のように設定します．

CLEAR		[ON]	OFF
INIT		[ON]	OFF
IMPL		[ON]	OFF
Patch		[ON]	OFF
- - - - - - - - - - -		- - - - -	
IPL		[ON]	OFF
START			
	[PIMOS]	ON	OFF

2. PIMOS の立ち上げに必要なファイルは標準として次のように設定されています．必要に応じて変更して下さい．

   - Pseudo マルチ PSI の場合

CLEAR	FILE (.mbn)	[(Don't care on Pseudo)]
INIT	FILE (.mbn)	[(Don't care on Pseudo)]
INIT	FILE (.mbn)	[(Don't care on Pseudo)]
IMPL	FILE (.mbn)	[(Don't care on Pseudo)]
IMPL	FILE (.kbn)	[pmpsi:kl1kp.kbn]

PATCH FILE (.com) [pmpsi:kl1pat.com]
IPL FILE (.kbn) [pimos:pimos.kbn]
SYMBOL FILE (.sym) [pimos:pimos.sym]
START ADDRESS [202]

- マルチ PSI の場合

CLEAR FILE (.mbn) [mpsi:MCLEAR.MBN]
INIT FILE (.mbn) [mpsi:MINIT.MBN]
INIT FILE (.mbn) [mpsi:MIDAT.MBN]
IMPL FILE (.mbn) [mpsi:MPSI.MBN]
IMPL FILE (.kbn) [(Don't care on MPSI)]
PATCH FILE (.com) [mpsi:MPSIPAT.com]
IPL FILE (.kbn) [pimos:pimos.kbn]
SYMBOL FILE (.sym) [pimos:pimos.sym]
START ADDRESS [2000]

3. マウスで次の項目の "AUTO" を選択します.

Power On Auto Boot [AUTO] CONSOLE

4. 最後に，上記設定をセーブします.

[SAVE] EXIT

5. Console Panel を閉じて，終了します.

SAVE [EXIT]

3.3 CSP からの PIMOS の立ち上げ

オートブートが設定されていない場合は，CSP からコマンドで PIMOS を立ち上げます. CSP コマンドウインドウから boot と入力します.

```
$ boot
```

これによってブートが始まり，実機マルチ PSI では自動的に各プロセサに電源が入ります.

3.4 ログイン

PIMOS が正常に起動されると PIMOS にログインするための利用環境設定ウィンドウが開きます.

ここで，利用環境設定ファイル "pimos:pimos.users" に登録されているユーザ名を入力します．次の三つのユーザ名が標準として用意されています．

```
ShellUser, ListenerUser, shutdown
```

利用環境設定ファイルの詳細は「PIMOS マニュアル」[3] の 12.3 を参照してください．

```
User Name>>ShellUser
```

と入力すると，シェル (後述) が起動されます．

```
User Name>>ListenerUser
```

と入力すると，リスナ (後述) が起動されます．

KL1 プログラムのコンパイル，実行，デバッグはシェルおよびリスナの上で行なうことができます．

PIMOS にはログアウトがありません．また，複数回ログインすることができます．たとえば，ShellUser で二回ログインすると，シェルが 2 つ起動されます．

3.5 シャットダウン

PIMOS を終了するには，利用環境設定ウインドウで，次のように入力します．

```
User Name>> shutdown
```

シャットダウンを行うかどうかを確認するウインドウが表れるので，

```
Really(yes/no)? yes
```

を入力して下さい．すると，CSP ウインドウに次のようなメッセージが表示されます．

```
Do you want to exit from CSP ? [Y/N] : y
```

ここで，y を入力して下さい．

これで (Pseudo) マルチ PSI が終了します実機では各プロセサの電源が自動的に切れます)．

PIMOS の終了についての詳細は PIMOS マニュアル [3] の 12 章を参照して下さい．また，マルチ PSI の電源，CSP については CSP マニュアル [2] の 7 章を参照して下さい．

3.6 PIMOS の中断

何かの理由で PIMOS の実行を中断したい場合は，次の方法で CSP のプロンプトに制御を戻すことができます.

1. CSP コマンドウインドウ上でマウス (左ボタン) をクリックして，ウインドウを選択した後,

   Control-C

   を入力して下さい.

2. Control-C が利かない場合は，左上の『状態監視パネル』でマウスの左ボタンをクリックしたあと， abort キーを押します. ウインドウがあらわれたら

   "Return to Csp Toplevel"

   を選択して下さい.

中断した後， PIMOS の実行を再開するには， CSP コマンドウインドウから,

```
$go
```

と入力して下さい.

3.7 PIMOS の起動に失敗した時

Pseudo マルチ PSI では CAL を用いてリンクした PIMOS 本体のオブジェクト

```
pimos:pimos.kbn
pimos:pimos.ldb
pimos:pimos.sys
```

に何らかの問題があると，起動に失敗することがあります.

- CSP Console ウインドウに " Copying GC Memory Shortage " のようなメッセージが出た場合.
  第 2 章の最後の節に示したコンフィギュレーションファイルを変更してプロセッサの台数を減らして下さい. PIMOS を一旦シャットダウンしてシステムメニューから再起動すると，正常に起動できる場合があります.

- ユーザのプログラムにバグがある場合
  上記三つのファイルを一度消して, "sys>csp>PIMOS.DIR"からコピーし直し, PIMOS 本体のオブジェクトを作り直して下さい.

- PIMOS 本体がパッチのために大きくなりすぎた場合
  CAL によって何回もプログラムを PIMOS にリンクすると， PIMOS 本体のパッチが大きくなり，起動に失敗することがあります. この場合も，上と同様にして本体をコピーしなおし， CAL によるコンパイル・リンクと PIMOS の起動を最初からやりなおしてください.

- それ以外の場合
  同様に本体をコピーしなおし， PIMOS を起動した後， PIMOS 上のコンパイラでプログラムを実行してみて下さい. これが正しく動いたにも関わらず，ふたたび CAL でコンパイル・リンクすると PIMOS が正常に起動できない場合は， PIMOS あるいは CAL 自体のバグの可能性があるので，バグレポートを作成して下さい. (「CSP 操作説明書」 [2] 参照.)

第 4 章

KL1 プログラミング

並列論理型プログラム言語 KL1 は, 第五世代コンピュータ・プロジェクトの一環として新世代コンピュータ技術開発機構 (ICOT) で開発された並列論理型言語 GHC(Guarded Horn Clause) をベースに, 効率的なプログラミングやシステム記述に必要な機能を拡張した汎用のプログラミング言語です.

この章では KL1 について, 読者の方が多少なりとも Prolog でプログラムを書いた経験があることを前提として説明します.

4.1 KL1 の特徴

プログラミング言語 KL1 は次のような特徴を持ちます.

4.1.1 節の選択と実行

KL1 の述語の記述はコミットオペレータ "|"によって二つの部分に分けることができます. オペレータの左側の部分をガード部, 右側の部分をボディ部と呼びます.

ガード部には, 組み込み述語のみを記述できます. それによって, OR の関係に置かれた節から適するものを選ぶための条件を記述します.

```
a(X) :- integer(X) | b1(X,Y), b2(Y).      %(0-1)
a(X) :- X = 3 | c1(X,Y), c2(Y,Z), c3(Z).  %(0-2)
a(X) :- list(X) | d1(X,Y), d2(Y), d3(Y).  %(0-3)
```

仮に, ゴール?- a(3) が他の述語から呼ばれたとします. この時, (0-1),(0-2),(0-3) の各節が起動され, そのうち (0-1),(0 2) のガード部の integer(3) と 3 = 3 が成功します. しかし, KL1 の処理系はガード部が最初に成功した節を一つだけ選択し, 他の節は破棄します. そして, 選択した節のボディ部を実行します.

この場合, (0-1),(0-2) のボディ部はどちらか一方だけが実行されます. 仮に, (0-2) の節が選択されたとすると, そのボディ部にあるゴール c1, c2, c3 が実行されます. ただし, このゴールの実行順序は, ボディ部に記述された順番とは無関係です.

また, このゴールのように, 述語定義のボディ部に記述された各ゴールの実行は, 別々のプロセッサに割り当てることによって, 並列に実行することができます. ボディ部のゴールをプロセッサに割り付ける方法については次章で説明します.

4.1.2 ゴールの失敗

KL1 のコミットオペレータ "|"は Prolog のカットオペレータ "!"と似た働きをします. コミットオペレータの前にあるガード部では, ゴールが失敗しても, 実行は他の節に移りますが, コミットオペレータの後ろにあるボディ部では, ゴールが失敗すると, その親のゴールが失敗しプログラム全体の実行が失敗して終ります.

4.1.3 データ型

KL1 で用いることが可能なデータ型には以下のものがあります.

- 変数 (Var,_, etc.)
- 整数 (138, 16'8A, etc.)
- 浮動少数点数 (1.23, 1.0E10, etc.)
- アトム (a, 'ABC', etc.)
- リスト ([1,2,3], [a,[A,b],c|Y], etc.)
- ベクタ ({a,b,c}, {}, a(b), etc.)
- ストリング ("abc", "", etc.)
- モジュール (module#foo, etc.)

データ型の詳細については「PIMOS マニュアル」[3] の 2.1.3 を参照して下さい.

4.1.4 組み込み述語

KL1 で用意された組み込み述語には, ガード部で用いることができるもの, ボディ部で用いることができるもの, その両方で用いることができるものの 3 種類があります.

組み込み述語の詳細については「PIMOS マニュアル」[3] の 2.2 を参照して下さい.

4.2 KL1 プログラムの例

KL1 プログラムについて, 実際の例を見ることにしましょう. いま次のような sample1.kl1 というファイルを作成したとします (KL1 プログラムのファイル名には必ず ".kl1" という属性を付けて下さい).

```
:- module sample1.                          %(1-1)
:- public furui/3.                          %(1-2)

furui([],Even,Odd):- true |                 %(1-3)
        Even=[],
        Odd=[].
furui([N|List],Even,Odd):-                  %(1-4)
        N mod 2 =:= 0 |
        Even=[N|Tail],
        furui(List,Tail,Odd).
furui([N|List],Even,Odd):-                  %(1-5)
        N mod 2 =\= 0 |
        Odd=[N|Tail],
        furui(List,Even,Tail).              %(1-6)
```

ここで,

'=:=' は右辺と左辺の整数演算結果の同値性の比較を示します. これらの記法の詳細は「PIMOS マニュアル」[3] を参照して下さい.

(1-1) はプログラムのモジュール名が 'sample1' であることを示す宣言です. また,
(1-2) はモジュールの外部から参照可能な述語名とその引数の数を示す宣言です.
KL1 では各プログラムファイルの先頭に (1-1),(1-2) のような, モジュール名と外部参照可能な述語名の宣言が必要です.

他のモジュールのゴールを呼び出す時, あるいは, ユーザがリスナ (第9 章参照) からゴールを呼ぶ時には,

"モジュール名: ゴール名"

という形で呼びます. 今ゴール furui を, リスナから

```
?- sample1:furui([2,5,6,8,0,9],Even,Odd).
```

と呼ぶと, それが (1-4) あるいは (1-5) とマッチしますが, ガード部の実行は (1-4) の節に対するものだけが成功します. これによって, (1-4) のボディ部が起動され, 出力変数へのユニフィケーション

```
Even = [2|Tail]
```

と, ゴール

```
furui([5,6,8,0,9],Odd,Tail)
```

の再帰的な呼び出しが実行されます.

注意 : 上のプログラムの例のように, KL1 では必ず, (Even=[N|Tail] のような) 出力変数への値の束縛はボディ部で記述します. これは, 次節で述べるようなプロセス同期の機構からくる制約です.

再帰的に呼ばれたゴール furui([5,6,8,0,9],Odd,Tail) に対しては (1-5) のガード部が成功し, そのボディ部で,

```
Odd = [5|Tail]
```
及び
```
furui([6,8,0,9],Tail,Even)
```

が実行されます.
結局このプログラムの実行によって,

```
Odd = [5,9], Even = [2,6,8,0]
```

という結果が得られます.

上の例で, 節 (1-5) の代わりに次の節を置くことができます.

```
otherwise.
furui([N|List],Odd,Even):-                %(1-5')
        true |
        Odd=[N|Tail],
        furui(List,Tail,Even).
```

ここにある otherwise. は否定を記述するための略記法です. OR の関係にある節の間に otherwise. を挿入すると, KL1 の処理系はその上の節の選択が全て失敗した場合のみ, その下の節を選択します. よって, otherwise. 以下の節のガード部では, その上の節のガード部の条件が全て成り立たないことを確認済みであるとして, その他の条件のみを記述すればよいことになります.

otherwise. を用いた KL1 のプログラムの例をもう一つ示します.

```
:- module sample2.
:- public check/2.

check([], Out):- true | Out = []
check([M|In], Out):- atom(M) |
        Out = [atom|Out1], check(In, Out1).
check([M|In], Out):- integer(M) |
        Out = [integer|Out1], check(In, Out1).
check([M|In], Out):- string(M,_,_) |
        Out = [string|Out1], check(In, Out1).
check([M|In], Out):- list(M) |
        Out = [list|Out1], check(In, Out1).
check([M|In], Out):- vector(M,_) |
        Out = [vector|Out1], check(In, Out1).
otherwise.
check([_|In], Out):- true |                          %(2-1)
        check(In, Out).
```

節 (2-1) はこの述語が失敗しないために必要な処理の定義です. これによって, 例外的なデータは無視されます.

実行例

このプログラムを実行するとつぎのようになります.

```
?- sample2:check([1,a,"ds",23.5,{a,b}],X).

X = [integer, atom, string, vector]

yes.
?-
```

このように, otherwise. 以下の処理によって入力データ 23.5 がスキップされています.

4.3 ストリーム通信とプロセスの同期

次の例を考えましょう.

```
:- module sample3.
:- public go/3, generate/2.

go(Max,Even,Odd):- true |
        generate(Max,Numbers),                   %(3-1)
        sample1:furui(Numbers,Even,Odd).         %(3-2)

generate(0,Numbers):- true |
        Numbers=[].
otherwise.
generate(N,Numbers):- true |
        Numbers=[N|NewNumbers],
        N1 := N-1,
        generate(N1,NewNumbers).
```

ゴール go の第一引数に適当な自然数を与えると，ゴール generate が変数 Numbers にその数から 0 までの自然数のリストを生成します．一方，Numbers はモジュール sample1 で定義したゴール furui に渡され，furui は Numbers 中の自然数を偶数と奇数に分けます．

ここで，generate や furui のように再帰的な呼び出しによって実行されるゴールをプロセスと呼ぶことにします．

今，二つのプロセス (3-1),(3-2) が異なるプロセッサ上で並列に実行されている場合を考えましょう．これらのプロセスは同じ変数 Numbers を共有しています．プロセス generate はこの変数を次々と具体化していき，プロセス furui は具体化された値を先頭から順に参照していきます．

このように，KL1 のプロセスは同じ変数を共有することによって，データの通信を行なうことができます．この例のようなプロセス同士の通信をストリーム通信と呼びます．

furui の実行の進み方が早いときは，furui は generate が共有変数 Numbers にデータを渡すのを待つことになります．このような場合の処理の待ち合わせは自動的に行なわれます．

これを具体的に説明します．プロセス furui が呼び出された時，第一引数が具体化されていないと，

```
furui(List,Even,Tail)                                    %(1-6)
```

の呼び出しが，(1-3),(1-4) あるいは (1-5) にマッチしたときに，呼びだし側の変数 List を [] または，[N|List] に具体化するようなユニフィケーションが生じます．このような場合，KL1 ではユニフィケーションを中断 (Suspend) して，プロセス generate が変数 List を具体化するまで待ちます．これによって，プロセス furui の実行はプロセス generate よりも先に進むことはありません．

このようなプロセスの待ち合わせ機能によって，ユーザはプロセスの同期を自然に記述することができます．

4.4 マージャ

KL1 でプログラムを書くと，複数のプロセスから得られるストリームを一つのストリームに束ねて出力したいことがよくあります．これを行なうのがマージャです．マージャは Prolog の append と同様に KL1 で容易に記述できますが，よく使われるので組み込み述語として用意してあります．

組み込みのマージャは，任意個のストリームを入力として受けとり，入力ストリームのどれかにデータを受けとると，直ちにそれを出力ストリームに流します．従って，append のように入力リストの順番で出力データの順番が決まることはありません．

組み込み述語として用意されているマージャは次の形で呼びます．

```
merge(In,Out)
```

ここで，第一引数は，

```
In = { In1,In2, ... }
```

のような任意個の入力ストリームのベクタです．次にマージャを使用するプログラムの例を示しましょう．

```
:- module sample4.
:- public flatten/2.
flatten([],Out):- true|                          %(4-1)
        Out=[].
flatten([Top|Tail],Out):- true|                  %(4-2)
        flatten(Top,X1),
```

```
        flatten(Tail,X2),
        merge({X1,X2},Out).
flatten(A,Out):- atom(A),A\=[]|          %(4-3)
        Out=[A].
```

このプログラムは，第一引数が入れ子になったリストを受けとり，その中のアトムを一つのリストとして返します．例えば，

```
?- sample3:flatten([a,[b,c],[d,[e,f]]],X).
```

を実行すると，

```
X = [ c,a,f,e,d,b ]
```

のような結果が得られます．ここで，a,b,c,...などのアトムの順番はプログラムから決定できないことに注意しましょう．これは (4-2) の節中で呼ばれる二つの "flatten" の実行の順序が定まらないからです．

マージャは入力ストリーム X1，X2 のいずれかにデータが到着すると，そのデータを出力ストリームに返します．ゴール flatten が第一引数のデータを全て消費すると節 (4-1) が実行され，マージャの入力ストリーム Out が [] で具体化されます．このように，ストリームの Cdr 部が [] で具体化されると，ストリームは "閉じ" ます．マージャは入力トリームが全て閉じると出力ストリームを閉ざして実行を終了します．

sample3 のプログラム (4-2) が実行される度にマージャを生成しますが，これはプログラムの実行効率にとって好ましくありません．プログラムを次のように改めると，この問題は解決します．

```
:- module sample5.
:- public flatten/2.
flatten(In,Out):- true|
        flatten1(In,X),
        merge(X,Out).
flatten1([],Out):- true|
        Out=[].
flatten1([Top|Tail],Out):- true|
        flatten1(Top,X1),
        flatten1(Tail,X2),
        Out = {X1,X2}.
flatten1(A,Out):- atom(A),A\=[]|
        Out=[A].
```

4.5 簡単な入出力の方法

この節では KL1 プログラムのウインドウ及びファイルに対する簡単な入出力の方法を示します．各入出力はそれを行なうためのストーリームを獲得した後，そこに各種のメッセージプロトコルを流すことによって実現できます．

各入出力の方法の詳細は「PIMOS マニュアル」[3] の 3.5, 4, 5, 8.3 を参照して下さい．

4.5.1 標準入・出力

標準入・出力を用いると，プログラムはシェルまたはリスナのウインドウに対して直接入出力を行なうことができます．標準入・出力には，

標準入力，標準出力，標準入出力，メッセージ出力，標準インタラクション

の5種類があります．

入出力を実行するには，各デバイスへのストリームを獲得する必要があります．そのストリームに getc(C)，putc(C) などのメッセージを送ることにより処理を行います．これらのメッセージの詳細は「PIMOS マニュアル」[3] の 3.5 を参照して下さい．

ここでは，標準入力，標準出力，メッセージ出力へのストリームを獲得するためのプログラムの例として，次のようなモジュールを用意します．

```
:- module std_io.
:- public create/3.
:- with_macro pimos.

create(Input,Output,Message):- true|
    shoen:raise(pimos_tag#shell,get_std_in,Input),
    shoen:raise(pimos_tag#shell,get_std_out,Output),
    shoen:raise(pimos_tag#shell,get_std_mes,Message).
```

このモジュールを，

```
?- std_io:create(Input,Output,Message),
   Input=[ ... ], Output=[ ... ], Message=[ ... ].
```

と呼び出すと，

- Input に標準入力に対するストーリームが得られ，
  [...] には buffer:input_filter の提供するメッセージを送ることができます．
- Output に標準出力に対するストーリームが得られ，
  buffer:output_filter の提供するメッセージを送ることができます．
- Message にメッセージ出力に対するストーリームが得られ，
  buffer:output_filter の提供するメッセージを送ることができます．

バッファ及びフィルタについては，「PIMOS マニュアル」3.5 を参照して下さい．

4.5.2 新たなウインドウでの入出力

ウインドウを新たに生成し，それに対して入出力を行なうために，次のようなモジュールを用意しましょう．

```
:- module window.
:- public create/1.
:- with_macro pimos.

create(Window):- true|
```

```
        shoen:raise(pimos_tag#task,
            general_request,General_Request_Device),
        General_Request_Device=
            [window(normal(Window_Request_Device,_,_))],
        Window_Request_Device=
            [create(normal(Window_Device,_,_))],
        Window_Device=[
            set_size(char(50,25),_),
%           set_size(mouse,_),
            set_position(mouse,_),
%           set_position(at(0,0),_),
            set_title("Interactive Window",_),
            activate(_)| Stream ],
        buffer:interaction_filter(Window,Stream).        %(5-1)
```

このモジュールを次のように呼び出したとします.

```
?- window:create(Window),
   Window=[ putl("Hit return to exit."), getl(_) ].
```

このとき, 画面にウインドウの枠が現れるので, マウスをクリックして場所を定めて下さい. このウインドウに return を入力するとウインドウは消えます.

この例の変数 Window がウインドウに対して入出力を行なうメッセージを送るためのストリームです. (5-1) のフィルタによって, このストリームに, 入出力両用フィルタ (「PIMOS マニュアル」 [3] の.5.4 参照) のユーザ側プロトコルを送ることによってデータの入出力を行ないます.

4.5.3 ファイルへの入出力

ファイルへの入出力は, ウインドウと同様ファイルへのストリームを獲得したあと, そこに getc(C), putc(C) などのメッセージを送ることによって行ないます.

ファイルに対して入出力を行なうために, 次のようなモジュールを作りましょう.

```
:- module file.
:- public open/4.

open(FileName,Mode,File,Status):- true|
    shoen:raise(pimos_tag#task,
        general_request,General_Request_Device),
    General_Request_Device=
        [file(normal(File_Request_Device,_,_))],
    File_Request_Device=
        [open(FileName,Access)],
    ( Mode=r ->                                    %(6-1)
        Access=read(Result),
        ( Result=normal(Stream,_,_) ->
            buffer:input_filter(File,Stream),      %(6-2)
            Status=success
        ; otherwise
        ; true -> Status=Result )
```

```
; Mode=w ->
    Access=write(Result),
    ( Result=normal(Stream,_,_) ->
        buffer:output_filter(File,Stream),      %(6-3)
        Status=success
    ; otherwise
    ; true -> Status=Result )
; Mode=a ->
    Access=append(Result),
    ( Result=normal(Stream,_,_) ->
        buffer:output_filter(File,Stream),      %(6-4)
        Status=success
    ; otherwise
    ; true -> Status=Result ) ).
```

ここで，(6-1) は実行の条件分岐のマクロ記法です．マクロ記法については，「PIMOS マニュアル」[3] の 2.1.6 を参照して下さい．

ここに定義されている述語 open(FileName, Mode, File, Status) の各引数は次の通りです．

- FileName : オープンするファイル名 (ストリング型).
  "sample.kl1", ">sys>user>myname>sample.kl1" など．
- Mode : オープンのモード (r, w, a) の指定.
  r → read, w → write, a → append
- File : オープンしたファイルへのストリーム．
- Status : オープンの成功 (success)/ 失敗 (abnormal) を返す変数.

この述語の呼び出しによって変数 File にはオープンしたファイルへのストリームが返されます．これに対する入出力のメッセージは (6-2),(6-3),(6-4) のフィルタのサポートするものが使用できます．

- Mode = r の場合
  buffer:input_filter の提供するメッセージ,
- Mode = w, a の場合
  buffer:output_filter の提供するメッセージ

4.5.4 入出力を利用したプログラムの例

以下に入出力を利用したプログラムの例を示します．このプログラムをコンパイルする前に，上に示した三つのモジュール std_io, window, file をコンパイルしておいて下さい．このプログラムは指定したファイルの内容をウインドウに表示します．プログラムを，

```
?- sample6:demo.
```

で起動すると，ファイル名を聞いてくるので，適当なファイル名を入力して下さい．すると，ウインドウの枠が現れるので，マウスで位置を決めて下さい．そのウインドウにファイルの内容が表示されます．

```
:- module sample6.
:- public demo/0.

demo:- true|
    std_io:create(In,Out,Mes),
    In=[prompt("File name ? "),getl(FName)],
    Out=[putt('Openning file : '),putl(FName)],
    Mes=[putt(Status),nl],
    type_file(FName,Status).

type_file(FName,St):- string(FName,_,_)|
    file:open(FName,r,File,St),
    ( St=success ->
        window:create(Win),
        File=[getl(Line)|RF],
        type(Line,RF,Win)
    ; otherwise
    ; true -> true ).

type(-1,File,Win):- true|
    File=[],
    Win=[prompt("*** Hit Return key to exit.***"),
            getl(_)].
otherwise.
type(L,File,Win):- true|
    Win=[putl(L),flush(_)|W],
    File=[getl(N)|F],
    type(N,F,W).
```

4.6 デッドロック

ストリーム通信とプロセスの同期の節で述べたように，あるゴールのボディ部の実行が未定義の変数を具体化しようとした時，そのゴールの実行は中断されます．

あるゴールが待っている変数が，他のどのゴールによっても具体化されない時，中断されたゴールの実行は永遠に再開されません．このような状態をデッドロックと呼びます．

デッドロックは様々な原因で起きる可能性があります．例えば sample4 の例の (4-1) の節を

```
flatten([],[]):- true|true.                    %(4-1')
```

とすると，この節と呼び出し側のゴール?- flatten([],X1) がマッチした時，呼び出し側の変数 X1 を [] で具体化するユニフィケーションが生じるので，KL1 の処理系はこのユニフィケーションを中断します．ところが，変数 X1 は他のプロセスから具体化されないので，デッドロックを起こします．

また，同じ節を次のように記述すると，

```
flatten([],Out):- true|true.                   %(4-1")
```

マージャの入力として参照される変数 Out が永遠に具体化されず，デッドロックとなります．この例のように，KL1 の節中に一度しか現れない変数が誤って生じた場合，それがデッドロッ

クの原因となる可能性があります．このような変数は，CAL の変数チェックの機能を用いて調べることができます．（第10章 参照）

プログラムの実行時にデッドロックが生じた場合の対処の方法は，第6章を参照して下さい．

第 5 章

負荷分散

本章では，負荷分散の方法について簡単に説明します．現在の PIMOS では，プログラムの負荷分散は KL1 の各ゴール毎に，それを実行するプロセッサの番号を与えることで実現します．プロセッサ番号を与えられていないゴールは，それを呼び出したゴールと同じプロセッサ上で実行されます．

5.1 負荷分散の方法

負荷分散させるには，各プログラムのボディ部のゴールに対して，そのゴールを実行するプロセッサ番号を次のように指定します．

```
goal@process(プロセッサ番号)
```

ここで，プロセッサ番号は 0 以上の整数です．

プログラム例

```
:- module sample7.
:- public foo/0.

foo:- true |
        a@processor(0),
        a@processor(1),
        a@processor(2).
a :- true | true.
```

プロセッサ番号は変数でも指定できます．変数がプロセッサ番号に具体化されると，そのゴールはそのプロセッサに移動先します．

```
:- module sample8.
:- public foo/3.

foo(P1,P2,P3):- true|
        a@processor(P1),
        a@processor(P2),
        a@processor(P3).
a :- true | true.
```

5.2 負荷分散の例

次に，もう少し具体的な負荷分散の例を示します．

```
:- module sample9.
:- public distribute/1.

distribute(N):- true |
        current_processor(_,X,Y),                    %(1)
        PEs := X*Y,                                  %(2)
        fork(N, PEs, 0)@priority(*,4096).            %(3)

fork(0, PEs, PE):- true | true.
otherwise.
fork(N, PEs, PE):- true |
        PeNo := PE mod PEs,
        job_ext@processor(PeNo),                     %(4)
        NextPE := PE+1,
        N1 := N-1,
        fork(N1, PEs, NextPE).

job_ext :- true | job@priority(*,2000).              %(5)
job :- true | true.
```

(1) は組み込み述語で，それが実行されているプロセッサの番号と，X,Y それぞれの方向の使用できるプロセッサの台数を返します．
(2) で得られる 'PEs' は，使用できるプロセッサの台数です．
(3) では負荷分散を行なうゴール 'fork' を最高の優先度で呼んでいます．
(優先度の指定については「PIMOS マニュアル」[3] の 2.1.4 を参照して下さい).
(4) では，実際に仕事をするゴールを各プロセッサに投げています．
(5) は実際の仕事をするゴールを 'fork' よりも低い優先度で呼んでいます．

このプログラムに対して，次のようなゴールを呼ぶと，

```
?- sample9:distribute(8).
```

ゴール 'job' が 8 回実行されます．使用できるプロセッサが仮に 4 台だとすると，各ゴールは，プロセッサ番号が 0,1,2,3,0,1,2,3 の順でプロセッサに割り付けられます．ゴール 'job' よりも，ゴール 'fork' の方が優先度が高いため，(0 番のプロセッサでは)，'fork' が全てのゴールを投げた後で，実際に仕事をするゴール 'job' が実行されます．

5.3 効率の良い負荷分散を実現するために

マルチ PSI では，各プロセッサが個別のメモリを持つため，複数のプロセッサ上に分散されたゴールが，他のプロセッサ上にあるデータを参照するとプロセッサ間で通信が起こります．多くのデータを通信し合うゴールが別々のプロセッサ上にあると，通信のオーバーヘッドが大きくなり，並列に実行することによる実行速度の向上を抑える結果となることもあります．

効率の良い負荷分散を実現するために，次の点を考慮して下さい．

- 処理の小さいゴールは負荷分散させない．
- ゴールはなるべくデータのあるプロセッサ上で実行させる．
- アクセスの多いデータベースは，プロセッサ毎に分散管理させる．

- ゴールについていくデータはなるべくコンパクトにまとめる.
  (そのゴールが返すデータもできるだけコンパクトにまとめてください.)
- 負荷分散を行なうゴールは, 実際に仕事をするゴールよりも高い優先度で実行する.

負荷分散についての詳細は 「KL1 プログラミング」 [5] の 11.5 を参照してください.

第 6 章

KL1 プログラムの実行方法

この章では KL1 プログラムのコンパイル，実行，デバッグ，などの手続きを順を追って説明します.

6.1 KL1 プログラムのコンパイル

KL1 プログラム (*.kl1 ファイル) をコンパイルするには,

- PIMOS 上のセルフコンパイラ
- クロスシステム CAL

の 2 通りの方法があります. プログラムのデバッグの段階では PIMOS 上のセルフコンパイラを使用して下さい. CAL については，第 10 章で説明します.

6.2 セルフコンパイラの使い方

6.2.1 セルフコンパイラの起動

コンパイラは PIMOS のシェルから起動します. まず, PIMOS を起動して, ShellUser でログインして下さい.

USERS	Shell for ShellUser
User Name>>ShellUser	Shell>

ここで，カレントディレクトリを見ます.

```
Shell> cd
Illeegal filename for take .              : 無視して下さい.
The value of the task:directory is "sys>user>Taro".
Shell>
```

カレントディレクトリの変更は，フルパス名をストリングで指定して行います．

```
Shell> cd "ディレクトリ名"
```

次に，

```
Shell>  compile
```

と入力して下さい．シェルウインドウからコンパイラーを起動します．

```
** KL1 Compiler **
COMPILE>
```

このプロンプトに，コンパイルしたいファイルの名前を入力します (属性の ".kl1" は省略して下さい)．ファイルが複数あるときは，カンマで区切って一度に入力できます．
例えば， sample1.kl1，sample2.kl1，sample3.kl1 を一度にコンパイルするには，

```
COMPILE> sample1, sample2, sample3
```

と入力します．
カレントディレクトリ以外にあるファイルはフルパス名をストリングで入力して下さい．カレントディレクトリを変更したい場合は， !exit の入力によって一旦コンパイラを終了し， cd コマンドで変更した後，再びコンパイラを起動して下さい．

注意：Psedo マルチ PSI ではメモリの容量が不足する場合があるので，多くのファイルを同時にコンパイルすることはお勧めできません．コンパイルは何回かに分けて実行できます．ただし，分割コンパイルは以下のような原則に従って実行して下さい．

- 相互に参照し合うファイル (モジュール) ごとにグループを作る．コンパイルはグループごとに実行する．
- コンパイルは下位のモジュールのグループから実行する．すなわち，まず，他を参照しないグループを先にコンパイルし，次に，既にコンパイルされているモジュールのみを参照するグループをコンパイルする．

コンパイルの実行例

```
Compile File : **psi::>sys>**>sample1.kl1.1

Compile Module : sample1
furui/3
Compile Succeeded : sample1

Compile File : **psi***::>sys>***>sample2.kl1.1

Compile Module : sample2
check/2
Compile Succeeded : sample2

Compile File : **psi***::>sys>***>sample3.kl1.1

Compile Module : sample3
```

```
go/3
generate/2
Compile Succeeded : sample3

"sample1" Loaded
"sample2" Loaded
"sample3" Loaded

Compilation(s) Succeeded
COMPILE>
```

と表示され，コンパイルが完了します．
コンパイラを終了するには

```
COMPILE>!exit
```

を入力して下さい．

6.2.2 バッチコンパイル

```
Shell> compile(["sample1","sample2","sample3"])
```

のような入力によって，シェルプロンプトからバッチコンパイルを実行することもできます．コマンドファイル（第 8 章参照）を用いる場合などに利用できます．

6.3 プログラムの実行

コンパイルしたプログラムの実行は，リスナあるいは，シェル（第 8 章参照）を用いて行ないます．以下では，リスナを用いたプログラムの実行とデバッグの方法を説明します．

6.3.1 リスナの起動

リスナは次の二つのどちらかの方法で起動できます．

- 利用環境設定ウインドウに ListenerUser でログインします．

- Shell から listener コマンドで起動します，（この場合は，マウスでリスナウインドウの位置と大きさを決めます．）

USERS	Shell for ShellUser
User Name>>	Shell> listener &

シェルから起動したリスナはジョブとして管理できるので (第8章参照), 万一プログラムが暴走しても，リスナを強制的に終了することができます．したがって，デバッグ時にはシェルから起動することをお勧めします．

6.3.2 ゴールの実行

リスナは Prolog インタプリタとよく似た役割を果たします．ゴールは必ず，

?- モジュール名: ゴール.

の形で呼びます．

```
?- sample1:furui([1,2,3],E,O).
 E = [2]
 O = [1,3]
 yes.
```

リスナの起動時には変数プール (第 9 章参照) が働いており，一旦値の束縛された変数 (この例では E と O) はその値を持ち続けます．試しに，次のユニフィケーションを実行してみましょう．

```
?- X = [E,O].
 X = [[2],[1,3]]
 yes.
```

誤って変数プールに登録されている変数を用いて別のゴールを実行すると，そのゴールが失敗することがあるので注意しましょう．
例えば，上の例の E, O を用いて別のゴール furui を実行してみると，

```
?-  sample1:furui([5,2,3],E,O).
 Unification failure>> 5
                With > 1
                Pe   > 0
no.
```

のように失敗します．これは，すでに [1,3] に具体化されている変数 O を [5|Tail] とユニファイしようとしたことによります．
変数プールに登録されている変数を全て解放するには，ピリオドを入力します．

```
?- .
```

変数プールを必要としない場合は，forget. を入力して下さい (その逆は remember. です).

```
?- forget.
```

6.4 プログラムのデバッグ

KL1 プログラムのデバッグのツールとして，トレーサとインスペクタが用意されています.

6.4.1 トレーサ

KL1 のゴールには，Prolog のような実行の逐次性がありません. ある時点で実行可能なゴールはどれでも実行される可能性があります. そこで，リスナから起動するトレーサは，どれをトレースしたいか (あるいは全部) をユーザが指定することにより，並列に実行される他のゴールとは関係なく，指定されたゴールの実行のみをトレースするようになっています.

トレーサの起動

```
?- trace.
yes.
{trace}
?-
```

モジュール sample3 の述語 go の実行をトレースしてみます.

```
?- sample3:go(3,E,0).
@0@4096 0    sample3:go(3,A,B)              (a)
1      * (1)   generate(3,C)                (b)
2      * (2)   sample1:furui(C,A,B)?        (c)
```

(a) の行の @0@4096 0 の最初の 0 はそのゴールをレデュースしたプロセッサの番号，次の 4096 はゴールのプライオリティ，その次の 0 はゴールに与えられた識別番号です.
(b) と (c) の行は，ゴール go の子ゴールの表示です. 各行の先頭の 1，2 はゴールの識別番号，次の * はこれからそのゴールをトレースすることを示す印です. その次の (1),(2) はトレースコマンドを適用する子ゴールを指定するための番号です (トレーサの表示の詳細については「PIMOS マニュアル」10.6.2 を参照して下さい).

上の例では，(1),(2) のどちらの子ゴールもトレースモードが指定されていますが，トレースする必要のない子ゴールの番号を

```
t N1,N2,...
```

のように入力するとその番号で指定された子ゴールのトレースモードが反転します.
例えば，(c) のプロンプト? に対して，次のように入力すると，

```
2      * (2)   sample1:furui(C,A,B)? t 1    (c)
@0@4096 0    sample3:go(3,A,B)
1        (1)   generate(3,C)                 (d)
2      * (2)   sample1:furui(C,A,B)?         (e)
```

(d) の行の * 印が消えます．これで子ゴール (1) のトレースは抑制されます．（もう一度 t 1 を入力するともとに戻ります．）

次に，(e) のプロンプトで return を入力すると，実行が継続します．

```
@0@4096 2    sample1:furui([3,2,1],A,B)
                 B= [3|D]
3      * (1)     furui([2,1],A,D)?
```

ここで，この実行過程をこれ以上トレースする必要がない場合は，x コマンドを入力します．これによって，子ゴールのトレースモードが反転され，実行が継続します．

```
3      * (1)     furui([2,1],A,D)? x
 E = [2]
 O = [1,3]
yes.
?-
```

この例では，トレースが終了します．トレースモードが指定されているゴールが他にもある場合は，その実行過程をトレースします．

全てのゴールのトレースやめるにはコマンド notrace を入力します．
また，ヘルプメニューは help で表示されます．
トレースコマンドの詳細は「PIMOS マニュアル」10.6.3 を参照して下さい．

トレーサを終了するには，

```
?- notrace.（ntr. でもよい．）
```

と入力します．

トレーサには，必要なゴールのみをトレースするための機能として，二種類のスパイ機能が用意されています．

- リダクションスパイ
  指定したゴールがリデュースされたときに，それを報告します．
- フォークスパイ
  指定したゴールを子ゴールとして持つゴールをリデュースしたときに，それを報告します．

スパイゴールの指定は，

```
?- spy モジュール名: 述語名 / 引数の数.
```

のように行ないます．次に，スパイのモードを指定します．

```
?- spy_reduction.（sr. でもよい．）
あるいは，
?- spy_fork.（sf. でもよい．）
```

スパイの実行例

```
?- spy sample1:furui/3.
Spypoint set on sample1:furui/3
 yes.

?- sr.
 yes.

{SpyReduction}
?- sample3:go(3,E,O).
<0> sample3:go(3,A,B) -->
@0@4096      + sample1:furui([3,2,1],A,B)
                    B=[3|C]
1    + * (1)      furui([2,1],A,C)?
```

スパイゴールの実行時にも，通常のトレースコマンドが使用できます．
次のスパイゴールの実行までトレースを抑制したい場合は，以下のコマンドを入力して下さい．

- sr コマンド：リダクションスパイの指定
- sf コマンド：フォークスパイの指定

スパイゴールの指定の解除は，

```
?- nospy モジュール名:述語名/引数の数.
```

です．また，全スパイゴールの指定の解除は，

```
?- nospy.
```

です． spy モードを抜けるには，notrace． と入力してください．

6.4.2 インスペクタ

インスペクタは任意の KL1 データの値の表示および，変数プールへの受け渡しを行なうためのツールです．インスペクタはリスナのトップレベル及び，トレースの途中から呼び出すことができます．インスペクタの詳細な説明は「PIMOS マニュアル」[3] 第 11 章を参照して下さい．

- リスナからの起動．
  リスナから起動した場合，変数プールに保持されている変数の内容を調べることができます．変数プールに登録されている変数のリストは，

  ```
  ?- list.
  ```

  の入力によって表示できます．例えば，変数 X のデータを調べたいとき，

  ```
  ?- inspect(X)
  ```

  としてインスペクタを起動します．これに対して，ここで，変数 X に {1,2,3,a(b)} が束縛されているとすると，それが次のように表示されます．

  ```
  {1,2,3,a(b)}>
  ```

ここで，このデータに対して，インスペクタの提供しているデータ調査コマンドが適用できます．また，インスペクタの制御コマンド (exit など) も適用できます．例えば，このデータの 3 番目の要素 (最初の要素は 0 番目) を表示するには，次のようにコマンド me を用います．

```
{1,2,3,a(b)} > me 3
3 : a(b)
```

- トレースの途中からの起動
  トレースの途中からインスペクタを起動するには，

  ins ゴール番号

  を入力します (ゴール番号は省略可能です)．
  例

```
?- sample3:go(3,E,0).
 @0@4096 0    sample3:go(3,A,B)
 1      * (1)   generate(3,C)
 2      * (2)   sample1:furui(C,A,B)? ins 1
 generate(3,A)>
```

インスペクタで使用できるコマンドは help によって知ることができます．
また，インスペクタを終了するには， exit を入力してください．

6.5 再リンカーの使用

プログラムを一度コンパイルした後，デバッグによって一部のモジュールを更新することがよくあります．このとき全てのモジュールを最初からコンパイルしなおす必要はありません．更新したモジュールのみをコンパイルしてください．コンパイルしなおしたモジュールを参照するモジュールがある場合は，参照関係も更新する必要があります．このために，再リンクを行います．

再リンクはシェルから実行します．

```
Shell>relinker([sample1, sample2,...])
```

引数のリストにはモジュール名 (アトム) のリストを入れます．

6.6 コンパイル結果の保存と再開

KL1 プログラムの実行に必要なコンパイル結果の情報は， PIMOS のシャットダウンによって失われます．次回に PIMOS を立ち上げた時に，プログラムを再びコンパイルせずに実行するためには，コンパイル結果の情報をファイルに保存しておく必要があります．このために PIMOS ではアンローダを提供しています．

アンローダはシェルで実行します．

```
Shell>unload([sample1,sample2,...],"module.unl")
```

これによって， sample1, sample2, ... のコンパイル結果を "module.unl" というファイルに保存します．

アンローダで保存したモジュールの情報を PIMOS 上に載せるにはローダを使用します．

ローダもシェルで実行します．

```
Shell>load("module.unl")
```

これによって， sample1, sample2, ... のモジュールをシェルやリスナから呼ぶことができます．

6.7 デッドロックの対処法

実行しているプログラムが終了しない場合は，デッドロックが生じている可能性があります．デッドロックはガーベッジ・コレクションのときに検出されますが，ガーベッジ・コレクションはメモリの空き領域が不足しない限り自動的には起動されません．そこで，次のような状況がおきた場合にはデッドロックの疑いがあるとして，ガーベッジ・コレクタをコマンドにより起動して下さい．

- CSP コンソールパネル（「CSP 操作説明書」[2]7.2 参照）の Silent Mode が OFF で，状態監視パネルが黒くなっている．
- プログラムの実行を Control-C で何回か中断し，コマンド r を入力し，リダクション数を調べても値が増加しない（第9 章参照）．

6.7.1 ガーベジコレクタの起動

ガーベジコレクタを起動するには，次の二通りの方法があります．

- リスナに Control-C を入力し，g コマンドで起動する．
- シェルに gc を入力して起動する．

GC はどこから起動してもかまいません．デッドロックが検出されると，プログラムを実行したウインドウに報告されます．（シェルに報告される場合は，プロンプト Shell> に return を入力して下さい．）

6.8 プログラムの評価

PIMOS では，KL1 プログラムの実行時の性能を評価するために，以下の機能を提供しています．

- タスクの資源消費量（reduction 数）及び，実行時間の表示
  リスナで，

  ```
  ?- statistics.
  ```
  （st. でもよい．）

  を入力すると，タスクの資源消費量および実行時間を表示するモードに入ります．

  ```
  ?- sample3:go(3,E,O).
  E = [2]
  O = [1,3]
  24 reductions
  118 msec
  yes.
  ```

  （表示モードの解除は nostatistics. または nst. です．）

- パフォーマンス・メータ（Pseudo Multi PSI では使えません．）
  マルチ PSI の各 CPU の稼働率を表示するためのツールとして，パフォーマンス・メータが用意されています．パフォーマンス・メータはシェルから起動します．

  ```
  Shell> pmeter
  ```

  詳細は「PIMOS マニュアル」[3] 付録 B を参照して下さい．

第 7 章

トラブル対処法

この章では，ユーザの KL1 プログラムによって PIMOS あるいは SIMPOS にトラブルが生じた場合の対処法について簡単に説明します．トラブル対処の詳細な説明は「CSP 操作説明書」[2] の第 12 章を参照して下さい．

7.1 PIMOS 上のトラブル

- KL1 のプログラムの暴走やその他の理由で，PIMOS が正常に動作しなくなった場合，CSP コマンドウインドウに Control-C を入力して下さい．プロンプト $ が現れます．$ が現れないとき，つまり Control-C が利かない場合は，左上の状態監視パネルをマウスでクリックした後，ABORT キーを入力し，現れたメニューで，Return to CSP Toplevel を選択して下さい．プロンゾトが現れたら，exit コマンド

  ```
  $ exit
  ```

  で一旦 CSP を終了し，システムメニューから (Pseudo) Multi-PSI を起動しなおして下さい．

- KL1 のプログラムの実行中にメモリの不足や，その他のエラーが原因で PIMOS が停止し，CSP コマンドウインドウにプロンプト $ が現れた場合も同様にして PIMOS を再起動して下さい．

- メモリの不足によって PIMOS が停止した場合，CSP コマンドウインドウに，

  ```
  Copying GC Memory Shortage : xxx words
  ```

  というエラーメッセージが表示されます．Pseudo Multi-PSI ではこのような場合，コンフィギュレーションファイルを変更して，使用するプロセッサの台数を減らしてください．一旦シャットダウンして PIMOS を立ち上げなおすと，同じプログラムが正常に動作する場合もあります．コンフィギュレーションファイルの変更については，「CSP 操作説明書」[2] の第 4 章，6 章を参照して下さい．

7.2 SIMPOS 上のトラブル

ユーザの KL1 プログラムの実行に何らかの障害が起こると，SIMPOS が停止する場合があります．この時，PSI-II の CSP に制御が移り，コンソールウインドウ (PSI の立ち上げ時に表示されるウインドウ) が現れて，次のような，メッセージとプロンプト $ が表示されます．

```
%F-SRGHLT, CPU HALT BY STOP REGISTER
```

```
MPC = ....                PMPC = ....
CIR0 = ....
CIR1 = ....

  ......................

$
```

このような場合の対処法を以下に示します.

1. SIMPOS のコンソールのプロンプトで deb コマンドを入力し，デバッグモードに入ります.

   ```
   $deb
   ```

2. プロンプトが > に変わるので，ここで，go と入力します.

   ```
   >go
   ```

3. CTRL キーと BREAK キーを同時に押すと，コンソールウインドウが隠れます. (Pseudo) マルチ PSI の CSP コマンドウインドウに戻って下さい.

4. CSP コマンドウインドウにプロンプト $ が表示されていない場合は Control-C を入力してプロンプト $ に戻ります.

5. このプロンプトに対し，exit コマンドで CSP を終了します.

   ```
   $exit
   ```

6. システムメニューからもう一度 (Pseudo) Multi-PSI を起動すると，PIMOS は正常に動きます.

- 上の 4,5 の操作で，(Pseudo) Multi-PSI を終了できない場合は，SIMPOS のシステムメニューから process を選択し，プロセスマニュピュレータ (「SIMPOS 操作説明書」[7] 参照.) を起動し，(Pseudo_)multi_psi_main_program に対して，exterminate を実行した後，SIMPOS をシャットダウンして下さい.

- 上の 3 の操作で，SIMPOS が正常に動作しない場合は，BREAK キーを押して SIMPOS のコンソールのプロンプト '>' に戻ります. その後，次の手順で SIMPOS をシャットダウンした後，SIMPOS を再起動して下さい.

  ```
  >go/tr
  @q simpos
  @< cr >
  DC> OK to SHUT DOWN ??  [Y/N] -> [1/0]
  RC> 1
  .....
  >q
  $bo
  ```

- 以上の操作が全て実行できない場合は，リセットボタンを押して下さい.

第 8 章

シェル

この章では，シェルの主な機能について簡単に説明します.

8.1 初期化ファイルの設定

第 2 章で示した利用環境設定ファイルの変更によって，シェル起動時の初期ディレクトリは，各ユーザのホームディレクトリに設定されます．そのディレクトリの下に，

```
shell.com
```

というファイルを作り，そこにシェルの環境変数の設定や，コマンドを書き込んでおくと，シェルの起動時にそのコマンドが自動的に実行されます.
"shell.com" の例

```
set history = 20
set rscinc = infinite
set prompt = "% "
listener(mouse,char(50,20),"font:test_11") &
```

この例では，シェル起動時に以下が実行されます.

- ヒストリースタックの長さを 20 とする.
- 資源不足時に追加する資源量 (「PIMOS マニュアル」 8.2.6 参照) を infinite とする.
- プロンプトを "% " とする.
- リスナをバックグラウンドで起動する.

8.2 シェルの基本的なコマンド

- シェルの終了
  `Shell> exit`
- カレントディレクトリの変更
  `Shell> cd ">sys>csp>SYSTEM.DIR"`
- コマンドファイル (実行したいシェルコマンドを書き込んだファイル) の実行
  `Shell> take "test.com"`
- コマンド一覧の表示
  `Shell> help`

8.3 ヒストリー機能

ヒストリー機能を使うためには，まずヒストリースタックの長さを設定します.

```
Shell> set history=20
```

この設定は初期化ファイルに書き込んでおくと便利です.
ヒストリースタックの内容は，

```
Shell> history
```

で表示されます.
ヒストリースタックのコマンドは，表示されたコマンドの番号を入力すると実行できます．ただし，直前のコマンドは 0 を入力すると実行できます.

```
Shell> 3
```

8.4 タスクの実行

シェルのプロンプトでは，シェルコマンドと KL1 のゴールを実行することができます．(ただし，シェルからのゴールの呼び出しには，変数を用いることができません．また，述語名が go で与えられているゴールの呼び出しは，go を省略し，モジュール名と引数のみでそのゴールを呼ぶことができます．)

8.4.1 タスクの実行方式

- タスクの実行には，フォアグラウンドでの実行とバックグラウンドでの実行があります.
- タスクには標準入力，標準出力，標準メッセージ出力があり，デフォルトでシェルウインドウが指定されていますが，必要に応じてこれをファイルまたは新しいウインドウに変更できます.
- パイプ "|" を利用して複数のタスクを結合できます．パイプによって結合されたタスクをジョブと呼びます.

8.4.2 タスクの実行例

- フォアグラウンドでのリスナの実行
  `Shell> listener`
- バックグラウンドでのリスナの実行
  `Shell> listener &`
- 複数のタスクの投入
  `Shell> listener & listener & compiler`
- ユーザ定義の KL1 プログラムの実行
  `Shell> sample6:demo`
- 標準入力をファイルに指定したコマンドの実行
  `Shell> cat <= file("me:sample1.kl1")`
  これで，ファイルの内容を表示します．(ただし， cat は空白行をファイルの終了とみなします．)

- 標準出力をファイルに指定したコマンドの実行
  `Shell> cat => file(^"file.new")`
  キー入力をファイルに出力します (return キーで終了します).
  ファイル名の前にある ^ は write モードの指定です. これを省略すると append モードになります.

- 標準入力をウインドウに指定し, 標準出力をファイルに指定したコマンドの実行
  `Shell> cat <= window("window name") => file(^"file.new")`

- 標準メッセージ出力をファイルに指定した実行
  `Shell> compile(["sample1","sample2","sample3"]) -> file(^"compile.log")`
  バッチコンパイル (「PIMOS マニュアル」 [3] の 7.2.4 参照) を実行します.

- パイプの利用
  `Shell> grep(":-") <= file("sample1.kl1") | lc`
  ファイル中に文字列 ":-" を含む行を取り出し, その数を数えます.

8.5 ジョブの管理

シェルには, ジョブの管理を行なうコマンドとして以下のものがあります.

- Control-C : フォアグラウンドのジョブをバックグラウンドに移します.

```
Shell> compile
COMPILE> ^C
Shell>
```

- status : 起動中の全ジョブの情報を表示します.

```
Shell> status
       1 --> running compile
```

- stop JobNo : JobNo で指定したジョブを停止します.
  `Shell> stop 1`

- fore JobNo : JobNo で指定したジョブをフォアグラウンドに移し, そのジョブが止まっていた場合には再開します.
  `Shell> fore 1`

- back JobNo : JobNo で指定したジョブをバックグラウンドに移し, そのジョブが止まっていた場合には再開します.
  `Shell> back 1`

- kill JobNo : JobNo で指定したジョブを終了します.
  `Shell> kill 1`

- kill all : 全てのジョブを終了します.

8.6 シェル・ユーティリティプログラム

シェル・ユーティリティプログラムはシェルからの起動を前提として作成するプログラムです．プログラムの実行を開始する述語の名前を go に統一し，そのモジュール名をコマンドとして入力することによって起動します．

ユーザは各自のシェルユーティリティプログラムによって，新たなシェルコマンドを作ることができます．（「PIMOS マニュアル」[3], 8.3 参照）
例えば，Shell ユーティリティの一つである Echo プログラムは次のように記述されています．

```
:- with_macro pimos.
:- module echo.
:- public go/1.

go(Arg):- true|
        shoen:raise(pimos_tag#shell,get_std_out,Out),
        Out = [putt(Arg),nl,flush(_)].
```

実行例

```
Shell> echo([a,b,c])
[a,b,c]
Shell> echo("a b c")
"a b c"
```

第 9 章

リスナ

第 6 章で示したように，リスナにはトレース，インスペクタなどのデバッグのための機能，デッドロック検出機能，変数プールなどの機能があります．この章では，その他のリスナの機能のうち，主なものについて簡単に説明します．

9.1 コマンド

- リスナの終了
  ?- exit.

- 公開述語の呼びだし

  ?- "モジュール名":"ゴール".

- 局所述語の呼びだし
  プログラム中で，公開述語宣言 (public 宣言) をしていない述語も次のようにして呼び出すことができます．

  ?- "モジュール名"::"ゴール".

  例

  ?- sample9::fork(100,32,0).

- 組み込み述語の呼び出し
  KL1 のボディ部で実行可能な組み込み述語も呼び出すことが可能です．
  例

  ?- current_processor(PE,X,Y), PEs := X*Y.

- ヘルプ

  ?- help.
  ?- help all.
  ?- help basic.
  など．

- statistics コマンド (st でもよい)
  ゴールの消費した資源量 (リダクション数) と，実行時間を表示するモードを ON にします．(逆は nostatistics または nst)

- デフォルトモジュールの設定

```
?- default_module "モジュール名".
```

によって，モジュール名のデフォルトを設定することができます．デフォルトとして指定されたモジュール中のゴールは，ゴール名のみで呼び出すことができます．
例

```
?- default_module sample9.
```

公開述語の呼びだし．

```
?- distribute(10).
```

局所述語の呼びだし．

```
?- :fork(10,8,0).
```

- その他のコマンド
  各コマンドの詳細，及びその他のコマンドについては「PIMOS マニュアル」[3] の 10.5 を参照して下さい．

9.2 実行の中断

リスナで実行中のプログラムは，Control-C(アテンションキー) の入力によって中断することができます．実行を中断した状態で，次のコマンドが使用できます．

- < *cr* > : 実行の継続
- g : ガーベジコレクタの起動と実行の継続
- r : 資源消費量 (reduction 数) の表示
- s : プログラム (タスク) の実行状態の表示
- @ : プログラム (タスク) の資源状態の表示
- t : トレースを指定したゴールのうち，まだリデュースされていないゴールの表示
- k : トレース中に実行を保留したゴールの表示
- a : 実行の放棄
- b : 実行の中断 (exit. で復帰)
- h : コマンドの一覧

9.3 標準入出力

標準入出力はデフォルトでリスナウインドウが設定されていますが，これをファイルあるいは新たなウインドウに指定することができます．
例

- 標準出力をファイル "demo.out" に指定

```
?- sample6:demo => file(^"demo.out").
```

- 標準入力を新たなウインドウに指定

```
?- sample6:demo <= window("any name").
```

- 標準メッセージ出力をファイル "demo.mes" に指定

```
?- sample6:demo -> file(^"demo.mes").
```

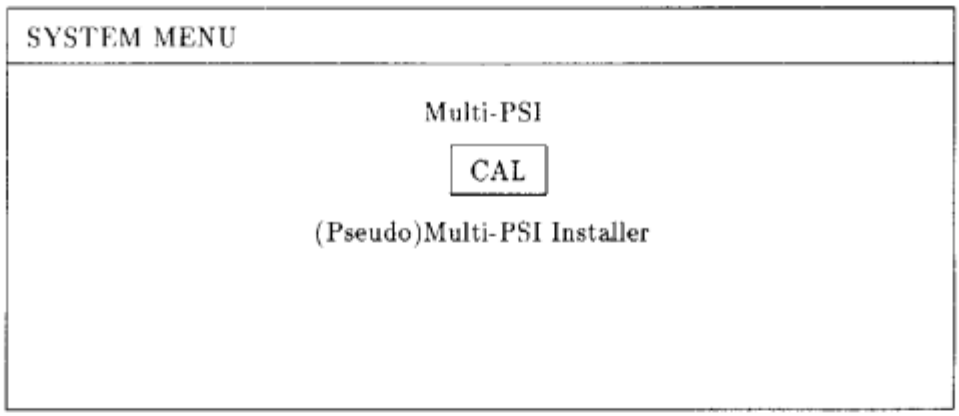
第 10 章

CAL

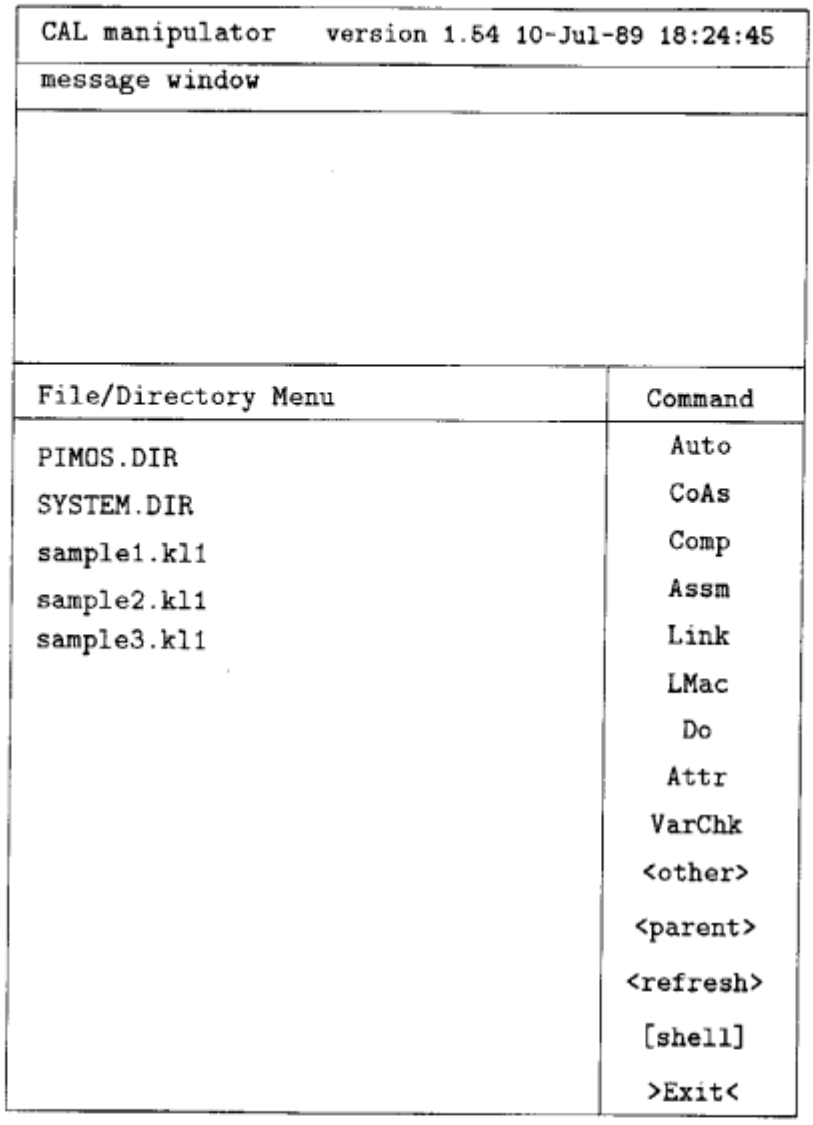
CAL は KL1 プログラムのコンパイル・アセンブル・リンクを実行して，マルチ PSI の CSP がロードできるバイナリのオブジェクトファイル (*.kbn) を作るシステムです．PIMOS のプログラムはすべて KL1 で記述され，CAL によってコンパイルされています．あなたの KL1 プログラムを CAL でコンパイルし，PIMOS 本体のオブジェクトにリンクすると，PIMOS を立ち上げ時に，そのプログラムの実行に必要な情報が CSP 上にロードされます．

10.1 CAL の起動

1. システムメニューで "CAL" を選択します．



2. メニューを選択すると，マウスがウインドウ指定の状態に変わります．(SIMPOS の立ち上げ後，最初に CAL を起動する際にはしばらく時間がかかります．) そこで，ボタンをクリックしてウインドウの位置と大きさを決めて下さい．



CAL は次のような機能を持つ三つのサブウインドウを持ちます.

- message window
  実行状態の表示, コマンドの入力を行うウインドウです.
- File/Directory Menu
  カレントディレクトリにある KL1 ソースファイル (*.kl1), コンパイル・アセンブル実行後のコードファイル (*.rkp), サブディレクトリを表示するウインドウです. マウスによってコンパイル・アセンブル (CoAs) を行なう (複数の) ファイルを指定します.
- Command
  CAL のコマンドのメニューを表示するウインドウです. 各コマンドはマウスで選択して実行します.

10.2 CAL の使い方

CAL によるコンパイル・アセンブル・リンクは次のような手順で行ないます. まず, 各プログラムをコンパイル・アセンブルしてコードファイル (*.rkp) を各ファイル毎に作ります. 次に, CAL のパーシャルリンク (plink) の機能によってそれらを PIMOS 本体のオブジェクト

```
pimos:pimos.kbn
```

にリンクします.

1. ディレクトリの変更が必要なときは, File/Directry Menu で, <other> を選択します. message window が入力待ちになるのでディレクトリ名を入力します.
   例

   ```
   other>me:
   ```

2. コンパイル・アセンブルしたいファイルをマウスで選択します. File/Directory Menu の表示の移動は, ファイルマニピュレータと同様の操作で行ないます. 一度に複数個のファイルが選択できます.

3. Command ウインドウの CoAs を選択してコンパイル・アセンブルを実行します.

4. message window に実行の様子が表示されます. 正常にコンパイルされた場合, 次のような表示になります.

   ```
   **psi***::>sys>user>***>sample1.kl1.1
   *Compile Start ....
   furui/3,END
   *Assemble Start ....
   End.
   *Output File Name
   **psi***::>sys>user>***>sample1.rkp

   *CoAs end !!
   ```

5. Command ウインドウの Link を選択します.

6. PIMOS 本体のオブジェクトに対してパーシャルリンクを実行することを指定します.

   ```
   >> plink pimos:pimos.kbn

   ok.
   P>>
   ```

7. 先ほどコンパイルしたファイル (*.rkp) を一つずつロードします.

   ```
   P>> load  sample1.rkp
   **psi***::>sys>user>***>sample1.rkp.1
    StartPC:XXXXXXXX
     EndPC:YYYYYYYY

    ok.
   **psi***::>sys>user>***>sample2.rkp.1
   .....
   ```

ここで，load * を入力すると，カレントディレクトリ中の *.rkp ファイルが全てロードされます．

8. psave コマンドでパーシャルリンクした内容をセーブします．

```
P>> psave
Saving ....
**>pimos.kbn
**>pimos.ldb
**>pimos.sym

 ok.
>>
```

9. リンカを終了します．

```
>> exit
```

10. KL1 のソースプログラムを修正して再びコンパイルしなおす時は，Command ウインドウから，

```
<refresh>
```

を選択して，File/Directory Menu の表示を更新して下さい．

CAL を終了するコマンドは，Command メニューの >Exit< です．

注意：パーシャルリンクは PIMOS にパッチを当てることによって実現されています．従ってパーシャルリンクのたびに，PIMOS 本体のオブジェクトは大きくなります．パッチが大き過ぎると CSP へのロードに失敗することがあります．また，ユーザのプログラムにバグがある場合もロードの失敗が生じることがあります．PIMOS の立ち上げに失敗した場合は，第 3 章の最後の節を参照してこれに対処して下さい．

10.3 セルフコンパイラと CAL の使い分け

- プログラム開発の段階では PIMOS 上のセルフコンパイラを使用して下さい．
- 変更する可能性のないモジュールで頻繁に使うものは CAL でコンパイルすると便利です．
- 現行のバージョンでは，CAL でコンパイルした方が，セルフコンパイラでコンパイルするよりも，プログラムの実行速度は早くなります．

10.4 変数チェックの機能

CAL には，KL1 プログラム中の各節を調べ，一度しか現れない変数を message window に報告する機能があります．この機能によって，デッドロックを引き起こす原因となる可能性がある変数を知ることができます．

変数チェックの機能を使用するためには，File/Directory Menu でファイルを指定した後，Command メニュー の VarChk をクリックしてください．

さらに詳細な CAL の説明は「Multi-PSI/V2 クロスシステム (V2/CAL) の使用法」[4] を参照して下さい．

第 11 章

その他のユーティリティ

この章では， PIMOS の機能のうち知っていると便利なプールとタイマについて，例を用いて説明します.

11.1 プール

プールは任意の KL1 データを格納し，また取り出すことのできるデータベースです．プールには，検索用のキーを付けてデータを格納するキー付きのプールと，データをそのまま格納するキーなしのプールがあります． PIMOS では，キー付き，キーなしそれぞれに，用途に応じて次のようなプールを提供しています.

- キーなし : bag, stack, queue, sorted_bag
- キー付き : keyed_bag, keyed_set, keyed_sorted_bag, keyed_sorted_set

このうち，比較的よく使うと思われる bag, keyed_bug, keyed_set の使い方を例を用いて説明します．プールについての詳細は，「PIMOS マニュアル」[3] の 3.3 を参照して下さい.

- Bag : データの格納と取り出しの機能のみを有する最も基本的なプールです.
  呼び出し方

  ```
  pool:bag(S)
  ```

  プールへのデータの受け渡しは，この変数 S にメッセージプロトコルを送ることによって行ないます．メッセージプロトコルの詳細については「PIMOS マニュアル」[3] の 3.3 を参照して下さい.
  例

  ```
  ?- pool:bag(S),
     S=[empty(A), put("str"), put(atom), put({vect}),
        get(B), get_all(C), get_if_any(D)].
  A = yes
  B = {vect}
  C = [atom,"str"]
  D = {}
  S = [ ... ]
  yes.
  ```

  ここでは，まずプールが空かどうかを問い合わせ，変数 A に yes という答を得ます．次に，三つのデータ "str"(ストリング型), atom(アトム型), {vect}(ベクタ型) をプールに収納します．そして，収納したデータのうちどれか一つを取り出すために， get(B) と

いうメッセージを送り，B に {vect} というデータを得ます．メッセージ get_all(C) を送ると，変数 C に残りの全ての要素のリストが得られます．(一般に，一度取り出したデータはプールから消えます．) 最後のメッセージ get_if_any(D) に注意してください．これは get( ) と同様に要素を 1 つとりだすメッセージですが，プールが空の場合 get( ) が失敗 (プログラムが fail) するのに対し，get_if_any( ) は，{ } を返します．

- Keyed Bag : キー付きのプール．ハッシュテーブルを利用してデータを検索します．
  呼びだし方

```
pool:keyed_bag(S)
```

例

```
?- pool:keyed_bag(S),
   S=[empty(A), put("atom",atom), put(str,"str1"),
      put(str,"str2"), put({vect},v(1)),
      empty(str,B), get(str,C), get_if_any("atom",D),
      get_and_put({vect},E,v(100)), get_all(F) ]
A = yes
B = no
C = "str2"
D = {atom}
E = {v(1)}
F = {{{vect},v(100)},str("str1")}
S = [ ... ]
yes.
```

ここでは，まずプール全体が空かどうかを問い合わせ，yes を得ます．次に，"atom" というキーに対して atom というデータ，str というキーに対して "str1", "str2" という二つのデータ，さらに，{vect} というキーに対して v(1) というデータをプールに収納します．ここで，str というキーに対して，プールが空かどうかを問い合わせ，no という回答を得た後，そのキーに対してデータを一つ取り出します．次に，今度は "atom" というキーに対して空かどうかを問い合わせを行なわずに，直接 get_if_any でデータを取り出しています．その次の get_and_put は {vect} というキーのデータを一つ取り出し，それを v(100) で置き換えています．最後の get_all によって，プール中の全てのデータをキーと共に取り出しています．

- Keyed Set : キーの重複を許さないキー付きのプール．

```
pool:keyed_set(S)
```

例

```
?- pool:keyed_set(S),
   S=[put(key,"str",A), put(key,atom,B), get(key,C)].
A = {}
B = {"str"}
C = atom
S = [ ... ]
yes.
```

keyed_set では，メッセージ put によって，あるキーに対して何らかのデータを格納する際，そのキーに対して別のデータがすでに収納されている場合は，そのデータが put の第3引数に返されます．この例では，最初の put の実行の際にはキー key に対するデータが存在しないので，変数 A に {} が返され，次の put の実行時には，変数 B に "str" が返されています．

11.1.1 プールを利用したプログラムの例

以下に示すプログラムは，指定したファイルの各行をプール Keyed Bag に取り込んだ後，ユーザの指定した行を表示します．このプログラムをコンパイルする前に，第 4 章で示したモジュール "file" をコンパイルしておいて下さい．

```
:- module sample10.
:- public go/1.
:- with_macro pimos.

go(Name):- true|
        file:open(Name,r,[getl(L)|F],success),
        std_inter(IO)@priority($,-100),
        IO=[putt('Number of Lines = '),putt(NL),nl,
            prompt("> "),gett(T)|IO1],
        pool:keyed_bag(Pool),
        read_file(L,F,P1,1,NL),
        display(T,IO1,P2),
        merge({P1,P2},Pool).

std_inter(IO):- true|
        shoen:raise(pimos_tag#shell,get_std_inter,IO).

read_file(-1,F,P,N,NL):- true|
        F=[],P=[],NL:=N-1.
otherwise.
read_file(L,F,P,N,NL):- true|
        P=[put(N,L)|P1],
        F=[getl(L1)|F1],
        N1 := N+1,
        read_file(L1,F1,P1,N1,NL).

display(exit,IO,P):- true|
        IO=[],P=[].
display(N,IO,P):- integer(N)|
        P=[get_if_any(N,L)|P0],
        ( L={St} ->
          P0=[put(N,St)|P1],
          IO=[putt(N),putt(' : '),putl(St),
              prompt("> "),gett(T)|IO1]
        ; otherwise
        ; true -> IO=[prompt("> "),gett(T)|IO1],
                  P1=P0 ),
        display(T,IO1,P1).
```

```
otherwise.
display(_,IO,P):- true|
        IO=[prompt("> "),gett(T)|IO1],
        display(T,IO1,P).
```

実行例

```
?- sample10:go("sample10.kl1").
Number of Lines = 42
> 5.
5 : go(Name):- true|
> 7.
7 : std_inter(IO)@priority($,-100),
> 1.
1 : :- module sample10.
> exit.
 yes.
```

11.2 タイマ

タイマはファイルやウインドウと同様，PIMOS の提供する資源の一つです．タイマを使って，時刻を知ることができます．また，これによってプログラムの所用時間を測定することも可能です．

タイマを利用するために，次のようなモジュールを作りましょう．

```
:- module timer.
:- public create/1.

create(Timer):- true|
        shoen:raise(pimos_tag#task,
                general_request,General_Request_Device),
        General_Request_Device=
                [timer(normal(Timer_Request_Device,_,_))],
        Timer_Request_Device=
                [create(normal(Timer,_,_))].
```

このモジュールに対して，

```
?- timer:create(Timer), Timer = [...].
```

を実行すると，タイマデバイスへのストリームが Timer に得られます．このストリームに以下のメッセージを送ることによって，タイマの機能を利用できます．

- get_count(^Result)
  0時0分から現在までの経過時間をミリ秒単位で返します．Result には，normal(Count) が返ります．ここで，Count が経過時間です．

- on_at(Count,^Result)
  ミリ秒単位で指定した時刻を Count に与えると， Result に normal(Now) を返し，その時刻が来ると Now が wake_up に具体化されます.

- on_after(Count,^Result)
  ミリ秒単位で指定した時間を Count に与えると， Result に normal(Now) を返し，その時間を経過すると Now が wake_up に具体化されます.

11.2.1 タイマを利用したプログラムの例

ここでは，タイマを利用したプログラムの例として，アラームのプログラムを示します. タイマデバイスのストリームにメッセージ on_at(Count,^Result) を送ると，変数 Result には normal(Now) という値が返され，変数 Now は指定した時刻になると wake_up に具体化されます. アラームは，この変数が具体化された時に，メッセージを出力するプロセスによって実現されています.

```
:- module alarm.
:- public go/3.

go(H,M,Mes):-
        integer(H),integer(M), string(Mes,_,_)|
        Time := ( H*3600 + M*60 )*1000,
        timer:create( [on_at(Time,normal(Now))] ),
        alarm( Now, Mes ).
alarm( wake_up, Mes ):- true|
        shoen:raise( pimos_tag#shell, get_std_mes, Stream),
        Stream=[ putl(Mes) ].
```

このプログラムをコンパイルする前に，上に示したモジュール timer をコンパイルしておいて下さい. このプログラムはシェル・ユーティリティとして使用できます (第 8 章参照).
実行例

```
Shell> alarm(17,30,"### Time is up. ###")&
Shell> ......

Shell> (Shell Command or <cr> )
### Time is up. ###
```

指定した時間がくると, ### Time is up. ### のメッセージが表示されます.

参考文献

[1] (Pseudo) マルチ PSI システム管理マニュアル

[2] マルチ PSI/V2 コンソールシステム (CSP) 操作説明書

[3] PIMOS マニュアル (第 1.5 版)

[4] Multi-PSI/V2 クロスシステム (V2/CAL) の使用法

[5] KL1 プログラミング 入門編 / 初級編 / 中級編

[6] 小型化 PSI/SIMPOS システム管理説明書

[7] 小型化 PSI/SIMPOS 操作説明書 (1),(2)